Chapitre 4

Le routage au fil des ans

Va toujours par le chemin le plus court, et le plus court est le chemin tracé par la nature.

Marcus Aurelius ANTONINUS (MARC-AURÈLE)

LE ROUTAGE tel que nous l'avons déjà évoqué dans le chapitre présentant les circuits reconfigurables consiste en la génération d'une manière de relier différents points d'un graphe. A partir d'une liste de points ou coordonnées à relier, un algorithme de routage doit pouvoir fournir un ensemble de chemins permettant de les connecter. De plus, des contraintes telles que la longueur maximale d'un chemin ou le nombre de virages maximal peuvent y être ajoutées, en fonction du problème à résoudre.

Dans ce chapitre, nous allons tout d'abord présenter quelques exemples de tous les jours (ou presque) dans lesquels le routage joue un rôle central. Nous partirons ensuite des solutions proposées par la nature puis détaillerons les approches algorithmiques qui furent développées durant les cinquante dernières années, et ce dans un ordre chronologique. Nous verrons ensuite les différents systèmes qui furent proposés dans l'optique d'accélérer le traitement du routage, en commençant par les coprocesseurs, et en finissant par des systèmes purement matériels, qui nous amèneront tout naturellement aux travaux de cette thèse qui seront présentés dans le chapitre suivant.

4.1 Pourquoi un plus court chemin ?

La question peut sembler inutile. Il est vrai qu'à part un touriste se promenant dans une ville sans but précis, ou un processus de recherche aléatoire ne visant pas de point défini, les exemples sont plutôt rares où les trajectoires ne sont pas destinées à directement relier deux points. Lorsqu'un taxi vous emmène à l'aéroport, pour autant que le chauffeur soit honnête, il empruntera le chemin le plus court afin de rendre le temps de la course (et donc le coût) le plus faible possible. De manière générale, nos déplacements sont guidés par la minimisation du chemin à parcourir, l'homme étant toujours à la recherche du moindre effort. Sur le plan technique, les réseaux de

communication de type Internet sont créés sur la base de protocoles visant à réduire le temps de transfert de l'information entre deux points du globe. Des algorithmes de routage dynamique sont ici nécessaires à l'envoi de paquets au travers de routeurs, dans un réseau global tolérant aux pannes.

La réalisation de systèmes digitaux de type VLSI (Very Large Scale Integrated Circuits) ou de cartes PCB nécessite également la recherche du plus court chemin. En effet, relier deux transistors par un fil de métal ou deux circuits par un fil de cuivre implique un délai d'autant plus grand que le fil est long. De plus, la tâche de relier tous les points donnés est dite NP-dure (cf. page 45), c'est-à-dire qu'elle est non triviale et qu'une méthode déterministe ne peut lui trouver une solution en un temps polynomial. Notons donc ici la distinction entre la recherche du plus court chemin et le problème de routage : ce dernier nécessite de réaliser plusieurs connexions entre différents points, et peut dès lors utiliser la recherche du plus court chemin dans la génération de chaque liaison.

Nous allons poursuivre en présentant quelques solutions, naturelles, puis algorithmiques, au problème de la recherche du plus court chemin, puis nous continuerons avec le problème plus général du routage. Nous ne traiterons pas des systèmes de routage dynamique de type réseaux de communications, notre approche étant plutôt basée sur une grille régulière d'éléments. En revanche nous accorderons une attention particulière à toutes les solutions qui furent développées dans le cadre de l'optimisation des outils de conception de circuits.

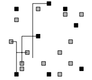
4.2 Aparté naturel

La nature s'efforce toujours de dépenser le moins d'énergie possible, ce qui la mène tout naturellement à chercher le chemin le plus court dans diverses circonstances. Les animaux, tels les fourmis, ont, au cours de l'évolution, développé divers systèmes leur permettant d'économiser leur énergie en empruntant des chemins les plus courts possibles. Les bulles de savon optent pour des formes minimisant la pression présente sur leur membrane, et offrent un moyen de trouver la membrane de taille minimum reliant plusieurs points. Un gaz lâché en un point d'un labyrinthe s'étendra de manière régulière jusqu'à la sortie. En remontant alors le gradient chimique, il est possible de reconstituer le plus court chemin de la source à la sortie.

Ces exemples naturels sont autant de sources d'inspiration pour les chercheurs travaillant sur des problèmes nécessitant l'optimisation d'un chemin entre plusieurs points. La présente thèse ayant été grandement influencée par ces phénomènes naturels, nous les présentons brièvement ici.

4.2.1 Les fourmis

La plupart des colonies de fourmis sont en quête perpétuelle de nourriture. Des éclaireuses partent à la recherche de nouvelles sources, et lorsqu'elles en ont trouvé une, reviennent au nid, déposant au passage des phéromones. Ces phéromones, hormones olfactives, servent aux ouvrières à retrouver la nourriture fraîchement découverte. En passant sur ce chemin, elles déposeront également des phéromones, accentuant la force du chemin. En conséquence, plus un tracé est utilisé, plus il sera marqué de phéromones, et plus les fourmis auront tendance à suivre cette piste olfactive forte, et donc à suivre leurs congénères en direction de leur nourriture. De plus, lorsqu'un



chemin existant se trouve bloqué par un événement quelconque, les fourmis sont capables de trouver, toujours selon la même méthode, un détour par lequel elles peuvent contourner l'obstacle.

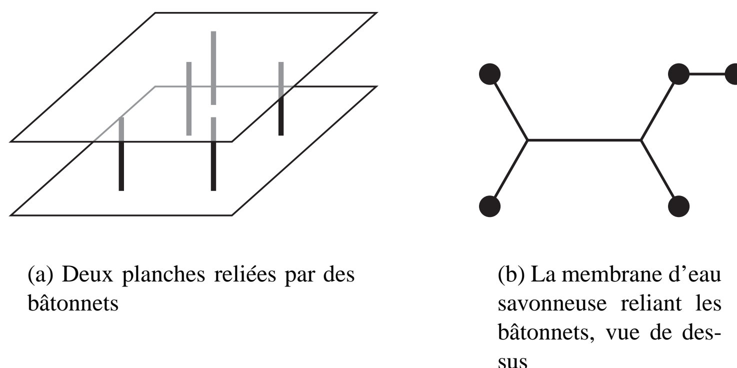
Basée sur ce principe, l'"Ant Colony Optimisation" (ACO), introduite en 1991 par Colorni, Dorigo, et Maniezzo [43], permet de résoudre des problèmes NP-complets, tels que celui du voyageur de commerce. Cependant, la méthode développée par les fourmis pour trouver le chemin le plus court semble difficilement adaptable aux systèmes digitaux, c'est pourquoi nous allons, dans les deux sections suivantes, explorer les bulles de savon, les sons, et l'expansion des gaz.

4.2.2 Les bulles de savon

Les bulles de savons sont un bel exemple de connexions de points grâce à un réseau minimal [76, 77, 109]. En 1892 déjà, Boyz présentait à un auditoire ses travaux sur les bulles de savon :

> "Quelle que soit la complication de la carcasse, on ne voit jamais plus de trois membranes se couper le long d'une crête, ou de quatre arêtes ou de six membranes se rencontrer en un point. [...] Pendant la formation des bulles, on voit parfois un nombre trop grand de membranes se rencontrer en un point ou le long d'une arête, mais l'une d'elles glisse aussitôt et l'ensemble redevient stable. Dans tous ces systèmes, les plans qui passent par une même arête s'y rencontrent sous des angles égaux." [29](pp. 47-48)

Ces quelques phrases, qui peuvent sembler quelque peu abruptes, définissent les propriétés essentielles des membranes d'eau savonneuse. Lorsque plusieurs bulles de savon sont accolées, elles s'appareillent de manière à minimiser la tension superficielle de leur membrane. Les bords extérieurs sont toujours des cercles, et les parois intérieurs séparant deux membranes se croisent toujours selon des angles égaux. Ce phénomène peut être exploité pour trouver un chemin de taille totale minimale entre plusieurs points (un arbre de Steiner, cf. page 86).



(a) Deux planches reliées par des bâtonnets

(b) La membrane d'eau savonneuse reliant les bâtonnets, vue de dessus

Figure 4.1 : *Deux planches reliées par des clous et la membrane connectant ces bâtonnets.*

La procédure est comme suit. Prenez une paroi transparente et collez-y des bâtonnets sur une de ces faces. Placez ensuite une autre paroi à l'autre extrémité des bâtonnets de manière à ce que les bâtonnets touchent ladite paroi (Figure 4.1(a)). Trempez

la construction dans de l'eau savonneuse et retirez-en la. Vous verrez alors une membrane d'eau savonneuse reliant tous les bâtonnets (Figure 4.1(b)). Il est très intéressant de noter que la membrane ainsi formée correspond au chemin minimum permettant de relier les différents bâtonnets[1]. La raison en est la minimisation de l'énergie contenue dans la membrane. Un fait remarquable est qu'à chaque intersection de trois pans de membrane, les angles les séparant sont exactement de 120 degrés, toujours pour des raison d'énergie.

Dès lors il semble être prometteur de tester des architectures de type hexagonales permettant justement la création de liaisons à 120 degrés.

4.2.3 Le gaz, le son, la lumière

Un gaz lâché en un point d'un espace quelconque, s'étendra dans cet espace de manière uniforme, à la manière d'un son. Cette "exploration" est similaire à un front d'onde (sonore ou lumineuse), qui avance dans l'espace. A un temps t, tous les points sur le front d'onde sont à la même distance de l'origine (dans le cas d'un milieu homogène), et la direction opposée à l'expansion indique la direction du plus court chemin menant à la source. La figure 4.2 présente le front d'onde d'un son ou l'expansion d'un gaz, à partir d'une source (disque noir), où pour trouver le chemin le plus court entre la source et un point quelconque (disque blanc), il suffit de remonter perpendiculairement au front.

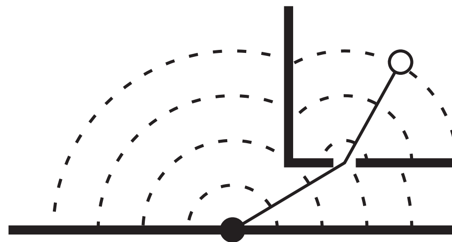


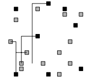
Figure 4.2 : *L'expansion d'un gaz, ou le front d'onde d'un son.*

Cette expansion est similaire à une recherche en largeur d'abord, qui sera explicitée plus loin, et servira de modèle pour la réalisation d'un système de routage dynamique réalisé en matériel. En effet, l'expansion peut être vue comme un processus parallèle, chaque point du front d'onde au temps $t + \Delta t$ étant "calculé" au même instant par les lois de la nature. Le matériel, de par ses possibilités de calcul parallèle, semble donc être en mesure de s'inspirer de ce phénomène pour étendre un "front d'onde" sur une structure cellulaire.

4.2.4 De la ficelle

Avant de passer aux bases théoriques, citons une méthode originale pour résoudre le problème du plus court chemin entre deux villes. Minty, en 1957, propose, dans une note de 12 lignes dans le journal Operations Research [161], une solution à base de ficelle et de clous. Il suffit de prendre un clou par ville, et de les relier par des bouts de ficelle dont la longueur correspond à la distance séparant chaque paire de villes

[1]Suivant la configuration des bâtonnets, il peut s'agir d'un minimum local, et un nouveau plongeon dans l'eau savonneuse peut modifier le résultat.



reliées par une route. En tirant ensuite sur les clous qui représentent les villes à relier, le chemin le plus court se trouve tout naturellement en suivant les bouts de ficelle tendus.

Par cette approche, il est également possible de trouver en une fois tous les chemins entre un point (la source) et tous les autres, en suspendant des poids à tous les clous. En tenant le clou de la source, tous les autres clous pendront par une suite de bouts de ficelle représentant le plus court chemin à la source.

