THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS VI

SPÉCIALITÉ INFORMATIQUE

Présentée par Damien DUPUIS

Pour obtenir le titre de
DOCTEUR DE L'UNIVERSITÉ PARIS VI

KNIK
Routeur global pour la plate-forme CORIOLIS

Soutenue le 17 juin 2009,
devant le jury composé de

M.	Michel ROBERT	Rapporteur
M.	Marc SEVAUX	Rapporteur
M.	Pierre FOUILHOUX	Examinateur
M.	Alain GREINER	Examinateur
M.	Christian MASSON	Examinateur
Mme	Alix MUNIER KORDON	Directrice de thèse

A mon amour, ma femme.

Remerciements

Au sein de cet environnement instable et turbulent,
un seul élément reste constant : le changement.

TENZIN GYATSO, 14^{eme} dalaï-lama

Je souhaite tout d'abord exprimer toute ma reconnaissance à Alain Greiner, pour m'avoir accueilli au sein du département Sytem On Chip (SOC) du Laboratoire d'Informatique de Paris 6 (LIP6) et pour m'avoir offert l'opportunité de réaliser cette thèse.

Je remercie vivement Monsieur Michel Robert, professeur et directeur du Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) et Monsieur Marc Sevaux, professeur et directeur adjoint du Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance (LabSTICC), pour avoir accepté d'être membres de mon jury et rapporteurs de mon manuscrit.
J'adresse également mes remerciements à Monsieur Pierre Fouilhoux, maître de conférences au LIP6, et à Monsieur Alain Greiner, professeur au LIP6, tous deux examinateurs de ma thèse. Je remercie Madame Alix Munier Kordon, ma directrice de thèse, pour son encadrement et ses conseils avisés sur la rédaction de ce manuscrit.

Je tiens à remercier tout particulièrement Christian Masson avec qui j'ai eu la chance et le plaisir de travailler tout au long de ma thèse. Qu'il sache que je lui suis reconnaissant pour tous ses conseils, sa disponibilité, sa patience et sa grande gentillesse.

Cette thèse s'inscrivant dans le projet **CORIOLIS**, je souhaite remercier tous les gens ayant contribué à son développement : Christophe Alexandre, Jean-Paul Chaput, Hugo Clément, Sophie Dupuis et Marek Sroka.

A titre personnel, je remercie tous mes amis qui se sont intéressés de près ou de loin à cette thèse et plus particulièrement Mathieu, notre « *kamarad* » et Elodie & Pierre, les « *bloggers wannabe* ».
Je remercie également tous les membres de ma famille dont la curiosité les aura souvent poussés à me demander « *Alors, cette thèse c'est pour quand ?* ». J'ai une pensée toute particulière pour mes parents qui dès mon plus jeune âge m'ont donné goût à l'informatique et m'ont toujours soutenu dans mes choix par la suite.

Enfin, je remercie ma femme, Sophie, qui m'apporte joie, amour et soutien depuis presque dix ans et avec qui je partagerai bientôt le bonheur d'être parent.

Résumé

Les outils d'aide à la synthèse physique de circuits VLSI (Very Large Scale Integration) sont fortement dépendants de la technologie utilisée. L'évolution récente des technologies nanométriques et la taille des problèmes à traiter ont entraîné un regain d'intérêt pour l'étude et le développement d'outils de placement / routage dans le milieu académique. Le but de cette thèse est l'étude et la mise en œuvre d'un outil de routage global se situant, dans une chaîne de CAO (Conception Assistée par Ordinateur), entre la phase de placement et celle de routage détaillé.

La phase de routage global construit un tracé approximatif à partir d'une modélisation simplifiée des ressources de routage. Son principal objectif est d'effectuer la répartition globale des interconnexions en respectant les ressources disponibles. La solution produite est définie par un ensemble d'arbres de Steiner dont chacun relie les connecteurs du signal (*net*) auquel il est associé.

Dans cette thèse, nous présentons **KNIK** un outil de routage global intégré à la plate-forme de conception VLSI **CORIOLIS**.

Les ressources de routage sont modélisées à l'aide d'une structure mémoire compacte et légère qui permet de représenter toute solution partielle ou complète du tracé des nets au cours du traitement. Sur la base de cette structure, nous avons mis en œuvre une approche séquentielle basée sur l'algorithme de Dijkstra pour construire une solution initiale ainsi qu'une méthode originale de *ripup & reroute* permettant de résoudre les problèmes de sur-congestion.

Nous avons développé un ensemble d'outils modulaires d'instrumentation, d'analyse et de visualisation qui nous a permis de valider et d'évaluer notre outil sur les jeux de circuits de tests de référence. Les performances obtenues sont comparables à celles des meilleurs routeurs globaux académiques actuels.

Mots Clefs :
routage global, algorithme de Dijkstra, arbre de Steiner, *ripup & reroute*, plate-forme de conception VLSI

Abstract

Tools for the design of VLSI (Very Large Scale Integration) circuits are strongly dependent on the technology used. Recent developments in nanoscale technologies and growing size of circuits have led to renewed interest in the study and development of academic place and route tools. The aim of this thesis is to study and implement a global routing tool which takes place between placement and detailed routing.

The global routing tool builds a coarse solution from a simplified model of routing resources. Its main objective is to solve the problem of the overall distribution of interconnections within the available resources. This solution is defined by a set of Steiner trees : each tree links the connectors of the net it is associated with.

In this thesis we present **KNIK**, a global routing tool integrated into **CORIOLIS**, a VLSI design platform.

The routing resources are modelled with a compact and lightweight memory structure which allows to represent any partial or complete solution. Based on this structure, we implemented a sequential approach using Dijkstra's algorithm to construct an initial solution and an original method of *ripup & reroute* to solve overflow problems.

We developed an evaluation, analysis and visualization platform that allows us to validate our tool on reference benchmarks suites. Its performances are similar to those of current best academic global routers.

Keywords :
global routing, Dijkstra's algorithm, Steiner tree, *ripup & reroute*, VLSI design platform

Table des matières

Table des figures

Liste des tableaux

Introduction

L'évolution récente des technologies nanométriques et la taille grandissante des problèmes à traiter sont à l'origine de nouvelles études et développements dans le cadre de la synthèse physique de circuits VLSI (Very Large Scale Integration). Les flots de conception classiques ont cédé la place à de nouveaux flots permettant aux différents outils d'intéragir tout au long du processus progressif de réalisation d'un circuit.

Les trois principaux outils de la phase de placement / routage d'un flot de conception sont :

- l'outil de **placement** qui permet d'obtenir un placement de toutes les cellules du circuit sans recouvrement,
- l'outil de **routage global** qui, à partir d'une modélisation simplifiée des ressources de routage, permet de construire rapidement un tracé approximatif des interconnexions utilisé par le placement ou le routage détaillé,
- l'outil de **routage détaillé** qui résout la combinatoire locale du tracé, régie par des règles de dessin complexes.

Le projet **CORIOLIS** [Ale07] définit une plate-forme de prototypage rapide pour les circuits VLSI, construite autour d'une base de données centralisée écrite en C++ et bénéficiant d'outils de visualisation graphique performants. Parmi les outils nécessaires au flot décrit dans [Ale07], nous disposons d'un outil de placement performant. Cette thèse a pour but le développement d'un outil de routage global au sein de cette plate-forme. Cet outil doit respecter deux contraintes.

Tout d'abord, il doit être rapide. En effet, le processus de synthèse par raffinement progressif de la plate-forme **CORIOLIS** implique une utilisation conjointe du placement et du routage global : les directives globales définies par le routeur servent à guider le placement des cellules de façon à réduire la longueur des interconnexions locales. De ce fait, la phase de routage global est appelée à plusieurs reprises mais ne doit pourtant pas ralentir le processus général. De plus, lorsqu'il est utilisé pour construire une solution initiale pour le routeur détaillé, notre outil doit être suffisament rapide pour traiter de très gros circuits.

La seconde contrainte concerne les performances de notre outil. La solution qu'il construit doit être suffisament de bonne qualité pour envisager un routage détaillé. Comme nous le verrons par la suite, il existe plusieurs critères pour mesurer la qualité d'une solution, mais le principal est la minimisation de la sur-congestion. Nous souhaitons donc que notre outil crée des solutions ne contenant aucune violation des ressources de routage, ou tout du moins, le moins possible.

En plus de **KNIK**, notre outil de routage global, nous avons également développé un environnement spécifique d'évaluation et de visualisation ainsi que l'infrastructure nécessaire au chargement et à la sauvegarde des jeux de circuits de référence qui ont été élaborés par la communauté académique en liaison étroite avec l'industrie.

Cette thèse a été réalisée au sein du département SOC (System On Chip) du laboratoire LIP6 (Laboratoire d'Informatique de Paris 6) de l'Université Pierre et Marie Curie, dans l'équipe de développement du projet **CORIOLIS**. Ce manuscrit présente nos travaux de recherche et se décompose en quatre chapitres.

Chapitre 1 : Problématique

Le routeur global utilise une modélisation simplifiée des ressources de routage basée sur une structure de graphe de routage. Dans ce premier chapitre, nous commençons par décrire cette structure permettant de représenter les ressources disponibles et de déterminer un arbre d'interconnexion pour chaque signal (*net*) du circuit. Nous définissons ensuite la notion originale de composante connexe[1] qui permet de représenter très simplement sur ce graphe tous les connecteurs à relier d'un net et toutes les étapes intermédiaires de la construction d'un arbre d'interconnexion.

Nous présentons ensuite les trois critères qui permettent de juger la qualité d'une solution de routage : congestion, longueur totale des interconnexions et nombre de vias. Les deux derniers critères étant antagonistes avec celui de minimisation de la congestion, la recherche d'une bonne solution de routage global consiste à trouver un bon compromis entre ces trois critères.

Par la suite nous présentons les deux grandes familles de méthodes permettant de construire une solution initiale du problème de routage global : les **approches concurrentes**, qui traitent tous les nets simultanément, et les **approches séquentielles**, qui, à l'inverse, considèrent les nets individuellement et séquentiellement. Nous verrons que les approches concurrentes ne sont pas adaptées au traitement de grands circuits. Les approches séquentielles simplifient les algorithmes de traitement mais nécessitent la définition d'un ordonnancement des nets qui, s'il est mal choisi, peut conduire à des blocages empêchant la construction d'une solution valide. Parmi les approches séquentielles, les méthodes de type *Maze routing* sont, à priori, les plus adaptées pour résoudre le problème.

Les solutions générées sont améliorées grâce à une procédure incrémentale (que l'on nomme usuellement *ripup & reroute*) qui vise à déloger les nets contribuant

1. Le terme de composante connexe ne correspond pas à la définition classique de la théorie des graphes mais à la notion de connexité électrique.

aux conflits afin de les retracer selon d'autres chemins. Nous présentons différents scénarios de mise en œuvre du *ripup & reroute*.

Chapitre 2 : Méthodes de résolution

Dans ce chapitre nous nous intéressons aux méthodes séquentielles de type *Maze routing* qui sont à priori les plus adaptées à la résolution du problème de routage global, comme nous l'avons montré dans le chapitre 1. Nous définissons dans un premier temps la fonction de coût, prenant en compte les trois critères précédemment cités, et utilisée pour construire un arbre d'interconnexion optimal à l'aide de ces méthodes.

Une approche séquentielle, donc gloutonne, ne permet pas d'anticiper les conflits futurs résultant des choix effectués au cours des premières étapes. Afin d'y remédier, nous utilisons une technique d'estimation anticipée de la congestion, qui permet d'éviter que des décisions arbitraires ne soient prises pour les premiers nets traités. Nous présentons différentes méthodes pour calculer et actualiser dynamiquement cette estimation, ainsi que les avantages et inconvénients de chacune.

Dans la dernière section de ce chapitre nous présentons l'algorithme de Dijkstra qui est mis en œuvre dans notre outil. Nous commençons par décrire l'algorithme de Dijkstra et sa variante A* qui permettent de construire une chaîne de coût minimal entre deux sommets quelconques d'un graphe de routage. Nous présentons ensuite différentes extensions permettant d'utiliser ces algorithmes pour interconnecter des composantes connexes ainsi qu'une approche multi composantes considérant simultanément toutes les composantes connexes d'un net.

Chapitre 3 : KNIK routeur global pour la plate-forme CORIOLIS

Dans ce chapitre nous décrivons la mise en œuvre de **KNIK** notre outil de routage global. Nous présentons tout d'abord notre structure de graphe de routage (régulier ou non) qui permet de modéliser les ressources disponibles et offre un ensemble de fonctionnalités utiles aux algorithmes. Parmi ces fonctionnalités nous détaillons la recherche du sommet associé à un point quelconque de la surface du circuit, la fonction d'estimation anticipée de la congestion et la matérialisation du routage. Nous abordons aussi l'aspect visualisation du graphe, de sa congestion et de la matérialisation des nets.

Nous introduisons ensuite les notions de `netStamp` et `connexId` qui permettent de représenter et de manipuler très simplement les composantes connexes sur ce graphe de routage. Elles sont particulièrement bien adaptées aux méthodes séquentielles, puisqu'elles permettent de passer outre la réinitialisation du graphe. Or la réinitialisation de tous les sommets et toutes les arêtes du graphe à chaque nouveau net traité

serait trop longue et augmenterait considérablement le temps d'exécution de notre outil. Pour expliquer l'utilisation du `netStamp` et du `connexId`, nous détaillons les fonctions d'initialisation des composantes connexes d'un net, de propagation du coût des sommets et enfin de fusion de deux composantes.

Nous présentons ensuite les spécifités de notre mise en œuvre de l'algorithme de Dijkstra en illustrant notamment sa modularité par l'étude de l'impact du coût des vias sur le coût d'une solution. Pour finir nous détaillons les différentes fonctions de notre mise en œuvre du *ripup & reroute* : l'identification des portions de nets (segments) à dérouter, le déroutage d'un segment et enfin le reroutage d'un net.

Chapitre 4 : Résultats

Dans ce quatrième et dernier chapitre nous présentons l'environnement de test ainsi que l'ensemble des résultats expérimentaux obtenus avec notre outil. Nous commençons par présenter l'environnement de test que nous avons développé au sein de la plate-forme **CORIOLIS**. Il nous permet de charger, de sauvegarder, d'évaluer et de visualiser des solutions de routage global aux formats des jeux de circuits de test académiques.

Nous présentons ensuite un premier jeu de résutats pour les circuits de l'ispd98 qui mettent en évidence le fait que notre première approche de *ripup & reroute* est trop simple et ne suffit pas à éliminer toute la sur-congestion d'un circuit. Nous verrons que pour tous les circuits, notre outil converge vers une solution non valide.

Nous introduisons alors une technique de négociation de la congestion dont nous détaillons la mise en œuvre et l'intégration dans l'outil **KNIK**. Grâce à cette technique notre outil construit une solution valide pour tous les circuits du jeu de test de l'ispd98 et pour six des huit circuits de l'ispd07.

Nous analysons en détails ces nouveaux résultats et nous les comparons à ceux obtenus par le routeur détaillé **FGR** [RM07] qui, s'étant placé premier du concours de routage global de l'ispd07, est devenu la référence des routeurs globaux dans le milieu académique.

Chapitre 1 Problématique

1.1 Introduction

Dans un flot de conception classique [GP92] [dCA], où chaque étape est réalisée par un outil indépendant des autres, le routage global intervient après le placement détaillé des cellules du circuit et juste avant le routage détaillé.

Le routage global et le routage détaillé ont pour but de construire pour chaque signal (*net*) du circuit un cheminement, à travers les ressources de routage, interconnectant l'ensemble de ses connecteurs. Tandis que le routeur détaillé prend en compte l'ensemble des règles de dessin définies pour chaque couche de métal du circuit, le routeur global utilise un modèle simplifié des ressources de routage basé sur un **graphe de routage**. Grâce à ce modèle simplifié, le routeur global construit pour chaque net du circuit un cheminement « approximatif » qui sera finalisé par le routeur détaillé.

De façon à pouvoir gérer de gros circuits et les problèmes liés aux technologies nanométriques, de nouveaux flots de conception sont apparus [fC] [ACC+05] [ASCM06]. Ces flots, utilisant une base de données centralisée et partagée par tous les outils de conception, ont modifié l'utilisation du routeur global.

En effet le routeur global est désormais aussi utilisé conjointement avec les outils de placement global et d'analyse temporelle anticipée. Cette nouvelle utilisation a suscité ces dernières années un regain d'intérêt pour le routage global dans le monde académique, comme le montre l'apparition récente de nouveaux outils [CP06] [PC06] [RM07] [CLYP07] [CJX+07], de nouveaux jeux de circuit de tests dédiés au routage global [NSY08] et même de concours de routage global [oPDGRCa] [oPDGRCb].

Dans ce chapitre, nous présentons le problème de routage global ainsi que les structures permettant de le modéliser. Nous commençons par décrire la structure de graphe de routage permettant de représenter les ressources disponibles et de déterminer un arbre d'interconnexion pour chaque net. Nous définissons ensuite la notion originale de composante connexe, ainsi que les critères permettant de juger de la qualité d'une solution de routage global : congestion, longueur totale des

interconnexions et nombre de vias.

Dans une seconde section, nous présentons les deux grandes familles de méthodes permettant de construire une solution initiale du problème : les approches concurrentes, qui traitent tous les nets simultanément, et les approches séquentielles, qui considèrent les nets individuellement et séquentiellement. La solution initiale générée est améliorée grâce à une procédure incrémentale, dite de *ripup& reroute*, qui vise à éliminer la sur-congestion. Nous présentons différents scénarios de mise en œuvre de *ripup & reroute*.

1.2 Formalisation du problème de routage global

Comme nous l'avons dit précédemment le problème de routage global se formule sur un **graphe de routage**. Ce dernier permet de modéliser de façon simplifiée les ressources de routage disponibles et de déterminer un cheminement « approximatif » pour chaque net appartenant à l'ensemble des nets à router, noté N.

Dans cette section nous présentons en détails la structure de graphe de routage ainsi que la manière de représenter un net sur ce graphe. Puis nous détaillons les critères permettant de calculer le coût d'une solution de routage global. Enfin nous présentons des variantes du graphe de routage (bidimensionnel ou irrégulier) ainsi que les fonctions associées.

1.2.1 Graphe de routage

Définition du graphe de routage

La surface du circuit est divisée en un pavage régulier, dont la hauteur des bandes horizontales est égale à celle d'une cellule de bibliothèque pré-caractérisée (*slice*). La largeur des bandes verticales est du même ordre de grandeur.
Chaque intersection de deux bandes (horizontale et verticale) définit un pavé qui recouvre une partie du circuit pour chaque couche de métal ainsi qu'une ou plusieurs parties de cellules ou macro-blocs.

Nous notons $G(S, A)$ le graphe de routage formé de l'ensemble des sommets S et celui des arêtes A, tels que :

- il existe un sommet $s_i \in S$, avec $i \in \{1, \ldots, |S|\}$, pour chaque paire pavé + couche de métal.
- il existe une arête $a_{ij} \in A$ reliant les sommets s_i et s_j s'ils sont associés à deux pavés adjacents et qu'il existe des ressources (pistes de routage / vias) permettant de passer de l'un à l'autre.

1.2 Formalisation du problème de routage global

Le graphe $G(S, A)$ ainsi défini est un graphe **connexe**, c'est-à-dire que quels que soient $s_i, s_j \in S$, il existe toujours une chaîne (une succession d'arêtes) qui permet d'atteindre s_j à partir de s_i. La figure 1.1 présente un exemple de graphe de routage tridimensionnel.

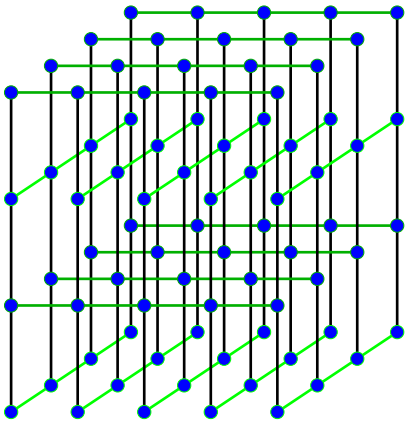


FIGURE 1.1 – Graphe de routage tridimensionnel

Pour bien modéliser les ressources de routage disponibles, il faut tout d'abord comprendre la nature et l'organisation de ces ressources. Jusqu'à l'année 2007, il n'existait qu'un seul jeu de circuits de test pour le routage global. Ce jeu de circuits dérive de celui défini lors de l'*ISPD98 placement benchmarks contest* [ben] et, du fait de son ancienneté, ne considère qu'une paire de couches de métal.

En 2007, lors de l'*ISPD'07 global routing contest* [oPDGRCa] un nouveau jeu de huit circuits de test « actualise » le problème de routage global. En 2008, une étude [NSY08] présente les caractéristiques de ce jeu de test.

Les technologies nanométriques actuelles fournissent six à huit couches de métal, voire plus, pour effectuer les interconnexions. Ces couches de métal sont numérotées de 1 pour la plus basse (la plus proche du substrat) à l pour la plus haute (l représentant le nombre de couches de métal) et sont nommées *métalX* où *X* est le numéro de couche.

La direction privilégiée des pistes de routage est alternée : les couches paires sont à dominante horizontale tandis que les couches impaires sont à dominante verticale, comme l'illustre la figure 1.2.

(a) Couche impaire (b) Couche paire

FIGURE 1.2 – Représention des couches de métal sur le graphe de routage

Les couches 1 et 2 (les plus basses) sont généralement dédiées aux interconnexions internes des cellules de la bibliothèque pré-caractérisée, la couche 2 contenant les connecteurs des cellules. Selon la technologie et la bibliothèque utilisée, ces interconnexions peuvent occuper jusqu'à 80% des pistes de routage disponibles, laissant ainsi très peu de ressources libres pour les connexions entre cellules.

Dans le cas de conception hiérarchique, lorsque les technologies offraient peu de couches de métal, les macro-blocs étaient considérés comme des obstacles opaques : dans la zone couverte par un macro-bloc, toutes les ressources de routage étaient utilisées, les connecteurs du macro-bloc situés à la périphérie permettaient ainsi de l'interconnecter au reste du circuit.

Aujourd'hui, l'approche dite « *over the cell routing* » prédomine. On utilise des macro-blocs pré-tracés, dans lesquels les fils d'interconnexions internes utilisent prioritairement les couches de métal 3 et 4 et les connecteurs situés à l'intérieur du macro-bloc sont représentés le plus souvent par des segments équipotentiels (faisant partie de l'interconnexion interne) visibles des couches de métal hautes. Les fils d'interconnexions entre macro-blocs se situent sur les couches supérieures.

Ces macro-blocs sont vus par le routeur global comme des « obstacles poreux » : certaines zones d'un macro-bloc peuvent utiliser 100% des ressources disponibles, tandis que d'autres zones, appelées « transparences », définissent des zones dans lesquelles pour certaines couches toutes les ressources sont disponibles.

Pour représenter les ressources de routage disponibles, nous associons à chaque arête $a_{ij} \in A$ (reliant s_i et $s_j \in S$) une **capacité** $cap(a_{ij})$ qui représente le nombre de pistes de routage disponibles, sur la couche de métal, reliant les deux pavés associés à s_i et s_j.

Cette capacité, qui ne varie pas au cours du traitement, peut cependant être réduite à l'origine du fait d'obstacles (tels que des pré-tracés dans un macro-bloc) interdisant l'utilisation de certaines pistes de routage.

Nous associons aussi une **longueur** à chaque arête $a_{ij} \in A$, notée $long(a_{ij})$ et définie par la distance centre à centre des deux pavés associés aux sommets qu'elle relie. Cette distance, exprimée en distance de Manhattan, est normalisée lorsque le graphe de routage est régulier (en la divisant par le pas de grille du graphe).

Les arêtes « transversales » du graphe de routage permettent de représenter les changements de couches de métal, c'est-à-dire les « **vias** ». Un via permet d'assurer la connexité électrique entre deux couches de métal adjacentes. Il est composé de deux contacts (un sur chacune des couches de métal qu'il relie) et d'un « trou » dans l'oxyde isolant séparant les deux couches de métal.

1.2 Formalisation du problème de routage global

Pour deux couches de métal non adjacentes, on parle de « **vias empilés** » (*stacked vias*) qui traversent toutes les couches intermédiaires (métal et oxyde) créant des contacts sur toutes les couches de métal concernées.

De plus, pour les technologies inférieures à 130nm, la résistivité des vias augmente fortement comparativement à celle des fils d'interconnexion, tandis que leur fiabilité diminue. Pour pallier ce problème, il est donc nécessaire de doubler leur largeur, ce qui influe grandement sur les obstructions générées [LW06].

Afin d'éviter de complexifier inutilement la modélisation, le modèle approximé des ressources de routage ne prend pas en compte les ressources locales aux pavés telles que les vias ; aucune capacité n'est donc associée aux arêtes transversales.

Représentation d'un net sur le graphe de routage

Tout d'abord, nous définissons la notion d'**arbre d'interconnexion** qui va nous permettre de représenter un net sur le graphe qu'il soit routé partiellement, totalement ou pas du tout.

Définition 1.1.

Un **arbre d'interconnexion** $\mathcal{A}$ est un sous-graphe partiel de $G(S, A)$ connexe sans cycle qui interconnecte un sous-ensemble de sommets de S. Il représente un tracé approximatif de l'interconnexion d'une partie ou de tout un net.

En entrée du routage global, chaque net $n_i \in N$ (où $i \in \{1, \ldots, n\}$ est l'indice du net et n est le nombre de nets à router) est représenté par un ou plusieurs arbres d'interconnexion représentant chacun une partie électriquement connexe du net.

Le routage global du net est achevé lorsque toutes les parties du net sont interconnectées, c'est-à-dire lorsqu'un seul arbre d'interconnexion représente le net sur le graphe $G(S, A)$.

En entrée du routeur global, chaque net $n_i \in N$ est composé d'un ensemble de connecteurs (pins) à relier, noté P_i avec $i \in \{1, \ldots, n\}$.

Généralement, un connecteur $p_{ij} \in P_i$ (où $j \in \{1, \ldots, |P_i|\}$ est l'indice du connecteur) appartient à une cellule de bibliothèque pré-caractérisée. Il se situe la plupart du temps sur une seule couche de métal et est recouvert totalement par un pavé. Nous associons ce connecteur p_{ij} au sommet $s \in S$ correspondant (même couche de métal et pavé associés) qui contient son représentant dans le graphe $G(S, A)$.

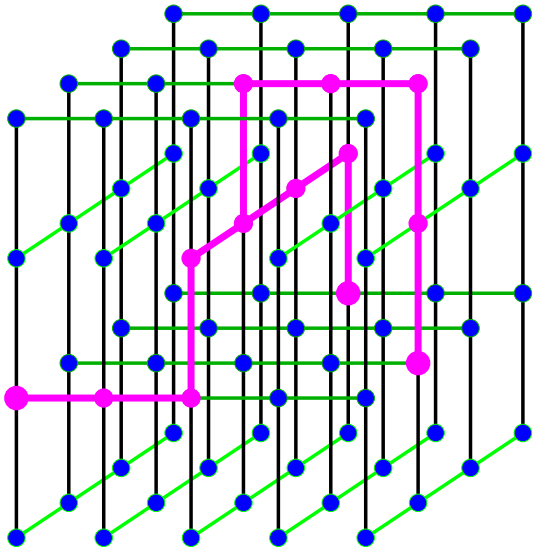


FIGURE 1.3 – Exemple d'arbre d'interconnexion sur un graphe tridimensionnel

Il faut noter toutefois que :

- Si plusieurs connecteurs appartenant au net n_i et de même couche sont recouverts par un seul pavé (soit parce que ce sont des connecteurs électriquement équivalents d'une cellule, soit parce qu'ils doivent être connectés par ce net) ils n'ont qu'un seul sommet s correspondant. Notons que dans le cas particulier où tous les connecteurs du net n_i sont recouverts par un seul pavé et sont de même couche, le net est ignoré par le routage global puisque, sur le graphe $G(S, A)$, il n'est représenté que par un seul sommet $s \in S$. (Toutefois l'occupation des ressources locales de ce net doit être prise en compte.)
- Si un connecteur $p_{ij} \in P_i$ est « réparti », c'est-à-dire qu'il est recouvert par plusieurs pavés adjacents (par exemple un « grand » connecteur à l'intérieur d'un macro-bloc), il a un représentant dans chacun des sommets associés à ces pavés.
- Si un connecteur $p_{ij} \in P_i$ est « empilé », c'est-à-dire qu'il est constitué d'une superposition de plusieurs connecteurs sur des couches de métal adjacentes (électriquement reliées par des vias), il a un représentant dans chacune des couches associées.

Un connecteur $p_{ij} \in Pi$, peut donc être représenté par plusieurs sommets du graphe $G(S, A)$. Ces sommets, offrant plusieurs points de liaisons pour le routeur global, sont dit « électriquement connexes ».
Nous notons $T_{ik}, k \in \{1, \ldots, k_i\}$ les ensembles de sommets de S électriquement connexes du net n_i. k_i est le nombre de composantes connexes du net n_i (voir définition 1.2).

D 'après les « règles » d'association entre les connecteurs et les sommets définies précédemment, le nombre k_i est toujours inférieur ou égal au nombre de connecteurs

$|P_i|$. Ces ensembles nous permettent de définir la composante connexe d'un net.

Définition 1.2.

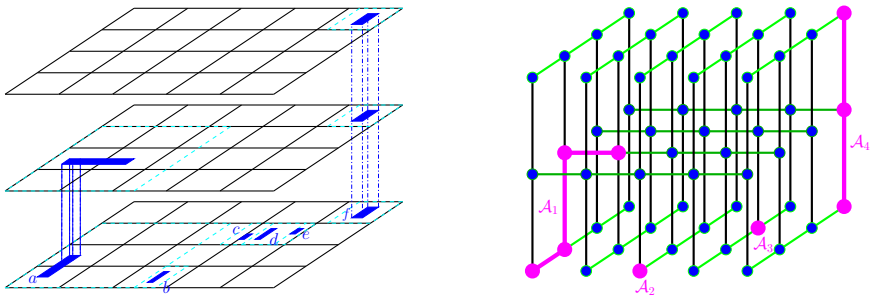
Une **composante connexe** du net $n_i \in N$ est un sous graphe partiel $\mathcal{A}_{ik}(T_{ik}, A_{ik})$ de $G(S, A)$ tel que :
- T_{ik} est un ensemble de sommets électriquement connexes
- toute arête $a \in A_{ik}$ est une arête de A
- $\mathcal{A}_{ik}$ est connexe et sans cycle ($\mathcal{A}_{ik}$ est un arbre)

Une composante connexe permet ainsi de représenter une connexion possible entre les sommets d'un ensemble $T_{ik}, i \in \{1, \ldots, n\}, k \in \{1, \ldots, k_i\}$.

Notons que le terme de composante connexe ainsi défini, ne correspond pas à la définition classique de la théorie des graphes. Le cas particulier d'une composante connexe ne représentant qu'un sommet $s \in S$ de $G(S, A)$ est appelé une composante connexe **ponctuelle**.

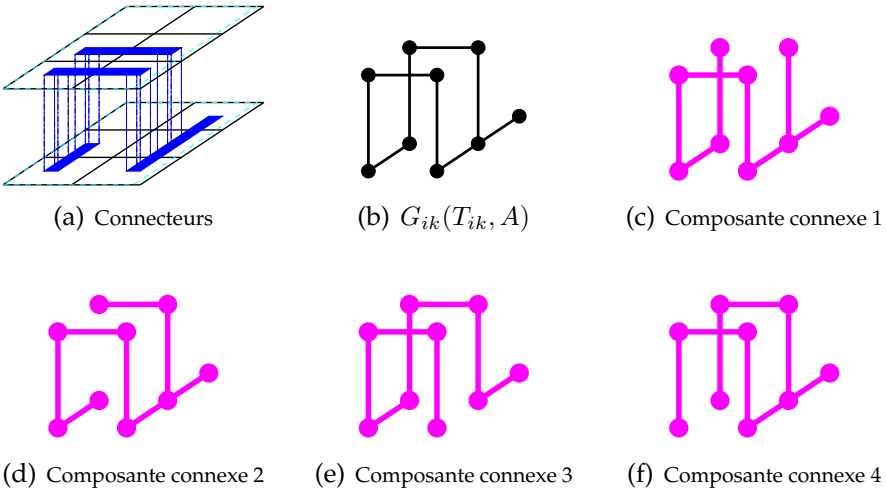
L'ensemble des composantes connexes du net $n_i \in N$ représente l'ensemble des connecteurs à relier P_i. Cette représentation est injective, c'est-à-dire qu'à tout connecteur $p_{ij} \in P_i$, nous associons une unique composante connexe $\mathcal{A}_{ik}$, mais une même composante connexe peut représenter plusieurs connecteurs.
L'existence de la composante connexe est assurée du fait que le sous graphe $G_{ik}(T_{ik}, A)$ est connexe puisque les pavés correspondant aux sommets de T_{ik} sont tous adjacents. Dans certains cas, pour un ensemble T_{ik} il existe plusieurs composantes connexes possibles, puisque le graphe $G_{ik}(T_{ik}, A)$ contient des cycles (voir figure 1.5). On en choisit alors une de manière arbitraire.

La figure 1.4 présente la correspondance entre les connecteurs a, b, c, d, e et f d'un net et les composantes connexes associées : $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ et $\mathcal{A}_4$. La composante connexe $\mathcal{A}_3$ représente les trois connecteurs c, d et e.



(a) Connecteurs (b) Composantes connexes

FIGURE 1.4 – Correspondance entre connecteurs et composantes connexes



(a) Connecteurs (b) $G_{ik}(T_{ik}, A)$ (c) Composante connexe 1

(d) Composante connexe 2 (e) Composante connexe 3 (f) Composante connexe 4

FIGURE 1.5 – Plusieurs composantes connexes possibles pour un graphe $G(T_{ik}, A)$

Quelle que soit la forme de la composante connexe, dès que l'un de ses sommets est atteint par le routeur global, toute la composante connexe est interconnectée.

Grâce à la notion de composante connexe d'un net, router le net $n_i \in N$, équivaut à trouver un arbre d'interconnexion $\mathcal{A}_i$, reliant toutes ses composantes connexes. La figure 1.6 présente un exemple d'arbre d'interconnexion (en marron) reliant les composantes connexes $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ et $\mathcal{A}_4$ (en magenta).

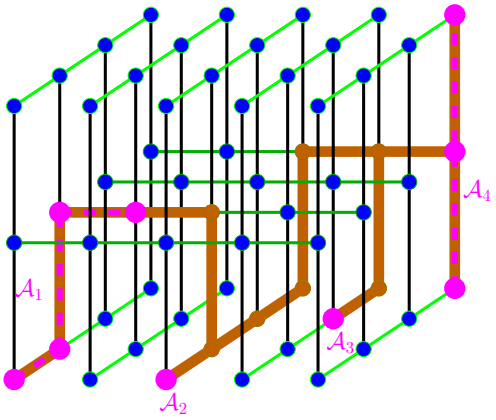


FIGURE 1.6 – Exemple de routage pour le net de la figure 1.4

Compléments sur le graphe de routage

Comme nous l'avons dit précédemment, certaines cellules ou macro-blocs contiennent des **connecteurs répartis**. Dans le cas d'un macro-bloc suffisamment grand, les connecteurs constituant le connecteur réparti sont recouverts par des pavés

disjoints. Or, la notion de composante connexe d'un net telle que nous venons de la définir ne permet pas de représenter ce cas.

L'interconnexion électrique reliant ces connecteurs en interne du macro-bloc n'est pas visible depuis le routeur global. Ce dernier les considère comme autant de points de liaison possibles pour relier le signal interne au macro-bloc au reste du net considéré.

Pour que le routeur global prenne en compte le fait que cette interconnexion existe, nous définissons pour chaque connecteur réparti du net $n_i \in N$ un graphe $G'_{ik}(T'_{ik}, A'_{ik})$ (où i est l'indice du net et k celui de composante connexe), dit « graphe dédié », tel que :

- T'_{ik} soit l'ensemble des sommets électriquement connexes représentant les connecteurs constituant le connecteur réparti,
- G'_{ik} soit une clique (tous les sommets sont deux à deux adjacents).

Chacun des graphes dédiés G'_{ik} représente une interconnexion existant à l'intérieur d'un macro-bloc. Pour le routeur global, utiliser une arête de l'un de ces graphes ne consomme aucune ressource. Toutefois l'occupation des ressources de routage par les interconnexions internes est prise en compte puisque, dans le graphe $G(S, A)$, la capacité des arêtes correspondantes est réduite.

Grâce aux graphes dédiés G'_{ik}, nous pouvons définir les composantes connexes $\mathcal{A}'_{ik}$ représentant les connecteurs répartis du net n_i.

Comme nous l'avons dit précédemment, lorsqu'il existe plusieurs composantes connexes pour un même G_{ik}, nous en sélectionnons une de manière arbitraire. Pour simplifier la mise en œuvre de la structure de données représentant les composantes connexes, nous restreignons le nombre de composantes connexes $\mathcal{A}'_{ik}$ possibles, en ne considérant que celles qui sont des chaînes de G'_{ik}. Il existe toujours une telle composante connexe puisque G'_{ik} est une clique.

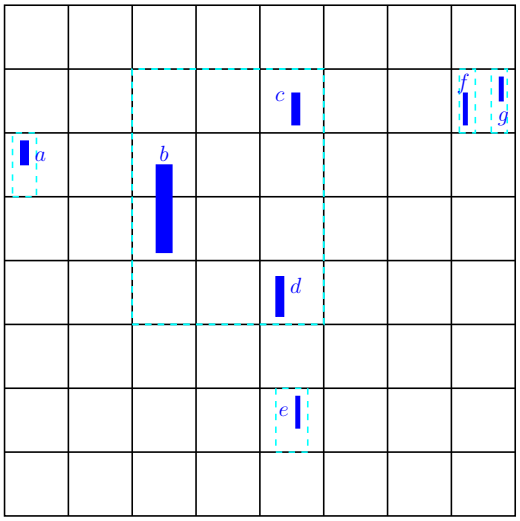
Pour chaque net $n_i \in N, i \in \{1, \ldots, n\}$, le routage global considère les graphes G, G_{ik} et G'_{ik} ainsi que les composantes connexes $\mathcal{A}_{ik}$ et $\mathcal{A}'_{ik}, k \in \{1, \ldots, k_i\}$ pour trouver une solution de routage global.
Cette solution est un arbre d'interconnexion $\mathcal{A}_i$ qui relie sur l'union des graphes G, G_{ik} et G'_{ik} les composantes connexes $\mathcal{A}_{ik}$ et $\mathcal{A}'_{ik}$.

Il est important de noter que chacun des graphes dédiés G'_{ik} « n'existe » que pendant la durée de traitement du net $n_i \in N$.

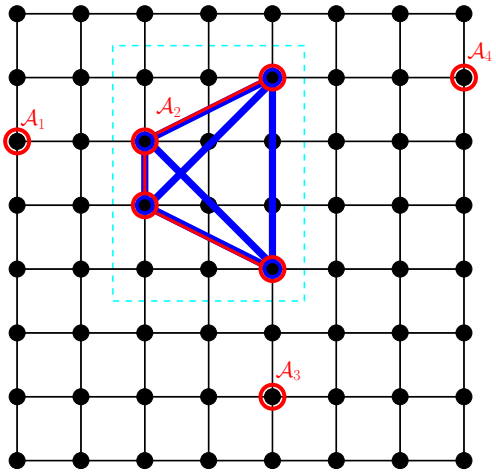
Considérons l'exemple de la figure 1.7(a), dans lequel un net se décompose en sept connecteurs, quatre appartenant à des cellules de bibliothèque pré-caractérisée (a, e, f et g) et trois constituant un connecteur réparti de macro-bloc (b, c et d). Par

souci de clarté, nous considérons un graphe de routage $G(S, A)$ bidimensionnel, mais le raisonnement reste identique pour un graphe tridimensionnel.

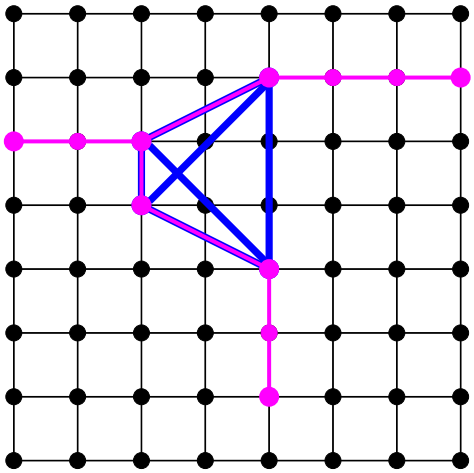
Les connecteurs a, e, f et g sont très simplement associés aux composantes connexes ponctuelles $\mathcal{A}_1$, $\mathcal{A}_3$ et $\mathcal{A}_4$ (voir figure 1.7(b)). Pour b, c et d formant le connecteur reparti, nous considérons le graphe dédié (clique représentée en bleu) et nous choisissons la composante connexe $\mathcal{A}_2$ représentée en rouge.
En considérant l'union du graphe de routage et du graphe dédié, l'algorithme de routage global construit un arbre d'interconnexion dont un exemple est donné sur la figure 1.7(c).



(a) Placement des cellules et macro-blocs



(b) Composantes connexes



(c) Net routé

FIGURE 1.7 – Des connecteurs au net routé

1.2 Formalisation du problème de routage global

Une solution au problème de routage global est formée par l'ensemble des arbres d'interconnexion construits $\{\mathcal{A}_i, i \in \{1, \ldots, n\}\}$. Sachant que chaque arbre d'interconnexion occupe un certain nombre de ressources de routage, voyons maintenant comment évaluer la qualité de cette solution.

1.2.2 Fonction de coût du routage global

La qualité d'une solution de routage global est fonction de trois critères, qui sont, par ordre de priorité décroissante : la **congestion** du circuit, la **longueur totale des fils d'interconnexion** et le **nombre de vias**.

Congestion

La notion de congestion est définie par le **taux de congestion**, qui est le rapport entre le nombre de ressources utilisées et le nombre de ressources disponibles initialement, pour chaque arête.

Sur le graphe, les ressources disponibles sont représentées par la capacité d'une arête. Nous y associons une **occupation** représentant les ressources utilisées. L'occupation $occ(a)$ pour une arête $a \in A$, est égale au nombre d'arbres d'interconnexion l'utilisant.

Pour l'arête $a \in A$, le taux de congestion est :

$$tauxCongestion(a) = \frac{occ(a)}{cap(a)}.$$

Lorsque ce taux de congestion est proche de 1, on dit que l'arête est **congestionnée**. Bien qu'il reste des pistes de routage non occupées, il vaut mieux éviter d'utiliser cette arête pour faciliter le travail du routeur détaillé.

Lorsqu'il est strictement supérieur à 1, on parle de **sur-congestion**. Les ressources ne sont pas respectées et le routeur détaillé devra produire plus d'effort pour essayer de résoudre le problème.

Le taux de congestion permet de facilement localiser les **zones congestionnées** du circuit. Ces zones sont définies par un ensemble d'arêtes proches, fortement congestionnées.
Le routage global doit éviter le plus possible de générer de telles zones.

En effet, si une arête sur-congestionnée se situe au milieu d'arêtes non ou peu congestionnées, le routeur détaillé peut facilement résorber cette congestion en « déportant » certains segments vers les pistes de routage des pavés adjacents.
En revanche, dans une zone congestionnée il ne peut pas résoudre le problème par de petites modifications locales ; une vision d'ensemble de la zone congestionnée est

nécessaire et donc un effort de calcul plus important.

Pour quantifier l'excès dû à la sur-congestion, nous utilisons la notion de **dépassement** :

$$\forall a \in A, dep(a) = \begin{cases} occ(a) - cap(a) & \text{si } occ(a) > cap(a) \\ 0 & \text{sinon} \end{cases}$$

Nous définissons aussi le **dépassement total** :

$$depassementTotal = \sum_{a \in A} dep(a)$$

Bien que l'absence de sur-congestion ne garantisse pas l'existence d'une solution de routage détaillé, du fait notamment de l'approximation des ressources, la minimisation du dépassement total est l'objectif prioritaire de la majorité des algorithmes de routage global.

Longueur totale des fils d'interconnexion

Comme le montrent [Ott98] et [DK99] la longueur des fils d'interconnexion est un critère déterminant à la fois pour les performances d'un circuit mais aussi pour sa fiabilité.

Le temps de propagation d'un signal sur un fil d'interconnexion augmente avec la longueur du fil.
De plus, avec les technologies nanométriques apparaît un problème de bruit sur les longs fils. Si deux fils d'interconnexion restent voisins sur une trop grande longueur, un effet capacitif (*crosstalk*) perturbe les signaux de chacun.

Il est donc nécessaire de minimiser la longueur totale des fils d'interconnexion. Pour cela, il faut que les arbres d'interconnexion soient de longueur minimale. Cette longueur est définie par la somme des longueurs des arêtes de l'arbre :

$$\forall n_i \in N, i \in \{1, \ldots, n\} : longueur(\mathcal{A}_i) = \sum_{a \in \mathcal{A}_i} long(a)$$

Pour prendre en compte de façon globale la longueur de toutes les interconnexions du circuit, nous définissons le critère de **longueur totale des interconnexions** :

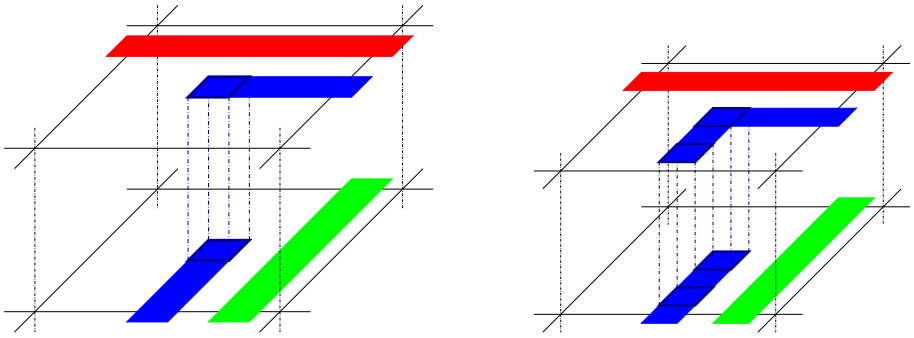
$$longueurTotale = \sum_{n_i \in N} longueur(\mathcal{A}_i)$$

Bien que réduire la longueur totale des interconnexions permet de diminuer les temps de propagation dans les fils et d'éviter les bruits parasites, cela peut conduire à une augmentation de la congestion. Les arbres d'interconnexion entre deux composantes connexes deviennent plus « directs », avec peu ou pas de détours, ce qui peut créer des zones congestionnées.

1.2 Formalisation du problème de routage global

Nombre de vias

Comme nous l'avons dit précédemment, les vias sont des éléments physiques dont le nombre total doit être réduit. En effet, ils sont très résistifs ce qui dégrade les performances temporelles du circuit. De plus pour les technologies les plus récentes, ils doivent être dédoublés, c'est à dire que deux vias sont placés côte à côte pour n'en former qu'un. Ce dédoublement occupe alors plus de ressources, obstruant plusieurs pistes de routage pour certaines couches de métal, comme l'illustre la figure 1.8.



(a) Via simple (b) Via doublé

FIGURE 1.8 – Exemple de doublement d'un via

Il faut aussi noter que la réduction du nombre total de via est un moyen pour le routeur global de réduire la congestion locale aux pavés.

Cette congestion locale (quantifiée par le rapport entre le nombre de ressources utilisées et le nombre de ressources disponibles à l'intérieur d'un pavé) n'est pas prise en compte dans le modèle approximé des ressources de routage du routeur global.

Seule la congestion sur les arêtes est prise en compte, ce qui peut conduire à des configurations valides pour le routeur global mais non réalisables pour le routeur détaillé.

Dans l'exemple suivant (voir figure 1.9), nous considérons une configuration de routage global comprenant un pavé couvrant trois couches de métal ($metal1$, $metal2$ et $metal3$) et dont les frontières horizontales (resp. verticales) couvrent deux pistes de routage sur chaque couche de métal à dominante verticale (resp. horizontale).

Nous représentons les portions des arbres d'interconnexion traversant le pavé. Celui-ci est représenté par les trois sommets $s1$, $s2$ et $s3$.
Sachant que chacune des arêtes à une capacité de 2, nous pouvons constater qu'aucune d'entre elles n'est en sur-congestion.

Si maintenant nous essayons de réaliser une affectation aux pistes de routage

pour les pavés associés aux sommets $s1$, $s2$ et $s3$, nous constatons qu'il n'existe pas de solution valide : l'impossibilité de trouver une solution valide provient du fait que dans le pavé associé au sommet $s2$, il n'y a pas assez de ressources de routage libres.

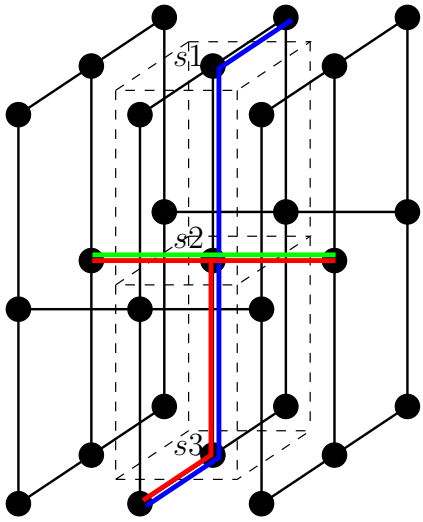
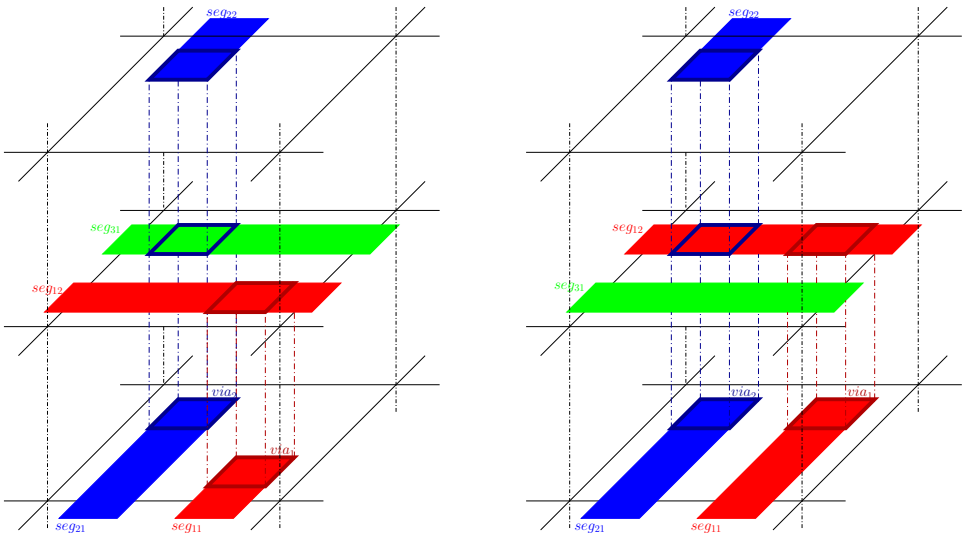


FIGURE 1.9 – Partie du graphe de routage contenant les sommets $s1$, $s2$ et $s3$

La figure 1.10 présente différentes configurations menant à une violation des ressources de routage du pavé associé à $s2$. Dans cet exemple, nous considérons trois nets matérialisés par les segments seg_{11} et seg_{12} pour le net n_1, seg_{21} et seg_{22} pour n_2 et enfin seg_{31} pour n_3.



(a) (b)

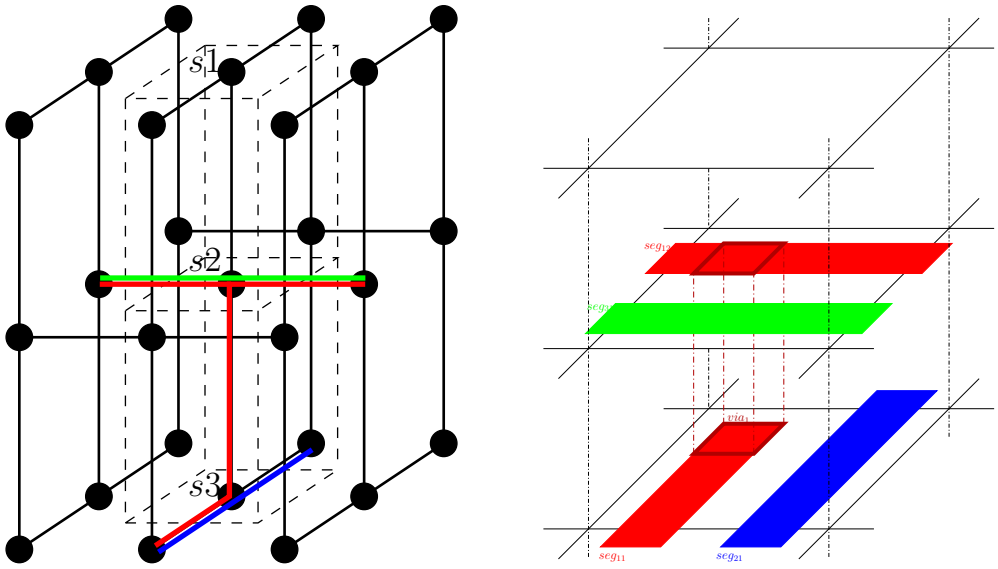
FIGURE 1.10 – Différentes affectations aux couches de métal possibles

1.2 Formalisation du problème de routage global

Quelle que soit la configuration, les segments seg_{12} et seg_{31} occupent les deux pistes de routage disponibles dans le pavé associé à $s2$ ne laissant aucune place pour le via_2. Or le net n_2 doit traverser la couche $metal2$ pour interconnecter les deux segments seg_{21} et seg_{22}.

Bien qu'il n'ait aucune information sur la congestion locale au pavé, le routeur global peut tenter de pallier ce problème en essayant de réduire le nombre de vias générés. Dans cet exemple, le routeur global peut générer un segment seg_{21} sur la couche $metal1$ traversant tout le pavé au lieu de deux segments (seg_{21} et seg_{22}) interconnectés par un via (via_2).

La configuration reste globalement la même, sauf qu'en réduisant le nombre de via, la congestion locale au pavé associé à $s2$ diminue ce qui permet au routeur détaillé de trouver une solution valide d'affectation aux pistes de routage (voir figure 1.11).



(a) Partie du graphe (b) Affectation aux pistes

FIGURE 1.11 – Diminution du nombre de vias

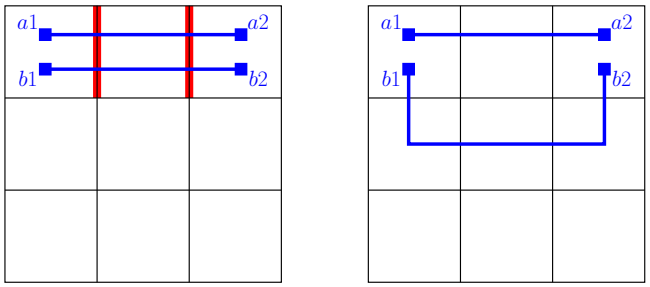
Ainsi, le routeur global en réduisant le **nombre de vias** qu'il génère peut éviter de créer des zones congestionnées locales à un pavé qui gênent le routeur détaillé.

Mais réduire le nombre de vias, revient à réduire les possibilités de changement de couches, c'est-à-dire l'utilisation des arêtes transversales pour les arbres d'interconnexion. La majorité des connecteurs se situant sur les deux premières couches de métal (les plus basses), ces dernières deviennent saturées tandis que les couches hautes se « vident », augmentant ainsi la congestion entre les pavés.

Compromis

L'optimisation de la longueur totale des fils d'interconnexion comme la réduction du nombre de vias, peuvent conduire à une augmentation de la congestion sur le graphe de routage. Ces deux derniers critères sont donc « antagonistes » avec celui de minimisation du dépassement total.

Les figures 1.12 et 1.13 illustrent ce phénomène à l'aide d'exemples très simples. L'exemple de la figure 1.12 considère deux nets a et b ayant chacun deux connecteurs à relier ($a1$,$a2$ et $b1$,$b2$). Chaque frontière du pavage couvre une piste de routage, la capacité des arêtes est 1.

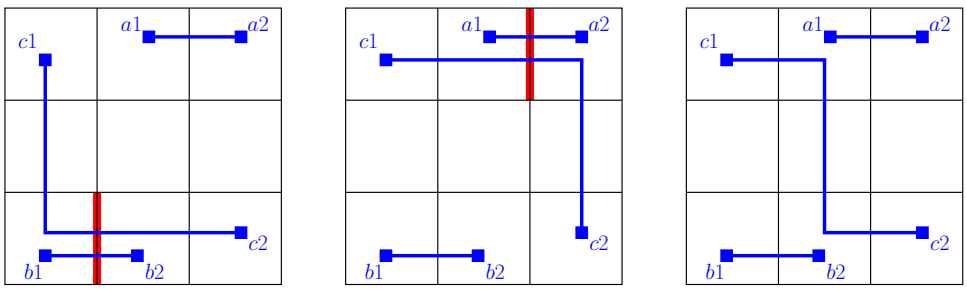


(a) (b)

FIGURE 1.12 – Compromis longeur vs congestion

Sur la figure 1.12(a), la longueur totale des fils d'interconnexion est minimale (une valeur de 4 exprimée en pas de grille), mais nous pouvons constater que le critère de congestion n'est pas respecté puisque les frontières marquées de rouge ont une occupation de 2. A l'inverse, sur la figure 1.12(b), il n'y a aucune sur-congestion, mais la longueur totale des fils d'interconnexion est plus grande (une valeur de 6).

Dans l'exemple de la figure 1.13, nous considérons trois nets : a, b et c ayant chacun deux connecteurs à relier ($a1$,$a2$, $b1$,$b2$ et $c1$,$c2$). La capacité des arêtes est de 1.



(a) (b) (c)

FIGURE 1.13 – Compromis nombre de vias vs congestion

1.2 Formalisation du problème de routage global

Les deux premières figures 1.13(a) et 1.13(b) présentent les deux configurations donnant des longueurs totales de fils d'interconnexion minimales (valeur de 1 pour les nets a et b, et 4 pour le net c. Soit une longueur totale minimale de 6). Là encore le critère de congestion n'est pas respecté puisque les frontières marquées de rouge ont une occupation de 2.

Comme précédemment, la dernière figure 1.13(c), présente une configuration sans sur-congestion. Cette fois la longueur est la même, mais le nombre de vias est accru (deux au lieu d'un). En effet pour passer d'une arête horizontale à une arête verticale (ou inversement), un via est nécessaire pour interconnecter les deux couches de métal et pour cette dernière configuration le net c présente deux changements de direction. Or comme nous l'avons dit, un trop grand nombre de vias peut conduire à une forte congestion locale susceptible d'empêcher le routeur détaillé de trouver une solution valide.

La recherche d'une « bonne » solution au problème de routage global équivaut donc à la recherche d'un bon compromis entre ces trois critères.

1.2.3 Autres modélisations

Occupation mémoire

Dans la partie précédente nous avons présenté le graphe de routage sous sa forme tridimensionnelle. Mais cette modélisation est assez gourmande en mémoire. En effet la taille des données pour la mettre en œuvre croît proportionnellement au nombre de couches de métal et au nombre de pavés.
Nous notons w le nombre de pavés pour la largeur du circuit, h celui pour sa hauteur et l le nombre de couches de métal utilisées.
Le nombre de sommets du graphe est :

$$|S| = w \times h \times l.$$

Le nombre total d'arêtes est la somme des nombres d'arêtes transversales, horizontales et verticales, où :

$$|Atransversales| = \quad w \times h \times (l-1)$$

$$|Ahorizontales| = \begin{cases} (w-1) \times h \times \frac{l}{2} & \text{si } l \text{ est pair} \\ (w-1) \times h \times \frac{(l-1)}{2} & \text{sinon} \end{cases}$$

$$|Averticales| = \begin{cases} w \times (h-1) \times \frac{l}{2} & \text{si } l \text{ est pair} \\ w \times (h-1) \times \frac{(l+1)}{2} & \text{sinon} \end{cases}$$

Notons que les termes en $\frac{l}{2}$ sont issus de l'alternance de direction privilégiée pour les couches de métal, les couches impaires ayant une dominante verticale et les couches

paires une dominante horizontale.

Plus concrètement, prenons l'exemple du circuit **newblue3**, faisant partie de la suite de benchmarks fournie pour les **Global Routing Contest** de l'**ISPD** en 2007 et 2008 [oPDGRCa]. Ce circuit est constitué de 973 pavés de large pour 1256 de haut et utilise 6 couches de métal. Le nombre d'éléments formant le graphe de routage (sommets et arêtes du graphe) est supérieur à 20 millions.
En considérant que la représentation mémoire d'un sommet occupe 64 octets et celle d'une arête 128, la quantité de mémoire utilisée pour représenter le graphe de routage avoisine les 2 giga-octets (Go) sur un processeur 32 bits et dépasse les 3,2 Go sur un processeur 64 bits.

Les technologies nanométriques actuelles permettent l'utilisation d'un plus grand nombre de couches de métal avec une taille du circuit qui augmente (relativement à la taille d'une cellule de bibliothèque pré-caractérisée). Il n'est donc pas absurde d'imaginer un graphe de taille 2000 par 2000 sur 10 couches de métal. En revanche il semble difficile de le représenter en mémoire et de l'utiliser convenablement puisque la quantité de mémoire nécessaire dépasse les 11 Go en 32 bits et 18 Go en 64 bits.

Graphe de routage bidimensionnel

Pour pallier à ce problème d'occupation mémoire, nous définissons un graphe de routage **bidimensionnel**. Soit $G(S, A)$ le graphe tel que :

- il existe un sommet $s \in S$ pour chaque pavé, un seul sommet représentant toutes les couches de métal.
- il existe une arête $a_{ij} \in A$ reliant les sommets s_i et s_j s'ils sont associés à des pavés adjacents et qu'il existe des ressources de routage permettant de passer de l'un à l'autre.

Le graphe $G(S, A)$ est une projection du graphe de routage tridimensionnel que nous avons précédemment défini dans un plan. Les sommets représentant un même pavé pour plusieurs couches de métal sont fusionnés en un seul, tout comme les arêtes. La capacité d'une arête dans le graphe bidimensionnel est égale à la somme des capacités des arêtes correspondantes dans le graphe tridimensionnel.

Les couches de métal ne sont plus différenciées si ce n'est de par leur direction privilégiée. Les arêtes verticales (resp. horizontales) du graphe correspondent à des couches de métal à dominante verticale (resp. horizontale), c'est-à-dire des couches paires (resp. impaires).

Par construction le graphe de routage bidimensionnel est connexe. De plus, les notions définies pour le graphe tridimensionnel (invariabilité de la capacité, longueur

d'une arête, arbre d'interconnexion, composante connexe, ...) restent valables avec les mêmes définitions pour le graphe bidimensionnel $G(S, A)$.

Le nombre d'éléments du graphe est fortement réduit puisqu'il ne dépend plus du nombre de couches de métal.
Avec les mêmes notations que précédemment, le nombre d'éléments est :

$$\begin{aligned} |S| = &\quad w \times h \\ |A| = &\quad (w-1) \times h + w \times (h-1) \end{aligned}$$

Pour un graphe de routage bidimensionnel de 2000 pavés par 2000, on obtient une occupation mémoire de l'ordre du Go en 32 bits, et de 2 Go en 64 bits.

Typiquement sur un graphe bidimensionnel, un arbre d'interconnexion est en fait un arbre de Steiner rectilinéaire.

Définition 1.3.

Un **arbre de Steiner** est une solution au problème d'optimisation combinatoire consistant à construire un arbre $\mathcal{A}$ reliant un ensemble de sommets S. Contrairement à l'**arbre couvrant** (qui est aussi une solution), l'arbre de Steiner s'autorise à utiliser d'autres sommets que ceux de S, appelés **nœuds de Steiner** ou **points de Steiner**.
Dans le cas d'un arbre de Steiner **rectilinéaire**, les arêtes ont la particularité qu'elles appartiennent à un espace à norme de Manhattan (et non pas l'espace euclidien).

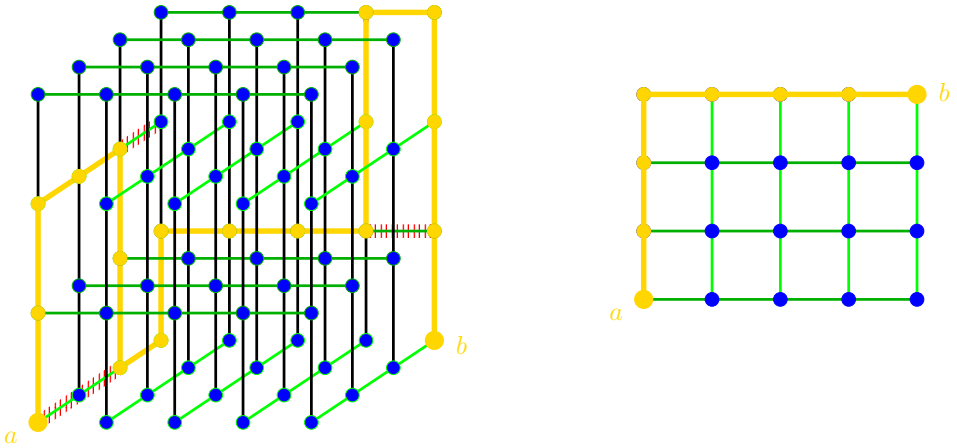
Affectation aux couches de métal

L'utilisation d'un graphe de routage bidimensionnel permet non seulement un gain non négligeable en terme d'occupation mémoire mais aussi une simplification des algorithmes de routage. Il est en effet plus simple de trouver un arbre d'interconnexion respectant les capacités des arêtes lorsque l'on considère un ensemble d'arêtes à grande capacité sur une même couche (cas du graphe bidimensionnel), plutôt qu'un ensemble d'arêtes de petite capacité réparties sur plusieurs couches de métal (cas du graphe tridimensionnel).

L'agrégation des couches de métal de même type (paires ou impaires) en une seule, réduit la taille du graphe de routage et par conséquent le nombre de solutions possibles à explorer. Les algorithmes de routage sont ainsi significativement accélérés.

Mais en contre-partie de ces améliorations, il devient impossible de connaître le nombre exact de vias générés ainsi que les couches qu'ils relient.
La figure 1.14(a) montre un exemple d'arbre interconnectant les sommets a et b (représentant des connecteurs situés sur la couche $metal1$) sur le graphe tridimensionnel.

Des détours ont été réalisés pour éviter d'utiliser les arêtes sur-congestionnées.
L'arbre illustré sur la figure 1.14(b) représente la même interconnexion entre les sommets a et b mais sur un graphe de routage bidimensionnel.



(a) Représentation tridimensionnelle (les arêtes hachurées en rouge sont sur-congestionnées)

(b) Projection bidimensionnelle

FIGURE 1.14 – Représentations d'un arbre d'interconnexion

Les arêtes congestionnées présentées sur le graphe de la figure 1.14(a) ne sont plus visibles puisque les couches de métal ne sont plus considérées individuellement.

Sur la représentation tridimensionnelle, il est très simple de quantifier le nombre de vias et les couches de métal qu'ils relient. En effet, chaque succession d'arêtes transversales de l'arbre d'interconnexion représente un via, comme l'illustre la figure 1.15 sur laquelle nous avons annoté les vias sur l'arbre.

Nous comptons donc :

- un via $metal1$ - $metal2$ ($via3$),
- deux vias $metal1$ - $metal3$ ($via1$ et $via2$),
- un via $metal2$ - $metal4$ ($via4$),
- un via $metal1$ - $metal4$ ($via5$).

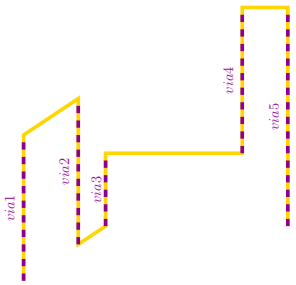


FIGURE 1.15 – Graphe de routage

Sur la figure 1.14(b), nous pouvons compter au maximum trois vias :

- un via $metal1$ - $metalX$ pour relier le connecteur du sommet a. Dans le cas où la connexion se fait sur la couche $metal1$, il n'y a pas de via. Sinon la couche $metalX$ représente une couche impaire différente de métal1.
- un via $metalX$ - $metalY$ représentant le coude où $metalX$ est une couche impaire (dominante verticale) et $metalY$ une couche paire (dominante horizontale)
- un via $metal1$ - $metalY$ pour relier le connecteur du sommet b. La couche $metalY$ étant paire, il y a donc toujours un via pour relier le connecteur b situé sur la couche impaire $metal1$.

Des cinq vias que nous comptions sur le graphe tridimensionnel, nous n'en dénombrons plus que trois au maximum. Ceux générés précédemment pour éviter les arêtes congestionnées n'ont aucune correspondance. Plus exactement les vias servant à relier des arêtes de même type (horizontales ou verticales) sur des couches différentes sont « absorbés » par la projection bidimensionnelle. De plus, il n'est plus possible de connaître les deux couches reliées par un via.

Ce problème met en évidence le fait qu'il est nécessaire d'associer au routage global sur un graphe bidimensionnel une phase supplémentaire qui partant d'une solution bidimensionnelle reconstruit une solution tridimensionnelle. Cette phase est appelée **affectation aux couches de métal** (*layer assignment*).

A première vue, une méthode simple consiste à transposer chaque segment d'un arbre d'interconnexion du graphe bidimensionnel dans le graphe tridimensionnel.

Définition 1.4.

Un **segment d'un arbre d'interconnexion** est un ensemble d'arêtes consécutives de même direction (verticale ou horizontale) reliant des sommets de degré 1 ou 2. Ce degré est calculé sur l'arbre d'interconnexion et non pas le graphe $G(S, A)$, c'est-à-dire que seules les arêtes utilisées par l'arbre d'interconnexion et incidentes au sommet sont considérées.

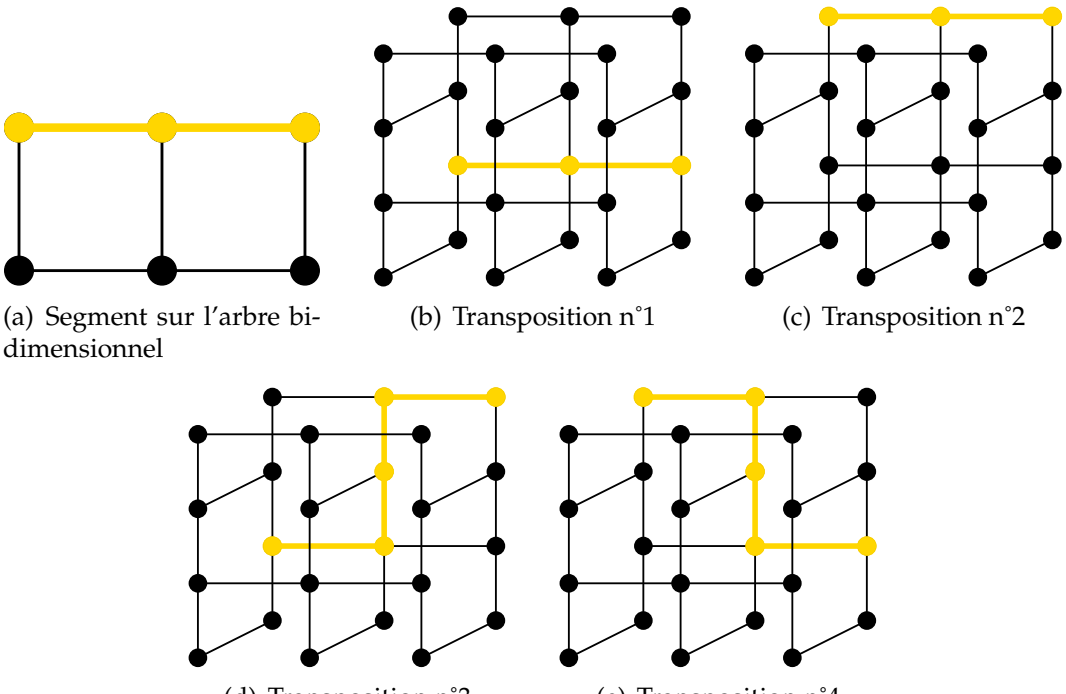
Dans le reste de cette section, lorsque nous parlons de segment d'un graphe, nous désignons un segment d'un arbre d'interconnexion ainsi défini.

L'idée simple consistant à transposer tel quel chaque segment du graphe bidimensionnel sur une couche du graphe tridimensionnel ne permet pas de trouver une solution valide. Assez rapidement, il devient impossible de transposer un segment complet sur une unique couche du graphe tridimensionnel en respectant les ressources de routage. Or la méthode d'affectation aux couches de métal doit préserver la validité d'une solution.

Définition 1.5.

Une solution de routage global est dite **valide** lorsqu'elle ne contient aucune sur-congestion, c'est-à-dire que $\forall a \in A, dep(a) = 0$.

Partant d'une solution bidimensionnelle valide, la transposition doit créer une solution tridimensionnelle valide. Il est en fait préférable de considérer les arêtes composant un segment et de chercher à les transposer individuellement sur le graphe tridimensionnel. Chaque arête du graphe bidimensionnel correspond à plusieurs arêtes dans le graphe tridimensionnel. Ainsi, chaque segment du graphe bidimensionnel correspond à plusieurs chaînes d'interconnexion tridimensionnelles comme le montre la figure 1.16.



(a) Segment sur l'arbre bidimensionnel (b) Transposition n°1 (c) Transposition n°2

(d) Transposition n°3 (e) Transposition n°4

FIGURE 1.16 – Transpositions possibles d'un segment d'un graphe bidimensionnel vers un graphe tridimensionnel

Dans [RM08], l'auteur démontre mathématiquement que partant d'une solution bidimensionnelle valide, il est toujours possible de trouver une solution tridimensionnelle valide si le nombre de vias générés n'est pas restreint. Cette démonstration est faite à l'aide d'un algorithme de construction d'une solution tridimensionnelle tel l'algorithme 1.1.

Cet algorithme utilise la notion d'ensemble bipoint :

Définition 1.6.

Un **ensemble bipoint** est un arbre $\mathcal{B}_i$ associé à l'arbre d'interconnexion $\mathcal{A}_i$ d'un net $n_i \in N$.
Soit S_i l'ensemble des sommets de $\mathcal{A}_i$ et A_i l'ensemble de ses arêtes, on définit alors $\mathcal{B}_i$ tel que :

- l'ensemble des sommets V_i est constitué des sommets associés aux connecteurs de n_i et des éventuels noeuds intermédiaires de $\mathcal{S}_i$ ($V_i \subset S_i$),
- l'ensemble des arêtes B_i est tel que $(s, d) \in B_i$ s'il existe une chaîne de s dans d dans $\mathcal{A}_i$ constituée uniquement de sommets de $S_i \setminus V_i$.

Notons que pour chaque $\mathcal{A}_i$ l'ensemble bipoint $\mathcal{B}_i$ est unique.

L'algorithme d'affectation aux couches de métal suivant est du même type que celui utilisé par l'auteur de [RM08] :

Algorithme 1.1 Affectation aux couches de métal

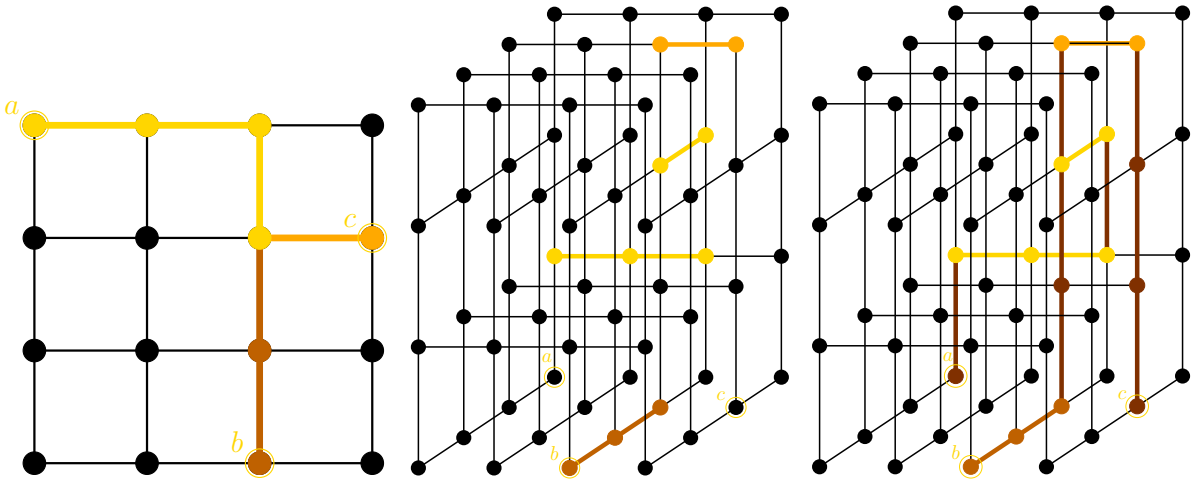
Entrée : Une solution de routage global $\mathcal{S}$ valide, constituée d'un ensemble d'arbres d'interconnexion $\mathcal{A} = \{\mathcal{A}_i, i \in \{1, \ldots, n\}\}$ sur un graphe bidimensionnel $G(S, A)$, le graphe tridimensionnel $G_{3d}(S_{3d}, A_{3d})$ associé.

