University of Liège

Faculty of Applied Sciences
Department of Electrical Engineering and Computer Science
Montefiore Institute

Single-Player Games: Introduction to A New Solving Method

Combining Classical State-Space Modelling with a Multi-Agent Representation

Academic year
2005-2006

DEA in Applied Sciences
presented by
Van Lishout François

Acknowledgements

First of all, I would like to express my deepest gratitude to my supervisor, professor Pascal Gribomont, head of the Artificial Intelligence Research Unit of the University of Liège. From the very beginning of our collaboration, he gave me the freedom to work in my favorite sub-field of the Artificial Intelligence, the game programs. He also found, again and again, very pertinent critics to my ideas and developments. They all contributed to an improvement of my work.

I am very grateful to the staff of the Electrical Engineering and Computer Science Department of the University of Liège, for all the help given, as well for my study as for my research. A special thanks to my colleague and friend Samuel Hiard, particulary for the long hours that we have spent together talking about our respective research activities. For me, it was always the occasion to make an internal summary of my recent advances and sometimes to discover that they did not work ! Let me also thank professor P.A. de Marneffe for the precious advices given for the presentation of my algorithms, professor L. Wehenkel for the state-of-the-art literature he suggested me to consult, and professor B. Boigelot for its encouragements. Finally, let me thank the members of my jury for the time that they will spent on reading this pages...

My most affectionate thanks go to my family and my close relations. My mother, who was always there to listen to me in the moments of doubt. In the memory of my father, who learned me how to work with eagerness. My girl-friend, for her constant support. It was not always easy for her to see that she could not turn my attention off my work. My friends, for which I had to little time.

François Van Lishout
Liège
August 2006

Abstract

In many games, the machine has become stronger than the best human players. Machines have already beaten the human World Champion in famous games like Checkers, Chess, Scrabble and Othello. However, mankind has not been humbled by chips in all games. The best human players are still stronger than computers in games like Go, Poker, Chinese Chess and Hex. In this thesis, we will focus on a new way to model single-player games in order to improve the performances of the machine.

Classically, games are described as search problems. Each game situation is considered as a node in a graph. The arcs represent legal moves from one position to another. Solving a single-player game requires to solve a state-space problem, i.e. find one path that leads to a solution-state of the game. The heart of this thesis consists in exploring the idea that most single-player games can also be modelled as multi-agent systems. The agents are no longer the players of the game as for multi-player games, but primitive game elements depending on the particular game. Furthermore, instead of facing each other, the agents collaborate to achieve a common objective. This new representation leads to new interesting resolution techniques, even more when both modelling methods are combined. It is demonstrated on the game of Sokoban, a challenging one-player puzzle for which mankind still dominates the machine.

For now, the best documented Sokoban solver called *Rolling Stone* uses single-agent search techniques with a lot of problem-dependent improvements, and is able to solve 59 problems of a difficult 90-problem test suite [8]. In [10], the fact that only trivial problems can be solved by using classical state-space techniques without other enhancements is demonstrated. Our program *Talking Stones* solves already 9 mazes of the same benchmark without any problem-dependant enhancement.

This thesis will also give an original presentation of the classical state-space algorithms. Usually, only a high-level description with abstract data types is given. Here we will present all the algorithms with the semi-formal-method proposed in [4]. The invariants specified in this work have not been taken from the literature. They have all been reconstructed from the original idea of the algorithm in order to produce a clear description of the latter and to prove its correctness. This approach has led to a contribution for the A* algorithm. The practical performances have indeed been improved for a particular implementation choice of practical interest.

Contents

Acknowledgements **i**

Abstract **ii**

Contents **iii**

List of Figures **iv**

1 Introduction **1**
1.1 State-Space Representation 1
1.1.1 Single-Player vs Multi-Player Games 2
1.1.2 Size of the Search Space 2
1.1.3 Search Strategies 2
1.2 Multi-Agent Systems 3
1.3 The New Modelling Method 3
1.3.1 The Game of Sokoban 4
1.3.2 Classical State-Space Techniques 6
1.3.3 Generalization of the Method 6
1.4 Overview of the Rest of the Document 7

2 Single-Agent Search **8**
2.1 Framework Presentation 8
2.1.1 The State-Space 8
2.1.2 Useful Functions and Procedures 9
2.2 Graph Search is Really Tree Search 10
2.2.1 Acyclic Graphs 10
2.2.2 Cyclic Graphs 11
2.3 Blind Methods 12
2.3.1 Depth-First Search 12
2.3.2 Depth-First Search with Cycle Detection 16
2.3.3 Depth-Limited Depth-First Search 19
2.3.4 Iterative Deepening 21
2.3.5 Breadth-First Search 22
2.3.6 Efficiency Analysis 24
2.4 Heuristically informed methods 26
2.4.1 The A* algorithm 27
2.4.2 The IDA* algorithm 40

3 The Game of Sokoban **44**
3.1 Rules and Consequences 45
3.1.1 Notation 45
3.1.2 Difficulty of the game 45
3.1.3 Optimality 47
3.1.4 Efficient Representation of a Maze 47
3.2 State of the Art 48

4 The New Multi-Agent Modelling Approach **49**
4.1 Solving a Particular Subclass of Problems 49
4.1.1 Definition of the Subclass 49
4.1.2 Protocol for Solving the Mazes of the Subclass 51
4.1.3 Efficient Implementation of the Protocol 52
4.1.4 Results 60
4.2 Embedding the New Approach into a State-Space Algorithm 61
4.2.1 Choosing the Right State-Space Algorithm 61
4.2.2 Results 61
4.2.3 Comparison with *Rolling Stone* 62
4.3 Generalization of the Method to Other Games 63
4.4 Extensions of the Method 63

Conclusion **65**

Scheme Implementation **67**

Bibliography **94**

List of Figures

1.1 Portion of the state space for Connect-Four 1
1.2 A Sokoban maze and a particular solution 4

2.1 The finite search tree corresponding to an acyclic graph 10
2.2 The infinite search tree corresponding to a cyclic graph 11
2.3 The finite tree of all the possible acyclic paths of a cyclic graph 11
2.4 Invariant for depth-first search 13
2.5 Invariant for depth-first search with cycle detection 16
2.6 Invariant for breadth-first search 23
2.7 Construction of the heuristic estimate $f(n)$ used by the A* algorithm. . 28
2.8 Invariant for the A* algorithm 30

3.1 The last maze of our 90-problem benchmark 44
3.2 Examples of deadlocks 46

4.1 A Sokoban maze that is solvable stone-by-stone 50
4.2 A Sokoban maze that is not solvable stone-by-stone 50

Chapter 1

Introduction

Games have always fascinated mankind and attracted the attention of the AI research community. Writing game-playing programs is not just a diverting activity, it also has many applications in real-life problems. Indeed, when facing a problem humans often act as if they would be playing a game: they consider a number of different *moves* on their way to solve the problem. Some people consider even the life as a big game, where every alive being tries to maximize his well-being.

1.1 State-Space Representation

Classically, games are described as *search problems*. As an example, consider the game of Connect-Four. Given any game configuration, there is a finite number of moves that a player can make. Each of them leads to another position, which will allow the opponent a finite number of responses, and so on until it ends with a win or a tie. We can therefore represent the game of Connect-Four by considering each board situation as a *node* in a graph. The *arcs* in the graph represent legal moves from one board situation to another. These nodes correspond thus to different *states* of the game board. The resulting structure is called a *state-space graph*.

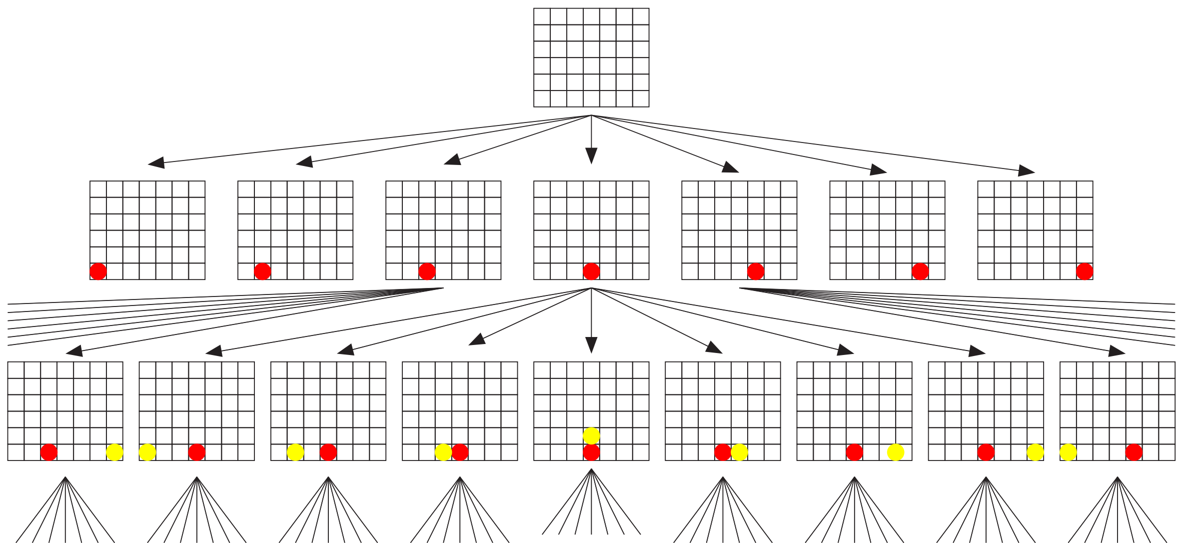

Figure 1.1: Portion of the state space for Connect-Four

A *path* between two nodes of a graph is a sequence of arcs, such that the end point of any arc is the beginning point of the following one in the sequence. The problem of finding a particular path in a state-space graph is called a *state-space problem.* Figure 1.1 shows that starting the construction of the state-space graph with an empty board will lead to a graph representing all the possible games of Connect-Four.

1.1.1 Single-Player vs Multi-Player Games

Solving a single-player game requires to solve one state-space problem. Indeed, the game starts in a particular state and the program has just to find one path that leads to a *goal node*, i.e. a solution state of the game. The moves to play are then given by this path. This document mainly focuses on single-player games. As we will see, even if only one state-space problem needs to be solved, this will be far from trivial for interesting games.

In two-player games, the problem is not as easy, as finding one path that leads to a win does not really help. Indeed, this path will contain positions where it is the opponent's turn. He will therefore have the freedom to decide what to play, not necessarily the moves of our winning path. In practice, two-player games are attacked with the minimax algorithm, but this is not covered in this document focused on single-player games [1]. The problem becomes even harder when the number of opponents grows.

1.1.2 Size of the Search Space

Obviously, the size of the search space plays a crucial role for solving state-space problems. Intuitively, the depth of the solutions is also an important factor. The size of the search space of Connect-Four depicted in Figure 1.1, has been estimated at 10^{14} [1]. It seems thus impossible to visit all states in reasonable time. We need more ingenious search strategies.

The problem of the size of the state-space graph is not specific to the game of Connect-Four. Indeed, all the interesting games are characterized by a huge search-space. A few more two-player examples are the game of Chess, where the state-space has been estimated at 10^{120}, which is a number larger than the number of molecules in the universe or the number of nanoseconds that have passed since the big bang [12], and the game of Go, where the size is even larger than for Chess, i.e. 10^{170} for the classical 19×19 board [16]. A Single-Player example is the game of Sokoban, which will be the main focus of this document, and where the state-space is a cyclic graph estimated at 10^{98} [10].

1.1.3 Search Strategies

Many different algorithms have been created for exploring search spaces. They can be separated in two complementary groups:

[1] for more information about the minimax algorithm and its variants consult for example [2].

- Blind methods, or uninformed search methods, do not use any problem-dependent information to guide through the search. They explore the state-space in a predefined way that is the same for all state-space problems.
- Heuristically informed methods use *heuristics* to determine in which order the different paths should be analyzed, i.e. to explore the most promising ones first.

A *heuristic* is a problem-dependent set of expert-rules for selecting lines that have a high probability of success. Heuristics are not foolproof: even the best game strategy can be defeated. For Connect-Four, the heuristic could for example be a function that favors paths leading to board configurations where the player's tokens are strongly connected together and the opponent's ones are dispersed all around the board.

The most popular strategy that uses heuristics is called the *best-first search* strategy. Humans use this natural strategy every day. For example, when planning to visit an old friend in another town, humans do not take the charts of all the existing roads of the world and explore all the possible combinations of roads until they found the best one leading to the desired destination. Instead, they use their experience to find a near-optimal route. When humans play games, they do not consider all possible moves in every possible position: they examine only moves that experience has shown to be effective. So heuristics can be seen as models of the experience of a problem.

Nevertheless, for some difficult games, machines are still far from beating humans and the research community mainly tries to do it with the existing modelling methods, focusing on new resolution methods. This document will instead introduce a new method, which seems to have never been tried till now. Actually, the main focus of this document will consist in proving that this method may lead to new interesting techniques and testing it on the game of Sokoban.

1.2 Multi-Agent Systems

According to [15], for game theorists a game is an abstraction of a situation where players, or agents, interact by making moves. Based on the moves made by the players, there is an outcome, or payoff, to the game. Standard games such as Poker and Chess are games in this sense. The game theorist's notion of game, however, encompasses far more than what we commonly think of as games. Standard economic interactions such as trading and bargaining can also be viewed as games, where players make moves and receive payoffs.

1.3 The New Modelling Method

The heart of this thesis consists in exploring the idea that most single-player games can also be modelled as multi-agent systems. The agents are not, as usually, the players of the game, but more primitive game elements depending on the particular game. This leads to new interesting resolution techniques. This means thus that it will even be possible to solve single-player games using this multi-agent modelling approach.

1.3.1 The Game of Sokoban

The new game-modelling method will be demonstrated on a particular game: the game of Sokoban. This choice is not random. First, it is a one-player puzzle which has not been solved yet by the AI community. For now, the best documented solver called *Rolling Stone* uses single-agent search techniques with a lot of problem-dependent improvements, and is able to solve 59 problems of a difficult 90-problem test suite [10]. Furthermore, man still dominates machine in this domain.

Second, an unlimited set of different starting positions can be created by varying the size and the difficulty of the component problems. Different solving methods are therefore easy to compare, as there will always exist a test-set that can highlight the limits of the solving strategies. In this paper, we will compare our results to *Rolling Stone*'s ones on the same 90-problem benchmark. The latter can be found at [6] with their results. Chapter 3 will be dedicated to this game, explaining notably why it is so challenging. Let us just give the simple rules of the game yet, in order to be able to explain the contributions of this document in the domain.

A Sokoban maze is a grid composed of unmovable walls, free squares, exactly one man, and as many stones as goal squares. The player controls the man and the man can only push stones (not pull). Furthermore, only one stone can be pushed at a time. The objective of the game is to push all stones on goal squares.

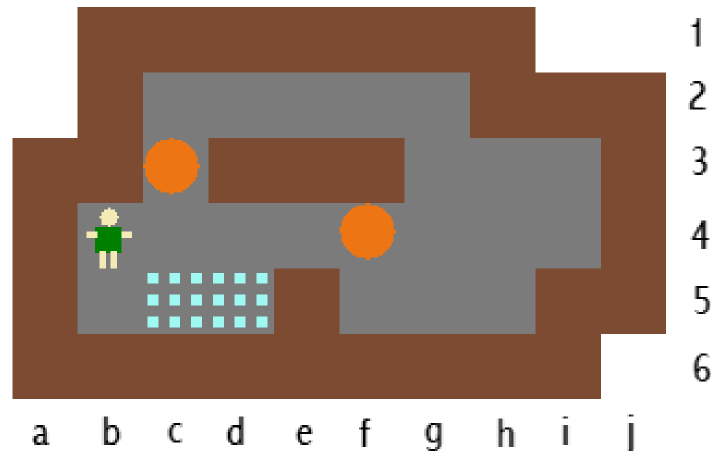

Figure 1.2: A Sokoban maze and a particular solution: b4-c4-d4-e4-f4-g4-g3-g2-f2-e2-d2-c2-c3-c4-b4-b5-c5-c4-d4-e4-f4-g4-g3-h3-i3-i4-h4-g4-f4-e4-d4-e4-f4-g4-g3-g2-f2-e2-d2-c2-c3-c4. The corresponding stone-moves notation of the solution: f4-g4-h4, c3-c4-c5-d5, h4-g4-f4-e4-d4-c4-c5.

For a better understanding of the game, a solution to the problem of Figure 1.2 is given. It is first described as an ordered list of coordinates of the squares that the man must pass by, to bring all the stones on goal areas (starting with the initial square of the man). A shorter notation giving only the lists of the stone-moves is also provided. Note that this notation makes the implicit hypothesis that each stone-move of the solution is valid, i.e. that the man can reach the square adjacent to the stone to effectively make the push.

The idea of the new multi-agent representation is that every stone of the maze can be seen as an agent whose aim is to reach one of the goal squares, and the global goal is to find a solution for which everyone achieves his objective. It is also possible to impose some kind of optimality like minimizing the global number of agent moves. In this view of the problem, the man is only a puppet which can be called by the stones when they want to be pushed. It is important to note that the multi-agent notion which has been introduced is only conceptual and that it does not imply multi-agent programming. We will write an algorithm which requests a central solver to decide the order of the stones to be solved. This work presents thus solely a new way to view the problem which leads to new interesting resolution ideas.

We will define a particular subclass of Sokoban mazes that has been completely solved by a protocol based on our pure multi-agent representation. Intuitively, a maze is in this class if the stones are solvable one by one. Only one problem of our 90-problem benchmark is in the subclass (problem 78) and can be solved by this protocol. This is not surprising, as problems that are directly solvable stone by stone are uncommon in difficult benchmarks. Instead of elaborating other protocols for solving more general problems with the pure new multi-agent modelling approach, another idea has been developed.

As it is often the case in AI, trying to understand how the human player solves problems helps to find new algorithms. In the case of Sokoban, one of the talents of the human player consists in recognizing very soon in the resolution process that he can reach a configuration that is easy to solve (stone-by-stone). This suggests a new solving method for difficult games.

The method consists in using a classical state-space algorithm, but one in which the nodes whose corresponding state of the game is solvable by the new multi-agent modelling approach are defined as success nodes. This means that when the search reaches such a node, the search terminates successfully. The solution is then obtained by appending the solution path found by the state-space algorithm to the solution found by the multi-agent modelling method.

The offspring of success nodes are no longer reachable and can be considered to have been pruned out of the state-space. In practice, the size of the state-space will decrease substantially. On the other hand, more computation time is needed at each node as the multi-agent modelling approach is called for each node to determine whether it is a success node. However, the time lost by these calls is largely compensated by the time won by having less nodes to visit. Our program *Talking Stones* implements this idea.

At the time being, our program *Talking Stones* is not a contribution in the domain of Sokoban. It solves only 9 mazes of the benchmark whereas the state-of-the-art program *Rolling Stone* solves 59. However, the latter is based on the IDA* algorithm with a lot of really interesting problem-dependent enhancements. These are presented in [10] and it is well explained why each of them contribute to a substantial decrease of the search-tree size. On the other hand, the fact that no problem of the benchmark can be solved with the pure IDA* approach without these enhancements, even with a clever heuristic, is also demonstrated in [10]. We have not implemented these enhancements yet and we plan to inject them within our new method in future works. As *Rolling stone* has enjoyed tremendous progress by adding them to its initial pure IDA* approach (from 0 mazes solved to 59), we hope to benefit from the same kind of progression.

1.3.2 Classical State-Space Techniques

The classical state-space techniques can be reused in this approach, as tools that agents can use to solve subproblems of the game. For this purpose, the most popular best-first search algorithm will be intensively used: the *A* algorithm* [13, 14]. The latter will therefore be studied very carefully in this work. Note that even if the algorithm is presented in many AI books, it is often only presented at a rather high level and the fact that there exists a variety of possible practical implementations depending on the problem is often hidden [2].

This thesis will give an original presentation of the A* algorithm in particular and more generally of all the classical state-space algorithms. Usually, only a high-level description with abstract data types is given. Here we will present all the algorithms with the semi-formal-method proposed by [4]. The invariants specified in this work have not been taken from the literature. They have all been reconstructed from the original idea of the algorithm in order to produce a clear description of the latter and to prove its correctness. This approach has even led to a contribution for the A* algorithm. For one particular implementation choice for the abstract objects, we have discovered that it is possible to rewrite the algorithm in order to improve the practical performances.

1.3.3 Generalization of the Method

We can decompose the new solving method for difficult single-player games in three layers:

- The high-level layer is a classical state-space algorithm where a node is a success node if the medium layer can solve it. The choice of the algorithm depends on the characteristics of the game.
- The medium-level is a protocol based on a multi-agent representation of the game. The agents are primitive game elements depending on the particular game. The agents have to communicate together to find a common solution. They can use the algorithms of the low-level as tools for solving subproblems.
- The low-level is a set of algorithms for solving subproblems of the game. Classical state-space algorithms can be used but not exclusively.

We believe that it is possible for almost all games to determine primitive game elements that have to reach some goal. In puzzles like the 24-tile puzzle, the agents could be defined as the tiles. In this representation, each tile aims to reach its final destination but cannot move without altering the position of other agents. In the game of Sokoban, all the agents are instances of stones of the maze and have thus the same characteristics. For other games however, we could define agents that have their own personality. For the game of solitaire for example, the agents could be the 52 cards. Each agent is now unique. Note that for such imperfect information game, we must consider that only a subset of the agents is visible. The other agents can thus be seen as being in an unknown queue, waiting for entering into play.

[2]That does not mean that the well-known variations of the A* algorithm are often hidden. It means merely that the best way to implement the A* algorithm itself is not often mentioned.

1.4 Overview of the Rest of the Document

Chapter 2 examines the most used search strategies. First, blind methods algorithms will be presented: the depth-first search and its variants, the breadth-first search and the iterative deepening depth-first search. Then, heuristically informed methods will be studied: the A* and the IDA* algorithm. Usually, only a high-level description with abstract data types of all these algorithms is given. Here we will present all the algorithms with the semi-formal-method proposed by [4]. The invariants specified in this work have not been taken from the literature. They have all been reconstructed from the original idea of the algorithm in order to produce a clear description of the latter and to prove its correctness.

Our new solving method will be demonstrated on the game of Sokoban. Chapter 3 introduces this game and give several arguments to show why this game is so challenging. The precise definition of optimal Sokoban-solutions that we have chosen will than be given. We will justify briefly why we have chosen this definition instead of another also commonly chosen one. This chapter will also contain a short overview of the state-of-the-art in Sokoban programs.

Chapter 4 is dedicated to our new solving method applied to the real case of Sokoban. We will first define a particular subclass of Sokoban problem which have been completely solved by our multi-agent modelling approach. We will see that this class is too particular to be interesting for itself. However, we will see that it is possible to combine the iterative deepening algorithm that will be presented in chapter 2 with our new method and achieve better results. We will than give a generalization of the method and sketch how it could be used for other single-player games. Finally, different ways to extend the method will be suggested for future works.

Chapter 2

Single-Agent Search

This chapter will describe classical search strategies commonly used for solving state-space problems. Deliberately, the most important strategies have been selected, letting other approaches in the dark. For more information, consult the references used in this chapter: [2, 7, 12, 17]. In this document, all the algorithms will be presented with the semi-formal method and the pseudo-code proposed by [3, 4].

2.1 Framework Presentation

2.1.1 The State-Space

Recall that a state-space problem consists in finding one path between a starting node and a goal node of a state-space graph. If the graph models a game, the states represent positions of the game board. The states are obviously problem-dependent. It must thus be supposed that the following data type is given:

type STATE = "*problem-dependent*";

The state-space graph is of course not present in memory. The space needed for stocking it would be too large for interesting problems. In fact, it is not necessary: each state-space algorithm starts with a beginning node and uses a function to generate the successors of the nodes that it wishes to expand. Typically, it is a function that "knows" the rules of the game and can compute all the possible moves in a given situation, obtaining the list of all the possible one-move modifications of the board. The following data type can thus be defined and the following function must be provided by the application:

type SUCCESSORS = list of STATE; [1]

find-successors(s: STATE): SUCCESSORS;
⇒ function that returns the list of the successors of the state s.

[1] list can be implemented as a linked list. It would take us too far away from the purpose of this document to explain in details how this can be done. We will restrict ourselves to specifying useful functions and procedures for manipulating lists. For a detailed implementation of linked lists, consult [3]. Note just that in this implementation, the constant *nil* is used to represent the empty list and that we will use the same convention here.

Our goal is to find a solution path, i.e. an ordered list of the states through which the game must evolve to reach a goal node. The different state-space algorithms will in fact manipulate lots of paths before finding the target one. This will be an essential data type of our framework:

type PATH = list of STATE;

Finding a goal node requires to know what a goal is. It depends of course of the problem studied. The following function must therefore be given:

goal-node?(s: STATE): boolean;
⇒ predicate true if and only if s is a goal state.

2.1.2 Useful Functions and Procedures

As lists will be intensively used by all search strategies, let us directly define useful functions and procedures that manipulate them, supposing that ELEM is the type of the elements of the list:

- initialize-list!(var l1: list of ELEM);
  ⇒ procedure that initialize the list $l1$ as an empty list.
- insert-front!(x: ELEM; var l1: list of ELEM);
  ⇒ procedure that inserts the element x in front of the list $l1$.
- insert-back!(x: ELEM; var l1: list of ELEM);
  ⇒ procedure that inserts the element x at the end of the list $l1$.
- get-first(l1: list of ELEM): ELEM;
  ⇒ function that returns a copy of the first element of the non-empty list $l1$.
- remove-first!(var l1: list of ELEM): ELEM;
  ⇒ function that removes the first element of the non-empty list $l1$ and returns this element.
- append(l1, l2: list of ELEM): list of ELEM;
  ⇒ function that returns the concatenation of the lists $l1$ and $l2$.
- empty?(l1: list of ELEM): boolean;
  ⇒ predicate true if and only if $l1$ is the empty list.
- member?(x: ELEM; l1: list of ELEM): boolean;
  ⇒ predicate true if and only if x is an element of $l1$.

