02285 AI and MAS

Assignment 4: Mandatory Programming Project

Due: *Friday 7 June at 13.00*

Thomas Bolander, Andreas Garnæs,
Martin Holm Jensen and Mikkel Birkegaard Andersen

1 Introduction

This document describes the mandatory programming project of 02285. The project is designed to be very flexible, allowing many different group sizes and levels of ambition. Successful implementations of the project range all the way from basic solutions using standard techniques to highly research relevant multi-agent systems. Due to the flexibility of the project, group sizes can range from 3 to 6 students. The expectations and assessment of your project will obviously depend on the group size.

2 Project background and motivation

The project is inspired by the recent developments in mobile robots for hospital use. Hospitals tend to have a very high number of transportation tasks to be carried out: transportation of beds, medicine, blood samples, medical equipment, food, garbage, mail, etc. Letting robots carry out these transportation tasks can save significantly on hospital staff resources.

The most successful implementation of hospital robots so far are the TUG robots by the company Aethon (http://www.aethon.com), see Figures 1–2. TUG robots were first employed in a hospital in 2004, and is now in use in more than 100 hospitals in the US. An earlier hospital robot was the HelpMate robot developed in the late 1990s, see Figure 3.

In Denmark, there has been several recent research projects aiming at constructing mobile robots for carrying out transportation tasks at Bispebjerg Hospital near Copenhagen. One of the robot prototypes from these projects is shown in Figure 4. A tunnel system in the basement of Bispebjerg Hospital connects all buildings and wards (Figure 5). Currently, most transportation tasks are carried out by human hospital porters driving electrical trucks in the tunnel system (Figure 6). It is the work of these porters that the hospital wishes to be taken over by mobile robots in the future.

Figure 1: The TUG robot

Figure 2: The TUG robot tugging a container

Figure 3: The HelpMate robot in a hospital environment

Figure 4: The Nestor Robot at Bispebjerg Hospital

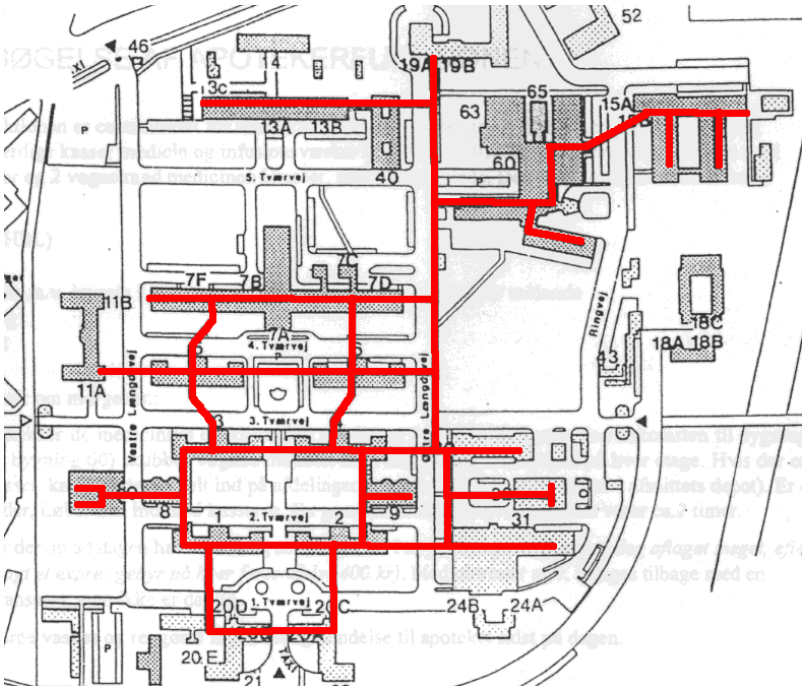


Figure 5: Tunnel system at Bispebjerg Hospital

Figure 6: A hospital porter at Bispebjerg Hospital

The ideal is to have a system of multiple mobile robots that can themselves distribute the transportation tasks between them in an efficient way, essentially making it into a multi-agent system. This is more advanced than the TUG robots, where each individual robot is manually assigned its individual tasks. Furthermore, ideally the robots should not only be able to move a single type of container as the TUG robot is, but many different types of physical objects, including hospital beds. Possibly, the easiest way to achieve this is to have different types of robots with different physical design and different abilities concerning which objects they can move. Some robots might e.g. be able to move very quickly with small items; others might be bigger and slower but able to move very large and heavy items like hospital beds; yet others might be both small and slow, but able to handle fragile items safely.