Sortie : Une solution de routage tridimensionnel $\mathcal{S}_{3d}$ valide, constituée de l'ensemble d'arbres d'interconnexion $\mathcal{A}_{3d} = \{\mathcal{A}_{3di}, i \in \{1, \ldots, n\}\}$, tel que pour chaque arbre d'interconnexion $\mathcal{A}_{3di}$, l'arbre d'interconnexion $\mathcal{A}_i$ soit une projection bidimensionnelle de $\mathcal{A}_{3di}$ sur $G(S, A)$.

pour tout $\mathcal{A}_i \in \mathcal{A}$ **faire**
  Décomposer $\mathcal{A}_i$ en un ensemble B de bipoints.
  pour tout $(s1, s2) \in B$ **faire**
    $sommetCourant$:= $s1$
    pour tout arête a de la chaîne interconnectant $s1$ et $s2$ **faire**
      Transposer a sur la couche de métal la plus proche de $sommetCourant$ et induisant le moins de congestion possible.
      $sommeCourant$:= $a.getOpposite(sommetCourant)$
    fin pour
  fin pour
  Rajouter les arêtes représentant les vias pour former l'arbre $\mathcal{A}_{3di}$.
fin pour

La figure 1.17 présente le déroulement de cet l'algorithme. Sur la figure 1.17(a) nous avons représenté l'arbre d'interconnexion sur le graphe bidimensionnel. Chacune des

chaînes représentant un des trois bipoints de l'arbre possède une couleur. La transposition des arêtes de ces chaînes est visible sur la figure 1.17(b) puis l'arbre tridimensionnel résultant est présenté sur la figure 1.17(c).



(a) Arbre bidimensionnel (b) Affectation des segments (c) Arbre tridimensionnel construit

FIGURE 1.17 – Illustration de l'algorithme 1.1

Le problème de cet algorithme est qu'il génère un grand nombre de vias. L'auteur de [RM08] propose alors d'exécuter une itération de *ripup & reroute* sur chaque net pour réduire le nombre de vias. Les résulats présentés dans [RM07] (section 5.4) confirment que cette approche donne des résultats équivalents au routage tridimensionnel (coût variant de -2.1% à 2.8%) mais en réduisant fortement le temps d'exécution (au minimum 33% plus rapide).

On peut noter que les routeurs globaux académiques actuels : Archer [OW07], BoxRouter 2.0 [CLYP07], FGR [RM07], MaizeRouter [Mof08], NTHU-Route [GWW08] utilsent la même approche. Ils projettent dans un premier temps le problème tridimensionnel original sur un graphe de routage bidimensionnel, puis réalisent un routage global avant de convertir la solution bidimensionnelle obtenue en une solution tridimensionnelle à l'aide d'une méthode d'affectation aux couches de métal telle que celle précédemment décrite.

Graphe de routage irrégulier

Comme nous l'avons évoqué précédemment, le pavage servant de base au graphe de routage n'est pas toujours régulier, notamment lorsque le routeur global est utilisé pour guider la phase de placement global.

Pendant cette phase, la taille des pavés est variable mais est plus grande que

celle utilisée dans le cas d'un routage global intervenant juste avant le routage détaillé. L'approximation du modèle de ressources est donc plus importante : les capacités des arêtes sont plus importantes ainsi que le nombre de nets locaux à chaque pavé.

Hormis sa structure non régulière, ce graphe est défini et utilisé de la même façon que précédemment.

Chaque sommet représente un pavé et chaque arête représente les ressources disponibles entre les deux pavés associés aux sommets qu'elle relie. Le nombre de ces ressources disponibles est stocké dans la capacité de l'arête.

Pour chaque net, les connecteurs à relier sont représentés par un ensemble de composantes connexes, qui une fois reliées par un chemin sans cycle, forment l'arbre d'interconnexion du net.

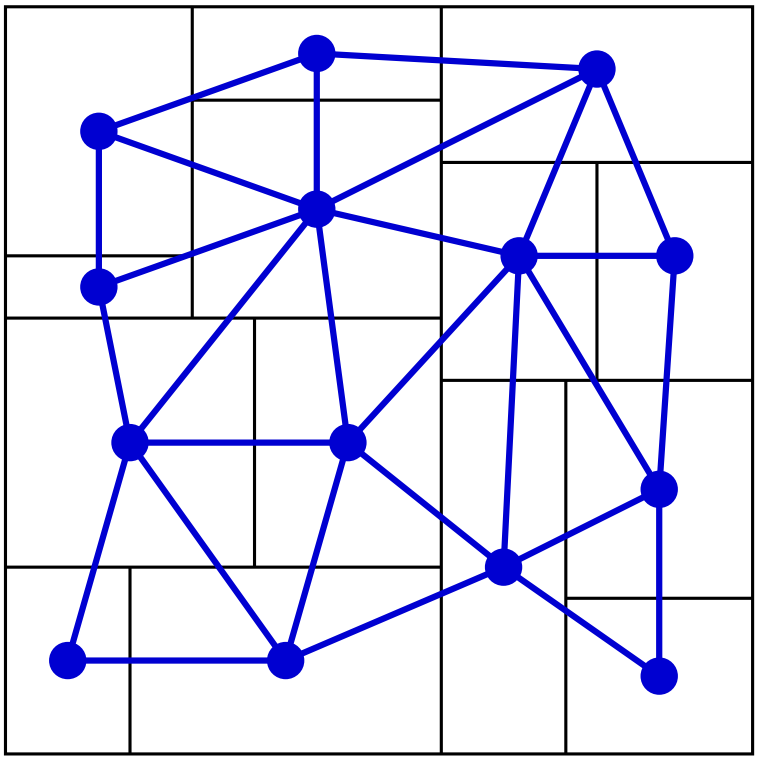


FIGURE 1.18 – Exemple de graphe de routage bidimensionnel irrégulier

1.3 Approches de résolution

Toutes les méthodes permettant de résoudre le problème de routage global utilisent le même schéma en deux phases :

Phase 1 : Construction d'une solution de routage global (*ie* un ensemble d'arbres d'interconnexion $\mathcal{A}_i, i \in \{1, \ldots, n\}$) offrant un bon compromis entre longueur totale des interconnexions, nombre de vias et congestion.

Phase 2 : Amélioration de la solution contruite en éliminant toute sur-congestion pour permettre au routeur détaillé de finaliser le routage.

Chaque méthode offre une mise en œuvre différente de chaque phase, mais globalement ces deux phases sont toujours présentes.
Dans cette partie nous allons tout d'abord présenter les deux grandes familles de méthodes permettant de résoudre la *phase 1* : les méthodes concurrentes et les méthodes séquentielles puis nous nous intéresserons au *ripup & reroute* qui permet de résoudre la *phase 2*.

1.3.1 Approches concurrentes

Une première façon de traiter le problème est de considérer simultanément l'ensemble des nets à router, de construire un ensemble de solutions $\mathcal{S} = \{\mathcal{S}_j, j \in \{1, \ldots, m\}\}$ et de choisir la meilleure solution, m étant le nombre de solutions construites.

Ces approches sont appelées approches **concurrentes** puisqu'elles traitent simultanément tous les nets à router. Parmi ces approches, il existe deux formulations usuelles du problème, une à base de programmation linéaire en nombres entiers (*Integer Linear Programming ILP*) [CP06] [MÖ6] [HRM08] [YAV07] et une autre à base de flots de transport (*Multi-commodity flow*) [Alb01]

Programmation Linéaire en Nombres Entiers (PLNE)

Les méthodes à base de PLNE commencent par construire pour chaque net $n_i \in N, i \in \{1, \ldots, n\}$ un ensemble d'arbres d'interconnexion $\mathcal{S}t\mathcal{A}_i = \{\mathcal{A}_{i,1}, \ldots, \mathcal{A}_{i,k_i}\}, i \in \{1, \ldots, n\}$ et k_i représente le nombre d'arbres construits pour le net n_i.

A chaque arbre $\mathcal{A}_{i,k}$, $i \in \{1, \ldots, n\}$ et $k \in \{1, \ldots, k_i\}$ est associée une variable $x_{i,k} \in \{0, 1\}$ telle que :
- $x_{i,k} = 1$ si l'arbre $\mathcal{A}_{i,k}$ est sélectionné pour le net n_i,

- $\sum_{k \in \{1,\ldots,k_i\}} x_{i,k} = 1$ pour $i \in \{1,\ldots,n\}$, c'est-à-dire qu'il ne peut y avoir qu'un seul arbre sélectionné pour un même net.

Le coût de chaque arbre est noté $cost(\mathcal{A}_{i,k})$ et prend en compte les trois critères précédemment cités.

Les techniques de PLNE cherchent à minimiser :

$$\sum_{(i,k) \in \{1,\ldots,n\} \times \{1,\ldots,k_i\}} cost(\mathcal{A}_{i,k}) x_{i,k}$$

L'étude [YAV07] présente plus en détails la formulation en PLNE et les méthodes utilisées pour prendre en compte les différents critères de coût.

L'inconvénient de ces méthodes est que le temps de résolution en PLNE augmente exponentiellement avec le nombre d'arbres construits k_i pour chaque net $n_i \in N$. Or pour assurer la faisabilité et la qualité de la solution il faut que ce nombre d'arbres considérés soit suffisamment grand, pour que les algorithmes de PLNE puissent trouver une solution respectant toutes les contraintes.

Multi flots

Une autre formulation utilisée dans les approches concurrentes est celle considérant le problème de routage global comme un problème de multi-flots (*Multi-commodity flow*).

Chaque net $n_i \in N, i \in \{1,\ldots,n\}$ est considéré comme une marchandise, associée à une demande (égale à 1 dans le cas du routage global), qu'il faut acheminer à travers un réseau de transport représenté par le graphe $G(S, A)$. Chaque arête $a \in A$ possède une capacité $cap(a)$ (telle que celle précédemment définie) ainsi qu'une variable $f(a)$ représentant la quantité de flot traversant l'arête.

Usuellement, les algorithmes de flots cherchent à minimiser le coût total de transport tout en respectant les contraintes de demande (la quantité de flots acheminée pour chaque net doit être égale à sa demande) et de capacité (pour chaque arête $a \in A$, la quantité de flot traversant l'arête $f(a)$ ne doit pas excéder sa capacité $cap(a)$).

Comme le montre l'étude [WGYM05], les méthodes à base de multi flots telles que celle présentée dans [Alb01] se montrent très efficaces pour résoudre des problèmes difficiles dans des petits circuits très denses. Mais leur temps d'exécution trop important les rend inutilisables pour des circuits plus grands.

Dans l'ensemble, les méthodes concurrentes, bien qu'elles donnent des résultats proches de l'optimum, ont une complexité et un temps d'exécution trop importants par rapport aux méthodes séquentielles plus traditionnelles.

1.3.2 Approches séquentielles

Les méthodes séquentielles considèrent les nets individuellement et successivement. Le problème de routage global est donc décomposé en sous problèmes consistant à trouver un arbre d'interconnexion $\mathcal{A}_i$ pour un net $n_i \in N$ respectant les critères de longueur totale des fils d'interconnexion, nombre de vias et congestion.

Cette technique a l'avantage de grandement simplifier les algorithmes de traitement mais souffre d'un défaut fondamental dû au fait que les nets sont traités un par un. Il n'y a aucun retour ou anticipation des traitements futurs permettant d'éviter de créer des zones congestionnées.

Lors de la construction d'un arbre, seule la congestion des arbres déjà construits est prise en compte. Les décisions prises lors de la création des premiers arbres d'interconnexion ne sont jamais remises en cause et peuvent conduire à des blocages empêchant d'obtenir une solution valide. Les méthodes séquentielles sont avant tout des méthodes gloutonnes.

Toutes les méthodes séquentielles fonctionnent en deux étapes consécutives ; la première ordonne les nets à router, tandis que la deuxième trouve un arbre d'interconnexion pour chaque net dans l'ordre précédemment défini.
De façon à obtenir une meilleure prise en compte de la congestion, certains routeurs ajoutent une étape intermédiaire permettant de calculer une estimation anticipée de la congestion du circuit. Nous présenterons ultérieurement l'utilisation de cette étape supplémentaire.

Ordonnancement des nets

La première étape des méthodes séquentielles est l'ordonnancement des nets. Elle détermine un ordre fixe pour le traitement des nets lors de la seconde étape.
Cet ordre, qui ne varie pas au fur et à mesure des traitements, est lié à la notion de **degré de liberté** d'un net. Ce degré de liberté est en quelque sorte une évaluation du nombre de configurations de routage envisageables. Plus le nombre de configurations (ou topologies) possibles à coût identique est grand et plus le degré de liberté augmente.

Les critères permettant de définir le degré de liberté d'un net sont assez variés. L'idée directrice est d'estimer le nombre de topologies, pour cela on peut utiliser :
- le nombre de connecteurs à relier du net,
- le demi-périmètre de la boîte englobante du net,
- la surface de la boîte englobante du net,
- ...

1.3 Approches de résolution

Prenons l'exemple de la **surface de la boîte englobante** (c'est-à-dire, le rectangle de surface minimale qui englobe tous les connecteurs à relier du net). Plus la surface est grande et plus il existe de topologies permettant d'éviter les éventuelles zones congestionnées. Sur l'exemple de la figure 1.19, malgré les arêtes congestionnées (hachurées en rouge) il existe plusieurs chaînes de longueur équivalente reliant a et b et contenues dans leur boîte englobante.

A l'inverse, pour des connecteurs concentrés sur une petite surface, il peut être difficile voire impossible de trouver un arbre de longueur raisonnable qui respecte les ressources de routage. Sur l'exemple de la figure 1.20, il n'existe aucune chaîne reliant c et d contenue dans leur boîte englobante. Du fait des arêtes congestionnées, le routeur global aura tendance à créer une chaîne sortant de la boîte englobante. Il est donc préférable de traiter ce type de net en premier pour éviter que le routeur global soit gêné par la congestion et construise des chaînes plus longues.
Le degré de liberté peut donc être directement proportionnel à la surface de la boîte englobante des connecteurs.

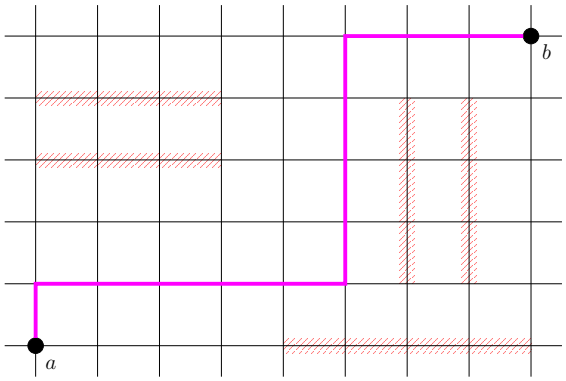


FIGURE 1.19 – Surface de la boîte englobante

La surface de la boîte englobante est suffisante pour trouver un chemin sans congestion.

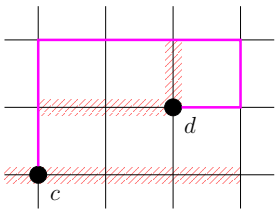


FIGURE 1.20 – Surface de la boîte englobante

Un détour est nécessaire pour relier les deux sommets.

Les distances étant normalisées par rapport au pas de grille du graphe de routage, les nets « plats », c'est-à-dire ceux dont les connecteurs à relier sont alignés (horizontalement ou verticalement), ont une surface de boîte englobante nulle. Ils sont alors tous égaux suivant le premier critère d'ordonnancement défini.

Bien sûr, les algorithmes ont tout intérêt à router ces nets « au plus court » pour minimiser à la fois la longueur de l'interconnexion mais aussi le nombre de vias et réduire les risques de création de zones congestionnées. Mais il serait intéressant de pouvoir les trier.

Pour cela nous définissons le critère de **facteur de forme d'un net**, défini par le

produit : $(h+1) \times (l+1)$ où h est la hauteur de la boîte englobante des connecteurs à relier et l sa largeur. Le facteur de forme est une sorte de « surface expansée », permettant ainsi d'ordonnancer les nets plats.

Dans le cas d'un routage global guidé par des contraintes temporelles, la **criticité** des nets est un autre critère important. La criticité d'un net est fonction du temps de propagation maximum acceptable sur les fils d'interconnexion pour que le circuit respecte les performances souhaitées.
Les nets critiques doivent donc être routés « au plus court » de façon à réduire ce temps de propagation et optimiser le chemin critique du circuit. Ils ont donc un faible degré de liberté. La criticité peut ne s'appliquer qu'à une portion d'un net, pour optimiser l'interconnexion entre un émetteur et un ou plusieurs récepteurs.

A l'inverse, les algorithmes n'ont pas besoin d'optimiser la longueur des arbres construits pour les nets non critiques (à fort degré de liberté), ils peuvent même l'augmenter pour éviter des zones congestionnées sans que les performances du circuit ne diminuent.

Pour guider le choix d'un bon critère d'ordonnancement parmi ceux cités, nous pouvons noter que le critère doit être rapide à calculer pour ne pas ralentir l'algorithme de routage global et qu'en pratique un critère « simple » tel que celui de facteur de forme donne de bon résultats.

Méthodes usuelles de construction d'arbres de Steiner

La deuxième étape des méthodes séquentielles consiste à construire un arbre d'interconnexion pour chaque net $n_i \in N, i \in \{1, \ldots, n\}$ dans l'ordre fixé par la première étape.

Le problème de construction d'un arbre d'interconnexion est en fait un problème de construction d'arbre de Steiner, on peut se demander s'il est possible / performant d'utiliser les méthodes usuelles de construction d'arbre de Steiner.

Il existe en effet plusieurs outils performants permettant de construire en temps logarithmique (par rapport au nombre de sommets à relier) des arbres euclidiens ou rectilinéaires de longueur proche de l'optimum. Citons notamment FLUTE [Chu04] [fRCE] qui permet de construire très rapidement des arbres de Steiner rectilinéraires de longueur minimale pour des nets ayant au maximum 9 connecteurs et proche de l'optimum au-delà, et FastSteiner [RA] qui peut construire en seulement quelques secondes un arbre de Steiner proche de l'optimum reliant plusieurs dizaines de milliers de connecteurs.

Malheureusement ces outils ne construisent un arbre quasi-optimal qu'en terme

de longueur de l'arbre. En effet toutes les méthodes de construction d'arbres de Steiner se basent sur une métrique, les sommets à relier sont vus comme des points dans un plan dans lequel l'inégalité triangulaire est vérifiée. Or dès que l'on cherche à intégrer le critère de congestion dans le graphe, le coût des arêtes ne vérifie plus cette inégalité et il n'est donc plus possible d'utiliser ces algorithmes pour construire un arbre d'interconnexion.

Le routeur global « FastRoute » [PC06] présente une technique de prise en compte de la congestion dans la construction d'arbres de Steiner.

Cette technique utilise le modèle de la grille de Hanan [Han66]. Cette grille est formée par l'ensemble des lignes verticales et horizontales traversant les sommets correspondant aux connecteur à relier du net (voir figure 1.21 en bleu). De plus, le théorème de Hanan énonce qu'il existe un arbre de Steiner optimum dont tous les sommets intermédiaires (points de Steiner) sont situés sur la grille de Hanan.
L'utilisation de la grille de Hanan permet donc de simplifier et surtout d'accélérer les algorithmes de construction d'arbres de Steiner optimaux en limitant l'espace de recherche des solutions.

L'idée conductrice du routeur global « Fastroute » est de conserver l'intérêt apporté par la grille de Hanan tout en essayant de gérer et réduire la congestion du circuit.
Lors du traitement de chaque net à router, une congestion moyenne est calculée pour chaque ligne et chaque colonne formée par la grille de Hanan associée (voir figure 1.21). En fonction de cette congestion moyenne, le graphe de routage est virtuellement déformé pour que le calcul de la longueur des arbres prenne en compte la congestion.
La congestion moyenne d'une colonne (resp. ligne) est calculée en fonction de la somme des occupations et capacités des arêtes horizontales (resp. verticales) couvertes par celle-ci. Pour définir la congestion moyenne, nous notons :

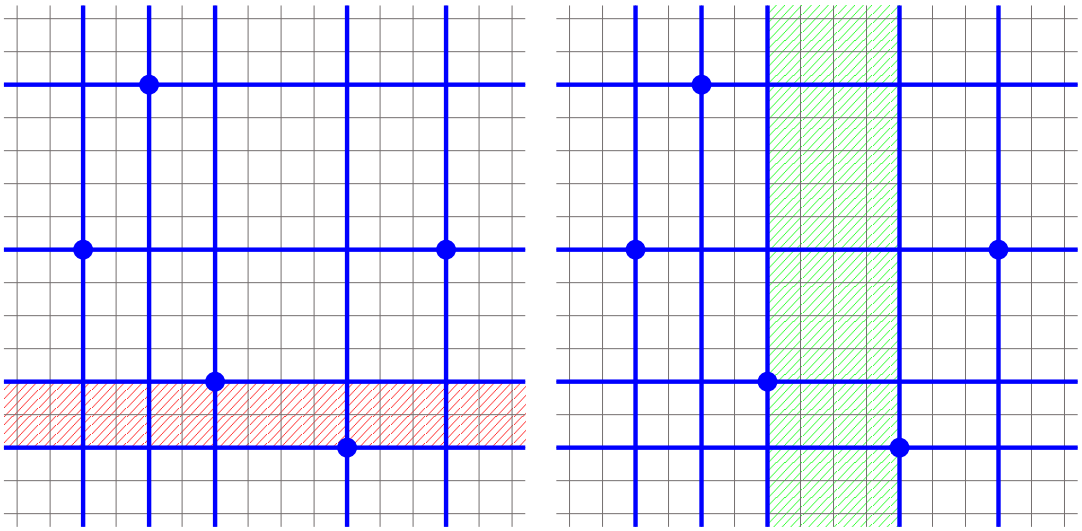
- $\mathcal{H}$ l'ensemble des arêtes horizontales couvertes par une colonne de la grille de Hanan
- $\mathcal{V}$ l'ensemble des arêtes verticales couvertes par une ligne de la grille de Hanan

Et nous définissons :

$$congestionMoyenne(colonne) = \frac{\sum_{a\in\mathcal{H}} occ(a)}{\sum_{a\in\mathcal{H}} cap(a)}$$

$$congestionMoyenne(ligne) = \frac{\sum_{a\in\mathcal{V}} occ(a)}{\sum_{a\in\mathcal{V}} cap(a)}$$

La congestion est en quelque sorte « convertie » en longueur, ce qui permet l'utilisation d'un algorithme de construction d'arbres de Steiner rectilinéaires de longueur minimale pour construire l'arbre d'interconnexion du net.



(a) Une ligne (hachurée en rouge) (b) Une colonne (hachurée en vert)

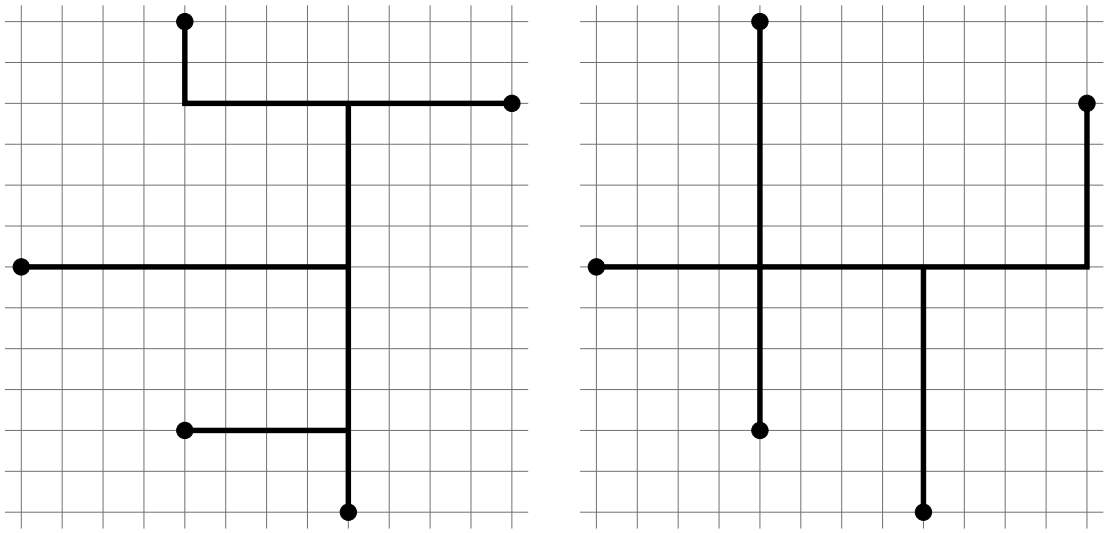
FIGURE 1.21 – Exemple de ligne et de colonne de la grille de Hanan

Il est important de noter que la déformation appliquée au graphe de routage n'est valable que pour le net en cours de traitement.

Prenons un exemple pour illustrer cette méthode. Les figures 1.22(a) et 1.22(b) présentent deux exemples d'arbres de Steiner reliant les cinq connecteurs du net considéré. Ces deux arbres sont de longueur minimale équivalente (32 en unités normalisées).
Nous allons calculer la congestion moyenne d'une colonne, puis déformer le graphe de routage en fonction de cette congestion et voir l'impact sur le choix des arbres de Steiner à priori équivalents.
Sur la figure 1.23(a), on considère la grille de Hanan associée aux cinq connecteurs du net considéré. La zone hachurée représente la colonne dont nous souhaitons calculer la congestion moyenne. Ce calcul prend en compte l'occupation et la capacité de toutes les arêtes horizontales couvertes par la colonne (en pointillés rouge sur la figure 1.23(a)).

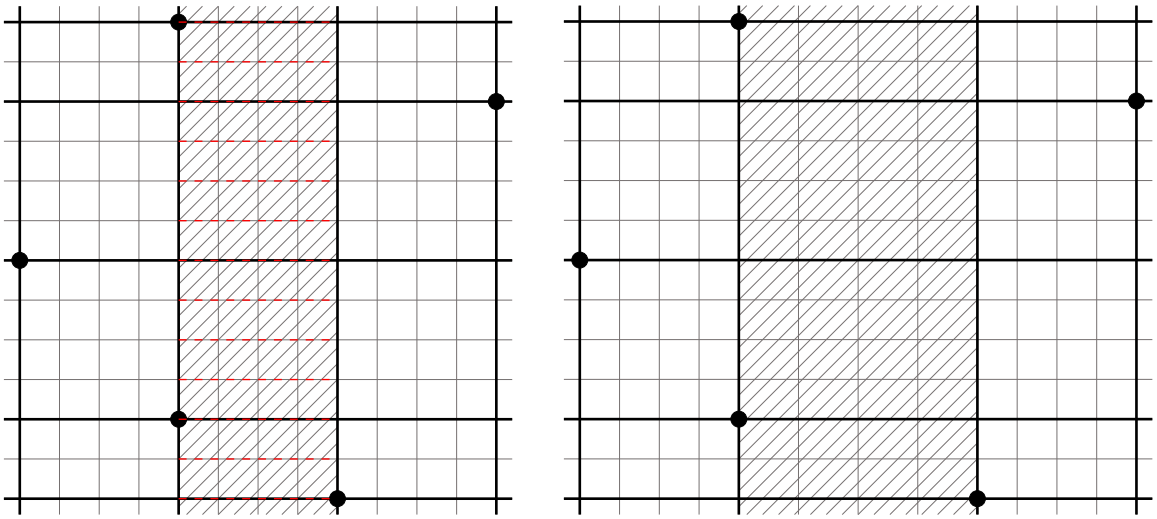
La largeur de la colonne est alors déformée proportionnellement à la congestion moyenne, plus il y a de congestion et plus la largeur augmente, comme l'illustre la figure 1.23(b).

Si l'on reporte les deux arbres des figures 1.22(a) et 1.22(b) sur le graphe déformé 1.24(a) et 1.24(b), ils n'ont plus des longueurs équivalentes, 38 unités normalisées pour le premier et 34 pour le deuxième. Ce dernier est meilleur puisqu'il utilise moins d'arêtes traversant la colonne congestionnée.



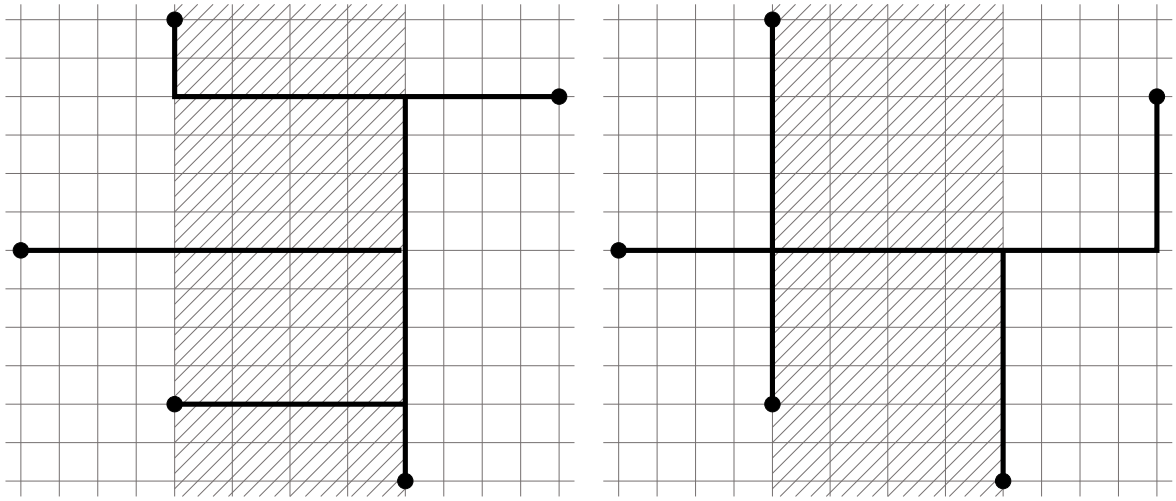
(a) Arbre d'interconnexion $\mathcal{A}_1$ (b) Arbre d'interconnexion $\mathcal{A}_2$

FIGURE 1.22 – Exemple de deux arbres d'interconnexion de longueur équivalente



(a) Congestion moyenne (b) Déformation associée

FIGURE 1.23 – Congestion moyenne d'un colonne et déformation associée



(a) Arbre d'interconnexion $\mathcal{A}_1$ reporté sur le graphe déformé

(b) Arbre d'interconnexion $\mathcal{A}_2$ reporté sur le graphe déformé

FIGURE 1.24 – Exemple de déroulement de l'algorithme de Fastroute

Bien que cette technique offre une possibilité pour prendre en compte la congestion dans les algorithmes de construction d'arbre de Steiner, l'utilisation d'une congestion moyenne (et non d'une congestion exacte pour chaque arête) implique que les arbres construits ne sont pas optimaux en terme de congestion. Il est donc nécessaire d'avoir des méthodes supplémentaires pour modifier les arbres et améliorer la congestion.

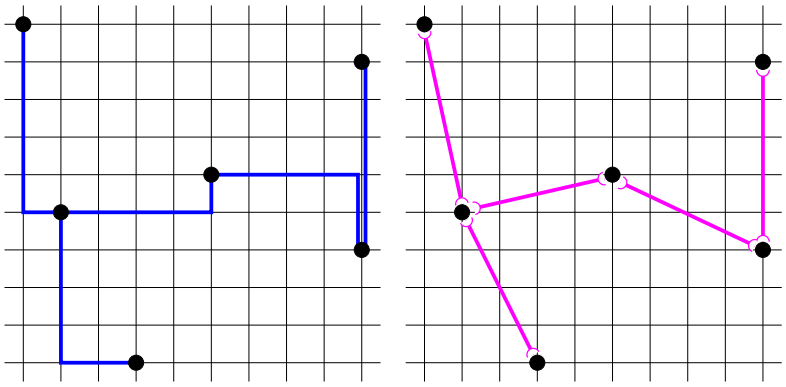
Méthodes de construction d'arbres de Steiner pour le routage global

Les algorithmes utilisés par les méthodes dédiées au routage global construisent une chaîne de coût minimal reliant un sommet source s à un sommet destination d sur le graphe $G(S, A)$. Il est donc nécessaire de créer un ensemble bipoint $\mathcal{B}_i$ pour chaque net $n_i \in N$.
Cette construction, que l'on nomme **décomposition en bipoint**, est une phase d'initialisation pour les méthodes dédiées.

Décomposition en bipoints

L'arbre d'interconnexion $\mathcal{A}_i$ nécessaire à la construction de l'ensemble bipoint $\mathcal{B}_i$ (voir la définition 1.6) peut être un arbre couvrant rectilinéaire ou un arbre de Steiner rectilinéaire comme nous le présentons ci-après.

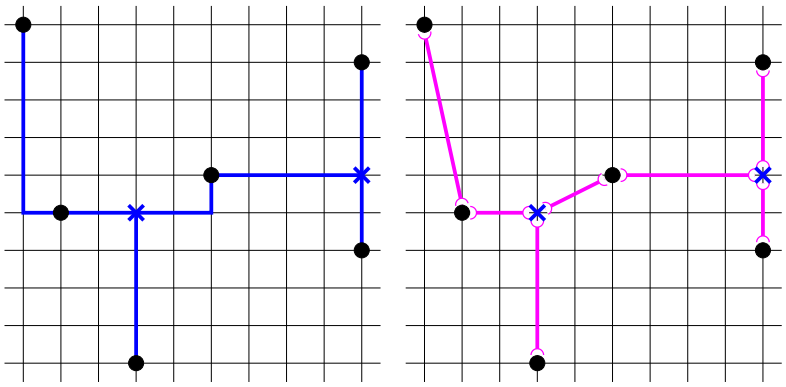
Dans le cas d'un arbre couvrant, la décomposition en bipoints est évidente comme le montre la figure 1.25.



(a) Arbre couvrant (b) Décomposition associée

FIGURE 1.25 – Arbre couvrant et décomposition en bipoints associée

Dans le cas d'un arbre de Steiner, certains sommets de S n'appartenant pas à T_i, sont utilisés pour obtenir un arbre de longueur optimale (représentés par des croix bleues sur le figure 1.26). Ces sommets sont appelés nœuds de Steiner et sont considérés comme des sommets supplémentaires de T_i.
Les bipoints de B_i, les prennent donc en compte, comme le montre la figure 1.26.



(a) Arbre de Steiner (b) Décomposition associée

FIGURE 1.26 – Arbre de Steiner et décomposition en bipoints associée

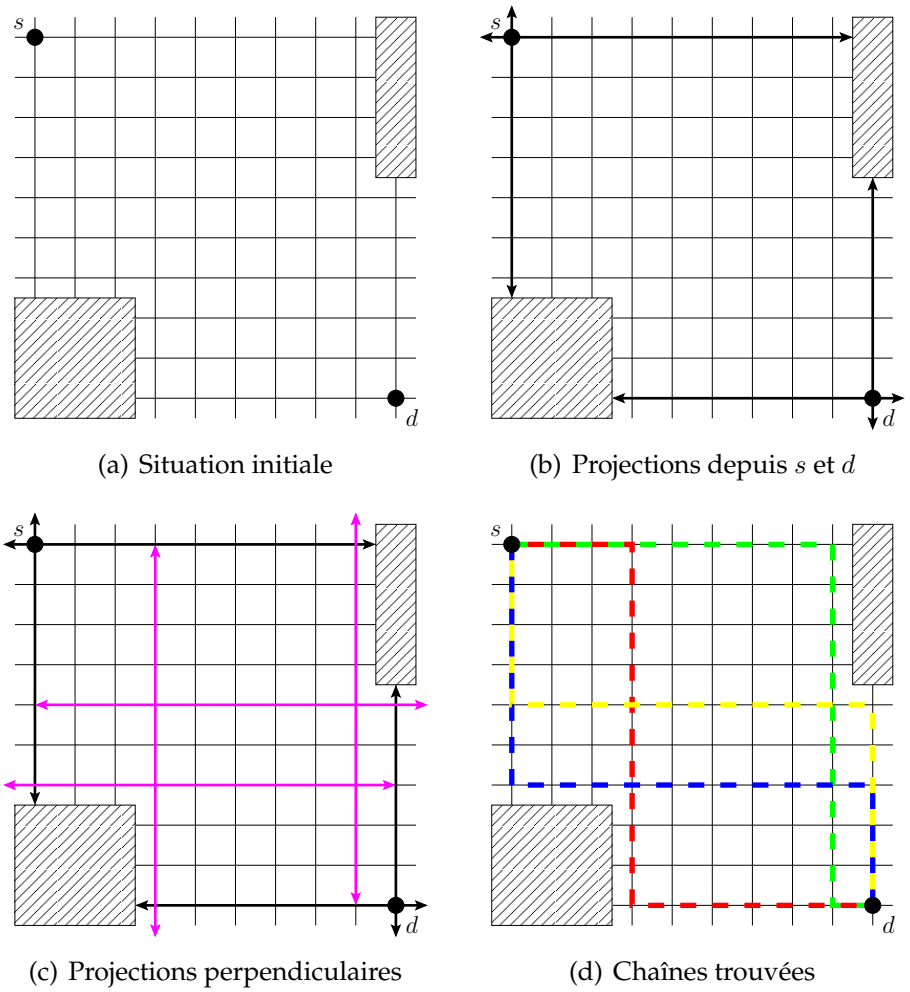
A priori la décomposition utilisant un arbre de Steiner est meilleure puisque, par définition, la longueur de l'arbre construit est inférieure à celle d'un arbre couvrant. Mais l'étude comparative menée dans [RM08] tend à montrer que, bien qu'ayant une longueur totale des fils d'interconnexion plus faible, le routage global issu d'une décomposition utilisant les arbres de Steiner utilise un nombre de vias plus important.

Pour l'ensemble des circuits de test de l'ISPD2007 [oPDGRCa], la longueur totale des fils d'interconnexion est réduite de 0.5% avec une décomposition en arbre de Steiner, tandis que le nombre de vias augmente de 1,8% et le temps d'exécution de

22%. Sachant que le temps total de construction d'arbres pour la décomposition est non significatif par rapport au temps d'exécution total du routeur global pour des circuits actuels, il est plus intéressant d'utiliser une décomposition utilisant des arbres couvrants.

Une fois la décomposition en bipoints effectuée, il faut construire une chaîne reliant les sommets de chaque bipoint $b \in B_i$ sur le graphe $G(S, A)$.

Routage par projection (*line probing*)
Une première possibilité est d'utiliser une technique d'exploration par projection (*Line probing*). A partir de la source s et de la destination d (correspondant au bipoint à traiter), des lignes sont projetées dans les 4 directions. Lorsqu'une ligne rencontre un obstacle, une ligne perpendiculaire est projetée. Lorsqu'une ligne issue de s intersecte une ligne issue de d, une chaîne d'interconnexion est trouvée.



(a) Situation initiale (b) Projections depuis s et d

(c) Projections perpendiculaires (d) Chaînes trouvées

FIGURE 1.27 – Déroulement de l'algorithme d'exploration par projection

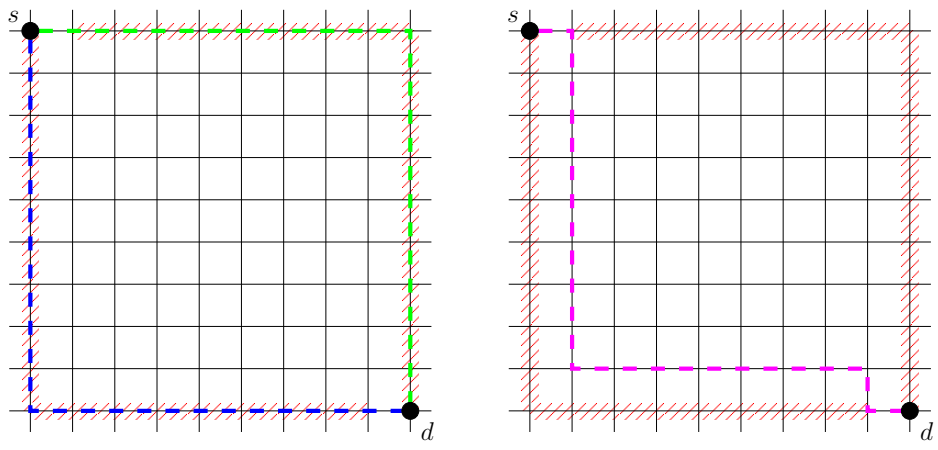
1.3 Approches de résolution

La figure 1.27 illustre le déroulement d'une exploration par projection. Les zones hachurées représentent des obstacles au routage.

Si plusieurs chaînes sont trouvées, l'algorithme choisit celle de coût minimal et dans le cas d'une égalité, le choix est fait arbitrairement.

Le coût dépend des trois critères précédemment cités mais la gestion de la congestion reste limitée du fait que seules les arêtes en sur-congestion peuvent faire obstacle aux lignes de projection.
La figure 1.28 illustre ce problème. Les arêtes hachurées en rouge ont un taux de congestion proche de 1 (mais inférieur), il est donc nécessaire de les éviter. Tel que nous avons décrit l'agortihme de *line probing*, les chaînes potentielles reliant s à d sont celles illustrées sur la figure 1.28(a). Ces deux chaînes ont un coût similaire assez important puisqu'elles traversent beaucoup d'arêtes congestionnées.

Pour relier s à d il est plus pertinent d'utiliser une chaîne telle que celle illustrée en 1.28(b) de coût inférieur, mais pour cela il faudrait que l'algorithme puisse éviter (c'est à dire considérer comme obstacles) les arêtes congestionnées.
Mais dans ce cas, il devient difficile voire impossible de relier un connecteur contenu dans une zone fortement congestionnée.



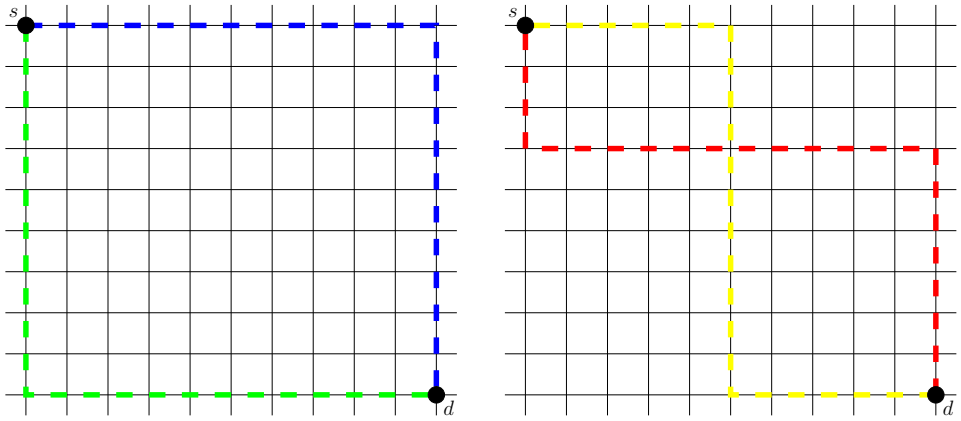
(a) Chaînes trouvées (b) Chaîne de coût inférieur

FIGURE 1.28 – Mauvaise gestion de la congestion pour le *line probing*

Les méthodes de *line probing* ont une complexité en $O(L)$, où L représente le nombre de lignes projetées. Elles ne garantissent pas de trouver la chaîne de coût minimale reliant s à d du fait du nombre limité de topologies d'arbres prises en compte et la gestion de la congestion étant limitée. Elles sont aujourd'hui peu utilisées dans les algorithmes de routage global.

Routage suivant des motifs (*Pattern routing*)
Partant de la constatation que dans les circuits actuels, la majorité des bipoints sont routés avec une chaîne à zéro, un ou deux coudes, les algorithmes de *pattern routing* limitent fortement le nombre de topologies possibles pour un bipoint.

En plus des chaînes « directes » (sans coudes), seuls les motifs de type L (un coude) et Z (deux coudes) sont pris en compte (voire uniquement le type L pour certains algorithmes).



(a) Motif en L (b) Motif en Z

FIGURE 1.29 – Motifs de *pattern routing*

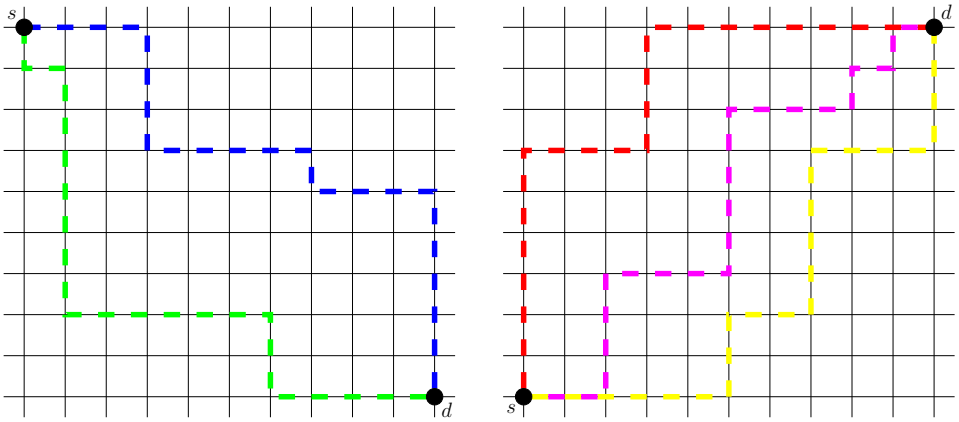
Le nombre de motifs considérés est simple à calculer. Pour des sommets s et d, nous notons l le nombre de lignes couvertes par la boîte englobante de s et d, et c le nombre de colonnes. Dans l'exemple de la figure 1.29 : $l = 10$ et $c = 11$.
Si s et d ne sont pas sur une même ligne ou une même colonne, le nombre de motifs en L possibles est constant égal à 2, tandis que le nombre de motifs en Z est égal à $(l - 2) + (c - 2)$ (17 dans notre exemple).

Cette forte limitation du nombre de topologies possibles accélère significativement les algorithmes de construction d'arbre d'interconnexion mais en contre partie, les solutions trouvées n'ont pas toujours une congestion satisfaisante (comme pour les méthodes de *line probing*, voir figure 1.28).

Routage par exploration monotone (*Monotonic routing*)
Les méthodes d'exploration monotone [PC07] permettent de considérer plus de topologies (et donc une meilleure gestion de la congestion) tout en gardant une grande rapidité d'exécution.

Ces méthodes recherchent parmi l'ensemble des chaînes monotones reliant la source s à la destination d, celle de coût minimal. Une **chaîne monotone** est une chaîne

reliant s à d et toujours dirigée de s vers d. Cette fois le nombre de topologies est égal à $\frac{(l+c-2)!}{(l-1)!(c-1)!}$ (92378 dans l'exemple de la figure 1.30).



(a) (b)

FIGURE 1.30 – Exemples de chaînes monotones

Il est important de noter que par construction, une chaîne monotone ne peut pas sortir de la boîte englobante de s et d, ce qui limite les possibilités pour éviter les éventuelles zones congestionnées à l'intérieur de cette boîte.

Routage par exploration exhaustive (*Maze routing*)

Les méthodes de *maze routing* considèrent toutes les chaînes possibles reliant s à d sur l'ensemble du graphe de routage $G(S, A)$ ou plus exactement sur l'ensemble de la fenêtre d'exploration.

La **fenêtre d'exploration** définit une portion du graphe de routage $G(S, A)$ contenant les sommets s et d et dans laquelle l'algorithme de *maze routing* cherche une chaîne de coût minimal reliant s à d. La taille de la fenêtre d'exploration peut être adaptée en fonction des zones congestionnées et varie de la boîte englobante de s et d à l'ensemble du graphe $G(S, A)$.

Se basant sur l'algorithme de Dijkstra [adD], les méthodes de *maze routing* garantissent de trouver la chaîne de coût minimal si elle existe.

La technique utilisée par l'algorithme de Dijkstra est assez simple. Chaque sommet $u \in S$ gère le coût de la chaîne minimale depuis la source s et une référence au sommet prédécesseur par lequel cette chaîne l'a atteint.

L'algorithme procède par une succession d'opérations de propagation du coût des sommets. Nous présenterons cette propagation en détails dans le chapitre 2.

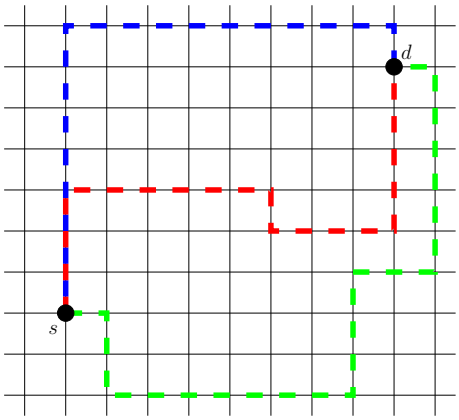


FIGURE 1.31 – Exemples de chaînes construites par l'algorithme de Dijkstra

Le nombre de topologies considérées n'est plus fonction de la taille de la boite englobante de s et d, mais de celle de la fenêtre d'exploration. Si à l'intérieur de cette fenêtre, la congestion ne permet pas de trouver une chaîne sans sur-congestion, il suffit d'agrandir la fenêtre d'exploration pour offrir plus de topologies possibles.

La figure 1.32 illustre l'agrandissement de la fenêtre d'exploration dans le but d'éviter de créer des zones sur-congestionnées. Les arêtes hachurées en rouge ont leur occupation égale à leur capacité, c'est à dire à la limite de la sur-congestion. Pour la figure 1.32(a), en considérant la fenêtre d'exploration (en bleu) il n'est pas possible de construire une chaîne reliant s à d sans créer de sur-congestion.
Si l'on considère maintenant une fenêtre d'exploration plus grande, plusieurs chaînes sans sur-congestion sont possibles, dont un exemple est donné sur la figure 1.32(b).

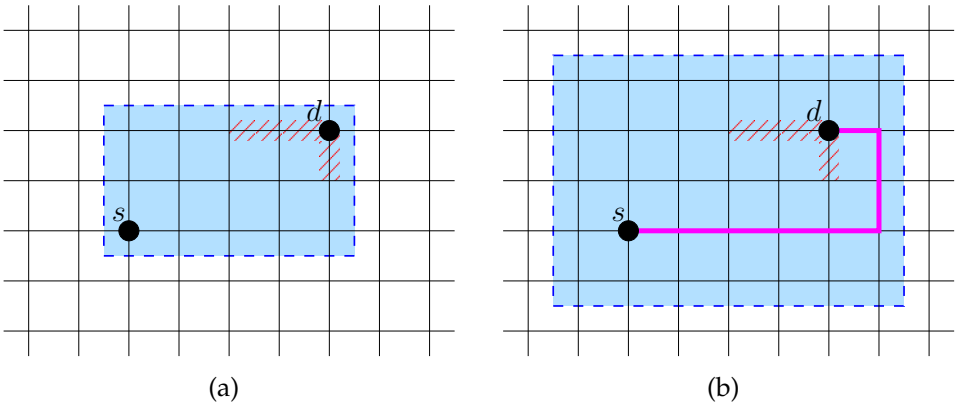


FIGURE 1.32 – Agrandissement de la fenêtre d'exploration

1.3 Approches de résolution

1.3.3 *Ripup & reroute*

Le *ripup & reroute* correspond à ce que nous avons précédemment décrit comme la *Phase 2* des méthodes de routage globale (voir page 30). Quelle que soit la méthode de routage global utilisée, la prise en compte d'un compromis entre les trois critères : longueur totale des fils d'interconnexion, nombre de vias et congestion, conduit souvent à une solution non valide.

Pour améliorer cette solution et tenter d'éliminer les éventuelles zones sur-congestionnées, il est nécessaire d'utiliser des méthodes gloutonnes de *ripup & reroute*.

Principe

Partant d'une solution non valide, les algorithmes de *ripup & reroute* procèdent par itérations successives. A chaque itération, les nets traversant des zones sur-congestionnées sont déroutés puis reroutés, en prenant en compte la congestion actualisée.

L'algorithme généralement utilisé pour le *ripup & reroute* est le suivant :

Algorithm 1.1 Algorithme général de *ripup & reroute*

Entrée : Une solution de routage global $\mathcal{S}$ non valide sur un graphe de routage $G(S, A)$

Sortie : Une solution de routage global $\mathcal{S}'$ sur $G(S, A)$.

$\mathcal{S}' := \mathcal{S}$
tant que $\mathcal{S}'$ n'est pas une solution valide **faire**
  Construire M l'ensemble des nets à dérouter.
  pour tout $n \in M$ **faire**
    $derouter(n)$.
  fin pour
  pour tout $n \in M$ **faire**
    $rerouter(n)$.
  fin pour
fin tant que

Les différentes mises en œuvre existantes [RM07] [CP06] se distinguent sur différents critères :

- le choix des nets à dérouter à chaque itération (l'ensemble M)
- l'ordre dans lequel les nets sont reroutés
- le coût associé à la traversée d'une zone congestionnée

Pour illustrer ces différences, nous présentons différents scénarios de mise en œuvre du *ripup & reroute*.

Scénarios de mise en œuvre

Scénario n˚1

Un premier scénario consiste, à chaque itération, à dérouter intégralement chaque net traversant au moins une arête sur-congestionnée. Une fois ces nets déroutés, il n'y a plus de sur-congestion, ils sont alors reroutés par un *maze router* prenant en compte la congestion actualisée.

L'ordre dans lequel les nets sont reroutés a une certaine importance et peut être déterminé suivant leur degré de liberté tel que décrit précédemment.

L'objectif est qu'au fur et à mesure des itérations, le nombre de nets à dérouter décroisse. Toutefois il n'existe aucune garantie de convergence d'une telle méthode de *ripup & reroute*.
En effet, rien ne garantit que le désengorgement de certaines zones ne conduise à l'engorgement d'autres zones et ceci de façon cyclique.
Une limite au temps d'exécution est donc usuellement fixée (limite temporelle ou nombre d'itérations).

Ce premier scénario a l'avantage d'être très simple, il suffit de savoir identifier les nets traversant au moins une arête sur-congestionnée. En revanche il peut générer un très grand nombre de nets à dérouter à chaque itération, ralentissant ainsi la vitesse de convergence vers une solution valide.

Scénario n˚2

Le deuxième scénario classe les arêtes en sur-congestion par ordre décroissant de dépassement. A chaque itération chacune des arêtes est traitée. Ce traitement consiste à dérouter l'ensemble des nets traversant l'arête puis à les rerouter.

L'avantage de ce scénario réside dans le fait que parmi les nets déroutés de la première arête traitée, certains traversent potentiellement une autre arête sur-congestionnée et les rerouter permet de désengorger aussi cette autre arête.

Au final le nombre de nets à dérouter à chaque itération est réduit.

Scénario n˚3

Le troisième scénario essaie d'optimiser le traitement en limitant le nombre de nets à dérouter. A chaque itération, l'ensemble des nets à dérouter est construit en fonction de leur contribution à la congestion.

1.3 Approches de résolution

La contribution d'un net à la congestion peut par exemple être exprimée en fonction du nombre d'arêtes en sur-congestion que son arbre d'interconnexion traverse. Une fois l'ordre des nets déterminé, chaque net est successivement dérouté puis rerouté. Entre chaque traitement d'un net, la congestion est actualisée.

De cette façon, pour désengorger une zone sur-congestionnée, seuls les nets contribuant le plus à la congestion seront reroutés et non pas tous les nets traversant la zone. De plus le choix des nets à dérouter peut prendre en compte leur degré de liberté. En effet si un net à un très faible degré de liberté, le dérouter ne présente aucun intérêt puisque l'arbre d'interconnexion issu du reroutage a une très forte probabilité d'être le même. Il est donc plus intéressant de dérouter des nets à fort degré de liberté.

Une légère alternative à ce scénario consiste à dérouter non pas le net ayant la plus forte contribution à la congestion, mais un ensemble de net dont la contribution dépasse un certain seuil (par exemple tous les nets dont la contribution est supérieure ou égale à 90% de la contribution maximale).
L'intérêt de cette alternative est de réduire le nombre d'appels à la méthode d'actualisation de la congestion qui, si la mise en œuvre n'est pas optimisée, peut être assez coûteuse en temps d'exécution.

Scénario n°4

Ce dernier scénario repose sur la notion de composante connexe définie précédemment. Grâce à celle-ci, il est possible de segmenter un net en plusieurs morceaux. Ainsi pour le *ripup & reroute* il devient possible de ne dérouter que des portions de nets.

Il suffit d'identifier les portions de nets contribuant le plus à la congestion pour les dérouter / rerouter à chaque itération, de la même façon qu'on le fait pour les nets dans le scénario 3.
Grâce à cela, les portions de net traversant des zones non-congestionnées sont conservées, limitant ainsi les fluctuations de la congestion dans celles-ci et accélérant le traitement du *maze router*.

Cette approche consistant à ne dérouter qu'une portion de net n'a encore jamais été étudiée : nous proposons dans ce manuscrit une mise en œuvre possible afin d'en évaluer les performances.

Comme nous l'avons dit précédemment, pour les deux derniers scénarios, l'actualisation de la congestion est faite après avoir dérouté / rerouté un net ou une portion de net. Ceci implique que la méthode d'actualisation est appelée de nombreuses fois au sein d'une même itération de *ripup & reroute*.
De ce fait, il est nécessaire d'avoir une méthode d'actualisation de la congestion rapide. Or cette rapidité repose sur une mise en œuvre efficace des structures gérant la congestion.

Conclusion

Dans ce chapitre nous avons présenté la façon de modéliser le problème de routage global en vue d'une résolution automatique à l'aide d'un algorithme dédié.

Le graphe de routage est l'élément central de cette modélisation puisqu'il permet de représenter les ressources disponibles et tout ou partie d'un net du circuit (grâce aux composantes connexes). Ce graphe peut avoir une structure bi ou tridimensionnelle mais la structure bidimensionnelle semble la plus appropriée puisqu'elle est plus légère en mémoire, plus simple à manipuler et couplée à un algorithme d'affectation aux couches, elle permet de traiter le problème en trois dimensions.

Nous avons présenté la fonction de coût qui permet de juger de la qualité d'une solution de routage global en tenant compte des critères de congestion, de longueur totale des fils d'interconnexion et du nombre de vias. Cette fonction permet de générer une solution initiale offrant un bon compromis entre les trois critères, qui sera ensuite améliorée lors d'une phase de *ripup & reroute* pour éliminer toute la sur-congestion.

Pour construire la solution initiale, il existe deux familles de méthodes. Les approches concurrentes considèrent tous les nets simultanément mais ne permettent pas de traiter de gros circuits. A l'inverse, les approches séquentielles traitent les net individuellement et séquentiellement. Parmi ces dernières, nous avons vu que les méthodes de type *Maze routing* sont, à priori, les plus à mêmes de créer une solution de coût optimal, ce qui nous pousse à les étudier plus en détails dans le chapitre suivant.

Enfin, nous avons présenté plusieurs scénarios de *ripup & reroute* pour améliorer la solution construite, dont un qui repose sur la notion originale de composante connexe.

Chapitre 2 Méthodes de résolution

Dans ce chapitre nous nous focalisons sur les méthodes séquentielles de type *Maze routing* qui, comme nous l'avons vu dans le chapitre précédent, sont les plus adaptées à la résolution du problème de routage global. Nous commençons par définir la fonction de coût prenant en compte les trois critères précédemment cités et utilisée pour construire un arbre d'interconnexion optimal à l'aide de ces méthodes.

Nous détaillons ensuite la technique d'estimation anticipée de la congestion qui permet d'éviter de prendre de mauvaises décisions pour les premiers nets traités en calculant des probabilités de congestion servant à guider l'algorithme de construction d'arbres d'interconnexion. Nous étudions différentes méthodes pour calculer et actualiser cette estimation.

Dans la dernière section, nous présentons l'algorithme de Dijkstra mis en œuvre dans notre outil de routage global. Partant de l'algorithme de Dijkstra permettant de construire une chaîne de coût minimal reliant deux sommets, nous détaillons les différentes extensions permettant d'interconnecter deux composantes connexes (ponctuelles ou non) ainsi qu'une approche multi composantes considérant simultanément toutes les composantes d'un net.

De plus, nous présentons l'agorithme A* qui est une variante de l'algorithme de Dijkstra utilisant une évaluation du coût restant vers la composante cible pour réduire le domaine de recherche.

2.1 Fonction de coût

Pour définir la fonction de coût du routage global permettant d'identifier la « meilleure » solution de routage global parmi plusieurs, nous définissons, tout d'abord, le coût d'une solution $\mathcal{S} = \{\mathcal{A}_i, i \in \{1, \ldots, n\}\}$ comme étant égal à la somme des coûts des arbres d'interconnexion la constituant. Le coût d'un arbre étant lui-même

égal à la somme des coûts des arêtes qu'il utilise :

$$cout(\mathcal{S}) = \sum_{i \in \{1,\dots,n\}} cout(\mathcal{A}_i)$$
$$cout(\mathcal{A}_i) = \sum_{a \in \mathcal{A}_i} cout(a)$$

Le coût d'une arête prend en compte les trois critères définis dans le chapitre précédent : longueur totale des fils d'interconnexion, nombre de vias et congestion. Le coût d'une arête se décompose en trois termes :

$$cout(a) = coutLongueur(a) + coutVias(a) + coutCongestion(a) \tag{2.1}$$

2.1.1 Longueur totale des fils d'interconnexion

Sur le graphe bidimensionnel $G(S, A)$, chaque arête a possède une longueur $long(a)$, qui est la distance de Manhattan centre à centre des pavés associés aux sommets qu'elle relie.

L'approche la plus simple consiste à définir que, pour une arête $a \in A$, le coût associé à la longueur est égal à la longueur de l'arête.

$$coutLongueur(a) = long(a)$$

Mais l'ordre de grandeur de la longueur des arêtes varie fortement suivant la phase de routage global que l'on considère. Lorsque le routeur global est utilisé pour guider le placement global, le pavage servant de base au graphe de routage est assez grossier, la longueur d'une arête est donc grande. A l'opposé, lorsque le routage global est utilisé pour initialiser le routeur détaillé, la grille de routage est beaucoup plus fine et par conséquent les arêtes plus courtes.

Or, d'après la formule 2.1, si le terme associé à la longueur est suffisamment grand, il peut « absorber » les autres termes, favorisant la construction d'un arbre d'interconnexion de longueur minimale, au détriment des autres critères.

La longueur $long(a)$ est donc normalisée en la divisant par le pas de grille du graphe régulier. Notons qu'il est important d'avoir un mécanisme similaire dans le cas d'un graphe de routage irrégulier. La longueur d'une arête peut alors être normalisée en la divisant par la plus petite des longueurs d'arêtes.

Le coût associé à la longueur d'une arête a est donc :

$$coutLongueur = longNormalisee(a)$$

Dans le cas d'un graphe de routage tridimensionnel, les arêtes transversales (représentant les vias) ont un coût $coutLongueur$ nul.

De plus, les arêtes des graphes $G'_{ik}(T'_{ik}, A'_{ik})$, utilisés pour construire les composantes connexes associées à des connecteurs répartis, ont un coût total $cout(a)$ nul. En effet si un arbre d'interconnexion utilise une arête $a \in A'_{ik}$, il n'utilise aucune ressource supplémentaire. Le coût total $cout(a)$ étant nul, chacun des trois coûts $coutLongueur(a)$, $coutVias(a)$ et $coutCongestion(a)$ est nul.
De cette manière les algorithmes de routage sont contraints à utiliser les arêtes des graphes $G'_{ik}(T'_{ik}, A'_{ik})$.

2.1.2 Nombre de vias

Comme nous l'avons vu dans le chapitre précédent, les vias sont représentés sur le graphe tridimensionnel par les arêtes transversales. L'utilisation d'une arête transversale par un arbre d'interconnexion équivaut physiquement à la création d'un via reliant les deux couches extrémités de l'arête.

Pour réduire le nombre de vias utilisés il suffit donc de donner un coût suffisamment important aux arêtes transversales (les autres arêtes ayant un coût de vias nul). Ce coût est en général supérieur au coût de traversée d'une arête non congestionnée. Par exemple, pour les circuits du jeu de test de l'ISPD2007 [oPDGRCa], le coût d'un via est fixé à une valeur de 3, c'est-à-dire trois fois le coût d'une arête non congestionnée fixé à 1.
Nous verrons par la suite que dans notre outil nous avons confirmé cette valeur de manière expérimentale.
Notons aussi que si ce coût est trop important, l'arbre d'interconnexion construit sera minimal en terme de nombre de vias, mais les risques de générer des zones sur-congestionnées augmentent.

Dans le cas d'un graphe de routage bidimensionnel, il n'y a aucune arête transversale. Il n'est donc à priori pas possible de considérer le coût des vias.
Pourtant nous sommes capables d'évaluer une borne inférieure du nombre de vias en comptant ceux dus aux changements de direction et ceux servant à relier les connecteurs (voir chapitre précédent page 23). Puisque la projection des arêtes transversales sur le graphe bidimensionnel correspond aux sommets du graphe, il est logique de penser à rajouter un coût correspondant aux vias sur les sommets, mais dans ce cas la fonction de coût d'un arbre d'interconnexion énoncée précédemment n'est plus valable puisqu'elle ne prend en compte que le coût des arêtes.

Le but étant de calculer le coût d'un arbre d'interconnexion, les arêtes ne sont en fait pas considérées indépendamment les unes des autres. Le calcul du coût d'un arbre d'interconnexion se fait récursivement en partant d'une arête puis en explorant

les arêtes voisines de l'arbre. Il est alors possible de comparer le type (horizontal ou vertical) d'une arête explorée (notée a_{ex}) et le type de l'arête qui a conduit à celle-ci (notée a_{src}). Si le type n'est pas le même, c'est qu'il y a un changement de direction (un coude) dans l'arbre, c'est-à-dire un via, pour interconnecter les différentes couches de métal. Nous pouvons alors associer à l'arête explorée a_{ex} le coût du via.
La formule 2.1 devient donc dépendante de l'arête p précédant a dans le calcul du coût de l'arbre :

$$cout(a,p) = coutLongueur(a) + coutVias(a,p) + coutCongestion(a) \tag{2.2}$$

De la même façon, les arêtes servant à relier un connecteur auront un $coutVias$ non nul, si le connecteur est sur une couche d'un type différent de celui de l'arête (paire ou impaire).

2.1.3 Congestion

Le coût lié à la congestion dépend du taux de congestion des arêtes. Dans la plupart des routeurs, ce coût est nul tant que le taux de congestion est strictement inférieur à 1, puis croît abruptement lorsque qu'il devient supérieur ou égal à 1.

$$coutCongestion(a) = \begin{cases} 0 & \text{si } tauxCongestion(a) < 1 \\ 10 & \text{sinon} \end{cases}$$

Le coût est limité à une valeur maximale (10 dans cet exemple) plutôt que de le rendre infini, de cette façon il reste possible d'utiliser une arête sur-congestionnée pour les algorithmes de routage. Ainsi il est toujours possible de construire une solution de routage global même si certaines arêtes sont sur-congestionnées.

La figure 2.1 illustre cette fonction de coût. Lorsque l'occupation de l'arête dépasse sa capacité, le coût devient grand pour empêcher les algorithmes de routage d'utiliser cette arête. Le coût de congestion d'une arête n'est donc pénalisant que lorsque son occupation dépasse sa capacité, ce qui ne permet pas d'éviter de générer des zones congestionnées.

Comme l'ont montré les études [Lin84] et [CM98], une fonction de coût avec un accroissement linéaire, telle que celle de la figure 2.2, est plus efficace.

La définition de cette fonction est :

$$coutCongestion(a) = \begin{cases} 0 & \text{si } tauxCongestion(a) < 0.8 \\ m & \text{si } tauxCongestion(a) > 1.4 \\ \frac{m}{3} \times (5 \times tauxCongestion(a) - 4) & \text{sinon} \end{cases}$$

Le facteur m représente la valeur maximale du coût de congestion (ici $m = 10$).

2.1 Fonction de coût

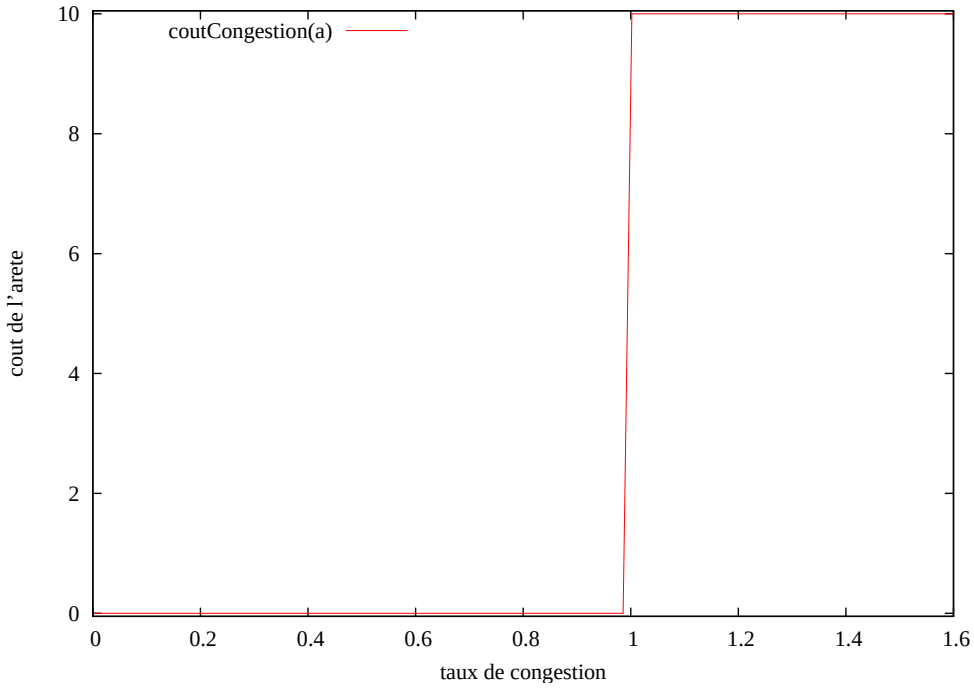


FIGURE 2.1 – Fonction de coût de la congestion

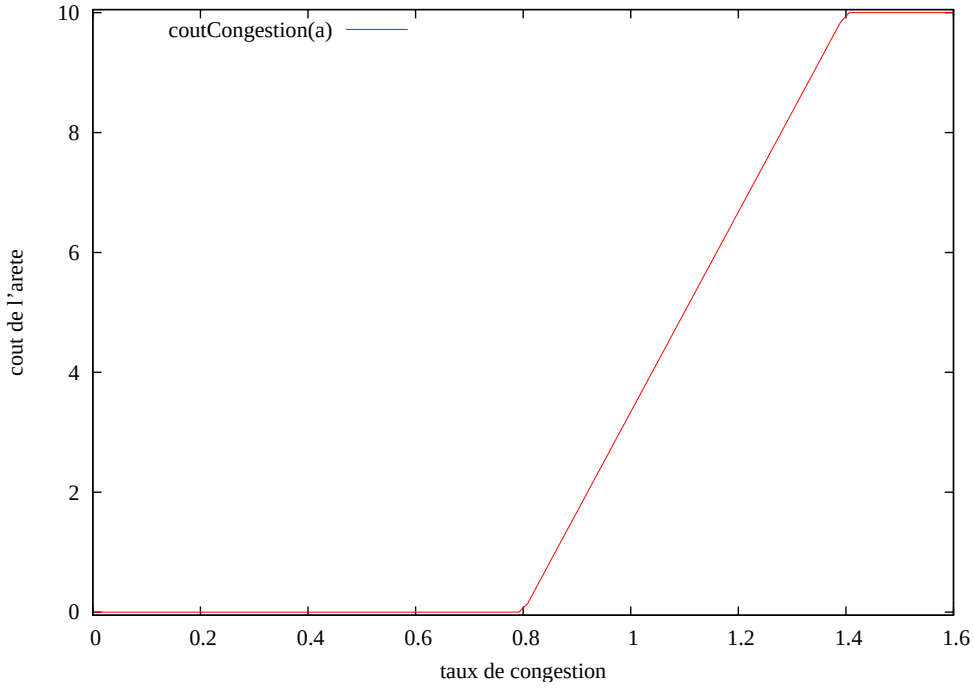


FIGURE 2.2 – Fonction de coût de la congestion

Dès que l'occupation d'une arête dépasse 80% de sa capacité le coût augmente linéairement jusqu'à atteindre sa valeur maximale pour une occupation dépassant de 40% la capacité de l'arête. Tant que le taux de congestion d'une arête est inférieur à 0.8, celle-ci est considérée comme non congestionnée et peut donc être utilisée par l'algorithme de routage.
Dès que son taux de congestion dépasse 0.8, les algorithmes ont tendance à l'éviter au profit d'autres arêtes moins congestionnées. Il reste possible d'utiliser l'arête, mais plus son occupation augmente et plus son coût augmente jusqu'à atteindre une valeur limite.

L'auteur de FastRoute [PC06] propose d'aller plus loin en « lissant » la fonction de coût de la congestion grâce à l'utilisation d'une fonction exponentielle, comme le montre la figure 2.3.

$$coutCongestion(a) = \frac{h}{1 + e^{-k \times (tauxCongestion(a) - 1)}}$$

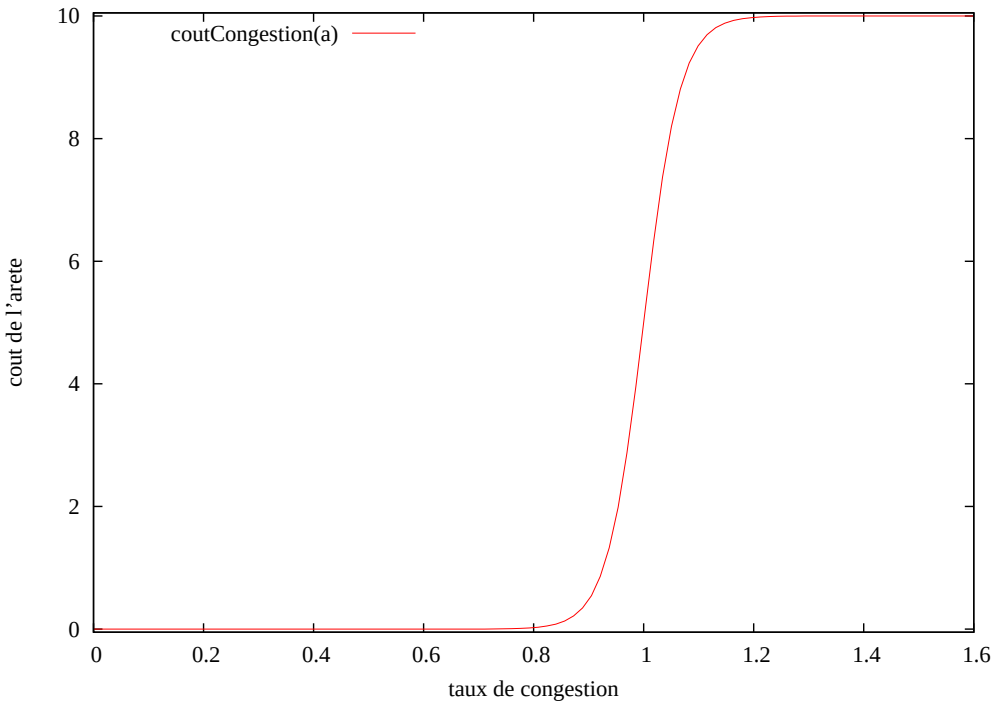


FIGURE 2.3 – Fonction de coût de la congestion

Les coefficients h et k sont des paramètres permettant de régler respectivement la valeur du maximum et la pente de la courbe, comme l'illustrent les figures 2.4 et 2.5.

2.1 Fonction de coût

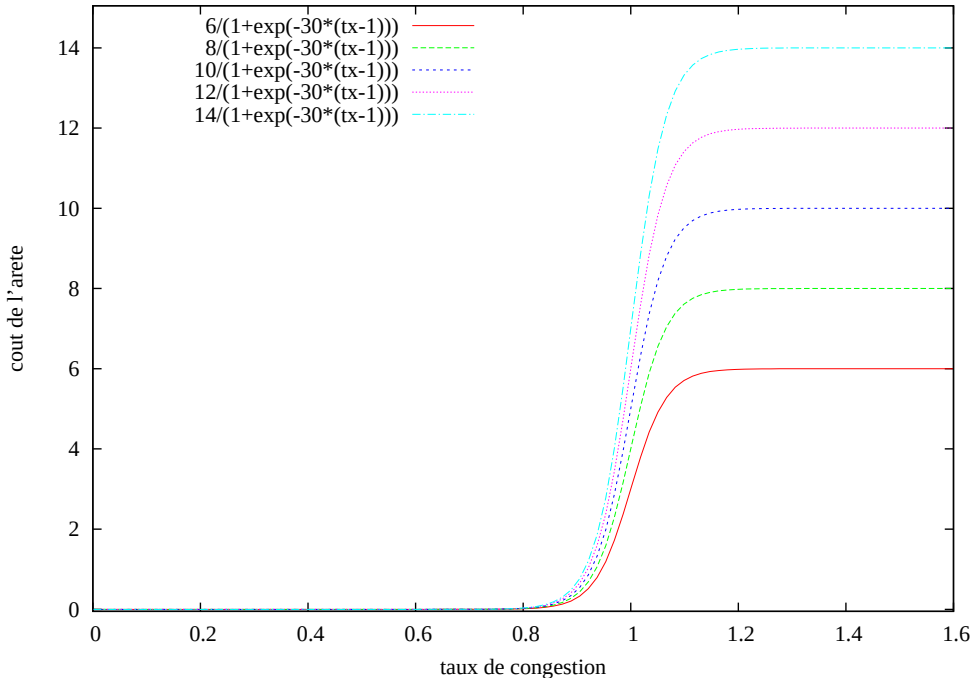


FIGURE 2.4 – Impact de la variation du paramètre h

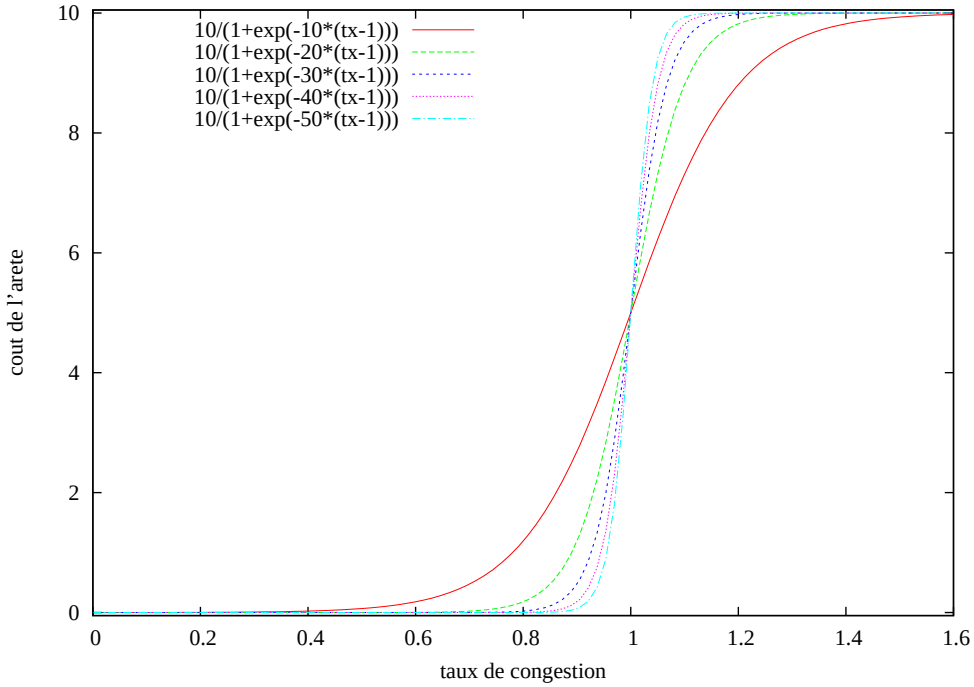


FIGURE 2.5 – Impact de la variation du paramètre k

2.2 Estimation anticipée de la congestion

2.2.1 Principe

Comme nous l'avons évoqué précédemment, les méthodes séquentielles font souvent appel à des algorithmes d'estimation anticipée de la congestion. Le but de ces méthodes est d'estimer la congestion future du circuit afin d'éviter que des décisions arbitraires et non remises en cause par la suite soient prises lors de la construction d'arbres d'interconnexion pour les premiers nets traités.

