Université de Liège
Faculté des Sciences Appliquées
Institut Montefiore

L'intelligence artificielle et les jeux : cas du Sokoban

Année académique
2006 - 2007

Travail de fin d'études présenté par
Jean-Noël Demaret
en vue de l'obtention du grade de
Licencié en Informatique.

Promoteur : Professeur Pascal Gribomont, responsable du Service d'Intelligence Artificielle de l'Université de Liège.

Remerciements

Je tiens avant tout à remercier Monsieur Gribomont pour l'enthousiasme communicatif dont il fait preuve dans ses cours et qui m'a amené à me passionner pour les deux domaines sous-jacents à ce travail que sont l'intelligence artificielle et la programmation fonctionnelle. Je le remercie également pour la confiance qu'il m'a accordée en acceptant d'être mon promoteur ainsi que pour son accueil et le temps qu'il n'a jamais hésité à me consacrer chaque fois que je suis venu frapper à la porte de son bureau.

Je remercie particulièrement François Van Lishout, assistant de Monsieur Gribomont, qui m'a proposé de poursuivre les recherches qu'il avait entreprises sur le Sokoban. Les échanges que nous avons eus ont représenté pour moi un des aspects les plus intéressants de ce travail. C'est grâce à eux que l'intérêt qu'a suscité chez moi ce sujet n'a jamais cessé d'être réanimé tout au long de cette année.

Je remercie les membres du jury pour le temps, que j'espère agréable, qu'ils ont passé à lire mon travail.

Je remercie mes parents pour m'avoir offert, durant toutes ces années, un cadre dans lequel j'ai pu réaliser mes études dans des conditions idéales. Je les remercie également pour les relectures, parfois fastidieuses, qu'ils ont faites de mon travail.

Je remercie Anouchka pour la patience dont elle a fait preuve pour supporter mes indisponibilités et mes moments de doute, pour ses encouragements, ainsi que pour l'aide ponctuelle qu'elle n'a jamais hésité à m'apporter.

Jean-Noël Demaret
Août 2007

Résumé

Depuis toujours, les chercheurs en intelligence artificielle ont considéré les jeux comme des sujets d'exploration privilégiés afin de développer des systèmes informatiques capables de simuler les capacités de raisonnement de l'être humain.

L'objectif principal de ce travail est la réalisation d'un programme capable de résoudre des problèmes de Sokoban. Le Sokoban est un jeu d'origine japonaise dans lequel les performances de l'être humain restent significativement supérieures à celles des meilleurs solveurs existants actuellement. En effet, le Sokoban possède des caractéristiques qui font de lui un problème particulièrement difficile et intéressant. Les plus importantes sont la taille gigantesque de l'espace d'états d'un problème (10^{98} pour un problème de taille 20×20) et la présence de situations de deadlock, *i.e.* la possibilité qu'une suite de coups conduise à une configuration dans laquelle le problème devient insoluble.

Le cheminement suivi dans ce travail comporte deux grandes étapes. Dans un premier temps, nous montrons comment le Sokoban peut être abordé comme un problème de recherche dans un espace d'états. Nous présentons les principaux algorithmes de recherche et leur application au problème spécifique du Sokoban. Enfin, nous montrons que ces méthodes classiques sont insuffisantes pour résoudre des problèmes de Sokoban non triviaux.

Dans un second temps, nous développons une méthode originale de résolution basée sur la planification par la décomposition en une suite de sous-problèmes. Nous décrivons une telle décomposition et démontrons son intérêt pour une classe particulière de problèmes de Sokoban. Nous ajoutons ensuite à notre programme la capacité d'apprendre certaines informations à partir des erreurs qu'il aura commises. Nous montrons que l'application de ce concept élémentaire d'apprentissage permet à notre programme de résoudre 54 problèmes d'un benchmark classique de 90 problèmes difficiles. L'implémentation en Scheme du programme que nous avons ainsi développé est disponible à l'adresse : `http://www.student.montefiore.ulg.ac.be/~demaret/tfe/`.

Finalement, nous présentons différentes pistes que nous avons envisagées afin d'améliorer les performances de notre programme. En particulier, nous montrons que l'une de ces pistes permettrait à notre programme de résoudre 61 problèmes du benchmark et de rivaliser dès lors avec le meilleur solveur actuel ayant fait l'objet d'une publication (59 problèmes résolus).

Table des matières

Chapitre 1

Introduction

1.1 L'intelligence artificielle

Il est aisé d'admettre que la personne capable de battre un champion du monde d'échecs fait preuve d'une grande intelligence. Pourquoi n'en est-il pas de même pour l'ordinateur capable d'accomplir les mêmes prouesses ? Sans doute parce que l'ordinateur s'appuie principalement sur ses énormes capacités de calcul pour envisager un grand nombre de coups possibles et qu'il ne joue pas réellement d'une façon que l'on pourrait qualifier d'intelligente. Par ailleurs, l'ordinateur se contente avant tout d'exécuter des suites d'instructions sans avoir conscience du fait qu'il est en train de jouer aux échecs. Effectivement, la perspective de demander à notre adversaire informatique s'il a apprécié la partie nous paraît pour le moins absurde.

Ces considérations nous amènent à distinguer deux approches de l'intelligence artificielle. La première, qui sera l'approche adoptée dans ce travail, se donne pour objectif de faire accomplir par une machine des comportements réputés intelligents. La notion de comportements intelligents demanderait bien sûr à être définie mais cet exercice, que l'on devine complexe, sortirait du cadre d'un travail dont l'objet est avant tout la résolution d'un problème précis d'intelligence artificielle. Nous nous reposerons dès lors sur la définition proposée dans [5] :

Un comportement intelligent est une action contrôlée complexe.

Cette définition établit un lien entre intelligence et complexité qui nous semble fondamental et qui correspond à l'intuition que nous pouvons en avoir. En effet, dans une science comme l'éthologie, la faculté d'accomplir un certain nombre de tâches complexes est utilisée comme métrique afin d'évaluer l'intelligence d'une espèce animale. De même, des théories issues de la psychologie identifient les différents stades du développement intellectuel d'un enfant aux tâches de plus en plus complexes qu'il est en mesure de réaliser. Cette première approche prend donc le parti d'aborder l'intelligence sous un angle essentiellement pragmatique en s'attachant à ses manifestations plutôt qu'à sa nature. Nous l'opposerons à l'approche de l'intelligence artificielle qui a pour but de développer des systèmes capables de penser et qui s'appuie sur l'hypothèse forte que la pensée se situe dans le domaine du calculable. Cette

volonté de développer des entités intelligentes artificielles situe cette deuxième approche à l'intersection de l'informatique et des sciences cognitives. De plus, le fait qu'elle remette en cause des concepts comme la pensée ou la conscience et donc l'essence même de l'Homme lui confère de fortes implications philosophiques. Développer plus avant cette approche de l'intelligence artificielle irait bien au delà-là du cadre de ce travail ainsi que de celui de nos connaissances. Nous tenions néanmoins à l'introduire, d'une part par comparaison avec la première approche, plus traditionnelle, et d'autre part pour l'intérêt personnel que nous lui portons.[1].

Comme nous l'avons annoncé, l'approche adoptée par ce travail part de la définition d'un système intelligent comme d'un système capable d'effectuer une tâche complexe. Cependant, la plupart des tâches complexes que nous connaissons sont effectuées dans un monde qui est tout aussi complexe. On peut dès lors craindre que la conjugaison de ces deux niveaux de complexité rende extrêmement périlleuse la réalisation d'un tel système. Il existe cependant un monde qui est régi par des règles simples et dans lequel l'intelligence règne en maître : le monde des jeux.

1.2 Les jeux

> ***A reductionnist approach to research is often used in science: use the simplest domain that is sufficent to explore the problems.*** **[7]**

Les chercheurs en intelligence artificielle se sont depuis toujours intéressés aux jeux. On peut relever plusieurs raisons à cet intérêt : (1) les jeux ont des règles simples qui sont faciles à modéliser sur une machine ; (2) les jeux mettent en avant les capacités de raisonnement pur de l'être humain ; (3) un jeu est un univers minimal dans lequel un ou plusieurs agents agissent de concert ou les uns contre les autres dans le but de réaliser des objectifs précis ; un jeu peut dès lors être vu comme une abstraction de beaucoup de situations de la vie. Pour toutes ces raisons, les jeux offrent un cadre idéal pour le développement et l'expérimentation de systèmes capables de simuler les capacités de raisonnement de l'être humain.

Certains jeux comme le *Puissance 4* ont pu être totalement résolus par des techniques issues de l'intelligence artificielle. Un jeu est considéré comme résolu s'il est possible de déterminer a priori l'issue de la partie si les joueurs utilisent chacun une stratégie optimale. Dans le cas du *Puissance 4*, le joueur qui commence la partie est sûr de gagner. Pour d'autres jeux comme les échecs ou les dames, l'ordinateur a été capable de battre des champions du monde. Néanmoins, il existe des jeux où les performances du joueur humain restent supérieures à celles des machines dont les capacités de calcul augmentent pourtant continuellement. C'est notamment le cas du jeu de *Go* et du *Sokoban*.

[1] Nous invitons le lecteur intéressé à consulter, à titre d'illustration, le livre d'Alain Cardon : *Modéliser et concevoir une machine pensante, Approche de la conscience artificielle*, aux éditions Vuibert.

1.3 Le Sokoban

Le Sokoban est un jeu inventé en 1980 par le japonais Hiroyuki Imabayashi. Dans ce jeu, le joueur[2] se déplace à l'intérieur d'un labyrinthe et doit amener des pierres dans des emplacements particuliers, que nous appellerons goals (cf. figures 1.1 et 1.2). Le joueur peut seulement pousser les pierres et il ne peut en pousser qu'une seule à la fois. Résoudre un problème de Sokoban consiste à trouver une séquence de déplacements du joueur qui conduit à une situation où chaque goal est occupé par une pierre (cf. figure 1.3).

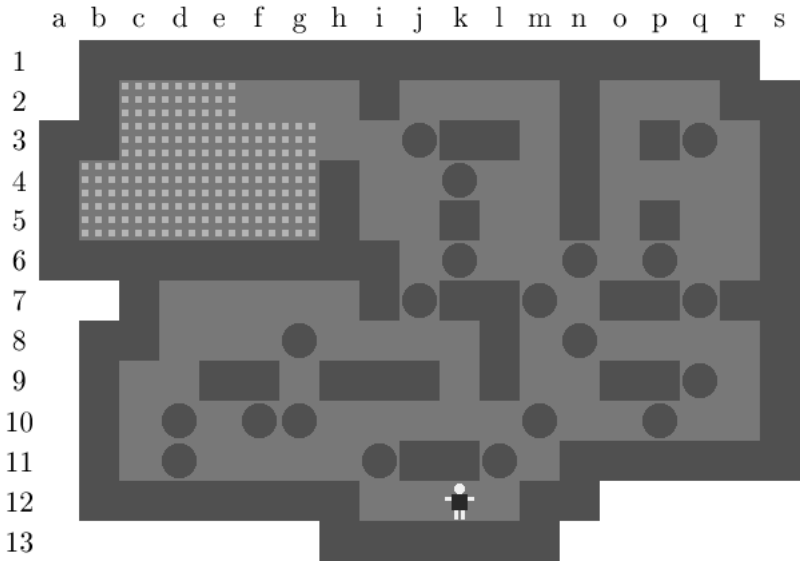

Figure 1.1 – Un problème de Sokoban.

La plupart des problèmes de Sokoban admettent plusieurs solutions. Une solution est dite *optimale* si elle minimise soit le nombre de poussées, soit le nombre de déplacements effectués par le joueur. Ces deux critères sont souvent antagonistes et l'optimisation de l'un se fait la plupart du temps au détriment de l'autre. Dans ce travail, nous ne considérerons l'optimalité d'une solution qu'en terme de poussées. Une action de jeu élémentaire, un coup, ne sera dès lors pas un déplacement du joueur mais la poussée d'une pierre.

Figure 1.2 – Eléments constitutifs d'un problème de Sokoban.

Un problème de Sokoban est composé de cinq types d'éléments : le joueur (a), les pierres (b), les goals (c), les murs (d) et les positions inoccupées (e).

Le Sokoban possède plusieurs caractéristiques qui font de lui un problème intéressant du point de vue de l'intelligence artificielle :

1. Le Sokoban est un jeu dans lequel l'homme reste supérieur à la machine et il constitue dès lors un défi pour les chercheurs en intelligence artificielle.

[2]Dans ce travail, nous identifierons le joueur (humain ou informatique) à sa représentation dans le jeu.

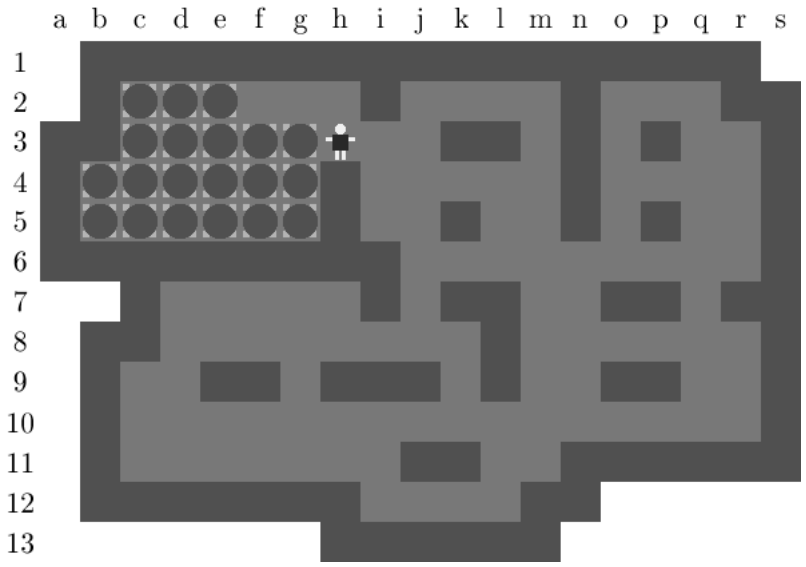

Figure 1.3 – Un problème de Sokoban résolu.

2. Le Sokoban est capable, en dépit de ses règles simples, de proposer des problèmes variés et d'une extrême complexité. La création de problèmes de Sokoban est d'ailleurs un art à part entière. A notre connaissance, au moins une tentative d'automatisation a déjà été entreprise par une équipe japonaise. D'après [6], leur programme a été capable de générer des problèmes qui ont été jugés de bon niveau par des experts humains mais dont la longueur des solutions et donc la complexité étaient limitées. Il est au demeurant fort probable que la conception d'un programme capable de créer des problèmes complexes de Sokoban soit aussi difficile que la réalisation d'un programme capable de les résoudre.
3. Trois caractéristiques du Sokoban font de lui un problème difficile à résoudre par les algorithmes classiques de recherche : (1) le facteur de branchement d'un problème de Sokoban, *i.e.* le nombre de coups possibles à partir d'une situation de jeu donnée, est très élevé (plus de 100 pour certains problèmes) ; (2) la solution d'un problème peut être très longue ; par exemple, la solution optimale du problème 39 de notre benchmark comporte plus de 600 poussées ; (3) la taille de l'espace d'états d'un problème de Sokoban est considérable ; elle a été estimée à 10^{98} pour un problème de taille 20×20. Par comparaison, dans le cas des échecs, le facteur de branchement moyen est de 35, la longueur d'une partie est d'environ 50 coups par joueur et la taille de l'espace d'états est évaluée à 10^{40}.
4. Une heuristique est une fonction qui utilise les connaissances a priori d'un domaine afin d'orienter la recherche vers les parties les plus prometteuses de l'espace d'états. Or, une telle fonction est particulièrement complexe à mettre au point et coûteuse en temps de calcul dans le cas du Sokoban.
5. Contrairement à d'autres jeux, un coup mal joué dans une partie de Sokoban peut conduire à une situation de deadlock, *i.e.* une situation de jeu dans laquelle il n'est plus possible d'amener certaines pierres vers un goal et où le problème devient dès lors insoluble. Ces situations de deadlock sont dues au fait que les règles du Sokoban permettent au joueur de pousser une pierre mais pas de la tirer. Une fois qu'une

poussée est effectuée, il est donc dans la plupart des cas impossible de revenir dans la situation de jeu précédente (cf. figure 1.4).

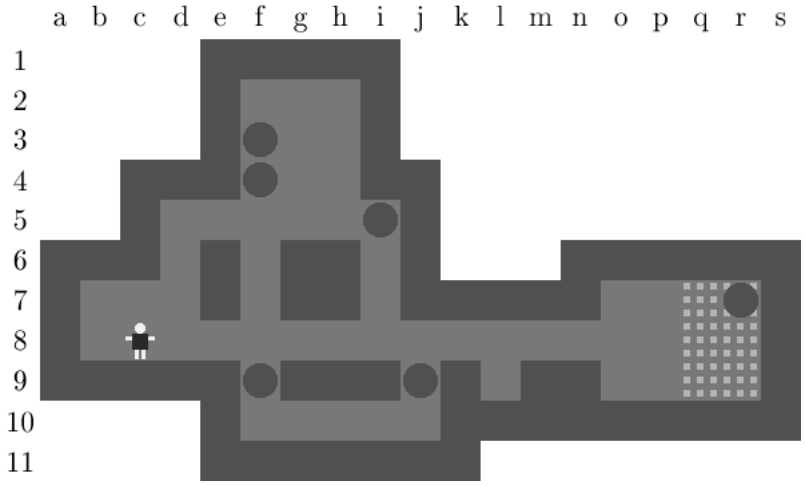

Figure 1.4 – Exemple de situation de deadlock.

Il n'existe aucune possibilité de pousser les pierres *f3*, *f4* et *i5* qui sont donc définitivement immobilisées. La pierre *f9* (resp. *j9*) peut être poussée dans la position *f10* (resp. *j10*) mais elle s'y retrouverait également immobilisée. Cette situation de jeu est dès lors une situation de dealock puiqu'il est impossible d'amener chaque pierre vers un goal et donc de résoudre le problème.

Projet de ce travail

Ce travail s'inscrit dans le cadre des recherches sur le Sokoban menées au Service d'Intelligence Artificielle de l'Université de Liège. Dans les chapitres qui suivent, nous montrerons d'abord que les algorithmes classiques de recherche ne permettent de résoudre que des problèmes simples de Sokoban. Nous tenterons ensuite de démontrer le potentiel d'une approche basée sur la planification par la décomposition en sous-problèmes. Nous verrons comment cette méthode nous permet de réduire sensiblement la profondeur de l'arbre de recherche. Celui-ci est un arbre dont la racine représente une situation de jeu initiale d'un problème et dont chaque noeud a comme successeurs les situations de jeu qui sont accessibles en un coup. Nous verrons également comment utiliser un concept simple de l'apprentissage afin d'exclure de l'espace de recherche un nombre important de situations de jeu et d'ainsi réduire le facteur de branchement de l'arbre. Finalement, nous ouvrirons quelques pistes qui nous semblent intéressantes pour améliorer les performances de notre programme.

Afin d'évaluer les performances des différentes versions de notre programme, nous utiliserons un benchmark de 90 problèmes issus de l'implémentation XSokoban[3] du jeu. Ce benchmark est utilisé par la plupart des concepteurs de solveurs de problèmes de Sokoban ; il nous sera donc utile pour comparer les performances de notre programme avec celles des programmes déjà existants.

[3] `http://www.cs.cornell.edu/andru/xsokoban.html`

Chapitre 2

Etat de l'art

De par la simplicité de son concept et l'intérêt des problèmes qu'il propose, le Sokoban a acquis rapidement une grande popularité. C'est donc assez naturellement qu'un certain nombre de chercheurs en intelligence artificielle ont commencé à l'étudier et à développer des programmes capables de résoudre des problèmes qu'il propose.

Les deux meilleurs programmes existant actuellement sont les solveurs japonais *deepgreen*[1] et *Sokoban Automatic Solver*[2]. Ils sont capables de résoudre respectivement 69 et 78 problèmes sur les 90 du benchmark. Malheureusement, aucun de ces projets n'a été l'objet d'une publication et leur code source n'a pas été diffusé. Une nouvelle version de Sokoban Automatic Solver, diffusée en mai 2007, soit plus de 4 ans après la précédente, est désormais capable de résoudre 86 problèmes du benchmark.

A contrario, le programme *Rolling Stone* [3], développé à l'Université d'Alberta au Canada, a été l'objet de nombreuses publications tout au long des étapes de son développement. *Rolling Stone* utilise l'algorithme de recherche classique IDA* [13] auquel ont été ajoutées de nombreuses améliorations spécifiques qui lui ont permis d'être en mesure de résoudre 59 problèmes du benchmark. Les auteurs ont en effet démontré que l'algorithme IDA* seul ne permettait de résoudre aucun des 90 problèmes.

Actuellement, *Rolling Stone* est le meilleur solveur ayant fait l'objet d'une publication et il est considéré comme la référence par la communauté des chercheurs s'intéressant au Sokoban. Le score de 60 problèmes résolus est dès lors le cap à franchir pour un solveur voulant démontrer son efficacité.

Talking Stones [2] est un programme conçu à l'Université de Liège par François Van Lishout. *Talking Stones* utilise une approche originale qui s'appuie sur une représentation multi-agent du problème. En intelligence artificielle, un agent est une entité autonome qui poursuit un but précis. Classiquement, les agents sont les joueurs d'un jeu. L'approche de *Talking Stones* consiste à voir les éléments primitifs du jeu (les pierres dans le cas du

[1] `http://www2.tokai.or.jp/deepgreen/sokoban/index.htm`
[2] `http://www.ic-net.or.jp/home/takaken/e/soko/index.html`

Sokoban) comme étant des agents qui collaborent entre eux pour atteindre un but commun. Le programme intègre un algorithme capable de résoudre la totalité de la sous-classe des problèmes de Sokoban satisfaisant aux trois conditions ainsi définies par l'auteur :

1. Il doit être possible de déterminer à l'avance et indépendamment de la position des pierres et du joueur, l'ordre dans lequel les goals pourront être remplis sans provoquer de situation de deadlock.
2. Il doit être possible d'amener au moins une des pierres dans le premier goal devant être rempli, sans avoir à modifier la position d'une autre pierre.
3. Pour chaque pierre satisfaisant à la condition précédente, la situation de jeu obtenue en retirant la pierre et en remplaçant le premier goal par un mur doit aussi appartenir à la sous-classe (cf. figure 2.1).

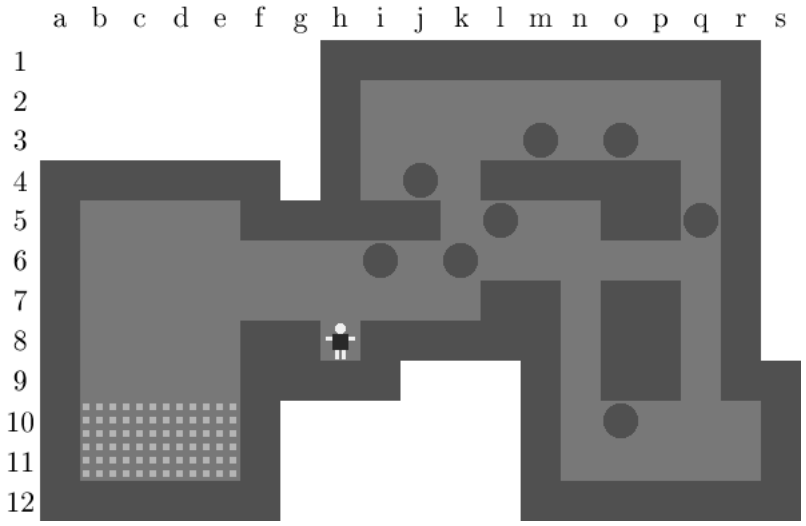

Figure 2.1 – Un problème appartenant à la sous-classe résolue par *Talking Stones*.

Les goals de ce problème peuvent être remplis dans l'ordre *b11*, *c11*, *d11*, *e11*, *b10*, *c10*, *d10*, *e10* sans provoquer de situation de deadlock. La pierre *i6* peut être amenée dans le goal *b11* sans que la position d'une autre pierre ne doive être modifiée. De la même manière, la pierre *k6* peut ensuite être amenée dans le goal *c11*, puis la pierre *o10* dans le goal *d11*, etc.

Talking Stones utilise ensuite un algorithme de recherche classique, l'iterative-deepening, afin de trouver une situation de jeu appartenant à la sous-classe, le problème pouvant alors être résolu immédiatement. La première version du programme, bien que n'intégrant aucune amélioration spécifique du domaine, est déjà capable de résoudre 9 problèmes du benchmark.

Les quelques programmes présentés ci-dessus ne constituent évidemment pas une liste exhaustive et d'autres projets gravitant autour du Sokoban ont été menés dans le monde. Certains de ces travaux ont parfois utilisé le Sokoban comme exemple afin de démontrer les limites d'une méthode qui se révélait efficace dans d'autres domaines. Un certain nombre de ces autres programmes sont présentés dans [3] ; aucun d'entre eux ne s'avère cependant capable d'atteindre les performances de *Rolling Stone*.

A notre connaissance, aucun solveur existant à ce jour n'est donc capable de résoudre les 90 problèmes du benchmark, encore moins de façon optimale. De plus, aucun d'eux ne semble raisonnablement proche de réaliser cet objectif. En effet, *Sokoban Automatic Solver*, qui affiche les performances les plus impressionnantes quant au nombre de problèmes résolus, ne permet pas d'obtenir des solutions optimales.

Il nous semble également important de souligner le fait que même si un solveur était capable d'une telle prouesse, cela ne signifierait pas pour autant que le problème de la résolution d'un problème de Sokoban pourrait être considéré comme solutionné. Ce benchmark de 90 problèmes est en effet loin d'être représentatif de l'ensemble des instances de problèmes de Sokoban existantes. Il contient les 50 problèmes du jeu original auxquels 40 problèmes supplémentaires plus ou moins difficiles ont été ajoutés. Ces problèmes supplémentaires ont été conçus dans l'esprit des problèmes originaux et présentent des configurations similaires.

Ce constat nous permet de mettre en lumière une difficulté supplémentaire présente dans les problèmes de Sokoban et qui provient de l'extrême hétérogénéité des problèmes proposés. La configuration d'un problème joue en effet un rôle prépondérant dans la méthode utilisable pour le résoudre et il semble très difficile de faire émerger une stratégie générale utilisable avec n'importe quel type de configuration. Il y a donc peu de chance que les premiers programmes capables de solutionner les 90 problèmes du benchmark se montrent aussi efficaces face à un autre ensemble de problèmes.

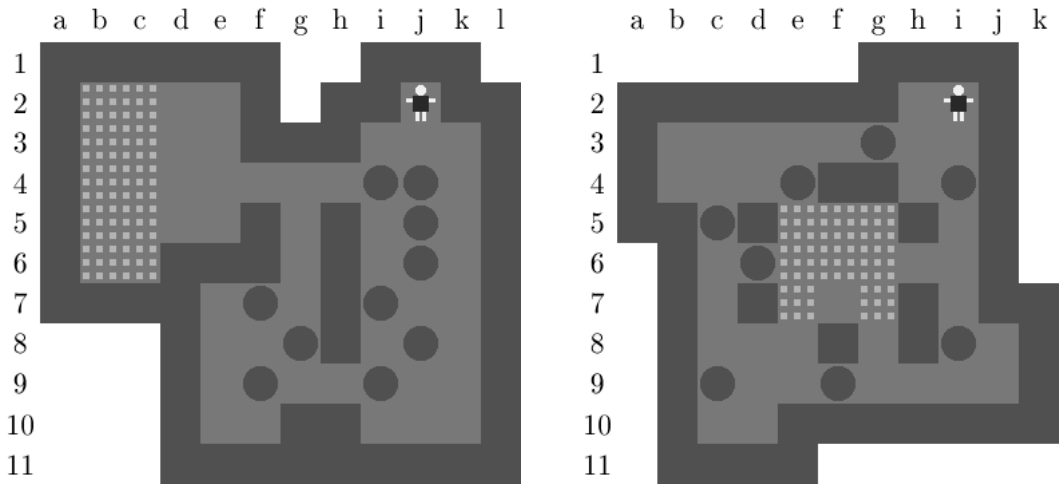

Figure 2.2 – Deux problèmes de Sokoban présentant des configurations très différentes.

Ces remarques ne doivent cependant pas nous conduire à minimiser l'importance de l'objectif vers lequel nous avançons. En effet, même si elle ne constitue pas une fin en soi, la conception d'un solveur capable de résoudre les 90 problèmes du benchmark représentera sans aucun doute un premier pas important vers l'objectif beaucoup plus ambitieux que constitue la création d'un programme capable de rivaliser dans un cadre plus général avec l'intelligence humaine.

Chapitre 3

Chercher

Une introduction aux algorithmes de recherche sera présentée dans ce chapitre. Elle nous conduira à la réalisation d'une première version de notre programme.

3.1 Problèmes de recherche en intelligence artificielle

La méthode la plus immédiate pour résoudre un problème de Sokoban est de le considérer comme un problème de recherche dans un espace d'états. Les problèmes de recherche sont courants en intelligence artificielle. En effet, résoudre un problème consiste souvent à rechercher une bonne solution parmi un ensemble de solutions possibles. Dans un problème complexe, la taille de cet ensemble rend généralement impossible une recherche exhaustive parmi les candidats solutions. La résolution d'un tel problème nécessitera dès lors de faire preuve d'une certaine intelligence afin d'être menée à bien. La taille de l'espace d'états (ou de solutions) est un des critères qui permet d'évaluer la complexité d'un problème de recherche. Pour rappel, celle-ci a été évaluée à 10^{98} pour un problème de Sokoban de taille 20×20.

Les problèmes de recherche de chemin sont un exemple classique de problème de recherche. Dans leur énoncé le plus simple, ils consistent à trouver un chemin entre deux positions d'un labyrinthe (cf. figure 3.1).

