Solving Sokoban by Searching in Feature-Space

yaronsh@google.com

This research has been done as a side project in Google Research IL.

Abstract - Sokoban is a puzzle computer game that was invented in Japan. The objective of the game is to push boxes in a maze from their start positions to target squares. Sokoban is a challenging game that requires logic, planning and abstraction, and as such it has always been of interest to the AI community. Despite many years of research, no one has been able to produce a program that plays at a human level, or even solves the standard test suite. This set of 90 levels, called XSokoban, has been used as a benchmark for most academic studies.
In this paper we present a novel search algorithm, called FESS (Feature Space Search). FESS works by projecting the playing board onto a smaller space defined by features. A solution in the original board space translates to a path from the projection of the initial state to the projection of the final state. FESS tries to find such a path in the feature space. Whenever progress is being made, this progress is rewarded with a permanent allocation of computation time.
In addition to FESS, we introduce new domain-specific concepts such as hotspots, sinks and advisors. Putting everything together, we have created the first documented program that solves all 90 XSokoban levels.

Introduction

Sokoban was invented in 1981 by Hiroyuki Imabayashi and published in 1982 by Thinking Rabbit [1]. Despite being relatively old, it is still a very popular game. New levels keep being published, and a search on Google play returns at least 100 implementations of the Sokoban game.
Sokoban levels range in difficulty from easy to extremely difficult. From a scientific point of view, Sokoban has been shown to be NP-hard [2]. However, most levels are meant to be challenging, creative and fun, so they can be expected to be solved in a reasonable time.
An early attempt to build an automated solver was the "Rolling Stone" program [3] from 1997. Timo Virkkala's Master thesis [4] surveys the research until 2011. In 2017, DeepMind presented their attempts of solving Sokoban using deep learning [5]. Recent studies have used deep reinforcement learning and Monte Carlo Tree Search [6][7].
Despite all these efforts, humans are still better at the game. To quote from the Wikipedia page: "*The more complex Sokoban levels are, however, out of reach even for the best automated solvers*" [1].
Of particular interest is the XSokoban level set [8]. This set became the de facto Sokoban test suite after Junghanns and Schaeffer used it in their early work [3]. The difficulty of these levels ranges from easy to medium. Up until now, no one has been able to develop an algorithm that solves the entire collection [9].
In this paper, we introduce a new search algorithm designed to overcome the inherent difficulties of Sokoban. The solver makes extensive use of *features* - numerical characteristics of

a Sokoban position. A substantial part of the search takes place in a space defined by the features, hence the name FESS (Feature Space Search). We show that this approach is very effective. Using the FESS algorithm, we have been able to produce the first documented program that solves all 90 XSokoban levels. Furthermore, the entire set is solved in less than ten minutes, which traditionally is the time limit for solving a single level [9].

The paper is organized as follows: In **Section 1** we describe the game and explain what makes it so difficult. We discuss some technical details about move notation and about the search tree data structure. In **Section 2** we introduce features. We begin with two simple features, connectivity and packed-boxes, and show how the game is represented in feature space. **Section 3** presents the basic framework of the FESS algorithm. **Section 4** adds two more useful features: room connectivity and hotspots. In **Section 5** we introduce advisors - domain specific heuristics that aim to reduce the effective branching factor by prioritizing moves. In **Section 6** we discuss the packing order plan for a Sokoban level and show that FESS is also useful for computing this packing plan. As a result of this discussion, we introduce another important feature: OOP (out of plan). In **Section 7** we show our experimental results for the XSokoban level set. The solver program uses FESS as its search algorithm and uses all the discussed features and advisors. In **Section 8** we conclude and suggest future research.

Section 1 - Description of the game

Figure 1 shows the first of the XSokoban levels. The player (represented here as the cool guy with the hat) can move around in all four directions. The level also contains boxes, shown as round objects. Standing next to a box, the player can push the box one square ahead, unless there is a wall or another box on the other side. The objective of the game is to push all boxes to targets, shown as red squares.

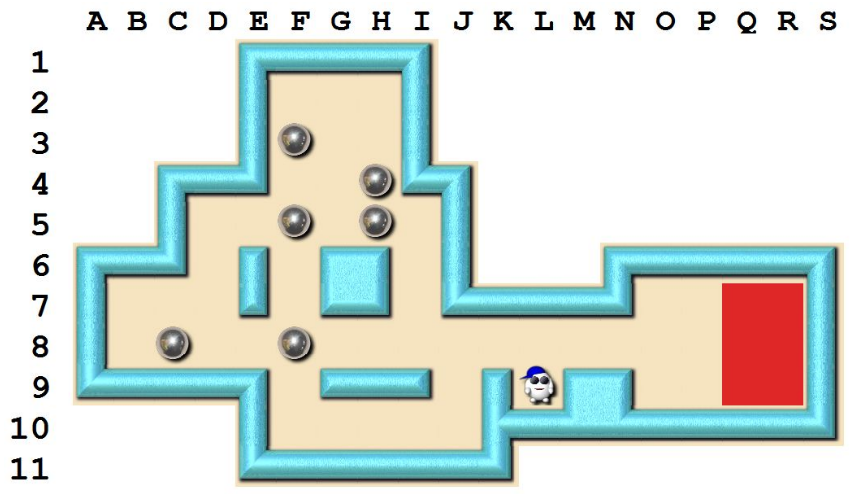

Figure 1

This may sound like an easy task, but some levels can be very difficult to solve, requiring hours or even days of thinking.

What makes Sokoban so difficult?

- The branching factor can be quite large. It can often be in the order of hundreds, as in the game of Go.
- The length of the solution. Some levels require hundreds of moves to be solved.
- The first two factors combined can make the number of possible positions enormously large. For the XSokoban levels it is estimated to be 10^{18} [3], meaning that a simple brute force approach is unlikely to succeed.
- The need to generalize or focus on subproblems. Human players usually understand when moves are “the same” in the sense that they have a similar effect on the board and on future plans. Humans are also very good at isolating relevant parts of the board and forming solutions or conclusions based on such an analysis. However, it is difficult to formalize and implement such rules in a solver program.
- The presence of deadlocks. Deadlocks are positions that can never be solved, no matter which moves are played next.

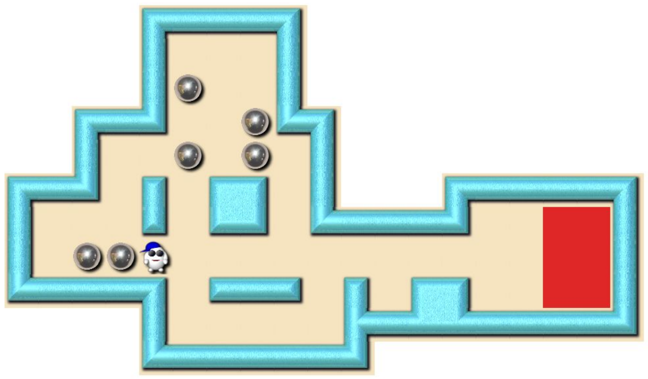
Figure 2a

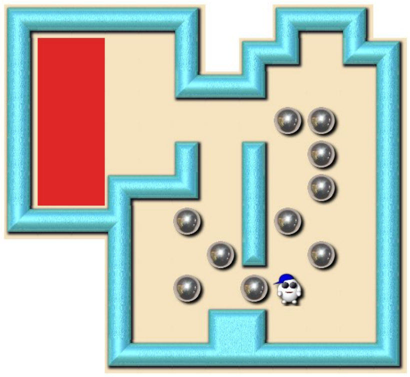
Figure 2b

Figure 2a shows one of the simplest deadlocks, a 2x2 pattern. Figure 2b shows a more complex case in which the lower left room cannot be entered.

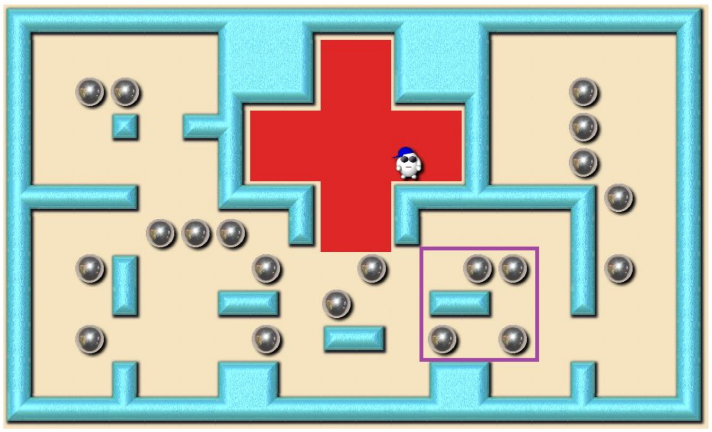

Figure 3

Figure 3 shows a more subtle example from level #61 after making an erroneous move. If any of the four marked boxes is removed, the other three boxes can reach their targets. But the four boxes are entangled is such a way that they cannot escape from deadlock.

Detecting deadlocks is a crucial skill for a Sokoban player. Deadlocks can, however, be much more complicated than shown in the examples above, and a solver program cannot always detect the situation.

This discussion has important implications for the choice of the search algorithm. BFS is a natural choice, but it is usually impractical because of the exploding branching factor. DFS uses less memory than BFS, but a DFS from a deadlocked position may last forever. The A* search algorithm requires a lower bound on the quality of the position, but in Sokoban it is difficult to find such (useful) lower bounds. Timo Virkkala's thesis [4] compares these and other search algorithms. In practice, none of them have been good enough to solve all 90 XSokoban levels.

Moves and notations

In most published Sokoban papers and solvers, the basic game action is a one step player movement in any direction, thus an action is either a push or just a move. For example, there is a 254 moves solution to level #1, out of which 97 moves are also pushes.

In our solver, we prefer to represent moves using a macro move notation. A macro move describes a push of a box from place (X_1,Y_1) on the board to place (X_2,Y_2). It does not describe the exact order of pushes and moves needed for this maneuver. In this notation, the 254 moves solution to level #1 looks like this:

(H,5)-(G,5) preparing a path to the upper room

(H,4)-(H,3) opening the upper room

(F,5)-(F,7) opening a path to the left room

(F,8)-(R,7) packing a box

(C,8)-(R,8) packing a box

(F,7)-(R,9) packing a box

(G,5)-(Q,7) packing a box

(F,3)-(Q,8) packing a box

(H,3)-(Q,9) packing a box

Now the solution is much shorter (9 macro moves instead of 254 moves).
The price to pay is a huge increase in branching factor. Theoretically, each box can now be sent not just to four neighbor squares, but to any square on the board. Using macro moves results in a larger branching factor but fewer steps compared to the usual approach with single step moves.

