02285 AI and MAS, SP20
Warmup Assignment
Due: Monday 17 February at 20.00

Exercises by: Martin Holm Jensen, Thomas Bolander and Mathias Kaas-Olsen

This assignment is to be carried out in **groups of 2–3 students**. Your solution is to be handed in via DTU Inside: Go to Assignments, then choose Warmup Assignment. You should hand in two *separate* files:

1. A **pdf file** containing your answers to the questions in the assignment.
2. A **zip file** containing the relevant source code files and level files (the ones you have modified or added, and *only* those).

Introduction

The purpose of this assignment is to recap some of the basic techniques used in search-based AI, and bring all students up to a sufficient and comparable level in AI search basics. You are already supposed to be familiar with these techniques in advance, as these are covered by the prerequisite courses. However, some of you might not be familiar with all of the techniques, and others might simply need a brush-up. If you are less familiar with the relevant notions, or have become a bit rusty in using them, please read Chapter 3 of Russell & Norvig.

For the assignment, we provide you with an incomplete search client in Java or Python, which you will finish implementing. To obtain the implementation, download the archive `searchclient.zip`. The Java client we provide is in the directory `searchclient_java`, and the Python client is similarly in the `searchclient_python` directory. The client is an implementation of the Graph-Search algorithm in Figure 3.7 of Russell & Norvig, where the code uses the same names as the book with the exception that the node and state representations are merged into a single data structure. The directory `levels` contain example levels, some of which are referenced later in this assignment. You are welcome to design additional levels to experiment with, and for testing your search client. The archive also contains a `readme-searchclient.txt` file for each of the search clients, found in their respective directories, which describes how to run the server using that search client.

The exercises described later are written with references to the Java search client, but if you are working with the Python version then all the concepts and names should translate more or less directly. We expect you to have a development environment set up to work in the command-line, though you are also free to use an IDE if you configure it yourselves.

The following sections describe the setting for the search domain and introduce the environment server that the search client will communicate with.

Figure 1: KIVA robots at Amazon.

Figure 2: The TUG robot tugging a container.

Domain Background

The search domain is inspired by the developments in mobile robots for hospital use and systems of warehouse robots like the KIVA robots at Amazon, see Figure 1. In both applications, there is a high number of transportation tasks to be carried out.

Among the most successful and widely used implementation of hospital robots so far are the TUG robots by the company Aethon, see Figure 2. TUG robots were first employed in a hospital in 2004, and is now in use in more than 100 hospitals in the US. In 2012–2013, TUG robots were also tested and used at a Danish hospital, Sygehus Sønderjylland, the first hospital in Europe to employ them.

The goal of this assignment is to implement a search-based AI for a transportation robot at a hospital or warehouse in a simulated environment.

Levels

The environment will be represented by grid-based structures, called *levels*. A level contains *walls*, *boxes*, *goal cells* and an *agent*. The walls are used to represent the physical layout of the environment. The agent represents the robot. The boxes represent the items that the robot has to move. Each item has to be moved to one of the corresponding goal cells.

A level can be represented textually, making it easy to design levels using any text editor (using a fixed-width font) and saving them as ASCII text files. Levels are constructed according to the following conventions:

- *Walls.* Wall items are represented by the symbol `+`.
- *Agent.* The agent is represented by the number `0`.
- *Boxes.* Boxes are represented by capital letters `A`, `B`, . . . , `Z`. The letter is used to denote the *type* of the box, e.g. one could use the letter `B` for hospital beds. There can be several boxes of the same type, that is, having the same letter.

```
++++++++++++++
+           0+a+
+  +++++++++A+
+             +
+  ++++++++   +
+B+b          +
++++++++++++++
```

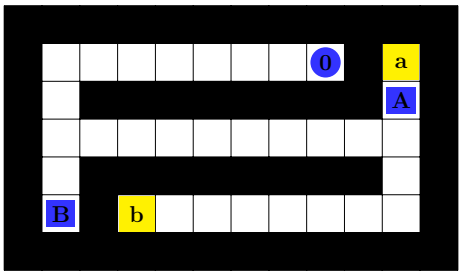


Figure 3: The textual representation of a level (left), and its graphical visualisation (right).