A constant for the empty list is also useful:

const EMPTY_LIST = *nil*;

2.2 Graph Search is Really Tree Search

In this chapter, strategies for solving state-space problems in state-space trees will be given. In this section, the fact that graph search unfolds into tree search so that some nodes are possibly duplicated will be demonstrated.

As explained in the previous section, the graph is not present in memory. Each search strategy starts with the initial state of the problem and use the function *find-successors* for generating the successors of previously generated nodes. The order in which they are generated and removed from memory depends on the particular algorithm. To be more precise, nodes are generated for expanding previously generated paths until a path leading to a goal node is found (the first path being the one-node path which contains only the initial state of the problem).

Conceptually, the different search strategies can therefore be seen as various ways to explore the tree of all the possible paths that can be constructed from the initial state of the graph. This construction is only virtual and will never be computed. It is just useful for explaining the algorithms and proving their correctness. This section will be organized in two complementary parts: the cases of acyclic and of cyclic graphs.

2.2.1 Acyclic Graphs

A *cycle* is a path that contains at least two nodes that represent the same state of the problem and that are therefore labelled with the same etiquette (we will call such nodes *clones*). Acyclic graphs are graphs that contains no cycle. The state-space graph of the game of Connect-Four presented in chapter 1 (Figure 1.1) was an example of such a graph. Indeed, after each move the number of chips increases and the same position can therefore not occur twice in the course of the game (but can be reached in several ways, i.e. by several paths).

For an acyclic graph, the number of possible paths is finite and the tree representing all the possible paths of the graph is thus finite too. Figure 2.1 shows an example of the conceptual construction.

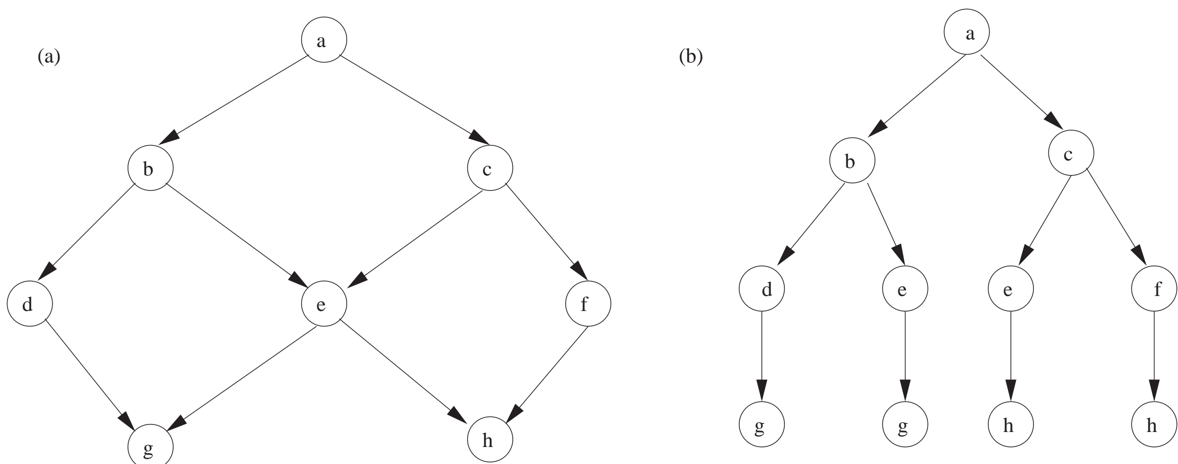

Figure 2.1: (a) An acyclic state-space graph: a is the start node.
(b) The tree of all the possible paths from a.

2.2.2 Cyclic Graphs

Cyclic graphs contains at least one cycle. The state-space of Sokoban is an example of such a graph. Indeed, the man can push a stone to the left, come back to push it on its starting square, and finally come back on its own starting square, attaining the same state of the game again.

For a cyclic graph, the number of possible paths is infinite and the tree representing all the possible paths of the graph is thus infinite too. Figure 2.2 shows an example of the conceptual construction.

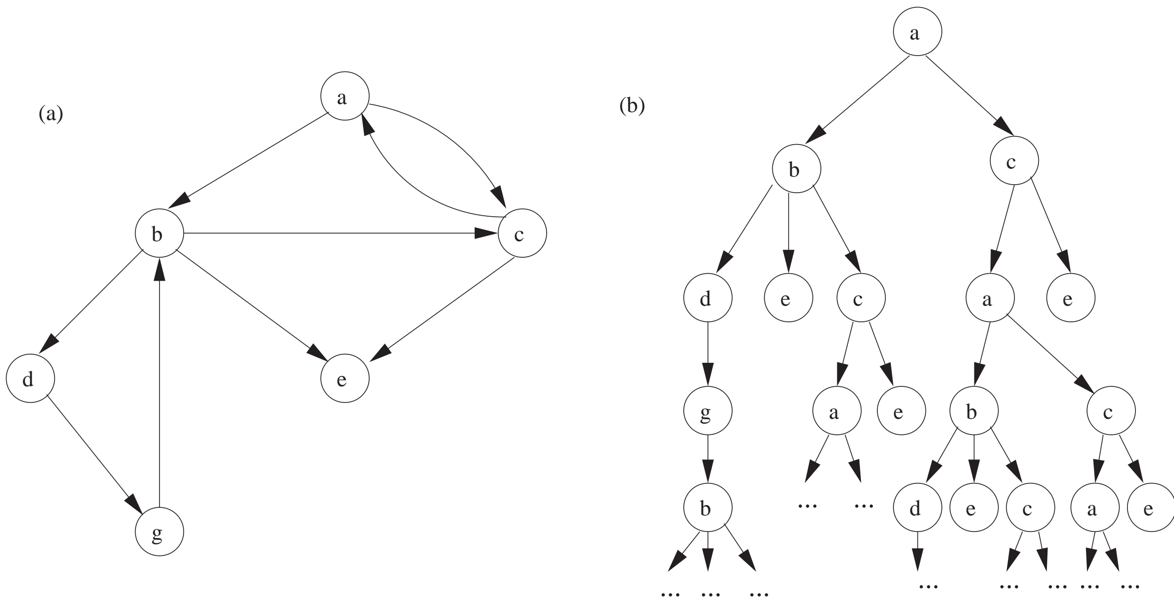

Figure 2.2: (a) A cyclic state-space graph: a is the start node.
(b) The infinite tree of all the possible paths from a.

The good news however is that the nodes (and in particular the goal nodes) of the graph have all at least one clone at a finite level of the tree. This can be deduced by the fact that the graph has only a finite number of acyclic paths that starts at the root node. A finite tree of those acyclic paths can thus be constructed, as showed on Figure 2.3 for the graph of Figure 2.2.

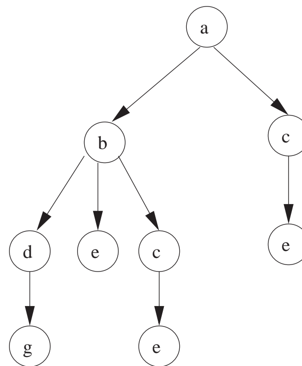

Figure 2.3: The tree of all the possible acyclic paths from a in the graph of Figure 2.2.

Each node of the graph appears exactly once in the tree, and the infinite tree containing also the cyclic paths is obviously only an extension of this tree. Algorithms that visit all the nodes at a particular depth before exploring deeper nodes will therefore terminate if a goal node exists. Algorithms with cycle detection (which avoid cycles) will only explore the tree of the acyclic paths and terminate thus even when no goal node exists. Some algorithms will however have no termination guarantee.

In the rest of the chapter, we will only consider the types of tree presented in this section. The trees can therefore be either finite or contain cyclic infinite branches. No trees with acyclic infinite branches will be considered (the only way to obtain such a tree would consist in starting with an infinite graph, but games are characterized by huge but finite graphs).

2.3 Blind Methods

This section presents algorithms for finding one path in a state-space tree, given no information about how to order the choices at the nodes.

2.3.1 Depth-First Search

The easiest way to explore the state-space tree consists in using a direct depth-first strategy. The idea of this search is simple. To find a goal node from a given node N, if N is already a goal node the search terminates, else pick one of the children of N and work forward from that node[2]. Other possibilities at the same level are memorized and will be explored only when there is no more chance of reaching a goal node using the original choice.

The idea of the algorithm consists in working with a list *open* of paths to expand (a stack is appropriate too). Initially, it contains only one path: the path between the initial state of the problem and itself. At each iteration, the first path of *open* is extracted. If this path ends with a goal state, a solution path is found. Else, new paths are created by extending the removed path to all successors of the terminal state, and the new paths are added to the front of the list. The algorithm terminates when the solution path is found (it is then the first element of *open*), or when *open* is empty (no solution). However, we will see that for some kind of state-space trees the algorithm will never reach one of those termination cases.

For efficiency reasons, the paths will not be represented as usually as the ordered list of its constituting nodes from the starting to the final one. Indeed, testing if the final node is a goal node would require time proportional to the length of the list. Therefore, the paths will be represented in the reverse order: from the final node to the initial one of the sequence. This leads to a constant time, as the final node is yet the first of the list and can be accessed immediately.

[2] In this chapter, we will suppose that the successors are picked from left to right. Note that this hypothesis is not constraining as the construction of the tree is only conceptual. The successors will be generated by the function find-successors and it can be decided where they will be placed in the virtual tree. Thus, this notion of left and right does only exist in the conceptual tree and not in the original problem.

2.3.1.1 Invariant

The invariant will be constructed using the following virtual coloring of the nodes:

- White nodes: nodes that have not yet been generated by the search.
- Blue nodes: nodes that have been generated but not *visited*. A node is said to be visited if we know that it is not a goal node.
- Green nodes: nodes that have been visited and that have at least one blue offspring.
- Black nodes: nodes that have been visited and whose offsprings have all been visited.

The strategy of the depth-first search consists in keeping all the possible path from the root node to a blue node in *open*, and to order them in favor to the deepest and in case of a tie the leftmost blue nodes. At the beginning of the algorithm, all the nodes of the conceptual tree are white except the initial state of the problem which is blue. At each step of the algorithm, if the first node of the first path of *open* (the deepest blue node and in case of a tie the leftmost one) is a goal node the algorithm terminates. Else, it depends on whether this node is:

- An inner node: the node is colored green and its successors are all colored blue.
- A leaf: the node is colored black as well as all its ancestor that have no blue brother.

Figure 2.4 shows an example of the coloration after a few steps for a particular tree.

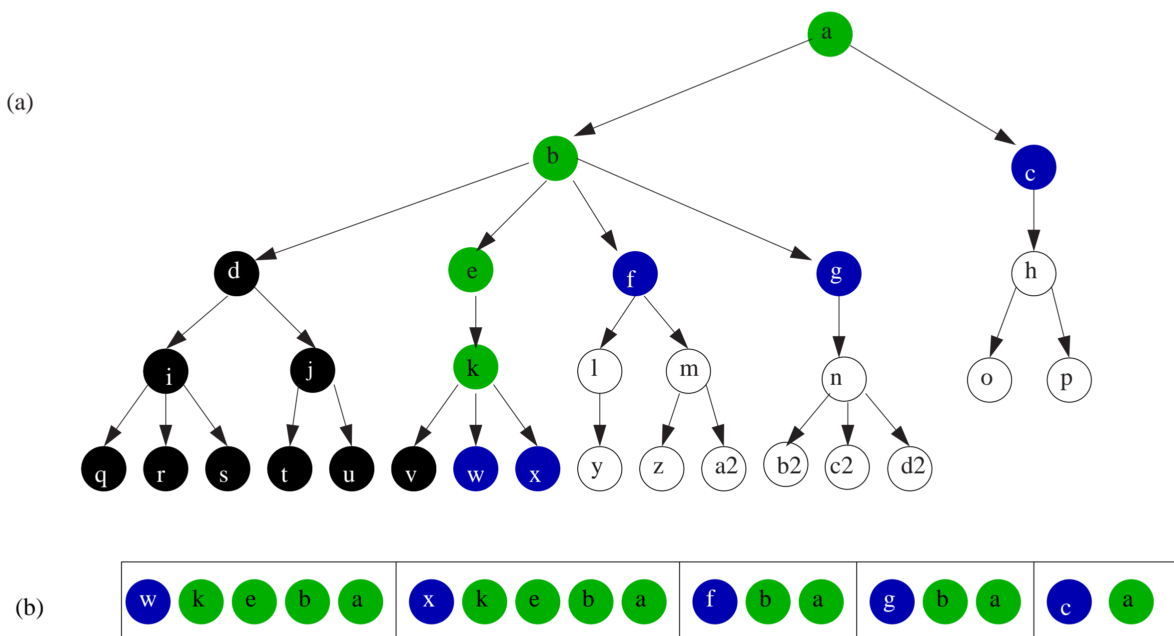

Figure 2.4: Schematized view of the coloration before the path w-k-e-b-a will be extracted from *open*: (a) The state-space tree. (b) The contents of *open*.

This coloration is only conceptual and will not be implemented. It is just useful for constructing a simple invariant that helps understanding the algorithm and proves its correctness. The strategy of the depth-first search can now be expressed as maintaining the following invariant:

> P: *open* is composed of all the possible paths between the root node of the tree and a blue node (ordered in favor to paths constructed from the deepest blue nodes, and in case of a tie the leftmost ones). The black nodes are all the left brothers of the green nodes and of the deepest and leftmost blue node. The white nodes are all the offsprings of the blue nodes.

The guardian B of the loop is true as long as *open* is not empty and as the first node of the first path of *open* is not a goal node. This means that after the loop testing if *open* is empty permits to decide whether a solution has been found or not.

2.3.1.2 Program

```
function depthfirst(start-state: STATE): PATH;
var  first-state: STATE;
     first-path: PATH;
     open: list of PATH;
     successor-list: SUCCESSORS;
begin
initialize-list!(first-path); insert-front!(start-state, first-path);
initialize-list!(open); insert-front!(first-path, open); {P}
do  not empty?(open) cand not goal-node?(get-first(get-first(open))) →
    first-path := remove-first!(open);
    first-state := get-first(first-path);
    successor-list := find-successors(first-state);
    "Add all the possible expansions of first-path with nodes of
    successor-list in front of open" {P}
od; {P and not B}
if  empty?(open) → depthfirst := EMPTY_LIST
□  not empty?(open) → depthfirst := get-first(open)
fi
end.
```

The operation *"Add all the possible expansions of first-path with nodes of successor-list in front of open"* can easily be computed. We can consider that the nodes of *successor-list* are ordered from right to left. This hypothesis can again be formulated without restriction because the tree is conceptual and the location of the successors in the virtual tree can thus be chosen. From this, a simple loop solves the problem. At each iteration, the first successor is removed from *successor-list*, a new path is created by adding this successor in front of *first-path* and the created path is added in front of *open*. The last added path will be the one that leads to the leftmost successor.

The simple invariant of this inner loop can be stated as follows:

> P_2: *successor-list* contains consecutive successors of the first node of *first-path* ordered from right to left and *open* begins with all the possible one-node expansions of *first-path* created with successors that are not in *successor-list* and ordered from left to right.

In fact, before the loop, *successor-list* contains all the successors and *open* begins with no expansion of *first-path*. At each step, the first node of *successor-list* is removed and the corresponding expansion is added in front of *open*. The guardian B_2 of this inner loop is true until *successor-list* is empty.

```
{P2}
do not empty?(successor-list) →
    insert-front!(insert-front!(remove-first!(successor-list),first-path),open)  {P2}
od {P2 and not B2}
```

The final code is obtained by replacing in the main code the operation *"Add all the possible expansions of first-path with nodes of successor-list in front of open"* by the code of the inner loop here above.

2.3.1.3 Termination

The termination of the inner loop is trivial. The command *remove-first!(successor-list)* is executed at each iteration, so the size of *successor-list* will be decremented. The loop will thus end after a number of iteration equal to the size of *successor-list*. As the latter has been produced by the function *find-successors* and as this function returns a finite list of successors, the number of iteration will be finite too.

The main loop is more critical. Actually, we have no termination guarantee for general state-space trees. Indeed, if an infinite branch exists in the tree and if the first goal node is situated to the right of the first infinite branch, it will never be found. In fact, the nodes of the infinite branch must all be colored green before nodes situated to the right of the branch will be considered and it would take an infinite time to color them.

Intuitively however, if the state-space is finite this problem will not appear and the main loop will terminate. More formally, in a finite tree, the number of possible paths is finite. Therefore, the number of new paths that can be added to *open* is finite. At each iteration, a path is removed from *open* and new paths are potentially added. By construction, the same path cannot be added twice. This means that after a finite number of iteration, it will be impossible to add new paths to *open*. From this point, paths will only be removed and the size of *open* will decrease. The loop will terminate when the list is empty.

The efficiency of the depth-first search algorithm will only be investigated at the end of this section, where a comparison between the different blind methods will be done. The condition that the tree should be finite is worrying, as it will not be the case for many games. The program proposed in the next subsection solves this problem.

2.3.2 Depth-First Search with Cycle Detection

This algorithm is only a slight modification of the pure depth-first algorithm where only the finite sub-tree of the acyclic paths of the state-space tree is explored. A list *open* will again be used but this time only acyclic paths will be added. To achieve this, only the successors that have no clones in the current path will be used for expansion.

2.3.2.1 Invariant

The invariant will once again be constructed using a virtual coloring of the nodes. The definition of the different colors remains the same but the notion of visited has changed. A node is now said to be visited if we know that it is not a goal node **or if we know that it is a clone of one of its ancestors**.

To achieve the new coloration of the nodes, the same coloring as before is used except for the case of inner nodes. The list of all the successors of the inner node is again generated but this time all the successors that are clones of one of their ancestor are colored black as well as all their descendant [3]. The latter are indeed all clones, as the offsprings of a clone are also offsprings of the original node. The successors that have not been colored black are then colored blue. If all the successors where colored black, then the treatment that was performed on leaf nodes will be performed on this inner node (directly after the black coloration) and the coloring mechanism continues.

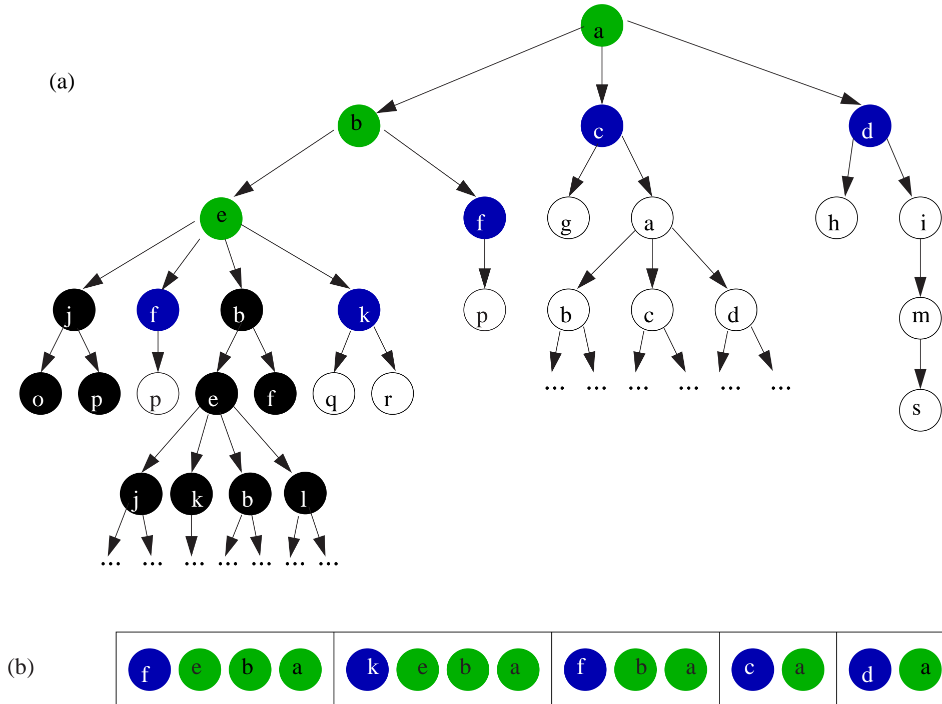

Figure 2.5: Schematized view of the coloration before the path f-e-b-a will be extracted from *open*: (a) The infinite state-space tree. (b) The content of *open*.

[3]As a clone has an infinite number of descendant, an infinite number of nodes will have to be colored black. However, recall that the coloring is only conceptual. That means that no infinite node-painting will be performed. The aim of this virtual coloration is only to model the fact that no acyclic solution exists in the subtrees of clones and that these nodes can be excluded from the search.

Figure 2.5 shows an example of the coloration after a few steps for a particular infinite tree. The invariant is only a slight modification of the invariant of the pure depth-first search algorithm:

> P: *open* is composed of all the possible paths between the root node of the tree and a blue node (ordered in favor to paths constructed from the deepest blue nodes, and in tie cases the leftmost ones). The black nodes are all the left brothers of the green nodes and of the deepest and leftmost blue node **and all the children of the green nodes that are clones of one of their ancestor as well as the offsprings of those nodes**. The white nodes are all the offsprings of the blue nodes.

2.3.2.2 Program

The main code is the same as before except that the operation *"Add all the possible expansions of first-path with nodes of successor-list **that are not member of first-path** in front of open"* will be used instead of the other one.

This operation can be computed using a similar inner loop as before, with the difference that at each iteration, the new path with the first successor of *successor-list* is only created and added in front of *open* if this successor is not a member of *first-path*. The invariant P_2 is the same as before except that we add *"all the possible one-node expansions of first-path created with successors that are **neither in successor-list nor in** **first-path**"* instead of simply *"not in successor-list"*. The guardian B_2 is exactly the same as before.

```
{P2}
do  not empty?(successor-list) →
      first-successor := remove-first!(successor-list);
      if  member?(first-successor, first-path) → skip
      □ not member?(first-successor, first-path) →
          insert-front!(insert-front!(first-successor, first-path), open)
      fi {P2}
od {P2 and not B2}
```

2.3.2.3 Termination

The termination of the inner loop can be proved with the same argument as before because the function *remove-first!* is once again called at each iteration. This time only acyclic paths are added to the list *open*. As we consider only infinite trees with cyclic infinite branches, the number of acyclic paths is finite and the same argumentation as before proves the termination of the main loop.

2.3.2.4 Variant

Another way to achieve this algorithm consists in keeping in addition to the *open* list of paths to expand, a *closed* list of the visited nodes. Instead of checking if a successor is in the path from which it was generated, the *closed* list will be consulted. The clones of already visited nodes will thus be detected directly.

The definition of the different colors remains the same but the notion of visited changes again. A node is now said to be visited if we know that it is not a goal node **or if we know that it is a clone of a previously visited node**. To achieve the new coloration of the nodes, the same coloring as usually is used except for the case of inner nodes. The list of all the successors of the inner node is again generated but this time all the successors that are clones of a green or a black node are colored black as well as all their descendant. The invariant P is easy to modify: the black nodes are all the left brothers of the green nodes and of the deepest and leftmost blue node and all the children of the green nodes **that are clones of a green or a black node** as well as the offsprings of those nodes. The invariant P_2 must merely be adapted for performing the membership test on *closed*.

```
function depthfirst-cycledetection(startstate: STATE): PATH;
var  first-state, first-successor: STATE;
     first-path: PATH;
     open: list of PATH;
     closed: list of STATE;
     successor-list: SUCCESSORS;
begin
initialize-list!(first-path); insert-front!(startstate, first-path);
initialize-list!(open); insert-front!(first-path, open); initialize-list!(closed); {P}
do  not empty?(open) cand not goal-node?(get-first(get-first(open))) →
    insert-front!(first-state, closed);
    first-path := remove-first!(open);
    first-state := get-first(first-path);
    successor-list := find-successors(first-state); {P2}
    do  not empty?(successor-list) →
        first-successor := remove-first!(successor-list);
        if  member?(first-successor, closed) → skip
        □  not member?(first-successor, closed) →
            insert-front!(insert-front!(first-successor,first-path),open)
        fi {P2}
    od {P and P2 and not B2}
od; {P and not B}
if  empty?(open) → depthfirst-cycledetection := EMPTY_LIST
□  not empty?(open) → depthfirst-cycledetection := get-first(open)
fi
end.
```

The termination of the inner and main loops are the same as for the first variant. The membership test will take time proportional to the length of *closed*. As *closed* contains also the states of the current path, this methods performs less efficiently than the preceding one. However, the advantage of this method is that for some problems, *closed* can be implemented as an indexed vector or a hash-table instead of a list and that the membership test will thus take a constant time. The first variant of the algorithm took time proportional to the length of the paths of *open* which are already implemented as list and it was therefore not possible to go faster.

2.3.3 Depth-Limited Depth-First Search

Another way to ensure to termination of the depth-first search algorithm is to limit the depth of the search. All the nodes situated deeper than the limit will be ignored and the algorithm terminates for all type of trees (even infinite trees with acyclic infinite branches). The problem of this method is that the goal nodes situated deeper than the limit are also neglected. The algorithm could thus conclude wrongly that no solution exists. It can therefore be used only when the fact that the first solution cannot be deeper than a certain depth is known a priori.

A simple but inefficient way to implement this search strategy would consist in modifying the classical depth-first algorithm such that paths of the same length as the depth limit are not expanded. The inefficiency comes from the fact that computing the length of a path takes a time proportional to its length. The classical implementation consists therefore in memorizing the length of each path of *open* in another list called *open-elem-sizes* (keeping them both in the same list is of course also possible). Computing the length of a new created path consists now merely in adding one to the length of the path that was just expanded.

