02285 AI and MAS

Mandatory Assignment 3

Due: Tuesday 2 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 (aka Mandatory Assignment 3). The project is designed to be very open-ended and flexible, allowing many different group sizes, types of solutions 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. Group sizes can range from 3 to 5 students. The expectations and assessment of your project will obviously depend on the group size.

2 Project background and motivation

The project is partly inspired by the 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. In 2012–2013, TUG

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.

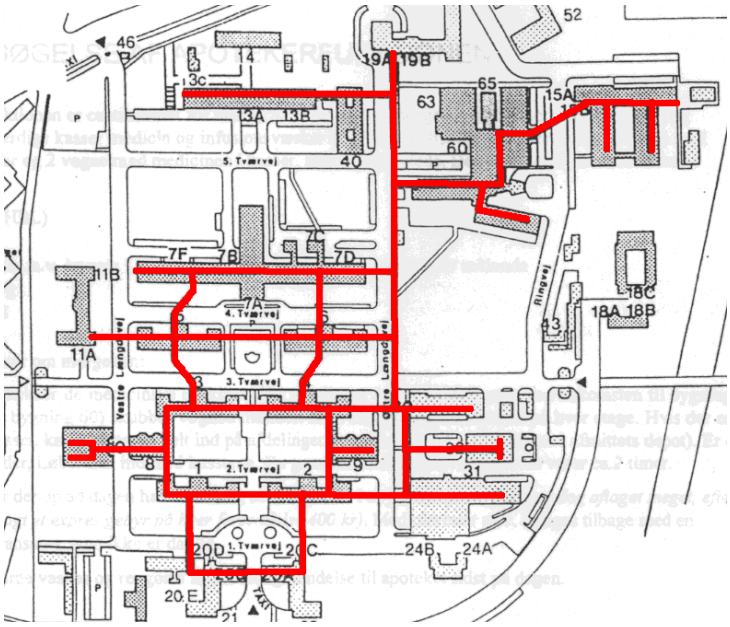


Figure 5: Tunnel system at Bispebjerg Hospital

Figure 6: A hospital porter at Bispebjerg Hospital.

robots were also tested and used at a Danish hospital, Sygehus Snderjylland, the first hospital in Europe to employ them. An earlier hospital robot was the HelpMate robot developed in the late 1990s, see Figure 3.

In Denmark, there has been several research projects aiming at constructing custom 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.

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 (or elsewhere) might work. It is essentially a *toy version* of a real multi-robot system for transportation tasks. Some of the challenges in this project are shared with the challenges of creating a real multi-robot system, but of course there would be many more challenges in creating a real such system.

3 Levels

The environment will be represented by grid-based structures, called *levels*. A level contains *walls*, *agents*, *boxes* and *goal cells*. 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.

A level can be represented textually, making it easy to design levels using any text editor (using a fixed-width font!) and saving them as pure text files. 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 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.

Figure 7 shows a level based on a grid version of the tunnel system at Bispebjerg Hospital. It might seem as an over-simplification to model the tunnel system as a grid, but in fact environments of mobile robots are often modelled exactly this way, although usually by much more fine-grained grids (some robots experimented with at Bispebjerg Hospital modelled the tunnel system by a grid in which each cell represented 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>
<color>: <object>, <object>, ..., <object>
...
<color>: <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 (same letter)