- *Goals cells.* Goal cells are represented by small letters `a`, `b`, ..., `z`. Again, the letter is used to denote the *type* of the goal cell. Each goal cells must have a box of corresponding type moved on top of the goal cell (e.g. an `a` goal with an `A` box).

A level is *solved* when all goal cells have boxes of corresponding types moved on top.

Figure 3 shows a full textual description of a simple level (left) accompanied by its graphical visualisation (right). We can think of this level as having two subgoals: 1) get `A` into its goal cell, and 2) get `B` into its goal cell. Agent `0` can solve one subgoal at a time, that is, it can e.g. first choose to move to the far right and push `A` into its goal cell, and then afterwards go back and pick up `B`. Or it can first choose to take `B` into its goal cell and then afterwards take care of `A`. But in fact, in this level, the optimal solution can not be achieved by solving the subgoals independently. Here the optimal solution is first to pick up box `B`, move it to the rightmost column, push box `A` into place, and finally move box `B` into place. This means that in the optimal solution we have to interleave the solutions to the two subgoals.

Files containing textual representations of levels are given the extension `.lvl`. The archive `searchclient.zip` contains a few such files with example levels. You are free to experiment with levels of your own construction if you want. Levels are not allowed to contain tab characters, and the last line of a level should be followed by a newline. Single-agent level names generally begin with `SA` and multi-agent levels with `MA`, e.g. `SAsimple1.lvl` and `MAsimple1.lvl`. In this assignment, only single-agent levels will be considered.

Actions

A grid cell in a level is called *occupied* if it contains either a wall, an agent or a box. A cell is called *free* if it is not occupied. The agent can perform the following actions:

1. *Move action.* A move action is represented on the form

   $$Move(\textit{move-dir-agent}),$$

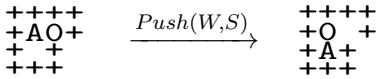
   where *move-dir-agent* is one of N (north), W (west), S (south), or E (east). $Move(N)$ means to move one cell to the north of the current location. For a move action to be successful, the following must be the case:

   - The neighboring cell in direction *move-dir-agent* is currently free.

2. *Push action.* A push action is represented on the form

   $$Push(\textit{move-dir-agent}, \textit{move-dir-box}).$$

Here *move-dir-agent* is the direction that the agent moves in, as above. The second parameter, *move-dir-box*, is the direction that the box is pushed in. The following example illustrates a push:



Here the agent, `0`, moves west and the box, `A`, moves south. The box is "pushed around the corner." For a push action to be successful, the following must be the case:

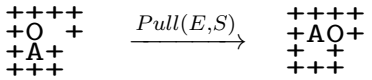
- The neighbouring cell of the agent in direction *move-dir-agent* contains a box β.
- The neighbouring cell of β in direction *move-dir-box* is currently free.

The result of a successful push will be that β moves one cell in direction *move-dir-box*, and that the agent moves to the previous location of β. Note that the second condition above ensures that it is not possible for an agent and a box to swap positions by simply performing an action like e.g. $Push(W, E)$.

3. *Pull action.* A pull action is represented on the form

$$Pull(\textit{move-dir-agent}, \textit{curr-dir-box}).$$

The first parameter, *move-dir-agent*, is as above. The second parameter, *curr-dir-box*, denotes the current direction of the box to be pulled (as seen from the position of the agent). The following example illustrates a pull, reversing the push shown above:



For a pull action to be successful, the following must be the case:

- The neighbouring cell of the agent in direction *move-dir-agent* is currently free.
- The neighbouring cell of the agent in direction *curr-dir-box* contains a box β of the same color as the agent.

The result of a successful pull will be that the agent moves one cell in direction *move-dir-agent*, and that β moves to the previous location of the agent. Note that the first condition above ensures that it is not possible for an agent and a box to swap positions by simply performing an action like e.g. $Pull(S, S)$.

