Programming Project Assignment

AI @ DIS, FA19

Part I due: 4 Oct 2019 at 20.00

Part II due: 15 Nov 2019 at 20.00

Thomas Bolander, Mikkel Birkegaard Andersen, Andreas Garnæs,
Martin Holm Jensen, Mathias Kaas-Olsen

Group sizes and group work. This assignment is to be carried out in **groups of 2-3 students**. All exercises are intended to be solved as group work, not by splitting the exercises between you. Some are straightforward, but others are more challenging and will require discussions and brainstorming in the group.

Registering your group. It is important that you register your group on Canvas. Please do this immediately. Go to *People* on Canvas, and click *Groups*. Then click *+Group*, choose a (cool) name for you group and choose the group members. By default, this will be your group in all the mandatory group work throughout the course.

Handing in. For both parts of the programming project, you need to hand in two separate files via Canvas (handing in as a group):

1. A **pdf file** containing your answers to the questions in the assignment (no Word files or other formats). If you scan handwritten material, make sure it is scanned in sufficiently high quality and reasonably scaled when included, so it is easily readable when printed.
2. A **zip file** containing the relevant Java (or Python) source files and level files (the ones you have modified or added, and *only* those).

The *front page* of your pdf file should contain:

- The name of the group (as it is registered on Canvas).
- The full names of all group members (as they appear on Canvas).
- A *group declaration* briefly specifying who did what in terms of ideas, programming, answering the theoretical questions, report writing, etc.

Make sure to only use **private repositories** for sharing work in your group.

Figure 1: KIVA robots at Amazon.

Figure 2: The TUG robot tugging a container.

1 Project background

The project is partly 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. Since 2012, TUG robots have also been applied at a Danish hospital, Sygehus Sønderjylland, the first hospital in Europe to employ them.

The goal of this programming project is to implement a simplified simulation of transportation robots at a hospital or in a warehouse.

2 Levels

The environment will be represented by grid-based structures, called *levels*. A level contains *walls*, *boxes*, *goal cells*, and *agents*. The walls are used to represent the physical layout of the environment. The agents represents the robots. The boxes represent the items that the robots have to move. The goal cells represent destinations for items of specific types; each goal cell must have a corresponding box moved on top of it to solve a level.

A level is represented textually, making it easy to design levels using any decent text editor (with a monospaced font) and saving the levels in ASCII-encoded text files. Each level file has the following format:

```
#domain
hospital
#levelname
<name>
```

```
#colors
<colors>
#initial
<initial>
#goal
<goal>
#end
```

The items in angle brackets (e.g. `<name>`) are placeholders for content described below, and all lines are terminated by either a line-feed character (LF) or a carriage-return followed by a line-feed (CRLF).

The first two lines indicate the domain of the problem, which for this project is the hospital domain, and occur verbatim in level files.

The `<name>` field is replaced by the level's name. By convention, single-agent levels begins with `SA`, and multi-agent levels with `MA`.

Levels are constructed with initial and goal states using the following conventions:

- *Free cells* are represented by spaces (ASCII value 32).
- *Walls* are represented by the symbol `+`.
- *Agents* are represented by the numbers `0,1,...,9`, with each number identifying a unique agent, so there can be at most 10 agents present in any given level. The first agent should always be named `0`, the second `1`, etc. Hence single-agent levels, those whose names starts with `SA`, only contain agent `0`.
- *Boxes* are represented by capital letters $\texttt{A}, \texttt{B}, \ldots, \texttt{Z}$. 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 (i.e. same letter) in a level.

The `<initial>` and `<goal>` specifications consist of lines with these symbols as a top-down map, which defines the initial state and goal states of the planning problem respectively. The two specifications must have exactly matching configurations of walls, and should consist of at most $2^{15} - 1$ rows each of at most $2^{15} - 1$ columns. It is required that the agent and all boxes in a level are in areas entirely enclosed by walls. Each symbol in the initial state specifies that a corresponding object starts in that position, and each symbol in the goal state specifies that for the level to be solved, an object of the symbol's type must occupy that cell. The goal states of the planning problem are then all states which have objects in the configuration shown in `<goal>`, where excess objects can be anywhere (i.e. the agent and/or some boxes do not necessarily have to have a goal cell they must reach to solve a level).