2.3.3.1 Invariant

The invariant will be constructed with the same coloring of the nodes as for the general depth-first search algorithm, except that we consider that the nodes that are deeper than the depth limit have all been colored black before the algorithm starts. As usually, this coloration is conceptual; in this case its only purpose is to model the fact that those nodes are ignored from the very start. The different steps of the coloring of the nodes remain the same as for the general depth-first algorithm except that when the successors of an inner node are black the inner node is considered as a leaf node. The invariant can now be defined as follows:

> P: *open* is composed of all the possible paths between the root node of the tree and a blue node (ordered in favor to paths constructed from the deepest blue nodes, and in tie cases the leftmost ones). *Open-elem-sizes* is the list of the length of the path of *open* in the same order. The black nodes are all the left brothers of the green nodes and of the deepest and leftmost blue node. The white nodes are all the offsprings of the blue nodes.

The guardian B remains the same as for the general depth-first search algorithm.

2.3.3.2 Program

```
function limited-depthfirst(start-state: STATE; depth-limit: integer): PATH;
var  first-state: STATE;
     first-path: PATH;
     open: list of PATH;
     open-elem-sizes: list of integer;
     current-size: integer;
     successor-list: SUCCESSORS;
begin
initialize-list!(first-path); insert-front!(start-state, first-path);
initialize-list!(open); insert-front!(first-path, open);
initialize-list!(open-elem-sizes); insert-front!(1, open-elem-sizes); {P}
do  not empty?(open) cand not goal-node?(get-first(get-first(open))) →
     first-path := remove-first!(open);
     current-size := remove-first!(open-elem-sizes);
     if   current-size = depth-limit → skip
     □   current-size < depth-limit →
         first-state := get-first(first-path);
         successor-list := find-successors(first-state);
         "Add all the possible expansions of first-path with nodes of
         successor-list in front of open and actualize open-elem-sizes"
     fi {P}
od; {P and not B}
if   empty?(open) → limited-depthfirst := EMPTY_LIST
□  not empty?(open) → limited-depthfirst := get-first(open)
fi
end.
```

The operation *"Add all the possible expansions of first-path with nodes of successor-list in front of open* ***and actualize open-elem-sizes****"* is a slight modification of the corresponding operation of the depth-first search. The invariant P_2 remains the same except that a sentence must be added to express that *open-elem-sizes* must always contain the list of the length of the paths of *open* in the same order.

```
{P2}
do  not empty?(successor-list) →
       insert-front!(insert-front!(remove-first!(successor-list),first-path),open);
       insert-front!(1 + current-size, open-elem-sizes) {P2}
od {P2 and not B2}
```

2.3.3.3 Termination

The termination of the inner loop is exactly the same as its counterparts. The termination of the main program consists again in proving that the number of different paths that can be added to *open* is finite and that we will eventually reach a point where paths can only be removed. This is clear since the depth of the search has been limited and the explored tree is therefore finite and contains a finite number of different paths from the root.

The difficulty when using this algorithm is to chose a good depth-limit. A too small value would lead to a high probability that all the goal nodes are ignored and a too large value would lead to an important time complexity (non-solution paths will be expanded very deep wasting a lot of time). The next algorithm solves this problem.

2.3.4 Iterative Deepening

To avoid the difficulty of the choice of the depth limit, we can execute the depth-limited depth-first search iteratively by increasing the depth limit at each iteration until a solution is found. An integer *current-limit* will represent the current depth-limit. It will be initialized to zero and increased by one at each iteration. The variable *current-solution* will be the returned value of the function *limited-depthfirst* for the current depth-limit. Initially it will be the empty list and it will change only in the last iteration, i.e. when it becomes the least deep and leftmost solution path. The algorithm can then terminate and the guardian B which is true as long as *current-solution* is the empty list is appropriate. The algorithm makes obviously the hypotheses that at least one goal node exists. This is not worrying for practical games which have at least one finite solution.

2.3.4.1 Invariant

P: *current-solution* is the leftmost solution path of the state-space tree that has a length exactly equal to *current-limit* (empty list if no such solution exists).

2.3.4.2 Program

```
function iterative-deepening(start-state: STATE): PATH;
var  current-limit: integer;
     current-solution: PATH;
begin
current-limit := 0; current-solution := EMPTY_LIST; {P}
do  empty? current-solution →
      current-limit := current-limit + 1;
      current-solution := limited-depthfirst(start-state, current-limit) {P}
od; {P and not B}
iterative-deepening := current-solution
end.
```

2.3.4.3 Termination

The algorithm terminates only if a goal node exists in the state-space tree. Recall that we consider only finite trees or infinite trees with cyclic infinite branches and that therefore at least one goal node must be situated at a finite depth if a goal node exists in the tree.

The advantage of this method is that it finds the shortest solution path. The algorithms seen till now did not have this convenient property. They find the leftmost solution paths and this is not necessarily the shortest. The inconvenient of the iterative deepening algorithm is that at each iteration, the paths previously computed have to be recomputed again. However, in the efficiency analysis of the different blind methods we will show that it will not affect the time complexity so much. Typically, when the branching factor is not too small, most of the time will be spent in the last iteration and repeated computations with lower limit values adds relatively little to the total time.

2.3.5 Breadth-First Search

This algorithm starts with the same idea as the depth-first search algorithm: to find a goal node from a given node N, if N is already a goal node the search is terminated, else another node of the state-space is picked and the algorithm works forward from that node. The difference is that the picked node is no more the leftmost child of the node, but the closest node to the start node that was not visited yet. When several such nodes exist, the leftmost one is conventionally chosen. Thus, if the previous picked node is not the rightmost of a particular level, the next picked node will be its right neighbour. Else, it will be the leftmost node of the next level. The algorithm progress thus in breadth, level by level. Hence the name of the algorithm.

The idea of the implementation consists in working with a list *open* similar to the one used by the depth-first search strategy. The only difference, is that all the expanded paths will be added to the back of the list instead of the front [4].

2.3.5.1 Invariant

The invariant will be constructed using the following virtual coloring of the nodes:

- White nodes: nodes that have not yet been generated by the search.
- Green nodes: nodes that have been visited. From now and for the rest of the document, the original notion of visited will be used (nodes that are known non-goal nodes).
- Blue nodes: nodes that have been generated but not visited.

[4] Note that in the implementation of lists as linked lists that we have suggested, it is possible to keep two pointers that indicates the locations of the first and the last element of the list. The function *insert-back!* can thus be implemented in time constant. The only operation that needs to be done is merely to link the last element of the list (that was linked to *nil*) to the new element and to link this new element to *nil*.

At the beginning of the algorithm, all the nodes of the conceptual tree are white except the initial state of the problem which is blue. At each step of the algorithm, if the least deep and leftmost blue node is a goal node the algorithm terminates. Else, this node is colored green. If no blue node exists, all the nodes have been visited and no solution exists. As usual, this coloration is only conceptual and will not be implemented.

The strategy of the algorithm can be expressed as maintaining the invariant:

> P: *open* is composed of all the possible paths between the root node of the tree and a blue node (ordered in favor to paths constructed from the least deep blue nodes, and in tie cases the leftmost ones). The green nodes are the ancestor of the blue nodes and the white nodes their offsprings.

Figure 2.6 shows an example of the coloration and the corresponding contents of *open* after a few steps for a particular tree.

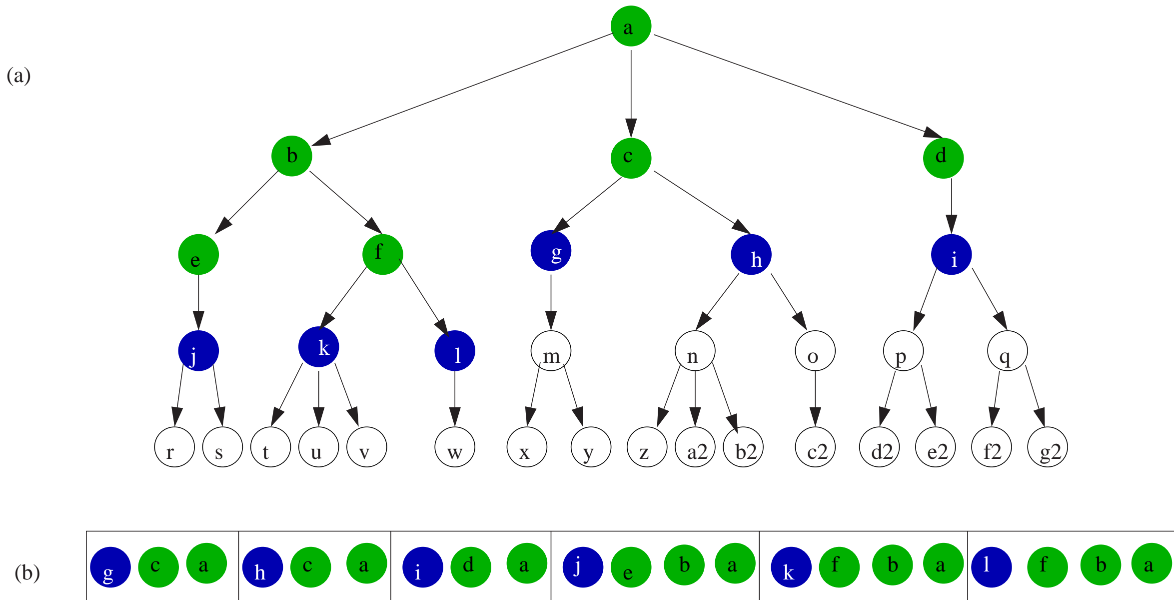

Figure 2.6: Schematized view of the coloration before the path g-c-a will be extracted from *open*: (a) The state-space tree. (b) The contents of *open*.

The main code is the same as for the depth-first search except that the operation *"Add all the possible expansions of first-path with nodes of successor-list to the* ***back*** *of open"* is used instead of the other. This can be computed by considering this time that the nodes of *successor-list* are ordered from left to right. Considering the opposite choice as for the depth-first case is not problematic. The explored tree is still conceptual and the nodes must not be placed at the same positions than for the depth-first search.

A simple loop performs the operation. At each iteration, the first successor is removed from *successor-list*, a new path is created by adding this successor in front of *first-path* and the created path is added to the back of *open*. The paths added at the end of *open* will therefore be ordered in favor to the ones containing the leftmost successors and the invariant is preserved. The loop terminates when the list is empty. The simple invariant of this inner loop can be stated as follows:

P_2: *successor-list* contains consecutive successors of the first node of *first-path* ordered from left to right and *open* ends with all the possible one-node expansions of *first-path* created with successors that are not in *successor-list* and ordered from left to right.

The guardian B_2 is the usual one for the actualization of *open*, i.e. it is true as long as *successor-list* is not empty.

```
{P2}
do not empty?(successor-list) →
    insert-back!(insert-front!(remove-first!(successor-list),first-path),open)   {P2}
od {P2 and not B2}
```

2.3.5.2 Termination

The termination of the inner loop can be proved with the usual argument. In fact, the function *remove-first!* is once again called at each iteration and *successor-list* will eventually become an empty list after a finite number of steps.

The algorithms terminates if a goal node exists at a finite depth. Indeed, at each iteration a node is colored green. Furthermore, no node is colored green before all the nodes of the less deep levels are all colored green. As the number of nodes of each level is finite, the level of the goal node will be reached after a finite number of steps and finding the goal in this level will also require a finite number of steps.

If no goal node exists, the algorithm terminates if and only if the state-space tree is finite. The fact that exactly one node is colored green at each iteration demonstrates this fact. Indeed, if the tree is finite, all the nodes will be green after a finite number of steps and the algorithm concludes that no solution exists. In the infinite case it would take an infinite time to color all the nodes and the procedure would never terminate.

2.3.5.3 Variants

A variant with cycle detection can also be constructed. Limiting the depth of the search makes however less sense as the algorithm works already level by level.

2.3.6 Efficiency Analysis

This subsection compares the main blind methods on two important measures of complexity:

- The time complexity: correspond to the order of magnitude of time needed for finding a solution. It is measured as the total amount of nodes generated by the search strategy until a solution is found.
- The space complexity: correspond to the order of magnitude of memory used during the execution of the algorithm. It is measured as the maximum number of nodes that must be kept in memory during the execution of the algorithm.

The different uninformed search methods will be compared on a generic state-space tree for which each inner node has exactly b successors and the only solution is the rightmost node situated at a depth d. This is the worst place possible at depth d because we have supposed that the different strategies works from left to right. Note that the performances of all algorithms become better when the first solution moves to the left, but this can only occur by chance as an uninformed method cannot order the nodes so that the most promising are explored first.

The number of nodes grows exponentially with the depth, so the number of nodes generated by the breadth-first search strategy is $1+b+b^2+b^3+\ldots$ The time complexity is thus $O(b^d)$. The breadth-first search maintains all the b^d candidate paths in memory. Those paths are composed of a maximum of d nodes and the space-complexity is thus $O(b^d)$ too.

In the depth-first search category, we will not consider the pure algorithm because it has no termination insurance and both complexities may be infinite. We will rather study the complexities of the depth-first search limited to a depth of d_{max} so that $d \leq d_{max}$. The total number of nodes of the depth-limited tree is $b^{d_{max}}$ and the nodes will all be generated except the offspring of the solution node. The time complexity is therefore $O(b^{d_{max}})$. With a cycle-detection mechanism, the complexity remains generally the same except for special state-space trees for which the number of cycle is very high.

The maximum number of nodes in memory occurs when the search has reached the leftmost leaf. At this precise moment, the number of green nodes is $d_{max} - 1$; i.e. all the nodes of the leftmost branch except the leaf. The number of blue nodes is $(b-1) \times (d_{max} - 1) + 1$; i.e. each green node has exactly $(b-1)$ blue children except the deepest one which has one more. The number of paths in memory is equal to the number of blue nodes. All paths are bounded by a size of d_{max}. The space complexity is therefore only $O(d_{max}^2)$.

The iterative deepening algorithms performs $(d+1)$ depth-first searches limited respectively by a depth of $0, 1, \ldots, d$. The maximum number of nodes in memory occurs in the last iteration when the search reaches the leftmost node of depth d. The same argumentation as for the depth-limited depth-first search shows that the space complexity is $O(d^2)$.

The start node will be visited $(d+1)$ times (at each iteration), the children of the start nodes d times (at each iteration except the first one), etc. The number of generated nodes is thus:

$$(d+1) \times 1 \;+\; d \times b \;+\; (d-1) \times b^2 \;+\; \ldots \;+\; 1 \times b^d$$

This gives also a time complexity of $O(b^d)$. The conclusion is that even if the iterative deepening seems very coarse at the first glance, the regenerating of nodes is in fact surprisingly small. It can be shown that the ratio between the number of nodes generated by iterative deepening and those generated by breadth-first search is approximately $\frac{b}{b-1}$ for $b \geq 2$ [2]. This means that when the branching factor is high the difference becomes practically unnoticeable. The following tabular summarizes the conclusions:

Algorithm	Time complexity	Space complexity	Shortest solution
Breadth-first	b^d	b^d	yes
Depth-limited Depth-first	$b^{d_{max}}$	d_{max}^2	no
Iterative deepening	b^d	d^2	yes

Table 2.1: Efficiency comparison of the three main blind methods

The column *Shortest solution* indicates whether each algorithm finds the shortest solution first or not. For some problems however the arcs may have different *costs* and the optimal solution would not be the shortest but the one for which the sum of the costs of the arcs is minimal. It is not possible to achieve such optimality with the pure blind methods presented here. The A* and IDA* algorithms presented in the next section are designed to ensure this cost-optimality.

The *cost* of an arc is a number which indicates how difficult it is to make the problem evolve from the initial state of the arc to its target state. For some problems it is indeed convenient to do so. Consider for example a path-finding application. A robot must go from its starting point to a target point of a game area. The latter is composed of forest, lakes and deserts. We could for example consider that it is more difficult to travel through a forest than a desert, and even more difficult to traverse a lake. A convenient way to model this is to associate a cost of 1 to the arcs that represents a one-step move in the deserts, 2 in the forest and 3 in a lake. The problem is now to find the solution-path with minimum cost.

The iterative algorithm combines obviously the best properties of breadth-first search and depth-first search. It is thus often the best choice for practical AI problems when the problem is not too complex. The depth-first search algorithm is also very often used. The language PROLOG for example works in depth-first manner. However, the blind methods that we have seen make nothing to fight against the combinatorial explosion because they consider each possibility to be equally probable. For complex AI problems, methods based on heuristics are more appropriate. This is the subject of the next section.

2.4 Heuristically informed methods

In chapter 1 we have seen that a heuristic is a function for selecting lines that have a high probability of success. The search efficiency may improve spectacularly by using clever heuristics to guide through the search. In this section, the most famous algorithm that uses heuristics and its most important variant will be presented: the A* and IDA* algorithms.

From now, we will suppose that a cost has been associated to the arcs of the state-space tree. This means that a function $cost(s_1, s_2)$ defines the cost of moving from each state s_1 of the state-space tree to each successor s_2 of s_1. If the problem does not require to associate different costs to the arcs, the function will simply use a value of 1 for each.

2.4.1 The A* algorithm

The A* algorithm was first presented in 1968 [13] and small corrections to the original paper were made four years later by the same authors in [14]. The A* algorithm is a best-first search algorithm, i.e. an algorithm that tries to explore the best candidate in the sense of an heuristic first. In almost all sources, the A* algorithm is presented using abstract data types.

In this document we will study the A* algorithm very carefully, using a software engineering approach. We will first reconstruct the algorithm from the original idea and therefore use the same abstract data types as in [13]. We will then show that their exist different ways to implement the abstract objects and give the corresponding theoretical performances. We will finally add our contribution by showing that for a particular choice which is often done in practice it is even possible to adapt the algorithm to this particular choice and improve the performances. We will also show that it is far from trivial to decide which choice is the best. The constant factors hidden in the theoretical performances may favor one method for some practical problems and another for others.

The principle of the best-first search strategy is simple. It consists in keeping in memory a set *open* of candidate nodes and a set *closed* of visited nodes. Initially, the former contains only the root node of the state-space tree and the latter is empty. At each iteration, the most promising node of *open* is extracted and added to *closed*. If this node is a goal node the algorithm terminates successfully. Else, its successors are generated and for each: if the successor is in *open* or *closed* and the existing one is as good or better then the successor is discarded. Else, the possible old occurrences of the successor are removed from *open* and *closed* and the successor is added to *open*. The algorithm can thereafter start the next iteration with the actualized versions of *open* and *closed*. It terminates when *open* is empty (no solution) or when a solution node has been found. Note that each node has to maintain a pointer to the node from which it was generated in order to be able to reconstruct the final solution-path.

To be more precise about the mechanism used to select the most promising node, let us define a useful function: $f(n)$ takes a node of the graph as argument and returns an estimates of the cost of the optimal path that goes through this node. From this, the extracted node of *open* is merely its member with the minimum f-value. The A* algorithm is the particular best-first algorithm obtained by using the function $f(n) = g(n) + h(n)$; where $g(n)$ is an estimate of the cost of the best path from the start node to n and $h(n)$ an estimate of the remaining cost necessary to reach a goal node. Figure 2.7 on the next page illustrates the definition of the f-function.

The term $g(n)$ is calculated as follows: if we have to evaluate node n it means that a path from the initial node s to node n has been found. We can use the sum of the costs of the arcs of this path as an estimation of the optimal path from s to n. It is indeed only an estimation as there may exist a better path from s to n not found yet by the search. The term $h(n)$ is more worrying, as the part of the search-space behind node n has not been explored yet. Therefore, the function $h(n)$ must be constructed from the expert-knowledge of the problem. There is no general method for doing this, as it really depends on the particular problem.

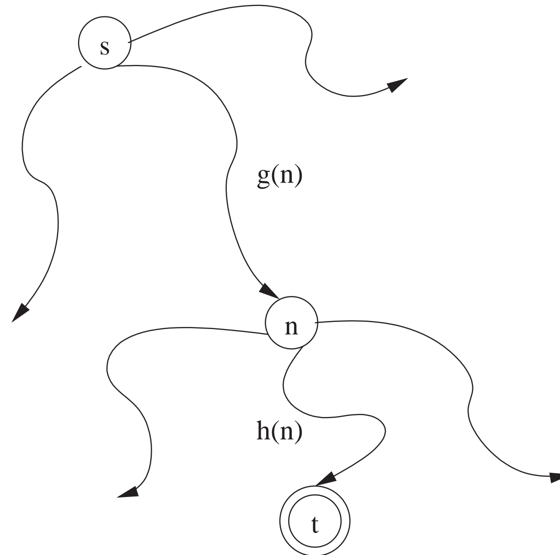

Figure 2.7: Function used by the A* algorithm to estimate the cost of the optimal path that goes from s to t via n: $f(n) = g(n) + h(n)$.

2.4.1.1 Definition of the Abstract Data Types

The A* algorithm is constructed on two different set of nodes: *open* and *closed*. These are abstract data types as they can for examples be implemented as lists, sorted lists, indexed vectors, hash tables, etc. The A* algorithm needs to perform some operations on *open* that have not to be executed on *closed*. The most interesting implementations are thus not necessarily the same for both. This leads to a wide range of possibilities.

In the blind methods, the nodes were simply the states of the games. Both notions have in fact been used as synonyms in the previous section. This can no longer be done here because a node must now contain at least four different information:

1°) STATE: the state of the game which corresponds to the node.

2°) FATHER: a pointer to the father of the node[5].

3°) F-VALUE: the f-value associated with the node.

4°) G-VALUE: the g-value associated with the node.

Note that the h-value can also be added but it is not necessary as it can be obtained from the f- and g- values. We have chosen to keep these two values in memory instead of the h-value plus another one because the h-value will only be calculated for determining the f-value and serves no other purpose. In contrast, the f-value is notably needed when the most promising node of *open* has to be determined, for testing if a particular node is more promising than another, etc. The g-value of a node should also be accessible directly as it is needed for calculating the g-value of each successor without having to recalculate the sum of the costs of the arcs of the path from the root node to the successor (enabling to merely add the g-value of the father to the cost of the arc between the father and its successor).

[5]The root is the only node without father and this will be represented by the constant *nil*.

The notion of node is obviously abstract too but this is less critical than for *open* and *closed*. Indeed, a node can simply be implemented as a record containing the four information enumerated above. A more sophisticated implementation choice is not needed for a structure containing only four members! We can thus define the following data type:

```
type NODE = record
              state: STATE;
              father: NODE;
              f-value: integer;
              g-value: integer
            end;
```

As the implementation of *open* and *closed* are not necessarily the same, we will define the abstract operations that will be performed on them separately, even if the operations of *closed* are conceptually the same as the corresponding operations of *open*. The operations for *open* are:

- initialize-open!(var open: set of NODE);
  ⇒ procedure that initialize the set *open* as an empty set.
- insert-in-open!(x: NODE; var open: set of NODE);
  ⇒ procedure that inserts the node x in the set *open*.
- get-open-best(open: set of NODE): NODE;
  ⇒ function that returns a copy of the node with the lowest f-value of the non-empty set *open*.
- remove-open-best!(var open: set of NODE): NODE;
  ⇒ function that removes the node with the lowest f-value of the non-empty set *open* and returns this node.
- empty-open-set?(open: set of NODE): boolean;
  ⇒ predicate true if and only if *open* is an empty set.
- get-open-member(x: NODE; open: set of NODE): NODE;
  ⇒ function that returns the node of *open* with the same state as x if such a node exists, and the constant *nil* else.
- erase-open-member!(x: NODE; var open: set of NODE);
  ⇒ procedure that removes the member x of *open*.

The operations for *closed* are initialize-closed!, insert-in-closed!, get-closed-member and erase-closed-member!. Their definitions are the same as those of their open-counterparts (but their implementation may be different and we must thus define them separately to be able to recognize in which case which implementation is meant).

2.4.1.2 Invariant

The invariant will be constructed using the following virtual coloring of the nodes:

- White nodes: nodes that have not yet been generated by the search.
- Black nodes: nodes that have been generated but are clones of a generated node which has an f-value smaller than them or are clones of a previously generated node with the same f-value.
- Blue nodes: nodes that have been generated but not visited (and does not satisfy the condition to be black nodes).
- Green nodes: nodes that have been visited (and does not satisfy the condition to be black nodes).

The strategy of the algorithm consists in keeping all the blue nodes in the *open* set and all the green nodes in the *closed* set. At the beginning of the algorithm, all the nodes of the conceptual tree are white except the initial state of the problem which is blue. At each step of the algorithm, if the blue node with the lowest f-value is a goal node the algorithm terminates. Else, this node is colored green, the list of all the successors of the node is generated and for each:

- If the successor is a clone of a blue node or a green node and its f-value is greater or equal, it is colored black.
- Else the successor is colored blue and its possible clones are colored black.

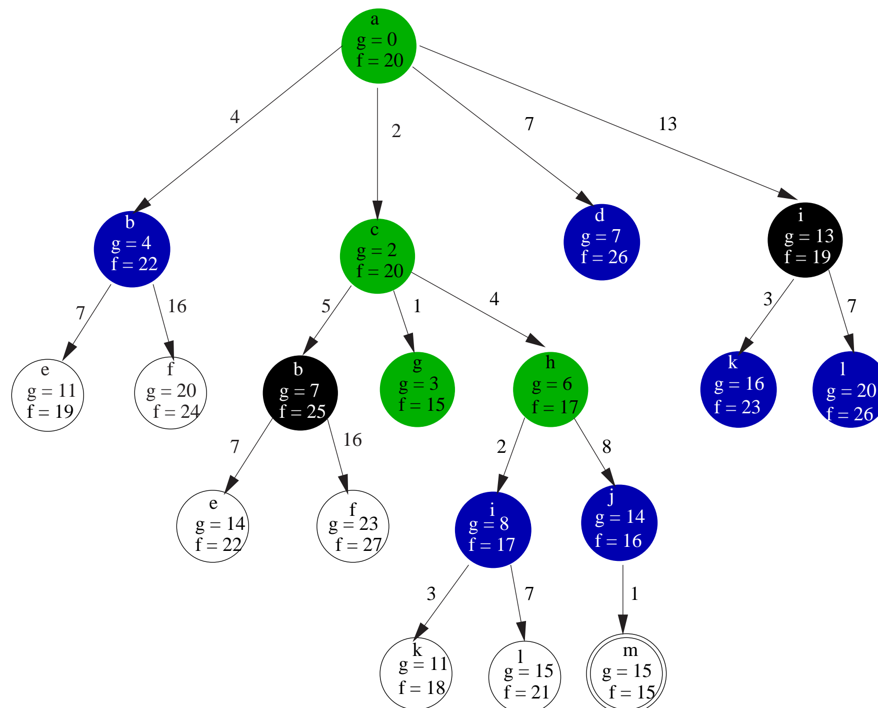

Figure 2.8: Schematized view of the coloration before node j will be expanded and the goal node m will be found.