The search tree

The solver uses a tree data structure. Each node in the tree represents a game board position found by the search. The root of this tree is the initial position. A node also contains the feature values for the position. Features are discussed in section 2, but for the moment consider them simple characteristics of the position. For example, the number of boxes on targets is a usable feature.
In addition to the position, each node also keeps a list of all possible macro moves from that position, including the resulting feature values after making the moves.
The search algorithm explores new parts of the search tree by *expanding* a node. One of the moves from the node is selected and applied, and then the new position is added to the search tree as a new leaf node. For this new node, the list of new possible moves and feature values is computed and stored.
The search tree employs a transposition table to avoid analyzing identical positions more than once. Another functionality supported by the tree is deadlock marking. If a node has no valid moves, it is marked as deadlocked. If all children of a node are deadlocked, the node becomes deadlocked too (and this may continue upward to the root). Deadlocked nodes are ignored by the search.

Section 2 - Features

In this section we present some intuition for the FESS algorithm. Section 3 will present the algorithm in a more formal manner.
FESS works by extracting features from the game board. These features should be good indications of progress toward a solution. In this section we consider two useful features: connectivity and packed-boxes.

Connectivity

Usually, the boxes divide the board into closed regions. The player can move around freely within a region, but cannot move to another region without first pushing boxes. Figure 4 shows the four regions in the initial position of level #1. We say that this position has a connectivity of four.

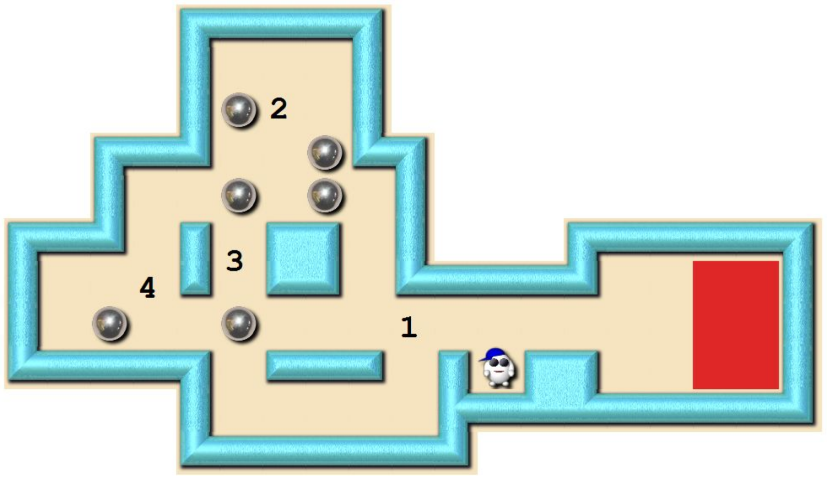

Figure 4

Usually, it is a good idea to reduce the connectivity in order to have more move options. A connectivity of one means that the player can move anywhere on the board. However, some levels require a temporary increase in connectivity in order to be solved. This is usually the “tricky” move required to solve the level.

Packed boxes

This feature counts the number of boxes that have been moved to targets according to the packing plan. In simple levels, this is usually the number of boxes on targets. In section 6 we show that sometimes, the packing plan involves parking boxes on non-target squares. For the moment, assume that this feature simply counts the number of boxes on targets.

Feature space

Feature space is the main concept behind the FESS algorithm. It is defined as a Cartesian product of features. Each element in this product is called a *cell*. A board position can be projected onto feature space by computing its feature values and selecting the matching cell.

As a very simple example, recall the solution to level #1.
(H,5)-(G,5) preparing a path to the upper room
(H,4)-(H,3) opening the upper room
(F,5)-(F,7) opening a path to the left room
(F,8)-(R,7) packing a box
(C,8)-(R,8) packing a box
(F,7)-(R,9) packing a box
(G,5)-(Q,7) packing a box
(F,3)-(Q,8) packing a box
(H,3)-(Q,9) packing a box

In this example, we define the feature space as the two-dimensional product of the features “connectivity” and “packed boxes”. Figure 5 shows the projected solution.

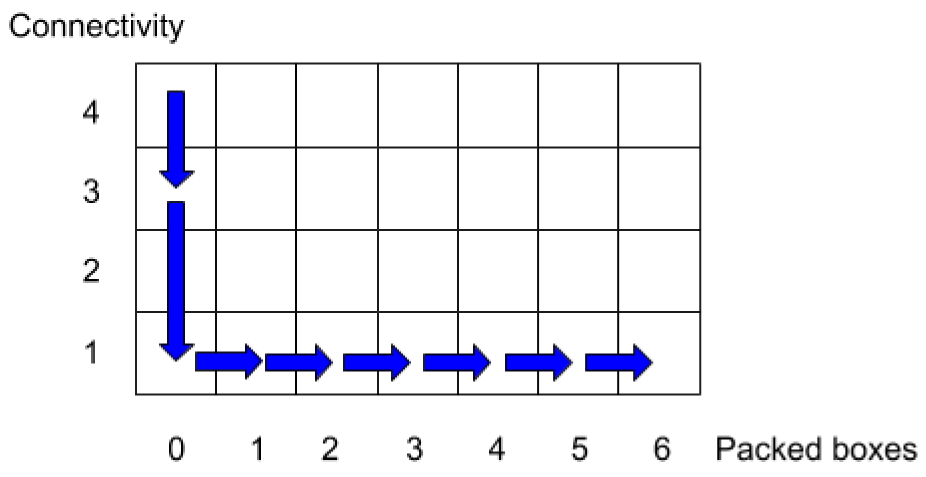

Figure 5

This example is very simple. First, the connectivity is improved and then the boxes are packed. In practice, a solution usually involves some combination of the two: Improving connectivity gives access to more boxes, and packing boxes gives more room for maneuvering. As an example, let us look at level #5 (Figure 6a). Consider the plan that pushes a nearby box up and then pushes another box down (Figure 6b).

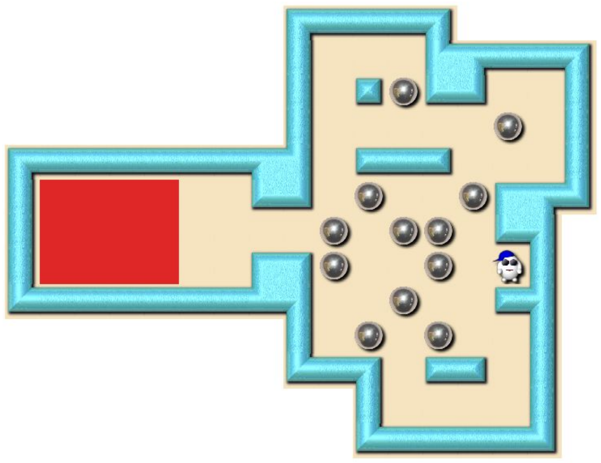

Figure 6a

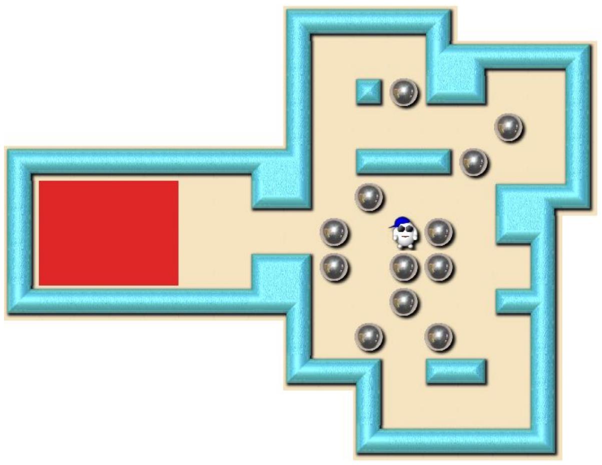

Figure 6b

The first move does not improve connectivity (it opens a new space but closes the upper right room). The second move improves connectivity by one. Now boxes can be pushed into the target area, as shown in Figure 7.

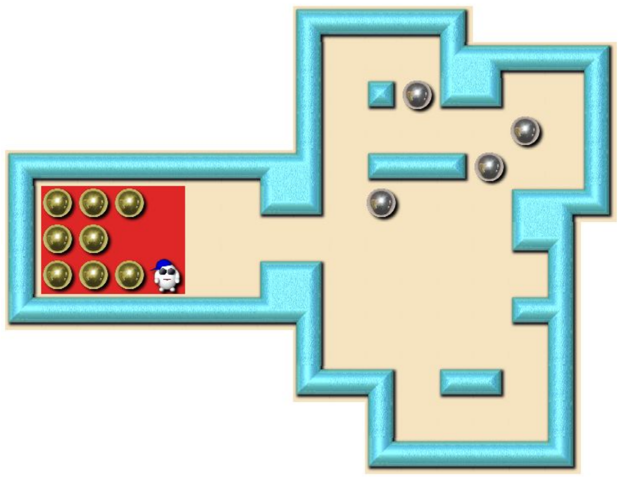

Figure 7

After having cleared the area this way, the first pushed box can now be pushed again. Then it becomes possible to pack the three remaining boxes. Figure 8 shows what the solution looks like in feature space.

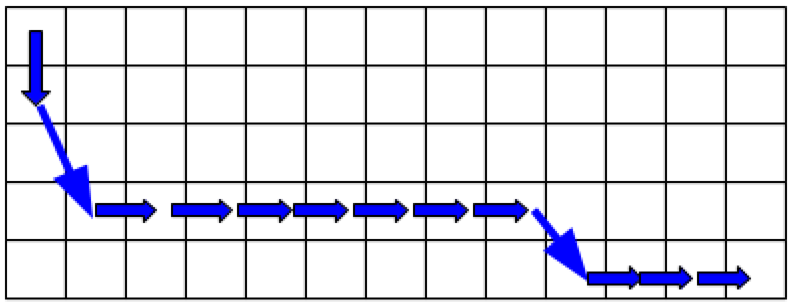

Figure 8

However, progress in feature space does not guarantee that a solution will be found.
Figure 9a shows level #1 again, this time after the move (F,8)-(F,6) which improves connectivity by two, and the move (C,8)-(R,7) which packs a box. At this point, the solver may realize that the position is deadlocked (or do a few more moves before realizing it). Figure 9b shows this plan in feature space.