Une fois la congestion estimée calculée, les algorithmes de routage global prennent en compte à la fois cette estimation anticipée et la congestion due aux arbres d'interconnexion déjà créés sur le graphe de routage. La congestion est en fait décomposée en deux composantes :

- la **congestion estimée** : issue de l'estimation anticipée,
- la **congestion instantanée** : calculée à partir des arbres d'interconnexion existant sur le graphe de routage.

Chaque arête $a \in A$ du graphe $G(S, A)$ possède une congestion instantanée et une congestion estimée, le taux de congestion de l'arête a est alors calculé en fonction de ces deux composantes. Ce découpage en deux composantes est aussi décrit dans [HM03].

2.2.2 Construction

Pour estimer la congestion, les routeurs globaux utilisent des méthodes probabilistes. Chaque net à router $n_i \in N, i \in \{1, \ldots, n\}$ est décomposé en bipoints puis pour chaque bipoint, on considère un ensemble de chaînes possibles pour interconnecter les deux sommets du bipoint, à la manière du *Pattern routing*.

En général, toutes les chaînes d'un bipoint ont la même probabilité égale à $\frac{1}{k}$, où k représente le nombre de chaînes considérées pour interconnecter les sommets du bipoint. Mais il est tout à fait possible de considérer des probabilités différentes pour chaque chaîne, le raisonnement qui suit restant valide.

Pour une arête $a \in A$, la congestion estimée est égale à la somme des probabilités des chaînes considérées qui la traversent.

Dans l'exemple suivant, nous considérons deux nets, a et b composés des connecteurs à relier (a_1, a_2, a_3) et (b_1, b_2, b_3, b_4). Nous allons détailler le calcul de l'estimation de congestion pour une décomposition à partir d'arbres couvrants puis nous comparerons les probabilités obtenues avec celles que l'on peut obtenir lorsque l'on utilise une décomposition à partir d'arbres de Steiner.

2.2 Estimation anticipée de la congestion

La figure 2.6 présente un exemple de décomposition en bipoint à l'aide d'arbres couvrants. L'estimation probabiliste que nous utilisons ne considère que les chaînes de 0, 1 ou 2 coudes.

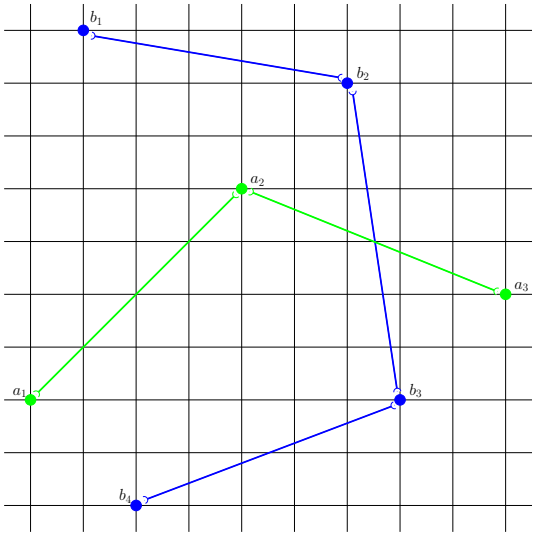
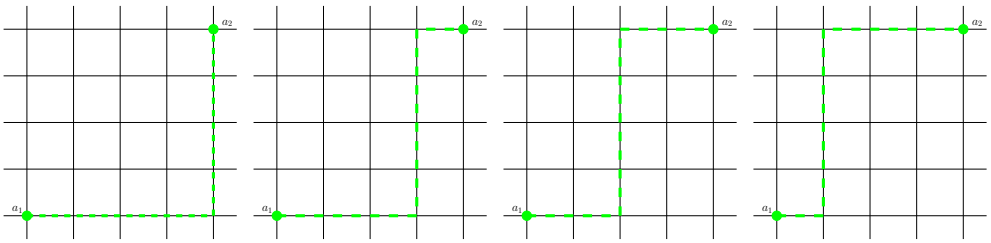
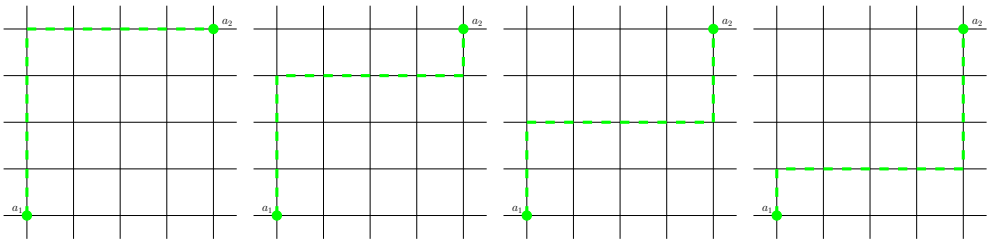


FIGURE 2.6 – Exemple de décomposition en bipoints à l'aide d'arbres couvrants

La figure 2.7 représente les huit chaînes possibles pour relier les connecteurs a_1 et a_2. Chacune de ces chaînes a une probabilité de $\frac{1}{8}$ répercutée sur les arêtes qu'elle traverse. Lorsque deux chaînes traversent la même arête, leurs probabilités s'ajoutent pour donner la congestion estimée de l'arête.



(a) Chaîne 1 (b) Chaîne 2 (c) Chaîne 3 (d) Chaîne 4



(e) Chaîne 5 (f) Chaîne 6 (g) Chaîne 7 (h) Chaîne 8

FIGURE 2.7 – Toutes les chaînes à 1 ou 2 coudes reliant a_1 et a_2

La figure 2.8 présente l'estimation de congestion des arêtes traversées par les chaînes présentées précédemment.

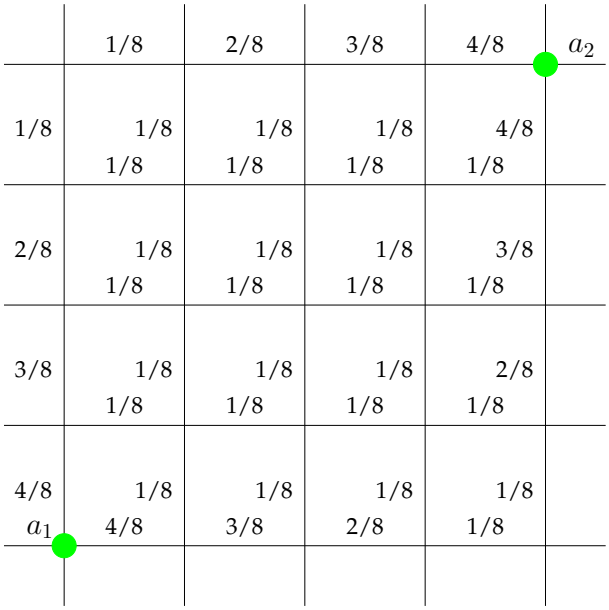


FIGURE 2.8 – Estimation probabiliste de congestion entre a_1 et a_2

Au final pour les deux nets a et b, nous obtenons l'estimation illustrée sur la figure 2.9.

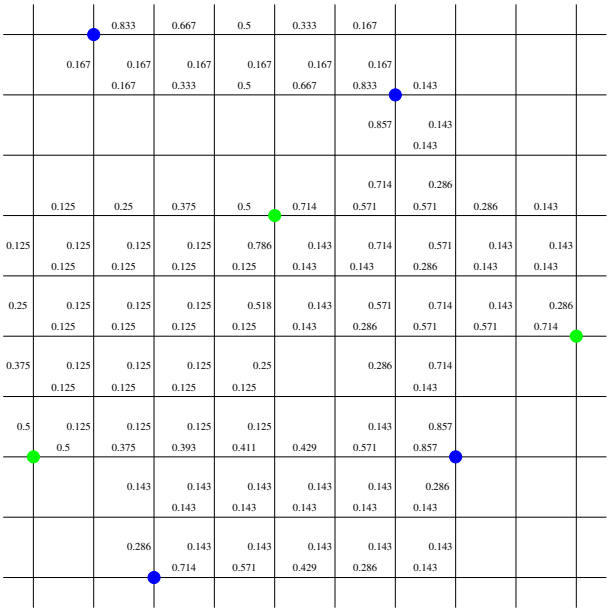
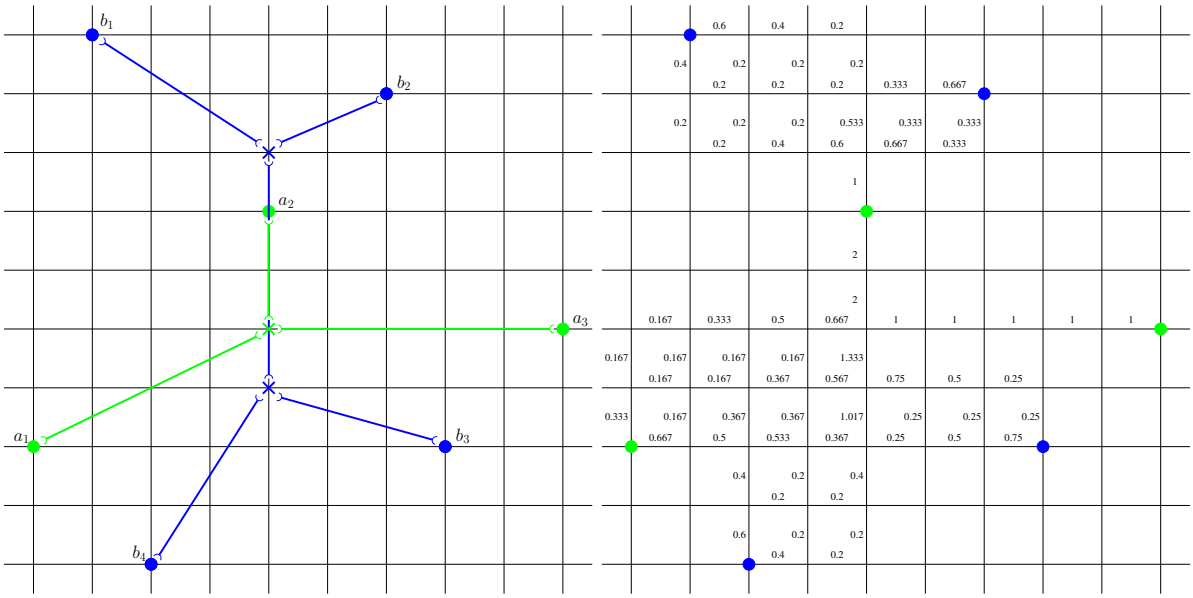


FIGURE 2.9 – Estimation probabiliste de la congestion à partir d'arbres couvrants

De la même façon nous pouvons calculer l'estimation de la congestion à partir d'une

décomposition en bipoints utilisant des arbres de Steiner, comme illustré sur les figures 2.10(a) et 2.10(b).



(a) Décomposition (b) Estimation

FIGURE 2.10 – Estimation probabiliste de la congestion à partir d'arbres de Steiner

Il est intéressant de constater que la répartition des arêtes dont l'estimation n'est pas nulle est très différente suivant que le calcul est fait à partir d'arbres couvrants ou d'arbres de Steiner.

L'utilisation d'arbres couvrants donne une répartition plus diffuse de la congestion tandis que l'utilisation des arbres de Steiner produit des zones plus précises de sur-congestion.

Pour calculer le taux de congestion d'une arête en fonction de ses congestions estimée et instantanée, le plus simple est de considérer la somme de ces deux congestions comme une seule congestion et de calculer son rapport à la capacité de l'arête :

$$tauxCongestion(a) = \frac{congestionInstantanee(a) + congestionEstimee(a)}{cap(a)}$$

2.2.3 Utilisation

Les méthodes séquentielles se décomposent en deux étapes consécutives : ordonnancement des nets puis construction d'un arbre d'interconnexion pour chaque net.

Le calcul de l'estimation de la congestion est fait avant la première étape. Lors de la deuxième étape, l'algorithme utilisé pour construire les arbres d'interconnexion prend en compte le taux de congestion tel que nous venons de le définir. De cette façon, lors du traitement des premiers nets, l'algorithme à connaissance des zones qui risquent d'être congestionnées et peut donc les éviter.

Au fur et à mesure que l'algorithme construit les arbres d'interconnexion, la congestion instantanée des arêtes augmente. Mais la congestion estimée doit-elle aussi varier ? Nous allons étudier deux gestions possibles de la congestion estimée :

- gestion statique : la congestion estimée ne varie jamais,
- gestion dynamique : la congestion est mise à jour à chaque fois qu'un arbre d'interconnexion est construit.

Nous verrons que, même si la deuxième méthode présente un désavantage en termes de temps de calcul, elle est beaucoup plus efficace pour optimiser le coût des arbres d'interconnexion construits tout en évitant de créer des zones sur-congestionnées.

Gestion statique de la congestion estimée

La gestion statique de la congestion estimée implique que l'estimation calculée initialement ne varie pas en cours de traitement. Pour chaque net, l'algorithme de construction d'arbres d'interconnexion tente d'éviter les zones estimées congestionnées et les zones réellement congestionnées (dues à la congestion instantanée).

Pour étudier l'impact de la gestion statique de la congestion, nous considérons un exemple composé de trois nets a, b et c formés des connecteurs (a_1, a_2, a_3), (b_1, b_2) et (c_1, c_2, c_3) (voir figure 2.11).

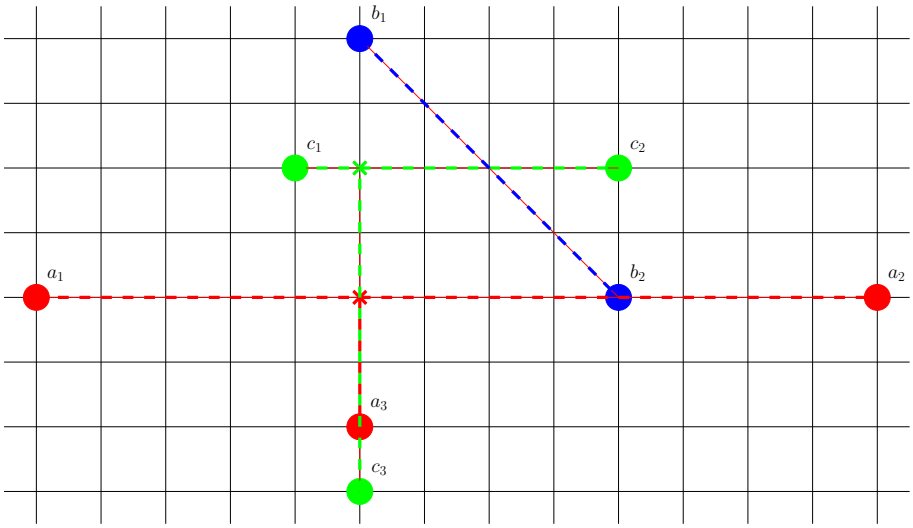


FIGURE 2.11 – Décomposition en bipoints des nets a, b et c

2.2 Estimation anticipée de la congestion

La figure 2.12 illustre le résultat de l'estimation de la congestion à partir d'arbres de Steiner. La capacité des arêtes du graphe de routage est égale à 1. La congestion estimée des arêtes sur-congestionnées est encadrée.

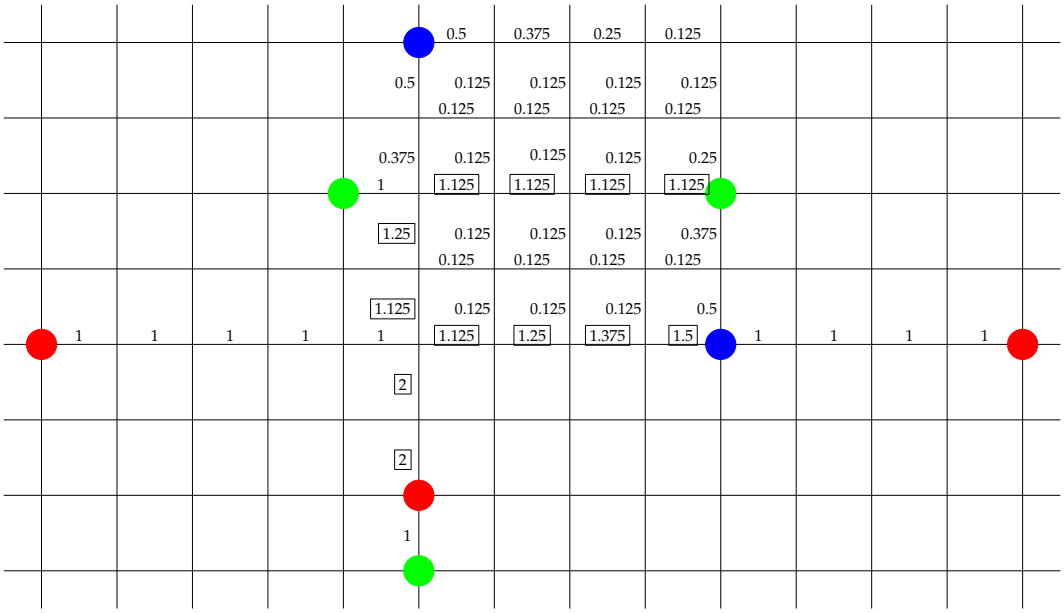


FIGURE 2.12 – Estimation anticipée de la congestion à partir d'arbres de Steiner

Une fois l'estimation de la congestion calculée, nous devons ordonner les nets. Pour cela nous allons calculer le facteur de forme de chaque net en considérant les longueurs normalisées.
Rappelons que pour un net $n \in N$, le facteur de forme de n est :

$FF(n) = (l + 1) \times (h + 1)$ où l est la largeur de la boîte englobante de n et h sa hauteur.

Nous obtenons : $FF(a) = 42$, $FF(b) = 25$ et $FF(c) = 36$. Ainsi l'algorithme va d'abord chercher à construire un arbre d'interconnexion pour le net b, puis le net c et enfin le net a.

Sur les figures 2.13, 2.14 et 2.15 nous présentons les trois étapes correspondant à la construction des arbres d'interconnexion des nets. La congestion estimée des arêtes est représentée en noir tandis que la congestion réelle est représentée en magenta.

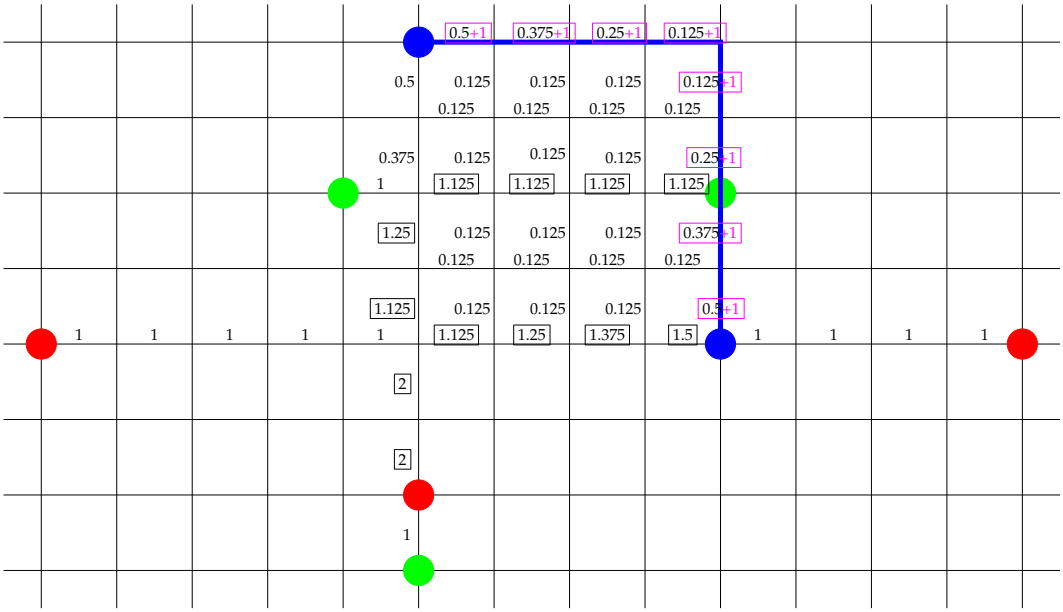


FIGURE 2.13 – Arbre d'interconnexion construit pour le net b

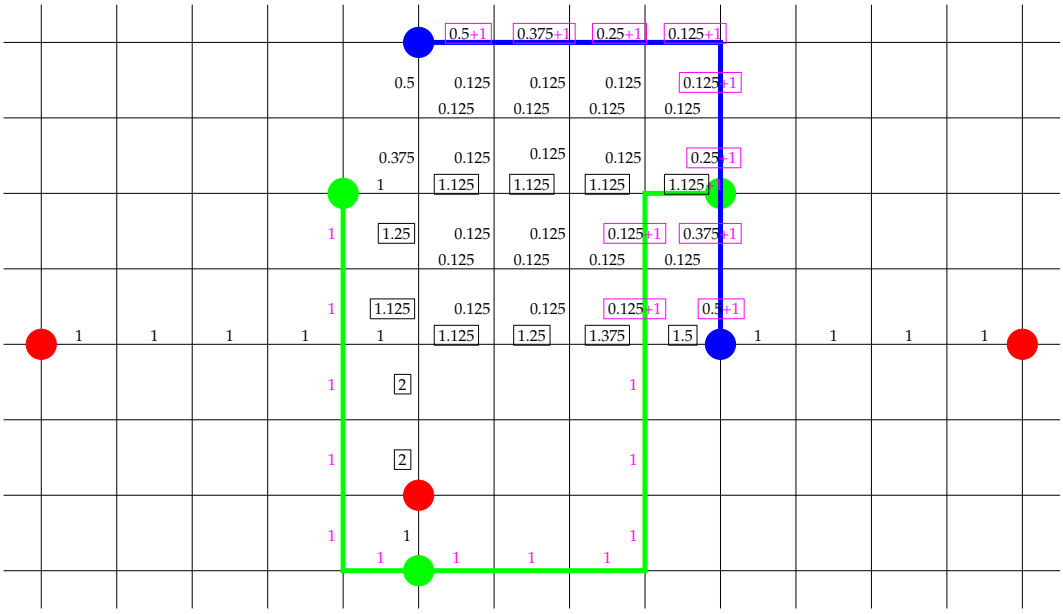


FIGURE 2.14 – Arbre d'interconnexion construit pour le net c

2.2 Estimation anticipée de la congestion

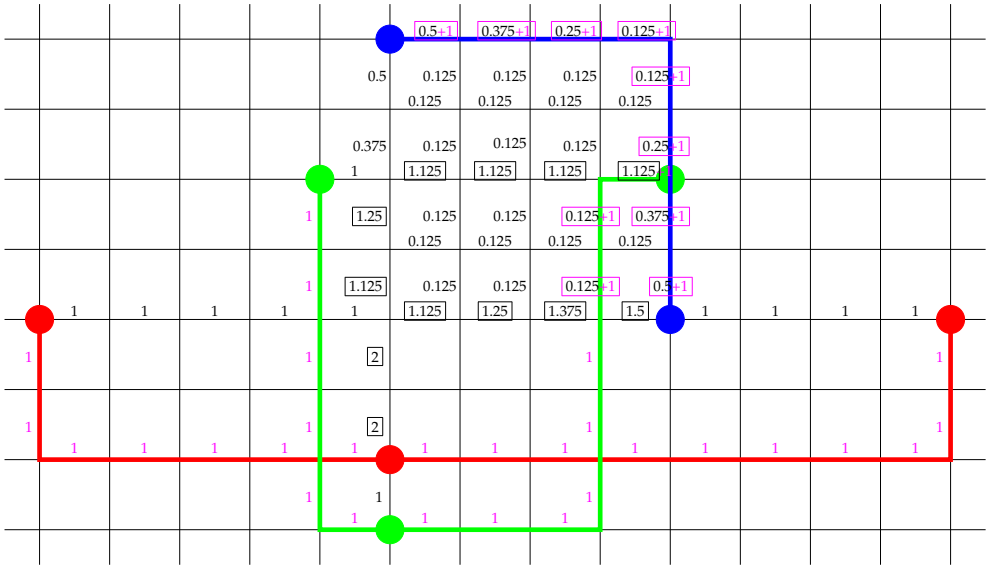


FIGURE 2.15 – Arbre d'interconnexion construit pour le net a

Nous constatons que les zones estimées sur-congestionnées (encadrées en noir) ont bien été évitées par l'algorithme de construction d'arbre au détriment de la longueur des arbres. Les arbres construits ne sont pas optimaux et il est assez facile de noter que la chaîne reliant c_2 à c_3 et contenant deux coudes peut être remplacée par une chaîne sans coude reliant c_1 à c_2 et ceci sans générer de sur-congestion (en ne considérant que la congestion instantanée). Ou encore de réduire le nombre de coudes utilisés dans l'arbre du net a à un seul.

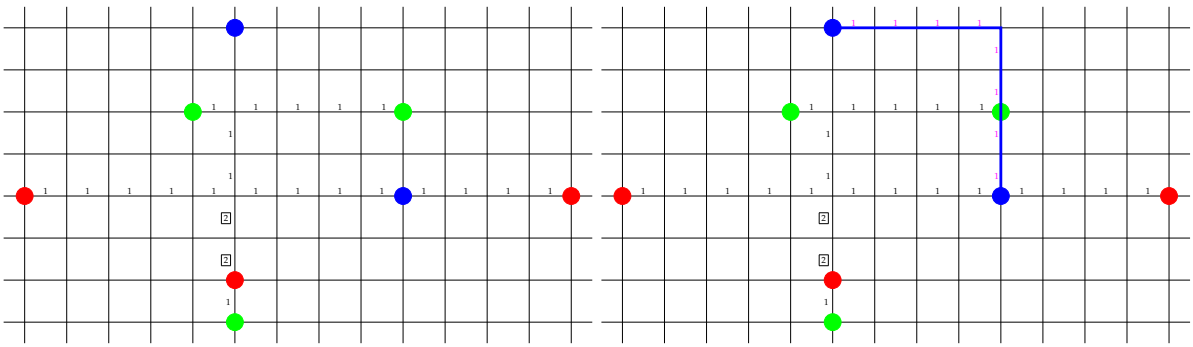
Ces mauvais résultats sont dus au fait qu'au cours du traitement, seule la congestion instantanée est mise à jour et pas la congestion estimée.

Gestion dynamique de la congestion estimée

Si l'on étudie de plus près le problème, on se rend compte que lorsque l'algorithme cherche à construire un arbre pour un net $n_i \in N, i \in \{1, \ldots, n\}$, il doit considérer la congestion instantanée due aux nets déjà routés sur le graphe, et la congestion estimée des nets non encore routés (hormis n).

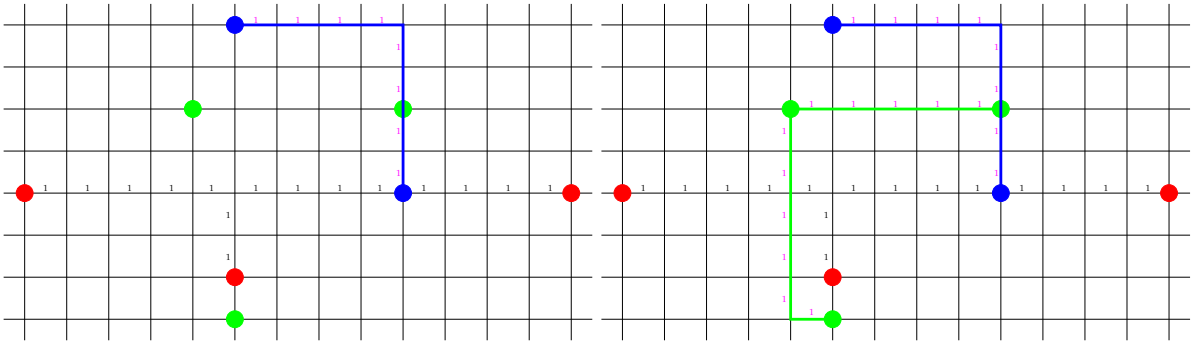
Ainsi l'algorithme commence par supprimer la participation à la congestion estimée du net, puis il construit un arbre d'interconnexion et enfin met à jour la congestion instantanée.

Si l'on reprend l'exemple précédent, seule la phase de construction des arbres d'interconnexion change. Les figures 2.16, 2.17 et 2.18 représentent le résultat de construction de chacun des arbres d'interconnexion (comme précédemment).



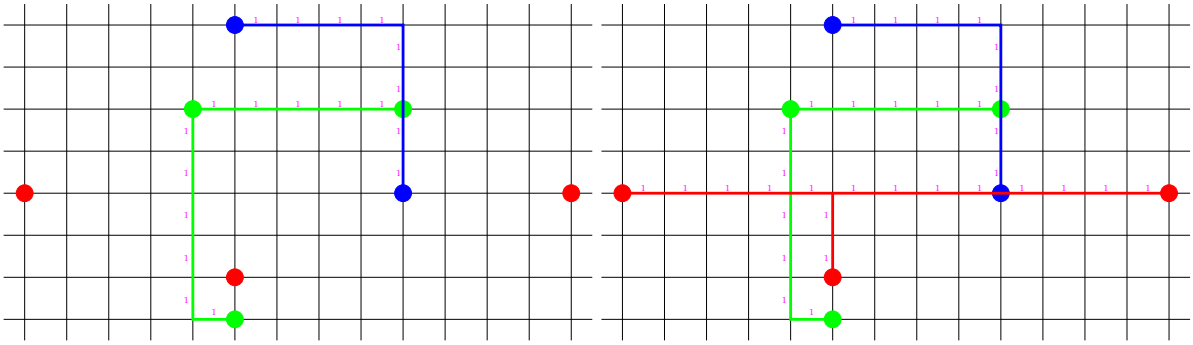
(a) Participation du net b supprimée (b) Arbre d'interconnexion

FIGURE 2.16 – Construction de l'arbre d'interconnexion pour le net b



(a) Participation du net c supprimée (b) Arbre d'interconnexion

FIGURE 2.17 – Construction de l'arbre d'interconnexion pour le net c



(a) Participation du net a supprimée (b) Arbre d'interconnexion

FIGURE 2.18 – Construction de l'arbre d'interconnexion pour le net a

Grâce à cette technique de gestion dynamique de la congestion estimée, les arbres d'interconnexion sont mieux optimisés, à la fois plus courts et avec moins de coudes (donc moins de vias).

Mais cette technique implique de pouvoir retrouver à tout moment les probabilités d'un net pour pouvoir supprimer sa participation, ce qui revient à être capable de reconstruire l'arbre couvrant, ou l'arbre de Steiner utilisé dans la phase de création de la congestion estimée.

Plutôt que de reconstruire l'arbre ayant servi à estimer la congestion d'un net, on peut imaginer une solution consistant à stocker en mémoire une correspondance entre un net et l'arbre utilisé. Le nombre très important de nets conduit rapidement à une occupation mémoire trop importante, rendant inutilisable ce genre d'approche.

Nous retenons une approche consistant à utiliser un algorithme de construction d'arbre déterministe qui, en fonction des sommets à relier du net, construit toujours le même arbre. De plus cet algorithme doit être rapide de façon à ne pas ralentir le routeur global.

Or comme nous l'avons décrit précédemment, les arbres construits pour calculer la congestion estimée sont des arbres rectilinéaires optimaux en terme de longueur seulement. Donc un algorithme de construction d'arbres de Steiner rectilinéaires quasi-optimaux comme FLUTE [fRCE] peut être utilisé. Nous verrons par la suite que nous procédons de cette façon dans notre routeur global.

2.3 Arbres d'interconnexion et algorithme de Dijkstra

Comme nous l'avons évoqué plus tôt, les méthodes de *Maze routing* permettent de construire la chaîne de coût minimal reliant deux sommets s et d du graphe de routage, en procédant par une succession d'opérations de progagation du coût des sommets.

Dans cette partie, nous commençons par détailler l'algorithme de Dijkstra utilisé pour construire la chaîne minimale reliant deux sommets s et d. Nous présentons ensuite la variante A*, puis différentes extensions de l'algorithme de Dijkstra pour prendre en compte les composantes connexes.

Enfin, nous présentons le traitement multi-composantes qui consiste à construire un arbre d'interconnexion reliant une ou plusieurs composantes connexes sources à plusieurs composantes connexes destination.

Dans tous les exemples qui suivent, les arêtes du graphe de routage sont associées à leur coût calculé tel que défini précédemment. Notons que ce coût est toujours positif ou nul.

2.3.1 Algorithme de Dijkstra uni-source uni-destination

L'**algorithme de Dijkstra** dans sa version originale permet de construire la chaîne de coût minimal entre un sommet s quelconque et tous les autres sommets d'un graphe connexe $G(S, A)$ dont les arêtes sont associées à un coût positif ou nul.

La technique utilisée est assez simple : chaque sommet $u \in S$ du graphe $G(S, A)$ gère le coût de la chaîne minimale depuis la source s, noté $c(u)$, ainsi qu'une référence au sommet précédent par lequel la chaîne a atteint le sommet u, notée $p(u)$.

L'algorithme procède par une succession d'opérations de propagation du coût de chaque sommet. Cette propagation consiste à comparer le coût actuel de chacun de ses voisins non traités avec la somme de son propre coût et du coût de l'arête les reliant. Si cette somme est inférieure au coût actuel du voisin, celui-ci est mis à jour et le sommet en cours de propagation devient son prédécesseur. L'ensemble des sommets déjà traités, c'est-à-dire dont le coût à été propagé, est noté P.

Voyons l'algorithme pour plus de détails :

Algorithme 2.2 Algorithme de Dijkstra uni-source uni-destination

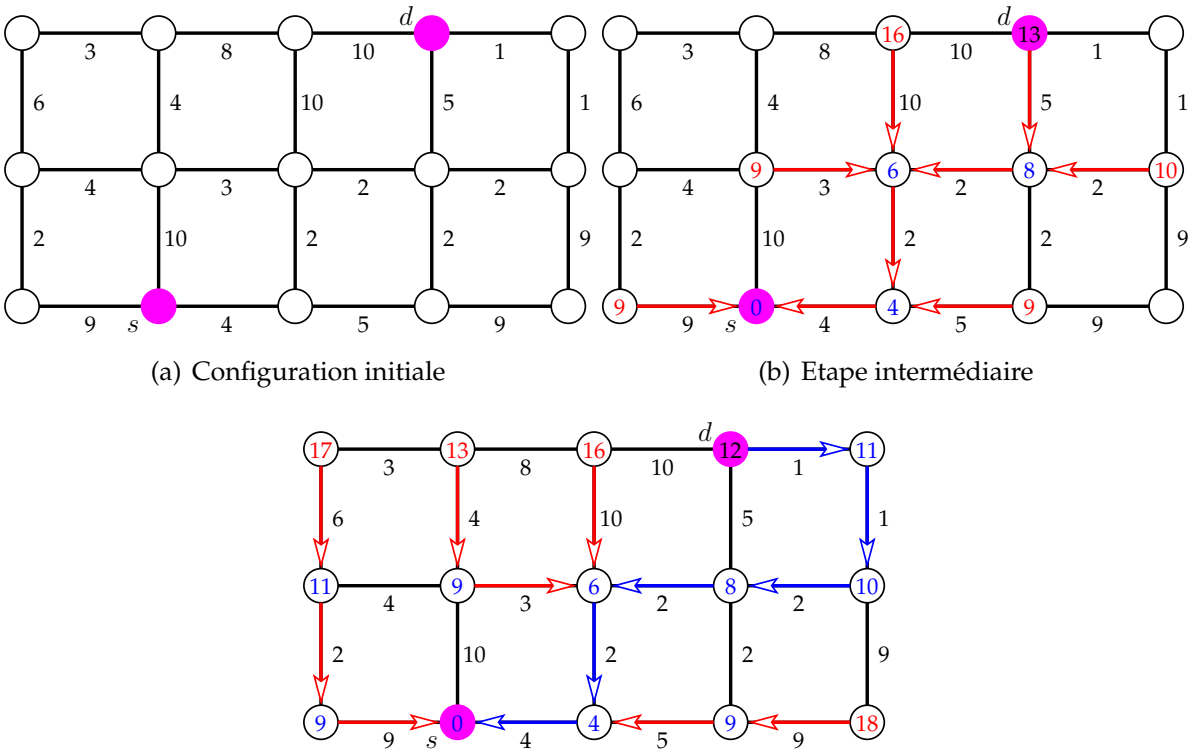
Entrée : un graphe de routage connexe $G(S, A)$, $s \in S$ le sommet source, $d \in S$ le sommet destination et $cout(\{u, v\}) : a = \{u, v\} \in A \rightarrow \mathbb{R}^+$ la fonction de coût des arêtes.

Sortie : une chaîne de coût minimal reliant s à d.

1: $c(s) := 0$
2: $c(v) := \infty$ pour chaque $v \in S \setminus \{s\}$
3: $P := \emptyset$
4: **tant que** $\exists v \in S \setminus P, c(v) < c(d)$ **faire**
5: Trouver un sommet $u \in S \setminus P$ tel que $c(u) = \min_{v \in S \setminus P} c(v)$
6: **pour tout** $v \in S \setminus P$ tel que $\{u, v\} \in A$ **faire**
7: **si** $c(v) > c(u) + cout(\{u, v\})$ **alors**
8: $c(v) := c(u) + cout(\{u, v\})$
9: $p(v) := u$
10: **fin si**
11: **fin pour**
12: $P := P \cup \{u\}$
13: **fin tant que**

2.3 Arbres d'interconnexion et algorithme de Dijkstra

La figure 2.19 présente un exemple de déroulement de l'algorithme de Dijkstra [1].



(a) Configuration initiale

(b) Etape intermédiaire

(c) Chaîne finale de coût minimal

FIGURE 2.19 – Déroulement de l'algorithme de Dijkstra

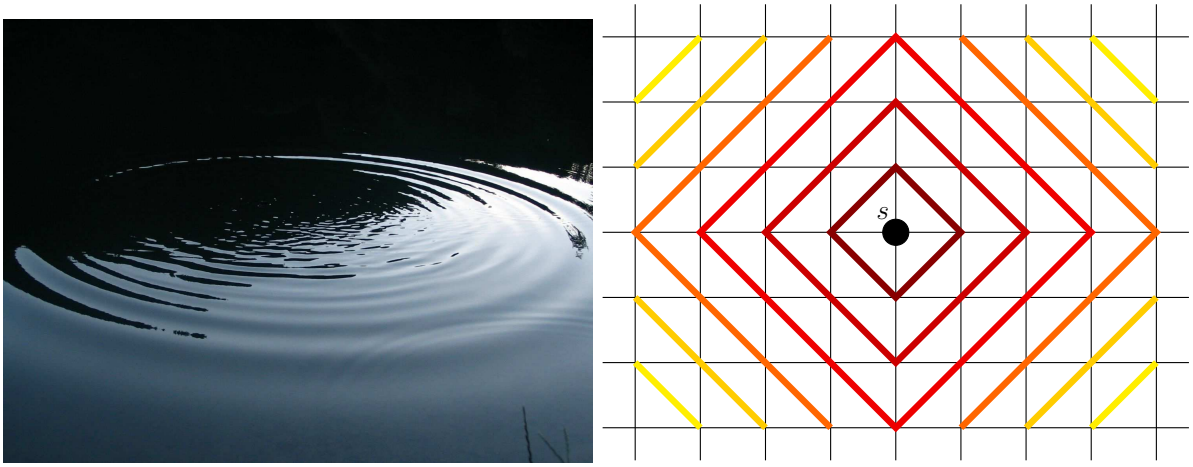
La figure 2.19(a) présente l'état initial du graphe de routage : les sommets source s et destination d sont représentés en magenta et la valeur annotée sur chaque arête correspond à son coût.

La figure 2.19(b) montre une étape du déroulement de l'algorithme. Le nombre au centre de chaque sommet $u \in S$ est le coût de la chaîne minimale actuellement trouvée pour relier s à u. Les sommets dont le coût est écrit en bleu sont les sommets déjà traités tandis que ceux ayant une coût écrit en rouge ont été explorés mais pas encore traités, c'est-à-dire que leur coût a été mis à jour mais pas encore propagé. Ces sommets représentent le bord de la vague. Le bord de la vague est l'ensemble des sommets $B \subset S$, tel que $\forall b \in B,\ b \in S \setminus P$ et $c(b) \neq \infty$.

$P \cup B$ forme donc l'ensemble dans sommets visités par l'algorithme et pour lesquels il existe une chaîne les reliant à s. Puisque le coût d'une arête est toujours positif ou nul, tous les sommets au-delà du bord de la vague auront un coût supérieur ou égal au coût minimum des sommets du bord de la vague.

1. L'ensemble des étapes de l'exécution de l'algorithme est donné en annexe page 151.

Le nom de bord de la vague est donné par analogie à la progression d'une onde à la surface de l'eau telle que l'illustre la figure 2.20.



(a) Propagation d'une onde à la surface de l'eau (b) Vagues de Dijkstra

FIGURE 2.20 – Analogie au bord de la vague

En effet si l'on considère un sommet source sur un graphe de routage dont toutes les arêtes ont un coût unitaire et que l'on relie entre eux les sommets de même coût après propagation, nous obtenons le résultat illustré par la figure 2.20(b).

Partant d'un sommet u exploré ou traité, il suffit de suivre les prédécesseurs représentés par les flèches pour rejoindre le sommet source.

Sur la figure 2.19(b), le sommet destination a été atteint par une chaîne dont le coût est de 13. Mais tant que dans le bord de la vague, il existe des sommets dont le coût est inférieur à cette valeur, il faut continuer les opérations de propagation car il est possible que le sommet destination soit atteint par une autre chaîne de coût inférieur.

Lorsque tous les sommets du bord de la vague ont un coût supérieur ou égal à celui du sommet destination, l'exécution est terminée. Comme le montre la figure 2.19(c) la chaîne minimale reliant s à d (représentée par les flèches bleues) a un coût inférieur à celle présentée sur la figure 2.19(b).

Lors de la recherche d'un sommet appartenant à $S \setminus P$ de coût minimum (ligne 5 de l'algorithme 2.2), si plusieurs sommets ont leur coût égal au minimum, le choix peut être fait aléatoirement ou en considérant l'ordre dans lequel ces sommets ont été atteints.

Notons enfin que dans cet exemple simple, la fenêtre d'exploration utilisée est

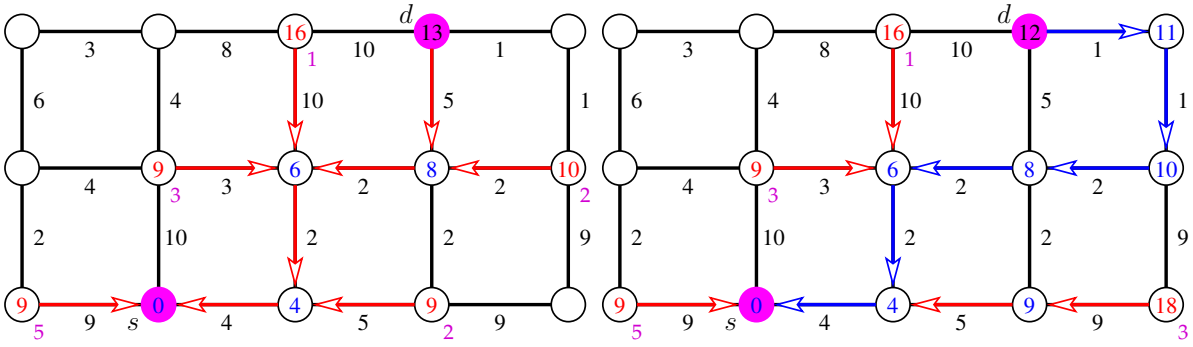
l'ensemble du graphe. Si nous avions utilisé une fenêtre d'exploration égale à la boîte englobante des sommets à relier, la chaîne présentée sur la figure 2.19(b) serait la chaîne de coût minimal.

2.3.2 Variante A*

L'**algorithme A*** [aA] est une variante de l'algorithme de Dijkstra qui utilise une évaluation du coût restant (*future cost*) vers la cible pour réduire le domaine de recherche. Le nombre de sommets traités est fortement réduit par rapport à l'algorithme de Dijkstra réduisant ainsi le temps d'exécution.

Le coût restant sert à guider la propagation du coût des sommets vers le sommet cible. Le coût d'un sommet $u \in S$ est la somme du coût par rapport au sommet source s et du coût restant.
Pour que l'algorithme A* construise toujours une solution de coût optimal, le coût restant d'un sommet $u \in S$ doit être une borne inférieure du coût de la chaîne optimale reliant u à d.

Reprenons l'exemple de la figure 2.19 et exécutons cette fois l'algorithme A*. L'évaluation du coût restant que nous utilisons est la distance Manhattan jusqu'à la cible exprimée en longueur normalisée. Cette évaluation vérifie la condition précédente.



(a) Etape intermédiaire (b) Chaîne finale de coût minimal

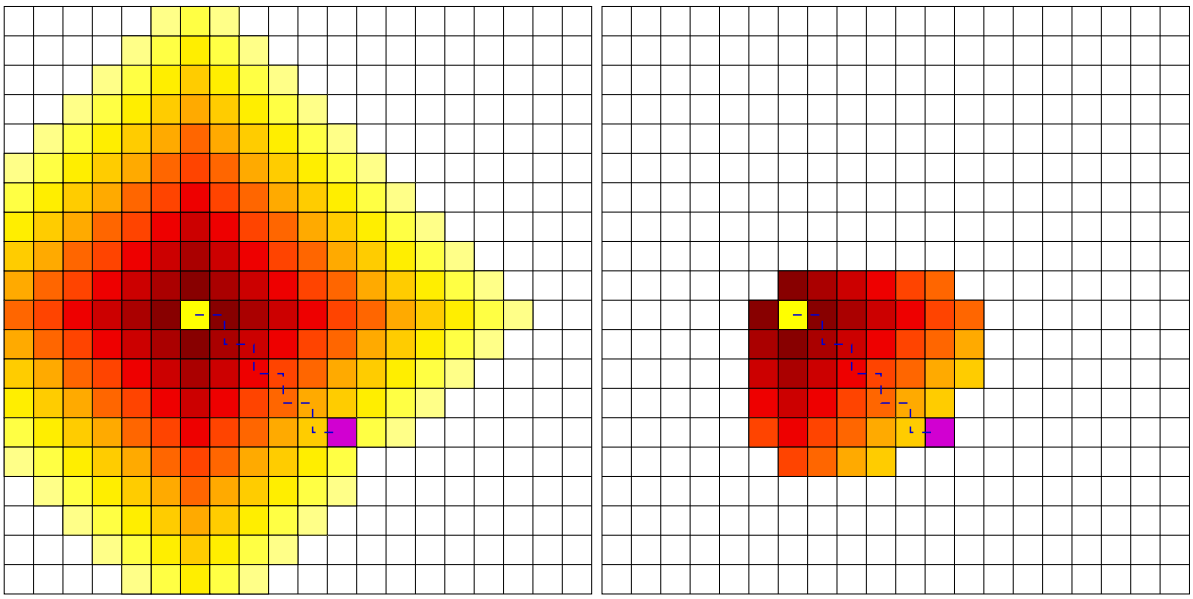
FIGURE 2.21 – Déroulement de l'algorithme A*

La figure 2.21(a) présente une étape du déroulement, le sommet destination a été atteint avec un coût de 13. Or certains sommets du bord de la vague ont un coût inférieur, il est donc encore possible d'atteindre le sommet destination avec un coût inférieur. Le coût des sommets est indiqué en leur centre tandis que le coût restant est indiqué sous le sommet et uniquement pour les sommets appartenant au bord de la vague.

La figure 2.21(b) présente le résultat de l'exécution de l'algorithme[1]. Comme précédemment, la chaîne de coût optimal construite est représentée en bleu.

Bien que l'exemple soit très simple, nous pouvons constater que le nombre de sommets traités de l'algorithme A* (six) est inférieur à celui de l'algorithme de Dijkstra (dix).

Dans le cas d'exemples ayant une zone d'exploration plus grande, l'écart est encore plus important.



(a) Sommets traités par l'algortihme de Dijkstra (b) Sommets traités par l'algorithme A*

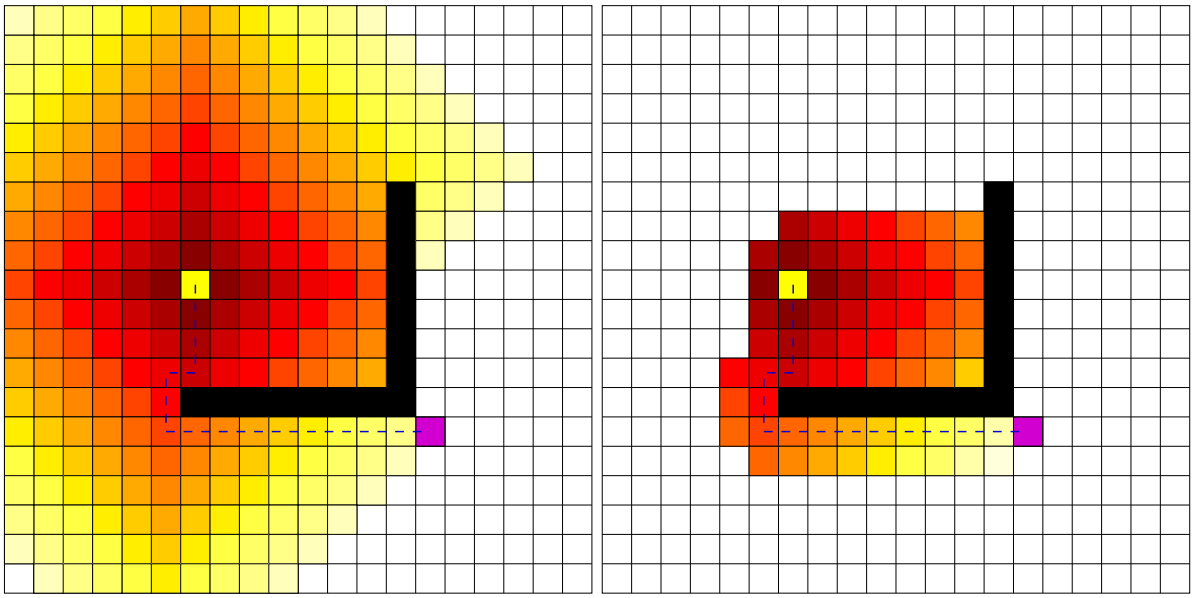
FIGURE 2.22 – Comparaison des algorithmes de Dijkstra et A*

La figure 2.22 compare les sommets traités par l'algorithme de Dijkstra et l'algorithme A* pour la construction d'une chaîne reliant un sommet source (représenté en jaune) et un sommet destination (représenté en violet). Le dégradé de couleur correspond aux vagues de Dijkstra comme définies pour la figure 2.20(b).

Pour cet exemple, le nombre de sommets traités est de 140 pour l'algorithme de Dijkstra et seulement 46 pour l'algorithme A*.

Dans le cas de la présence d'un obstacle, le nombre de sommets traités même s'il augmente, reste inférieur pour l'algorithme A*, comme l'illustre la figure 2.23. Sur cet exemple, le nombre de sommets traités est de 251 pour l'algorithme de Dijkstra contre 67 pour l'algorithme A*.

1. L'ensemble des étapes de l'exécution de l'algorithme est donné en annexe page 153.



(a) Sommets traités par l'algorithme de Dijkstra (b) Sommets traités par l'algorithme A*

FIGURE 2.23 – Comparaison des algorithmes de Dijkstra et A* avec un obstacle

Globalement l'algorithme A* est donc plus rapide du fait que le nombre de sommets traités est fortement réduit.

Il est important de noter que l'algorithme de Dijkstra peut être accéléré en diminuant la fenêtre d'exploration, mais, comme nous l'avons noté pour la figure 2.19, la chaîne construite peut ne pas être celle de coût optimal.
Dans le cas de la figure 2.22, si nous limitons la fenêtre d'exploration à la boîte englobante des sommets à relier, les sommets traités sont les mêmes.

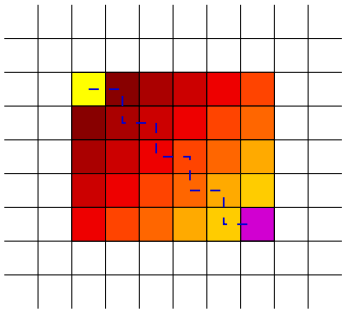


FIGURE 2.24 – Sommets traités pour une fenêtre d'exploration limitée à la boîte englobante des sommets

2.3.3 Extension de Dijkstra aux composantes connexes

Les deux algorithmes de *Maze routing* que nous venons de présenter permettent de trouver la chaîne de coût optimal reliant deux sommets du graphe de routage. Nous allons maintenant décrire comment l'algorithme de Dijkstra peut être utilisé lorsque nous considérons les composantes connexes.

Le principe de fonctionnement repose sur le fait que quelle que soit la composante connexe, dès que l'un de ses sommets est atteint, toute la composante connexe est interconnectée. Si les composantes connexes sont ponctuelles le problème reste le même que précédemment.

Partant de l'algorithme initial 2.2, nous considérons trois cas pour les composantes connexes source et destination :

1. la composante connexe source est ponctuelle et la composante connexe destination est non ponctuelle,
2. la composante connexe source est non ponctuelle et la composante connexe destination est ponctuelle,
3. les deux composantes connexes sont non ponctuelles.

Source ponctuelle, destination non ponctuelle

Dans ce premier cas, nous cherchons la chaîne de coût minimal reliant un sommet source $s \in S$ et une composante connexe destination $\mathcal{A}_{ik}$ du net $n_i \in N, i \in \{1, \ldots, n\}$ et $k \in \{1, \ldots, k_i\}$. L'ensemble des sommets électriquement connexes représentés par la composante connexe est noté T_{ik}.

L'algoritme est le suivant :

Algorithme 2.3 Algorithme de Dijkstra source ponctuelle - destination non ponctuelle

Entrée : un graphe de routage connexe $G(S, A)$, $s \in S$ le sommet source, $\mathcal{A}_{ik}$ la composante connexe destination et $cout(\{u, v\}) : a = \{u, v\} \in A \rightarrow \mathbb{R}^+$ la fonction de coût des arêtes.

Sortie : une chaîne de coût minimal reliant s à $\mathcal{A}_{ik}$.

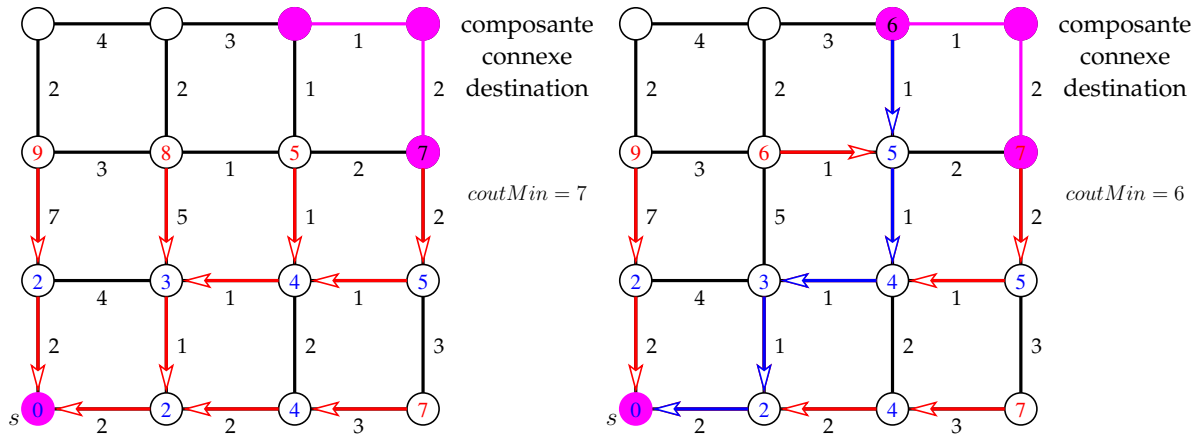
1: $c(s) := 0$
2: $c(v) := \infty$ pour chaque $v \in S \setminus \{s\}$
3: $P := \emptyset$
4: $couMin := \infty$
5: **tant que** $\exists v \in S \setminus P, c(v) < coutMin$ **faire**
6: Trouver un sommet $u \in S \setminus P$ tel que $c(u) = \min_{v \in S \setminus P} c(v)$
7: **pour tout** $v \in S \setminus P$ tel que $\{u, v\} \in A$ **faire**

8: **si** $c(v) > c(u) + cout(\{u, v\})$ **alors**
9: $c(v) := c(u) + cout(\{u, v\})$
10: $p(v) := u$
11: **si** $v \in T_{ik}$ et $c(v) < coutMin$ **alors**
12: $coutMin := c(v)$
13: **fin si**
14: **fin si**
15: **fin pour**
16: $P := P \cup \{u\}$
17: **fin tant que**

Cet algorithme est très proche du précédent (2.2). La principale différence réside dans le fait que le coût actuel avec le lequel la cible a été atteint n'est plus le coût d'un seul sommet ($c(d)$) mais le minimum du coût des sommets appartenant à T_{ik} c'est-à-dire $coutMin$.

De la même façon que précédemment, l'algorithme s'arrête lorsque tous les sommets du bord de la vague ont un coût supérieur ou égal à $coutMin$. Ainsi nous sommes sûr que la composante connexe est interconnectée par une chaîne de coût minimal.

La figure 2.25 illustre un cas pour lequel la première chaîne construite pour atteindre la composante connexe n'est pas celle de coût minimal (voir figure 2.25(a)). La chaîne de coût minimal étant celle représentée sur la figure 2.25(b) [1].



(a) Etape intermédiaire (b) Chaîne finale de coût minimal

FIGURE 2.25 – Déroulement de l'algorithme de Dijkstra pour une source ponctuelle et une destination non ponctuelle

1. L'ensemble des étapes de l'exécution de l'algorithme est donné en annexe page 154.

Source non ponctuelle, destination ponctuelle

Cette fois nous cherchons à construire la chaîne de coût minimal permettant de relier une composante connexe source $\mathcal{A}_{ik}$ du net $n_i \in N, i \in \{1, \ldots, n\}$ et $k \in \{1, \ldots, k_i\}$ et un sommet destination $d \in S$. L'ensemble des sommets électriquement connexes représentés par la composante connexe est noté T_{ik}.

L'algoritme est le suivant :

Algorithme 2.4 Algorithme de Dijkstra source non ponctuelle - destination ponctuelle

Entrée : un graphe de routage connexe $G(S, A)$, $\mathcal{A}_{ik}$ la composante connexe source, $d \in S$ le sommet destination et $cout(\{u, v\}) : a = \{u, v\} \in A \rightarrow \mathbb{R}^+$ la fonction de coût des arêtes.

Sortie : une chaîne de coût minimal reliant $\mathcal{A}_{ik}$ à d.

1: $c(s) := 0$ pour chaque $s \in T_{ik}$
2: $c(v) := \infty$ pour chaque $v \in S \setminus \{T_{ik}\}$
3: $P := \emptyset$
4: **tant que** $\exists v \in S \setminus P, c(v) < c(d)$ **faire**
5: Trouver un sommet $u \in S \setminus P$ tel que $c(u) = \min_{v \in S \setminus P} c(v)$
6: **pour tout** $v \in S \setminus P$ tel que $\{u, v\} \in A$ **faire**
7: **si** $c(v) > c(u) + cout(\{u, v\})$ **alors**
8: $c(v) := c(u) + cout(\{u, v\})$
9: $p(v) := u$
10: **fin si**
11: **fin pour**
12: $P := P \cup \{u\}$
13: **fin tant que**

Si ce n'est la phase d'initialisation du coût des sommets (lignes 1 et 2), rien ne change dans l'algorithme par rapport à l'algorithme 2.2.

La figure 2.26 illustre le résultat du déroulement de l'algorithme [1].

1. L'ensemble des étapes de l'exécution de l'algorithme est donné en annexe page 156

2.3 Arbres d'interconnexion et algorithme de Dijkstra

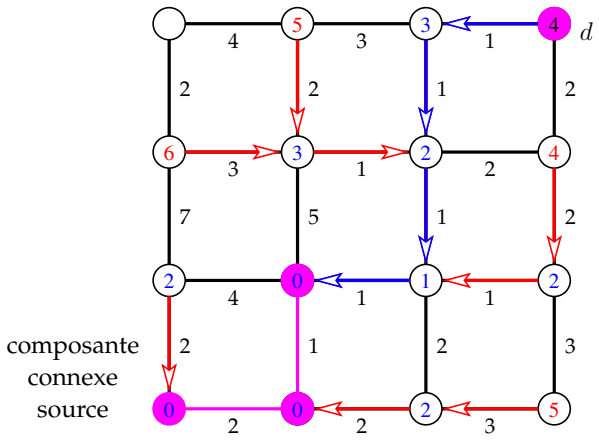


FIGURE 2.26 – Chaîne de coût minimal reliant $\mathcal{A}_{ik}$ et d

Source et destination non ponctuelles

Les modifications présentées dans les deux cas précédents peuvent être combinées pour construire une chaîne de coût minimal reliant deux composantes connexes $\mathcal{A}_{ik}$ et $\mathcal{A}_{ik'}$ du net $n_i \in N, i \in \{1, \ldots, n\}$ et $k, k' \in \{1, \ldots, k_i\}$.
L'ensemble des sommets électriquement connexes représentés par la composante connexe $\mathcal{A}_{ik}$ (resp. $\mathcal{A}_{ik'}$) est noté T_{ik} (resp. $T_{ik'}$).

L'algoritme est le suivant :

Algorithme 2.5 Algorithme de Dijkstra source et destination non ponctuelles

Entrée : un graphe de routage connexe $G(S, A)$, $\mathcal{A}_{ik}$ la composante connexe source, $\mathcal{A}_{ik'}$ la composante connexe destination et $cout(\{u, v\}) \ : a = \{u, v\} \in A \rightarrow \mathbb{R}^+$ la fonction de coût des arêtes.
Notons que par construction $\mathcal{A}_{ik} \cap \mathcal{A}_{ik'} = \emptyset$.

Sortie : une chaîne de coût minimal reliant $\mathcal{A}_{ik}$ à $\mathcal{A}_{ik'}$.

1: $c(s) := 0$ pour chaque $s \in T_{ik}$
2: $c(v) := \infty$ pour chaque $v \in S \setminus \{T_{ik}\}$
3: $P := \emptyset$
4: $coutMin := c(v)$
5: **tant que** $\exists v \in S \setminus P, c(v) < coutMin$ **faire**
6: Trouver un sommet $u \in S \setminus P$ tel que $c(u) = \min_{v \in S \setminus P} c(v)$
7: **pour tout** $v \in S \setminus P$ tel que $\{u, v\} \in A$ **faire**
8: **si** $c(v) > c(u) + cout(\{u, v\})$ **alors**
9: $c(v) := c(u) + cout(\{u, v\})$
10: $p(v) := u$
11: **si** $v \in T_{ik'}$ et $c(v) < coutMin$ **alors**
12: $coutMin := c(v)$

13: **fin si**
14: **fin si**
15: **fin pour**
16: $P := P \cup \{u\}$
17: **fin tant que**

La figure 2.27 illustre le résultat du déroulement de cet algorithme[1] pour construire une chaîne de coût minimal reliant les composantes connexes $\mathcal{A}_{ik}$ et $\mathcal{A}_{ik'}$.

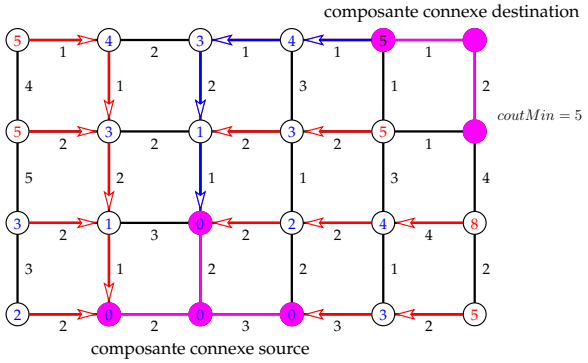


FIGURE 2.27 – Chaîne de coût minimal reliant $\mathcal{A}_{ik}$ et $\mathcal{A}_{ik'}$

2.3.4 Adaptation de l'algorithme A* aux composantes connexes non ponctuelles

Puisque l'algorithme A* est une variante de l'algorithme de Dijkstra il doit lui aussi pouvoir être adapté pour fonctionner avec les composantes connexes.

Le problème réside dans le fait de trouver une évaluation du coût restant pour chaque sommet du graphe et non pas un seul sommet comme précédemment. De plus la condition de non surestimation énoncée à la page 69 doit être respectée.

Une borne inférieure simple consiste à estimer le coût restant en calculant la distance de Manhattan jusqu'au plus proche sommet de la composante connexe cible. Pour connaître le coût restant d'un sommet $u \in S$, il est nécessaire de calculer les distances Manhattan de u jusqu'à chacun des sommets de la composante connexe cible. Le coût restant est alors le minimum de ces distances.

Lors de la propagation des coûts des sommets, pour chaque sommet traité il est donc nécessaire de calculer la distance minimale jusqu'à la composante connexe cible. Si la composante connexe cible est composée de nombreux sommets et que la distance

1. L'ensemble des étapes de l'exécution de l'algorithme est donné en annexe page 158.

séparant les deux composantes est grande (en terme de pas de grille), alors le nombre de calcul de distance entre deux sommets cumulé tout au long de la propagation des coûts peut être très grand.

Bien que ce cas soit assez rare et que la complexité du traitement soit linéaire, nous lui préférons une méthode consistant à calculer la distance de Manhattan par rapport à la boîte englobante de la composante connexe cible. De cette façon, quelle que soit la taille de la composante connexe le temps de calcul reste constant.

Dans le cas d'un sommet u contenu dans la boîte englobante de la composante connexe, il faut que le coût restant calculé soit égal à 0 pour éviter toute surestimation. Il existe aussi certaines configurations qui peuvent sous estimer assez fortement le coût restant, comme l'illustre la figure 2.28. La distance de Manhattan entre u et la boîte englobante est de 5 (en unités normalisées) alors que le sommet le plus proche de la composante connexe est distant 9 unités normalisées.

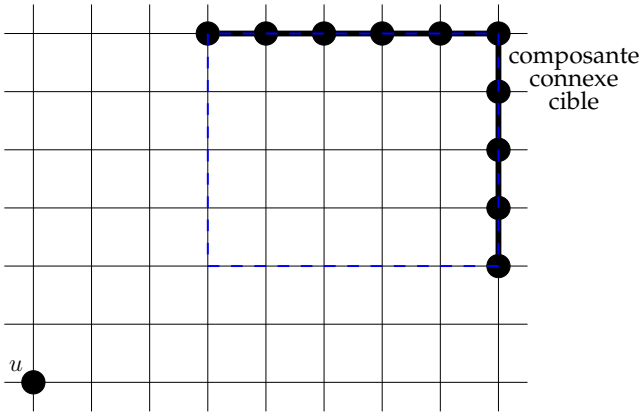


FIGURE 2.28 – Sous estimation du coût restant par rapport à la boîte englobante d'une composante connexe

L'auteur de FGR [RM07] propose une solution permettant d'utiliser l'algorithme A* pour rerouter un segment d'arbre d'interconnexion. Il démontre mathématiquement que si l'on considère deux arbres $\mathcal{A}_1$ et $\mathcal{A}_2$ sur un graphe de routage $G(S, A)$ et que l'on choisit arbitrairement pour chacun un sommet représentant (notés s_1 et s_2), alors la chaîne de coût minimal reliant $\mathcal{A}_1$ et $\mathcal{A}_2$ est la même que celle reliant s_1 à s_2 à condition que les arêtes traversées par les arbres $\mathcal{A}_1$ et $\mathcal{A}_2$ aient un coût nul.

Puisque par définition une composante connexe est un arbre, on peut adapter cette solution à l'interconnexion de deux composantes connexes.

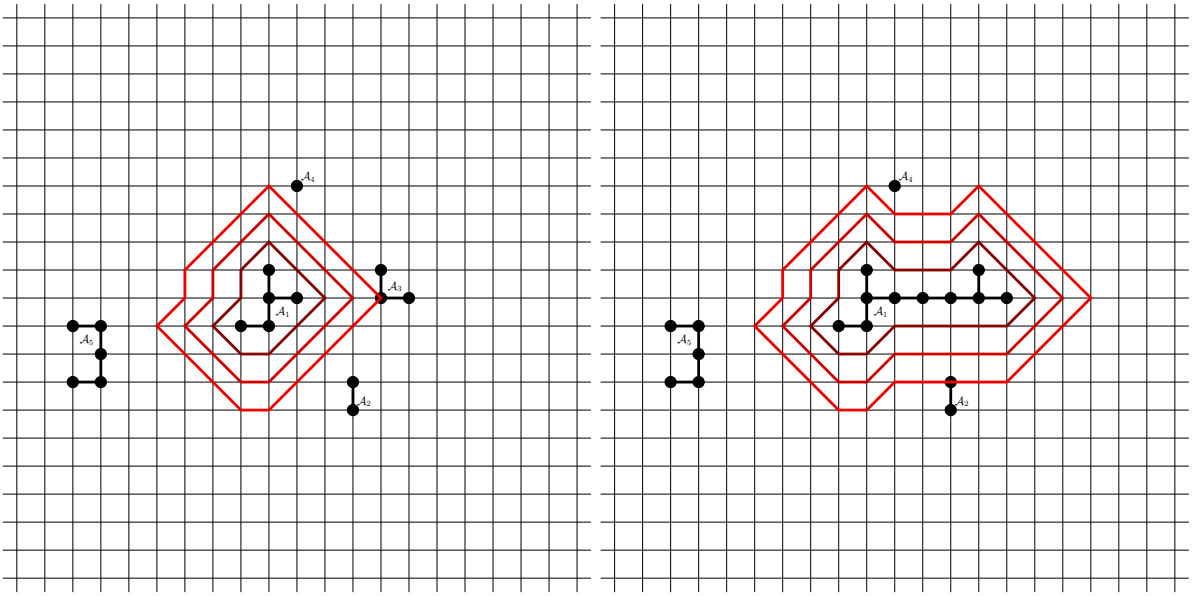
2.3.5 Traitement multi composantes

Dans tous les exemples précédents, nous cherchons à interconnecter deux composantes connexes, ponctuelles ou non. Mais dans le cas d'un net composé de plus de deux composantes connexes, la question se pose de savoir s'il faut utiliser un équivalent de la décomposition en bipoints ou si l'on peut traiter l'ensemble des composantes connexes du net globalement.

Une décomposition similaire à la décomposition en bipoints sous-entend d'être capable de construire un arbre couvrant ou un arbre de Steiner interconnectant les composantes connexes, ce qui, à priori, n'est pas un problème simple.

En revanche l'expansion par vagues de l'algorithme de Dijkstra se prête très bien à la manipulation multi composantes.

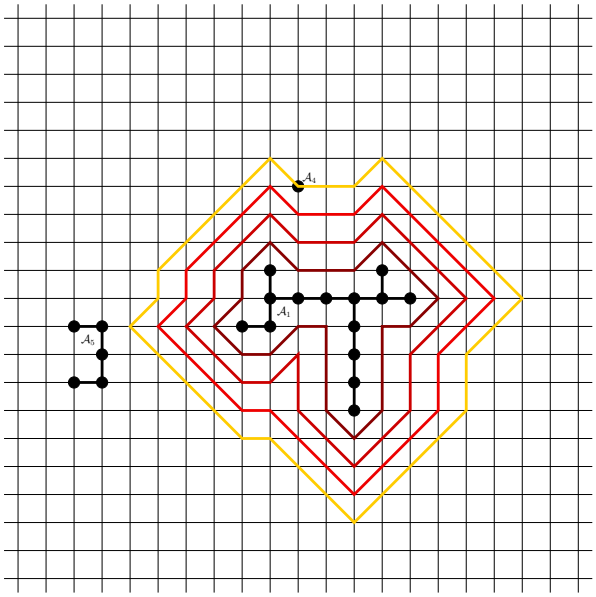
Prenons l'exemple concret d'un net composé de 5 composantes connexes : $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{A}_4$ et $\mathcal{A}_5$. La composante connexe $\mathcal{A}_1$ est considérée comme étant la source (par exemple parce qu'elle représente l'émetteur du net).
L'algorithme commence par propager le coût des sommets de $\mathcal{A}_1$, puis par propagation du bord de la vague finit par atteindre un sommet d'une autre composante connexe, par exemple $\mathcal{A}_3$ (voir figure 2.29(a)).



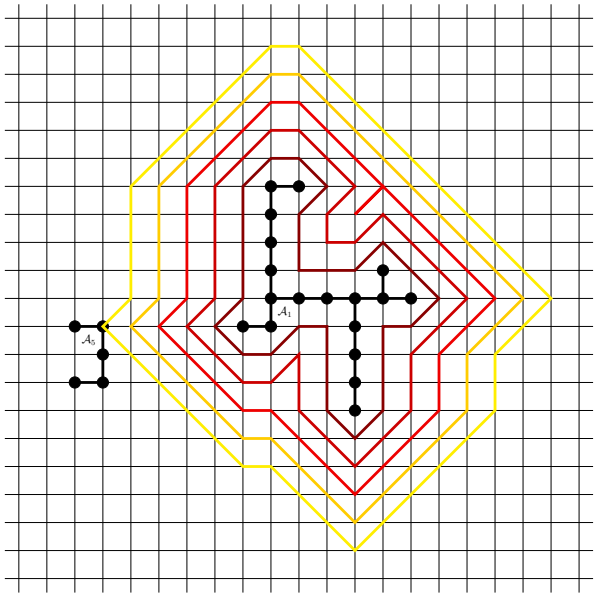
(a) Etape 1 (b) Etape 2

Grâce à la chaîne construite, les deux composantes connexes $\mathcal{A}_1$ et $\mathcal{A}_3$ sont interconnectées ; nous pouvons alors les fusionner pour former une nouvelle composante connexe $\mathcal{A}_1$. Cette nouvelle composante connexe devient la nouvelle source et le processus de propagation recommence (voir figure 2.29(b)). En procédant de la même manière pour

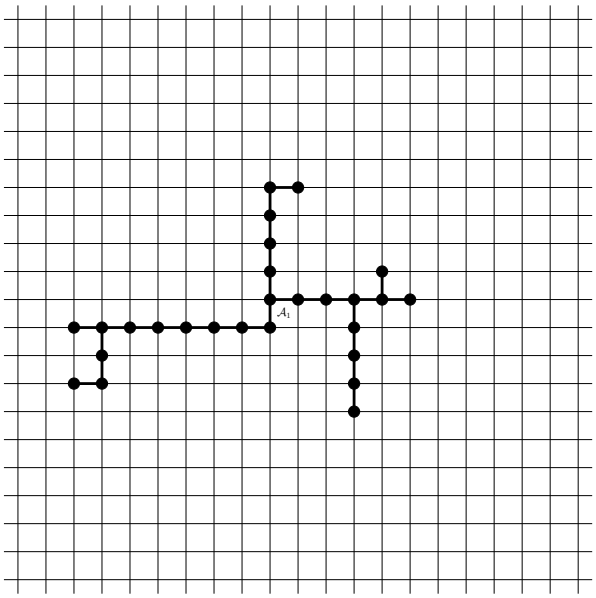
interconnecter les autres composantes connexes, l'algorithme finit par construire un arbre d'interconnexion reliant toutes les composantes connexes (voir figure 2.29(e)).