L'algorithme élémentaire qui permet de résoudre un tel problème démarre de la position initiale A et explore les successeurs de cette position, *i.e.* les 4 positions adjacentes à A. Il explore ensuite les successeurs des successeurs de A et ainsi de suite jusqu'à ce qu'à ce qu'un de ces successeurs soit la position B. Une manière plus efficace d'effectuer cette recherche sera d'utiliser un algorithme « intelligent » qui privilégiera l'exploration des successeurs les plus proches de B. Nous introduirons plus loin dans ce chapitre le fonctionnement d'un tel algorithme.

Les énigmes, connues pour mettre à l'épreuve nos capacités de raisonnement, peuvent

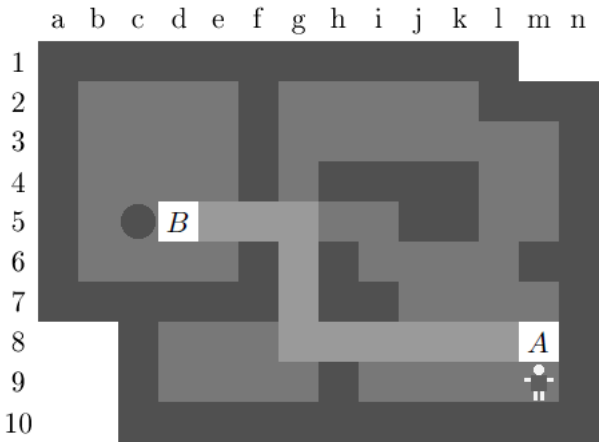

Figure 3.1 – Un exemple de problème de recherche de chemin.

également être abordées comme des problèmes de recherche. Un exemple classique d'énigme présenté dans [5] est *l'énigme du zèbre* :

Cinq maisons alignées,
Cinq nationalités,
Cinq couleurs,
Cinq boissons favorites,
Cinq marques de tabac,
Cinq animaux familiers,
et de plus...

1. *Les numéros des maisons sont 1, 2, 3, 4, 5.*
2. *L'Anglais habite la maison verte.*
3. *L'Espagnol possède un chien.*
4. *On boit du café dans la maison rouge.*
5. *On boit du thé chez l'Ukrainien.*
6. *La maison rouge suit la maison blanche.*
7. *Le fumeur de Old Gold élève des escargots.*
8. *On fume des Gauloises dans la maison jaune.*
9. *On boit du lait au numéro 3.*
10. *Le Norvégien habite au numéro 1.*
11. *Le fumeur de Chesterfield et le propriétaire du renard sont voisins.*
12. *Le fumeur de Gauloises habite à côté du propriétaire du cheval.*
13. *Le fumeur de Lucky Strike boit du jus d'orange.*
14. *Le Japonais fume des Gitanes.*
15. *La maison bleue jouxte celle du Norvégien.*

Qui possède le zèbre... et qui boit de l'eau ?

Une solution de cette énigme attribuera à chacune des cinq maisons cinq caractéristiques, *i.e.* une nationalité, une couleur, une boisson, une marque de tabac et un animal. Or, il y a 5! façons d'attribuer une caractéristique aux cinq maisons. La solution de cette énigme se trouve donc dans un ensemble de $(5!)^5$ solutions possibles. Enumérer et tester l'une

après l'autre un tel nombre de possibilités serait cependant particulièrement inefficace. Une solution plus intelligente sera de construire à partir des indices donnés une suite de solutions partielles. Le dixième indice nous permet par exemple de partir d'une solution partielle dans laquelle la première maison est occupée par le Norvégien. Le quinzième indice nous permet ensuite de compléter cette solution en fixant la couleur de la deuxième maison. Trois solutions partielles peuvent alors être construites pour chacune des possibilités de placer l'Anglais dans la maison verte (deuxième indice). La solution globale de l'énigme peut ainsi être obtenue en complétant peu à peu nos solutions partielles ou en les rejetant quand une contradiction avec l'ensemble des indices apparaît. La résolution de l'énigme s'apparente dès lors à une recherche dans un arbre dans lequel chaque noeud est une solution partielle et a comme successeurs les solutions partielles obtenues en ajoutant l'information contenue dans un nouvel indice.

Une troisième illustration de l'importance des problèmes de recherche dans le domaine de l'intelligence artificielle est le langage de programmation logique *Prolog*. Un algorithme de recherche est en effet au coeur du fonctionnement de ce langage qui est l'un des langages de prédilection des concepteurs d'applications d'intelligence artificielle. Pour répondre à la question qui lui est posée, *Prolog* utilise sa base de connaissances (le programme) afin de réduire la résolution de la question à la résolution d'une sous-question. Il procède ensuite de même avec la sous-question jusqu'à arriver à une sous-question dont la réponse est dans sa base de connaissances ou à une situation d'échec, *i.e.* une sous-question pour laquelle aucune réduction n'est possible. Plusieurs réductions sont parfois possibles pour une sous-question donnée ; *Prolog* procède donc en construisant un arbre, appelé arbre de recherche, dont le parcours se fait en profondeur d'abord. Un exemple d'un tel arbre de recherche est donné dans la figure 3.2. Le lecteur intéressé par une description plus approfondie (et plus formelle) du fonctionnement du langage *Prolog* pourra consulter avec profit le chapitre 11 de [5].

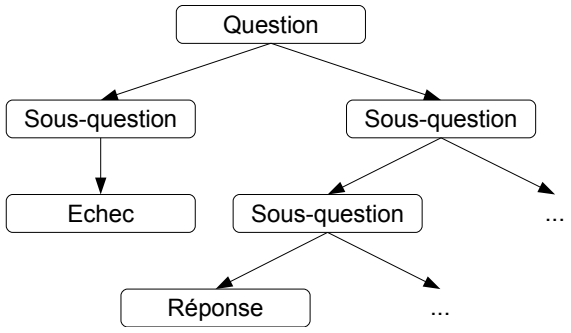

Figure 3.2 – Un arbre de recherche *Prolog*.

3.2 Le Sokoban comme un problème de recherche.

Dans le cas du Sokoban, la structure de l'espace d'états est un graphe dans lequel chaque noeud est une situation de jeu (un état) et a comme successeurs les situations de jeu accessibles en un coup, *i.e.* en une poussée (cf. figure 3.3).

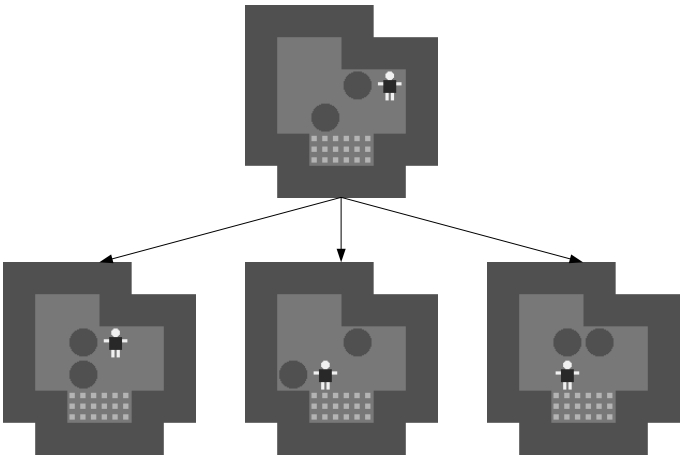

Figure 3.3 – Un noeud du graphe et ses successeurs.

Résoudre le problème consiste donc à trouver dans ce graphe un chemin entre le noeud représentant l'état initial du problème et un noeud représentant un état solution, *i.e.* une situation de jeu dans laquelle tous les goals sont occupés par une pierre. Opérationnellement, les algorithmes de recherche n'explorent pas le graphe mais l'arbre représentant l'ensemble des chemins possibles dans ce graphe. Cet arbre est appelé l'arbre de recherche (cf. figure 3.4).

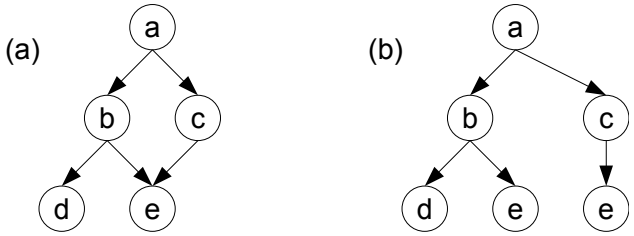

Figure 3.4 – Un graphe (a) et l'arbre de recherche correspondant (b).

Certains graphes d'états peuvent contenir des cycles. Ce sera souvent le cas du graphe correspondant à un problème de Sokoban. En effet, une séquence de poussées peut parfois conduire à revenir dans une situation de jeu rencontrée précédemment. La présence de cycles dans un graphe implique l'existence de chemins infinis dans ce graphe. L'arbre de recherche correspondant à ce graphe aura dès lors une profondeur infinie (cf. figure 3.5). La terminaison de certains algorithmes de recherche ne pourra être garantie sans la présence de mécanismes permettant de déceler la présence de ces cycles. Un algorithme de recherche sera dit *optimal* s'il renvoie toujours la solution de moindre coût, *i.e.* la solution correspondant au chemin de longueur minimale. Un algorithme de recherche sera dit *complet* s'il renvoie toujours une solution dans le cas où au moins un noeud solution existe dans le graphe.

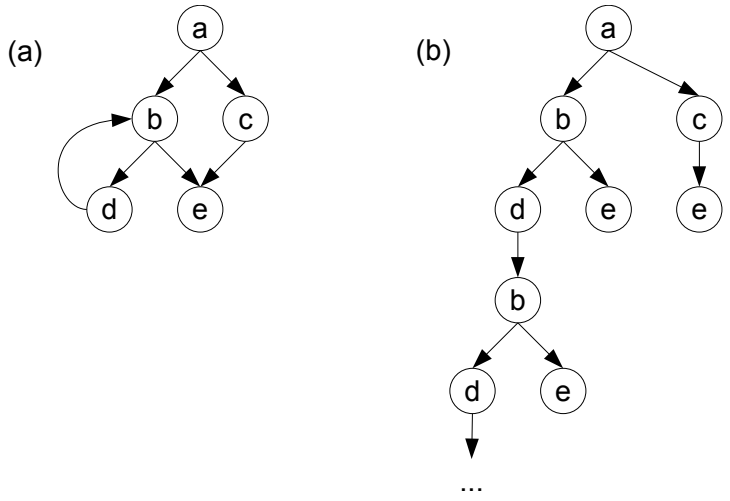

Figure 3.5 – Un arbre de recherche infini (b) dû à la présence d'un cycle (a).

3.3 Algorithmes de recherche classiques

Ces algorithmes sont relativement anciens et sont largement documentés. En particulier, une formalisation à l'aide d'invariants de leur implémentation est décrite dans [1]. Nous nous limiterons dès lors à en donner ici une description générale et nous détaillerons dans une section ultérieure l'implémentation des algorithmes utilisés par notre programme. Par ailleurs, notre présentation rejoindra l'approche utilisée par [8] en présentant d'abord un algorithme général de recherche qui sera ensuite particularisé afin d'implémenter les différentes méthodes. Cet algorithme général nous permettra de souligner la structure commune aux différents algorithmes.

Afin de s'adapter à la résolution d'un problème particulier, un algorithme de recherche prend trois arguments :

1. *start-node* : le noeud représentant la situation initiale du problème ;
2. *goal-node*? : un prédicat qui prend comme argument un noeud et qui renvoie vrai si et seulement si ce noeud est un noeud solution ;
3. *find-successors* : une fonction qui prend comme argument un noeud et qui renvoie la liste des successeurs de ce noeud.

Le coeur d'un algorithme de recherche est une liste de noeuds *open* contenant initialement le noeud représentant la situation initiale du problème, *i.e.* le noeud *start-node*. Soit $open_n$ le contenu de la liste open à l'étape n de l'algorithme et $first\text{-}node$ le premier élément (la tête) de cette liste, $open_{n+1}$ sera constitué du reste de $open_n$, *i.e.* la liste $open_n$ amputée de son premier élément, auquel on aura ajouté les successeurs du noeud $first\text{-}node$. Si à une étape donnée, la tête de *open* est un noeud solution, ce noeud est renvoyé et l'algorithme se termine. Notre algorithme général de recherche peut dès lors être décrit par le programme Scheme [10] :

```
(define search
  (lambda (start-node goal-node? find-successors)
    (let loop ((open (list start-node)))
      (if (null? open)
          #f
          (let ((first-node (car open)))
            (if (goal-node? first-node)
                first-node
                (loop (update-open ...)))))))))
```

Les méthodes de recherche diffèreront principalement dans la manière dont les successeurs de *first-node* seront ajoutés dans le reste de $open_n$. Décrire leur implémentation consistera donc à définir la fonction *update-open*.

Les algorithmes de recherche classiques se divisent en deux classes : (1) les algorithmes de recherche aveugle et (2) les algorithmes de recherche informée. Dans les algorithmes de recherche aveugle, le parcours de l'arbre se fait de façon systématique. A contrario, dans les algorithmes de recherche informée, une fonction particulière, appelée une heuristique, est utilisée afin d'orienter le parcours dans la bonne direction. Nous commencerons cet exposé, par une présentation des principaux algorithmes de recherche aveugle.

3.3.1 Méthodes de recherche aveugle

Recherche en largeur d'abord (Breadth-first search)

Dans cette première méthode, le parcours de l'arbre se fait en largeur d'abord. Un noeud situé à une profondeur d_1 sera donc toujours exploré avant un noeud de profondeur d_2 si $d_2 > d_1$ (cf. figure 3.6).

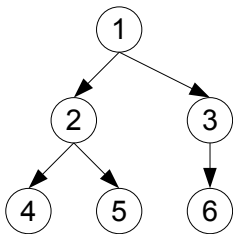

Figure 3.6 – Parcours en largeur d'abord d'un arbre de recherche.

L'implémentation d'un tel parcours se fait en ajoutant les successeurs du noeud courant, *i.e.* le noeud situé en tête de la liste open et que nous avons appelé *first-node*, à la fin du reste de la liste *open*. La recherche en largeur d'abord sera donc implémentée en définissant la fonction *update-open* de notre algorithme de recherche général par :

```
(lambda (first-node open find-successors)
  (append (cdr open) (find-successors first-node)))
```

La recherche en largeur d'abord est optimale puisque le noeud solution de profondeur minimale sera toujours atteint avant tout autre noeud solution situé plus profondément dans l'arbre. De même, sous l'hypothèse qu'un noeud solution existe, le type de parcours effectué garantit qu'une solution sera toujours trouvée en un temps fini. En effet, si un arbre peut avoir une profondeur infinie, le nombre de noeuds situés à une profondeur donnée sera, en revanche, toujours fini puisque le nombre de successeurs issus d'un noeud donné sera lui-même toujours fini. La recherche en largeur d'abord est donc également complète. Elle semble par conséquent être une méthode idéale. Néanmoins, une étude de sa complexité permet de mettre en lumière une faiblesse majeure dont souffre la méthode par rapport à d'autres algorithmes de recherche. La complexité d'un algorithme de recherche s'exprime en temps et en espace : la *complexité temporelle* correspond au nombre de noeuds qui auront dû être développés pour atteindre un noeud solution ; la *complexité spatiale* équivaut quant à elle au nombre maximal de noeuds qui auront été présents simultanément en mémoire pendant le processus de résolution du problème.

Afin d'étudier la complexité de l'algorithme de recherche en largeur d'abord, nous utiliserons l'arbre représenté dans la figure 3.7. Nous définissons b comme étant le facteur de branchement de cet arbre et d comme étant la profondeur du noeud solution recherché. Dans le pire des cas, *i.e.* si le noeud solution se trouve le plus à droite, le nombre de noeuds qui devront être développés pour atteindre le noeud solution est exprimé par la formule $(\sum_{i=0}^{d} b^i) - 1$. La complexité temporelle de notre algorithme sera donc de $O(b^d)$.

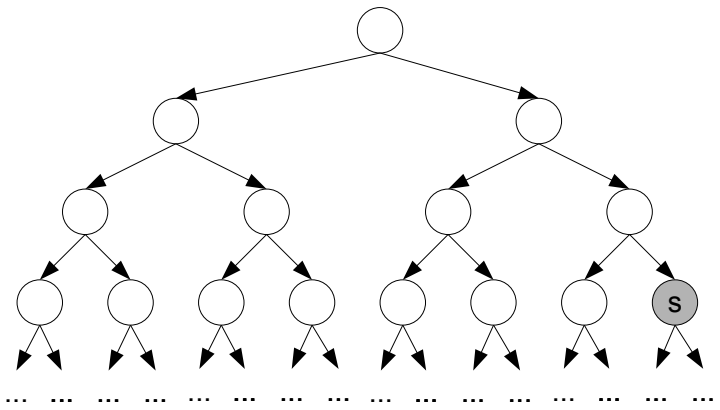

Figure 3.7 – Un arbre de recherche.

Dans cet exemple, le facteur de branchement b de l'arbre vaut 2 puisque chaque noeud possède deux successeurs. Le noeud solution recherché est le noeud s qui se trouve à une profondeur $d = 3$.

Par ailleurs, et toujours dans le cas le moins favorable, le nombre de noeuds présents en mémoire sera maximal lorsque le noeud solution sera atteint. A ce moment, les $b^d - 1$ noeuds se trouvant à gauche du noeud solution auront chacun été développés en b successeurs et le nombre total de noeuds présents en mémoire sera dès lors de $((b^d - 1) \times b) + 1$ noeuds. La complexité spatiale de la recherche en largeur d'abord est donc de $O(b^d)$. Cette complexité spatiale est le point faible de l'algorithme. En effet, les méthodes présentées par la suite et qui se baseront sur un parcourir non plus en largeur mais en profondeur de l'arbre, nous permettrons d'obtenir une complexité spatiale de $O(b \times d)$.

Recherche en profondeur d'abord (Depth-first search)

Cette méthode correspond à un parcours de l'arbre de recherche en profondeur d'abord. Dans un tel parcours, le successeur le plus à gauche d'un noeud est exploré en premier et on ne revient à l'exploration des autres successeurs que si l'exploration du sous-arbre issu de ce successeur ne permet pas d'atteindre un noeud solution (cf. figure 3.8).

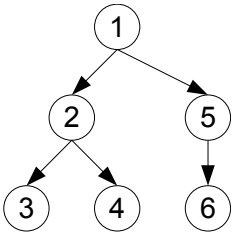

Figure 3.8 – Parcours en profondeur d'abord d'un arbre de recherche.

Cette méthode s'implémente en ajoutant les successeurs du noeud courant en tête du reste de la liste *open*. La fonction *update-open* de notre algorithme de recherche générale sera dès lors définie comme suit :

```
(lambda (first-node open find-successors)
  (append (find-successors first-node) (cdr open)))
```

Ce type de parcours permet d'obtenir une complexité spatiale plus intéressante que la recherche en largeur d'abord. En effet, le nombre de noeuds présents en mémoire ne dépendra plus du nombre de noeuds situés à la profondeur du noeud courant mais du nombre de noeuds présents sur la branche à laquelle il appartient. Pour une profondeur d donnée et un facteur de branchement b, le nombre de noeuds en mémoire sera ici maximal quand le noeud courant, *i.e.* le noeud situé en tête de la liste *open*, sera le noeud de profondeur d situé le plus à gauche de l'arbre. A cet instant, la liste *open* contiendra, en plus du noeud courant, les $d \times (b-1)$ successeurs non encore développés des d noeuds qui précédent le noeud courant sur la branche à laquelle il appartient. Nous passerons ainsi d'une complexité exponentielle de $O(b^d)$ pour la recherche en largeur d'abord à une complexité linéaire de $O(b \times d)$ pour la recherche en profondeur d'abord.

Cette méthode de recherche n'est cependant pas optimale puisque le premier noeud solution rencontré sera le noeud solution situé le plus à gauche dans l'arbre et non celui situé à la profondeur la plus faible (cf. figure 3.9).

Inconvénient plus grave, la méthode n'est pas complète. En effet, comme nous l'avons vu, l'arbre de recherche aura une profondeur infinie si le graphe auquel il correspond contient au moins un cycle. Ce sera la plupart du temps le cas dans un problème de Sokoban. Or, si une branche infinie se trouve à gauche du noeud solution le plus à gauche, le parcours de cette branche infinie ne se terminera jamais et le noeud solution ne sera jamais atteint (cf. figure 3.10). La terminaison de l'algorithme ne peut donc être garantie et la méthode n'est par conséquent pas complète.

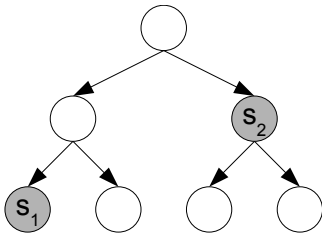

Figure 3.9 – Arbre pour lequel la recherche en profondeur d'abord ne sera pas optimale.

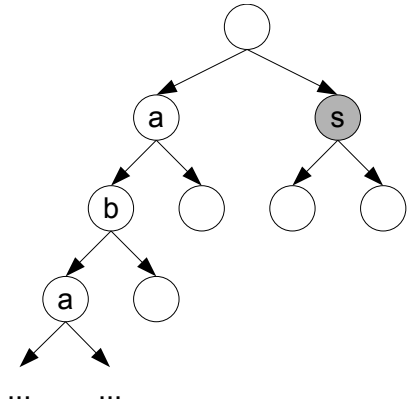

Figure 3.10 – Arbre pour lequel la recherche en profondeur d'abord aura une exécution infinie. Le noeud solution s ne sera donc jamais atteint.

Recherche limitée en profondeur (Depth-limited search)

La recherche limitée en profondeur est une variante de la méthode précédente dans laquelle une profondeur maximale de recherche est fixée a priori. Un noeud situé en tête de la liste *open* et qui se trouve à la profondeur maximale n'est donc plus développé et ses successeurs sont ignorés. Soit la fonction *depth* qui prend comme unique argument un noeud et qui renvoie la profondeur de ce noeud, et *depth-limit* la profondeur maximale de recherche, la méthode s'implémente en définissant la fonction *update-open* par :

```
(lambda (first-node open find-successors)
  (if (= (depth first-node) depth-limit)
      (cdr open)
      (append (find-successors first) (cdr open))))
```

Cette variante règle le problème de la terminaison de l'algorithme de recherche en profondeur d'abord mais ne garantit toujours pas l'optimalité de la solution renvoyée. Par ailleurs, la méthode ne peut être considérée comme complète que par rapport au fragment de l'arbre formé par les noeuds situés à une profondeur inférieure ou égale à la profondeur maximale. En effet, un noeud solution ne sera jamais atteint s'il se trouve à une profondeur plus importante que cette profondeur maximale.

La complexité de cet algorithme s'étudie par rapport à la profondeur maximale fixée ; nous l'appellerons d_{max}. Le cas le moins favorable est ici celui d'un arbre dans lequel le noeud solution recherché est le noeud de profondeur d_{max} situé le plus à droite. Soit b le facteur de branchement de cet arbre, $(\sum_{i=0}^{d} b^i) - 1$ noeuds précéderont le noeud solution dans le parcours effectué. La complexité temporelle de cet algorithme sera donc de $O(b^{d_{max}})$. L'occupation en mémoire d'une recherche en profondeur d'abord dépend, comme nous l'avons vu, du nombre de feuilles présentes sur la branche à laquelle appartient le noeud courant. Or, dans le cas de la recherche en profondeur limitée, la longueur d'une branche sera bornée par la profondeur maximale fixée. Le complexité spatiale de cet algorithme sera donc de $O(b \times d_{max})$.

Recherche itérative en profondeur (Iterative-deepening)

La recherche itérative en profondeur est une méthode qui concilie l'optimalité de la recherche en largeur d'abord et la complexité spatiale linéaire de la recherche en profondeur d'abord. La méthode utilise une recherche limitée en profondeur, dont la profondeur maximale est augmentée itérativement tant qu'un noeud solution n'est pas rencontré. Son implémentation peut être décrite comme suit :

```
(define iterative-deepening
  (lambda (start-node goal-node? find-successors)
    (let loop ((depth-limit 0))
      (or (limited-depth-first depth-limit start-node goal-node? find-successors)
          (loop (+ depth-limit 1))))))
```

L'optimalité de l'iterative-deepening découle du principe même de son fonctionnement. En effet, le premier noeud solution rencontré sera toujours le noeud solution de profondeur minimale puisque tous les noeuds situés à une profondeur inférieure auront déjà été explorés au moins une fois lors des itérations précédant l'itération courante.

La complexité spatiale de la méthode découlera quant à elle du type de parcours qui sera utilisé dans chaque itération. Soit un arbre dont le facteur de branchement est b et dans lequel le noeud solution recherché se trouve à la profondeur d, ce noeud solution sera atteint à la $(d+1)^{\text{ième}}$ itération et la profondeur maximale de recherche sera dès lors bornée par d. La parcours utilisé étant un parcours en profondeur d'abord, la complexité spatiale de l'iterative-deepening sera donc de $O(b \times d)$.

On pourrait a priori penser que l'optimalité et la complexité spatiale raisonnable de l'iterative-deepening sont obtenues au prix d'une augmentation importante de sa complexité temporelle. En effet, à chaque itération, les noeuds qui avaient été explorés lors de l'itération précédente sont explorés à nouveau. Une analyse plus poussée permet cependant de se rendre compte que la quantité de travail supplémentaire effectué est en fait modeste.

Dans l'arbre que nous avons utilisé précédemment, $(d+1)$ itérations étaient nécessaires pour atteindre le noeud solution recherché. En partant de cet exemple, on peut établir la

formule qui nous permettra de calculer le nombre de noeuds visités par l'iterative-deepening. En effet, lors de ces $(d+1)$ itérations, le noeud racine aura été visité $(d+1)$ fois, les b noeuds de profondeur 1 auront été visités d fois, les b^2 noeuds de profondeur 2 auront été visités $(d-1)$ fois, ... et les b^d noeuds de profondeur d auront été visités une seule fois. On obtient ainsi la formule $\sum_{i=0}^{d}((d+1-i) \times b^i)$ qui nous permet de conclure que la complexité temporelle de l'iterative-deepening reste de $O(b^d)$.

Afin d'illustrer cette étonnante conclusion, nous utiliserons un exemple numérique extrait de [8]. En reprenant une dernière fois notre arbre précédent et en fixant le facteur de branchement b à 10 et la profondeur du noeud solution d à 5, le nombre de noeuds visités par la dernière itération d'une recherche itérative en profondeur est obtenu par la formule $\sum_{i=0}^{d} b^i$ et vaudra

$$1 + 10 + 100 + 1.000 + 10.000 + 100.000 = 111.111$$

Le nombre total de noeuds visités par l'ensemble des itérations de l'algorithme sera quant à lui calculé par la formule que nous avons obtenue précédemment et sera égal à

$$6 + 50 + 400 + 3.000 + 20.000 + 100.000 = 123.456$$

L'effort supplémentaire induit par les multiples itérations de l'algorithme est donc, comme nous l'avions annoncé, relativement modeste. En effet, il ne représente ici qu'environ 11 % du temps total nécessaire à la découverte d'un noeud solution.

Soit b le facteur de branchement d'un arbre de recherche, d la profondeur à laquelle se trouve le noeud solution recherché et d_{max} la profondeur maximale fixée, le tableau 3.1 résume les complexités temporelles et spatiales ainsi que les caractéristiques des différents algorithmes de recherche aveugle que nous venons de présenter.

Algorithme	**Complexité temporelle**	**Complexité spatiale**	**Optimalité**	**Complétude**
breadth-first search	b^d	b^d	oui	oui
depth-limited search	$b^{d_{max}}$	$b \times d_{max}$	non	non
iterative-deepening	b^d	$b \times d$	oui	oui

Tableau 3.1 – Complexité et caractéristiques des différents algorithmes de recherche aveugle.

L'iterative-deepening allie à la fois une complexité spatiale raisonnable et la possibilité d'obtenir une solution optimale. Il apparaît donc comme le choix le plus intéressant lorsque, comme dans le cas d'un problème de Sokoban, la taille de l'espace d'états à explorer est importante et que la profondeur de la solution recherchée n'est pas connue. Dans cette situation, la taille de l'espace d'états rend clairement inadaptée la recherche en largeur d'abord. Par ailleurs, la non-optimalité d'une recherche en profondeur d'abord ne signifie pas seulement que la solution trouvée risque de ne pas être optimale. Elle implique également que la complexité temporelle rencontrée sera supérieure à $O(b^d)$. En effet, dans l'hypothèse où

l'arbre n'a pas une profondeur infinie, le parcours effectué par l'algorithme sera susceptible de descendre jusqu'à la profondeur maximale m de cet arbre. Or cette profondeur m sera probablement largement supérieure à la profondeur d du noeud solution recherché et le temps $O(b^m)$ nécessaire à la résolution du problème sera par conséquent largement supérieur au temps $O(b^d)$ rencontré avec l'iterative-deepening.

3.3.2 Méthodes de recherche informée

Dans les algorithmes de recherche aveugle, le parcours de successeur en successeur est systématique et aucun noeud n'est privilégié par rapport à un autre. A contrario, les algorithmes de recherche informée utilisent une fonction afin d'évaluer la distance (le nombre de noeuds) qui sépare un noeud donné du noeud solution recherché. Cette fonction est appelée une *heuristique*. Elle exploite les connaissances a priori du domaine concerné afin d'orienter la recherche vers les noeuds les plus prometteurs. Un exemple simple d'heuristique est celle qui est habituellement utilisée dans les problèmes de recherche d'un chemin dans un labyrinthe, problèmes que nous avons introduits précédemment. Soit une position de départ de coordonnées $(x1, y1)$ et la position cible de coordonnées $(x2, y2)$, cette heuristique, appelée *distance de Manhattan*, est la fonction f définie par $f((x1, y1), (x2, y2))) = |x1 - x2| + |y1 - y2|$. Elle calcule la distance « à vol d'oiseau » entre la position $(x1, y1)$ et la position $(x2, y2)$, sachant que les déplacements ne se font que dans les quatre directions : gauche, haut, droite et bas.