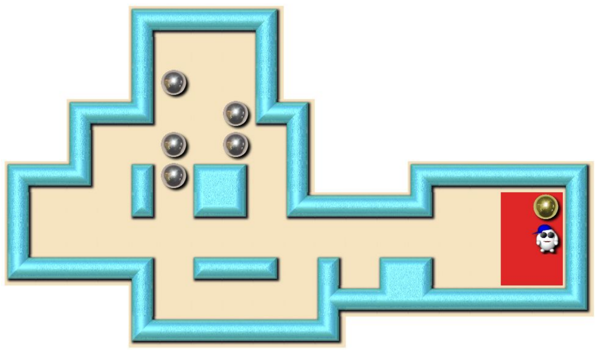
Figure 9a

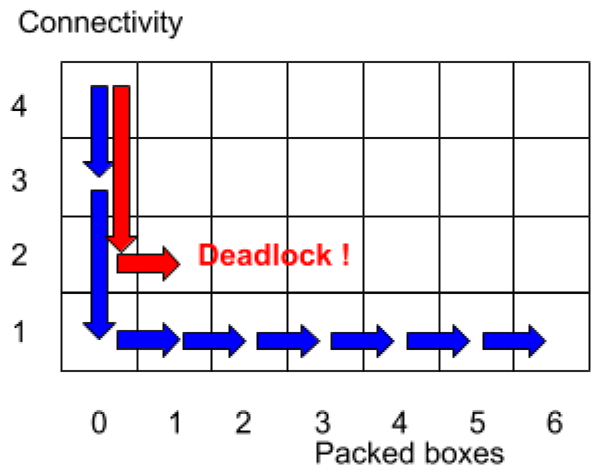

Figure 9b

This example illustrates that several plans can coexist in feature space, although some of them may be deadlocked.

Section 3 - The FESS algorithm

After this discussion of paths in feature space, we are finally ready to present the FESS algorithm.

```
Algorithm FESS (Feature Space Search)

Initialization:
Set the initial position as the root of the search tree
Assign a weight of zero to the root
Project the initial position onto a cell in feature space

Search:
While no solution has been found:
    Pick the next cell in feature space
    Consider all positions in the search tree that project onto this cell
    Consider all unexpanded moves from these positions
    Choose the move with the least weight. In case of equality, prioritize by other features
        Add the new position to the search tree
        Assign a weight to this child: parent's weight + move weight
        Project the new position onto feature space, adding a new cell if necessary
```

FESS works by going *cyclically* over all active cells in feature space. For each such cell, the algorithm examines all search tree positions that projected onto the cell. Now the algorithm

considers the possible moves from these positions, but only moves that were not already expanded. Among all these moves, the algorithm picks the move with the least weight.
We have not defined “weight” yet. For the moment, think of it as an estimate of how “bad” the move is. In section 5 we define how this weight is computed.
If several moves have the same weight, the algorithm picks the best move by using other feature values as tiebreakers. Prioritizing moves is discussed in section 7.
The selected move is applied, and the resulting position is added to the search tree. The new position is now projected onto feature space. If a new cell is created, it is added to the list of active cells.

Analysis of the algorithm

FESS was designed with the intention to address two of Sokoban’s problems: the long solutions and the deadlocks. For FESS, it does not matter how long the solution is. As long as new cells are discovered, the algorithm allocates computation time to them. The attention given to these new cells allows them to advance further in feature space. Compare this to BFS, where plans with a large number of moves cannot practically be found.

The other aspect is dealing with deadlocks. Even though deadlocked positions can advance in feature space, they will eventually stall. From that point on, whenever such positions are picked by the algorithm, they just gain more weight. Sooner or later, this allows positions on a solution path, with lower weights, to be selected from the cell. The algorithm repeatedly visits all cells in feature space, so it does not commit to a single plan like DFS. Instead, it considers all plans whether they are deadlocked or not.

Conceptually, FESS can be considered a combination of:

- A strategic plan (opening rooms, packing boxes) which takes place in feature space.
- A tactical search for the positions and moves that allow transitions between cells. This part takes place in position space.

Proof of completeness

We will now show that FESS solves all levels, given enough time.
Consider the positions along the solution path. If we ignore competing plans for the moment, these positions will be projected onto feature space, one by one. FESS considers all cells, so when it visits the cell with the current position along the path, FESS will select the position and expand its children. At that step, the search may continue to the next cell, or continue in the same cell but with a bigger weight. Note that the algorithm never prunes any moves. It may assign heavy weights to moves so they deprioritize, but eventually they will all be considered.

The next thing to show is that the next move along the solution path will be picked from its cell, despite any competing plans. Suppose that the correct move has the weight of W at this point.

Suppose also that the minimum weight assigned to a move is one. Therefore, after W+1 steps all other plans have weights greater than W, so the right move with weight W will be chosen.

The last thing to show is that the number of competing plans of length W+1 is finite. This is true because the branching factor is bounded by the level size and the number of boxes.

Comparison to other search algorithms

FESS can be seen as a mixture of several search algorithms. By changing focus between the cells, FESS can search deeper very quickly when progress is being made, much like DFS. When progress in feature space is slow, FESS repeatedly revisits cells, trying to expand to other nearby nodes. This systematic search from each cell is similar to BFS. Finally, within a specific cell, FESS prioritizes moves by their weights. Then it acts exactly like Dijkstra's algorithm, finding the shortest paths from the cell's entry point to positions exiting the cell.

Restricting the number of cells

The FESS algorithm, as presented so far, allocates new cells even if positions get worse. For example, consider a cell with two packed boxes and a connectivity of five, denoted by (2,5). Suppose that the algorithm picks a move that increases connectivity by one. If a new cell with attributes (2,6) is created, this cell will be allocated computation time despite clearly being worse. Moreover, this cell can spawn even worse cells like (2,7) and (1,6). In order to restrain this undesirable behavior, the algorithm is slightly modified. Instead of projecting the new position onto feature space, the position is associated with the cell holding its best ancestor. This way, the position (2,6) is actually projected onto the cell (2,5), but with a bigger weight caused by the extra move. In this cell, the position will have to compete with positions having smaller weights. In this way, "bad moves" are deprioritized, but they will eventually be checked. Admittedly, projecting onto an ancestor's cell is a less elegant solution than projecting onto the cell of the state itself, but it has worked very well for us. In Section 8 we suggest another way of dealing with this problem.

Section 4 - Room connectivity and hotspots

This section presents two domain-specific features used by our solver.

Rooms and room-connectivity

Sokoban levels are usually composed of rooms connected by corridors or tunnels. For a human player it is obvious what a room is: a spacious place in which boxes can be stored and the player can move around freely. Defining it formally, however, proved slightly tricky. In the end, we settled for this definition: A *room* is a connected area of 2x3 squares.

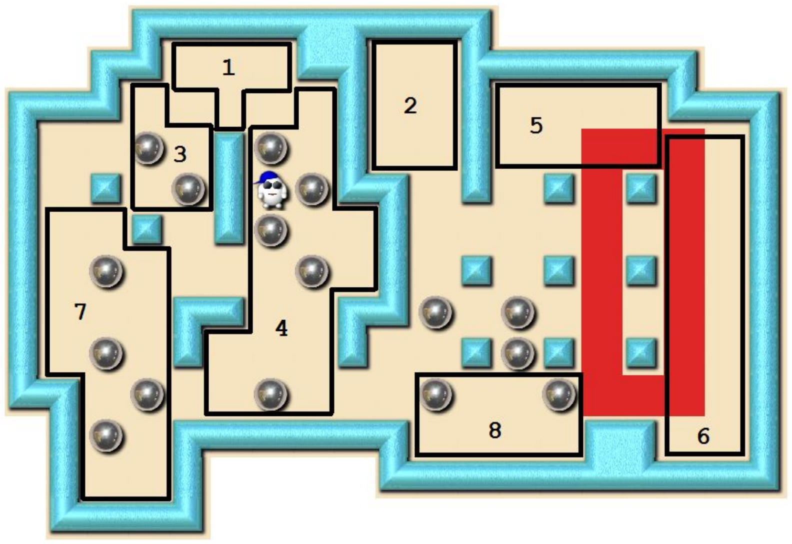

Figure 10

Figure 10 shows level #13 partitioned into rooms, using this definition. Note the open spaces near the targets. They are not big enough to qualify as being rooms, according to our definition. Note also the square connecting room 1 and room 4. By definition, this square can belong to any of these two rooms. For simplicity, we do not assign the square to any of them.

Room connectivity

For a given level, a graph can be computed which shows how rooms are connected. A vertex in this graph corresponds to a room. There is an edge between vertices if there is a direct path between the two rooms, that does not go through a third room. For the example shown in Figure 10, the adjacency matrix is:

	1	2	3	4	5	6	7	8
1	0	0	**1**	**1**	0	0	0	0
2	0	0	0	**1**	**1**	**1**	0	**1**
3	**1**	0	0	**1**	0	0	**1**	0
4	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**
5	0	**1**	0	**1**	0	**1**	0	**1**
6	0	**1**	0	**1**	**1**	0	0	**1**
7	0	0	**1**	**1**	0	0	0	0
8	0	**1**	0	**1**	**1**	**1**	0	0

The graph assumes that there are no boxes on the board.
However, in actual gameplay, some of the edges may be broken by boxes blocking the path. In Figure 10, for example, there is no direct path from (some) squares in room 4 to (some) squares in room 8.
In order to compute the broken edges for a specific room, a BFS is started from a square inside the room. This search is allowed to go through corridors and squares in the same room, but not through other rooms. If the search does not touch a neighbor room (according to the adjacency

matrix), then the edge is broken. If necessary, additional searches are invoked to cover all the squares in the room.
The number of broken edges is referred to as *room-connectivity*. Often, it can capture topological information missed by the normal connectivity feature.

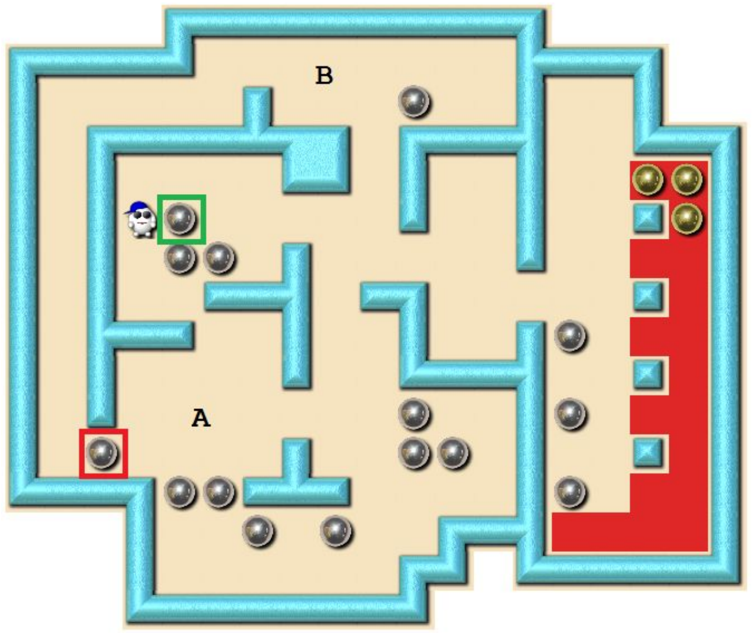

Figure 11

Figure 11 shows level #20 after some moves. In the rooms graph, rooms A and B are connected by a corridor. However, in Figure 11 the box marked in red blocks this corridor. A path from room A to room B must go through other rooms, so the A-B edge is broken. Note that the broken edge prevents the box marked in green from being pushed into the target area. However, if the A-B edge is restored, such moves will be possible. Although the connectivity is one (the player can go everywhere), room-connectivity recognizes that some maneuvers are obstructed.