4.3 Les Bases théoriques

Le problème du routage consiste en trouver une manière de relier plusieurs points grâce à des liaisons physiques tout en respectant certaines contraintes. Typiquement, dans un espace discrétisé composé d'un ensemble de points P, étant donné une liste $L = \{L_i\}$ de liaisons $L_i = (S_i, D_i)$ à créer avec $S_i \in P$ et $D_i \in P$, un tel algorithme doit retourner une liste de connexions $C = \{C_i\}$ où $C_i = (S_i, P_{i,1}, P_{i,2}, ..., P_{i,j-1}, P_{i,j}, D_i)$. Dans la plupart des cas, les contraintes impliquent qu'un point ne peut appartenir à deux chemins de source différente, comme c'est le cas pour les circuits imprimés. De plus, il est en général intéressant de générer des chemins les plus courts possibles, afin de minimiser les délais sur les fils, ou le temps de parcours dans le cas d'un chauffeur de taxi cherchant à amener un client à bon port.

4.3.1 Arbre de poids minimal

Avant les solutions au plus court chemin, ce fut le problème de la création de l'arbre couvrant de poids minimum (Définition 4.1) qui fut traité par les scientifiques.

Définition 4.1. *L'arbre couvrant minimal (minimum spanning tree) d'un graphe, dont un exemple est montré à la figure 4.3, est l'arbre contenant tous les nœuds (ou terminaux) du graphe, et dont la somme des poids des arêtes est minimale.*

L'arbre couvrant de poids minimum permet de relier plusieurs points par un réseau de poids minimum. Les réseaux électriques, par exemple, en tirent parti, afin de minimiser la quantité de câbles qui servent à interconnecter l'ensemble des points du réseau.

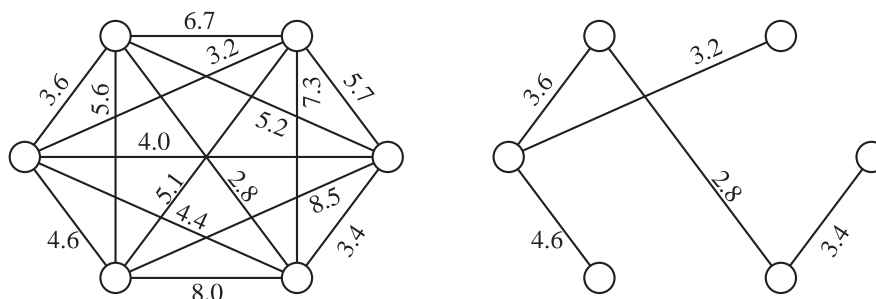


Figure 4.3 : *A droite, l'arbre couvrant minimal du graphe de gauche.*

Kruskal

Kruskal est le premier, en 1956, à proposer une méthode de création de l'arbre couvrant minimal d'un graphe non-orienté [130], et ce dans l'optique de prouver un théorème sur l'unicité de l'arbre couvrant minimal publié en tchèque en 1926 [28]. Partant d'un graphe $G = (V, E)$, où V est un ensemble de nœuds, et E un ensemble d'arêtes, associées à un poids et connectant ces nœuds, il permet de créer l'arbre de poids minimum.

Dans son article, il présente trois algorithmes donnant le même résultat. Nous n'en explicitons qu'un seul, les deux autres faisant appel à une approche très similaire. L'approche, triviale, consiste en la création d'un sous-graphe du graphe initial par ajout successif des arêtes. A chaque pas, une nouvelle arête qui ne crée pas de cycle, et dont le poids est minimum, est ajoutée à la liste finale (Algorithme 4.1). Lorsque toutes les arêtes du graphe initial ont été testées, le graphe final constitue l'arbre de poids minimum.

Algorithme 4.1 Algorithme de Kruskal

Entrées : Un graphe $G = (V, E)$, contenant un ensemble de nœuds $V = \{v_x\}$ et un ensemble d'arêtes E

Résultat : Un arbre F de poids minimum

1: Classer les arêtes par ordre de longueur croissante $(e_1, e_2, \cdots, e_M)$
2: $F = \emptyset$
3: **Pour** $i = 1, \cdots, M$ **Faire**
4: **Si** $F \cup \{e_i\}$ est un arbre **alors**
5: poser $F = F \cup \{e_i\}$
6: **Fin si**
7: **Fin faire**

Le temps d'exécution de l'algorithme de Kruskal est $O(|E| log|V|)$.

Prim

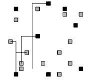
Peu de temps après, en 1957, Prim publie un article [187] où il présente une autre méthode de création de l'arbre couvrant minimal. Avant de présenter son algorithme, nous introduisons quelques termes nécessaire à sa compréhension. Un *terminal isolé* est un terminal qui, à un instant du déroulement de la construction, n'est connecté à aucun autre. Un *fragment* est un ensemble de terminaux interconnectés par des liens directs, et finalement un *fragment isolé* est un fragment qui n'est connecté à aucun terminal externe à lui-même[2].

Son algorithme est basé sur deux principes simples :

- **Principe 1 :** N'importe quel terminal isolé peut être connecté à un plus proche voisin.
- **Principe 2 :** N'importe quel fragment isolé peut être connecté à un plus proche voisin par l'arête disponible la plus courte.

En appliquant ces deux principes judicieusement, l'arbre couvrant minimal est construit très facilement. Au départ, on sélectionne un terminal du graphe, que l'on

[2]Si nous avons quatre nœuds a, b, c, d reliés comme ceci : $a - b - c$, avec d comme terminal isolé, alors $a - b$, $b - c$, et $a - b - c$ sont des fragments, et $a - b - c$ est un fragment isolé.



connecte avec le terminal qui lui est relié par l'arête de poids le plus faible. Ce premier lien constitue un fragment isolé. A chaque pas suivant, il suffit de choisir l'arête de poids le plus faible reliant un nœud au fragment isolé existant, et d'ajouter ce nœud au fragment.

La simplicité de l'algorithme en rend l'exécution manuelle possible, et l'auteur va même jusqu'à suggérer qu'il pourrait être utile dans le cas où un message doit être passé à tous les membres d'un organisation secrète sans qu'il ne soit intercepté par l'ennemi. Pour ce faire, une probabilité d'interception est définie entre chaque couple de membres, et sert de poids aux arêtes. L'arbre couvrant minimal permet alors de trouver le moyen de faire passer le message à tous les membres en minimisant le risque d'être pris par l'ennemi.

La formulation initiale de Prim ne fut pas des plus formalisées, mais nous retranscrivons ici sa déclaration formelle (Algorithme 4.2).

Algorithme 4.2 Algorithme de Prim

Entrées : Un graphe $G = (V, E)$, contenant un ensemble de nœuds $V = \{v_x\}$ et un ensemble d'arêtes $E = \{e_{ij}\}$

Résultat : Forêt F de poids minimum

1: $F =$ ensemble vide
2: $S = \{s$, un sommet de V$\}$
3: **Tant que** S différent de V **Faire**
4: Trouver e_{ij} avec un poids minimum entre $v_i \in S$ et $v_j \in V - S$
5: $F = F \cup e_{ij}$
6: $S = S \cup v_j$
7: **Fin tant que**

Nous pouvons d'ores et déjà noter que les algorithmes de Kruskal et de Prim ne se prêtent pas instinctivement à une implémentation matérielle décentralisée. En effet, si nous partons d'un système où les nœuds sont des cellules, une vision globale est nécessaire, pour trier correctement les arêtes en fonction de leur poids, ou au moins pour pouvoir comparer tous les poids des arêtes reliées à un nœud.

4.3.2 Plus court chemin

Les algorithme de Kruskal et de Prim ne permettent que de trouver l'arbre couvrant minimal d'un graphe, c'est-à-dire qu'ils connectent forcément tous les nœuds ensemble, dans un même sous-graphe. Bien qu'utiles dans certaines applications, ils ne permettent pas de trouver le chemin le plus court entre deux points du graphe.

La première méthode de résolution de ce problème que nous avons pu trouver est à attribuer à Ford et Fulkerson. En 1956 [72], ils proposent une méthode pour trouver le flot maximal entre deux points dans un graphe non orienté, et constatent que trouver le plus court chemin entre deux points dans un graphe G revient à trouver le flot maximal entre ces deux points dans le graphe dual de G.

Dantzig

Alors que le résultat de Ford et Fulkerson n'est vu que comme une dualité du problème de flot maximal, dans un article de 1957, Dantzig [51] résout directement le plus

court chemin, et propose une approche basée sur la résolution, fort joliment illustrée, d'un problème de contraintes, qui peut être résolu grâce à la méthode du simplex [50]. Son algorithme débute par la création d'un arbre quelconque incluant tous les nœuds du graphe. A chaque nœud est assigné une valeur correspondant à sa distance au nœud d'origine. Les arêtes qui ne sont pas présentes dans l'arbre y sont ajoutées petit à petit, si elles font décroître la distance à l'origine des autres points. Lorsqu'une arête est ajoutée, une autre en est enlevée, afin de conserver la structure d'arbre, et le processus continue jusqu'à ce que plus aucune arête ne puisse être ajoutée. Le plus court chemin entre le point d'origine et tous les autres est alors disponible en suivant les arêtes de l'arbre.

Bellman

Une année plus tard, Bellman [22] propose une approche par la programmation dynamique, dont l'application est quasiment identique à celle de Dantzig. Les poids des arêtes sont stockés dans une matrice T, de taille $N \times N$, le poids entre le nœud i et j étant caractérisé par t_{ij}, et le but de l'algorithme est de trouver le chemin le plus court [3] entre le nœud 1 et le nœud N.

Définissons :

$$\begin{aligned} f_i &= \text{le coût requis pour aller de } i \text{ à } N\text{, pour } i = 1, 2, \cdots, N-1, \\ &\quad \text{en utilisant une politique optimale,} \\ \text{avec } f_N &= 0. \end{aligned} \tag{4.1}$$

Il suffit, pour trouver la solution, d'appliquer itérativement les équations suivantes :

$$\begin{aligned} f_i^{(k+1)} &= \underset{j \neq i}{Min}[t_{ij} + f_i^{(k+1)}]\,, \quad i = 1, 2, \cdots, N-1, \\ \text{avec } f_N^{(k)} &= 0\,, \\ \text{et } f_i^{(0)} &= \underset{j \neq i}{Min}\, t_{ij}\,, \quad i = 1, 2, \cdots, N-1 \end{aligned} \tag{4.2}$$

pour $k = 0, 1, 2, \cdots, N-1$

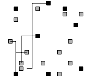
Le problème de cette méthode, qui procède par approximations successives, est le temps de calcul, qui nécessite $N-1$ pas, où chaque pas nécessite $O(N^2)$ opérations.

Dijkstra

Suivant de près Bellman, Dijkstra fut le second, en 1959, à proposer un algorithme cité par tous, dans [59]. Il y propose une manière de trouver le chemin de poids le plus faible entre deux points d'un graphe non-orienté pondéré dont les arêtes ont un poids positif ou nul. Peut de temps après, en 1960, Whiting et Hillier [253] on présenté le même algorithme, en suggérant que la méthode pouvait aisément être appliquée aux graphes orientés.

Nous avons traduit la partie de l'article de Dijkstra traitant de ce problème dans l'algorithme 4.3. Cette première version littéraire correspond à l'algorithme 4.4, qui présente une facette nettement plus mathématique. Le temps d'exécution de l'algorithme est de $O(|V|^2)$, et peut être réduit à $O(|E| log|V|)$ suivant l'implémentation.

[3]Nous utilisons le terme chemin le plus court comme synonyme du chemin le moins coûteux.



Algorithme 4.3 Algorithme du plus court chemin de Dijkstra (version traduite)

On considère n points (nœuds), dont certaines paires sont connectées par une arête ; la longueur de chaque arête est donnée. Nous nous restreignons au cas où il existe au moins un chemin entre n'importe quels deux nœuds.
Problème : Trouver le chemin ayant la distance totale minimale entre deux nœuds P et Q.
Nous utilisons le fait que, si R est un nœud sur le chemin minimal de P à Q, la connaissance de ce dernier implique la connaissance du chemin minimal entre P et R. Dans la solution présentée, les chemins minimaux entre P et n'importe quel autre nœud sont construits par ordre de longueur jusqu'à ce que Q soit atteint.
Les nœuds sont subdivisés en trois ensembles :

- A. Les nœuds pour lesquels le chemin minimum depuis P est connu ; les nœuds seront ajoutés à cet ensemble par ordre de longueur de chemin minimum depuis P ;
- B. Les nœuds candidats à un transfert dans l'ensemble A. Cet ensemble comprend tous les nœuds qui sont connectés à au moins un nœud de l'ensemble A, mais qui ne sont pas dans A eux-mêmes ;
- C. Les autres nœuds.

