LISA
Layered Interactive Sokoban Assistant

B. Stoeller [0426857], T. Kostelijk [0418889]
{bram.stoeller, mailtjerk}@gmail.com

June 25th, 2010

Abstract

This paper provides a literature research on solving the Japanese game Sokoban. Inspiration is emerged from previous work to develop LISA (Layered Interactive Sokoban Solver). LISA assists the player towards solving the puzzle. AI techniques are used to display strategic information about the game is visual assistance layers.

1 Introduction

In many games the computer has become stronger than the human, like chess. However there are games where the human performs significantly better, one of them is Sokoban. Sokoban is a Japanese single player game where the player has to push stones to certain goal positions. The player can only push one stone at a time and has to rearrange stones in a clever way to make a clear path.

Let's compare a chess game with solving a complex level of Sokoban. The Chess player has on average each turn a choice between 35 different moves. The Sokoban player has more then 100 different possibilities. The chess game takes about 50 moves to finish. Where some optimal solutions in Sokoban require 600 pushes [1]. The state-space in Chess is 10^{40} where a complex Sokoban level has 10^{98} states. This makes Sokoban intractable to solve with current AI techniques.

Most of the research done in AI tries to solve problems, which is very useful. There is not much attention for the case where the human and the computer collaborate in order to solve a problem. In this report we took this approach where the computer assists the player towards a solution.

The outline is as follows. First we describe what previous work is done on solving Sokoban. Then we describe how we approached the problem. After that we give a detailed description of our implementation. Section 5 presents the results. Finally a discussion and conclusion are drawn and several ideas are proposed for further research.

2 Previous work

In previous work solving the Sokoban domain is addressed by mainly three different approaches, Single agent, Multi-agent, and State abstraction.

2.1 Single-agent

The single agent approach is the most popular branch of research and lots of interesting work is done. Junghanns et al. [2] [3] developed a popular Sokoban solver called *Rolling Stone*. They use transposition tables, deadlock detection and macro moves to reduce the search tree of Sokoban. Furthermore they use the minimal solution length for the depth of their iterative deepening A* algorithm. Despite of the advanced AI techniques they used, they left a standard test suite for 1/3 unsolved.

2.2 Multi-agent

Van Lishout [4] took a multi-agent approach. They presented the stones as individual agents which have to collaborate. The man can be called by the stones when they want to be pushed. They used conditions like: at least one stone should directly reachable and have a free goal path. The only collaboration between the agents they used is that the stone closest to the goal has precedence. This seems to be insufficient. Although the approach seemed to be very innovative the results where poor, they could only solve one problem of the standard test suite.

2.3 State abstraction

A nice way to handle the complexity of the problem is state abstraction. In [5] they use an abstract representation which decomposes the puzzle into two types of areas: rooms and tunnels. The maze is reduced to a graph of rooms linked by tunnels. In this way the problem is split up into a local component (moves within a room) and a global component (moves between rooms and tunnels). Planning is first done on a global level and then on a local level.

This seems to be a nice approach. We started implemented this method with STRIPS and Prolog but aborted soon. In the abstract state they store if a room has at least one stone that can be pushed into a specific tunnel. We figured this abstraction is not deterministic and explain this with an example.

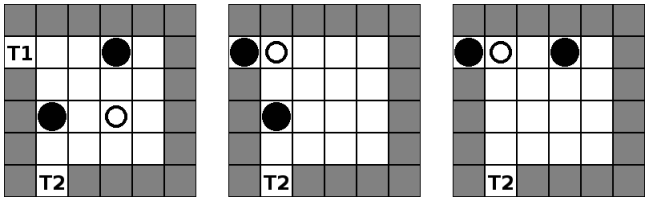

Figure 1

Suppose we have a room with two tunnels T1 and T2 as shown in figure 1. The room has two stones which both can be pushed by the man (displayed by the small circle) into T1. Only the left stone can be pushed into T2. If a stone is pushed to T1 the abstract state does not concern which stone this is. The right stone that could reach both tunnels or the left stone that could only reach T2. After this abstract move it is unsure if a stone can be pushed in T2. This makes the abstract state planner non deterministic. A solution for this would require a set of strong assumptions. This hypothesis supports the fact that they only solve 10 problems of the standard test suite.