The goal of this programming project is to implement a much simplified simulation of how a multi-robot system at Bispebjerg Hospital might work.

3 Levels

The environment will be represented by grid-based structures, called *levels*. A level contains *walls*, *agents*, *boxes*, *goal cells* and possibly a number of *unknown items*. The walls are used to represent the physical layout of the environment, e.g. the corridors in the basement of Bispebjerg Hospital. The agents represent the hospital robots. The boxes represent the items that the robots have to move. Each item has to be moved to one of the corresponding goal cells. The unknown items represent objects that are not part of the agents' initial maps of the world, e.g. boxes, trash or beds placed by humans but not scheduled for transportation. Unknown items can not be moved.

A level can be represented textually, similarly to Sokoban levels. Levels are constructed according to the following conventions:

- *Walls.* Wall items are represented by the symbol `+`.
- *Agents.* Agents are represented by the numbers `0`, `1`, ..., `9`. Agent numbers are unique, 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.
- *Boxes.* Boxes are represented by capital letters `A`, `B`, ..., `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

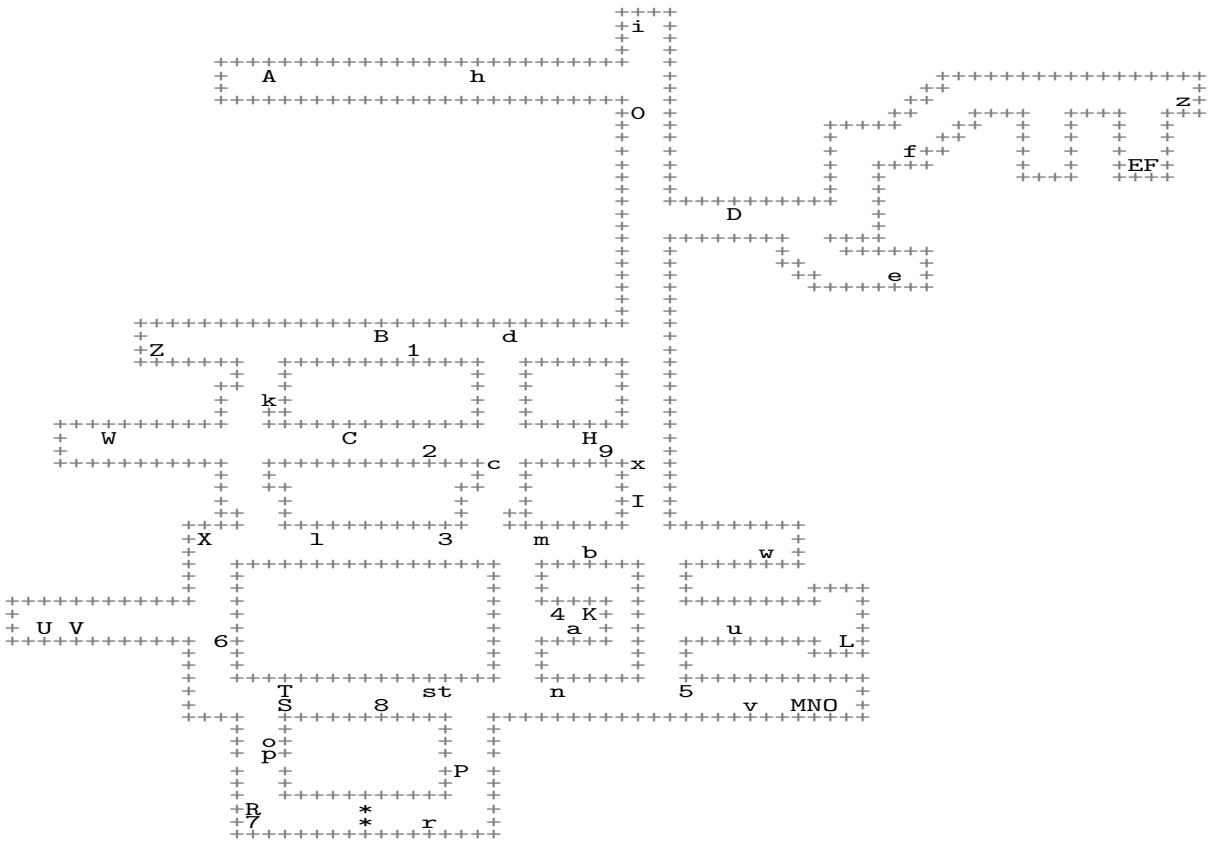


Figure 7: The Bispebjerg tunnel system modelled as a grid

type, that is, having the same letter.

- *Goals cells.* Goal cells are represented by small letters `a`, `b`, ..., `z`. Again, the letter is used to denote the *type* of the goal cell. Boxes have to be moved to goal cells of the same type, that is, having the same letter. Thus, a box with letter `A` has be to moved to one of the goal cells with letter `a`, etc.

- *Unknown items.* Unknown items are represented by the symbol `*`.

Figure 7 shows a level based on a grid version of the tunnel system at Bispebjerg Hospital. In the figure, passage through the bottom corridor has been blocked by two unknown items.

It might seem as an over-simplification to model the tunnel system as a grid, but in fact environments of mobile robots are usually modelled exactly this way, although usually by much more fine-grained grids (the robots currently experimented with at Bispebjerg Hospital model the tunnel system by a grid in which each cell represents a $10cm \times 10cm$ square).

To allow different agents to have 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. The idea here is that we can model scenarios where the different robots have different abilities concerning which objects they can handle. 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 *allowable colors* for agents and boxes are:

blue, red, green, cyan, magenta, orange, pink, yellow.

To represent the colors of agents and boxes as part of the textual level representation, each such representation is prepended by a *color declaration.* A color declaration is of the form

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

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

where each `<color>` is any allowable color, and each `<object>` is either the name of a box type (`A`, ..., `Z`) or the name of an agent (`0`, ..., `9`). Note that boxes of the same type always gets the same color. Each color, agent (number) and box type (capital letter) must occur at most once in the color declaration. Any agent or box type not mentioned in the color declaration will be given the *default color*, blue.

Example. The following is a full textual description of a simple level with two agents:

```
red: 0,A
green: 1,B
++++++++++++
+          0+a+
+ ++++++++A+ +
+           +
+B++++++++  +
+b+1        +
++++++++++++
```

In this level, agent `0` can move box `A` but not `B`, and vice versa for agent `1`.

Files containing textual representations of level are given the extension `.lvl`. The file sharing folder `Programming Project` on CampusNet contains a few such files with example levels. You are expected to construct additional levels yourself during the project.

4 Actions

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

1) *Move action.* A move action is represented on the form

$$Move(\textit{move-dir-agent}),$$

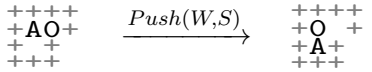
where *move-dir-agent* is one of N (north), W (west), S (south), or E (east). $Move(N)$ means to move one cell to the north of the current location. For a move action to be successful, the following must be the case:

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

2) *Push action.* A push action is represented on the 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 seen to be "pushed around the corner." For a push action to be successful, the following must be the case:

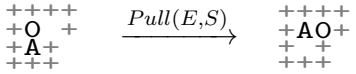
- The neighbouring cell of the agent in direction *move-dir-agent* contains a box β of the same color as the agent.
- The neighbouring cell of β in direction *move-dir-box* is currently free.

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

3) *Pull action.* A pull action is represented on the 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 (as seen from 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. Note that the first condition above ensures that it is not possible for an agent and a box to swap positions by simply performing an action like e.g. $Pull(S, S)$.