As usually, this coloration is only conceptual and will not be implemented. Note that the offsprings of black nodes are not necessarily black nodes, as can be seen on Figure 2.8 for a particular tree. The invariant can now be constructed easily:

> P: *open* is composed of all the blue nodes of the state-space tree and *closed* of all the green nodes. The black nodes are clones of at least one blue or green node with an f-value less or equal than them. The white nodes are the remaining nodes.

The guardian B of the loop is true as long as *open* is not empty and as the best node of *open* is not a goal node.

2.4.1.3 Program

```
function a-star(start-state: STATE): PATH;
var  current-node, best-node, open-member, closed-member: NODE;
     open, closed: set of NODE;
     solution-path: PATH;
     successor-list: SUCCESSORS;
begin
current-node.state := start-state;
current-node.father := nil;
current-node.g-value := 0;
current-node.f-value := h(start-state);
initialize-open!(open); initialize-closed!(closed);
insert-in-open!(current-node, open); {P}
do  not empty-open-set?(open) cand not goal-node?(get-open-best(open).state) →
      best-node := remove-open-best!(open);
      insert-in-closed!(best-node, closed);
      successor-list := find-successors(best-node.state);
      "Actualize open and closed" {P}
od; {P and not B}
if  empty-open-set?(open) → a-star := EMPTY_LIST
□  not empty-open-set?(open) → "Generate solution path"
fi
end.
```

The operation *"Actualize open and closed"* can be concretized by a simple loop. At each iteration, the first member of *successor-list* is removed. Recall that the members of the latter are not nodes but states of the game. The node corresponding to the removed state of *successor-list* must thus first be created (notably by calculating its g- and f-values). If the created node is a clone of a node of *open* or *closed* and the clone has an f-value smaller or equal then the created node is ignored. Else, the possible clone is removed from *open* or *closed* and the created node is inserted in *open*. If we note $open_0$ and $closed_0$ the contents of *open* and *closed* before the loop, we can define the invariant as follows:

P_2: *successor-list* contains the states of some successors of *best-node*. *open* contains $open_0$ **plus** all the successors of *best-node* whose states are not in *successor-list* except those that are clones of a node of $open_0$ or $closed_0$ with an f-value greater or equal than the f-value of the clone and **minus** all the elements of $open_0$ that are clones of a successor of *best-node* whose state is not in *successor-list* and have an f-value greater than the one of the successor. *closed* contains $closed_0$ minus all the elements of $closed_0$ that are clones of a successor of *best-node* whose state is not in *successor-list* and have an f-value greater than the one of the successor.

In fact, before the loop, *successor-list* contains all the successors of *best-node* and *open* and *closed* are exactly $open_0$ and $closed_0$. At each step the first state of successor-list is removed and if the corresponding node is a clone of a node of *open* or *closed* and its f-value is greater or equal then the node is ignored else the node is added to *open* and its possible clones are removed from *open* and *closed*. The guardian B_2 is true as long as there are successors to treat, i.e. as long as *successor-list* is not empty.

```
{P2}
do  not empty?(successor-list) →
      current-node.state := get-first(successor-list);
      current-node.father := best-node;
      current-node.g-value := best-node.g-value
                              + cost(best-node.state, current-node.state);
      current-node.f-value := current-node.g-value
                              + h(remove-first!(successor-list));
      open-member := get-open-member(current-node, open);
      closed-member := get-closed-member(current-node, closed);
      if (open-member ≠ nil cand open-member.f-value ≤ current-node.f-value) or
         (closed-member ≠ nil cand closed-member.f-value ≤ current-node.f-value)
          → skip
      □ (open-member = nil cor open-member.f-value > current-node.f-value) and
         (closed-member = nil cor closed-member.f-value > current-node.f-value)
          → insert-in-open!(current-node, open);
            if open-member = nil → skip
            □ open-member ≠ nil → erase-open-member!(open-member, open)
            fi;
            if closed-member = nil → skip
            □ closed-member ≠ nil → erase-closed-member!(closed-member, closed)
            fi
      fi {P2}
od {P2 and not B2}
```

The operation *"Generate solution path"* is much more simple to implement. The idea of the implementation consists in keeping in memory a variable *current-node* which is initially the goal node that has just been found and a variable *solution-path* which contains initially the path between the goal state and itself. At each iteration, the father of *current-node* becomes the new *current-node* and its state is added in front of the *solution-path.* The loop terminates when *current-node* is the root node and this is achieved by using a guardian B_3 which is true as long as the father of *current-node* is not the constant *nil.* From this, the invariant is easy to define:

> P_3: *current-node* is a node of the optimal solution path and *solution-path* is the path between the state of this node and the state of the goal node.

```
current-node := get-open-best(open); initialize-list(solution-path);
insert-front!(current-node.state, solution-path); {P3}
do   current-node.father ≠ nil →
     current-node := current-node.father;
     insert-front!(current-node.state, solution-path) {P3}
od; {P3 and not B3}
a-star := solution-path
```

2.4.1.4 Termination

The termination of the two inner loops is trivial. In the first one the procedure *remove-first!* is called at each iteration on the finite list *successor-list.* The latter will thus be empty after a finite number of steps and the loop terminates. In the second one *current-node* becomes the father of the preceding content of *current-node* at each iteration. As the solution path is finite, we will eventually reach the root node and this loop terminates too.

We will now prove that the main loop also terminates. The A* has thus the convenient property to always terminate, also when no goal node exists. Recall that we study only finite state-space graphs and that we have not to consider trees with infinite acyclic branches. The termination proof can therefore be separated in two cases:

- The case of a finite tree of N nodes: let n_{color} be the number of nodes of the given color. Then the function $t = (N+1) - 2 \times n_{green} - n_{blue} - n_{black}$ proves the termination. Indeed, at each iteration the blue node with the lowest f-value becomes green. If this node is a leaf no other color changes are performed and the function t diminishes by one (n_{green} augments by one and n_{blue} diminishes by one). If this node is an inner node the N_s successors are all colored black or blue. If none of the successors have clones the function t diminishes by $1 + N_s$ (n_{green} augments by one and $n_{black} + n_{blue}$ augments by $N_s - 1$). The blue clones does not affect t as they are repainted in black. However, each green clone will be changed into a black node and will thus make the function t augment by one. In the worst case all the successors are blue and have a green clone to repaint in black but t still diminishes by one in this case (n_{green} diminishes by $N_s - 1$, n_{blue} augments by $N_s - 1$ and n_{black} augments by N_s).

- The case of an infinite tree without any acyclic infinite path: we will prove that the algorithm only explores the finite tree of the acyclic paths. As we have proven that the algorithm terminates for finite trees this proves the termination in this case. When the search reaches a node that is a clone of one of its ancestor, the f-value of the clone must be smaller. Indeed, the cost of the arcs are strictly positive number so the g-value of the clone is smaller and the h-value are exactly the same as the state of the game is the same. The node will thus be colored black and its offsprings will remain white for the rest of the search. The nodes that are clones of one of their ancestor will thus all be black or white and will never be members of *open*. This proves that the nodes of the finite tree of the acyclic paths are the only ones that could be explored and proves thus the termination.

2.4.1.5 Implementation of *open*

In this section inspired from [5], we will present some possible implementations for *open*. We will not present all the possible choices that have ever been tried but only the most usual and of course the most efficient ones. To be able to decide which ones are the best, we must first identify which operations are the most critical. We have defined seven different operations that will be performed on *open*.

The most critical operation is the membership test *get-open-member*. It will indeed be executed on all the successors and is thus the most often called function. The second most used operation is the insertion operation *insert-in-open!*. It will be executed on all the blue successors. The operations *get-open-best* and *remove-open-best!* will each be called exactly once at each iteration. These are therefore also critical operations and we will only discuss the implementation of the latter here since the case of the former is only a part of the latter (for removing the best node, we must first find it).

The operation *remove-open-member!* is much less critical. It will indeed be called only in the rare cases when a successor is better than one of its clones. This occurs in fact rarely because the A* algorithm tries to expand the most promising nodes first and practice shows that with a good h-function the clone with the lower f-value is almost always the first one found. Finally, the operations *initialize-open!* and *empty-open-set?* are not critical as they can always both be implemented in time constant.

We can summarize this by asserting that the ideal implementation would optimize membership test, insertion, removing best and removing member in this order. Here are the most usual basic implementations:

- Unsorted array or list

  Membership test is slow, $O(N)$ where N is the size of *open*[6], to scan to entire structure. Insertion is instantaneous as the node can be added in front of the list (for arrays at the end to avoid to have to shift the elements before the insertion). This gives $O(1)$. Finding and removing the best element is $O(N)$ as we have again to scan the whole structure (and in the case of arrays to shift elements after the deletion). Even the removing member operation is $O(N)$.

[6] Note that the size of *open* is variable and that the efficiency will thus depend on the way that the nodes are inserted and removed from *open*. For some applications, the size of *open* will remain of reasonable size in the course of the treatment but it will grow exponentially for others.

- Sorted list

  Keeping the list sorted permits to improve the performance of the find and remove best operations. The best node will merely by at the head of the list and this gives time $O(1)$. However, the insertion operation will now take time $O(N)$ as we have to scan the list to find the right insertion place. The efficiency of the other operations remains the same.

- Sorted array

  Keeping the array sorted (so that the best is at the end) permits again to improve the find and remove operations to $O(1)$ but depreciate the insertion operation to $O(N)$. However, the membership test can now be executed in $O(logN)$ by using a binary search. The removing member operation takes time $O(N)$ to move all the right elements to the left after the deletion.

- Indexed array

  If the state-space tree is finite and the number of states of the game is not too high to create an indexed array of all the states, this structure can be envisaged. A problem-dependent indexing function $i(s)$ which associates an index to each state of the game must be provided. The membership test is $O(1)$, as we merely have to check whether the $i(s)^{th}$ element contains any data. Insertion is $O(1)$, we have just to actualize the $i(s)^{th}$ element. Find and remove best is $O(Total_{states})$, where $Total_{states}$ is the total number of states, since we have to search the entire structure. This is considerably bigger than the worst value encountered so far (in $O(N)$ the N was usually only a small subset of the nodes of the state-space). Finally, the removing member operation is only $O(1)$.

- Hash table

  When the number of states of the game is high and we have not enough memory to create an indexed array, a hash table with a hash function $h(s)$ that maps each state s into a hash code can be envisaged. In this case, the membership test is expected $O(1)$ (it will grow in case of collisions; it depends thus crucially of the quality of the hash function), insertion is expected $O(1)$, and remove best is $O(Max_{hash})$ where Max_{hash} is the number of hash codes. We have indeed to scan the entire structure. The removing member operation is again $O(1)$.

- Binary dictionary

  A binary dictionary is a binary tree (a tree in which every node has at most two children) with an ordering relation between the nodes of the tree. For each node n of the tree, if n_f is the f-value associated with n, all the nodes in the left-subtree of n have an f-value smaller than n_f and all the nodes in the right-subtree of n have an f-value greater or equal than n_f. This data structure is efficient when the tree is *balanced*, i.e. when the left and right sub-trees of each node have almost the same number of nodes. Consult [2] for an efficient implementation of binary dictionaries. Membership and removing member are $O(N)$ to scan the tree for the member. Insertion and remove-best are $O(logN)$. An efficient implementation of the insert and remove operations on binary dictionary which preserves the balance of the tree can be found in [2].

None of the basic data structures presented here is entirely satisfactory. To circumvent this, we can use an hybrid data structure that takes the best properties of the best basic data structures. A good choice when the number of nodes is not too high is to use a combination of an indexed array and a binary dictionary. The membership test will be performed on the indexed array leading to a complexity of $O(1)$. The insertion in the indexed array is $O(1)$ and in the binary dictionary we have $O(logN)$. The get best operation will be performed on the binary dictionary in time $O(logN)$. Once the node is found, we can compute its index and removing it from the index array will thus take time $O(1)$. Removing it from the binary dictionary will take time $O(logN)$. Finally, the removing member operation is $O(1)$ in the indexed array and $O(logN)$ in the binary dictionary. The following tabular summarizes the results:

Data Type	Membership Test	Insertion	Remove Best	Remove Member
Unsorted array/list	$O(N)$	$O(1)$	$O(N)$	$O(N)$
Sorted list	$O(N)$	$O(N)$	$O(1)$	$O(N)$
Sorted array	$O(logN)$	$O(N)$	$O(1)$	$O(N)$
Indexed array	$O(1)$	$O(1)$	$O(Total_{states})$	$O(1)$
Hash table	$O(1)$	$O(1)$	$O(Max_{hash})$	$O(1)$
Binary dictionary	$O(N)$	$O(logN)$	$O(logN)$	$O(N)$
Hybrid structure	$O(1)$	$O(logN)$	$O(logN)$	$O(logN)$

Table 2.2: Efficiency comparison of the most usual implementations for *open*.

The hybrid structure has the best theoretical performances. However, it is not forcingly the best choice in practice. Indeed, for some problems the average number of nodes in *open* is not so high and another implementation could outperform the hybrid structure. Indeed, the hidden factors in the theoretical performances begins to play a crucial role when N is not large and this must not be neglected. In this work, we have chosen to advocate two structures: the hybrid structure in cases where the number of states is high and the sorted list else. The choice of the latter could seem curious if we consider only the comparison table. However, section 2.4.1.7 will demonstrate that the A* can be rewritten in this case so that all the $O(N)$ operations are merged in a single scan of the list. This contribution makes the choice of sorted list very interesting in many practical applications. The most known domain of application of the A* algorithm is a good example: the path-finding problems. Indeed, in such problem the map is usually composed of a number of cells that is not so enormous and the performances of the improved A* with sorted lists are promising.

2.4.1.6 Implementation of *closed*

The choice for *closed* is easier. In fact, the operations that are performed on this structure does not require any ordering of the nodes which facilitates the task. For the same reasons as for *open*, the critical operations are *get-closed-member*, *insert-in-closed!* and *erase-closed-member!*. In practical implementations the *closed* set is sometimes implemented as a list but this gives only a complexity of $O(N)$ for the member and erase operations. The most used structures are actually the indexed arrays and hash tables. The particular choice depends once more on the number of states. In both cases, the complexity of all critical operations is $O(1)$ and we cannot expect less!

2.4.1.7 Improvement of the A* when *open* is a Sorted List

Algorithms based on abstract data types are good for understanding high-level ideas. However, lots of examples exists in the literature to show that such kind of abstraction can lead to non-optimal code [4]. We will show that it is also the case for the A* when *open* is implemented as a sorted list and we will give a solution to this problem. This is a contribution in the domain.

As the sorted lists are thought to be used in problems for which the number of states is not too high, we will implement *closed* as an indexed array. The number of states of the game must now be passed as additional argument to the A* function to be able to define the array. In this document we will suppose that when an array is created all its elements are initialized at *nil*. A problem-dependent function $i(s)$ which associates an unique number to each state s of the game must be provided by the application.

Normally, implementing the A* consists simply in choosing particular data structures for *open* and *closed*, implementing the seven resp. four basic operations and injecting everything in the pure A* abstract code. If we would do so, the operation *"Actualize open and closed"* would not be optimally implemented. For each successor, *open* would indeed be potentially entirely scanned a first time to decide if the successor is a clone of a node of *open* or not, a second time to insert the successor in *open* and a third time to remove the possible clone of the successor.

We will show that a single scan of the list permits to do all the operations that are needed. The idea consists in scanning *open* until one of the following condition is satisfied:

1°) A node with an f-value greater than the f-value of the successor has been found.

2°) A clone of the successor has been found.

3°) *open* has been entirely scanned without finding any node that satisfies the two other conditions.

If we are in the first case, the successor can be inserted just before the node on which the scanning has stopped. This node is indeed the first one with an f-value greater than the successor and inserting the successor before this node preserves the sort. If we are in the second case the clone is as good or better than the successor and the latter can be ignored. If we are in the third case the successor can be inserted at the end of *open*.

As *open* contains at most one clone of the successor, in the two last cases *open* has already been correctly actualized. However, in the first case *open* could still contain a clone with an higher f-value than the successor. The rest of the list must thus be scanned until a clone is found (it will then be removed) or all the elements have been scanned.

The invariants and guardians of the main loop and the two inner loops remain the same. The new implementation will however achieves P_2 more efficiently. The main code must merely be adapted to the implementation choices that we have made for *open* and *closed*.

```
function a-star(start-state: STATE; total-N: int): PATH;
var  current-node, best-node: NODE;
     open: list of NODE; closed: array[total-N] of NODE;
     solution-path: PATH; successor-list: SUCCESSORS;
begin
current-node.state := start-state; current-node.father := nil;
current-node.g-value := 0; current-node.f-value := h(start-state);
initialize-list!(open); insert-front!(current-node, open); {P}
do  not empty?(open) cand not goal-node?(get-first(open).state) →
      best-node := remove-first!(open); closed[i(best-node.state)] := best-node;
      successor-list := find-successors(best-node.state);
      "Actualize open and closed" {P}
od; {P and not B}
if  empty?(open) → a-star := EMPTY_LIST
□  not empty?(open) → "Generate solution path"
fi
end.
```

The operation *"Generate solution path"* remains exactly the same except that the function *get-first* will be used instead of *get-open-best*. The operation *"Actualize open and closed"* is fundamentally different:

```
{P2}
do  not empty?(successor-list) →
     current-node.state := get-first(successor-list);
     current-node.father := best-node;
     current-node.g-value := best-node.g-value
                              + cost(best-node.state, current-node.state);
     current-node.f-value := current-node.g-value
                              + h(remove-first!(successor-list));
     if closed[i(current-node.state)] ≠ nil cand
        closed[i(current-node.state)].f-value ≤ current-node.f-value → skip
     □ closed[i(current-node.state)] = nil cor
        closed[i(current-node.state)].f-value > current-node.f-value →
        if closed-member = nil → skip
        □ closed-member ≠ nil → closed[i(current-node.state)] := nil
        fi;
        actualize-open(current-node, open)
     fi {P2}
od {P2 and not B2}
```

The function *actualize-open* is the one that performs the scanning of *open* already explained. It is of course possible to define it in the usual iterative style but this requires to use a temporary list that is not needed in the following recursive implementation.

```
function actualize-open(current-node: NODE; open: list of NODE): list of NODE;
begin
if empty?(open) → insert-front!(current-node, open)
☐ not empty?(open) cand get-first(open).state = current-node.state → skip
☐ not empty?(open) cand get-first(open).f-value > current-node.f-value →
   insert-front!(current-node, remove-clone(current-node, open))
☐ not empty?(open) cand get-first(open).state ≠ current-node.state
   cand get-first(open).f-value ≤ current-node.f-value →
   insert-front!(get-first(open), actualize-open(current-node, remove-first!(open)))
fi
actualize-open := open
end.
```

The recursive scheme is sound as the function *remove-first!* is applied to *open* before the recursive call. The only function that must still be defined is *remove-clone*. It returns the list of nodes passed as second argument where the possible clone of the node passed as first argument has been removed.

```
function remove-clone(current-node: NODE; open: list of NODE): list of NODE;
begin
if empty?(open) → remove-clone := open;
☐ not empty?(open) cand get-first(open).state = current-node.state →
   remove-clone := remove-first!(open);
☐ not empty?(open) cand get-first(open).state ≠ current-node.state →
   insert-front!(get-first(open), remove-clone(current-node, remove-first!(open)))
fi
end.
```

2.4.1.8 Efficiency Analysis

The heuristically informed methods are much more difficult to analyze than the blind methods because everything depends of the quality of the heuristic. Let $h(n)$ be the heuristic used and $h^*(n)$ the perfect heuristic that would always return the real cost of the optimal path to the nearest goal. In an ideal world, it would be possible to design $h(n)$ so that it is exactly $h^*(n)$ and the A* algorithm would find the optimal path directly. However, as the location of the goal nodes is not known[7], it is practically impossible to create such a perfect heuristic for practical problems.

[7] The state-space problem would not exists if a path to a goal node was already discovered

For this reason, heuristics are computed from the expert knowledge of the problem and return only an estimation of the real distance. The time complexity of A* is exponential in the worst case, but is polynomial when the heuristic function meets the condition $|h^*(n) - h(n)| \leq O(log\, h^*(n))$ [13]. More problematic than the time complexity is the space complexity of the A*. It is proportional to the size of the search space. Several variants of A* have been invented to solve this problem and this document will focus on the iterative deepening A*, which is the subject of the next section.

Another convenient property of the A* algorithm is that it is *admissible* when $h(n) \leq h^*(n)$ for all nodes n of the state-space [2]. An algorithm is admissible if it always return an optimal solution when a solution exists. Consider for example the problem of finding a path from your home to your office. The flying distance between your position and the office is an example of admissible heuristic. This heuristic can indeed never overestimate the distance as the line is the shortest path between two points. A* is also optimally efficient for any heuristic $h(n)$, meaning that no algorithm employing the same heuristic will expand fewer nodes than A* (except when there are several partial solutions where h exactly predicts the cost of the optimal path) [13].

2.4.2 The IDA* algorithm

The iterative deepening A* (IDA*) was first presented in 1985 [11] and is probably the most famous variant of the A*. It consists in applying the idea of the iterative deepening which works so well in the uninformed world to the A* algorithm. This time, the algorithm will no longer perform successive depth-first search by increasing the depth of the search but by increasing the maximum cost of the solution path.

Initially, the cost is limited to the heuristic estimate of the initial state (h(*start-state*)). At each iteration, the nodes with an f-value greater than the cost limit are ignored (we consider here that the heuristic is admissible and that those nodes can thus not be on a solution path of the length of the cost limit) and all the remaining nodes are visited in depth-first manner. When no solution is found at a precise iteration, this proves that no solution of length equal to the cost limit exists. The cost limit can be increased and a new iteration starts. When a solution exists, the cost limit will eventually reach the value of the optimal solution path and the latter will be found.

The implementation is the same as for the iterative deepening except that we will no longer call the *limited-depthfirst* function at each iteration but a function *f-limited-depthfirst* that we will define. An integer *current-limit* will again represent the current cost-limit. It will this time be initialized to h(*start-state*) $- 1$ and increased by one at the beginning of each iteration. The variable *current-solution* represents still the potential solution of the current limit. Initially it is the empty list because a solution of cost h(*start-state*)-1 cannot exist as we consider that the heuristic is admissible. It will obviously remain an empty list until the last iteration, i.e. when *current-solution* becomes the solution of optimal cost and the guardian B becomes false.

2.4.2.1 Invariant

P: *current-solution* is a solution path of a cost of *current-limit* (empty list if no such solution exists).

2.4.2.2 Program

```
function iterative-deepening-a-star(start-state: STATE): PATH;
var  current-limit: integer;
     current-solution: PATH;
begin
current-limit := h(start-state)-1; current-solution := EMPTY_LIST; {P}
do  empty? current-solution →
    current-limit := current-limit + 1;
    current-solution := f-limited-depthfirst(start-state, current-limit) {P}
od; {P and not B}
iterative-deepening-a-star := current-solution
end.
```

The heart of the program is the function *f-limited-depthfirst*. Its implementation is based on a list *open* which contains the generated but not visited nodes. Initially, it contains only the initial state of the game. At each iteration, if the first node of *open* is a goal node the algorithm terminates. Else, this nodes is removed from *open* and its successors that have an f-value smaller or equal than *current-limit* are added to the front. This process terminates when *open* is empty (no solution within the current limit) or a solution has been found (the goal node is then the first node of *open* and the solution path can be constructed in the same way as in the A* algorithm).

The invariant of the function *f-limited-depthfirst* will be constructed with the same coloring of the nodes as for the depth-first search algorithm, except that we consider that the nodes that have an f-value greater than the cost-limit are all colored black before the algorithm starts. As usually, this coloration is conceptual; in this case its only purpose is to model the fact that those nodes are ignored from the very start. Initially, the start state of the problem is the only blue node. At each iteration, if the first node of *open* (the deepest blue node of the state-space tree and in case of a tie the leftmost one) is a goal node the algorithm terminates. Else, it depends on whether this node is:

- An inner node that has at least one white successor: the node is colored green and its white successors are all colored blue.
- A leaf or an inner node whose successors are all black: the node is colored black as well as all its ancestor that have no blue brother.

This gives the following invariant:

P_2: *open* is composed of all the blue nodes of the state-space (ordered in favor to the deepest and in case of a tie the leftmost ones). The green nodes are the ancestors of the blue nodes. The black nodes are the left brothers of the green nodes and of the deepest and leftmost blue node as well as all the nodes that have an f-value greater than *current-limit*. The white nodes are the offsprings of the blue nodes that are not black.