3 Approach

As described before it is infeasible to create a full solver for Sokoban using current AI techniques. This section explains how we approached the problem of assisting a player towards a solution. We developed LISA, this stands for:

- Layered (Different layers of assistance are present)
- Interactive (The player can toggle the layers and the system reacts on pushes)
- Sokoban (The game)
- Assistant (It assists the player towards a solution)

First we created a tile game engine and implemented Sokoban in this framework (see Appendix A). After that, the Sokoban implementation is extended to support visual assistance layers.

3.1 Visual assistance layers

Several visual assistance layers are created that can visually assist the player while solving a Sokoban puzzle. These layers can be toggled on or off by pressing a key. In this way the degree of assistance can be adjusted by the player. The layers are:

- The *room-tunnel layer* highlights what squares belong to a tunnel and which belong to a room.
- The *dead squares layer* highlights all positions in the level from where a stone can never reach a goal area.
- The *possible pushes layer* highlights all stones that can be pushed directly.
- The *deadlock prevention layer* highlights pushes that result in a deadlock.
- The *deadlock alert layer* highlights all stones and walls which are involved in a deadlock at the moment the deadlock is created.

4 Implementation

Before we present the results of LISA we provide a description of the Implementation. We implemented LISA in Python and Pygame. For readability purpose we limit the implementation details and describe a large part on a conceptual level.

4.1 Room tunnel layer

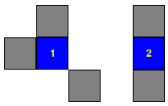

Figure 2

To solve Sokoban levels it is important to know how rooms and tunnels are divided. It keeps the players attention on more important calculations if we highlight this. The room tunnel division is explained in figure 2. If one of these patterns (or a mirrored instance) holds, the position is classified as a tunnel, if not the position is classified as a room.

4.2 Possible pushes

To determine which stones can be pushed by the man, a path planning method is needed. Using a path planner like A* to find all possible paths from the man to all positions next to all stones would be to expensive to perform in real-time. Furthermore it would also be overkill because the exact path is not needed. We implemented a breadth-first search algorithm instead. It starts from the position of the man spreading out in every direction. This is done until the entire reachable area is covered. Every time a stone is found, the position of the stone and the direction from which it is reached is stored in a list.

Every impossible push is removed from this list of reachable stones to create a list of possible pushes. A push is impossible when a stone is pushed against another stone or a wall. Executing such a push would result in the same state as before the push and is therefore not considered to be possible. In many cases such a possible push would lead to an insolvable state. This is discussed in the next section.

4.3 Deadlock detection

"In Sokoban, moves can lead to states that provably cannot lead to solutions" [2]. These states are called deadlocks. One of the first skills a player develops is to prevent these deadlocks. It takes a lot of effort detecting these deadlocks which turns the players attention away from the bigger picture. Visualising these deadlocks would help the player a lot.

There are three types of deadlocks: trivial, local and global. Trivial deadlocks are for example moving a stone into a corner (the man can never get behind the stone). A local deadlock considers interaction with other local stones (more details on this is section 5). A global deadlock considers complex interaction with stones located over a large portion of the field. The detection of global deadlocks involves a full solution. Because this is intractable with current research we only consider trivial and local deadlocks in this project.

If there are no deadlocks in the grid, pushing a stone can introduce one. Therefore we would have to apply the deadlock detection only on this last pushed stone. However, for the purpose of our interactive solver, the search is not only applied to this last pushed stone, but also to all possible pushes. Each possible push is executed by the engine, deadlocks are tested against this stone and then the push is undone, remembering if it would lead to a deadlock or not. Finally the pushes that lead to a deadlock are marked in the *deadlock prevention layer*.

4.3.1 Trivial deadlocks

A stone pushed into a corner is deadlocked, we call this corner-square a dead square. If two corners are connected, which means they are supported by a straight wall, the path between the corners are also classified as dead squares.