4. *No-op action.* The action $NoOp$ represents the persistence action (do nothing). The No-op action is always successful.

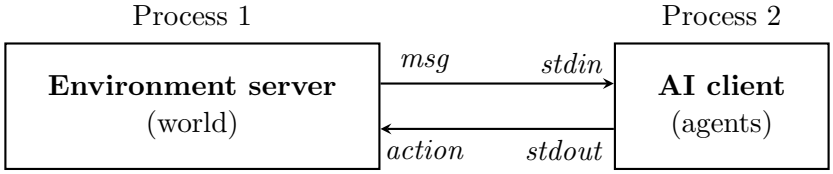
If an agent tries to execute an action that does not satisfy the conditions for being successful, the action will fail. Failure corresponds to performing a no-op action, that is, doing nothing. So if e.g. an agent tries to move into an occupied cell, it will simply stay in the same cell.

```
<SERVER>                    <CLIENT>
1 +++++                     1 -
2 +0Aa+                     2 -
3 +++++                     3 -
4                           4 -
5 -                         5 [Move(E)]
6 [false]                   6 -
7 -                         7 [Push(E,E)]
8 [true]                    8 -
```

Figure 4: Example of interaction between server and client.

Environment server

To simulate the environment, we provide an environment server. It is in the archive `searchclient.zip`. The server represents the actual state of the world, and the search client interacts with the environment by communicating with the server. Communication happens through the *standard input*, *standard output*, and *standard error* streams (`System.in`, `System.out`, and `System.err` in Java). The search client program can thus be implemented in any programming language. The interaction between server and client is depicted below:



The protocol for communicating with the server is specified by the following steps:

1. The server sends a description of the level to the client, following the level format conventions of Section . The level is sent one line at a time, terminated by an empty line.
2. The client sends a string to the server $[a]$, where a is the action. A valid action string could e.g. be `[Push(S,W)]`.
3. The server replies with $[p]$, where each p is either `true` or `false` according to whether the action has been successful or not.
4. If the level is not solved, go to step 2.

Figure 4 illustrates an initial segment of an example interaction between a server and client. The left and right columns show what the server and client sends, respectively. The symbol '`-`' denotes that the process is waiting for input.

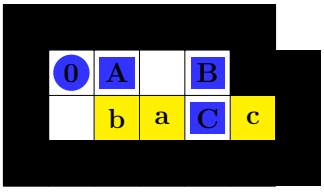
Once a sequence of actions has led to all goal cells being occupied by boxes of the correct types, the server will write `success` to its *stdout* stream (presumably the shell). If a timeout is set and this value is exceeded, the server will instead write `timeout` and terminate the client. Violation of the communication protocol or the occurrence of unrecoverable errors will similarly terminate the server and provide a brief error message. As *standard in* and *standard out* of the

client process are used for communication with the server, the search client can use the *standard error* stream for writing messages to the terminal - this may be useful in debugging a search client.

Example. Consider the following level:

```
+++++++
+0A B++
+ baCc+
+++++++
```

The environment server will present this level graphically as follows:



As seen, the agent is represented by a circle and boxes by squares. Walls are black and goal cells are yellow. Figure 5 shows a sequence of actions in this level and the corresponding sequence of states. Goal cells turn green when they become occupied by boxes of the correct type.

Benchmarking

Throughout the following exercises you are asked to benchmark and report the performance of the client. For this you should use the values printed after "Found solution of length xxx". In cases where your client runs out of memory or your search takes more than 3 minutes (with the "-t 180" server argument), use the latest values that have been printed (put "-" for solution length). You should allocate as much memory as possible to your client in order to be able to solve as many levels as possible – normally allocating half of your RAM is reasonable. The `readme-searchclient.txt` file in the archive explains how to adjust the memory settings.

