02285 AI and MAS, SP18
Warmup Assignment
Due: Monday 12 February at 20.00

Martin Holm Jensen and Thomas Bolander

This assignment is to be carried out in **groups of 2–3 students**, with 3 as the recommended number. Your solution is to be handed in via CampusNet: 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 Java source files and level files (the ones you have modified or added, and *only* those).

Introduction

In this assignment you will get a sneak peek of the programming project in this course. You can read more about this project in the file `prog_proj_assignment.pdf`. *Before reading on, you should read sections 1–6 of* `prog_proj_assignment.pdf`. The following will refer to the concepts introduced there. (You don't have to download and look at `environment.zip`).

For this assignment, we provide you with a client implemented in Java, which you will improve. To obtain the implementation, download the archive `searchclient.zip`. The Java client we provide is in the directory `searchclient`. It is a full implementation of the Graph-Search algorithm in Figure 3.7 of Russell & Norvig. We have named the relevant Java methods according to the naming conventions in Russell & Norvig. It is this search client that you are expected to base your solution on. The directory `sampleclients` contains some additional clients, most of them very simple. The directory `levels` contain example levels, some of them referred to in this assignment or the readmes. You are very welcome to design additional levels to experiment with, and for testing your client.

The archive contains two readme files, `readme-server.txt` and `readme-searchclient.txt`. As explained in `prog_proj_assignment.pdf`, `readme-server.txt` contains many examples on how to invoke the server and covers all its functionality. For this assignment, it should be enough to look at the first part of this readme, the part that describes the command line parameters (some of the later parts of the readme refer to multi-agent levels that are not considered in this assignment). The other readme, `readme-searchclient.txt`, describes how to run the server using the provided Java search client.

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 all supposed to be familiar with these techniques. 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 consult Chapter 3

Level	**Client**	**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 3.

of Russell & Norvig. In particular, as mentioned above, the implemented Java search client is based on the Graph-Search algorithm in Figure 3.7 of Russell & Norvig.

Since some of you might not be familiar, or comfortable, with Java, we also provide a search client written in Python (in `searchclient_python.zip` in the `sampleclients` directory). The Python client is however not as thoroughly tested as the Java client, and is provided "as is". Please talk to the teacher or teaching assistant if you plan to use Python instead of Java. The text below refers to the Java version.

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 (Windows), Terminal (Unix, Linux & Mac)).

Benchmarking

Throughout the exercises you are asked to benchmark and report the performance of the client (solver). For this you should use the values printed after "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 your JVM runs 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 JVM 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.

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 `Strategy.java`.

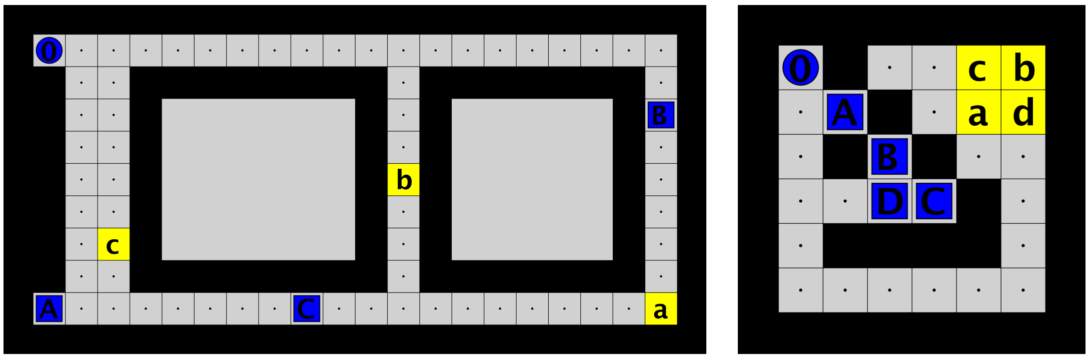

Figure 1: The (relatively simple) levels `SAFirefly.lvl` (left) and `SACrunch.lvl` (right).

a) 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.*

b) 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).*

c) 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.*

d) On the levels `SAD1.lvl` and `SAD2.lvl`, DFS is much more efficient than BFS. But this is not always the case. Design a level `friendofBFS.lvl` on which your BFS client finds a solution almost immediately generating no more than 1,000 nodes, but where the DFS client generates at least 50,000 nodes. In the Node class, the order in which the children of a node are added to the frontier is randomised using a pseudo-random number with a fixed seed. On certain levels, the behaviour of BFS and DFS can be very sensitive to this order. To make sure that your `friendofBFS` level is not exploiting the order in the default random seed, try out BFS and DFS on the level with 3 different random seeds, making sure that the constraints on the number of generated nodes are satisfied for all three. Try out seed values 1, 2 and 3 by changing the number in the following line of the Node class:

```
private static final Random RND = new Random(1)
```

Remember to compile before running BFS and DFS again with the new seed. You should at least allow the client to use 2 GB of memory, preferably more. Report your benchmarks on `friendofBFS.lvl`. Why does BFS perform so much better than DFS on your level? *Give a brief, but conceptually precise, answer. Make sure to include your* `friendofBFS` *level in the pdf file of your submission.*

```
++++++++++++
+OA       a+
++++++++++++
```