Il est intéressant de remarquer que l'efficacité de cette heuristique dépendra grandement de la configuration du labyrinthe. En effet, dans une configuration très ouverte, la destination pourra effectivement être atteinte en utilisant un parcours plus ou moins direct et dont la longueur sera proche de la distance de Manhattan (cf. figure 3.11). A contrario, dans une configuration très fermée et tortueuse, de nombreux détours seront nécessaires pour atteindre la position cible et l'estimation fournie par la distance de Manhattan ne procurera donc aucune information réellement utile (cf. figure 3.12).

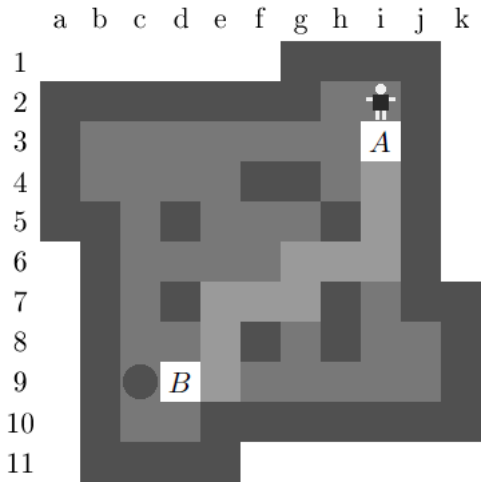

Figure 3.11 – Configuration dans laquelle la distance de Manhattan est efficace.

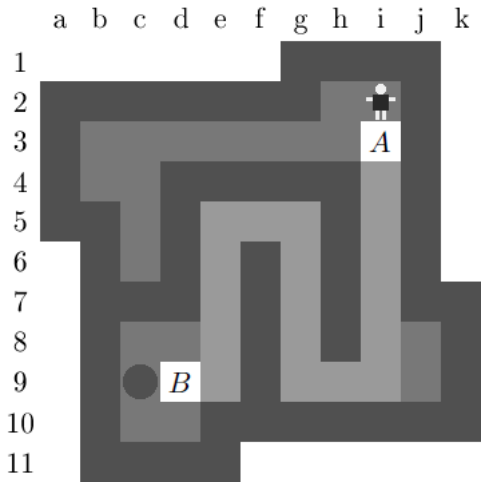

Figure 3.12 – Configuration dans laquelle la distance de Manhattan est inefficace.

Développer une fonction d'heuristique, même pour un problème aussi simple, n'est donc pas une chose facile. On peut dès lors imaginer la difficulté que cela représente dans le cas d'un problème aussi complexe que celui du Sokoban. Nous reviendrons sur ce dernier point à la fin de ce chapitre.

Une heuristique sera dite *admissible* si elle ne surestime jamais la distance minimale entre le noeud à évaluer et le noeud solution recherché. C'est le cas de la distance de Manhattan puisque la distance à parcourir entre deux points ne pourra jamais être inférieure à la distance à vol d'oiseau. Certains algorithmes de recherche informée requièrent l'utilisation d'une heuristique admissible pour garantir leur optimalité.

Recherche gloutonne (Greedy search)

La recherche gloutonne effectue un parcours de l'arbre de recherche dans lequel le noeud le plus prometteur est exploré en premier. On appelle cette stratégie une recherche par le meilleur d'abord (best-first search). Dans ce type d'algorithme, la liste *open* est maintenue triée de telle sorte que son premier élément soit toujours le noeud le plus prometteur par rapport à une fonction de coût donnée. Dans le cadre d'une recherche gloutonne, ce noeud est celui qui est estimé comme étant le plus proche du noeud solution recherché. Soit une fonction d'heuristique h telle que nous l'avons définie plus haut, le noeud situé en tête de la liste *open* sera donc le noeud n pour lequel $h(n)$ est minimal.

Soit *insert* une fonction qui prend comme arguments deux listes de noeuds et une fonction de coût, et qui renvoie une liste dans laquelle les éléments des deux listes sont triés par ordre croissant de coût, la recherche gloutonne s'implémente en donnant à la fonction *update-open* de notre algorithme de recherche général la définition :

```
(lambda (first-node open find-successors)
  (insert (find-successors first-node) (cdr open) h))
```

Si une bonne fonction d'heuristique est utilisée, la recherche gloutonne permettra de réduire substantiellement le nombre de noeuds qui devront être explorés avant d'atteindre le noeud solution. Cependant, cette méthode souffre des mêmes inconvénients que la recherche en profondeur d'abord : elle n'est ni optimale, ni complète. Un exemple de sa non-optimalité est présenté dans [8]. Par ailleurs, le parcours effectué par une recherche gloutonne sera, dans le pire des cas, le même que celui effectué par une recherche en profondeur d'abord. Sa complexité temporelle sera dès lors de $O(b^m)$ où b est le facteur de branchement de l'arbre et m sa profondeur maximale. De plus, puisque tous les noeuds seront gardés en mémoire, la complexité spatiale de l'algorithme sera également de $O(b^m)$. Ce dernier sera donc inadapté pour l'exploration d'espaces d'états de très grande taille.

Algorithme A*

L'algorithme A* est une variante de la recherche par le meilleur d'abord dans laquelle le calcul du coût d'un noeud fait intervenir l'effort qui a été fourni pour atteindre ce noeud. Soit la fonction g qui prend comme argument un noeud et qui renvoie sa profondeur, et la fonction d'heuristique h qui prend comme argument un noeud et qui renvoie l'évaluation de la distance qui sépare ce noeud d'un noeud solution, la fonction f qui prend comme argument un noeud n et qui renvoie son coût est définie par $f(n) = g(n) + h(n)$.

L'implémentation de l'algorithme A* est similaire à celle de la recherche gloutonne puisque seule la fonction utilisée pour évaluer le coût d'un noeud est différente. La fonction *update-open* sera ainsi définie par :

```
(lambda (first-node open find-successors)
  (insert (find-successors first-node) (cdr open) f))
```

Il est prouvé dans [8] que la fonction de coût utilisé permet à l'algorithme A* d'être optimal et complet à condition que la fonction h soit une heuristique admissible. Néanmoins, à l'instar de la recherche gloutonne, la complexité spatiale de l'algorithme A* est exponentielle.

Algorithme IDA*

L'algorithme IDA* applique le principe utilisé par l'iterative-deepening à la recherche informée. La limite incrémentée à chaque itération n'est ici plus définie par rapport à la profondeur d'un noeud mais par rapport au coût de celui-ci. Soit f la fonction de coût que nous avons définie précédemment et *f-limit* la limite fixée de l'itération courante, seuls les noeuds n pour lesquels $f(n) < f\text{-}limit$ seront explorés au cours de cette itération. L'algorithme IDA* peut ainsi s'implémenter par la fonction :

```
(define ida-star
  (lambda (start-node goal-node? find-successors)
```

```
(let loop ((f-limit 0))
  (or (f-limited-depth-first f-limit start-node goal-node? find-successors)
      (loop (+ f-limit 1))))))
```

La fonction *f-limited-depth-first* s'implémente quant à elle en définissant la fonction *update-open* de notre algorithme de recherche général par :

```
(lambda (first-node open find-successors)
  (if (= (f first-node) f-limit)
      (cdr open)
      (append (find-successors first-node) (cdr open))))
```

Le fonctionnement itératif utilisé permet à l'algorithme IDA* d'avoir la complexité spatiale linéaire d'un parcours en profondeur d'abord tout en étant optimal et complet. Ce sera donc la méthode à privilégier pour l'exploration d'espaces d'états de taille gigantesque tels que ceux qui sont rencontrés lors de la résolution de problèmes de Sokoban. Cette méthode de recherche est ainsi celle qui est utilisée par le solveur *Rolling Stone*.

3.4 Implémentation version 1

La première version de notre programme reposera sur l'utilisation des algorithmes de recherche classiques que nous avons introduits dans la première partie de ce chapitre. Cette première version nous permettra d'une part de démontrer que ces méthodes sont loin d'être suffisantes pour s'attaquer aux 90 problèmes de notre benchmark et d'autre part de mettre en place un ensemble d'outils qui nous seront nécessaires à l'implémentation des protocoles de résolution plus évolués et plus efficaces que nous présenterons par la suite.

3.4.1 Représentation d'un problème de Sokoban

Un problème de Sokoban est défini par quatre éléments : (1) la position de départ du joueur, (2) la position de départ des pierres, (3) la disposition des murs et (4) la disposition des goals.

Le concept de position est commun à ces quatre éléments. Afin de pouvoir représenter la position d'un objet, nous utiliserons l'indice de cette position dans le tableau de jeu (cf. figure 3.13 (a)). Ces indices sont attribués de gauche à droite et de haut en bas.

Une représentation sous forme de coordonnées, telle que l'illustre la figure 3.13 (b), pourrait sembler plus naturelle mais nous conduirait à effectuer des opérations inutiles. En effet, comme nous le verrons plus loin, le codage de certains éléments du jeu, la position des murs par exemple, se fera sous la forme d'un vecteur. Or, pour déterminer si une position particulière est occupée par un mur, les coordonnées de cette position devraient être transformées en un indice afin de pouvoir consulter l'entrée correspondante du vecteur. Dans le

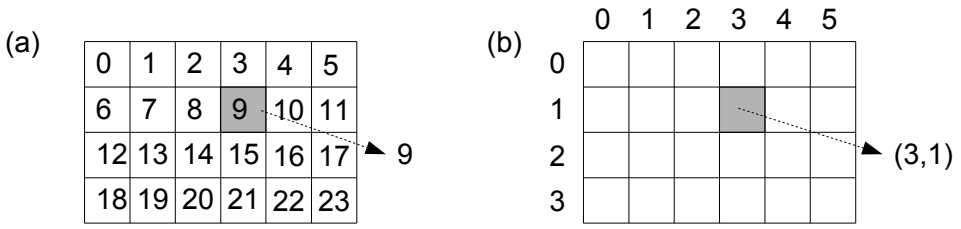

Figure 3.13 – Représentation d'une position par son indice dans le tableau de jeu (a) et par ses coordonnées (b).

cas d'une opération aussi courante, ces transformations répétées auraient des conséquences négatives non négligeables sur les performances de notre programme. L'utilisation d'un indice pour représenter une position nous permettra donc de faire l'économie de ces opérations de transformation.

Par ailleurs, la représentation sous forme d'indices, nous permettra de réaliser facilement les deux opérations que nous devrons effectuer sur les positions : (1) déterminer les successeurs d'une position et (2) déterminer la position d'arrivée d'une poussée.

Soit *max-col* le nombre de colonnes que contient le tableau de jeu, les indices des quatre successeurs d'une position se trouvant à l'indice p s'obtiennent en utilisant les formules de la table 3.2 (cf. figure 3.14 (a)).

Successeur	**Indice**
gauche	$p - 1$
haut	$p - max\text{-}col$
droite	$p + 1$
bas	$p + max\text{-}col$

Tableau 3.2 – Indices des successeurs d'une position d'indice p.

De façon similaire, soit *man-pos* l'indice de la position du joueur et p l'indice de la position d'une pierre contiguë, l'indice de la position d'arrivée de cette pierre suite à une poussée s'obtient par la formule $p + (p - man\text{-}pos)$ (cf. figure 3.14 (b)).

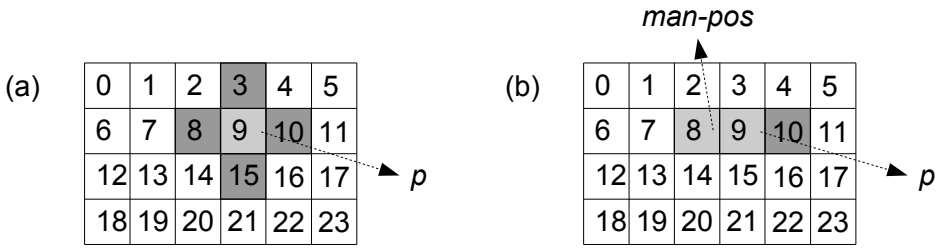

Figure 3.14 – Détermination des successeurs d'une position (a) et détermination de la position d'arrivée d'une poussée (b).

Les positions du joueur et des pierres définissent une situation de jeu donnée. Ces deux informations sont associées à chaque noeud de l'arbre de recherche. Notre préoccupation

sera dès lors de les stocker de la manière la plus compacte possible. La position du joueur[1] sera stockée telle quelle dans une variable *man-pos*. La position des n pierres sera stockée dans une liste *stone-list* de n positions.

Cependant la représentation de la position des pierres sous forme d'une liste ne sera pas toujours adaptée. En effet, soit une liste de n pierres, déterminer si une position donnée est occupée par une pierre prendra un temps $O(n)$. Or, dans les algorithmes nécessitant un parcours du tableau de jeu cette opération sera répétée de nombreuses fois. Il deviendra alors intéressant de stocker la position des pierres dans un vecteur *stone-vector*. Ce vecteur sera tel que *stone-vector*$[i]$ sera vrai ssi la position i est occupée par une pierre. La création de ce vecteur prendra un temps $O(n)$ au début de l'algorithme et nous permettra ensuite de déterminer si une pierre occupe une position donnée en un temps $O(1)$. La complexité temporelle liée à la réalisation de m tests passera dès lors de $O(m \times n)$ à $O(n + m)$.

La disposition des murs est une information statique et sera stockée dans un vecteur *wall-vector* selon le même principe que le vecteur *stone-vector* ; *wall-vector*$[i]$ sera vrai ssi une pierre se trouve à la position i. La disposition des n goals, qui est également statique, sera quant à elle représentée sous la forme d'une liste de n positions, *goal-list*, et sous la forme d'un vecteur *goal-vector* toujours selon le même principe que *stone-vector*.

Le mode de représentation exacte des vecteurs *stone-vector*, *wall-vector* et *goal-vector* est au niveau de l'implémentation légèrement différent de celui qui vient d'être présenté. Les deux premières entrées de chaque vecteur sont en effet utilisées pour stocker les valeurs *max-col* et *max-row*, qui sont respectivement le nombre de colonnes et le nombre de lignes que contient le tableau de jeu. Comme l'illustre la figure 3.15, cette modification est sans conséquence puisqu'il suffit d'attribuer les indices au tableau de jeu en commencant à la valeur 2, pour retrouver une correspondance exacte entre ces indices et les indices des entrées du vecteur. Les opérations sur les positions présentées plus haut restent également valables sans aucune modification.

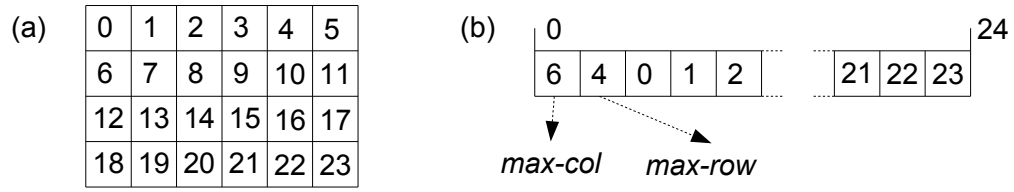

Figure 3.15 – Représentation du tableau de jeu (a) sous la forme d'un vecteur (b).

3.4.2 Recherche des poussées possibles

Pour qu'une pierre puisse être poussée dans une direction, il faut d'une part que (1) la position adjacente soit libre, *i.e.* qu'elle ne soit occupée ni par un mur, ni par une autre pierre, et d'autre par que (2) la position se trouvant dans la direction opposée puisse être atteinte par le joueur. Une première idée est de parcourir la liste des pierres et de tester pour chacune d'elles et pour chaque direction les deux conditions précitées. Cette méthode

[1] Dans la suite de ce travail, nous identifierons une position à son indice dans le tableau de jeu.

suppose néanmoins, dans le pire des cas, d'effectuer pour chaque pierre quatre recherches d'un chemin vers la position du joueur. Une idée plus intéressante est de calculer une fois pour toutes l'ensemble des positions accessibles par le joueur et de coder cet ensemble dans un vecteur *man-vector* tel que *man-vector*$[i]$ soit vrai ssi la position i est accessible par le joueur. Par la suite, déterminer si une position est accessible par le joueur serait ainsi aussi rapide ($O(1)$) que de déterminer si une position est occupée par un mur.

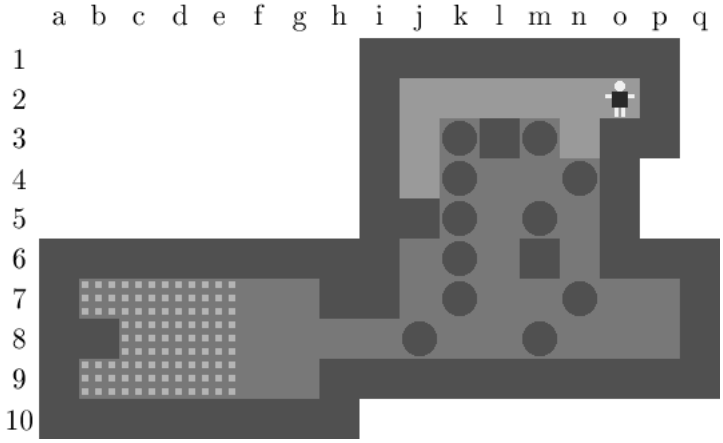

Figure 3.16 – Ensemble des positions accessibles par le joueur (en gris clair).

L'observation de la figure 3.16 nous suggère néanmoins une troisième méthode qui va nous permettre de combiner la détermination de l'ensemble des positions accessibles par le joueur et le parcours de différentes pierres en un seul traitement. En effet, la zone constituée par cet ensemble est délimitée par un certain nombre de murs et par les pierres accessibles par le joueur. L'exploration du tableau de jeu nécessaire à sa détermination passera donc par toutes ces pierres et par toutes les positions à partir desquelles le joueur serait susceptible de les pousser. La seule opération supplémentaire à effectuer sera dès lors de vérifier si la position d'arrivée de la pierre est libre. Ces considérations nous amènent à l'algorithme suivant :

```
(define find-stone-pushes
  (lambda (man-pos stone-list wall-vector)
    (let ((closed-vector (make-closed-vector ...))
          (begin
            (vector-set! closed-vector man-pos #t)
            (let loop ((open (list man-pos))
                       (push-list '()))
              (if (null? open)
                  push-list
                  (let ((first-pos (car open)))
                    (let loop2 ((successor-list (find-coord-successors ...))
                                (updated-open (cdr open))
                                (updated-push-list push-list))
                      (if (null? successor-list)
                          (loop updated-open updated-push-list)
                          (let ((first-successor (car successor-list))
                                (loop2 (cdr successor-list)
```

```
                                    (update-open ...)
                                    (update-push-list ...)))))))))))))))
```

La liste *open* est la liste des positions à explorer. Le vecteur *closed-vector* est tel que *closed-vector*$[i]$ est vrai ssi la position i a déjà été visitée. Ce vecteur permet d'éviter qu'une même position soit explorée deux fois. Lorsqu'une position est explorée, la liste de ses successeurs, *i.e.* la liste des quatre positions adjacentes, est parcourue. Si un successeur est une position accessible au joueur et qu'elle n'a pas encore été visitée, elle est ajoutée dans la liste *open* et est marquée comme visitée. La fonction *update-open* est dès lors définie par :

```
(define update-open
  (lambda (first-successor updated-open)
    (if (or (wall? ...) (stone? ...) (closed? ...))
        updated-open
        (begin
          (vector-set! closed-vector first-successor #t)
          (cons first-successor updated-open)))))
```

Par ailleurs, si un successeur est une position occupée par une pierre, on regarde si la position située « derrière » cette pierre par rapport à la position explorée est libre. Si c'est le cas, une poussée est possible et on ajoute cette poussée à la liste. Ainsi, *updated-push-list* est la fonction :

```
(define update-push-list
  (lambda (first-successor updated-push-list)
    (if (and (stone? ...) (pushables? ...))
        (cons (make-push ...) updated-push-list)
        updated-push-list)))
```

3.4.3 Elimination des clones dans l'arbre de recherche

Dans le Sokoban, comme dans beaucoup d'autres jeux, une même situation de jeu peut être atteinte par différentes séquences de coups. De nombreux clones d'une même situation de jeu seront par conséquent présents dans l'arbre de recherche. L'identification et l'élimination de ces clones s'avère dès lors capitale afin d'éviter de développer inutilement plusieurs fois le même sous-arbre (cf. figure 3.17).

Une table de transposition, aussi appelée table de hashage, va nous permettre de déterminer en un temps $O(1)$ si une situation de jeu a déjà été rencontrée. Son principe sera de générer à partir d'une situation de jeu donnée une valeur numérique unique et d'utiliser cette valeur comme indice dans une table. Cette valeur unique assure une correspondance bijective entre une situation de jeu donnée et une entrée de la table. Cette table est dès lors utilisée comme une base de données permettant de connaître le statut, déjà rencontrée ou non, d'une situation de jeu. Lors de l'opération de développement d'un noeud, le statut de chaque successeur est lu dans la table. Si le successeur n'a pas encore été rencontré, il est

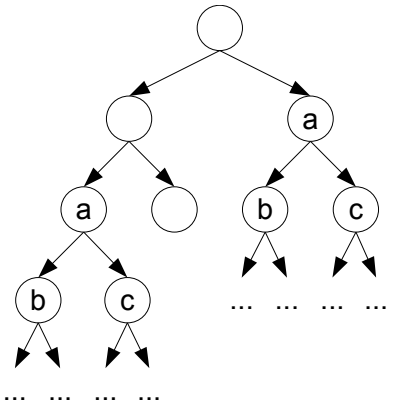

Figure 3.17 – Développement inutile d'un même sous-arbre.

ajouté à la liste des successeurs du noeud et une valeur particulière est placée dans l'entrée de la table qui lui correspond. Si le successeur a déjà été rencontré, il est simplement ignoré.

Avant de définir la fonction, appelée fonction de hashage, qui nous permettra de générer la valeur numérique unique, il nous faut définir précisément les informations qui caractérisent une situation de jeu. Dans un problème de Sokoban, une situation de jeu est définie par deux éléments : (1) la position du joueur et (2) la position des pierres. Puisque nous avons fait le choix de ne prendre en compte que l'optimalité en terme de poussées, nous pouvons considérer les déplacements du joueur comme gratuits. Nous pouvons dès lors considérer que deux situations de jeu qui ne diffèrent que par la position du joueur sont équivalentes s'il existe un chemin pour le joueur[2] entre ces deux positions. L'information à prendre en compte n'est donc pas tant la position réelle du joueur que l'ensemble des positions qui lui sont accessibles. Il est dès lors commode de pouvoir caractériser cet ensemble par une position particulière. Nous définirons cette position que nous appellerons la *position normalisée du joueur* comme étant la position d'indice le plus faible appartenant à l'ensemble des positions accessibles par le joueur, *i.e.* la première position rencontrée en parcourant le tableau de jeu de gauche à droite et de haut en bas (cf. figure 3.18).

Soit S_1 et S_2, deux situations de jeu dans lesquelles les pierres occupent les mêmes positions. La position normalisée permet bien de déterminer s'il existe un chemin entre les deux positions du joueur. En effet, une disposition des pierres donnée divise le tableau de jeu en différentes zones d'accessibilité[3] qui sont chacune définie par une position normalisée. Or, si les positions normalisées du joueur des situations S_1 et S_2 sont les mêmes, c'est que les joueurs se trouvent dans la même zone et donc qu'il existe un chemin entre ces deux positions.

Finalement, une situation de jeu sera donc caractérisée dans notre programme par (1) la position normalisée du joueur et (2) la position des pierres. Soit un problème dans lequel n

[2]Dans ce travail, nous définirons un *chemin pour le joueur* comme étant la possibilité d'un déplacement du joueur entre deux positions sans que ce déplacement n'implique la modification de la position d'une pierre.

[3]Deux positions libres, *i.e.* qui ne sont occupées ni par un mur, ni par une pierre, sont dans la même *zone d'accessibilité* ssi il existe un chemin pour le joueur entre ces deux positions.

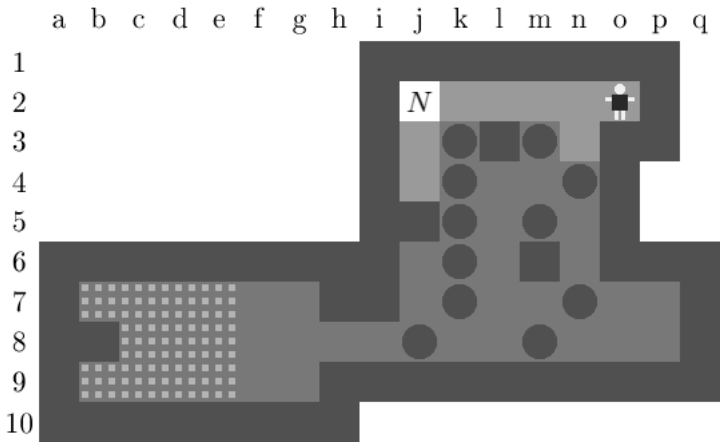

Figure 3.18 – Position normalisée du joueur (N).

positions du tableau de jeu sont susceptibles d'être occupées par une pierre, la position des pierres de ce problème sera représentée par une chaîne de n bits dont le $i^{\text{ème}}$ bit vaudra 1 si la $i^{\text{ème}}$ position accessible est occupée par une pierre. La valeur numérique unique associée à une situation de jeu sera dès lors égale à la valeur obtenue en concaténant la représentation binaire de la position normalisée du joueur avec la chaîne de bits représentant la position des pierres.

Collisions dans la table

Utiliser telle quelle la valeur numérique obtenue comme indice dans la table ne serait cependant pas réaliste. En effet, cela supposerait l'utilisation d'une table de taille bien trop importante. Une solution à ce problème est d'utiliser comme indice dans la table non pas la valeur numérique générée mais cette valeur modulo la taille de la table, cette dernière pouvant dès lors être fixée arbitrairement.

Cette solution n'est cependant pas idéale puisqu'elle introduit un problème de collision dans la table. En effet, si le nombre de situations de jeu potentiellement rencontrées est supérieur au nombre d'entrées disponibles dans la table, la correspondance bijective qui existait entre une situation et une entrée est perdue. Plusieurs situations de jeu différentes seront dès lors susceptibles de correspondre à un même indice de la table.

Afin de gérer ces collisions, chaque entrée de la table ne contiendra plus seulement la valeur du statut d'une situation de jeu mais la liste des situations de jeu qui ont été ajoutées dans la table et dont la valeur numérique unique modulo la taille de la table est égale à l'indice de cette entrée.

Cette modification de la structure de la table aura peu de conséquences sur l'opération d'ajout dans la table. Elle consistera simplement à ajouter la nouvelle situation de jeu en tête de la liste se trouvant à l'entrée correspondante. A contrario, l'opération de lecture dans la table se compliquera. Elle ne consistera plus à simplement consulter l'entrée associée avec

la situation de jeu recherchée mais bien à parcourir la liste se trouvant à cette entrée. Soit n le nombre de situations de jeu présentes dans cette liste, l'opération de lecture ne sera donc plus effectuée en un temps $O(1)$ mais en un temps $O(n)$.

L'art de la conception d'une table de transposition consiste donc à trouver une fonction de hashage et une taille pour la table qui répartissent de la façon la plus équitable possible les situations de jeu entre les entrées disponibles. Ce travail ne pousse pas plus avant cet aspect du problème et nous avons appliqué le principe élémentaire de conception qui consiste à choisir pour la taille de la table un nombre premier.

Conséquences de l'élimination des clones

L'utilisation d'une table de transposition pour identifier et éliminer les clones rencontrés peut cependant avoir des conséquences négatives sur le comportement de certains algorithmes de recherche. La figure 3.19 nous montre le cas d'un arbre de recherche dans lequel l'élimination des clones empêche l'algorithme depth-limited search d'atteindre un noeud solution qui se trouve pourtant à une profondeur inférieure à la profondeur maximale d_{max}.

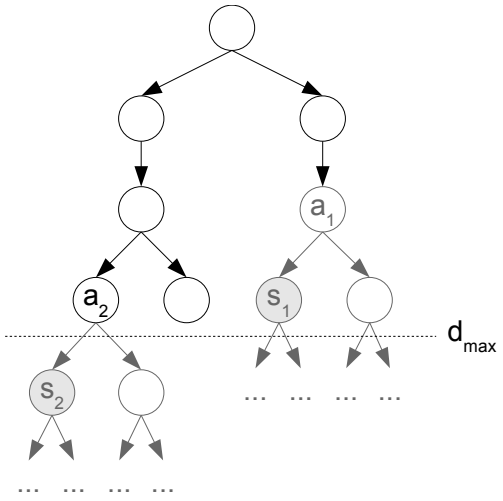

Figure 3.19 – Conséquences de l'élimination des clones.

Le noeud a_1 de cet arbre est la racine d'un sous-arbre qui contient le noeud solution s_1 situé à une profondeur inférieure à la profondeur maximale. Le noeud a_2 est un clone de a_1 situé plus bas dans l'arbre de telle sorte que le noeud solution s_2 appartenant au sous-arbre dont il est la racine est quant à lui inaccessible. De plus, le noeud a_2 se trouve plus à gauche dans l'arbre et sera donc visité avant le noeud a_1 lors d'un parcours en profondeur d'abord. A l'instant où le noeud a_1 sera atteint, le noeud a_2 aura donc déjà été ajouté dans la table. Par conséquent, le noeud a_1 sera identifié comme un clone et il sera ignoré. Le noeud solution s_1 ne sera ainsi jamais atteint.