(c) Etape 3



(d) Etape 4



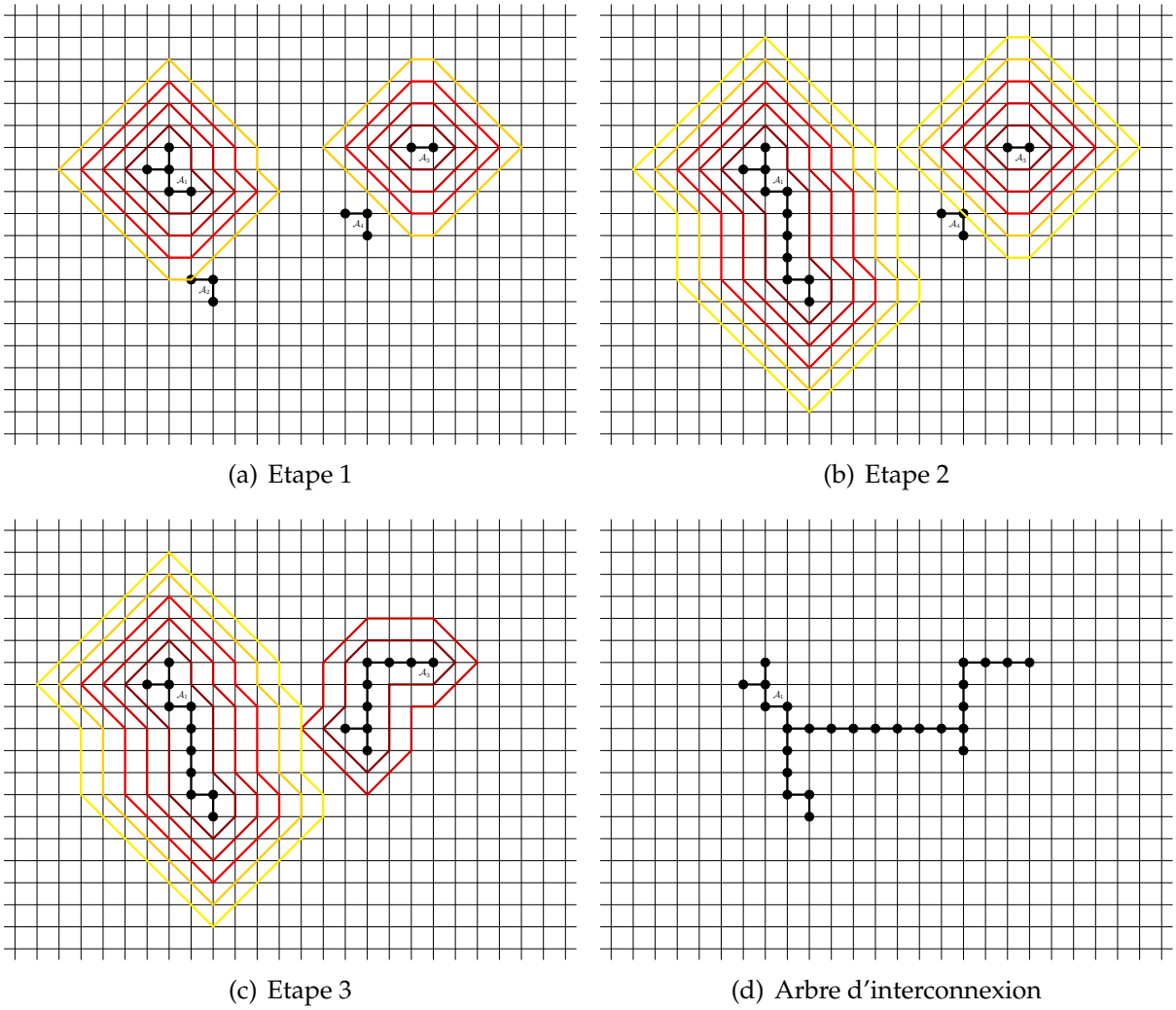
(e) Arbre d'interconnexion

FIGURE 2.29 – Etapes d'un exemple de traitement multi composantes

Cette approche permet même de considérer plusieurs composantes connexes sources simultanément. Il y a alors plusieurs vagues de propagation (et donc plusieurs bords de vagues), lorsque l'une d'elles rencontre une composante connexe source ou une autre vague, une chaîne d'interconnexion est créée.

La figure 2.30 présente un exemple de déroulement de l'algorithme de Dijkstra avec plusieurs composantes connexes sources.

Les composantes connexes $\mathcal{A}_1$ et $\mathcal{A}_3$ sont considérées comme des composantes sources. L'algorithme de Dijkstra propage les coûts jusqu'à ce que la composante $\mathcal{A}_1$ rencontre la composante $\mathcal{A}_2$. Ces deux composantes sont alors fusionnées et la composante résultante devient une nouvelle source (voir figures 2.30(a) et 2.30(b)).



(a) Etape 1 (b) Etape 2

(c) Etape 3 (d) Arbre d'interconnexion

FIGURE 2.30 – Etapes d'un exemple de traitement multi composantes avec plusieurs composantes connexes sources

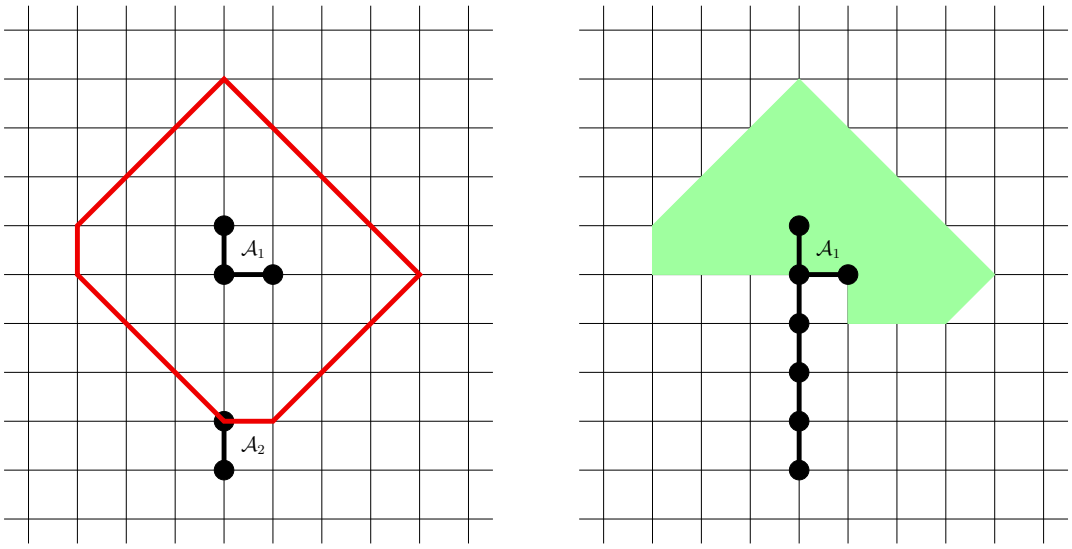
La figure 2.30(c) illustre le cas où les expansions de deux composantes connexes se rencontrent et la figure 2.30(d) présente l'arbre d'interconnexion construit.

2.3 Arbres d'interconnexion et algorithme de Dijkstra

Il est important de noter que pour accélérer l'agorithme de Dijkstra, il est préférable, après fusion de deux composantes connexes, de ne pas réintialiser le coût des sommets parcourus, ni le bord de la vague correspondant à la composante source fusionnée. En effet, la chaîne de coût minimum vers cette composante source ne varie pas pour certains des sommets parcourus, même après fusion des composantes.

Sur la figure 2.31(a), nous avons représenté le bord de la vague lorsque la propagation des coûts des sommets de $\mathcal{A}_1$ a atteint la composante $\mathcal{A}_2$.
Après fusion des deux composantes (figure 2.31(b)), les sommets couverts par la zone verte ont toujours un coût minimal ; il n'est donc pas nécessaire de réinitialiser leur coût. De plus, les sommets à la limite de cette zone (dont le coût doit être propagé) sont déjà présents dans le bord de la vague, puisque nous ne l'avons pas réinitialisé.

Les autres sommets du bord de la vague seront mis à jour lors de la propagatoin du coût des sommets de la nouvelle composante connexe source, il n'est donc pas nécessaire de les supprimer du bord de la vague.



(a) Etape 1 (b) Etape 2

FIGURE 2.31 – Non réinitialisation des coûts des sommets après fusion de deux composantes connexes

Cette technique nous permet d'économiser le temps de parcours des sommets de la zone verte (correspondant à une insertion puis une extraction du bord de la vague) et favorise la mise à jour du coût de sommets déjà présent dans le bord de la vague, ce qui présente un intérêt comme nous le verrons dans le chapitre suivant.

Conclusion

Dans ce chapitre nous avons détaillé l'utilisation des méthodes séquentielles de type *Maze routing*, qui sont les plus adaptées pour résoudre le problème de routage global.

Pour construire un arbre d'interconnexion de coût minimal, nous définissons une fonction de coût prenant en compte la congestion instantanée, la longueur totale des interconnexions, le nombre de vias ainsi qu'une estimation anticipée de la congestion. Cette estimation permet d'éviter que des décisions arbitraires et non remises en cause par la suite soient prises lors du traitement des premiers nets. Nous avons vu que pour obtenir les meilleurs résultats, cette estimation doit être actualisée au fur et à mesure des nets traités.

L'algorithme de Dijkstra et sa variante l'algorithme A* que nous avons présentés, utilisent cette fonction de coût pour construire des arbres d'interconnexion optimaux. Nous avons vu que ces deux algorithmes peuvent être adaptés au traitement de composantes connexes (ponctuelles ou non).

De plus nous avons proposé une approche multi composantes pour l'algorithme de Dijkstra, permettant de traiter simultanément toutes les composantes connexes d'un net. Cette approche est celle utilisée dans notre outil de routage global dont nous détaillons la mise en œuvre dans le chapitre suivant.

Chapitre 3 KNIK routeur global pour la plate-forme CORIOLIS

Dans ce chapitre nous détaillons les structures de données utilisées dans **KNIK** l'outil de routage global de la plate-forme **CORIOLIS**. Notre outil utilise une méthode séquentielle à base d'algorithme de Dijkstra étendue aux composantes connexes.

Nous présentons tout d'abord le graphe de routage qui est la structure centrale de notre outil. Il permet de modéliser simplement les ressources de routage et offre toutes les fonctionnalités nécessaires à l'utilisation d'un algorithme de routage global.

Ensuite, nous expliquons comment les composantes connexes sont représentées sur le graphe de routage et utilisées par notre outil. Nous introduisons les concepts de `netStamp` et `connexId` qui permettent une gestion intelligente du graphe de routage en association avec une méthode séquentielle.

Pour finir, nous présentons les spécificités de notre mise en œuvre de l'algorithme de Dijkstra et du *ripup & reroute* adaptée à l'utilisation du graphe de routage et des composantes connexes.

3.1 Graphe de routage

Le graphe de routage étant la structure centrale de notre outil, nous avons défini en son sein, plusieurs fonctionnalités utiles pour le routage global.

Dans cette section, nous présentons dans un premier temps notre mise en œuvre du graphe qui permet une gestion rapide et légère pour un graphe régulier ou irrégulier.
Nous détaillons ensuite l'ensemble des fonctionnalités additionnelles du graphe : nous présentons d'abord, la technique utilisée pour retrouver le sommet du graphe associé aux coordonnées d'un point quelconque de la surface du circuit. Puis nous détaillons le calcul de l'estimation anticipée de la congestion ainsi que la façon dont elle est gérée sur le graphe de routage. Nous introduisons ensuite le principe de matérialisation du routage global qui permet de transformer un arbre d'interconnexion en un ensemble de segments physique de métal occupant certaines ressources de routage. Enfin nous

présentons la représentation graphique du graphe de routage ainsi que l'exploration de la structure mémoire associée permise par la plate-forme **CORIOLIS**.

3.1.1 Mise en œuvre

Le graphe de routage utilisé par l'outil **KNIK** est un graphe bidimensionnel $G(S, A)$ tel que celui que nous avons décrit précédemment.

Ce graphe est le dual d'un pavage, régulier ou non, de la surface du circuit. Chaque sommet représente un pavé et les arêtes incidentes le relient aux sommets représentant des pavés adjacents.

Puisque le graphe peut être irrégulier, il n'est pas possible de le représenter sous forme matricielle en mémoire. Nous proposons une structure simple composée de deux objets :
- le `vertex` représentant un sommet du graphe,
- l'`edge` représentant une arête.

Chaque arête du graphe est de type horizontal ou vertical. Par définition, une arête verticale est associée à une frontière horizontale, c'est-à-dire qu'elle relie deux sommets associés à des pavés voisins par une frontière horizontale. De la même façon une arête horizontale est associée à une frontière verticale.

Pour pouvoir parcourir rapidement et simplement le graphe de routage, notre mise en œuvre oriente les arêtes du graphe vers les abscisses et ordonnées positives. Pour une arête horizontale (resp. verticale) son sommet source est toujours le plus à gauche (resp. le plus bas) et son sommet destination le plus à droite (resp. le plus haut). En pratique chaque objet `edge` possède un pointeur `from` vers son sommet source et un pointeur `to` vers son sommet destination.

Ceci nous permet de définir quatre types d'arêtes incidentes à un sommet $s \in S$:
- `HEdgeOut` : les arêtes horizontales dont s est la source,
- `VEdgeOut` : les arêtes verticales dont s est la source,
- `HEdgeIn` : les arêtes horizontales dont s est la destination,
- `VEdgeIn` : les arêtes verticales dont s est la destination.

Dans la majorité des cas, un sommet $s \in S$ ne possède, au maximum, qu'une arête de chaque type. Il n'y a que dans le cas d'un pavage irrégulier que plusieurs arêtes incidentes au sommet peuvent être de même type (voir figure 3.1).

Pour alléger la structure, chaque objet `vertex` ne possède qu'un seul pointeur vers une arête par type. Les autres arêtes de même type, si elles existent, sont chaînées à la première grâce aux pointeurs `nextTo` et `nextFrom` de l'objet `edge`.

3.1 Graphe de routage

Le pointeur `nextFrom` (resp. `nextTo`) sert à chaîner les arêtes de même type ayant le même sommet `from` (resp. `to`).

Le parcours des arêtes incidentes à un sommet se fait dans le sens inverse des aiguilles d'une montre en partant de la première arête de type `HEdgeOut`. Les arêtes sont donc chaînées dans le même sens (voir figure 3.1). Une fois toutes les arêtes de type `HEdgeOut` parcourues, on passe à la première de type `VEdgeOut` et ainsi de suite jusqu'à la dernière.

La figure 3.1 présente un exemple de graphe de routage irrégulier sur lequel nous avons représenté les pointeurs utilisés dans la structure mémoire. Ces pointeurs sont représentés par des flèches dont la base est associée à l'objet possédant le pointeur tandis que l'extrémité pointe vers l'objet ciblé.

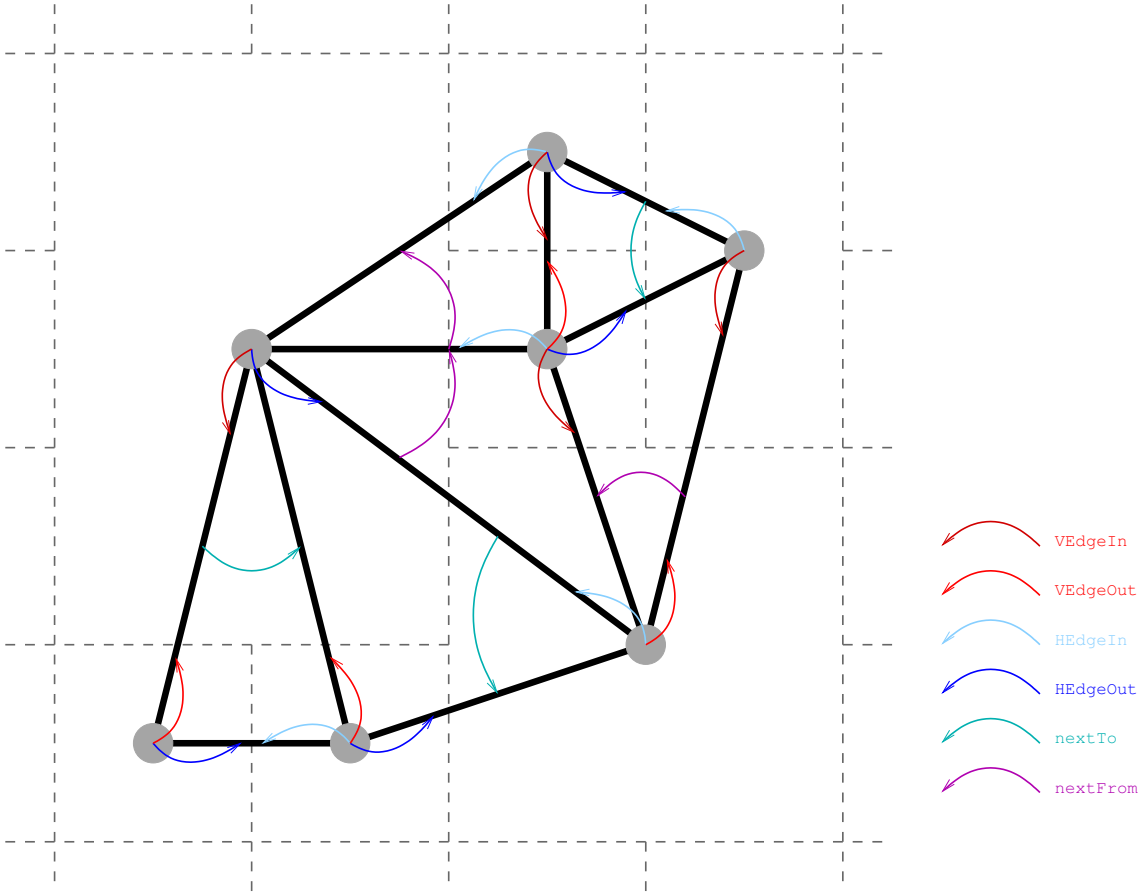


FIGURE 3.1 – Chaînage des arêtes de même type

Pour représenter les graphes dédiés servant de base aux composantes connexes représentant des connecteurs répartis de macro-blocs, nous ajoutons à l'objet `vertex` un booléen (`looseComponent`) servant à identifier rapidement qu'un `vertex` appartient ou non à une composante connexe construite sur un graphe dédié. Nous

associons à ce booléen un vecteur de pointeurs sur `edge` représentant les arêtes de la composante connexe. Notons que nous ne représentons pas en mémoire les arêtes du graphe dédié qui n'appartiennent pas à la composante connexe puisque l'algorithme de routage ne les utilise jamais.

Dans l'optique d'une utilisation du graphe pour le routage global, nous ajoutons à chaque arête une capacité (`capacity`), une occupation instantanée (`realOccupancy`) et une occupation estimée (`estimatedOccupancy`) et à chaque sommet un pointeur vers le sommet prédécesseur (`predecessor`) ainsi qu'un coût (`cost`) par rapport au sommet source (pour représenter les chaînes de routage).
Nous obtenons alors une empreinte mémoire assez faible (52 octets pour l'objet `vertex` et 48 pour l'objet `edge`).

Les circuits de tests conçus pour le routage global (ispd98 [ben] et ispd07 [oPDGRCa]) fournissent tous les éléments nécessaires à la construction d'un graphe de routage régulier : la position des sommets et la capacité des arêtes suivant la couche de métal (horizontale ou verticale).

En revanche, dans le cadre d'une utilisation de notre outil sur des circuits réels, ces deux informations sont définies grâce à des fonctions internes à la plate-forme **CORIOLIS** .

Comme nous l'avons décrit dans le chapitre 1 nous considérons pour le pavage associé au graphe une hauteur de pavé égale à la hauteur d'une cellule de bibliothèque pré-caractérisée et une largeur du même ordre de grandeur, adaptée à la largeur totale du circuit. Cette taille est bien adaptée puisqu'elle est suffisament grande pour obtenir un graphe de routage de taille raisonnable tout en étant suffisamment petite pour que la topologie des nets, locale à chaque pavé, offre suffisamment de degré de liberté au routage détaillé sans trop de complexité. Il faut noter en outre que le nombre de nets locaux à un pavé augmente avec sa taille ce qui génère des obstacles de routage que la modélisation simplifiée ne prend pas en compte.

La capacité des arêtes dépend, elle, des données technologiques. La plate-forme **CORIOLIS** dispose d'un module permettant, à partir d'un intervalle de coordonnées $[y_1, y_2]$ (resp. $[x_1, x_2]$), de calculer le nombre de pistes de routage horizontales (resp. verticales) disponibles sur l'ensemble des couches de métal considérées, c'est-à-dire la capacité. Pour ce calcul, le module prend en compte les obstacles et les pré-tracés éventuels (alimentations, horloges, ...). Pour calculer la capacité d'une arête horizontale (resp. verticale), il nous suffit alors de faire appel à ce module en lui fournissant l'intersection des intervalles $[y_{min}, y_{max}]$ (resp. $[x_{min}, x_{max}]$) des deux pavés associés aux sommets qu'elle relie.

Au final, nous avons défini une structure occupant peu de ressources mémoire

et permettant un accès simple et rapide à tous les éléments du graphe de routage.

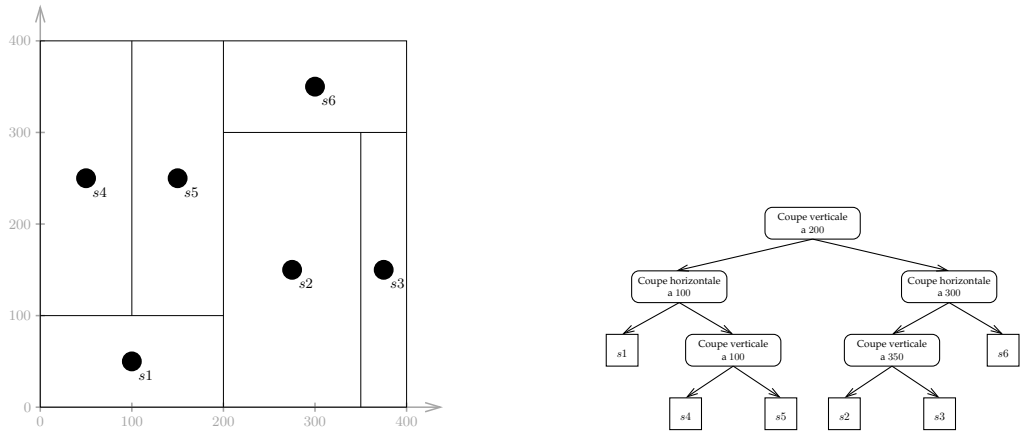
3.1.2 Recherche d'un sommet associé à un point quelconque de la surface du circuit

Pour construire un arbre d'interconnexion d'un net sur le graphe de routage, il faut d'abord décomposer ce net en composantes connexes. Pour cela il est nécessaire d'être capable de retrouver le sommet associé à un point quelconque de la surface du circuit. Cette association est définie par le pavé contenant le point.

Arbre de découpage

Il s'agit donc d'être capable de retrouver le sommet associé à un point quelconque de la surface du circuit. Pour cela nous avons tout d'abord utilisé une structure de type « arbre de découpage » (*slicing tree*).
A partir du pavage de la surface du circuit, on construit un arbre binaire dont les nœuds représentent les coupes verticales et horizontales du pavage et les feuilles représentent les sommets du graphe associés aux différents pavés.

La figure 3.2(a) présente un exemple de pavage ainsi que les sommets associés aux pavés. Les axes représentant la métrique du plan, permettent de trouver la coordonnée en $lambda$ [1] associée à chaque coupe (horizontale ou verticale).
La figure 3.2(b) illustre l'arbre de découpage associé. Considérons le point de coordonnées $(150, 210)$ et utilisons l'arbre de découpage pour retrouver le sommet associé.



(a) Pavage de la surface du circuit (b) Arbre de découpage associé

FIGURE 3.2 – Principe de l'arbre de découpage

1. Le $lambda$ est l'unité de mesure utilisée dans les technologies symboliques.

Partant de la racine de l'arbre il suffit de comparer les coordonnées du point avec celle de la coupe associée au nœud de l'arbre. Pour le premier nœud, la coupe est verticale avec une coordonnée x égale à $200\ lambdas$. La coordonnée x du point étant inférieure (150) il faut suivre le fils gauche du nœud.

En procédant de la même façon pour les nœuds suivant, on atteint une feuille de l'arbre : $s5$. Le point de coordonnées $(150, 210)$ est donc associé au sommet $s5$, comme nous pouvons le vérifier visuellement sur la figure 3.2(a).

Dans le cas d'une coordonnée de point égale à celle de la coupe, on choisit par convention de suivre le fils droit du nœud.

Dans l'exemple précédent nous avons choisi la coupe du nœud racine de façon à obtenir un arbre de découpage équilibré ce qui permet d'accélérer la recherche. Le choix de la coupe racine est très important puisqu'il détermine le fait que l'arbre de découpage est équilibré ou non. Or un déséquilibre augmente le temps de recherche pour les feuilles les plus profondes.

De plus la mise en œuvre de ce type d'arbre n'est pas simple et peut conduire à une lenteur générale de l'algorithme de recherche : cette fonction de recherche d'un sommet associé à un point quelconque du circuit est utilisée intensivement lors de la création des composantes connexes, de la phase de mise à jour dynamique de la congestion ou encore de *ripup & reroute*. Il est donc nécessaire qu'elle soit (très) rapide.

Matrice de `vertex`

Dans le cas d'un graphe régulier, la division des coordonnées du point par le pas de grille suffit à trouver les indices de ligne et colonne du sommet. La limite à l'utilisation de cette méthode et donc à une exécution rapide réside dans le fait de considérer un graphe irrégulier.

Nous définissons une structure qui permet de « voir » le graphe de routage irrégulier comme un graphe régulier et d'utiliser des méthodes très simples de calcul des indices. Pour cela nous complétons virtuellement les lignes de coupes du pavage (comme illustré sur la figure 3.3).

Nous obtenons alors un pavage régulier mais dont la largeur et la hauteur des pavés n'est pas constante. Nous utilisons deux vecteurs d'indexation (`rowsIndex` et `columnsIndex`) qui associent les coordonnées des coupes aux indices de lignes et colonnes d'une matrice. Cette matrice contient des pointeurs vers les objets `vertex` du graphe de routage. Un même `vertex` peut être référencé par plusieurs cases de la matrice.

3.1 Graphe de routage

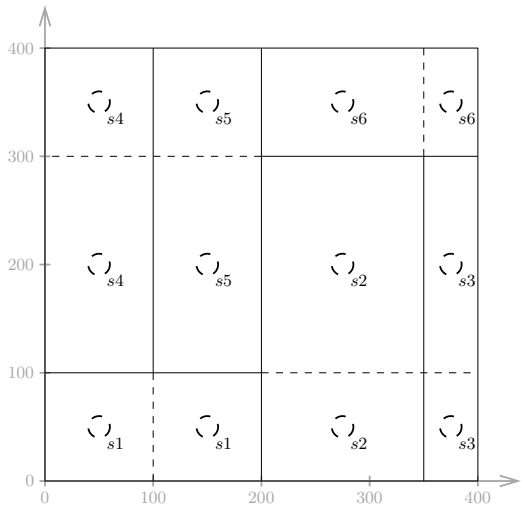


FIGURE 3.3 – Découpage virtuel du graphe irrégulier

Les deux vecteurs d'indexation contiennent non seulement les coordonnées des coupes mais aussi les coordonnées des frontières du circuit pour faciliter la recherche d'indice.

Pour trouver le sommet associé au point de coordonnées (x, y), on commence par chercher les indices associés.
Dans le vecteur `rowsIndex`, on cherche l'indice i tel que `rowsIndex[`i`]` $<= y$ et `rowsIndex[`$i+1$`]` $> y$. Puis on cherche de la même façon l'indice j correspondant à la coordonnée x, à partir du vecteur `columnsIndex`.
Une fois les indices i et j trouvés, il suffit de suivre le pointeur contenu dans `matrice[`i`][`j`]` pour atteindre le sommet associé.

La structure correspondant à l'exemple de la figure 3.2(a) est la suivante :

`matrice`

	0	1	2	3
0	$s1$	$s1$	$s2$	$s3$
1	$s4$	$s5$	$s2$	$s3$
2	$s4$	$s5$	$s6$	$s6$

	0	1	2	3	4
`rowsIndex`	0	100	300	400	
`columnsIndex`	0	100	200	350	400

A partir du point de coordonnées $(150, 210)$, on trouve les indices $i = 1$ et $j = 1$, puis en suivant le pointeur contenu dans `matrice[1][1]` on atteint bien le sommet $s5$.

Grâce à l'utilisation de la STL (Standard Template Library [Lib]), la recherche d'un intervalle contenant une valeur à l'intérieur d'un vecteur trié est très rapide. Cette structure de données permet donc de gérer un graphe de routage irrégulier tout en conservant une recherche très rapide.

3.1.3 Evaluation de la congestion

Dans l'outil **KNIK** , nous avons choisi de mettre en œuvre une méthode utilisant congestion instantanée et estimation anticipée de la congestion de façon à éviter de prendre des décisions arbitraires pour les premiers nets traités. Le taux de congestion d'une arête est calculé à la volée à partir de l'occupation instantanée, l'occupation estimée et la capacité de l'arête. Les occupations instantanées et estimées correspondent aux congestions instantanées et estimées définies page 56.

Toutes ces valeurs étant directement liées aux arêtes du graphe, nous introduisons les trois attributs suivants dans l'objet `edge` :
- `capacity` représentant la capacité de l'arête,
- `realOccupancy` représentant l'occupation instantanée,
- `estimatedOccupancy` représentant l'occupation estimée.

Le calcul de l'estimation anticipée de la congestion se fait en deux temps. Il faut tout d'abord décomposer chaque net en un ensemble bipoints. Puis pour chaque bipoint on calcule les probabilités en ne considérant qu'un nombre restreint de chaînes d'interconnexion reliant les deux sommets du bipoint. Il suffit ensuite de reporter les probabilités sur l'occupation estimée des arêtes du graphe de routage.

Décomposition en bipoints à l'aide de FLUTE

La méthode mise en œuvre dans l'outil **KNIK** est similaire à celle décrite dans le chapitre 2. Nous utilisons la bibliothèque **FLUTE** [fRCE] pour décomposer les nets en bipoints.

L'outil **FLUTE** construit un arbre de Steiner rectilinéaire de longueur minimale à partir d'un ensemble de points. Tant que le nombre de points est inférieur ou égal à 9, l'arbre construit est optimal. Au-delà l'arbre reste très proche de l'optimum.

L'outil **FLUTE** étant rapide[2] et déterministe il peut être utilisé dans le cadre de

2. La construction d'un arbre pour un million de points se fait en 531 secondes sur ordinateur équipé d'un processeur Pentium 4 1.8GHz et de 256 Mo de mémoire vive.

la gestion dynamique de la congestion anticipée.

L'ensemble des points à relier est donné à **FLUTE** sous la forme des coordonnées (x, y) des points. **FLUTE** renvoie l'ensemble bipoint de l'arbre construit sous la forme d'une liste des couples de points représentés par leur coordonées x, y. A partir de cet ensemble bipoints et en considérant la grille de Hanan, il est très simple de reconstruire l'ensemble des arbres de Steiner rectilinéaires de longueurs minimales équivalentes interconnectant les points.

Prenons l'exemple de la figure 3.4 pour lequel nous considérons un net composé de quatre connecteurs à relier. L'ensemble bipoint construit par **FLUTE** est représenté sur la figure 3.4(b). En considérant la grille de Hanan (représentée en noir), on peut aisément construire les quatres arbres de Steiner de longueur minimale équivalente représentés sur les figures 3.4(c), 3.4(d), 3.4(e) et 3.4(f).

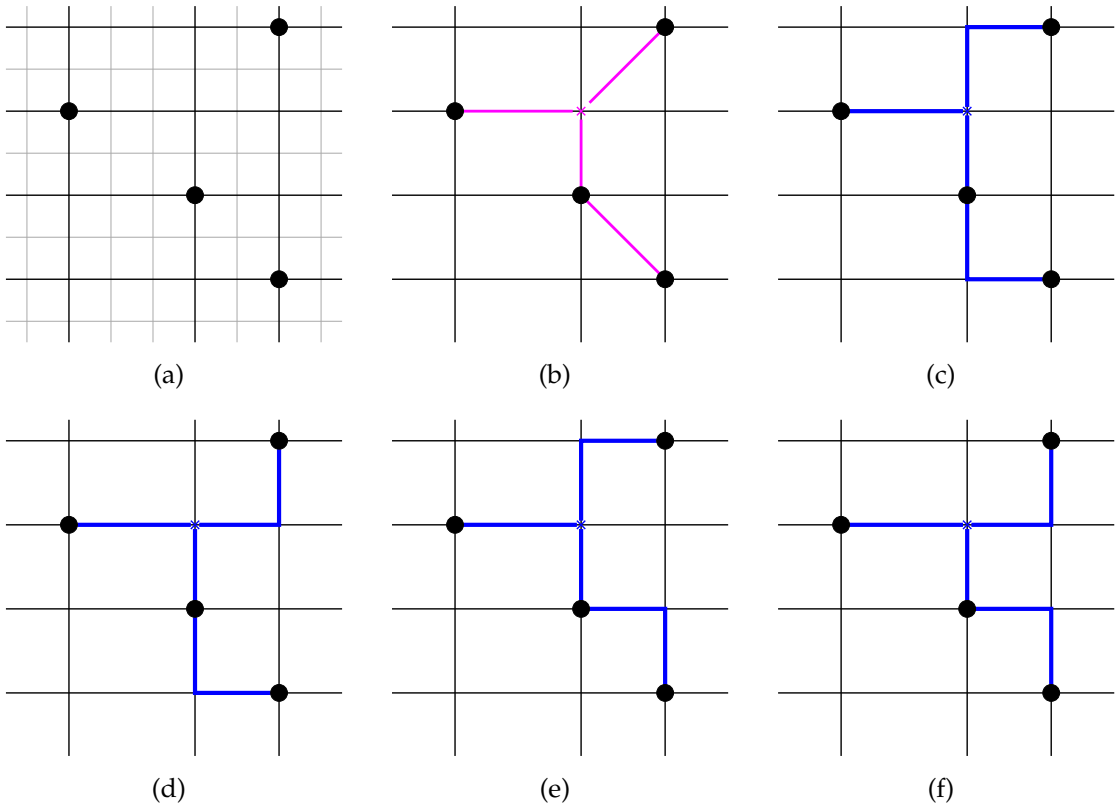


FIGURE 3.4 – Ensemble bipoint créé par **FLUTE** et arbres de Steiner associés

Les points à relier de cet exemple correspondent à des composantes connexes ponctuelles. Dans le cas de composantes connexes non ponctuelles, une méthode simple consiste à ne fournir à **FLUTE** que le sommet représentant de chaque composante connexe ; on rejoint alors le cas précédent. La figure 3.5 illustre cette méthode. Sur les trois composantes connexes ($\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$) présentées sur la figure 3.5(a), deux sont

non ponctuelles. Leur sommet représentant apparaît sous forme d'un cercle magenta plein. Ne connaissant que ces sommets, l'outil **FLUTE** construit l'arbre représenté sur la figure 3.5(b).

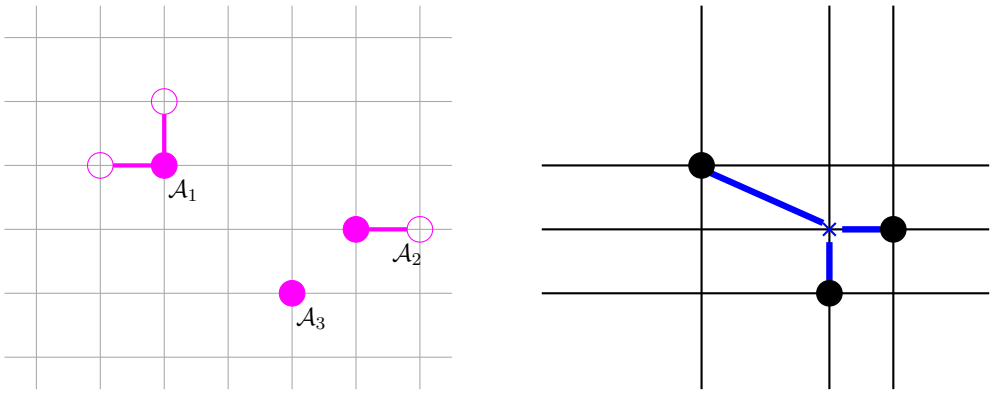


FIGURE 3.5 – Arbre construit par l'outil **FLUTE** en considérant les représentants des composantes connexes

Mais la réduction d'une composante connexe à son seul représentant peut conduire, dans certains cas, à une estimation de la congestion faussée. L'arbre construit par **FLUTE** est optimal pour relier les représentants des composantes connexes, mais si l'on considère les composantes connexes complètes, l'arbre n'est plus optimal. En calculant l'estimation de congestion à partir de cet arbre, on considère une congestion qui ne reflète pas la future congestion instantanée.

La figure 3.6 illustre ce problème. Considérant les représentants des trois composantes connexes $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_3$ de la figure 3.6(a), l'outil **FLUTE** construit l'arbre illustré sur la figure 3.6(b) tandis que l'arbre de longueur minimale reliant les composantes connexes est représenté sur la figure 3.6(c).

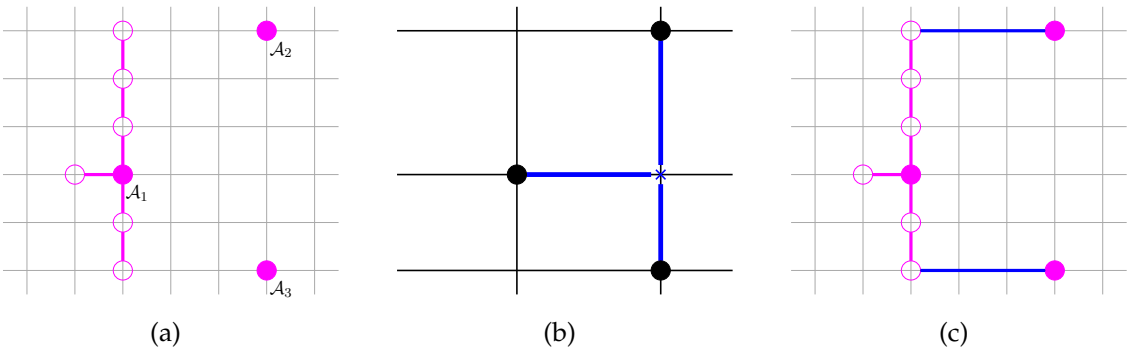


FIGURE 3.6 – Exemple de cas particulier menant à une estimation de congestion faussée

Pour pallier ce problème nous proposons une autre méthode consistant à fournir en entrée de **FLUTE** tous les sommets formant la composante connexe. De cette manière, l'outil **FLUTE** a connaissance de toutes les possibilités de connexion pour les composantes connexes.

La figure 3.7 présente l'arbre construit par **FLUTE** pour l'exemple de la figure 3.6(a) en utilisant cette méthode. Cet arbre est bien un arbre de longueur minimale reliant les composantes connexes (voir figure 3.6(c)) et permet donc de calculer une estimation de la congestion plus juste sans pour autant complexifier le traitement.

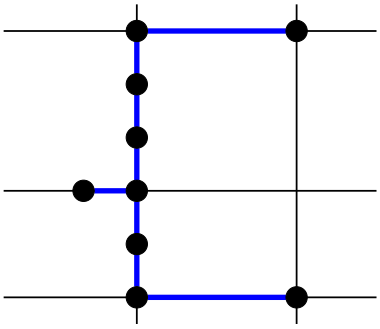
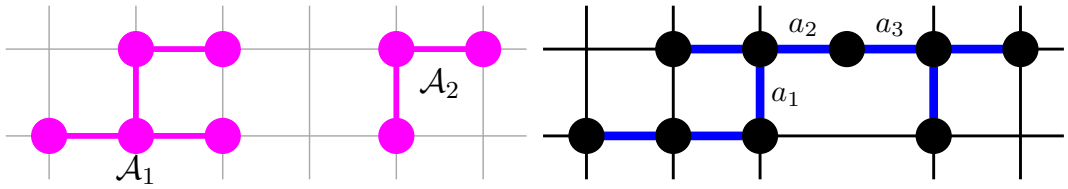


FIGURE 3.7 – Arbre construit par **FLUTE** en considérant tous les sommets des composantes

Il est important de noter que contrairement à la première méthode, pour calculer les probabilités, il ne faut ici pas tenir compte de toutes les arêtes de l'arbre construit par **FLUTE**. Les arêtes de l'arbre appartenant à une composante connexe n'occuperont aucune ressource et donc ne doivent pas entrer dans le calcul de l'estimation.

Dans le cas particulier d'une composante connexe représentant un connecteur étalé, le graphe associé $G(T_{ik}, A)$ peut contenir des cycles ; un problème apparaît alors quant aux arêtes qui doivent être considérées pour le calcul des probabilités. Si le graphe $G(T_{ik}, A)$ contient des cycles, il existe plusieurs composantes connexes possibles et le routeur en choisit une arbitrairement. Mais rien n'assure que l'arbre construit par **FLUTE** recouvre exactement les arêtes de la composante connexe choisie, comme l'illustre la figure 3.8.
On constate bien que la composante connexe $\mathcal{A}_1$ présentée sur la figure 3.8(a) n'est pas couverte pas l'arbre construit par **FLUTE** représenté sur la figure 3.8(b). L'arête notée a_1, n'appartient à aucune des deux composantes connexes et ne doit pourtant pas être considérée pour le calcul des probabilités puisqu'elle relie deux sommets déjà électriquement connexes.

Pour pallier ce problème, il suffit de considérer que toute arête de l'arbre construit par **FLUTE** reliant deux sommets appartenant à la même composante connexe doit être ignorée, même si elle n'appartient pas à une composante connexe. Sur l'exemple de



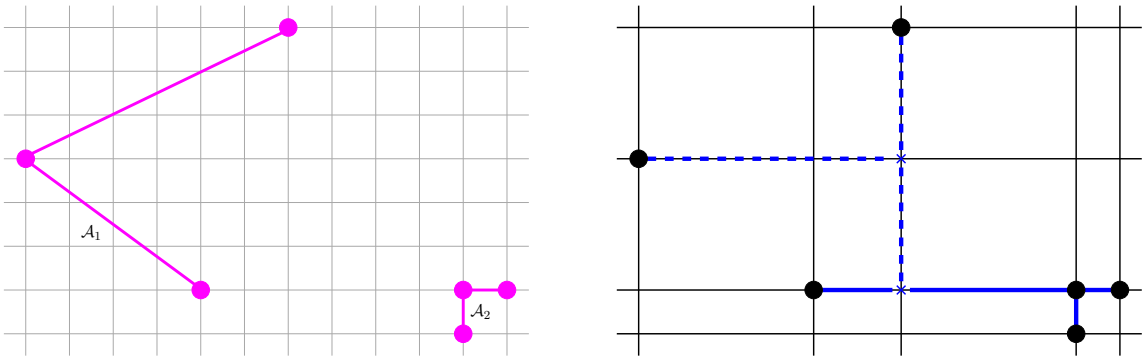
(a) (b)

FIGURE 3.8 – Non correspondance entre l'arbre construit par **FLUTE** et les composantes

la figure 3.8(b), seules les arêtes a_2 et a_3 sont alors considérées. Nous présentons plus loin les détails du mécanisme utilisé pour calculer l'estimation à partir d'un arbre créé par **FLUTE** , et quelles arêtes de cet arbre sont prises en compte.

Malheureusement cette formulation ne suffit pas à gérer tous les cas de composantes connexes. Pour le cas d'une composante connexe représentant un connecteur réparti de macro bloc, **FLUTE** ne peut pas prendre en compte le graphe dédié associé. Si l'on considère tous les sommets de la composante connexe, la congestion estimée calculée à partir de l'arbre construit par **FLUTE** contient des zones faussement congestionnées.

Dans l'exemple de la figure 3.9, nous considérons deux composantes connexes : $\mathcal{A}_1$ représentant un connecteur réparti dans un macro bloc et $\mathcal{A}_2$ un connecteur étalé (voir figure 3.9(a)). En observant l'arbre construit par **FLUTE** pour relier tous les sommets des composantes sur la figure 3.9(b), on constate que l'estimation de congestion est faussée puisque l'arbre que construirait le routeur global sans prise en compte de la congestion ne contient aucune des arêtes représentées en pointillés.



(a) (b)

FIGURE 3.9 – Arbre construit par **FLUTE** pour un connecteur réparti avec sur-estimation

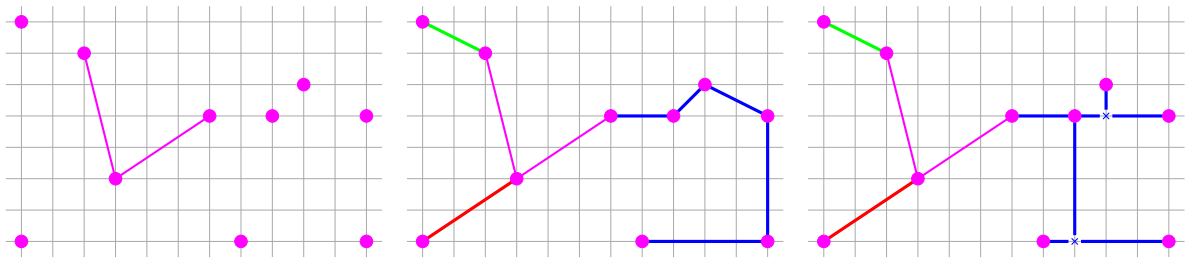
Nous proposons un traitement en deux étapes pour réaliser la décomposition en

bipoints d'un net possédant au moins une composante connexe représentant un connecteur réparti.

La première étape consiste à construire un arbre couvrant à l'aide d'un algorithme de Prim [adP] en considérant en entrée à la fois les sommets constituant les composantes connexes mais aussi les arêtes. Puis pour calculer les probabilités, nous ne considérons que les arêtes de l'arbre créé par l'algorithme de Prim.

Dans un cas simple comme celui de la figure 3.9(a), cette étape suffit à construire une décomposition satisfaisante pour calculer l'estimation de la congestion. Mais dans certains cas l'arbre peut conduire à une estimation faussée.

La deuxième étape permet d'améliorer la décomposition en bipoints. En ne considérant que les arêtes construites par l'algorithme de Prim, nous obtenons un ensemble de sous graphes connexes. Parmi ces sous graphes, nous considérons ceux ayant plus de deux sommets et nous utilisons **FLUTE** pour reconstruire une décomposition en bipoints. La figure 3.10 illustre cette méthode.



(a) Composantes connexes (b) Arbre couvrant (c) Amélioration du sous graphe (bleu) par **FLUTE**

FIGURE 3.10 – Décomposition en bipoints grâce à l'algorithme de Prim puis par **FLUTE**

Calcul des probabilités et report sur le graphe

Une fois la décomposition en bipoints effectuée, nous pouvons calculer les probabilités servant à estimer la congestion. L'outil **KNIK** utilise la méthode décrite dans le chapitre 2. Pour des sommets non alignés, nous ne considérons que les chaînes à un seul coude.

Partant de l'ensemble bipoint, il suffit de calculer les probabilités pour l'ensemble des chaînes considérées puis de reporter ces probabilités sur les arêtes du graphe de routage. Pour cela il suffit, pour chaque chaîne, d'ajouter la valeur de sa probabilité à l'ensemble des arêtes qu'elle recouvre.

La figure 3.11 illustre cette procédure.

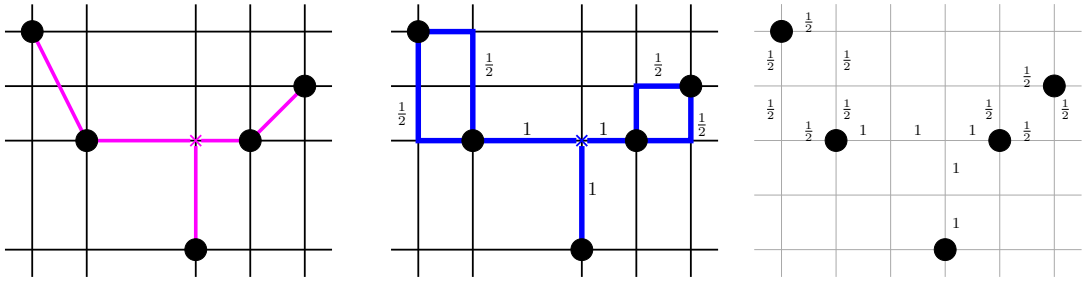


FIGURE 3.11 – Calcul et report des des probabilités de congestion

Cette procédure fonctionne très bien du moment que l'on considère un graphe de routage régulier. Les coordonnées d'un sommet d'un bipoint construit par **FLUTE** correspondent toujours à un sommet du graphe de routage puisqu'elles représentent soit un sommet du graphe donné en entrée de **FLUTE**, soit un nœud de Steiner rajouté par **FLUTE**. Or ces nœuds de Steiner sont toujours sur la grille de Hanan et correspondent donc à un sommet du graphe de routage.

Dans le cas d'un graphe de routage irrégulier, le report des probabilités sur le graphe de routage n'est pas simple. Prenons l'exemple de la figure 3.12 sur laquelle les deux sommets du bipoint (s, d) sont horizontalement alignés. L'outil **KNIK** ne considère qu'une seule chaîne de probabilité 1 pour les relier. La question est de savoir sur quelles arêtes du graphe devons-nous reporter cette probabilité ?

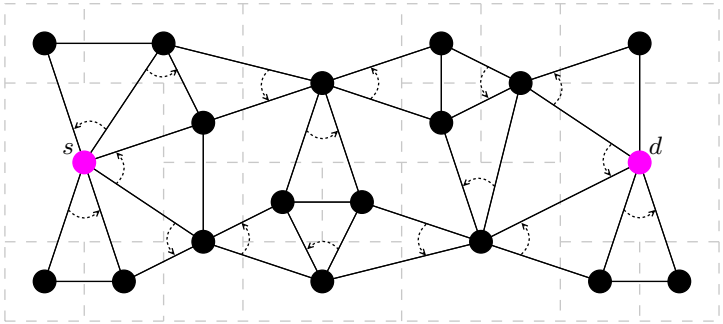


FIGURE 3.12 – Report des probabilités sur un graphe irrégulier

Dans le cas d'un bipoint dont les sommets sont horizontalement (resp. verticalement) alignés, il suffit de partir du sommet le plus à gauche (resp. le plus bas) et de suivre les arêtes de type `HEdgeOut` (resp. `VEdgeOut`). Si plusieurs arêtes de même type existent, on choisit celle menant au sommet le plus proche de la cible. De plus, dans le cas de sommets horizontalement alignés, le pavé associé au sommet suivant et le pavé cible doivent toujours avoir un intervalle commun de coordonnées y. La même condition s'applique aux coordonées x dans le cas d'un alignement vertical.

La figure 3.13 présente (en bleu) les arêtes du graphe sur lesquelles les probabilités doivent être reportées.

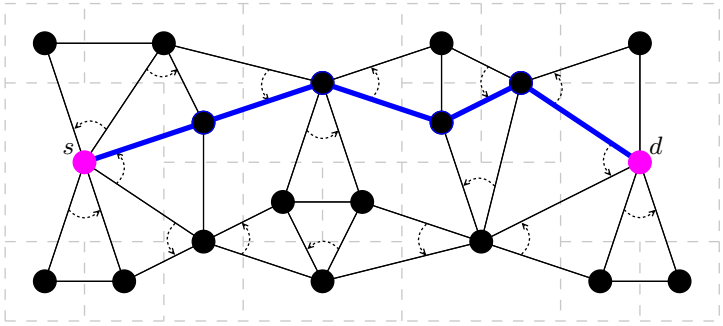


FIGURE 3.13 – Chaîne de report des probabilités pour un graphe irrégulier

Lorsque les deux sommets d'un bipoint ne sont pas alignés, le raisonnement reste assez simple. Il s'agit de trouver les arêtes sur le graphe correspondant aux deux chaînes considérées.

Pour la première chaîne, on suit dans un premier temps les arêtes verticales (`VEdgeOut` ou `VEdgeIn`) tant que la coordonnée y du sommet suivant est comprise dans l'intervalle des coordonnées y du pavé associé au sommet cible. Puis on suit les arêtes horizontales (`HEdgeOut` puisque l'on considére que le sommet source du bipoint est toujours le plus à gauche).

Pour la deuxième chaîne on commence par suivre les arêtes horizontales tant que la coordonnée du sommet suivant est comprise dans l'intervalle des coordonnées x du pavé associé au sommet cible, puis on suit les arêtes verticales.

La figure 3.14 présente (en rouge et bleu) les deux chaînes d'arêtes sur lesquelles les probabilités doivent être reportées.

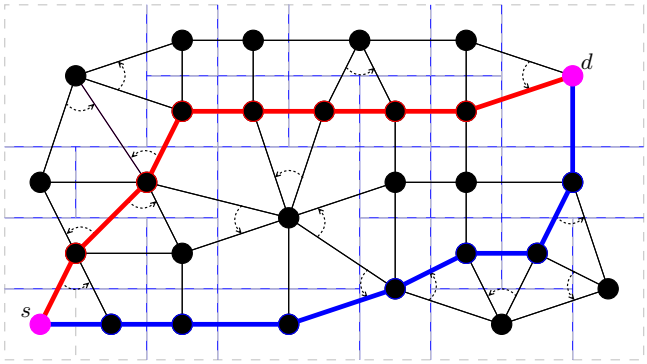


FIGURE 3.14 – Chaînes de report des probabilités pour un graphe irrégulier

Notons enfin qu'il est possible que deux sommets d'un bipoint correspondent au même sommet du graphe comme l'illustre la figure 3.15 sur laquelle nous avons représenté l'arbre construit par l'outil **FLUTE** et le pavage du circuit et où les sommets s et b correspondent au même pavé. Dans un tel cas, nous ignorons simplement le bipoint.

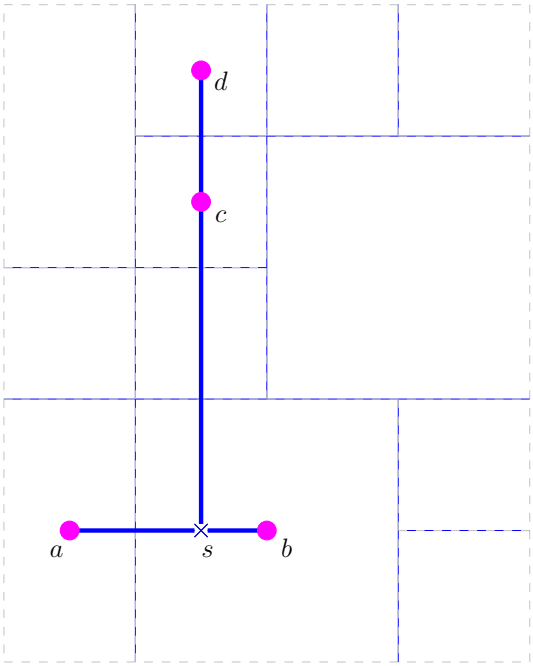


FIGURE 3.15 – Sommets d'un bipoint correspondant au même sommet du graphe de routage

Mise à jour de la congestion estimée

L'outil **KNIK** permet une gestion statique ou dynamique de la congestion estimée. Dans les deux cas le calcul de l'estimation est fait au début de l'algorithme, avant de construire les arbres d'interconnexion des nets à router. Pour la gestion statique, la congestion estimée est calculée une fois pour toute et n'est jamais mise à jour par la suite. En revanche dans le cas d'une gestion dynamique, avant de construire un arbre d'interconnexion pour un net, l'algorithme de routage supprime la congestion estimée correspondant au net.

Supprimer la congestion estimée d'un net consiste à décomposer le net en bipoints puis à retrancher les probabilités correspondantes à la congestion estimée des arêtes du graphe de routage. Il est impératif que lors de cette phase, la décomposition en bipoints soit exactement la même que celle utilisée pour calculer l'estimation, d'où l'importance du déterminisme de l'outil **FLUTE**.

3.1.4 Matérialisation du routage

Outre la recherche d'un sommet et la gestion de la congestion que nous avons présenté, notre mise en œuvre du graphe offre la fonctionnalité de **matérialisation** du routage global.

Une fois l'arbre d'interconnexion construit par l'algorithme de Dijkstra, nous le matérialisons, c'est-à-dire que nous transformons chaque segment de l'arbre (tel que défini page 25 du chapitre 1) en un segment physique de métal occupant certaines ressources de routage. Pour cela nous associons à chaque segment de l'arbre deux vias (un à chaque extrémité) et une couche de métal correspondant à la direction du segment (horizontale ou verticale).

Nous ne considérons que les couches de métal spéciales notées `GHLayer` (GlobalHorizontalLayer) et `GVLayer` (GlobalVerticalLayer) pour les segments et la couche `GVia` pour les vias.

L'association entre une extrémité d'un segment de l'arbre et un via est très simple puisque par définition, les extrémités d'un segment d'un arbre sont associées soit à un coude soit à une descente sur un connecteur (voire les deux).

La mise à jour de l'occupation réelle des arêtes du graphe de routage se fait automatiquement lors de la création ou de la suppression d'un segment. Chaque net, une fois son arbre d'interconnexion créé par l'algorithme de Dijkstra, est matérialisé de façon à ce que le routeur global prenne en compte l'occupation des arêtes associées pour la construction des arbres d'interconnexion des nets suivants.

La figure 3.16 présente un exemple de matérialisation d'un arbre d'interconnexion.

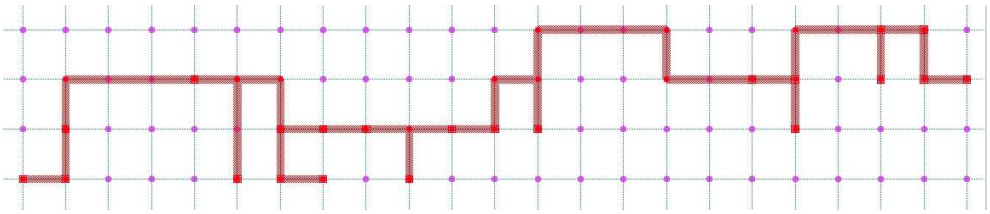
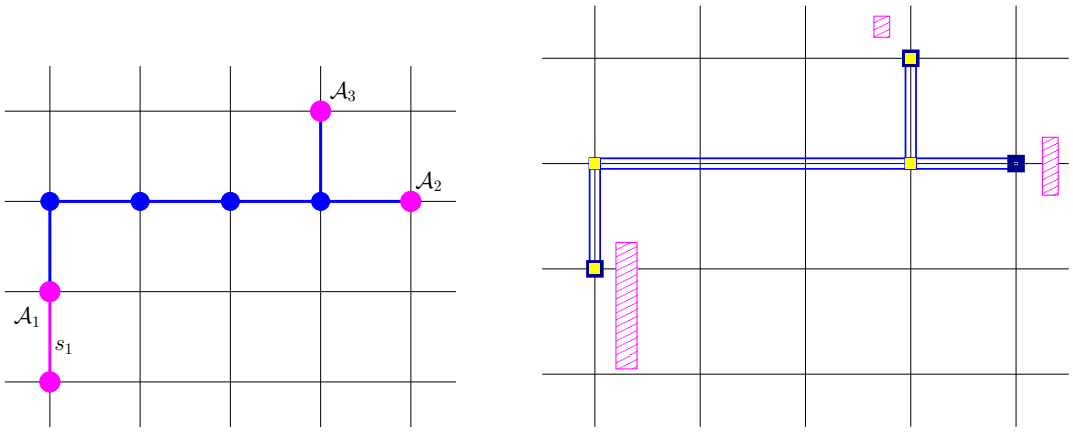


FIGURE 3.16 – Exemple de matérialisation d'un arbre d'interconnexion

Dans le cas d'un connecteur réparti, il ne faut pas matérialiser toutes les arêtes de l'arbre construit par l'algorithme de routage global. En effet les arêtes appartenant à l'origine à la composante connexe représentant le connecteur réparti ne doivent pas être associée à un segment et voir leur occupation augmentée, puisque l'interconnexion physique existe déjà grâce au connecteur.

Par exemple, sur la figure 3.17, le segment s_1 de la composante $\mathcal{A}_1$ n'a aucune correspondance dans la matérialisation.



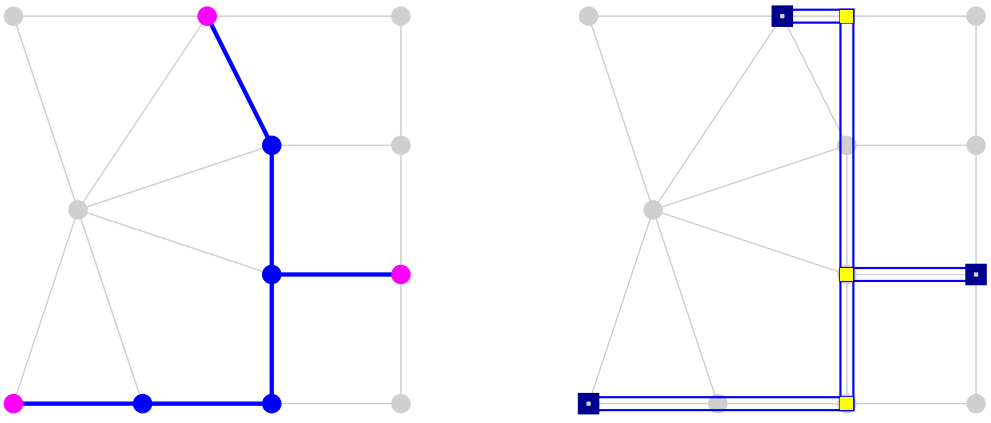
(a) Arbre d'interconnexion

(b) Vue physique (matérialisation et connecteurs)

FIGURE 3.17 – Non matérialisation des arêtes appartenant à une composante connexe représentant un connecteur réparti

De la même façon si le net contient un connecteur réparti de macro blocs, alors les arêtes appartenant au graphe dédié ne sont pas matérialisées.

Notons enfin que dans le cas d'un graphe irrégulier, lorsque l'on rencontre une arrête reliant deux sommets non alignés (horizontalement ou verticalement), nous la matérialisons à l'aide d'un coude, c'est-à-dire un segment horizontal et un segment vertical ainsi qu'un via pour les interconnecter (voir figure 3.18).



(a) Arbre d'interconnexion

(b) Matérialisation

FIGURE 3.18 – Matérialisation sur un graphe de routage irrégulier

3.1.5 Fonctionnalités graphiques

L'outil **KNIK** est intégré à la plate-forme **CORIOLIS** et profite de ce fait de tout l'environnement graphique existant. Ainsi, il est possible à tout moment de stopper le déroulement de l'algorithme de routage global et d'obtenir une représentation graphique de la structure mémoire correspondant au graphe de routage.

La représentation graphique du graphe de routage est utile pour vérifier le comportement des algorithmes. Par exemple, elle permet de visualiser les composantes connexes d'un net après initialisation, puis de suivre graphiquement l'évolution d'un net au fur et à mesure du routage global. La figure 3.19 illustre la représentation du graphe de routage (les sommets sont en magenta et les arêtes en pointillés gris) ainsi que les composantes connexes initiales d'un net (représentées par les rectangles rouges).

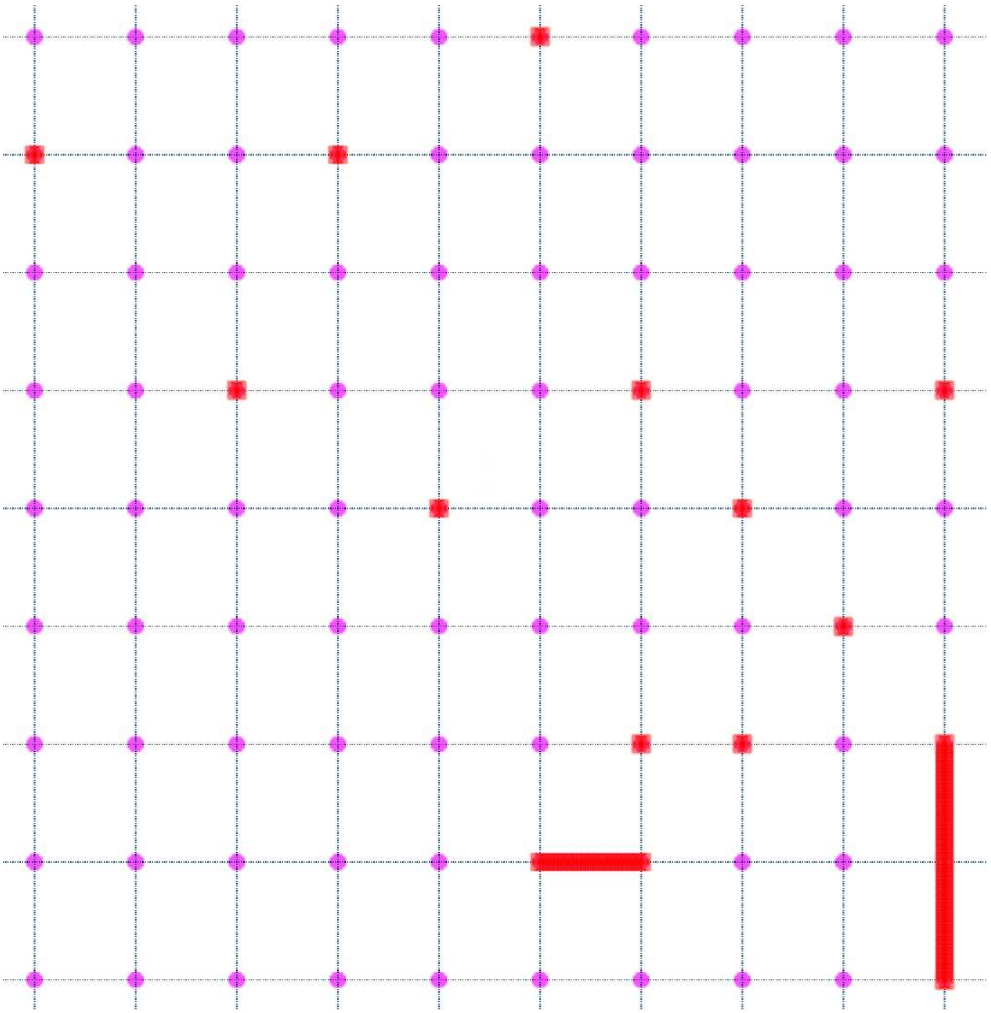


FIGURE 3.19 – Représentation graphique des composantes connexes d'un net

L'inspecteur inclu dans la plate-forme **CORIOLIS** permet de s'assurer de la valeur des attributs des objets de la base de données. Par exemple on peut facilement consulter la capacité d'une arête ou même son occupation à n'importe quel moment de l'algorithme (voir figure 3.20), ou encore de vérifier le bon chaînage des arêtes du graphe (grâce aux pointeurs `nextTo` et `nextFrom`).

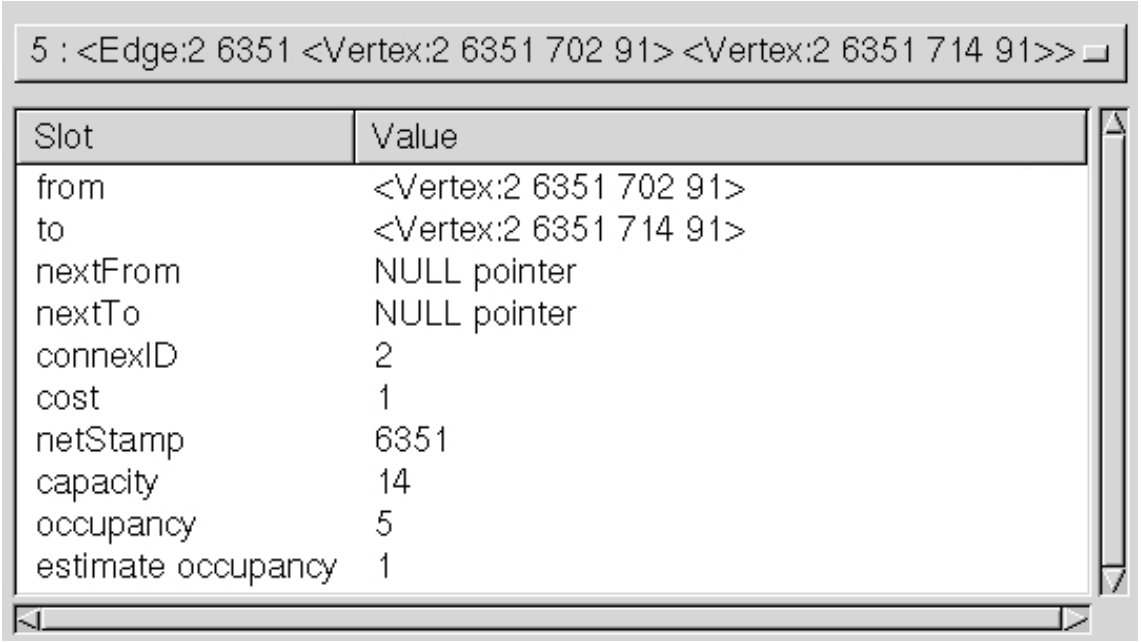


FIGURE 3.20 – Visualisation des attributs d'une arête du graphe de routage.

Les attributs `netStamp` et `connexId` de l'objet `edge` sont présentés dans la section suivante.

3.2 Gestion des nets

La composante connexe est le point clé de la représentation des nets dans l'outil **KNIK**. Tout au long de l'algorithme de Dijkstra, un net n'est représenté que par ses composantes connexes.

Dans cette section nous présentons la mise en œuvre des composantes connexes dans **KNIK** qui utilise les notions de `netStamp` et `connexId` permettant une gestion simplifiée des composantes connexes. Puis nous illustrons l'utilisation des composantes connexes à l'aide des fonctions d'initialisation des composantes connexes, de propagation du coût des sommets et enfin de fusion de deux composantes.

3.2 Gestion des nets

3.2.1 Mise en œuvre des composantes connexes

Une composante connexe est un arbre et de ce fait nous pouvons la représenter sur le graphe de routage en « marquant » chaque arête et sommet qui la composent.

Les algorithmes de routage implémentés dans le cadre de notre outil traitent les nets de manière séquentielle. A chaque instant, il n'y a donc qu'un seul net en cours de traitement sur le graphe de routage.

Nous définissons sur le graphe de routage un identificateur, noté `netStamp`, permettant d'identifier le net en cours de traitement. Le premier net traité est associé au `netStamp` 0, puis le `netStamp` est incrémenté à chaque nouveau net traité. La valeur du `netStamp` varie donc dans $\{0, \ldots, n-1\}$, où n est le nombre de nets à router du circuit.

De plus, nous ajoutons à chaque `vertex` et `edge` du graphe un `netStamp` dont la valeur définit la validité de l'objet pour le traitement en cours.

Si le `netStamp` de l'objet (`vertex` ou `edge`) est égal à celui du net en cours de traitement, l'objet est dit valide, c'est-à-dire que tous ses attributs peuvent être pris en compte par les algorithmes sans risquer de confusion avec les itérations précédentes. Sinon, le `netStamp` de l'objet est inférieur au `netStamp` du graphe, signifiant que l'objet a été modifié lors du traitement d'un net précédent et non réinitialisé. Le `netStamp` des objets peut aussi prendre la valeur spéciale -1 signifiant que l'objet n'a jamais été parcouru depuis la création du graphe de routage.

Connaissant le net à traiter, les algorithmes ont alors besoin de savoir si un sommet (ou une arête) fixé du graphe appartient à l'une des composantes connexes de ce net. Pour cela nous associons à chaque `vertex` et `edge` du graphe un autre identificateur, noté `connexId`, permettant d'identifier les différentes composantes connexes d'un net.

Pour le net en cours de traitement, la valeur du `connexId` varie dans $\{0, \ldots, k_{\texttt{netStamp}}\}$ où $k_{\texttt{netStamp}}$ est le nombre de composantes connexes initiales du net. De plus, si un objet n'appartient à aucune composante connexe, son `connexId` a une valeur spéciale égale à -1.

Cette méthode d'enrichissement du graphe utilisant le `connexId` et le `netStamp` est très légère en mémoire. De plus, elle est très simple d'utilisation puisqu'il suffit aux algorithmes de connaître un `vertex` représentant la composante connexe pour la parcourir. Partant de ce `vertex`, on parcourt récursivement les arêtes de même `connexId` pour atteindre les autres sommets de la composante connexe.

Un autre point fort de cette méthode est qu'elle permet de passer outre la réinitialisation du graphe de routage à chaque nouveau net traité. En effet, lorsque l'algorithme atteint un objet du graphe, il commence par vérifier que le `netStamp` de celui-ci correspond bien au `netStamp` du net actuellement traité. Si ce n'est pas le cas, la valeur actuelle du `connexId` de l'objet est ignorée et modifiée en fonction du traitement en cours. Le `netStamp` est lui aussi mis à jour pour correspondre à celui du net traité.

3.2.2 Manipulation des composantes connexes

Initialisation des composantes connexes d'un net

En parcourant les connecteurs à relier d'un net, on initialise les composantes connexes grâce aux règles énoncées page 10 du chapitre 1. « Initialiser une composante connexe » consiste à reporter la valeur du `netStamp` du net actuellement traité et le `connexId` de la composante connexe à tous les `vertex` et les `edge` correspondant à cette composante.

Pour illustrer cette initialisation, considérons l'exemple de la figure 3.21. Le net à traiter est composé de deux connecteurs à relier c_1 et c_2 et son `netStamp` vaut 9. Nous ne considérons que la portion du graphe de routage correspondant à la boîte englobante des connecteurs.

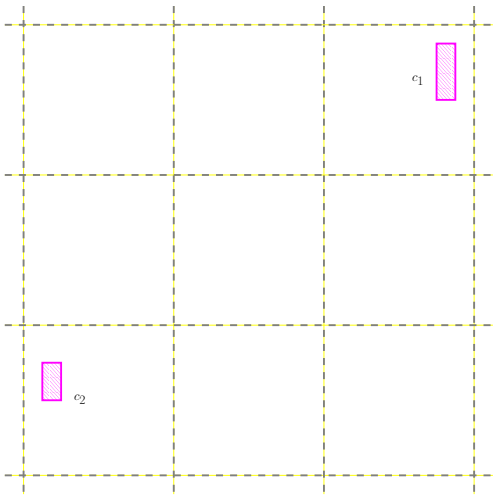
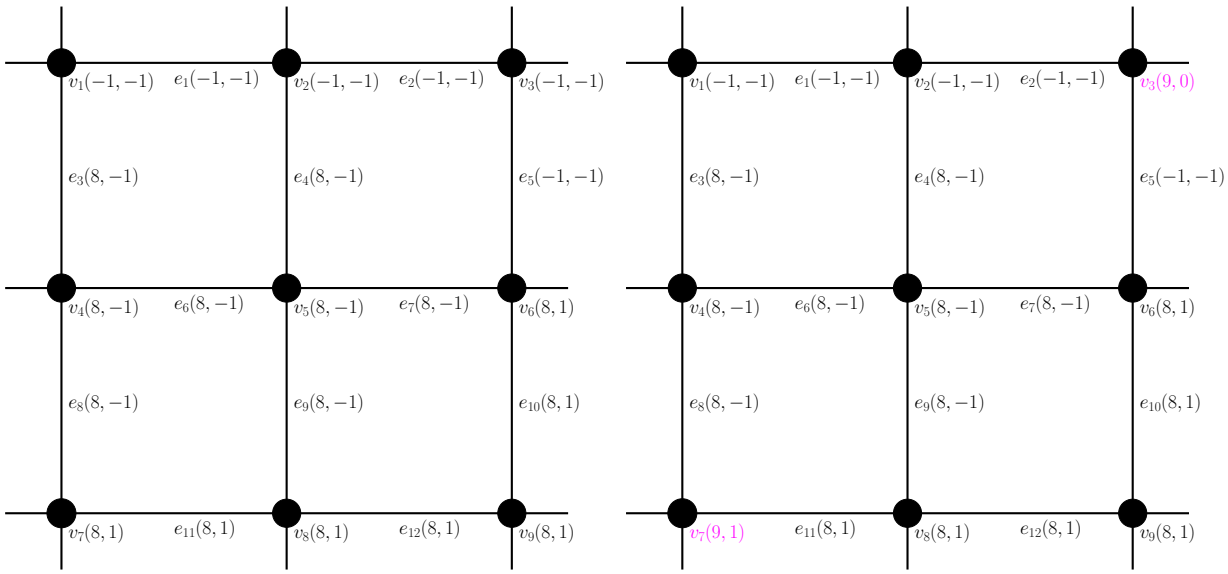


FIGURE 3.21 – Connecteurs à relier du net

Sur la figure 3.22(a), nous avons représenté les valeurs (`netStamp`,`connexId`) correspondant aux nets précédemment routés. On constate que certains sommets et arêtes ont été parcourus par l'algorithme de Dijkstra lors du traitement du net précédent (`netStamp` $= 8$) et qu'une portion de son arbre d'interconnexion est visible (v_7, e_{11}, v_8, e_{12}, v_9, e_{10} et v_6).



(a) Etat du graphe de routage avant initialisation

(b) Etat du graphe de routage après initialisation

FIGURE 3.22 – Initialisation des composantes connexes d'un net

Après initialisation des composantes connexes du net courant (`netStamp` $= 9$), nous obtenons deux composantes connexes ponctuelles. La première est composée du sommet $v_3(9,0)$ et représente le connecteur c_1. La deuxième est composée du sommet $v_7(9,1)$ et représente le connecteur c_2.

Propagation du coût des sommets

Dans la propagation du coût des sommets de l'algorithme de Dijkstra, un point important consiste à être capable de savoir si le sommet atteint appartient ou non à une composante connexe destination du net actuellement traité.

En pratique cette vérification est très simple. On commence par comparer le `netStamp` du `vertex` atteint à celui du net actuellement traité. S'ils sont différents, le `vertex` atteint ne peut appartenir à une composante connexe du net du fait de la façon dont les composantes ont été initialisées. S'ils sont égaux et que le `connexId` du sommet atteint est différent de -1 et différent du `connexId` de la composante connexe source, alors l'algorithme a atteint une composante connexe destination.

Pour chaque `vertex` exploré par l'algorithme de Dijkstra, le `netStamp` est mis à jour à la valeur du `netStamp` du net en cours de traitement. L'algorithme est capable à tout moment de savoir si le coût par rapport à la composante source d'un sommet est valable pour le traitement en cours. De cette façon on peut passer outre une réinitialisation du coût des `vertex` pour chaque nouveau net traité. Notons que

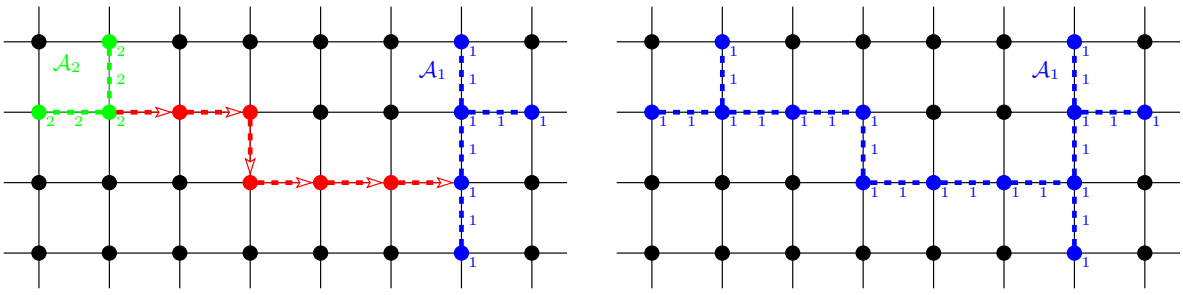
lors de la propagation du coût des sommets, le `connexId` des `vertex` ou des `edge` n'est jamais modifié.

