02285 AI and MAS
Warmup Assignment
Due: Tuesday 9 February 2016 at 13.00

Martin Holm Jensen and Thomas Bolander

This assignment is to be carried out in **groups of two**. It is allowed, but not recommended, to work alone. If you wish to form a group of 3 people, you have to consult the teachers to get their acceptance first. You will be required to do more if you work in a group of 3 people.

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).

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 `searchclient.zip`. This archive also contains the two levels `SAD1.lvl` and `SAD2.lvl`. You can find more information on running the client in the `readme` files.

The purpose of the following 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 of Russell & Norvig. In particular, the implemented Java search client is based on the GRAPH-SEARCH algorithm in Figure 3.7 of Russell & Norvig, and the relevant Java methods are named accordingly.

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 hits the 5 minute limit, use the latest values that have been printed (put "-" for solution length). If you experience a lot of swapping when benchmarking, you can decrease the value of `limitRatio` in the `Memory` class.

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`.

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. Conclude what the length of the shortest solution for this level is. Explain how you know it is a shortest solution.

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 is the case for `SAD1.lvl`. (You can also try to experiment with levels of intermediate complexity between `SAD1.lvl` and `SAD2.lvl`).

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.

d) Design a level `friendofDFS.lvl` on which your DFS client finds a solution almost immediately (at most half a second, expanding no more than 200 nodes), but where the BFS client explores at least 10.000 nodes (and possibly exhausts its memory). Report your benchmarks on `friendofDFS.lvl`. Why does DFS perform so much better than BFS on your level?

e) Design a level `friendofBFS.lvl` on which your BFS client finds a solution almost immediately (at most half a second, expanding no more than 200 nodes), but where the DFS client explores at least 10.000 nodes (and possibly passes the standard timeout limit of 5 minutes). Report your benchmarks on `friendofBFS.lvl`. Why does BFS perform so much better than DFS on your level?

Level	Client	Time	Memory Used	Solution length	Nodes Explored
SAD1	BFS				
SAD1	DFS				
SAD2	BFS				
SAD2	DFS				
friendofDFS	BFS				
friendofDFS	DFS				
friendofBFS	BFS				
friendofBFS	DFS				

Table 1: Benchmarks table for Exercises 1 and 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 optimisations include reduced memory footprint of nodes and the use of more clever data structures. Your benchmarks must be reported in a format like that of Table 1. To complete this exercise you will need to modify `Node.java` and `SearchClient.java`.

a) 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 Table **??**. (To avoid `ArrayIndexOutOfBounds` exceptions, you should make sure that `MAX_ROW` and `MAX_COLUMN` are large enough to contain the border of the level.)

b) The locations of boxes in a level are *not* static. Explain which data structure would allow you to save memory in most levels, while still offering good performance when it comes to lookup. In terms of running time, what would the impact of such a modification be on `isGoalState()` and `getExpandedNodes()`?

Exercise 3 (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`.

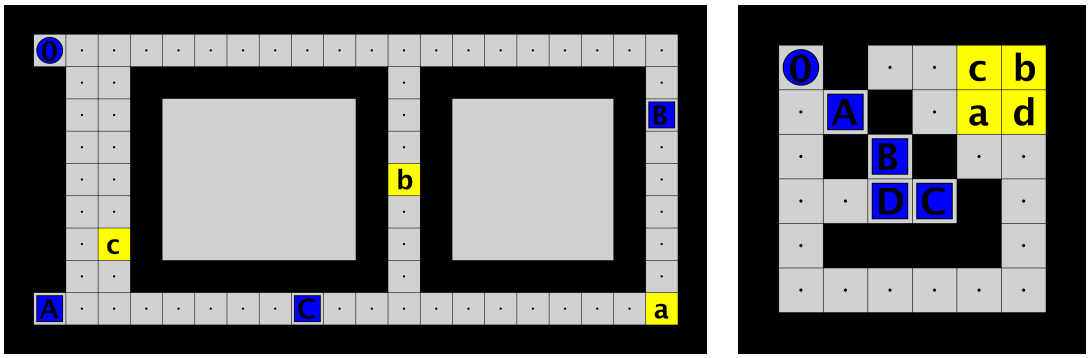

Figure 1: The (relatively simple) competition levels `Firefly.lvl` (left) and `Crunch.lvl` (right) from the 2011 competition at DTU.

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, and state which one you have used.

b) `AStar`, `AStarWeighted` and `Greedy` are implementations of the abstract `Heuristic` class, each of which implement a distinct evaluation function `f(n)`. As the names suggest, they implement $A^\star$, $WA^\star$ and *greedy best-first search*, respectively.[1] Currently, the crucial *heuristic function* `h(n)` in the `Heuristic` class always returns 0. Implement a better heuristic function (or several ones), then benchmark and report the results of your best-first search client with each of the three types of evaluation functions. Explain your findings.

   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.

c) Argue whether or not your heuristic function is admissible.

d) Informed search with a fairly simple search heuristics can already solve a number of levels relatively efficiently, and can potentially solve some of the simplest competition levels from the previous years. Benchmark the performance of best-first search with your heuristics on the two competition levels `Firefly.lvl` and `Crunch.lvl` from the 2011 competition at DTU, shown in Figure 1. Both were solvable by all the submitted competition clients. Firefly is quite simple and has a shortest solution of length 60. Crunch is a little more challenging, and has a shortest solution of length 98. Analyse your findings, that is, discuss which best-first

[1] $WA^\star$ is not described in Russell & Norvig. It is a simple variant of $A^\star$ where the evaluation function $f(n) = g(n) + h(n)$ is replaced by $f(n) = g(n) + Wh(n)$ for some constant $W > 1$.

Level	**Evaluation**	**Time**	**Mem Used**	**Solution length**	**Nodes Explored**
`SAD1`	A*				
`SAD1`	WA*				
`SAD1`	Greedy				
`SAD2`	A*				
`SAD2`	WA*				
`SAD2`	Greedy				
`friendofDFS`	A*				
`friendofDFS`	WA*				
`friendofDFS`	Greedy				
`friendofBFS`	A*				
`friendofBFS`	WA*				
`friendofBFS`	Greedy				
`Firefly`	A*				
`Firefly`	WA*				
`Firefly`	Greedy				
`Crunch`	A*				
`Crunch`	WA*				
`Crunch`	Greedy				

Table 2: Benchmarks table for Exercise 3, using the best-first search client.

search algorithms can solve which levels and why.