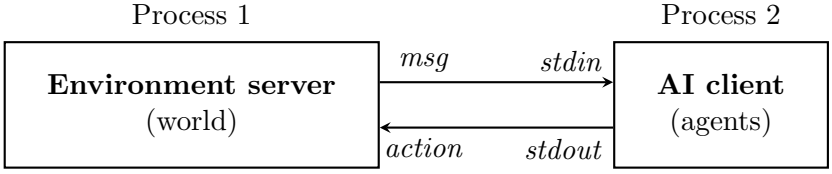
4) *No-op action.* The action *NoOp* represents the persistence action (do nothing). The No-op action is always successful.

If an agent tries to execute an action without the conditions for it to be successful being satisfied, the action will fail. Failure corresponds to performing a no-op action, that is, 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 is inhabited by several agents, these agents can perform simultaneous actions. The actions of the individual agents are assumed to be completely synchronised, hence we can *joint actions* of the form $[a_0, \ldots, a_n]$. In a joint action $[a_0, \ldots, a_n]$, a_0 is the action performed by agent `0`, a_1 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. If this happens, only one of the involved agents will be successful in performing its action. Which agent will be successful is decided non-deterministically.

5 Environment server

To simulate the environment, an environment server is available on CampusNet. It is in the archive file `environment.zip` in the file sharing folder "Mandatory 3". The server represents the actual state of the world, and agents interact with the environment by communicating with the server. All agents are represented by a single client program, *which you must implement*. The client communicates with the server through the standard streams *standard in* and *standard out* (`System.in` and `System.out` in Java). The client program can thus be implemented in any language of your choice. The interaction between server and client is depicted below:



The protocol for communicating with the server is specified by the following steps:

```
<SERVER>
1 +++++
2 +0Aa+
3 +++++
4
5 -
6 [false]
7 -
8 [true]
```

```
<CLIENT>
1 -
2 -
3 -
4 -
5 [Move(E)]
6 -
7 [Push(E,E)]
8 -
```

Figure 8: Example of interaction between server and client

1) The client is sent a description of the environment, as described in section 3, except unknown items (`*`) are replaced by empty squares (the client does thus not initially know where the unknown items are).

2) The client sends a joint action to the server $[a_0, ..., a_n]$, where a_i is the action of agent i. A valid joint action could be `[Move(N), Push(S,W), NoOp, Pull(E,N)]` for an environment with four agents. See the description of the possible actions in Section 4.

3) The server replies with a list $[p_0, p_1, \ldots, p_n]$, where each p_i is either `true` or `false` according to whether the action of agent i has been successful or not. This reply can be used to figure out the positions of the unknown items in a level, e.g. if a move into a cell not occupied by any walls, boxes or other agents fail.

4) Go to step 2.

Figure 8 illustrates a complete interaction between a server and client. The left and right columns show what the server and client sends, respectively. The symbol '`-`' denotes that the process is waiting for input.

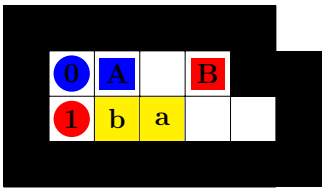
Once a sequence of joint actions has lead to all goal cells being occupied by boxes of the correct types, the server will write "success" to its *stdout* stream (presumably the shell). If a timeout is set and this value is exceeded, the server will instead write "timeout" and terminate the client (similar to the competition setting, see section 8). Violation of the communication protocol or the occurrence of unrecoverable errors will similarly terminate the server and provide a brief error message. As *standard in* and *standard out* of the client process are used for communication with the server, your implementation must use the *standard error* stream for writing to the shell.

The archive on CampusNet containing the environment server also contains various example clients and levels to try out. The `README` file contained in the archive contains many examples on how to invoke the server and covers all its functionality. The source files of the example clients are commented to provide further hints about implementation, and you may base your own implementation on these.