Les arêtes sont également subdivisés en trois ensembles :

- I. Les arêtes permettant de construire les chemins les plus courts entre P et les nœuds de l'ensemble A ;
- II. Les arêtes candidates à un transfert dans l'ensemble I ; Une et une seule des arêtes de cet ensemble est connectée à chaque nœud de l'ensemble B ;
- III. Les autres arêtes (rejetées ou non encore considérées).

Au départ de l'algorithme, tous les nœuds sont dans l'ensemble C et toutes les arêtes sont dans l'ensemble III. Nous transférons maintenant le nœud P dans l'ensemble A et répétons ensuite les points suivants.
Pas 1. Considérons toutes les arêtes connectées au nœud qui vient d'être transféré dans l'ensemble A avec les nœuds R dans l'ensemble B ou C. Si le nœud R est dans l'ensemble B, nous vérifions si l'arête r crée un chemin plus court de P à R que le chemin connu qui utilise les arêtes de l'ensemble II. Si ce n'est pas le cas, l'arête r est rejetée ; si toutefois l'arête r crée une connexion plus courte entre P et R que celle déjà obtenue, elle remplace la branche correspondante dans l'ensemble II et cette dernière est rejetée. Si le nœud R est dans l'ensemble C, il est ajouté à l'ensemble B et l'arête r est ajoutée à l'ensemble II.
Pas 2. Chaque nœud de l'ensemble B peut être connecté au nœud P par un seul chemin si nous nous restreignons aux arêtes de l'ensemble I et d'une de l'ensemble II. En ce sens, chaque nœud de l'ensemble B connaît sa distance au nœud P ; le nœud ayant la distance minimum à P est transféré de l'ensemble B à l'ensemble A, et l'arête correspondante est transférée de l'ensemble II au I. Nous retournons ensuite au pas 1 et répétons le processus jusqu'à ce que le nœud Q soit transféré dans l'ensemble A. La solution a ensuite été trouvée.

Algorithme 4.4 Algorithme du plus court chemin dans un graphe de Dijkstra

Entrées : Un graphe non-orienté $G = (V, E)$, des coûts c_{ij} sur les arêtes, et deux nœuds v_a et v_b
Résultat : Le plus court chemin entre v_a et v_b

1: $Val(v_j) = c_{aj}$, pour tout $v_j \in V$
2: $T = V; P = \{v_a\}; Val(v_a) = 0$
3: **Répéter**
4: Trouver $v_t \in P$ tel que $Val(v_t) < Val(v_i)$, pour tout autre $v_i \in P$
5: **Si** $v_t = v_b$ **alors**
6: Remonter le chemin des prédécesseurs de v_b à v_a, et sortir de l'algorithme
7: **Fin si**
8: $T = T - \{v_t\}$
9: $P = P \cup \{v_t\}$
10: **Pour tout** $v_j \in T$ **Faire**
11: **Si** $Val(v_t) + c_{tj} < Val(v_j)$ **alors**
12: $Val(v_j) = Val(v_t) + c_{tj}$
13: Prédécesseur de $v_j = v_t$
14: **Fin si**
15: **Fin faire**
16: **Jusqu'à** ce que T soit vide
17: Il n'y a pas de chemin entre v_a et v_b

La figure 4.4 présente l'exécution de l'algorithme de Dijkstra (Algorithme 4.4) pour un graphe simple dans lequel nous voulons trouver le chemin minimal entre le

nœud présent en haut à gauche, et celui positionné en bas à droite.

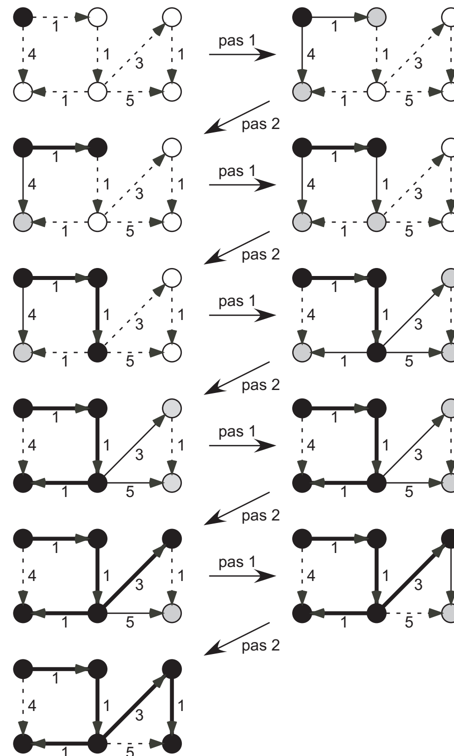


Figure 4.4 : *Exemple d'exécution de l'algorithme de Dijkstra.*

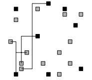
Il est intéressant de noter que Pollack et Wiebenson, dans un article de 1960 [186], attribuent la paternité d'un algorithme quasiment semblable à Minty, en ne citant comme référence qu'une "Personal Communication". Agissant cette fois sur un graphe orienté, il est présenté par l'algorithme 4.5.

Algorithme 4.5 Algorithme du plus court chemin dans un graphe de Minty

Entrées : Un graphe orienté $G = (V, E)$, et deux nœuds v_a et v_b
Résultat : Le plus court chemin entre v_a et v_b
1: $Val(v_a) = 0$
2: **Tant que** v_b n'est pas atteint **Faire**
3: **Pour** toute arête e_{ij} telle que la valeur de v_i a été définie et la valeur de v_j ne l'a pas été **Faire**
4: Calculer $Val(v_i) + c(e_{ij})$
5: **Fin faire**
6: Sélectionner l'arête e_{ij} pour laquelle cette somme est minimale
7: $Val(v_j) = Val(v_i) + c(e_{ij})$
8: **Fin tant que**

Moore

La même année que Dijkstra, Moore [165] écrit un article dans lequel il présente quatre algorithmes, dont un résolvant le même problème que Dijkstra, mais avec une



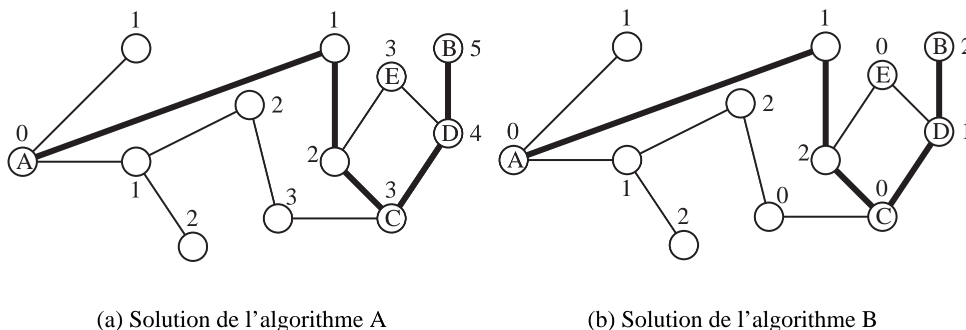
efficacité moindre. Nous ne réquisitionnerons toutefois pas plus de lignes sur des algorithmes similaires à celui de Dijkstra, pour lequel d'autres optimisations ont été effectuées ultérieurement par différentes équipes [41, 183]. En effet, le routage dans les circuits électroniques, que nous allons entre autre développer dans le chapitre suivant, traite, de manière générale, du plus court chemin entre deux points, et ce dans une grille de points. Ce problème est un cas particulier des trois autres algorithmes de Moore, qui sont appliqués à des graphes non pondérés (dont les poids des arêtes sont identiques). Ils permettent donc de trouver le plus court chemin entre deux points du graphe avec différentes implémentations plus ou moins coûteuses en terme de mémoire et de temps d'exécution.

Algorithme A Pour trouver le chemin le plus court entre un point A et un point B, l'algorithme donne tout d'abord la valeur 0 au point A. Au pas suivant, la valeur 1 est assignée à tous les points reliés au point A. Au pas suivant, tous les points non assignés reliés aux points de valeur 1 se voient assigner la valeur 2. Et ainsi de suite, jusqu'à arriver au point B. La valeur du point B donne sa distance au point A, et il suffit de revenir en arrière, en suivant la décrémentation des numéros jusqu'à arriver à A, pour trouver le chemin le plus court (Algorithme 4.6). Dans le cas où plusieurs chemins sont possibles, comme à partir du point D de la figure 4.5(a), qui peut choisir entre C et E, il suffit d'en prendre un au hasard.

Algorithme 4.6 Algorithme A du plus court chemin de Moore

Entrées : Un graphe $G = (V, E)$, et deux nœuds v_a et v_b
Résultat : Le plus court chemin entre v_a et v_b

1: $i = 0$
2: **Tant que** v_b n'est pas atteint **Faire**
3:     **Pour** tout nœud v_j non visité ayant une arête reliée à un nœud de valeur i **Faire**
4:         $Val(v_j) = i + 1$
5:     **Fin faire**
6:     $i = i + 1$
7: **Fin tant que**
8: Parcourir le chemin de v_b à v_a



(a) Solution de l'algorithme A (b) Solution de l'algorithme B

Figure 4.5 : *Le chemin le plus court (trait gras) entre deux points A et B, dans un graphe non pondéré, par les algorithmes de Moore.*

Algorithme B En 1959, la puissance des ordinateurs n'était pas ce que l'on peut qualifier d'exceptionnelle. Les développeurs devaient donc faire particulièrement attention à la taille de leurs programmes, et surtout à la mémoire nécessaire à leur exécution. Dans cette optique, Moore propose une amélioration de son algorithme : au lieu de stocker un entier pour chaque nœud, il est possible de ne stocker que la valeur de cet entier modulo 3, c'est-à-dire une valeur pouvant être 0, 1, ou 2. De cette manière, seulement 2 bits sont nécessaires pour chaque nœud, et le chemin entre B et A peut être retrouvé en décrémentant les valeurs modulo 3. Sur la figure 4.5(b), nous pouvons observer, par exemple, que depuis le point C, qui a la valeur 0, le point avec valeur 2 sera choisi.

Algorithme C Toujours poussé par le souci de minimiser la taille de la mémoire nécessaire, il va jusqu'à proposer une implémentation ne nécessitant qu'un bit par nœud. Ce bit indique si le nœud a été visité ou non, et est initialement placé à 0 dans tous les nœuds. Le principe est relativement semblable aux approches précédentes, et part du point A, en y plaçant la valeur 1. A chaque pas, tous les nœuds dont la valeur est 0 et qui sont reliés à un nœud de valeur 1 voient leur valeur passer à 1, et ce jusqu'à trouver le point B. La liaison ayant permis d'arriver à B est alors mémorisée comme faisant partie du plus court chemin, et le processus est relancé, pour trouver le plus court chemin entre A et le prédécesseur de B. Au total, le processus est lancé d fois, si la distance entre A et B vaut d. Il est donc nettement plus coûteux en terme de temps d'exécution que les variantes A et B.

Les trois approches de Moore sont intéressantes dans le sens où elles offrent une possibilité de parallélisation par un système synchrone. En effet, les lignes 3-5 de l'algorithme 4.6 peuvent sans autre être exécutées en parallèle.

4.3.3 Algorithme de Lee

Le plus influant des articles sur le sujet des algorithmes de routage est sans nul doute celui de Lee [137], qui est cité par tous ses successeurs. Il y présente une méthode de spécification de problèmes de traitement de patterns, qu'il applique à la création de chemins dans un graphe régulier. Nous allons décrire son formalisme, et nous montrerons que sa solution au plus court chemin est identique à celle proposée par Moore.

Commençons par définir un ensemble de cellules (Lee est le premier à introduire le terme *cellule*) : $C = \{c^1, c^2, \cdots\}$. Pour chaque cellule $c^i \in C$, nous définissons son *1-voisinage* $N(c^i) = \{c^i_1, c^i_2, \cdots, c^i_n\}$, qui doit obéir aux deux règles suivantes :

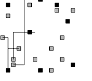
N1) Tout 1-voisinage a exactement n cellules, où $n \geq 1$.