The guardian B_2 must remain true as long as a termination case has not been reached, i.e. as long as *open* is not empty (a solution may still exist) and the first node of *open* is not a goal node (a solution have not been found yet).

```
function f-limited-depthfirst(start-state: STATE; current-limit: int): PATH;
var  current-node, succ-node: NODE;
     open: list of NODE;
     successor-list: SUCCESSORS;
begin
current-node.state := start-state; current-node.father := nil;
current-node.g-value := 0; current-node.f-value := h(start-state);
initialize-open!(open); insert-in-open!(current-node, open); {P2}
do  not empty?(open) cand not goal-node?(get-first(open)) →
      current-node := remove-first!(open);
      successor-list := find-successors(current-node.state);
      "Add all the successors that have an f-value less or equal than
      current-limit in front of open" {P2}
od; {P2 and not B2}
if  empty?(open) → f-limited-depthfirst := EMPTY_LIST
□  not empty?(open) → "Generate solution path"
fi
end.
```

The operation *"Generate solution path"* remains exactly the same as for the A* algorithm except that the function get-first will be used instead of get-open-best. The function *"Add all the successors that have an f-value less or equal than current-limit in front of open"* can be concretized by a simple loop. At each iteration, the first member of *successor-list* is removed. Recall that the members of the latter are not nodes but states of the game. The node corresponding to the removed state of *successor-list* must thus first be created. If its f-value is greater than *current-limit* then the node can be ignored. Else, the created node is inserted in front of *open*. We can once more consider that the successors of *successor-list* are ordered from right to left and note $open_0$ the content of *open* before the loop. This gives the following invariant:

> P_3: *successor-list* contains the states of consecutive successors of *current-node* (ordered from right to left). *open* contains $open_0$ where all the successors of *current-node* whose states are not in *successor-list* except those that have an f-value greater than *current-limit* has been added in front of the list (ordered from left to right).

In fact, before the loop, *successor-list* contains all the successors of *current-node* and *open* is exactly $open_0$. At each step the first state of *successor-list* is removed and if and only if the f-value of the corresponding node is less are equal than *current-limit* the node is added to the front of *open*. The guardian B_3 is the usual one for the actualization of open (true as long as *successor-list* is true).

```
{P3}
do  not empty?(successor-list) →
      succ-node.state := get-first(successor-list);
      succ-node.father := current-node;
      succ-node.g-value := current-node.g-value
                           + cost(current-node.state, succ-node.state);
      succ-node.f-value := succ-node.g-value
                           + h(remove-first!(successor-list));
      if succ-node.f-value ≤ current-limit → insert-front!(succ-node, open)
      □  succ-node.f-value > current-limit → skip
      fi {P3}
od {P3 and not B3}
```

2.4.2.3 Termination

The IDA* algorithm terminates if only if a goal node exists in the state-space tree for the same reason as the iterative deepening algorithm.

2.4.2.4 Efficiency Analysis

The IDA* is more efficient than the A* algorithm for the following reasons:

- Each iteration is a depth-first search. The space complexity is thus $O(d^2)$ where d is the depth of the tree scanned in the last iteration, i.e. the tree composed of the nodes of the state-space that have an f-value which is less or equal than the cost of the optimal solution path. This is considerably smaller than the linear space complexity of the A*.
- A simple list *open* is used in the IDA* algorithm whereas more complicated structure are needed for the A*.
- The use of a *closed* set is not absolutely necessary as most of the clones have already been eliminated because of their too high f-value.
- In the last iteration of the IDA* algorithm, the number of generated nodes is potentially much less than for the A* algorithm as all the nodes with an f-value greater than the cost of the optimal solution path have been pruned out of the tree.

Note that the regenerating of the same nodes in the different iterations is again virtually negligible as it was the case for the iterative deepening. Note also that the IDA* algorithm is admissible and optimally efficient in the same sense as the A* algorithm and that the quality of the heuristic plays the same crucial role. The only drawback of the IDA* compared to the A* is that it cannot be used when the existence of a solution is not assured (only if we limit the resources of the program but the program will be very inefficient to determine that no solution exists).

Chapter 3

The Game of Sokoban

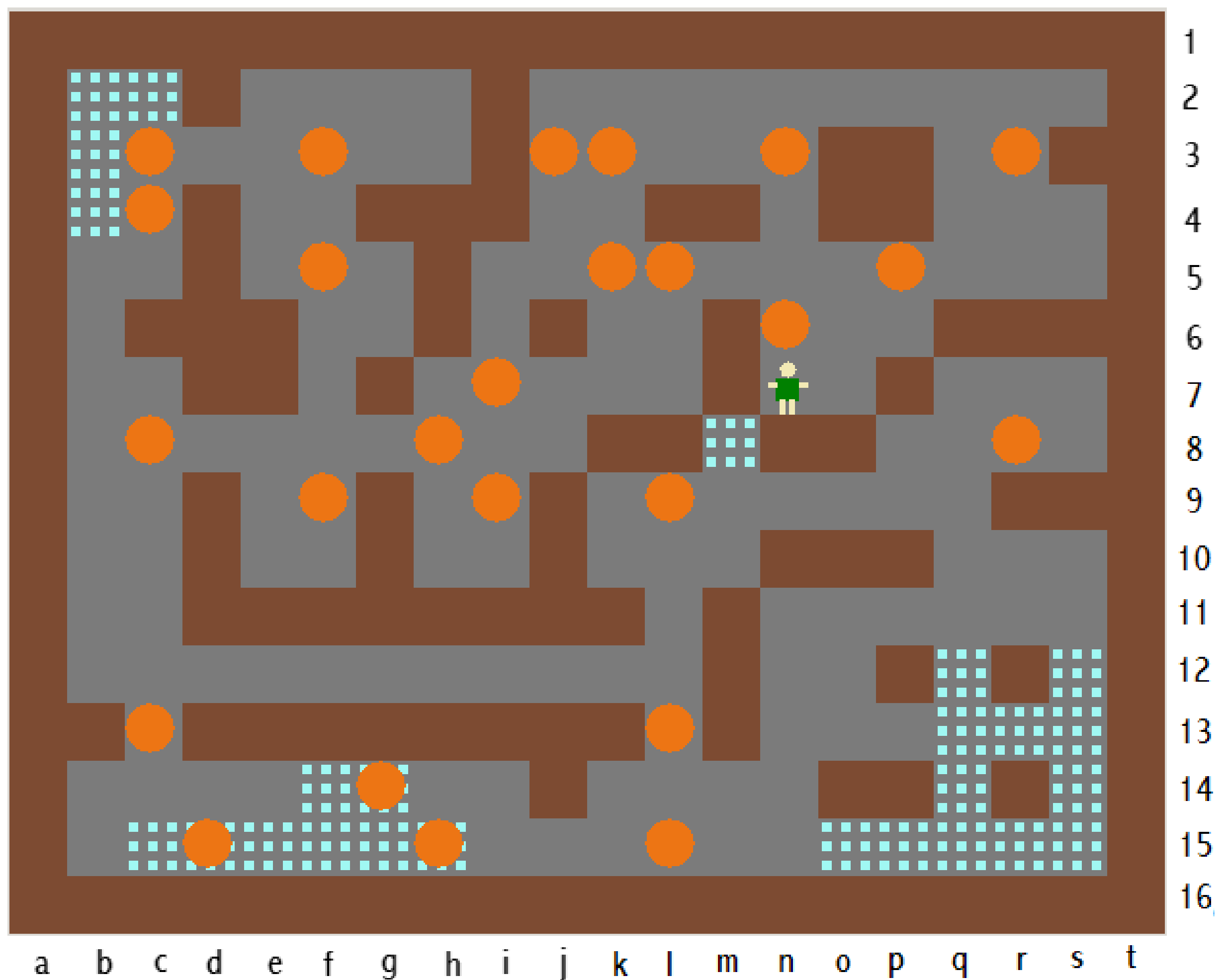

Figure 3.1: The last maze of our 90-problem benchmark

The game of Sokoban was created in 1982 by *Thinking Rabbit*, a computer games company in the town of Takarazuka (Japan). The game was invented by *Hiroyuki Imabayashi*. In Japanese terms, the word *Sokoban* means a warehouse man. Because of the clearness and beauty of the rules, and the intellectually challenging complexity of the composed problems, Sokoban quickly became a popular pastime. Several versions of the game appeared over the years, among which are PC, Macintosh and Unix versions.

3.1 Rules and Consequences

The rules of the game have already been mentioned in the introduction. Recall that a Sokoban maze is a grid composed of unmovable walls, free squares, exactly one man, and as many stones as goal squares. The player controls the man and the man can only push stones (not pull). Furthermore, only one stone can be pushed at a time. The objective of the game is to push all stones on goal squares. Figure 3.1 is a very difficult example taken from our benchmark.

3.1.1 Notation

In the introduction, we have given an example of solution for the simple case of the problem of Figure 1.2. We have introduced two different notations and from now we will keep the shorter one: the stone-move notation. Recall that it consists in giving the lists of the stone-moves of the solution and that it makes the implicit hypothesis that each stone-move is valid, i.e. that the man can reach the square adjacent to the stone to effectively make the push.

To be more precise, each square of the maze will be identified using a coordinate notation. We have chosen to use a convention inspired from the game of Chess: the vertical axis will be labelled with numbers (from 1 to the vertical size of the game) and the horizontal axis with letters (in the alphabetic order beginning with a) starting in the upper left corner. The notation that represents the moves of a particular stone is merely the sequence of the coordinates of the squares through which the stone progresses. For example, the only way to move the $n6$ stone of the maze of Figure 3.1 to the $m5$ square is written: *n6-n5-m5*. The global notation is just an extension of this: as long as the same stone is moved we keep using the notation that has just been introduced and as soon as another stone is moved we start a new sequence separating both by a comma. For example, moving the $n3$ stone of the maze of Figure 3.1 to $n2$ and then the $j3$ stone to $j5$ is written: *n3-n2, j3-j4-j5*.

3.1.2 Difficulty of the game

With such simple rules, the game seams to be simple too. In [9] the following reasons explains why Sokoban is such a difficult game:

- The combination of long solution lengths (from 97 to 674 stone pushes in the test set) and potentially large branching factor (up to 136) make Sokoban difficult for conventional algorithms to solve. The size of the search space for 20×20 Sokoban mazes has been estimated at 10^{98}.
- Sokoban solutions are inherent sequential; only limited parts of a solution are interchangeable. Subgoals are often interrelated and thus cannot be solved independently. Attempts to decompose problems are also ineffective. For example, removing a single stone from a problem may make it trivial to solve, offering no insights as how to solve the problem.
- A simple and effective lower bound on the solution length of a Sokoban problem remains elusive. The best lower-bound estimator is expensive to calculate, and is often ineffective.

- The underlying structure of Sokoban can be represented by a direct graph, meaning that some moves are not reversible. Consequently, there are deadlock states from which no solution can be reached.

In fact, the problem of the existence of deadlock states is the heart of the game of Sokoban. Even one single wrong move can make the task of the player impossible. For example, if a stone is pushed into a corner it is paralyzed forever and the maze becomes unsolvable (recall that all the stones must reach a goal square and that a single deadlocked stone makes the whole maze unsolvable).

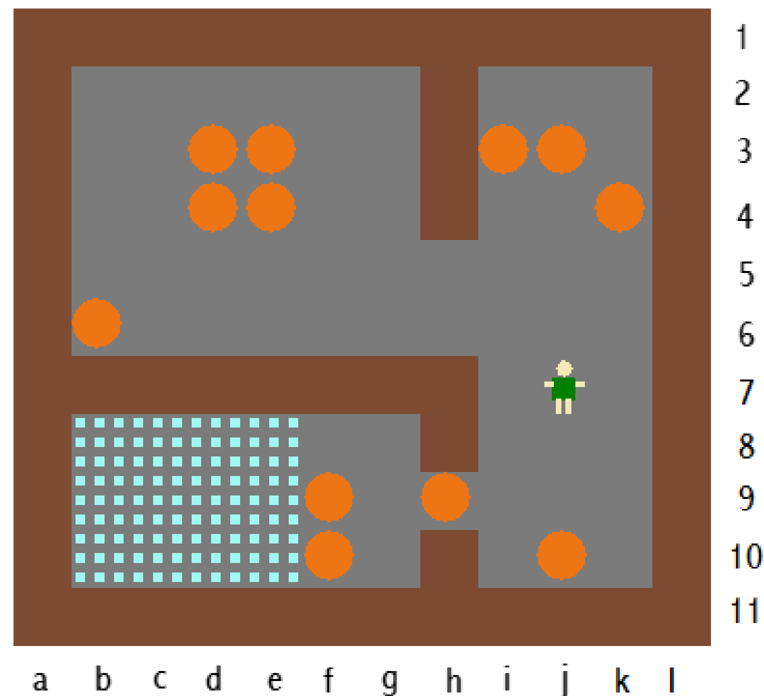

Figure 3.2: Examples of deadlocks

Figure 3.2 shows a palette of deadlock examples. The stones $b6$, $d4$, $d3$, $e4$ and $e3$ are unmovable forever and are clearly deadlocked. The stones $f10$, $f9$ and $h9$ would be solvable if the man was on square $g9$ (*f9-e9-d9-c9-b9, f10-e10-d10-c10-b10, h9-i9-j9-i9-h9-g9-f9-e9-d9-c9-c8-b8* would park them properly in the goal area). However, the man is on $j7$ and can only push the stone $h9$ to $g9$ blocking the entrance of the goal area forever (the man cannot push two stones at a time). This examples shows that the position of the man is also of importance in the context of deadlocks.

Even if we would remove the stones $f10$, $f9$ and $h9$ the stones that have not been mentioned yet would still be deadlocked. The stones $i3$, $j3$ and $k4$ can indeed only move up and the man has therefore no chance to bring them down to the goal area (the stone $k4$ can go down if we first move the stone $j3$ up but the latter would be deadlocked forever and the former would not even become solvable as it is against the border and cannot move to the left). The stone $j10$ would require to reach the 9th rank to become solvable but can only move laterally.

3.1.3 Optimality

Solving Sokoban mazes may be very challenging but assuring that the solution is optimal is even more ambitious. However, the definition of optimality must first be selected. Two possibilities are indeed to consider: we can either want to minimize the number of man-moves or the number of stone-moves. It is uncommon to achieve both target simultaneously. The solution of Figure 1.2 for example was man-moves optimal but not stone-moves optimal. It was indeed possible to do without the stone move g4-h4 at the beginning of the solution but this would have lead to more man motion. We have chosen the stone-moves optimality as it suits our multi-agent modelling approach very well. Indeed, considering that the agents are the stones of the maze, we can merely define the optimal solution as the one that minimizes the number of agents moves.

3.1.4 Efficient Representation of a Maze

Representing the maze simply by an array of the contents of the squares of the maze is not sufficient as it would require to scan the complete array each time the location of the man or the stones are needed. In this work we will therefore represent the game by a record of four elements:

- wall-vector: a boolean vector where each element is true if and only if the corresponding square of the board is a wall. The ith element of the vector corresponds to the ith square of the board counting them in the reading order (from left to right and from up to down) starting with zero in the upper left corner.
- man-coord: the coordinates of the man.
- stone-list: the list of the coordinates of the stones.
- goal-list: the list of the coordinates of the goal squares.

The idea of this representation is to separate the information that does not change in the course of the game (wall-vector and goal-list) from the variable data (man-coord and stone-list). Deciding which moves the man can do will take time $O(1)$ to find the location of the man, $O(1)$ to decide if the adjacent squares are walls or not and for those who are not walls $O(N_{stones})$ to find out if there is a stone on the square or not. As the number of stones is considerably smaller than the total number of squares this complexity is a good improvement over the simple array representation.

Note that we could even have achieved $O(1)$ for deciding if a square contains a stone by using an additional variable stone-vector similar to wall-vector for the stones. However, this vector would require an update each time a stone-move is tried in the resolution process. This update is not so worrying but more problematic is the fact that we will have to backtrack when the tried moves do not lead to a solution which requires to keep plenty of old versions in memory.

Deciding which stones are movable and what moves they can do is also efficient: it consists in scanning the list of the stones which takes $O(N_{stones})$ and for each to test which adjacent squares are accessible ($O(1)$ to see if it is not a wall and $O(N_{stones})$ to see if it is not a stone) and which of those can be reached by the man (we will investigate this question later).

For each stone, deciding whose goal square is the nearest one will take time $O(N_{goals})$. It would have been much more if we had chosen to use a variable goal-vector similar to wall-vector for the goals. The choice of a list representation is also judicious considering that we plan to decide in advance in which order the goal squares will be filled and that it is obviously easier to achieve it from a list representation than from an array representation. We can now define more formally the data type that will be used for representing Sokoban positions:

```
type GAME-INFO = record
                     wall-vector: array[N_squares] of boolean;
                     man-coord: COORD;
                     stone-list: list of COORD;
                     goal-list: list of COORD
                 end;
```

The data type COORD corresponds to the coordinates of a square as defined in the notation section, and can formally been defined as follows:

```
type COORD = record
                 x: char;
                 y: int
             end;
```

We also need data types for representing stone-paths and Sokoban solutions. The former are list of the coordinates of the squares through which the stones must go and the latter are list of stone-path:

```
type STONE_PATH = list of COORD.

type SOKOBAN_SOLUTION = list of STONE_PATH.
```

3.2 State of the Art

State-of-the-art solvers model Sokoban as a state-space problem. The states of the graph are all the possible states of the game which can be obtained by varying the position of the man and the stones. The arcs represent legal one-square stone-moves from a position to another. In [8] the best overall program is stated to be *deepgreen* which could solve 62 problems of the benchmark. However, no details are known about this program (no publication, no open source) and the author assumes that *deepgreen* builds on efforts of the strong Japanese Sokoban community.

Rolling Stone is the actual best documented Sokoban solver. It uses the Iterative Deepening A* as basis for exploring the state-space. In [10], the fact that only trivial problems can be solved by using the IDA* without other enhancements is demonstrated (even with a clever heuristic function, no maze of the difficult 90-problem benchmark can be solved). A lot of enhancements like transposition table, move ordering, deadlock tables, pattern search, and further problem-dependent improvements where implemented to achieve good performances (59 problems solved) [8]. However, this strategy seems to have reached its limits as the most difficult instances are still far from being solved by such methods.

Chapter 4

The New Multi-Agent Modelling Approach

This chapter presents our new multi-agent modelling approach on the real case of the game of Sokoban. A particular class of Sokoban problems will be defined and we will show that our method solves all the problems of that class. We will first give a high-level description of our solving protocol. Then we will continue with a precise and efficient way to implement it.

We will then show that the class which has been solved is unfortunately very particular and that only one problem of our benchmark can be solved by our protocol. However, we will than present a way to embed the latter in a classical state-space algorithm and achieve better results. Finally, a generalization and future extensions of our modelling method will be given.

4.1 Solving a Particular Subclass of Problems

Let us define a particular subclass of Sokoban mazes which have been completely solved by the new method with a rather simple protocol. To be more precise, we will write a protocol that solves mazes if (but not only if) they belong the class. We can thus apply this protocol to any Sokoban maze but we have the guarantee to find a solution only if it is in the class.

4.1.1 Definition of the Subclass

Intuitively, a maze is in this class if the stones are solvable one by one. To be more precise, the maze must satisfy the following conditions:

- *Goal-ordering-criterium:* it must be possible to determine in advance the order in which the goal squares will be filled without introducing deadlocks, independently from the position of the stones and the man.
- *Solvable-stone-existence:* it must be possible to bring at least one stone to the first selected goal square without having to move other stones.
- *Recursive-condition*: for each stone which satisfies the previous condition, the maze obtained by removing that stone and replacing the selected goal square by a wall must also by in the class.

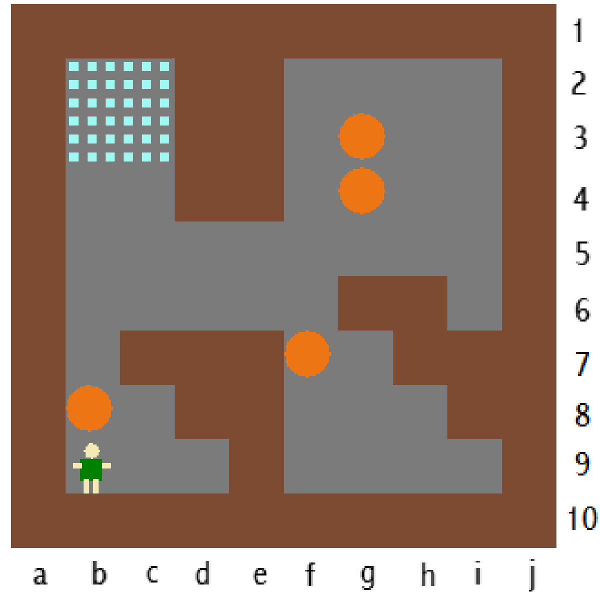

Figure 4.1: An example of maze that is in the defined subclass.

The maze of Figure 4.1 is a Sokoban problem which satisfies all the conditions:

- The following filling order is adequate: $b2$, $c2$, $b3$ and finally $c3$.
- The $b8$ stone is the only one that can reach the square $b2$ without requiring other stone-moves.
- The maze obtained by removing the stone $b8$ and replacing the square $b2$ by a wall is obviously also in the class. This comes from the fact that each remaining stone can reach each remaining goal. The order in which the stones will be solved can thus be chosen freely.

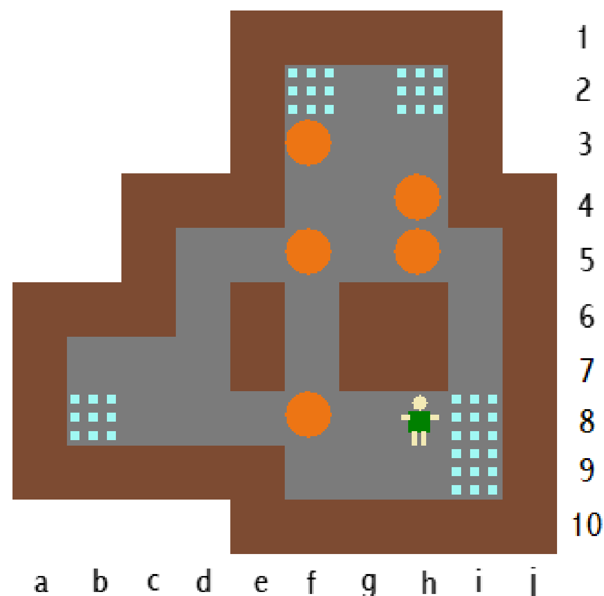

Figure 4.2: An example of maze that is not in the defined subclass.

The maze of Figure 4.2 is not in the class. It is a good compilation of non-satisfied conditions. The goal squares are divided into several goal areas and it is not possible to determine in advance the order in which the goal squares should be filled without considering the position of the stones and the man.

Furthermore, in the initial position, only one stone can reach some goals without other stone-moves (the stone $f8$ can reach $b8$ and $i8$). However, removing the stone and replacing the square $i8$ by a wall would make the goal square $i9$ unreachable. Even by replacing the square $b8$ by a wall, the corresponding maze would still not be in the class. Indeed, none of the four remaining stones can reach a goal without other stone moves.

4.1.2 Protocol for Solving the Mazes of the Subclass

The idea of the new multi-agent modelling approach is that every stone of the maze can be seen as an agent whose aim is to reach one of the goal squares, and the global goal is to find a solution for which everyone achieves his objective. Instead of facing each other, the agents have thus to collaborate to achieve a common objective. In this view of the problem, the man is only a puppet which can be called by the stones when they want to be pushed.

Due to the *recursive-condition*, solving an agent cannot introduce a deadlock situation. This facilitates the elaboration of the algorithm. Indeed, we do not have to write a communication protocol that an agent must initiate when he can reach the first goal square to determine if it could cause damage to other agents. From this, a simple protocol can be used for finding a non-optimal solution:

1°) If no agent (stone) exists, the maze is already solved and the solution path is the empty list.

2°) Else, select the first goal square.

3°) Select an agent that was not chosen before and compute if it can reach the selected goal square.

- If it cannot, return to step 3.
- If it can, move the agent to the goal square, replace the goal square reached by the agent by a wall and delete the agent. Go back to step 1. The solution path is the solution path of the deleted agent added in front of to the recursive solution path produced by the result of step 1.

This protocol is easy to modify for obtaining shorter solutions at the cost of more computation time. In step 3, the first solvable agent will no longer be chosen directly. Instead, the solvability of all the agents will be computed and the one that has found the shortest path to the goal square be selected. By this means, a situation in which a stone has to make a detour around another stone to reach a goal square cannot occur anymore. The stone that would have been on the way would have already been solved. This strategy does not guarantee optimality (counterexamples can be constructed easily) but practice shows that in most real cases it leads to optimal or near-optimal solutions.

The termination of the algorithm is assured for mazes that are in the subclass, as the *solvable-stone-existence* condition assures that it will always exist at least one agent that can reach the first goal square at step 3. The *recursive-condition* assures that all the mazes of the subclass will be solved by this protocol.

The algorithm could also find an answer for mazes that are not in the class. Consider for example a maze for which the stones are solvable one-by-one but only in a precise order. The algorithm could try this order first by chance and find the solution. It is easy to modify the protocol so that it always terminates. We can simply answer that no such easy multi-agent exists when no more agent that has not been selected before can be chosen at step 3.

4.1.3 Efficient Implementation of the Protocol

In the practical implementation, we will notably have to create a procedure for selecting the first goal square. We will also have to decide how to compute the solvability of a stone efficiently. This will obviously first require to work out if the man can at least reach a square adjacent to the stone to make the push. Proving that it is not the case for the unreachable stones will necessitate a complete exploration of the *zone* controlled by the man, i.e. the set of squares that he can reach. In order to avoid to redo this computation for all the stones, we will from the very start create a vector *ind-zone-vector* whose elements are true if and only if the corresponding squares of the maze are reachable by the man without requiring any stone move.