Example. Consider the following level:

```
blue: 0,A
red: 1,B
++++++
+0A B++
+1ba  +
+++++++
```

If the -g option is specified (see README) in the environment server this level will be presented graphically as follows:



As seen, agents are represented by circles and boxes by squares. Walls are black and goal cells are yellow. Unknown items are grey (none in this level). Figure 9 shows a sequence of joint actions in this level and the corresponding sequence of states. Goal cells turn green when they become occupied by boxes of the correct type.

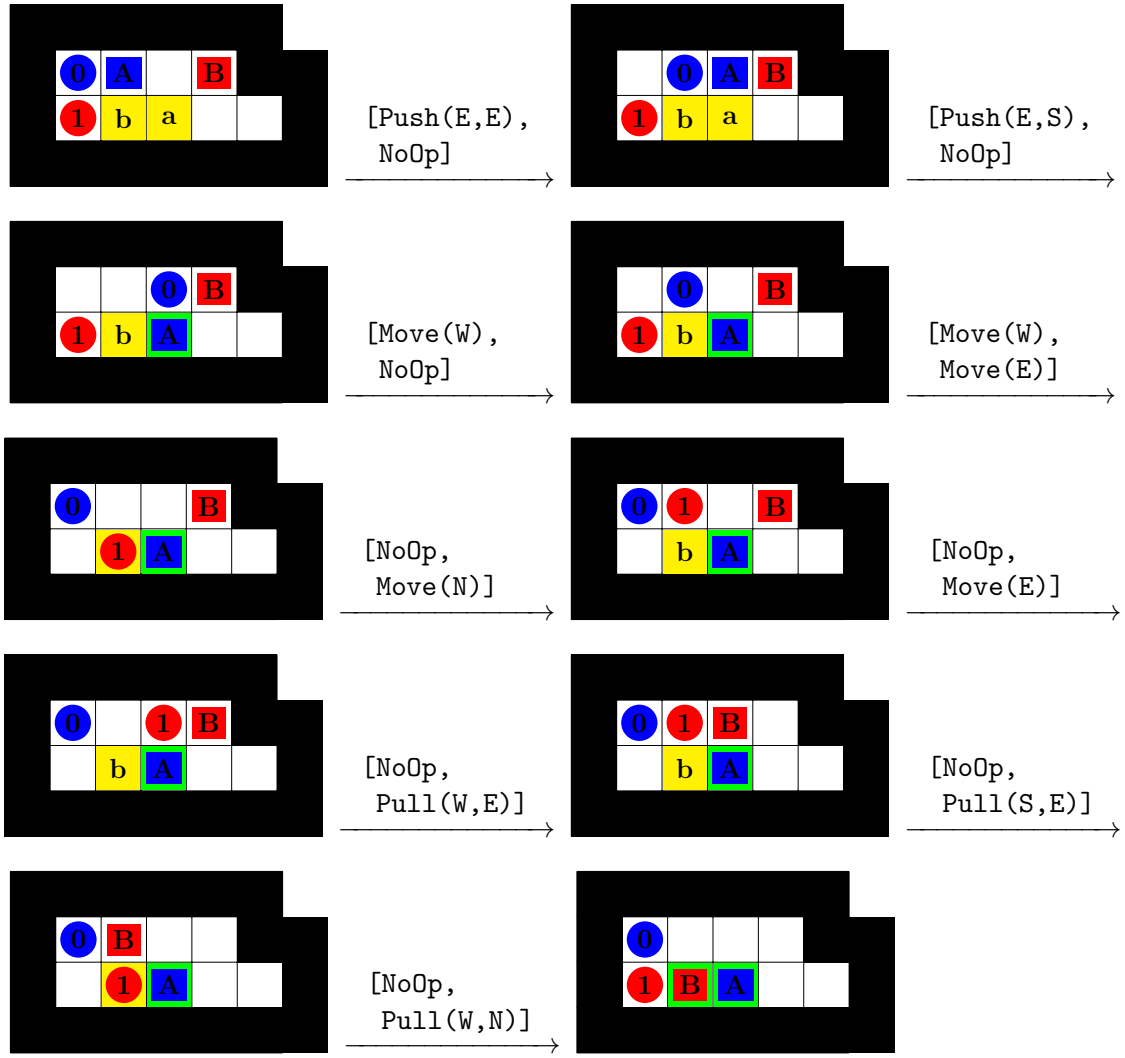


Figure 9: A sequence of joint actions and the corresponding states

6 Goal of the project

The goal of the present programming project is to implement an AI client that can complete arbitrary levels. *Completing* a level means to perform a sequence of joint actions that will result in all goal cells being occupied by boxes of the correct type (same letter). Note that this does not necessarily imply that all boxes end up in goal cells, as there might be more boxes than goal cells (e.g. more clean beds available than the number needed to be transported to the different wards). We will only consider levels that *can* actually be completed by *some* sequence of joint actions. In the example given in Figure 9, the

agents do not only complete the level, but also return to their start positions afterwards. Returning to the start positions is not a required part of completing a level. You are free to implement your AI client in whatever programming language you prefer.

7 Tracks

Since levels can contain multiple agents, the problem environment is inherently a *multiactor environment*. Since levels can contain unknown items, the problem environment is also a *partially observable environment*. Simplified problems can be obtained by either restricting to single-agent levels and/or levels without unknown items. Thus, the problem naturally divides into the following 3 *tracks* in order of increasing complexity:

- *FOSA track (Full Observability, Single-Agent track).* In this track, only levels with a single agent and no unknown items are considered. That is, there is assumed to be only one agent, agent `0`, and no `*`s. This track is the simplest and is subsumed by the two other tracks.
- *FOMA track (Full Observability, Multi-Agent track).* In this track, multiple agents are allowed, but no unknown items.
- *POMA track (Partial Observability, Multi-Agent track).* In this track, all possible levels are allowed. This track is the most complex and it subsumes the two other tracks.

In the programming project, you are required to provide a solution to at least the first two tracks. Large groups should aim at all 3 tracks.

8 Competition

