02285 AI and MAS
Warmup Assignment
Due: Monday 13 February at 20.00

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

Group sizes and group work. This assignment is to be carried out in **groups of 3 students**. Groups of 2 or 4 students are also allowed. Groups of 2 students will be expected to deliver the same as groups of 3 students. Groups of 4 students are however expected to deliver more. The exercises are still the same, but groups of 4 will be expected to deliver a more substantial and polished hand-in (e.g. doing more work on the heuristics). All exercises are intended to be solved as group work, not by splitting the exercises between you. You will of course be forced to split some subtasks between you, but it is essential that all group members have been involved in all parts of the assignment, so you should at least make sure to review, revise and improve on everything made by your team members. All initial brainstorming about how to solve a given task should certainly also be done in the full group.

Collaboration policy. It is not allowed to show solutions to other groups or see the solutions of other groups. You are not allowed to share any part of your solutions, neither verbally nor in writing, and all formulations should be your own, independent of the other groups. Violations of these conditions will be considered as violations of the academic honesty and will be reported.

Handing in. Your solution is to be handed in via DTU Learn. You should hand in two *separate* files:

1. A **movie file** (mp4, mov or other standard movie format) containing a video presentation of your solution to the assignment. The video can be at most 10 minutes long, anything after the 10 minute mark will not be assessed. The file `guide_for_video.pdf` provides general instructions for what to include in the video and how to make it. The first slide has to contain the names and study IDs of all students of the group as well as a brief *group declaration*: A brief specification of who did what. This can for instance be done in the form of a small table. The purpose of the group declaration is to allow assessment differentiation in case some group members did significantly more or less than others.
2. A **zip file** containing the relevant source code files and level files (the ones you have modified or added, and *only* those).

Make sure to only use private repositories for sharing work in your group. Using public repositories will be treated as a violation of the academic honesty, as you are then—implicitly or explicitly—violating the collaboration policy stated above. We take this very seriously!

Important about your hand-ins and how they will be assessed. The assignments involve both practical implementation and theoretical considerations. The course is not a programming course, so you will be evaluated more on your conceptual understanding of the involved AI techniques than on the quality and elegance of your code (though you should of course strive to make good code as well). The assessment will hence primarily focus on the quality of the video presentation you hand in, in particular your ability to formulate yourselves in mathematically and technically precise ways using the appropriate technical concepts. This is also important in your benchmark analyses. You should use your theoretical understanding to get behind the numbers and try to understand and precisely convey why you get the numbers you get. This is a very important skill to acquire in AI. *Conciseness* will be a very important parameter of the assessment. Using the technical terms in a precise and correct way can usually lead to descriptions that are *both* more brief and more precise. It is for instance much better to say "the branching factor grows exponentially with the number of agents" than "when the number of agents become large, there are so many more options of what the agents can do, the number of possible options simply explode".

1 Goal of the assignment

The main 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. Furthermore, a goal is to give you some insights into and practical experience with the domain to be used in the programming project. Finally, a goal is to create the basis and motivation for the remaining curriculum of the course.

2 Preparation

Before reading on, make sure you have carefully read the document `hospital_domain.pdf` that describes the hospital domain in detail. You might also need to (re)read Chapter 3 of R&N, 3ed, to brush up on the details of the techniques and concepts required to solve this assignment.

3 What we provide and what you need

For this assignment, we provide you with a basic client implemented in Java. To obtain the implementation, download the archive `searchclient.zip` and unzip it somewhere sensible. When you have unpacked the repository, open a command-line interface (command prompt/terminal) and navigate to the folder `searchclient_java`. 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 assignments, 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 (JDK) installed before the continuing with the exercises 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).

Compiling and running the code is supposed to always take place from the command line. You are welcome to edit the Java source files in whatever editor you like. You can of course use your favourite IDE as editor, but you should still compile and run the program from the command line.

Benchmarking

Throughout the exercises you are asked to benchmark and report the performance of the client. For this you should use the value xxx printed on the line "Found solution of length xxx" as well as the values on the two lines preceding it. The last lines starting with "[client][info]" and "[server][info]" are simply additional messages from the *MAvis* server that can be ignored. 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, using the `-Xmx` option (see `readme-searchclient.txt` for details). Normally allocating half of your RAM is reasonable, e.g. 8GB on a machine with 16GB (the option is then `-Xmx8g`).