Comme nous l'avons vu dans la première partie de ce chapitre, le fonctionnement de l'algorithme iterative-deepening repose sur l'utilisation d'une recherche limitée en profondeur, dont la profondeur maximale est augmentée itérativement. On peut dès lors craindre

que le problème identifié précédemment ait également un impact sur le fonctionnement de l'iterative-deepening. Cette crainte est justifiée. Et bien que les conséquences de la suppression des clones ne soient pas aussi dramatiques puisqu'un noeud solution pourra ici être atteint, elles ne sont pas négligeables car elles feront perdre à l'iterative-deepening sa propriété d'optimalité.

En effet, la troisième itération du parcours de l'arbre utilisé dans le cas précédent devrait normalement permettre d'atteindre le noeud solution s_1. Cependant, à partir de cette troisième itération, le parcours en profondeur d'abord passera préalablement par le noeud a_2 et le noeud a_1 ne sera dès lors plus visité. Le noeud solution s_2 sera finalement atteint à la cinquième itération mais la solution renvoyée par l'algorithme ne correspondra pas à la solution optimale du problème.

Exploitation de l'itération avec l'iterative-deepening

Ces considérations nous ont conduit à développer une version plus efficace de l'iterative-deepening. L'idée consiste à stocker dans une table la profondeur minimale à laquelle un noeud a été rencontré et de conserver cette table entre toutes les itérations de l'algorithme. Cette variante règle le problème de l'optimalité de l'algorithme. En effet, à la quatrième itération, le noeud a_2 ne sera plus développé aux dépends du noeud a_1 puisque la table permettra à l'algorithme de savoir que ce noeud a_2 est le clone d'un noeud situé à une profondeur inférieure (cf. figure 3.20).

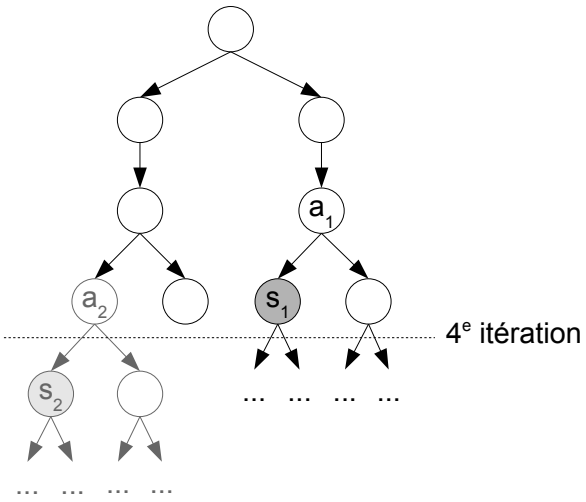

Figure 3.20 – Comportement de notre variante de l'iterative-deepening.

Cependant un noeud peut également avoir des clones situés à la même profondeur. Ne développer un noeud que s'il se trouve à la profondeur minimale stockée dans la table ne suffit donc pas pour éliminer toutes les possibilités de développer plusieurs fois un même sous-arbre. Nous devons en plus mettre en place un mécanisme qui nous permette de savoir si le noeud situé à la profondeur minimale a déjà ou non été visité durant l'itération courante.

Un moyen simple de marquer un noeud de la table est de changer le signe de l'information

de profondeur qui lui est associé. Ainsi, en associant à chaque itération un signe positif ou négatif et en faisant varier ce signe à chaque itération, on pourra déterminer avec certitude si un noeud donné a déjà ou non été visité au cours de l'itération courante.

Soit une itération négative dans laquelle un noeud c de profondeur d_1 est rencontré, si aucune entrée de la table ne correspond au noeud c, on est certain que ce noeud ne possède aucun clone à une profondeur inférieure. En effet, tous les noeuds de l'arbre situés à une profondeur inférieure ont été ajoutés dans la table à l'itération précédente. Le noeud c est donc ajouté dans la liste *open* et une entrée associée à la $-d_1$ est ajoutée dans la table. Si, par contre, le noeud c correspond à une entrée de la table associée à une valeur d_2, trois cas peuvent se présenter :

1. Si $d_1 > |d_2|$, le noeud c est le clone d'un noeud situé à une profondeur inférieure et est donc ignoré.
2. Si $d_1 = |d_2|$ et que $d_2 < 0$, *i.e.* que d_2 a le même signe que l'itération courante, le noeud c est le clone d'un noeud situé à la même profondeur et qui a déjà été visité au cours de l'itération. Le noeud c est dès lors ignoré.
3. Si $d_1 = |d_2|$ et que $d_2 > 0$, *i.e.* que d_2 a le signe inverse de celui de l'itération courante, on est certain que le noeud c est visité pour la première fois au cours de l'itération et il est donc ajouté dans la liste *open*.

On remarquera que le cas où $d_1 < |d_2|$ n'a pas été envisagé. En effet, le parcours effectué par l'iterative-deepening garantit que le clone situé à une profondeur inférieure sera toujours visité en premier lieu et donc ajouté en premier dans la table.

3.4.4 Situations de deadlock n'impliquant qu'une seule pierre.

La préoccupation principale d'un joueur de Sokoban est d'éviter les situations de deadlock. Or, certaines d'entre elles s'avèrent aisément identifiables par une analyse préalable du tableau de jeu. Il s'agit des situations de deadlock n'impliquant qu'une seule pierre. Une telle situation se présente si une pierre de la situation de jeu se trouve en position de deadlock, *i.e.* dans une position à partir de laquelle il n'existe aucune solution pour l'amener vers un goal (cf. figure 3.21).

Afin d'exploiter cette information, notre programme construit un vecteur *deadlock-vector* tel que *deadlock-vector*$[i]$ soit vrai ssi la position i est une position qui ne peut pas être occupée par une pierre car, dans le cas contraire, cette pierre se trouverait en position de deadlock. Ce vecteur est ensuite utilisé par l'algorithme chargé de déterminer les poussées possibles à partir d'une situation de jeu donnée afin d'éliminer du résultat les poussées qui amèneraient une pierre dans une position de deadlock. La construction du *deadlock-vector* nous permet ainsi de réduire le nombre de successeurs d'une position de jeu et donc de réduire le facteur de branchement de l'arbre de recherche.

La méthode utilisée pour construire le vecteur *deadlock-vector* procède en deux étapes. Les positions de deadlock immédiatement identifiables sont les positions qui ne sont pas

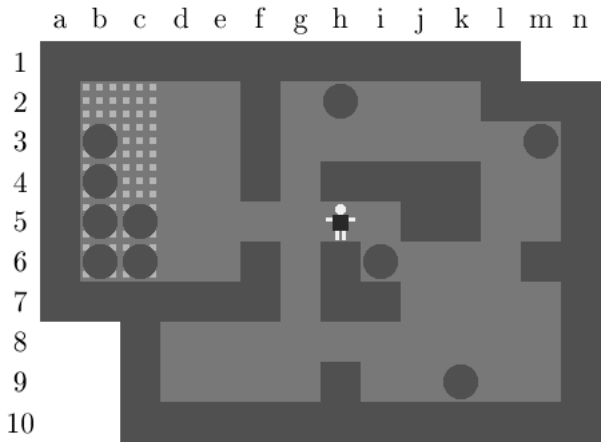

Figure 3.21 – Exemples de pierres se trouvant en position de deadlock.

occupées par un goal et qui sont entourées par au moins deux murs se trouvant dans des directions perpendiculaires. Une pierre se trouvant dans une telle position est en effet définitivement immobilisée par ces deux murs contigus (cf. figure 3.22). La première étape de notre méthode est donc de parcourir une première fois le tableau de jeu et de stocker ces positions dans le vecteur *deadlock-vector*.

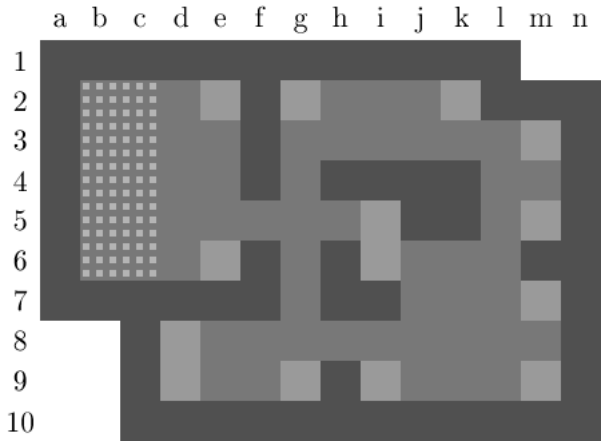

Figure 3.22 – Identification des positions de deadlock entourées par au moins deux murs.

La deuxième étape de la méthode est d'identifier des lignes de positions de deadlock contiguës situées entre deux positions identifiées lors de la première étape. Une pierre se trouvant dans une telle ligne sera en effet incapable d'en sortir et donc d'être amenée vers un goal (cf. figure 3.23).

Afin d'identifier ces lignes, on prolonge chaque position identifiée lors de la première étape vers la gauche et vers le haut. Si l'un de ces prolongements atteint une position stockée dans le *deadlock-vector*, sans passer par une position occupée par un goal ou une position d'échappement, une ligne est identifiée et les positions qu'elle contient sont ajoutées dans le *deadlock-vector*. Nous entendons par *position d'échappement*, une position dont les deux positions contiguës situées dans les directions perpendiculaires à la ligne ne sont pas

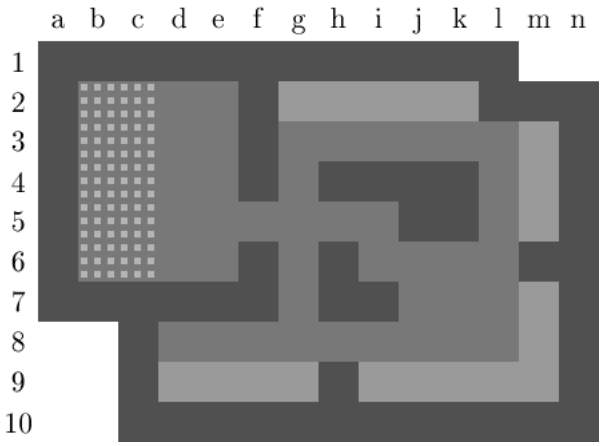

Figure 3.23 – Identification des lignes de positions de deadlock contiguës.

occupées par des murs (cf. figure 3.24).

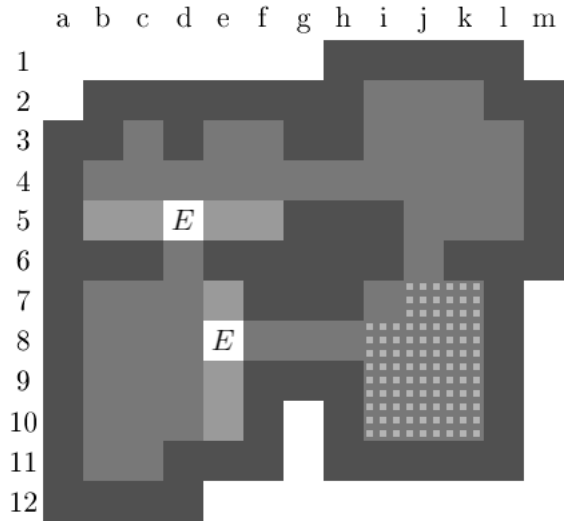

Figure 3.24 – Positions d'échappements (E) permettant à une pierre de sortir d'une ligne de positions de deadlock contiguës.

Une seconde exécution de la deuxième étape est de plus effectuée afin de prendre en compte certaines situations plus particulières dans lesquelles une simple exécution de l'étape ne permet pas d'identifier toutes les lignes de positions de deadlock. Ce sera notamment le cas d'une ligne dont l'une des extrémités se trouve dans une autre ligne et qui ne pourra dès lors être identifiée qu'après que cette dernière ligne ait été elle-même identifiée lors d'une première exécution de la deuxième étape.

3.4.5 Situations de deadlock induites par la dernière poussée

La situation de départ d'un problème de Sokoban n'est jamais une situation de deadlock. Un deadlock ne peut donc apparaître que lors d'une poussée mal choisie. Il est donc intéressant de pouvoir analyser les conséquences de la dernière poussée sur la situation de jeu afin de s'assurer que celle-ci n'induit pas une situation de deadlock.

La méthode que nous avons implémentée dans notre programme est encore limitée et ne permet d'éviter qu'un seul type de deadlock. Elle s'avère néanmoins intéressante puisqu'il s'agit sans doute du type de deadlock le plus souvent rencontré dans la résolution d'un problème : les deadlocks « en carré ». Dans ce type de deadlock, quatre pierres ou murs occupent des positions contiguës de telle sorte qu'ils forment un carré (cf. figure 3.25).

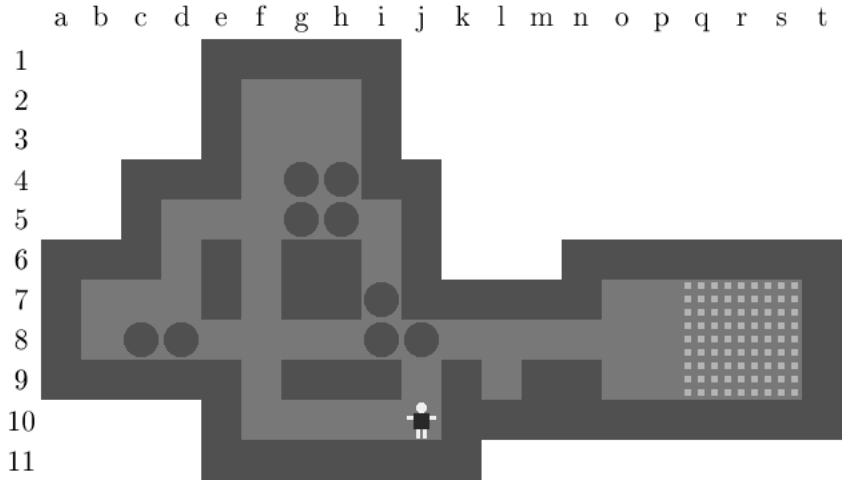

Figure 3.25 – Exemples de deadlocks « en carré ».

Une pierre dans une position donnée peut potentiellement être impliquée dans quatre deadlocks « en carré » (cf. figure 3.26). La méthode d'identification consiste donc à tester ces quatre carrés de quatre positions pour chaque position d'arrivée d'une poussée possible. Si l'un de ces carrés est complètement occupé par des pierres ou des murs, la poussée est ignorée lors de la construction de la liste des successeurs d'un noeud.

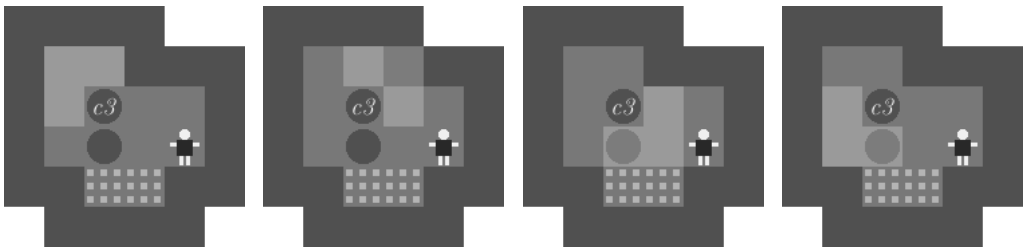

Figure 3.26 – Les quatre carrés occupés par la pierre *c3*.

Une technique intéressante utilisée par le programme *Rolling Stone* est de précalculer et de stocker dans une base de données toutes les configurations de deadlock possibles dans une zone bien définie. La base de données utilisée par *Rolling Stone* contient le statut, deadlock

ou non, de toutes les configurations possibles pour des zones de 5×4 positions. Cette base de données contient approximativement 22 millions d'entrées et a demandé plusieurs semaines de temps de calcul pour être générée.

Bien qu'elle présente un intérêt certain, l'implémentation d'une telle solution n'est pas triviale et demande des moyens importants. Nous avons dès lors pris la décision de ne pas chercher à l'implémenter dans un premier temps. Nous verrons néanmoins que la méthode que nous présenterons dans la troisième partie de ce travail comblera en partie son absence en stockant dans une base de données les configurations de deadlock rencontrées au fur et à mesure de la résolution d'un problème. Cette base de données, qui sera dès lors locale au problème courant, nous permettra ainsi d'exclure de la liste des successeurs d'un noeud les configurations de jeu contenant ces configurations de deadlock.

3.4.6 Résolution d'un problème de Sokoban

Comme nous l'avons vu précédemment, la résolution d'un problème particulier à l'aide d'un algorithme de recherche se fait en passant à cet algorithme trois arguments : (1) *start-node*, le noeud représentant la situation initiale du problème, (2) *goal-node*?, un prédicat permettant d'identifier le noeud solution recherché et (3) $find\text{-}successors$, une fonction renvoyant la liste des successeurs d'un noeud. La définition de ces trois éléments constituera donc la dernière étape de la réalisation d'un programme capable de résoudre nos premiers problèmes de Sokoban.

La définition du noeud *start-node* représentant la situation initiale du problème est immédiate. Elle nous permettra cependant d'introduire la représentation que nous avons choisie pour les noeuds de l'arbre de recherche.

Deux informations sont associées à un noeud de l'arbre : (1) la situation de jeu qu'il représente et (2) la séquence de poussées qui a permis d'atteindre cette situation à partir de la situation de jeu initiale du problème. Une situation de jeu est définie par la position du joueur et la position des différentes pierres. Nous avons déjà discuté de la représentation de ces deux éléments dans ce chapitre : la position du joueur est stockée dans une variable *man-pos* et la position des pierres est représentée par une liste de positions *stone-list*. Une situation de jeu sera dès lors définie par une paire $(man\text{-}pos, stone\text{-}list)$.

Une poussée a pour conséquence de déplacer une pierre de sa position initiale vers une position adjacente. Dans le Sokoban, chaque position ne peut être occupée que par une seule pierre. La connaissance de la position initiale d'une poussée permet donc d'identifier de façon univoque la pierre déplacée. Une poussée sera dès lors définie par le couple formé par ces deux positions. Soit $from\text{-}coord$ la position de départ d'une pierre et *to-coord* sa position d'arrivée, une poussée sera représentée par une paire $(from\text{-}coord, to\text{-}coord)$. Enfin, une séquence de poussées sera simplement définie par une liste de poussées *push-list*.

Nous sommes maintenant en mesure de représenter un noeud de l'arbre de recherche par une paire $((man\text{-}pos, stone\text{-}list), push\text{-}list)$. Dans le cas du noeud représentant la situation initiale du problème, *push-list* est la liste vide et ce noeud pourra dès lors être construit

par l'expression :

```
(cons (cons man-pos stone-list) '())
```

La définition de la fonction *goal-node?* est également immédiate. En effet, afin de déterminer si le noeud qui lui est passé en argument est un noeud solution, la fonction devra parcourir la liste des pierres contenues dans la situation de jeu représentée par le noeud et vérifier que chaque pierre occupe une position de goal. On a ainsi :

```
(define goal-node?
  (lambda (node)
    (let loop ((stone-list (node-stone-list node)))
      (or (null? stone-list)
          (and (vector-ref goal-vector (car stone-list))
               (loop (cdr stone-list)))))))
```

Pour un problème comportant n pierres et en utilisant le vecteur *goal-vector* tel que nous l'avons défini précédemment, cette opération se fera en un temps $O(n)$.

Enfin, la définition de la fonction *find-successors* utilise deux opérations que nous avons décrites précédemment : la recherche des poussées possibles et la recherche des deadlocks induits par la dernière poussée. La liste des successeurs du noeud passé en argument sera obtenue en parcourant la liste des poussées possibles et en construisant un noeud successeur pour chaque poussée qui ne provoque pas de deadlock identifiable par notre méthode. On arrive ainsi à la définition :

```
(define find-successors
  (lambda (node)
    (let loop ((next-push-list (find-stone-pushes ...))
               (successor-list '()))
      (if (null? next-push-list)
          successor-list
          (if (lead-to-deadlock? ...)
              (loop (cdr next-push-list)
                    successor-list)
              (loop (cdr next-push-list)
                    (cons (make-successor-node ...) successor-list)))))))
```

Soit un noeud de départ a_1, la construction d'un noeud successeur à partir d'une poussée est une opération qui consiste à « appliquer » cette poussée au noeud a_1. Nous définissons *man-pos*$_1$, *stone-list*$_1$ et *push-list*$_1$ comme étant respectivement la position du joueur, la liste des positions des pierres et la liste de poussées associées au noeud a_1. Soit une poussée qui amène une pierre d'une position p_1 à une position p_2 adjacente, le noeud a_2 résultant de l'application de la poussée au noeud a_1 est construit à partir des éléments *man-pos*$_2$, *stone-list*$_2$ et *push-list*$_2$ ci-après définis : *man-pos*$_2$ est la position p_1 ; *stone-list*$_2$ est la liste

stone-list$_1$ dans laquelle la position p_1 à été remplacée par la position p_2 ; enfin, *push-list*$_2$ est construite en ajoutant la poussée $(p1, p2)$ en tête de la liste *push-list*$_1$.

Une fois ces trois éléments définis, nous pouvons lancer la résolution d'un problème de Sokoban en les passant comme arguments à l'algorithme de recherche choisi :

```
(iterative-deepening start-node goal-node? find-successors)
```

Si la recherche aboutit, le noeud solution trouvé est renvoyé. Ce noeud contient la séquence de poussées qui a été effectuée pour atteindre la situation de jeu finale qu'il représente. A partir de cette séquence, il est facile de reconstituer la séquence de déplacements du joueur initialement recherchée en transformant chaque poussée en une séquence de déplacements.

Soit *man-pos* la position courante du joueur et une poussée qui fait passer une pierre de la position p_1 à la position p_2, on calcule d'abord la position p_3 qui est la position que le joueur devra occuper afin de pouvoir pousser la pierre vers la position p_2 (cf. figure 3.27). Une recherche de chemin classique permet ensuite d'obtenir la séquence de déplacements permettant au joueur de rejoindre cette position p_3. La séquence de déplacements correspondant à la poussée est alors obtenue en ajoutant à la séquence précédente le déplacement supplémentaire réalisé par le joueur pour amener la pierre dans la position p_2. Enfin, la position p_1 devient la position courante du joueur qui sera utilisée dans la transformation de la poussée suivante en une séquence de déplacements. La solution voulue du problème est ainsi reconstituée en concaténant les séquences obtenues.

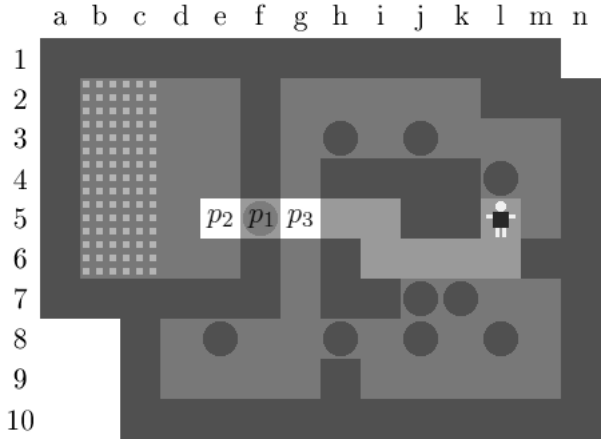

Figure 3.27 – Transformation d'une poussée en une séquence de déplacements du joueur.

3.4.7 Limites par rapport au benchmark de 90 problèmes

Ces algorithmes classiques sont cependant insuffisants pour résoudre les problèmes non triviaux de notre benchmark. En effet, l'obtention d'une solution acceptable (raisonnablement proche de la solution optimale) pour le premier et le plus simple des problèmes a

nécessité plus de 8 heures de temps de calcul. Nous verrons que la méthode que nous développerons dans les chapitres suivants nous permettra d'obtenir, sur la même machine, le même résultat en moins d'1 seconde.

Le lecteur attentif aura remarqué qu'aucun algorithme de recherche informée n'a été implémenté ni expérimenté dans cette première partie de notre travail qui avait pour principal objectif de démontrer les limites des algorithmes classiques dans le domaine du Sokoban. Ce choix est motivé par différentes raisons. D'une part, comme expliqué dans [3], une fonction d'heuristique est difficile à concevoir et coûteuse à calculer dans le cas du Sokoban. En effet, soit n le nombre de pierres du problème, l'heuristique utilisée par le solveur *Rolling Stone* a une complexité temporelle de $O(n^3)$. D'autre part, [3] démontre également que l'utilisation d'un algorithme de recherche informée ne permet de résoudre aucun des problèmes de notre benchmark sans l'ajout d'améliorations spécifiques. Finalement, l'approche basée sur une fonction d'heuristique a déjà été largement étudiée par les concepteurs de *Rolling Stone*. Le développement de ce solveur, qui fait office de référence actuelle, a pris plusieurs années et a donné lieu à de nombreuses publications. Il semble cependant que les performances de ce programme, et donc de cette approche, aient atteint un certain palier. En effet, *Rolling Stone* ne semble plus être actuellement l'objet de recherches et les récentes publications sur le Sokoban en provenance de l'Université d'Alberta[4] exploraient des méthodes de résolutions différentes. Dans [4], par exemple, on peut lire :

> *Heuristic search has led to impressive performance in games such as Chess and Checkers. However, for some two-player games like Go and Shogi, or puzzles like Sokoban, approaches based on heuristic search seem to be of limited value.*

Le principal facteur qui a mis en échec les algorithmes classiques que nous avons expérimentés est la taille gigantesque de l'arbre de recherche rencontré lors de la résolution d'un problème de Sokoban non trivial. Comme nous l'avons déjà évoqué, la taille de l'espace d'états d'un problème de Sokoban de taille 20×20 a été évaluée à 10^{98}. La taille d'un arbre de recherche est la conséquence de deux facteurs : sa profondeur et son facteur de branchement. Puisque nous avons décidé de mettre dans un premier temps de côté les méthodes à base d'heuristique, agir sur ces deux facteurs sera pour nous le seul moyen de surmonter la difficulté induite par la taille de l'arbre. Afin d'atteindre cet objectif, nous nous sommes tourné vers une approche inspirée par des méthodes de planification et d'apprentissage. Cette approche originale sera l'objet des chapitres suivants.

[4] *Rolling Stone* a été conçu par une équipe de chercheurs issue de l'Université d'Alberta, au Canada.

Chapitre 4

Planifier

> ***Strategic objectives which a human player can see are often well over the brute froce search horizon.*[7]**

Le parcours systématique de tous les coups possibles est une approche qui convient au fonctionnement d'une machine mais qui est loin de correspondre à la démarche adoptée par un joueur humain confronté à un problème de Sokoban. Après avoir décidé de nous lancer dans des recherches sur le Sokoban, la première chose que nous avons faite a été de nous confronter nous-même à la résolution de quelques problèmes. Cette première approche avait pour but de nous faire prendre conscience de la méthode que nous utilisions « spontanément » pour solutionner un problème de Sokoban. Bien qu'une telle observation sur nos propres comportements ne soit pas aisée à réaliser, nous sommes tout de même arrivé à en extraire une ligne directrice : nous nous fixions des objectifs stratégiques. Ces objectifs, dont un exemple est donné dans la figure 4.1, étaient de différentes natures, mais le plus important était que nous nous les fixions avant de chercher la séquence de coups qui nous permettrait de les réaliser. Cette conclusion nous a amené à aborder les problèmes de Sokoban sous un nouvel angle, celui de la planification.

4.1 Planification hiérarchique

Dans l'introduction de ce travail, nous avons choisi de définir un système intelligent comme étant un système capable de réaliser des tâches complexes. La planification est une méthode communément employée pour organiser et simplifier la réalisation de telles tâches. Elle implique couramment de diviser une tâche réputée complexe en sous-tâches de moindre importance et d'ordonnancer ces sous-tâches entre elles. La séquence de sous-tâches ainsi produite peut être vue comme un plan permettant de guider la réalisation de la tâche complète. On peut également la voir comme une première solution au problème, faisant abstraction des détails de la réalisation de chaque sous-tâche. Cette division des tâches en sous-tâches implique une notion de hiérarchie entre les tâches et cette approche de la planification porte le nom de planification hiérarchique.

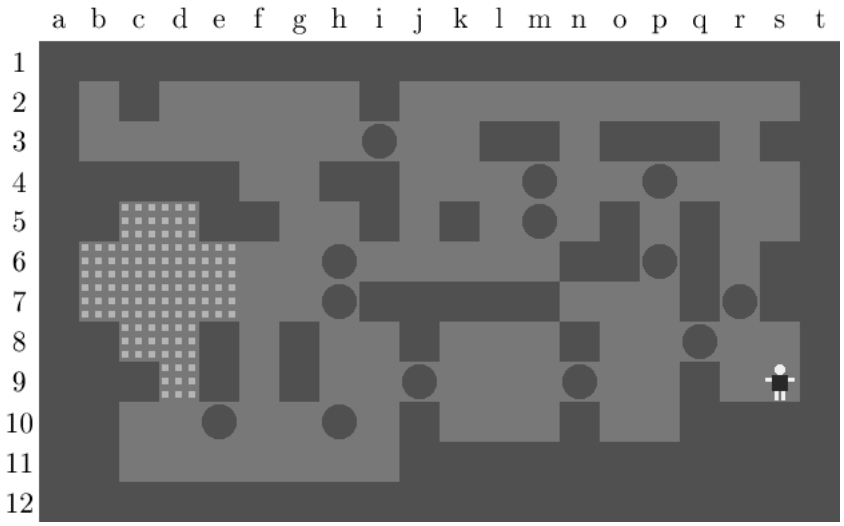

Figure 4.1 – Exemple d'objectif stratégique.

La pierre *e10* de ce problème peut être amenée directement dans le goal *d5*. Une stratégie intelligente sera dès lors de chercher une séquence de coups qui permette au joueur d'atteindre la position *f10* afin qu'il puisse pousser cette pierre vers sa position définitive.

Dans le cadre d'un problème de recherche, l'utilisation de la planification hiérarchique introduit un niveau supplémentaire de recherche impliquant des actions de plus haut niveau que les actions élémentaires (les coups dans le cadre d'un jeu) utilisées par un algorithme de recherche classique pour explorer l'espace d'états d'un problème. Dans cette approche, le problème est d'abord résolu en utilisant ces actions de haut niveau. Puis, la réalisation de chacune des actions de haut niveau est transformée en une suite d'actions élémentaires afin d'obtenir la solution voulue du problème (cf. figure 4.2).

