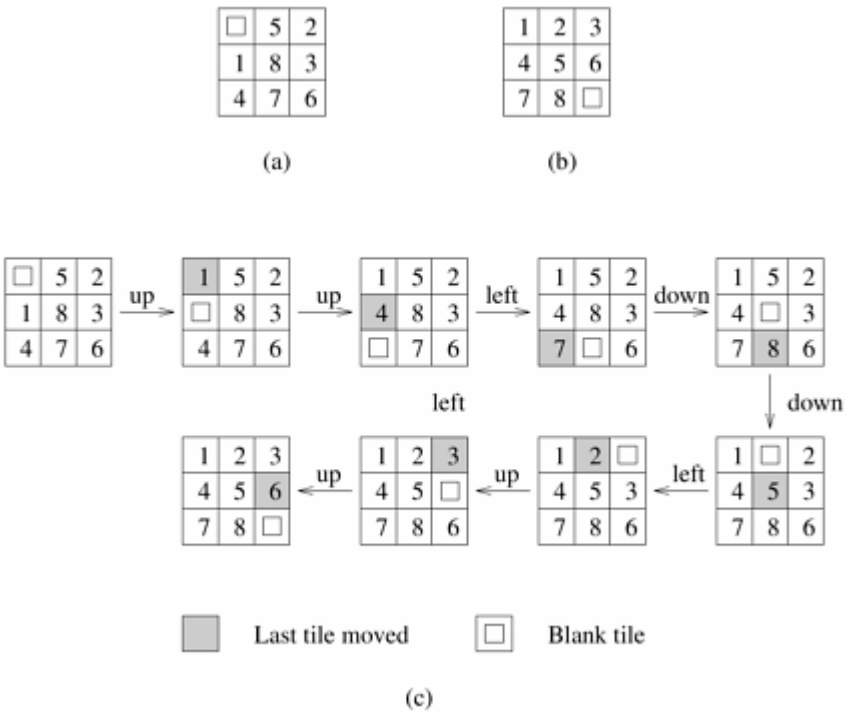
Projet de programmation parallèle en UPC

Le (N^2-1)-puzzle

I – Le jeu

Le jeu du (N^2-1)-puzzle consiste en une grille contenant N×N cases, numérotées de 1 à N-1 et où une des cases est vide. Une case peut être déplacée dans la case vide si sa position initiale est adjacente à cette case vide. Le déplacement créé donc une case vide à la position initiale de la case déplacée. Les positions initiales et finales des cases sont spécifiées au départ, et l'objectif est de déterminer la plus petite suite de déplacements permettant de passer des positions initiales aux position finales. La figure ci-dessous montre une séquence de jeu pour un 8-puzzle.



L'ensemble de toutes les séquences de déplacements qui mènent à la configuration finale est trop grand pour être construit entièrement et énuméré. On pose alors un problème d'optimisation discrète de recherche du chemin de coût minimal dans un graphe, d'un nœud initial à un nœud final donnés.

Il est possible d'estimer le coût nécessaire pour atteindre le nœud final à partir d'un nœud intermédiaire d'un état x en utilisant une heuristique.

Soient h la fonction d'heuristique, h(x) l'estimation en x, g la fonction de coût du nœud initial à un nœud quelconque, et g(x) sa valeur en x. Si h(x) est une borne inférieure du coût nécessaire pour atteindre l'état final à partir de l'état x pour tout x, alors h est dite admissible.

On définit l(x) = h(x) + g(x). Si h est admissible, alors l(x) est une borne inférieure du coût d'un chemin complet en étendant le chemin courant de x jusqu'à l'état final.