N2) $c^j \in N(c^i) \Leftrightarrow c^i \in N(c^j)$, qui signifie que le graphe est non-orienté.

Basé sur la fonction N, nous définissons, pour toute cellule c^i ayant un voisinage $N(c^i) = \{c^i_1, c^i_2, \cdots, c^i_n\}$, la fonction $d_k(c^i) = c^i_k$, pour $k = 1, 2, \cdots, n$. Dès lors, $d_k(c^i)$ est la k-ème cellule du 1-voisinage.

Définissons un ensemble fini de symboles $S = \{s^1, s^2, \cdots, s^m\}$.

Définissons un mapping Γ de C dans $C \times S$, qui associe un symbole $s^j \in S$ à chaque cellule $c^i \in C$. Nous devons donc, pour chaque problème particulier, définir $\Gamma(c^i) = (c^i, s(c^i))$, qui peut, dans l'exemple du routage, indiquer si une cellule est libre ou occupée.



Pour deux cellules c^i et c^j distinctes, nous définissons un chemin entre ces deux cellules par $p(c^i, c^j) = \{c^0 = c^i, c^1, c^2, \cdots, c^m = c^j\}$, de manière à ce que $c^{i+1} \in N(c^i)$ pour tout $i = 0, 1, \cdots, m-1$. $\pi(c^i, c^j)$ est alors l'ensemble de tous les chemins entre c^i et c^j.

Définissons une carte d'admission M, du domaine $\pi(c^i, c^j)$ à l'ensemble $\{0, 1\}$. Chaque chemin tel que $M(p(c^i, c^j)) = 1$ est défini comme étant admissible, les autres étant inadmissibles, et l'ensemble des chemins admissibles est appelé $\pi^*(c^i, c^j)$.

Avec toutes ces définitions, nous disposons d'un quintuple (C, S, N, Γ, M), que nous appelons un $\mathcal{C} - space$.

Dans le cas de la création de chemins, qui nous intéresse, nous devons encore définir un vecteur de fonctions $F = (f_1, f_2, \cdots, f_r)$, où chaque fonction f_i est définie de $\pi^*(c^i, c^j)$ dans $\mathbb{N}$. f_i nous donne une fonction de coût du chemin entre deux cellules, et le but de l'algorithme est alors de trouver le chemin de plus bas coût entre deux cellules données.

Pour que l'algorithme fonctionne, il faut que les fonctions f soient monotones, c'est-à-dire que si $p(c^i, c^k)$ est un sous-chemin de $p(c^i, c^j)$, alors $f(p(c^i, c^k)) \leq f(p(c^i, c^j))$.

Durant l'exécution de l'algorithme, une masse cellulaire $(m_1, m_2, \cdots, m_r)$ est associée à chaque cellule. Une masse cellulaire $m = (m_1, m_2, \cdots, m_r)$ sera plus petite que la masse $m' = (m'_1, m'_2, \cdots, m'_r)$ si $m_i = m'_i$ pour $i = 0, 1, \cdots, k$, mais que $m_{k+1} < m'_{k+1}$, avec $0 \leq k \leq r$.

Se basant sur toutes ces définitions, l'algorithme 4.7 présent la solution générale de Lee au problème du routage d'un chemin.

Création de chemin dans une grille Pour le problème de trouver le plus court chemin dans une grille régulière de cellules reliées à leurs quatre voisines, nous posons les définitions suivantes :

Les fonctions de coordonnées d_1, d_2, d_3,et d_4, pour une cellule c^i, sont définies ainsi : $d_1(c^i)$ est la cellule en haut, $d_2(c^i)$ est la cellule de droite, $d_3(c^i)$ est la cellule du bas, et $d_4(c^i)$ est la cellule de gauche.

Si une cellule peut être occupée ou inoccupée, nous pouvons définir M comme ceci : Pour un chemin $p(c^*, c^{**}) = \{c^0 = c^*, c^1, \cdots, c^{n-1}, c^n = c^{**}\}$, $p(c^*, c^{**})$ est admissible, c'est-à-dire, $M(p(c^*, c^{**}) = 1$, si $s(c^i)$ est inoccupée, pour $1 \leq i \leq n$.

Ensuite, le vecteur F doit être défini en fonction du problème à résoudre. Par exemple, pour trouver un chemin minimisant le contact avec des cellules occupées, nous pouvons définir F comme étant une seule fonction f comme ceci :

1) $f(p(c^*, c^*)) = 0$.

2) Si $s(c^i)$ est inoccupée, alors

$$f(p(c^*, c^i)) = \begin{cases} (min\{f(p(c^*, c^j))|c^j \in N(c^i)\}) + R(c^i) \\ \text{pour tout} f(p(c^*, c^j))\text{ayant été définie,} \\ \text{et est indéfinie sinon} \end{cases} \quad (4.3)$$

Où $R(c^i) =$ le nombre de cellules $c^j \in N(c^i)$ qui sont occupées.

Pour le problème du plus court chemin, nous pouvons définir F comme étant une seule fonction f comme ceci :

1) $f(p(c^*, c^*)) = 0$.

2) Si $s(c^i)$ est inoccupée, alors

Algorithme 4.7 Algorithme de Lee

Entrées : $\mathcal{C} - space$ (C, S, N, Γ, M), un vecteur $F = (f_1, f_2, \cdots, f_r)$, une cellule initiale c^* et une cellule finale c^{**}

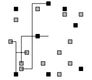
Résultat : Le chemin de coût le plus faible entre c^* et c^{**}

1: $L = \{c^*\}$
2: Toutes les cellules de C reçoivent une masse $(0, 0, \cdots, 0)$
3: **Tant que** $c^{**} \notin L$ ou $L \neq \emptyset$ **Faire**
4: $L1 = \emptyset$
5: **Pour tout** cellule $c \in L$ **Faire**
6: déterminer l'ensemble des cellules admissibles $\{c^i\} \in N(c)$ dont la masse n'a pas été déterminée
7: Ajouter cet ensemble à L1
8: **Fin faire**
9: **Pour tout** cellule $c^i \in L1$ **Faire**
10: Calculer une masse cellulaire possible, en trouvant parmi les voisins $c^j \in N(c^i)$ le r-tuple minimal $(f_1(p(c^*, c^j, c^i)), \cdots, f_r(p(c^*, c^j, c^i)))$
11: **Fin faire**
12: Les masses des cellules ayant une masse possible minimum $\{c^j\}$ sont mises à jour avec celle-ci
13: Pour chacune de ces cellules, la coordonnée d'origine de l'expansion est stockée
14: $L = L \cup \{c^j\}$
15: Enlever de L les cellules dont toutes les voisines ont une masse calculée
16: **Fin tant que**
17: **Si** $L = \emptyset$ **alors**
18: Il n'y a pas de solution
19: **Sinon**
20: Partant de c^{**}, parcourir la chaîne de coordonnées jusqu'à atteindre c^*
21: **Fin si**

$$f(p(c^*, c^i)) = \begin{cases} (min\{f(p(c^*, c^j)) | c^j \in N(c^i)\}) + 1 \\ \text{pour tout} f(p(c^*, c^j)) \text{ayant été définie,} \\ \text{et est indéfinie sinon} \end{cases} \tag{4.4}$$

Il est intéressant de noter que Lee, dans son article, ne propose pas directement de solution pour le plus court chemin, mais une pour minimiser la distance totale ET le contact avec des cellules occupées. Pour ce faire, il utilise deux fonctions, dont une est celle de l'équation 4.4. Les lecteurs ont donc extrapolé que le plus court chemin était créé avec cette fonction-ci, et c'est cet algorithme qui est toujours cité. Or, par le mécanisme de son algorithme général, disposer d'une simple fonction $f(p(c^*, c^i)) = 0$ suffit à trouver le plus court chemin entre deux points. Ceci évite de devoir stocker la distance à l'origine de chaque point, ce qui est toujours reproché à Lee, et ne nécessite que de mémoriser le prédécesseur de chaque cellule.

Cette dernière version est cruellement semblable à celle de Moore, mais appliquée à une grille régulière d'éléments connectés à leurs n voisins. Son approche y est, comme nous l'avons vu, nettement plus mathématique, alors que Moore ne le présentait que d'une façon littéraire. L'algorithme de Lee permet en outre de placer des contraintes comme par exemple la minimisation non pas seulement de la distance mais



également du nombre de croisements de chemins.

A partir de maintenant, nous nous référons au cas de la recherche du plus court chemin grâce à la fonction 4.4 comme étant l'algorithme de Lee. Appliqué à une grille d'éléments reliés à leurs quatre voisins, il fonctionne à la manière dont une vague se propage à partir de la source, créant une structure en forme de diamant. Dans ce que nous appellerons dorénavant la phase d'expansion, un front d'onde atteint les cellules les unes après les autres et lorsqu'une cellule est touchée par le front d'onde, elle stocke sa distance à la source en incrémentant de 1 la distance de la cellule par laquelle l'expansion est arrivée. La phase d'expansion se termine lorsque la destination est atteinte, signe que le chemin est trouvé. Il ne reste plus alors qu'à remonter jusqu'à la source en passant d'une cellule de distance d à une de distance $d-1$ jusqu'à arriver à la source, qui a une distance de 0 (phase de rétropropagation). La figure 4.6 montre, pour un exemple, les pas de l'algorithme lors de l'expansion, ainsi que le chemin généré.

L'aspect fondamental de l'algorithme de Lee réside en son parallélisme intrinsèque. En effet, dans sa présentation, Lee propose une approche séquentielle, où les cellules sont ajoutées au front d'onde les unes après les autres, mais il est clair qu'avec un système parallèle la phase d'expansion pourrait être plus efficace. Alors que la solution séquentielle a une complexité de $O(d^2)$ pour une distance entre la source et la destination de d, la solution parallèle offre une complexité de $O(d)$.

4.3.4 Variations sur Lee

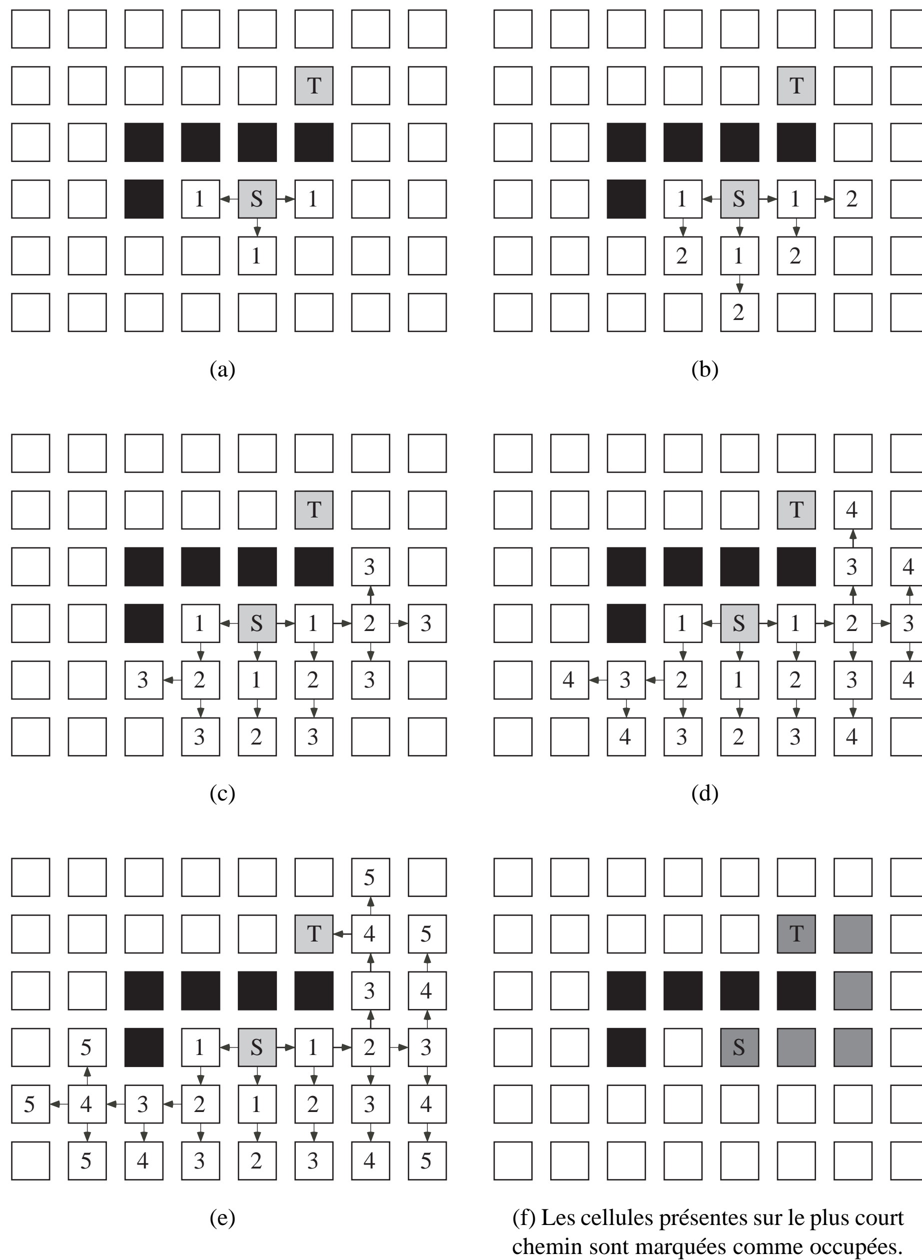
De nombreuses variations sur l'algorithme de Lee ont été proposées. Nous en présentons ici les principales, mais le lecteur pourra en trouver des secondaires dans [42, 92, 93, 94, 128, 163]. La plupart de ces variations, ainsi que l'algorithme principal, ont été implémentées sur des systèmes parallèles dans le but de les optimiser, dont notamment [25, 200, 262].