The idea of the protocol is recursive. The basic case occurs when the list of the coordinates of the goal squares is empty. In this case the solution is the empty list. Otherwise the variables *first-goal* (the first goal square) and *movable-stones* (the list of the movable stones) will be computed. Then a loop will repeatedly remove the first stone of *movable-stones* and compute its solvability until the list has been emptied or a solvable stone has been found. The variable *stone-path* will be used for detecting the latter case. It will be initialized to the empty list. At each iteration the solution path found for the current stone will be affected to it. The loop terminates when *stone-path* is a solution-path or when no solution path exists which has been proven when *movable-stones* is empty.

In the latter case the algorithm can return a new constant NO_SOLUTION (we cannot use the empty list as it is our basic solution case). In the former case a recursive call can be performed after having removed the solved stone and solved goal from the corresponding lists and having actualized the position of the man and wall-vector (the solved goal becomes a wall). If the recursive call returns the constant NO_SOLUTION the maze is not in the class and we can return this constant. Else, the solution path is obtained by adding the stone-path found by the loop in front of the other stone moves given by the recursive solution.

Very important is also to ensure that the game data has not been affected by the function. Otherwise some goal squares could still having been replaced by walls even when no solution has been found. For this purpose, and as we do not want to make plenty of copies of the vector, its elements will merely be set back to their old values directly after the recursive call. In contrast, the stone list, goal list and the man coordinates are not large and can be copied. The guardian B of the loop is true as long as *movable-stones* is not empty and the variable *stone-path* is the empty list.

We can now define the invariant of the loop:

P: *movable-stones* is the list of the movable stones for which we do not know if they are solvable or not. *first-stone* is a solvable stone if and only if *stone-path* is not the empty list. All the other stones are known to be unsolvable.

```
function talking-stones(game-info: GAME-INFO): SOKOBAN_SOLUTION;
var  first-goal, first-stone: COORD; stone-path: STONE_PATH;
     movable-stones: list of COORD; new-game-info: GAME-INFO;
     ind-zone-vector: array[N_squares] of boolean;
begin
if  empty?(game-info.goal-list) → talking-stones := EMPTY_LIST
□  not empty?(game-info.goal-list) →
   stone-path := EMPTY_LIST; "Find the first goal square";
   "Find the vector of the squares that are reachable by the man";
   "Find the list of the movable stones"; {P}
   do  not empty?(movable-stones) and empty?(stone-path) →
        first-stone := remove-first!(movable-stones);
        stone-path := "Find a stone-path between first-stone and first-goal"
   od; {P and not B}
   if  empty?(stone-path) → talking-stones := NO_SOLUTION
   □  not empty?(stone-path) →
       new-game-info.stone-list := erase-one(game-info.stone-list, first-stone);
       new-game-info.goal-list := erase-one(game-info.goal-list, first-goal);
       if  empty?(get-rest(stone-path)) →
           new-game-info.man-coord := game-info.man-coord
       □  not empty?(get-rest(stone-path)) →
           new-game-info.man-coord := get-first(get-rest(stone-path))
       fi;
       game-info.wall-vector[coord-to-index(first-goal)] := true;
       new-game-info.wall-vector := game-info.wall-vector;
       rec-solution := talking-stones(new-game-info);
       game-info.wall-vector[coord-to-index(first-goal)] := false;
       if  rec-solution = NO_SOLUTION →
           talking-stones := NO_SOLUTION
       □  rec-solution ≠ NO_SOLUTION →
           talking-stones := insert-front!(stone-path, rec-solution)
       fi
   fi
fi
end.
```

The function *get-rest* takes as argument a list and returns the list without its first element. In contrast to *remove-first!* it does not affect the list. The function *erase-one* returns the list passed as first argument where the element passed as second element has been removed. The function *coord-to-number* takes as argument the coordinates of a square of the maze and returns the corresponding vector-index. All these functions are very easy to implement and it would be wasting time to explain how here. A Scheme implementation of *erase-one* and *coord-to-number* is given in the appendix and *get-rest* is simply the predefined function *cdr*.

The affectation of *new-game-info.man-coord* request an explanation. It depends in fact of *stone-path*. If the latter is a list of a single coordinate this means that the stone was already on the target goal square and that no stone-moves has been performed. In this case the man is still on its starting square and its coordinates remains the same. Otherwise, some stone moves has been performed and the man is located on the square on which the stone was just before having reached the goal square. It is thus the second element of *stone-path*.

4.1.3.1 Computing the Order in which the Goals will be Filled

In the first version of *Talking Stones* presented here, a very simple rule for ordering the goal squares has been chosen. The first goal square will be the one that has the maximum adjacent walls (in case of a tie any of them). The next goal squares are chosen by following the same strategy, but by considering that the previously chosen goal squares are walls. Thus, the goal squares which can only be reached from a single adjacent square will be filled first and we will avoid to block their entrance. This simple criterion has been chosen because it is easily implemented and works for most of the mazes that satisfy the *goal-ordering-criterion*. However, counterexamples can be constructed and more complicated ordering rules will be considered in later versions.

The operation *"Find the first goal square"* can now be implemented without difficulty. When it is called all the preceding solved goals have already been replaced by walls. The idea is thus simply to scan the goal-list and keep the current member with the maximum adjacent walls in memory. Note that when the process reaches a goal with three adjacent walls it can stop directly (it is indeed the maximum because if all the adjacent squares were occupied by walls the maze would be unsolvable and we do not consider such mazes in this work).

The idea of the implementation is to use the following variables:

- *current-best*: the coordinates of the current best goal square. Initially it is the first goal of *game-info.goal-list*.
- *best-value*: the number of adjacent walls to *current-best*.
- *current-list*: the list of the goals that could still be better than *current-best*. Initially it is the list *game-info.goal-list* without its first element.

The guardian B_2 of the loop is true as long as *current-list* is not empty and *best-value* is smaller than three. The following invariant is adequate:

> P_2: *current-best* is the best goal square among the goals which are not in *current-list*.

```
var  current-best, front-goal: COORD;
     best-value, front-goal-value: int;
     current-list : list of COORD;
begin
current-list := get-rest(game-info.goal-list);
current-best := get-first(game-info.goal-list);
best-value := count-walls(current-best, wall-vector); {P2}
do  not empty?(current-list) and current-best-value < 3 →
    front-goal := remove-first!(current-list);
    front-goal-value := count-walls(front-goal, wall-vector);
    if current-best-value < front-goal-value →
       current-best := front-goal; current-best-value := front-goal-value
    □ current-best-value ≥ front-goal-value → SKIP
    fi {P2}
od; {P2 and not B2}
first-goal := current-best
end.
```

The function *count-walls* counts simply how many of the four adjacent squares to the inner-square whose coordinates are passed as first argument are walls, i.e. have true values in the corresponding cells of the wall-vector passed as second argument. Again, this function is very easy to implement and it would be wasting time to explain how here. A Scheme implementation is given in the appendix.

4.1.3.2 Computing the vector of the man-reachable squares

As most of the squares of the board are unreachable (the walls, the stones, the outside squares, some of the inside squares, ...) we will first initialize *ind-zone-vector* with *false* values. We will suppose that a predefined function *initialize-array!* which initializes all cells of the array passed as first argument to the boolean value passed as second argument is given. This operation is performed in time $O(1)$ for most languages.

Initially, one square is for sure reachable by the man, i.e. its actual square. The corresponding cell of *ind-zone-vector* can thus be directly set to *true*. The easiest way to find all the other reachable squares (without any stone-move) consists in using a depth-first search strategy with cycle-detection. A list *open* of the coordinates of known reachable-squares will be used. Initially it contains only the current coordinates of the man. At each iteration, the first element of *open* is extracted and its successors (i.e. the squares that the man can reach in one step) are computed. The successors whose corresponding cell of *ind-zone-vector* are already *true* can be ignored (hence the cycle-detection). The others will be set to *true* and their coordinates added in front of *open*. The algorithm terminates when *open* is empty. This will eventually occur as exactly one element is removed from *open* at each iteration and only a finite number of elements can be added (the man can only reach a finite number of squares and as we detect cycles the same coordinates cannot be added twice).

The main difference between this algorithm and the state-space algorithm that performs a depth-first search with cycle detection is that the latter ends directly when it finds a goal square and the former always generates all the offsprings. The virtual coloring of the nodes for constructing the invariant is thus different:

- White nodes: nodes that have not yet been generated by the search.
- Blue nodes: nodes that have been generated but not *expanded*. A node is said to be expanded if its successors have been generated or if we now that it has no successors.
- Green nodes: nodes that have been generated and expanded.

Initially, the node corresponding to the actual position of the man is the only blue node and all the other nodes are white. At each iteration, the deepest blue node is colored green (in case of a tie the leftmost one) and its white successors as well as their clones are colored blue. The algorithm terminates when all the nodes have been painted in blue. The guardian B_2 of the loop is true as long as *open* is not empty. The strategy of the algorithm can now be expressed as maintaining the following invariant:

> P_2: *open* contains all the coordinates of the squares whose corresponding nodes are blue (ordered in favor to the deepest blue nodes, and in case of a tie the leftmost ones). The cells of *ind-zone-vector* are *true* if and only if the corresponding nodes are blue or green.

The function *"Find the vector of the squares that are reachable by the man"* can finally be written as follows:

```
var  open, successors-list: list of COORD;
begin
initialize-array!(ind-zone-vector, false);
initialize-list!(open); insert-front!(game-info.man-coord, open);
ind-zone-vector[coord-to-number(first-square)] := true; {P2}
do  not empty?(open) →
    successor-list := find-new-successors(remove-first!(open), ind-zone-vector,
                              game-info.wall-vector, game-info.stone-list);
    open := append(successor-list, open); {P3}
    do  not empty?(successor-list) →
        ind-zone-vector[coord-to-number(remove-first!(successor-list))] := true
        {P3}
    od {P2 and P3 and not B3}
od {P2 and not B2}
end.
```

The function *find-new-successors* returns the list of the squares that the man could reach in one step if the man was on the square whose coordinates are passed as first argument except those whose corresponding cell of the vector *ind-zone-vector* (passed as second argument) are *true* and by taking account of the position of the walls and of the stones (third and forth arguments). This function is trivial to implement and as usual a Scheme implementation is given in the appendix.

The inner loop extracts the successors from the list returned by *find-new-successors* one-by-one and sets the corresponding cells of *ind-zone-vector* to *true* until the list has been emptied. The guardian B_3 of this inner loop is thus true as long as *successor-list* is not empty and the invariant P_3 states that *successor-list* contains the successors whose corresponding cells of *ind-zone-vector* are *false*.

A more subtle difference between this program and the depth-first search algorithm with cycle-detection presented in chapter 2 is that all the clones are colored blue at the same time. This program is thus more efficient as *open* can now never contain clones of already treated nodes.

4.1.3.3 Computing the list of the movable stones

Deciding if a stone is movable consists merely in computing if at least one if its adjacent squares is reachable by the man while the opposite adjacent square is free (is nor occupied by a wall neither by a stone). This is easy to achieve efficiently as the vector *ind-zone-vector* enables to decide if a square is reachable by the man in $O(1)$, the vector *game-info.wall-vector* enables to decide if a square is occupied by a wall in $O(1)$ and the list *game-info.stone-list* enables to decide if a square is occupied by a stone in $O(N_{stones})$.

From this, a simple loop solves the problem. An auxiliary variable *current-list* will keep the coordinates of the stones which have not been considered yet. It will be initialized to *game-info.stone-list*. The variable *movable-stones* which must contain the list of the coordinates of the movable stones after the execution of the function will be initialized to the empty list. At each iteration, the first stone will be extracted from *current-list*. If it is movable it will be added in front of *movable-stones* and otherwise nothing has to be done at this iteration. The algorithm terminates when *current-list* is empty which will eventually occur after N_{stones} steps.

For clarity reasons a boolean variable *the-stone-can-move* will be used. It will be set to *true* if the current stone is movable and *false* otherwise. In fact, the condition for testing if a stone is movable is extremely long. Without this variable the code would thus be consequently longer as the condition and its contrary would both have to appear in the *if*-condition.

The guardian B_2 of the loop is true as long as *current-list* is not empty. The strategy of this function can be expressed as maintaining the following invariant:

> P_2: *current-list* contains the coordinates of the stones for which we do not now yet if they are movable or not. *movable-stones* contains the coordinates of the movable stones that are not in *current-list*.

The operation *"Find the list of the movable stones"* performed in the main function *talking-stones* can finally be written as follows:

```
var  front-stone, left-coord, right-coord, up-coord, down-coord: COORD;
     current-list: list of COORD;
     the-stone-can-move: boolean;
begin
current-list := game-info.stone-list;
initialize-list!(movable-stones); {P2}
do  not empty?(current-list) →
    front-stone := remove-first!(current-list);
    left-coord.x := front-stone.x - 1; left-coord.y := front-stone.y;
    right-coord.x := front-stone.x + 1; right-coord.y := front-stone.y;
    up-coord.x := front-stone.x; up-coord.y := front-stone.y - 1;
    down-coord.x := front-stone.x; down-coord.y := front-stone.y + 1;
    the-stone-can-move :=
        (ind-zone-vector[coord-to-index(left-coord)]
        and not game-info.wall-vector[coord-to-index(right-coord)]
        and not member(right-coord, game-info.stone-list)
    or (ind-zone-vector[coord-to-index(right-coord)]
        and not game-info.wall-vector[coord-to-index(left-coord)]
        and not member(left-coord, game-info.stone-list)
    or (ind-zone-vector[coord-to-index(up-coord)]
        and not game-info.wall-vector[coord-to-index(down-coord)]
        and not member(down-coord, game-info.stone-list)
    or (ind-zone-vector[coord-to-index(down-coord)]
        and not game-info.wall-vector[coord-to-index(up-coord)]
        and not member(up-coord, game-info.stone-list);
    if the-stone-can-move → insert-front!(front-stone, movable-stones)
    □ not the stone-can-move → SKIP
    fi {P2}
od {P2 and not B2}
end.
```

4.1.3.4 Finding a stone-path between two squares of a maze

This is a typical path-finding problem, i.e a particular subclass of state-space problems where the question is to find the shortest route between two points of a map. We have seen in chapter 2 that the heuristically informed methods outperforms the blind method when a good heuristic which estimates the remaining distance to the objective is provided.

For this problem we can simply use the manhattan distance between the initial square and the target square. If their coordinates are respectively (i_{col}, i_{row}) and (t_{col}, t_{row}), the manhattan distance is defined as:

$$|i_{col} - t_{col}| + |i_{row} - t_{row}|$$

This heuristic is admissible. Indeed, consider the maze without the walls and the other stones. The stone could in this case reach the target square in a number of step exactly equal to the manhattan distance. It will indeed have to perform $|i_{col} - t_{col}|$ steps to reach the target column and $|i_{row} - t_{row}|$ steps to reach the target row. When the walls and the other stones are present the number of step can only increase (it can even become infinite if the target square is unreachable). The real distance is thus greater or equal than the manhattan distance hence the admissibility of the heuristic.

We have seen that the IDA* algorithm is the best choice when we have the insurance that at least one goal node exists. This is not the case here as the target square could be inaccessible. We will thus use the A* algorithm and the only remaining question is which particular implementation to use. We have seen that this choice depends mainly on the size of the state-space. We will now show that it is very small which means that an implementation with an indexed vector for *closed* and a list for *open* is the best choice as explained in chapter 2 (we will thus use our contribution, i.e. the program given in section 2.4.1.7 which consists in rewriting the abstract code of the A* algorithm for the particular implementation choice).

In the initial position, the stone and the man can be on any square of the board. This is no longer true for the other nodes of the state-space as each successor is obtained by moving the stone one-square ahead and in the resulting position the man must be on an adjacent square to the stone. For a 20×20 maze the stone can be placed on at most 18×18 squares (the borders are walls). The man can only be on 4 different adjacent squares. The number of states (without counting the initial state) is thus clearly bounded by $18 \times 18 \times 4 = 1296$. A smaller bound can even by found by considering the specials cases of a stone in a corner or against a wall. This is not necessary as that would complicate the indexing function unnecessarily.

From this, it is easy to write an indexing function which associates a unique number to each state of the state-space. Let assign the number zero to the initial state. Let *stone-square-number* be the number associated to the square on which the stone is situated: if we exclude the border squares, zero is associated to the upper left corner and the next naturals are associated to the next squares in the reading order. From this, using the following equation for associating the numbers to the states guaranty the unicity: $4 \times$ *stone-square-number* $+$ (1, 2, 3 or 4 depending on the man-position, resp. up, down, left and right from the stone). Note that when the stone is on the downiest right corner the man cannot be on the right square. The maximum number that can be associated is thus 1295 for 20×20 mazes. we can thus use an array in the range 0...1295.

Conceptually, the states of the problem contains the whole information of the corresponding position of the game. However, in our practical case, keeping the position of the stone and of the man is enough. Indeed, the position of the target goal, the walls and the other stones does not change and can be stocked as a constant information independent of the particular node. We can now define the states of the problem:

```
type STONE_PATH_STATE = record
                            stone-coord: COORD;
                            man-coord: COORD
                        end;
```

The operation *"Find a stone-path between first-stone and first-goal"* can be written as follows:

```
var  init-state: STONE_PATH_STATE;
begin
init-state.stone-coord := first-stone;
init-state.man-coord := game-info.man-coord;
a-star(init-state, 4 × (N_col − 2) × (N_row − 2));
end.
```

All what it remains to do is to define the problem-dependent functions used by the A* algorithm. The cost of all arcs will simply be one as each stone-move is equally difficult. The g-value of a node will simply be one more than the one of its father. The h-function is the manhattan distance already defined. The *goal-node?* predicate is simply a predicate which takes as argument a state and the coordinates of the goal node and returns true if and only if the coordinates of the stone are equal to those of the goal.

The function *find-successors* is not so easy to implement. In fact, the vector *ind-zone-vector* can no longer be used for deciding whether a square is reachable by the man or not. Indeed, after each stone move the set of the reachable squares can change completely. A stone-move can for example open a path to a large area of the maze that was not reachable before. Furthermore, in many case the man will be able to reach the push-square and computing all the reachable squares is wasting time. For this reason, we will use the A* algorithm to work out if a path to the push-square exists. The heuristic will again be the manhattan distance, but this time it will be computed on the coordinates of the man and of the push-square. The implementation is similar to the one of the stone-path.

4.1.4 Results

Only one problem of our 90-problem test suite is in the subclass just defined (problem 78) and can be solved by this protocol. This is not surprising, as problems that are directly solvable stone by stone are uncommon in difficult benchmarks. Instead of elaborating other protocols for solving more general problems with the pure new multi-agent modelling approach, another idea has been developed.

As it is often the case in AI, trying to understand how the human player solves problems helps to find new algorithms. In the case of Sokoban, one of the talents of the human player consists in recognizing very soon in the resolution process that he can reach a configuration that is easy to solve (stone-by-stone). This suggests a new solving method for difficult games.

4.2 Embedding the New Approach into a State-Space Algorithm

The method consists in using a classical state-space algorithm, but one in which the nodes whose corresponding state of the game is solvable by the new multi-agent modelling approach are defined as success nodes. This means that when the search reaches such a node, the search terminates successfully. The solution is then obtained by appending the solution path found by the state-space algorithm to the solution found by the multi-agent modelling approach.

The offspring of success nodes are no longer reachable and can be considered to have been pruned out of the state-space. In practice, the size of the state-space will decrease substantially. On the other hand, more computation time is needed at each node as the multi-agent modelling approach is called for each node to determine whether it is a success node. However, the time lost by these calls is largely compensated by the time won by having less nodes to visit. Our program *Talking Stones* implements this idea.

4.2.1 Choosing the Right State-Space Algorithm

The size of the search space remains huge for difficult problems. As the branching factor is rather high, a good memory management is required. Therefore, an iterative deepening algorithm is a good choice. We have thus to choose between the pure iterative deepening algorithm and the IDA* algorithm. The latter is naturally the best choice if we can find a good heuristic.

However, the heuristics commonly used for games are functions that try to minimize a particular distance to the objective. For Sokoban, it is generally the distance between the stones and the goals (using a minimum matching algorithm to assign the stones to the goals). This does not help much in the context of the new method. Indeed, almost every Sokoban mazes start in a configuration where the situation seems to be nearly blocked. The initial strategy consists therefore in finding a few moves to make more space for the man. Those moves have no reasons to be moves that diminish the global stone-goal distance. Defining a good heuristic function is thus far from trivial. For this reason, we have simply used the original iterative deepening algorithm in our first version of *Talking Stones*. We decided to limit the number of generated nodes to 20000.

4.2.2 Results

The algorithm seems rather naive; it simply tests all the possible moves until a position in which the stones are solvable one-by-one is reached. It makes nothing for avoiding deadlock moves and seems thus to be too coarse for solving serious problems. Surprisingly, this strategy is already able to solve 9 problems of the 90-problem test suite.

Problem	Generated Nodes	Depth of the multi-agent solution	Time
1	76	3	6 sec
2	7155	4	8 min 23 sec
3	31	2	5 sec
5	67	2	25 sec
6	2849	4	4 min 20 sec
51	972	4	31 sec
54	2761	3	19 min 4 sec
78	0	0	< 1 sec
82	173	3	12 sec

Table 4.1: Results obtained by Talking Stones on the 90-problem benchmark

4.2.3 Comparison with *Rolling Stone*

Rolling Stone has demonstrated that a pure IDA* approach without other enhancements cannot solve any problem of the benchmark and that immense progress can be obtained with them. Implementing some of those enhancements within *Talking Stones* looks thus promising for solving more instances.

One enhancement needs a special comment: Move Ordering. In *Rolling Stone*, moves which preserves the *inertia* of the stones are favored. This means that if the previous move was performed on a particular stone, moving this stone again will be considered first. From this, if *Rolling Stone* reaches a node that is solvable by the multi-agent modelling approach, it will find a solution similar to the one found by *Talking Stones* (solving the stones one-by-one, but possibly in another order).

However, the big difference is that our program will in this case require much less time and space to find the solution. Indeed, if we consider a maze composed of N stones, *Talking Stones* will simply call the A* algorithm at most N times to find a first solvable stone, $N-1$ times for a second one, and so on. This gives a complexity of N^2 calls to the A* algorithm. Each call is relatively fast as the size of a Sokoban maze is small and so is the number of possibilities. To be more precise, for a 20×20 maze the stone can be placed on at most 18×18 squares (the borders are walls). A stone can move in at most 4 directions. The number of possibilities is thus clearly bounded by $18 \times 18 \times 4 = 1296$. A smaller bound can even be found by considering that a stone cannot move if it is in a corner and can at most move in 2 directions if it is against a wall.

An implementation based on the IDA* will do much more. At each step, it will generate all the possible moves for the N stones. For each position that can be obtained by performing one of those moves, it will compute its heuristic distance to the final goal. Then, it will select the most promising move not tried so far (in case of a tie one that preserves the inertia) and continue from that point. As the branching factor of the game of Sokoban is important, the IDA* algorithm will rapidly have to keep a large amount of candidate nodes to expand in memory. In the case of a maze solvable stone-by-stone, this is not needed as most of the candidate moves will never been tried.

4.3 Generalization of the Method to Other Games

We can decompose the new solving method for difficult single-player games in three layers:

- The high-level layer is a classical state-space algorithm where a node is a success node if the medium layer can solve it. The choice of the algorithm depends on the precise game. If the branching factor is important an iterative deepening algorithm is appropriate. The IDA* should be preferred to the pure iterative deepening algorithm when expert-knowledge of the game is provided and a good heuristic function can be constructed. If the branching factor is small, the memory is not critical and the A* algorithm is indicated. When no good heuristic function can be constructed, the breadth-first search algorithm is the best suited one at this level. See [17] for more information on the most adapted state-space algorithms according to the branching factor.
- The medium-level is a protocol based on a multi-agent representation of the game. The agents are primitive game elements depending on the particular game. The agents have to communicate together to find a common solution. They can use the algorithms of the low-level as tools for solving sub-problems.
- The low-level is a set of algorithms for solving subproblems of the game. Classical state-space algorithms can be used but not exclusively.

We believe that it is possible for almost all games to determine primitive game elements that have to reach some goal. In puzzles like the 24-tile puzzle, the agents could be defined as the tiles. In this representation, each tile aims to reach its final destination but cannot move without altering the position of other agents. In the game of Sokoban, all the agents are instances of stones of the maze and have thus the same characteristics. For other games however, we could define agents that have their own personality. For the game of solitaire for example, the agents could be the 52 cards. Each agent is now unique. Note that for such imperfect information game, we must consider that only a subset of the agents is visible. The other agents can thus be seen as being in an unknown queue, waiting for entering into play.

4.4 Extensions of the Method

Our actual implementation is sequential. However, each agent has to solve subproblems independently from the others. It should thus be possible to parallelize the processes. Tests must be done to determine if significant computing time can be won by these means.

The protocol that has been defined in the multi-agent modelling approach has the advantage to be easy to implement and to work well for mazes of the defined subclass. More general protocols that imply a communication process between the agents can presumably be found. Furthermore, the way to combine the state-space algorithms and the multi-agent modelling approach chosen here is not the only possibility. Other approaches are already considered.

After all these enhancements, we hope that the new method will have proven its strength. The next evolution would consist in extending it to multi-player games. In this case, teams of agents would face each other. Take the game of Chess as an example. We could model the game as a war between a white team of agents that collaborate together to checkmate a special black agent (the black king) and try to protect a special white agent (the white king). And inversely for the black team. A second example is the game of Go. Here the goal of the white and black teams of stones would be to control as many areas of the game as possible.

Conclusion

In this work we have presented a new modelling method for single-player games. Our idea has been to model the game as a multi-agent system where the agents are primitive game elements depending on the particular game. We have demonstrated the method on the game of Sokoban. It is important to note that the multi-agent notion which has been introduced is only conceptual and that it does not imply multi-agent programming. Our program is a solver for a single player puzzle. The method that has been presented requests a central solver to decide the order of the stones to be solved. This work presents thus solely a new way to view the problem which leads to new interesting resolution ideas.