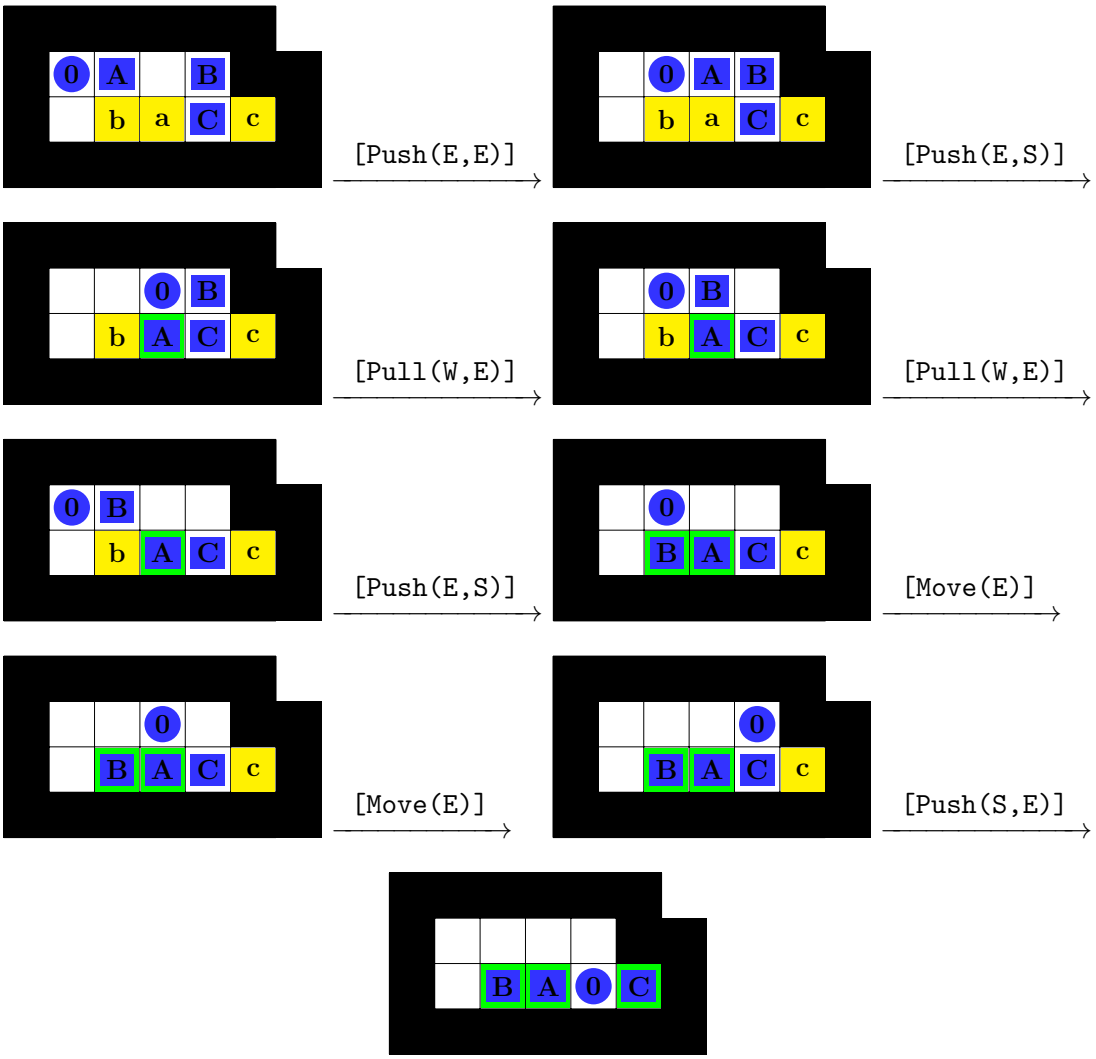


Figure 5: A sequence of actions and the corresponding states

Exercise 1 (Search Strategies)

In this exercise we revisit the two evergreens: Breadth-First Search (BFS) and Depth-First Search (DFS). Your benchmarks must be reported in a format like that of Table 1. To complete this exercise you only need to modify the `Strategy` and `State` files.