In the exam period there will be a competition where your different solutions will compete against each other in each of the tracks that you have provided a solution to. In each track, you will be given a number of levels belonging to that track, and your implemented AI client should then try to complete each level as fast as possible and using as few (joint) actions as possible. In the competition, the environment server will time out after 5 minutes, so your AI only has 5 minutes to complete each level. Furthermore, your client can use at most 20.000 joint actions to solve the level.

Your client will get two scores for each level, an *action score* and a *time score*. These are both numbers between 0 and 1 with 1 being the best. If a level is not solved within the time and action limits, both scores will be 0. Otherwise, the scores are calculated as follows:

$$\text{action score of your client} = \frac{\text{fewest number of joint actions among all successful clients}}{\text{number of joint actions used by your client}}$$

$$\text{time score of your client} = \frac{\text{time spent by the fastest among all successful clients}}{\text{time spent by your client}}$$

For each track that you have provided a client for, both your action and time scores for the individual levels in the track will be summed up, and for each track two winners will be announced: one for having the best action score, and one of having the best time score.

Each group has to design and submit one level within each of the tracks that they provide a solution for. The submitted levels will then form the basis for the set of competition levels. These levels are required to be of size at most 70x70 grid cells. Note that you can only score better on your own level than the competing clients if the level is:

- Simple enough for your own client to be able to solve it within the given time and action limits.
- Complex enough to give challenge to the clients of the other groups.

Note that a level doesn't necessarily have to be of a big size in order to be challenging, as the greatest challenge lies in combinatorial complexity.

Each group has to choose a group name of at most 9 letters in the range `[a-zA-Z]`, e.g. `DeepGreen` or `MASochist`. When you have formed your group of 3 to 6 students and chosen your name, you should use the assignment "Group Registration" on CampusNet to upload a text file `group.txt` of the following form:

```
<grpname> <studid1>
<grpname> <studid2>
...
<grpname> <studidn>
```

where `<grpname>` is the name you have chosen for your group, and `studid1` to `studidn` are the student numbers of the group members (of the form `sNNNNNN`). For instance, the 4 person group `DeepGreen` from the future would upload a file named `group.txt` with contents:

```
DeepGreen s159111
DeepGreen s161112
DeepGreen s151448
DeepGreen s140304
```

Schedule

Tuesday 27 March at 13.00. Deadline for the registration of groups. Use the assignment "Group Registration" on CampusNet. Upload the file `group.txt` as described above.