To allow modelling of different agents having different abilities concerning which boxes they can and can't move, agents and boxes are given colors. An agent can only move a box that has the same color as itself. If e.g. we use boxes of type B to represent beds, and if these are red, then only the red agents can move beds.

The *allowed colors* for the agent and boxes are:

blue, red, cyan, purple, green, orange, pink, grey, lightblue, brown.

To represent the colors of the agent and boxes as part of the textual level representation, each level has the colors section (`#colors`). The color declaration `<colors>` is of the form

```
#domain
hospital
#levelname
MAExample
#colors
red: 0, A
green: 1, B
#initial
++++++++++++
+        0+ +
+ ++++++++A+ +
+            +
+B++++++++ + +
+ +1         +
++++++++++++
#goal
++++++++++++
+          +A+
+ ++++++++ + +
+            +
+ ++++++++ + +
+B+          +
++++++++++++
#end
```

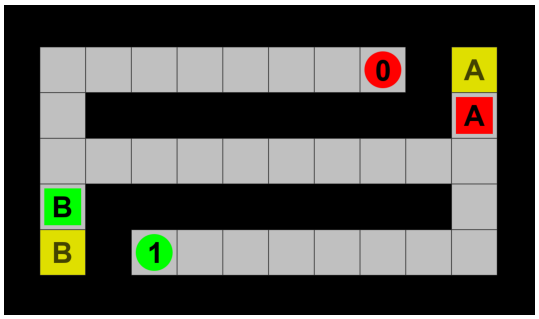


Figure 3: The textual representation of a level (left), and its graphical visualisation (right).

```
<color>: <object>, <object>, ..., <object>
<color>: <object>, <object>, ..., <object>
...
<color>: <object>, <object>, ..., <object>
```

where each `<color>` is an allowed color, and each `<object>` is either the name of a box type (`A`, ..., `Z`) or the name of the agent (`0`,...,`9`). Note that this specification forces all boxes of the same type to have the same color (e.g. there can't be both a blue and a red `A` box; all `A` boxes must have the same color). The agent and all box types that are used in a level must occur exactly once in the level's color declaration.

Figure 3 shows a full textual description of a simple level (left) accompanied by its graphical visualization (right). In this level, agent `0` can only move box `A`, and agent `1` only box `B`.

3 Actions

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

a) *Move action.* A move action is represented in the textual form

$$Move(move\text{-}dir\text{-}agent)$$

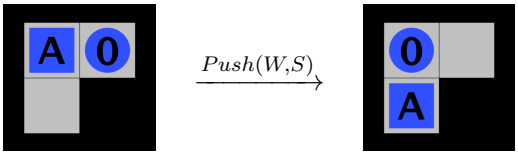
where *move-dir-agent* is one of N (north), W (west), S (south), or E (east). $Move(N)$ means to move one cell to the north of the current location. For a move action to be successful, the following must be the case:

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

b) *Push action.* A push action is represented in the textual 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:

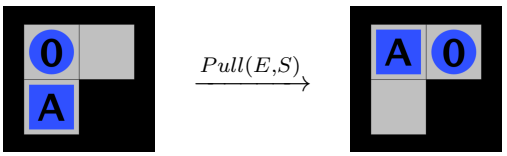
- The neighbouring cell of the agent in direction *move-dir-agent* contains a box β of the same color as the agent.
- The neighbouring cell of β in direction *move-dir-box* is currently free.

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 the agent and the box to swap positions by simply performing an action like $Push(W, E)$.

c) *Pull action.* A pull action is represented in the textual 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, relative to 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.

d) *No-op action.* The NoOp action is represented in the textual form $NoOp$. The action represents the agent doing nothing. It has no parameters and is always successful.

If the 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, i.e. doing nothing. So if e.g. an agent tries to move into an occupied cell, it will simply stay in the same cell.