Exercise 1 (Familiarise yourselves with the code)

Consult `readme-searchclient.txt` to find out how to invoke the server and client. Try it out on a simple level like `SAD1.lvl`. Currently the client is only producing a fixed sequence of actions, so it can't solve any levels. Look at all the source code files provided in the `searchclient` folder. Try to get an overview of what the code inside each source file does. You don't have to understand all the code in detail, but should at least try to get an overview. The files are as follows:

Action This class defines the available actions. Currently, only the no-op and move actions of the hospital domain are supported.

Color This class defines the available colors (of agents and boxes).

Frontier This file defines the frontier classes for implementing Graph-Search. Currently, only the queue frontier for BFS has been implemented (in `frontierBFS`).

GraphSearch This is where you are later going to implement Graph-Search. For now, the code just returns a fixed sequence of actions. This class is also responsible for writing status messages about the search to the terminal.

Heuristic This class is for defining your heuristic functions.

Memory This class is for keeping track of the memory usage of the client.

NotImplementedException This is an exception to mark the parts of the code that hasn't yet been implemented (and that you will implement).

SearchClient This is the main class for communicating with the server. It first reads a level from the server and constructs an initial state from it (a `State` object). It invokes the search method of the `GraphSearch` class, and if this search completes with a non-empty plan, the plan will be sent to the server (and the server will visualise it).

State This class defines the states in the hospital domain. Make sure you understand how all the relevant information about a state of a level in the hospital domain is represented: agent positions and colors; wall positions; box positions and colors; goal positions.

This question doesn't have to be addressed in the video you hand in, but you're very welcome, if you can make it fit. If so, provide a brief high-level overview of how the code is structured, e.g. making a graphical illustration of the individual classes, what they contain, and how they interact.

Exercise 2 (Breadth-First Search for Multi-Agent Pathfinding)

In these first exercises we only consider multi-agent pathfinding problems. This domain has already been implemented in the client via the `State` and `Action` classes. You will be implementing the Graph-Search algorithm of the textbook (3ed) yourself in order to be able to solve levels using breadth-first search (BFS). You can also find the Graph-Search algorithm on the slides, so feel free to implement it based on the slides, if that is simpler. The client already contains an implementation of the frontier for BFS via the `FrontierBFS` class. To complete this exercise you only need to modify the `GraphSearch` file.

1. Solve the level `MAPF00.lvl` by hand. Test the correctness of your solution by hardcoding it into the `search` method in `graph_search.py`. How does your approach to solving the level differ from how Graph-Search would do it?

2. Now try to implement the Graph-Search algorithm yourself in `GraphSearch.java`. You will need to call methods from both `Frontier.java` and `State.java`. Test your implementation by running the BFS client on the `MAPF00.lvl` level.

3. Run your BFS GraphSearch client on the `MAPF00.lvl`, `MAPF01.lvl`, `MAPF02.lvl`, `MAPF02C.lvl`, `MAPF03.lvl`, `MAPF03C.lvl`, `MAPFslidingpuzzle.lvl` and `MAPFreorder2.lvl` levels and report your benchmarks (filling in the relevant lines of Table 1). Make sure to provide a lot of RAM to the client. Set the speed of solution playback relatively low, so you have time to see what happens, e.g. `-s 500`. Explain what is causing the differences observed between the performance of the client on the different levels. Make sure to explain yourselves in a brief but conceptually and technically precise way, using the relevant notions from the course curriculum, e.g. notions such as *branching factor*, *state space size*, *search tree*, etc. Analyse the numbers you get as precisely as possible, e.g. if the number of generated states grows by a factor x whenever a new agent is added to a specific level, why is that?