Akers

En 1967, Akers publie un article [6] où il propose une variante de l'algorithme de Lee, appliquée à une grille rectangulaire de cellules, dans laquelle sont placées des barrières. Au lieu de mémoriser, dans chaque cellule, la distance à l'origine, il lui suffit d'y placer une valeur binaire (Moore pouvait le résoudre avec une valeur ternaire pour le cas d'un graphe non régulier), réduisant ainsi la quantité de mémoire nécessaire à la bonne marche de l'algorithme.

Il part de l'observation suivante : Dans l'algorithme de Lee, une cellule qui a été atteinte par la phase d'expansion et numérotée X, a comme voisines des cellules qui sont soit vides, soit des murs, soit qui contiennent la valeur $X-1$ ou $X+1$. Il suffit donc, pour pouvoir tracer le chemin durant la phase de rétropropagation, que le prédécesseur ait une valeur différente de celle de son successeur, et la séquence $1,1,2,2,1,1,2,2,\cdots$ remplit parfaitement cette condition.

L'algorithme de Lee est donc adapté de manière à placer la séquence susmentionnée au lieu de la séquence $1,2,3,4,5,\cdots$ dans les cellules visitées. Il ne reste plus qu'à la phase de rétropropagation de parcourir la séquence dans l'ordre correct, en fonction de la manière $(11,12,21,$ ou $22)$ dont l'expansion a atteint le point B (Figure 4.7).



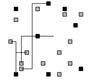
(a) (b) (c) (d) (e) (f) Les cellules présentes sur le plus court chemin sont marquées comme occupées.

Figure 4.6 : *Déroulement de l'algorithme de Lee (S est une source, et T est une destination).*

Rubin

Prenant encore plus de recul, en 1974, Rubin [197] propose des améliorations à l'algorithme de Lee dans l'optique de réduire le temps de calcul. Les deux plus importantes sont les suivantes :

Recherche bidirectionnelle Une solution de Pohl suggère de lancer la phase d'expansion depuis la source ET la destination. De cette manière, le nombre de cellules



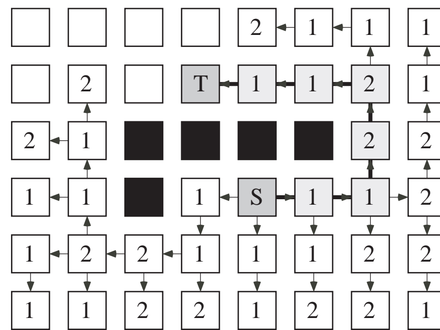


Figure 4.7 : *Le plus court chemin trouvé grâce à l'algorithme de Akers.*

explorées peut être réduit, la somme des cellules visitées par les deux vagues étant plus petite que celle d'une seule vague.

Profondeur d'abord Une autre méthode, développée par Rubin, utilise une recherche en profondeur d'abord. L'expansion se poursuit prioritairement dans la direction de la destination, tant qu'aucun obstacle n'est présent. Le temps d'exécution peut être grandement réduit par cette approche, mais une connaissance globale de la position de la destination est nécessaire, ce qui n'est pas la panacée dans le cas d'une implémentation matérielle.

Hoel

En 1976, Hoel [100] propose deux nouvelles variations à l'algorithme de Lee, pour des coûts de chemins quelconques. La première est purement algorithmique, vouée à une implémentation logicielle, et propose d'utiliser plusieurs listes pour le stockage des cellules présentes sur la frontière, afin d'améliorer la rapidité de la recherche de la prochaine cellule à étendre. La deuxième est très intéressante, et vise à réduire la mémoire nécessaire à l'exécution de l'algorithme. Alors que l'approche de Akers qui plaçait des 1 et des 2 dans les cellules atteintes par l'expansion n'est valable que dans le cas où il n'y a pas de coûts différents entre les cellules, celle de Hoel est applicable à des coûts différenciés. Seuls 2 bits par cellule sont nécessaires, et permettent de définir si la cellule est inoccupée, occupée, ou sur le front, de même que l'origine de l'expansion : occupé=tout droit=0, frontière=virage à gauche=1, inoccupé=2, virage à droite=3. Cette réduction, qui économise un maximum la mémoire, a toutefois un coût. Durant l'exécution de la création d'un chemin, la configuration initiale (cellules inoccupées ou occupées) est perdue, et une mémoire secondaire où le tableau initial (avec les cellules occupées) est stocké est nécessaire. Après chaque création d'un nouveau chemin, il doit être mis à jour avec les nouvelles cellules occupées. Finalement, il est intéressant de noter que cette approche peut être appliquée à un voisinage de 6, sans que des bits supplémentaires ne soient nécessaire (voir l'article [100] pour plus de détails).

Mikami et Tabuchi

En 1968, Mikami et Tabuchi [156] introduisent un nouvel algorithme, appelé line-search, pour le routage d'un circuit imprimé dont un des côtés peut contenir des lignes

horizontales, et l'autre des verticales, les deux couches étant reliées par des trous. Leur but est d'offrir une alternative moins coûteuse en terme de temps d'exécution et de mémoire à celui de Lee et de Akers. Au lieu de travailler sur une grille de cellules, l'algorithme stocke des lignes, déterminées par seulement trois coordonnées. La recherche s'effectue alternativement depuis la source et la destination, par l'extension de lignes étendues dans les quatre directions. Les lignes originaires de la source sont comparées à celles originaires de la destination afin de détecter la fin de l'algorithme, lorsque deux lignes se croisent. En outre, l'algorithme exploite les lignes déjà connectées à la source ou à la destination, rendant le mécanisme de liaison, initialement point-à-point, point-à-chemin ou chemin-à-chemin. De cette manière, il évite de créer de nouvelles lignes qui risqueraient de surcharger la carte inutilement.

Cet algorithme garantit de trouver une solution si elle existe, mais ne fournit pas forcément le chemin le plus court entre les deux points, mais le chemin le plus simple, c'est-à-dire avec le moins de coins possibles. Bien que sa structure, telle que présentée dans [156], ne se prête pas à une implémentation parallèle, son concept peut l'être, et nous nous en inspirerons en page 128.

Backtracking

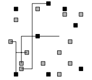
Le routage à la manière de Lee peut conduire à une impasse si trop de chemins doivent être créés. Cette congestion peut être évitée par l'utilisation de backtracking. Agrawal et Breuer, en 1977, introduisent un algorithme basé sur deux principes supplémentaires : le backtracking et la déviation. Les connexions y sont réalisées dans un ordre quelconque, mais après chaque ajout, un test détermine si les connexions restantes sont forcément bloquées par celui-ci. Si c'est le cas, l'ajout est supprimé et un autre chemin est cherché. Dans le cas où aucun des plus courts chemins n'est acceptable, une déviation de taille δ est autorisée, et agrandit le chemin de δ cellules.

Cet algorithme, dont tous les détails peuvent être trouvés dans [5], garantit de trouver une solution au problème général du routage de N chemins, si une telle solution existe. Toutefois, le prix à payer pour une telle réussite est le temps de calcul, qui peut s'avérer prohibitif, de par les nombreuses phases potentielles de backtracking.

Soukup

En 1978, Soukup [223] propose une amélioration de l'algorithme de Lee, qui exploite une recherche en profondeur d'abord, quasiment de la même manière que Rubin. L'expansion se poursuit dans le sens de la destination, tant qu'aucun obstacle n'est rencontré. Lorsque cette expansion de type line-search n'est plus possible, l'algorithme de Lee prend le pas, jusqu'à ce que l'obstacle ait été contourné. Le temps d'exécution peut être grandement réduit par cette approche, mais une connaissance globale de la position de la destination est nécessaire, ce qui n'est pas la panacée dans le cas d'une implémentation matérielle.

Les figures 4.8 et 4.9 illustrent la différence entre l'approche de Soukup et celle de Lee dans le nombre de cellules visitées lors de la recherche de la destination. Dans la figure 4.8, les points noirs ont été atteints par une expansion de type line-search, alors que les blancs l'ont été par une expansion de type Lee.



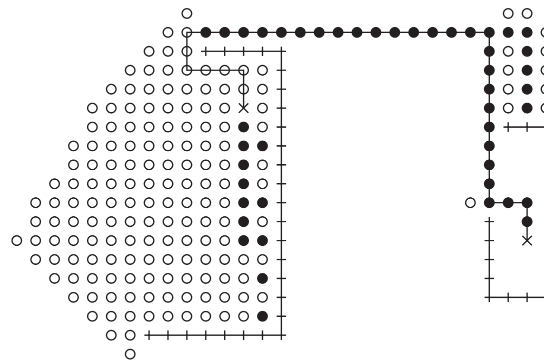


Figure 4.8 : *Nombre de cellules visitées par l'algorithme de Soukup.*

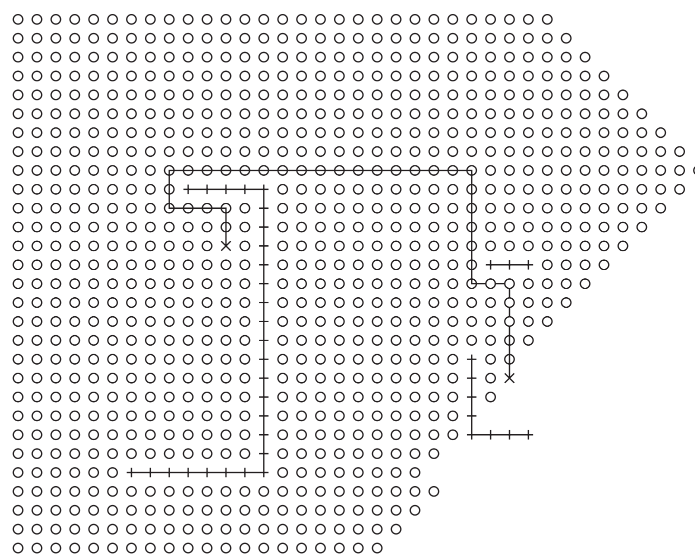


Figure 4.9 : *Nombre de cellules visitées par l'algorithme de Lee, à comparer avec l'approche de Soukup, figure 4.8.*

Watanabe

Bien plus tard, en 1986, Watanabe et Sugiyama [251], présentent un algorithme de routage parallèle, qu'ils appellent PAR, pour Parallel Adaptable Routing. Leur première variante, PAR-1, n'est qu'une adaptation de l'algorithme de Lee, avec comme paramètre supplémentaire la taille de la vague d'expansion. A chaque pas, les cellules présentes sur le front d'onde s'étendent de D_{ex} cellules. Dès lors, si $D_{ex} = 1$, leur algorithme correspond à celui de Lee, et si $D_{ex} = \infty$, il est semblable au line-search de Mikami.

