ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Evolving a compact, concept-based Sokoban solver

Master thesis

SCHOOL OF COMPUTER AND COMMUNICATION SCIENCES

Tom Schaul

April 18, 2005

Under the supervision of:

Prof. Boi Faltings (EPFL)
Dr. Eric Baum (CCLS)

Abstract

This paper reports on an attempt to come closer to an understanding of understanding, in the domain of artificial intelligence. Our approach is based in the idea that understanding means exploiting underlying structure. Our goal is compact code that is adapted to the structure of its problem domain. We want to achieve this compactness by applying Occam's Razor in an evolutionary framework. As a problem domain we chose the game Sokoban; for evolution we used genetic programming set in a Hayek economy.

We have gained insights into how hard this problem is and developed a framework in which to tackle it. Using concepts derived directly from playing Sokoban, an adapted representation language and the Hayek economic system, we laid the foundation for evolving such compact code. Although the complete project goes beyond the scope of this thesis, here we demonstrate experimentally that the approach is viable, because our system can evolve code that solves interesting Sokoban instances.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to take the opportunity to thank all those people who have supported me most during the time of this project:

Prof. Boi Faltings who understood what kind of project I was looking for, and then found one that perfectly matched my interests.

Dr. Eric Baum for devising the project, accepting to supervise my work and keep refreshing it with many fertile ideas.

Prof. David Waltz and the people of the Center for Computational Learning Systems, for their support and for making me feel welcome in their lab.

Marisa Marciano, Eliane Reuille and Nancy Burroughs, for greatly helping me with the administrative aspects of doing my diploma project abroad.

The residents of the International House, for their gift of friendship, and for making me feel at home in this great city of New York.

My friends Alex, Andr, Bob, Caroline, Jo, Lex, Martin, Phil and all others who contributed many suggestions and ideas.

Claudia Clopath for her love, and for giving me boosts of motivation when I needed them.

Last but not least, I would like to thank very warmly my parents whose continued support has made all this possible in the first place.

Chapter 1

Introduction

1.1 Background

We believe that it is appropriate to describe *understanding* as the *computational exploitation of underlying structure*. This statement, as well as a justification and its implications are put forward in the book 'What is Thought' [1]. There, Baum argues that evolution has created minds that understand, and that this has been achieved by the process of building a highly *compact* system that behaves correctly in a large number situations, which therefore necessarily exploits underlying structure. This idea of compactness is well known, in the form of the principle of Occam's Razor:

> *Pluralitas non est ponenda sine necessitate*

which we can rephrase as the principle of eliminating everything that is not absolutely essential.

The criteria of compactness, requires us to have a condensed representation, and as the problems we encounter are structured (usually even to a very high degree), the most efficient way to achieve that is to exploit this underlying structure [2] [3]. Further generalizing this representation makes it possible to solve new problems with a minimum of effort.

We are not trying to explain how exactly the human brain exploits underlying structure (at least not in *this* project), but we are concerned with making good use of this idea in the field of artificial intelligence (AI). More specifically we want to research the feasibility of using Occam's razor in a learning process that will solve logical problems, by evolving appropriate compact code, and by using a *divide and conquer* approach. The code will be compact by virtue of code reuse, i.e. modular programs where modules correspond to real concepts. It should furthermore generalize enough to be capable of quickly solving different problems of the same domain because the solution program is built by and out of such previously learned concepts [4].

Complexity theory indicates that building such compact, powerful programs is a hard optimization problem [5]. A human programmer is no more able to build such a compact program by hand than he would be able to solve huge traveling salesman problems by hand. If programs that understand are to be built, they will have to be written by machines (at least partially), e.g. by using genetic programming techniques. A more mixed approach would be to use a kind of powerful CAD[1] tool, which unfortunately does not yet exist.

In practice, an additional criterion cannot be left aside: if we want to use such a compact program in real applications, we have to *trust* it. And there is no better way of building trust than understanding, so we want our resulting code to be *human-readable*. We think that this feature may possibly be a positive side-effect of evolving a result that closely matches the underlying structure. It is possible that many of our concepts may turn out to be entities in the evolved modular structure.

Nature threw massive computational resources at the problem of building such understanding programs (in the form of evolution), and it is an open question whether the currently available computational resources will allow a solution.

1.2 Our approach

We are addressing this problem, not in its entirety, but in a restricted domain. The domain we chose is the one of *two dimensional navigation problems*, because many general purpose concepts are involved, humans excel in the domain and easily learn how to adapt. More particularly, we decided to start with a specific problem that embodies many of the difficulties of the domain: the one-player game **Sokoban**. Chapter 2 is describing this game extensively, and its section 2.7 is justifying why it is a very appropriate problem for our purposes.

We believe that the structure of the Sokoban domain can reasonably be described in terms of entities that have a close relationship with human concepts, both nameable and implicit. For sake of simplicity, we will call them **concepts** (see also section 2.5). Furthermore we assume that (a) human players can solve many interesting Sokoban instances[2], (b) when doing that they use a number of high-level concepts, (c) sometimes they have to learn a new concept in order to progress, (d) these concepts are simple, unique, general and composable, and (e) learning how to play consists in understanding the relevant concepts (i.e. learning how to exploit the underlying structure) and in learning how to use known concepts.

1.3 Previous work

In the field of artificial intelligence, concept learning has not been a significant focus. Instead, most methods have generated concepts implicitly and in a distributed fashion (e.g. neural networks, genetic algorithms). Other systems have attempted the composition of hand-constructed concept structures

[1]CAD: Computer-aided design

[2]Most of those instances cannot be solved by search-based algorithms.

(e.g. SOAR [6], LABYRINTH [7]). However, we are not aware of any previous research that uses concepts to evolve compact code.

Junghanns and Schaeffer have constructed a program that solves many Sokoban problems that humans find interesting[8]. This was a tour-de-force, as no other approach of which we are aware is able to solve even the simpler Sokoban instances. But their program was nonetheless a mainstream AI approach: it is based on search and gains its advantage over previous AI search programs from complex hand-coded modules that restrict search. But the method (a) requires a lot of human hand-coding on a domain-by-domain basis, (b) is not obviously extensible to domains beyond Sokoban (c) does not achieve human levels of understanding even on Sokoban, and (d) does not output any code for solving the complete problem class.

Baum and Durdanovic have taken a similar approach to ours, but addressed different problem domains[9]. Their work is based on an evolutionary framework in an artificial economy which encourages compact and efficient code. They have been able to evolve compact code that solves large problems in Blocks World efficiently, and without any hand-coded parts. They have found similar but more moderate results on the problem of Rubik's Cube. However, these problems can be solved by purely computational means with standard AI techniques.

We will adopt domain-specific ideas from Junghanns and Schaeffer's work, and build an evolutionary framework inspired by Baum and Durdanovic's.

1.4 Objectives

Our goal is to build a program that learns to understand how to reason about two dimensional navigation problems well enough to solve human-solvable Sokoban problems. Along the way, we intend to gain deeper knowledge on the following questions:

1. Does the problem space have an underlying structure that can be described sufficiently well with concepts?
2. How difficult is it to code those concepts manually?
3. How efficiently can evolution combine them?
4. Can we evolve a compact program that solves simple Sokoban instances?
5. How well does such a program scale to hard or very hard ones?

Note. We realize that these objectives are ambitious, especially given the severe time restraints imposed by this project being done within the framework of a diploma thesis, i.e. six months. It is thus obvious that we cannot achieve everything that we would like to, but we are positive that the work will be continued.

1.5 Plan

We will start by analyzing the problem domain of Sokoban (chapter 2), and based upon that, we will develop a set of hand-coded modules that capture meaningful concepts and actions in the domain (chapter 3).

Then we will design an appropriate representation language that can describe meaningful algorithms in the program domain and make use of the developed modules (chapter 4).

This language will be the basis for evolution in an evolutionary framework based on the Hayek system mentioned above, which we describe in chapter 5.

The experimental setup is described in chapter 6, and the empirical findings are shown in in chapter 7. Finally, in chapter 8 we will critically evaluate our results, conclude and talk about the future work.

Chapter 2

Sokoban

Sokoban is a classic puzzle game invented by Hiroyuki Imabayashi in 1980[1]. Over the years many versions for all platforms have been developed, it is still actively played today, and new levels are created all the time. The simplicity and elegance of the rules have made Sokoban one of the most popular logic games.

2.1 Rules

If you are familiar with the game, you may skip this section, which explains the few simple rules of Sokoban. Otherwise, we strongly recommend you to try and play the game a bit, after reading it. Some of the available versions are listed in the bibliography [10] [11] [12].

'Sokoban' is Japanese and means 'warehouse'. The goal of the game is to push barrels into storage locations in a crowded warehouse. We call a problem instance **level**, and it corresponds to a situation in a warehouse. Formally, it is defined as a two-dimensional **grid** where each space can be one of the following:

- free space
- a wall
- a barrel
- a storage location
- the pusher
- a barrel in a storage location

Each space is uniquely identified by its coordinates on the grid.

By convention there is always a closed wall around the level, and there are exactly as many **barrels** as there are **storage locations**. The **pusher** corresponds to a human worker who is in charge of properly ordering the warehouse. There is exactly one. He can **move** freely to adjacent free spaces

[1] The game is Copyright ©1982 Thinking Rabbit, a company of which Hiroyuki Imabayashi is the president.

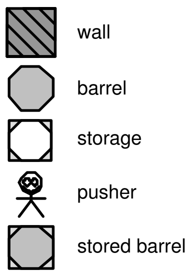

Figure 2.1: Symbols used

(but not diagonally). He can also **push** (not pull) a barrel, but only one at a time. The level is said to be **solved** when every barrel is on some storage location (barrels are not paired with specific storage locations).

Have a look at figure 2.2 for small example, which shows the main steps of its solution. The meaning of the symbols are explained in figure 2.1.

Solutions in Sokoban are subject to the topology of the level and the placement of the barrels. They are also inherently sequential: only limited parts of a solution are interchangeable because subgoals are often interrelated.

2.2 Complexity

It has been proved that solving Sokoban is NP-hard and PSPACE-complete [13] [14]. Now this statement could be the end of a project like this one already, if it was not for an intriguing observation: the levels that humans consider the hardest and the most interesting are usually not large (they usually have grids between 8x8 and 20x20).

If you want to get a feeling for how hard a compact level can be, you might want to try to solve the level in figure D.6 (page 51). It has only 6 barrels, 5 of which are already in a correct location, and still the solution is surprisingly long and complex.

Other than the theoretical results, we also have empirical evidence that quite small levels (of grids smaller than 20x20) that are interesting to human players cannot be tackled by brute force. It turns out that applying a traditional AI approach like using an efficient domain-independent search algorithm[2] does not yield a solution, even on a simple level (see figure 2.3, how long does it take you to solve it?), and with a very substantial computational effort [8].

Let's have a closer look at some of the reasons why this is the case: the *branching factor* in the search can go up to a multiple of the number of barrels, and the *depth* is usually more than 100 pushes, and can go up to more than a thousand [8] (for the example given in figure 2.3, it is exactly 97). This can give us a state space of over 10^100 states but only one goal state.