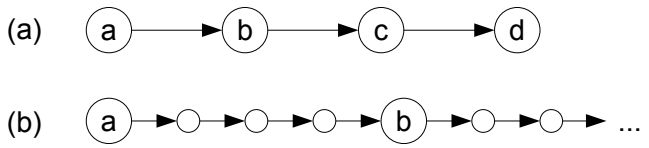

Figure 4.2 – Utilisation de la planification hiérarchique.

Le problème est d'abord résolu en terme d'actions de haut niveau (a). Chaque action de haut niveau est ensuite transformée en une suite d'actions de bas niveau afin d'obtenir la solution recherchée (b).

Nous avons déjà appliqué le principe de la planification hiérarchique dans ce travail en choisissant de définir un coup comme étant une poussée et non un déplacement du joueur. En effet, chaque poussée résulte d'un ou plusieurs déplacements du joueur et peut dès lors être vue comme une action de plus haut niveau qu'un déplacement. Résoudre un problème de Sokoban en terme de poussées consiste donc à résoudre le problème d'un point de vue de plus haut niveau que de le résoudre directement en terme de déplacements du joueur. Il est aisé de se convaincre que le choix du déplacement comme action élémentaire nous aurait conduit à parcourir un arbre de recherche dont la taille aurait été sensiblement plus importante. En effet, à titre d'exemple, si la résolution du problème 39 de notre benchmark

nécessite plus de 600 poussées, elle nécessite également plus de 1500 déplacements du joueur. Or, comme nous l'avons vu, la solution en terme de déplacements du joueur, qui est la solution recherchée, s'obtient très facilement à partir de la solution exprimée en poussées. Les bénéfices de l'utilisation de la planification hiérarchique sont donc dans ce cas évidents.

La résolution d'un problème de Sokoban en terme de poussées nous prive cependant de la possibilité d'obtenir une solution optimale en terme de déplacements du joueur. Cette restriction n'est pas essentielle dans le cadre de ce travail puisque nous avons fait le choix de ne considérer que l'optimalité en terme de poussées. Cependant, elle nous permet de mettre en lumière une conséquence plus générale de l'utilisation de la planification hiérarchique. En effet, lors de la construction de la solution finale, chaque transformation d'une action de haut niveau en une suite d'actions élémentaires est traitée comme un problème indépendant. Il est de ce fait difficile et souvent impossible d'obtenir une optimisation globale de la séquence complète d'actions élémentaires formant la solution du problème. L'utilisation de la planification hiérarchique n'est donc pas une solution miracle et, comme cela est souvent le cas dans la résolution de problèmes par des moyens informatiques, son efficacité est avant tout le résultat d'un compromis.

4.2 Etat de l'art

Des recherches visant à résoudre des problèmes de Sokoban par des méthodes de planification ont déjà été réalisées et ont donné lieu à une publication. Dans [4], les auteurs explorent les possibilités de l'utilisation du langage et de l'algorithme de planification *STRIPS* [14]. Ils y démontrent l'efficacité de l'utilisation d'une représentation abstraite d'un problème de Sokoban qui décompose le tableau de jeu en pièces interconnectées par des tunnels. Bien que l'approche soit intéressante, elle n'a pas encore obtenu de résultats réellement significatifs puisque le programme réalisé est actuellement capable de résoudre seulement 10 problèmes du benchmark.

Le point commun entre l'approche explorée dans [4] et l'approche que nous développerons dans ce chapitre est le concept de décomposition. En effet, de même que, comme nous l'avons vu, l'intelligence semble intimement liée à la complexité, la planification semble indissociable du concept de décomposition. Les deux approches différeront cependant dans l'objet de cette décomposition. Dans [4], le tableau de jeu est divisé en sous-zones de jeu et les mouvements des pierres à l'intérieur de ces sous-zones sont traités comme des sous-problèmes indépendants. Par ailleurs, les pierres seront amenées à passer d'une sous-zone à l'autre et ces dernières seront donc liées par une relation de voisinage. La décomposition utilisée peut dès lors être vue comme une décomposition spatiale du problème.

L'approche de la planification utilisée dans ce travail sera par contre de décomposer l'objectif poursuivi, *i.e.* le remplissage de tous les goals d'un problème de Sokoban, en une suite de sous-objectifs de moindre importance. Afin de résoudre le problème, il faudra dès lors résoudre les uns après les autres les sous-problèmes définis par cette suite de sous-objectifs. Les sous-problèmes seront par conséquent liés par une relation de précédence et on pourra dès lors parler d'une décomposition temporelle du problème global.

4.3 Planification dans le cas du Sokoban

Une question nous vient alors naturellement à l'esprit : comment décomposer un problème de Sokoban en une suite de sous-problèmes ? La décomposition que nous avons choisie repose sur l'observation suivante : la solution d'un problème de Sokoban peut se décomposer en une suite de séquences de poussées à la fin desquelles une pierre est amenée dans sa position définitive, *i.e.* le goal qu'elle occupera dans la situation de jeu finale. Chacune de ces séquences peut être elle-même divisée en deux phases : (1) une phase de dégagement, durant laquelle un certain nombre de poussées sont effectuées sur différentes pierres, et (2) une phase de rangement, durant laquelle une suite de poussées ne concernant plus qu'une seule pierre amènent celle-ci dans sa position définitive.

La phase de planification de notre méthode consistera à déterminer l'ordre dans lequel les goals devront être remplis. Nous appellerons la liste ordonnée de goals ainsi produite *l'ordonnancement des goals*. Chaque sous-problème consistera dès lors à trouver un moyen de remplir un goal donné, *i.e.* une séquence de poussées telle que nous l'avons décrite.

Il est intéressant de noter qu'un ordonnancement quelconque sera toujours la décomposition valide d'une solution. Notre décomposition n'exclut en effet pas que des goals soient remplis lors de la phase de dégagement d'une séquence. Soit un ensemble de n goals à remplir, dans le pire des cas, *i.e.* le cas où le premier goal de notre ordonnancement est en fait le dernier goal à devoir être rempli, la phase de dégagement de la séquence remplira les $(n-1)$ premiers goals à devoir être remplis et la phase de rangement amènera la dernière pierre dans le goal cible. La résolution du premier sous-problème aura ainsi conduit à résoudre la totalité du problème et les séquences de poussées correspondant à la résolution des $(n-1)$ sous-problèmes restants seront dès lors vides. Cependant, on se doute aisément qu'une telle situation fait perdre tout son intérêt à la méthode et qu'elle n'est utile que pour démontrer sa complétude théorique.

De fait, un ordonnancement n'aura de sens que s'il divise réellement le remplissage des n goals en n séquences non vides, *i.e.* s'il correspond à l'ordre dans lequel les goals seront réellement remplis. En effet, le gain principal de la méthode sera de réduire sensiblement la profondeur de l'arbre de recherche associé à la résolution de chaque sous-problème, en remplaçant la phase de rangement d'une séquence par un coup unique que nous appellerons une macro-poussée (cf. figure 4.3). Nous définirons dès lors une *macro-poussée* comme étant une suite de poussées d'une même pierre.

Notre programme sera donc adapté à la sous-classe des problèmes de Sokoban pour lesquels il est possible de déterminer à l'avance et sans ambiguïté l'ordre dans lequel les goals devront être remplis. Ce sera principalement le cas des problèmes ne contenant qu'une seule zone de goals et qu'une seule entrée réellement utilisable (cf. figure 4.4).

Une *zone de goals* est un ensemble de goals occupant des positions contiguës. Une *entrée* est intuitivement une position à partir de laquelle une pierre peut pénétrer dans une zone de goals. Nous verrons dans une prochaine section une définition plus formelle des entrées en décrivant la méthode utilisée pour les identifier.

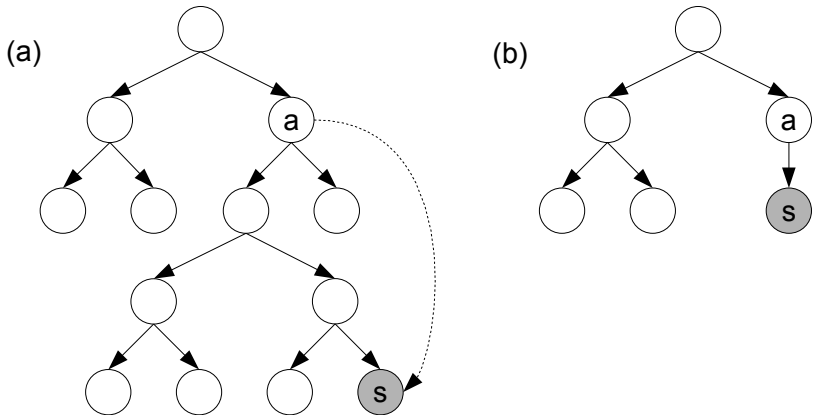

Figure 4.3 – Utilisation d'une macro-poussée.

Dans l'arbre (a), une macro-poussée permet d'atteindre en un seul coup le noeud solution recherché s à partir du noeud a. La profondeur de l'arbre de recherche est ainsi sensiblement réduite puisque l'arbre effectivement exploré est l'arbre (b).

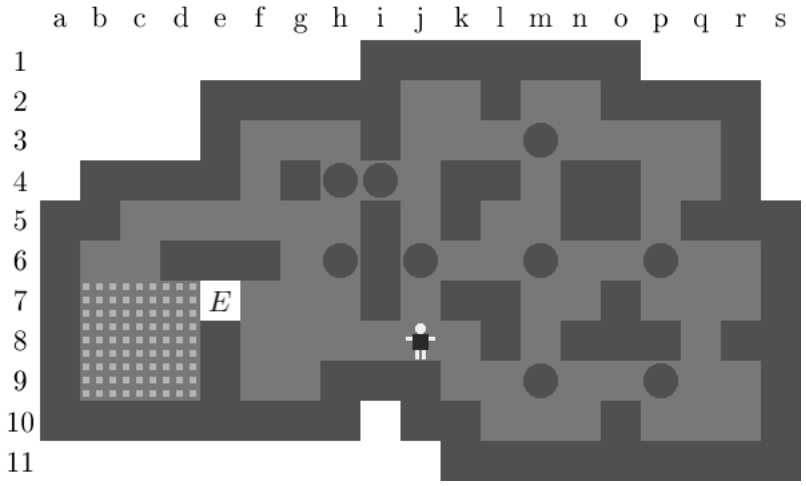

Figure 4.4 – L'unique entrée utilisable (E) du problème 43 de notre benchmark.

Notre approche rejoindra donc en un sens l'approche choisie par le programme *Talking Stone* qui était de proposer un protocole de résolution complet pour une sous-classe de problèmes de Sokoban. Afin de pouvoir résoudre d'autres types de problèmes, *Talking Stone* utilisait un algorithme de recherche classique pour trouver une séquence de poussées conduisant à une situation de jeu appartenant à la sous-classe. La méthode utilisée par *Talking Stone* était ainsi complète puisque le noeud solution recherché faisait lui-même partie de la sous-classe. Cependant, comme notre approche, elle ne se révélait efficace qu'avec des problèmes présentant certaines caractéristiques, *i.e.* des problèmes dans lesquels une situation de jeu appartenant à la sous-classe pouvait être atteinte en un nombre raisonnable de coups.

Il est par ailleurs intéressant de noter que la sous-classe des problèmes susceptibles d'être résolus efficacement par notre méthode englobe la sous-classe des problèmes résolus immédiatement par *Talking Stone*. En effet, ces problèmes devaient présenter trois caractéristiques pour appartenir à la sous-classe, la première étant que l'ordre dans lequel les goals devaient être remplis pouvait être déterminé à l'avance.

Protocole de résolution

Conceptuellement, notre méthode de résolution s'organisera autour de trois agents qui seront chacun responsable de la résolution d'un problème particulier (cf. figure 4.5) :

1. Le premier agent sera chargé de déterminer l'ordonnancement des goals en analysant la situation de jeu initiale.
2. Le deuxième agent aura pour tâche de remplir les uns après les autres les goals du problème dans l'ordre défini par l'ordonnancement.
3. Le troisième agent sera responsable de la résolution du sous-problème consistant à remplir le goal courant, *i.e.* le premier goal non rempli de l'ordonnancement, à partir d'une situation de jeu donnée.

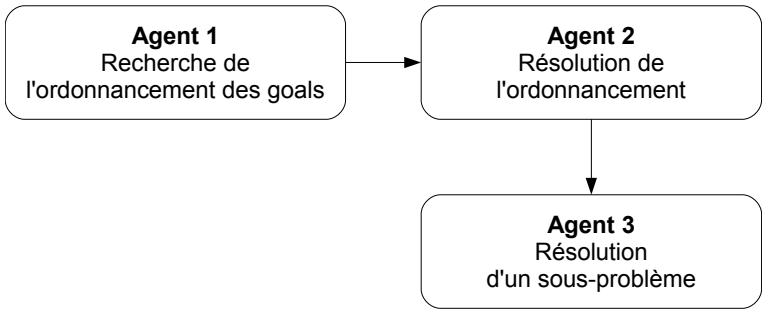

Figure 4.5 – Schéma général de notre protocole de résolution.

A partir de ces trois agents, la méthode de résolution utilisée par notre programme peut être décrite ainsi :

Pour commencer, le premier agent analyse la situation de jeu initiale et détermine l'ordre dans lequel les goals devront être remplis. Cet ordonnancement ainsi que la situation de

jeu initiale sont ensuite passés en arguments au deuxième agent. Ce dernier cherche alors un moyen de remplir le premier goal de l'ordonnancement. Pour ce faire, il fait appel au troisième agent et lui passe en arguments la situation initiale du problème ainsi que la position du goal cible. Le troisième agent lance alors une recherche qui a pour objectif de trouver une séquence de coups qui amène à une situation de jeu dans laquelle le goal cible est rempli. Cette recherche utilise un algorithme de recherche classique. Le noeud solution trouvé est retourné au deuxième agent qui utilise la situation de jeu contenue dans ce noeud comme point de départ à partir duquel un moyen de remplir le second goal de l'ordonnancement devra être trouvé. Si à une étape du processus, un moyen de remplir le $n^{\text{ième}}$ goal ne peut être trouvé, une autre solution permettant de remplir le $(n-1)^{\text{ième}}$ goal est recherchée.

Opérationnellement, le second agent peut être vu comme un agent de recherche effectuant un parcours en profondeur d'abord dans un arbre dont la racine est la situation initiale du problème et dont un noeud situé à une profondeur n correspond à une situation de jeu dans laquelle les $(n-1)$ premiers goals de l'ordonnancement sont remplis. Dans cet arbre, un noeud de profondeur n a comme successeurs les situations de jeu accessibles en un nombre quelconque de coups et dans lesquelles les n premiers goals de l'ordonnancement sont remplis. De plus, l'ordre d'apparition des successeurs d'un noeud correspondant au nombre de coups qui auront été nécessaires pour les atteindre, le successeur le plus à gauche et qui sera donc visité en premier sera le successeur ayant pu être atteint en un nombre minimum de coups.

4.4 Implémentation version 2

Dans la section précédente, nous nous sommes appuyé sur le paradigme multi-agent pour décrire le protocole de résolution mis en oeuvre dans notre programme. Cette description était essentiellement conceptuelle et avait pour objet d'en éclairer le fonctionnement. Les trois agents que nous avons introduits n'apparaissent donc pas explicitement dans l'implémentation qui sera décrite dans cette section. Nous tâcherons néanmoins de préciser le lien existant entre chaque fonction présentée et les agents de notre modèle conceptuel.

4.4.1 Recherche de l'ordonnancement des goals

La recherche de l'ordonnancement des goals correspond à la tâche que nous avons assignée au premier de nos trois agents. Cette opération, qui se situe en amont du processus de résolution, a une importance critique puisque l'efficacité de la méthode que nous avons introduite repose sur le choix d'un « bon ordonnancement », notion qu'il convient de préciser.

Un ordonnancement est une permutation de la liste des n goals d'un problème. La détermination d'un ordonnancement consiste donc à sélectionner une permutation particulière parmi les $n!$ permutations possibles de cette liste. L'ordonnancement que nous recherchons est un ordonnancement qui correspond à l'ordre réel dans lequel les goals du problème

pourront être remplis. Nous qualifierons d'effectif un tel ordonnancement. Formellement, un ordonnancement sera *effectif* pour un problème donné si ce problème admet au moins une solution dans laquelle les goals sont remplis dans l'ordre défini par cet ordonnancement. Ce critère d'effectivité est cependant difficilement exploitable. En effet, prouver qu'un ordonnancement est effectif implique de trouver une solution au problème et rend par conséquent l'ordonnancement des goals sans objet. Il s'avère dès lors indispensable d'identifier une caractéristique moins contraignante qui nous permette de générer un « bon ordonnancement » à partir d'une analyse opérationnellement raisonnable de la situation de jeu initiale d'un problème.

Une condition nécessaire pour qu'un ordonnancement soit effectif est que cet ordonnancement ne crée pas de situation de deadlock, *i.e.* qu'il ne conduise pas à une situation dans laquelle la position des pierres occupant les goals déjà remplis rend inaccessible au moins un des goals encore non remplis. Nous dirons d'un goal qu'il est *inaccessible* s'il n'a pas encore été rempli et qu'aucune pierre ne peut plus y être amenée (cf. figure 4.6).

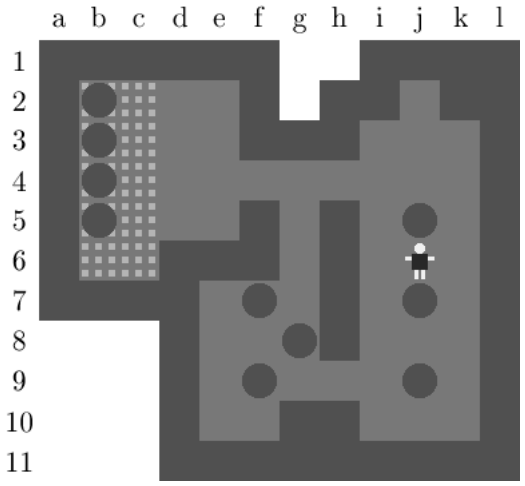

Figure 4.6 – Le goal *b6* est rendu inaccessible par les pierres occupant les goals déjà remplis.

Nous appellerons ordonnancement *consistant*[1] un ordonnancement qui garantit qu'aucun goal du problème ne sera jamais rendu inaccessible par la position des pierres occupant les goals déjà remplis. Cette dernière précision est importante puisque la consistance d'un ordonnancement n'exclut pas qu'un goal soit rendu inaccessible par la position des autres pierres du problème. La figure 4.7 illustre une telle situation. L'unique zone de goals de ce problème possède deux entrées, E_1 et E_2, et un seul goal encore non rempli. Ce goal n'est pas rendu inaccessible par la position des pierres occupant les goals déjà remplis puisqu'une pierre arrivant par l'entrée E_1 pourrait y être amenée. Cependant, aucune pierre ne peut être amenée vers cette entrée E_1 et la situation de jeu est dès lors une situation de deadlock.

On ne peut ainsi pas exclure qu'un ordonnancement, bien que consistant, soit tel que

[1] Le terme « consistant » fait ici référence à une non-contradiction. On peut, en effet, considérer qu'un ordonnancement qui rend inaccessible l'un des goals du problème contient en son sein une contradiction puisqu'il conduit irrémédiablement à une situation de deadlock et ce quelle que soit la configuration des pierres du problème.

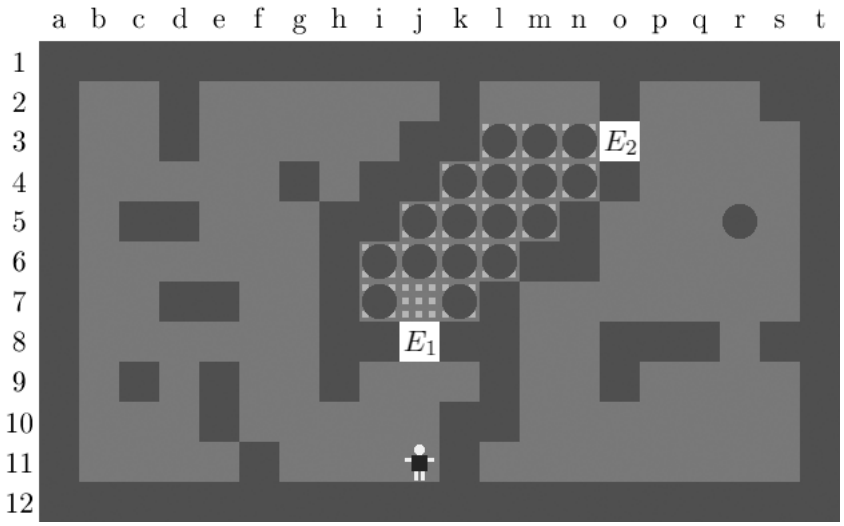

Figure 4.7 – Le goal $j7$ est inaccessible car aucune pierre ne peut être amenée vers l'entrée E_1.

toutes les séquences de poussées qui remplissent les goals dans l'ordre qu'il définit conduisent à une situation de deadlock semblable à celle de la figure 4.7. Un ordonnancement consistant n'est donc pas toujours un ordonnancement effectif. Soit $\mathcal{O}$ l'ensemble des $n!$ ordonnancements envisageables dans un problème de n goals, $\mathcal{O}_{consistants}$ l'ensemble des ordonnancements consistants et $\mathcal{O}_{effectifs}$ l'ensemble des ordonnancements effectifs, nous pouvons établir la relation

$$\mathcal{O} \subseteq \mathcal{O}_{consistants} \subseteq \mathcal{O}_{effectifs}$$

Prouver qu'un ordonnancement est consistant est une opération relativement légère. Elle consiste à remplir un à un les goals du problème dans l'ordre décrit par l'ordonnancement et à vérifier à chaque étape si tous les goals encore non remplis sont accessibles. La consistance sera ainsi le critère que nous utiliserons afin de générer un « bon ordonnancement ». Cette solution n'est néanmoins pas idéale puisqu'elle ne garantira pas que l'ordonnancement produit sera toujours effectif. La condition de consistance se révèlera cependant suffisante pour une sous-classe de problèmes de Sokoban présentant une caractéristique aisément identifiable.

Dans la section précédente, nous avions défini la sous-classe de problèmes de Sokoban susceptibles d'être résolus efficacement par notre méthode comme étant la sous-classe de problèmes pour lesquels un ordonnancement des goals pouvait être déterminé à l'avance et sans ambiguïté. Nous avions ensuite affirmé que ces problèmes étaient principalement ceux qui ne contenaient qu'une seule entrée (et donc, qu'une seule zone de goals). Cette dernière assertion peut maintenant être explicitée. En effet, dans une telle configuration, l'accessibilité des goals ne dépendra que de l'occupation de la zone de goals et non de la possibilité d'amener une pierre vers l'une ou l'autre entrée. On aura dès lors la relation

$$\mathcal{O}_{consistants} = \mathcal{O}_{effectifs}$$

Dans ce cas, l'ordonnancement des goals pourra donc être déterminé sans ambiguïté puisque l'ordonnancement produit sera toujours un ordonnancement effectif et non un ordonnancement choisi arbitrairement dans le sur-ensemble des ordonnancements consistants.

Premier algorithme d'ordonnancement des goals

Nous arrivons ainsi à un premier algorithme d'ordonnancement qui sera utilisé lorsque le problème analysé ne comporte qu'une seule entrée. Le principe de cet algorithme est de partir d'une situation de jeu dans laquelle la zone de goals est entièrement occupée par des pierres et de la vider progressivement en évacuant une par une ces pierres par l'unique entrée (cf. figure 4.8). L'ordonnancement des goals peut de cette façon être obtenu en inversant l'ordre dans lequel les goals auront pu être vidés.

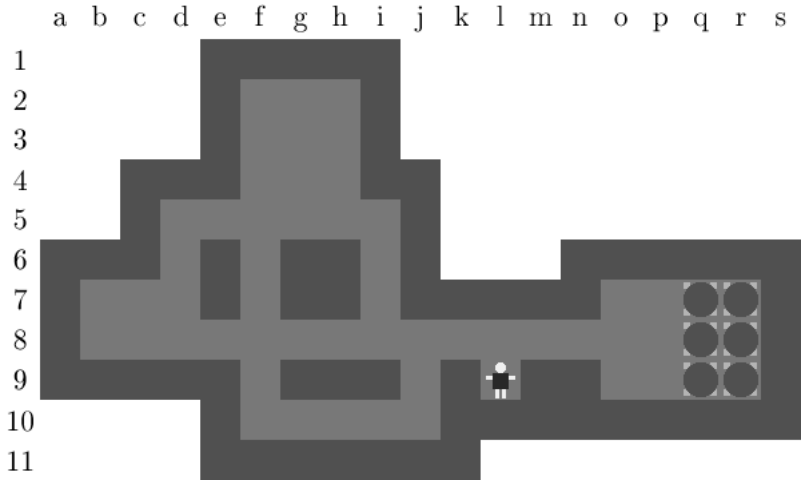

Figure 4.8 – Situation de jeu de départ du premier algorithme d'ordonnancement.

Pour trouver une pierre qui puisse être évacuée, l'algorithme parcourt la liste des pierres occupant la zone de goals et génère, pour chacune d'elles, une situation de jeu temporaire dans laquelle cette pierre a été retirée de la zone de goal et placée sur l'unique entrée. Il recherche ensuite une séquence de poussées permettant de ramener la pierre vers sa position initiale. Si une telle séquence existe, la pierre est retirée de la zone de goal (cf. figure 4.9).

Notre premier algorithme d'ordonnancement des goals peut ainsi être défini par :

```
(define find-one-entrance-scheduling
  (lambda (entrance-coord goal-list)
    (let loop ((remaining-goal-list goal-list)
               (scheduled-goal-list '()))
      (if (null? remaining-goal-list)
          scheduled-goal-list
          (let loop-goal ((iterated-goal-list remaining-goal-list))
            (and (pair? iterated-goal-list)
                 (let* ((goal-coord (car goal-list))
                        (stone-list (replace-first goal-coord entrance-coord
```

```
                                        remaining-goal-list)))
    (if (find-push-path stone-coord goal-coord stone-list ...)
        (loop (remove-first goal-coord remaining-goal-list)
              (cons goal-coord scheduled-goal-list))
        (loop-goal (cdr goal-list)))))))))))
```

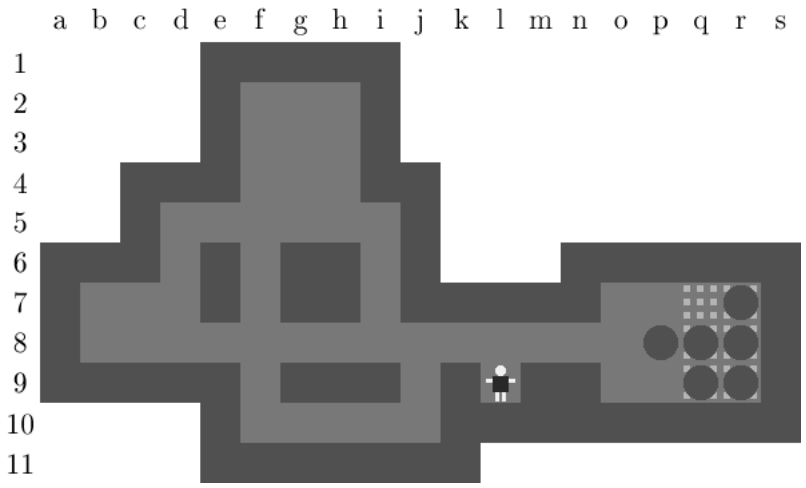

Figure 4.9 – Evacuation de la pierre occupant le goal *q7*.

La pierre a été retirée de son goal et placée sur l'unique entrée du problème (*p8*). La pierre pourra, dans cet exemple, être évacuée puisqu'il est possible de la ramener vers sa position d'origine.

Deuxième algorithme d'ordonnancement des goals

Si le problème à résoudre contient plusieurs entrées, la condition de consistance ne permettra pas de générer un ordonnancement qui soit avec certitude effectif. En effet, dans ce type de configuration, l'accessibilité des goals sera influencée par les possibilités d'amener ou non une pierre vers une entrée donnée. Or, ces possibilités évolueront en fonction de la modification de la position des pierres au fur et à mesure de la résolution du problème. Certaines entrées finiront ainsi par être bloquées par les pierres occupant les goals déjà remplis, tandis que d'autres entrées finiront par devenir inaccessibles si aucune pierre ne peut plus y être amenée. L'analyse préalable de la situation initiale du problème ne permet pas d'inférer ces évolutions qui sont à l'origine de la différence qui sépare un ordonnancement consistant d'un ordonnancement effectif. Par conséquent, l'ordonnancement consistant qu'elle produira ne sera pas toujours effectif.

Malgré cette incertitude sur l'effectivité de l'ordonnancement généré, notre programme s'avérera tout de même capable de résoudre un certain nombre de problèmes comportant plusieurs entrées. Ce sera notamment le cas des problèmes qui présentent une configuration telle qu'il s'avère possible de faire voyager une pierre d'une entrée à l'autre et dans lesquels il est par conséquent toujours possible d'amener une pierre vers une entrée donnée.

L'algorithme utilisé pour générer l'ordonnancement d'un problème à plusieurs entrées

repose sur un principe similaire à celui de l'algorithme précédent. En effet, il consistera également à vider étape par étape une zone de goals[2] initialement remplie. Il se différenciera cependant par le fait que chaque étape consistera à retirer de la zone de goals non plus une seule pierre mais l'ensemble des pierres susceptibles d'être évacuées.