If a level has several agents, these agents can perform simultaneous actions. The actions of the individual agents are assumed to be completely synchronised, hence we consider joint actions, which have the textual representation

```
<action0>; <action1>; ... ; <action9>
```

In a joint action, `<action0>` is the action performed by agent `0`, `<action1>` is the action performed by agent `1`, etc. Which cells are occupied is always determined at the beginning of a joint action, so it is e.g. not possible for one agent to move into a cell in the same joint action as another one leaves it. Simultaneous actions can be conflicting if two or more agents try to move either themselves or boxes into the same cell, or if two agents attempt to move the same box. If this happens, then neither agent will succeed in their action, and both agents perform a *NoOp* instead.

4 Server

To simulate the environment, an environment server (`server.jar`) is provided. The server loads a level and tracks the actual state of the world, and agents interact with the environment by communicating with the server through a client. The client communicates with the server through the standard streams *stdin*, *stdout*, and *stderr*. Clients can thus be implemented in any programming language that can read from and write to these streams.

The server and client use a text-based protocol to communicate over the streams. The protocol text is ASCII encoded, and proceeds as follows:

1. The client sends its name to the server, terminated by a newline (CRLF or LF). This is the name of the client that will be shown in the GUI of the environment simulation.

2. The server sends the contents of the level file to the client, exactly as it occurs byte-for-byte (with the addition of a final CRLF or LF if the level file is not properly terminated by a newline).

3. The client sends the server either a joint action (specified above) or a comment (which is any string starting with a hashtag symbol (`#`)). The line is terminated by a newline.

4. If the client's message was a comment, then the server prints this message to its own *stdout*.

5. If the client's message was a joint action, then the server simulates the action and sends back a line of the form

   ```
   <success0>; <success1>; ... ; <success9>
   ```

   where each `successN` is either `true` or `false` indicating whether that agent's action succeeded of failed.

6. Steps 3-5 are repeated until the client shuts down, or the server terminates the client. After the client shuts down or is terminated, the server will write a brief summary of the result to its own *stdout*.

Server		Client
	1	`ExampleClient`
`#domain`	2	
`hospital`	3	
`#levelname`	4	
`SAExample`	5	
`#colors`	6	
`blue: 0, A`	7	
`#initial`	8	
`+++++`	9	
`+0A +`	10	
`+++++`	11	
`#goal`	12	
`+++++`	13	
`+0 A+`	14	
`+++++`	15	
`#end`	16	
	17	`Move(E)`
`false`	18	
	19	`Push(E,E)`
`true`	20	
	21	`Move(W)`
`true`	22	

Table 1: Example of interaction between server and client.

The client receives the messages from the server on its stdin, and sends its own messages to the server on its stdout. Anything the client writes on its stderr is directly redirected to whatever the server's own stderr is connected to (typically the terminal).

Table 1 illustrates a complete interaction between a server and client. The left and right columns show what the server and client send, respectively. The given exchange will lead to the level being solved.

For details on different modes and options for the server, run the server with the -h argument:

```
java -jar server.jar -h
```

5 Goal of the project

The goal of the programming project is to implement and experiment with AI clients that can complete a wide range of levels. *Completing* a level means to perform a sequence of joint actions that will result in all goal cells being occupied by boxes or agents of the corresponding types. We will only consider levels that *can* actually be completed by *some* sequence of actions.

6 What we provide

For this assignment, we provide you with basic clients implemented in Java and Python. To obtain the implementations, download the archive `searchclient.zip` and unzip it somewhere sensible. You can choose to work with either the Java or Python client as you please. Do note, though, that you can expect significantly better performance when working in Java, that is, you will probably be able to solve more levels, and much faster (could be one order of magnitude or more). The clients contain a full implementation of the GraphSearch algorithm in Figure 3.7 of Russell & Norvig. It is this search client that you are expected to base your solution on. When you have decided which client to work with, open a command-line interface (command prompt/terminal) and navigate to the relevant folder, either `searchclient_java` or `searchclient_python`. This directory contains a readme file for the client, named `readme-searchclient.txt`. This file shows a few examples of how to invoke the server with the client, and how to adjust memory settings. The directory `levels` contains example levels, some of them referred to in this assignment. You are very welcome to design additional levels to experiment with, and for testing your client.

To complete the default Java version of the assignment, it is required that you can compile and execute Java programs. You should therefore make sure to have an updated version of a Java Development Kit installed before the continuing with the assignment questions below. Both *Oracle JDK* and *OpenJDK* will do. Additionally you should make sure your PATH variable is configured so that 'javac' and 'java' are available in your command-line interface (command prompt/terminal). If you use the Python version, you should similarly make sure that you have an updated version of Python that you can run from the command-line interface.