[2]We refer here to **Blackbox 3.3**, the winner of AIPS'98 fastest planner competition, the experiment is described in [8]

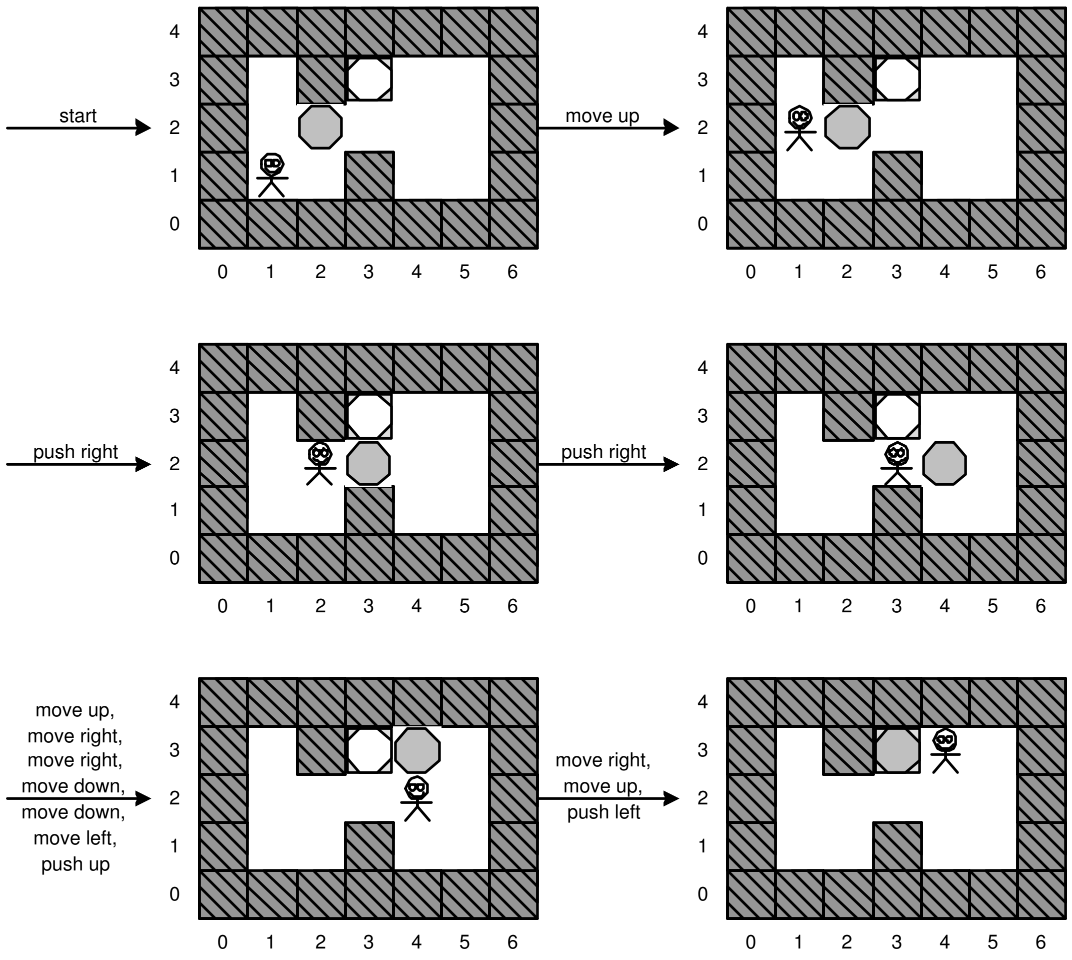

Figure 2.2: A walk through of the optimal (shortest) solution to a simple example (having only one barrel). In the last state, the level is solved.

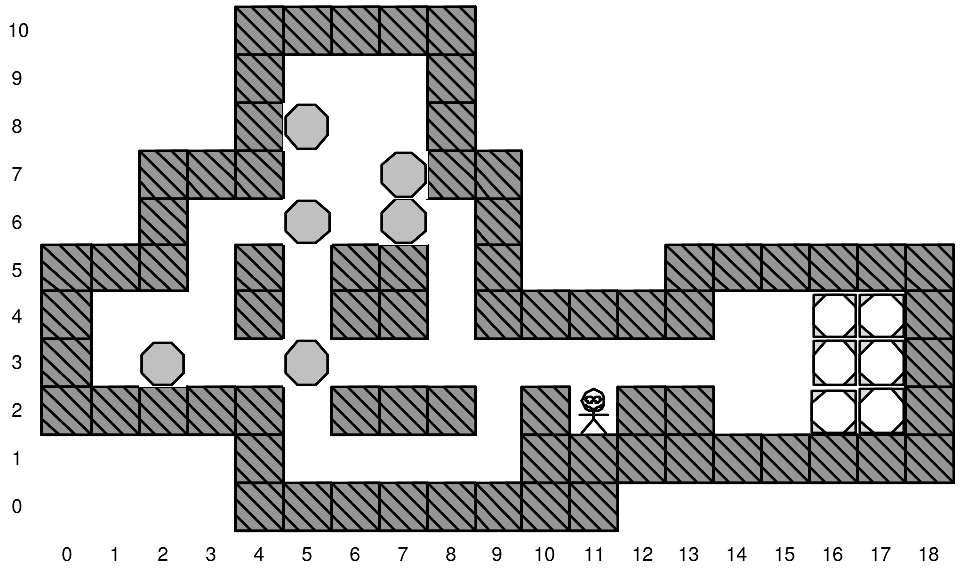

Figure 2.3: A simple level, by Thinking Rabbit, Inc.

2.3 Optimality

For a given level, it is possible to distinguish different solutions that differ in the number of moves or the number of pushes that the pusher does. Different versions of Sokoban count either, but for this paper, we will consider the moves to be of secondary importance, and only count pushes.

However, in accordance with human playing behavior, we do not place the main focus on the problem of finding the shortest possible solution, but rather on quickly finding one that solves the level in a reasonable number of pushes.

2.4 Levels

Sokoban is only as interesting as the levels you are playing. We distinguish several types of levels:

traditional levels: they are designed with a lot of effort, moderately large and are often built around a specific concept. Some of these are grouped in collections with incrementally increasing difficulty. Example: figure 2.3 or figure D.7 (page 52).

minimalistic levels: similar to the previous type, but smaller and generally with no space that is not strictly necessary. They may or may not be based on a concept, but they usually have little or no structure. This category seems to be the most appropriate for a search-based approach to the game. Example: figure D.6(page 51).

thematic levels: the focus here is on the visual appearance of the maze, they may be fun to play but are not necessarily challenging. They can be very large.

generated levels: these randomly generated levels depend strongly on the quality of the generating program. They are generally small and can be interesting. Sometimes they are just hard and confusing though.

trivial levels: this is a set of very small levels (only very few barrels) that we wrote ourselves, they are not challenging for humans, but can be used for testing and for bootstrapping the evolution. Examples: figure 2.2 and D.1 to D.5.

We have gathered level collections from various authors from the Internet[10] (mainly of the traditional type) and have written a trivial collection. To extend these data, particularly at the easy end of the spectrum, we have developed the tools to *simplify* existing levels. The following means all guarantee that the simpler level remains solvable:

- remove a number of barrels (and the same number of storage locations)[3]
- remove a number of walls, thus changing the structure and increasing the number of possible solutions.
- for levels where we have a *complete trace* of a solution, we can produce variants that capture the situation at different stages. Each stage corresponds to a new, partly solved instance, that is easier to solve than the original level.

With these tools in hand, we are able to produce enough training instances for our evolutionary framework. Apart from that, we also want our levels to be *incrementally* harder, reducing the gaps in difficulty as much as possible, because such a large gap might halt the evolutionary process. We are aware that our simplifications do not necessary close these gaps: removing a single barrel can make the level trivial to solve, without offering any insight on how to solve the original instance.

There is no unique way of measuring the difficulty of a level[4], although there are different relevant criteria:

1. size of the grid
2. number of barrels
3. length of the solution
4. average solution time (human player)
5. size of the search space
6. number and complexity of concepts involved

[3]This is actually a bit more tricky than it sounds: we have to be careful to remove a storage location which can be (in some solution) occupied by the removed barrel.

[4]In fact, human players still rely on a subjective impression, i.e. the common version of the game [10] will ask the player for their opinion after completing each level.

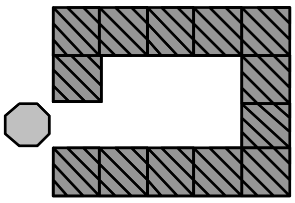

Figure 2.4: Counter-intuitive turn: in order to push this barrel to the left, we first need to push it inside, get behind it and then push it back (we assume that the pusher is not inside at the start)

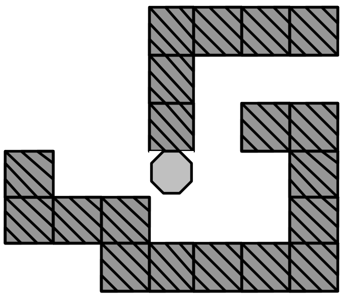

Figure 2.5: This is a one-way passage: the pusher can only go from the left to the right (push right, push left) if it does not want to put the barrel into a deadlock. After having passed through, it is again only possible to go through from left to right. There is a complete level based on this concept in the original collection, it is shown in figure D.7 (page 52)

The original set of levels (that Junghanns and Schaeffer [8] used as their reference) is not clearly following any of these: but it continuously introduces new concepts over time, and it is necessary to assimilate those concepts to be able to solve some of the later levels.

Finally, as we cannot determine an order of difficulty *a priori*, we will take our best guess — which we base on increasing size and adding concepts — and refine it based on our experimental results.

Note. For the specific subproblem of deadlock detection, described in section 3.4.6, we developed some additional tools for generating and manipulating levels; these tools however are not used anywhere else in the project.

2.5 Concepts

We have been speaking of concepts already, so let's have a look at a few concrete examples now: figures 2.4, 2.5 and 2.6 illustrate a few simple ones, which are all related to the static structure of the level. Other ones are for example tunnels (what goes in has necessarily to come out) or pusher-passages (open for the pusher, but cannot be used to push a barrel through).

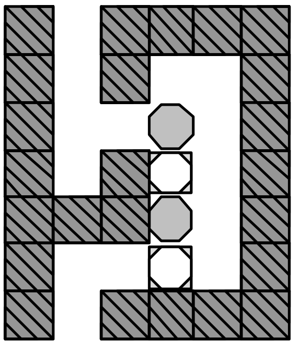

Figure 2.6: This pattern shows a passage that can be passed only once (enter at the bottom, push down both barrels, exit at the top) and is then forever blocked.

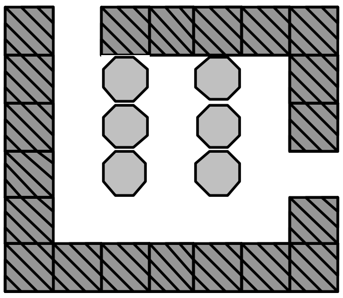

Figure 2.7: The room was empty before; now the maximum of 6 barrels have been temporarily placed inside, where they do not disturb. It is possible to get all of them out of the room again (to where the storage locations are).

An example of a more sophisticated concept is illustrated in figure 2.7, which is the idea of temporarily storing a large (maximal) number of barrels in a room (for later use), without causing a deadlock. A particular concept, arguably the most important of all, will be analyzed extensively in the next section: deadlocks.

We will not attempt to enumerate all possible concepts because there are too many, and some of them are not easy to describe. Instead, we leave it to the enthusiastic reader to discover more of them by playing the game. But such a deep knowledge of the game is not necessary for the understanding of this project, nor is it required for us, because — and this is exactly the purpose of using an evolutionary framework — our system will eventually discover these concepts by itself.

2.6 Causality and deadlocks

Unlike many other logical problems, Sokoban is deeply pervaded by the notion of causality: the fundamental rule (push not pull) has as a consequence that most

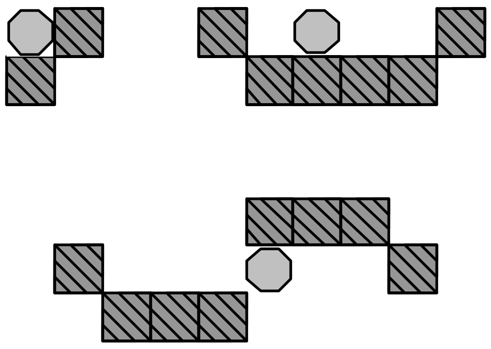

Figure 2.8: Static deadlocks, consisting of one barrel and the configuration of walls around it.

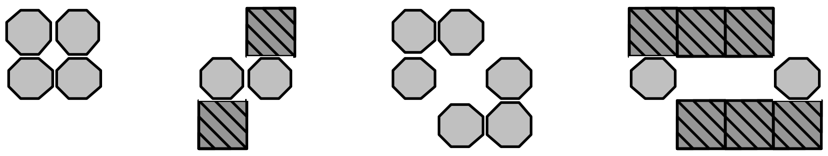

Figure 2.9: Simple, local deadlock patterns.

manipulations are not reversible. Since all barrels need to end up in a storage location, it is common that one push leads to a situation that cannot lead to a solution, *whichever* manipulations come after it. We call such a situation a **deadlock**.