Hotspots

A box is defined as a *hotspot* if turning it into a wall reduces the number of reachable targets for any of the other boxes. Figure 12 shows some examples of hotspots in level #16.

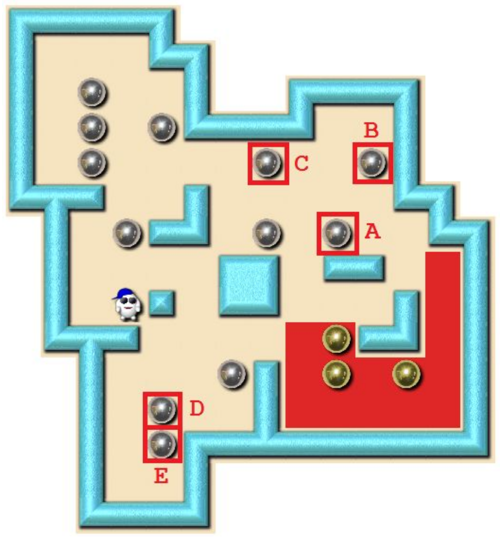

Figure 12

Box A prevents B from reaching any target as long as it remains in place.
Box B prevents C from reaching any target.
Box C blocks all boxes on the upper left side.
Boxes D and E block each other; one of them must move in order to free the other.

Note how hotspots can identify the major obstacles for solving level #16. Box A must be pushed so that box B can be pushed, thereby freeing the other boxes. Note also that box C is disruptive, but this goes unnoticed by the connectivity feature and the room-connectivity feature. In general, eliminating hotspots has the desirable effect that it improves the mobility.

Computing hotspots

At the level preprocessing stage, we check all pairs of X,Y squares for being in a hotspot relation. First, a box is placed on square X, and we count the number of reachable targets from this square. Then another box is added on square Y and we count again. If the number has changed, then square Y is a hotspot square for square X. This result is stored in a table for all pairs of X,Y squares.
During the search phase, we examine all pairs of boxes. Using the table, we can tell if a box on square Y is a hotspot for a box on square X. The total number of hotspots can then be used as a feature.

Section 5 - Advisors

As discussed in section 1, the branching factor can be very large. It can be hundreds of moves and sometimes even more than a thousand. A human player can, however, identify “similar” moves in the sense that they have the same effect on the board. For example, if two or more boxes can be packed to a target square, it usually does not matter which one is packed. If a box is pushed in order to clear the way to a room, it usually does not matter at which square the box is parked. If a box just moves around in a big room, this does not substantially change the general plan. Although it is difficult to formally define “similar”, humans are very good at understanding it intuitively.

To address the problems with the large branching factor and the moves similarity, we introduce the *advisors* concept. Advisors are domain-specific heuristics that aim to pick promising moves. Whenever a position is added to the search tree, the solver consults a small number of advisors in order to mark the most promising moves. The search prioritizes these moves, but eventually all other moves will be considered too. This prioritization mechanism is implemented in FESS by using the weight attribute associated with moves (see the algorithm description in section 3). Moves chosen by the advisors are assigned a weight of 1, and all other moves are assigned a weight of M. Choosing the value of M is discussed in section 7.
Using advisors, the solver typically considers only a small number of moves in each position. Other moves are checked only when the advisors do not find a working plan.

In this section, we present the advisors used by our solver. Each advisor is allowed to choose at most one move.

Packing advisor

This advisor considers only moves that increase the number of packed boxes, and rejects all other moves. If several packing moves are available, it chooses the best one (see section 7 for the definition of “best”). Conceptually, this advisor cares only about packing. For example, it does not matter if a packing move worsens the connectivity.

Connectivity advisor

Very similarly to the packing advisor, this advisor considers only moves that improve connectivity.

Room-connectivity advisor

This advisor considers only moves that improve room connectivity.

Hotspots advisor

This advisor considers only moves that reduce hotspots.

Explorer advisor

This advisor opens a path to a free square which enables a new push. Most of the time, this advisor agrees with the connectivity advisor. However, sometimes it finds other moves.

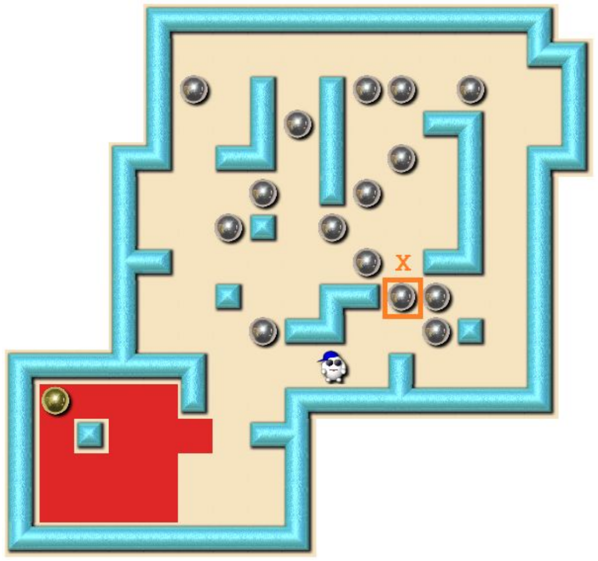

Figure 13a

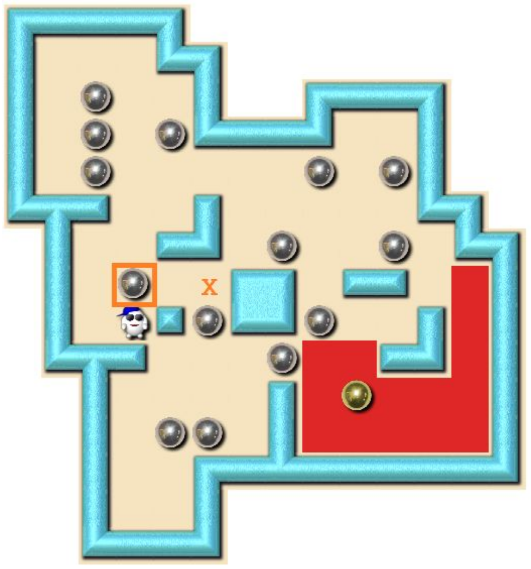

Figure 13b

Figure 13a shows an example from level #40. The explorer advisor suggests pushing the orange box up by two squares, as this gains access to square X. From this square, the player can push the left side neighboring box, a move that was unavailable before. Note that pushing the marked box actually worsens the connectivity.
The explorer advisor was designed for forcefully breaking into closed rooms. However, it turned out to have other useful behaviors as well, notably its enthusiasm for going through *doors*. Opening a door gives the player access to new areas of the board at the expense of closing off the current access area. This is a frequently occurring theme in Sokoban levels.
Figure 13b shows a door example from level #16. Pushing the marked box up does not improve connectivity, but since it gains access to X, the explorer advisor may select this move.

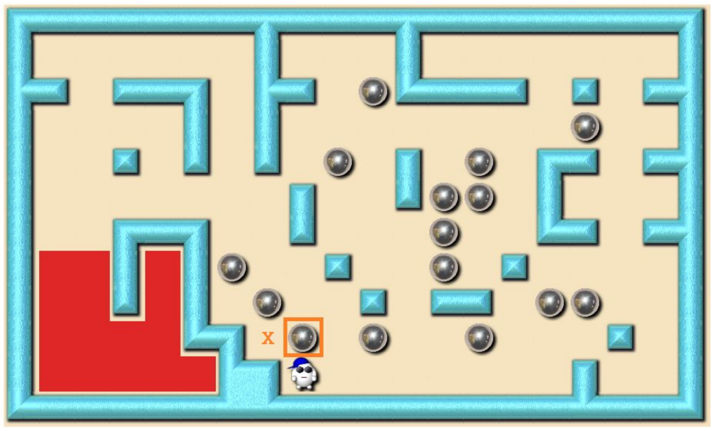

Figure 14

This advisor is also good at opening diagonals. Figure 14 shows an example from level #68. The explorer advisor will push the marked box, even though this move does not improve the connectivity. This push is indeed an essential part of the solution.

Opener advisor

This is probably the most interesting advisor. First, it finds the hotspot that blocks the largest number of boxes. Clearly, this box should be pushed away. It is, however, often blocked by other boxes. In such cases, the opener advisor tries to open up by pushing nearby boxes away from the hotspot. Only moves that do not worsen the connectivity are considered.

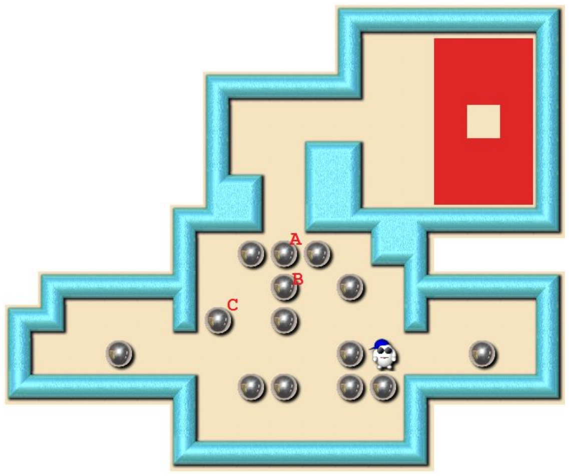

Figure 15a

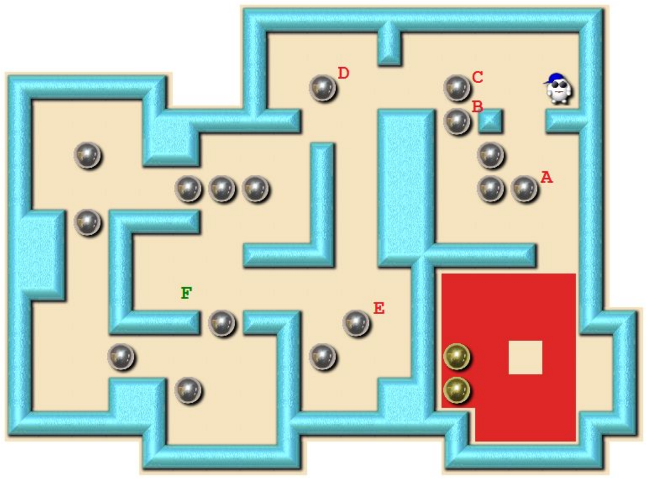

Figure 15b

Figure 15a shows an example from level #9. Here the box marked by A is a hotspot for all the other boxes, but it cannot move at the moment. The opener advisor will try to push the nearby box B, but the deadlock pattern database prunes all of its moves (see the appendix). Box C, however, can be pushed down by two squares without increasing the connectivity. In the next step, the opener advisor will push box B to the left, and in the next step it will finally clear box A.