Exercise 3 (Depth-First Search for Multi-Agent Pathfinding)

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

Level	Strategy	States Generated	Time/s	Solution length
MAPF00	BFS			
MAPF00	DFS			
MAPF01	BFS			
MAPF01	DFS			
MAPF02	BFS			
MAPF02	DFS			
MAPF02C	BFS			
MAPF02C	DFS			
MAPF03	BFS			
MAPF03	DFS			
MAPF03C	BFS			
MAPF03C	DFS			
MAPFslidingpuzzle	BFS			
MAPFslidingpuzzle	DFS			
MAPFreorder2	BFS			
MAPFreorder2	DFS			
BFSfriendly	BFS			
BFSfriendly	DFS			

Table 1: Benchmarks table for uninformed search.

1. 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 the `MAPF00.lvl`, `MAPF01.lvl`, `MAPF02.lvl`, `MAPF02C.lvl`, `MAPF03.lvl`, `MAPF03C.lvl`, `MAPFslidingpuzzle.lvl` and `MAPFreorder2.lvl` levels and report your benchmarks (fill in the relevant lines of Table 1). Does DFS find solutions faster or slower than BFS and why? And how about the length of the solutions found? Also briefly explain how your implementation of DFS differs from the implementation of BFS.

2. As you probably noticed, DFS often finds solutions much more quickly than BFS. Yet these are then not guaranteed to be optimal. However, DFS is not always quicker at finding solutions than BFS. Design a level `BFSfriendly.lvl` on which your BFS client finds a solution spending significantly less time than the DFS client. Try to design a level where the difference in performance between BFS and DFS is as large as possible. Include benchmark results for both DFS and BFS on your level (filling in the relevant lines of Table 1) as well as videos of both algorithms solving it. Why does BFS perform better than DFS on this level?

The previous assignment illustrated that even relatively simple problems in AI are completely infeasible to solve with naive, uninformed search methods. We will now look at informed search methods, where the search has a sense of direction and closeness to the goal, which can make the search much more efficient.

Exercise 4 (Informed Search and Heuristics)

Your next task is to implement an informed search strategy and then provide it with some proper information via the heuristic function. When referring to $f(n)$, $g(n)$ and $h(n)$ below, they are used in the same way as in the textbook and the slides. To complete this exercise you will need to modify `Frontier.java` and `Heuristic.java`.

1. 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 always returns 0. This means that $A^\star$ will always expand a node n with minimal value of $f(n) = g(n) + h(n) = g(n)$, that is, a node of minimal depth in the search tree. This makes $A^\star$ essentially behave as a BFS search (except potentially for the order in which nodes at a certain depth are expanded). Before moving on to the next question, you should check that your $A^\star$ indeed behave in this way (for instance that, as BFS, it always finds optimal solutions).

2. Implement a *goal count heuristic*: A heuristic function $h(n)$ that simply counts how many goal cells are not yet covered by an object of the right type (so far all goal cells are agent goals, so $h(n)$ should simply count how many agents are not at their destination). The code of the goal count heuristic should replace the current code of `h(n)` in the `Heuristic` class.

   To make sure that the heuristic function values are computed correctly, you should do some testing by printing both the state s and its value $h(s)$ whenever the method `h` is called in `Heuristic.java`. Printing to the terminal is done using `System.err.println`. Doing this might be even more useful in the later exercises when you are going to implement more complex heuristic functions. Note, however, that for large levels, you are going to print a lot of states and heuristic values to the terminal, so try to pick your test levels carefully (you are always welcome to design your own levels suitable for testing various aspects of your code).

   Benchmark your heuristic on the levels `MAPF00.lvl`, `MAPF01.lvl`, `MAPF02.lvl`, `MAPF02C.lvl`, `MAPF03.lvl`, `MAPF03C.lvl`, `BFSFriendly.lvl`, `MAPFslidingpuzzle.lvl` and `MAPFreorder2.lvl` by first filling in the relevant cells of Table 2. Then compare your benchmark results to uninformed search (BFS and DFS) and analyse the differences: Are $A^\star$ and greedy best-first search faster and/or better, how much, and why?

3. Now try to design your own improved heuristic function h' to allow informed search to solve more levels and solve them faster. Try to make it as efficient as possible, so that it can solve as many levels of possible. Since best-first search expands nodes with lower h-values before nodes with higher h-values (at least if they have the same distance from the initial state), the h-values should ideally always decrease when getting closer to the goal. Heuristic values should also not be too expensive to compute, as they have to be computed for each generated state. But preprocessing can help. For instance you might

consider to initially once and for all to compute all exact distances between pairs of cells in the level, and then you can later look these up in constant time when computing your heuristic values.