In our implementation a square is classified as a corner-square if two diagonal adjacent neighbouring squares appear to be a wall. For each corner-square the program walks to possible corner-squares in the horizontal and vertical direction. If every square on the path is free and has a wall on his square to the bottom/top (for horizontal direction) or left/right (for vertical direction) the squares are classified as dead squares. For details on this see figure 3 and section 5.

Now all dead squares are collected, the deadlocks are detected if a stone is pushed to a dead square. Next we explain the local deadlocks.

4.3.2 Create deadlock patterns

We first investigated and developed the most common deadlock patterns. The patterns are stored in a text file which is read to load the patterns one by one. For every pattern we apply the *duplicate*, *mirror* and *unique* operation. A pattern is always seen from the stone the *man* can push. Therefore we duplicate every pattern and centre each duplication on a different stone. The duplication is done n times where n is the number of stones present in the pattern.

The next module duplicates the patterns for all possible mirror configurations. Three mirror types are used (horizontal, vertical, diagonal) which can be applied or not, causing $2^3 = 8$ different mirror configurations. In this way all possible transpositions are covered. The combination of the duplication and mirror module can generate equivalent configurations , those are found and discarded.

4.3.3 Create deadlock table

A deadlock table is a table which for every configuration stores a pattern list of patterns that are consistent with this configuration. The configuration is a combination of a position coordinate and a tile. A tile can have values *wall*, *stone* or *free*.

The deadlock table is filled as follows. For every pattern we retrieve all positions and their concerned tiles and we update the deadlock table on the entry with this position and tile, by adding the current pattern id to the list of consistent patterns.

4.3.4 Evaluate deadlocks

The next step is to evaluate the grid and find all deadlocks. As stated before, pushing a stone can introduce a deadlock. This last pushed stone must be part of the deadlock pattern to be responsible for creating it. Therefore our search algorithm is defined for one stone only. When determining possible pushes that lead to a deadlock, this algorithm is repeated for each stone.

The squares near the centre are more probable to be part of the deadlock pattern than the squares further away. Therefore each deadlock pattern is centred on the last pushed stone. From this centre stone an extensive pattern search is initialised. Starting at one of the positions closest to the centre and expanding the search area until the search area is larger than the largest deadlock pattern. This expansion of the search space is implemented by iterating through a list of relative coordinates sorted by the distance from each coordinate to the centre. In the end a list of deadlocks that match is returned. See algorithm 1.

Algorithm 1 DetectDeadlock(D, T, p)

Require: A deadlock table D, a grid of squares T and a stone position p.
Ensure: Which deadlocks occur in T at p?

```
1:  Q ⇐ getDeadlockIndexes(D)
2:  C ⇐ getRelativeCoordinates(D)
3:  for c ∈ C do
4:     i ⇐ p + c
5:     if isFree(T_i) then
6:        Q ⇐ Q − D[(c, wall)] ∪ D[(c, stone)]
7:     else if hasStone(T_i) then
8:        Q ⇐ Q − D[(c, wall)]
9:     else if isWall(T_i) then
10:       (do nothing, all patterns are consistent)
11:    end if
12: end for
13: return Q
```

It is possible that no deadlock pattern is consistent anymore before the search through all relative coordinates is completed. This would mean the search could terminate earlier. It is also possible that more than one deadlock matches the grid, but we are concerned about if there is any deadlock instead of the exact subset of deadlocks.

For efficiency purposes our Python implementation checks every iteration if Q is already empty (i.e. none of the deadlocks are consistent anymore) and it checks if any of the deadlock patterns is completely covered by the grid. In the latter case that particular pattern is returned and the remaining patterns are discarded.

5 Results

We now show our results by presenting some deadlocks and the assistance layers.