Un deuxième algorithme, PAR-2, présente, lui, une intéressante solution pour les connexions multipoints, où une source doit être connectée à plusieurs destinations. Son but est de construire un arbre de Steiner quasi-minimum (Définition 4.2) entre les différents points, afin de minimiser le nombre de ressources nécessaires. Pour y parvenir, une phase d'expansion part de la source, jusqu'à atteindre une destination (Figure 4.10(a)). Ensuite, une deuxième vague part de ce point pour retrouver la source (Figure 4.10(b)), et permet de définir l'ensemble des plus courts chemins entre les deux points (Figure 4.10(c)). Pour relier le point suivant, une expansion est lancée depuis

cet ensemble de points (Figure 4.10(d)), et de la même manière une nouvelle zone est créée, les deux zones étant reliées par un point P (Figure 4.10(e)). Il ne reste alors plus qu'à relier les trois points en passant par ce point P (Figure 4.10(f)). Le procédé peut être réitéré dans le cas où plus de trois points sont à connecter, suivant le même principe.

Définition 4.2. *L'arbre de Steiner de poids minimum d'un graphe connecte tous les nœuds de ce graphe par un arbre de poids minimum, qui peut inclure des noeuds supplémentaires appelés points de Steiner (Figure 4.1(b), page 69).*

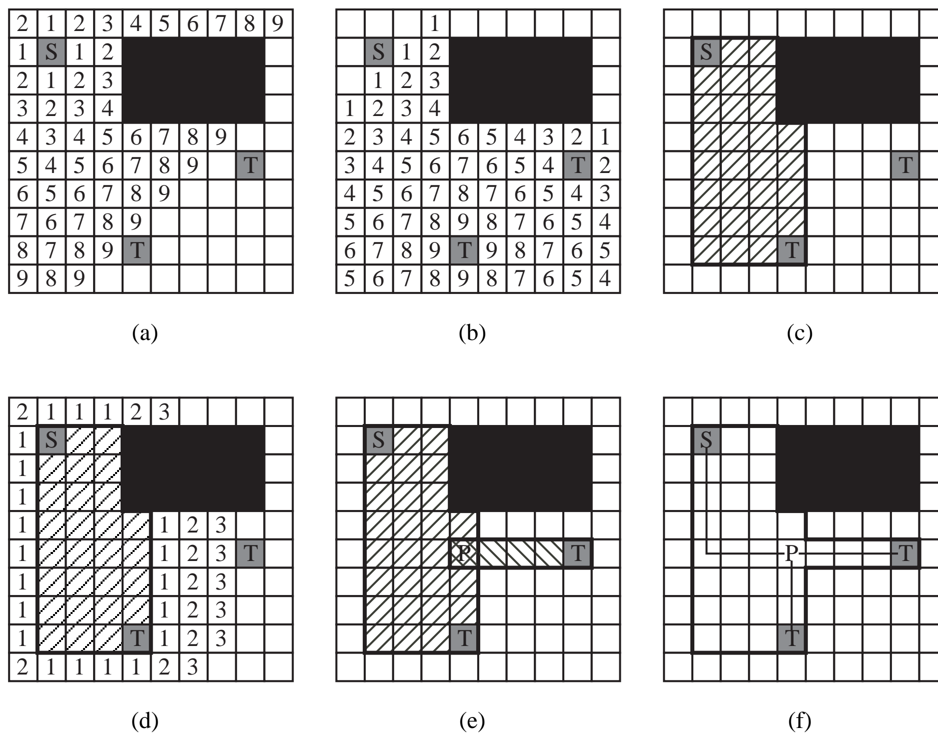
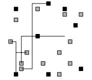
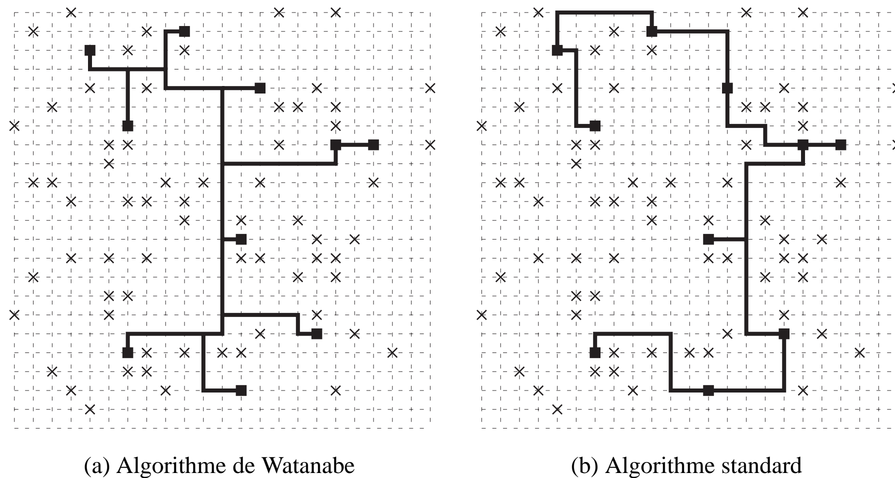


Figure 4.10 : *Principe de base de l'algorithme PAR-2 pour la construction d'un arbre de Steiner quasi-minimum.*

La figure 4.11 illustre la différence entre un réseau de connexions réalisé à l'aide de PAR-2, comparé avec celui créé par un algorithme standard.

Une technique de Rip-up a été ajoutée à leur système, et permet de détruire un chemin déjà créé s'il en bloque un nouveau en phase d'expansion. Cette approche de Rip-up a également été exploitée par d'autres équipes [42, 248], et permet de grandement améliorer le nombre de connexions qui peuvent être routées. Nous noterons toutefois que la parallélisation d'une telle technique a un prix, qui est soit celui d'un contrôle global capable de détecter un conflit entre deux chemins, soit celui d'un ajout non négligeable de complexité au niveau des cellules responsables du routage, qui doivent pouvoir détecter un conflit, et détruire un chemin déjà routé.





(a) Algorithme de Watanabe (b) Algorithme standard

Figure 4.11 : *Comparaison de l'algorithme de Watanabe et de l'approche standard.*

Automates cellulaires

Deux équipes ont présenté, en 1996, une solution au routage grâce aux automates cellulaires à deux dimensions. Contrairement aux algorithmes précédemment exposés, la structure d'un automate cellulaire offre ici un parallélisme intrinsèque, qui peut parfaitement être appliqué à l'algorithme de Lee. Ces deux solutions sont basées sur des automates synchrones, où les états de toutes les cellules sont mis à jour simultanément.

Hochberger/Hoffmann La solution proposée par Hochberger et Hoffmann [98] fait intervenir 14 états, dont 4 pour indiquer l'origine de l'expansion, et 4 pour la configuration du chemin entre la source et la destination. Ils proposent également une solution permettant la gestion de la création de plusieurs chemins en parallèle, basée sur des priorités données à ces chemins. Cette amélioration nécessite 24 états, additionnés à deux registres supplémentaires qui mémorisent une appartenance de la cellule à un chemin, ainsi que la priorité de ce chemin.

Adamatzky Adamatzky [3], quant à lui, présente une solution axée sur la recherche du plus court chemin, capable de gérer des coûts entre les cellules de l'automate. Un coût c y induit un retard d'autant de coups d'horloge dans la propagation du signal d'expansion, et le nombre d'états de cette solution dépend du coût maximal autorisé.

4.3.5 A* et algorithmes évolués

Alors que la plupart des algorithmes précédents effectuent une recherche en largeur d'abord, il est possible de chercher en profondeur d'abord, c'est-à-dire, dans un processus séquentiel, d'étendre le dernier nœud visité plutôt que le premier. Sur le plan de l'implémentation logicielle, ceci correspond à utiliser une liste LIFO (Last In First Out) plutôt que FIFO (First In First Out). La recherche best-first utilise une connaissance de la distance potentielle au but pour choisir le nœud à étendre. Il s'agit toujours du nœud qui est potentiellement le plus proche de la destination. Dans le cas d'une grille à deux dimensions, comme pour le problème qui nous intéresse, la distance à

la destination est simplement la distance de Manhattan (Définition 4.3) entre le nœud courant et la destination. Cet algorithme garantit de trouver une solution si elle existe, mais elle n'est pas forcément la plus courte.

Définition 4.3. *La distance entre deux points du plan,* $A = (a_x, a_y)$ *et* $B = (b_x, b_y)$*, vaut* $|a_x - b_x| + |a_y + b_y|$.

L'algorithme de recherche A* [89, 199], qui est grandement utilisé en intelligence artificielle, est de type best-first, et tire parti d'un heuristique qui combine le coût $g(n)$ du chemin entre la source et le nœud courant avec le coût potentiel $h(n)$ entre ce nœud et la destination : $f(n) = g(n) + h(n)$. Le nœud ayant un $f(n)$ le plus faible est alors celui qui est étendu. Dans notre cas, $g(n)$ est calculé durant l'expansion, et correspond à la distance minimal entre le noœud et la source, tandi que $h(n)$ est la distance de Manhattan entre le nœud n et la destination. Le chemin trouvé par cet algorithme est toujours optimal si $h(n)$ ne surestime jamais le coût nécessaire à atteindre la destination. L'avantage de cet algorithme est que son nombre d'itérations est grandement réduit par rapport à une recherche en largeur d'abord. Toutefois, il ne se prête pas à une implémentation parallèle, car il nécessite une vision globale nécessaire à l'évaluation de $h(n)$, et que son essence même est séquentielle.

D'autres algorithmes de routage, dont notamment [34, 97, 151, 173], dédiés aux FPGAs, ont également vu le jour après les années 85, date de création des premiers FPGAs. Le problème global y est légèrement différent que pour les ASICs, de par la structure fixe des circuits reconfigurables, et le nombre limité de ressources de routage disponibles. L'optimisation y a pour but de router l'ensemble des connexions demandées, ainsi que de minimiser les délais de propagation. Nous n'étudierons pas en détail ces algorithme, car leur structure ne se prête pas à une implémentation purement distribuée. Ils nécessitent tous, à la manière de A*, une vision globale, qui aide à l'optimisation du routage.

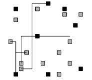
4.4 Approches Matérielles

Un des buts de cette thèse étant d'explorer une implémentation matérielle du routage, nous allons maintenant nous intéresser aux réalisations matérielles de tels systèmes, dont la plupart ont tiré parti du parallélisme intrinsèque de l'algorithme de Lee. À part les deux solutions de Shannon et Rapaport, toutes ont été réalisées dans l'optique du routage des circuits imprimés ou électroniques.

Nous allons présenter plusieurs types de systèmes qui furent développés, pour la grande majorité, dans les années 80 :

- Les coprocesseurs développés pour accélérer certaines parties de l'algorithme de Lee, comme le stockage des listes par exemple.
- Les processeurs spécialisés.
- Les architectures MIMD (Multiple Instruction Multiple Data), où un ensemble de processeurs indépendants, et reliés entre eux, résolvent le problème.
- Les architectures SIMD (Single Instruction Multiple Data), où un ensemble de processeurs simples exécutent le même code au même instant.
- Les architectures purement matérielles, qui nous amèneront tout naturellement vers notre solution.

Avant de poursuivre, citons ici l'approche matérielle de Shannon au problème du labyrinthe, où trouver un point est identique à rechercher un chemin entre une source et



une destination. En 1952 [211], il présente une machine mécanique, entre autre composée de 70 relais électriques, capable de résoudre cette tâche dans un labyrinthe de taille 5x5. Elle mémorise le chemin qu'elle a déjà parcouru pour explorer tout l'espace à la recherche du but, grâce à un senseur détectant les murs et le but.