We have defined a particular subclass of Sokoban mazes that has been completely solved by a protocol based on our pure multi-agent representation. It is of course possible to write an algorithm which determines efficiently if a maze is in the class. This has not been done here since it was not necessary. Our protocol solves indeed all the mazes of the subclass but can also solve some out of it. Furthermore, we have discovered that even when a Sokoban problem is not in this very particular class, it can often become so after a few moves.

We propose thus a new method. It consists in using a classical state-space algorithm, but one in which the nodes whose corresponding state of the game are solvable by the multi-agent modelling approach are defined as success nodes. This means that when the search reaches such a node, it terminates successfully. The solution is then obtained by appending the solution path found by the state-space algorithm to the solution found by the multi-agent modelling approach. Thus, we have not to test if a maze is in the class, but merely search for a configuration that is solvable by an algorithm based on a multi-agent representation.

At the time being, our program *Talking Stones* is not a contribution in the domain of Sokoban. It solves only 9 mazes of the benchmark whereas the state-of-the-art program *Rolling Stone* solves 59. However, the latter is based on the IDA* algorithm with a lot of really interesting problem-dependent enhancements. These are presented in [10] and it is well explained why each of them contribute to a substantial decrease of the search-tree size. On the other hand, the fact that no problem of the benchmark can be solved with the pure IDA* approach without these enhancements, even with a clever heuristic, is also demonstrated in [10]. We have not implemented these enhancements yet and we plan to inject them within our new method in future works. As *Rolling stone* has enjoyed tremendous progress by adding them to its initial pure IDA* approach (from 0 mazes solved to 59), we hope to benefit from the same kind of progression.

This thesis has also given an original presentation of the classical state-space algorithms. Usually, only a high-level description with abstract data types is provided. Here we have presented all the algorithms with the semi-formal-method proposed in [4]. The invariants specified in this work have not been taken from the literature. They have all been reconstructed from the original idea of the algorithm in order to produce a clear description of the latter and to prove its correctness. This approach has led to a contribution for the A* algorithm. The practical performances have indeed been improved for a particular implementation choice of practical interest.