Benchmarking

Throughout the exercises you are asked to benchmark and report the performance of the client. For this you should use the values printed on the lines just before "Found solution of length xxx" (the length of a solution is the number of steps in it, that is, the number of moves made by the agent in order to solve the level). In cases where you run out of memory or your search takes more than 3 minutes (you can set a 3 minute timeout using the option `-t 180`), 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.

—Assignment Part I—

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 2. To complete this exercise you only need to modify `Frontier.java`.

a) The client contains an implementation of breadth-first search via the `FrontierBFS` 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 2.*

b) Run the BFS client on `SAD2.lvl` and report your benchmarks. Explain which factors make `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. Whenever relevant, use the relevant notions from the course curriculum to assist you in making the answer as clear and technically precise as possible.*

c) Modify the implementation so that it supports depth-first search (DFS). Specifically, implement the class `FrontierDFS` such that `SearchClient.search()` behaves as a depth-first search when it is passed an instance of this frontier. 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 2 as well as very briefly explain how your implementation of DFS differs from the implementation of BFS.*

d) 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` (included as a separate file under *Assignments* on Canvas) and report how many states are generated by each of the two algorithms. You should allow the client to use at least 2 GB of memory, preferably more (but note that you might risk running out of memory no matter how high you set it). Why is there such a huge difference between the number of generated states? *For this question, you need to fill in the relevant lines of Table 2 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 relevant notions from the course curriculum to assist you in making the answer as clear and technically precise as possible.*

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

f) Benchmark the performance of both BFS and DFS on the two levels `SAFirefly.lvl` and `SACrunch.lvl`, shown in Figure 4. *For this question, you only have to fill in the relevant lines of Table 2.*

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

Level	Frontier	Time	Memory Used	Solution length	States Generated
SAD1	BFS				
SAD1	DFS				
SAD2	BFS				
SAD2	DFS				
SAfriendofDFS	BFS				
SAfriendofDFS	DFS				
SAfriendofBFS	BFS				
SAfriendofBFS	DFS				
SAFirefly	BFS				
SAFirefly	DFS				
SACrunch	BFS				
SACrunch	DFS				

Table 2: Benchmarks table for Exercises 1 and 2.

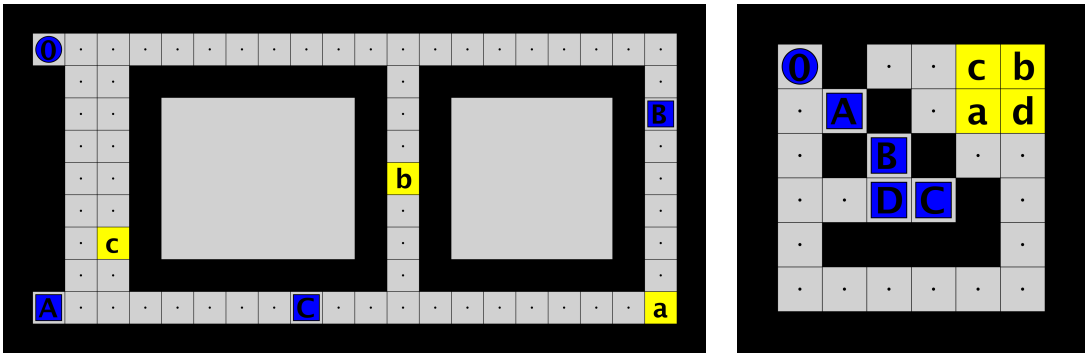


Figure 4: The (relatively simple) levels `SAFirefly.lvl` (left) and `SACrunch.lvl` (right).

```
#domain
hospital
#levelname
SAsoko1_08
#colors
blue: 0, A
#initial
++++++++++
+0A      +
++++++++++
#goal
++++++++++
+       A+
++++++++++
#end
```