4.4.1 Routage de FPGA

L'approche de DeHon, Huang, et Wawrzynek ne cadre pas dans les catégories que nous venons d'énumérer, mais présente une intéressante réalisation d'un FPGA auto-routable. Les FPGAs, comme nous l'avons vu au chapitre 2, ont des réseaux d'interconnexions qui deviennent de plus en plus complexes. Il serait dès lors appréciable de disposer de tels circuits capables de configurer eux-mêmes leur routage, sans faire appel à un logiciel spécialisé. Dans cette optique, DeHon, Huang, et Wawrzynek [58, 107], ont donc développé un système matériel possédant de tels capacités. Il s'agit de pouvoir router un réseau de type Hierarchical Synchronous Reconfigurable Array (Figure 4.12), qui a la caractéristique de posséder un unique ensemble de switchbox entre une source et une destination. Le routage global devient donc une simple affaire locale, qui peut être gérée par quelques transistors, présents à chaque switch point. Cette approche est très intéressante et prometteuse, mais ne peut malheureusement être appliquée qu'à un réseau de type HSRA, ce qui en limite grandement l'application à d'autres systèmes.

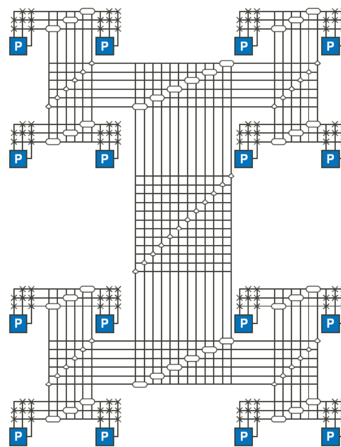


Figure 4.12 : *La topologie d'un tableau HSRA*

4.4.2 Coprocesseur

Une des façons d'accélérer un algorithme de routage est d'augmenter les performances d'un processeur standard en lui ajoutant un coprocesseur capable d'optimiser certaines tâches inhérentes à de tels algorithmes. Damm et Gethöffer [49] offrent un "noyau de routage" capable de gérer les fonctions de coût ainsi que la liste de cellules, qui peut être relié à un processeur pour accélérer le déroulement de l'algorithme. Chan et Schlag [38], quant à eux, ont créé un système de routage distribué sur plusieurs stations de travail, et dont une partie est accélérée par du matériel spécifique, notamment pour le calcul de la fonction de coût.

4.4.3 Processeur

Plus complexe qu'un coprocesseur, plusieurs approches ont proposé l'architecture d'un processeur spécialisé dans le routage. Connecté à un processeur hôte, qui ne s'occupe que de charger la liste des cellules à router, il gère seul l'entièreté du routage d'un circuit. Spiers et Edwards [224] en ont proposé une réalisation basée sur un seul processeur, et Won, Sahni, et El-Ziq [256], ont, eux, réalisé un processeur à trois niveaux de pipeline pour l'accélération de la phase d'expansion. Une autre implémentation pour résoudre le problème du routage sans grille, c'est-à-dire non cellulaire, mais dans le plan réel, a été réalisée par Sato, Kubota et Ohtsuki [206], à l'aide de Content Addressable Memory appliquées à un algorithme de type line-expansion.

4.4.4 SIMD

Alors que les accélérateurs de type coprocesseurs ou processeurs exécutent l'algorithme de routage de manière séquentielle, tout en l'accélérant, les architectures SIMD permettent de profiter du parallélisme matériel. Des éléments de calcul (PE, pour Processing Element) y sont commandés par une unité centrale, et exécutent au même moment les mêmes opérations. Cette approche peut donc grandement améliorer la phase d'expansion de l'algorithme de Lee.

Les deux algorithmes parallèles de Watanabe, par exemple, ont été développés dans le but d'être implémentés sur un réseau de processeurs, l'Adaptive Array Processor (AAP-1) [125, 126, 227]. Il s'agit d'une grille de 256×256 éléments de calcul de un bit, qui semble être la plus grande à avoir été construite. Parmi les autres implémentations que l'on peut trouver, dont [4] [26] [131] [228] [250] , peu travaillent avec un PE par cellule. En effet, la taille des circuits à router dépasse facilement 256×256 (taille de l'AAP-1), et donc le nombre de PEs pourrait devenir ingérable. Dès lors, certains utilisent une grille toroïdale, où un PE peut servir à plusieurs cellules [228], d'autres exécutent un routage grossier, qu'ils affinent ensuite [131], et d'autres encore utilisent un tableau virtuel, dont des parties sont chargées dans les PEs [26].

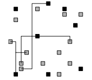
4.4.5 MIMD

Plus générale que l'architecture SIMD, les MIMD sont composés de petits processeurs, qui au lieu d'exécuter tous le même code au même moment, sont indépendants. De nombreuses implémentations ont vu le jour, ainsi que plusieurs manières de gérer un grand tableau de cellules [105] [120] [138] [174] [229][202][195]. Certaines ont une architecture en arbre binaire, où les processeurs sont organisés de façon hiérarchique, d'autres forment une grille. L'avantage de l'approche MIMD est qu'elle peut tirer partie des architectures déjà réalisées pour résoudre d'autres problèmes. Toutefois, la non possession d'un tel réseau d'ordinateurs peut impliquer un investissement nettement plus important que la réalisation de matériel spécialisé.

4.4.6 Analogique

Rapaport

Rapaport et Abramson [189], en 1959, proposent, quant à eux, une machine analogique capable de résoudre le problème du plus court chemin dans un graphe non-orienté dont les poids ne sont pas forcément égaux. Leur système est basé sur des



timers, qui sont placés sur les arêtes du graphe, avec un seuil correspondant au poids de l'arête. Le processus démarre de la source, qui lance les timers des arêtes lui étant connectées, et lorsqu'un timer arrive à son terme, il allume une lumière et lance les timers reliés à son nœud d'arrivée. De cette manière, l'arbre des chemins les plus courts est construit depuis la source, jusqu'à arriver à la destination, qui bloque le mécanisme.

Leur approche est élégante, et permet de résoudre le problème pour un graphe de taille quelconque, en reliant entre eux plusieurs de leurs prototypes.

Cheng

Un routeur totalement analogique est proposé par Cheng, Tanaka, et Yamada [40], en 1991. L'idée est de pouvoir y router un chemin dans une grille rectangulaire de cellules, et de le faire en considérant les liaisons entre voisines comme des résistances. Au centre d'une cellule, une source de courant peut être imposée, avec un potentiel positif ou négatif. Il suffit alors de placer un haut voltage pour la source, et un faible pour la destination, de laisser le réseau se stabiliser, et de parcourir la distribution de potentiel du plus grand au plus petit.

L'implémentation physique est réalisée à l'aide de transistors, qui permettent de bloquer certaines liaisons, et de placer des sources et des destinations. Le blocage de cellules se fait en imposant un haut voltage, obligeant ainsi le potentiel à contourner l'obstacle. Ce système analogique trouve donc un chemin entre une source et une destination, mais ne trouve pas forcément le chemin le plus court, étant donné que plusieurs chemins possibles sont exploitables par une descente de gradient.

4.4.7 Tableau de cellules pour la réalisation de circuits

A notre connaissance, quatre réalisations matérielles de l'algorithme de Lee sous forme de tableau de cellules existent à ce jour, les trois premières datant du début des années 80, et la quatrième du début du millénaire. Elles sont toutes quatre basées sur une implémentation parallèle de l'algorithme de Lee, où les cellules stockent l'origine de l'expansion sur 2 ou 4 bits.

Breuer

En 1981, Breuer et Shamsa [30] proposent l'architecture d'une L-machine, une implémentation de l'algorithme de Lee basée sur des L-Cellules matérielles (Figure 4.13). Chaque cellule est composée de 7 bascules JK et de 75 portes logiques, communique directement avec ses quatre voisines, et est adressable par une unité de contrôle global. Trois des bascules codent l'état de la cellule parmi cinq, et les quatre restantes mémorisent l'origine de la phase d'expansion, en codage 1 parmi 4.

En 1981, la taille des circuits intégrés n'était pas celle d'aujourd'hui. Breuer et Shamsa présentent donc également une méthode visant à relier plusieurs circuits entre eux. Certains y ont des interfaces sur les quatre côtés, d'autres sur trois, et d'autres encore sur deux, une interface y servant à connecter deux circuits entre eux. Dans le premier cas, pour un circuit contenant $n \times n$ cellules, le nombre de pins est $2log_2(n) + 4n + 8$, et pour le cas d'un circuit fonctionnant seul, il est de $4log_2(n) + 8$.

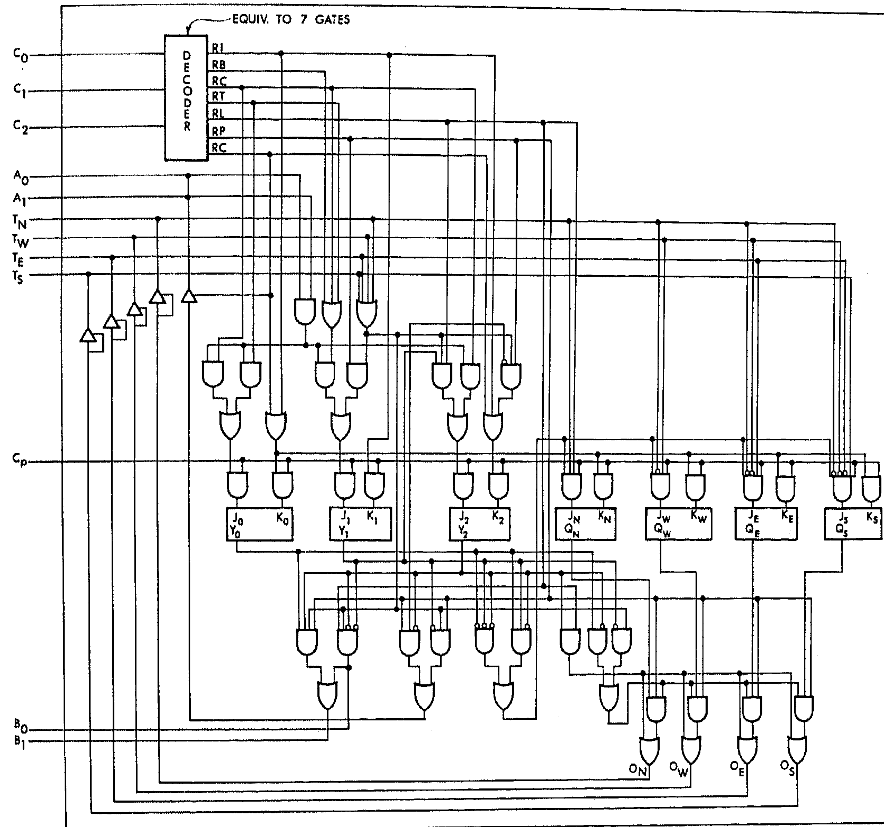
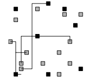


Figure 4.13 : *Implémentation d'une L-cellule de Breuer.*

Iosupovicz

Quasi simultanément, Iosupovicz [108] développe son implémentation, faite également d'une unité de contrôle, d'un tableau de cellules, et d'interfaces pour interconnecter plusieurs circuits. Sa cellule contient 6 bascules et 34 portes logiques[4] (Figure 4.14). Quatre des bascules définissent l'état de la cellule, et les deux dernières définissent l'origine de l'expansion (chez Breuer il s'agit d'un codage en 1 parmi M, qui nécessite plus de bascules, mais moins de logique additionnelle). La différence entre les deux implémentations réside dans l'interfaçage entre circuits. Iosupovicz propose une amélioration au modèle de Breuer, en diminuant le nombre de pins nécessaires. Pour que quatre circuits puissent être connectés à un autre, ce dernier ne nécessite que $4log_2(n-1)+14$ pins au lieu de $2log_2(n)+4n+8$. Ceci implique toutefois une communication plus complexe, où seuls les changements d'états des cellules de bord sont envoyés aux circuits voisins. L'exécution peut donc être ralentie, lorsque plusieurs cellules changent d'état sur le même bord.

[4]Ce nombre de 34 portes logiques est tiré de l'article, mais il semble clair, au vue de la figure 4.14, qu'il a été calculé en considérant qu'une porte à 5 entrée est identique à une à 2 entrées. Il faut donc revoir cette évaluation à la hausse, rendant le nombre de portes logiques relativement équivalent aux autres implémentations.



