# 02285 AI and MAS
# Mandatory Assignment 1
## *Due: Tuesday 10 February 2015 at 13.00*

Martin Holm Jensen and Thomas Bolander

## Formalities

This exercise is assessed individually, so, per the DTU Study Handbook, the following rules apply:

- Discussing the exercises with your fellow students *is* allowed.
- Your answers must be your own, *from the moment you start writing down anything.*
- You must hand in your solutions individually.
- Indicate on the first page of your hand-in who you had discussions with.
- *All* parties involved in a breach of these rules will be held responsible.

Your solution is to be handed in via CampusNet: Go to Assignments, then choose Mandatory Assignment 1. 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).

Even though this is an individual assignment, we strongly encourage collaboration with fellow students, in particular, we encourage you to work in **groups of two** on this assignment. This means that we allow two collaborating students to submit identical source code, but the report should still be written individually according to the rules stated above. Remember to state in the report who you collaborated with. If you wish to form a group of 3 people and submit identical source code, you have to consult the teachers (Thomas and Martin) to get their acceptance first. More will be required if you work in a group of 3 people.

### Introduction

In this assignment you will get a sneak peek of the programming project in this course (aka Mandatory Assignment 3). You can read more about this project on CampusNet in the file `assignment03.pdf` located in the folder `Mandatory Assignment 3`. *Before reading on, you should read sections 1–6 of* ***`assignment03.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` from the file sharing folder `Mandatory Assignment 1`. 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. You are all supposed to be familiar with these techniques from the prerequisite course *02180 Introduction to Artificial Intelligence* or a similar course from another university. However, some of you might not be familiar with all of the techniques, and others might simply need a brush-up. The purpose of this assignment is to bring all students up to a sufficient and comparable level in AI search basics before beginning to introduce the new curriculum of the present course, which heavily relies on these basics. As you are already expected to be familiar with the relevant curriculum for this assignment, there is no required text book reading involved. However, for those of you who 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. As an alternative to Russell & Norvig, an ultrabrief introduction to all the relevant concepts can be found in Section 2.3 of Geffner & Bonet (find the link to this book on the CampusNet welcome page).

### Benchmarking

Throughout the exercises you're 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

| Level | Client | Time | Memory Used | Solution length | Nodes Explored |
|---|---|---|---|---|---|
| `SAD1` | BFS | | | | |
| `SAD1` | DFS | | | | |
| `SAD2` | BFS | | | | |
| `SAD2` | DFS | | | | |
| `custom` | BFS | | | | |
| `custom` | DFS | | | | |

Table 1: Benchmarks for Exercise 1

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.

   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 `custom.lvl` on which your DFS client finds a solution almost immediately (at most half a second, expanding no more than 150 nodes), but where the BFS client explores at least 10000 nodes (and possibly exhausts its memory). What level did you come up with? Report your benchmarks on `custom.lvl`.

## 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 2. 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

| Level | Client | Time | Memory Used | Solution length | Nodes Explored |
|---|---|---|---|---|---|
| `SAD1` | BFS | | | | |
| `SAD1` | DFS | | | | |
| `SAD2` | BFS | | | | |
| `SAD2` | DFS | | | | |
| `custom` | BFS | | | | |
| `custom` | DFS | | | | |

Table 2: Benchmarks for Exercise 2, after optimisations.

regardless of the actual size of a level. Rectify these two flaws and report your new benchmarks in Table 2. (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 3. 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, 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

[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 | | | | |
| `custom` | A* | | | | |
| `custom` | WA* | | | | |
| `custom` | Greedy | | | | |
| `Firefly` | A* | | | | |
| `Firefly` | WA* | | | | |
| `Firefly` | Greedy | | | | |
| `Crunch` | A* | | | | |
| `Crunch` | WA* | | | | |
| `Crunch` | Greedy | | | | |

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

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. 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. Explain your findings.