Before describing the consequences of this, we suggest that you have a look at figures 2.8 and 2.9 for an illustration of different kinds of deadlocks. A deadlock can be formed by as few as one or as many as all the barrels of the instance. Figure 2.10 shows such a deadlock situation that is not trivial.

Deadlocks are a critical element: there is no way to recover from bad push at an early stage other than to **undo** everything up to that point and try again. It is therefore important to recognize potential deadlocks as early as possible; this is one of the first things one learns when starting to play the game (because of the high amount of frustration). Unfortunately this is not always possible, thus one learns to live with it: after backtracking to the bad push, one generally tries to redo more or less the same manipulations, and so — usually — most of the effort was not in vain.

2.7 Conclusion: Why Sokoban

We conclude this chapter by pointing out the main arguments that made us choose Sokoban over other logical problems or games:

- it is a *real* problem, not an academic one,
- the rules and the framework are simple and easy to understand,

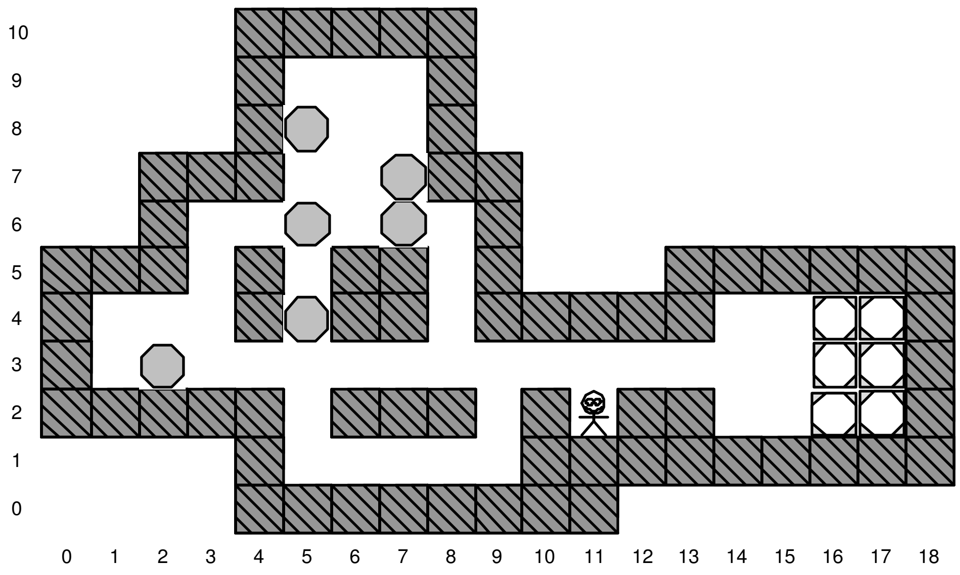

Figure 2.10: An example of a non-local deadlock: this level can never be solved because of the barrel in (5,4), but the barrels in (5,6) and (7,6) contribute to the situation too.

- there are enough problem instances available for learning,
- it is very *hard* and it cannot generally be tackled by a standard AI approach
- while the problem instances go smoothly from easy to hard, which makes incremental learning possible, and, last but not least,
- it is actually a very *fun* game!

Chapter 3

Domain-specific modules

3.1 Definitions

Before describing the modules, we will put forward a few definitions of terms that we will use repeatedly:

Definition. x is said to be **reachable** from y, if the pusher can move from x to y without pushing any barrel.

Definition. A **zone** is a set of positions in a level where each one is reachable from each other. See also figure 3.1.

Definition. A **barrier** is the set of barrels between two zones, so that, was any of the barrels to be removed, the two zones would be connected (and form a single zone). See figure 3.1 for an illustration.

Definition. y is said to be **barrel-reachable** from x if, when all other barrels would be removed, a barrel in x could somehow be pushed all the way to y.

Definition. A **chamber** is a set of positions in a level where each one is barrel-reachable from each other.

Definition. A **strait** is a position through which all barrels have to pass (in the same direction) on their way to a storage location. Not all levels have a strait. If there is more than one, we use the one that is closest to the storage area.

Definition. The **focus** is a subset of all the barrels. Barrels in focus are those being considered for pushes, or those that have been pushed recently. Barrels not in focus are assumed to be inactive, fixed[1].

3.2 Representation of the game

Instead of simply storing the grid of a level (and its contents), we built a more complex datastructure to represent a game. In addition to storing the level and

[1] A normal consequence of this is that when a certain (sequence of) manipulations is not possible, it may be necessary to increase the focus and try again.

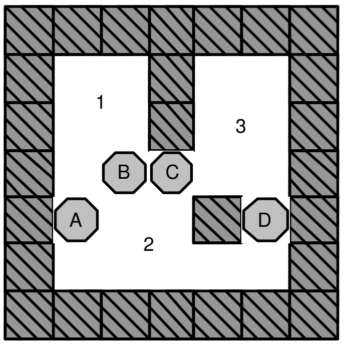

Figure 3.1: The barrier between zone 1 and 2 is composed by barrels A and B, the one between zone 2 and 3 by barrels C and D. One might be tempted to say that B and C are between zones 1 and 3 but this is not a barrier because removing B or C would not connect the zones.

keeping track of where the barrels and the pusher currently are, it also keeps additional information up to date:

- the **forbidden positions** that can be statically determined: if a barrel should ever be put there this would by itself put the game into deadlock. These positions are filtered out at the source and never considered for pushes. Figure 3.2 offers an illustration of this concept.
- the **static structure** of the level is a directed graph showing which chambers are barrel-reachable from which other chambers. When augmented by the number of barrels and storage locations in each chamber, this graph may provide us with additional explicit constraints on how to solve the level; in some cases it can even tell us that we are deadlocked[2].
- the **dynamic structure** of the level, consisting of all the zones. We keep track of how they are related to each other by maintaining (incrementally updating) an undirected graph in which each zone and each barrel has its corresponding node, and links in the graph are established between those which are physically touching in the level[3].
- the **trace** of all manipulations that have been performed on it. This includes all pushes and all changes to the focus.

3.3 Low-level functions

Without going into detail, here are some brief descriptions of the low-level functions on which the complex modules are built; some of these will also be used in the instruction set of the representation language (section 4.4):

[2]Details on the implementation of the static structure can be found in appendix C.2
[3]Details about this dynamic graph can be found in appendix C.1

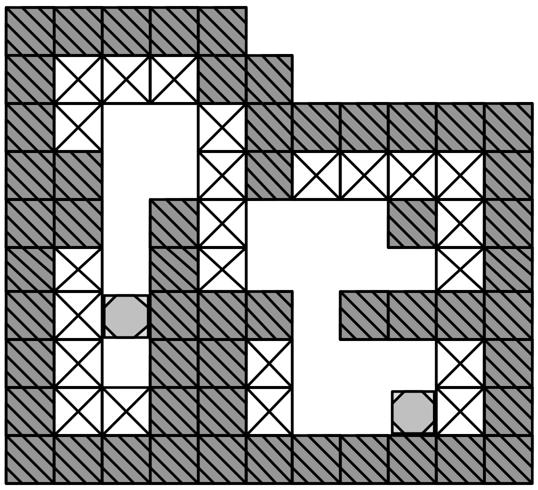

Figure 3.2: The spaces marked with a cross are forbidden for any barrel: once a barrel would be put there, the level would necessarily be in a deadlock. Note that this generally excludes a significant number of possibilities for pushes.

- a function to determine if the level is solved,
- a function that does a push if it is legal, i.e. if nothing hinders the push, if it does not lead to a forbidden position and if the pusher can reach the position from where he has to push. This function also moves the pusher to where he is needed automatically,
- a function to undo the last push,
- functions to set **milestones** in the trace, which can be used to quickly come back to later, e.g. after getting stuck in a deadlock.
- a **hashing** function that allows us to determine if two states are equivalent, and therefore evoid cycles,
- functions that add a barrel to or remove it from the focus.

3.4 High-level modules

This section describes the high-level modules that we have built. They may call each other, sometimes recursively. Their implementations are not final: by having access to new modules, or new versions of old modules, we can develop a new version which can do its task with less constraints. Not all versions have been fully implemented at this stage of the project; these are flagged by `[not implemented]` . Our main focus has not been on developing the most sophisticated (and often recursive) versions of these modules, because we wanted to achieve the best results with simple building blocks, at least as long as this is possible[4]. Those versions will be useful at a later stage.

[4]We have also resisted the temptation of writing something by hand that could serve as a complete solver

3.4.1 Push barrel to destination

The `getBarrelTo` module finds a sequence of pushes which will bring a specific barrel to a (far-away) destination.

Version 1: does not push any other barrel around.

Version 2: includes scenarios in which the barrel needs to be pushed in one direction and then back the same way (c.f. figure 2.4).

Version 3: apart from manipulating the selected barrel it does only pushes that are reversible (and reverses them at the end). `[not implemented]`

3.4.2 Move pusher into zone

The `getPusherTo` module looks for a sequence of pushes which will allow the pusher to get into a different zone (or to a specific position), without creating a (detectable) deadlock. It can also be used to try and **open up** an adjacent zone, which is a common subgoal when playing Sokoban. Furthermore it can be used to get **behind** a barrel or a barrier.

The planning algorithm that this involves is based on the concept of **relevance**. A barrel can be relevant if it:

- is directly blocking a way to the goal,
- blocks a push of a relevant barrel, or
- forms a deadlock with a pushed relevant barrel.

Version 1: internally uses version 3 of `detectDeadlock`.

Version 2: (recursive) internally uses version 5 of `detectDeadlock`. `[not implemented]`

3.4.3 Reversible push-sequences

This module determines if a sequence of pushes is reversible, i.e. there exists a second (reverse) sequence which, when executed after this one, leads back to the original situation.

Version 1: tests only the exact symmetric sequence.

Version 2: same, but may change the order in the reverse sequence. `[not implemented]`

Version 3: (recursive) may also try to make different temporary pushes, if such pushes are reversible. `[not implemented]`

Version 4: (recursive) may furthermore try to push previously untouched barrels temporarily, if such pushes are reversible. `[not implemented]`

All versions are correct, but even the fourth version is not complete, as there might be, for example, a structural constraint that forces a barrel to come back a different way (see figure 3.3).

3.4.4 Ordering of storage locations

Often the storage locations are grouped in one area, with only a limited number of access points. It simplifies finding an overall solution if we know in which order to fill the individual storage locations. It avoids the situation in which we have almost

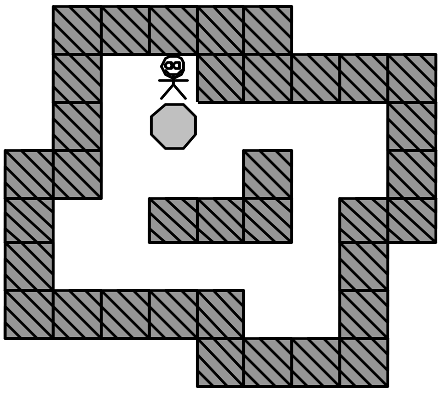

Figure 3.3: Would pushing the barrel down once be reversible? Yes, but we need to go all around. Imagining a situation like this set in a complex level should make us believe that there cannot be a simple and *complete* algorithm for determining reversible sequences.

solved a level but then noticed that we placed a barrel badly, and cannot undo that without losing most of the solution.

The `storageOrder` module is executed only once, at the beginning, and the sequential order that is found is stored and used the rest of the time. This is sufficient because the order depends only on the static structure of the level (and the original positions of the barrels).

Version 1: works on levels that have a *strait*, in the strict sense of the definition.

Version 2: works on levels that have all storage locations in the same chamber, but so that a barrel will not exit that chamber once it is inside. `[not implemented]`

Version 3: works in general on levels that have all storage locations in the same chamber. `[not implemented]`

The instruction that uses this module (`bestStorage`) will return the storage locations in order, starting with the first one that is not occupied by a barrel. For levels where the module cannot determine any order, this instruction will still return an approximately reasonable answer, based on the heuristic of filling the furthest positions first.