A chaque étape de son exécution, l'algorithme commence par construire la liste des entrées accessibles. Une entrée sera accessible si les pierres occupant encore la zone de goals à cette étape n'empêchent pas une pierre d'y être amenée (cf. figure 4.10).

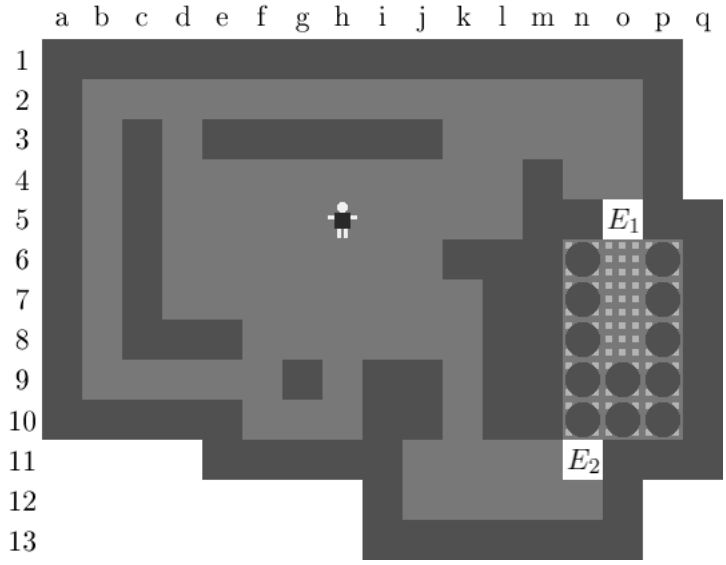

Figure 4.10 – L'entrée E_2 est rendue inaccessible par les pierres occupant la zone de goals.

L'algorithme parcourt ensuite la liste des entrées accessibles et établit la liste des pierres pouvant être évacuées par une de ces entrées (cf. figure 4.11). L'opération qui permet de déterminer si une pierre peut être évacuée par une entrée donnée est similaire à celle utilisée dans l'algorithme précédent. Finalement, toutes les pierres évacuables sont retirées de la zone de goals. L'ordonnancement des goals est ainsi généré en concaténant les listes de goals évacuables dans l'ordre inverse de celui dans lequel elles auront été obtenues.

Soit *find-first-goals* une fonction qui prend comme arguments la liste des entrées du problème et la liste des goals encore occupés par une pierre à l'étape courante, et qui renvoie les listes des goals qui peuvent être vidés à cette étape, notre deuxième algorithme d'ordonnancement est défini par la fonction :

```
(define find-multiple-entrances-scheduling
  (lambda (entrance-list goal-list)
    (let loop ((remaining-goal-list goal-list)
               (scheduled-goal-list '()))
      (if (null? remaining-goal-list)
```

[2]Il peut éventuellement exister plusieurs zones de goals dans un problème. Cette possibilité ne modifie en rien la méthode utilisée puisque cette dernière ne prendra en considération que la liste des entrées du problème et traitera donc ces différentes zones comme une zone unique.

```
scheduled-goal-list
(let* ((entrance-list (find-reachable-entrances ...))
       (first-goal-list (find-first-goals ...)))
  (if (null? first-goal-list)
      #f
      (loop (difference remaining-goal-list first-goal-list)
            (append first-goal-list scheduled-goal-list)))))))
```

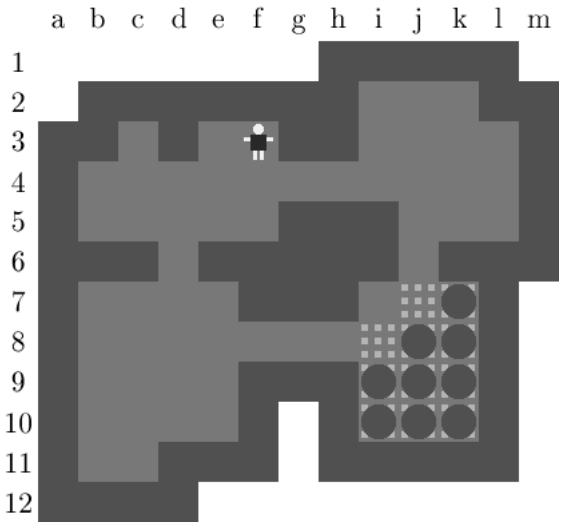

Figure 4.11 – Les pierres *k7*, *j8* et *i9* pourront être évacuées à cette étape de l'algorithme.

L'intérêt de cet algorithme d'ordonnancement est qu'il permet une certaine répartition de l'arrivée des pierres entre les différentes entrées d'un problème. En effet, un algorithme qui n'évacuerait qu'une seule pierre par étape serait susceptible d'évacuer toutes les pierres par la première entrée rencontrée. Un tel ordonnancement conduirait par conséquent à rechercher par la suite une solution improbable dans laquelle les goals du problème seraient remplis par une unique entrée.

Troisième algorithme d'ordonnancement des goals

L'ordonnancement produit par la méthode précédente ne sera pas toujours effectif et ne permettra donc pas de résoudre l'entièreté des problèmes de notre benchmark. Pour cette raison, un troisième algorithme d'ordonnancement a également été implémenté. Ce dernier produira un ordonnancement sensiblement différent qui permettra de résoudre quelques problèmes supplémentaires.

Cet algorithme repose sur une notion d'encerclement des goals et correspond à une version plus évoluée de la méthode d'ordonnancement proposée dans [1]. L'encerclement d'un goal se définit par rapport à l'occupation des quatre positions qui l'entourent. Il s'exprime sous la forme d'une valeur numérique que nous appellerons le taux d'encerclement et dont le calcul fait intervenir deux facteurs : (1) le nombre de positions adjacentes occupées par un mur ou par une pierre et (2) la limitation des possibilités de déplacement d'une pierre

occupant potentiellement ce goal.

Le taux d'encerclement d'un goal sera ainsi égal au nombre de positions adjacentes occupées par un mur ou une pierre. De plus, sa valeur sera augmentée d'une unité si une pierre occupant potentiellement ce goal est bloquée dans ses déplacements verticaux. De même, sa valeur sera augmentée d'une unité si une pierre occupant potentiellement ce goal est bloquée dans ses déplacements horizontaux (cf. figure 4.12).

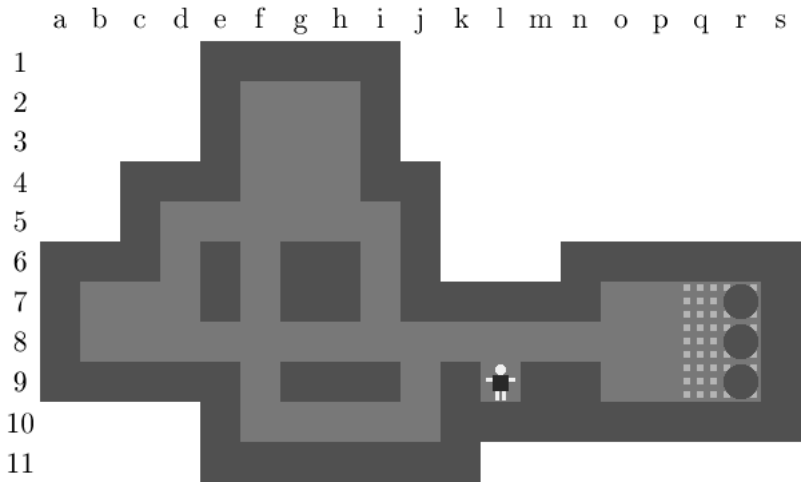

Figure 4.12 – Calcul du taux d'encerclement d'un goal.

Le goal $q7$ a une de ses positions adjacentes occupée par un mur (+1) et une autre par une pierre (+1). De plus, les déplacement d'une pierre à partir de sa position sont bloqués verticalement (+1) et horizontalement (+1). Le taux d'encerclement de ce goal vaut dès lors 4.

Le principe utilisé par les deux méthodes d'ordonnancement précédentes était de partir d'une zone de goals remplie et d'évacuer progressivement les pierres qui l'occupaient. Cette troisième méthode part au contraire d'une zone de goals vide et cherche un moyen de la remplir entièrement. A chaque étape de son exécution, l'algorithme tente d'ajouter une pierre dans la zone de goals de façon à ce que cet ajout ne rende pas inaccessibles certains goals encore non occupés. Si, à une étape, cet ajout s'avère impossible, le choix effectué lors de l'étape précédente est remis en cause et un autre goal doit alors être rempli. Le fonctionnement de cet algorithme correspond donc au fonctionnement classique d'un algorithme de recherche.

Afin que la sélection du prochain goal à remplir ne soit pas complètement arbitraire, la liste des goals non encore remplis est triée par ordre décroissant de taux d'encerclement. L'algorithme essayera ainsi de remplir en premier les goals qui possèdent le plus grand taux d'encerclement. On peut, en effet, escompter que le placement d'une pierre dans le goal le plus encerclé sera le moins susceptible de conduire à une situation de deadlock. Cette exploitation d'une connaissance a priori du domaine pour améliorer les performances d'une recherche en privilégiant certains choix par rapport à d'autres correspond à l'utilisation d'une heuristique.

Chaque fois que l'algorithme tente de placer une nouvelle pierre, il vérifie que ce choix ne

rend pas inaccessibles certains goals encore non remplis. Un goal sera ici considéré comme accessible s'il possède au moins deux positions contiguës libres dans une des quatre directions gauche, haute, droite ou basse, *i.e.* s'il existe deux positions susceptibles d'être occupées par le joueur et par une pierre de façon à ce que cette pierre puisse être poussée dans le goal (cf. figure 4.13).

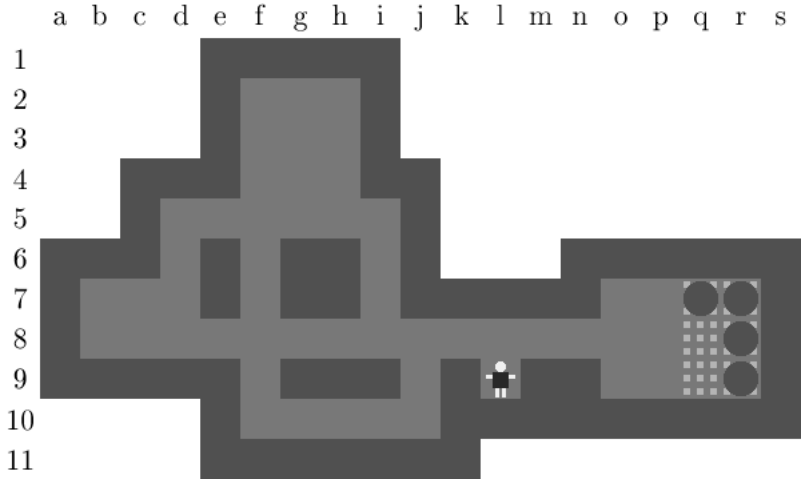

Figure 4.13 – Vérification de l'accessibilité d'un goal.
Le goal *q8* est accessible puisque les positions p8 et o8 permettent à une pierre d'y être poussée.

Recherche des entrées

L'algorithme utilisé pour identifier les entrées des zones de goals d'un problème procède en deux étapes. La première identifie les entrées potentielles en examinant les positions adjacentes des goals. Pour qu'une position adjacente à un goal soit une entrée potentielle, il faut (1) qu'un goal n'occupe pas cette position et (2) qu'une position soit accessible pour le joueur afin qu'une pierre occupant l'entrée potentielle puisse être poussée dans le goal adjacent (cf. figure 4.14).

La deuxième étape de l'algorithme parcourt la liste des entrées potentielles précédemment créées et construit une liste des entrées réelles. Une entrée potentielle est une entrée réelle si au moins une pierre de la situation de jeu initiale peut y être amenée sans passer par une des autres entrées potentielles (cf. figure 4.15).

Afin de déterminer si une pierre donnée peut être amenée vers une entrée potentielle, l'algorithme crée une situation de jeu temporaire dans laquelle toutes les autres pierres ont été retirées. Il cherche ensuite un moyen d'amener cette pierre à la position occupée par l'entrée potentielle courante sans passer par une position occupée par une autre entrée potentielle. Cette opération correspond à une recherche classique de chemin légèrement modifiée afin de tenir compte du fait que la pierre ne doit pas passer par certaines positions.

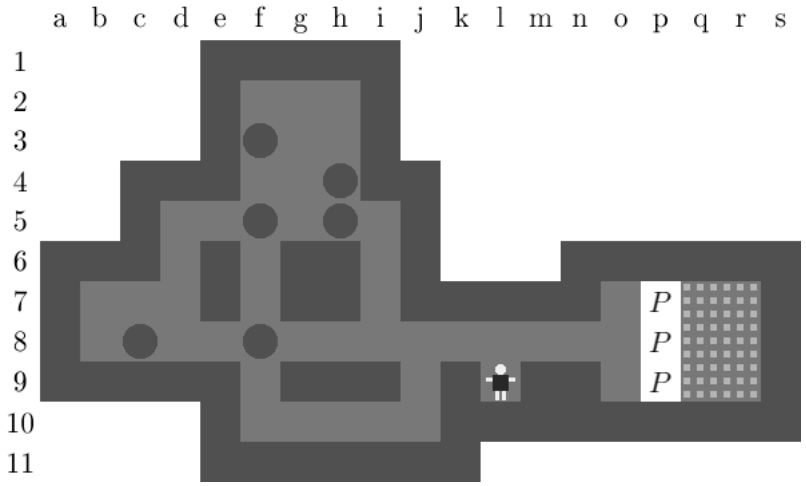

Figure 4.14 – Les entrées potentielles (P) du premier problème de notre benchmark.

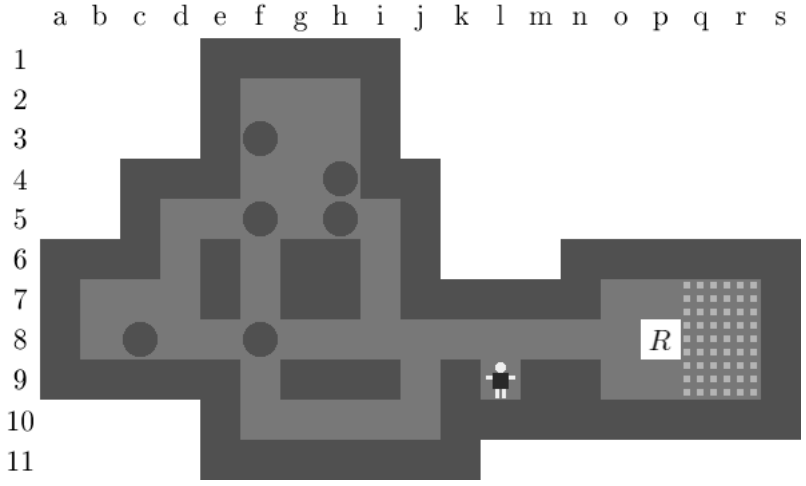

Figure 4.15 – La seule entrée réelle (R) du premier problème de notre benchmark.

4.4.2 Résolution de l'ordonnancement

L'algorithme de résolution de l'ordonnancement correspond à la mise en oeuvre du deuxième agent de notre modèle multi-agent. Cet algorithme prend comme arguments le noeud représentant la situation initiale du problème et l'ordonnancement des goals généré par les algorithmes que nous avons présentés dans la section précédente. Son objectif est de trouver une séquence de poussées qui remplissent les uns après les autres les goals du problème dans l'ordre défini par l'ordonnancement (cf. figure 4.16).

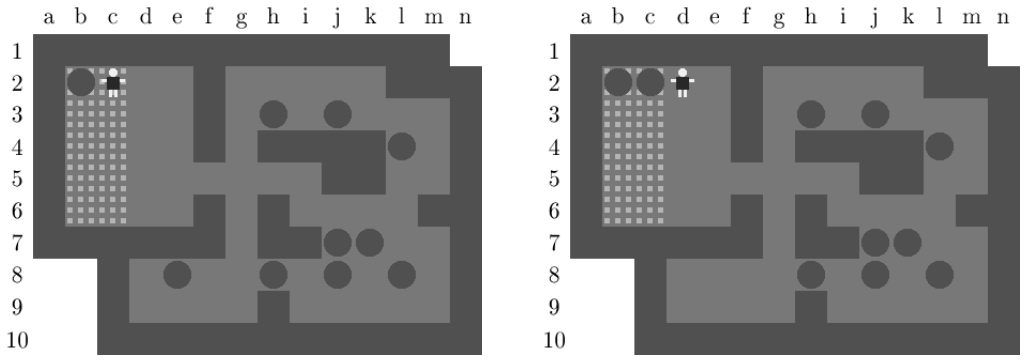

Figure 4.16 – Les deux premières étapes de la résolution du problème 2.

A chaque étape de son exécution, l'algorithme devra résoudre le sous-problème consistant à remplir un goal cible à partir de la situation de jeu issue de la résolution du sous-problème précédent. Nous appellerons le premier goal de l'ordonnancement à ne pas être rempli, le goal cible courant. Une étape de l'exécution de l'algorithme peut être décrite ainsi : soit un noeud *subsolution-node* dans lequel les n premiers goals de l'ordonnancement sont remplis et *target-goal* le goal cible courant, *i.e.* le $(n+1)^{\text{ième}}$ goal de l'ordonnancement ; l'algorithme lance une recherche qui va tenter de remplir le goal *target-goal* en partant de la situation de jeu représentée par le noeud *subsolution-node* ; si une solution est trouvée, elle devient le point de départ d'une recherche qui a pour objectif de remplir le goal suivant de l'ordonnancement, *i.e.* le $(n+2)^{\text{ième}}$ goal ; si, par contre, la recherche échoue, notre algorithme revient à la recherche lancée à l'étape précédente et tente d'obtenir une autre solution pour remplir le $n^{\text{ième}}$ goal. Si une solution permettant de remplir le dernier goal de l'ordonnancement est trouvée, l'algorithme se termine et renvoie le noeud solution trouvé.

La résolution de l'ordonnancement fonctionne donc à la manière d'un algorithme de recherche dans lequel les choix qui conduisent à une impasse sont remis en question. Afin d'implémenter ce retour en arrière, la mise en oeuvre de nos algorithmes de recherche classiques a dû être modifiée afin de ne plus renvoyer seulement la première solution trouvée à un sous-problème mais la suite des solutions rencontrées lors du parcours effectué dans l'arbre de recherche. Nous appellerons la fonction qui nous permettra d'obtenir cette suite de solutions, un générateur de solutions. Un générateur de solutions est une fonction sans argument qui renvoie le prochain noeud solution rencontré. Soit un noeud start-node, une fonction *goal-node* ? et une fonction $find\text{-}successors$ tels que nous les avons définis dans le chapitre précédent, le générateur de solutions correspondant à un parcours de type iterative-deepening sera obtenu par l'expression :

```
(make-solution-generator
 (iterative-deepening start-node goal-node? find-successors))
```

L'implémentation de ces générateurs de solutions fait appel à des notions avancées du langage Scheme, telles que les macros et les évaluations retardées (lazy evaluations). Sa description approfondie sortirait dès lors du cadre de la résolution d'un problème de Sokoban. Nous invitons néanmoins le lecteur intéressé par ce sujet à consulter le code source de notre programme[3] ainsi que, par exemple, [12] qui constitue une excellente référence en la matière.

Le fonctionnement de l'algorithme de résolution de l'ordonnancement repose sur une liste open contenant des structures de données particulières que nous appellerons des méta-noeuds. Un *méta-noeud* est une liste de quatre éléments relatifs à la résolution d'un sous-problème donné :

1. *solution-generator* : un générateur de solutions, *i.e.* une fonction sans argument qui renvoie la solution suivante du sous-problème ;
2. *start-node* : le noeud représentant la situation de jeu initiale du sous-problème, *i.e.* la situation de jeu à partir de laquelle une suite de poussées permettant de remplir le goal cible devra être trouvée ; ce noeud est le noeud solution issu de la résolution du sous-problème précédent ;
3. *remaining-goal-list* : la liste ordonnancée des goals encore non remplis dans la situation de jeu correspondant au noeud start-node ; le premier élément de cette liste est donc le goal cible du sous-problème ;
4. *filled-goal-list* : la liste des goals déjà remplis dans la situation de jeu du noeud start-node.

Soit ($solution\text{-}generator_1, start\text{-}node_1, remaining\text{-}goal\text{-}list_1, filled\text{-}goal\text{-}list_1$) le méta-noeud correspondant à la résolution du sous-problème consistant à remplir le $n^{\text{ième}}$ goal de l'ordonnancement et $subsolution\text{-}node_1$ un noeud solution de ce sous-problème, le méta-noeud relatif à la résolution du sous-problème suivant est la liste ($solution\text{-}generator_2$, $start\text{-}node_2, remaining\text{-}goal\text{-}list_2, filled\text{-}goal\text{-}list_2$) dans laquelle :

- $solution\text{-}generator_2$ est un générateur de solutions qui permettra d'obtenir la suite de noeuds solutions de ce sous-problème ;
- $start\text{-}node_2$ est le noeud $subsolution\text{-}node_1$;
- $remaining\text{-}goal\text{-}list_2$ est la liste $remaining\text{-}goal\text{-}list_1$ amputée de son premier élément ;
- $filled\text{-}goal\text{-}list_2$ est une liste dont la tête est le premier élément de la liste $remaining\text{-}goal\text{-}list_1$ et le reste est la liste $filled\text{-}goal\text{-}list_1$.

L'algorithme de résolution de l'ordonnancement peut maintenant être défini comme suit :

[3] Ce code est disponible à l'adresse : `http://www.student.montefiore.ulg.ac.be/~demaret/tfe/`.

```
(define solve-scheduled-goal-list
  (lambda (start-node scheduled-goal-list)
    (let loop ((open (list (make-start-meta-node ...))))
      (if (null? open)
          #f
          (let* ((first-meta-node (car open))
                 (subsolution-node (meta-node-next-solution first-meta-node))
                 (remaining-goal-list (meta-node-remaining-goal-list
                                       first-meta-node)))
            (if (pair? subsolution-node)
                (if (null? (cdr remaining-goal-list))
                    subsolution-node
                    (loop (cons (make-next-meta-node ...) open)))))
          (loop (cdr open))))))
```

4.4.3 Résolution d'un sous-problème

La méthode de résolution que nous avons introduite repose sur la décomposition d'un problème de Sokoban en une suite de sous-problèmes consistant à trouver un moyen de remplir un goal cible à partir d'une situation de jeu donnée. Dans notre représentation multi-agent, la responsabilité de la résolution d'un sous-problème particulier a été confiée à notre troisième agent. Du point de vue de la planification hiérarchique, cet agent est un agent de bas niveau puisqu'il résoudra le problème qui lui est posé en terme d'actions élémentaires de jeu, *i.e.* en terme de poussées. Cette résolution se fera au moyen d'un algorithme de recherche classique, l'iterative-deepening, et son implémentation passera donc par la définition des fonctions *goal-node*? et *find-successors* qui seront passées en arguments à cet algorithme. Pour rappel, *goal-node*? est un prédicat qui prend comme seul argument un noeud et qui est vrai ssi ce noeud est un noeud solution, et *find-successors* est une fonction qui prend comme unique argument un noeud et qui renvoie la liste des successeurs de ce noeud.

Résoudre un sous-problème consiste à trouver une séquence de poussées permettant de remplir un goal cible. L'opération effectuée par la fonction *goal-node*? consiste donc à vérifier si une pierre de la situation de jeu associée au noeud passé en argument occupe ce goal cible. Contrairement à la fonction que nous avons définie dans la première version de notre programme, la fonction *goal-node*? ne peut pas ici être définie directement puisque chaque sous-problème aura un goal cible qui lui est propre. Nous passerons donc par une fonction *make-goal-node*? qui nous permettra de générer une fonction *goal-node*? adaptée à un sous-problème donné. Cette fonction prend comme unique arguement un goal cible et renvoie la fonction *goal-node*? correspondante :

```
(define make-goal-node?
  (lambda (target-goal)
    (lambda (node)
      (let ((stone-list (node-stone-list node)))
        (member target stone-list)))))
```

L'implémentation de la fonction *find-successors* sera semblable à celle de la fonction

décrite dans la première version de notre programme. Elle intégrera cependant une recherche des possibilités de macro-poussées vers le goal cible. Comme nous l'avons vu, une macro-poussée est une suite de poussées d'une même pierre. La recherche des macro-poussées consiste à rechercher les possibilités d'amener une pierre de la situation de jeu courante vers le goal cible afin de résoudre immédiatement le sous-problème (cf. figure 4.17). L'utilisation des macro-poussées permet de réduire sensiblement la profondeur de l'arbre de recherche associé à la résolution de chaque sous-problème. En effet, une macro-poussée permet d'atteindre en un coup un noeud solution situé plus profondément dans l'arbre de recherche. Cette réduction est particulièrement intéressante puisque le nombre de noeuds à explorer et donc le temps de calcul nécessaire à la résolution d'un sous-problème augmentent exponentiellement avec la profondeur de l'arbre.

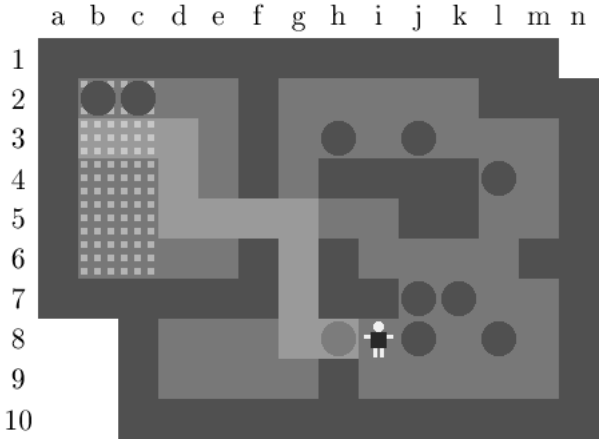

Figure 4.17 – Une macro-poussée de la pierre *h8* vers le goal cible *b3*.

La première étape de notre algorithme consistera donc à rechercher les possibilités de macro-poussées vers le goal cible du sous-problème. Si des macro-poussées sont trouvées, les successeurs normaux du noeud sont ignorés et seuls les successeurs issus de ces macro-poussées sont renvoyés. Cette coupure[4] effectuée dans l'arbre de recherche peut a priori paraître risquée puisqu'elle revient à ignorer l'exploration de certaines possibilités de coups. Une analyse par cas permet cependant de justifier son utilisation. Deux situations peuvent se présenter suite à la résolution d'un sous-problème par une macro-poussée. Dans la première, la solution obtenue pour le sous-problème s'avère être une bonne solution et les sous-problèmes suivants peuvent être résolus sans qu'elle doive être remise en cause. La coupure effectuée est ici sans conséquence puisque les successeurs ignorés n'auraient de toute façon pas été explorés.

La seconde situation à envisager est celle où la solution issue de la macro-poussée n'est pas une bonne solution et ne permet dès lors pas de résoudre les sous-problèmes suivants. On peut dans ce cas montrer que l'exploration des successeurs ignorés aurait été inutile. En effet, dans l'hypothèse où le problème considéré ne comporte qu'une seule entrée, la résolution des sous-problèmes suivants dépend exclusivement de la position des pierres n'occupant pas les goals déjà remplis. On peut dès lors affirmer que si les sous-problèmes suivants

[4]On parle de « coupure » dans un arbre de recherche quand certains successeurs d'un noeud sont volontairement ignorés. Cette opération équivaut, en effet, à amputer l'arbre de recherche des sous-arbres issus de l'exploration de ces successeurs.

n'ont pas pu être résolus c'est que ces pierres induisaient une situation de deadlock. On peut également affirmer que le noeud à partir duquel la macro-poussée avait été effectuée était une situation de deadlock puisqu'il contenait ces mêmes pierres. Or, une situation de deadlock est irrémédiable et il n'existe ainsi aucune séquence de poussées permettant de revenir à une situation de jeu normale. L'exploration de ces successeurs ignorés aurait dès lors été inutile puisqu'elle aurait conduit à explorer des sous-arbres issus d'une situation de deadlock.

Si aucune possibilité de macro-poussée vers le goal cible n'est identifiée, la liste des successeurs normaux du noeud est générée à partir de la liste des poussées possibles comme c'était le cas dans la première implémentation de notre programme. Afin de ne pas modifier les positions des pierres occupant les goals remplis lors de la résolution des sous-problèmes précédents, les éventuelles poussées de ces pierres sont ignorées. Notre algorithme intègre également une amélioration spécifique décrite dans [3]. Elle consiste à trier la liste des successeurs d'un noeud de façon à ce que les successeurs issus d'une poussée impliquant la même pierre que la poussée précédente soient explorés en premier. Ce tri de la liste des successeurs permet de privilégier l'exploration des suites de poussées d'une même pierre. Dans [3], les auteurs parlent d'ailleurs de cette méthode en utilisant le terme « inertie ». L'ajout de cette notion d'inertie exploite une connaissante empirique du domaine afin de privilégier certains noeuds jugés plus prometteurs et est donc à rapprocher de l'utilisation d'une heuristique.