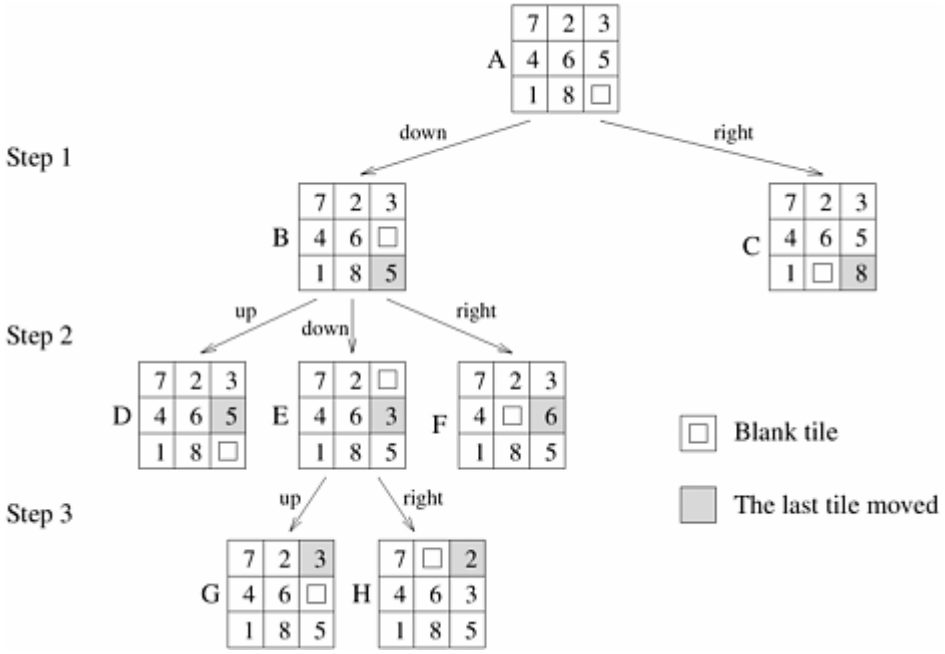
Dans le cas du puzzle, on représente chaque case par le couple de ses coordonnées. La distance de Manhattan entre les positions (i,j) et (k,l) est définie par |i-k| + |j-l|. La somme de toutes les distances de Manhattan entre la position initiale et finale pour toutes les cases est une estimation du nombre de déplacements requis. Cette estimation est l'heuristique de Manhattan qui est admissible.

II. Recherche en profondeur par l'algorithme IDA* (Iterative Deepening A*)

Cet algorithme effectue une recherche dans un arbre, où l'état initial s est la racine de l'arbre. On fixe une borne de coût maximum c. L'algorithme développe une branche de l'arbre jusqu'à que la valeur l(x) de l'état courant est supérieur au coût maximum c, ou jusqu'à que l'on ait atteint l'état final. Si l(x) > c, alors l'algorithme effectue un retour arrière afin d'explorer une autre branche. Si aucune solution n'est trouvée avec le coût maximum c, on augmente c puis on recommence la recherche à partir de la racine (nouvelle itération).

Au départ, on fixe le coût maximum à l(s) = h(s), où s est l'état initial. Pour une nouvelle itération, on fixe c à la valeur minimale des l(x) des nœuds qui n'ont pas pu être développés à l'itération précédente.

On montre ci-dessous quelques pas de l'algorithme pour le 8-puzzle.