3.4.5 Temporary placement of barrels `[not implemented]`

A practical module would be one that could best use the available free space for temporary placement of barrels that are hindering our progress elsewhere (see also figure 2.7). We have not implemented this module for lack of time.

3.4.6 Deadlock detection

Before describing the `detectDeadlock` module, we have to acknowledge that building a module that can detect *any* deadlock is a hard problem, in fact it is as hard as building one that solves Sokoban directly (for a justification of this statement, please have a look at figure 3.4). However, having a module that can detect a large number

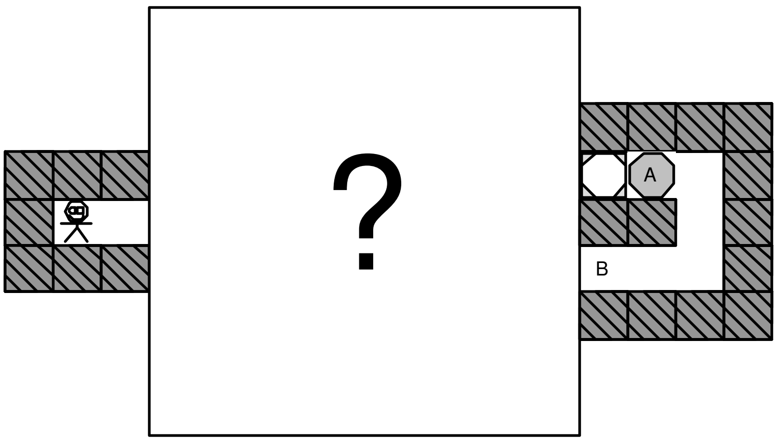

Figure 3.4: The question that we will ask our deadlock detector is the following: is barrel A causing a deadlock? The answer is yes if the position marked by B can never be reached by the pusher. This however is dependent on what is in the big rectangle: it can be a Sokoban level of any complexity. We can even imagine that the access to B is only possible after solving that level completely. In that particular (but possible) case, detecting a deadlock is equivalent to finding a solution for the level.

of deadlock situations will be very helpful. So the purpose of this module is to detect a large number of deadlock situations, as early as possible.

Version 1: detects only static deadlocks (c.f. 2.8).

Version 2: detects structural deadlocks as well, i.e. a chamber without an exit (for barrels) which has less storage locations than barrels .

Version 3: in addition to those before, detects simple pattern-based deadlocks (like those in figure 2.9, but not exclusively).

Version 4: (recursive) for each barrier (with not enough storage locations behind it), makes a call to `getPusherTo` (version 2) to see if it is possible to get behind its barrels without causing a deadlock. This is not guaranteed to have a finite execution time.`[not implemented]`

We originally had a different approach to the problem of detecting deadlocks: we designed a pattern language adapted to Sokoban and tried to evolve a set of efficient patterns that would be capable of detecting the most common deadlocks. We deemed the results to be insufficient for our purposes and discarded the idea again, but not without taking over some of the patterns we found into version 3 of this module.

Note. There is a kind of symmetry between the concepts of deadlock and of reversible: one has very grave consequences and the other almost none. While solving a Sokoban level we try to avoid all pushes that lead to a deadlock, but we also avoid playing around too much with reversible manipulations because they are mostly meaningless. So the choices that are actually interesting are those that fall in-between these extremes; they are the ones a search-based solver should consider.

Chapter 4

Representation Language

4.1 Objective

The heart of the agents in our artificial economy are two pieces of code, one determines how much the agent will bid for a given state of the game, and the other one determines the action that it will perform on the game, in case it wins the auction. This procedure is described in detail in the next chapter; here our focus is on how those two pieces of code can be constructed.

The evolutionary process requires that the agents are structured in a way that they can be **generated** randomly, as well as undergo **mutations**. Furthermore, we want them to have a substantial *expressive power* because the problem domain is complex, although we do not think that they need to be Turing-complete. But the quality of the representation language for this code is not only determined by its expressive power, but also by the proportion of possible expressions that are actually *meaningful*. We therefore need a sufficiently restrictive syntax on one hand, and a minimal set of instructions that can still cover the domain on the other hand. The design of our representation language is thus striving for this delicate balance between expressive power and meaningful restrictions. If we can achieve that, then the selection process of our artificial economy will be able to reduce the set of meaningful code-fragments by the criteria of usefulness and efficiency.

4.2 Syntax

The syntax of our representation language is based on **typed S-expressions** [15]. These are recursively defined as a symbolic expression (as in Lisp), consisting of either a symbol or a list of S-expressions. A comprehensive way to look at them is to see them as a n-ary tree where each leaf and each node is a symbol. Each symbol has a **type**, and if it is not a terminal symbol, it has a determined sequence of types which it takes as arguments (i.e. subtrees). See figure 4.1 for an illustration.

4.2.1 Types

We have a relatively large number of data types in our instruction set. Apart from basic types like integers, booleans or lists, we also have many domain-specific ones.

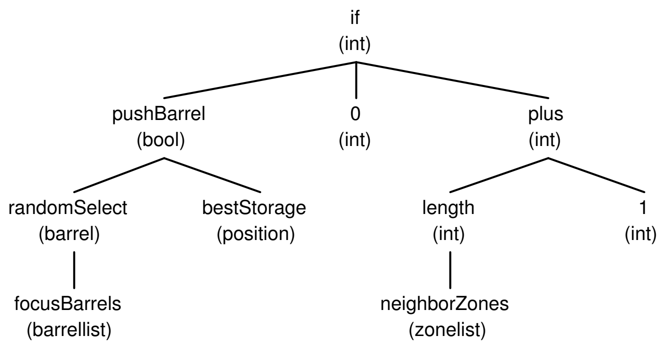

Figure 4.1: An example S-expression, shown in its tree representation. Its canonical form is:
if(pushBarrel(randSelect(focusBarrels),bestStorage),0,plus(length(neighborZones),1)).

basic	bool int
domain-specific	position direction barrel push sequence zone

Table 4.1: This table contains a listing of types used in our representation language. Not listed are the *list*-types: there is one for each standard type, but there are no lists of lists.

These are introduced to increase the meaningfulness of generated S-expressions, and to reduce the number of failures or exceptions during execution.

Example. Although in the implementation position and barrel are each equivalent to a pair of integers, we name them differently for they have different uses: a push instruction is only meaningful if its first argument is a barrel and the second is a position without a barrel.

The complete list of types is given in table 4.1. We have paid attention to code this part of the project in such a way that it is very easy to extend, reduce or modify the set of types because we believe this will be necessary as the project progresses and the needs of the representation language change.

4.2.2 Special treatment: loops

At the current stage of the project, we have only implemented one type of loop expressions: they iterate over the elements of a list. This guarantees a finite execution, but not necessarily a short one, as it is possible to have an unlimited[1] number of levels of inner loops. The tree on figure 4.2 shows how such a loop is represented.

[1]Not quite unlimited: there are parameters that restrict the maximal depth of an S-expression, so they indirectly limit the number of inner loops as well

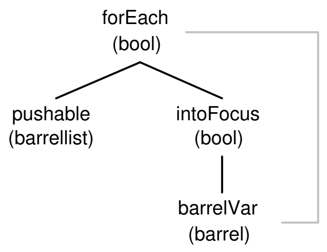

Figure 4.2: An S-expression that contains a loop: the root node (`forEach`) executes the expression of its right subtree for as many times as there are elements in the list returned by its left subtree. The `barrelvar` node is a variable that iterates over these elements.

Loop expressions return a boolean value, for `forEach` it is `True` if all its iterations (there must be at least one) return `True`, and for the analogous loop expression `forAny` the behavior is the same but the return value is True if at least one of its iteration returns True.

Although we will only see how generation and mutation work in sections 4.5 and 4.6, we can already specify the particular treatment that is required there for loops: the pool of symbols available at a certain node only contains the iterator-variables of the loop expressions that are its ancestors in the S-expression-tree. Furthermore, a specific type of mutation for loops will be explained in that section.

4.3 Execution

Execution of S-expression is done similarly to programming languages: for operators (both arithmetic and logical ones) as well as domain-specific instructions, all arguments are executed and the results are used in the encompassing instruction (the parent node of the arguments). The arguments of functional instructions like loops or conditional clauses are only executed if, and as often as, they need to be.

4.3.1 Exception handling

In case of an execution failure (i.e. a push that is not possible, a selection from an empty list) the execution of the whole expression is canceled, and the system continues normally. If this happens while calculating a bid, we assume that the agent did not bid at all, and if it happens while acting on the game, we assume that the action failed and undo any manipulations that have been done before the exception.

4.3.2 Time limits

We allow each S-expression only a limited amount of execution time. We measure here the total elapsed time instead of the actual processing time, for no other reason than to keep it simple. The parameters that determine this time are set in a way that the time restriction is very loose; our goal is to avoid only the worst cases[2]. A timeout has the same consequences as a normal exception.

[2]Our current representation makes infinite loops theoretically impossible, but future extensions might add that possibility, if the expressive power was increased.

Name	return type	arguments	
plus	int	int, int	
minus	int	int, int	
times	int	int, int	
and	bool	bool, bool	
or	bool	bool, bool	
not	bool	bool	
equals	bool	int, int	
lessthan	bool	int, int	
morethan	bool	int, int	
next	position	position, direction	
previous	position	position, direction	
dir	direction	position, position	positions must be next to each other
distance	int	position, position	manhattan distance
boolval	int	bool	type conversion
pushval	push	barrel, position	type conversion
isEmpty	bool	list	
length	int	list	
select	any	list, int	return type depends on list elements
randSelect	any	list	idem
if	bool/int	bool, bool/int, bool/int	either all 3 are bool or all 3 are int
forEach	bool	list, bool	see section 4.2.2
forAny	bool	list, bool	idem

Table 4.2: Basic instructions

4.4 Instruction set

The tables in this section list the instructions that are used in our representation language. The descriptions are brief, but all the important instructions are described in their context, in chapter 3. The notation 'type1, type2/type3' means that the instruction takes two arguments, the first of type1 and the second of either type2 or type3. We distinguish three categories:

- Table 4.2 shows basic instructions, like arithmetic operators or control flow commands; these can be evaluated independently of the state of the game.
- Table 4.3 shows **perceptive** instructions: they have access to information about the state of the game, but are not allowed to modify it.
- Table 4.4 shows **active** instructions: these are the ones that can act on the game. They all have a boolean return type, and the value True indicates that they have in fact changed the state of the game. The value False means that they failed (the state is left unchanged).
- Table 4.5 show **shortcuts** for common combinations of instructions. They do not add any new functionality but should make expressions more compact and speed up evolution.

The instruction set is in an early stage of developement, and it does not pretend to be complete or optimal. As for the data types, we have also constructed the instruction set in a way that it is very easy to extend, reduce or modify it at later stages of the project. Specifically, each instruction can be removed or added to the set individually,

Name	return type	arguments	brief description
solved	bool		is the game solved?
reachable	bool	position	from where the pusher is now
reversible	bool	sequence	see section 3.4.3
detectDeadlock	bool	barrel	see section 3.4.6
focusBarrels	barrellist		all barrels in focus
focussable	barrellist		barrels that are reachable but not in focus
allPushes	pushlist		all currently possible pushes
getPushes	pushlist	barrel	all pushes for the specified barrel
allStorage	positionlist		all storage locations
reachableStorage	positionlist		all those reachable by the pusher
usableStorage	positionlist		all reachable ones that are empty
neighborZones	zonelist		adjacent to the pusher's zone
pusher	position		the position of the pusher
furthestBarrel	barrel		of those in focus, relative to the pusher
biggestZone	zone		of the adjacent ones
bestStorage	position		see section 3.4.4
getBarrelTo	sequence	barrel, position	see section 3.4.1
getPusherTo	sequence	position	see section 3.4.2
getBehind	sequence	zone	break through the barrier and into the zone
getStrictBehind	sequence	zone	same, but without building any new barrier

Table 4.3: Perceptive instructions

Name	arguments	brief description
pushBarrel	push	does the push if it is legal
executeSequence	sequence	does all the pushes in the sequence
intoFocus	barrel	adds a barrel to the focus
outOfFocus	barrel	removes a barrel from focus

Table 4.4: Active instructions. The return type is not mentioned anymore because it is always `bool`.