Fusion de deux composantes connexes

Notre mise en œuvre de l'algorithme de Dijkstra raisonne en composantes connexes. A partir d'une composante connexe source, il construit une chaîne d'interconnexion vers une composante destination, puis « fusionne » les deux composantes. La composante connexe résultante devient alors la nouvelle composante source.

La fusion de deux composantes connexes interconnectées par une chaîne est assez simple. On commence par modifier le `connexId` de la composante connexe destination pour qu'il soit égal à celui de la composante connexe source. Puis à partir du sommet de la composante connexe destination atteint par l'algorithme de Dijkstra, on remonte jusqu'à la composante connexe source en suivant les pointeurs `predecessor`. Lors de cette étape, les `connexId` de tous les `vertex` et `edge` parcourus sont modifiés pour correspondre à celui de la composante connexe source. Le `netStamp` des objets n'est pas modifié.

La figure 3.23 illustre un exemple de fusion de deux composantes. Nous considérons les deux composantes $\mathcal{A}_1$ et $\mathcal{A}_2$ (en bleu et vert sur la figure 3.23(a)) de `connexId` respectifs 1 et 2. Nous avons représenté en rouge la chaîne les interconnectant et la valeur du `connexId` pour les `vertex` et `edge` des composantes connexes. La figure 3.23(b) présente la composante connexe résultant de la fusion. Le `connexId` de cette nouvelle composante est égal à celui de la précédente composante connexe source.



(a) Les deux composantes connexes interconnectées

(b) La nouvelle composante connexe source correspondante

FIGURE 3.23 – Exemple de fusion de deux composantes connexes

3.3 Mise en œuvre de l'algorithme de Dijkstra

Dans notre mise en œuvre de l'algorithme de Dijkstra, nous utilisons les composantes connexes telles que nous venons de les définir. Nous utilisons un algorithme de Dijkstra multi composantes (uni-source et multi-destinations) tel que décrit dans la section 2.3.5 du chapitre 2.

Dans cette section, nous détaillons les différents points importants de notre mise en œuvre de l'algorithme de Dijkstra. Nous présentons la phase d'initialisation et la fonction de coût de la congestion. Puis nous détaillons la structure de file de priorité utilisée par notre algorithme. Enfin nous illustrons la flexibilité de notre mise en œuvre par l'étude du coût des vias.

3.3.1 Initialisation de l'algorithme de Dijkstra

Traiter toutes les composantes connexes du net simultanément permet de passer outre la phase de décomposition en bipoints du net, puisque l'algorithme ne considère plus les composantes par paire. Le seul choix à faire est celui de la composante connexe source.

Dans l'outil **KNIK**, nous avons choisi de définir comme composante connexe source d'un net celle dont le sommet représentant est le plus au centre de la boîte englobant toutes les composantes connexes du net.
En cas d'égalité nous choisissons arbitrairement le représentant dont la coordonnée x est la plus faible (ou y en cas d'égalité des coordonnées x). Ce choix est valable puisque par construction il est impossible que deux composantes connexes aient le même sommet représentant.

En choisissant la composante la plus au centre, nous réduisons sensiblement le nombre de sommets explorés par l'algorithme de Dijkstra, réduisant ainsi le temps d'exécution.

3.3.2 Fonction de coût de la congestion

La fonction de coût utilisée dans l'outil **KNIK** est telle que celle définie dans le chapitre 2. Pour le coût lié à la congestion, nous utilisons une fonction « lissée » comme définie page 54 :

$$coutCongestion(a) = \frac{h}{1 + e^{-k \times (tauxCongestion(a) - 1)}}$$

Les paramètres h et k ont pour valeurs respectives 10 et 30. Le taux de congestion est calculé à la volée en fonction des attributs `realOccupancy`, `estimatedOccupancy`

et `capacity` d'une arête :

$$tauxCongestion(a) = \frac{\texttt{realOccupancy} + \texttt{estimatedOccupancy}}{\texttt{capacity}}$$

Rappelons que dans le cas d'une gestion dynamique de la congestion, la participation de l'occupation estimée décroît au bénéfice de l'occupation réelle au fur et à mesure du traitement.

3.3.3 File de priorité

Un point clé d'une mise en œuvre efficace de l'algorithme de Dijkstra réside dans la structure mémoire utilisée pour représenter le bord de la vague. Cette structure doit fournir les fonctions suivantes :

- `insert(vertex,cost)` qui permet d'insérer un nouveau sommet dans le bord de la vague, avec un coût fixé,
- `extractMin()` qui permet de récupérer le sommet de coût minimum et de le supprimer du bord de la vague,
- `decreaseCost(vertex,newCost)` qui permet de modifier le coût d'un sommet déjà présent dans le bord de la vague. Notons que par définition de l'algorihme de Dijkstra (voir page 66 du chapitre 2), ce coût ne peut être que réduit.

Traditionnellement pour l'algorithme de Dijkstra, on utilise une structure de **file de priorité** [Quea] qui, si l'on considère que le sommet ayant le plus petit coût est le plus prioritaire, offre toutes les fonctionnalités nécessaires.

Non seulement les fonctions `insert` et `extractMin` doivent être performantes, mais aussi la fonction `decreaseKey`. En effet, la forte hétérogénéité des coûts des arêtes induit, pour un sommet donné, de nombreuses reconvergences par des chemins plus courts, entraînant des appels fréquents à cette fonction.

Parmi les différentes mises en œuvre de la file de priorité, certaines sont plus particulièrement adaptées à l'algorithme de Dijkstra [Queb]. Notons qu'une des meilleures mises en œuvre est celle utilisant un tas de Fibonacci [hea].

Notre mise en œuvre, plus simple, est basée sur un *set* de la *STL* [sS], qui est en fait un arbre binaire équilibré. Ce *set* est trié par ordre décroissant de priorité des sommets (et un critère secondaire de coordonnées des sommets en cas d'égalité). L'insertion d'un nouveau sommet et l'extraction du plus prioritaire (premier élément du *set*) se font avec une complexité $O(\log n)$ (où n représente le nombre de sommets dans la vague).

Pour la mise en œuvre de la fonction `decreaseKey`, on associe à chaque sommet présent dans la vague sa place dans la file, ce qui permet de l'extraire et de le

réinsérer avec une complexité $O(\log n)$.

3.3.4 Flexibilité de la mise en œuvre

Tout au long du développement de l'outil **KNIK**, nous avons veillé à ce qu'il reste modulaire, c'est-à-dire qu'en changeant la valeur d'un paramètre ou en modifiant une fonction simple, il soit possible de changer le comportement général de l'algorithme.

Parmi les éléments modulables, citons notamment :
- les paramètres de la fonction de coût de congestion des arètes : h et k,
- la gestion de l'estimation anticipée de la congestion (statique ou dynamique),
- la valeur du coût des vias,
- la fonction de choix de la composante connexe source.

Comme nous l'avons vu dans le chapitre 2, le coût des vias a été fixé à une valeur de 3 pour les circuits de tests du l'ISPD'07 [oPDGRCa]. Puisque l'outil **KNIK** offre la possibilité de faire varier le coût des vias nous avons voulu vérifier quel était son impact sur la solution de routage global.

Pour des valeurs de coût des vias variant de 1 à 4 nous avons relevé[3] pour chacun des circuits de tests de l'ISPD'98 [ben], la longueur totale des fils d'interconnexion (exprimée en unité normalisée de pas de grille de routage), le nombre de vias correspondant à un coude ou une descente sur un connecteur et enfin le dépassement total du circuit. Pour bien voir l'impact du coût des vias sur ces trois critères, nous relevons les valeurs avant la phase de *ripup & reroute*.

La figure 3.24 présente la variation du nombre de vias en fonction de leur coût et confirme que le nombre de vias est inversement proportionnel à leur coût.

3. Le tableau des relevés est donné en annexe page 161.

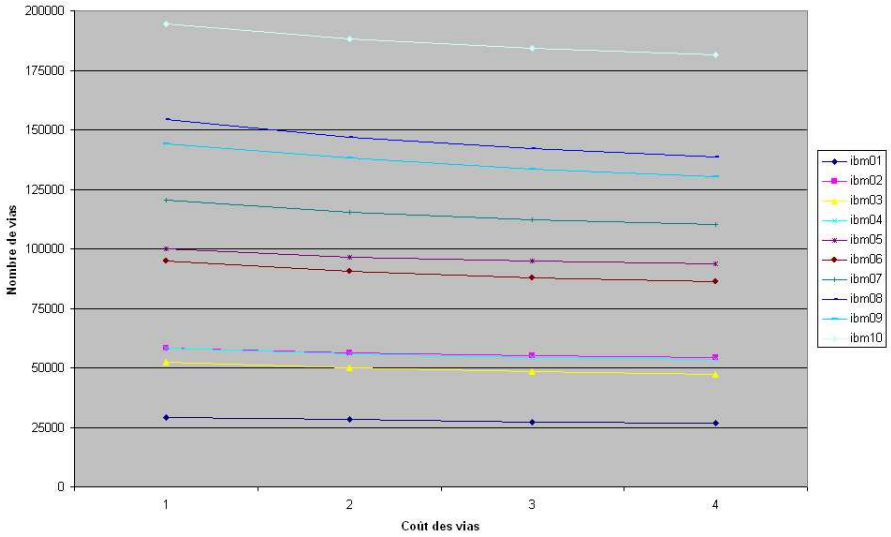


FIGURE 3.24 – Variation du nombre de vias en fonction du coût d'un via

Bien sûr diminuer le nombre de vias tend à générer des arbres d'interconnexion plus directs et de ce fait plus courts, comme l'illustre la figure 3.25.

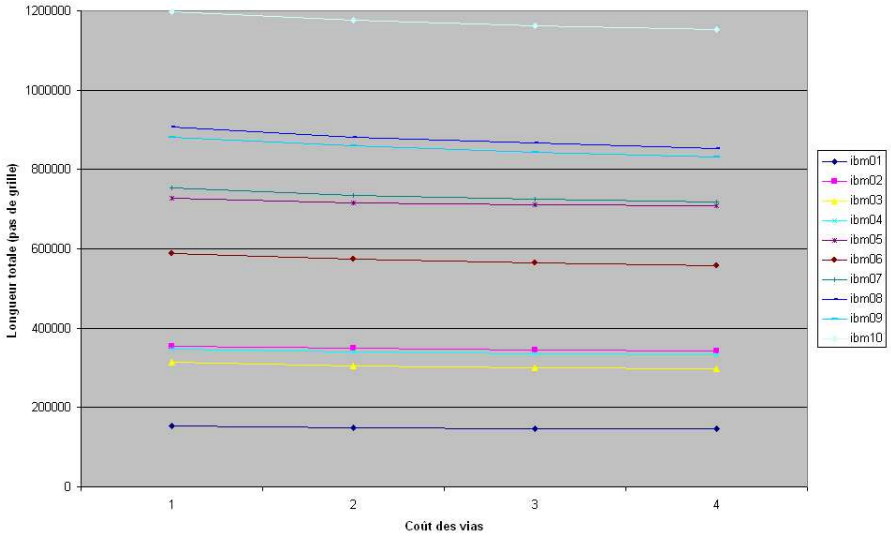


FIGURE 3.25 – Variation de la longueur totale des segments en fonction du coût d'un via

Mais comme nous l'avons montré précédemment, la diminution de la longueur totale des interconnexions et du nombre de vias entraîne une augmentation de la congestion. La figure 3.26 illustre bien ce phénomène [4]. L'augmentation du coût des vias fait

4. Sur cette figure, le circuit *ibm05* n'apparaît pas car le dépassement total est nul sauf pour un coût de via égal à 4.

diminuer le dépassement total tant que le coût est inférieur ou égal à 3. Au delà, la congestion du circuit augmente.

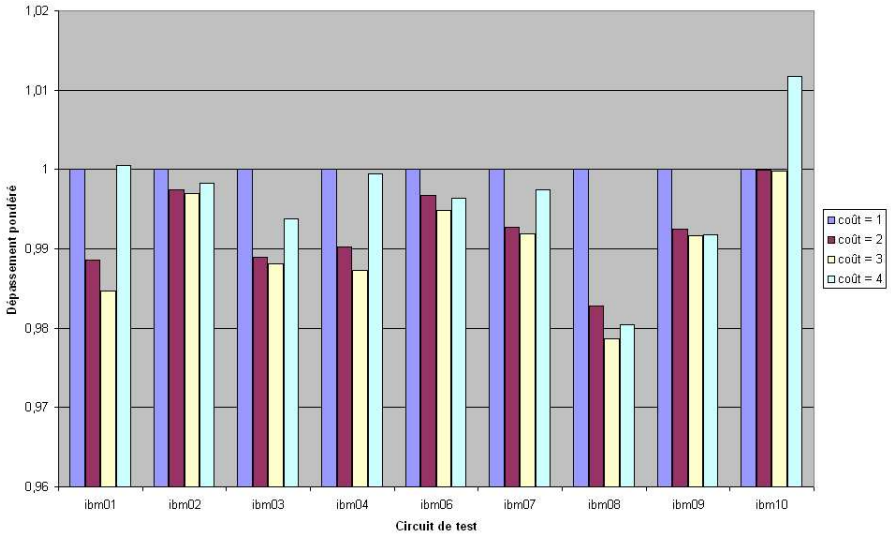


FIGURE 3.26 – Variation du dépassement total en fonction du coût d'un via

Etant donné que la prise en compte des vias a pour but de réduire la congestion du circuit, nous retenons la valeur 3 pour le coût des vias.

3.4 Mise en œuvre du *ripup & reroute*

Dans la très grande majorité des cas, la solution initiale construite en utilisant l'algorithme de Dijkstra n'est pas valide, c'est-à-dire que certaines arêtes ont un dépassement non nul. Pour tenter d'éliminer cette sur-congestion, il est commun d'utiliser une méthode de *ripup & reroute*.

A la fin du chapitre 1 nous avons défini plusieurs scénarios de *ripup & reroute* dont notamment un utilisant les possibilités offertes par les composantes connexes. En segmentant le net en plusieurs morceaux, il est possible de ne rerouter que certaines portions. L'approche proposée consiste à identifier les portions de net contribuant le plus à la congestion puis à les rerouter.

Notre mise en œuvre du *ripup & reroute* utilise un tel scénario. Nous utilisons les fonctionnalités du graphe de routage pour obtenir la liste des segments de routage global contribuant aux sur-congestions. Nous les déroutons puis nous réutilisons l'algorithme de Dijkstra pour construire une nouvelle solution.

Nous présentons ici la fonction de coût utilisée pour identifier les segments par-

ticipant à la congestion, la métode pour dérouter un segment et enfin les changements appliqués à l'algorithme de Dijkstra pour fonctionner en mode *ripup & reroute*.

3.4.1 Identification des portions de nets à rerouter

Pour identifier les portions de net à rerouter, nous calculons un coût de contribution à la congestion pour chaque segment de la matérialisation du routage global. Ce coût est basé sur le dépassement des arêtes recouvertes par le segment considéré. Les segments ayant le coût le plus élevé sont ceux contribuant le plus à la congestion du circuit.

L'idée la plus simple pour calculer ce coût consiste à faire la somme des dépassements des arêtes recouvertes par le segment :

$$cout(seg) = \sum_{a \in seg} dep(a) \tag{3.1}$$

De cette façon, les segments recouvrant plusieurs arêtes ayant un fort dépassement seront reroutés en premier, ce qui tend à faire rapidement décroître le dépassement total du circuit. Ce coût (3.1) n'est pourtant pas suffisamment précis, au sens où dans certains cas deux segments ont le même coût alors que visuellement il est facile de dire lequel devrait être rerouté en premier.

La figure 3.27 illustre ce problème. Nous considérons les arbres d'interconnexion de trois nets (les connecteurs sont représentés par les carrés pleins). Les arêtes du graphe de routage ayant une capacité égale à 2, les trois segments s_1, s_2 et s_3 ont un coût de contribution à la congestion (3.1) égal à 1.
On constate que l'arbre contenant le segment s_3 peut être facilement remplacé pour construire une solution sans dépassement.

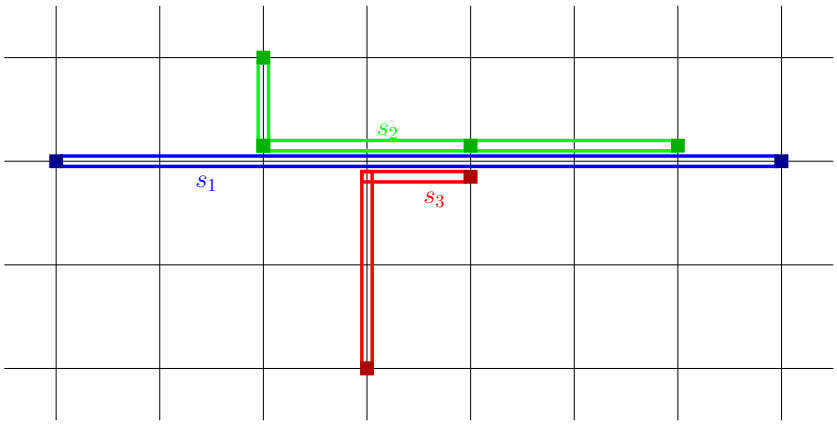


FIGURE 3.27 – Calcul du coût d'un segment

Pour prendre en compte de tels cas, nous considérons deux autres valeurs :
- $NbTot(seg)$: représentant le nombre total d'arêtes recouvertes par le segment,
- $NbDep(seg)$: représentant le nombre d'arêtes recouvertes par le segment et ayant un dépassement non nul.

En comparant ces deux valeurs pour un même segment, on constate que si elles sont égales ou du moins très proches, cela signifie que le segment est entièrement (ou presque) contenu dans une zone sur-congestionnée. Si maintenant $NbTot$ est très supérieur à $NbDep$, le segment traverse une ou plusieurs petites zones sur-congestionnées.

Ainsi, rerouter un segment ayant une somme des dépassements faible et un rapport $\frac{NbDep}{NbTot}$ faible ne réduira que très peu la congestion. A l'inverse, un segment ayant beaucoup de dépassement et un rapport proche de 1 est plus intéressant à rerouter.

En répercutant ces observations sur le coût de contribution à la congestion d'un segment, nous obtenons la nouvelle définition suivante :

$$cout(seg) = \frac{NbDep(seg)}{NbTot(seg)} \times \sum_{a \in seg} dep(a) \tag{3.2}$$

Dans le cas d'un graphe régulier, le calcul de ce coût est très simple puisque l'identification des arêtes recouvertes par un segment est immédiate. Dans le cas d'un graphe irrégulier, nous utilisons la même méthode que celle décrite pour le report des probabilités (voir page 96).

3.4.2 Déroutage d'un segment

Une fois que l'on a identifié les segments à rerouter, il faut les dérouter. Dérouter un segment signifie le supprimer et donc découper l'arbre d'interconnexion auquel il appartient en deux composantes connexes disjointes. La suppression du segment libère des ressources de routage : l'occupation de chaque arête du graphe de routage recouverte par le segment est décrémentée en conséquence.

Une fois le segment supprimé, l'arbre d'interconnexion est découpé en deux composantes connexes disjointes qui doivent être interconnectées. Dans certains cas, il est possible que certaines portions des composantes connexes résiduelles soient inutiles.

Prenons l'exemple de la figure 3.28(a), sur laquelle nous considérons l'arbre d'interconnexion construit pour relier trois connecteurs c_1, c_2 et c_3. Le segment s_3, recouvrant deux arêtes sur-congestionnées, doit être rerouté.

Après suppression du segment s_3, l'arbre est décomposé en deux composantes

connexes résiduelles : $\mathcal{A}_1$ et $\mathcal{A}_2$ (voir figure 3.28(b)). Si nous relançons l'algorithme de routage global sur cet arbre, nous obtenons un nouvel arbre d'interconnexion représenté sur le figure 3.28(c).

Sur ce nouvel arbre, on constate que le segment s'_2 ne sert à rien. En effet par définition, un segment sert à relier deux éléments physiques (connecteurs ou segments), mais l'extrémité haute de s'_2 n'est plus attachée à rien.

De plus il est facile de voir que pour relier les composantes $\mathcal{A}_1$ et $\mathcal{A}_2$ de la figure 3.28(b), il est possible de construire une chaîne plus simple (voir la figure 3.28(d)). Ceci montre que pour la figure 3.28(b), lors de la suppression du segment s_3 nous devrions dérouter aussi le segment s_2.

En pratique, lorsque nous déroutons un segment nous étudions les contacts sur lesquels il repose. Si un contact n'est associé à aucun connecteur et qu'il n'y a qu'un seul segment reposant sur ce contact (hormis celui dérouté), alors ce segment est aussi dérouté. Et lorsqu'un contact n'a plus aucun segment s'appuyant sur lui, il est supprimé.

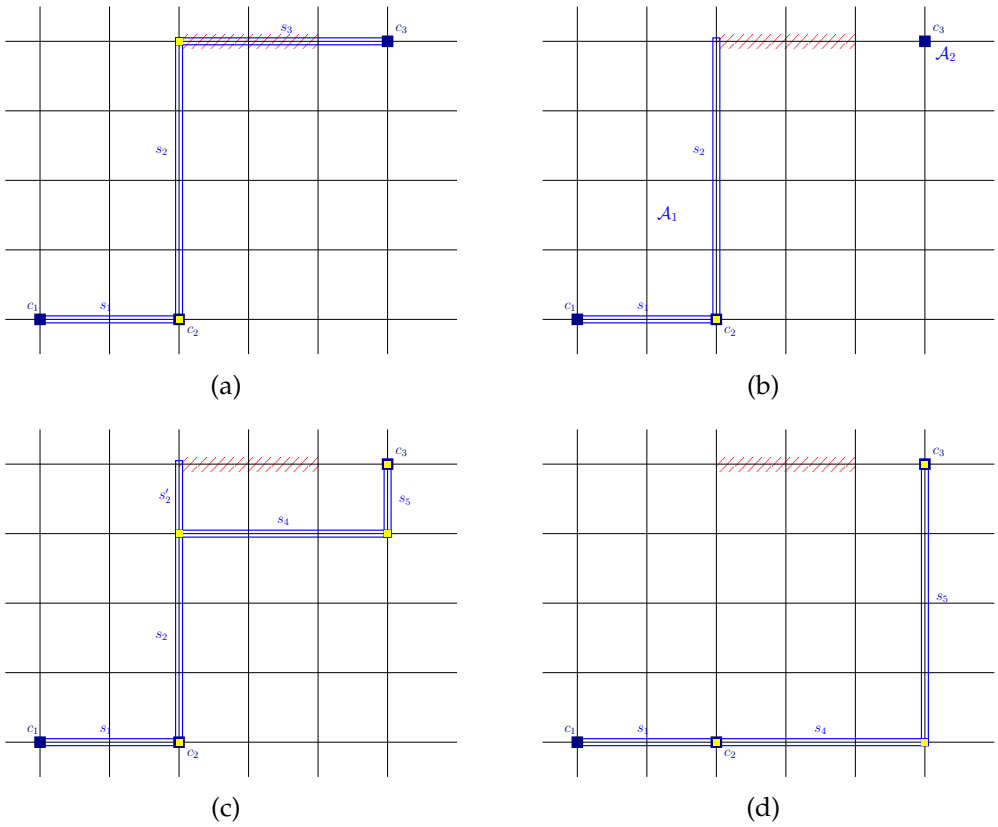


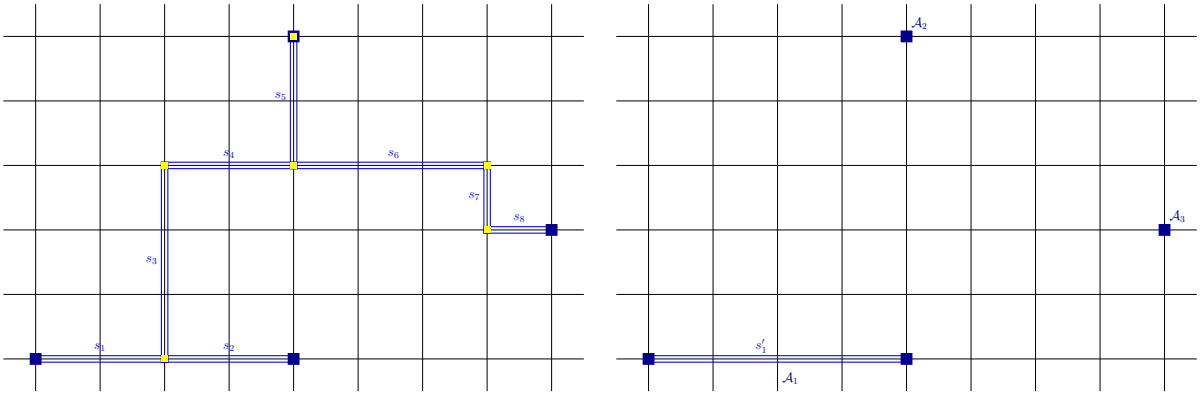
FIGURE 3.28 – Reroutage d'un segment donnant lieu à des portions de composantes connexes inutiles

3.4 Mise en œuvre du *ripup & reroute*

Cette technique permet d'optimiser les arbres d'interconnexion générés après reroutage mais peut dans certains cas entraîner la suppression d'un grand nombre de segments et donc augmenter le temps de calcul du reroutage.

Considérons l'exemple de la figure 3.29(a) et imaginons que nous souhaitons rerouter les segments s_3 et s_7. La suppression de s_3 implique celle de s_4, puis en supprimant s_7 on supprime aussi s_8 et s_6 ainsi que s_5 (puisque s_4 a été supprimé). Au final le reroutage considère les trois composantes connexes illustrées sur la figure 3.29(b).

Notons aussi que nous avons défini une procédure de « nettoyage » pour fusionner deux segments de même direction s'appuyant sur un même contact associé uniquement à ces deux segments, comme s_1 et s_2 sur la figure 3.29(a) devenant s'_1.



(a) (b)

FIGURE 3.29 – Suppressions de segments de proche en proche

Une fois le ou les segments supprimés, il faut réinitialiser les composantes connexes du net considéré à partir des connecteurs et segments existants. Pour cela nous considérons à la fois les connecteurs du net et la matérialisation.

Transformer la matérialisation en composantes connexes est tout aussi simple que la construction de la matérialisation à partir d'un arbre d'interconnexion (voir page 99). Il est important de prendre en compte les connecteurs notamment pour le cas des connecteurs répartis.

Sur la figure 3.30, la matérialisation correspond à l'arbre d'interconnexion après suppression du segment reliant les connecteurs c_3 et c_4. L'autre « trou » dans la matérialisation est dû au connecteur réparti c_2. Après réinitilisation des composantes connexes, on obtient les deux composantes $\mathcal{A}_1$ et $\mathcal{A}_2$ présentées sur la figure 3.30.

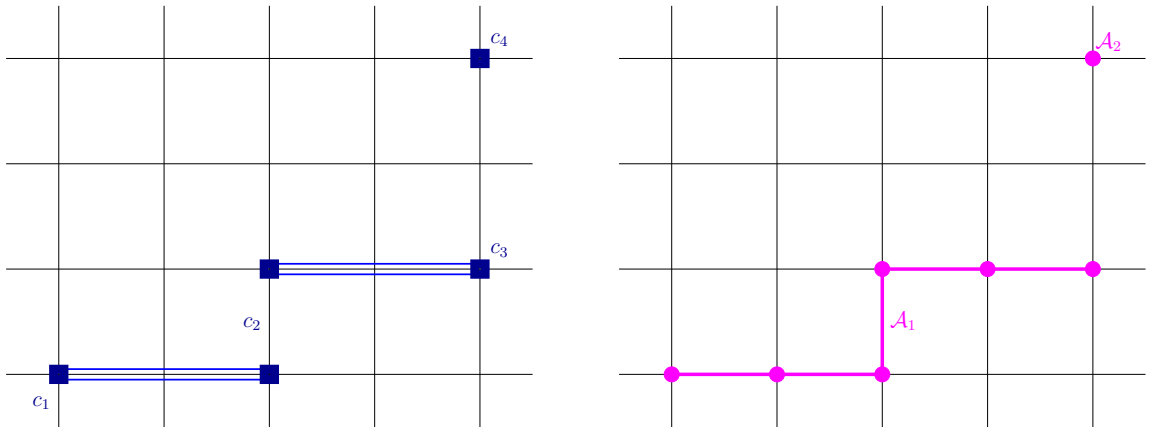


FIGURE 3.30 – Reconstruction des composantes connexes avec présence d'un connecteur réparti

3.4.3 Reroutage des nets

Une fois les segments participant le plus à la congestion déroutés, il suffit de relancer l'algorithme de Dijkstra pour reconstruire les arbres d'interconnexion des nets correspondant aux segments déroutés. L'ordre dans lequel ces nets sont reroutés dépend de leur facteur de forme, exactement comme lors du routage global hors *ripup & reroute*.

Contrairement à la phase de routage précédant le *ripup & reroute*, le nombre de nets à rerouter est assez restreint. De plus, seule la congestion instantanée est considérée. Du point de vue de l'algorithme de Dijkstra, nous ne modifions qu'une seule chose : la taille de la fenêtre d'exploration.

Le but du *ripup & reroute* étant d'éliminer toute la sur-congestion du circuit, si nous limitons la fenêtre d'exploration du Dijkstra à la boîte englobante des composantes connexes d'un net, il est très probable que l'algorithme ne pourra pas construire de solution valide pour certains nets.

Pour la phase de *ripup & reroute*, la fenêtre d'exploration de l'algortihme de Dijkstra est donc étendue à l'ensemble de la surface du circuit, ce qui, dans le cas de très grands circuits, ralentit fortement la vitesse d'exécution de notre algorithme (comme nous le verrons dans le chapitre 4).

Conclusion

Dans ce chapitre nous avons présenté les détails de la mise en œuvre de notre outil de routage global.

Sur la base d'une structure de graphe de routage bidimensionnel régulier ou non, nous avons défini un ensemble de fonctionnalités nécessaires aux algorithmes de routage global. Nous avons introduit les notions de `netStamp` et `connexId` qui permettent de représenter et de manipuler très simplement les composantes connexes sur ce graphe de routage.

Nous avons mis en œuvre un algorithme de Dijkstra traitant simultanément une composante connexe source et une ou plusieurs composantes connexes destinations ainsi qu'une technique de *ripup & reroute* consistant à ne dérouter qu'un certain nombre des segments contribuant le plus à la sur-congestion du circuit. L'évaluation de cette contribution est calculée en prenant en compte le nombre d'arêtes en sur-congestion recouvertes par le segment, le nombre total de ces arêtes et la somme de leur dépassement.

Nous avons donc à notre disposition un outil de routage global comprenant une première méthode simple de *ripup & reroute* que nous allons pouvoir tester et comparer aux autres routeurs globaux académiques existant. Nous verrons dans le chapitre suivant les points faibles de cette approche de *ripup & reroute* et nous présenterons la technique de négociation de la congestion qui permet de corriger ces défaults.

Chapitre 4 Résultats

Dans ce chapitre nous présentons et analysons les résultats de notre routeur global **KNIK**, pour les jeux de circuits de test ispd98 [ben] et ispd07 [oPDGRCa]. Ces deux jeux de circuits sont des références dans le domaine du routage global académique et les résultats des routeurs globaux sont toujours évalués et comparés sur ces circuits.

Dans un premier temps, nous présentons l'environnement d'évaluation que nous avons développé pour que notre outil interagisse correctement avec ces deux jeux de circuits de test.

Nous présentons ensuite une première série de résultats qui met en évidence le fait que notre approche de *ripup & reroute* (présentée dans le chapitre 3) est trop simple et ne suffit pas à éliminer toute la sur-congestion d'un circuit. Nous introduisons alors une technique de négociation de la congestion dans notre outil.

Dans les deux sections suivantes, nous analysons les résultats de notre outil avec la négociation de la congestion pour les circuits de l'ispd98 puis ceux de l'ispd07.

Tous nos résultats sont comparés à ceux du routeur global **FGR** (*Fairly Good Router*) [RM07] développé par Jarrod A. Roy et Igor L. Markov. Ce routeur, distribué librement sur Internet [aFGR], est notre référence puisqu'il s'est placé premier de la catégorie « *2D Global Routing* » lors du concours de routage global de l'ispd07.

De façon à ce que la comparaison ait un sens, tous les tests ont été exécutés dans les mêmes conditions. Nous avons utilisé la version 1.0 de **FGR** disponible sur Internet. L'ordinateur utilisé est équipé d'un processeur Intel Quad Core Q6600 [Spe], de 6 Go de mémoire vive et utilise une distribution linux Ubuntu 8.10 64bits [Edi].

4.1 Environnement d'évaluation

Les circuits de tests de l'ispd98 et l'ispd07 sont fournis au format *Labyrinth* [ifs] et les résultats de routage global doivent respecter le format *BoxRouter* [ofs]. En plus de notre outil de routage global, nous avons développé un outil permettant de charger

ces circuits et de sauvegarder la solution construite dans le bon format.

Notre outil permet en outre de calculer les valeurs utiles à l'analyse des résultats. Ces valeurs sont celles que nous avons définies dans le chapitre 1 : le dépassement total du circuit, la longueur totale des interconnexions et le nombre de vias.

Les valeurs calculées sont exactement les mêmes que celles renvoyées par le script d'évaluation des solutions fournis lors de l'ispd07 et nommé `grc-eval`. L'avantage de notre outil est que le temps nécessaire au chargement et à l'analyse d'un circuit et de sa solution est bien inférieur à celui du script. Par exemple pour le circuit *ibm10*, l'analyse avec notre outil ne nécessite que $16,21$ secondes contre $93,01$ secondes avec le script.

Grâce aux facilités offertes par la plateforme **CORIOLIS**, il est possible de visualiser le routage global dans son ensemble mais aussi pour un net particulier (voir figure 4.1).

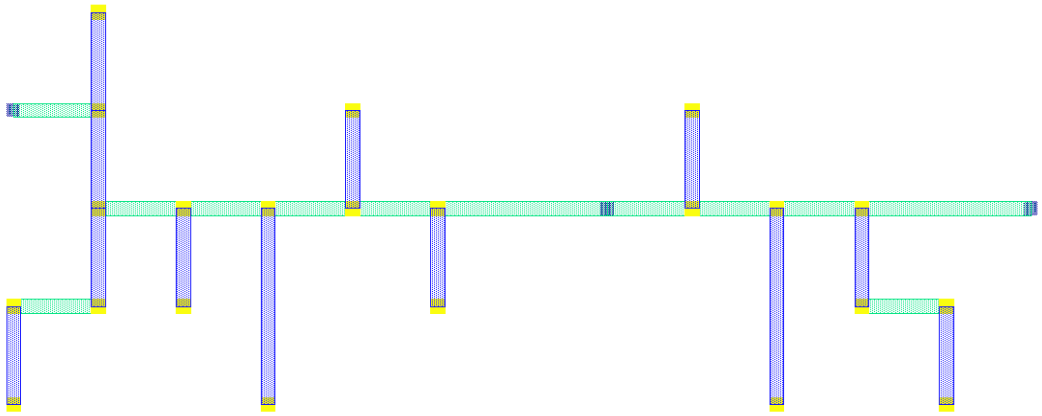


FIGURE 4.1 – Affichage du net `net8219` du circuit *ibm01* après routage global effectué par **KNIK**

Notre outil d'analyse permet aussi de générer des cartes de congestion. Une carte de congestion est une représentation graphique du taux de congestion des arêtes du graphe de routage. Un exemple de carte de congestion ainsi que l'échelle de couleur utilisée sont illustrés sur la figure 4.2. Plus une arête est claire et plus son taux de congestion est proche de 1, c'est-à-dire plus elle est congestionnée. L'échelle de couleur utilisée est une échelle de température et par analogie nous disons que les arêtes les plus congestionnées représentent les points chauds du circuit. Les arêtes sur congestionnées sont identifiées à l'aide d'un contour de couleur turquoise.

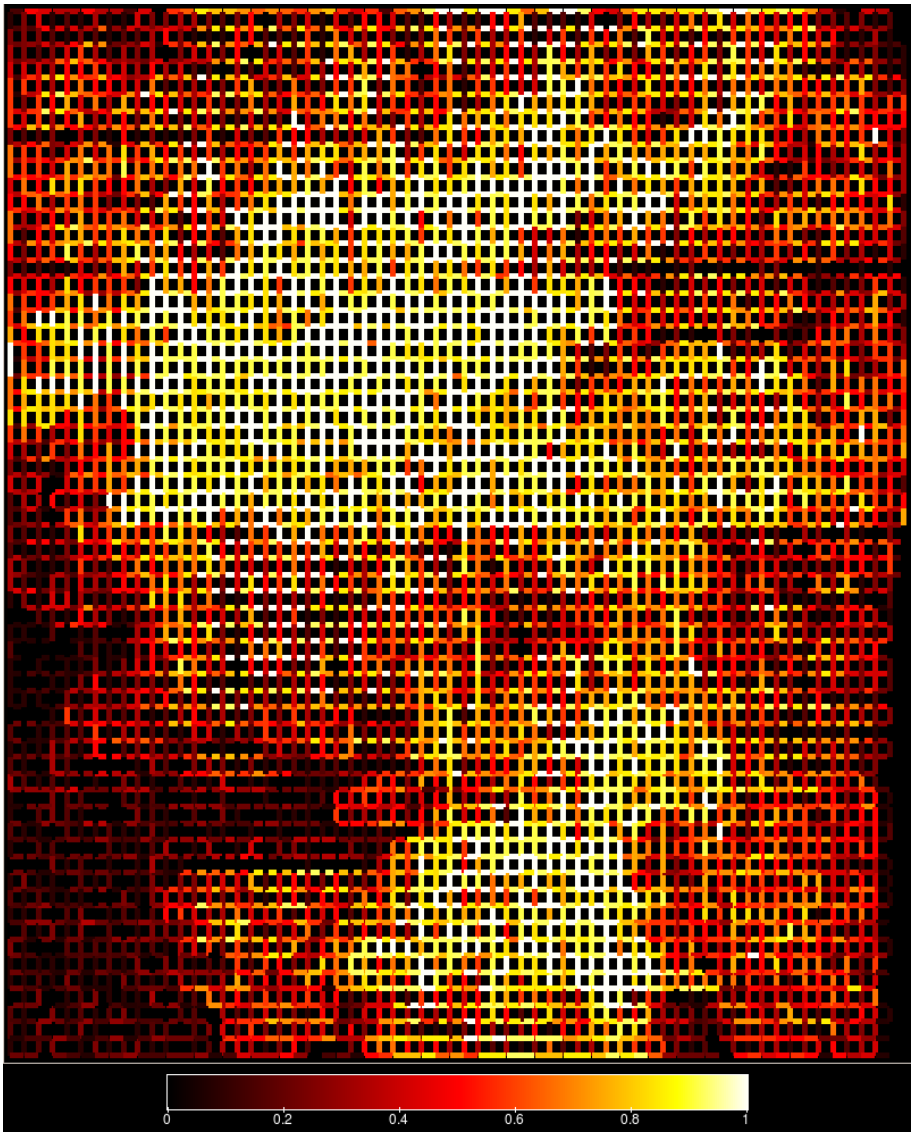


FIGURE 4.2 – Carte de congestion du circuit *ibm01* après routage global effectué par **KNIK**

Notons enfin que les solutions de routage d'autres routeurs globaux peuvent être lues et analysées par notre outil à condition qu'elles respectent le format *BoxRouter*.

4.2 Etude du *ripup & reroute* simple

Dans cette section, nous présentons une première série de résultats qui met en évidence le fait que notre approche de *ripup & reroute* est trop simple et ne suffit pas à éliminer toute la sur-congestion d'un circuit. Nous introduisons alors une technique de négociation de la congestion dans notre outil.

4.2.1 Premiers résultats

Dans un premier temps nous avons étudié les résultats d'un algorithme de *ripup & reroute* simple, qui à chaque itération déroute l'ensemble des segments participant le plus à la sur-congestion puis les reroute à l'aide de l'algorithme de Dijkstra.

Pour cela nous avons utilisé l'outil **KNIK** pour construire une première solution de routage global et l'améliorer grâce au *ripup & reroute*. Le nombre d'itérations de *ripup & reroute* a été limité à 20. Nous avons relevé à chaque itération le dépassement total des arêtes du graphe de routage.

Sur la figure 4.3 nous avons représenté l'évolution du dépassement total au fur et à mesure des itérations pour chacun des circuits de l'ispd98 et l'ispd07[1]. Comme les dépassements totaux ne sont pas du même ordre de grandeur pour chaque circuit, nous les avons normalisés en les divisant par la valeur relevée après routage global (itération 0).

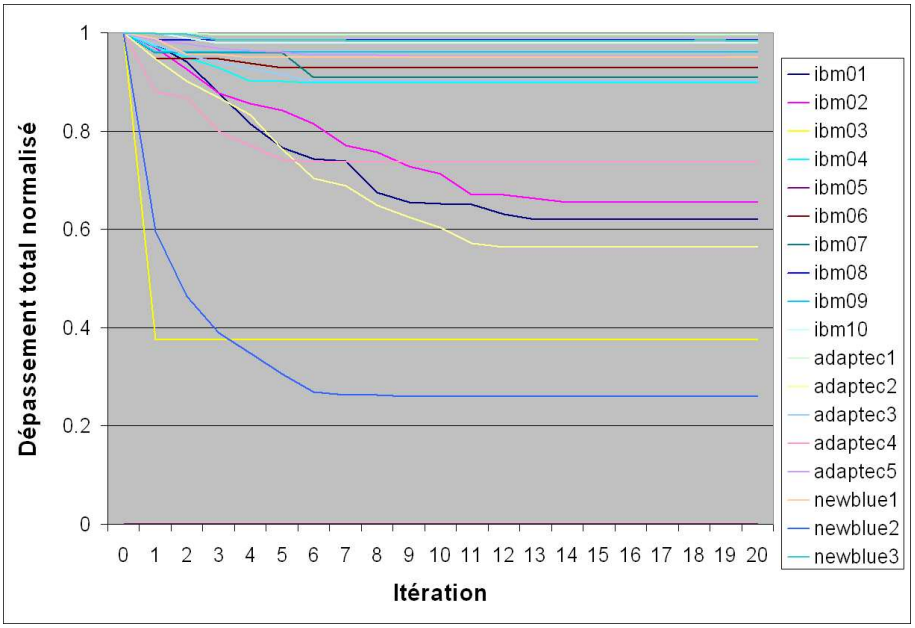


FIGURE 4.3 – Evolution des dépassements totaux pour les circuits de l'ispd98 et l'ispd07

On constate facilement que l'algorithme de *ripup & reroute* proposé ne suffit pas pour supprimer toute sur-congestion d'un circuit. En effet à partir d'un certain nombre d'itérations, l'algorithme ne parvient plus à réduire le dépassement total.

Plus exactement, d'une itération à l'autre, notre outil peut sélectionner le même ensemble de segments à rerouter (ayant les coûts les plus grands) car l'algorithme de

1. Les relevés complets sont fournis en annexe page 163

Dijkstra reroute à l'identique les segments. En effet pour des segments noyés dans une zone de sur-congestion, l'algorithme de Dijkstra ne parvient pas à trouver une chaîne d'interconnexion de coût inférieur à celle passant pas les zones sur-congestionnées.

Si l'on considère le segment de la figure 4.4[2], on constate bien que l'algorithme de Dijkstra en considérant le coût des arêtes tel que nous l'avons précédemment défini ne peut pas rerouter ce segment autrement. Les segments déroutés sont donc reroutés à l'identique restant ainsi les segments participant le plus à la congestion. L'ensemble des segments à dérouter ne change pas et l'outil **KNIK** tombe dans une impasse.

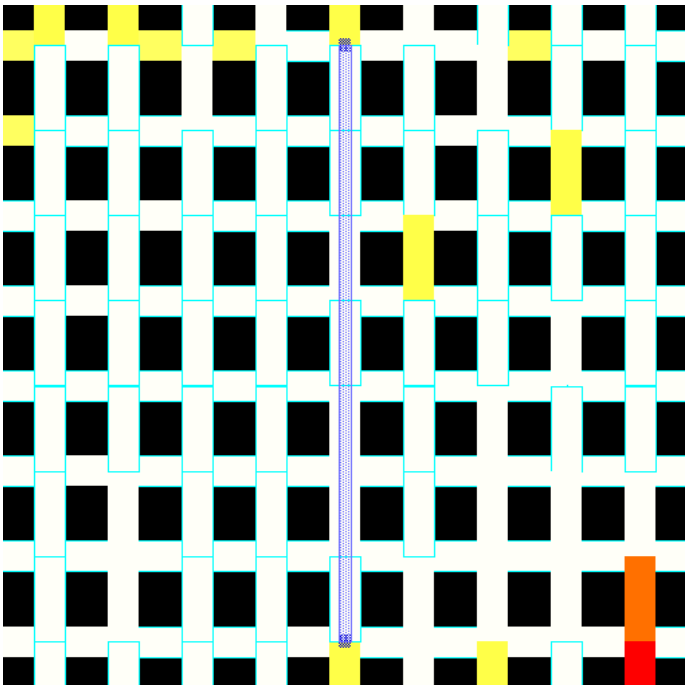


FIGURE 4.4 – Segment noyé dans une zone de sur-congestion

4.2.2 Technique de négociation de la congestion

Pour pallier ce problème, nous utilisons une technique de négociation de la congestion semblable à celle définie pour le routage de circuits FPGA dans [LC95].

Cette technique introduit un **coût d'historique de congestion** sur les arêtes du graphe de routage. Avant chaque itération de *ripup & reroute*, on procède à une analyse du graphe de routage et pour chaque arête ayant un dépassement non nul, son coût d'historique est incrémenté d'un facteur constant. Le coût de l'arête utilisé par l'algorithme de Dijkstra prend alors en compte ce nouveau coût.

2. Les arêtes représentées avec un contour de couleur turquoise ont un dépassement non nul.

De cette façon, au fur et à mesure des itérations de *ripup & reroute*, le coût d'une arête resur-congestionnée par l'algorithme de Dijkstra, va rapidement croître. Au bout d'un certain temps, le coût de l'arête est suffisamment grand pour que la chaîne de coût minimal construite par l'algorithme ne la recouvre plus.

Pour une arête $a \in A$, le coût d'historique de congestion à l'itération $n+1$ est défini par :

$$hCost(a, n+1) = \begin{cases} \alpha(a) \times hCost(a, n) & \text{si } tauxCongestion(a) \leq 1 \\ \alpha(a) \times (hCost(a, n) + hInc) & \text{sinon} \end{cases}$$

où $hInc$ est le facteur d'incrémentation constant. Une valeur élevée de $hInc$ permet une convergence rapide vers une solution valide, mais induit une augmentation significative de la longueur totale des interconnexions. Nous avons fixé sa valeur à $1,5$ après une étude sur les circuits de l'ispd98 (voir tableau en annexe page 165).

Le facteur $\alpha(a)$ sert à éviter que, dans les futures itérations, le coût d'historique d'une arête ayant été congestionnée ne perturbe les calculs. Nous le définissons de la manière suivante :

$$\alpha(a) = \begin{cases} tauxCongestion(a) & \text{si } tauxCongestion(a) \leq 1 \\ e^{\ln(8) \times (tauxCongestion(a) - 1)} & \text{sinon} \end{cases}$$

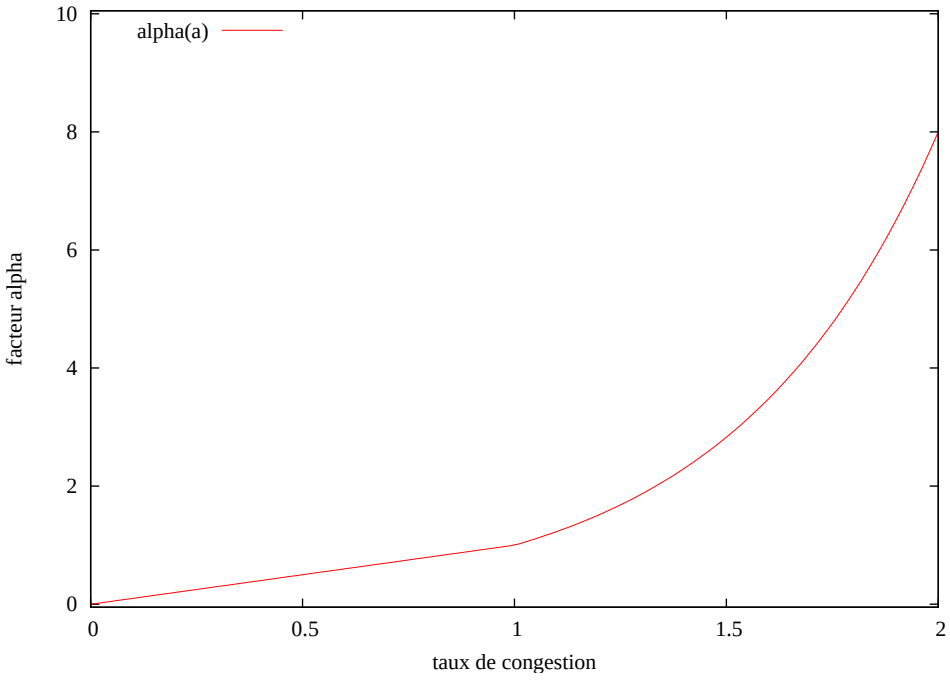
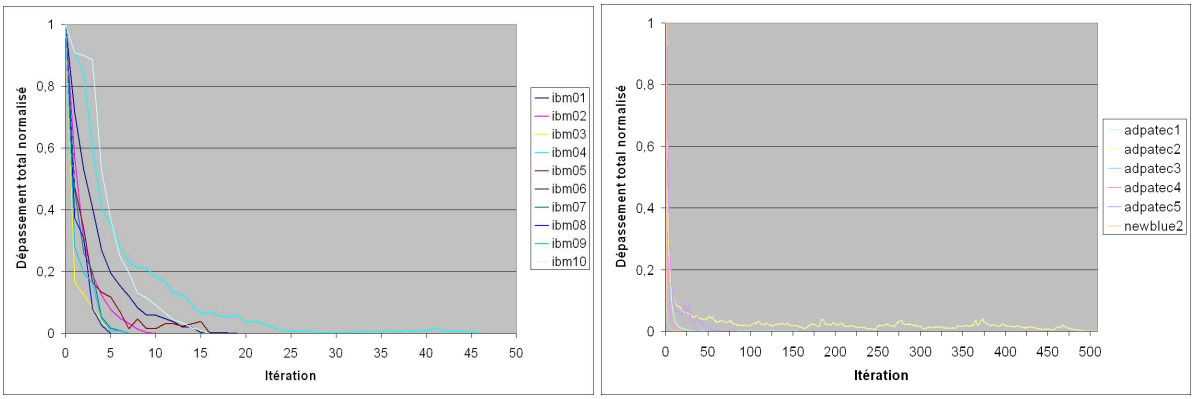


FIGURE 4.5 – Valeur du facteur α en fonction du taux de congestion de l'arête

La formule pour calculer le coût d'une arête $a \in A$ décrite page 50 du chapitre 2 reste valable si ce n'est qu'elle dépend désormais de l'itération de *ripup & reroute* courante et que nous lui ajoutons un terme prenant en compte le coût d'historique de congestion :

$$cout(a, n+1) = long(a) + coutVia(a) + coutCongestion(a) + hCost(a, n+1)$$

L'introduction de cette technique de négociation de la congestion dans l'outil **KNIK** nous a permis de construire une solution sans sur-congestion pour l'ensemble des circuits du jeu de test de l'ispd98 ainsi que pour six circuits (sur huit) du jeu de test de l'ispd07 (voir figure 4.6).



(a) ispd98 (b) ispd07

FIGURE 4.6 – Evolution des dépassements totaux avec négociation de la congestion

4.3 Résultats pour les circuits de l'*ispd98*

4.3.1 Caractéristiques des circuits

Les circuits du jeu de tests ispd98 ont été adaptés au routage global puisqu'à l'origine ils ont été définis pour une évaluation des algorithmes de placement.

Chacun des circuits, décrit au format *Labyrinth*, fournit un ensemble de nets composés de connecteurs ponctuels répartis à la surface du circuit. On ne considère que deux couches de métal : une pour les segments horizontaux et les connecteurs et une autre pour les segments verticaux. La taille du graphe de routage est fournie dans le fichier de test.

Le tableau 4.1 présente les caractéristiques des dix circuits de test.

	Nombre de connecteurs	Nombre de nets	Taille du graphe de routage (largeur×hauteur en nombre de sommets)
ibm01	44266	11507	64x64
ibm02	78171	18429	80x64
ibm03	75510	21621	80x64
ibm04	89591	26163	96x64
ibm05	124438	27777	128x64
ibm06	124299	33354	128x64
ibm07	164369	44394	192x64
ibm08	198180	47944	192x64
ibm09	187872	50393	265x64
ibm10	269000	64227	265x64

TABLE 4.1 – Caractéristiques des circuits du jeu de test ispd98

4.3.2 Résultats

Comme nous l'avons dit précédemment, la technique de négociation de la congestion nous permet, à l'aide de l'outil **KNIK**, de construire une solution sans sur-congestion pour les dix circuits de ce jeu de test.

Nous comparons donc nos résultats par rapport à ceux du routeur **FGR** en fonction des critères de temps d'exécution, de longueur des segments et de nombre de vias. Le routeur global **FGR** permet lui aussi de construire une solution sans sur-congestion pour tous les circuits du jeu de test.

Temps d'exécution

La figure 4.7 présente les temps d'exécution [3] de notre outil comparés à ceux du routeur global **FGR**. On constate que pour sept des dix circuits, notre outil **KNIK** est plus rapide que **FGR**. Les temps d'exécution sont réduits de 4,3% à 54,7%. De plus si l'on considère le temps nécessaire pour résoudre l'ensemble des circuits, l'outil **KNIK** est plus rapide avec une réduction du temps de l'ordre de 35%.

3. Le relevé complet est fournit en annexe page 166

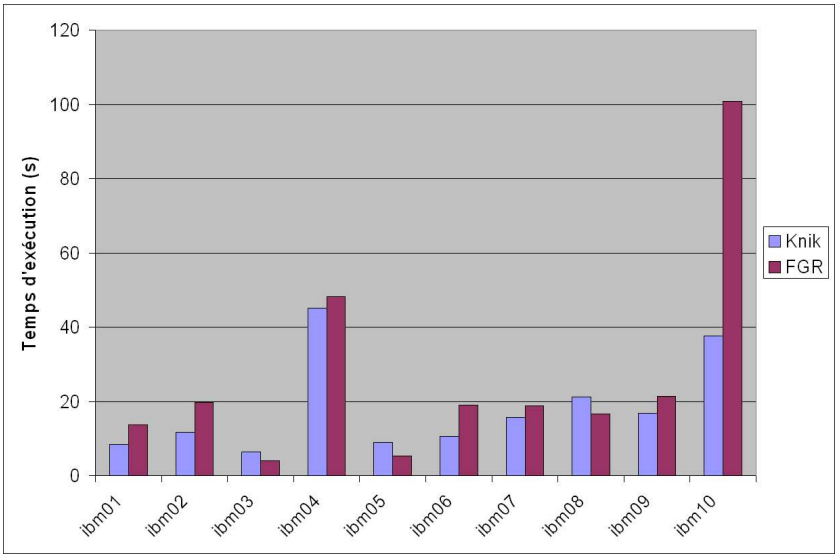


FIGURE 4.7 – Temps d'exécution comparés pour les circuits de l'ispd98

Cette réduction du temps d'exécution s'explique du fait que la solution initiale de routage global (avant *ripup & reroute*) produite par l'outil **KNIK** est meilleure en terme de sur-congestion que celle produite par **FGR**, comme l'illustre la figure 4.8.

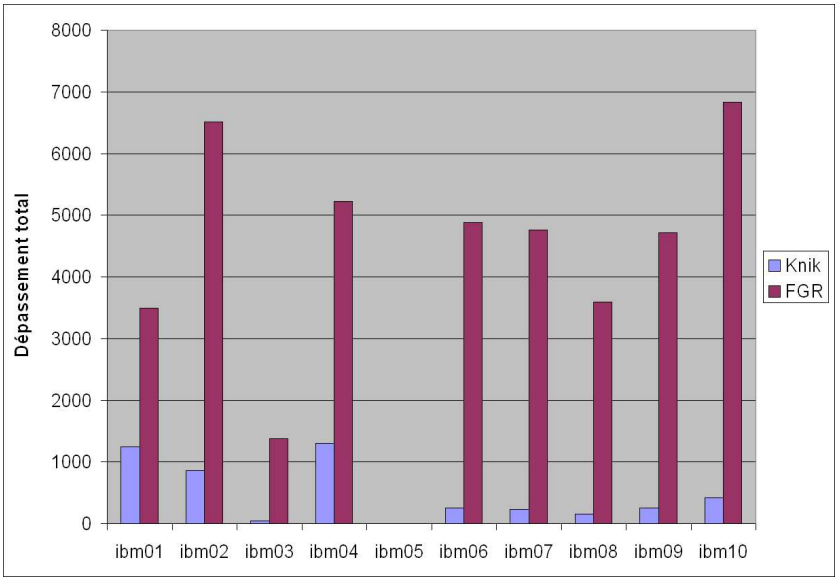


FIGURE 4.8 – Dépassements totaux comparés sans *ripup & reroute* pour les circuits de l'ispd98

Ainsi, si l'on étudie l'évolution des dépassements totaux au fur et à mesure des itérations de *ripup & reroute* pour les deux outils, on constate que l'outil **KNIK** converge plus rapidement vers une solution sans sur-congestion (voir figure 4.9 [4]).

4. Les évolutions pour les autres circuits de l'ispd98 sont fournis en annexe page 169.

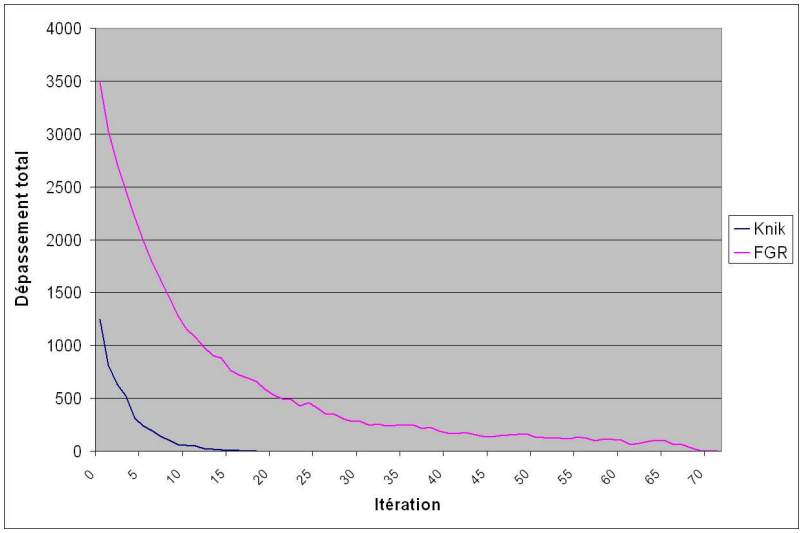


FIGURE 4.9 – Evolution comparée des dépassements totaux pour le circuit *ibm01*

Longueur des segments

En terme de longueur totale des segments, les outils **KNIK** et **FGR** produisent pour les circuits de l'ispd98 des solutions assez proches comme le montre la figure 4.10.

Bien que les solutions créées par **KNIK** aient toujours une longueur totale des segments plus importante, l'excédent est d'au maximum 1,7% par rapport à la solution créée par **FGR** . De plus si l'on considère la somme totale des longueurs de segments pour l'ensemble des circuits de l'ispd98 l'excédent des solutions créées par Knik est 0,60%.

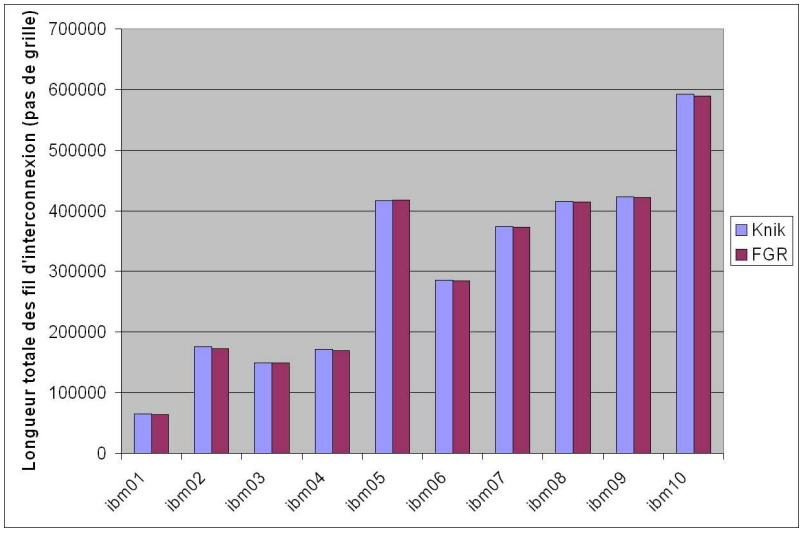


FIGURE 4.10 – Longueurs totales des segments comparées pour les circuits de l'ispd98

4.3 Résultats pour les circuits de l'*ispd98*

Nombre de vias

En comparant le nombre de vias des solutions créées par les deux outils (voir figure 4.11), on constate que **KNIK** génère un grand nombre de vias (de 12% à 16% de plus que l'outil **FGR**).

Ce surplus du nombre de vias provient du fait que **KNIK** n'offre qu'une prise en compte très simplifiée du nombre de vias. Seuls les vias correspondant aux changements de direction lors de la propagation du coût d'un segment sont considérés, comme nous l'avons décrit dans le chapitre 2.

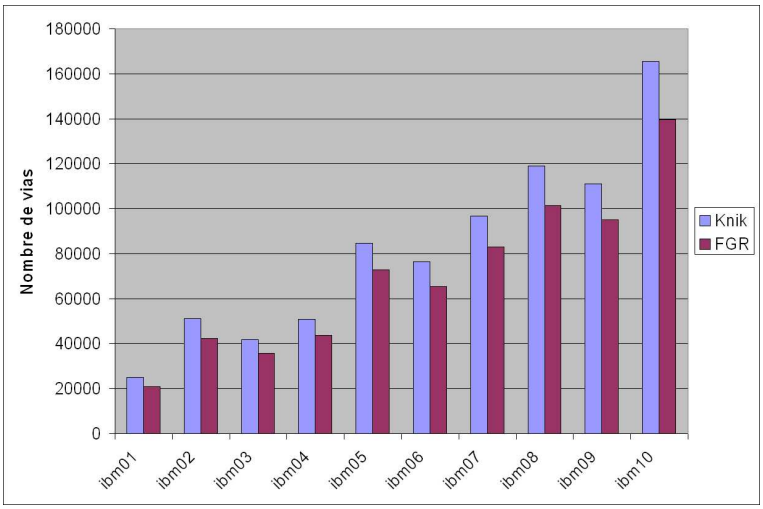


FIGURE 4.11 – Nombre de vias comparés pour les circuits de l'ispd98

La figure 4.12, comparant les routages effectués par **FGR** et **KNIK** du net *net10288* du circuit *ibm01*, met ce problème bien en évidence. Le net considéré est constitué de quatre connecteurs (représentés en rouge) associés à des sommets adjacents du graphe de routage. L'arbre d'interconnexion construit par **FGR** ne contient que deux vias (représentés en jaune) tandis que celui construit par **KNIK** en contient 4.

(a) **FGR** (b) **KNIK**

FIGURE 4.12 – Comparaison du routage du net *net10288* du circuit *ibm01*

Du point de vue de l'outil **KNIK**, les deux configurations ont en fait le même coût. En effet, si l'on considère que le net est situé dans une zone non congestionnée et que les arêtes ont toutes la même longueur, le coût des deux arbres d'interconnexion est le même. Les vias permettant d'interconnecter un segment vertical et un connecteur (placé sur la couche horizontale) ne sont pas pris en compte.

La solution pour pouvoir prendre en compte tous les vias consiste à étudier la composante connexe destination atteinte lors de la propagation des coûts des sommets. Si le sommet atteint est associé à un connecteur et que l'arête par laquelle on atteint le sommet est verticale, alors le coût de l'arête doit prendre en compte le coût d'un via.

Mais cette solution ne suffit pas. En effet, dans le cas d'un sommet atteint non associé à un connecteur, il faut étudier la « géométrie » de la composante connexe atteinte. Si cette composante correspond par exemple à un segment horizontal et que l'arête par laquelle on l'a atteinte est verticale, il faut compter le coût du via.

Une solution simple consiste à ajouter à chaque objet `vertex` une série de booléens permettant de savoir, si pour ce `vertex` et le net en cours de traitement :
- un connecteur lui est associé,
- un via a déjà été créé,
- une de ses arêtes `HEdgeOut` appartient à la même composante connexe,
- une de ses arêtes `VEdgeOut` appartient à la même composante connexe,
- une de ses arêtes `HEdgeIn` appartient à la même composante connexe,
- une de ses arêtes `VEdgeIn` appartient à la même composante connexe.

Cartes de congestion

L'étude des cartes de congestion des solutions générées par les deux outils [5], nous permet de mettre en évidence un autre problème de **KNIK** .

En comparant visuellement les deux cartes de congestion des figures 4.13 et 4.14 correspondant aux solutions créées par **FGR** et **KNIK** pour le circuit *ibm04*, on constate que **KNIK** génère une solution avec moins d'arêtes ayant un fort taux de congestion (blanches). En revanche les zones définies par les arêtes ayant un taux de congestion compris entre $0,7$ et $0,9$ sont plus étendues.

5. Toutes les cartes de congestion sont fournies en annexes à partir de la page 176.

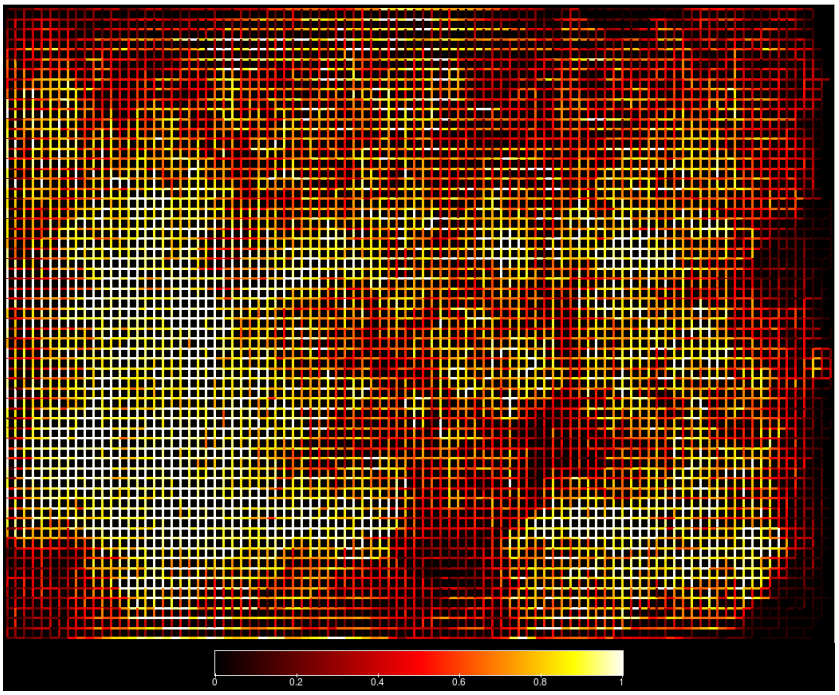


FIGURE 4.13 – Carte de congestion de la solution créée par **FGR** pour le circuit *ibm04*

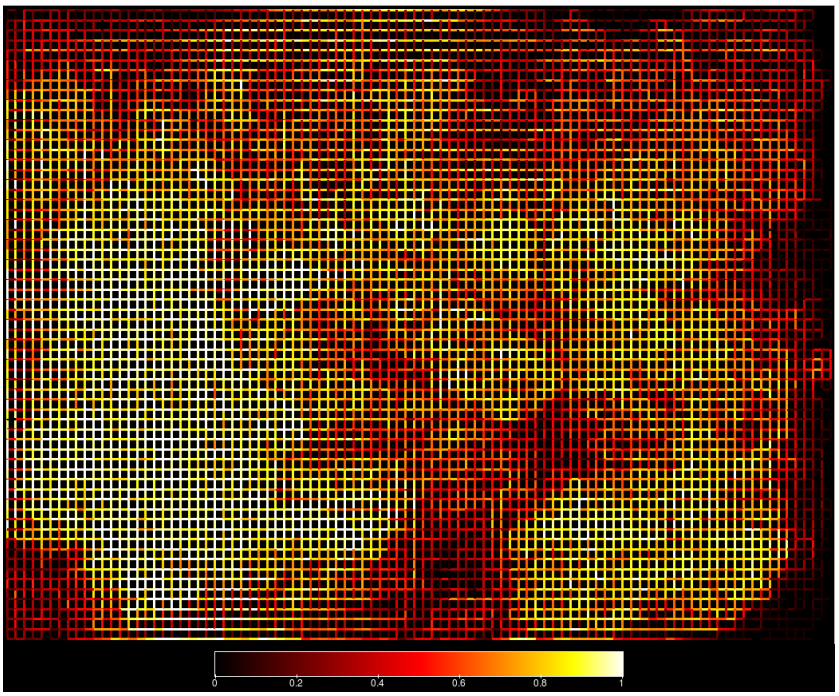


FIGURE 4.14 – Carte de congestion de la solution créée par **KNIK** pour le circuit *ibm04*

Une analyse détaillée du taux de congestion des arêtes du graphe de routage vient confirmer cette analyse visuelle. La figure 4.15 présente la répartition moyenne des arêtes suivant leur taux de congestion de l'ensemble des circuits de l'ispd98.

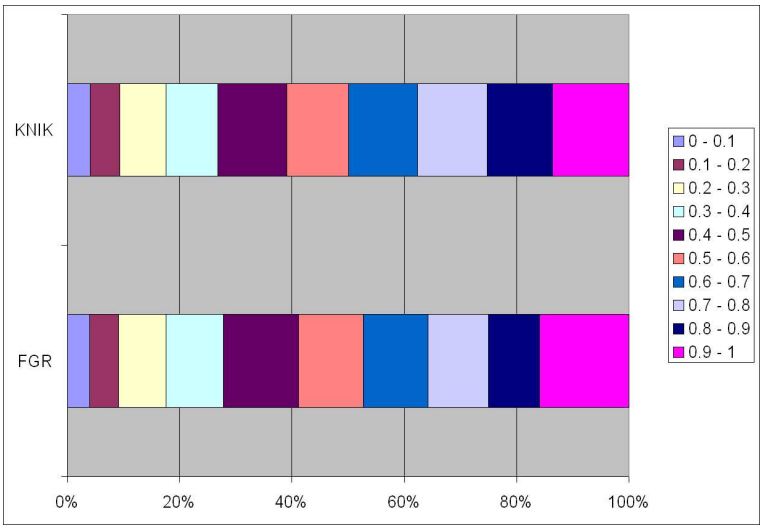


FIGURE 4.15 – Répartition moyenne des arêtes en fonction de leur taux de congestion pour les circuits de l'ispd98

On constate que **KNIK** génère des solutions dont 36,4% des arêtes ont un taux compris entre $0,6$ et $0,9$ (contre 31,4% pour **FGR**)[6] tandis que seulement 13,6% des arêtes ont un taux compris entre $0,9$ et 1 (contre 15,9% pour **FGR**).

Notons que cette répartition des arêtes n'influe pas sur le taux de congestion moyen des arêtes (voir tableau 4.2).

	KNIK	**FGR**
ibm01	0.61	0.60
ibm02	0.62	0.60
ibm03	0.59	0.59
ibm04	0.66	0.65
ibm05	0.49	0.49
ibm06	0.65	0.65
ibm07	0.53	0.53
ibm08	0.64	0.64
ibm09	0.61	0.61
ibm10	0.54	0.54

TABLE 4.2 – Taux de congestion moyens comparés pour les circuits de l'ispd98

6. La répartition pour chaque circuit est donnée en annexe page 174.

Globalement, cet étalement de la congestion du circuit se traduit par une augmentation des détours augmentant le nombre de vias et la longueur totale des segments.

4.4 Résultats pour les circuits de l'ispd07

4.4.1 Caractéristiques des circuits

Les circuits du jeu de tests ispd07 ont été spécialement conçus pour le routage global. Il existe une version deux dimensions et trois dimensions de chacun des circuits. Nous ne considérons que les circuits à deux dimensions puisque notre outil **KNIK** ne gère que ceux là.

Les circuits sont décrits au format *Labyrinth* et contiennent une description d'obstacles sous la forme d'une réduction de la capacité des arêtes du graphe de routage. Contrairement aux circuits de l'ispd98, toutes les arêtes de même direction d'un graphe de routage n'ont donc pas la même capacité.

Le tableau 4.3 présente les caractéristiques des huit circuits de ce jeu de tests.

	Nombre de connecteurs	**Nombre de nets**	**Taille du graphe de routage** (largeur×hauteur en nombre de sommets)
adaptec1	942705	219794	324x324
adaptec2	1063632	260159	424x424
adaptec3	1874576	466295	774x779
adaptec4	1911773	515304	774x779
adaptec5	3492790	867441	465x468
newblue1	1237104	331663	399x399
newblue2	1771849	463213	557x463
newblue3	1929360	551667	973x1256

TABLE 4.3 – Caractéristiques des circuits du jeu de test ispd07

Que ce soit en termes de nombre de connecteurs, de nets ou en taille du graphe de routage, ces circuits sont beaucoup plus « gros » que les circuits de l'ispd98.
L'ordre de grandeur du temps d'exécution nécessaire à la construction d'une solution sans sur-congestion passe de quelques secondes à plusieurs minutes voire plusieurs heures.

4.4.2 Résultats

Aucun des deux outils n'a été en mesure de construire une solution sans sur-congestion pour les circuits *newblue1* et *newblue3* dans une limite de vingt-quatre

heures d'exécution. En revanche pour les six autres circuits des solutions valides existent.

Longueur des segments et nombre de vias

Globalement les résultats de ces circuits mettent en évidence les problèmes précédemment évoqués.

La longueur totale des segments générés par l'outil **KNIK** est en moyenne 3,7% plus grande (voir figure 4.16) tandis que le nombre de vias est 15,4% plus important (voir figure 4.17).

La répartition des arêtes suivant leur taux de congestion est assez différente (voir figure 4.18) notamment parce que les circuits du l'ispd07 sont globalement moins denses. Mais, comme pour les circuits de l'ispd98, on constate que **KNIK** crée des solutions dont le pourcentage des arêtes ayant un taux compris entre $0,6$ et $0,9$ (11,9%) est supérieur à celui des solutions créees par **FGR** (10%)[7] tandis que le pourcentage d'arêtes ayant un taux compris entre $0,9$ et 1 est inférieur (4,7% pour **KNIK** contre 5,2% pour **FGR**).