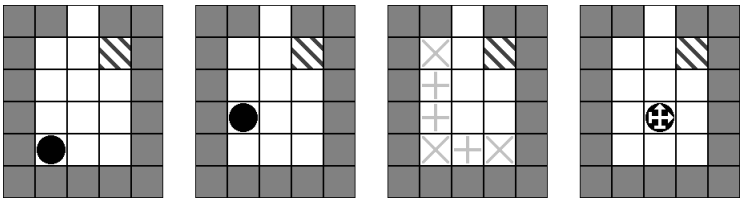

Figure 3

On the left of figure 3 we see the first deadlock situation. If a stone is pushed into a corner the man cannot get behind the stone. To the next you see the second deadlock situation. If a stone is pushed to a wall between corners, the man cannot get behind the stone. Both deadlock situations are of course not true when a goal square is in this corner or next to the wall.

We highlighted the positions of these trivial deadlocks as dead squares. The third image on the figure displays the dead square layer. The most right image displays the possible pushes- and deadlock prevention layer. Pushes that are possible are marked with an arrow. Pushes that result in a deadlock are marked with a blocked sign.

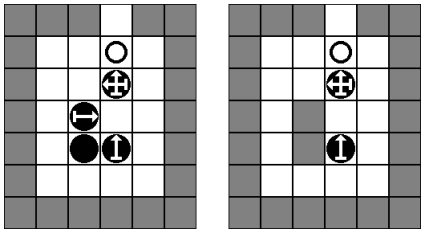

Figure 4

On the left of figure 4 we see another deadlock pattern. If the upper stone is pushed down the stones are in such a way aligned that they are stuck. On the right we have a similar deadlock. Notice that if in a pattern one or more stones are substituted by walls, the pattern always remains deadlocked.

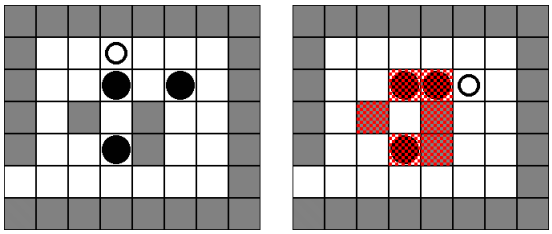

Figure 5

Obviously pushing the stone down in the left image of figure 5 gives the same deadlock situation as in figure 4. Pushing the upper right stone to the left results in a more advanced deadlock. Notice that a deadlock does not necessary mean that stones cannot move anymore. In this example some stones can still move but it will always result in another deadlock or in a situation where provably no solution can be found.

A clear example of the room-tunnel layer can be found in figure . The rooms and tunnels are divided as the implementation section.

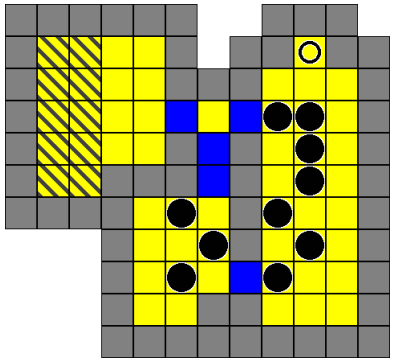

Figure 6

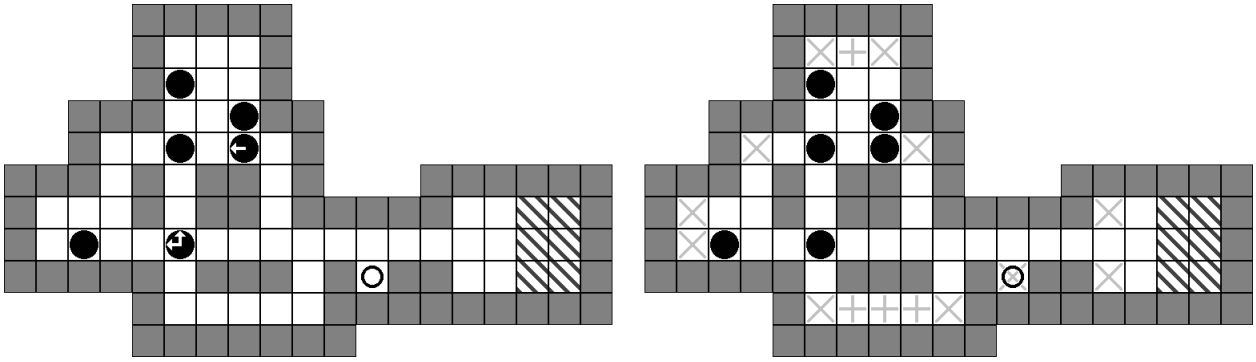

Figure 7

Figure 7 displays level 1 of the standard test suite. The left image shows the possible pushes and deadlock prevention layer, only two stones are reachable. Notice that there are three vertical tunnels. The dead square layer in the right image shows that it is impossible to push a stone through the left or right tunnel, the stones are forced to go through the middle tunnel. This saves a lot of computations.