Figure 15b shows another example, this time from level #30 after some initial moves. Box A is the hottest hotspot, but all attempts to push nearby boxes away worsen the connectivity. The opener advisor will push box E to the place marked by F; then push D to E, C to D, and after moving box B, the level is trivially solved.

The opener advisor is interesting because it is merely a simple rule: find the most disruptive box and push other boxes away from it. Yet, it often results in a behavior that almost seems like an intelligent understanding of the level.

OOP advisor

The description of this advisor requires terms we will first define in Section 6. In short, *OOP* ("out of plan") boxes interfere with the packing plan. They can be dealt with by pushing them into a *basin* area.
The OOP advisor is very similar in design to the opener advisor. First, it finds the OOP box closest to the basin. If this box can be pushed into the basin, the advisor chooses this move. Otherwise, it tries to make room for the OOP box by pushing nearby boxes into the basin. Only moves that do not worsen the connectivity are considered.

Section 6 - Packing plan

Early on, human Sokoban players learn that it is not enough to bring boxes to targets. Boxes must also be packed in a specific order.

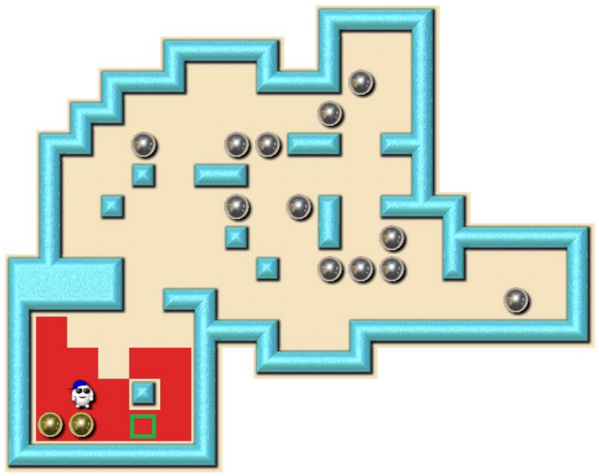
Figure 16a

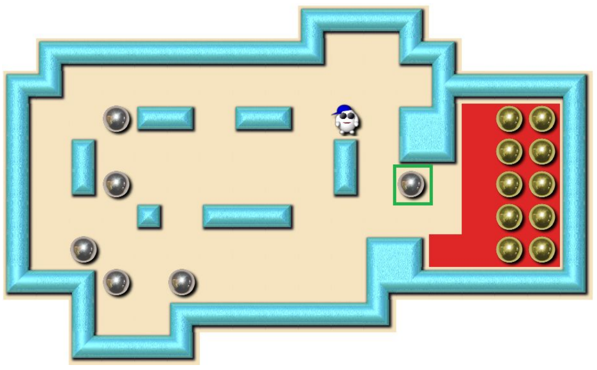
Figure 16b

Figure 16a shows level #11 after two boxes have been pushed to targets. The position is now deadlocked because it is impossible to fill the marked target.

Next, the human players learn that sometimes boxes should not be pushed to targets directly but instead to intermediate parking squares. Figure 16b shows a game position from level #47. The level cannot be solved unless a box is first parked on the marked square.
A useful technique for finding the packing order and the parking squares is to solve the level backwards: Start with the final position and *pull* boxes away from the targets. When all boxes have been removed from the targets, reversing the move order results in a plausible packing plan.
If pulling the boxes in reverse-mode can bring them to their initial places (in normal forward mode), then the level is solved. However, this is not as easy as it sounds. In the initial position, boxes are typically cramped together. Pulling them to such a state requires complex maneuvering, and this is not easier than solving the level directly. Another problem is, that as more and more boxes are pulled away from targets, other parts of the board become crowded, and pulling the next boxes gets harder and harder.
Some Sokoban solvers try a “meet-in-the-middle” approach. A forward search pushes boxes starting from the initial state while a backward search pulls boxes starting from the final state. If these two searches find a common position, then the level is solved. However, the number of possible positions is usually astronomical, and the solver runs out of memory long before the two search frontiers meet.

The sink room assumption

In this section we present a technique for determining the packing and parking order. The technique is best used for *goal-room themed* levels: levels in which all target squares reside in a few rooms. As all 90 XSokoban levels are goal-room themed levels, this technique has proven very useful for solving them.
For this technique to work, a special assumption is needed.
Sink assumption: There is a room near the target area that can hold an infinite number of boxes. This room is called the sink room.

If this assumption holds, then the packing order calculation removes boxes from the board if they can be pulled into the sink room. Since the sink room can hold an infinite number of boxes, we do not need to worry about their actual position in the room.
Then the skeleton plan for solving a level is:

1. Planning: pull boxes from targets to the sink room and remove them from the board.
2. Solving: push all boxes to the sink room and remove them from the board.
3. Once all the boxes have been pushed into the sink room, push them into the target area.

In other words, the sink room is considered a supplier of an infinite number of boxes, and the packing plan is how we use this supply. Obviously, this is just an outline of the solution. In reality, the room is finite so boxes must be inserted and extracted at the same time.

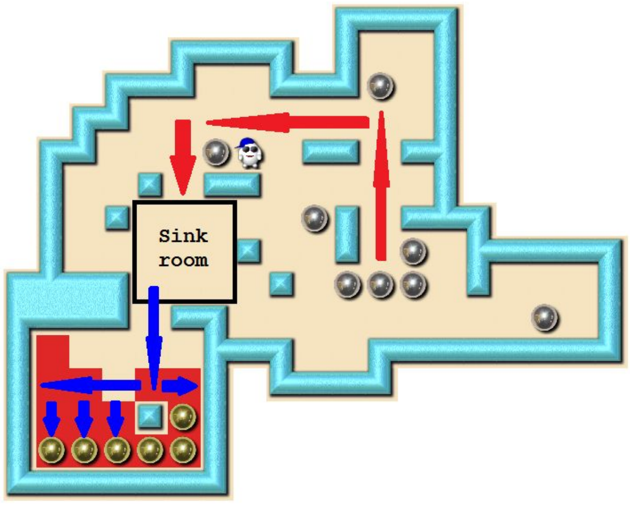

Figure 17

Figure 17 shows the sink room machinery in action for level #11. The red arrows represent pushing boxes into the sink room. The blue arrows represent pushing boxes from the sink room to the targets. In the packing search phase, the blue arrows are reversed, as boxes are pulled into the sink room.

The sink room strategy is in fact a relaxation of the “meet-in-the-middle” approach. We look for the same position when pushing forward and when pulling boxes backward *up to the sink room*. This relaxation makes the problem a whole lot easier.

Let us consider a more complicated case, level #15, shown in Figures 18a-d.

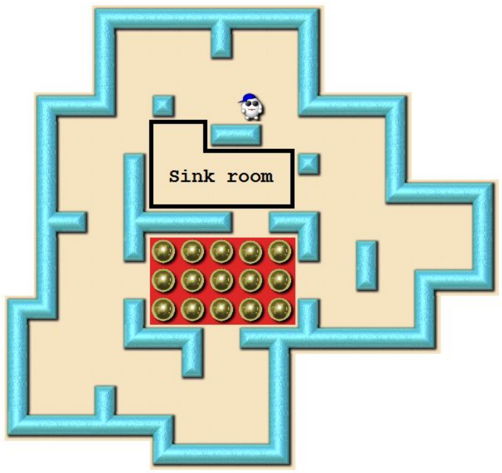

Figure 18a

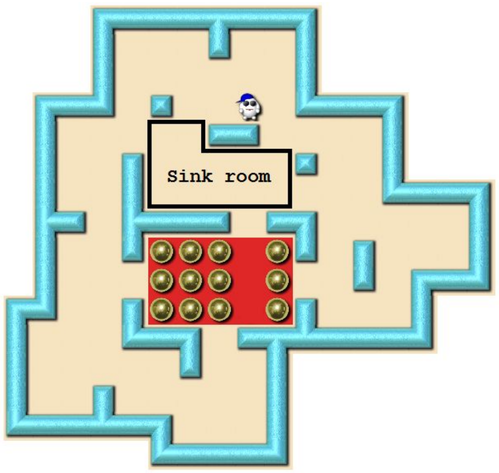

Figure 18b

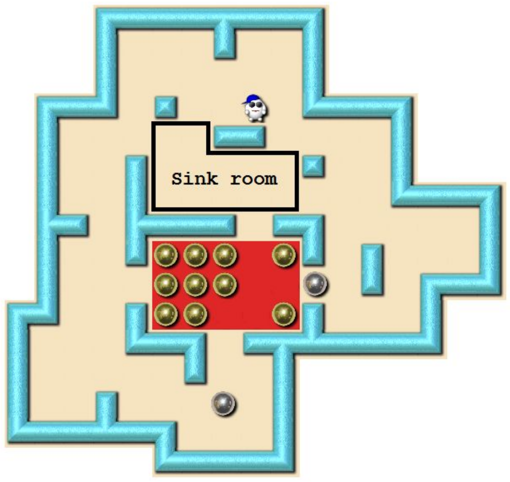

Figure 18c

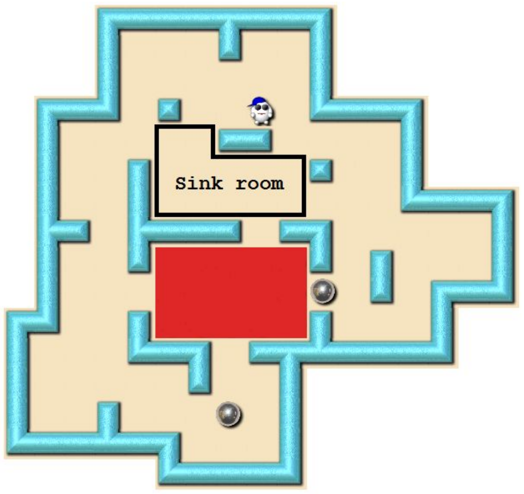

Figure 18d

Initially, three boxes can be eliminated via the sink room, which brings us to Figure 18b. Then no more eliminating moves are possible, so the next step is to pull two boxes as shown in Figure 18c. With the newly opened space, the rest of the boxes in the target area can be pulled into the sink room and removed from the board. This brings us to Figure 18d.
Considering these steps in reverse suggests a packing plan: First park two boxes as shown in Figure 18d and then push boxes to the configuration shown in Figure 18c. Then push the two parked boxes to their respective targets and fill the remaining column. Note that not only have we defined the packing order, but we have also identified two useful parking squares.

