02285 AI and MAS, SP2020

Programming Project

Due: Friday 5 June at 20.00

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

1 Introduction

This document describes the main programming project of the course. 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.

Among the most successful and widely used 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 robots were also tested and used at a Danish hospital, Sygehus

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.

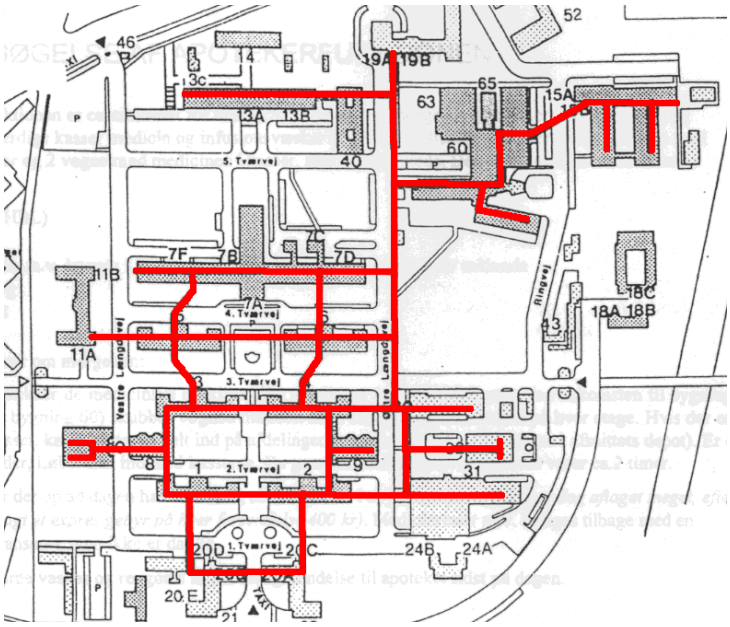


Figure 5: Tunnel system at Bispebjerg Hospital

Figure 6: A hospital porter at Bispebjerg Hospital.

Sønderjylland, 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, physical multi-robot system, but of course there would be many more challenges in creating a physical 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.

A level can be represented textually, making it easy to design levels using any decent text editor (with a monospaced font) and saving them as ASCII-encoded text files. Each level file has the following format:

```
#domain
hospital
#levelname
<name>
#colors
<colors>
#initial
<initial>
#goal
<goal>
#end
```

The items in diamond brackets (e.g. `<name>`) are placeholders for content described below.

The first two lines indicate the domain of the problem, which for this project is the hospital domain, and occur verbatim in level files for the project. Those two lines *must be terminated only with a line-feed character* (ASCII value 10)! The remaining lines can be terminated by either carriage-return and line-feed (CRLF) or just line-feed (LF).

The `<name>` field is replaced by the level's name. By convention, single-agent levels begin with `SA`, and multi-agent levels with `MA`.

Levels are constructed with initial and goal states using the following conventions:

- *Free cells.* Free cells are represented by spaces (ASCII value 32).
- *Walls.* Walls are represented by the symbol `+`.
- *Agents.* Agents are represented by the numbers `0`, `1`, ..., `9`, with each number identifying a unique agent, 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 (i.e. same letter) in a level.

The `<initial>` and `<goal>` specifications consist of lines with these symbols as a top-down map, which defines the initial state and goal states of the planning problem respectively. The two specifications must have exactly matching configurations of walls, and can consist of at most $2^{15}-1$ rows each of at most $2^{15}-1$ columns. Note that for the competition, *this is further limited to* 50 *rows and* 50 *columns*! It is required that every agent and box in a level are in areas entirely enclosed by walls. Each symbol in the initial state specifies that a corresponding

object starts in that position, and each symbol in the goal state specifies that for the level to be solved, an object of the symbol's type must occupy that cell. The goal states of the planning problem are then all states which have objects in the configuration shown in `<goal>`, where excess objects can be anywhere (i.e. not every agent or box necessarily has to have a goal cell they must reach to solve a level).

To allow modelling of different agents having 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. 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 *allowed colors* for agents and boxes are:

blue, red, cyan, purple, green, orange, pink, grey, lightblue, brown.