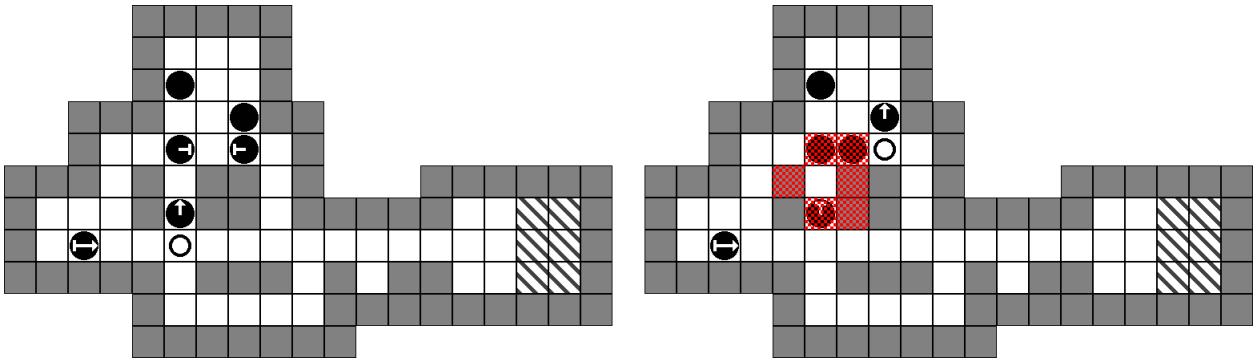

Figure 8

On the left of figure 8 a stone is pushed up and the deadlock prevention layers shows some interesting results. The middle tunnel becomes unreachable because the two stones above have a block sign. If the left blocked stone is pushed to the right it is easy to see that it causes a local 4-stone deadlock. The right figure displays what happens when the blocked stone to the right is pushed to the left, the more advanced deadlock of example 5 occurs.

We leave it to the reader what the correct solution for this level is.

Finally figure 9 shows our assistance layers on a more complex level (level 35 of the standard test suite).

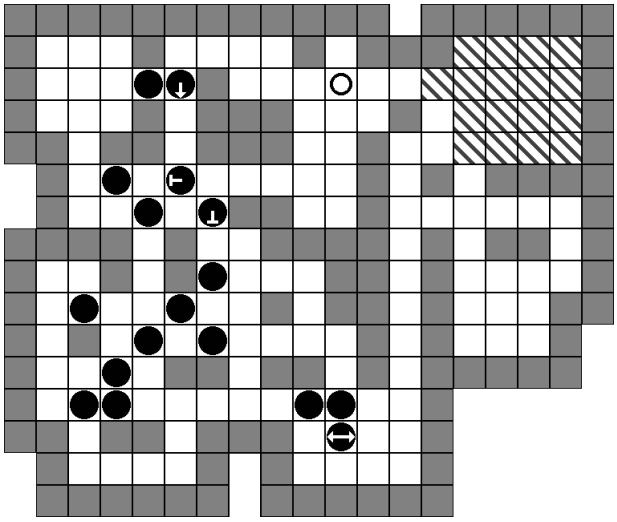

Figure 9

6 Discussion

6.1 Complexity

We explained how we mirror and duplicate patterns for deadlocks. Although this grows linear in the number of unique patterns, it provides the system a large number of patterns to evaluate. A detailed complexity analysis is out of the scope of this report, what can be stated is that this number of patterns could lead to insufficient resources. This is not tested empirically. We covered a large part of possible deadlocks and our system worked in real-time.