La fonction *find-successors* peut finalement être définie par :

```
(define find-successors
  (lambda (node)
    (let ((macro-list (find-macros ...)))
      (if (pair? macro-list)
          macro-list
          (let ((next-push-list (find-stone-pushes ...)))
            (let loop ((next-push-list (sort-next-pushes ...))
                       (successor-list '()))
              (if (null? next-push-list)
                  successor-list
                  (if (or (push-filled-goal? ...)
                          (lead-to-deadlock? ...))
                      (loop (cdr next-push-list)
                            successor-list)
                      (loop (cdr next-push-list)
                            (cons (make-successor-node ...) successor-list)))))))))))
```

Recherche des macro-poussées

La recherche des macro-poussées consiste à parcourir la liste des pierres qui n'ont pas encore été amenées dans un goal et à lancer pour chacune d'elle une recherche qui tentera de trouver un moyen de l'amener dans le goal cible courant sans avoir à modifier la position d'une autre pierre. Cette dernière recherche correspond à une recherche de chemin classique

dans un arbre dont la racine est la position initiale de la pierre et dont chaque noeud a comme successeurs les positions accessibles par cette pierre en une poussée. La fonction *find-macros* se définit comme suit :

```
(define find-macros
  (lambda (node target-goal filled-goal-list)
    (let ((stone-list (node-stone-list node)))
      (let loop ((iterated-stone-list (difference stone-list filled-goal-list))
                 (macro-list '()))
        (if (null? iterated-stone-list)
            macro-list
            (let* ((stone-coord (car iterated-stone-list))
                   (push-path (find-push-path stone-coord target-goal ...)))
              (if (pair? push-path)
                  (loop (cdr iterated-stone-list)
                        (cons (make-macro-node ...) macro-list))
                  (loop (cdr iterated-stone-list)
                        macro-list)))))))))
```

Si une macro-poussée est identifiée, un noeud successeur est construit pour cette macro-poussée. Soit $((\textit{man-pos}_1, \textit{stone-list}_1), \textit{push-list}_1)$ le noeud à partir duquel la macro-poussée a été identifiée, *stone-coord* la position initiale de la pierre impliquée et (p_1, p_2) la dernière poussée de la liste de poussées correspondant à la macro-poussée, le noeud successeur issu de la macro-poussée est le noeud $((\textit{man-pos}_2, \textit{stone-list}_2), \textit{push-list}_2)$ dans lequel :

- $\textit{man-pos}_2$ est la position p_1 ;
- $\textit{stone-list}_2$ est la liste $\textit{stone-list}_1$ dans laquelle *stone-coord* a été remplacée par p_2 ;
- $\textit{push-list}_2$ est la concaténation de la liste de poussées correspondant à la macro-poussée et de $\textit{push-list}_1$.

Chapitre 5

Apprendre

Une caractéristique importante d'un agent intelligent est sa faculté d'apprendre. Sous sa forme la plus simple, celle-ci se traduit par son aptitude à ne pas commettre deux fois une même erreur. Dans le cas de la résolution d'un problème de Sokoban, commettre une erreur c'est créer une situation de deadlock. Ce chapitre s'attache à la mise en place d'un mécanisme destiné à éviter qu'une situation de deadlock rencontrée au cours de la résolution d'un problème ne soit reproduite une seconde fois.

La première idée que nous avons développée dans cet objectif a été d'utiliser une table de transposition afin de garder en mémoire les situations de jeu à partir desquelles aucun successeur n'avait pu être trouvé et qui étaient par conséquent identifiées comme des situations de deadlock. Cette table permettait d'exclure par la suite ces situations de jeu de la liste des successeurs d'un noeud exploré. L'idée sous-jacente à l'utilisation de cette table était d'exploiter le parcours itératif de l'arbre de recherche effectué par l'algorithme iterative-deepening lors de la résolution d'un sous-problème (cf. section 4.4.3). En effet, si l'ensemble des successeurs d'un noeud étaient présents dans la table à la fin d'une itération n, ce noeud était lui-même ajouté dans la table au cours de l'itération $(n+1)$. Ainsi, au fur et à mesure des itérations, les sous-arbres dont toutes les feuilles représentaient des situations de deadlock étaient progressivement éliminés de l'arbre de recherche.

Cependant, les expériences effectuées sur notre benchmark de 90 problèmes nous ont montré que l'efficacité de cette première idée était limitée. En effet, bien qu'elle permette la plupart du temps à l'algorithme de recherche de développer entre 5 % et 10 % de noeuds en moins, le coût supplémentaire en temps de calcul occasionné par l'utilisation de la table de transposition la rendait finalement improductive. Nous n'avons donc pas conservé cette idée et nous nous sommes tourné vers une méthode plus intéressante permettant d'exclure de l'arbre de recherche non pas une mais un ensemble de situations de jeu à partir d'une situation de deadlock identifiée.

Cette seconde idée repose sur l'observation suivante : dans la plupart des cas, une situation de deadlock n'est pas induite par la position de l'ensemble des pierres du problème mais par un sous-ensemble d'entre elles. On peut dès lors considérer la position des pierres qui ne font pas partie de ce sous-ensemble comme quelconque. En conséquence, toutes les

situations de jeu qui contiennent ce sous-ensemble, que nous appellerons un pattern, sont également des situations de deadlock (cf. figure 5.1). Or, il peut exister un nombre potentiellement très élevé de telles situations de jeu. Ce nombre sera généralement inversement proportionnel au nombre de pierres présentes dans le pattern. Identifier et garder en mémoire ce sous-ensemble de pierres permet donc d'exclure de l'arbre de recherche un nombre important de situations de jeu et de réduire ainsi sensiblement son facteur de branchement.

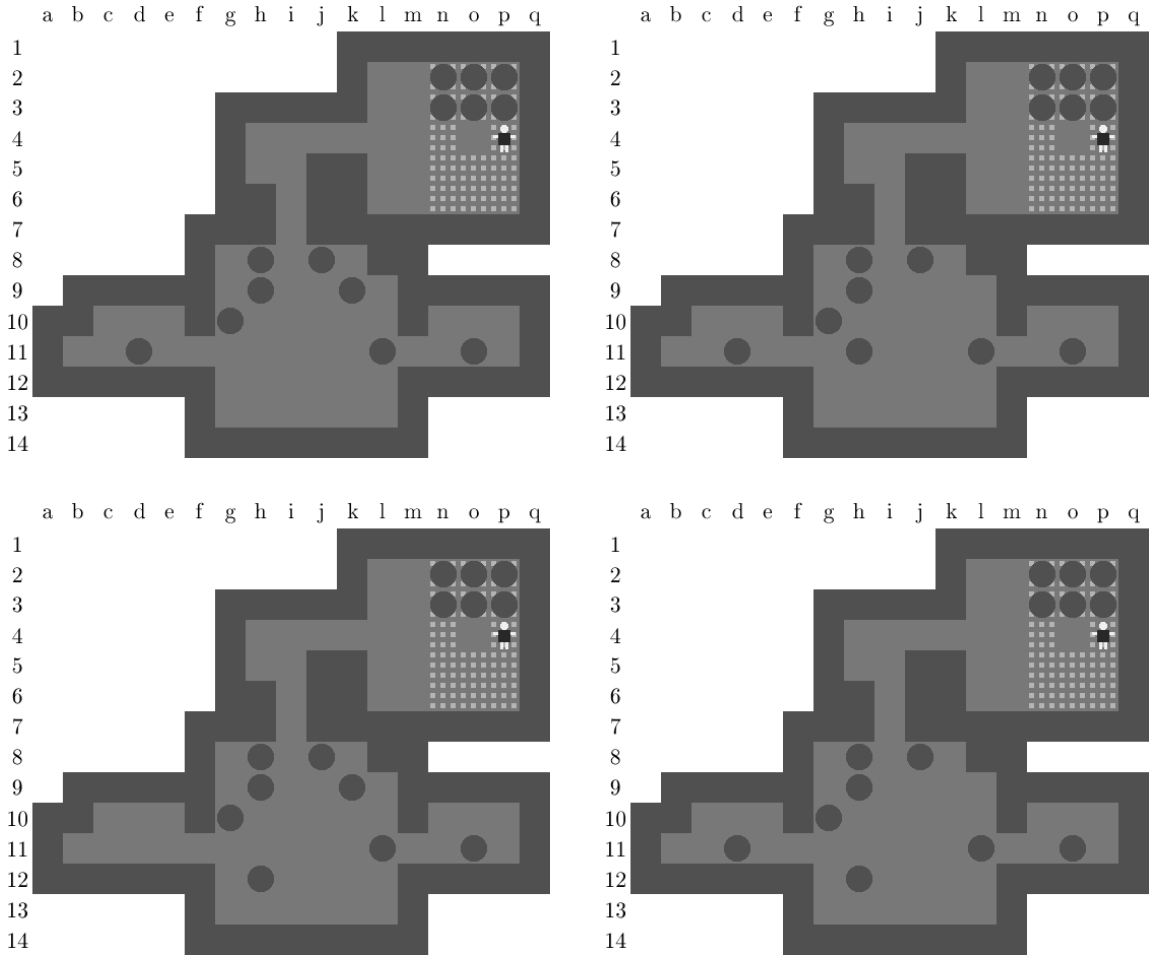

Figure 5.1 – Exemples de situations de deadlock contenant le même pattern (*h8*,*h9*,*g10*).

C'est l'observation des exécutions de la deuxième version de notre programme sur des problèmes de notre benchmark qui nous a conduit à envisager l'identification de ces sous-ensembles de pierres impliquant une situation de deadlock. Les situations de jeu de la figure 5.1 sont extraites de l'une de ces exécutions. Il s'agit d'une suite de quatre solutions obtenues pour le sous-problème consistant à remplir le sixième goal de l'ordonnancement du problème 9 du benchmark. L'intérêt de l'identification des patterns apparaît ici comme évident. En effet, si le pattern commun à ces solutions avait pu être identifié lors de la remise en question de la première solution, les trois solutions suivantes auraient pu être rejetées immédiatement et n'auraient pas conduit à l'exploration inutile de trois sous-arbres issus de trois situations de deadlock.

Un mécanisme comparable est également mis en oeuvre dans le solveur *Rolling Stone*.

Dans le chapitre 5 de [3], les auteurs décrivent comment leur programme analyse certaines situations de jeu jugées critiques (par une heuristique spécifique) afin d'identifier les conflits potentiellement existants à l'intérieur de sous-ensembles de pierres. Les sous-ensembles conflictuels identifiés sont gardés en mémoire et ensuite utilisés afin d'ajuster l'évaluation de la distance entre un noeud donné et le noeud solution recherché, évaluation générée par la fonction d'heuristique utilisée par *Rolling Stone*. Cet ajustement prend la forme de l'application d'une pénalité, *i.e.* une augmentation du coût évalué d'un noeud. Par exemple, si l'analyse effectuée permet de déterminer que les pierres d'un sous-ensemble se trouvent en position de deadlock, la pénalité appliquée à un noeud contenant ce sous-ensemble sera infinie. On peut, en effet, considérer que la distance entre un noeud représentant une situation de deadlock et un noeud solution est infinie puisque ce noeud solution ne pourra jamais être atteint.

5.1 Identification des patterns

L'identification des patterns consiste à identifier et à stocker des sous-ensembles de pierres impliquant une situation de deadlock, puis à les utiliser pour exclure les situations de jeu les contenant de la liste des successeurs d'un noeud. Ne conserver en mémoire que les sous-ensembles de pierres est cependant insuffisant. En effet, certaines situations de deadlock dépendent également de la position du joueur ou, plus exactement, de l'ensemble des positions qui lui sont accessibles (cf. figure 5.2). Il faudra donc associer à chaque sous-ensemble de pierres cette position afin de ne pas exclure de l'arbre de recherche des situations de jeu qui seraient erronément identifiées comme étant des situations de deadlock. Un pattern stocké dans la base de données sera ainsi constitué d'un sous-ensemble de pierres et d'une position du joueur.

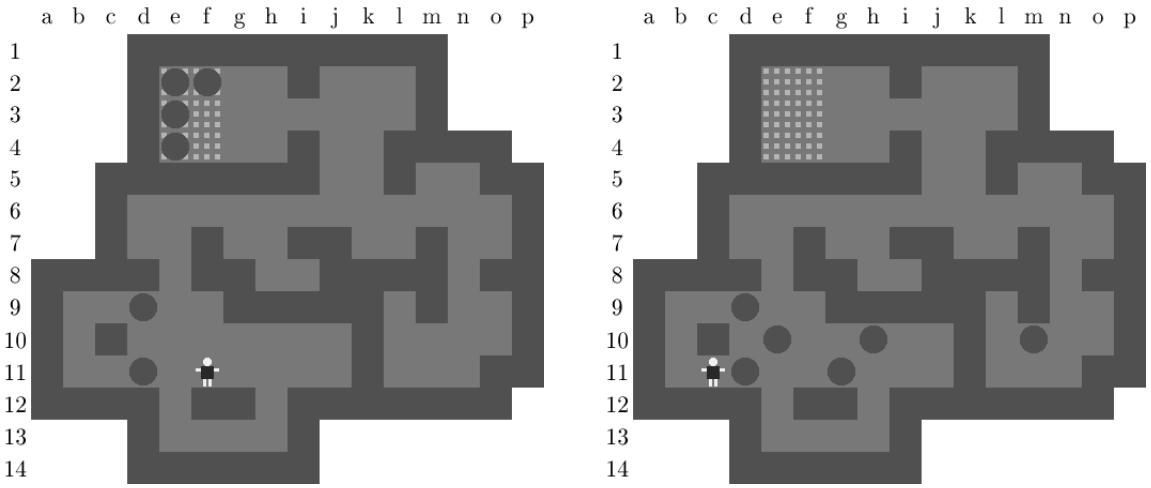

Figure 5.2 – Exemple de situation de deadlock où la position du joueur est déterminante.

Dans le problème 17 du benchmark, le couple de pierres (*d9,d11*) implique ou non une situation de deadlock en fonction de la position du joueur. La situation de gauche est une situation de deadlock puisqu'il n'existe aucun moyen d'amener ces pierres vers un goal. La situation de droite est la situation initiale du problème.

Afin d'intégrer l'identification des patterns dans notre programme, nous devons répondre aux trois questions suivantes : (1) à partir de quelles situations de jeu le processus d'identification des patterns doit-il être lancé ? (2) quels sous-ensembles de pierres d'une situation de jeu faut-il examiner ? et (3) comment déterminer si un sous-ensemble de pierres donné provoque une situation de deadlock ?

Afin de répondre à la première question, nous devons définir les cas où une situation de jeu devra faire l'objet d'une identification des patterns. Le fait même que cette question soit posée nous suggère que cette identification ne pourra être faite sur l'ensemble des situations de jeu rencontrées. Effectivement, même si cette opération n'a pas encore été définie, on se rend aisément compte qu'elle nécessitera l'utilisation d'un algorithme de recherche et qu'elle sera par conséquent relativement gourmande en temps de calcul. Il s'avèrera ainsi plus judicieux de ne l'exécuter que pour un nombre restreint de situations de jeu pour lesquelles on pourra prédire avec certitude l'existence d'au moins un sous-ensemble de pierres impliquant un deadlock.

Dans le chapitre précédent, nous avons discuté du cas d'un noeud qui était solution d'un sous-problème mais à partir duquel les sous-problèmes suivants n'avaient pas pu être résolus. Nous avons vu qu'on pouvait affirmer que la situation de jeu associée à ce noeud était une situation de deadlock et que ce deadlock était induit par la position des pierres n'occupant pas les goals déjà remplis de l'ordonnancement. Une telle situation de jeu est un candidat idéal pour une identification des patterns qui pourra dès lors se limiter à l'analyse du sous-ensemble de pierres n'occupant pas les goals déjà remplis. Les situations de jeu qui feront l'objet d'une identification des patterns seront ainsi les situations associées aux noeuds solutions rejetés des sous-problèmes.

Cependant, afin de réduire le nombre de situations analysées, une situation de jeu issue d'un noeud solution rejeté ne fera l'objet d'une identification des patterns que si aucune solution n'a pu être trouvée au problème suivant. En d'autres termes, seules les situations de jeu associées à des feuilles de l'arbre exploré par le second agent seront analysées (cf. section 4.3). On peut justifier la mise en place de cette restriction par les considérations suivantes :

- Les déplacements des pierres impliquant une situation de deadlock sont par nature limités voire impossibles. Un pattern présent dans une solution à un sous-problème sera ainsi la plupart du temps également présent dans les solutions des sous-problèmes suivants. Si un noeud solution a permis de résoudre le sous-problème suivant, l'identification des patterns à partir de ce noeud sera souvent inutile puisque les éventuels patterns qu'il contient auront pu être identifiés lors de la remise en cause des solutions obtenues pour les sous-problèmes suivants (cf. figure 5.3).
- Si une solution au sous-problème suivant a pu être trouvée, c'est qu'une des pierres appartenant au sous-ensemble des pierres n'occupant pas les goals déjà remplis a pu être amenée dans le goal cible suivant. Cette pierre n'était dès lors pas impliquée dans la situation de deadlock. A contrario, si aucune solution au sous-problème suivant n'est trouvée, c'est que le sous-ensemble des pierres n'occupant pas les goals déjà remplis forme une situation de deadlock dans laquelle toutes les pierres sont impliquées. Dans ces situations, l'identification des patterns sera ainsi facilitée grâce au fait que le nombre de pierres à prendre en compte dans l'analyse aura été réduit au maximum

par les résolutions successives des sous-problèmes.

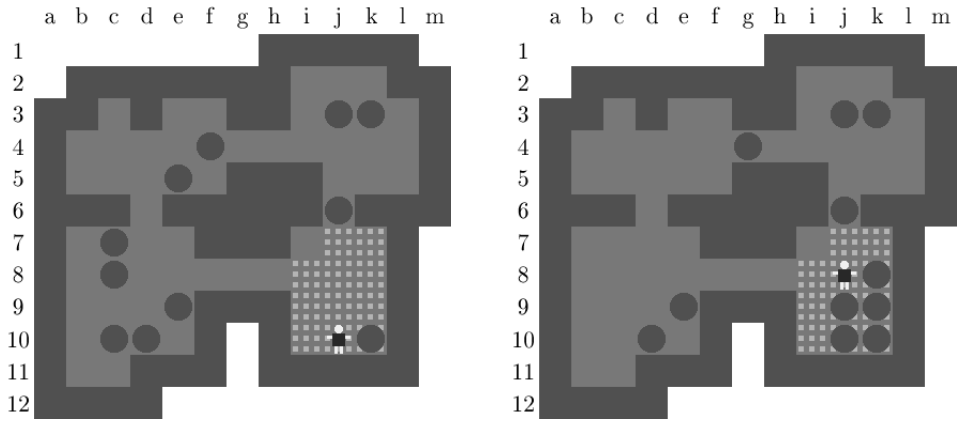

Figure 5.3 – Propagation d'un pattern.

Le pattern (*e9,d10*) créé lors de la résolution du premier sous-problème (à gauche) est toujours présent dans la solution du cinquième sous-problème (à droite).

Quels sous-ensembles de pierres d'une situation de jeu faut-il examiner ? Pour cette question également, une analyse exhaustive ne sera pas envisageable. En effet, le nombre de sous-ensembles de pierres existant croit exponentiellement avec le nombre de pierres présentes dans la situation de jeu examinée[1]. Il s'avère dès lors nécessaire de définir des sous-ensembles de pierres de façon à ce qu'ils soient pertinents et que leur nombre reste raisonnable. Lors de nos expérimentations, nous avons pu remarquer que beaucoup de patterns étaient formés par des groupes de pierres contiguës. Nous avons dès lors choisi d'utiliser ces groupes de pierres contiguës, que nous appellerons clusters, comme constituants des sous-ensembles. Précisons qu'une pierre isolée sera considérée comme faisant partie d'un cluster dont elle sera le seul élément. Les sous-ensembles examinés seront ainsi les sous-ensembles constitués d'un, deux ou trois clusters (cf. figure 5.4).

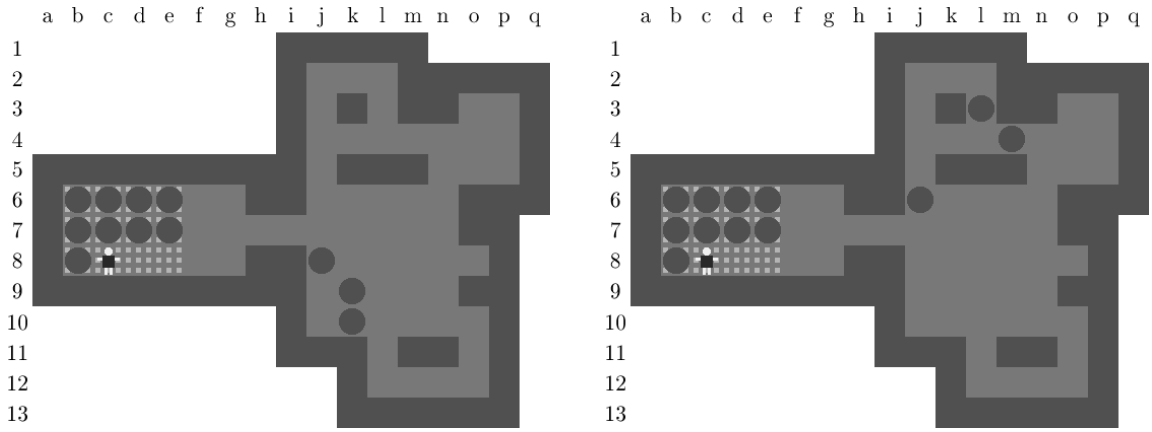

Figure 5.4 – Situations de deadlock induites par un (à gauche) et deux clusters (à droite).

Comment déterminer si un sous-ensemble de pierres donné implique une situation de deadlock ? Comme nous l'avons déjà évoqué, cette opération fera appel à un algorithme de

[1] On peut générer $(2^n - 1)$ sous-ensembles non vides distincts à partir d'un ensemble de n éléments.

recherche. Elle consistera à rechercher un moyen de remplir le goal cible du sous-problème pour lequel aucune solution n'a pu être trouvée, à partir d'une situation de jeu temporaire dans laquelle les pierres n'appartenant pas au sous-ensemble examiné ont été retirées. S'il n'existe aucune possibilité d'amener une pierre vers le goal cible, le sous-ensemble est identifié comme étant un pattern et est ajouté dans la base de données (cf. figure 5.5).

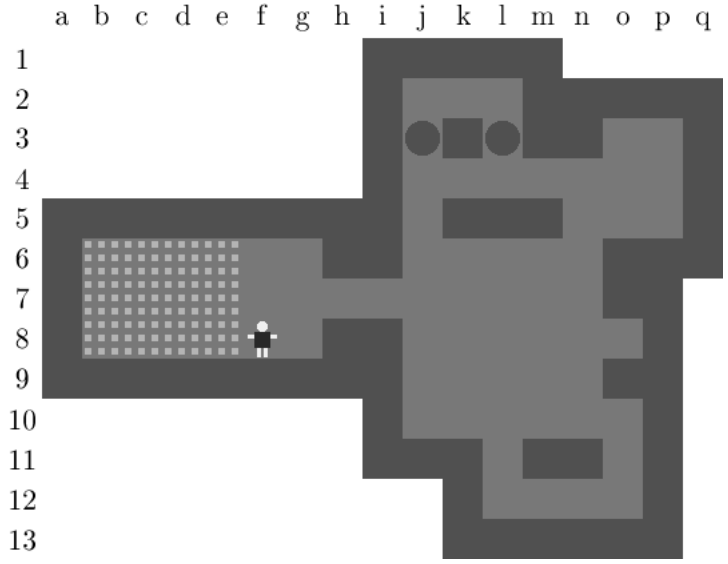

Figure 5.5 – Examen d'un sous-ensemble de pierres.

Dans cet exemple, le goal cible est le goal *e6*. Si aucune des pierres du sous-ensemble examiné ne peut être amenée dans ce goal, ce sous-ensemble sera identifié comme étant un pattern.

5.2 Implémentation version 2 (suite)

5.2.1 Recherche des patterns

L'algorithme d'identification des patterns prend comme arguments la position du joueur, une sous-liste des pierres du problème et un goal cible. Nous avons déjà évoqué les grandes étapes de son exécution en présentant la méthode d'identification des patterns utilisée par notre programme. Nous détaillerons donc dans cette section l'implémentation de chacune de ces étapes.

Recherche des clusters

La recherche des clusters consiste à regrouper les pierres de la sous-liste en clusters de pierres contiguës. Afin de construire la liste des clusters, l'algorithme identifie le cluster auquel appartient la première pierre de la sous-liste et retire ensuite de cette sous-liste les

pierres appartenant à ce cluster. Il recommence alors plusieurs fois cette opération jusqu'à ce que toutes les pierres de la sous-liste aient été placées dans un cluster.

La construction de la liste de clusters peut être définie par la fonction :

```
(define find-stone-clusters
  (lambda (stone-list)
    (let loop ((stone-list stone-list)
               (cluster-list '()))
      (if (null? stone-list)
          cluster-list
          (let ((stone-cluster (find-stone-cluster (car stone-list) stone-list)))
            (loop (difference stone-list stone-cluster)
                  (cons stone-cluster cluster-list)))))))
```

Pour identifier le cluster auquel appartient une pierre donnée, l'algorithme explore les successeurs de la position de cette pierre afin de déterminer les pierres qui lui sont directement contiguës. Il explore ensuite les successeurs des positions de ces pierres et ainsi de suite jusqu'à ce que toutes les pierres du cluster aient été rencontrées.

Parcours des sous-ensembles de clusters

Les sous-ensembles examinés seront composés d'un, deux ou trois clusters. L'algorithme commence ainsi par construire la liste des sous-ensembles composés d'un seul cluster. Il parcourt ensuite cette liste et examine chaque sous-ensemble afin d'identifier les éventuels patterns. Il procède ensuite de la même manière avec les sous-ensembles composés de deux, puis de trois clusters.

Par ailleurs, un sous-ensemble ne devra pas être examiné si :

- Un sous-ensemble équivalent a été examiné précédement au cours de la résolution du problème. Deux sous-ensembles seront équivalents s'ils contiennent les mêmes pierres et qu'il existe un chemin pour le joueur entre les deux positions du joueur associées à ces sous-ensembles. Une table de transposition est ainsi utilisée afin d'assurer qu'un sous-ensemble donné n'est examiné qu'une seule fois.
- Toutes les pierres présentes dans ce sous-ensemble occupent des positions de goals. En effet, un pattern qui contiendrait un tel sous-ensemble de pierres exclurait de facto le noeud solution recherché.
- Un sous-ensemble de pierres inclus dans ce sous-ensemble a déjà été ajouté dans la base de données durant l'analyse de cette situation de jeu. Le pattern éventuellement identifié serait en effet inutile puisqu'il serait inclus dans le pattern déjà ajouté.

Le parcours des sous-ensembles de clusters sera ainsi défini par :

```
(let* ((cluster-list (find-stone-clusters ...))
       (n (length cluster-list)))
  (let loop ((k 1))
    (if (and (<= k 3) (< k n))
        (let loop-subset ((subset-list (make-k-clusters-subset-list ...)))
          (if (null? subset-list)
              (loop (1+ k))
              (let ((stone-subset (car subset-list)))
                (begin
                  (if (not (or (stone-subset-in-hashtable? ...)
                               (stone-subset-in-goal-list? ...)
                               (stone-subset-in-inserted-pattern-list? ...)))
                      (examine-stone-subset ...))
                  (loop-subset (cdr subset-list)))))))))
```

Examen d'un sous-ensemble

L'examen d'un sous-ensemble aura pour objectif de déterminer si ce sous-ensemble est un pattern, *i.e.* s'il implique une situation de deadlock. Cette opération débute par la création d'une situation de jeu temporaire dans laquelle seules les pierres du sous-ensemble examiné sont présentes. Cette situation temporaire est utilisée comme point de départ d'une recherche qui tentera de trouver un moyen d'amener l'une des pierres du sous-ensemble vers le goal cible. Cette recherche équivaut à la résolution d'un sous-problème et sera dès lors confiée au troisième agent de notre modèle multi-agent (cf. section 4.3). Une recherche en profondeur limitée à quelques coups sera cependant utilisée afin de restreindre le temps de calcul consacré à cette opération. Trois cas pourront se présenter à la fin du processus de recherche :

1. Une solution pour remplir le goal cible a été trouvée. Dans ce cas, notre méthode considère que le sous-ensemble examiné n'induit pas de situation de deadlock.
2. La recherche a échoué mais certains noeuds n'ont pas pu être explorés complètement à cause de la profondeur maximale fixée. Dans ce cas, notre méthode ne peut conclure et le sous-ensemble n'est pas ajouté dans la base de données. Il serait effectivement hasardeux d'ajouter dans la base de données un sous-ensemble qui ne serait pas avec certitude un pattern.
3. La recherche a échoué et tous les noeuds de l'arbre de recherche ont été explorés complètement. Dans ce cas, la méthode peut conclure avec certitude que le sous-ensemble examiné implique une situation de deadlock et que toutes les situations de jeu qui le contiennent sont par conséquent des situations de deadlock. Le pattern est donc ajouté dans la base de données.

Finalement, si aucun sous-ensemble n'a été identifié comme étant un pattern, la sous-liste de pierres passée en argument à l'algorithme est ajoutée dans la base de données. En effet, les pierres de cette sous-liste sont les pierres n'occupant pas les goals déjà remplis d'un noeud solution à partir duquel le goal cible suivant n'a pas pu être rempli. Comme nous l'avons vu, ces pierres induisent une situation de deadlock et les pierres de la sous-liste forment donc un pattern.

5.2.2 Insertion d'un pattern dans la base de données

Comme défini plus haut, les deux informations associées à un pattern sont (1) la liste des pierres du sous-ensemble impliquant la situation de deadlock et (2) la position du joueur de cette situation. Un pattern de n pierres sera représenté par une liste de $(n+1)$ positions. Le premier élément de cette liste correspondra à la position du joueur et les n éléments suivants correspondront aux positions des n pierres du sous-ensemble.

La base de données des patterns sera implémentée sous la forme d'une liste de patterns. L'insertion d'un pattern se fera donc en ajoutant ce pattern en tête de cette liste. Bien qu'elle semble triviale, cette opération demande cependant certaines précautions. Effectivement, on ne peut pas exclure qu'un pattern soit inclus dans un autre. Nous dirons que le pattern A est inclus dans le pattern B si toutes les pierres du pattern B sont présentes dans le pattern A et s'il existe un chemin pour le joueur entre les positions du joueur du pattern B et du pattern A. Dans cette définition, la notion d'inclusion peut sembler contre intuitive puisque le pattern qui contient le plus de pierres est inclus dans l'autre. Elle fait cependant référence à la quantité d'information associée à un pattern, *i.e.* au nombre de situations de jeu qu'il peut potentiellement exclure. En effet, toutes les situations de jeu exclues par le pattern A seront également exclues par le pattern B puisque ces situations de jeu contiendront de facto ce pattern B. En ce sens, il est naturel de dire que le pattern A est inclus dans le pattern B (cf. figure 5.6).