1. The client contains an implementation of breadth-first search via the `StrategyBFS` class. Run the BFS client on the `SAD1.lvl` level and report your benchmarks. *For this question, you only have to fill in the relevant lines of Table 1.*

2. Run the BFS client on `SAD2.lvl` and report your benchmarks. Explain which factor makes `SAD2.lvl` much harder to solve using BFS than `SAD1.lvl`. (You can also try to experiment with levels of intermediate complexity between `SAD1.lvl` and `SAD2.lvl`). *Give a brief, but conceptually precise, answer (using the relevant theoretical concepts from the book).*

3. Modify the implementation so that it supports depth-first search (DFS). Specifically, implement the class `StrategyDFS` (which extends `Strategy`) such that when an instance is passed to `SearchClient.search()` it behaves as a depth-first search. Benchmark your DFS client on `SAD1.lvl` and `SAD2.lvl` and report the results. *For this question, you should fill in the relevant lines of Table 1 as well as very briefly explain how your implementation of DFS differs from the implementation of BFS.*

4. On the levels `SAD1.lvl` and `SAD2.lvl`, DFS is much more efficient than BFS. But this is not always the case. Run BFS and DFS on the level `SAfriendofBFS.lvl` and report how many nodes are generated by each of the two algorithms. Why is there such a huge difference between the number of generated nodes? *For this question, you need to fill in the relevant lines of Table 1 as well as give a brief, but conceptually precise, explanation of why there is such a big difference in the performance of the two algorithms on this level. Whenever relevant, use the notions from the course curriculum to assist you in making the answer as clear and technically precise as possible.*

5. Answer the same question as the previous one, but this time for the level `SAfriendofDFS.lvl`.

Level	Strategy	Time	Memory Used	Solution length	Nodes Generated
`SAD1`	BFS				
`SAD1`	DFS				
`SAD2`	BFS				
`SAD2`	DFS				
`friendofDFS`	BFS				
`friendofDFS`	DFS				
`friendofBFS`	BFS				
`friendofBFS`	DFS				
`SAFirefly`	BFS				
`SAFirefly`	DFS				
`SACrunch`	BFS				
`SACrunch`	DFS				

Table 1: Benchmarks table for Exercises 1 and 2.

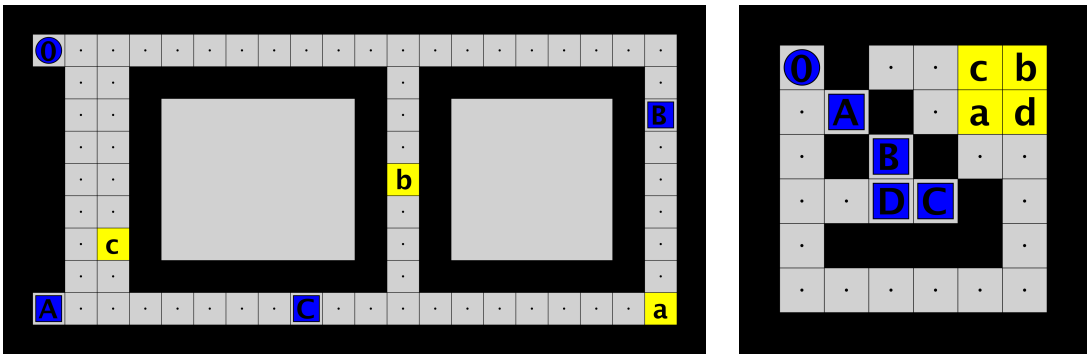


Figure 6: The (relatively simple) levels `SAFirefly.lvl` (left) and `SACrunch.lvl` (right).

6. Benchmark the performance of both BFS and DFS on the two levels `SAFirefly.lvl` and `SACrunch.lvl`, shown in Figure 6. *For this question, you only have to fill in the relevant lines of Table 1.*

Exercise 2 (Optimisations)

While the choice of search strategy can provide huge benefits on certain levels, code optimisation gives you across the board performance improvements and should not be neglected. Such optimisations include reduced memory footprint of states and the use of more clever data structures. In this exercise, we consider two simple such optimisations.