Before implementing anything, first think about what exactly you want the heuristic function to compute. Then make this precise by describing the heuristic function either in mathematical notation or pseudocode (or both). Only when you have a precise description of your heuristic, should you start implementing it. It is always advisable to create pseudocode first. Your video should include a description of the heuristic either using mathematical notation or pseudocode (or both).

When implementing your heuristic, you may find it useful to do some preprocessing of the initial state in the `Heuristic` constructor. After having implemented the first version of your heuristic, it might be worth to do some benchmarking to check how well it performs. You can e.g. use this to compare different choices of heuristics, and see what works best. A well-designed heuristic should give a significant improvement over the goal count heuristic. Greedy best-first normally gives much greater improvements over uninformed search than $A^\star$, so it is advised to primarily benchmark using greedy-best first search. Doing benchmarkings with greedy best-first search also often give valuable insights into your heuristic, 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).

In your video, make sure to not only present the formal description of your heuristic (in mathematical notation and/or pseudocode), but also describe the intuition behind it, how it estimates solution lengths, and what its strengths and weaknesses are.

4. Benchmark the performance of $A^\star$ and greedy best-first search with your heuristic by filling in a table similar to Table 2. Analyse the improvements over uninformed search and informed search with the goal count heuristic by comparing the new benchmarks with your earlier benchmarks. You can also illustrate the improvements in your hand-in by including videos of the various algorithms solving the same levels.

Exercise 5 (Adding support for pushing and pulling boxes)

In this exercise, you will extend the set of actions to include pushing and pulling boxes. However, to avoid making things too complicated, from now on we restrict attention to single-agent levels (you are of course still welcome to consider the multi-agent case if you want some extra challenge). To define the new actions, we need to extend the implementations of Actions(s) and Result(s, a). Start by rereading the definition of applicability in `hospital_domain.pdf` to recall exactly what conditions have to be satisfied for an action to be applicable. In the client, applicability is defined by the `isApplicable` method of `State.java`. The Result function is implemented as the `State` constructor with parameters `parent` and `jointAction`: It takes an existing state `parent` and a (joint) action `jointAction` and creates the new state resulting from executing the (joint) action in the existing state. So to sum up, in the code, the functions Actions and Result are defined via: 1) the definition of the action set in `Action.java` (the names of all possible actions); 2) the definition of the `State` constructor with parent and joint action parameters; 3) the definition of `isApplicable` in `State.java`.

Level	Eval	Heuristic	States Generated	Time/s	Solution length
`MAPF00`	$A^\star$	Goal Count			
`MAPF00`	Greedy	Goal Count			
`MAPF01`	$A^\star$	Goal Count			
`MAPF01`	Greedy	Goal Count			
`MAPF02`	$A^\star$	Goal Count			
`MAPF02`	Greedy	Goal Count			
`MAPF02C`	$A^\star$	Goal Count			
`MAPF02C`	Greedy	Goal Count			
`MAPF03`	$A^\star$	Goal Count			
`MAPF03`	Greedy	Goal Count			
`MAPF03C`	$A^\star$	Goal Count			
`MAPF03C`	Greedy	Goal Count			
`MAPFslidingpuzzle`	$A^\star$	Goal Count			
`MAPFslidingpuzzle`	Greedy	Goal Count			
`MAPFreorder2`	$A^\star$	Goal Count			
`MAPFreorder2`	Greedy	Goal Count			
`BFSFriendly`	Greedy	Goal Count			
`BFSFriendly`	$A^\star$	Goal Count			

Table 2: Benchmarks table for informed search.

1. Extend the code to support pushes and pulls by modifying the above mentioned classes and methods. Use the code for the *Move* action as an inspiration for the *Push* and *Pull* actions. First try to get an overview of what has to be modified in the existing code, and then do the necessary modifications afterwards.