Figure 2: `SAsoko1_12.lvl`.

```
++++++++++++
+OA       a+
+          +
+          +
+          +
+          +
+          +
+          +
+          +
+          +
+          +
++++++++++++
```

Figure 3: `SAsoko2_12.lvl`.

```
++++++++++++
+OA       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
+ A       a+
++++++++++++
```

Figure 4: `SAsoko3_12.lvl`.

e) Design a level `friendofDFS.lvl` on which your DFS client finds a solution almost immediately generating no more than 1,000 nodes, but where the BFS client generates at least 50,000 nodes. As in the previous questions, you should check both algorithms with random seeds 1, 2 and 3 and make sure that the constraints on the number of generated nodes are satisfied for all. Report your benchmarks on `friendofDFS.lvl`. Why does DFS perform so much better than BFS on your level? *Give a brief, but conceptually precise, answer. Make sure to include your* `friendofDFS` *level in the pdf file of your submission.*

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

Exercise 2 (State spaces and solution lengths)

a) Consider a level of the form in Figure 2. Let n denote the width of the level excluding walls. The level `SAsoko_12.lvl` has a total width of 12 including walls, so $n = 10$ 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 possible states) in Θ-notation? *Report your answers as well as the calculations/reasoning leading to the answers.*

b) Consider a quadratic level of the form in Figure 3. Let n denote the width and breadth of the level excluding walls (so $n = 10$ in Figure 3). Answer the same questions as you did for Figure 2. *Report your answers as well as the calculations/reasoning leading to the answers.*

c) Consider a quadratic level of the form in Figure 4. Let n denote the width and breadth of the level excluding walls. Answer the same questions as for Figure 2 and 3.[1] Is the size of the state space polynomial in n, that is, is it in $O(n^k)$ for some k? *Report your answers as well as the calculations/reasoning leading to the answers.*

d) Run your BFS client on the different sizes of `SAsoko1`, `SAsoko2` and `SAsoko3` 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.*

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

Level	Evaluation	Time	Mem 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				

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

This exercise shows that even relatively simple problems in AI are completely infeasible to solve with naive, uninformed search methods. Later 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 3 (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 nodes and the use of more clever data structures. In this exercise, we consider two simple such optimisations.

The `Node` class contains two flaws that results in an excess use of memory: 1) The location of walls and goals are static (i.e. never changes between two nodes), yet each node contains its own copy, 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 `Node.java` and `SearchClient.java`. *Your answer to this question should include: 1) the new benchmarks (for the same levels as before, except for the `SAsoko` levels just choose the biggest of each type that you are able to solve); 2) a few words on how significant the improvement is; 3) a few sentences explaining how you modified the code.*

Exercise 4 (Heuristics)

Uninformed seach 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 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). Your benchmarks must be reported in a format like that of Table 2. To complete this exercise you will need to modify `Strategy.java`, `Heuristic.java` and `SearchClient.java`.

a) Write a best-first search client by implementing the `StrategyBestFirst` class. The `Heuristic` argument in the constructor must be used to order nodes. As it implements the `Comparator<Node>` 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 `Heuristic.java` throws a `NotImplementedException`. Implement a heuristic function (or several ones). Remember that `h(n)` should estimate the length of a solution from the node `n` to a goal node, while still being cheap to calculate. You may find it useful to do some preprocessing of the initial state in the `Heuristic` constructor.

   For this question, you should in the report include your choice of data structure for the frontier as well as a detailed and mathematically precise description of the heuristics you have implemented.

b) Informed search with a fairly simple search heuristics can already solve a number of levels relatively efficiently. Benchmark the performance of best-first search with your heuristics by filling in Table 2. Analyse the 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.*

c) Argue whether or not your heuristic function is admissible and consistent (cf. Chapter 3 of Russell & Norvig). *Be brief, but mathematically precise, in your answer.*

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

Consider the following level `SASimple0`:

```
++++
+O +
+  A+
+a  +
++++
```

a) 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.*

Exercise 6 (Heuristics, state spaces and solution lengths)

a) In Exercise 2 you were asked to run your BFS client on various levels of the forms shown in Figures 2–4. Now run your best-first search client on `SAsoko1_48.lvl`, `SAsoko2_48.lvl` and `SAsoko3_48.lvl` and compare with the BFS benchmarks.

If with your current heuristics you are still unable to solve `SAsoko3_48.lvl`, then consider ways to improve the heuristics to make it solvable. Ideally, you should provide a new heuristics so that `SAsoko3_48.lvl` can be solved, or you should at least sketch how such a heuristics could be constructed. *Hints*: Running greedy-best first search can often give valuable information about your heuristics and its potential weaknesses. You can add additional levels of intermediate sizes to experiment with, if necessary.

For this question, you should: 1) very briefly comment on the improvements of best-first search over BFS on the `SAsoko` *levels; 2) give a precise mathematical description of your improved heuristics, or at least sketch your ideas for how to define such a heuristics.*

b) *OPTIONAL.* There are several levels of varying complexity provided in `searchclient.zip`. What are the most difficult ones you can now solve using best-first search?