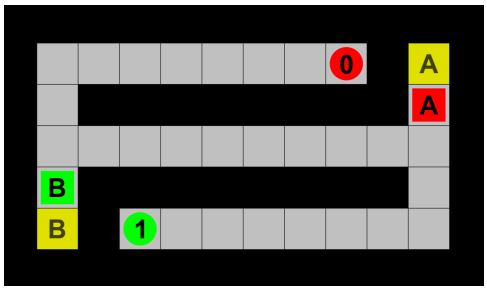
To represent the colors of agents and boxes as part of the textual level representation, each level has the colors section (`#colors`). The color declaration `<colors>` is of the form

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

where each `<color>` is an allowed color, and each `<object>` is either the name of a box type (`A`, ..., `Z`) or the name of an agent (`0`, ..., `9`). Note that this specification forces all boxes of the same type to have the same color (e.g. there can't be both a blue and a red `A` box; all `A` boxes must have the same color). Each agent and box type may occur at most once in the color declaration, and all agents and box types that occur in the initial state must explicitly have a color specified.

Example. The following is a full textual representation (left) of a simple level with two agents, accompanied by its graphical visualisation (right):

```
#domain
hospital
#levelname
MAExample
#colors
red: 0, A
green: 1, B
#initial
++++++++++++
+          0+ +
+ +++++++++A+
+            +
+B++++++++  +
+ +1         +
++++++++++++
#goal
++++++++++++
+          +A+
+ ++++++++  +
+           +
+ ++++++++  +
+B+         +
++++++++++++
#end
```



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

4 Actions

A grid cell in a level is called *occupied* if it contains either a wall, an agent or a box. 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 in the textual form

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

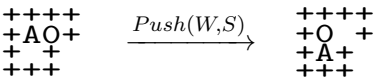
   where *move-dir-agent* is one of N (north), W (west), S (south), or E (east). For instance, $Move(N)$ means an agent attempts to move one cell to the north of its current location. For a move action to be successful, the following must be the case:

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

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

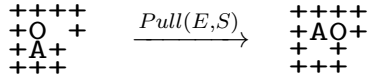
   - The neighbouring cell of the agent in direction *move-dir-agent* contains a box β of the same color as the agent.
   - The neighbouring cell of β in direction *move-dir-box* is currently free.

   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 in the textual 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 is textually represented as

$$NoOp$$

and represents an agent doing nothing. The No-op action is always successful.

If an agent attempts to execute an action that does not satisfy the conditions for being successful, the action will fail. Failure corresponds to performing a no-op action, i.e. 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 has 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*, which have the textual representation

`<action0>`;`<action1>`;...;`<action9>`

In a joint action, `<action0>` is the action performed by agent `0`, `<action1>` 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, or if two agents attempt to move the same box. If this happens, then neither agent will succeed in their action, and both agents perform a $NoOp$ instead.

5 Server

To simulate the environment, an environment server (`server.jar`) is provided. The server loads a levels and tracks the actual state of the world, and agents interact with the environment by communicating with the server. All agents are controlled by a single client program, *which you will implement*. The client communicates with the server through the standard streams *stdin*, *stdout*, and *stderr*. The client program can thus be implemented in any language of your choice which can read from and write to these streams.

The server and client use a text-based protocol over the standard streams to communicate. The protocol text is ASCII encoded, and proceeds as follows:

1. The client sends its name to the server, terminated by a newline (CRLF or LF).
2. The server sends the contents of the level file to the client, exactly as it occurs byte-for-byte[1].

[1] With the addition of a final LF if the level file is not properly terminated by a newline.

Server		Client
	1	`ExampleClient`
`#domain`	2	
`hospital`	3	
`#levelname`	4	
`SAExample`	5	
`#colors`	6	
`blue: 0, A`	7	
`#initial`	8	
`+++++`	9	
`+0A +`	10	
`+++++`	11	
`#goal`	12	
`+++++`	13	
`+0 A+`	14	
`+++++`	15	
	16	`Move(E)`
`false`	17	
	18	`Push(E,E)`
`true`	19	
	20	`Move(W)`
`true`	21	

Table 1: Example of interaction between server and client.

3. The client sends the server either a joint action (specified above) or a comment (which is any string starting with a hashtag symbol (`#`)). The line is terminated by a newline.

4. If the client's message was a comment, then the server prints this message to its own *stdout*.

5. If the client's message was a joint action, then the server simulates the action and sends back a line of the form

   `<success0>;<success1>;...;<success9>`

   where each `<successN>` is either `true` or `false` indicating whether that agent's action succeeded or failed respectively.