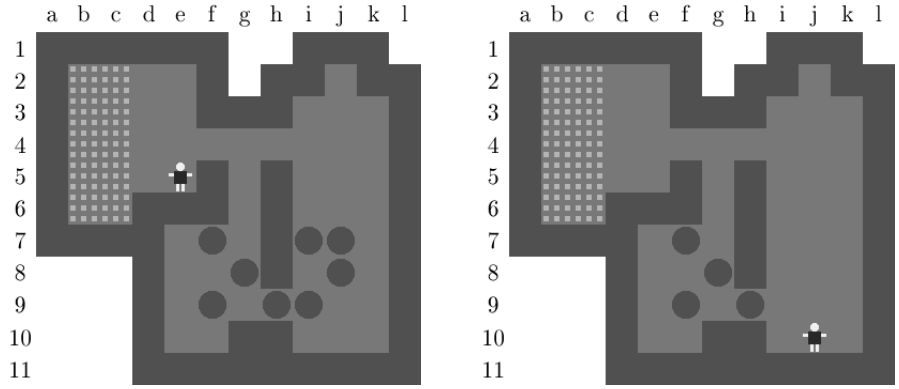

Figure 5.6 – Inclusion entre deux patterns.

Le pattern de gauche est inclus dans celui de droite. En effet, toutes les situations de jeu qui contiennent le pattern de gauche contiennent également le pattern de droite.

Deux situations peuvent donc se présenter lors de l'ajout d'un pattern dans la base de données :

1. Le pattern à ajouter est inclus dans l'un des patterns déjà présents dans la base de données. Dans ce cas, le nouveau pattern ne permet d'exclure aucune situation de jeu supplémentaire et il n'est dès lors pas ajouté.
2. Le pattern à ajouter inclut un ou plusieurs patterns de la base de données. Dans ce cas, ces patterns doivent être supprimés de la base de données puisque l'ajout du nouveau pattern les rend inutiles.

Lors de l'ajout d'un pattern, la base de données doit donc être parcourue afin de vérifier la pertinence de cet ajout et de supprimer les éventuels patterns devenus inutiles. Soit le prédicat *subpattern*? qui prend deux arguments, deux patterns, et qui est vrai ssi le premier pattern est inclus dans le second, la fonction d'insertion d'un pattern dans la base de données est définie par :

```
(define insert-pattern!
  (lambda (pattern-db man-pos stone-subset)
    (let (inserted-pattern (make-pattern man-pos stone-subset))
      (let loop ((pattern-list (pattern-db-pattern-list pattern-db))
                 (updated-pattern-list '()))
        (if (null? pattern-list)
            (update-pattern-db pattern-db (cons inserted-pattern updated-pattern-list))
            (let ((first-pattern (car pattern-list))
                  (if (not (subpattern? inserted-pattern first-pattern))
                      (if (subpattern? first-pattern inserted-pattern)
                          (loop (cdr pattern-list)
                                updated-pattern-list)
                          (loop (cdr pattern-list)
                                (cons first-pattern updated-pattern-list)))))))))))
```

5.2.3 Reconnaissance des patterns

La reconnaissance des patterns a pour objectif de déterminer si un pattern stocké dans la base de données est présent dans une situation de jeu donnée. La première étape de cet algorithme consiste à créer un vecteur *stone-vector* correspondant à la situation de jeu examinée. Ce vecteur est tel que *stone-vector*$[i]$ est vrai ssi une pierre occupe la position i dans cette situation de jeu. Il permet ainsi de déterminer en un temps $O(1)$ si une pierre occupe une position donnée et donc de déterminer en un temps $O(n)$ si les pierres d'un pattern de n pierres sont présentes dans la situation de jeu.

Dans une seconde étape, la liste des patterns de la base de données est parcourue et chacun des patterns est recherché dans la situation de jeu examinée. Cette opération consiste à d'abord vérifier si chaque pierre du pattern est présente dans la situation de jeu et à ensuite vérifier l'existence d'un chemin pour le joueur entre la position du joueur du pattern et la position du joueur de la situation de jeu. Si ces deux conditions sont remplies, la présence de ce pattern dans la situation de jeu examinée est établie et cette dernière est dès lors identifiée comme étant une situation de deadlock.

On définit ainsi la fonction :

```
(define match-pattern
  (lambda (pattern-db man-pos stone-list)
    (let ((stone-vector (make-stone-vector stone-list ...)))
      (let loop ((pattern-list (pattern-db-pattern-list pattern-db)))
        (and (pair? pattern-list)
             (or (let* ((first-pattern (car pattern-list))
```

```
            (p-man-pos (pattern-man-pos first-pattern))
            (stone-subset (pattern-stone-subset first-pattern)))
      (and (match-stone-subset stone-subset stone-vector)
            (find-man-path p-man-pos man-pos stone-subset ...)))
(loop (cdr pattern-list))))))))
```

Chapitre 6

Résultats

Dans la première partie de ce travail, nous avons montré les limites de l'utilisation des algorithmes de recherche classiques pour résoudre des problèmes de Sokoban non triviaux. Comme déjà signalé, la résolution du premier et du plus simple des 90 problèmes de notre benchmark a demandé, en effet, plus de 8 heures de temps de calcul à la première version de notre programme.

La double approche que nous venons de développer en nous inspirant de méthodes de planification et d'apprentissage se révèle beaucoup plus efficace puisque la deuxième version de notre programme est capable de résoudre 54 problèmes du benchmark. Ainsi, bien qu'elle ne nous permette pas encore d'atteindre le score de 59 problèmes résolus affiché par le meilleur solveur actuellement documenté, les perspectives offertes par notre méthode sont, sans aucun doute, très encourageantes.

Le tableau 6.1 présente les données relatives à la résolution de ces 54 problèmes. La deuxième colonne correspond à la longueur — exprimée en poussées — des solutions générées par notre programme. Le nombre total de noeuds développés durant la résolution d'un problème est reporté dans la troisième colonne. Ce total englobe les noeuds explorés lors de la résolution de chaque sous-problème et lors de la recherche des patterns. La quatrième colonne correspond, quant à elle, au temps de calcul qui a été nécessaire pour résoudre le problème[1]. Enfin, la valeur de la cinquième colonne est la profondeur maximale qui a été fixée pour la résolution de chaque sous-problème. L'introduction de cette limitation permet d'accélérer la résolution d'un problème en évitant que la recherche ne s'enfonce trop profondément dans l'exploration inutile de l'arbre d'un sous-problème. Seule la résolution du premier sous-problème, *i.e.* le sous-problème consistant à remplir le premier goal de l'ordonnancement, n'est pas limitée. En effet, l'arbre de recherche associé à ce sous-problème est unique puisqu'il est issu du noeud représentant la situation initiale du problème. Son exploration ne sera par conséquent jamais superflue puisque le noeud solution recherché du sous-problème se trouve avec certitude dans cet arbre.

Cependant, la fixation d'une profondeur maximale peut également nuire à la résolution

[1] La machine utilisée est un ordinateur de bureau standard (CPU à 1.8GHz et 512 Mo de RAM).

d'un problème si la résolution d'un de ses sous-problèmes demande une exploration plus profonde que la limite fixée. Une bonne stratégie est dès lors d'incrémenter progressivement cette limite. Nous avons ainsi, dans un premier temps, essayé de résoudre les problèmes avec une profondeur maximale fixée à 5. Puis dans un second temps, nous avons tenté de résoudre les problèmes qui n'avaient pas pu être résolus, en fixant cette limite à 10.

Dernière remarque, la plupart des problèmes comprenant plusieurs entrées ont été résolus en utilisant notre deuxième algorithme d'ordonnancement. Seuls les problèmes 34 et 72 ont nécessité l'utilisation du troisième algorithme, basé sur l'encerclement des goals, pour être résolus.

Problème	Poussées	Noeuds explorés	Temps	Limite de profondeur (des sous-problèmes)
1	97	97	1 s	5
2	137	291	3 s	5
3	160	32	1 s	5
4	361	6858	1 min 26 s	5
5	149	133	4 s	5
6	110	193	2 s	5
7	114	427	4 s	5
8	248	77	14 s	5
9	241	2193	27 s	5
10	536	710	3 min 51 s	5
11	249	83711	27 min 25 s	5
12	222	31	10 s	5
13	266	17918	11 min 2 s	5
17	213	1427	13 s	5
19	318	454	20 s	5
21	163	2402	36 s	5
24	560	403229	3 h 2 min	10
25	390	516000	6 h 11 min	5
26	197	1171	19 s	5
27	371	10183	1 min 46 s	5
34	186	503	12 s	5
36	521	147434	1 h 5 min	10
37	360	1078	1 min 5 s	5
38	91	14749	2 min 9 s	10
43	152	1166	16 s	5
45	312	942964	3 h 13 min	10
51	132	121	6 s	5
53	208	15	8 s	5
54	271	162	19 s	5
55	122	15	6 s	5
56	243	35	7 s	5
57	243	65	21 s	5
58	283	306	10 s	5
59	250	272	13 s	5
60	172	182	8 s	5
61	283	310	21 s	5
62	251	1649	56 s	5
63	439	40	10 s	5
64	407	154	8 s	5
65	231	122	10 s	5
67	415	580	25 s	5
68	351	395	13 s	10
70	369	2161	1 min 34 s	5
72	358	86	9 s	5
73	453	51	9 s	5
75	403	2133007	6 h 23 min	5
76	290	90592	27 min 27 s	5
78	154	8	2 s	5
79	176	204	11 s	5
80	235	344	10 s	5
81	185	2604	40 s	5
82	173	66	4 s	5
83	202	1040	19 s	5
84	161	695	52 s	10

Tableau 6.1 – Problèmes du benchmark résolus par notre programme.

Chapitre 7

Perspectives

Nous introduirons dans ce chapitre quelques pistes plus ou moins concrètes que nous avons envisagées et qui nous semblent intéressantes à explorer afin d'améliorer les performances de notre programme.

7.1 Réarrangement des goals

Au cours de nos recherches, nous nous sommes aperçu qu'un certain nombre de problèmes pour lesquels notre méthode échouait à trouver une possibilité d'ordonnancement des goals, devenaient accessibles et parfois facilement solubles par notre programme à condition de réarranger préalablement la disposition des goals. En effet, la résolution de ces problèmes impose que certaines pierres ne soient pas directement amenées vers leur position définitive mais soient d'abord placées dans une position temporaire afin de ne pas bloquer l'accès à certains goals. Un exemple d'un tel réarrangement est donné dans la figure 7.1.

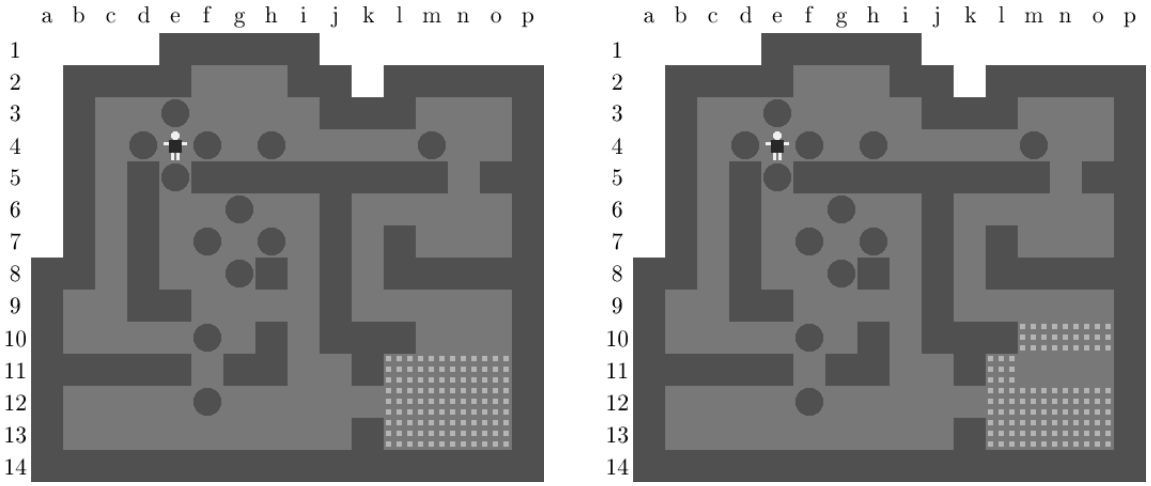

Figure 7.1 – Réarrangement des goals (à droite) dans le problème 87 (à gauche).

Il est aisé de se convaincre que l'obtention d'une solution au problème de départ à partir de la solution trouvée au problème réarrangé est presque immédiate. Dès lors, la conception d'une procédure permettant d'automatiser ce réarrangement nous permettrait d'ajouter 7 problèmes à la liste des problèmes résolus par notre programme. Celui-ci serait dès lors capable de résoudre 61 des 90 problèmes de notre benchmark et d'ainsi rivaliser avec le meilleur solveur actuel ayant fait l'objet d'une publication (59 problèmes résolus). Le tableau 7.1 présente les données relatives à la résolution de ces 7 problèmes supplémentaires.

Problème	**Poussées**	**Noeuds explorés**	**Temps**	**Limite de profondeur (sous-problèmes)**
35	394	168	21 s	5
39	688	1646578	5 h 50 min	10
44	177	736375	2 h 14 min	10
46	277	53584	14 min 31 s	10
47	208	264	5 s	5
86	134	143	2 s	5
87	250	80	4 s	5

Tableau 7.1 – Problèmes du benchmark résolus en utilisant le réarrangement des goals.

Un réarrangement valide devra posséder les deux caractéristiques suivantes : (1) il devra être possible de trouver un ordonnancement des goals pour le problème réarrangé ; (2) en partant du problème réarrangé résolu, il devra être possible de trouver une séquence de poussées qui permettent d'atteindre la situation de jeu dans laquelle le problème initial est résolu.

Il est intéressant de noter que la procédure de réarrangement s'insérerait naturellement dans le processus de résolution adopté par notre programme. Elle consisterait, en effet, à ajouter un niveau de planification supplémentaire et prendrait la forme d'un quatrième agent chargé de réarranger les goals du problème initial (cf. figure 7.2). Le processus de résolution ainsi augmenté pourrait dès lors être décrit ainsi :

- Si un ordonnancement des goals peut être trouvé pour le problème initial, le nouvel agent passe cet ordonnancement ainsi que la situation de jeu initial à l'agent ayant pour tâche de remplir les goals dans l'ordre défini par l'ordonnancement. La solution trouvée est ensuite renvoyée comme solution globale du problème. Dans ce cas, l'introduction du nouvel agent ne modifie donc pas le processus de résolution du problème.
- Si aucune possibilité d'ordonnancement n'est trouvée, le nouvel agent réarrange les goals de façon adéquate, *i.e.* de façon à ce que le réarrangement utilisé possède les deux caractéristiques que nous avons décrites. Il passe ensuite la situation de jeu réarrangée ainsi que l'ordonnancement correspondant à l'agent chargé de remplir les goals, qui lui retourne la solution trouvée. Cette solution n'est que partielle puisqu'elle permet de résoudre le problème réarrangé et non le problème initial. Afin de compléter cette solution partielle, le nouvel agent génère une situation de jeu dans laquelle la disposition des goals est celle de la situation de jeu initiale et dans laquelle les pierres occupent les positions des goals réarrangés. Il passe cette situation de jeu ainsi que son ordonnancement à l'agent chargé de remplir les goals. Pour reconstruire la solution globale, il lui suffit finalement de concaténer les deux solutions partielles obtenues. Il est également envisageable que la séquence de poussées permettant de passer du problème réarrangé résolu à la résolution du problème initial ait été obtenue lors

du processus de réarrangement des goals. Dans cette hypothèse, le deuxième appel à l'agent chargé de remplir les goals devient inutile et la solution globale peut être obtenue immédiatement.

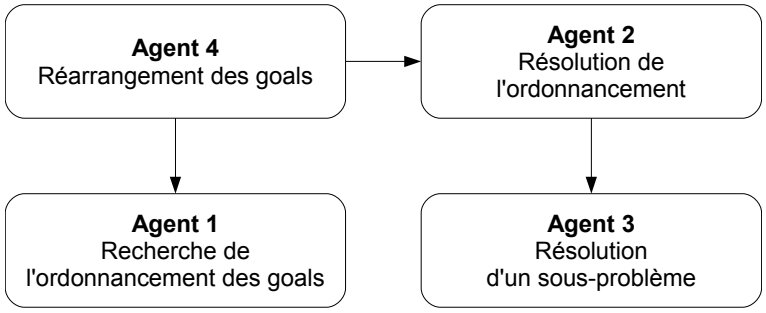

Figure 7.2 – Intégration d'un agent supplémentaire chargé de réarranger les goals.

7.2 Gestion de plusieurs entrées

Le principal problème qui limite actuellement les performances de notre programme est son incapacité à gérer convenablement la présence de plusieurs entrées et par conséquent la présence de plusieurs zones de goals dans un problème de Sokoban (cf. figure 7.3).

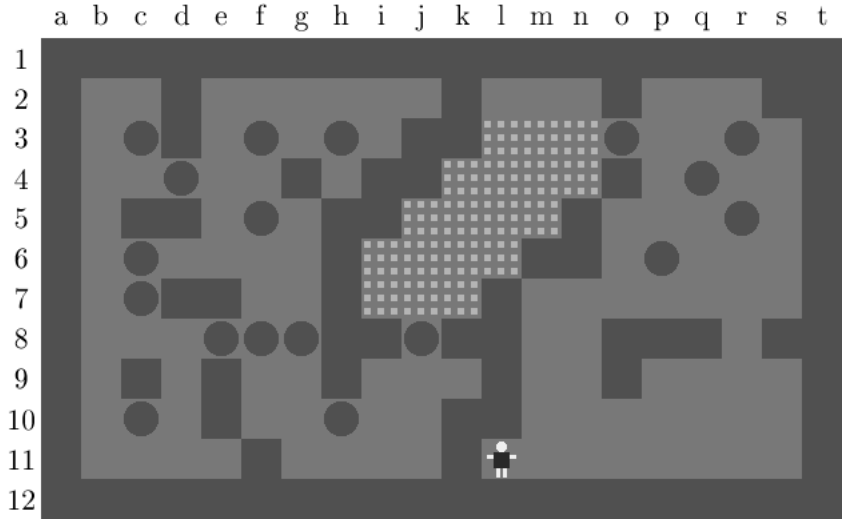

Figure 7.3 – Problème qui met en défaut notre méthode.

En effet, si un problème comporte plusieurs entrées, l'analyse préalable du tableau de jeu ne permet pas de déterminer a priori un ordonnancement qui corresponde avec certitude à l'ordre dans lequel les goals du problème pourront être remplis. Cette impossibilité est la conséquence du fait que cette analyse préalable ne permet pas de connaître à l'avance l'entrée par laquelle la pierre suivante rentrera dans une zone de goals. Or, tous les goals ne peuvent pas nécessairement être atteints à partir de n'importe quelle entrée et la modification de la position des pierres au cours de la résolution du problème peut rendre

inaccessibles certaines entrées. La méthode d'ordonnancement des goals utilisée par notre programme ne prend pas en compte ces différentes possibilités et pose un choix arbitraire. L'ordonnancement obtenu n'est donc qu'un ordonnancement choisi arbitrairement parmi un ensemble d'ordonnancements possibles répondant aux critères pris en compte par l'algorithme d'ordonnancement que nous utilisons. Même si notre méthode se révèle satisfaisante pour un certain nombre de problèmes comportant plusieurs entrées, elle produit également des ordonnancements inadaptés dans le cas d'autres problèmes.

La solution que nous avons envisagée pour dépasser cette limitation est de ne plus fixer a priori un ordonnancement complet mais de recalculer à chaque étape du processus de résolution une liste de goals cibles possibles. Cette solution aurait pour conséquence d'étendre notre définition des sous-problèmes. En effet, ces derniers ne consisteraient plus à remplir un goal cible donné mais à trouver une séquence de poussées permettant de remplir un goal parmi un ensemble de goals possibles.

Soit une situation de jeu dans laquelle n goals ont déjà été remplis. L'analyse de cette situation de jeu nous permettrait de déterminer l'ensemble des entrées à partir desquelles une pierre serait susceptible de pénétrer dans une zone de goals et dès lors un ensemble de goals pouvant être remplis à partir de ces entrées. Cet ensemble de goals représenterait dès lors les différentes possibilités de remplir le $(n+1)^{\text{ième}}$ goal du problème. En plus de permettre la prise en compte des différentes possibilités dues à la présence de plusieurs entrées, un apport important de cette méthode serait que cet ordonnancement, que l'on pourrait qualifier de dynamique, tiendrait compte de la modification de la position des pierres induite par la résolution des sous-problèmes précédents.

7.3 Heuristique pour les sous-problèmes

Les méthodes de résolution que nous avons mises en oeuvre dans notre programme nous ont permis de réduire la profondeur et le facteur de branchement de l'arbre de recherche. Néanmoins, pour certains problèmes, ce facteur de branchement reste très élevé, et demeure donc un obstacle. C'est notamment le cas lorsque la phase de dégagement nécessaire à la résolution d'un sous-problème, *i.e.* au remplissage d'un goal cible, est particulièrement longue. Pour rappel, la phase de dégagement est la partie de la séquence de poussées solution d'un sous-problème qui ne peut être réduite à une macro-poussée. Par exemple, dans le problème 30 de notre benchmark, la phase de dégagement relative à la résolution du deuxième sous-problème comprend environ 35 poussées.

Il serait dès lors intéressant d'explorer les possibilités d'une méthode de résolution des sous-problèmes basée sur une recherche informée. On peut en effet espérer qu'une heuristique destinée à orienter la résolution d'un sous-problème soit plus simple et moins coûteuse qu'une heuristique ayant pour objectif la résolution du problème complet.

7.4 Améliorations spécifiques au domaine

Finalement, l'intégration de certaines améliorations spécifiques au domaine décrites dans [3] permettrait également d'améliorer les performances de notre programme. Nous avons déjà appliqué cette idée en ajoutant à notre programme le *Move Ordering* qui consiste à trier la liste des successeurs d'un noeud afin de privilégier les suites de poussées d'une même pierre. Les deux améliorations non encore intégrées qui nous semblent les plus intéressantes sont les *Deadlock Tables* et les *Tunnel Macros*.

Nous avons déjà présenté les *Deadlock Tables* dans le deuxième chapitre de ce travail. Cette amélioration consiste à précalculer, à déterminer et à stocker dans une base de données le statut, deadlock ou non, de toutes les configurations possibles d'une zone de taille définie. L'intégration de cette idée serait particulièrement intéressante afin d'améliorer l'efficacité de la détection des deadlocks induits par la dernière poussée et d'ainsi éviter à notre programme de se lancer inutilement dans l'exploration d'un sous-arbre issu d'une situation de jeu contenant une configuration deadlock.

Les *Tunnel Macros*, à l'instar de l'identification des deadlocks induits par une seule pierre, exploitent une information obtenue par une analyse préalable du tableau de jeu. Un tunnel est constitué d'un ensemble de positions contiguës, habituellement alignées, qui ne peuvent contenir plus d'une pierre simultanément sans qu'une situation de deadlock ne soit créée (cf. figure 7.4). L'identification de ces tunnels est intéressante à deux points de vue : (1) d'une part, elle permet d'éviter l'apparition d'un certain nombre de situations de deadlock en empêchant qu'un tunnel soit occupé par plus d'une pierre ; (2) d'autre part, elle permet de considérer l'ensemble des positions faisant partie du tunnel comme une position unique. En effet, la position exacte d'une pierre à l'intérieur d'un tunnel est sans importance. La seule information importante est de savoir si le tunnel est occupé ou non par une pierre. Cette réduction du nombre de positions potentiellement occupées par des pierres a pour conséquence bénéfique de réduire la taille de l'arbre de recherche.

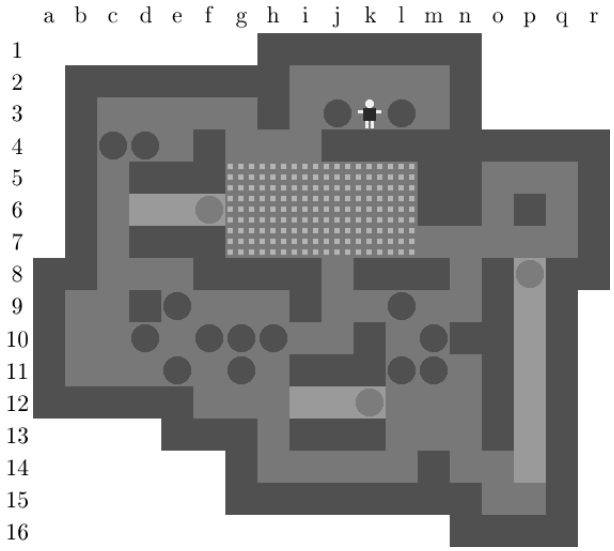

Figure 7.4 – Identification des tunnels présents dans le problème 14 du benchmark.

Chapitre 8

Conclusion

Ce travail avait pour objectif la conception d'un programme capable de résoudre des problèmes de Sokoban.

Nous avons d'abord montré comment un problème de Sokoban pouvait être abordé comme un problème de recherche dans un arbre. Les problèmes de recherche sont courants en intelligence artificielle et nous avons présenté un certain nombre d'algorithmes classiques destinés à les résoudre. Leur implémentation nous a permis de démontrer leur inefficacité face à la taille gigantesque des arbres de recherche rencontrés lors de la résolution de problèmes de Sokoban non triviaux.

En partant de notre propre expérience du jeu et de l'observation de nos propres comportements de résolution, nous avons ensuite développé une méthode de résolution originale basée sur la planification par la décomposition en sous-problèmes. Nous avons décrit une telle décomposition et démontré son intérêt pour une classe particulière de problèmes de Sokoban : les problèmes pour lesquels l'ordre dans lequel les goals doivent être remplis peut être déterminé à l'avance. L'exploitation de cette idée nous a conduit à la mise au point d'un protocole de résolution dont le fonctionnement conceptuel pouvait être décrit par l'interaction de trois agents de recherche : (1) le premier agent était chargé de générer un ordonnancement des goals à partir de l'analyse de la situation initiale du problème ; (2) le deuxième agent avait pour tâche de remplir les goals du problème dans l'ordre défini par l'ordonnancement ; (3) enfin, le rôle du troisième agent était de trouver une séquence de coups permettant de remplir un goal cible à partir d'une situation de jeu donnée.

La troisième étape de notre cheminement a été d'intégrer à notre programme un concept élémentaire d'apprentissage. En analysant ses erreurs et en conservant les informations récoltées dans une base de données, notre programme a été capable d'exclure de l'arbre de recherche un grand nombre de situations de jeu. Cette deuxième idée nous a permis d'arriver à résoudre 54 problèmes d'un benchmark classique de 90 problèmes difficiles. L'implémentation en Scheme de cette version de notre programme est disponible à l'adresse : `http://www.student.montefiore.ulg.ac.be/~demaret/tfe/`.

Nous avons finalement ouvert quelques pistes susceptibles d'améliorer les performances de notre programme. L'une d'elle, que nous avons appelée réarrangement des goals, s'avère particulièrement prometteuse. D'une part, elle s'intègre naturellement dans le protocole de résolution hiérarchique que nous avons mis en place. D'autre part, nous avons montré qu'elle permet à notre programme de résoudre 61 problèmes du benchmark et d'ainsi afficher des performances comparables à celles du meilleur solveur actuel ayant fait l'objet d'une publication (59 problèmes résolus).

Bien que nous soyons encore loin des performances de certains solveurs japonais qui restent malheureusement encore non documentés, la relative simplicité des idées mises en oeuvre dans notre programme laisse entrevoir une importante marge de progression, que ce soit par l'exploration des idées nouvelles que nous avons esquissées ou par l'implémentation d'améliorations spécifiques au domaine.

Bibliographie

[1] F. Van Lishout. *Single-player games: Introduction to a new solving method.* Université de Liège, 2006.

[2] F. Van Lishout et P. Gribomont. *Single-player games: introduction to a new solving method combining classical state-space modelling with a multi-agent representation.* Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'2006), pp 331-337, Namur, 2006.

[3] A. Junghanns. *Pushing the limits: New developments in single-agent search.* PhD thesis, University of Alberta, Edmonton, Canada, 1999.

[4] A. Botea, M. Müller and J. Schaeffer. *Using Abstraction for Planning in Sokoban.* University of Alberta, Edmonton, Canada, 2003.

[5] P. Gochet et P. Gribomont. *Logique, volume 3 : Méthodes pour l'intelligence artificielle.* Hermès, Paris, 2000.

[6] Y. Murase, H. Matsubara and Y. Hiraga. *Automatic making of sokoban problems.* In Pacific Rim Conference on AI, 1996.

[7] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu and D. Szafron. *A World Championship Caliber Checkers Program.* Departement of Computing Science, University of Alberta, Canada.

[8] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[9] S. Tanimoto. *The Elements of Artificial Intelligence: An Introduction Using LISP.* Computer Science Press, 1987.

[10] R. Kelsey, W. Clinger and J. Rees (eds.). *Revised^5 Report on the Algorithmic Language Scheme.* Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998.

[11] P. Gribomont. *Eléments de programmation en Scheme.* Dunod, Paris, 2000.

[12] J. Chazarain. *Programmer avec Scheme - De la pratique à la théorie.* International Thomson Publishing France, Paris, 1996.

[13] R.E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence*, 26(1):35-77, 1985.

[14] R. Fikes and N. Nilsson. *STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence*, 2:189-208, 1971.