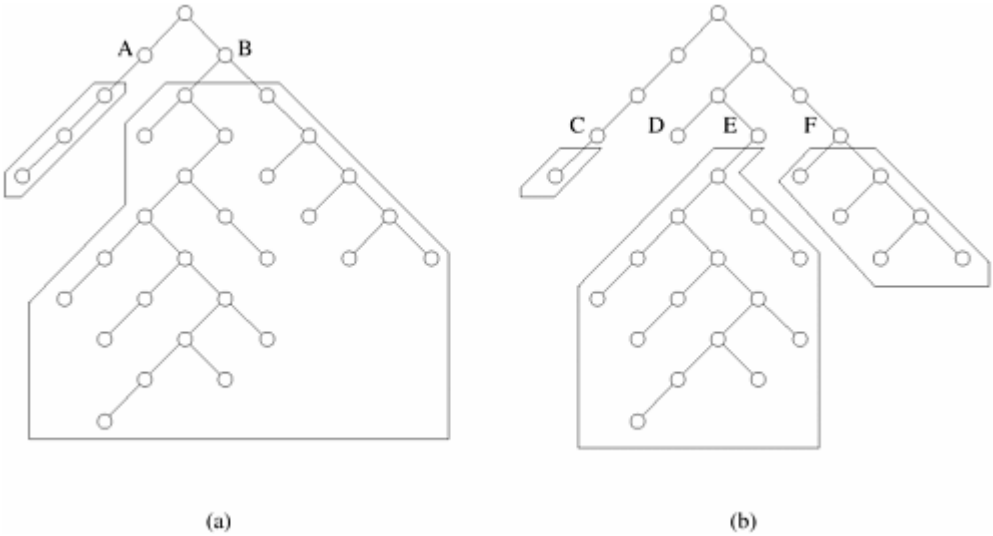
On remarque que les états D et G peuvent être supprimés car ils ne sont que des duplications de l'état B.

A chaque pas de l'algorithme, les nœuds non développés doivent être mémorisés. L'espace des états peut être représenté efficacement avec une pile. Les états non développés sont stockés avec leur état père. Par exemple, pour l'exemple précédent, la pile serait :

bas	A	C
	B	F
	E	
haut	H	

III. Recherche en profondeur IDA* parallèle

La question cruciale si l'on parallélise la recherche en profondeur est la distribution de l'espace de recherche aux processeurs. On peut statiquement allouer à chaque processeur l'exploration d'un sous-arbre. Mais chaque sous-arbre ne va pas induire le même nombre de développement d'états, entraînant ainsi un mauvais équilibrage de charge.



Il est difficile, voire impossible, d'estimer la taille de l'espace de recherche à l'avance. Il faut donc implémenter une stratégie de distribution dynamique du travail aux processeurs. Lorsqu'un processeur n'a plus de travail, il en demande à un autre processeur. Lorsqu'un processeur atteint l'état final, tous les processeurs stoppent leur exécution.

Stratégie de partage de travail :

Chaque processeur maintient sa propre pile locale. Lorsque celle-ci est vide, il demande des états non-développés d'une autre pile d'un autre processeur. Au départ, tout l'espace de recherche est alloué à un seul processeur. Lorsqu'il est inactif, un processeur sélectionne un donneur et examine sa pile locale. Si elle n'est pas vide, celle-ci est coupée en 2 piles, une pour le donneur et l'autre pour le demandeur. Il est préférable de couper vers le bas de la pile, afin d'allouer une quantité suffisamment importante de travail. Afin d'éviter d'allouer trop peu de travail, on fixe une limite de niveau de coupe au-dessus duquel on refuse de donner du travail au demandeur.

Stratégie de choix du donneur :

Stratégie cyclique asynchrone : chaque processeur gère une variable locale *target*. Lorsqu'un processeur est inactif, il utilise cette variable comme label d'un processeur donneur et tente d'obtenir du travail de sa part. La valeur de *target* est ensuite incrémentée modulo p, où p est le nombre total de processeurs. La valeur initiale de *target* pour chaque processeur est (son label + 1) modulo p. Mais il est possible avec cette stratégie que plusieurs processeurs fassent une demande au même donneur presque en même temps.

Stratégie cyclique globale : on utilise ici une seule variable partagée *target*. Ainsi les demandes seront destinées équitablement aux processeurs.

Scrutation au hasard : Le choix du donneur est aléatoire.