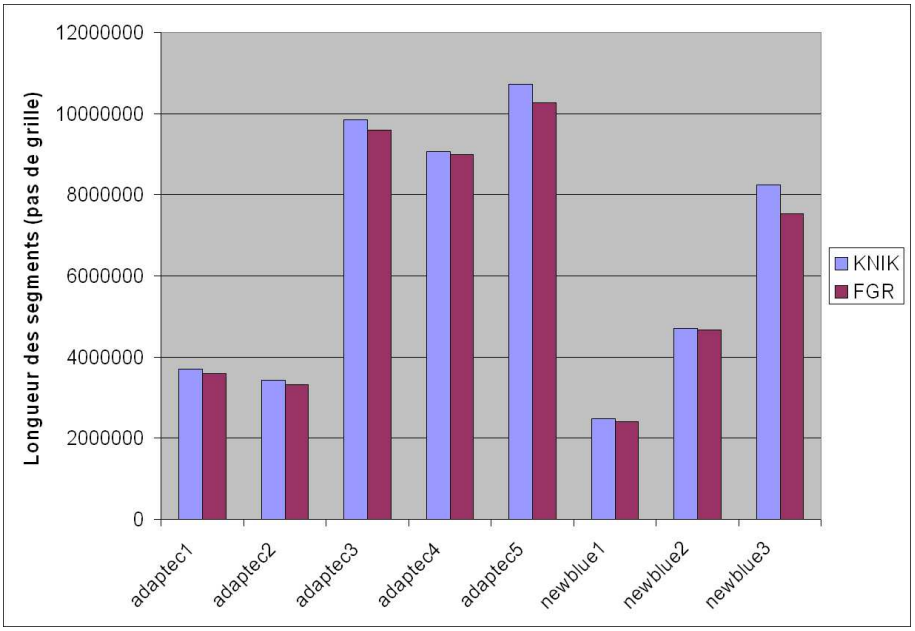


FIGURE 4.16 – Longueurs totales des segments comparées pour les circuits de l'ispd07

7. La répartition pour chaque circuit est donnée en annexe page 175.

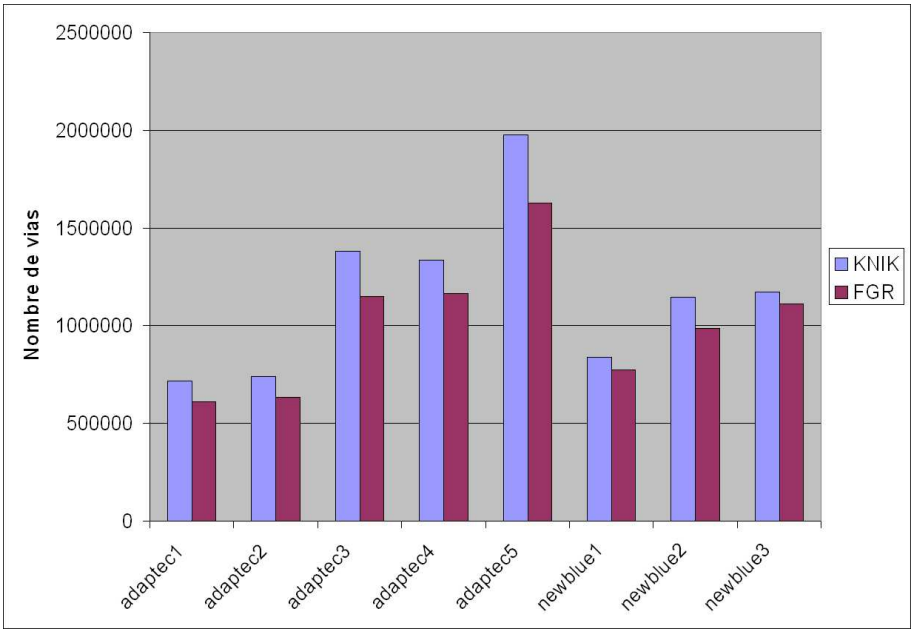


FIGURE 4.17 – Nombre de vias comparés pour les circuits de l'ispd07

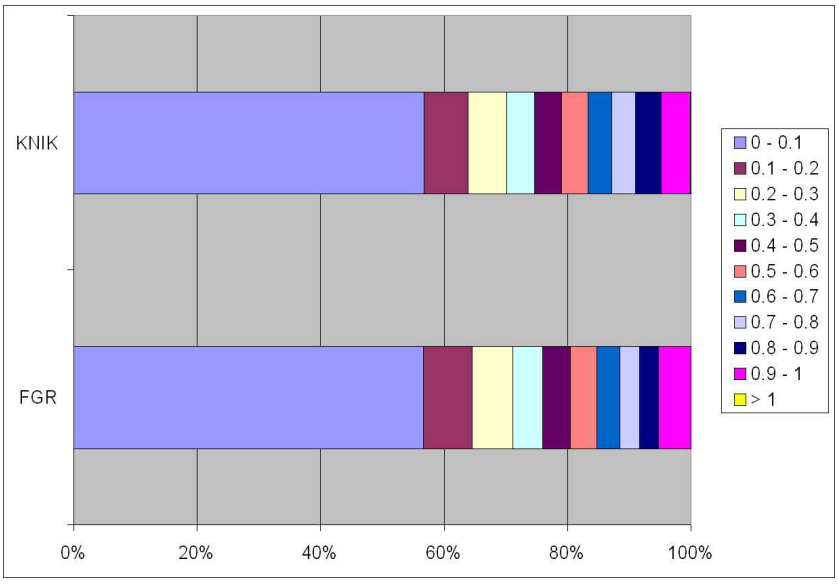


FIGURE 4.18 – Répartition moyenne des arêtes en fonction de leur taux de congestion pour les circuits de l'ispd07

Temps d'exécution

En ce qui concerne le temps d'exécution, l'outil **KNIK** construit des solutions valides plus rapidement pour seulement quatre circuits, comme le montre la figure 4.19.

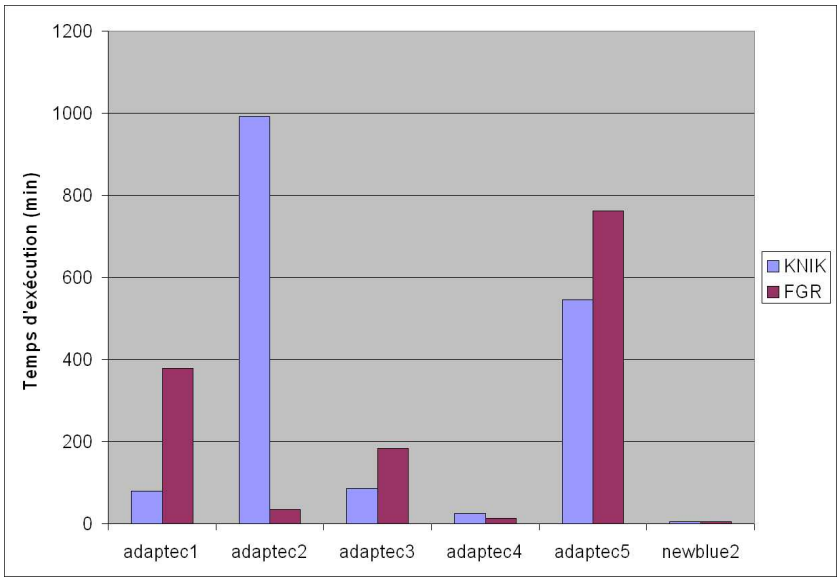


FIGURE 4.19 – Temps d'exécution comparés pour les circuits de l'ispd07

Le circuit *adaptec2* pose un gros problème à notre outil qui, bien qu'il réussisse à construire une solution sans sur-congestion, procède à de très (trop) nombreuses itérations de *ripup & reroute*.

La comparaison de l'évolution de la somme des dépassements au fur et à mesure des itérations de *ripup & reroute*, illustrée sur la figure 4.20, met bien en évidence le fait que l'outil **KNIK** « stagne » pendant un certain temps sans arriver à réduire significativement la sur-congestion (entre les itérations 160 et 420 environ).

Il est important de noter que le temps moyen d'une itération de *ripup & reroute* pour l'outil **KNIK** est bien supérieur à celui de l'outil **FGR** pour ce circuit : $116,9$ secondes pour **KNIK** contre seulement $6,9$ secondes pour **FGR** . En moyenne pour les circuits de l'ispd98, le temps d'exécution d'une itération est 3 fois plus long pour **KNIK** que pour **FGR** , et 1,8 fois plus long pour les circuits de l'ispd07.

Cette lenteur des itérations de *ripup & reroute* s'explique notamment par la nécessité d'augmenter la fenêtre d'exploration de l'algorithme de Dijkstra à l'ensemble du circuit. En effet, le temps nécessaire pour propager les coûts des sommets augmente avec la taille de la fenêtre d'exploration et donc la taille du circuit.

4.4 Résultats pour les circuits de l'ispd07

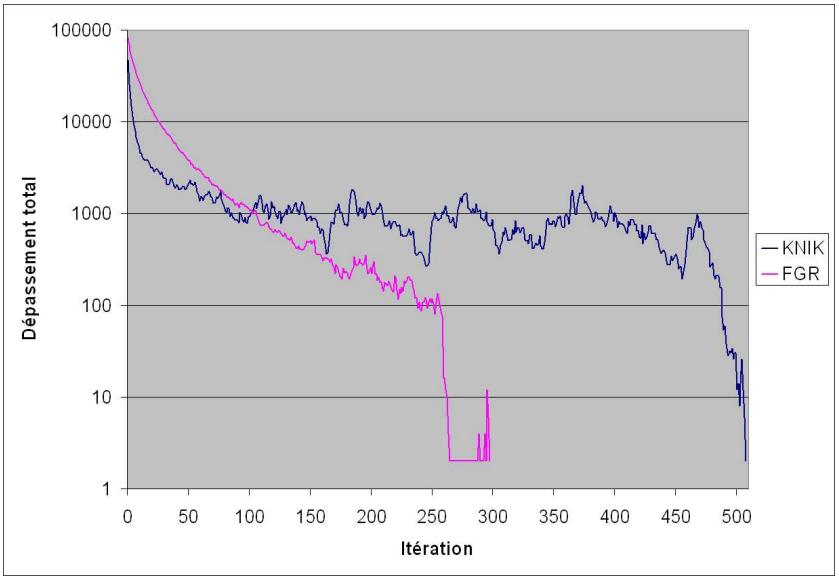


FIGURE 4.20 – Evolutions comparées des dépassements totaux en fonction des itérations de *ripup & reroute* pour le circuit *adaptec2*

La construction d'une solution initiale (sans *ripup & reroute*) par **KNIK** pour les 442005 nets du circuit *newblue3* s'effectue en $227,19$ secondes, soit en moyenne $1594,6$ nets routés par seconde. Alors que pendant les itérations de *ripup & reroute*, l'outil **KNIK** route en moyenne $2,1$ nets par seconde seulement !

Pour s'affranchir de ce problème de fenêtre d'exploration, il est possible d'utiliser un algorithme de type A* à la place de l'algorithme de Dijkstra. Cet algorithme utilise une évaluation du coût restant vers la cible pour orienter la propagation des coûts des sommets vers celle-ci, reduisant ainsi le nombre de sommets traités et donc le temps d'exécution.

Tout comme l'agorithme de Dijkstra, l'algorithme A* nécessite d'être adapté aux composantes connexes comme nous l'avons évoqué page 76 du chapitre 2. Dans le cas de traitement multi composantes le coût restant pour un sommet du graphe est égal au minimum des coûts restants par rapport aux différentes composantes connexes. De cette manière aucune composante connexe destination n'est favorisée.

Conclusion

La première conclusion que nous pouvons tirer de nos résultats est que notre approche à base de composantes connexes est valide tant pour le routage global par l'agorithme de Dijkstra que pour le *ripup & reroute*. En utilisant des algorithmes simples, nous parvenons à construire des solutions comparables à celles créées par le routeur global **FGR** qui est la référence actuelle dans le domaine du routage global académique.

Nous avons mis en évidence deux problèmes dans l'outil **KNIK**. Premièrement nous avons pu constater que le nombre de vias des solutions générées par **KNIK** est en moyenne plus important de 14% par rapport à **FGR**. Cette différence est dûe à notre modèle de prise en compte des vias trop simpliste. En effet, celui-ci ne gère pas les vias liés au sommet d'une composante connexe atteinte lors de la propagation du coût des vias.

Nous avons aussi constaté une lenteur à l'exécution lors des phases de *ripup & reroute* dûe à l'agrandissement de la fenêtre d'exploration à l'ensemble du circuit. Pour réduire le nombre de sommets explorés et donc accélérer le traitement, nous proposons d'utiliser un algorithme A* adapté au traitement multi composantes, à la place de notre algorithme de Dijkstra.

Enfin, en étudiant les cartes de congestion des circuits, nous avons constaté que notre outil **KNIK** a tendance à étaler la congestion plus que nécessaire. Les arêtes ayant un taux de congestion compris entre $0,9$ et 1 sont moins nombreuses que dans les solutions générées par **FGR**. Il serait intéressant d'étudier l'influence du point de montée de la fonction de coût de congestion des arêtes définie page 107. Notamment en le rappochant de 1 pour voir si le nombre d'arêtes ayant un taux de congestion compris entre $0,9$ et 1 augmente.

Conclusions et perspectives

Conclusions

Dans le cadre de cette thèse, nous avons développé **KNIK** un outil de routage global au sein de la plate-forme **CORIOLIS** en adéquation avec les besoins de celle-ci. Notre outil est rapide et performant parce qu'il est basé sur une mise en œuvre optimisée de l'algorithme de Dijkstra et qu'il utilise une structure de données légère et adaptée au problème.

Pour évaluer les performances de notre outil, nous avons développé un environnement d'évaluation permettant de charger un circuit de test et de construire une solution de routage global, qui peut être visualisée, évaluée et enfin, sauvegardée sur disque. Nous respectons les formats d'entrée / sortie de l'ispd07 ce qui nous permet, en outre, de charger un circuit de test et une solution correspondante, qu'elle ait été générée par **KNIK** ou tout autre routeur global respectant le format de sortie *BoxRouter*.

Pour la mise en œuvre de notre outil, nous avons opté pour une approche séquentielle, de façon à pouvoir traiter efficacement de gros circuits, et plus particulièrement une méthode de type *Maze routing* qui permet de définir une bonne fonction de coût utilisée par un algorithme simple et rapide, tel que l'algorithme de Dijkstra.

Nous avons défini une structure de données légère et compacte qui permet de représenter le graphe, régulier ou non, modélisant les ressources de routage. Nous avons rajouté toutes les fonctionnalités nécessaires à l'utilisation d'un algorithme de routage global sur cette structure, telles que la recherche du sommet associé à un point quelconque de la surface du circuit, ou encore les fonctions de visualisation du graphe de routage.

Par la suite, nous avons introduit la notion originale de composante connexe qui permet de représenter toute partie d'un net à n'importe quelle étape de l'algorithme de routage. Cette notion est particulièrement intéressante dans le cadre du *ripup & reroute* puisqu'elle permet de ne dérouter que les parties sur-congestionnées des nets, et non pas tous les nets dont une partie est sur-congestionnée. Nous avons montré que grâce aux deux identificateurs `netStamp` et `connexId`, il est possible de représenter et manipuler très simplement les composantes connexes sur notre structure de graphe de routage. De plus, nous avons présenté les extensions nécessaires à l'algorithme de Dijkstra pour l'adapter au traitement de composantes connexes.

Pour gérer la congestion nous avons défini une fonction de coût qui prend en compte

la congestion dûe aux arbres d'interconnexions construits et une estimation anticipée de la congestion permettant de guider les choix faits pour les premiers nets traités. Cette estimation est construite à l'aide de l'outil **FLUTE** en calculant des probabilités de congestion et est actualisée à chaque nouveau net traité par l'algorithme de Dijkstra.

Une fois une solution initiale construite, nous utilisons une phase de *ripup & reroute* pour éliminer la sur-congestion. Pour cela, nous avons défini une fonction de coût permettant d'évaluer la contribution à la sur-congestion pour chaque segment d'un net et ainsi ne rerouter que les parties sur-congestionnées des nets.

Nos résultats préliminaires ont mis en évidence que cette première approche de *ripup & reroute* est trop simple. Nous avons alors mis en œuvre une technique de négociation de la congestion inspirée de celle utilisée pour les circuits FPGA (Field-Programmable Gate Array) qui nous a permis d'obtenir des solutions valides (pour tous les circuits de l'ispd98 et pour six des huits circuits de l'ispd07) et comparables à celles construites par **FGR** qui est le routeur global de référence dans le milieu académique.

Perspectives

Dans la continuité de nos travaux, il serait intéressant d'étudier différents points. Tout d'abord il est nécessaire d'améliorer la prise en compte des vias et le temps d'exécution des itérations de *ripup & reroute*. Ensuite, nous pouvons profiter de la modularité offerte par notre outil et la plate-forme **CORIOLIS** pour étudier divers paramètres du routage global et même mettre en œuvre de nouveaux algorithmes.

Prise en compte des vias

Comme nous l'avons montré dans le chapitre 4, notre modèle de prise en compte des vias est trop simple et ne permet pas de considérer tous les vias lors du calcul du coût d'un arbre d'interconnexion.

Pour prendre en compte tous les vias et ainsi réduire leur nombre dans les solutions générées par **KNIK**, il faut être capable de savoir si un connecteur est associé à un `vertex` donné et, dans le cas où ce `vertex` appartient à une composante connexe, d'étudier la géométrie de cette dernière.

La solution simple que nous avons proposée dans le chapitre 4 doit être mise en œuvre et testée avec notre outil.

Temps d'exécution des phases de *ripup & reroute*

Nous avons constaté que, dans les phases de *ripup & reroute*, l'agrandissement de la fenêtre d'exploration à l'ensemble de la surface du circuit peut très fortement ralentir notre outil dans le cas de gros circuits.

Pour s'affranchir de ce défaut, nous proposons d'utiliser un algorithme A* adapté au traitement multi-composantes à la place de notre algorithme de Dijkstra. Le problème réside dans le fait de trouver une estimation de la borne inférieure du coût restant par rapport à une composante connexe non ponctuelle pour un sommet fixé. Comme nous l'avons vu dans le chapitre 2, une stratégie bien adaptée consiste à choisir d'une part un représentant ponctuel pour chaque composante cible afin de pouvoir calculer simplement la borne inférieure du coût, mais aussi, d'autre part, d'affecter aux arêtes de la composante cible un coût nul. Ainsi lorsque la vague atteint un sommet quelconque de la composante cible, elle est guidée directement vers son représentant.

Dans le cas d'un traitement multi-composantes, le coût restant d'un sommet est égal au minimum des coûts restants par rapport à chacune des composantes connexes destination. De cette façon, la propagation du coût des sommets est orientée vers les composantes destination sans être limitée à une fenêtre d'exploration empêchant de construire une solution sans sur-congestion.

Etude des paramètres du routage global

Comme nous l'avons constaté dans le chapitre 4, notre outil a tendance à étaler la congestion ce qui augmente sensiblement le nombre de vias et la longueur des fils d'interconnexion. Si nous étudions de plus près la fonction de coût liée à la congestion que nous avons définie dans le chapitre 3, nous constatons que pour un taux de congestion compris entre 0,9 et 1, le coût augmente rapidement de 0,5 à 5.

Il serait intéressant de « raidir » la pente de cette fonction de coût de façon à diminuer le coût pour des taux compris entre 0,9 et 0,96 et d'en étudier l'impact sur la répartition des arêtes en fonction du taux de congestion. Du fait de la modularité de notre outil, ces changements sont très simples puisqu'il suffit de faire varier les paramètres h et k de la fonction.

Prise en compte de contraintes temporelles

Comme nous l'avons vu dans ce manuscrit, l'un des objectifs principaux du routage global est de réduire la longueur déployée des arbres d'interconnexions de façon à réduire la charge capacitive sur les émetteurs et donc les délais de propagation des signaux. Pour optimiser ces délais, on peut aussi s'intéresser à la topologie des arbres d'interconnexion.

En effet, les contraintes sur les délais entre l'émetteur et les différents récepteurs peuvent fortement varier selon qu'ils font partie ou non d'un chemin critique. Or ces délais sont en première approximation liés au RC des chemins qui relient les émetteurs aux récepteurs. Dès lors, il devient judicieux de réaliser d'abord et au plus court les connexions entre émetteurs et leurs seuls récepteurs sur des chemins critiques (donc sans grandes contraintes de congestion), puis dans des passes ultérieures de finaliser pour chaque net, le raccordement des récepteurs restants (quitte à devoir faire des détours en raison de la congestion).

De telles stratégies ont été proposées dans [HNJR07], leur mise en œuvre selon le scénario ci-dessus se prette bien à l'approche multi composantes de notre modélisation.

Interaction avec le routage détaillé

Dans le cadre d'une utilisation conjointe avec l'outil de routage détaillé en cours de développement au sein de la plate-fome **CORIOLIS**, il nous paraît judicieux d'essayer de réutiliser notre structure de graphe pour le routage détaillé en l'enrichissant de façon à satisfaire les besoins de ce dernier. L'utilisation de cette structure commune simplifie l'interaction des deux outils. En effet, si le routeur détaillé détecte une zone pour laquelle il ne peut résoudre les problèmes locaux, il peut annoter la structure et demander au routeur global de recommencer un routage avec de nouvelles contraintes.

De plus, le routeur détaillé peut servir à valider notre outil de routage global en confirmant qu'il est possible à partir de la solution construite d'obtenir un routage détaillé valide. Cette validation n'est pas limitée à notre outil puisque notre environnement d'évaluation nous permet de charger les solutions générées par les routeurs globaux académiques concurrents. Après chargement d'une solution dans l'environnement d'évaluation, il est très simple de confirmer qu'elle peut conduire à un routage complet valide en utilisant le routeur détaillé.

Routage global analogique

Dans le cadre du routage global analogique, la problématique est légèrement différente du fait qu'on utilise des canaux de routage et non plus une approche « *over the cell routing* ». La principale contrainte liée à ce changement est que le graphe de routage associé à ces canaux est irrégulier.

A priori, notre modèle de graphe de routage irrégulier offre suffisamment de flexibilité pour être étendu à cette problématique. Il deviendrait alors possible d'utiliser notre outil pour générer une solution de routage global pour des circuits analogiques.

Bibliographie

[aA] Wikipedia : algorithme A*. http ://fr.wikipedia.org/wiki/algorithme_a*.

[ACC+05] Christophe Alexandre, Hugo Clement, Jean-Paul Chaput, Marek Sroka, Christian Masson, and Remy Escassut. Tsunami : An integrated timing-driven place and route research platform. In *DATE '05 : Proceedings of the 2005 Design, Automation, and Test in Europe*, pages 920–921, 2005.

[adD] Wikipedia : algorithme de Dijkstra. http ://fr.wikipedia.org/wiki/algorithme_de_dijkstra.

[adP] Wikipedia : algorithme de Prim. http ://fr.wikipedia.org/wiki/algorithme_de_prim.

[aFGR] FGR : a Fairly Good Router. http ://vlsicad.eecs.umich.edu/bk/fgr/.

[Alb01] Christoph Albrecht. Global routing by new approximation algorithms for multicommodityflow. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(5) :622–632, 2001.

[Ale07] Christohe Alexandre. *Coriolis : une plate-forme ouverte pour l'évaluation de flots de conception VLSI fortement intégrés*. PhD thesis, Université Pierre et Marie Curie Paris VI, septembre 2007.

[ASCM06] Christophe Alexandre, Marek Sroka, Hugo Clement, and Christian Masson. Zephyr : a static timing analyzer integrated in a trans-hierarchical refinement design flow. In *PATMOS '06 : Proceedings of the 2006 Power And Timing Modeling Optimization and Simulation conference*, pages 319–328, 2006.

[ben] ISPD98 / IBM benchmarks. http ://www.ece.ucsb.edu/∼kastner/labyrinth/benchmarks/.

[bh] Wikipédia : binary heap. http ://en.wikipedia.org/wiki/binary_heap.

[Chu04] Chris Chu. Flute : fast lookup table based wirelength estimation technique. In *ICCAD '04 : Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 696–701, 2004.

[CJX+07] Zhen Cao, Tong Jing, Jinjun Xiong, Yu Hu, Lei He, and Xianlong Hong. Dprouter : A fast and accurate dynamic-pattern-based global routing algorithm. In *ASP-DAC '07 : Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 256–261, 2007.

[CLYP07] Minsik Cho, Katrina Lu, Kun Yuan, and David Z. Pan. Boxrouter 2.0 : architecture and implementation of a hybrid and robust global router. In *ICCAD '07 : Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 503–508, 2007.

[CM98] Jason Cong and Patrick H. Madden. Performance driven multi-layer general area routing for pcb/mcm designs. In *DAC '98 : Proceedings of the 35th annual conference on Design automation*, pages 356–361, 1998.

[CP06] Minsik Cho and David Z. Pan. Boxrouter : a new global router based on box expansion and progressive ilp. In *DAC '06 : Proceedings of the 43rd annual conference on Design automation*, pages 373–378, 2006.

[CW05] Chris Chu and Yiu-Chung Wong. Fast and accurate rectilinear steiner minimal tree algorithm for vlsi design. In *ISPD '05 : Proceedings of the 2005 international symposium on Physical design*, pages 28–35, 2005.

[dCA] Chaîne de CAO Alliance. http ://www-asim.lip6.fr/recherche/alliance.

[DK99] Sylvester Dennis and Keutzer Kurt. Getting to the bottom of deep submicron ii : a global wiring paradigm. In *ISPD '99 : Proceedings of the 1999 international symposium on Physical design*, pages 193–200, 1999.

[Edi] Ubuntu 8.10 Desktop Edition. http ://www.ubuntu.com/products/whatisubuntu/810features/.

[fC] Plate forme Coriolis. http ://www-asim.lip6.fr/recherche/coriolis/.

[fRCE] FLUTE : Fast Lookup Table Based Technique for RSMT Construction and Wirelength Estimation. http ://home.eng.iastate.edu/∼cnchu/flute.html.

[GP92] Alain Greiner and François Pecheux. Alliance : A complete set of cad tools for teaching vlsi design. 1992.

[GWW08] Jhih-Rong Gao, Pei-Ci Wu, and Ting-Chi Wang. A new global router for modern designs. In *ASP-DAC '08 : Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 232–237, 2008.

[Han66] Maurice Hanan. On steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14 :255–265, 1966.

[hea] Wikipédia : Fibonacci heap. http ://en.wikipedia.org/wiki/fibonacci_heap.

[HM03] Raia T. Hadsell and Patrick H. Madden. Improved global routing through congestion estimation. In *DAC '03 : Proceedings of the 40th conference on Design automation*, pages 28–31, 2003.

[HNJR07] Renato F. Hentschke, Jaganathan Narasimham, Marcelo O. Johann, and Ricardo L. Reis. Maze routing steiner trees with effective critical sink optimization. In *ISPD '07 : Proceedings of the 2007 international symposium on Physical design*, pages 135–142, 2007.

[HRM08] Jin Hu, Jarrod A. Roy, and Igor L. Markov. Sidewinder : a scalable ilp-based router. In *SLIP '08 : Proceedings of the 2008 international workshop on System level interconnect prediction*, pages 73–80, 2008.

[ifs] Labyrinth input format specification. http ://www.ece.ucsb.edu/∼kastner/labyrinth/doc/inputformat.txt.

[KMZ03] Andrew B. Kahng, Ion I. Măndoiu, and Alexander Z. Zelikovsky. Highly scalable algorithms for rectilinear and octilinear steiner trees. In *ASPDAC '03 : Proceedings of the 2003 conference on Asia South Pacific design automation*, pages 827–833, 2003.

[lc] Wikipedia : liste chaînée. http ://fr.wikipedia.org/wiki/liste_chainée.

[LC95] McMurchie Larry and Ebeling Carl. Pathfinder : a negotiation-based performance-driven router for fpgas. In *FPGA '95 : Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 111–117, 1995.

[Lib] Standard Template Library. http ://www.sgi.com/tech/stl/.

[Lin84] Ralph Linsker. An iterative-improvement penalty-function-driven wire routing system. *IBM J. Res. Dev.*, 28(5) :613–624, 1984.

[LW06] Kuang-Yao Lee and Ting-Chi Wang. Post-routing redundant via insertion for yield/reliability improvement. In *ASP-DAC '06 : Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 303–308, 2006.

[MÖ6] Dirk Müller. Optimizing yield in global routing. In *ICCAD '06 : Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 480–486, 2006.

[Mof08] Michael D. Moffitt. Maizerouter : engineering an effective global router. In *ASP-DAC '08 : Proceedings of the 2008 conference on Asia and South Pacific design automation*, pages 226–231, 2008.

[MRM08] Michael D. Moffitt, Jarrod A. Roy, and Igor L. Markov. The coming of age of (academic) global routing. In *ISPD '08 : Proceedings of the 2008 international symposium on Physical design*, pages 148–155, 2008.

[NSY08] Gi-Joon Nam, Cliff Sze, and Mehmet Yildiz. The ispd global routing benchmark suite. In *ISPD '08 : Proceedings of the 2008 international symposium on Physical design*, pages 156–159, 2008.

[ofs] BoxRouter output format specification. http ://www.cerc.utexas.edu/ ∼thyeros/boxrouter/boxrouter.htm#output format.

[oPDGRCa] International Symposium on Physical Design Global Routing Contest 2007. http ://www.sigda.org/ispd2007/contest.html.

[oPDGRCb] International Symposium on Physical Design Global Routing Contest 2008. http ://www.sigda.org/ispd2008/contests/ispd08rc.html.

[Ott98] Ralph H. J. M. Otten. Global wires : harmful ? In *ISPD '98 : Proceedings of the 1998 international symposium on Physical design*, pages 104–109, 1998.

[OW07] Muhammet Mustafa Ozdal and Martin D. F. Wong. Archer : a history-driven global routing algorithm. In *ICCAD '07 : Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 488–495, 2007.

[PC06] Min Pan and Chris Chu. Fastroute : a step to integrate global routing into placement. In *ICCAD '06 : Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 464–471, 2006.

[PC07] Min Pan and C. Chu. Fastroute 2.0 : A high-quality and efficient global router. In *ASP-DAC '07 : Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 250–255, 2007.

[Quea] Wikipedia : Priority Queue.
http ://en.wikipedia.org/wiki/priority_queue.

[Queb] Priority Queues.
http ://www.theturingmachine.com/algorithms/heaps.html.

[RA] FastSteiner : Highly Scalable Rectilinear and Octilinear Minimum Steiner Tree Algorithms. http ://vlsicad.ucsd.edu/gsrc/bookshelf/slots/rsmt/faststeiner/.

[RLM06] Jarrod A. Roy, James F. Lu, and Igor L. Markov. Seeing the forest and the trees : Steiner wirelength optimization in placement. In *ISPD '06 : Proceedings of the 2006 international symposium on Physical design*, pages 78–85, 2006.

[RM07] Jarrod A. Roy and Igor L. Markov. High-performance routing at the nanometer scale. In *ICCAD '07 : Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 496–502, 2007.

[RM08] Jarrod A. Roy and Igor L. Markov. High-performance routing at the nanometer scale. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(6) :1066–1077, 2008.

[Spe] Intel® Core™2 Quad Processor Specifications. http ://www.intel.com/products/processor/core2quad/specifications.htm.

[sS] set STL. http ://www.sgi.com/tech/stl/set.html.

[TCR94] Charles Leiserson Thomas Cormen and Ronald Rivet. *Introduction à l'algorithmique*. Dunod, 1994.

[TDNS07] Taraneh Taghavi, Foad Dabiri, Ani Nahapetian, and Majid Sarrafzadeh. Tutorial on congestion prediction. In *SLIP '07 : Proceedings of the 2007 international workshop on System level interconnect prediction*, pages 15–24, 2007.

[WBG04] Jurjen Westra, Chris Bartels, and Patrick Groeneveld. Probabilistic congestion prediction. In *ISPD '04 : Proceedings of the 2004 international symposium on Physical design*, pages 204–209, 2004.

[WG05] Jurjen Westra and Patrick Groeneveld. Is probabilistic congestion estimation worthwhile ? In *SLIP '05 : Proceedings of the 2005 international workshop on System level interconnect prediction*, pages 99–106, 2005.

[WGYM05] Jurjen Westra, Patrick Groeneveld, Tan Yan, and Patrick H. Madden. Global routing : metrics, benchmarks, and tools. In *EDP '05 : Proceedings of the 2005 electronic design processes*, 2005.

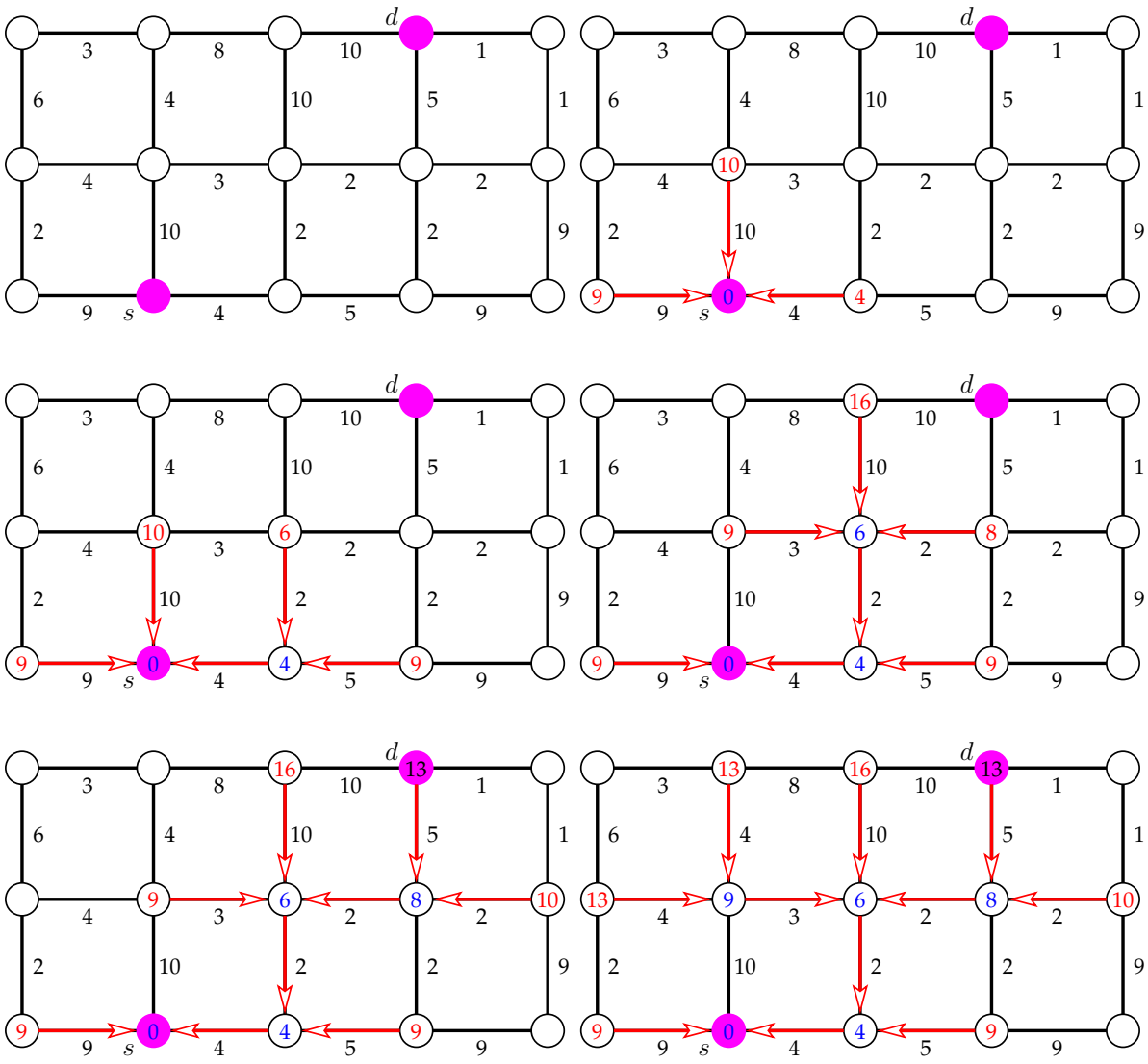
[YAV07] Zhen Yang, S. Areibi, and A. Vannelli. A comparison of ilp based global routing models for vlsi asic design. *Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on*, pages 1141–1144, Aug. 2007.

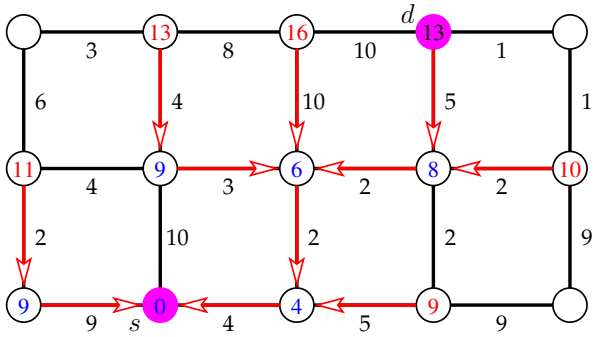
Annexes

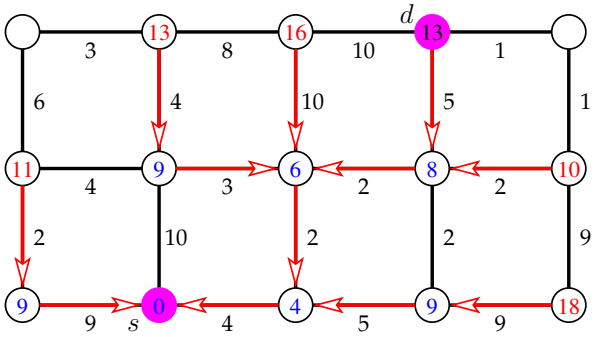
Annexe A
Algorithmes de Dijkstra

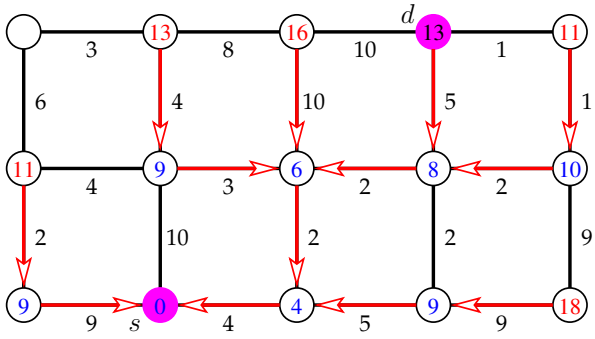
A.1 Source et destination ponctuelles

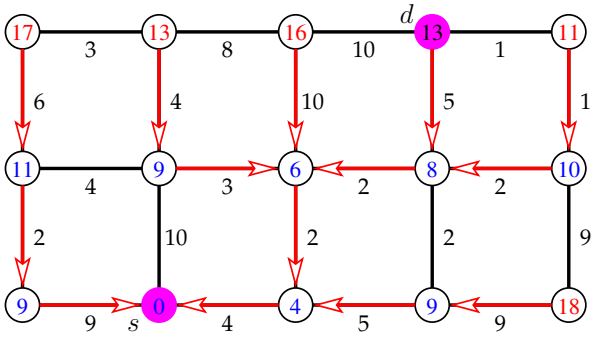
Nous présentons ici un exemple de déroulement de l'algorithme de Dijkstra pour une source et une destination ponctuelles.

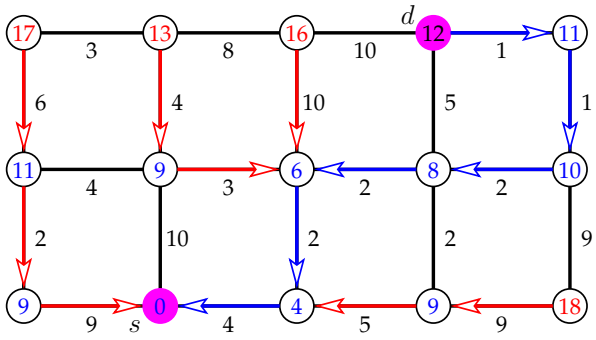



d
13
16
13
1
3
8
10
6
4
10
5
1
11
9
6
8
10
4
3
2
2
2
10
2
2
9
9
0
4
9
9
s
4
5
9


d
13
16
13
1
3
8
10
6
4
10
5
1
11
9
6
8
10
4
3
2
2
2
10
2
2
9
9
0
4
9
18
9
s
4
5
9

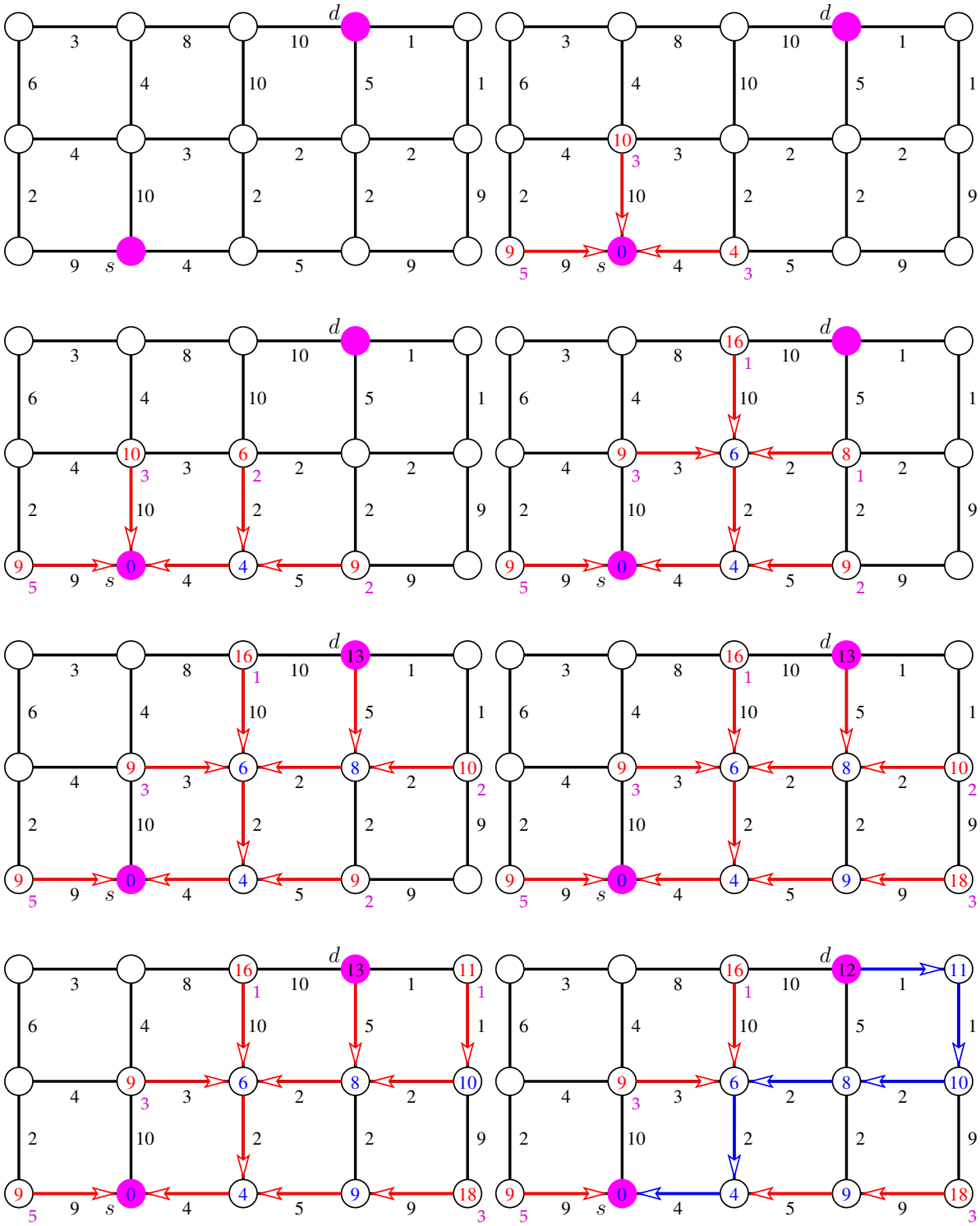

d
13
16
13
11
3
8
10
1
6
4
10
5
1
11
9
6
8
10
4
3
2
2
2
10
2
2
9
9
0
4
9
18
9
s
4
5
9


d
17
13
16
13
11
3
8
10
1
6
4
10
5
1
11
9
6
8
10
4
3
2
2
2
10
2
2
9
9
0
4
9
18
9
s
4
5
9


d
17
13
16
12
11
3
8
10
1
6
4
10
5
1
11
9
6
8
10
4
3
2
2
2
10
2
2
9
9
0
4
9
18
9
s
4
5
9

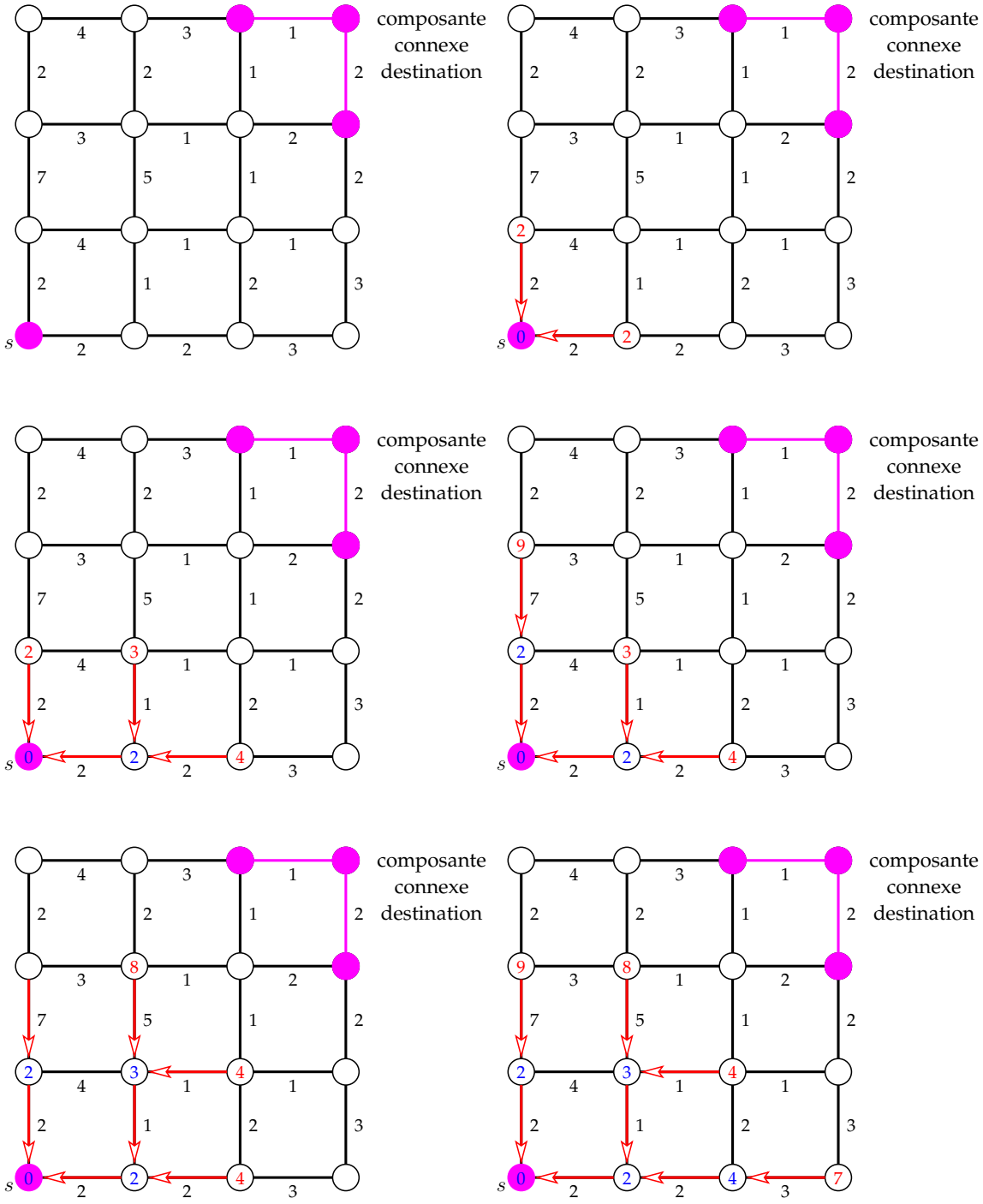
A.2 Variante A*

Nous présentons ici un exemple de déroulement de l'algorithme A*.

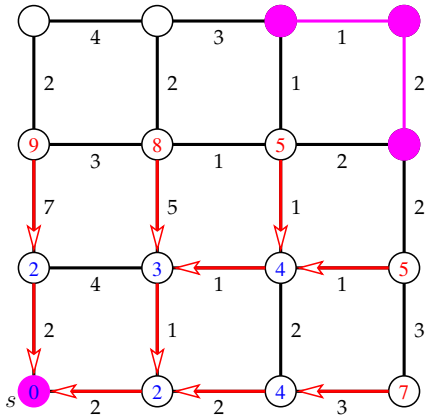


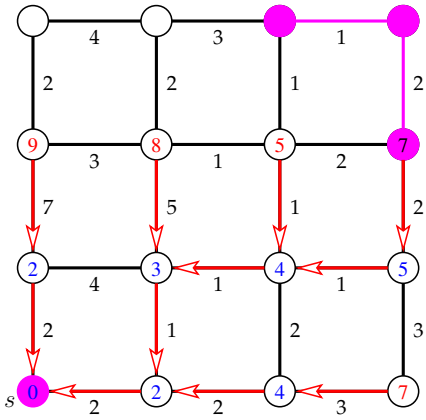
A.3 Source ponctuelle et destination non ponctuelle

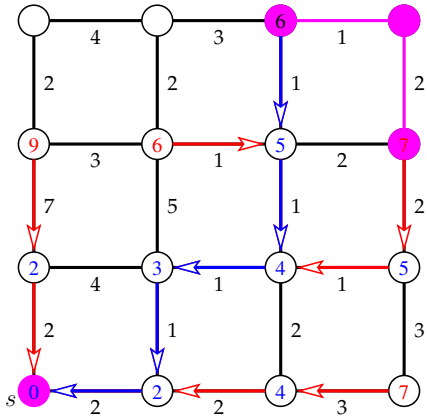
Nous présentons ici un exemple de déroulement de l'algorithme de Dijkstra pour une source ponctuelle et une destination non ponctuelle.



A.3 Source ponctuelle et destination non ponctuelle

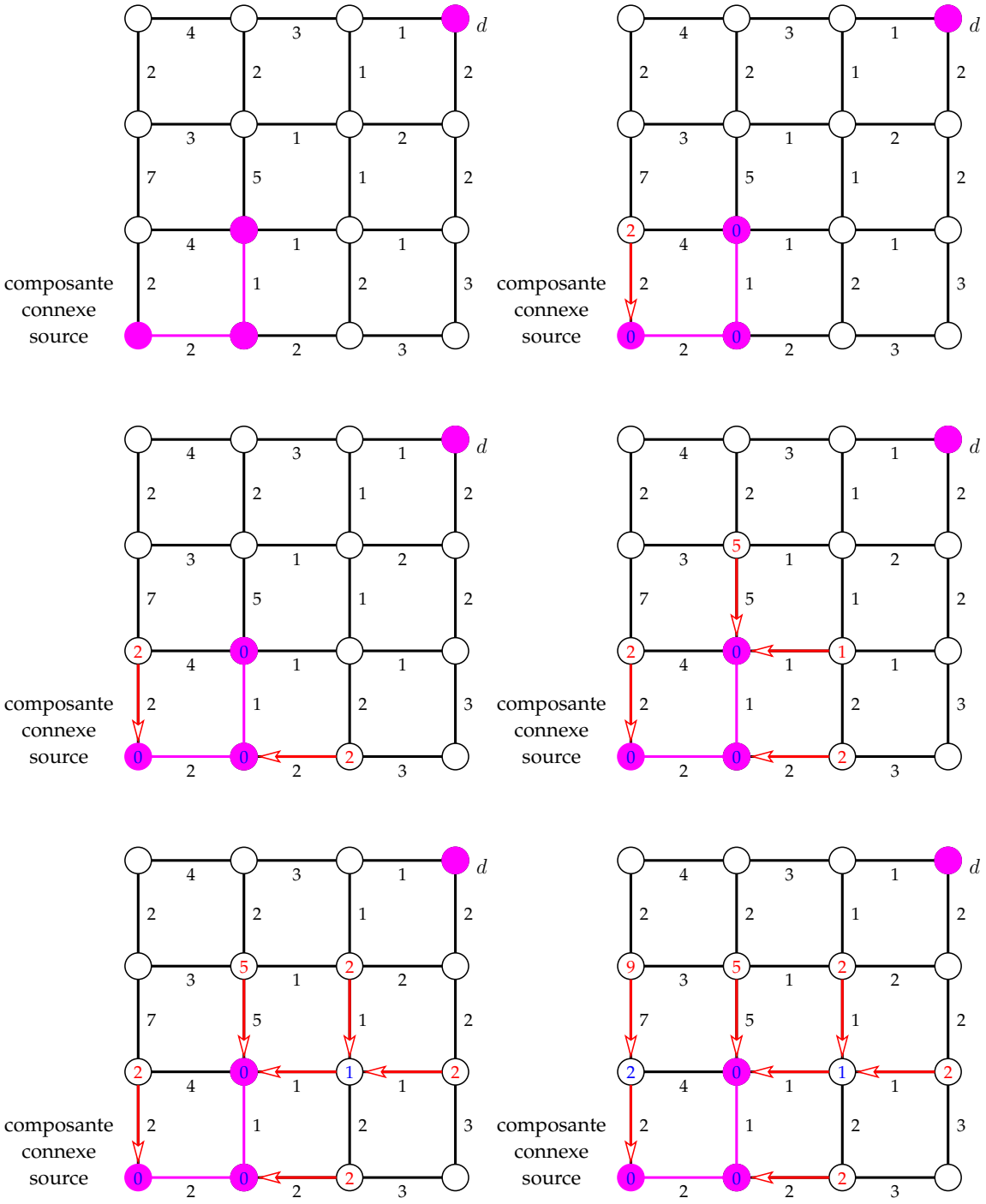




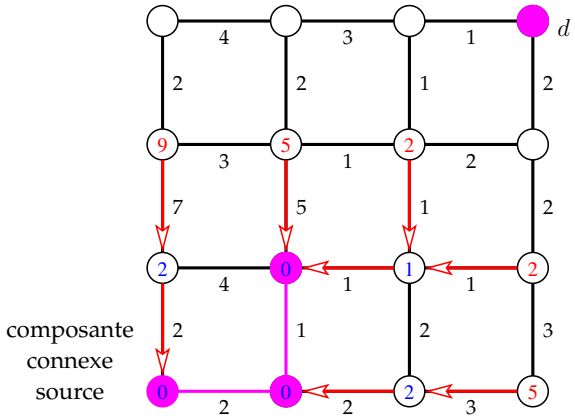


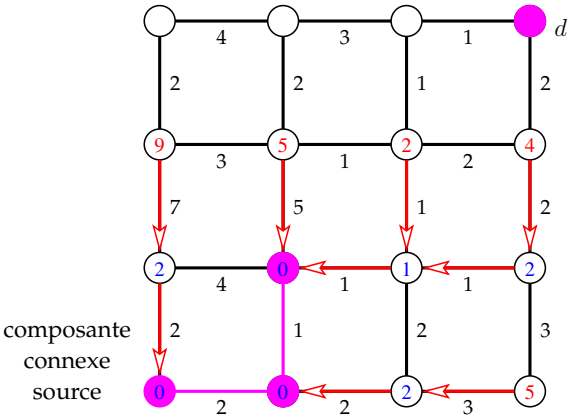
A.4 Source non ponctuelle et destination ponctuelle

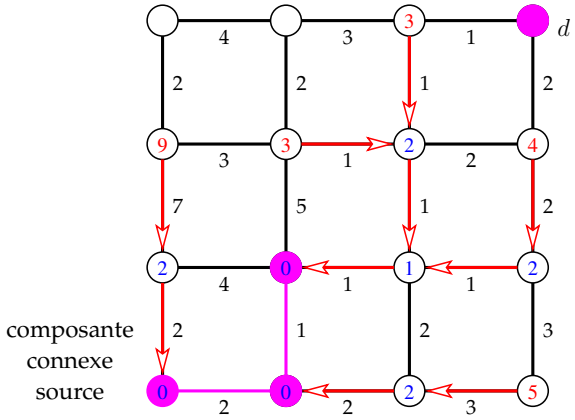
Nous présentons ici un exemple de déroulement de l'algorithme de Dijkstra pour une source non ponctuelle et une destination ponctuelle.

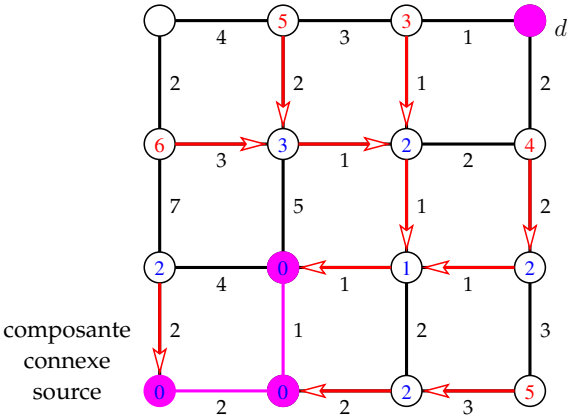


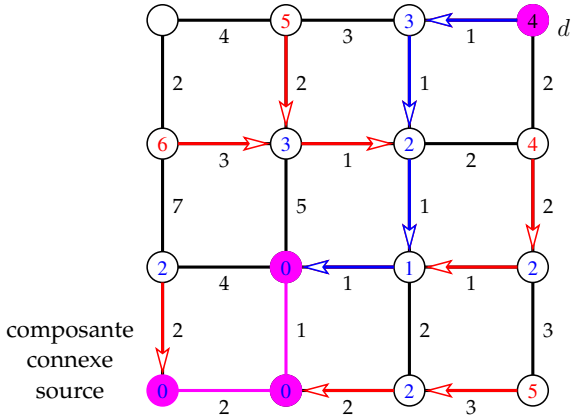
A.4 Source non ponctuelle et destination ponctuelle





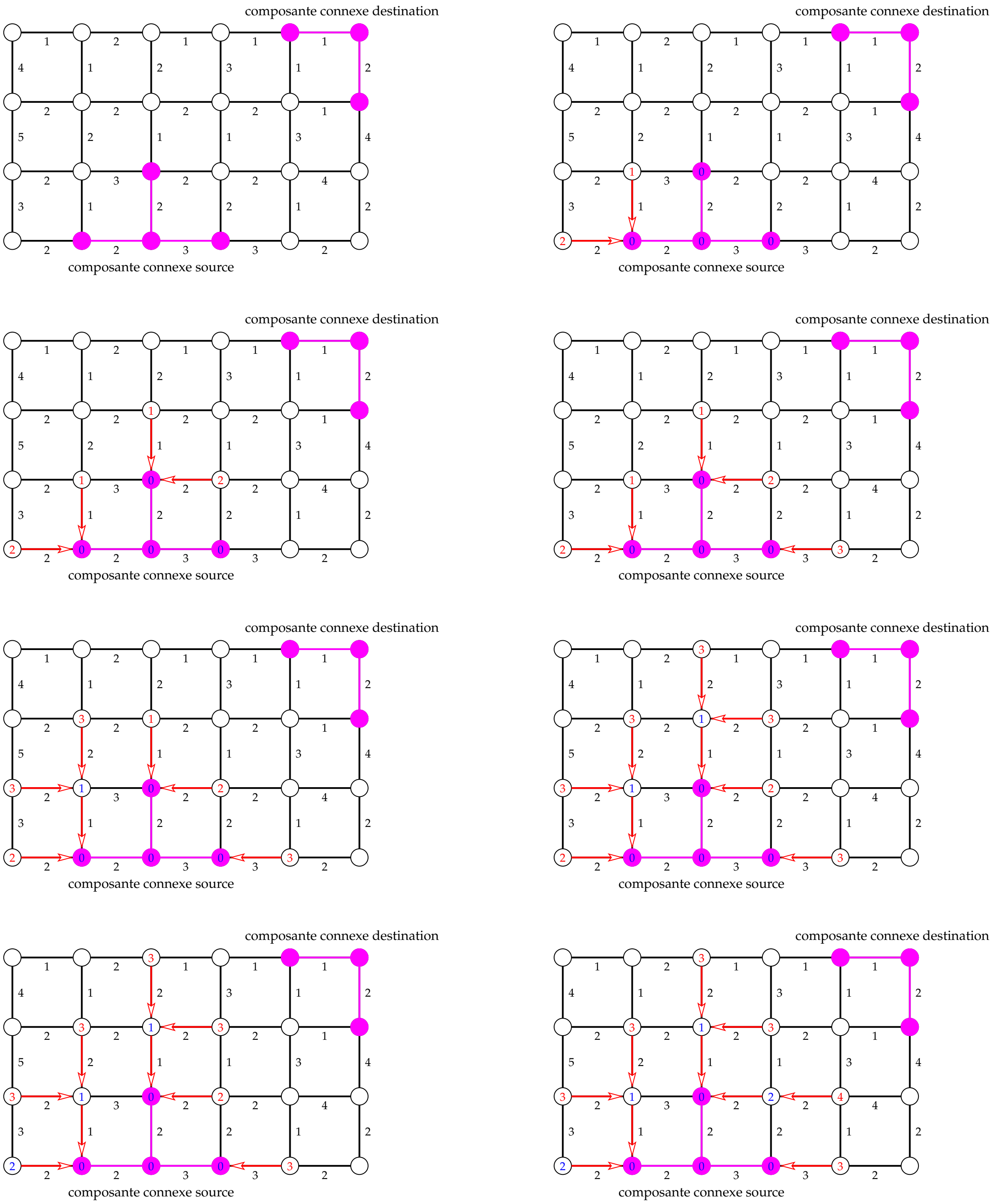




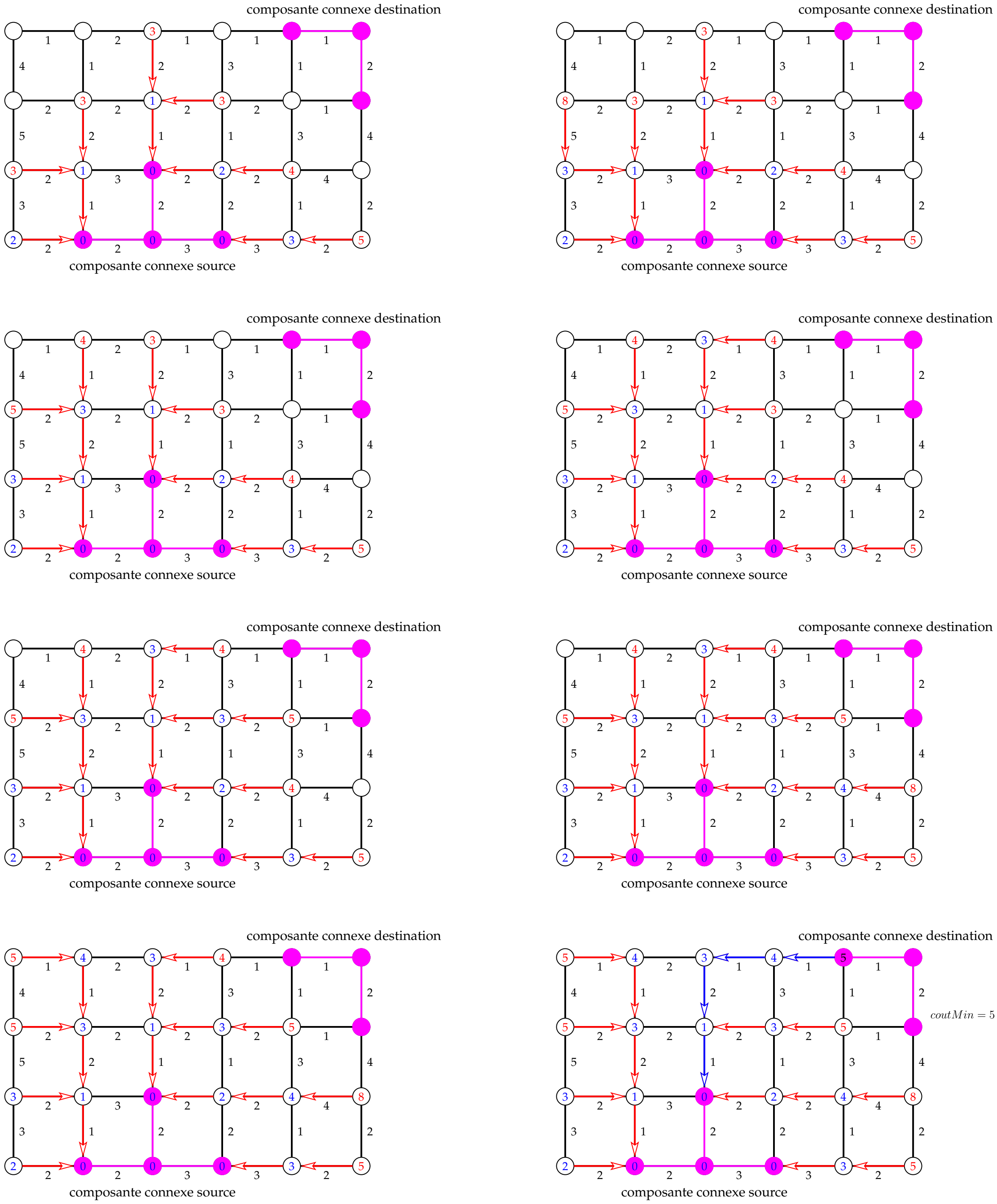


A.5 Source et destination non ponctuelles

Nous présentons ici un exemple de déroulement de l'algorithme de Dijkstra pour une source et une destination non ponctuelles.



A.5 Source et destination non ponctuelles



Annexe B
KNIK routeur global pour la plate-forme CORIOLIS

B.1 Flexibilité de la mise œuvre

Les trois tableaux suivants présentent les relevés du nombre de vias, de la longueur totales des fils d'interconnexion et du dépassement total des solutions de routage global créées par **KNIK** pour les circuits de l'ispd98. Les solutions sont celles construites par l'algorithme de Dijkstra sans *ripup & reroute*.

	Coût des vias						
	1	**2**		**3**		**4**	
ibm01	29296	28153	-3.9%	27358	-6.6%	26800	-8.5%
ibm02	58270	56350	-3.3%	55245	-5.2%	54415	-6.6%
ibm03	52383	49982	-4.6%	48391	-7.6%	47319	-9.7%
ibm04	58147	55888	-3.9%	54507	-6.3%	53649	-7.7%
ibm05	100091	96433	-3.7%	94991	-5.1%	93797	-6.3%
ibm06	94972	90643	-4.6%	87790	-7.6%	86118	-9.3%
ibm07	120647	115343	-4.4%	112263	-6.9%	110182	-8.7%
ibm08	154505	146769	-5.0%	142288	-7.9%	138626	-10.3%
ibm09	144181	138216	-4.1%	133659	-7.3%	130453	-9.5%
ibm10	194555	188324	-3.2%	184183	-5.3%	181323	-6.8%
Total	1007047	966101	-4.1%	940675	-6.6%	922682	-8.4%

TABLE B.1 – Relevé du nombre de vias

Remarque : Pour les trois tableaux, les pourcentages sont exprimés par rapport aux valeurs pour un coût de via égal à 1.

	Coût des vias						
	1	**2**		**3**		**4**	
ibm01	153429	149449	-2.6%	146585	-4.5%	146585	-4.5%
ibm02	354991	348646	-1.8%	344534	-2.9%	341543	-3.8%
ibm03	313955	305861	-2.6%	300615	-4.2%	296774	-5.5%
ibm04	346981	339677	-2.1%	334971	-3.5%	332120	-4.3%
ibm05	728019	716475	-1.6%	712124	-2.2%	708169	-2.7%
ibm06	588992	574342	-2.5%	564275	-4.2%	558275	-5.2%
ibm07	753211	735192	-2.4%	724605	-3.8%	717141	-4.8%
ibm08	906743	881133	-2.8%	865826	-4.5%	853061	-5.9%
ibm09	880204	859302	-2.4%	843318	-4.2%	831756	-5.5%
ibm10	1197925	1177121	-1.5%	1162731	-2.7%	1153011	-3.5%
Total	6224450	6087198	-2.2%	5999584	-3.6%	5938435	-4.6%

TABLE B.2 – Relevé de la longueur totale des fils d'interconnexion

	Coût des vias						
	1	**2**		**3**		**4**	
ibm01	3664	3622	-1.1%	3608	-1.5%	3666	+0.1%
ibm02	29516	29440	-0.3%	29428	-0.3%	29466	-0.2%
ibm03	15844	15668	-1.1%	15656	-1.2%	15746	-0.6%
ibm04	3466	3432	-1.0%	3422	-1.3%	3464	-0.1%
ibm05	0	0	-0.0%	0	-0.0%	4	–
ibm06	62444	62238	-0.3%	62120	-0.5%	62218	-0.4%
ibm07	44476	44150	-0.7%	44116	-0.7%	44362	-0.2%
ibm08	38148	37492	-1.7%	37336	-2.1%	37402	-1.9%
ibm09	126546	125592	-0.8%	125490	-0.8%	125498	-0.8%
ibm10	40612	40610	-0.0%	40604	-0.0%	41090	+1.2%
Total	364716	362244	-0.7%	361780	-0.8%	362916	-0.5%

TABLE B.3 – Relevé du dépassement total

Annexe C Résultats

C.1 Résultats pour algorithme de *ripup & reroute* simple

Les deux tableaux suivants présentent pour les circuits de l'ispd98 et l'ispd07 le relevé du dépassement total pour la solution initiale (itération 0) et vingt itérations successives de *ripup & reroute*. Les parties grisées mettent en évidence les itérations pour lesquelles le dépassement total ne varie plus.

Circuits de l'ispd98

Itération	ibm01	ibm02	ibm03	ibm04	ibm05	ibm06	ibm07	ibm08	ibm09	ibm10
0	1230	800	16	1268	0	226	196	130	254	382
1	1200	776	6	1232	0	214	188	128	244	382
2	1158	740	6	1204	0	214	188	128	244	378
3	1078	702	6	1178	0	214	188	128	244	374
4	1002	684	6	1142	0	212	188	128	244	374
5	942	674	6	1142	0	210	188	128	244	374
6	914	652	6	1138	0	210	178	128	244	374
7	908	616	6	1138	0	210	178	128	244	374
8	830	606	6	1138	0	210	178	128	244	374
9	806	582	6	1138	0	210	178	128	244	374
10	802	570	6	1138	0	210	178	128	244	374
11	800	536	6	1138	0	210	178	128	244	374
12	776	536	6	1138	0	210	178	128	244	374
13	762	530	6	1138	0	210	178	128	244	374
14	762	524	6	1138	0	210	178	128	244	374
15	762	524	6	1138	0	210	178	128	244	374
16	762	524	6	1138	0	210	178	128	244	374
17	762	524	6	1138	0	210	178	128	244	374
18	762	524	6	1138	0	210	178	128	244	374
19	762	524	6	1138	0	210	178	128	244	374
20	762	524	6	1138	0	210	178	128	244	374

TABLE C.1 – Relevé des dépassements totaux pour 20 itérations de *ripup & reroute* pour les circuits de l'ispd98

Remarque : Le circuit *ibm05* n'est pas représentatif puisqu'un routage global sans *ripup & reroute* permet de construire une solution valide.

Circuits de l'ispd07

Itération	adaptec1	adaptec2	adaptec3	adaptec4	adaptec5	newblue1	newblue2	newblue3
0	58720	46626	54684	8124	141414	18924	1370	125806
1	58714	44136	53370	7144	139230	18704	816	125560
2	58714	42008	52218	7048	138130	18086	636	125500
3	58476	40456	51324	6494	136844	18082	534	123880
4	58476	38854	50670	6262	136290	17976	476	123890
5	58476	35596	49990	6020	135770	17976	418	123888
6	58476	32822	49266	5990	134924	17976	368	123866
7	58476	32084	49266	5986	134920	17976	360	123892
8	58476	30240	49266	5986	134912	17976	360	123766
9	58476	29102	49266	5988	134816	17976	356	123790
10	58476	28130	49266	5986	134616	17976	356	123782
11	58476	26658	49266	5992	134586	17976	356	123794
12	58476	26286	49266	5986	134554	17976	356	123798
13	58476	26286	49266	5986	134548	17976	356	123788
14	58476	26286	49266	5988	134566	17976	356	123766
15	58476	26286	49266	5986	134556	17976	356	123762
16	58476	26286	49266	5986	134566	17976	356	123804
17	58476	26286	49266	5986	134572	17976	356	123764
18	58476	26286	49266	5986	134554	17976	356	123814
19	58476	26286	49266	5986	134554	17976	356	123820
20	58476	26286	49266	5986	134330	17976	356	123792

TABLE C.2 – Relevé des dépassements totaux pour 20 itérations de *ripup & reroute* pour les circuits de l'ispd07

Remarque : Du fait de la grande taille des circuist *adaptec5* et *newblue2*, vingt itérations ne suffisent pas à mettre en évidence ce phénomène.

C.2 Etude de la valeur de l'incrément $hInc$

Le tableau ci-dessous présente l'évolution du temps d'exécution (en secondes), de la longueur totale des segments (en pas de grille) et du nombre de vias pour différentes valeurs de l'incrément $hInc$ et pour tous les circuits de l'ispd98.

	Incrément $hInc$	**0.5**	**1**	**1.5**	**2**	**2.5**	**3**
ibm01	Temps d'exécution (s)	13.90	10.36	9.74	9.10	8.62	7.66
	Longueur des segments	64560	64956	65043	65041	65236	65115
	Nombre de vias	23924	24069	24138	24026	24195	24140
ibm02	Temps d'exécution (s)	17.26	15.01	13.54	12.00	11.74	11.44
	Longueur des segments	175567	175381	175271	175520	175556	175353
	Nombre de vias	49150	49156	49079	49083	49194	49141
ibm03	Temps d'exécution (s)	6,08	4,06	4,17	4,54	4,52	4,31
	Longueur des segments	149585	149479	149503	149565	149488	149426
	Nombre de vias	40388	40291	40218	40340	40292	40307
ibm04	Temps d'exécution (s)	76,15	42,70	45,51	42,99	34,28	37,98
	Longueur des segments	171602	171515	171468	171574	171929	172071
	Nombre de vias	49785	49779	49790	49820	49880	50016
ibm05	Temps d'exécution (s)	9,01	9,01	9,01	9,01	9,01	9,01
	Longueur des segments	417927	417927	417927	417927	417927	417927
	Nombre de vias	82370	82370	82370	82370	82370	82370
ibm06	Temps d'exécution (s)	19,17	15,06	16,43	16,74	13,18	13,25
	Longueur des segments	286140	285801	286258	286419	286032	286311
	Nombre de vias	74317	74217	74564	74401	74365	74252
ibm07	Temps d'exécution (s)	21,60	20,70	17,65	15,79	18,93	18,63
	Longueur des segments	374949	375209	375211	375037	375115	375466
	Nombre de vias	93536	93742	93687	93735	93656	93734
ibm08	Temps d'exécution (s)	18,84	21,50	22,98	19,41	19,37	25,05
	Longueur des segments	416574	416302	416228	416486	416567	416600
	Nombre de vias	115229	115226	115227	115275	115203	115229
ibm09	Temps d'exécution (s)	19,97	24,86	19,87	17,99	20,05	19,99
	Longueur des segments	425294	425390	425175	425228	425463	425550
	Nombre de vias	106611	106696	106624	106624	106731	106804
ibm10	Temps d'exécution (s)	75,86	53,01	48,71	38,46	46,73	40,27
	Longueur des segments	591882	592090	592131	592149	592497	592302
	Nombre de vias	160263	160411	160443	160383	160490	160464

TABLE C.3 – Relevé de valeurs pour l'étude de l'incrément de coût historique

Remarque : Le circuit *ibm05* n'est pas représentatif puisqu'un routage global sans *ripup & reroute* permet de construire une solution valide.

C.3 Relevé des résultas

Les tableaux suivants présentent les relevés du temps d'exécution (en secondes), de la longueur totale des segments (en pas de grille) et du nombre de vias des solutions créées par les outils **FGR** et **KNIK** pour les circuits de l'ispd98 et de l'ispd07.

Circuits de l'ispd98

	FGR	KNIK	KNIK vs. FGR
ibm01	13.66	8.24	-39.7%
ibm02	19.78	12.19	-38.3%
ibm03	3.94	4.20	+6.5%
ibm04	48.24	34.57	-28.3%
ibm05	5.33	8.60	+61.4%
ibm06	19.08	15.29	-19.9%
ibm07	18.88	13.53	-28.3%
ibm08	16.58	15.86	-4.3%
ibm09	21.33	15.59	-26.9%
ibm10	100.83	45.64	-54.7%
Total	267.65	173.70	-35.1%

TABLE C.4 – Temps d'exécution de FGR et **KNIK** pour les circuits de l'ispd98

	FGR	KNIK	KNIK vs. FGR
ibm01	64059	64741	+1.1%
ibm02	172305	175355	+1.8%
ibm03	149255	149367	+0.1%
ibm04	169495	171419	+1.1%
ibm05	417917	417960	+0.0%
ibm06	283994	285959	+0.7%
ibm07	372854	375168	+0.6%
ibm08	414246	416253	+0.5%
ibm09	421481	425358	+0.9%
ibm10	589556	592060	+0.4%
Total	3055162	3073640	+0.6%

TABLE C.5 – Longueurs des segments de FGR et **KNIK** pour les circuits de l'ispd98

C.3 Relevé des résultas

	FGR	KNIK	KNIK vs. FGR
ibm01	20762	24028	+15.7%
ibm02	42367	49167	+16.1%
ibm03	35751	40354	+12.9%
ibm04	43579	49705	+14.1%
ibm05	72918	82228	+12.8%
ibm06	65469	74279	+13.5%
ibm07	82927	93737	+13.0%
ibm08	101312	115271	+13.8%
ibm09	95216	106587	+11.9%
ibm10	139738	160460	+14.8%
Total	700039	795816	+13.7%

TABLE C.6 – Nombre de vias créés par FGR et **KNIK** pour les circuits de l'ispd98

Circuits de l'ispd07

	FGR	KNIK	KNIK vs. FGR
adaptec1	22673	4760	-79.0%
adaptec2	2093	59472	+2741.5%
adaptec3	11021	5125	-53.5%
adaptec4	823	1119	+36.0%
adaptec5	45725	32678	-28.5%
newblue1	86400	86400	0.0%
newblue2	336	283	-15.8%
newblue3	86400	86400	0.0%
Total	82671	103437	+25.1%

TABLE C.7 – Temps d'exécution de FGR et **KNIK** pour les circuits de l'ispd07

Remarque : Les circuits *newblue1* et *newblue3* n'entrent pas dans le calcul du total puisque aucune solution valide n'a pu être construite.

	FGR	KNIK	KNIK vs. FGR
adaptec1	3583817	3697354	+3.2%
adaptec2	3316202	3434075	+3.6%
adaptec3	9591656	9854361	+2.7%
adaptec4	8995548	9062071	+0.7%
adaptec5	10260240	10722362	+4.5%
newblue1	2413785	2480338	+2.8%
newblue2	4677656	4710562	+0.7%
newblue3	7527899	8252162	+9.6%
Total	50366803	52213285	+3.7%

TABLE C.8 – Longueurs des segments de FGR et **KNIK** pour les circuits de l'ispd07

	FGR	KNIK	KNIK vs. FGR
adaptec1	611327	715980	+17.1%
adaptec2	632791	739451	+16.9%
adaptec3	1148904	1380260	+20.1%
adaptec4	1164124	1336263	+14.8%
adaptec5	1627423	1977516	+21.5%
newblue1	773181	837033	+8.3%
newblue2	987621	1144001	+15.8%
newblue3	1112763	1171510	+5.3%
Total	8058134	9302014	+15.4%

TABLE C.9 – Nombre de vias créés par FGR et **KNIK** pour les circuits de l'ispd07

C.4 Evolution du dépassement total

Nous présentons ci-dessous l'évolution du dépassement total au fur et à mesure des itérations de *ripup & reroute* pour les outils **FGR** et **KNIK** sur les circuits de l'ispd98.