6.2 Deadlock development

The patterns we used for deadlock detection where designed by us. Because we are human and tend to make mistakes, it is tricky to develop these patterns. The larger the patterns are the more difficult it is to evaluate them. It would be nice to evaluate the patterns by computer.

As said before a stone in a deadlock pattern can always be replaced by a wall. This makes a stone in a deadlock more specific then a wall. It is important to maximise the number of stones when developing deadlock patterns. If one forgets this, some deadlock patterns would not be detected. Another comment on developing deadlocks is deadlock transpositions. Especially for patterns involving a large number of stones and walls, the mirror configurations do not cover all possibilities. Sometimes a few walls can be placed at a different position within the same pattern. It is import to cover all possibilities and to think twice when adding an advanced deadlock.

6.2.1 Uncaptured situations

When detecting a deadlock, the position of the man is not taken into account. In some levels the man is spawned with stones close around him. The system alerts for a deadlock but this is not the case because the man is inside the deadlock and can push the stones away from the inside. Furthermore global deadlocks are not defined, more on this in section 8.

7 Conclusion

We demonstrated that current research shows interesting results but even the latest AI techniques are insufficient to solve the entire Sokoban standard test suite. We took another approach where we assist the player towards the solution. Inspiration is done on previous work and own ideas are developed to create helpful visual layers. We developed LISA which gives the player ability to focus on the bigger picture of solving a level. It warns the player about deadlocks and makes playing Sokoban more fun.

LISA will be soon available on www.pygame.org

8 Further research

8.1 Reachable stone areas

A nice feature would be a layer that presents the possible positions a stone could take without moving other stones. This assists the player in thinking ahead. The possible positions could be highlighted when hovering over this stone with a mouse, or for (one of) the closest stone to the man.

8.2 Free goal path highlighting

A stone is said to have a free goal path if it can be pushed to the goal without moving other stones. In most of the cases this is proven to be the best move. It would be nice if this path would be highlighted in a new layer at the moment it becomes available. Note that it is not always the best solution. In advanced levels the order in which the stones are pushed into a goal room is restricted. Although it is not always the best solution, highlighting it would inform the player but leaves the decision to him.

8.3 Parking spot detection

Often a stone blocks an important route, a common task is to search for spots in the maze where you can temporarily park this stone. These parking spots are spots in the maze that do not block important routes. A common parking spot is the room parking spot, where a room can only keep one stone. The parking spot detection would require the extraction of important routes in the maze which is a interesting challenge.

8.4 Information overkill

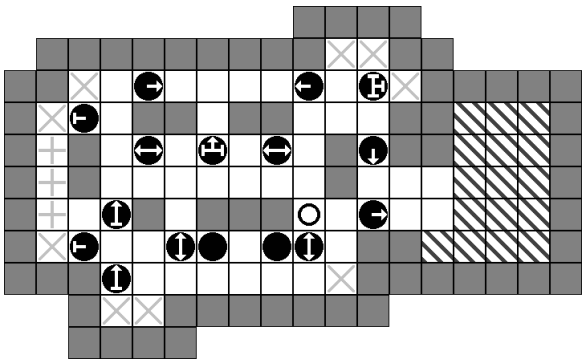

Figure 10

Our assistance layers are developed in levels with a low number of stones. During testing we discovered that this gives the player an information overkill in levels where the number of (reachable) stones is large, for example level 47 of the standard test suite (figure 10). It would be nice to do some research on filtering this information.

8.5 Global deadlocks

We showed an approach for detecting trivial and local deadlocks. A nice extension to our Sokoban assistant would be to detect global deadlocks. These can only be detected if a full solution of the game is provided for a specific state. The player should only use this layer when it is necessary because it reveals a large part of (if not the entire) the solution.

References

[1] J. Demaret, F. Van Lishout, and P. Gribomont, "Hierarchical planning and learning for automatic solving of sokoban problems," 2008.

[2] A. Junghanns and J. Schaeffer, "Sokoban: A challenging single-agent search problem," 1997, pp. 27–36.