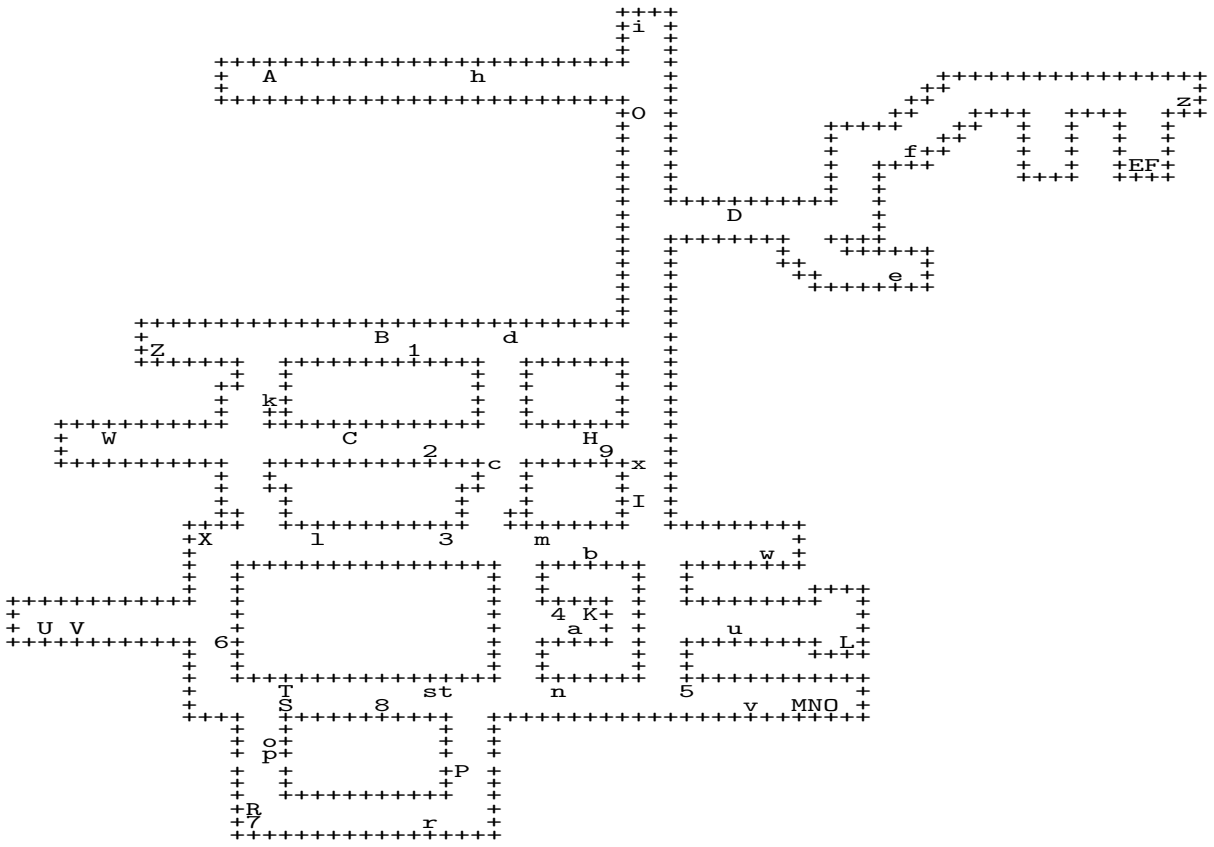


Figure 7: The Bispebjerg tunnel system modelled as a grid

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 levels are given the extension `.lvl`. The file sharing folder `Mandatory Assignment 3` on CampusNet contains a few such files with example levels (in `environment.zip`). 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 or a box. A cell is called *free* if it is not occupied. Each agent can perform the following actions:

a) *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.

b) *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:

```
++++          Push(W,S)     ++++
+AO+      ---------------->  +O +
+ +                          +A+
+++                          +++
```

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. $Push(W, E)$.

c) *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:

```
++++          Pull(E,S)     ++++
+O +      ---------------->  +AO+
+A+                          + +
+++                          +++
```

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. $Pull(S, S)$.

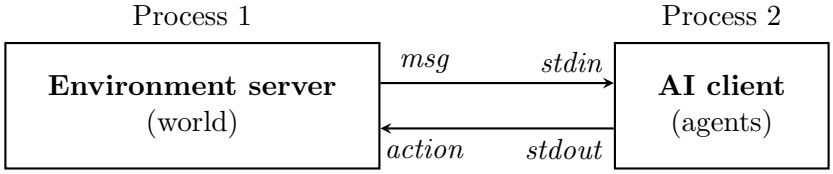
d) *No-op action.* The action $NoOp$ represents the persistence action (do nothing). The No-op action is always successful.