2. Run your extended BFS Graph-Search client on the `SAD1.lvl`, `SAD2.lvl` and `SAD3.lvl` levels and report your benchmarks (using a table similar to Table 1). Explain which factors make `SAD2.lvl` much harder to solve using BFS than `SAD1.lvl`, and `SAD3.lvl` much harder than `SAD2.lvl`. As always, be as clear and technically precise as possible.

3. Benchmark the performance of both BFS and DFS on the two levels `SAFirefly.lvl` and `SACrunch.lvl`, shown in Figure 1.

Exercise 6 (Heuristics for levels with pushing and pulling)

1. Revise your goal count heuristic from Exercise 4 so that it can handle single-agent levels with boxes. Benchmark greedy best-first search with the goal count heuristic against BFS and DFS on the levels `SAFirefly.lvl` and `SACrunch.lvl`. Analyse your results.

2. Design your own improved heuristic function h' to allow informed search to solve more levels and solve them faster. This is essentially the same question as in Exercise 4, except we are now looking at a different type of domain requiring a different type of heuristic (though you might still get inspiration from your old heuristic and the thoughts behind it). Try to challenge yourselves to make the most efficient heuristic possible in terms of being

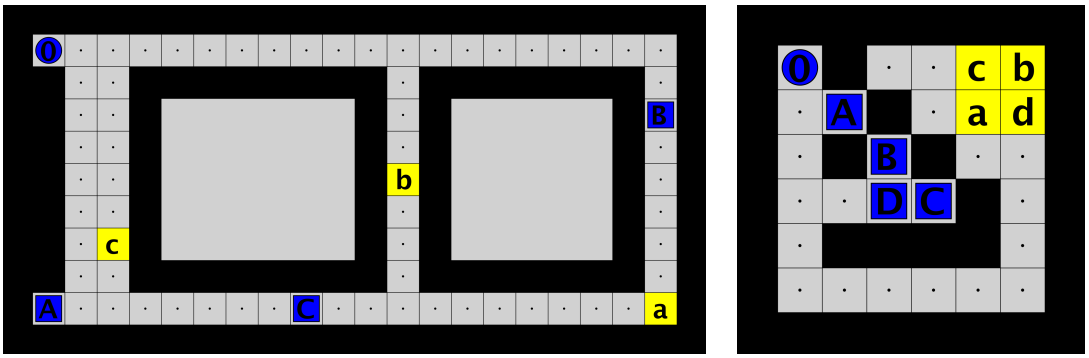

Figure 1: The levels `SAFirefly.lvl` (left) and `SACrunch.lvl` (right).

able to solve as many levels as possible (not necessarily optimally). For any heuristic function you consider, make sure to describe it in mathematical notation and/or pseudocode before implementing it. You should also try to compute a few heuristic values by hand first, to check whether it really manages to direct the search in the right direction (getting closer to the goal should reduce the h-value). Make sure to include the mathematical description of your heuristic and/or the pseudocode in the video that you hand in.

After implementing your first heuristic, you can try to benchmark it and think about possible ways to improve it (if you decide to try to improve it, describe the new version in mathematical notation and/or pseudocode before implementing it). To challenge yourselves, you should look at different types of levels with different features. You could for instance start by looking at the levels `SAsoko1_`n, `SAsoko2_`n and `SAsoko3_`n with $n = 4, 8, 16, \ldots, 128$. How many of those can you solve? Can you improve your heuristic to be able to solve more or all of them? How about levels of a complete different type?

For this assignment, it is important that you benchmark your heuristic on a wide variety of levels, including ones of your own design. In your hand-in, you should include videos of your solutions to levels that demonstrate interesting behaviour of your heuristic (good or bad). Make sure to both include videos that can highlight the strengths and videos that can highlight the weaknesses of your heuristic, and analyse these strengths and weaknesses. You should design your own levels for this part. If you make several distinct heuristics (e.g. first a simple one that is later improved), it can also be a good idea to benchmark the different heuristics against each other and comment on it. It's important to reflect on the benchmark results and the observed behaviour of best-first search with your heuristic (in particular the behaviour of greedy best-first search).

This last question is deliberately more open than the previous. Try to be creative and innovative, both in designing an efficient heuristic and in benchmarking it on different level types of your own design.