Figure 5: `SAsoko1_08.lvl`.

```
#domain
hospital
#levelname
SAsoko2_08
#colors
blue: 0, A
#initial
++++++++++
+0A      +
+        +
+        +
+        +
+        +
+        +
+        +
+        +
++++++++++
#goal
++++++++++
+       A+
+        +
+        +
+        +
+        +
+        +
+        +
+        +
++++++++++
#end
```

Figure 6: `SAsoko2_08.lvl`.

```
#domain
hospital
#levelname
SAsoko3_08
#colors
blue: 0, A
#initial
++++++++
+0A    +
+ A    +
+ A    +
+ A    +
+ A    +
+ A    +
++++++++
#goal
++++++++
+     A+
+     A+
+     A+
+     A+
+     A+
+     A+
++++++++
#end
```

Figure 7: `SAsoko3_08.lvl`.

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 contains two flaws that results in an excess use of memory: 1) The location of walls and goal cells are static (i.e. never changes between two states), yet each state contains its own copy, and 2) when we construct the initial state, the widths and heights of arrays are set to 130 regardless of the actual size of a level, and the number of colors are similarly fixed. Rectify these two flaws and report your new benchmarks in a format like that of Table 2. 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 `State.java` and `SearchClient.java`. *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 improvement is; 3) a few sentences explaining how you modified the code.*

Exercise 3 (State space growth)

Consider levels of the form shown in Figure 5–7. We let the *size* of such a level be the width of the level excluding walls. The level `SAsoko1_08.lvl` has size 8. In `searchclient.zip` we have provided you with levels of different sizes. What are the largest levels of each type that your BFS client can currently handle? Why is there such a big difference between the biggest size level you can handle of the three types? *To complete this exercise, you should: 1) Include benchmarks of the biggest sizes of `SAsoko1`, `SAsoko2` and `SAsoko3` levels that your BFS client can solve, in the style of Table 2; 2) Provide a brief explanation of why there is such a big difference between the biggest size level you can handle of the three types. Try to make your answer as mathematically precise as possible. Hint: think about the state space sizes.*

The previous exercise shows that even relatively simple problems in AI are completely infeasible to solve with naive, uninformed search methods. In part II of the programming project, 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 (Other levels, other domains)

Consider again the Sokoban levels found here:

https://sokoban-game.com/packs/sokogen-990602-levels/

Try to test your client on some of these levels, and see whether your optimised client can solve them (using BFS). Since Sokoban does not allow pulls and you can't push around corners, you will have to modify `Actions.java` for this exercise. You will also have to create the level files yourself for the levels you'd like to try out.

Also try out your client on the labyrinth levels `SAlabyrinth.lvl`, `SAlabyrinthOfStBertin.lvl`, and `SAmicromousecontest2011.lvl` (the latter is the one from the micro mouse contest that was considered in one of the previous exercises). Why are these labyrinth levels so much easier to solve than most of the previously considered levels in this assignment?

For this exercise, include the benchmarks of your selected Sokoban levels. Make sure to state the level numbers. Also include the discussion on why the labyrinth levels are simple for the search client (but no need to include the benchmarks for those).

—Assignment Part II—

Exercise 5 (Heuristics)

Uninformed search strategies can only take us so far. Your next task is to implement an informed search strategy and then provide it with some proper information via the heuristics 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 `Frontier.java` and `Heuristic.java`.

a) Write a best-first search client by implementing the `FrontierBestFirst` 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 appropriate data structures in the frontier.

`HeuristicAStar` and `HeuristicGreedy` 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` 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 worth 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.

b) Benchmark the performance of best-first search with your heuristics by filling in Table 3. 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 3; 2) analyse the improvements provided by $A^\star$ and greedy best-first search over BFS and DFS.*

Level	Evaluation	Time	Mem Used	Solution length	States Generated
SAD1	A*				
SAD1	Greedy				
SAD2	A*				
SAD2	Greedy				
SAfriendofDFS	A*				
SAfriendofDFS	Greedy				
SAfriendofBFS	A*				
SAfriendofBFS	Greedy				
SAFirefly	A*				
SAFirefly	Greedy				
SACrunch	A*				
SACrunch	Greedy				
SAsoko1_64.lvl	Greedy				
SAsoko2_64.lvl	Greedy				
SAsoko3_64.lvl	Greedy				

Table 3: Benchmarks table for Exercise 5, using the best-first search client.