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 consider *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 Assignment 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:

1. The client is sent a description of the environment, as described in section 3.

```
<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

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.

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.

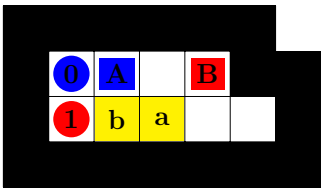
Once a sequence of joint actions has led to all goal cells being occupied by boxes of the correct types, the server will write `success` to its *stdout* stream (presumably the shell). If a timeout is set and this value is exceeded, the server will instead write `timeout` and terminate the client (similar 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. 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.

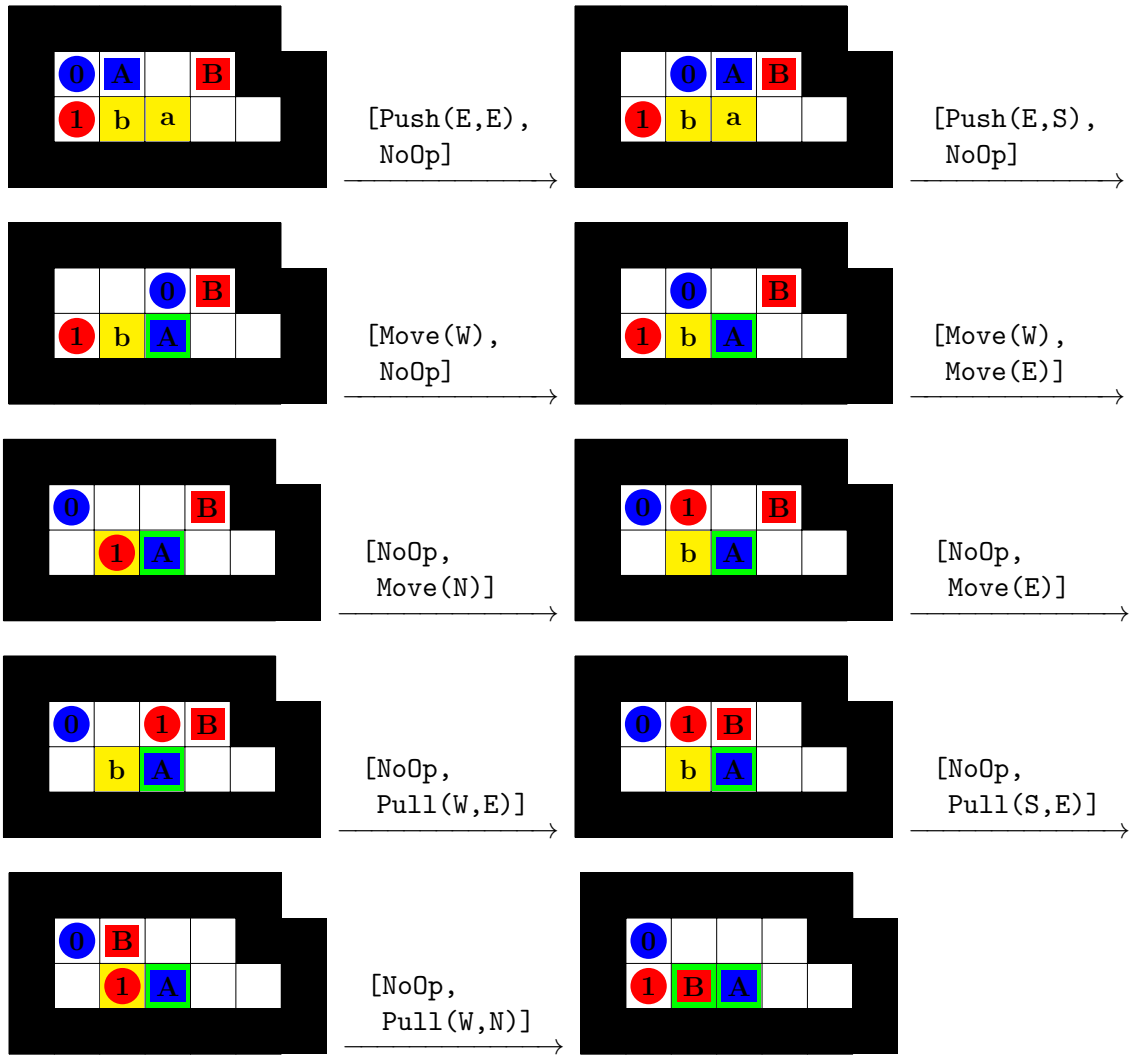


Figure 9: A sequence of joint actions and the corresponding states

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*. A natural simplification is the single-agent version where only single-agent levels are considered. Thus, the problem naturally divides into the following 2 *tracks*:

- *SA track (Single-Agent track).* In this track, only levels with a single agent are considered, that is, there is assumed to be only one agent, agent `0`.
- *MA track (Multi-Agent track).* In this track, multiple agents are allowed.

In the programming project, you are required to provide a solution to both tracks.

8 Competition

In the exam period there will be a competition where your different solutions will compete against each other. In each track (SA and MA), 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 (SA and MA), 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. So there will be 4 winners in total: 1) best time score in SA track; 2) best action score in SA track; 3) best time score in MA track; 4) best action score in MA track.

Each group has to design and submit one SA and one MA level. The submitted levels will then be considered for inclusion in the set of competition levels. These levels are required to be of size at most 70x70 grid cells. Note that a level doesn't necessarily have to be of big size in order to be challenging, as the greatest challenge lies in combinatorial complexity. Please only submit levels that your own client can solve within the time and action limit. The final competition levels will be a combination of selected levels submitted by the students and levels designed by the teachers. Levels will be selected to ensure variation in difficulty and features.

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 5 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, a 4 person group DeepGreen consisting of students with student numbers s129111, s131112, s091448 and s980304 would upload a file named `group.txt` with contents:

```
DeepGreen s129111
DeepGreen s131112
DeepGreen s091448
DeepGreen s980304
```

Schedule

Tuesday 10 March at 17.00. Deadline for the registration of groups. Use the assignment "Group Registration" on CampusNet. Upload the file `group.txt` as described above.

Wednesday 13 May at 10.00. Deadline for uploading your competition levels. Use the assignment "Competition Levels" on CampusNet. Upload one level for each track (2 in total). Your files containing the levels should have the following format:

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

Here `<track type>` is one of the two-letter combinations SA or MA, 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:

```
SADeepGreen.lvl
```

Don't zip the files, but upload 2 plain text files containing the levels. Remember that your levels can be at most 70x70 grid cells large. Levels are not allowed to contain tab characters, and the last line of a level should be followed by a newline. Levels that are incorrectly named or incorrectly formatted will not be considered for inclusion in the competition levels.

Monday 18 May at 8.00. 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`.

Tuesday 19 May at 20.00. 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.

Thursday 28 May 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 is required to participate. 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 submitted solutions. Note, however, that your result in the competition will not directly affect your grade. A group might for instance challenge itself by trying out advanced novel techniques or focus mostly on optimising a certain aspect of the solution, e.g. multi-agent communication and coordination. That group might not be able to produce the most efficient solution, but the novelty and quality of both solution and report could still be very high. Furthermore, a group of 5 people obviously has 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.

In addition to the report, it is very important that you include a separate document containing a *group declaration*. The group declaration should give a detailed specification of who did what in the project, and who has written which parts of the report. From the Study Handbook 2014 edition:

> **Group project with individual contributions stated**. Several students can submit individual projects on a common theme, contributing with their individual

section to a common report. Provided that the individual contributions are clearly marked in the common report, a subsequent individual oral examination is not required. The students contribution to the report can in itself be graded.

The completed project should be submitted electronically via CampusNet using the assignment "Mandatory Assignment 3". The deadline for submission is on the last day of the regular exam period:

Tuesday 2 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) A **pdf file** containing the group declaration, that is, a detailed description of who did what in terms of ideas, programming, literature search, report writing, etc. This is a very important document!
3) 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

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!