Name	equivalent expression
increaseFocus	intoFocus(randSelect(focussable))
openUpAZone	getStrictBehind(randSelect(neighborZones))
bringBarrelsHome	forAny(focusBarrels, getBarrelTo(barrelvar, bestStorage))

Table 4.5: Shortcut instructions. These instructions do not take any arguments, and their return type is `bool`.

with a minimum of effort. For the purpose of further refinement, we vary the subset of these instructions that are actually available for a given simulation, as described in section 6.3.

4.5 Random generation

All nodes in an S-expression are generated randomly, from the root downwards, while being subject to the following criteria:

1. Depending on the *use* of the generated S-expression for bidding or acting, the instruction set is different: as the S-expressions that are responsible for evaluating an agent's bid are not supposed to change the state of the game (i.e. do pushes), we eliminate all **active** instructions from their instruction set[3].
2. With an increasing *depth* in the expression tree, instructions are favored that have no arguments. This guarantees that an S-expression cannot grow indefinitely.
3. The generated instruction has to have the appropriate **return type** (i.e. the type of the value returned by its execution). It has to conform to the type that the parent node specified, or, in case of the root node, it must be `int` for bidding expressions and `bool` for action expressions.

In other words, any two instructions that are not differentiated by any of these criteria will be used with the same probability. This is the simplest way of doing it, and we are aware that this might be a drawback once the set of instructions becomes larger, in which case we are prepared to extend the model.

4.6 Mutation

An S-expression can undergo a mutation, which implies that it is changed at exactly one node. This change can be one of the following:

1. a simple substitution of the node by one with the same type and argument types, without changing the subtrees of the arguments. See figure 4.3.
2. a substitution of a subtree: replacing a subtree by a newly generated one (of the same type). See figure 4.3.
3. insertion of a new node with, if applicable, a newly generated argument-subtree. The original subtree is not changed. See figure 4.4.
4. a special purpose mutation which transforms a selection from a list into a loop over that list. See figure 4.5.

[3]Their allocated execution time is also smaller.

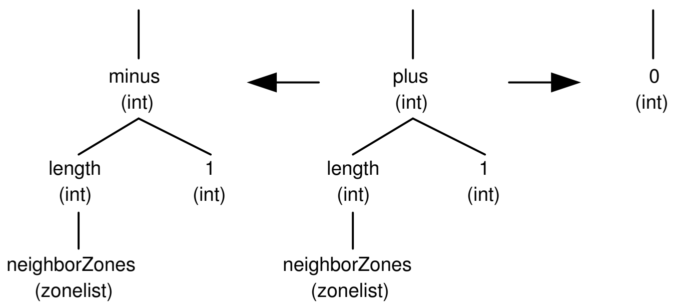

Figure 4.3: A mutation of type 1 on the left and of type 2 on the right, both times acting on the plus-node.

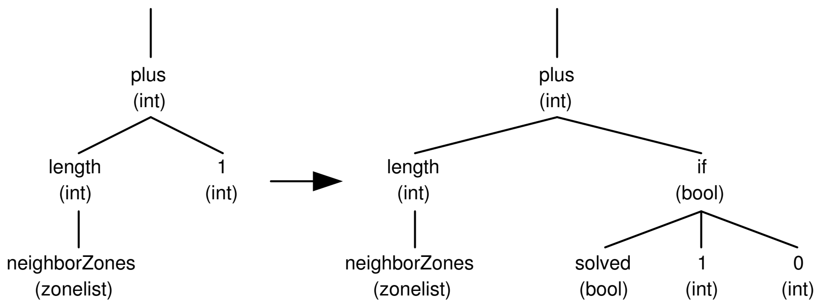

Figure 4.4: A mutation of type 3: the if-node has been inserted, and the subtrees of its condition and else clause have been generated.

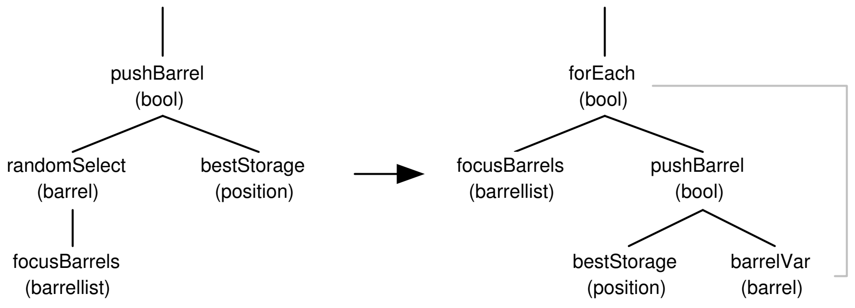

Figure 4.5: A mutation of type 4: the mutated node is randSelect, but the mutation changes the whole structure of the tree.

Chapter 5

Evolutionary framework

5.1 The Hayek economy

The fundamental idea behind the Hayek economy is *self-organization*. A group of agents compete among themselves for the right to perform an action (on the level), for many rounds. Each round starts with an **auction**, where each agent may choose to make a bid, and the highest bidder gains the right to perform an **action**. An action can consist of any combination of manipulations onto the state of the game. As a counterpart, it has to pay an amount equal to its bid to the winner of the previous round. Once the level is completely solved, the last agent 'sells' it to the bank for a high reward. Section 5.5 illustrates this economic procedure. The amount of the bid can be interpreted as the agent's estimate of the current state's value.

The structure of the system produces rational and cooperative agents that try to solve the level as good as they can, because this is the only way money can come into the system. Irrational agents are exploited and go broke.

To keep the agents' money from accumulating, all agents pay a small tax for each action they make. This also makes the agents sensitive to computational cost.

Contrary to genetic algorithms where the key element is usually a potentially very complex fitness function which has to be hand-coded, this system eventually learns its fitness function. It is implicit in the bids that agents make.

Giving *partial rewards* can speed the evolution, but we cannot give them, because we can never be sure that a situation that looks partially solved (many barrels in a storage location) is not in fact a deadlock. Fortunately this framework is not dependent on them.

5.2 Structure

The structure of the artificial economy is shown in figure 5.1.

- The **simulator** coordinates the system: it determines which levels should be attempted, in which order, and also how much effort should be allocated to each one[1].

[1]The simulator can also change the parameters of the system during execution to accommodate different types of levels.

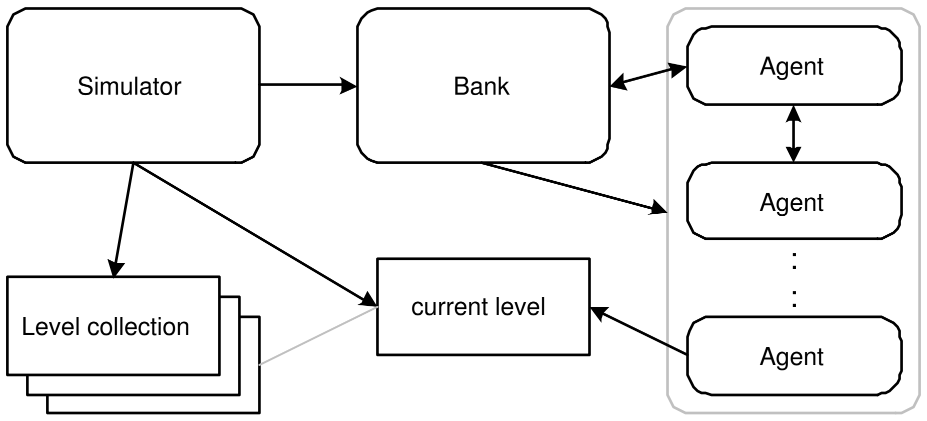

Figure 5.1: The structure of the evolutionary framework

- The **bank** has multiple tasks:
  1. hold auctions and let the highest bidder interact on the level,
  2. keep the system alive by providing credit for new agents,
  3. collect taxes, which are dependent on the computational effort that the agents require, and
  4. eliminate inefficient agents when they go broke.
- The **agents** do the actual work: they analyze the current state of the game, make a bid accordingly and, if they win, they do manipulations on the level. Their **genome** consists of the S-expressions: one that determines how its bid is calculated, and one that determines the action it will do.

5.3 Bidding

The type of auction we use is very simple: each agent is allowed to bid once (without knowledge on how high the other agents bid) and the one with the highest bid wins. The bids are usually in the same order of magnitude as the final reward, which, depending on the parameter settings, is between a thousand and a million. This is much higher than the number of agents, so it almost guarantees that no two bids are ever the same. Should that particular situation rise anyway, we randomly pick one of the winners. A sequence of auctions is illustrated in the example in section 5.5.

As mentioned before, the bid of an agent is determined by an S-expression. However, to achieve a smooth distribution of bids in the appropriate range (above a thousand) without requiring unnecessarily complex S-expressions, we do not directly take the returned `int` at face value, but put it in a linear function to produce the actual bid:

$$bid = a * bidexpr + b$$

The two constants (a and b) are determined at the birth of the agent, and can be negative. If the final bid is negative, we consider that the agent did not want to bid at all.

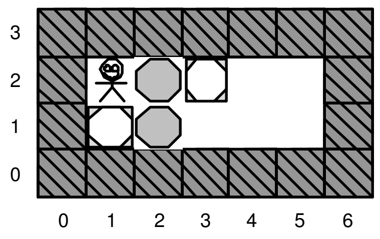

Figure 5.2: A simple level with two barrels, but which can easily be pushed into a deadlock.

An agent is allowed to have a constant bid ($a = 0$), in which case it pays less taxes, but it is also less flexible.

Note. We have devised an alternative bidding mechanism that works with negative bids as well as positive ones: a high negative bid could be interpreted as the agent believing that the state is in a deadlock. In that case, once the deadlock is detected by the bank, the last agent has to pay a penalty, but it can still make a profit if it receives enough money through its negative bid. We have not progressed far enough in the project to make use of this mechanism: we intended to keep it as simple as possible as long as possible.

5.4 Taxes

All agents pay taxes, for everything they do. One part of the taxes are proportional to the number of nodes in their S-expressions, the other part is determined dynamically and is approximately proportional to the execution time needed to process the instructions during bidding or acting. The taxes for bidding are lower, because these instructions get executed at each round, and we do not want agents to go broke before they even get a chance to act. There is a high tax associated with leading the game into a (detectable) deadlock. Furthermore there is a special tax for agents that bid high (and win the auction) but do not perform a successful action, in order to keep agents from holding the control for a long time without doing anything.

The existence of taxes guarantees that the agents will evolve to be efficient: an equivalent expression with less instructions (or faster ones) will out-compete its rival in the long run.

5.5 Example

This section shows a **cold run** of the artificial economy (i.e. without creation or elimination of agents) on the level shown in figure 5.2. We assume that the following two agents are in the system (there may be more, but they are not relevant).

Agent 1:
bid: $300 * length(E) + 200$
action: $executeSequence(E)$
with $E = getBarrelTo(randSelect(focusBarrels)), randSelect(allGoals))$

Agent 2:
bid: $400 * length(neighborZones) - 100$

round	winning bid	winning agent	manipulation	wealth agent 1	wealth agent 2
1	500	1	(2,2)→(3,2)	1000-30-500	1000-10
2	300	2	-	470-10+300	990-130-300
3	300	2	(3,2)→(4,2)	760-10	560-30
4	500	1	(2,1)→(1,1)	750-30-500	530-10+500
5	500	1	(4,2)→(3,2)	220-30	1020-10
6	1000	bank	-	190+1000	1010

Table 5.1: This table shows the state of the system at each bidding round. The wealth is shown in the following format: $wealth_{old} - \sum taxes \pm bid$.

action: $executeSequence(getBehind(randSelect(focusBarrels)))$

The table 5.1 shows what is happening in the system, and how the wealth of the two agents evolves. The agents start out with an initial wealth of 1000. They pay what they bid to the last winner (which may be themselves if they win repeatedly, like in rounds 3 and 5). At each round the agents pay a tax, to simplify the example, we made them 10 for bidding and an additional 20 for acting; in the real case, the numbers would be dependent on the actual execution. In round 2, the winning agent does not make a successful action (it might have attempted (2,1)→(3,1)→(4,1)→(5,1) which is not valid because it leads to a deadlock) and it therefore pays a special tax of 100. In the end, the bank buys the state for a final reward of 1000. The agents may not be the most efficient ones, but they both made profit.