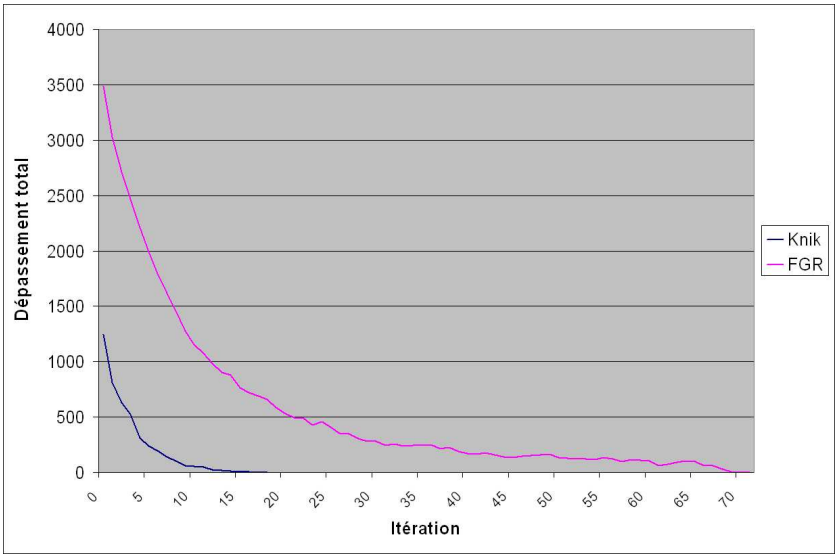


FIGURE C.1 – Dépassement total *ibm01*

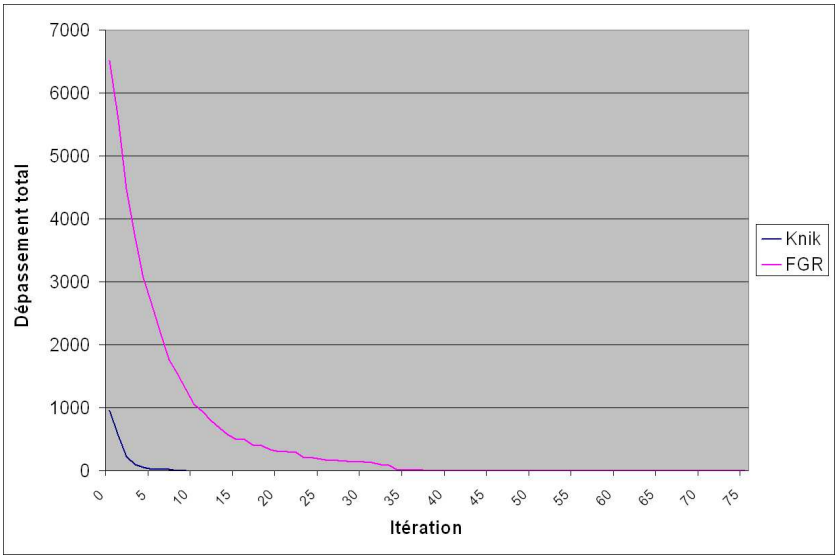


FIGURE C.2 – Dépassement total *ibm02*

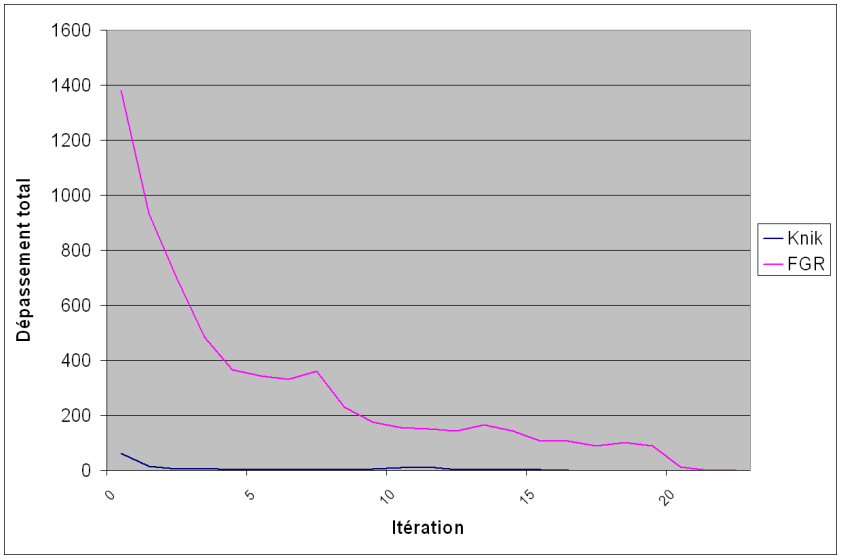


FIGURE C.3 – Dépassement total *ibm03*

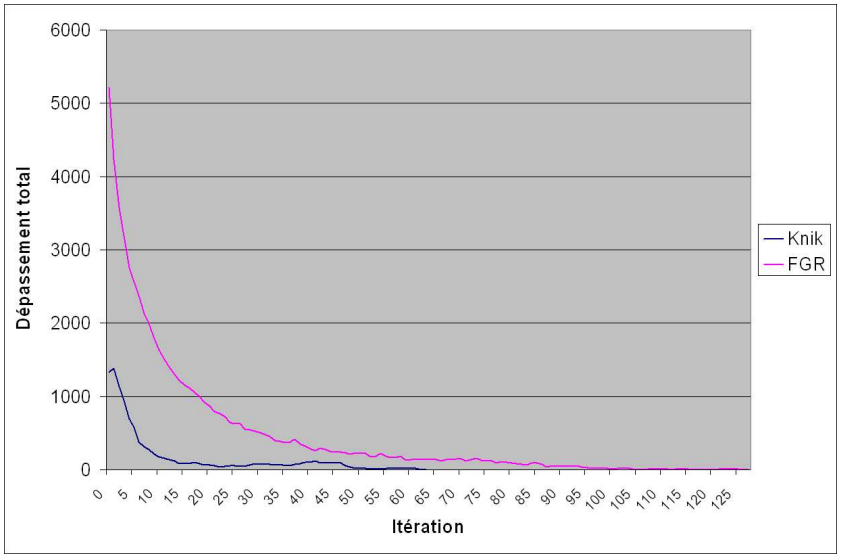


FIGURE C.4 – Dépassement total *ibm04*

C.4 Evolution du dépassement total

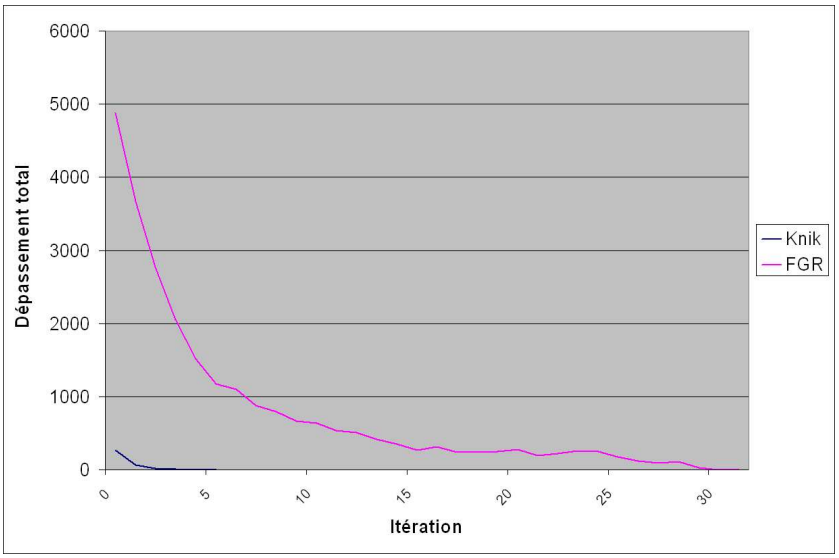


FIGURE C.5 – Dépassement total *ibm06*

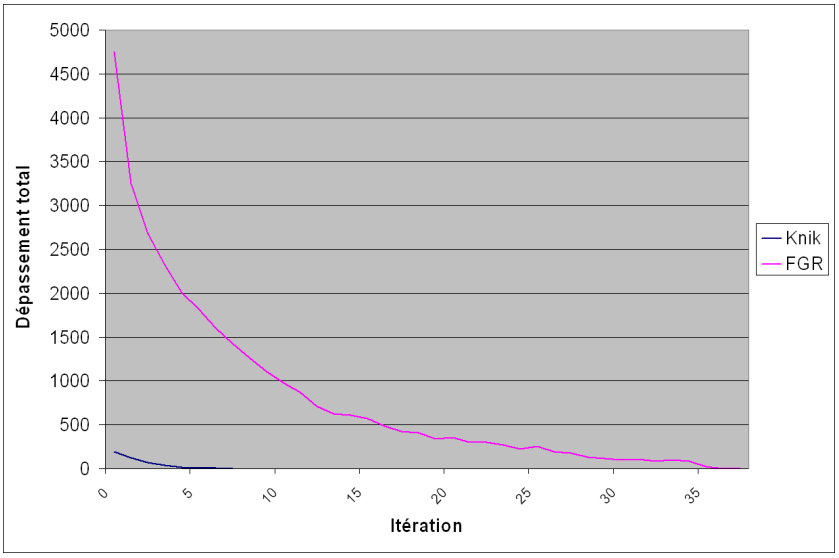


FIGURE C.6 – Dépassement total *ibm07*

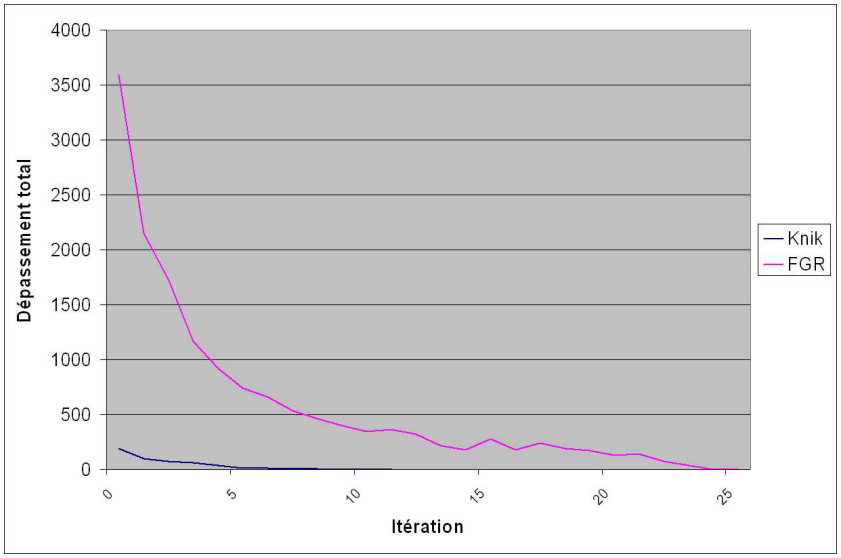


FIGURE C.7 – Dépassement total *ibm08*

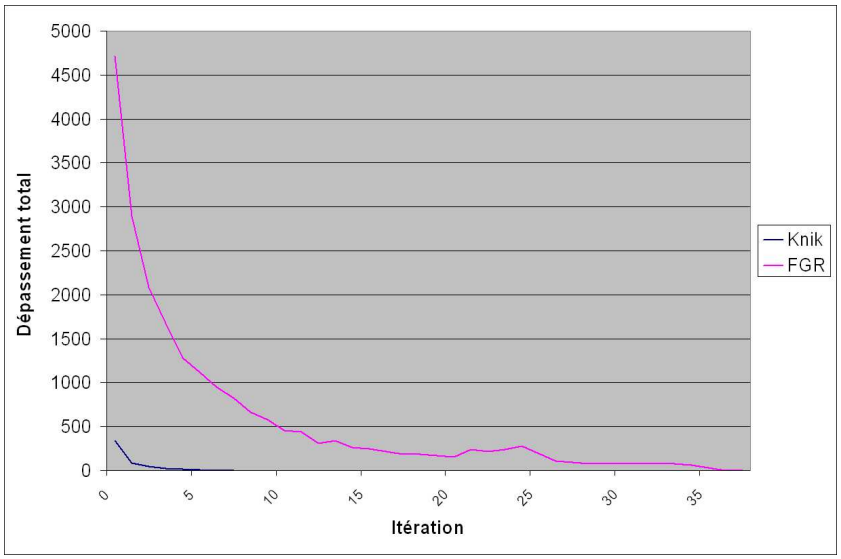


FIGURE C.8 – Dépassement total *ibm09*

C.4 Evolution du dépassement total

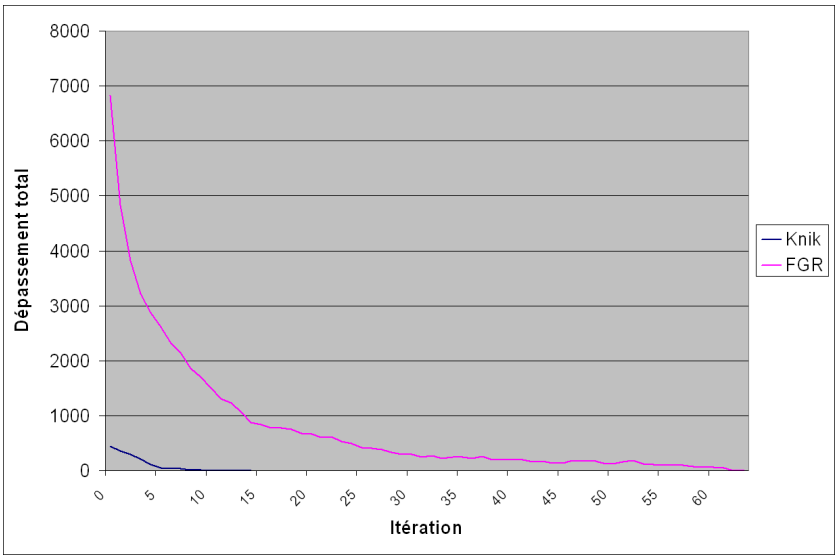


FIGURE C.9 – Dépassement total *ibm10*

C.5 Répartition des arêtes

Les deux tableaux suivants présentent la répartition des arêtes en fonction de leur taux de congestion pour les solutions créées par **FGR** et **KNIK** sur les circuits de l'ispd98 et l'ispd07.

		0-0.1	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8	0.8-0.9	0.9-1	Total
ibm01	**FGR**	594	425	636	559	1164	570	559	903	616	2038	8064
	KNIK	605	385	600	449	1126	603	581	947	798	1970	8064
ibm02	**FGR**	298	477	995	1054	1456	904	782	829	939	2362	10096
	KNIK	288	503	908	927	1360	819	798	973	1140	2380	10096
ibm03	**FGR**	673	660	871	1005	992	1027	1109	972	1077	1710	10096
	KNIK	674	699	897	867	933	969	1040	1176	1546	1295	10096
ibm04	**FGR**	514	455	600	1022	1102	1260	1685	1393	1470	2627	12128
	KNIK	508	411	604	908	1002	1099	1754	1670	1842	2330	12128
ibm05	**FGR**	770	975	1389	2183	3055	2848	2432	1424	717	399	16192
	KNIK	776	986	1308	2041	2714	2949	3394	1671	326	27	16192
ibm06	**FGR**	569	538	832	1438	1727	1823	2249	1837	1697	3482	16192
	KNIK	540	547	857	1299	1551	1769	2139	2024	2428	3038	16192
ibm07	**FGR**	885	1678	2560	3595	3518	2764	2692	1763	1830	3035	24320
	KNIK	975	1788	2541	3297	3310	2523	2792	2109	2381	2604	24320
ibm08	**FGR**	558	690	1284	1953	2931	3156	3262	3001	2776	4709	24320
	KNIK	509	712	1259	1907	2712	2813	3229	3527	3555	4097	24320
ibm09	**FGR**	1190	1203	2867	2565	4521	2780	3746	4717	3388	5471	32448
	KNIK	1373	1238	2710	2201	4038	2644	3851	5657	4318	4418	32448
ibm10	**FGR**	1264	2647	3606	3650	4461	4344	3037	3071	2610	3758	32448
	KNIK	1298	2599	3574	3366	4192	4160	3270	3547	3259	3183	32448
%	**FGR**	3.93%	5.23%	8.39%	10.21%	13.38%	11.53%	11.57%	10.69%	9.19%	15.88%	100%
	KNIK	4.05%	5.30%	8.19%	9.27%	12.31%	10.92%	12.26%	12.51%	11.59%	13.60%	100%

TABLE C.10 – Répartition des arêtes suivant leur taux de congestion pour les circuits de l'ispd98

C.5 Répartition des arêtes

		0-0.1	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8	0.8-0.9	0.9-1	>1	Total
adaptec1	**FGR**	30379	14504	17259	13080	19679	16339	17272	16908	23423	40461	0	209304
	KNIK	29317	13187	15451	11432	18135	15729	17544	18750	29075	40684	0	209304
adaptec2	**FGR**	183140	44791	22060	19851	14573	14569	15319	8732	13656	22013	0	358704
	KNIK	184469	39275	20101	18726	14732	14813	17097	11056	19106	19329	0	358704
adaptec3	**FGR**	549944	106340	97148	59073	54276	67556	51190	63265	53608	101939	0	1204339
	KNIK	555065	92501	89800	54808	52798	65329	50907	71788	71436	99907	0	1204339
adaptec4	**FGR**	470762	161670	145891	94273	79506	75634	51035	48733	33298	43537	0	1204339
	KNIK	469181	151103	140113	94336	83088	81254	58455	59924	41123	25762	0	1204339
adaptec5	**FGR**	98398	46649	39642	32019	30165	25385	29300	23088	34480	75181	0	434307
	KNIK	95633	41159	35413	29226	28979	24027	30114	25884	50794	73078	0	434307
newblue1	**FGR**	171004	24996	17501	14447	15309	15570	13946	13042	10848	20772	169	317604
	KNIK	174820	19290	16610	12863	13286	13763	13110	14381	15728	23101	652	317604
newblue2	**FGR**	301254	34359	34068	28823	28585	22563	22290	12663	14147	16010	0	514762
	KNIK	299360	33090	32578	28676	29363	23875	26206	16363	18591	6660	0	514762
newblue3	**FGR**	1981524	92128	69765	61345	58202	50804	45418	27561	26600	26453	2147	2441947
	KNIK	1985990	89864	64752	55025	50628	46201	48066	33361	35986	28226	3848	2441947
%	**FGR**	56.64%	7.86%	6.63%	4.83%	4.49%	4.31%	3.68%	3.20%	3.14%	5.18%	0.03%	100%
	KNIK	56.75%	7.17%	6.20%	4.56%	4.35%	4.26%	3.91%	3.76%	4.22%	4.74%	0.07%	100%

TABLE C.11 – Répartition des arêtes suivant leur taux de congestion pour les circuits de l'ispd07

C.6 Cartes de congestion

Dans cette section nous présentons toutes les cartes de congestion des solutions créées par **FGR** et **KNIK** sur les circuits de l'ispd98 et l'ispd07.

De façon à pouvoir facilement comparer les cartes de congestion des deux outils pour un même circuit, nous présentons pour chaque circuit la carte de congestion de la solution créée par **FGR** sur la page de gauche et celle de **KNIK** sur la page de droite.