The `State` class has two immediate flaws which result in an excess use of memory: 1) The location of walls and goals are static (i.e. are the same for all states), yet each `State` contains its own copy of the arrays, and 2) `MAX_ROW` and `MAX_COLUMN` are set to 70 regardless of the actual size of a level. Rectify these two flaws and report your new benchmarks in a format like that of Table 1. Briefly comment on how significant the improvements you achieve are in terms of time and memory consumption. To complete this exercise you will need to modify the `State` and `SearchClient` classes. *Your answer to this question should include: 1) the new benchmarks (for the same levels as before); 2) a few words on how significant the improvements are; 3) a few sentences explaining how you modified the code.*

Exercise 3 (State spaces and solution lengths)

1. Consider a level of the form in Figure 7. Let n denote the width of the level excluding walls. The level `SAsoko1_16.lvl` has a total width of 16 excluding walls, so $n = 16$ for this level. What is the length of a shortest solution in terms of n, given in Θ-notation? What is the size of the state space (number of reachable states from the initial state) in Θ-notation? *Report your answers as well as the calculations/reasoning leading to the answers.*

2. Consider a quadratic level of the form in Figure 8. Let n denote the width and breadth of the level excluding walls (so $n = 16$ in Figure 8). Answer the same questions as you did for Figure 7. *Report your answers as well as the calculations/reasoning leading to the answers.*

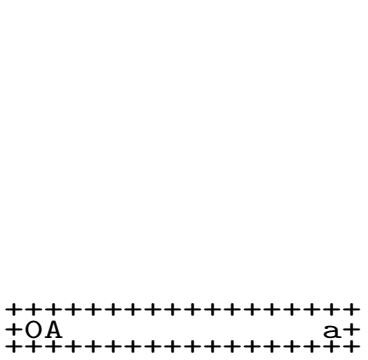


Figure 7: `SAsoko1_16.lvl`.

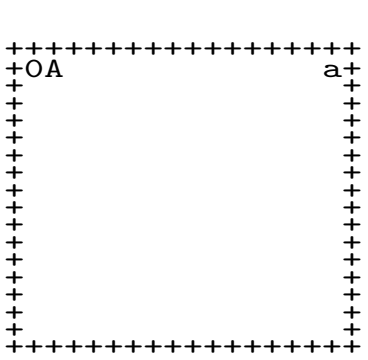


Figure 8: `SAsoko2_16.lvl`.

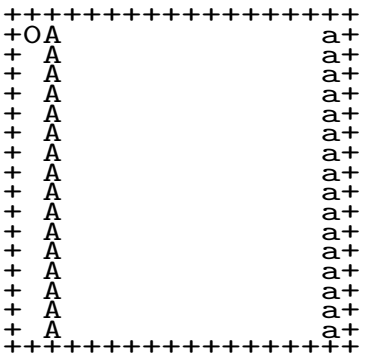


Figure 9: `SAsoko3_16.lvl`.

3. Consider a quadratic level of the form in Figure 9. Let n denote the width and breadth of the level excluding walls. Answer the same questions as for Figure 7 and 8.[1] Is the size of the state space polynomial in n, that is, is it in $O(n^k)$ for some constant k? *Report your answers as well as the calculations/reasoning leading to the answers.*

4. Run your (optimised) BFS client on the different sizes of the `SAsoko1`, `SAsoko2` and `SAsoko3` levels provided in `searchclient.zip`. Briefly relate your findings to your answers to the previous 3 questions. *For this question, you should include benchmarks of all the provided sizes of `SAsoko1`, `SAsoko2` and `SAsoko3` in the style of Table 1. Briefly analyse how the growth in solution length and number of generated nodes observed in your benchmarks relate to the theoretical complexity results from the previous questions.*

This exercise shows that even relatively simple problems in AI are completely infeasible to solve with naive, uninformed search methods. Next in this assignment we will look at informed search methods, where the search has a sense of direction and closeness to the goal, which makes the search much more efficient.