5.6 Agent creation and elimination

We start out with a set of randomly generated agents. All agents start life with a specified amount of money. Over time, as weak agents are eliminated, new random ones will be generated. We artificially keep the environment in balance by setting upper and lower bounds on the number of agents that can exist at a given time.

Apart from that, rich agents are allowed to create offspring. The genome of the children is a mutated version of their parents': the mutation can take place in the bidding expression or constants, or in the acting expression. Those children also share a percentage of their profit (or loss) with their parents (recursively).

In the special situation when no agent wants to bid for a state, new agents will be generated until one of them is willing to bid. This agent is added, and the system is allowed to go its course. This guarantees that the system cannot get stuck.

Agents are eliminated if they are bankrupt, or if the population limit is reached and another agent is added (for example in the situation mentioned above). In that case, one of the existing agents has to be eliminated, and we pick the one with the lowest score:

$$score = wealth + c * \frac{\sum auctions_{won}}{age}$$

This formula for the score was fixed *ad hoc*, with a parameter (c) that makes a trade-off between activity and economic success.

5.7 Behavior

Let's first look at some behaviors that are automatically eliminated by the system:

- Agents that bid too high lose in the long run, because this reduces their profit.
- Agents that bid too low will go broke because they lose money each round from taxes but they do not gain enough to compensate for that
- Agents that have a tendency to lead the state into a deadlock are punished directly if the deadlock can be detected, and indirectly because eventually no other agent will want to bid on a blocked state.
- Agents that are not acting on the state will continuously lose money until they are broke (this is sped up by the special tax).
- Agents that have very complex S-expression, or whose actions take a lot of time will lose a lot of money on taxes, and so they can only survive if their actions are very appropriate.

If all goes well, we reach an **equilibrium** where a small set of agents do all the work. During a run, their bids will continuously increase, until the level is solved (the last agent's bid is almost as high as the reward for solving the level). All involved agents make a profit that is high enough to cover their expenses (i.e. taxes).

There are other possible behaviors of the system, that lead to a less desirable state:

- a too high rate of turnover among the agents leads to an unstable pool of agents, which cannot end up in an equilibrium.
- a successful agent or group of agents can produce a large number of children[2], and this clan can dominate the economy so that there is no innovation anymore.
- high taxes make the environment too hard to survive in, even for 'good' agents: it leads to a system with only random agents.
- an agent can accumulate so much money on some levels that it is capable of dominating and thus immobilizing the economy for a long time on a level it cannot solve.

5.8 Seeding

In some simulations we make use of the special mechanism of **seeding** the system originally with some hand-coded agents. We write them in a way that we consider to be close to optimal for the given task (or at least close to how we would solve it). However we try to keep the code as simple as possible, so that the probability of randomly generating the exact agent is still acceptably large. We use seeding for levels that the system cannot solve, or for those which it can but we believe that the effort it requires is too large.

We keep track of how well those agents perform in the artificial economy, if they survive, and by which agents (if any) they are superseded. If they perform well, we take them out again, one after another, and see how and how quickly the system replaces them by evolved agents. If this replacement does not take place, we try to analyze the reasons for that: one possibility are agents that are quite complex, but that could not survive if amputated by one part of their code, i.e. they can not be evolved incrementally.

If the seeded agents do not perform well, we either did not code them well enough, or there are flaws in the workings of the economy (usually inappropriate parameter values). Thus, in any case, we can incrementally improve our system.

[2]We have a parameter that restricts this number and makes the scenario quite unlikely

Chapter 6

Experimental Setup

In brief, we test our system on individual levels or on sequences of levels, while in the same time varying the instruction set. We then analyze the outcomes qualitatively and quantitatively.

6.1 Simulation modalities

We consider a level to be **solvable** at the moment the system solves it for the first time. This is not powerful enough as a criterion though, because it might have been lucky or found the solution by trying many random manipulations. We say that it **understands** a level, if it can *repeatedly* solve it efficiently, i.e. the large majority of agents that act on the state actually contribute to its solution. We determine this by looking at the proportion of fruitless actions from start to solution[1].

At the moment a level is solved for the first time, money flows into the system in form of the reward. This is the first positive feedback, which can speed up the evolution, up to the level of understanding.

The simulator will have evolutionary runs on one level at a time, until it considers that the level is understood, or until a specified number of runs have been completed unsuccessfully. Then it repeats the procedure with the next level in the set.

If a run ends up in a (detectable) deadlock, we **restart** from the original position of the level, while keeping the agents (because the economic environment will eventually eliminate the responsible ones, even without our interference). If this happens a certain number of times, we **skip** the level.

For those deadlocks that we cannot detect the situation is slightly different: as we cannot distinguish a long but fruitful run from a random erring in an undetected deadlocked state, we have to make a trade-off: we limit the number of auctions in a run. We fix it to a number that we deem largely high enough for getting to the solution, and restart after reaching that limit. As before, the number of restarts is limited itself (see also appendix B).

Once a level is understood, we halt the evolution on that level. We are aware that may therefore not evolve the most efficient agents and the most appropriate bids. We are making this trade-off because at this stage of the project we are more interested

[1]A parameter determines the exact proportion, but it is ususally around 25%.

in qualitative results: we want to see how much effort and which instructions are necessary to understand a level.

After the evolution is halted, we do an additional experiment that we call **cold run**: we disable the generation/elimination of agents (this means that the pool of agents is kept artificially constant), and have the system solve the last level again, while observing the auctions.

6.2 Level sets

A run is either made on one level, or on a sequentially ordered set of levels. In the latter case, we assume that all levels but the last one are easy, and measure the performances only on that last one. The difference is that in one case the level is attempted with a starting pool of random agents, whereas in the second case that pool contains already a few agents that have proved to be useful on the previous levels in the set.

We now list all the level sets we used, and give them an identifier, for reference in the experimental data. The singleton with only the last level of the set *setname* is noted $setname_0$.

A1 contains only the level shown in figure D.1, page 49.

A2 adds the level shown in figure D.2 to the set A1.

A3 adds some more levels that have only one barrel to A2, the last one is shown in figure D.3.

A4 adds some more, longer, levels that have only one barrel to A3, the last one is shown in figure D.4.

B1 add some levels that have two barrels to A4, the last one is shown in figure 5.2, page 29.

B2 adds some more, longer, levels that have two barrels to B1, the last one is shown in figure D.5.

C1 B2 plus the first ‘interesting’ level (figure 2.3, page 8).

Similarly, the sets Cn (with $n \in [2, 90]$) are defined to be the extensions of C$n - 1$, with the last level being one of the traditional levels (number n) from Hiroyuki Imabayashi. So far we only built the first few of these sets, because the system was unable to solve any others than C1. Furthermore we have the tools in place to build much larger sets which would include simplified versions (see section 2.4) of the last level in front of it. This is supposed to favor incremental learning, but we have not exlored this option for lack of time.

6.3 Instruction sets

A certain number of basic instructions is always available to evolution: the minimal set which satisfies the condition that if any instruction was removed, some trivial levels[2] would not be solvable anymore (not to talk about understandable). These basic instructions are:

- `int`, `push`
- `allPushes`, `focusBarrels`

[2]The ones shown in figures D.1 to D.4 (page 49).

- `allStorage`, `usableStorage`, `reachableStorage`, `bestStorage`
- `biggestZone`, `neighborZones`
- `randselect`
- `pushBarrel`, `executeSequence`
- `getBarrelTo`

We vary the content of the instruction set by enabling or disabling different groups of instructions. Those groups are the following:

Logic: this group enables logical operators, like `if`, `and`, `not`.

Arithmetics: this enables all other mathematical functions, allowing more expressive power for determining the bid.

Loops: enabling this group allows looping S-expressions.

Low-level access: enables the instructions that can manipulate positions and directions directly (i.e. next, distance). This opens up more possibilities, but also increases the probability of generating meaningless code.

Advanced concepts enables the modules `getPusherTo` and `storageOrder`. This includes the instructions `bestStorage`, `getPusherTo`, `getStrictBehind` and `getBehind`. These are usually more expensive but also very powerful.

Focus: if this group is enabled, no barrels are originally in focus but instructions are available to increase or reduce the focus. If it is disabled, all barrels are in focus all the time.

Shortcuts: enables the shortcuts of table 4.5.

6.4 Measures

As mentioned in section 6.1, the *qualitative* measures are:

- is the level solvable?
- is the level understood?

We limit ourselves to only two *quantitative* measures, namely the total number of auctions required to achieve the solved/understood state and the number of auctions in a cold run (after the level is understood). We take the average of these numbers over multiple simulations.

There would be different measuring criteria, but we have picked the number of auctions because it is a good indicator of the evolutionary effort. We believe that it is more objective than the execution time which is affected by implementation details, processing power and processor load. Furthermore that measure would be subject to large irregularities, e.g. in some cases, agents are generated that use extensive amounts of computational power.

Note. To give a general impression of the computational effort involved in our simulations, we can give an average execution time (on an average PC) for 10^4 auctions: 30 minutes for a simple level and a reduced instruction set, 90 minutes for the full instruction set on a moderately large level. The latter is strongly influenced by the parameters that specify timeout limits. Fortunately the limit of 10^4 auctions is usually not reached because often, results are found significantly earlier.

Chapter 7

Empirical results

This chapter presents our empirical results. The different tables correspond to different instruction sets. Each table shows the performance of the system on the different level sets. The numbers are the number of auctions needed (applied on the last level of the set) to attain a certain qualitative state. These numbers are averages over 10 to 15 runs (more if the variance was high). The column named 'previous' shows the number of auctions spent on average to understand the preceding levels in the set; it is only applicable if the set has more than one level. Only runs that achieve the qualitative state are counted in the average, e.g. if B1 is understood only 10% of the time, the number shown in the cell understood/B1 is the average over only those runs. The limit for the total number of auctions is 10^4 for all runs. All other parameters of the system have been kept constant (see also appendix B). We will further interpret the data in the next chapter.

7.1 Basic instructions

The first batch of runs is based on the simplest instruction set. The table 7.1 shows the results. Level sets A1-A4 are reliably understood, B1-B2 are reliably solved, but rarely understood and not a single run has solved any of the C-levels during our experiments.

For the sets A1-A4, the solution was almost always given by a single agent. As it solved the levels by itself, the bidding expression is not relevant, it only needs to be high enough to keep on outbidding all other agents. Its action expression was usually similar to:

executeSequence(getBarrelTo(randSelect(focusBarrels), randSelect(usableStorage)))

but sometimes the expression for a simple random push dominated:

pushBarrel(randSelect(allPushes))

7.2 Adding direct-access instructions

This batch uses the basic instruction set and the instructions of direct access. We did a small-scale experiment to determine if this changes the qualitative results; we test:

- A4 to see how the performance on the simple levels changes,

Level set	solved	understood	previous	comment
A1	1.0	1.0		
A2	7.1	11.4	4.6	
$A2_0$	3.8	3.8		
A3	1.7	1.9	7.8	
$A3_0$	3.1	4.6		
A4	12.3	15.5	9.8	
$A4_0$	10.5	17.4		
B1	60	493	18.9	
$B1_0$	88	282		understood 25% of the time
B2	37	1083	38.4	
$B2_0$	45	1342		
C1	-	-	-	
C2	-	-	-	
C3	-	-	-	
C4	-	-	-	
C5	-	-	-	

Table 7.1: Results: basic instruction set

Level set	solved	understood	previous	comment
A4	3.0	80	10.9	very high variance for understanding
$B1_0$	60	228		understood 55% of the time
$B2_0$	98	656		
C1	-	-	-	

Table 7.2: Results: basic and direct-access instructions

- $B1_0$ and $B2_0$ to see if they are now easily understood and
- C1 to see if a harder level can now be solved at all.