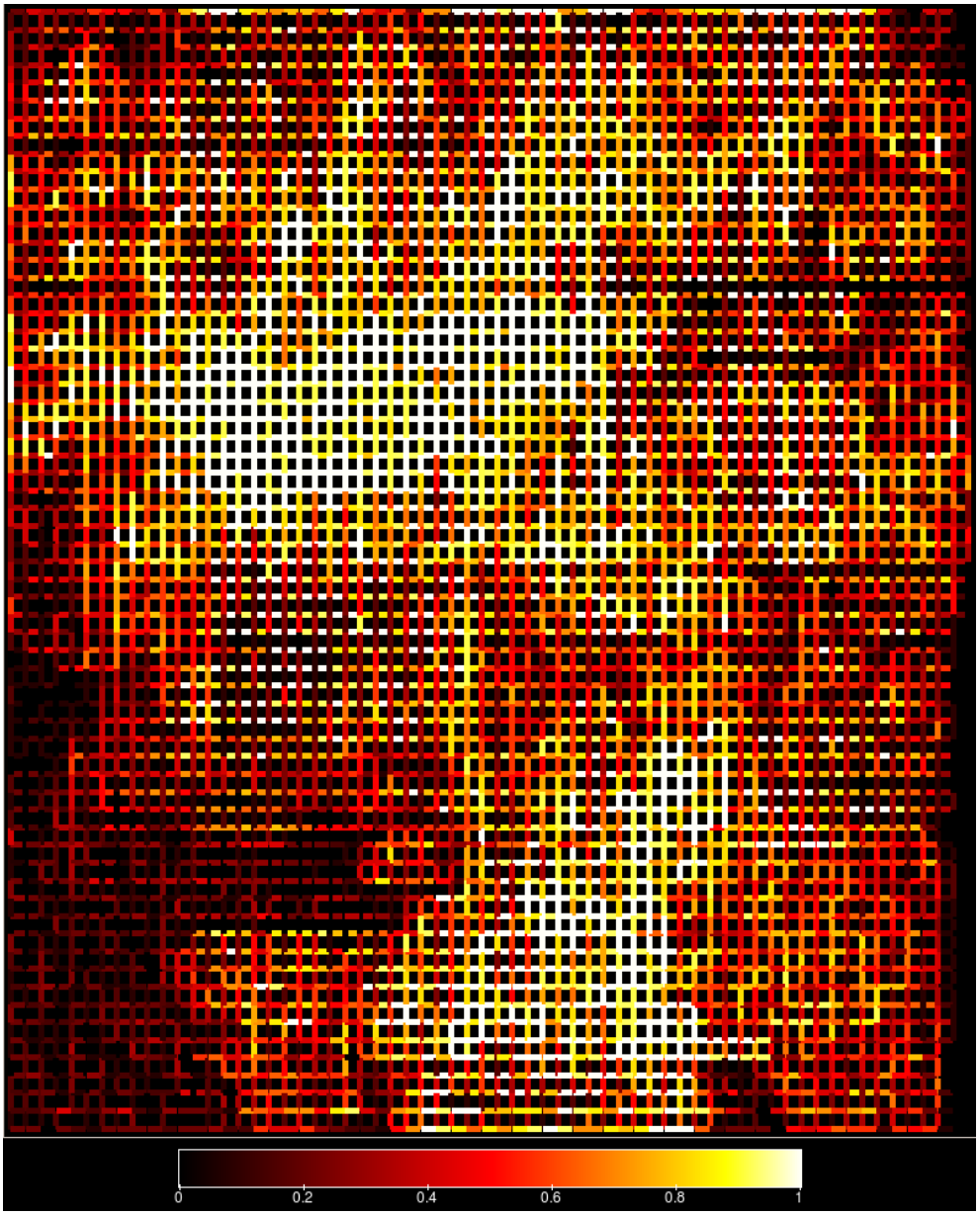


FIGURE C.10 – Carte de congestion du circuit *ibm01* routé par **FGR**

C.6 Cartes de congestion

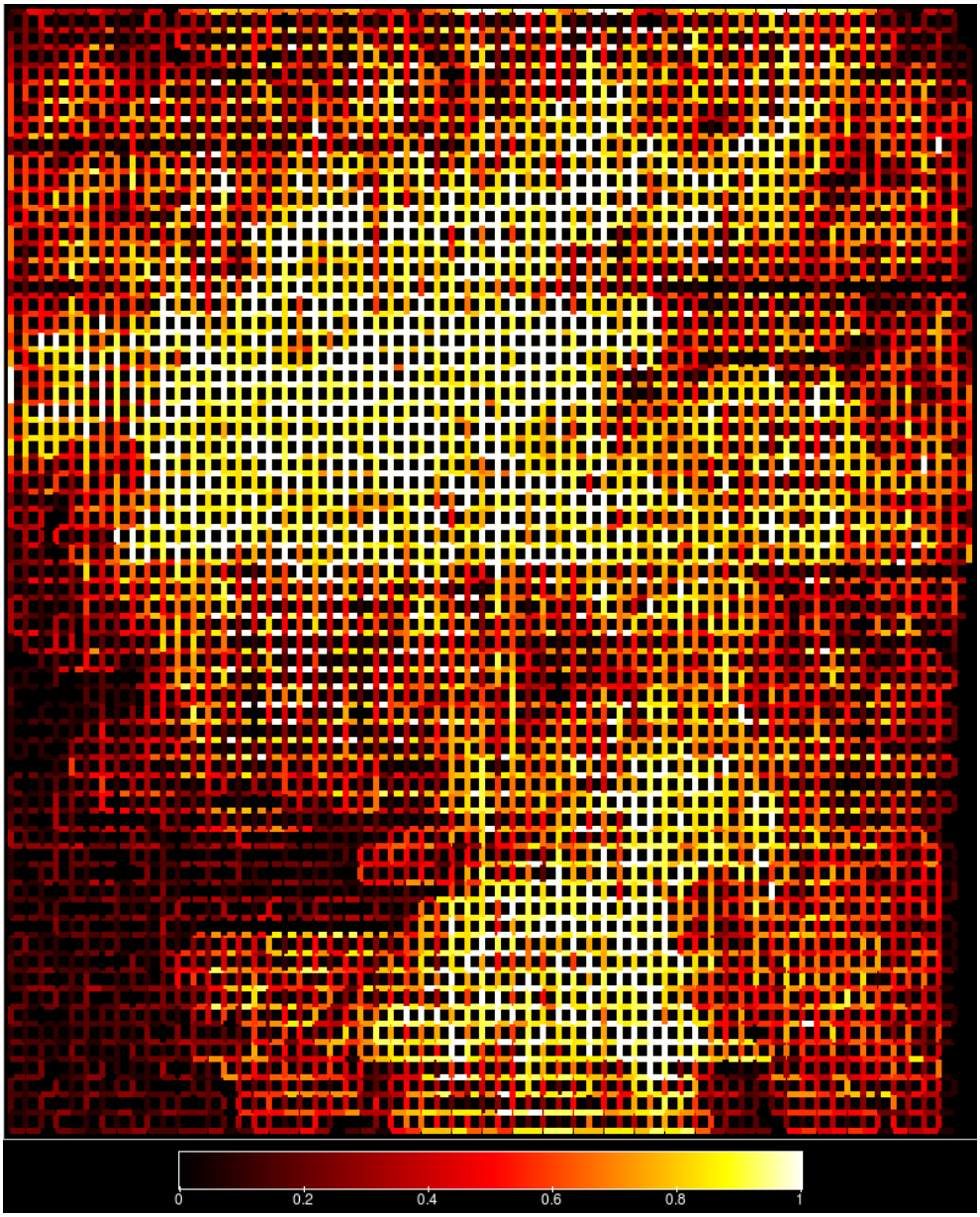


FIGURE C.11 – Carte de congestion du circuit *ibm01* routé par **KNIK**

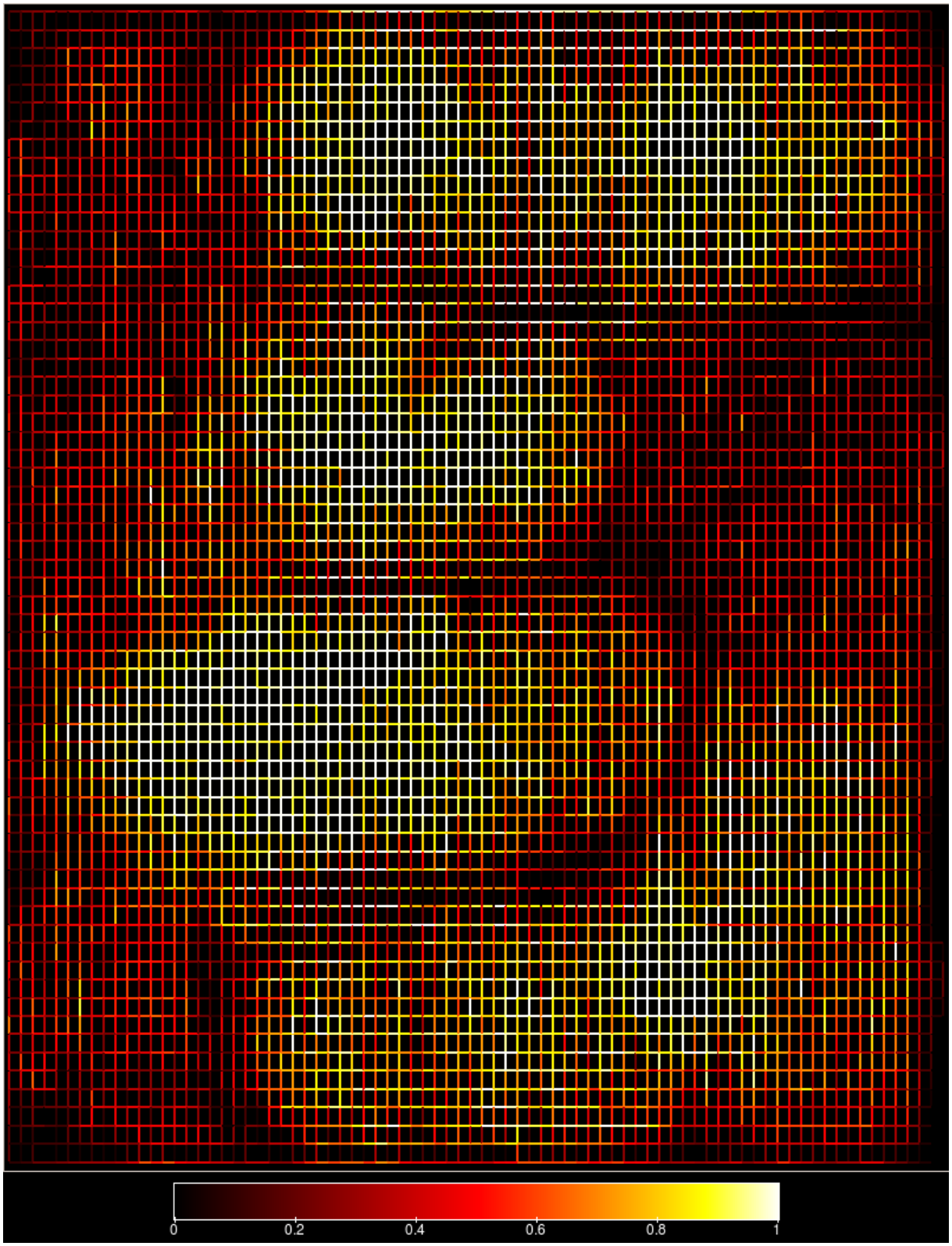


FIGURE C.12 – Carte de congestion du circuit *ibm02* routé par **FGR**

C.6 Cartes de congestion

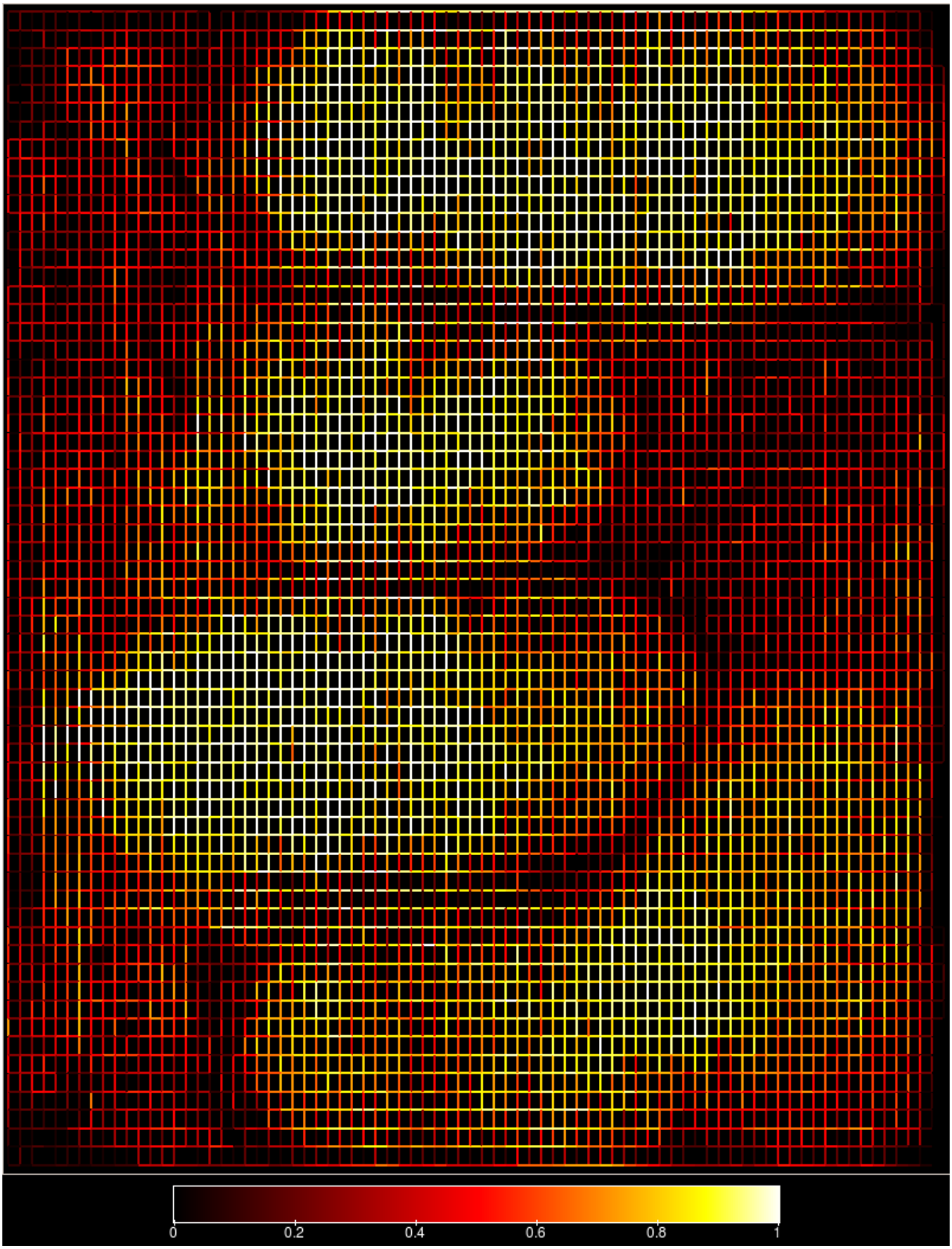


FIGURE C.13 – Carte de congestion du circuit *ibm02* routé par **KNIK**

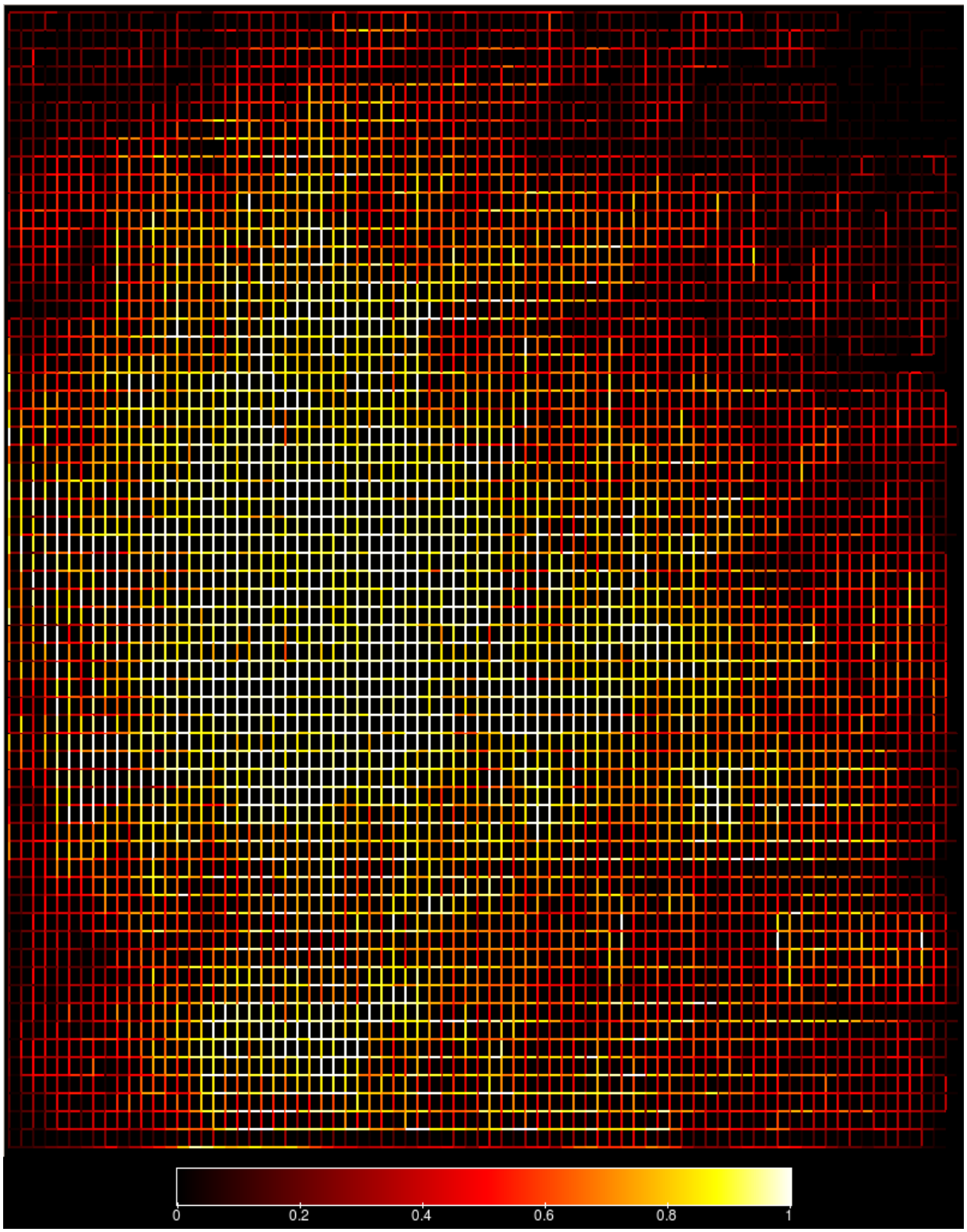


FIGURE C.14 – Carte de congestion du circuit *ibm03* routé par **FGR**

C.6 Cartes de congestion

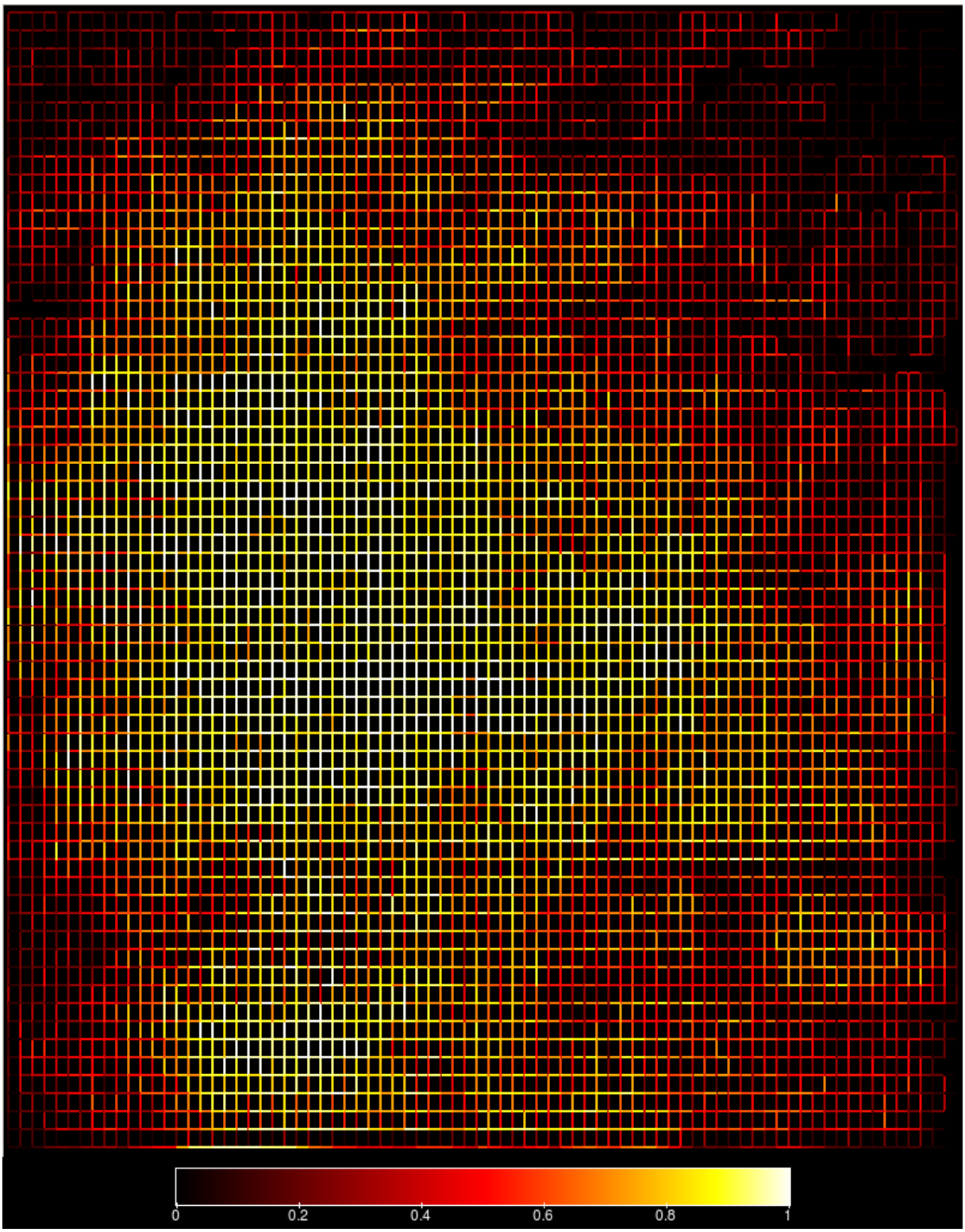


FIGURE C.15 – Carte de congestion du circuit *ibm03* routé par **KNIK**

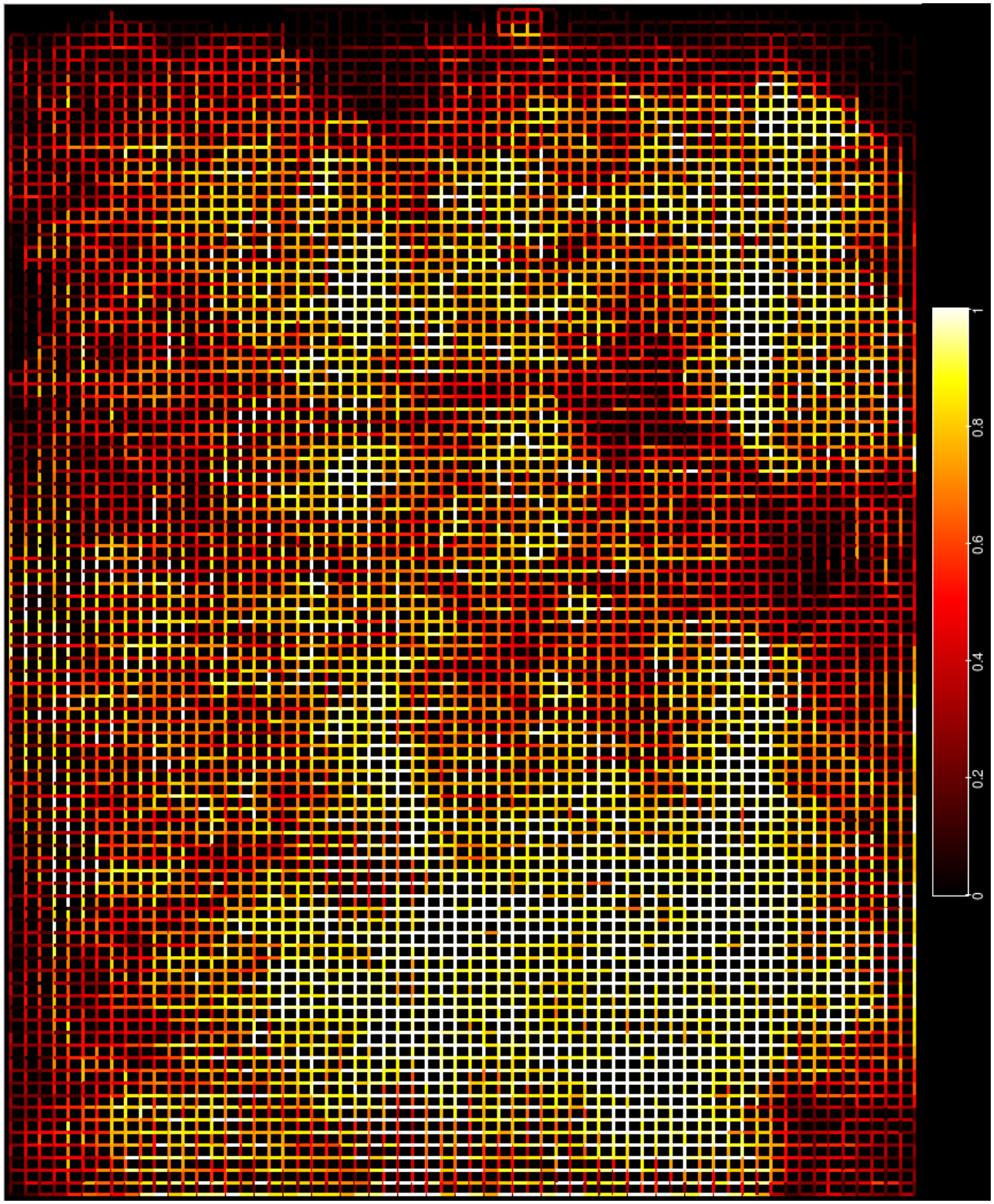


FIGURE C.16 – Carte de congestion du circuit *ibm04* routé par **FGR**

C.6 Cartes de congestion

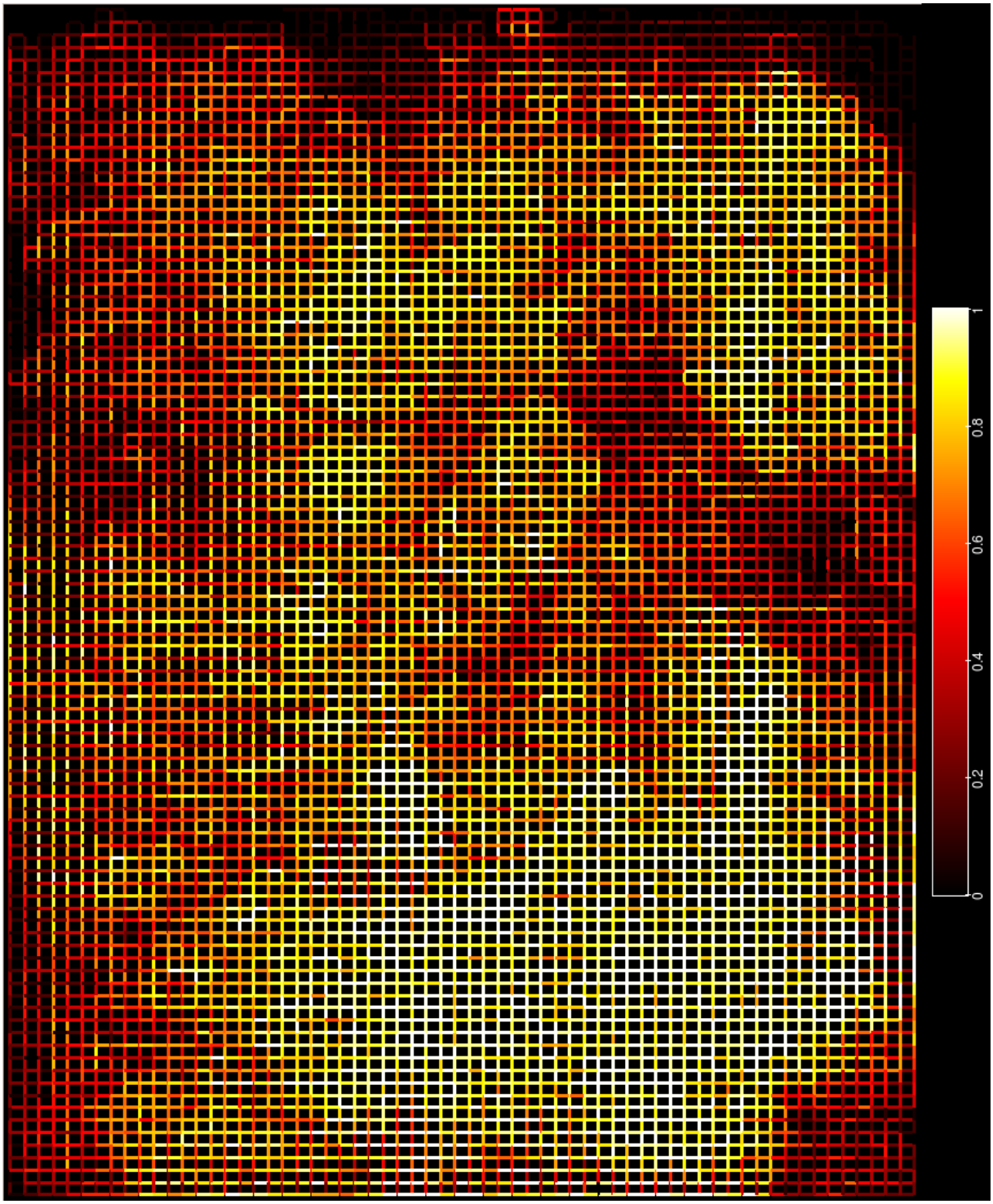


FIGURE C.17 – Carte de congestion du circuit *ibm04* routé par **KNIK**

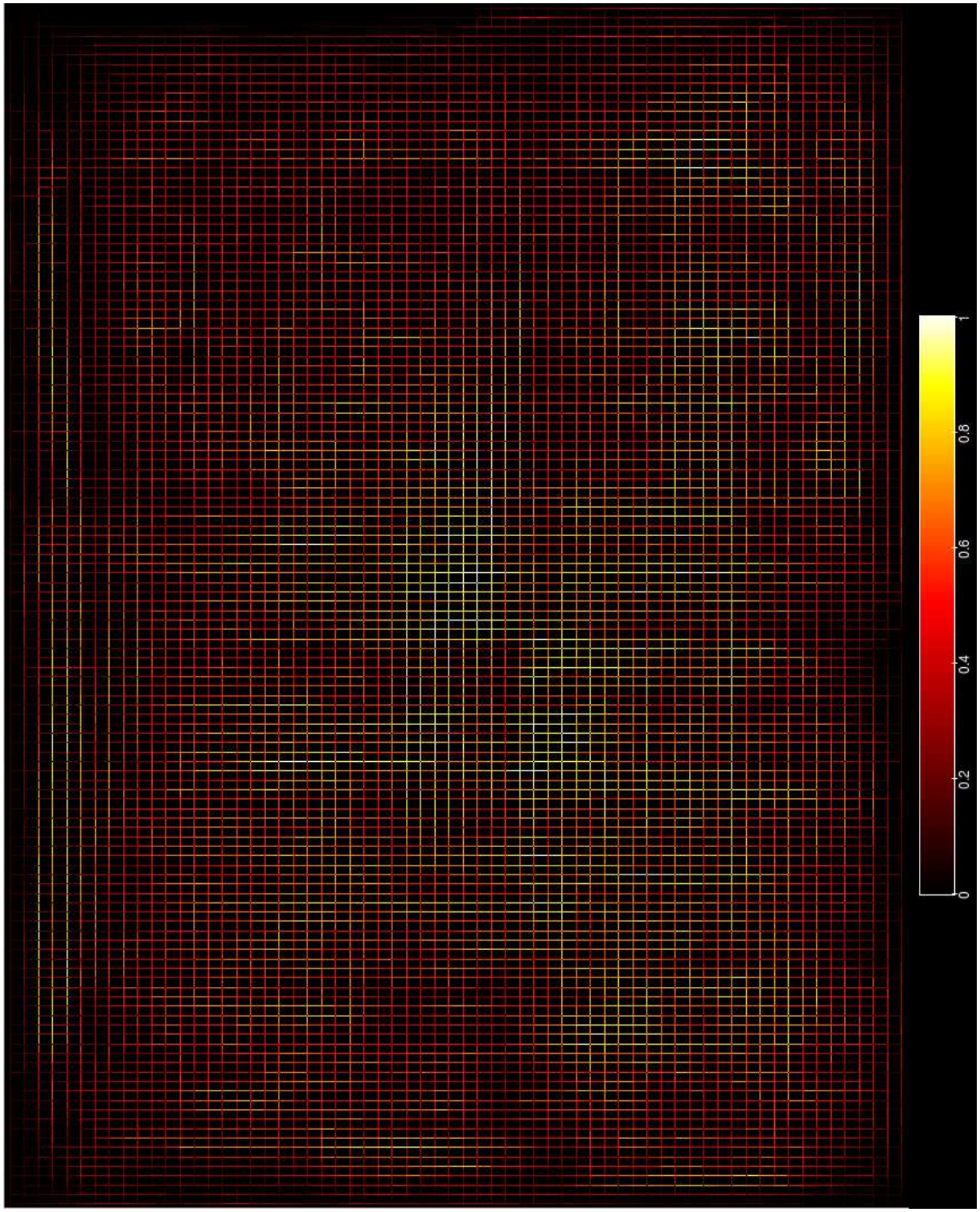


FIGURE C.18 – Carte de congestion du circuit *ibm05* routé par **FGR**

C.6 Cartes de congestion

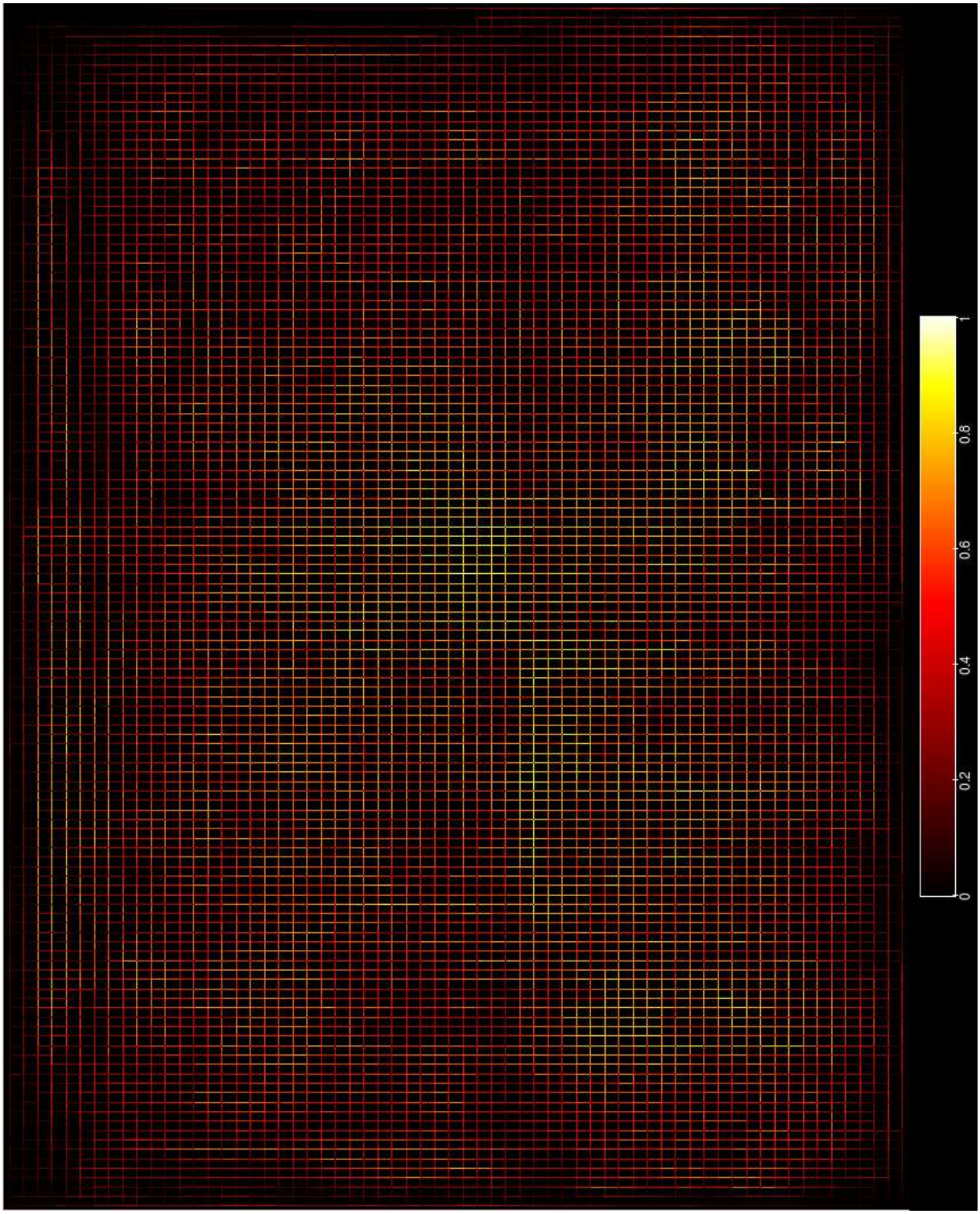


FIGURE C.19 – Carte de congestion du circuit *ibm05* routé par **KNIK**

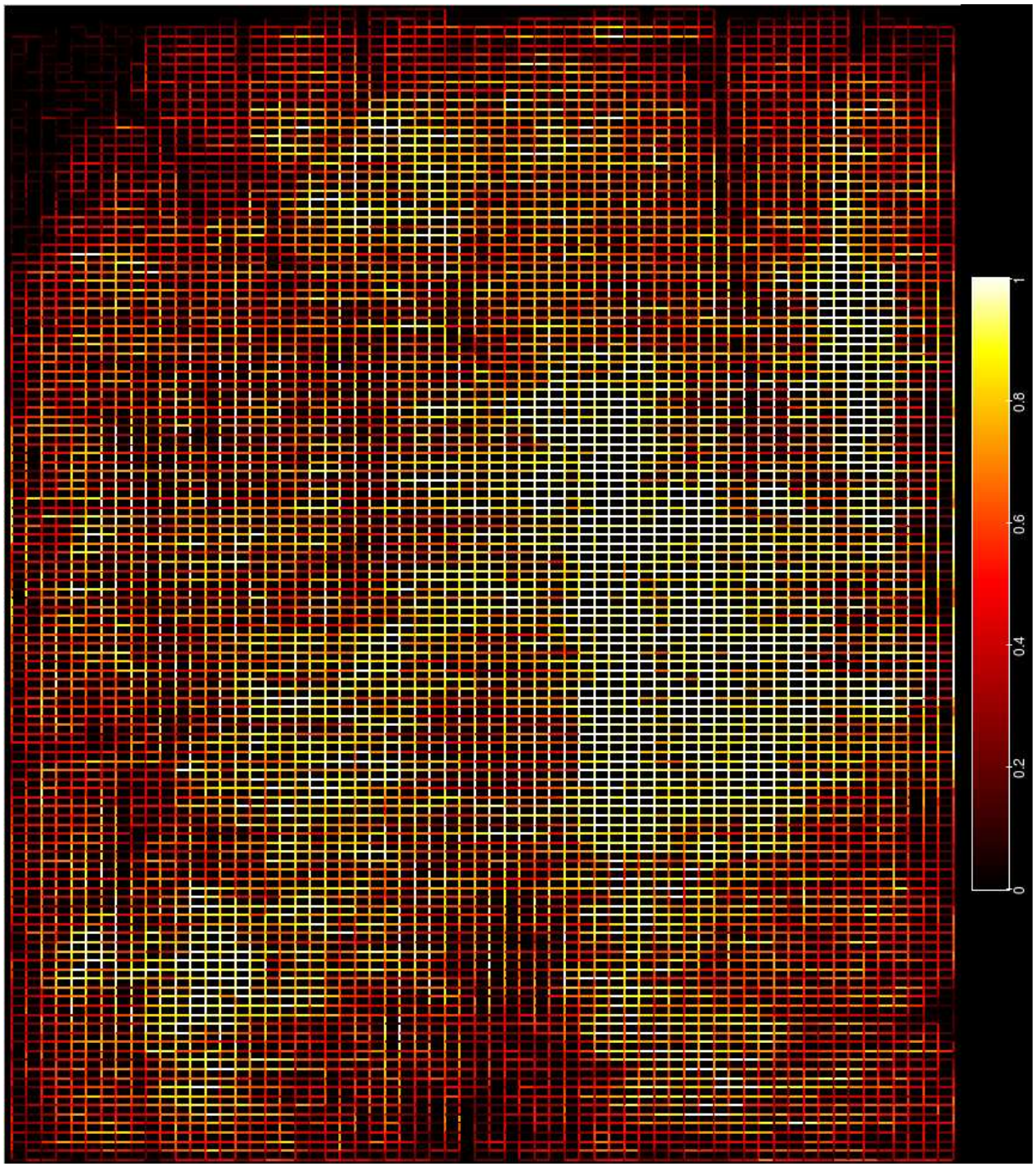


FIGURE C.20 – Carte de congestion du circuit *ibm06* routé par **FGR**

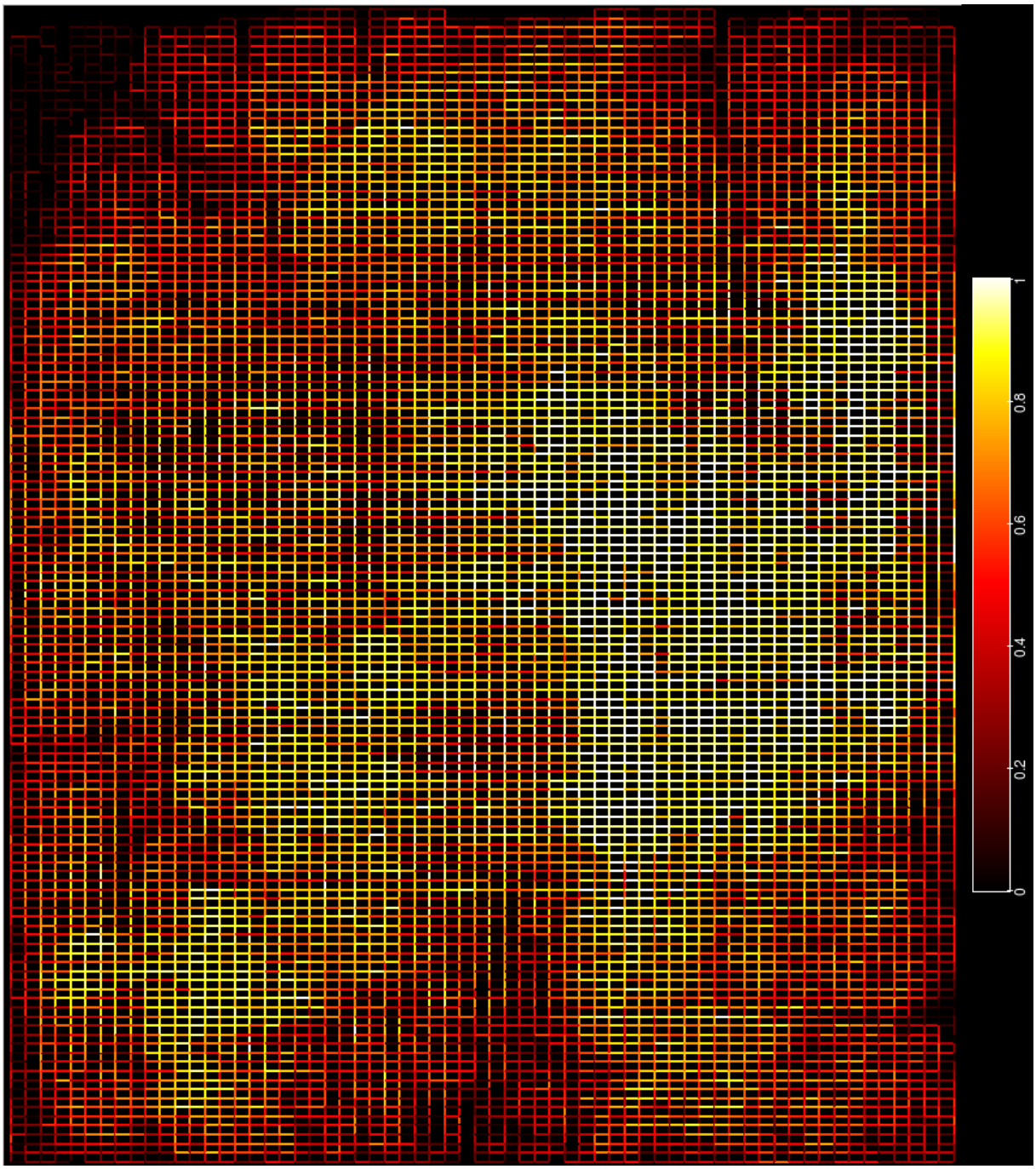


FIGURE C.21 – Carte de congestion du circuit *ibm06* routé par **KNIK**

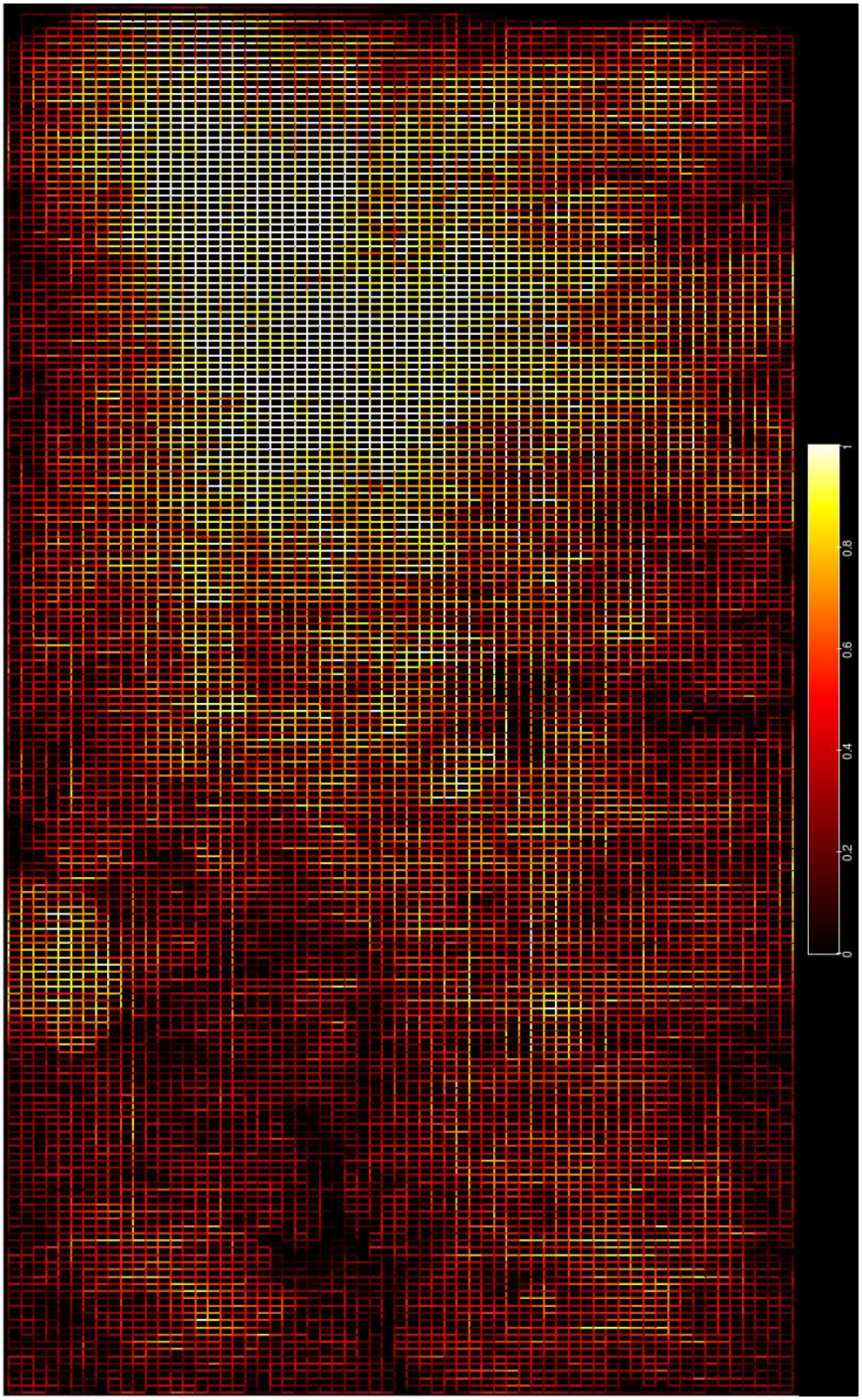


FIGURE C.22 – Carte de congestion du circuit *ibm07* routé par **FGR**

C.6 Cartes de congestion

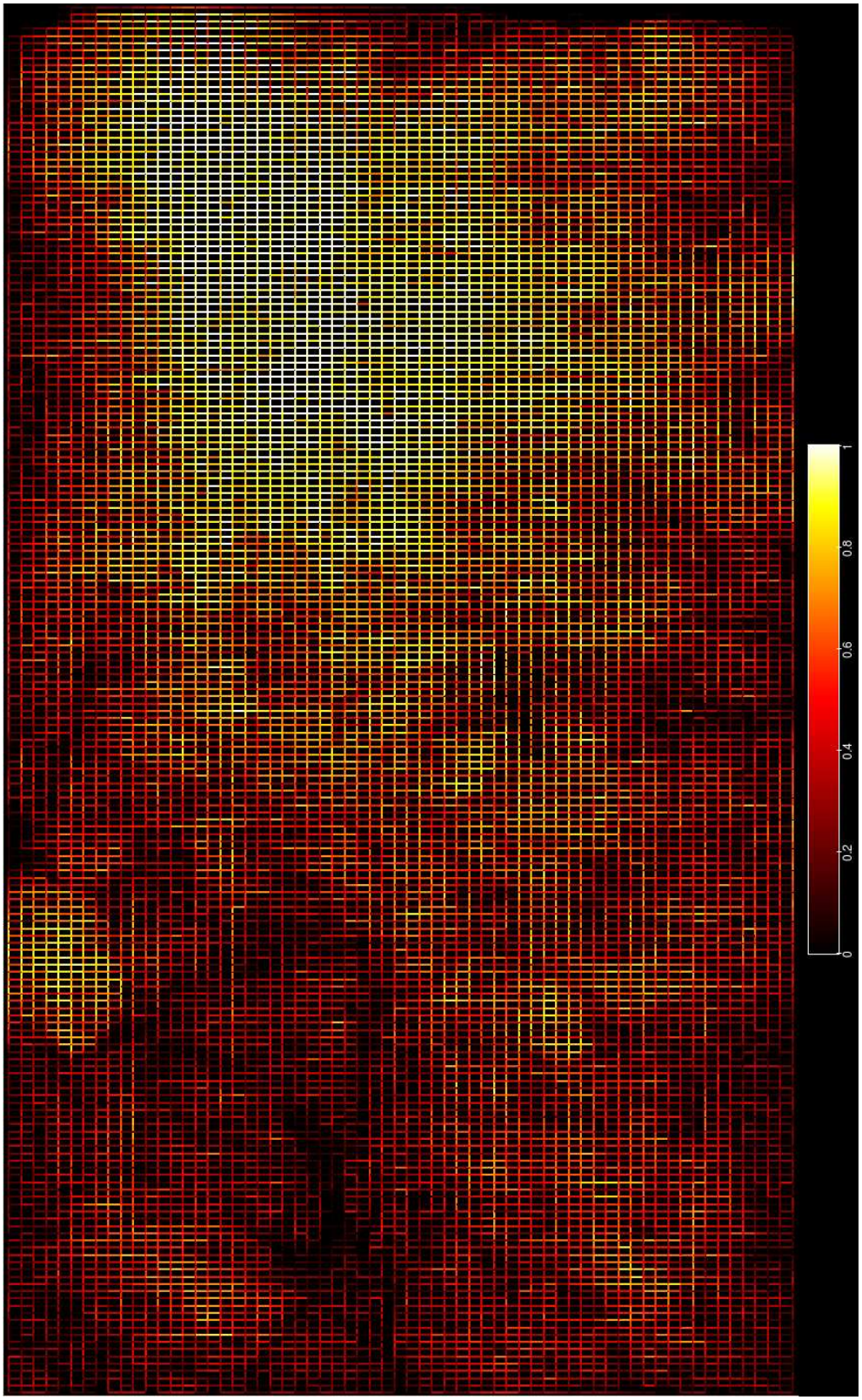


FIGURE C.23 – Carte de congestion du circuit *ibm07* routé par **KNIK**

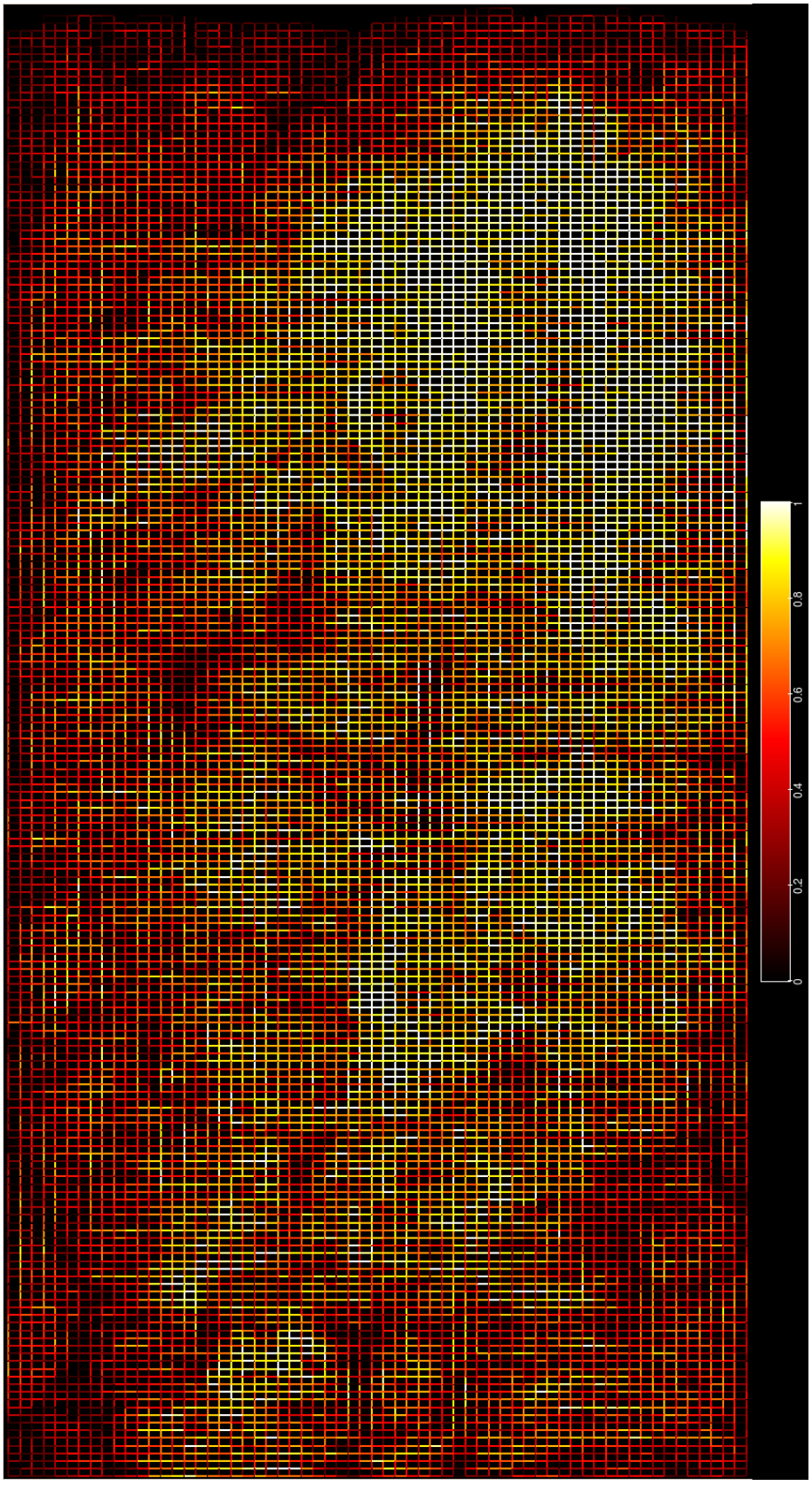


FIGURE C.24 – Carte de congestion du circuit *ibm08* routé par **FGR**

C.6 Cartes de congestion

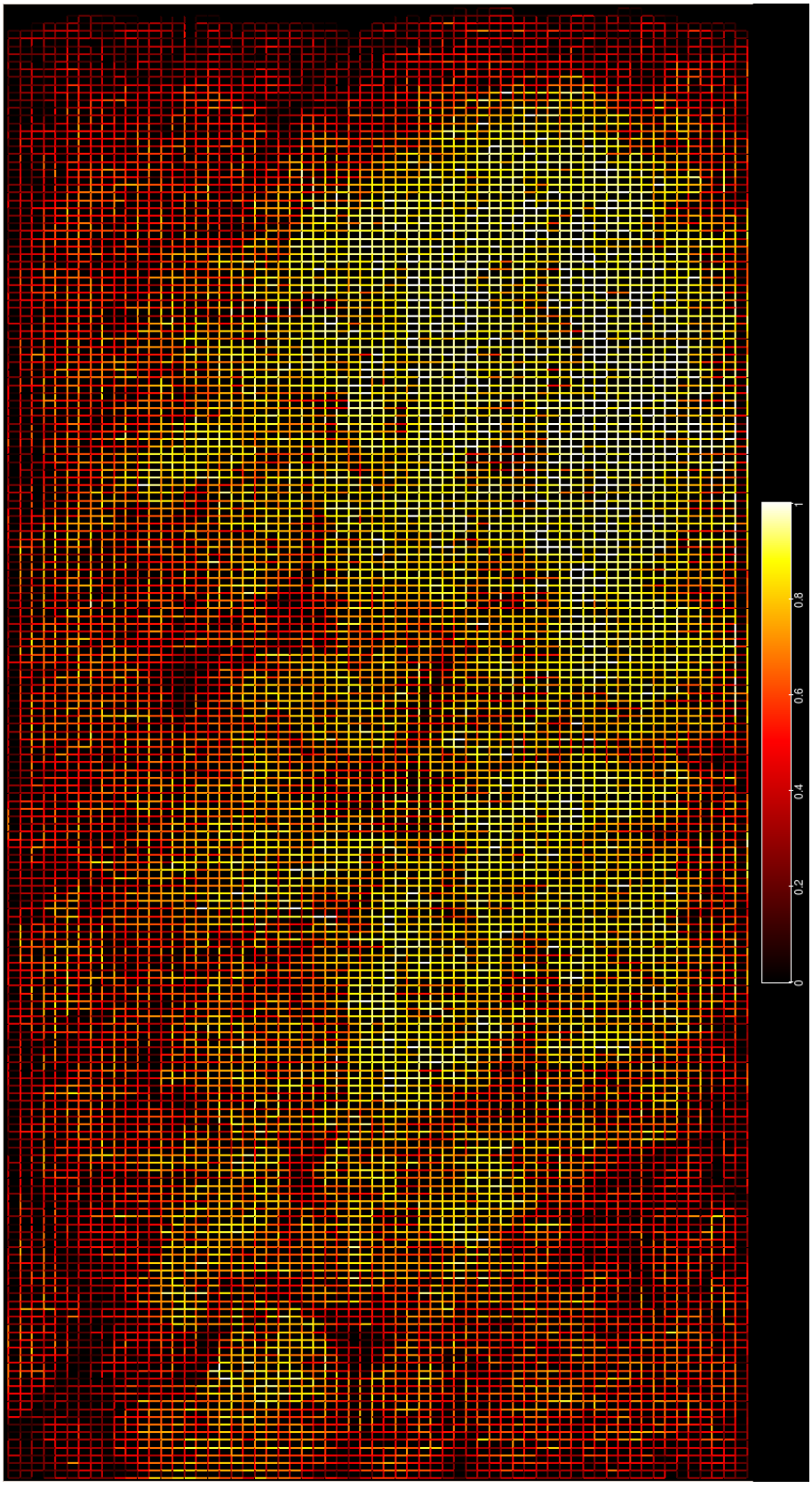


FIGURE C.25 – Carte de congestion du circuit *ibm08* routé par **KNIK**

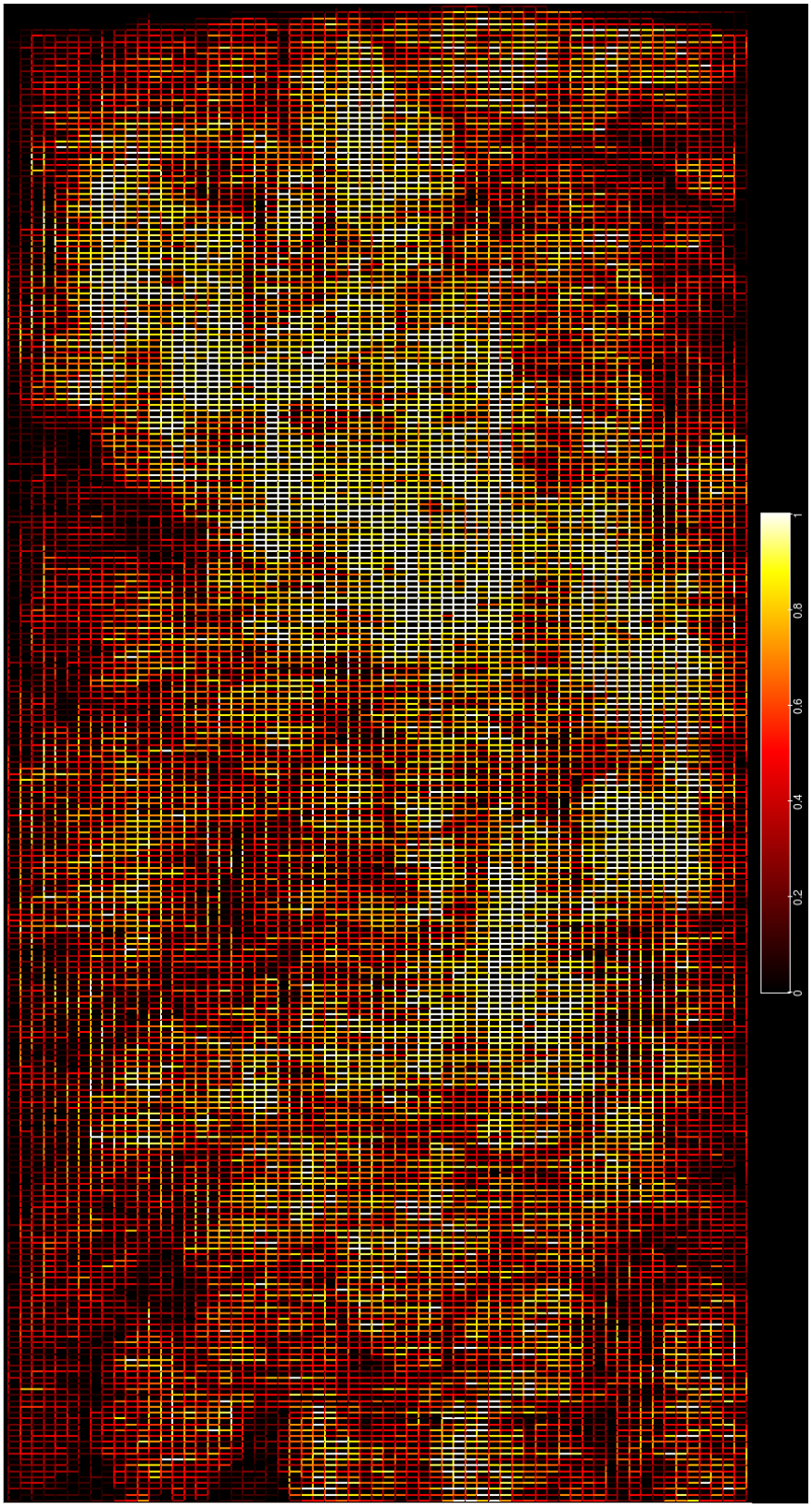


FIGURE C.26 – Carte de congestion du circuit *ibm09* routé par **FGR**

C.6 Cartes de congestion

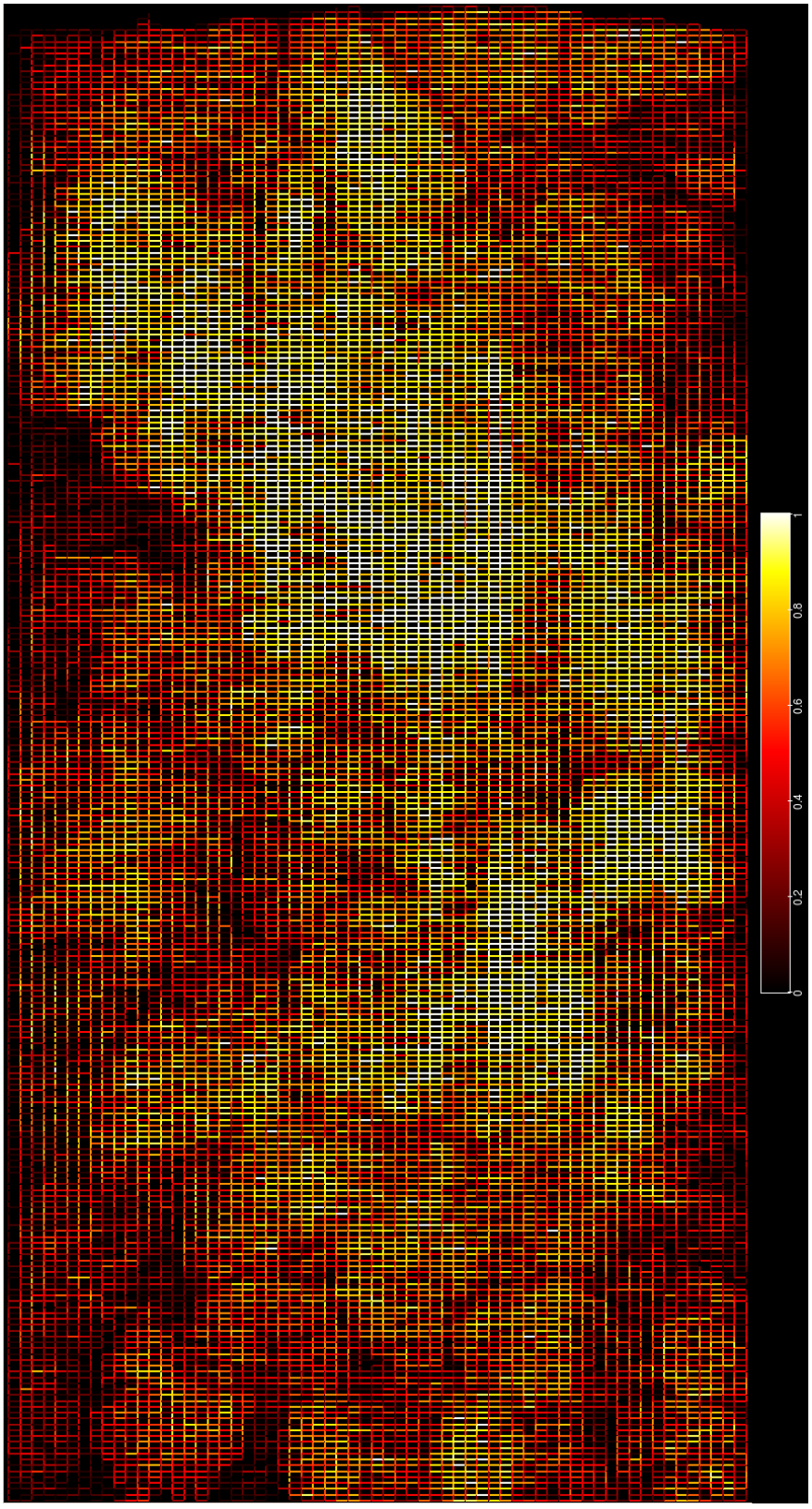


FIGURE C.27 – Carte de congestion du circuit *ibm09* routé par **KNIK**

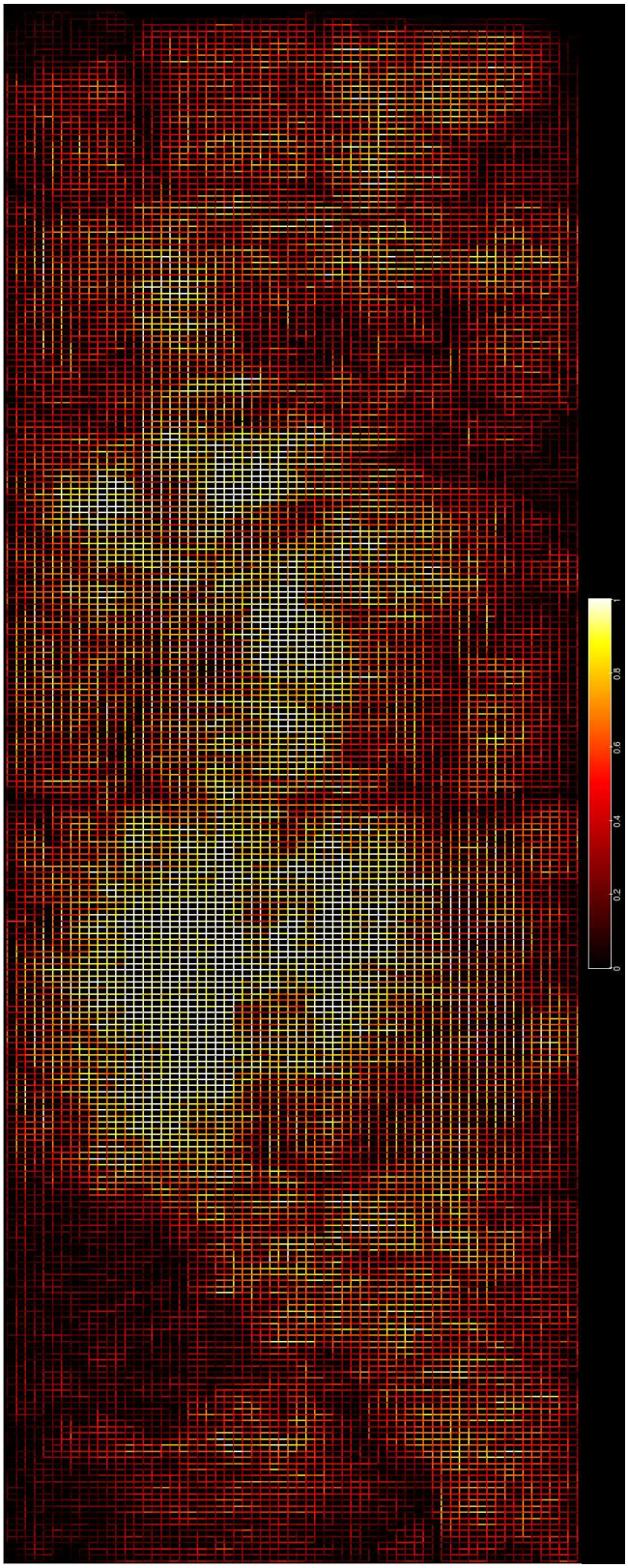


FIGURE C.28 – Carte de congestion du circuit *ibm10* routé par **FGR**

C.6 Cartes de congestion

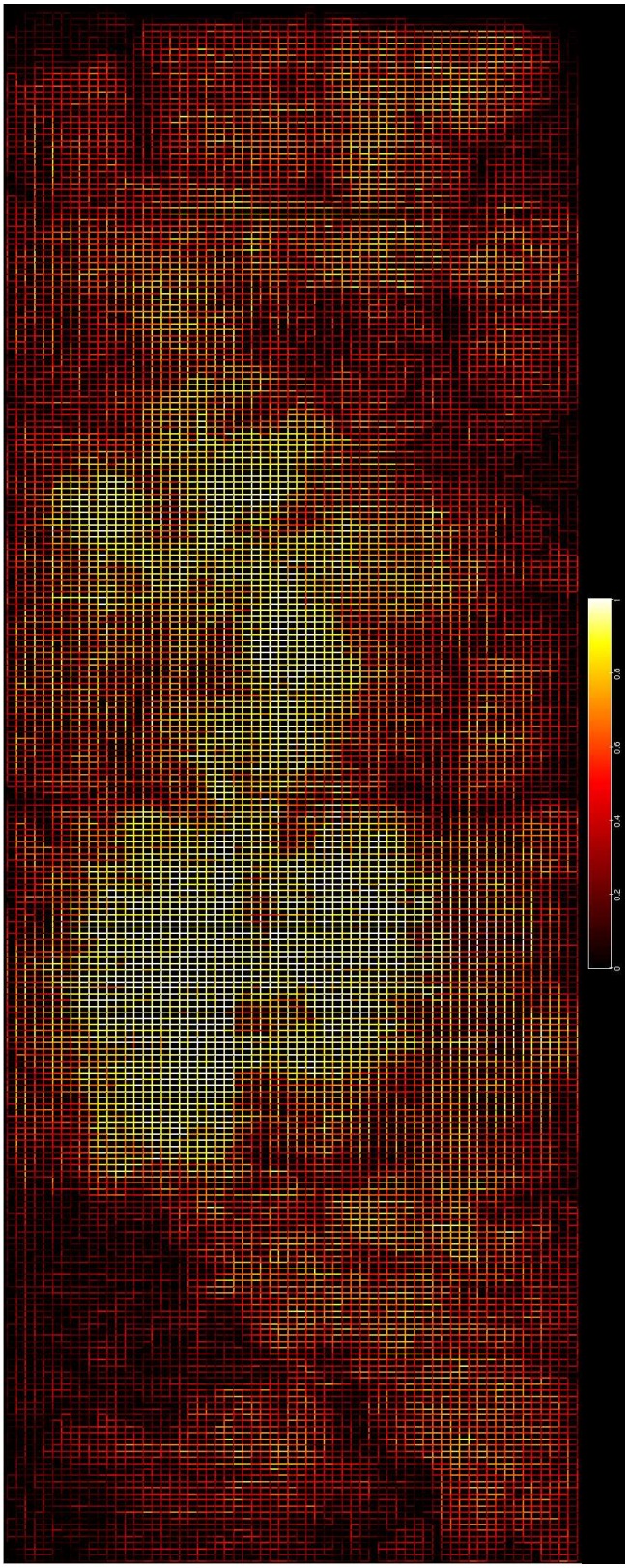


FIGURE C.29 – Carte de congestion du circuit *ibm10* routé par **KNIK**

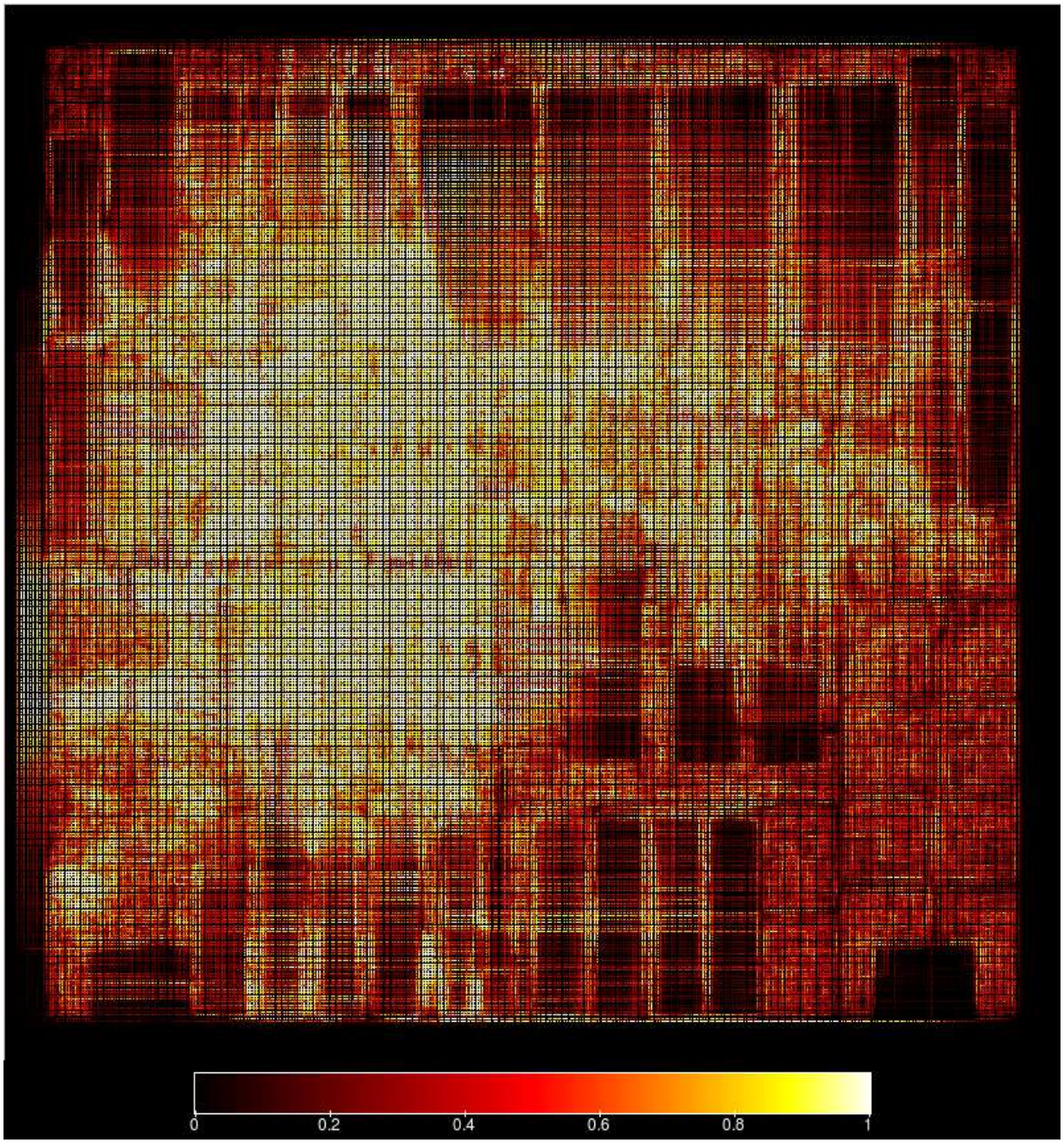


FIGURE C.30 – Carte de congestion du circuit *adaptec1* routé par **FGR**

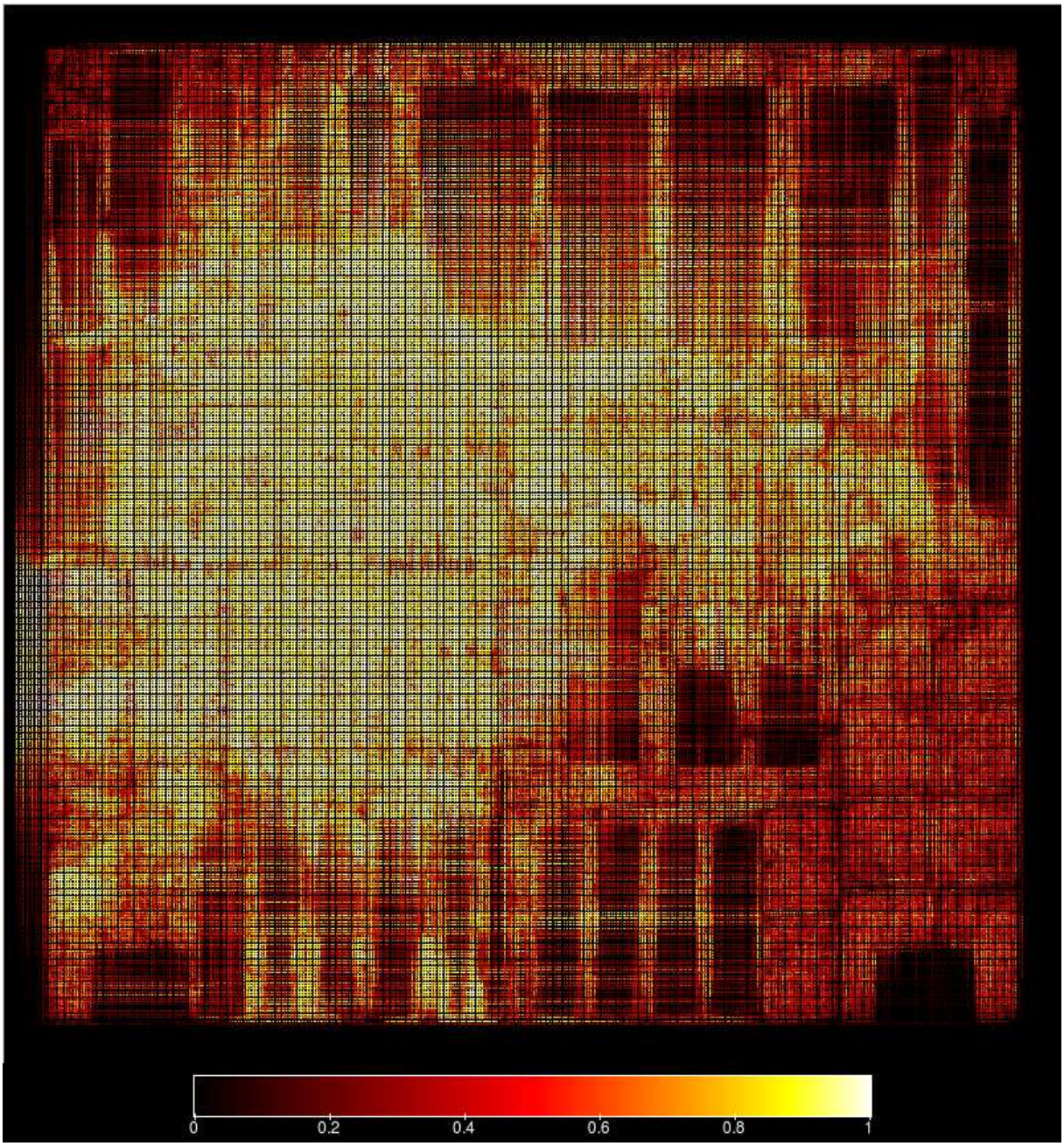


FIGURE C.31 – Carte de congestion du circuit *adaptec1* routé par **KNIK**

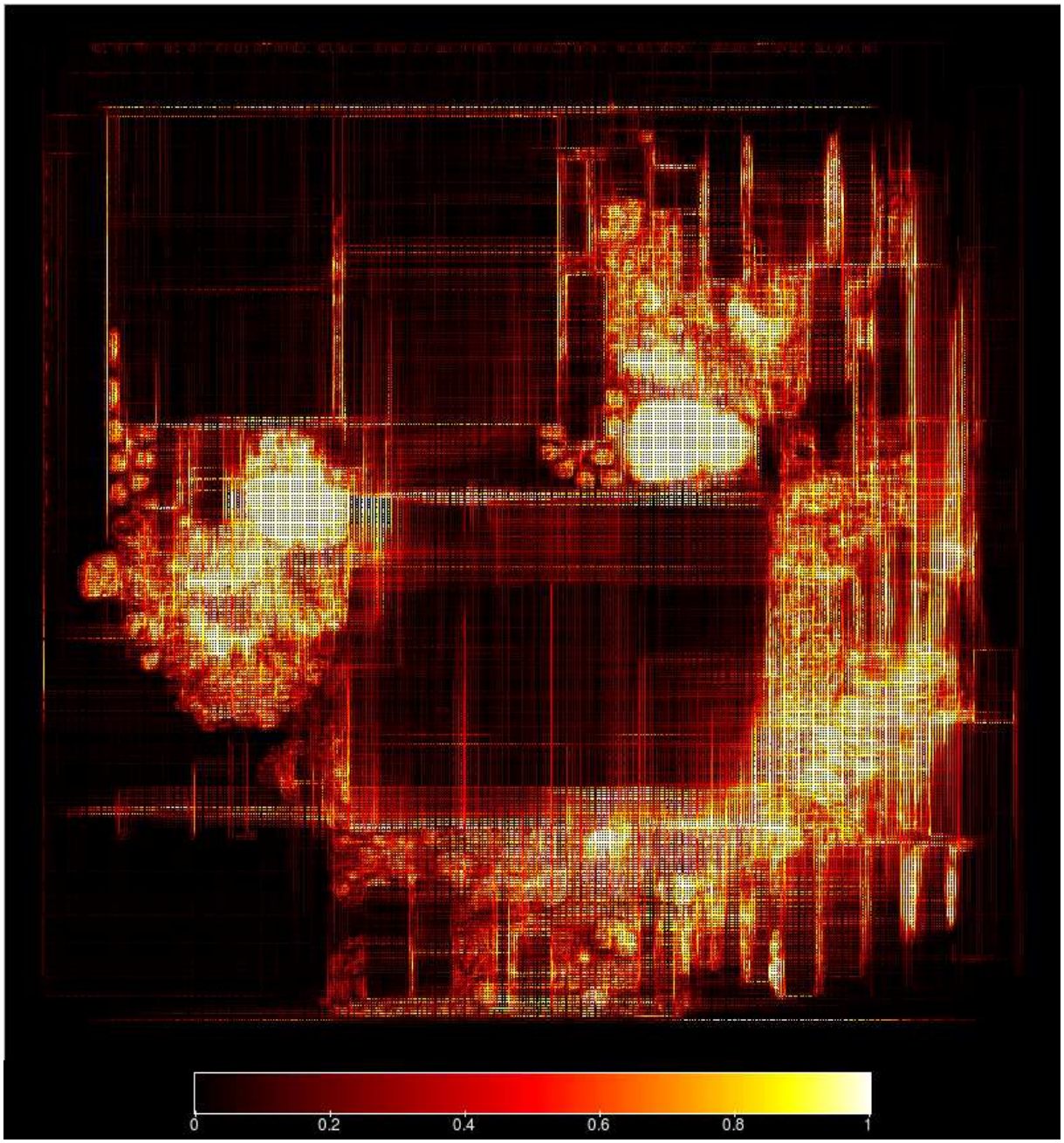


FIGURE C.32 – Carte de congestion du circuit *adaptec2* routé par **FGR**

C.6 Cartes de congestion

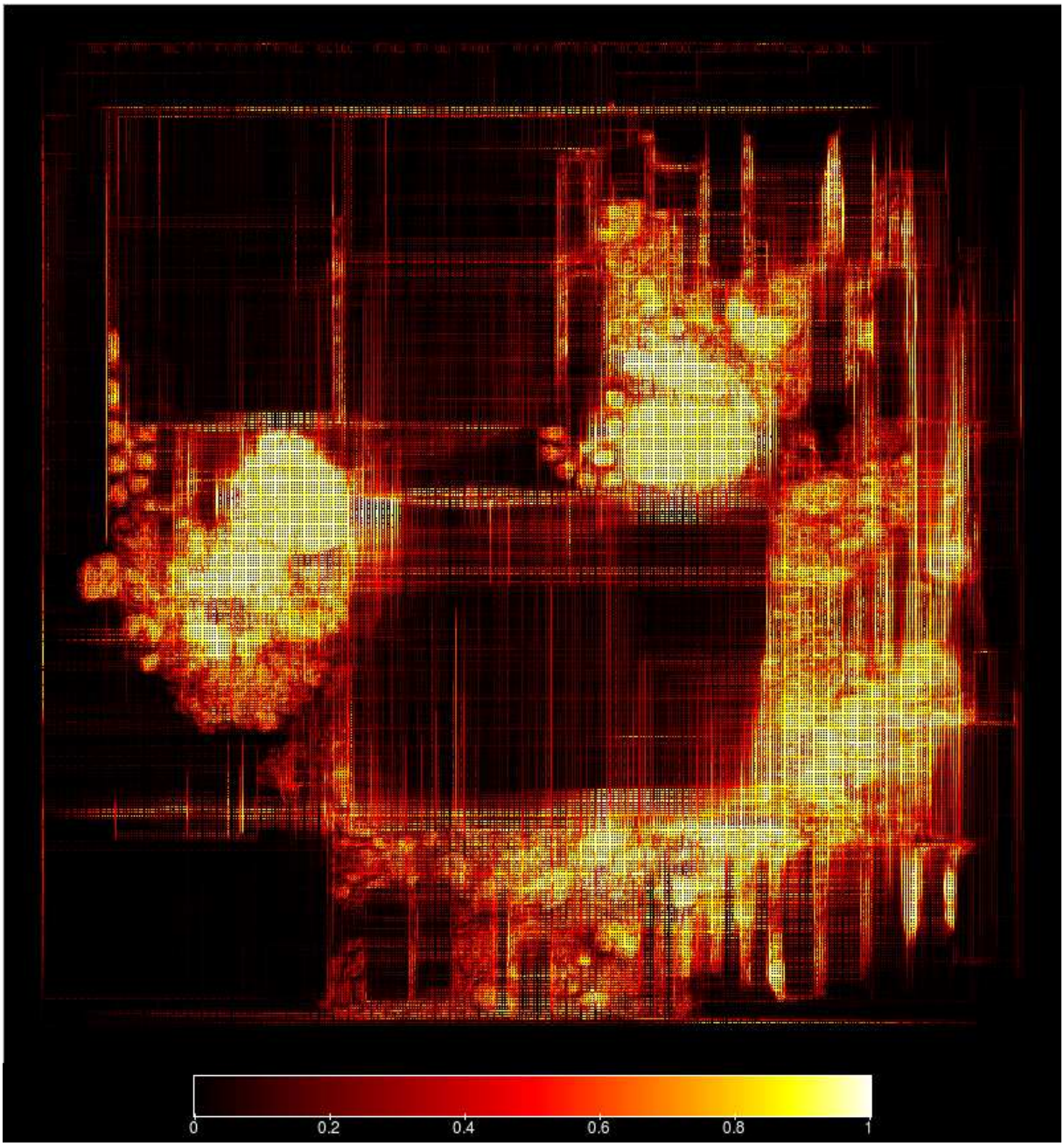


FIGURE C.33 – Carte de congestion du circuit *adaptec2* routé par **KNIK**

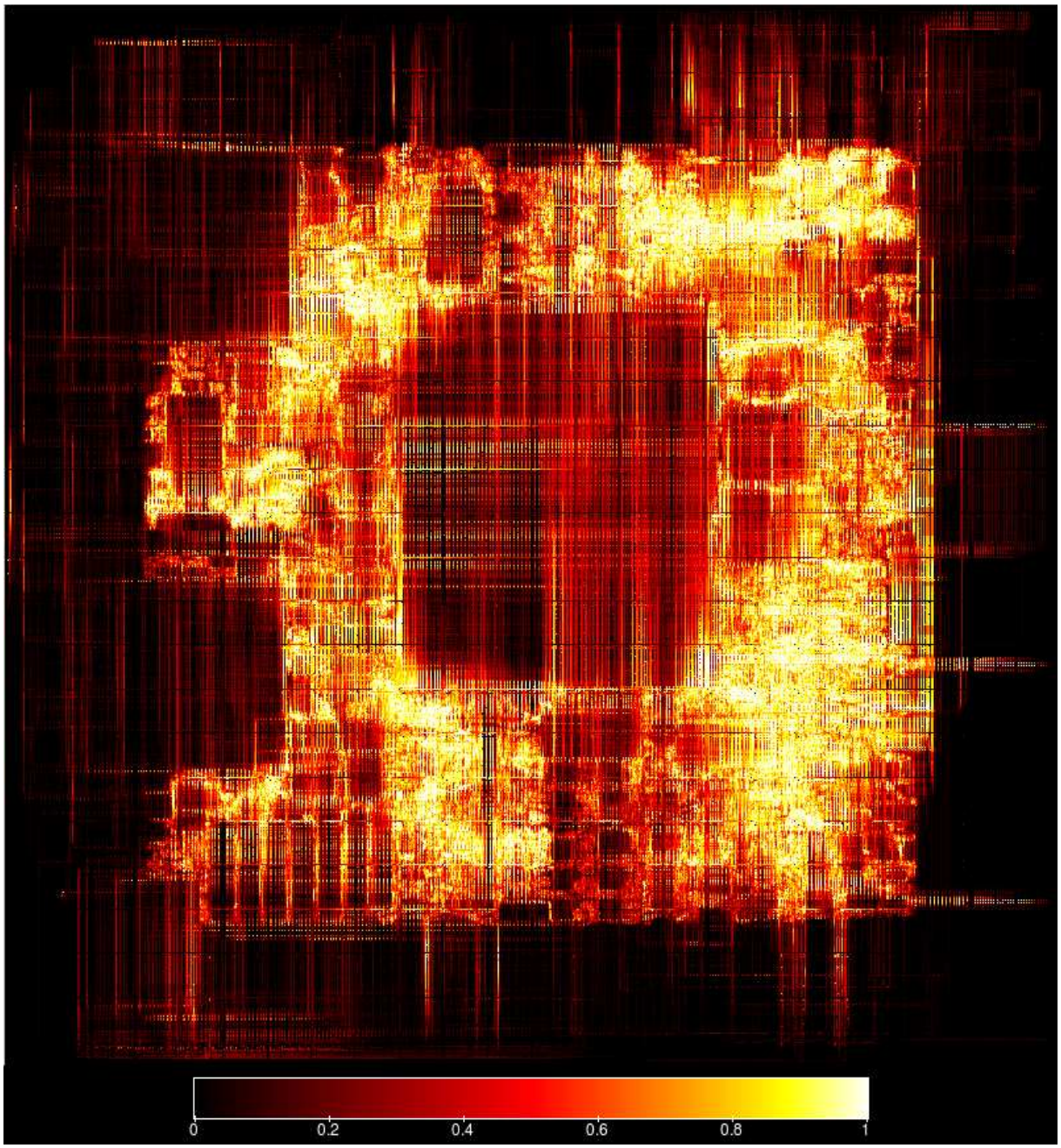


FIGURE C.34 – Carte de congestion du circuit *adaptec3* routé par **FGR**

C.6 Cartes de congestion

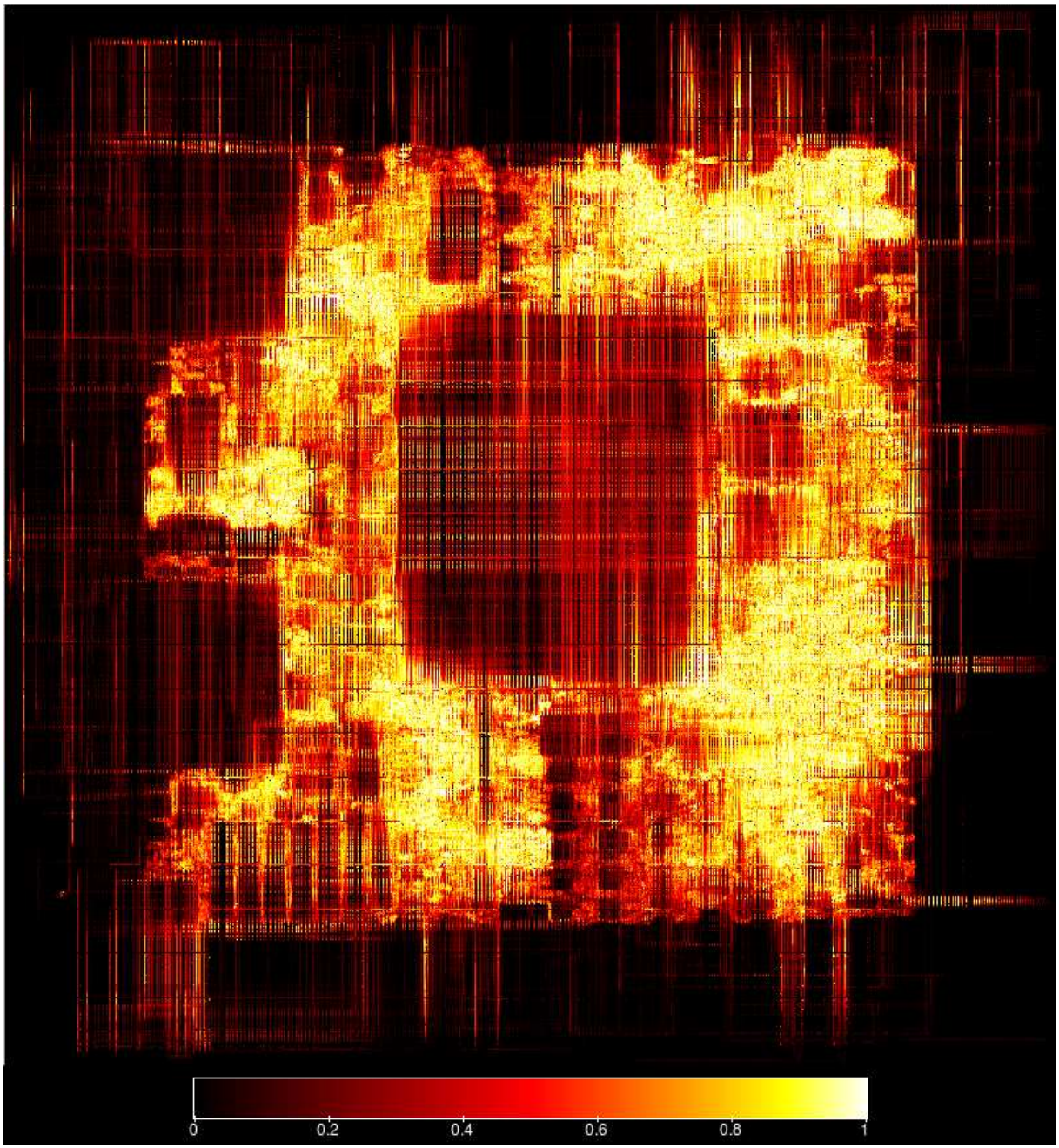


FIGURE C.35 – Carte de congestion du circuit *adaptec3* routé par **KNIK**

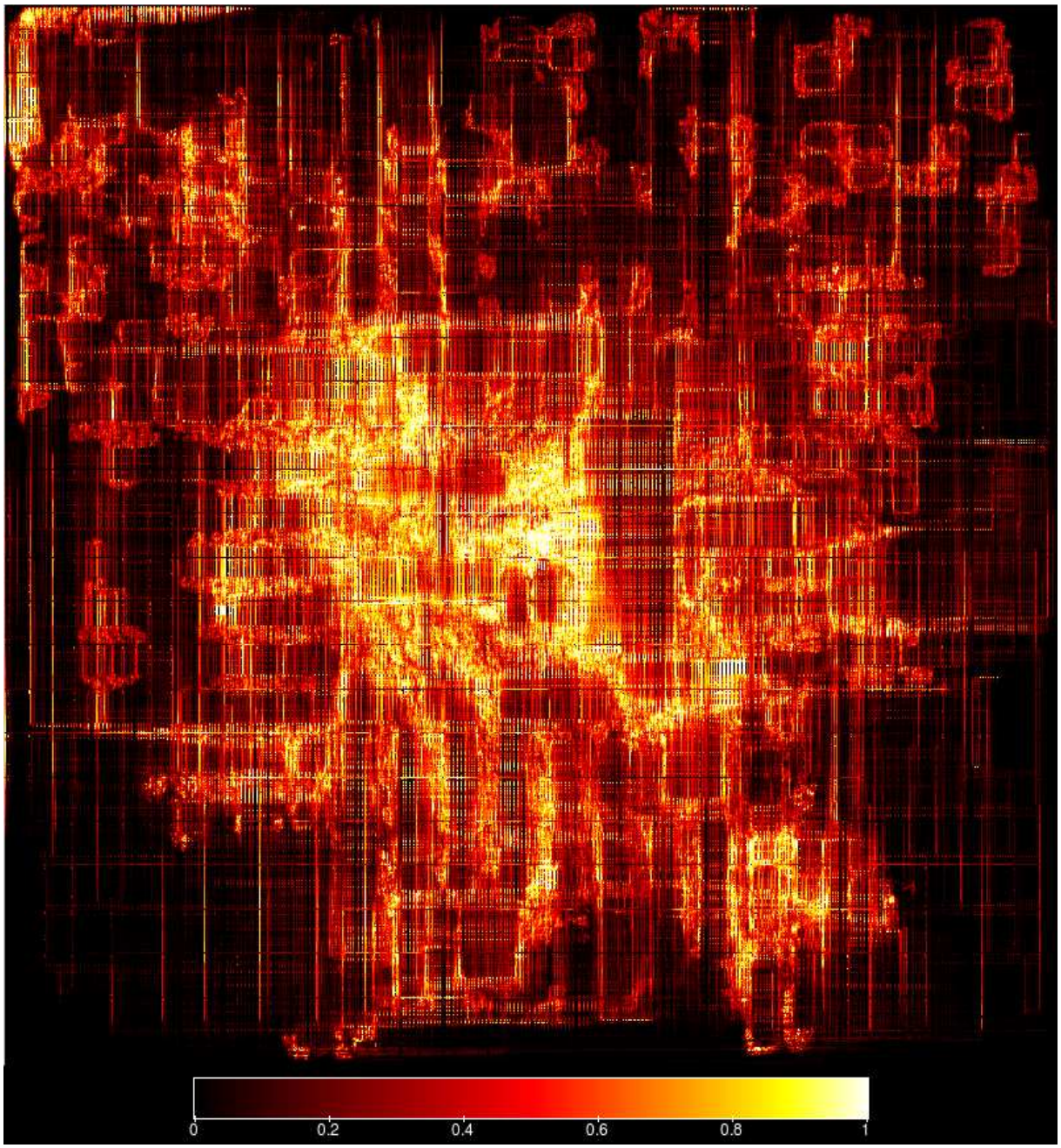


FIGURE C.36 – Carte de congestion du circuit *adaptec4* routé par **FGR**

C.6 Cartes de congestion

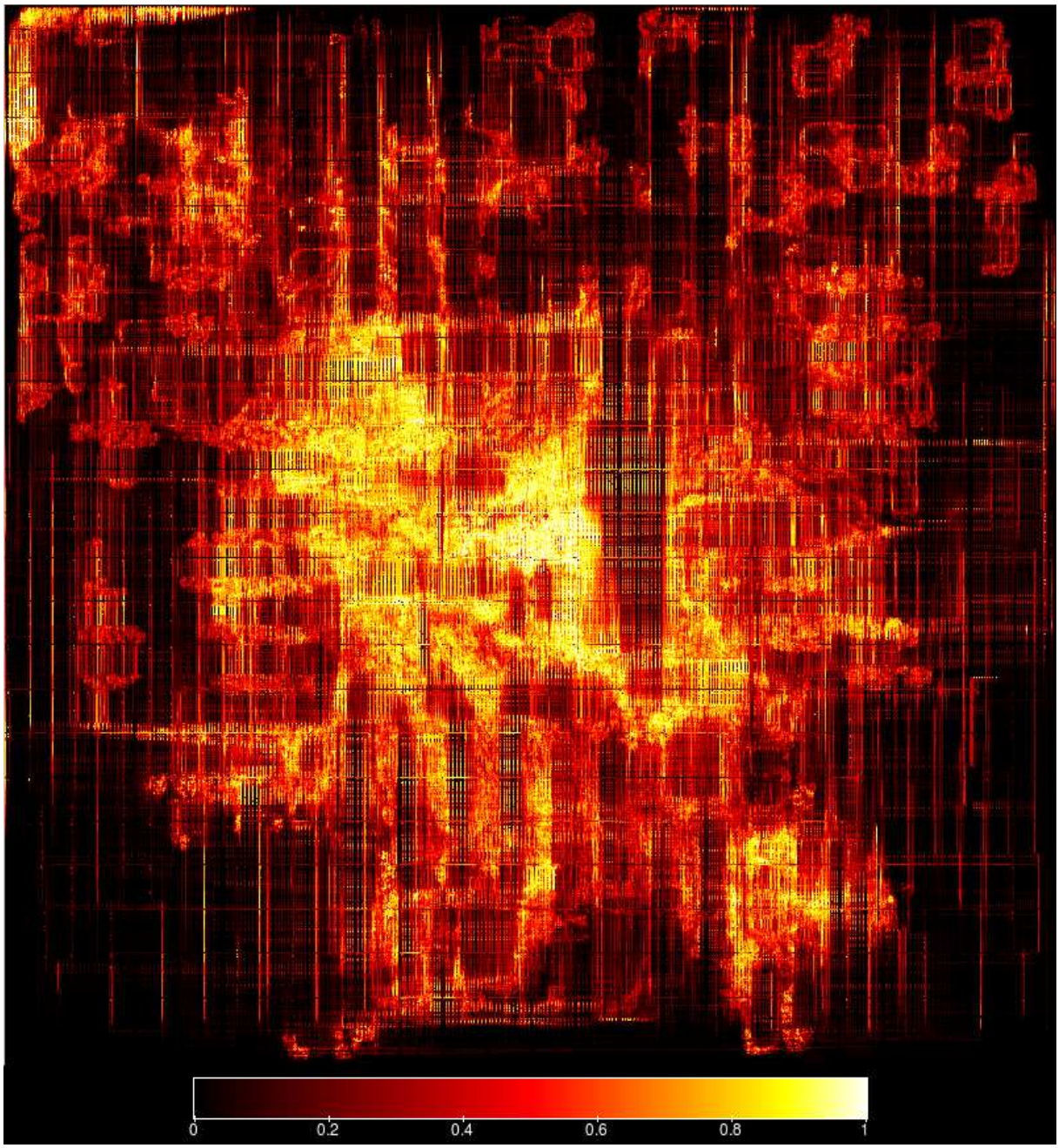


FIGURE C.37 – Carte de congestion du circuit *adaptec4* routé par **KNIK**

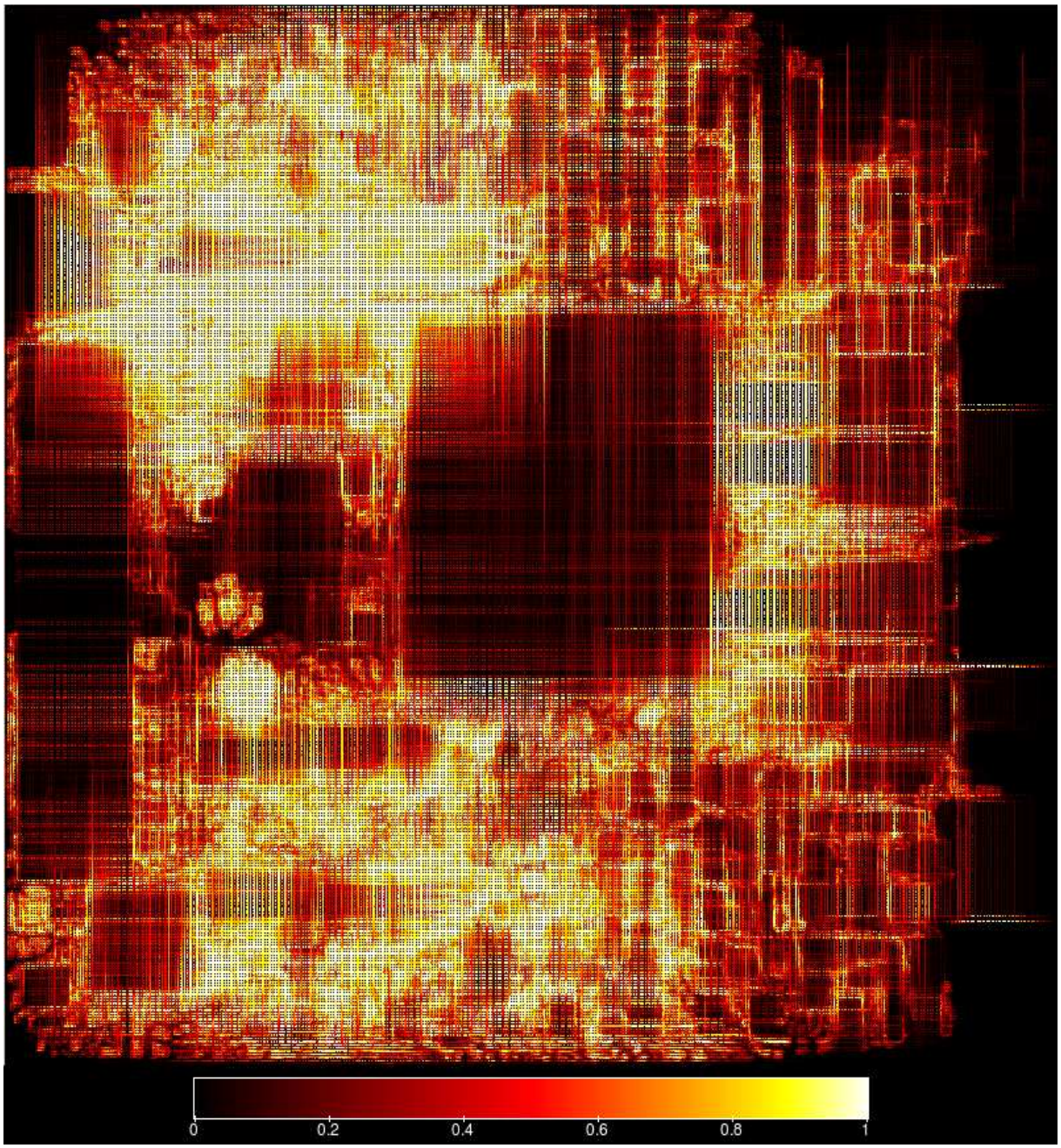


FIGURE C.38 – Carte de congestion du circuit *adaptec5* routé par **FGR**

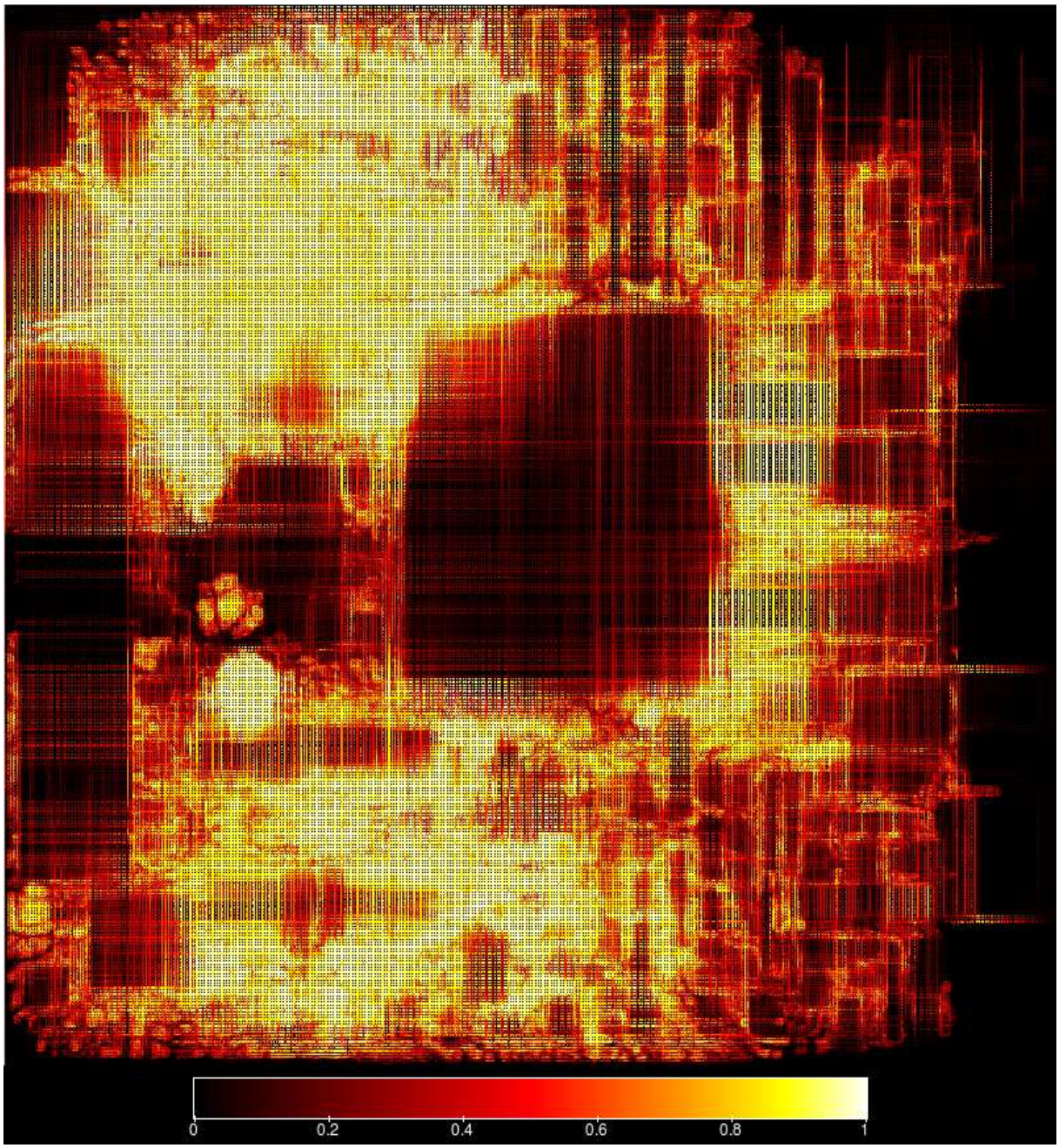


FIGURE C.39 – Carte de congestion du circuit *adaptec5* routé par **KNIK**

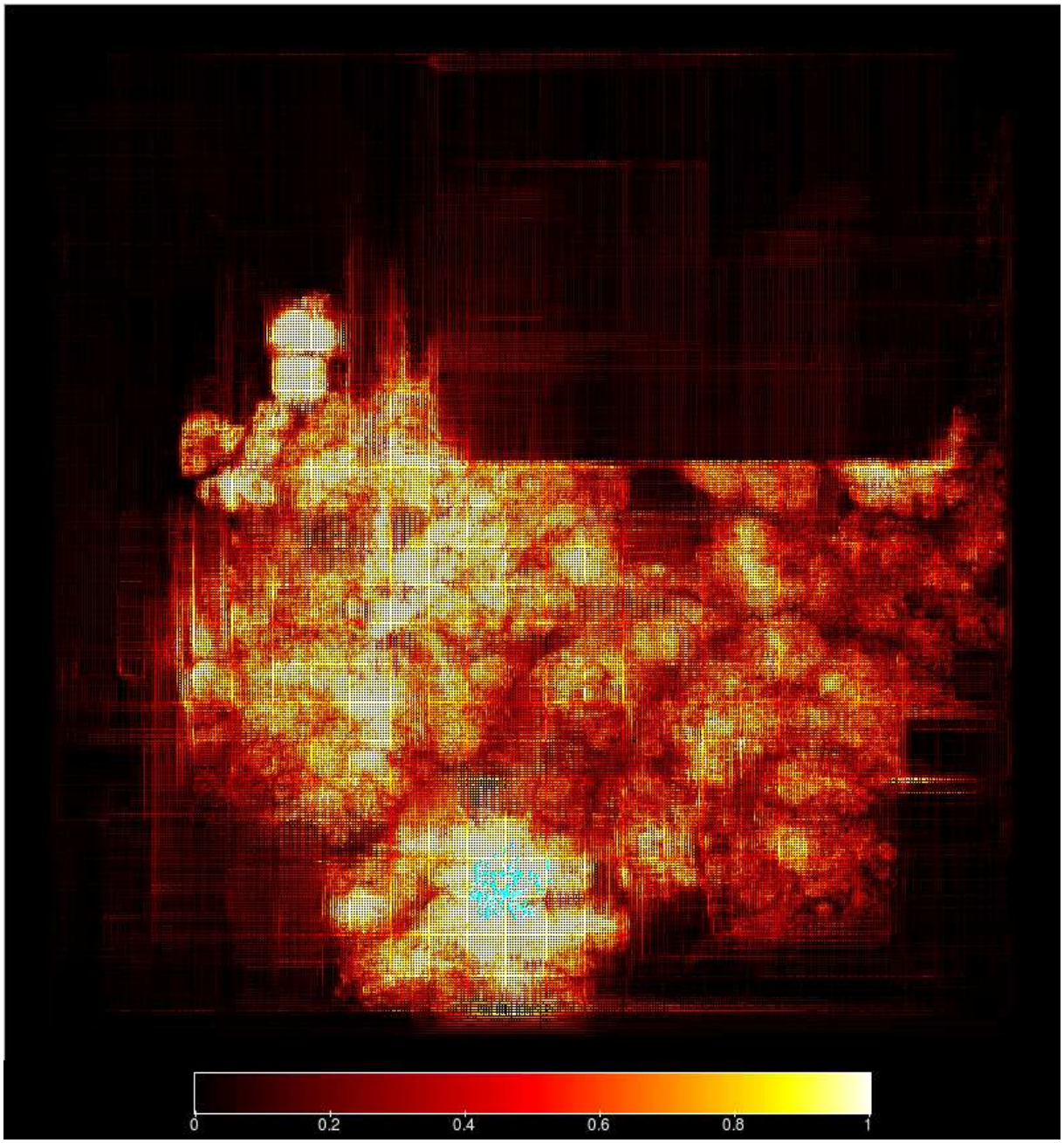


FIGURE C.40 – Carte de congestion du circuit *newblue1* routé par **FGR**

C.6 Cartes de congestion

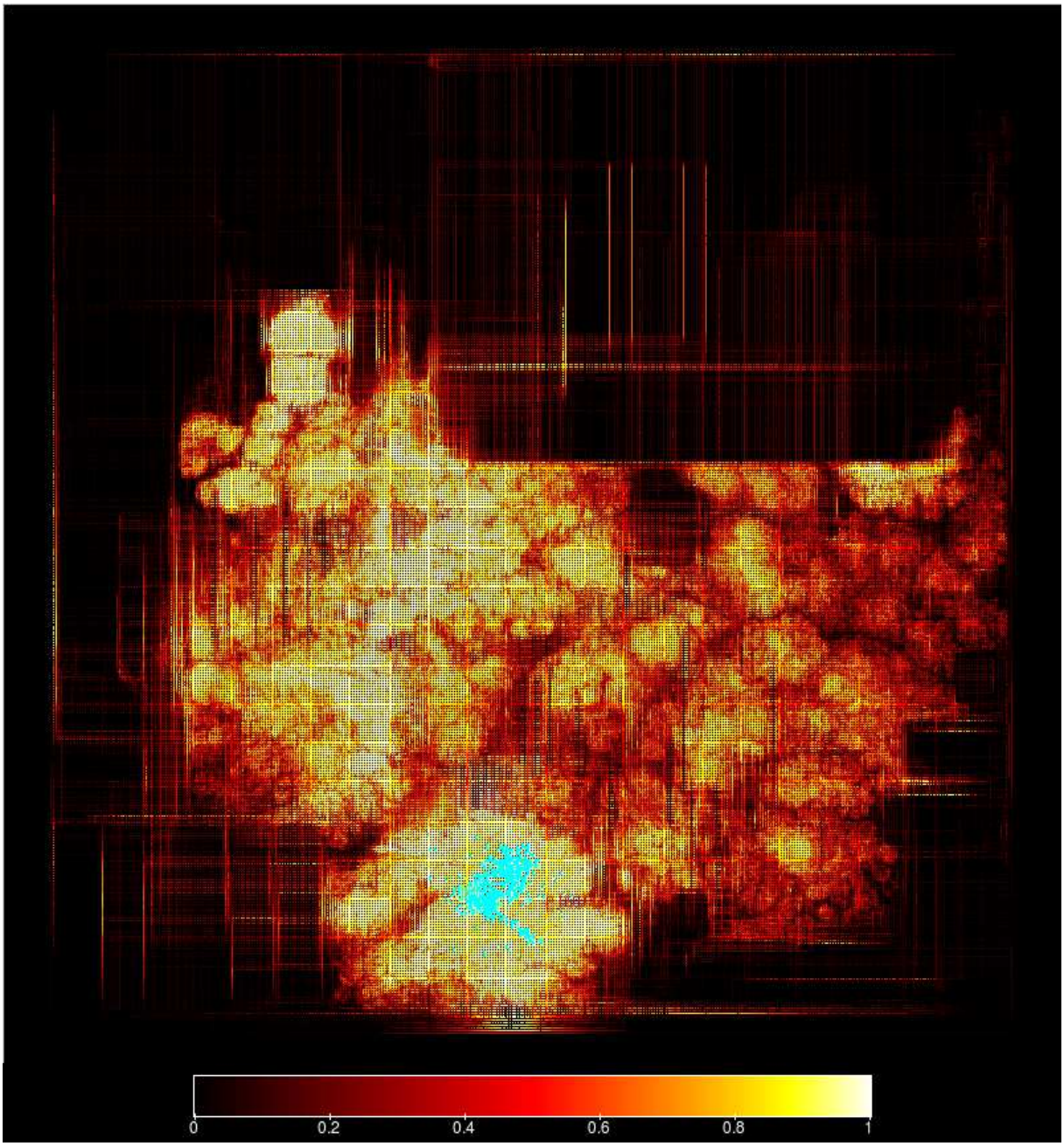


FIGURE C.41 – Carte de congestion du circuit *newblue1* routé par **KNIK**

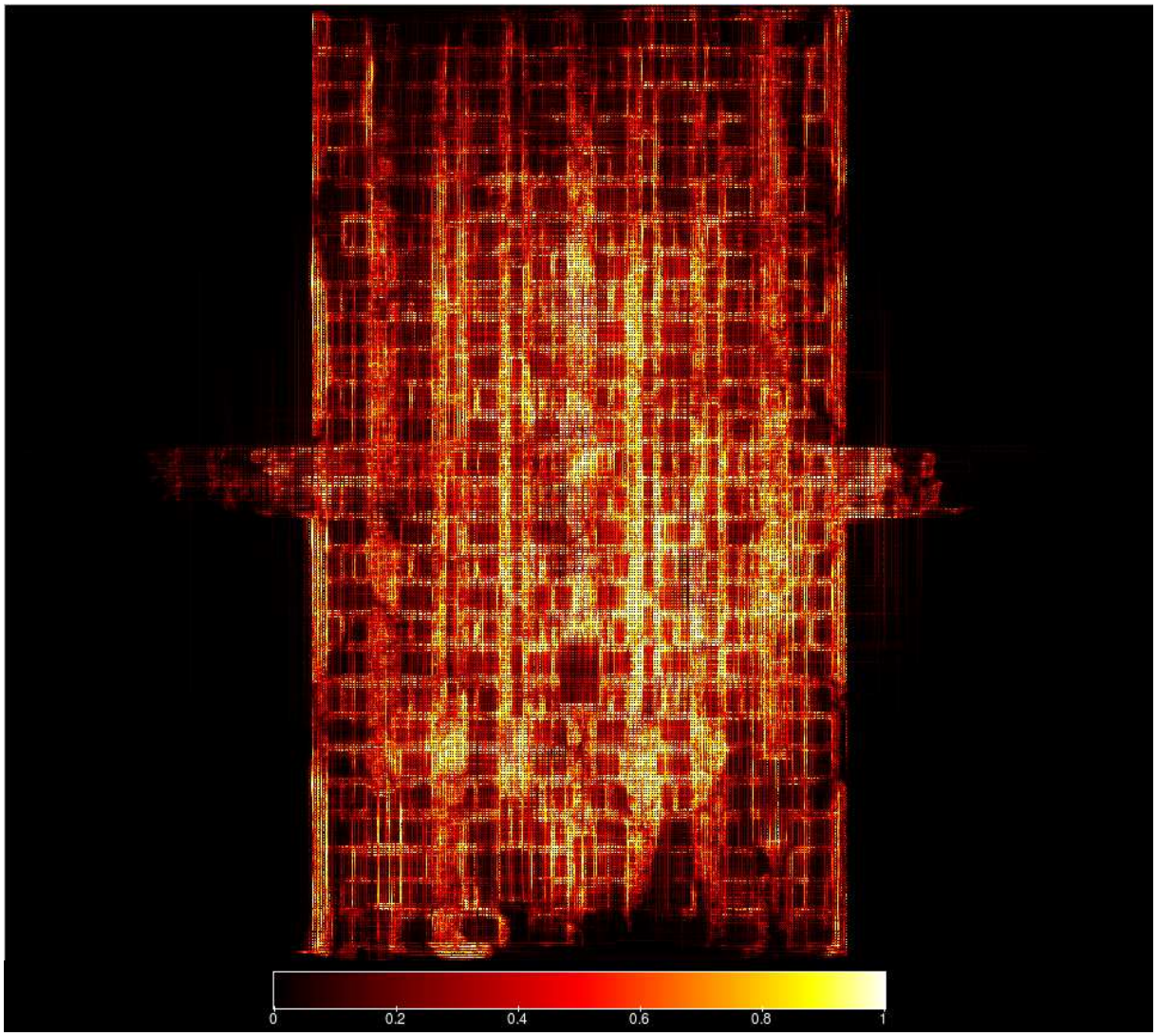


FIGURE C.42 – Carte de congestion du circuit *newblue2* routé par **FGR**

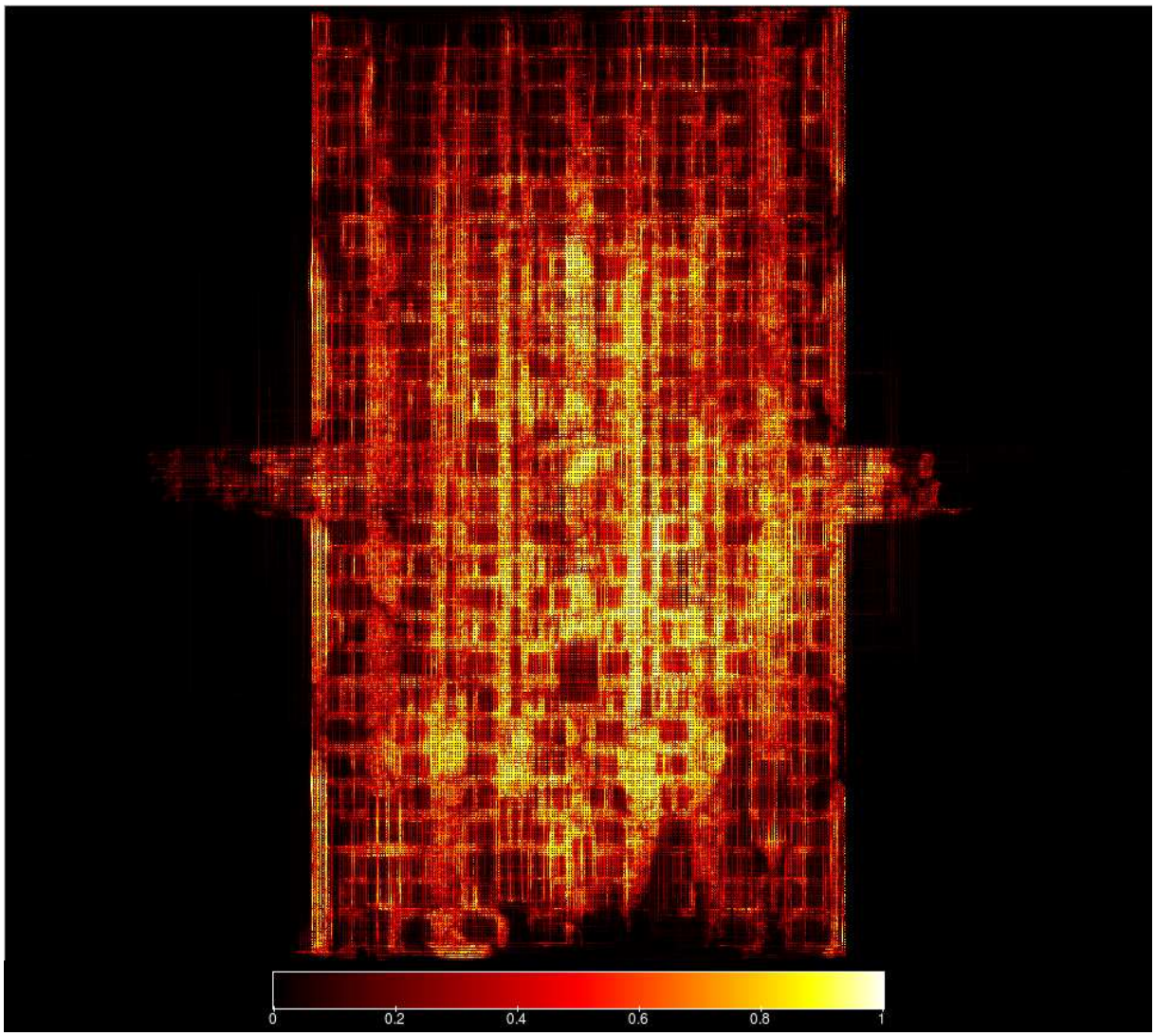


FIGURE C.43 – Carte de congestion du circuit *newblue2* routé par **KNIK**

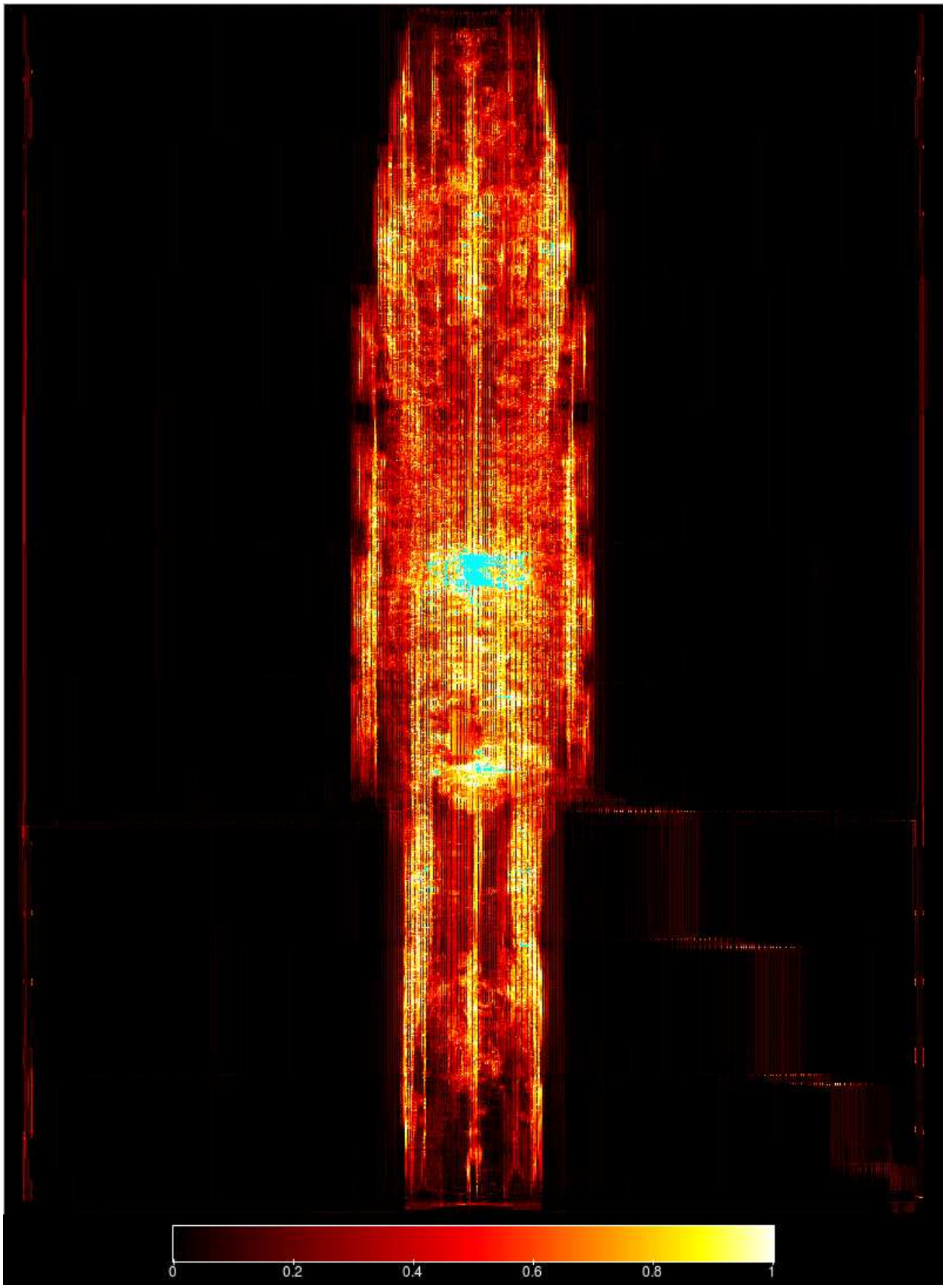


FIGURE C.44 – Carte de congestion du circuit *newblue3* routé par **FGR**

C.6 Cartes de congestion

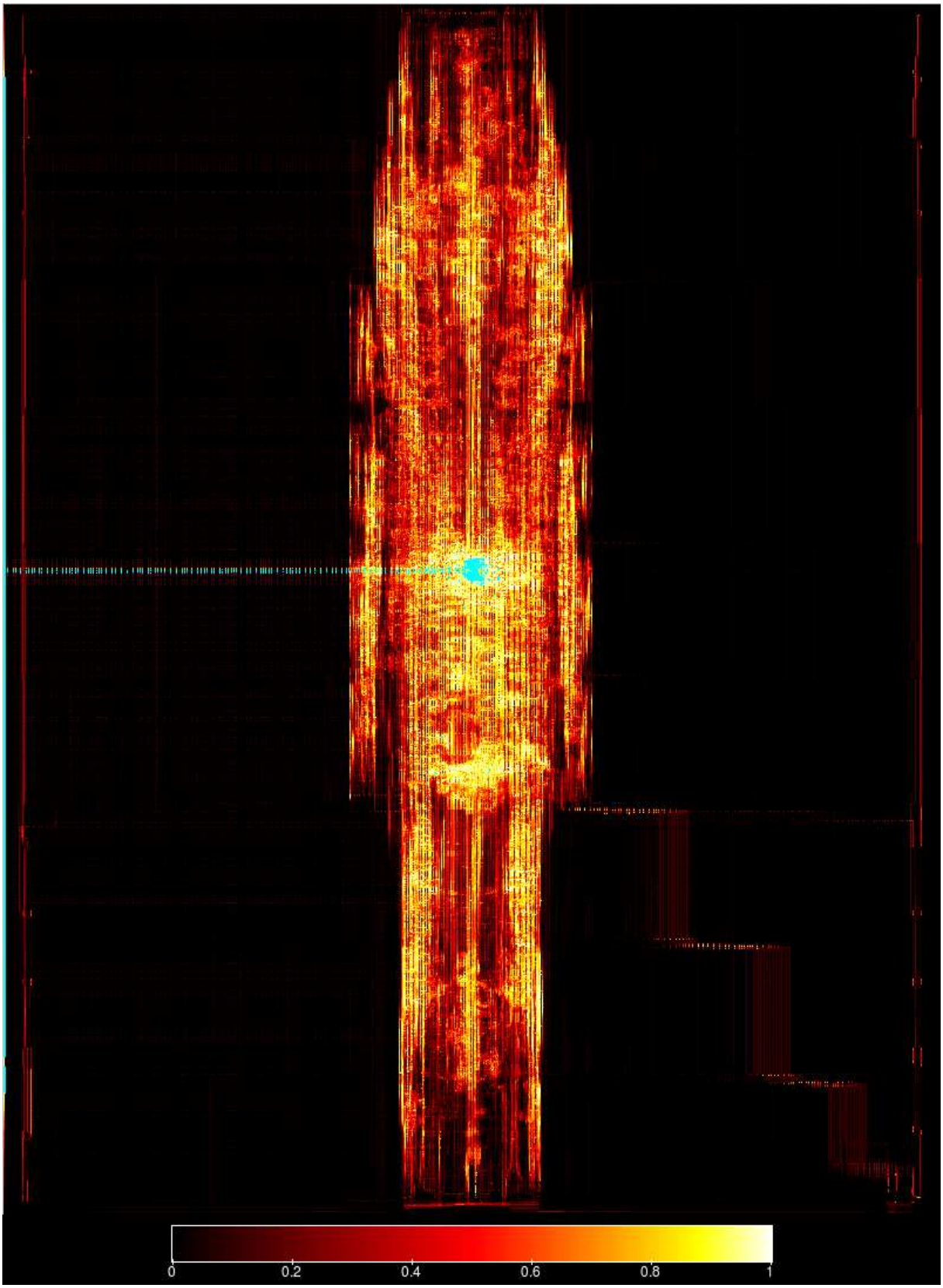


FIGURE C.45 – Carte de congestion du circuit *newblue3* routé par **KNIK**