Exercise 4 (Heuristics)

Uninformed search strategies can only take you so far. Your next task is to implement an informed search strategy and then provide it with some proper information via the heuristic function. Background reading for this exercise is Section 3.5 in Russell & Norvig (for those who need it). In particular, when referring to $f(n)$, $g(n)$ and $h(n)$ below, they are used in the same way as in Russell & Norvig (and almost all other texts on heuristic search). To complete this exercise you will need to modify the `Strategy` and `Heuristic` files.

1. Write a best-first search client by implementing the `StrategyBestFirst` class. The `Heuristic` argument in the constructor must be used to order states. As it implements the `Comparator<State>` interface it integrates well with the Java Collections library. Make sure you use an appropriate data structure for the frontier.

   `AStar` and `Greedy` are implementations of the abstract `Heuristic` class, implementing distinct evaluation functions `f(n)`. As the names suggest, they implement $A^\star$ and *greedy best-first search*, respectively. Currently, the crucial *heuristic function* `h(n)` in the `Heuristic`

[1] *Hint*: You are allowed to use the binomial coefficient in your Θ-expression.

Level	Strategy	Time	Memory Used	Solution length	Nodes Generated
SAD1	A*				
SAD1	Greedy				
SAD2	A*				
SAD2	Greedy				
friendofDFS	A*				
friendofDFS	Greedy				
friendofBFS	A*				
friendofBFS	Greedy				
SAFirefly	A*				
SAFirefly	Greedy				
SACrunch	A*				
SACrunch	Greedy				
SAsoko1_64.lvl	Greedy				
SAsoko2_64.lvl	Greedy				
SAsoko3_64.lvl	Greedy				

Table 2: Benchmarks table for Exercise 4, using the best-first search client.

class throws a `NotImplementedException`. Implement a heuristic function (or several ones). Remember that `h(n)` should estimate the length of a solution from the state `n` to a goal state, while still being cheap to calculate. You may find it useful to do some preprocessing of the initial state in the `Heuristic` constructor.

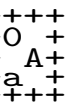
When designing your heuristics, it might be worthwhile to do some benchmarks to check how well it performs. You can e.g. use this to compare different choices of heuristics, and see what works best. You can do benchmarks on the levels you have previously benchmarked BFS and DFS on (the next question asks you to do this systematically). Best-first search using a good heuristics should give a significant improvement over BFS and DFS in most cases. But note that even a very good heuristics can give bad results on some types of levels, and a bad heuristics can give good results on some types of levels. A good heuristics should give improvements on as many different level types as possible (improvements in how long it takes to find a solution). Greedy best-first search normally gives the greatest improvements in computation time, and doing benchmarks with greedy best-first search often gives valuable insights into your heuristics, its strengths and weaknesses. You are of course also free to experiment with other level types than the ones used for the earlier benchmarks (either some of the levels provided in `searchclient.zip` or levels of your own design). *For this question, you should in the report include: 1) your choice of data structure for the frontier, 2) a detailed and mathematically precise specification of the heuristics you have implemented, and 3) a description of the reasoning and intuition behind your heuristics, including how it estimates a length of a solution, and how general and precise it is.*

2. Benchmark the performance of best-first search with your heuristics by filling in Table 2. Analyse the (potential) improvements over uninformed search by comparing the new benchmarks with your earlier benchmarks. *For this question, you need to: 1) fill in Table 2; 2) analyse the improvements provided by $A^\star$ and greedy best-first search over BFS and DFS; 3) if you are still unable to solve `SAsoko3_64.lvl` with greedy best-first search,*

sketch an idea for how to improve the heuristics so that you expect it to become solvable.

Exercise 5 ($A^\star$ search by hand)

Consider the following level:



Illustrate how $A^\star$ with your heuristics solves this level. *You should: 1) draw the graph generated by $A^\star$; 2) put numbers on the nodes of the graph according to the order in which they are generated using the names $n_0, n_1, n_2, \ldots$; 3) add the f, g and h values to each node of the graph.*