The results are not significantly different from before, as table 7.2 shows, some runs are a little better, some are a little worse. The winning agents are similar to before.

7.3 Adding general-purpose operations

This batch uses the basic instruction set plus logical and arithmetic operators as well as the loops (not the instructions of direct access anymore). We did another small-scale experiment to determine if this changes the qualitative results. Again, table 7.3 shows that the results are not significantly different from using only the basic instruction set. The main difference in the agents is that the bidding expressions are becoming more complex, which increases the overall execution time but might be an advantage at a later stage.

Level set	solved	understood	previous	comment
A4	14.0	15.2	31.2	
$B1_0$	70	208		understood 50% of the time
$B2_0$	111	680		
C1	-	-	-	

Table 7.3: Results: basic and general-purpose instructions

Level set	solved	understood	previous
A4	14.4	32.5	11.8
$A4_0$	9.1	9.8	
B1	13.4	55	19.4
$B1_0$	13.5	53	
B2	37.5	220	15.0
$B2_0$	43.8	298	
C1	165	-	21
$C1_0$	273	-	
C2	-	-	-
C3	-	-	-
C4	-	-	-
C5	-	-	-

Table 7.4: Results: full instruction set

7.4 All instructions

Our last experiment uses all instructions, including the shortcut expressions but not those related to focus (we consider it to be too early to introduce them). Table 7.4 shows the results for all level sets except the easiest. The following action expression (or a variation of it) was the most commonly found among successful (i.e. rich) agents:

or(bringBarrelsHome, openUpAZone)

The most important result is that the last level of C1 (see figure 2.3) can now be solved, even if it is not yet understood. Also, the performance on the levels in B1 and B2 is significantly improved.

7.5 Cold runs

For all the levels — except the ones of the C-sets, which have not been understood — the cold runs were very quick, usually involving only a single or very few agent actions. The only exception are the cases where the level can be efficiently solved by always doing a random push (i.e. every push leads us closer to the solution, like in the level shown in figure D.3): there the cold runs would be sometimes made up of the required number of such random pushes.

Chapter 8

Discussion

8.1 Critical Evaluation

Using the basic instruction set, the system solves levels with one barrel easily. This is not surprising because once the system has evolved an agent whose action is always

executeSequence(getBarrelTo(randSelect(focusBarrels), randSelect(usableStorage)))

all levels with one barrel are trivially solved by a single action of that agent. In other words, what we usually measure in the sets A1-A4 is how quickly such an agent evolves. The key here is the module `getBarrelTo`.

We observe that the performance is generally better if the system has been trained on easier levels before having to solve the last one. This is a common-sense result, and it suggests that our approach of using incrementally harder instances is reasonable.

The exception to this rule is $A2_0$ which is solved faster from scratch than by a set of agents evolved on the early levels of A2. We can understand that by looking at the actual agents: in fact the preceding levels biased the agents towards doing random pushes (this led to the most efficient solution for those levels) but this is not good for $A2_0$. Encountering this negative consequence of the incremental approach at such an early stage indicates that we should use it with care, and critically reevalute it continuously, as the system becomes more complex.

When comparing the effort necessary between solving a level for the first time and understanding it, we realize that for the sets A1-A4 it is very similar, and for some levels it is even exactly the same (A1, $A2_0$). This means that an agent that can solve the level has ususally understood everything about it as well, i.e. it is not probable that the level can be solved by pure luck.

For the levels B1-B2, this gap opens very wide, which means that our instruction set is less well adapted to understanding them. In fact, when observing the agents in detail, we realize that the system does not gain an understanding in a way we would like it to, it merely manages to trick our measuring criteria into believing that it does, while in reality continuing to make random choices. This also explains why $B1_0$ does not always reach the understanding criterion.

For the far more complex levels in the C-sets the basic instruction set, even with its first extensions are not sufficiently powerful to lead to a solution. Only when introducing in addition the modules `storageOrder` and `getPusherTo` can one of them be solved. The fact that it can not yet be understood shows that even the full instruction set is not powerful enough. We are either missing some more fundamental concepts, or we need to improve the possibilities of combining them.

This same full instruction set is leading to good results on understanding the B1-B2 levels; the power of the instructions is sufficient for them.

When looking at the successful agents for all experiments, we realize that the tendency seems to go toward very simple single agents, that are able to solve a level by themselves. We can explain that by the fact that there is a reward only at the very end of a solution, therefore cooperation is necessarily slow to evolve: agents start out looking for their own immediate reward, and single all-purpose agents can achieve that by simply keeping the control over the level.

Our good results concerning cold runs are not a surprise, as by our definition the system is only considered to understand the level if it is doing very efficient cold runs. Unfortunately we have not yet understood levels that are complex enough so that analyzing the respective cold runs in detail would be interesting.

8.2 Objectives revisited

Now we will try to analyze the previous parts from the perspective of the questions that we intended to resolve originally:

1. *Does the problem space have an underlying structure that can be described sufficiently well with concepts?* We believe that we can answer this question affirmatively: our analysis of the problem domain naturally led to such concepts, and we could successfully found the core of our evolved program on hand-coded modules that correspond to them.
2. *How difficult is it to code those concepts manually?* We have successfully coded some useful concepts, low-level and high-level ones. Some of them were necessary for our progress but involved complex algorithms (e.g. `getPusherTo` module).
3. *How efficiently can evolution combine them?* We have found that if the representation language and the instruction set are flexible enough, then evolution can find good combinations very quickly, i.e. in at most a few hundred iterations.
4. *Can we evolve a compact program that solves simple Sokoban instances?* Yes we can, and we can even evolve such a program quite easily, based in only a few fundamental concepts (and the corresponding instructions).
5. *How well does such a program scale to hard or very hard ones?* We have observed a moderate success in scaling from easy to medium instances. It does not permit a prediction on how well the system will scale to hard ones, but at least it allows us to stay optimistic.

8.3 Future Work

Now, as mentioned before, this project is meant to be but the first stepping stone of a much more ambitious project. It was done in less than six months, whereas the whole project is currently expected to last at least for three years. This is one of the reasons why this section is as large as it is: we try hereby to sketch an outline of what the next stages of the project may involve.

During our work, we have identified quite a few dead ends, but also a large number of paths branching off into different directions, which might lead to a deeper understanding.

Straight ahead

The following are the logical continuations of our work, they do not involve new ideas, but more time to develop the elements we have introduced.

More experiments: with our current experimental setup, we could invest many more days of execution time

More levels: we have based our experiments on a small set of simple levels; using what is available in the web, appropriately scaled down if necessary by the techniques we have seen, we could achieve much more robust results, and probably more complex agents as well.

Better versions of existing modules: we have described some versions that are not yet implemented, but certainly useful. There may be even more powerful versions, as soon as new concepts are available for use.

New concepts: when adding new levels, new high-level concepts will need to be developed, that make them solvable.

More instructions: these new concepts should be made usable by incorporating them in appropriate instruction. Furthermore our agents are, so far, quite restricted as to how much they can perceive about the world (for appropriate bidding), just as they have only a reduced set of possible actions. So the set of instructions should be expanded to include new ones that combine old and new concepts in a variety of ways. This might go hand in hand with careful expansion of the set of shortcut instructions.

Extended language: the current representation is far from being able to code any given algorithm. However we might have to expand on it for more representative power, i.e. by adding variables, or new control flow instructions.

Tuned parameters: we have not devoted much time to tuning the parameters involved in controlling the evolution and the economy. Much remains to be done to achieve the greatest diversity while properly rewarding the most efficient agents (see also appendix B).

Systematic use of seeding: so far, we have used the technique of seeding only for testing purposes: we believe that a systematic use of it may lead to significant improvements.

Along the way

Here are a few new ideas, which fit well into our current approach, but their usefulness remains to be proved:

Systematic building of level sets: we have determined the level sets subjectively, but although it is not easy to classify levels objectively (see section 2.4), a more systematic approach to building level sets should be attempted.

Crossover: additionally to mutation, this could be used to better exploit existing knowledge in the evolutionary process: a natural (although simplistic) way of doing this would be to take the bidding expression from one parent and the acting expression from the other.

Undo instruction: giving the power to undo manipulations of their predecessors to agents might completely change the behavior of the system, and might even be an important step toward resolving the causality problem

Bidding on failure: this idea of allowing negative bids is introduced in section 5.3.

Faster implementation: it might become necessary to make the implementation more efficient, for example by recoding some recurrent loops in C (see appendix A).

Evolved concepts: we predict that some concepts will be very hard to code by hand, but given the whole system that will be developed when they are encountered, it might be worthwhile to use a similar evolutionary framework, no longer to solve levels, but to evolve code for a specific concept.

Alternate paths

There are some alternate approaches, which do not fit immediately into our current work, but whose pursuit at a later stage might turn out to be a fruitful complement or replacement for our strategies.

Subgoals and planning: a significant part of human strategies to solve Sokoban instances involves some degree planning, sometimes with very clearly specified subgoals and constraints (i.e. get barrel A to position B before touching barrel C). It is an open question if that behavior could somehow be realized by our system of agents.

Chunking: we are already using shortcut instructions that are derived from frequently used expressions. We could imagine an extension that would automatically determine which instructions to chunk together into new ones, and that would manage this pool efficiently (i.e. instead of only accumulating new instructions).

Different representations: we have based our representation language on S-expressions, but for the sake completeness it would be good to try different representations and compare the results.

Meta learning: so far we have few mutation options but already a certain set of parameters. One could imagine of extending the options and then optimizing the parameters by means of meta-learning. In that case our system would also contain a number of creation-agents, which do not act themselves but produce offspring of the active agents by different means; they then participate in the profit of their creatures. For example, these creation-agents might attach a specific probability to each instruction for the purpose of generating the genome of new agents.

Psychological parallels: helpful information for this project might come out of a systematic psychological study on how human players (especially beginners) learn concepts in Sokoban, what these concepts are, and how they use them.

On the horizon

We are not able to see further into the future, but we can suppose that if the project progresses successfully and Sokoban is solved, its priorities will shift, mainly toward making the concepts generalizable to other problem domains. At that point there will be opportunities to use this knowledge in concrete applications, for example in robot motion planning problems [14]. It might also stimulate research on the nature of human understanding.

8.4 Conclusions

With an approach based on Occam's Razor and the idea that understanding means exploiting underlying structure, we have attempted to evolve compact concept-based code in the specific domain of Sokoban. We have gained insights into how hard this problem is and developed a framework in which to tackle it. Using concepts derived directly from playing Sokoban, an adapted representation language and the Hayek economic system, we laid the foundation for evolving such compact code. We have demonstrated experimentally that our approach is viable because our system can solve interesting Sokoban instances. It remains to be seen if it will be able to outperform the competing approaches to Sokoban, and how well the approach will generalize to broader problem domains.

This research has been a promising start with a relatively original approach, but much remains to be done before it can achieve its ambitious goal.

Appendix A

Implementation

A.1 Programming language

We use **Python**, version 2.3.1. Here is a short description of the language, taken from [16]:

> Python is an interpreted, interactive, object-oriented programming language. It combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems. New built-in modules are easily written in C or C++. The Python implementation is portable: it runs on many brands of UNIX, on Windows, OS/2, Mac, Amiga, and many other platforms.

We consider this language to be appropriate for our purposes, because it is a high-level language, has a simple syntax, and allows easy prototyping. Also, although we are not yet concerned with optimizing performance, it is good to know that we can achieve that at a later stage by recoding some parts in C, without rewriting the whole project.

Note. We did our main developement on the Windows platform, but executed some of the simulations on a Linux machine. At that time, no Python version 2.3.1 was available for that platform, so we had to make a number of minor adjustments to make the code compatible with version 2.2.

A.2 Tools

We have developed a few tools that contribute only indirectly to the project.

File storage: we used the `pickle` module of Python to store level collections as well as complete environments with evolved agents [1] .