A question may arise: if sinks are so useful, why not turn every room into a sink room? The answer is that some rooms are not fit to be sinks; they are too small to pack more than a few boxes, or they do not have enough boxes that can reach them. This leads us to another definition, closely related to sink rooms.
Definition: A *basin* is the set of squares from which a box can reach a given target square, when the board is in the final position.
Figure 19 illustrates four target squares and their corresponding basins.

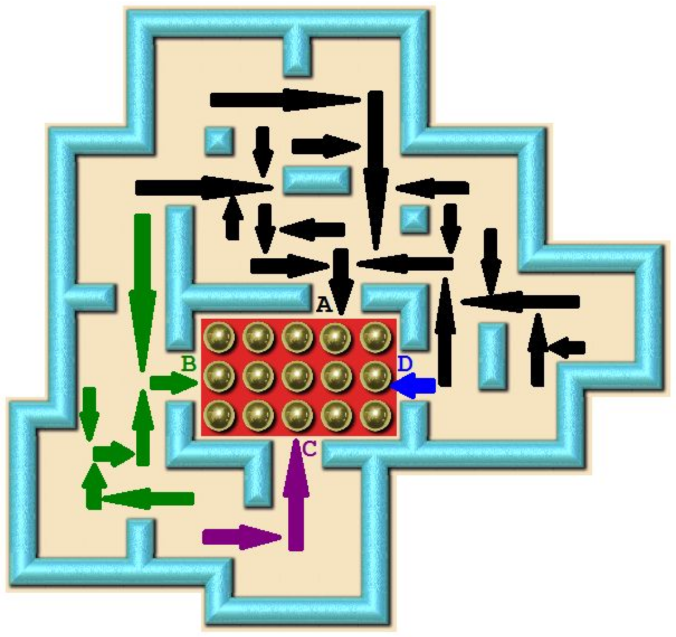

Figure 19

It can be seen that the basins are of different sizes. A’s basin is the largest one and, in fact, it includes B’s basin, but this is not shown in the figure.
Usually, a basin transfers its boxes to a room just before they reach the target squares. This makes rooms associated with large basins good sink room candidates. We choose the sink room to be the room associated with the basin that contains the largest number of boxes in the initial position.
The rationale behind calculating basins is that the final position shows us which paths are going to become permanently blocked. As the solver gets closer and closer to the solved game state, targets fill up. Paths may be closed in this process, but boxes in basins can still reach target squares. The solver exploits this fact by making the simplifying assumption that boxes in the sink room basin will not pose a problem in any of the solution steps.

Out of plan feature

Having defined basins, we can now define a very important feature: *Out of Plan (OOP) boxes*.
Definition: For a given step in the packing plan, the *OOP boxes* are the ones that are not in the basin and not already packed or parked.

OOP boxes are nothing but trouble! Let us see a couple of examples.

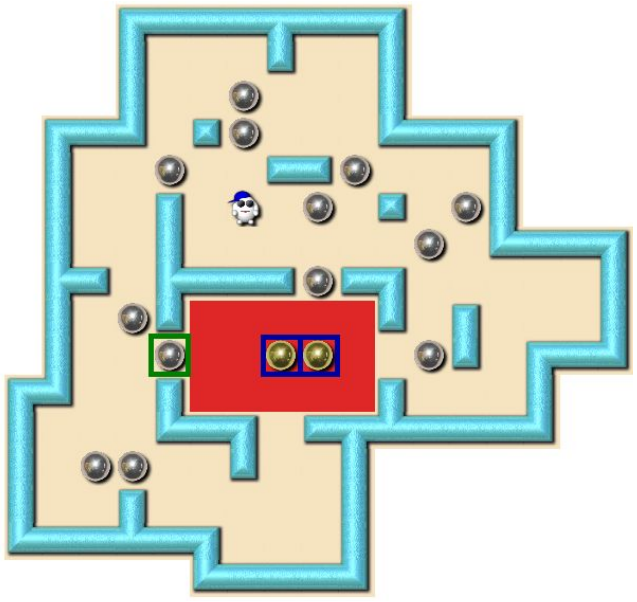
Figure 20

Figure 20 shows an example from level #15. For this level, we choose the sink room from Figure 18, and the basin “A” from Figure 19. These two can provide the largest number of boxes to targets.
Consider the packing phases illustrated in Figures 18a-d. Unpacked boxes are expected to be in the basin from which they will be sent to the sink room. Note, however, the boxes marked in blue in Figure 20. They are neither packed boxes nor parked boxes, and they are not in the basin either. They are OOP boxes, and they interfere with the plan of pushing boxes to the configuration shown in Figure 18d.
There is a similar problem with the box marked in green in Figure 20. On closer inspection, Figure 19 reveals that it is not in the “A” basin (the packed boxes prevent it from being pushed). This box is also an OOP box. If the solver ignores that and just starts packing boxes on the left side of the room, then the marked box will be immobilized and deadlock the level.

Let us examine another example, this time from level #74.

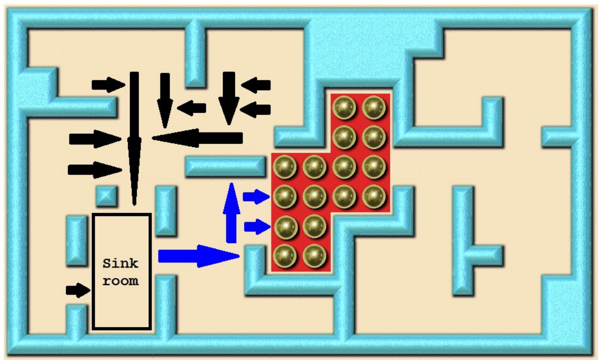

Figure 21a

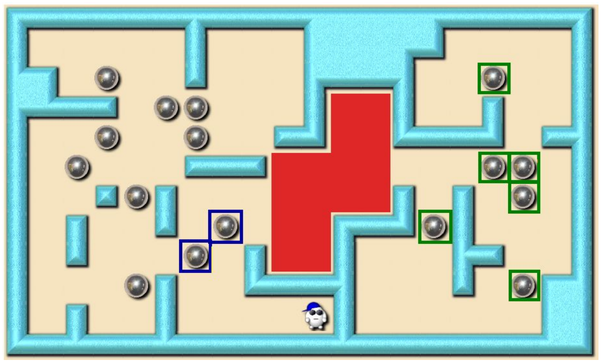
Figure 21b

The packing plan uses the sink room and the basin shown in Figure 21a. However, the initial position shown in Figure 21b contains OOP boxes on the right side, marked in green. If the packing plan ignores them, they will be fenced in behind the packed boxes and cause a deadlock. Note also the boxes marked in blue. They are also OOP boxes and interfere with the packing plan.

OOP boxes represent a problem best dealt with as soon as possible. Typically, this is done by pushing them into the basin area, or by packing an OOP box rather than a box from the sink.

Computing the packing plan

The general strategy for computing the packing order involves two rules:

1. If a box can be pulled to the sink room, remove the box from the board.
2. Otherwise, pull a box as far away from the targets as possible.

Rule 1 is easy to understand. Eliminating a box from the board is always a good move because it cannot obstruct future pulling plans. If such a move is available, the packing plan computation prunes all other moves.
Rule 2 cannot remove a box, but it creates new pulling opportunities. Paths may open and perhaps allow other boxes to be pulled into the sink room. However, a greedy approach does not always work here. Sometimes, pulling a box to the farthest square is not the right move.

In order to implement rule 2, we define a distance-score. First, we compute for each square its distance from its nearest target square. Then the distance-score for a position is simply the sum of these distances for all squares occupied by boxes. Maximizing this sum results in “getting boxes far away from targets”.

Using FESS to find a packing plan

The FESS algorithm is also useful when constructing a packing plan using a backward search. This section describes the features, advisors and score-comparisons in use.

Feature space

We use two features to define the feature space for the backward search. The first feature is “the number of boxes on the board”. As explained in rule 1 above, reducing this number is always desirable. The second feature is “the number of boxes on targets”. As explained in rule 2, this number should be minimized.

Packing plan advisor

The backward search uses only one advisor. It chooses a box on a target square and pulls it as far away as possible. There are, though, some restrictions:

1. Connectivity must not worsen.
2. The parking square must be accessible by pushing, or
3. The parking square holds a box in the initial position.

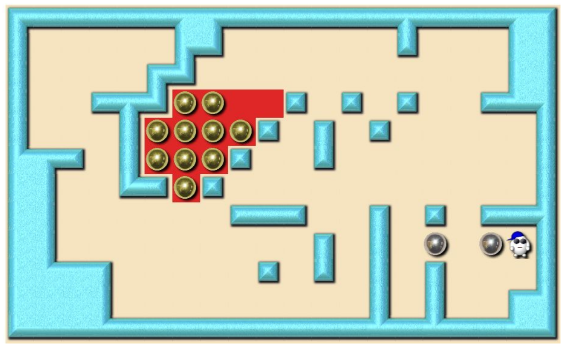
Figure 22a

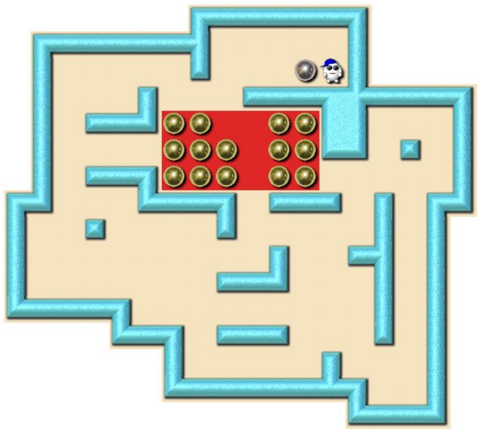
Figure 22b

Figures 22a-b illustrates the restrictions. In Figure 22a (from level #55) it is possible to *pull* two boxes to the shown position. However, during the forward search, it would be impossible to *push* two boxes into such a configuration.
Figure 22b shows a phase from the parking plan for level #14. Here a box has been pulled to the upper room, even though it is impossible to push a box there. This pull is allowed because there is already a box at this square in the initial position.
Advisors do not prune any moves but merely prioritize moves more likely to succeed. Therefore, there is nothing wrong in imposing a few helping restrictions.

Move ordering

FESS needs a way to choose the best move in case of weight equality. The packing search uses the distance-score to prioritize positions with boxes far away from targets.
Comparing distance-scores between positions requires the same number of boxes, but boxes can be removed from the board via the sink. This is, however, not really a problem because positions are grouped into cells in feature space by the feature ”number of boxes on the board”. This means that FESS can make valid distance-score comparisons of moves from the same cell.