6. Steps 3-5 are repeated until the client shuts down, or the server terminates the client.

The client receives the messages from the server on its *stdin*, and sends its own messages to the server on its *stdout*. Anything the client writes on its *stderr* is directly redirected to whatever the server's own *stderr* is connected to (typically the terminal).

Table 1 illustrates a complete interaction between a server and client. The left and right columns show what the server and client sends, respectively. The given exchange will lead to the level being solved.

After the client shuts down or is terminated, the server will write a brief summary of the result to its own *stdout*.

For details on different modes and options for the server, run the server with the `-h` argument:

```
java -jar server.jar -h
```

6 Goal of the project

The goal of the 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 or agents of the corresponding types. We will only consider levels that *can* actually be completed by *some* sequence of joint actions. You are free to implement your AI client in whatever programming language you prefer.

We ***strongly suggest*** that you start from scratch with your own architecture, rather than building on top of something like the search client from the warmup.

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 two *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 client which can handle levels in both tracks.

8 Competition

Towards the end of the course there will be a competition where your different clients 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 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 clients out after 3 minutes, so your AI only has 3 minutes to complete each level. Furthermore, your client can use at most 20.000 joint actions to solve a 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:

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

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

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 be included in the set of competition levels. These levels are required to be of size at most 50x50 grid cells. Note that a level does not necessarily have to be of big size in order to be challenging, as the greatest challenge lies in combinatorial complexity. You should only submit levels that your own client can solve within the time and action limits. *Any level submitted for the competition that the group's competition client cannot itself solve will automatically be excluded from the competition results!* The final competition levels will be a combination the levels submitted by the students and levels designed by the teachers. Levels will be selected to ensure variation in difficulty and features.

Schedule

Thursday 12 March at 20.00. Deadline for the registration of groups, that is, for submitting the `<group_name>.txt` file. Use the assignment "Group Registration" on DTU Inside.

Monday 18 May at 20.00. Deadline for submitting competition levels. Submit one level for each track (2 in total). The level file names must have the following format:

```
<track_type><group_name>.lvl
```

Here `<track_type>` is one of the two-letter combinations SA or MA, depending on which track the level belongs to. `<group_name>` is the same name as you used when registering your group. Examples of valid file names could be:

```
SADeepGreen.lvl
MADeepGreen.lvl
```

Do not zip the files, but submit 2 plain text files containing the levels. Remember that your levels can be at most 50x50 grid cells large. Levels that are incorrectly named or incorrectly formatted will not be considered for inclusion in the competition levels. Use the assignment "Submission of Competition Levels" on DTU Inside for the submission.

You can still improve your implementation after the submission of your competition levels, but you need to have a working and almost final implementation by this date in order to be able to submit relevant levels (that you know your client solves). You can afterwards improve your implementation until the release of the competition levels.

Thursday 21 May at 08.00. The set of competition levels will be made available, along with instructions for running the server on the competition levels.

Friday 22 May at 20.00. Deadline for submitting the result of running the server on the competition levels using your client. Use the assignment "Submission of Competition Solutions" on DTU Inside.

Tuesday 26 May at 14.00. 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. Each group should be prepared to give a short explanation of the behaviour of their client and the general ideas underlying their solution. 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 report prepared in the style of a conference article. Detailed guidelines on the report are in `guide_for_report.pdf`.

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 "Exam forms" section of the Structure and Rules on DTU Inside (quoted in September 2018):

> **Group projects with individualization**. Several students can contribute individual sections to a joint report. Provided that the students' individual contributions are clearly distinguishable in the joint report, a subsequent oral exam is not required. It is accepted that general descriptive sections such as the introduction and the conclusion are prepared jointly. However, the most important sections in a group project must be individualized.

The completed project should be submitted electronically via DTU Inside using the assignment "Programming Project". The deadline for submission is:

Friday 5 June at 20.00.

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 of your implemented software.

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

You have the time from submitting your competition results to the deadline for handing in the report to focus on report writing. You are also allowed to improve your implementation in this period, for instance if you get new ideas after seeing the competition results. But note that the quality of the report is very important, so make sure not to get caught up in last minute improvements of the client rather than spending time on writing a nice, clear and well-structured report.

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!