[1] The `pickle` module does not permit the storing of function objects (like those forming the instruction set), so we have to remove the references to the instruction set from all agents before saving and reestablish them after reloading from a file.

Logging: we wrote our own little tool to take care of logging the information of simulations, and writing them into files.

Performance tests: we used the profiling module (`profile`) that comes with Python, to analyze which parts of our code use up most of the execution time. We used it mainly to make the access to our main datastructure reasonably efficient, not on the evolutionary system anymore.

GUI: at an early stage we built a small graphical user interface as well, but it did not prove very useful, and we did not keep the new versions of our framework integrated with it.

Appendix B

Parameters

A large number of parameters are involved in this system, and while some are critical for the delicate balance described in the last section, many of them influence the efficiency of the evolution. As most of them have been set to *ad hoc* values and have not been further investigated, we will not give any numerical values. We will however try to give qualitative constraints where we can.

It is explained elsewhere (chapters 6 and 7) how different instructions can be enabled or disabled and how that influences the whole system.

Generation and mutation

The goal here is to favor a large diversity of S-expressions and at the same time not to make them more complex than necessary. For that we can tune:

- probabilities influencing how S-expressions are generated
- average depth of S-expressions
- probabilities for different kinds of mutations

Economic balance

We want to evolve many agents, let each a chance to be make profit if they are good, and eliminate them as quickly as possible if they are not. Furthermore we want to keep all the good properties of a working Hayek economy. All the following influence this domain:

- initial wealth
- final reward
- punishment taxes, e.g. for deadlocks
- taxes on using instructions
- difference in these taxes between bidding and acting
- wealth required for investment in offspring
- distribution of generated bidding constants

Stable system

We want the execution time to be regular enough for repeated experiments, so we have to avoid time traps. The following parameterize the means we have to guarantee this:

- minimal and maximal number of agents
- timeouts, different for bidding and acting
- number of auctions before restarting
- number of restarts before skipping the level
- how to increase all these numbers with increasingly difficult levels

Appendix C

Pseudocode

C.1 Dynamic structure: graph of zones

We have built a datastructure that keeps track of zones (as defined page 14) and barrels. It is an undirected graph where each node is either a barrel or a zone, and if two nodes are linked, that means that they have adjacent spaces on the level grid.

Each push modifies the graph, using two auxiliary functions that refer to it:

```
FUNCTION PushBarrel (fromPos, toPos)
   IF push is legal THEN
      move Pusher in front of fromPos
      RemoveBarrel (fromPos)
      AddBarrel (toPos)
   ENDIF
END FUNCTION

FUNCTION RemoveBarrel (Pos)
   SET N to the node corresponding Pos
   remove the barrel at Pos
   define N to be a new zone
   FOR each node Ni that is connected to N in the graph
      IF Ni is a zone (not a barrel) THEN
         merge Ni with N
      ENDIF
   END LOOP
END FUNCTION

FUNCTION AddBarrel (Pos)
   SET N to the node corresponding Pos
   add a new node B
   define B to be a new barrel at Pos
   IF the zone N can be split at Pos THEN
      split N into the zones N1 and N2
      remove N
      add N1
      add N2
      connect N1 to B
      connect N2 to B
      FOR each node Ni that was connected to N
         IF Ni is touching N1 THEN
            connect Ni to N1
         ELSE
            connect Ni to N2
         ENDIF
      END LOOP
   ELSE
      connect B to N
   ENDIF
END FUNCTION
```

C.2 Static structure: graph of chambers

We construct a directed graph of chambers using the following algorithm:

```
create a directed graph G
FOREACH space S
IF S is not a wall AND S is not forbidden THEN
      create node N containing only S
      add N to G
   ENDIF
END LOOP

FOREACH two adjacent spaces S1, S2 and their respective nodes N1, N2
   IF a barrel could get from S1 to S2 in one push THEN
      link N1 to N2
   END IF
END LOOP

WHILE something changes
   FOREACH two nodes N1, N2
      IF G contains a path from N1 to N2 and a path from N2 to N1 THEN
         merge N1 and N2
      END IF
   END LOOP
END LOOP
```

Appendix D

Some levels

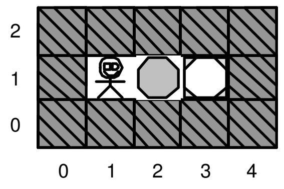

Figure D.1: The simplest possible level.

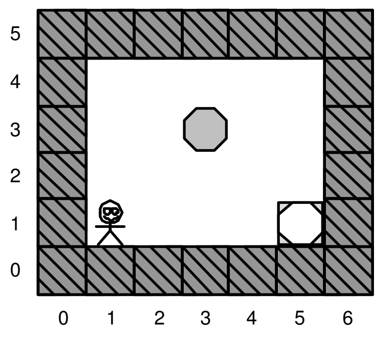

Figure D.2: A very simple level with one barrel.

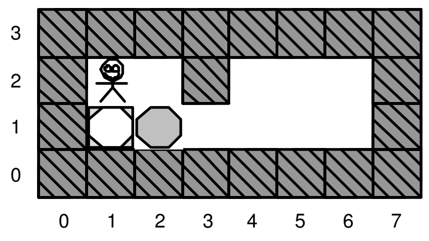

Figure D.3: A simple level with one barrel, introducing a concept.

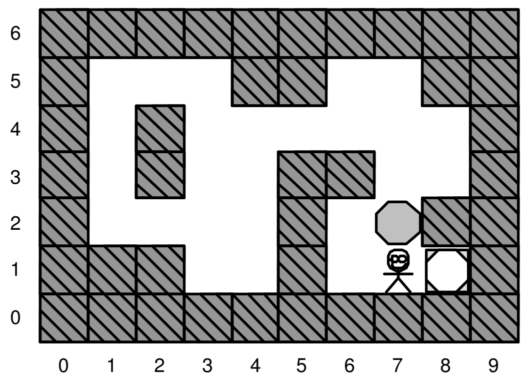

Figure D.4: A level with one barrel and a longer solution.

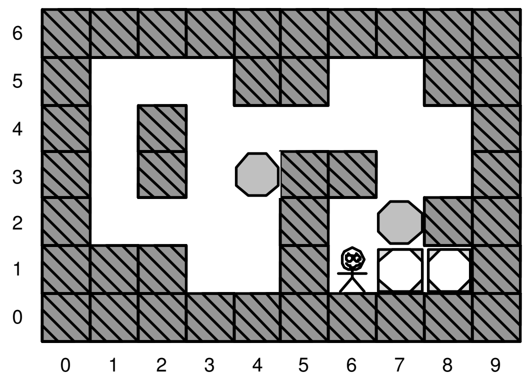

Figure D.5: A level with two barrels and a longer solution.

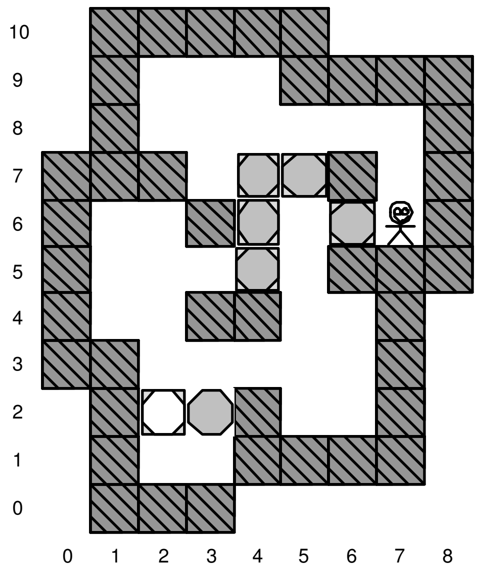

Figure D.6: A small but very difficult level from Aymeric du Peloux[17].

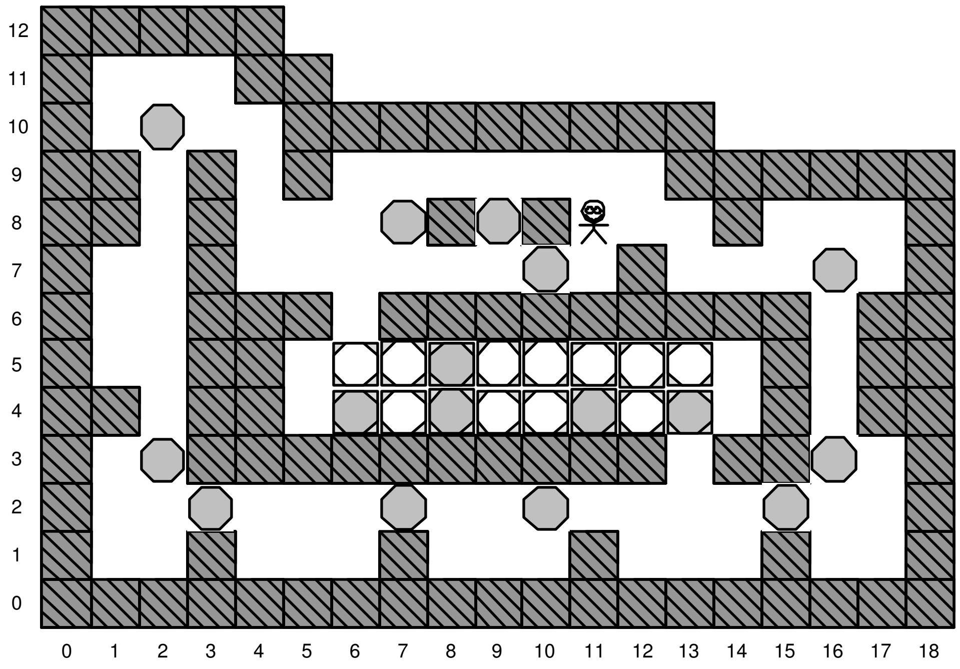

Figure D.7: Level 29 from the original set of levels by Hiroyuki Imabayashi using the concept of one-way passages described in section 2.5.

Bibliography

[1] E. B. Baum, *What is Thought.* MIT Press, 2004.

[2] N. Tishby and N. Slonim, “Data clustering by markovian relaxation and the information bottleneck method.,” in *NIPS*, pp. 640–646, 2000.

[3] X. Zhang, J. S. Fetrow, W. A. Rennie, D. L. Waltz, and G. Berg, “Automatic derivation of substructures yields novel structural building blocks in globular proteins.,” in *ISMB*, pp. 438–446, 1993.

[4] D. L. Waltz, “The importance of importance: Aaai presidential address,” *AI Magazine*, pp. 18–36, Fall 1999.

[5] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[6] J. E. Laird, A. Newell, and P. S. Rosenbloom

[7] K. Thompson and P. Langley, “Concept formation in structured domains,” pp. 127–161, 1991.

[8] A. Junghanns and J. Schaeffer, “Sokoban: Enhancing General Single-Agent Search Methods Using Domain Knowledge,” *Artificial Intelligence*, vol. 129, no. 1-2, pp. 219–251, 2001.

[9] E. B. Baum and I. Durdanovic, “Evolution of cooperative problem solving in an artificial economy,” *Neural Computation*, vol. 12, no. 12, pp. 2743 – 2775, 2000.

[10] B. Källmark, “Sokoban for windows.” `http://www.sourcecode.se/sokoban/`. A very complete, commercial version of the Sokoban game, for the Windows platform.

[11] S. Lindhurst, “Sokoban for the macintosh.” `http://members.aol.com/SokobanMac/`. A slim and popular version of Sokoban for the Mac.

[12] P. Online, “Sokoban v2.0.” `http://www.pimpernel.com/sokoban/sokoban.html`. A simple applet-based version of Sokoban, playable online.

[13] J. Culberson, “Sokoban is pspace-complete,” *Informatics 4, fun with Algorithms*, pp. 65–76, 1999.

[14] D. Dor and U. Zwick, “Sokoban and other motion planning problems.,” *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[15] D. J. Montana, “Strongly typed genetic programming,” *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1994.

[16] “Python programming language.” `http://www.python.org/`. The official website.

[17] A. du Peloux, “Sokoban.” `http://membres.lycos.fr/nabokos/`. A popular french Sokoban webpage, with a large set of unique and very small levels, all playable online.