Completing the plan

The search runs iteratively until all boxes have been removed from the board via the sink, or until the number of iterations exceeds a predetermined limit. At this point, the program selects the best position in the search tree in terms of “boxes on the board” and “distance to targets”. The path from this node to the root determines the packing and parking plan.

Section 7 - experimental results

This section presents our experimental results. First, we report the running times for solving the XSokoban levels, and then we discuss the parameter choices we made.

Solution times

The following table shows the time required for solving all 90 XSokoban levels.
System architecture: Intel(R) Core(TM) i5-6600K CPU @ 4.5 GHz, 32 GiB RAM

1	0:00	16	0:03	31	0:02	46	0:00	61	0:02	76	0:02
2	0:00	17	0:00	32	0:05	47	0:01	62	0:00	77	0:13
3	0:01	18	0:11	33	0:01	48	0:00	63	0:01	78	0:00
4	0:00	19	0:01	34	0:13	49	0:01	64	0:00	79	0:00
5	0:00	20	0:00	35	0:01	50	0:09	65	0:00	80	0:01
6	0:01	21	0:01	36	0:10	51	0:00	66	0:02	81	0:00
7	0:00	22	0:02	37	0:00	52	0:20	67	0:04	82	0:00
8	0:01	23	0:16	38	0:01	53	0:00	68	0:03	83	0:01
9	0:00	24	0:03	39	0:02	54	0:00	69	0:01	84	0:00
10	0:05	25	0:08	40	0:22	55	0:01	70	0:00	85	0:02
11	0:01	26	0:01	41	0:27	56	0:01	71	0:02	86	0:00
12	0:02	27	0:00	42	0:01	57	0:00	72	0:00	87	0:01
13	0:00	28	0:01	43	0:00	58	0:01	73	0:01	88	0:01
14	0:01	29	0:32	44	0:09	59	0:00	74	0:15	89	0:29
15	0:08	30	0:01	45	0:01	60	0:01	75	0:01	90	0:18
										Total	5:30

Feature ordering

Recall that the FESS algorithm chooses the least weight move or the best move in case of equality. In order to define “best”, we need a feature ordering. Our solver uses a lexicographical ordering of features prioritized this way:

1. OOP: OOP boxes interfere with the packing plan. The first priority is to get rid of them.
2. Packed boxes: They seem more important than connectivity. As an example, consider Figure 15a. Here it is impossible to pack the boxes without worsening the connectivity.
3. Connectivity: Improving it clears the way to new areas of the board.
4. Room connectivity: It is useful to have two ways of accessing a room.
5. Hotspots: They stand in the way and must be dealt with eventually, but usually they are not the immediate problem.
6. Mobility: This is a feature we have not introduced yet. Typically, mobility refers to the number of possible moves in a position. However, such a feature is too expensive to compute, so we approximate it by the number of box sides the player can reach.

Although we currently use this lexicographical order with good results, there may be linear combinations of features which are even better. This is left as a topic for future research.

Setting the weights

In section 5 we said that moves chosen by the advisors have a weight of 1, and that all other moves have a weight of M to be decided later. In this section we present the results of trying different values for M. For simplicity, we call non-advisor moves *difficult* moves.
When we started our experiments with different values of M, it quickly became clear that as M increases, the search tree gets smaller. The best results were obtained by using high values like M=100. We suggest here an explanation for this observation. Consider a solution path in feature space where most cell transitions are found by advisors, but one cell transition requires two difficult moves. Assuming a branching factor of 100, finding the difficult transition requires at least 100^2 nodes. Clearly, all other cell transitions are negligible compared to this value. Therefore, we speculate that the level’s complexity is best estimated by the maximum number of difficult moves required to shift between cells. By setting large values of M, we *indirectly* guide the search algorithm this way:

1. Try to find a solution with no difficult moves (its weight is smaller than M).
2. If this fails, try to find a solution with one difficult move (weight smaller than 2M).
3. If this fails, try to find a solution with two difficult moves, etc.

We made the following experiment to test this hypothesis. The weight of advisor moves was set to zero, and the weight of difficult moves was set to one, so the weight simply counted the difficult moves. We expected the results to be identical to the results with the 1:100 ratio. The results were, however, even better!

In order to understand that, recall that FESS chooses the move with the least weight, and *in case of equality* chooses the best move. Consider a position A where several advisor moves can be applied to tidy up the position without changing the cell: disable hotspots, increase mobility etc. Call the resulting position B. After making the advisor moves, FESS with weights 1:100 will favor position A because it has the lowest weight. For FESS with weights 0:1, positions A and B have the same weight, but position B will be selected for expansion because some of the problems in position A have been dealt with by the advisors. Position B does indeed seem to be the more promising candidate in this case.
Note that using zero-weights breaks the completeness proof in section 3. This can be fixed by noting that the number of possible board positions is finite. Consequently, the moves on the solution path will be selected for expansion in each feature space cell sooner or later.

Choosing features for feature space

The obvious features for measuring search progress are OOP, packed-boxes and connectivity. As an experiment, we added room connectivity as a fourth feature. Some levels were solved faster, but for other levels the extra cells actually increased the solution time. In general, levels with long, complex solutions benefited the most from adding this feature. Based on this observation, the solver does use room connectivity in feature space at the cost of the easier levels being solved somewhat slower. This means that feature space is 4-dimensional in our solver. The features ”hotspots” and “mobility” are not used in feature space but just for comparing scores.

Section 8 - Discussion and future work

We have presented a new search algorithm, FESS, which has proven very effective for solving the XSokoban level set. FESS deals with two problems that make Sokoban difficult: deadlocks and long solutions. In order to deal with the large branching factor, we have introduced advisors: domain specific heuristics that prioritize promising moves. We have developed several concepts for computing a packing plan, such as sinks, basins, and the OOP feature. These concepts have been useful for solving goal-room themed levels. We believe that new features and new advisors can be found for solving other level types as well.

Future research

Automatic generation of features

The features and advisors in this paper were handcrafted. It would have been more impressive if the program had learned them by itself, as DeepMind has demonstrated in the AlphaZero project [16]. However, such an approach is way beyond the scope of this paper.

Refined weights

The weights we have used are very simplistic: A move is either good (chosen by an advisor) or bad (a "difficult move"). Maybe the solver can be further improved by using more refined weights. It is, however, not clear how to pick such weights, as the discussion is section 7 shows.

Prioritizing cells

FESS treats all cells in feature space equally. It sounds reasonable to allocate more computation time to cells on the frontier, as they are more likely to find a solution quickly. However, if the correct plan is left behind in an inner cell, the solution time will actually increase. More research is required on how to prioritize cells better.

We also remind of the problem that FESS has with cells resulting from bad moves, as discussed in "Section 3 - restricting the number of cells". The solution suggested there was to project onto a cell of a good ancestor. A prioritized cell selection could instead solve the problem by assigning low priorities to less promising cells.

Comparison to Q-learning

One may speculate whether FESS is similar to the Q-learning algorithm [17], which also uses feature space. We believe this not to be the case for the following reasons:

- Q-learning is an exploration process in which a strategy is learned. FESS works without such a learning process.
- Q-learning rewards progress by adding a numerical value. FESS rewards progress by allocating computation time.
- In Q-learning, the policy determines the next cell after applying an action (or determines a probability of cells). In FESS, there is no policy that guarantees shifts from one cell to another, or associates such probabilities.
- Q-learning is a single Markov decision process. FESS advances simultaneously from different cells.
- An agent that follows a policy learned by Q-learning might get deadlocked. FESS has an inherent mechanism for avoiding it.

Considering these differences, FESS is an algorithm in its own right and not just a form of Q-learning.

Acknowledgements

We wish to thank Björn Källmark for his permission to use his beautiful “classic” skin. Björn’s “Sokoban for Windows” program was used to produce the diagrams in this paper.

References

[1] Wikipedia - Sokoban entry : https://en.wikipedia.org/wiki/Sokoban
[2] Dorit Dor, Uri Zwick: SOKOBAN and other motion planning problems, Comput. Geom. 1999
[3] Junghanns, Schaeffer: Sokoban: A Challenging Single-Agent Search Problem, IJCAI 1997
[4] Timo Virkkala: Solving Sokoban
http://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf
[5] DeepMind: Agents that imagine and plan
https://deepmind.com/blog/article/agents-imagine-and-plan
[6] Victor Ge: Solving planning problems with deep reinforcement learning and tree search
https://www.ideals.illinois.edu/bitstream/handle/2142/101086/GE-THESIS-2018.pdf
[7] Mattia Crippa: Monte Carlo Tree Seach for Sokoban
https://www.politesi.polimi.it/bitstream/10589/140141/1/2018_04_Marocchi_Crippa.pdf
[8] XSokoban Home Page - Cornell CS : http://www.cs.cornell.edu/andru/xsokoban.html
[9] Sokoban Solver Statistics - Level Set Summary - XSokoban :
http://sokobano.de/wiki/index.php?title=Solver_Statistics_-_XSokoban_-_Thinking_Rabbit_%26_Various_Authors
[10] JSoko - https://www.sokoban-online.de/
[11] YASS solver -
http://sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Brian_Damgaard_about_the_YASS_solver#PI-Corral_pruning
[12] Leme, Pereira, Ritt, Buriol: Solving Sokoban Optimally with Domain-Dependent Move Pruning: https://dl.acm.org/citation.cfm?id=2916392.2916681
[13] Brian Damgaard: Solving Sokoban Optimally with Domain-Dependent Move Pruning - Stolen Ideas, Lies, and Incompetence
http://www.sokobano.de/wiki/index.php?title=Solving_Sokoban_Optimally_with_Domain-Dependent_Move_Pruning_-_Stolen_Ideas,_Lies,_and_Incompetence
[14] Sokoban WIKI - http://sokobano.de/wiki/index.php?title=Solver
[15] Florent Diedler: The Sokolution solver
http://sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Florent_Diedler_about_the_Sokolution_solver
[16] DeepMind: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm https://arxiv.org/abs/1712.01815
[17] Watkins: Learning from Delayed Rewards, Cambridge, 1989
http://www.cs.rhul.ac.uk/~chrisw/thesis.html
[18] How to detect deadlocks :
http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks#Detecting_corral_deadlocks
[19] Andreas Junghanns: Pushing the Limits: New Developments in Single-Agent Search
https://www.researchgate.net/publication/2305703_Pushing_the_Limits_New_Developments_in_Single-Agent_Search

Appendix - Deadlock detection

All Sokoban solver programs need good deadlock detection mechanisms in order to succeed. Therefore, even though not directly related to FESS, this section describes the algorithms we use, so that our results are reproducible.

Deadlock tables

This is a common technique in Sokoban solvers. The solver has a list of patterns. If a pattern matches a subarea of the level, then the level is deadlocked.