[3] A. Junghanns, "Pushing the limits: New developments in single-agent search," 1999.

[4] F. Van Lishout and P. Gribomont, "Single-player games: Introduction to a new solving method combining state-space modelling with a multi-agent representation," in *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence (BNAICâ2006)*. Citeseer, 2006, pp. 331–337.

[5] A. Botea, M. Müller, and J. Schaeffer, "Using abstraction for planning in sokoban," 2002, pp. 360–375.

Appendices

A Game engine

We developed a game engine using Python[1] and Pygame[2]. This engine is designed to implement every arbitrary tile game. It has three main classes `Game`, `Field` and `Tile`. I refer to the instance of a class using the term *object* (e.g. a `Game` object). A `Game` objects contains one or more `Field` objects which in turn contains any arbitrary number of `Tile` objects. The `Game` and `Tile` class have to be extended to create a playable game and the `Field` class may be extended to implement extra functionalities, but this is not necessary.

A.1 Python & Pygame

Pygame is a cross-platform library designed to make it easy to write multimedia software, such as games, in Python. We used it to implement our game engine because of its simplicity and its completeness.

A.2 Tiles

The Tile class is extended for every possible tile for a game. All properties of a tile are defined in the `config` method (see figure 11). The first property is `canBeAccessedBy` which contains a list of `Tile` extensions (i.e. tile types) that can access the tile (e.g. a `TileAgent`). It is possible to add `Tile` as the only element of this list to capture all tiles at once. The `canBeMovedBy` property defines which tile types can push the tile when they try to access it without having the privilege to do so (i.e. they are pushing against it). The `canMoveInDirection` property defines in which direction(s) the tile can move (provided that a privileged tile pushes it). When a tile is at the edge of a field and some other tile tries to push it off the edge, the `canFallOffEdge` property defines if the tile falls of the edge (i.e. is being removed) or just stays where it is. Finally the `src` property defines which image represents the concerned tile. Notice that all tile types should have an image of equal size. This size should be 32×32 pixels. In any other case the dimensions of the images should be communicated to the field on which the tiles are placed (`Field.tileSize = (width, height)`).

```
class TileBox(Tile):
    def config(self, args = None):
        self.canBeAccessedBy = []
        self.canBeMovedBy = [TileAgent]
        self.canMoveInDirection = ALL
        self.canFallOffEdge = False
        self.src = 'img/box.png'
```

Figure 11: Definition of the tile type `TileBox` which extends the `Tile` class.

As can be seen in figure 11 it is possible to supply some arguments for the `config` method. These arguments are passed from the constructor function. They can be used to define some small differences in tiles. For example an `TileSlider` is an object that can be pushed in some specific directions. A `TileSlider` may need some direction arguments which defines in which direction the slider can slide. In this way it is possible to create two different sliders using the same class (e.g. `TileSlider(LEFT | RIGHT)` and `TileSlider(UP | DOWN)`).

A.3 Field

The `Field` class is the core of our engine. Its three main methods are `addTileType(char, tile)`, `addTile(pos, char)` and `move(pos, dir)`. The first one is used to add a new tile type. Each `tile` type is identified by a `char`acter. An instance of each tile object is stored in a dictionary, indexed by its character. The second method adds a tile to a certain `pos`ition on the field. The character identifying the tile is stored in a dictionary indexed by the position which is a tuple `(x, y)`. Tiles can be stacked (e.g. first add a `TileFloor` and then a `TileAgent` to the same position). The third and most interesting method `move` will be explained at the end of this section.

[1] Python: www.python.org
[2] Pygame: www.pygame.org

Furthermore the `Field` class contains some useful methods like `getNeighbour(pos, dir)` to return the neighbour position and `nextTo(pos, char, dir)` to check whether a position is next to some tile. The `dir`(ection) variable is defined by a combination of some constants `UP = 1`, `RIGHT = 2`, `DOWN = 4`, `LEFT = 8`. In this way it is possible to apply some fast bitwise combinations and comparisons (e.g. `ALL = (UP | RIGHT | DOWN | LEFT) = 15`)

A generalisation of the `nextTo` method is `checkNbh(pos, char, pattern, dirs)`, where the tile at the given `pos`ition is tested against the given `char`acter and where `pattern` is a list of eight characters which are tested against the eight squares around the tile at the given `pos`ition. Starting at the position defined by the `dirs` variable. `UP` means it starts at the position directly above the centre tile and walking around it in clockwise order. The value of this variable is set to `ALL` by default, meaning that all four rotations of the pattern are tested. Notice that mirror configurations are not included and should be specified by another `pattern`.