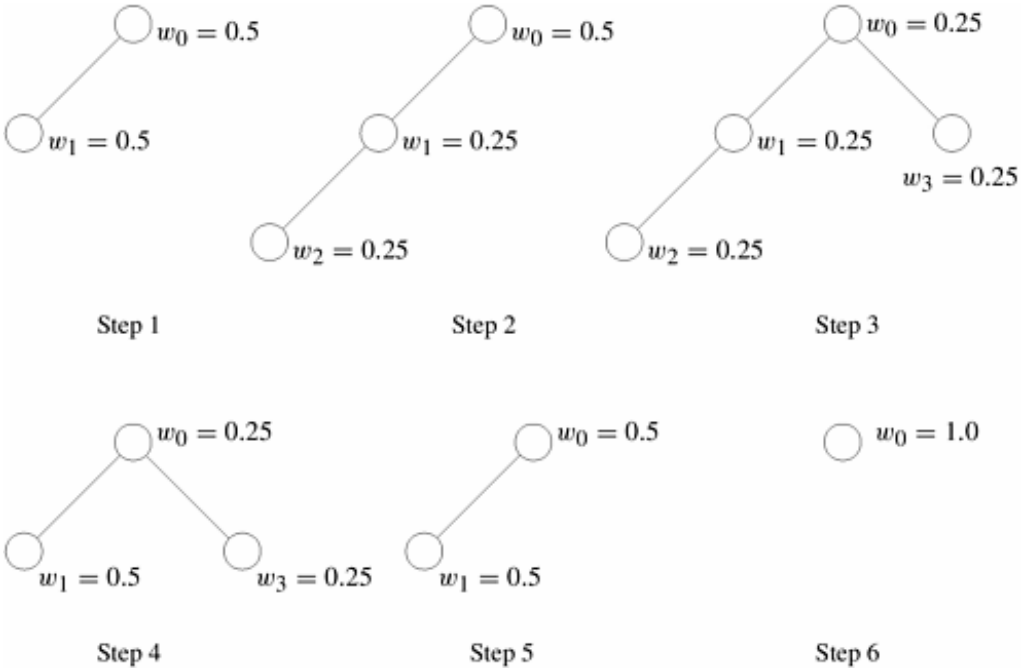
Détection de terminaison :

Algorithme de Dijkstra modifié : Les processeurs sont organisés en anneau. Chaque processeur peut être dans l'état blanc ou noir. Initialement, tous les processeurs sont blancs.

1. Lorsque le processeur P_0 est inactif, il envoie un jeton blanc au processeur P_1.
2. Si le processeur P_j donne du travail au processeur P_i avec j > i, alors P_j devient noir.
3. Si le processeur P_i est inactif et reçoit le jeton, alors il le transmet à P_{i+1}. Si P_i est noir, alors la couleur du jeton change à noir avant d'être transmis, sinon sa couleur n'est pas changée.
4. Après avoir transmis le jeton, P_i redevient blanc.

L'algorithme termine lorsque P_0 reçoit un jeton blanc et lorsqu'il est lui-même inactif.

Détection par arbre : Initialement, P_0 possède tout l'espace de recherche et on lui associe un poids de 1. Tous les autres processeurs ont un poids de 0. Lorsque son espace est coupé pour servir un autre processeur, son poids est divisé par 2, soit 0,5, et le demandeur voit son poids fixé à 0,5. Chaque fois qu'un processeur sert un autre processeur, son poids est divisé par 2. Lorsqu'un processeur termine son traitement, il renvoie son poids au processeur qui lui avait donné son travail. La terminaison est détectée lorsque le processeur P_0 retrouve son poids de 1.



IV – Travail à effectuer

- Implémenter l'algorithme IDA* parallèle en UPC selon toutes les variantes décrites ci-dessus (choix du donneur, terminaison).
- Essayer d'établir un rapport entre niveau de coupe maximum dans la pile des états à développer, taille du puzzle et nombre de processeurs.
- Expliquez vos choix de structures de données et de distributions de données, en les justifiant éventuellement par des mesures de performances.
- Pour votre meilleure implémentation, effectuer une étude d'accélération, d'efficacité et d'iso-efficacité pour plusieurs tailles de puzzle.
- Rédiger un rapport complet
- Remettre le rapport ainsi que les codes sources des programmes.

Documentation UPC : http://polytope/ipp/upc_manual.pdf