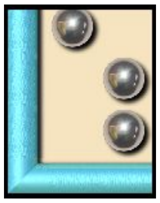

Figure 23

Figure 23 shows an example of such a deadlock pattern. The patterns used by our solver have a maximum size of 4x4 squares. The move generator rejects moves that result in such patterns.

PI-corral pruning

The solver uses the PI-corral pruning algorithm, a very helpful pruning technique invented by Matthias Meger and Brian Damgaard [4,10,11]. It is beyond the scope of this paper to describe the algorithm in detail, but we will provide a rough sketch here. Suppose there is a corral on the board, i.e., an area fenced in by boxes and walls. Suppose also, that the player can only get into the corral by first pushing a box into the corral. In that case, and if certain other conditions are met, then the solver needs only investigate the pushes going into the corral. All other pushes in the position can safely be ignored.

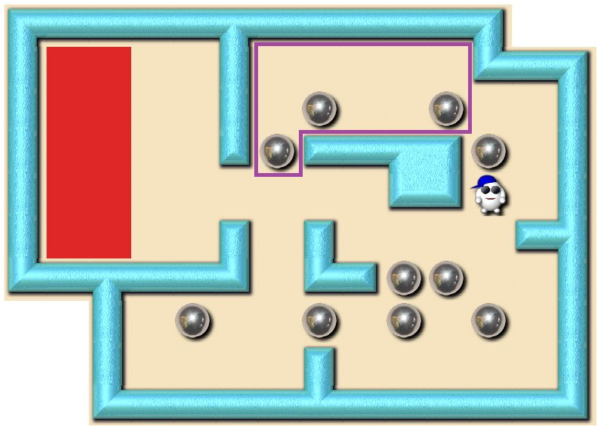
Figure 24

Figure 24 shows an example from level #2. The clever observation is that pushing a box into the marked corral does not interfere with the pushes outside the corral. Thus, the player may just as well push a box into the corral right away. The boxes outside the corral can be pushed later as if nothing has happened.

Implementing PI-corral pruning correctly is tricky because of the complicated pruning conditions. The reader is referred to Timo Virkkala's thesis [4] for a formal definition of the algorithm. Further material and studies of effectiveness can be found in the work by Pereira et al. [12,13] and on the Sokoban wiki [14].

Corral deadlock detection

Many solvers implement some form of corral deadlock detection, so we just sketch it here. Descriptions of the technique can be found on the Sokoban wiki [18]. Like PI-corral pruning, corral deadlock detection deals with corrals, but the objectives are different. When the solver finds a corral, i.e., an area fenced in by boxes and walls, a small search is launched in an attempt to open the corral. Boxes outside the corral are removed from the board. Then the search explores all possible moves until the corral is opened, or until the search runs out of moves, in which case the position is deadlocked.

Bipartite analysis

This technique is also well-known from the Sokoban literature [15,19], so a sketch will suffice here too. It is primarily used for detecting positions with packed boxes causing deadlocks. Consider level #41, for example.

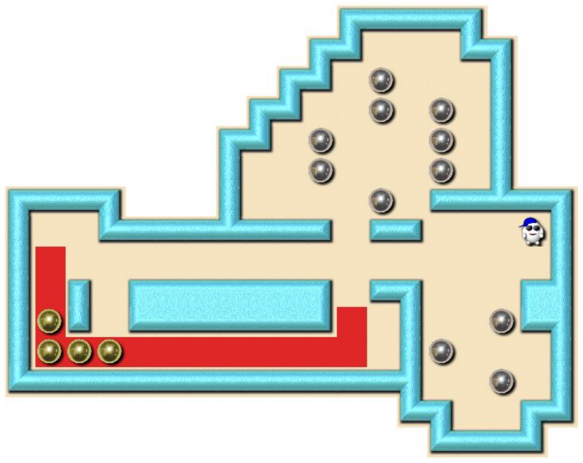
Figure 25a

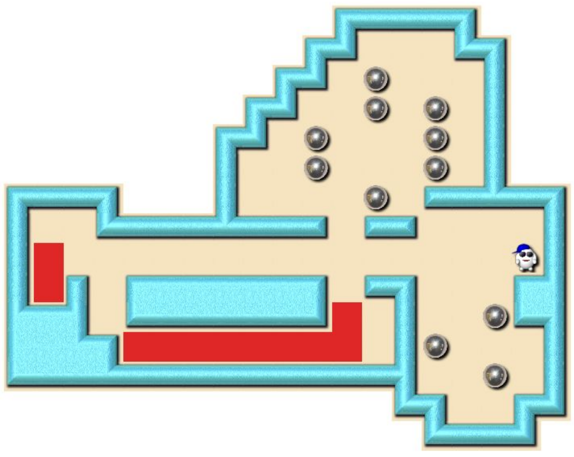
Figure 25b

In Figure 25a, each box can reach a target, and each target can be filled with a box. However, if we turn all immobilized boxes into walls (Figure 25b), we see that the three boxes in the rightmost room compete for two targets. Such deadlocks can be found by computing the bipartite graph, showing which boxes can reach which targets. If this graph does not have a perfect matching, then the position is deadlocked.

Retrograde analysis

Our solver uses a new deadlock detection mechanism which we call retrograde analysis. The idea is to identify boxes packed and immobilized too soon, thereby making it impossible to fill remaining target squares. This is detected by running a backward search from the final position after having turned immobilized boxes into walls. The procedure is best explained by an example.

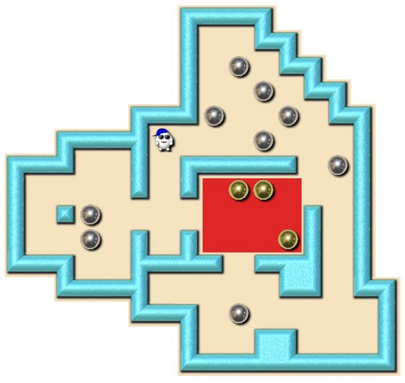
Figure 26a

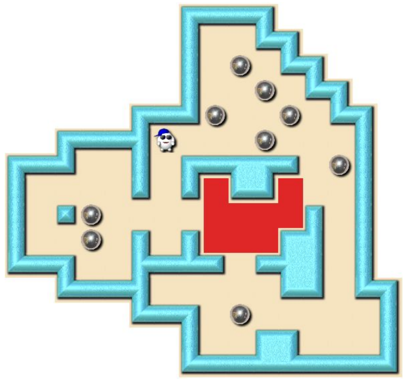
Figure 26b

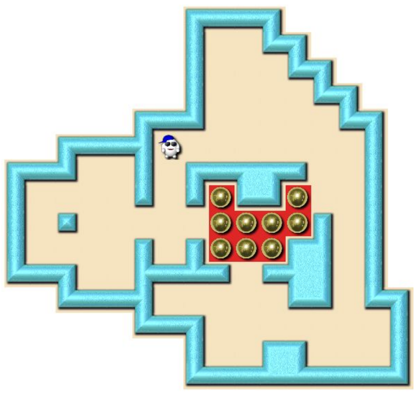
Figure 26c

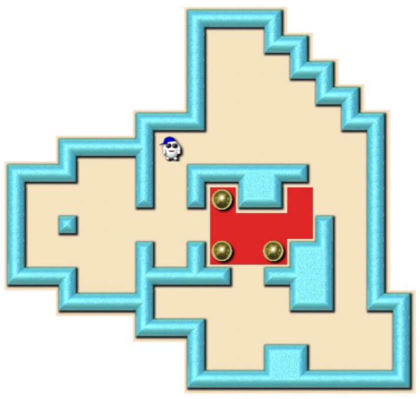
Figure 26d

Figure 26a shows level #49 after some of the boxes have been packed at wrong target squares. These boxes cannot move anymore and might just as well be considered walls (Figure 26b). If the level is solvable, it will reach Figure 26c with boxes at all the remaining targets. Now we start pulling boxes away from the target area. The boxes are removed from the board as soon as they can be pulled away from their target squares. Eventually, we reach Figure 26d. The remaining boxes cannot be pulled anywhere. This means we cannot solve the level, unless these boxes were already there before we started pulling boxes away from the targets squares. But some of these boxes are missing in Figure 26a, hence, there is a contradiction! This proves that the position in Figure 26a is unsolvable.

Retrograde analysis is very fast. Whenever a box can be pulled and removed from the board, the size of the player's access area increases by at least one square. That makes the running time linear in the size of the level.

We note that usually, the bipartite analysis detects most of the retrograde analysis deadlocks too. The retrograde analysis does, however, find a few more deadlocks. Since this check is so inexpensive in terms of computation time, it is almost a pity not to use it.

Room deadlock detection

This is another new deadlock type identified by our solver. Sometimes, the box configuration in a room makes it impossible to push the boxes out of the room. Figures 27a shows an example from level #44. Some of the boxes can be pushed, but they cannot be pushed out of the room. Figure 27b shows another example, this time from level #75. Here the boxes in the room cannot move without creating simple deadlocks. Figure 3 is yet another example.

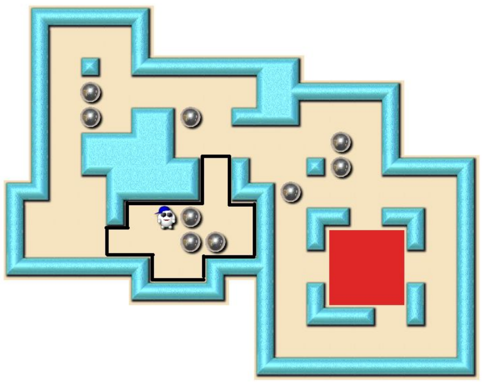
Figure 27a

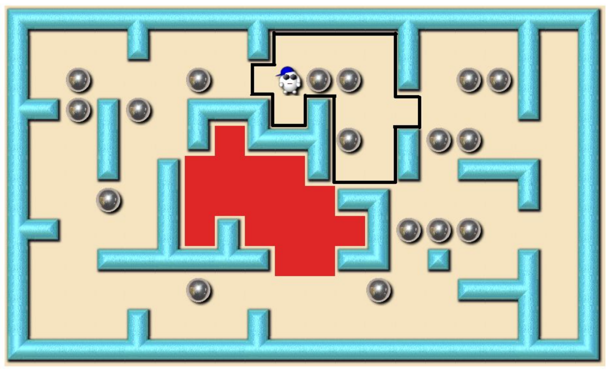
Figure 27b

Room deadlock detection works very similarly to corral deadlock detection. All boxes are removed from the board, except for the boxes in the room and its nearby corridors. Then a small search is launched in an attempt to push a box out of the room. The search explores all possible moves until a box is pushed out of the room, or until all boxes in the room have been pushed to targets. If the search runs out of moves, then the position is deadlocked.