Friday 24 May at 13.00. Deadline for uploading your competition levels. Use the assignment "Competition Levels" on CampusNet. Upload one level for each track you are signing up to (2–3 in total). Your files containing the levels should have the following format:

```
<track type><grpname>.lvl
```

Here `<track type>` is one of the four-letter combinations FOSA, FOMA or POMA, depending on which track the level belongs to. `<grpname>` is the same name as you used when registering your group. An example of a valid file name could be:

```
FOSADeepGreen.lvl
```

Don't zip the files, but upload either 2-3 plain text files containing the levels. Remember that you levels can be at most 70x70 grid cells large.

Tuesday 28 May at 13.00. The set of competition levels will be made available on CampusNet. At the same time, we will make a special "competition version" of the server available to be used to test your client on the competition levels. The competition server will be able in one run to test your client on all the competition levels and output the solutions to an encrypted file, `output.out`.

Friday 31 May at 13.00. Deadline for uploading the result, `output.out`, of running the competition server on the competition levels using your client. Use the assignment "Competition Solutions" on CampusNet. Note that only one file should be uploaded.

Monday 3 June at 13.00. Presentation of competition results. We will present the detailed scores of all groups, announce the winners, and show playbacks of some of the best solutions. This is the official day of examination in the course, so everybody should be able to participate. Each group should be prepared to give a short explanation of the behaviour of their client and the general ideas underlying their solution. Location is yet to be decided.

The goal of the competition is to motivate you to make the most efficient possible solution, and to allow comparison of the strengths and weaknesses of the various solution. Note, however, that your result in the competition will not affect your grade. Also, a group of 6 people obviously has a lot more resources for making an efficient solution than a group of 3.

9 Report

In addition to implementing your AI client, you should write a short report prepared in the style of a conference article. The report should be at most 15 pages long. It should focus on your problem analysis, your solution, and an analysis of your findings and results. The reader of the report is expected to be familiar with the project description given in this document. The reader is also expected to be familiar with the entire curriculum of the course, so there is no need to repeat *any* of that in the report. If you expect the reader to be familiar with additional work in the area, please make the necessary references. Also make sure to reference any specific work that you might have used in completing the project (books, articles, implementations, online resources, etc.).

Your report should describe your ideas, solutions, and original contributions. You should describe your implemented system in terms of its overall algorithmic functionality, not in terms of implementation details. Your report should make clear what methods within AI and MAS you have been implementing, using the correct computer science nomenclature. Finally, make to sure reflect on strengths and weaknesses of your final solution.

It is very important that your report is clear about who has done what in the project, and who has written which parts of the report. From http://shb.dtu.dk/default.aspx?documentid=2811&Language=en-GB&lg=&version=2011%2f2012:

> Group projects with both common and individual contributions. Common sections and individual contributions shall be indicated in the project. This can either be as pages or sections for which each student is responsible, or, it can be by a description of how each member of the group has contributed, e.g. by collecting data, experimental work, searching for information, model building, programing, analysis, editing etc.

The completed project should be submitted electronically via CampusNet using the assignment "Mandatory Assignment 3 Final Handin". The deadline for submission is:

Friday 7 June at 13.00.

Each group makes a single submission consisting of the following:

1) A **pdf file** containing the report. Please put the following information on the front page of your report:
   - Course number (02285), course name (AI and MAS), and date.
   - Name of the group.
   - Study numbers and full names of all group members.
2) Source code and executable code of your implemented software.

All these files should be archived in a single zip-file named `project02285.zip` to be uploaded to CampusNet (no `.rar`, `.tar`, `.gz` or other formats, please!).

10 Expected workload

As previously announced, the programming project constitutes approximately 3.75 of the 7.5 ECTS in the course. This means that the expected workload of the programming project is approximately 100 hours per student.

Good luck with your project. Be creative! Have fun!