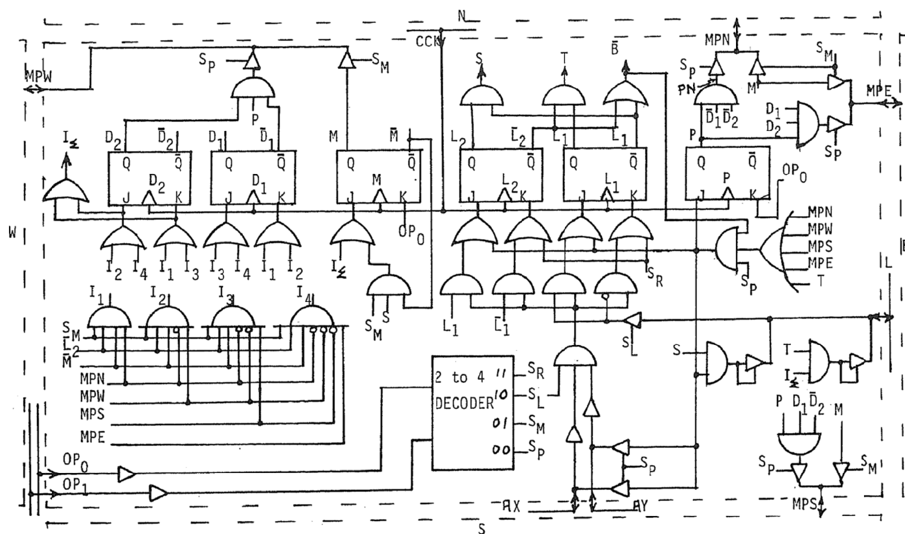


Figure 4.14 : *Implémentation d'une cellule de Iosupovicz.*

Ryan

L'approche prise par Ryan et Rodgers [201], en 1987, vise à router un système sur deux couches, où une cellule a cinq voisines, dont une est sa correspondante sur la couche opposée. Une cellule y est composée de 6 bascules et de 71 portes logiques (Figure 4.15).

La grande différence, en comparaison des autres approches, est leur manière de traiter des tableaux de grande taille. Ici, un système de fenêtre est utilisé, un processeur hôte devant envoyer les fenêtres les unes après les autres au système, tout en récupérant l'état de la fenêtre qui vient d'être calculée. Ceci se fait en décalant les données dans le tableau de cellules, qui est vu comme un grand registre à décalage.

Un routage d'un tableau de $qn \times qn$ points, dans un circuit physique de taille $n \times n$, se fait tout d'abord sur un système où une cellule physique correspond à un tableau de cellules virtuelles de taille $q \times q$. Ces cellules représentent la connectivité grossière du système, et un chemin peut y être trouvé. Ensuite, pour chaque cellule grossière sur le chemin à créer, le tableau de cellules fines correspondant est routé indépendamment, une cellule physique correspondant alors à une cellule réelle.

Nestor

Finalement, la quatrième réalisation est due à Nestor [175, 176], qui, en 2002, implémente un algorithme de Lee pour un routage multicouche, sur FPGA. Une cellule (Figure 4.16) est ici composée d'un séquenceur faisant office de machine d'états (8 états ne nécessitant pas d'autres bits de mémorisation pour l'origine), et d'un registre à décalage qui contient l'état des cellules sur les différentes couches. De cette manière, les couches sont mises à jour à tour de rôle, en multiplexant le temps d'utilisation de la cellule. Concernant l'implémentation sur un FPGA de Xilinx, le XC2S300E, une cellule nécessite 27 look-up tables à 4 entrées, en utilisant trois de ces LUTs comme des registres à décalage.

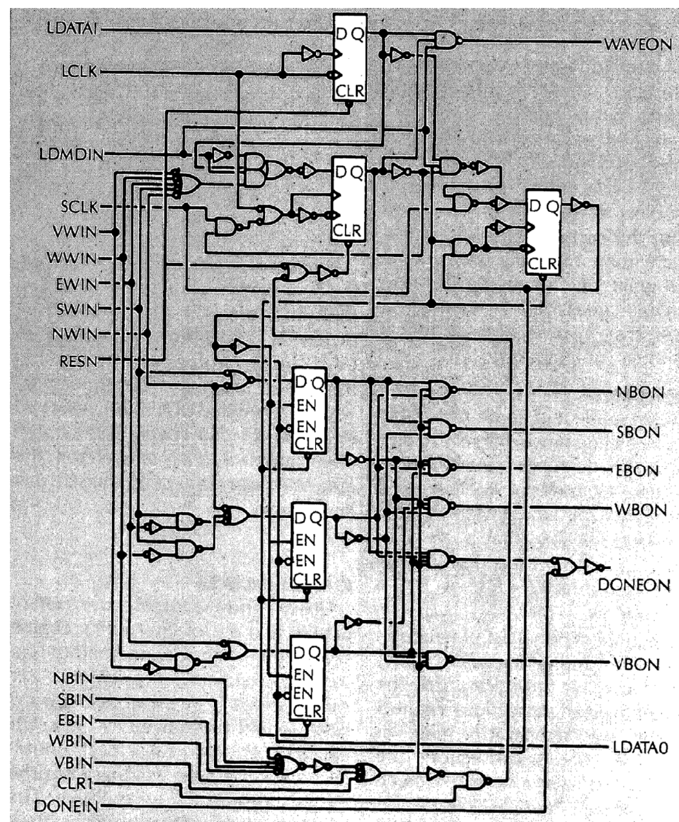


Figure 4.15 : *Implémentation d'une cellule de Ryan.*

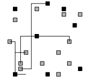
4.4.8 Tableau de cellules pour du matériel bio-inspiré

Toutes les implémentations que nous avons passées en revue sont basées sur le fait qu'une cellule est soit occupée, soit inoccupée, et sont toutes dédiées à l'optimisation du routage de circuits imprimés ou intégrés. Les cellules sont reliées entre elles par un graphe dont les arêtes ne sont pas orientées.

Dans le cadre de systèmes bio-inspirés, l'idée est d'offrir plus d'adaptabilité aux circuits électroniques. Dans cette optique, Moreno [169] propose un mécanisme capable de créer des chemins de données, en configurant dynamiquement des blocs de multiplexeurs, dans une structure semblable à celle exploitée par Ercal et Lee [68]. Le but y est de coupler les cellules de routage avec des éléments qui doivent pouvoir se transmettre des informations, et dont les connexions ne sont pas définies a priori, mais peuvent l'être durant le fonctionnement du système.

Une cellule, que Moreno appelle unité de routage, n'est autre qu'un contrôleur qui devrait, pour une réalisation réelle, être relié à un switchbox composé de cinq multiplexeurs (Figure 4.17) : quatre pour sélectionner la valeur à envoyer à chacune de ses voisines, et un pour la valeur à transmettre à un élément lui étant connecté. L'unité de routage peut être implémentée grâce à 47 LUTs d'un FPGA Virtex-E de Xilinx, et de 5 bascules. Le mécanisme de routage étant basé sur l'algorithme de Lee, deux d'entre elles servent à stocker l'origine de l'expansion, et les trois autres permettent de gérer le processus.

Pour son bon fonctionnement, une unité de routage possède quinze entrées, qui



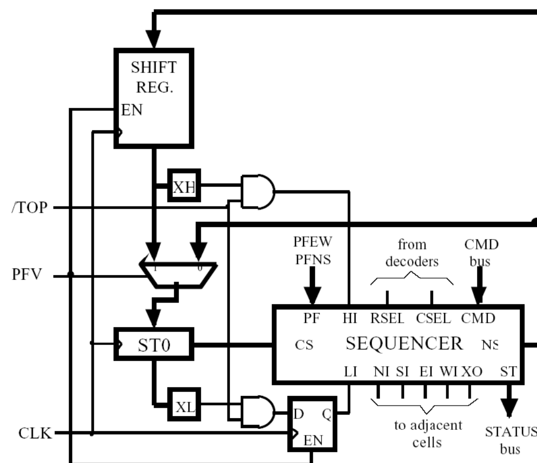
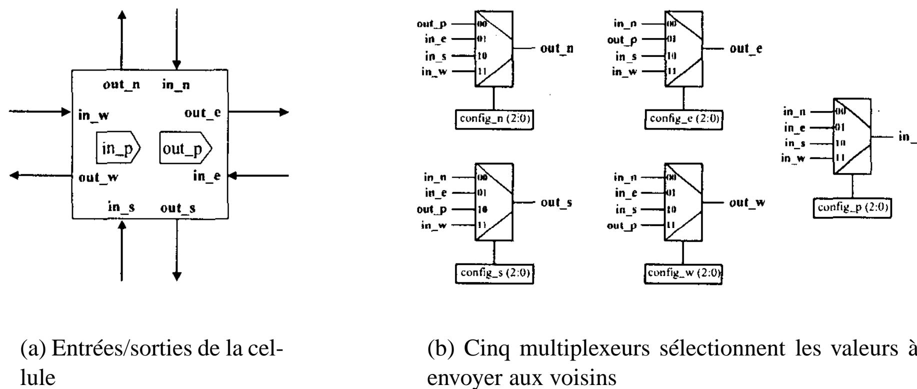


Figure 4.16 : *Implémentation d'une cellule de Nestor.*



(a) Entrées/sorties de la cellule

(b) Cinq multiplexeurs sélectionnent les valeurs à envoyer aux voisins

Figure 4.17 : *Une cellule de Moreno.*

définissent la configuration existante des multiplexeurs. Chacun d'eux possède des bits indiquant le signal à sélectionner (Figure 4.17(b)), et un autre en charge de préciser si le multiplexeur fait déjà partie d'un chemin existant où non. Le processus de routage doit effectivement prendre garde à ne pas effacer un chemin préexistant, mais peut réutiliser un tel chemin partant de la source qui cherche à être connectée.

Chaque unité de routage envoie des signaux à chacune de ses quatre voisines, via trois lignes directionnelles. Ce nombre élevé de connexions pourrait toutefois aisément être réduit, afin d'éviter des problèmes liés au nombre de pins des circuits électroniques.

4.5 Conclusion

Le problème du routage de circuits imprimés et intégrés a, nous venons de le voir, suscité le développement d'un nombre non négligeable d'algorithmes et d'implémentations matérielles. Les années 80 ont vu le nombre d'accélérateurs matériel croître de manière fulgurante, les circuits à router devenant de plus en plus gros, et la vitesse des

processeurs de l'époque n'étant pas à la hauteur de telles applications.

Toutes ces implémentations, qu'elles soient prévues pour des cartes à simple, double, ou multi-couche, visent à relier des points par des chemins, sans toutefois que ceux-ci ne soient directionnels. L'espace y est divisé en cellules, qui peuvent être occupées par un chemin existant, ou inoccupées, et donc candidates potentielles au passage d'un nouveau chemin. Notre intérêt se situe, quant à lui, dans la génération de chemins de données dirigés, basés sur des multiplexeurs responsables de router les signaux d'une source à une destination, au travers d'une grille de cellules.

Seul le système présenté par Moreno, spécialement conçu pour des applications bio-inspirées, traite de chemins directionnels. Son algorithme est semblable à celui de Lee, qui fut implémenté pour la réalisation de circuits imprimés, avec pour nuance la directionnalité des liens entre cellules. Nous pouvons cependant émettre trois remarques concernant cette implémentation.

Premièrement, la cellule de routage a été simplifiée au maximum, puisqu'elle ne contient qu'un petit contrôleur auquel il faut fournir la configuration courante des multiplexeurs. Il serait plus judicieux de considérer les multiplexeurs comme faisant partie intégrante de la cellule de routage.

Deuxièmement, le nombre de liaisons inter-unités de routage est trop important. Dans le cas de la réalisation d'un système multi-circuits, pour des circuits contenant un tableau de $n \times n$ unités de routage, pas moins de $24n$ pins seraient nécessaires au bon fonctionnement de l'algorithme, ce qui est loin d'être négligeable.

Troisièmement, comme dans toutes les implémentations que nous avons passées en revue, un contrôle global doit être présent afin de définir quelles sont les sources et destinations à connecter. Dans le cas de systèmes distribués, il serait plus judicieux qu'ils ne nécessitent pas un tel contrôle.

Dans le chapitre suivant, nous allons donc proposer un système où les cellules sont maîtres de leur destin, et ont la possibilité d'initier elles-mêmes des connexions, et ce de manière totalement distribuée.