Algorithm 2 MOVE(p, d)

Require: A position p and a direction d.
Ensure: Move the top most tile at p to d.

```
1: A ⇐ getTileType(p)
2: if d not in A.canMoveInDirection then
3:    return False
4: end if
5: p' ⇐ getNeighbourPosition(p, d)
6: if not onField(p') then
7:    if A.canFallOffEdge then
8:       removeTile(p)
9:       return True
10:   else
11:      return False
12:   end if
13: end if
14: B ⇐ getTileType(p')
15: if A in B.canBePushedBy(A) then
16:   canMove ⇐ move(p', d)
17:   if canMove then
18:      removeTile(p)
19:      addTile(p', A)
20:   end if
21:   return canMove
22: else
23:   return False
24: end if
25: return True
```

The method `move(pos, dir)` tries to move the top most tile at the given `pos`ition to the given `dir`ection. Lets call this tile A. First the direction is to be checked with the `canMoveInDirection` variable of A. If it can not move in this direction, the move fails and everything stays in place. To move A its destination position has to be checked for accessibility. If there is no tile at this destination position it is always accessible, but if there is a tile B, its `canBeAccessedBy` variable has to be checked for consistency with tile A. If B is not accessible for A, the only solution for A to move is to move B first. Again this can only happen if A is in the `canBePushedBy` variable of B. If it is not in there the move fails. But if A can move B the `move` method is called again, now for B (moving in the same direction as A).

A.4 Game

The `Game` class is extended to create a specific game like Sokoban or Cyberbox. In the constructor of the `Game` class a new `Field` object is initialised and the different tile types are added to this field.

Furthermore the `Game` class contains a set of events which are triggered at certain moments during

```
class Sokoban(Game):
    def __init__(self):
        self.field = Field()
        self.field.tileSize = (40, 40)

        self.field.addTileType('.', TileFloor())
        self.field.addTileType('O', TileMan())
        self.field.addTileType('*', TileGoal())
        self.field.addTileType('#', TileObject())
        self.field.addTileType('X', TileWall())

        self.background = (0, 0, 0)
        self.agentChar = 'O'
        self.goalChar = '*'
        self.goalReachingChar = '#'

    [...]
```

Figure 12: Constructor of the game `Sokoban` which extends the `Game` class.

the game. In this way it is fairly easy to create game specific behaviour like counting the number of moves execute some algorithm after the goal is reached. (Meaning all tiles of type `Game.goalChar` are occupied by at least one tile of type `Game.goalReachingChar`, or any other implementation of the `Game.goalReached()` function.)

- `onStart()` When the game is started (not initialised).
- `onStop()` When the game is stopped.
- `onQuit()` Just before the program quits.
- `onGoal()` When the goal is reached.
- `onNoAgent()` If no agent can be found on the filed (maybe he jumped of the edge of the field).
- `onMove(direction, moved)` The number of tiles that moved in this direction after the `move` method is called (moved > 0).
- `onPush(direction, moved)` The number of tiles that are pushed in this direction by the agent after the `move` method is called (pushed = moved - 1 > 0).
- `onNoMove(direction)` If no tiles moved after the `move` method is called (i.e. the move failed).
- `onUndoMove(moved)` The number of tiles that moved while undoing a move (moved > 0).
- `onUndoPush(moved)` The number of tiles excluding the agent that moved while undoing a move (pushed = moved - 1 > 0).
- `onNoUndoMove()` The player tried to undo a move, but nothing can be undone (anymore).
- `onReload()` After the game is reloaded.