Appendix - Scheme Implementation of Talking Stones

```
;%%%%%%%%%%%%%%%%%%%
;% I. MAIN FUNCTION %
;%%%%%%%%%%%%%%%%%%%

;solve-sokoban-maze is a function which takes two arguments:

;- maze-file: file containing a Sokoban maze to solve. The file
;must begin with the horizontal and then the vertical size of the
;board. The other lines represents the content of the different
;lines of the maze. A wall is represented by a '+', the man by a
;'@' if it is not on a goal square and '&' otherwise, the stones
;by a '$' if they are not on goal squares and '*' otherwise, an
;empty goal square is written '.' and the empty squares by '=' if
;they are inside the maze and else by a '0'.

;- solution-file: name of the solution file that will be created
;by the function.

;The function has as side effect to create the file solution-file
;containing the list of the man-moves that must be executed for
;solving the maze of maze-file and returns the execution time.

(define solve-sokoban-maze
 (lambda (maze-file solution-file)
  (let ((file-game-info (make-file-game-info maze-file)))
   (let ((wall-vector (car file-game-info))
         (man-coord (cadr file-game-info))
         (stone-list (caddr file-game-info))
         (goal-list (cadddr file-game-info))
         (t0 (runtime)))
    (let ((solution (talking-stones-iterative-deepening
                     (make-sokoban-start-coord man-coord stone-list)
                     (make-sokoban-game-info goal-list wall-vector)
                     sokoban-successors-func sokoban-talking-stones-func 20000)))
     (let ((t1 (runtime)))
```

```
      (begin
       (write-solution-to-file
              (stone-sol-2-man-sol solution man-coord stone-list wall-vector)
              solution-file)
       (- t1 t0))))))))

;%%%%%%%%%%%%%%%%%%%%%%
;% II. INPUT - OUTPUT %
;%%%%%%%%%%%%%%%%%%%%%%

;-----------------------
; accessors and builders
;-----------------------

;Different vectors for stocking information attached to squares
;of the board will be used. Let us define a common framework for
;them yet. The two first elements of all vectors will be the
;number of columns and the number of rows of the maze and the
;other elements the respective information of the squares of the
;board from left to right and up to down. wall-vector for example
;will have true values if and only if the corresponding square of
;the board is a wall.

(define acc-board-vector-col
 (lambda (board-vector)
  (vector-ref board-vector 0)))

(define acc-board-vector-row
 (lambda (board-vector)
  (vector-ref board-vector 1)))

(define acc-board-vector-square
 (lambda (board-vector col row)
  (vector-ref board-vector
   (make-board-vector-index col row (acc-board-vector-col board-vector)))))

(define write-board-vector-square!
 (lambda (board-vector square-coord content)
  (vector-set! board-vector
   (make-board-vector-index
          (car square-coord) (cdr square-coord)
          (acc-board-vector-col board-vector)) content)))

(define make-board-vector-index
 (lambda (col row max-col)
  (+ 2 col (* row max-col))))
```

```
;-------------------------
; read and write functions
;-------------------------

;make-file-game-info is a function which takes one argument:
;filename is a file containing the data of the maze. It returns a
;list whose first element is a wall vector (see above), the second
;element the coordinates of the man (man-col . man-row), the third
;element the list of the stone coordinates and the last element
;the list of the goal coordinates.

(define make-file-game-info
 (lambda (filename)
  (read-game-info (open-input-file filename))))

;read-game-info is a function which takes one argument:
;input-port is an input-port containing the maze data. It returns
;a list whose first element is a wall vector, the second element
;the coordinates of the man (man-col . man-row), the third element
;the list of the stone coordinates and the last element the list
;of the goal coordinates.

(define read-game-info
 (lambda (input-port)
  (let* ((max-col (read input-port))
         (max-row (read input-port))
         (length-vector (+ 2 (* max-col max-row))))
   (let ((wall-vector (make-vector length-vector)))
    (begin
        (vector-set! wall-vector 0 max-col)
        (vector-set! wall-vector 1 max-row)
        (let loop ((index 2)
                   (man-pos '())
                   (stone-list '())
                   (goal-list '()))
         (if (= index length-vector)
             (list wall-vector man-pos stone-list goal-list)
             (begin
              (if (= (modulo index max-col) 2) (read-char input-port))
              (let ((caract (read-char input-port))
                    (coordonnees (cons (modulo (- index 2) max-col)
                                       (quotient (- index 2) max-col))))
               (begin
                (if (eq? caract '#\+)
                    (vector-set! wall-vector index #t)
                    (vector-set! wall-vector index #f))
                (cond ((eq? caract '#\@)
                       (loop (1+ index) coordonnees stone-list goal-list))
```

```
                        ((eq? caract '#\&)
                         (loop (1+ index) coordonnees stone-list
                               (cons coordonnees goal-list)))
                        ((eq? caract '#\$)
                         (loop (1+ index) man-pos
                               (cons coordonnees stone-list) goal-list))
                        ((eq? caract '#\g)
                         (loop (1+ index) man-pos stone-list
                               (cons coordonnees goal-list)))
                        ((eq? caract '#\*)
                         (loop (1+ index) man-pos (cons coordonnees stone-list)
                               (cons coordonnees goal-list)))
                        (else (loop (1+ index)
                                    man-pos stone-list goal-list))))))))))))

;write-solution-to-file creates a file whose name is given as
;second argument and writes the list given as first argument in
;the new created file.

(define write-solution-to-file
 (lambda (ls output-file)
  (let ((output-port (open-output-file output-file)))
   (begin
    (write ls output-port)
    (close-output-port output-port)))))

;------------------------------------------------------------------
; transformation of stone-moves solutions into man-moves solutions
;------------------------------------------------------------------

; stone-sol-2-man-sol is a function which takes 4 arguments.
; - rev-sol is a solution of a Sokoban maze given in the stone-move
;  notation and in reverse order (from the final position of the
;  game to the initial one)
; - man-coord are the initial coordinates of the man
; - stone-list is the list of the stone coordinates
; - wall-vector is a vector of type board-vector where an element is
;   true if and only if the corresponding square is a wall. It returns
;   the man-move solution in the correct order.

(define stone-sol-2-man-sol
 (lambda (rev-sol man-coord stone-list wall-vector)
  (if (null? stone-list)
      (stone-sol-2-man-sol-aux (deep-reverse rev-sol) man-coord wall-vector)
      (begin
       (write-board-vector-square! wall-vector (car stone-list) #t)
       (stone-sol-2-man-sol rev-sol man-coord (cdr stone-list) wall-vector)))))
```

```
(define stone-sol-2-man-sol-aux
 (lambda (sol man-coord wall-vector)
  (cond ((null? sol) '())
        ((null? (cdar sol))
         (if (null? (cdr sol))
             (list man-coord)
             (stone-sol-2-man-sol-aux (cdr sol) man-coord wall-vector)))
        (else
         (let ((stone-coord (caar sol))
               (new-stone-coord (cadar sol)))
          (let ((stone-col (car stone-coord))
                (stone-row (cdr stone-coord))
                (new-stone-col (car new-stone-coord))
                (new-stone-row (cdr new-stone-coord)))
           (let ((push-col (- (* 2 stone-col) new-stone-col))
                 (push-row (- (* 2 stone-row) new-stone-row)))
            (let ((man-path (find-man-path man-coord (cons push-col push-row)
                             (make-man-path-game-info '() wall-vector))))
             (if (null? man-path) (list wall-vector)
             (begin (write-board-vector-square! wall-vector stone-coord #f)
                    (write-board-vector-square! wall-vector new-stone-coord #t)
                    (append (stone-sol-2-man-sol-aux (cons (cdar sol) (cdr sol))
                                     stone-coord wall-vector) man-path))))))))))

;deep-reverse takes a list of lists as argument and returns the
;reversed list where every sublist has also been reversed.

(define deep-reverse
 (lambda (ls)
  (deep-reverse-aux ls '())))

(define deep-reverse-aux
 (lambda (ls acc)
  (if (null? ls) acc
      (deep-reverse-aux (cdr ls) (cons (reverse (car ls)) acc)))))

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; III. Implementation of the A* and the IDA* combined with multi-agent %
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

;***********************************************************
; 3.1. A* implementation (with known maximum nodes amount) *
;***********************************************************

;-----------------------
; accessors and builders
;-----------------------
```

```
; node = ((node-coord . ancestror-number) . (g-value . f-value))

(define acc-node-coord caar)
(define acc-node-ancestror cdar)
(define acc-node-g-value cadr)
(define acc-node-f-value cddr)

(define make-node
 (lambda (node-coord ancestror-number g-value f-value)
  (cons (cons node-coord ancestror-number) (cons g-value f-value))))

(define same-node-pred
 (lambda (node1 node2)
  (equal? (acc-node-coord node1) (acc-node-coord node2))))

; list-of-functions = ((g-func . h-func) (goal-node-pred . successor-nodes-func)
;                       (coord-to-number-func . construct-solution-func))

(define acc-g-func caar)
(define acc-h-func cdar)
(define acc-goal-node-pred caadr)
(define acc-successor-nodes-func cdadr)
(define acc-coord-to-number-func caaddr)
(define acc-construct-solution-func cdaddr)

(define make-list-of-functions
 (lambda (g-func h-func goal-node-pred successor-nodes-func coord-to-number-func
          construct-solution-func)
  (list (cons g-func h-func) (cons goal-node-pred successor-nodes-func)
        (cons coord-to-number-func construct-solution-func))))

;----------------------------------------------
; A* implementation with numbering of the nodes
;----------------------------------------------

; a-star is a function which takes 5 arguments:
; - start-coord: coordinates of the starting node
; - goal-coord: coordinates of the target node
; - game-info: fixed game-data
; - list-of-functions: list of the problem-dependent functions
; - max-nodes: maximum number of nodes ;
; This function implements the A* algorithm which finds the
; shortest path between start-coord and a goal-coord (empty list if
; no such path exist), by holding account of the fixed game-data
;(game-info) and the list of the problem-dependent functions
;(list-of-functions).
```

```
(define a-star
 (lambda (start-coord goal-coord game-info list-of-functions max-nodes)
  (let ((closed-vector (make-vector max-nodes '())))
   (a-star-aux goal-coord
               (list (make-node start-coord -1 0
                     ((acc-h-func list-of-functions) start-coord goal-coord)))
               closed-vector game-info list-of-functions))))

; a-star-aux is a function which takes 5 arguments:
; - goal-coord: coordinates of the target node
; - open-list: sorted list (f-values) of nodes to expand
; - closed-vector: an element is an empty list if the corresponding node
;   was not visited yet and the representation of the node in the other case.
; - game-info: fixed game-data
; - list-of-functions: list of the problem-dependent functions
; This function implements the A* algorithm using open-list and
; closed-vector and by holding account of goal-coord, game-info and
; list-of-functions.

(define a-star-aux
 (lambda (goal-coord open-list closed-vector game-info list-of-functions)
  (cond ((null? open-list) '())
        (((acc-goal-node-pred list-of-functions) goal-coord
                                                  (acc-node-coord (car open-list)))
         ((acc-construct-solution-func list-of-functions) (car open-list)
                                                           closed-vector))
        (else
         (a-star-aux goal-coord
                     (update-open-closed goal-coord open-list closed-vector
                                         game-info list-of-functions)
                     closed-vector game-info list-of-functions)))))

;-------------------------------------------------
; functions to update open-list and closed-vector
;-------------------------------------------------

; update-open-closed is a function which takes 5 arguments:
; - goal-coord: coordinate of the target node
; - open-list: sorted list (f-values) of nodes to expand ;
; - closed-vector: an element is an empty list if the corresponding node
;   was not visited yet and the representation of the node in the other case.
; - game-info: fixed game-data ;
; - list-of-functions: list of the problem-dependent functions

; returns a list new-open-list and has as side effect to modify
; the content of closed-vector. new-open-list is achieved by
; removing the first element of open-list, adding it in
; closed-vector, calculating its successors and for each:
```

```
; * if succ-node is in open-list or closed-vector with a f-value less or
;   equal: succ-node is ignored
; * else, the potential old occurences of succ-node are removed from
; open-list and closed-vector and succ-node is inserted in the
; sorted open-list.

(define update-open-closed
 (lambda (goal-coord open-list closed-vector game-info list-of-functions)
  (let* ((node-current (car open-list))
         (node-number ((acc-coord-to-number-func list-of-functions)
                        (acc-node-coord node-current) game-info))
         (successors ((acc-successor-nodes-func list-of-functions) node-current
                        goal-coord game-info (acc-g-func list-of-functions)
                       (acc-h-func list-of-functions))))
   (begin
    (vector-set! closed-vector node-number node-current)
    (let loop ((succ-list successors)
               (new-open-list (cdr open-list)))
     (if (null? succ-list)
         new-open-list
         (let* ((succ-current (car succ-list))
                (succ-number ((acc-coord-to-number-func list-of-functions)
                               (acc-node-coord succ-current) game-info))
                (closed-vector-member (vector-ref closed-vector succ-number)))
          (if (and (not (null? closed-vector-member))
                   (<= (acc-node-f-value closed-vector-member)
                       (acc-node-f-value succ-current)))
              (loop (cdr succ-list) new-open-list)
              (let ((updated-open (update-open new-open-list succ-current)))
               (if (null? updated-open)
                   (loop (cdr succ-list) new-open-list)
                   (begin
                    (vector-set! closed-vector succ-number '())
                    (loop (cdr succ-list) updated-open))))))))))))

; update-open is a function which takes 2 arguments:
; - open-list: sorted list (f-values) of nodes to expand
; - node: a node
; returns a list, that can be:
; * open-list where node has been inserted (preserving the sort), if the node
;   is not member of open-list
; * the empty list, if the node is in open-list with a less or equal f-value
; * else, the open-list where node has been inserted (preserving the sort)
;   and the old occurence removed.

(define update-open
 (lambda (open-list node)
  (if (null? open-list)
```

```
      (list node)
      (let ((first (car open-list)))
       (cond ((< (acc-node-f-value node) (acc-node-f-value first))
              (cons node (remove-node node open-list)))
             ((same-node-pred first node) '())
             (else
               (let ((rec-open-list (update-open (cdr open-list) node)))
                (if (null? rec-open-list)
                    '()
                    (cons first rec-open-list))))))))

; remove-node is a function which takes 2 arguments
; - node: a node ;
; - node-list: a list of nodes
; returns node-list where the potential node with the same coordinates
; as node has been removed.

(define remove-node
 (lambda (node node-list)
  (cond ((null? node-list) '())
        ((same-node-pred node (car node-list)) (cdr node-list))
        (else (cons (car node-list) (remove-node node (cdr node-list)))))))

;*******************************************************
;3.2. Implementation of the IDA* combined with multi-agent
;*******************************************************

;-----------------------
; accessors and builders
;-----------------------

; id-node = (id-node-coord . id-node-path)

(define acc-id-node-coord car)
(define acc-id-node-path cdr)

(define make-id-node
 (lambda (id-node-coord id-node-path)
  (cons id-node-coord id-node-path)))

;---------------------------------------------------- ;
talking-stones-iterative-deepening implementation
;----------------------------------------------------

; talking-stones-iterative-deepening is a function of 5 arguments:
; - start-coord: coord of the start position
; - game-info: fixed game-data
```

```
; - successor-nodes-func: function that returns the list of the successors
;   of a node
; - talking-stones-func: function that finds a multi-agent solution to the
;   problem if the maze is in the solved sub-class
; - max-nodes: maximum number of nodes that will be generated
; implementation of the iterative deepening algorithm that finds the shortest
; path between start-coord and a node that is solvable by the function
; talking-stones-func by taking account of game-info and successor-nodes-func and
; by limiting the number of visited nodes to max-nodes
;(returns the empty list if it is reached).

(define talking-stones-iterative-deepening
 (lambda (start-coord game-info successor-nodes-func
          talking-stones-func max-nodes)
  (talking-stones-id-aux start-coord (list (make-id-node start-coord '()))
    (list 1) game-info successor-nodes-func talking-stones-func max-nodes 1)))

; talking-stones-id-aux is a function which takes 8 arguments:
; - start-coord: coord of the start position
; - open: list of paths to expand
; - open-elem-sizes: list of the length of the paths of open
; - game-info: fixed game-data
; - successor-nodes-func: function that returns the list of the successors
;   of a node
; - talking-stones-func: function that finds a multi-agent solution to
;   the problem if the maze is in the solved sub-class
; - max-nodes: maximum number of nodes that will be generated
; - pathlimit: the size of the solution must be known to be at least pathlimit
; implementation of the iterative deepening algorithm that increase
; pathlimit iteratively until a node that is solvable by the
; function talking-stones-func is found.

(define talking-stones-id-aux
 (lambda (start-coord open open-elem-sizes game-info successor-nodes-func
          talking-stones-func max-nodes path-limit)
  (cond ((zero? max-nodes) '())
        ((null? open)
          (talking-stones-id-aux start-coord (list (list start-coord)) (list 1)
                              game-info successor-nodes-func talking-stones-func
                              max-nodes (1+ path-limit)))
        (else
         (let ((talking-stones-solution
               (talking-stones-func (acc-id-node-coord (car open)) game-info)))
          (if (equal? talking-stones-solution 'nosolution)
              (if (= (car open-elem-sizes) path-limit)
                  (talking-stones-id-aux start-coord (cdr open)
                    (cdr open-elem-sizes) game-info successor-nodes-func
                    talking-stones-func (-1+ max-nodes) path-limit)
```

```
                  (let ((updated-open (update-open-and-sizes open
                         open-elem-sizes game-info successor-nodes-func)))
                   (talking-stones-id-aux start-coord (car updated-open)
                    (cdr updated-open) game-info successor-nodes-func
                    talking-stones-func (-1+ max-nodes) path-limit)))
               (append-reversefirst talking-stones-solution
                                   (acc-id-node-path (car open)))))))))

;-----------------------------------------------
; function to actualize open and open-elem-sizes
;-----------------------------------------------

;update-open-and-sizes is a function which takes 5 arguments
; - open: list of paths to expand
; - open-elem-sizes: list of the length of the paths of open
; - game-info: fixed game-data
; - successor-nodes-func: function that returns the list of the
;   successors of a node
; - pathlimit: the size of the solution must be known to be at least pathlimit
; Returns the new stack achieved by removing the first element and adding the
; children for which the f-value is <= pathlimit.

(define update-open-and-sizes
 (lambda (open open-elem-sizes game-info successor-nodes-func)
  (let ((successors (successor-nodes-func (car open) game-info)))
   (let loop ((succ-list successors)
              (new-open (cdr open))
              (new-open-elem-sizes (cdr open-elem-sizes)))
     (if (null? succ-list)
         (cons new-open new-open-elem-sizes)
         (loop (cdr succ-list)
               (cons (car succ-list) new-open)
               (cons (1+ (car open-elem-sizes)) new-open-elem-sizes)))))))

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
; IV. Sokoban specific functions that uses the A* and new IDA* %
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

;************************
;* 4.1. Man-path finder *
;************************

;----------------------
; accessors and builders
;----------------------

; man-path-game-info = (stone-list . wall-vector)
```

```
(define acc-man-path-stone-list car)
(define acc-man-path-wall-vector cdr)
(define make-man-path-game-info cons)

;-------------------
; man-path functions
;-------------------

;find-man-path is a function which takes 3 arguments:
; - man-coord: coordinates of the man
; - goal-coord: coordinates of the target
; - man-path-game-info: (stone-list . wall-vector)
; returns the shortest path between man-coord and goal-coord
;(represented by the reverse list of the coordinates of the
;travelling squares, including man-coord and goal-coord) if such
;a path exist, and in the other case the empty list.

(define find-man-path
 (lambda (man-coord goal-coord man-path-game-info)
  (let ((wall-vector (acc-man-path-wall-vector man-path-game-info)))
   (a-star man-coord goal-coord man-path-game-info
          (make-list-of-functions man-path-g-func man-path-h-func
                              man-path-goal-pred man-path-successors
                           man-path-coord-to-number man-path-construct-solution)
          (* (- (acc-board-vector-col wall-vector) 2)
             (- (acc-board-vector-row wall-vector) 2))))))

; man-path-g-func
; all arcs have cost 1

(define man-path-g-func 1+)

; man-path-h-func
; manhattan distance

(define man-path-h-func
 (lambda (start-coord goal-coord)
   (+ (abs (- (car start-coord) (car goal-coord)))
      (abs (- (cdr start-coord) (cdr goal-coord))))))

; man-path-goal-pred
; same coordinates

(define man-path-goal-pred equal?)

; man-path-successors is a function which takes 5 arguments:
; - node: a node
; - goal-coord: coordonate of the target node
```

```
; - man-path-game-info: (stone-list . wall-vector)
; - g-func: g function
; - h-func: h function
; returns the list of the successors of node
;(the other parameters are used for constructing them).

(define man-path-successors
 (lambda (node goal-coord man-path-game-info g-func h-func)
  (append (find-man-successor node goal-coord man-path-game-info
                              g-func h-func 1 0)
          (find-man-successor node goal-coord man-path-game-info
                              g-func h-func -1 0)
          (find-man-successor node goal-coord man-path-game-info
                              g-func h-func 0 1)
          (find-man-successor node goal-coord man-path-game-info
                              g-func h-func 0 -1))))

; find-man-successor is a function which takes 7 arguments:
; - node: a node
; - goal-coord: coordonate of the target node
; - man-path-game-info: (stone-list . wall-vector)
; - g-func: g function
; - h-func: h function
; - a: an integer
; - b: an integer
; returns a list containing the successor of node for
; which the coordinates are those of node on which the integers a
; and b have been added (if this is a valid successor, empty list else)

(define find-man-successor
 (lambda (node goal-coord man-path-game-info g-func h-func a b)
  (let ((node-coord (acc-node-coord node))
        (stone-list (acc-man-path-stone-list man-path-game-info))
        (wall-vector (acc-man-path-wall-vector man-path-game-info)))
   (let ((node-col (car node-coord))
         (node-row (cdr node-coord)))
    (let ((new-col (+ node-col a))
          (new-row (+ node-row b)))
     (let ((new-node-coord (cons new-col new-row)))
      (if (or (acc-board-vector-square wall-vector new-col new-row)
              (member new-node-coord stone-list))
          '()
          (let ((new-node-number (man-path-coord-to-number new-node-coord
                                                     man-path-game-info)))
           (if (equal? (acc-node-ancestror node) new-node-number)
               '()
               (let ((g-value (g-func (acc-node-g-value node))))
                (list (make-node new-node-coord
```

```
                       (man-path-coord-to-number node-coord man-path-game-info)
                       g-value
                       (+ g-value (h-func new-node-coord goal-coord))))))))))))

; man-path-coord-to-number is a function which takes 2 arguments:
; - node-coord: coordinates of a node (node-col . node-row)
; - man-path-game-info: fixed game-data
; (including max-col the number of columns of the maze)
; returns the number of the node of coordinates node-coord,
; achieved by the equation (node-col - 1) + (max-col - 2) * (node-row - 1)
; Example: for a maze of size 5 * 4, here is the classification:
; +++++
; +012+
; +345+
; +++++

(define man-path-coord-to-number
 (lambda (node-coord man-path-game-info)
  (let ((wall-vector (acc-man-path-wall-vector man-path-game-info)))
   (+ (- (car node-coord) 1)
      (* (- (cdr node-coord) 1) (- (acc-board-vector-col wall-vector) 2))))))

; man-path-construct-solution is a function which takes 2 arguments:
; - final-node: a node
; - closed-vector: a vector of nodes
; returns the reversed solution-path obtained by starting with
; final-node and following with its father found in closed-vector
; until the root node is reached.

(define man-path-construct-solution
 (lambda (final-node closed-vector)
  (let ((ancestror-number (acc-node-ancestror final-node)))
    (if (= ancestror-number -1)
        (list (acc-node-coord final-node))
        (cons (acc-node-coord final-node)
              (man-path-construct-solution
                   (vector-ref closed-vector ancestror-number)
                                       closed-vector))))))

;************************
;* 4.2. Stone-path finder *
;************************

;----------------------
; accessors and builders
;----------------------

; stone-path-node-coord = (stone-coord . man-coord)
```

```
(define acc-stone-path-stone-coord car)
(define acc-stone-path-man-coord cdr)
(define make-stone-path-node-coord cons)

; stone-path-game-info = (stone-list . wall-vector)

(define acc-stone-path-stone-list car)
(define acc-stone-path-wall-vector cdr)
(define make-stone-path-game-info cons)

;---------------------
; stone-path functions
;---------------------

;find-stone-path is a function which takes 4 arguments:
; - stone-coord: coordinates of a stone
; - man-coord: coordinates of the man
; - goal-coord: coordinates of the target
; - stone-path-game-info: (stone-list . wall-vector), where stone-list
;   is the list of the stones without stone-coord
; returns the shortest path between stone-coord and goal-coord (represented by
; the reverse list of the coordinates of the travelling squares of
; the stone, including stone-coord and goal-coord, without giving
; the man moves) if such a path exist, and the empty list else.

(define find-stone-path
 (lambda (stone-coord man-coord goal-coord stone-path-game-info)
  (begin
   (let ((wall-vector (acc-stone-path-wall-vector stone-path-game-info)))
    (a-star (make-stone-path-node-coord stone-coord man-coord) goal-coord
            stone-path-game-info
            (make-list-of-functions stone-path-g-func stone-path-h-func
                                    stone-path-goal-pred stone-path-successors
                                    stone-path-coord-to-number
                                    stone-path-construct-solution)
            (* 4 (- (acc-board-vector-col wall-vector) 2)
                 (- (acc-board-vector-row wall-vector) 2)))))))

; stone-path-g-func
; all arcs have cost 1

(define stone-path-g-func 1+)

; stone-path-h-func
; manhattan distance between stone and goal
```

```
(define stone-path-h-func
 (lambda (node-coord goal-coord)
  (let ((stone-coord (acc-stone-path-stone-coord node-coord)))
   (+ (abs (- (car stone-coord) (car goal-coord)))
      (abs (- (cdr stone-coord) (cdr goal-coord)))))))

; stone-path-goal-pred
; - goal-coord: coordinates of the target
; - node-coord: coordinates of a node
; returns true if and only if the stone-coord of node-coord are
; the same as goal-coord

(define stone-path-goal-pred
 (lambda (goal-coord node-coord)
  (equal? goal-coord (acc-stone-path-stone-coord node-coord))))

; stone-path-successors is a function which takes 5 arguments:
; - node: a node
; - goal-coord: coordonate of the target node
; - stone-path-game-info: (stone-list . wall-vector), where stone-list
;   is the list of the stones without stone-coord
; - g-func: g function
; - h-func: h function
; returns the list of the successors of node.

(define stone-path-successors
 (lambda (node goal-coord stone-path-game-info g-func h-func)
  (append (find-stone-successor node goal-coord stone-path-game-info
                                g-func h-func 1 0)
          (find-stone-successor node goal-coord stone-path-game-info
                                g-func h-func -1 0)
          (find-stone-successor node goal-coord stone-path-game-info
                                g-func h-func 0 1)
          (find-stone-successor node goal-coord stone-path-game-info
                                g-func h-func 0 -1))))

; find-stone-successor is a function which takes 7 arguments:
; - node: a node
; - goal-coord: coordonate of the target node
; - stone-path-game-info: (stone-list . wall-vector), where stone-list
;   is the list of the stones without stone-coord
; - g-func: g function
; - h-func: h function
; - a: an integer
; - b: an integer
; returns a list containing the successor of node for which the coordinates
; are those of node on which the integers a and b have been added
; (if this is a valid successor, empty list else)
```

```
(define find-stone-successor
 (lambda (node goal-coord stone-path-game-info g-func h-func a b)
  (let ((node-coord (acc-node-coord node))
        (stone-list (acc-stone-path-stone-list stone-path-game-info))
        (wall-vector (acc-stone-path-wall-vector stone-path-game-info)))
   (let ((stone-coord (acc-stone-path-stone-coord node-coord))
         (man-coord (acc-stone-path-man-coord node-coord)))
    (let ((stone-col (car stone-coord))
          (stone-row (cdr stone-coord)))
     (let ((new-stone-col (+ stone-col a))
           (new-stone-row (+ stone-row b))
           (push-col (- stone-col a))
           (push-row (- stone-row b)))
      (let ((new-stone-coord (cons new-stone-col new-stone-row))
            (push-coord (cons push-col push-row)))
      (if (or (acc-board-vector-square wall-vector new-stone-col new-stone-row)
              (member new-stone-coord stone-list)
              (acc-board-vector-square wall-vector push-col push-row)
              (member push-coord stone-list))
          '()
          (let* ((new-node-coord
                  (make-stone-path-node-coord new-stone-coord stone-coord))
                 (new-node-number (stone-path-coord-to-number new-node-coord
                                                      stone-path-game-info)))
           (if (equal? (acc-node-ancestror node) new-node-number)
               '()
               (if (null? (find-man-path man-coord push-coord
                           (make-man-path-game-info (cons stone-coord stone-list)
                                                    wall-vector)))
                   '()
                   (let ((g-value (g-func (acc-node-g-value node))))
                    (list (make-node new-node-coord
                     (stone-path-coord-to-number node-coord stone-path-game-info)
                     g-value
                     (+ g-value (h-func new-node-coord goal-coord))))))))))))))

; stone-path-coord-to-number is a function which takes 2 arguments:
; - node-coord: (stone-coord . man-coord)
; - stone-path-game-info: fixed game-data
;  (including max-col the number of columns of the maze)
; returns the number of the node of coordinates node-coord, achieved by
; the equation: 4 * [(stone-col - 1) + (max-col - 2) * (stone-row - 1)]
; + (1, 2, 3 or 4 depending on the man-position, resp. up, down, left and right
; from the stone). Notice: the first node is the only one for which the man
; is not for sure next to the stone, this node is labelled 0.
```

```
(define stone-path-coord-to-number
 (lambda (node-coord stone-path-game-info)
  (let ((stone-coord (acc-stone-path-stone-coord node-coord))
        (man-coord (acc-stone-path-man-coord node-coord))
        (wall-vector (acc-stone-path-wall-vector stone-path-game-info)))
   (let ((stone-col (car stone-coord))
         (stone-row (cdr stone-coord))
         (man-col (car man-coord))
         (man-row (cdr man-coord)))
    (let ((n (* 4 (+ (- stone-col 1)
                 (* (- stone-row 1) (- (acc-board-vector-col wall-vector) 2))))))
     (cond ((and (= stone-col man-col) (= stone-row (1+ man-row))) (+ 1 n))
           ((and (= stone-col man-col) (= (1+ stone-row) man-row)) (+ 2 n))
           ((and (= stone-row man-row) (= stone-col (1+ man-col))) (+ 3 n))
           ((and (= stone-row man-row) (= (1+ stone-col) man-col)) (+ 4 n))
           (else 0)))))))

; stone-path-construct-solution is a function which takes 2 arguments:
; - final-node: a node
; - closed-vector: a vector of nodes
; returns the reversed solution-path obtained by starting with
; final-node and following with its father found in closed-vector
; until the root node is reached.

(define stone-path-construct-solution
 (lambda (final-node closed-vector)
  (let ((ancestror-number (acc-node-ancestror final-node)))
    (if (= ancestror-number -1)
        (list (acc-stone-path-stone-coord (acc-node-coord final-node)))
        (cons (acc-stone-path-stone-coord (acc-node-coord final-node))
              (stone-path-construct-solution
               (vector-ref closed-vector ancestror-number) closed-vector))))))

;*******************************
;* 4.3. Sokoban-solution-finder *
;*******************************

;----------------------
; accessors and builders
;----------------------

; start-coord = (man-coord . stone-list)

(define acc-sokoban-man-coord car)
(define acc-sokoban-stone-list cdr)
(define make-sokoban-start-coord cons)
```

```
; sokoban-game-info = (goal-list . wall-vector)

(define acc-sokoban-goal-list car)
(define acc-sokoban-wall-vector cdr)
(define make-sokoban-game-info cons)

;---------------------
; list of id functions
;---------------------

; sokoban-successors-func
; - node: a node
; - sokoban-game-info: (goal-list . wall-vector)
; returns the list of the successors of node.

(define sokoban-successors-func
 (lambda (node sokoban-game-info)
  (let ((node-coord (acc-id-node-coord node))
        (wall-vector (acc-sokoban-wall-vector sokoban-game-info)))
   (let ((man-coord (acc-sokoban-man-coord node-coord))
         (stone-list (acc-sokoban-stone-list node-coord)))
    (let ((ind-zone-vector
           (find-independent-zone man-coord stone-list wall-vector)))
     (let loop ((current-list stone-list)
                (current-successors '()))
      (if (null? current-list)
          current-successors
          (let ((stone-moves (find-stone-moves (car current-list)
                                stone-list wall-vector ind-zone-vector)))
           (if (null? stone-moves)
               (loop (cdr current-list) current-successors)
               (loop (cdr current-list)
                     (append (make-new-successors node stone-moves)
                     current-successors))))))))))

; make-new-successors
; - node: a node
; - stone-moves: a list of the possible moves of one of the stones of stone-list
; (coord-stone (arr1 arr2 ...))
; returns the list of all the new possible successors
; (new-man-coord . new-stone-list)

(define make-new-successors
 (lambda (node stone-moves)
  (let ((stone-coord (car stone-moves))
        (arrival-list (cdr stone-moves)))
   (let* ((node-coord (acc-id-node-coord node))
          (stone-list (acc-sokoban-stone-list node-coord))
```

```
          (cutted-stone-list (erase-one stone-list stone-coord)))
    (let loop ((arrival-squares arrival-list)
               (successors-list '()))
     (if (null? arrival-squares)
         successors-list
         (loop (cdr arrival-squares)
               (cons (make-id-node (cons stone-coord
                                         (cons (car arrival-squares)
                                               cutted-stone-list))
                                   (cons-sokoban-solution stone-coord
                                           (car arrival-squares)
                                           (acc-id-node-path node)))
                      successors-list)))))))

; cons-sokoban-solution is a function which takes 3 arguments:
; - stone-coord: coordinates of a stone
; - arr-coord: arrival coordinates of this stone
; - prev-path: solution path (list of non-empty lists of stone-moves)
; add the stone move between stone-coord and arr-coord to prev-path.
; If prev-path is empty or starts with a stone move of another stone than
; stone-coord, a new deplacement is initiated with stone-coord, else the move
; is added to the current movement list of stone-coord.

(define cons-sokoban-solution
 (lambda (stone-coord arr-coord prev-path)
   (cond ((null? prev-path) (list (list arr-coord stone-coord)))
         ((equal? (caar prev-path) stone-coord)
          (cons (cons arr-coord (car prev-path)) (cdr prev-path)))
         (else (cons (list arr-coord stone-coord) prev-path)))))

; sokoban-talking-stones-func is a function of 2 arguments:
; - start-coord: (man-coord . stone-list)
; - sokoban-game-info: (goal-list . wall-vector)
; returns a talking-stones solution of the problem
; (the constant 'nosolution if no solution exist)

(define sokoban-talking-stones-func
 (lambda (start-coord sokoban-game-info)
  (let ((man-coord (acc-sokoban-man-coord start-coord))
        (stone-list (acc-sokoban-stone-list start-coord))
        (goal-list (acc-sokoban-goal-list sokoban-game-info))
        (wall-vector (acc-sokoban-wall-vector sokoban-game-info)))
   (sokoban-talking-stones-aux man-coord stone-list goal-list wall-vector))))

; sokoban-talking-stones-aux is a function of 4 arguments:
; - man-coord: coordinates of the man
; - stone-list: list of the stone coordinates
; - goal-list: list of the goal coordinates
```

```
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; returns a talking-stones solution of the problem
;(the constant 'nosolution if no solution exist)

(define sokoban-talking-stones-aux
 (lambda (man-coord stone-list goal-list wall-vector)
  (if (null? goal-list)
      '()
      (let ((first-goal (find-first-goal goal-list wall-vector))
            (ind-zone-vector (find-independent-zone man-coord
                              stone-list wall-vector)))
       (let ((possible-moves (find-movable-stones stone-list stone-list
                              wall-vector ind-zone-vector)))
        (if (null? possible-moves) 'nosolution
            (let loop ((movable-stones possible-moves))
             (if (null? movable-stones) 'nosolution
                 (let* ((first-stone (caar movable-stones))
                        (first-stone-path (find-stone-path first-stone
                         man-coord first-goal
                        (make-sokoban-game-info (erase-one stone-list first-stone)
                         wall-vector))))
                  (if (null? first-stone-path)
                      (loop (cdr movable-stones))
                      (let ((new-stone-list (erase-one stone-list first-stone))
                            (new-goal-list (erase-one goal-list first-goal))
                            (new-man-coord (if (null? (cdr first-stone-path))
                                            man-coord (cadr first-stone-path))))
                       (begin
                        (write-board-vector-square! wall-vector first-goal #t)
                        (let ((rec-solution (sokoban-talking-stones-aux
                                   new-man-coord new-stone-list
                                   new-goal-list wall-vector)))
                         (begin
                           (write-board-vector-square! wall-vector first-goal #f)
                           (if (equal? rec-solution 'nosolution)
                               'nosolution
                               (cons first-stone-path rec-solution))))))))))))))))

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;% V. Sokoban useful functions %
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

;********************************
;* 5.1. Independant zone finder *
;********************************
```

```
; find-independent-zone is a function which takes 3 arguments:
; - start-square: the coordinates of a square
; - stone-list: list of the stone coordinates
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; returns a vector of the same size as wall-vector. The two first elements are
; the same and the others are true if the man can reach the corresponding
; square and false else.

(define find-independent-zone
 (lambda (start-square stone-list wall-vector)
  (let ((ind-zone-vector (make-vector (+ 2 (* (acc-board-vector-col wall-vector)
                                      (acc-board-vector-row wall-vector))) #f)))
   (begin
    (vector-set! ind-zone-vector 0 (acc-board-vector-col wall-vector))
    (vector-set! ind-zone-vector 1 (acc-board-vector-row wall-vector))
    (depth-first-zone-search start-square stone-list
                             wall-vector ind-zone-vector)))))

; depth-first-zone-search is a function which takes 4 arguments:
; - start-square: the coordinates of a square
; - stone-list: list of the stone coordinates
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; - ind-zone-vector: a vector of the same size as wall-vector.
;   The two first elements are the same and the others are boolean values
;  (true only for reachable squares from start-square but not necessary for all).
; returns ind-zone-vector where the reachable squares from
; start-square are ALL represented by true.

(define depth-first-zone-search
 (lambda (start-square stone-list wall-vector ind-zone-vector)
  (let ((successors (find-successor-squares start-square stone-list
                                            wall-vector ind-zone-vector)))
   (begin
    (write-board-vector-square! ind-zone-vector start-square #t)
    (let loop ((succ-list successors))
     (if (not (null? succ-list))
         (begin
          (write-board-vector-square! ind-zone-vector (car succ-list) #t)
          (loop (cdr succ-list)))
         (let loop2 ((succ-list2 successors))
          (if (null? succ-list2)
              ind-zone-vector
              (begin
               (depth-first-zone-search (car succ-list2) stone-list
                                        wall-vector ind-zone-vector)
               (loop2 (cdr succ-list2))))))))))
```

```
; find-successor-squares is a function which takes 4 arguments:
; - start-square: the coordinates of a square
; - stone-list: list of the stone coordinates
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; - ind-zone-vector: a vector of the same size as wall-vector.
;   The two first elements are the same and the others are boolean values
;  (true only for reachable squares from start-square but not necessary for all).
; returns the list of the coordinates of the adjacent squares to
; start-square that are nor stones neither walls and for which the
; content of ind-zone-vector is #f.

(define find-successor-squares
 (lambda (start-square stone-list wall-vector ind-zone-vector)
  (append (find-successor-square start-square stone-list
                                 wall-vector ind-zone-vector 1 0)
          (find-successor-square start-square stone-list
                                 wall-vector ind-zone-vector -1 0)
          (find-successor-square start-square stone-list
                                 wall-vector ind-zone-vector 0 1)
          (find-successor-square start-square stone-list
                                 wall-vector ind-zone-vector 0 -1))))

; find-successor-square is a function which takes 6 arguments:
; - start-square: the coordinates of a square
; - stone-list: list of the stone coordinates

; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; - ind-zone-vector: a vector of the same size as wall-vector.
;   The two first elements are the same and the others are boolean values
;  (true only for reachable squares from start-square but not necessary for all).
; - a: an integer
; - b: an integer
; returns the empty list if the square of coordinates ((start-square-col + a) .
;(start-quare-row + b)) is a wall, a stone or is already visited
;(i.e. ind-zone-vector contains #t), and else a list containing
;the index of this square.

(define find-successor-square
 (lambda (start-square stone-list wall-vector ind-zone-vector a b)
  (let ((square-col (car start-square))
        (square-row (cdr start-square))
        (max-col (acc-board-vector-col wall-vector))
        (max-row (acc-board-vector-row wall-vector)))
   (let* ((new-col (+ square-col a))
          (new-row (+ square-row b))
```

```
          (new-coord (cons new-col new-row)))
      (if (or (>= new-col max-col)
              (< new-col 0)
              (>= new-row max-row)
              (< new-row 0)
              (acc-board-vector-square ind-zone-vector new-col new-row)
              (acc-board-vector-square wall-vector new-col new-row)
              (member new-coord stone-list))
          '()
          (list new-coord))))))

;******************************
;* 5.2. Movable stones finder *
;******************************

; find-movable-stones is a function which takes 4 arguments:
; - stone-tested: list of stones for which the movability is tested
; - stone-list: list of coordinates of all the stones of the game.
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; - ind-zone-vector: a vector of the same size as wall-vector.
;   The two first elements are the same and the others are boolean values
;  (true only for reachable squares from start-square but not necessary for all).
; returns the list of all the possible stone-moves from stones of
; stone-tested: ((stone1-coord . (arr1-1-coord ... arr1-n-coord))
;(stone2-coord . (arr2-1-coord ... arr2-n-coord)) ...)
; where stonei-coord is the coordinate of the movable stone and
;(arri-1-coord ... arri-n-coord) the non-empty list of the
;coordinates of the possible arrival squares.

(define find-movable-stones
 (lambda (stone-tested stone-list wall-vector ind-zone-vector)
  (if (null? stone-tested)
      '()
      (let ((first-stone-moves (find-stone-moves (car stone-tested)
                               stone-list wall-vector ind-zone-vector)))
       (if (null? first-stone-moves)
           (find-movable-stones (cdr stone-tested) stone-list
                                 wall-vector ind-zone-vector)
           (cons first-stone-moves (find-movable-stones (cdr stone-tested)
                               stone-list wall-vector ind-zone-vector)))))))

; find-stone-moves is a function of 4 arguments:
; - stone-coord: coordinates of a stone
; - stone-list: list of coordinates of all the stones of the game.
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; - ind-zone-vector: a vector of the same size as wall-vector.
```

```
;   The two first elements are the same and the others are boolean values
;  (true only for reachable squares from start-square but not necessary for all).
; returns an empty list if the stone stone-coord is not movable and
; (stone-coord . (arr-1-coord ... arr-n-coord)) else.

(define find-stone-moves
 (lambda (stone-coord stone-list wall-vector ind-zone-vector)
  (let ((stone-col (car stone-coord))
        (stone-row (cdr stone-coord))
        (max-col (acc-board-vector-col ind-zone-vector)))
   (let ((left-col (-1+ stone-col))
         (right-col (1+ stone-col))
         (up-row (-1+ stone-row))
         (down-row (1+ stone-row)))
    (let ((left-square-nr (make-board-vector-index left-col stone-row max-col))
          (right-square-nr (make-board-vector-index right-col stone-row max-col))
          (up-square-nr (make-board-vector-index stone-col up-row max-col))
          (down-square-nr (make-board-vector-index stone-col down-row max-col)))
     (let ((arrival-list (append (find-stone-move (cons left-col stone-row)
                                   (vector-ref ind-zone-vector right-square-nr)
                                   (vector-ref wall-vector left-square-nr)
                                   (member (cons left-col stone-row) stone-list))
                                 (find-stone-move (cons right-col stone-row)
                                  (vector-ref ind-zone-vector left-square-nr)
                                  (vector-ref wall-vector right-square-nr)
                                  (member (cons right-col stone-row) stone-list))
                                 (find-stone-move (cons stone-col up-row)
                                  (vector-ref ind-zone-vector down-square-nr)
                                  (vector-ref wall-vector up-square-nr)
                                  (member (cons stone-col up-row) stone-list))
                                 (find-stone-move (cons stone-col down-row)
                                  (vector-ref ind-zone-vector up-square-nr)
                                  (vector-ref wall-vector down-square-nr)
                                  (member (cons stone-col down-row) stone-list)))))
     (if (null? arrival-list)
         '()
         (cons stone-coord arrival-list))))))))

; find-stone-move is a function of 4 arguments:
; - square-coord: coordinates of a square adjacent to a stone
; - square-acc: true if and only if the man can reach the square
;   at the opposite of square-coord and adjacent to the same stone.
; - wall-present: true if square-coord is a wall
; - stone-present: true if square-coord is a stone
; returns the list (square-coord) if square-acc is true and arr-square-content is
; free, else the empty list.
```

```
(define find-stone-move
 (lambda (square-coord square-acc wall-present stone-present)
  (if (or (not square-acc) wall-present stone-present)
      '()
      (list square-coord))))

;**************************
;* 5.3. First goal finder *
;**************************

; find-first-goal is a function which takes 2 arguments:
; - goal-list: non-empty goal list
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; let's call w-nbr-i the number of walls adjacent to the ith goal
; of goal-list. The function returns the goal for which w-nbr-i is
; maximum (knowing that it cannot be greater than 3). If several
; goals are maximum, the first one is returned.

(define find-first-goal
 (lambda (goal-list wall-vector)
  (let* ((first-goal (car goal-list))
         (w-nbr-i (find-wall-number first-goal wall-vector)))
   (if (= w-nbr-i 3) first-goal
       (find-first-goal-aux (cdr goal-list) wall-vector
                            (cons first-goal w-nbr-i))))))

(define find-first-goal-aux
 (lambda (goal-list wall-vector current-best)
  (if (null? goal-list) (car current-best)
   (let* ((first-goal (car goal-list))
          (w-nbr-i (find-wall-number first-goal wall-vector)))
    (cond ((= w-nbr-i 3) first-goal)
          ((> w-nbr-i (cdr current-best))
           (find-first-goal-aux (cdr goal-list) wall-vector
                                (cons first-goal w-nbr-i)))
          (else (find-first-goal-aux (cdr goal-list) wall-vector
                                      current-best)))))))

; find-wall-number is a function of 2 arguments:
; - square-coord: coordinates of a square
; - wall-vector: vector of type board-vector, where an element is true
;   if and only if the corresponding square is a wall
; returns the number of walls adjacent to square-coord.

(define find-wall-number
 (lambda (square-coord wall-vector)
  (let ((square-col (car square-coord)) (square-row (cdr square-coord)))
```

```
   (+ (if (acc-board-vector-square wall-vector
                                   (1+ square-col) square-row) 1 0)
      (if (acc-board-vector-square wall-vector
                                   (-1+ square-col) square-row) 1 0)
      (if (acc-board-vector-square wall-vector square-col
                                   (1+ square-row)) 1 0)
      (if (acc-board-vector-square wall-vector square-col
                                   (-1+ square-row)) 1 0)))))

;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;% VI. Other useful functions %
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

; erase-one returns the list passed as first argument where the
; element passed as second element has been removed.

(define erase-one
 (lambda (ls elem)
  (if (equal? (car ls) elem)
      (cdr ls)
      (cons (car ls) (erase-one (cdr ls) elem)))))

; append-reversefirst returns the concatenation of the reverse of
; the list passed as first argument with the list passed as second argument.

(define append-reversefirst
 (lambda (l1 l2)
  (if (null? l1) l2
      (append-reversefirst (cdr l1) (cons (car l1) l2)))))
```

Bibliography

[1] V. Allis. A knowledge-based approach of connect-four, the game is solved: White wins. Master's thesis, Departement of Mathematics and Computer Science, Vrije Universiteit Amsterdam, 1988.

[2] I. Bratko. *PROLOG, Programming for Artificial Intelligence*. Addison-Wesley, third edition, 2001.

[3] P.A. de Marneffe. *Introduction à l'Algorithmique*. Université de Liège, 1999.

[4] P.A. de Marneffe. *Cours d'Algorithmique Avancée*. Université de Liège, 2006.

[5] http://theory.stanford.edu/∼amitp/GameProgramming/.

[6] http://www.cs.ualberta.ca/∼games/Sokoban/.

[7] http://www.wikipedia.com.

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

[9] A. Junghanns and J. Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In R. Mercer and E. Neufeld, editors, *Advances in Artificial Intelligence*, pages 1–15. Springer, Berlin, 1998.

[10] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(2):219–251, 2001. Republished in [16].

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

[12] G.F. Luger and W.A. Stubblefield. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*. Addison-Wesley, third edition, 1998.

[13] N.J. Nilsson P.E. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[14] N.J. Nilsson P.E. Hart and B. Raphael. Correction to [13]. *SIGART Newsletter*, 37:28–29, 1972.

[15] Y. Moses R. Fagin, J.Y. Halpern and M.Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1996.

[16] J. Schaeffer and J. van den Herik, editors. *Chips Challenging Champions*. Elsevier, 2002.

[17] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, third edition, 1992.