02285 AI and MAS
Warmup Assignment
Due: Monday 22 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 and more on the last exercise that is more open). 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 the code and text written by your fellow students. 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 **pdf file** containing your answers to the questions in the assignment (no Word files or other formats). If you scan handwritten material, make sure it is scanned in sufficiently high quality and reasonably scaled when included, so it is easily readable when printed.
2. A **zip file** containing the relevant Java (or Python) source files and level files (the ones you have modified or added, and *only* those).

The *front page* of your pdf file should contain:

- The name of the group (as it is registered on DTU Learn). Each group should choose a group name of at most 9 letters (restricted to the letters a-z and A-Z), e.g. `DeepGreen`, `MASochist` or `WatsOn`. This will allow you to preserve the group name for the programming project.
- Study numbers and full names of all group members (as they appear on DTU Learn).

Make sure to only use **private repositories** for sharing work in your group.

Assessment of the assignment This assignment involves 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 report you hand in, in particular your ability to formulate yourself in a mathematically and technically precise way 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* shorter and more precise. It is much better to write "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".

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 Russell & Norvig to brush up on the details of the techniques and concepts required to solve this assignment.

3 What we provide

For this assignment, we provide you with basic clients implemented in Java and Python. To obtain the implementations, download the archive `searchclient.zip` and unzip it somewhere sensible. You can choose to work with either the Java or Python client as you please. Do note, though, that you can expect significantly better performance when working in Java, that is, you will probably be able to solve more levels, and much faster (could be one order of magnitude or more). The clients contain a full implementation of the GraphSearch algorithm in Figure 3.7 of Russell & Norvig. It is this search client that you are expected to base your solution on. When you have decided which client to work with, open a command-line interface (command prompt/terminal) and navigate to the relevant folder, either `searchclient_java` or `searchclient_python`. 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 the default Java version of the assignment, 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 installed before the continuing with the assignment questions 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). If you use the Python version, you should similarly make sure that you have an updated version of Python that you can run from the command-line interface.

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 (Familiarize 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 (java version only) 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.

For this question, you don't have to hand anything in.

Exercise 2 (Graph Search for Multi-Agent Pathfinding)

In this exercise 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 yourself in order to be able to solve levels using breadth-first search (BFS). 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 `GraphSearch.java`. *For this question, report the solutions found (by stating the relevant action sequences), and briefly explain similarities and difference between your approach to solving these levels and how the* GRAPH-SEARCH *algorithm in the book would solve them.*

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. *For this question, include in the pdf file of your report a listing of your* GRAPH-SEARCH *code. Make sure the code is well commented and relates to the pseudocode of the* GRAPH-SEARCH *algorithm in the book.*

3. Run your BFS GRAPHSEARCH client on the `MAPF00.lvl`, `MAPF01.lvl`, `MAPF02.lvl` and `MAPF02C.lvl` levels and report your benchmarks. 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 the differences observed between the performance of the client on the different levels. *For this question, first fill in the relevant lines of Table 1. Then explain the differences in performance observed. Give brief, but conceptually and technically precise, answers, using the relevant notions from the course curriculum, e.g. notions such as* branching factor*,* state space size*,* search tree*, etc.*

Exercise 3 (Adding support for pushing and pulling boxes)

In this exercise, you will extend the set of actions to include pushing and pulling boxes. This of course also requires you to extend the implementations of ACTIONS(s) and RESULT(s, a).

Suppose a level has n agents ($n \leq 9$). The set ACTIONS(s) is then the set of joint actions $a_1|\cdots|a_n$ satisfying that 1) each a_i is applicable in s, and 2) no pair of actions a_i and a_j are conflicting. Start by rereading the definitions of applicability and conflicts in `hospital_domain.pdf` to recall exactly what those conditions are. In the client, applicability is defined by the `isApplicable` method and conflicts are defined by the `isConflicting` method, both in `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` and `isConflicting` in `State.java`.

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. *For this question, include in the pdf file of your report a listing of all code* changes*. Only include the code of the fields, constructors and methods you've modified. Make sure the code is well commented.*

2. Run your extended BFS GRAPH-SEARCH client on the `SAD1.lvl`, `SAD2.lvl`, `SAD3.lvl` and `SAfriendofBFS.lvl` levels and report your benchmarks. Explain which factors make `SAD2.lvl` much harder to solve using BFS than `SAD1.lvl`, and `SAD3.lvl` much harder than `SAD2.lvl`. *For this question, first fill in the relevant lines of Table 1. Then explain why there is such a huge difference between how difficult the different `SAD` levels are to solve. Give a brief, but conceptually precise, answer. Use the relevant notions from the course curriculum to assist you in making the answer as clear and technically precise as possible. For instance, you can refer to such notions as* branching factor*,* state space size*,* search tree*, etc.*

Exercise 4 (Depth-First Search)

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

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 `MAPF02C.lvl`, `SAD1.lvl`, `SAD2.lvl`, `SAD3.lvl` and `SAfriendofBFS.lvl` and report the results. Is DFS faster or slower than BFS and why? *For this question, you should fill in the relevant lines of Table 1 as well as very briefly explain how your implementation of DFS differs from the implementation of BFS. Furthermore, briefly comment on whether DFS is faster or slower than BFS and argue why.*

2. Benchmark the performance of both BFS and DFS on the two levels `SAFirefly.lvl` and `SACrunch.lvl`, shown in Figure 1. *For this question, you only have to fill in the relevant lines of Table 1, but please still make sure to spend a few minutes in your group for each benchmark to try to analyse why you get the results you get.*

Level	Frontier	States Generated	Time/s	Memory Used/MB	Sol length
MAPF00	BFS				
MAPF01	BFS				
MAPF02	BFS				
MAPF02C	BFS				
SAD1	BFS				
SAD2	BFS				
SAD3	BFS				
SAfriendofBFS	BFS				
SAFirefly	BFS				
SACrunch	BFS				
MAPF02C	DFS				
SAD1	DFS				
SAD2	DFS				
SAD3	DFS				
SAfriendofBFS	DFS				
SAFirefly	DFS				
SACrunch	DFS				

Table 1: Benchmarks table for uninformed search.

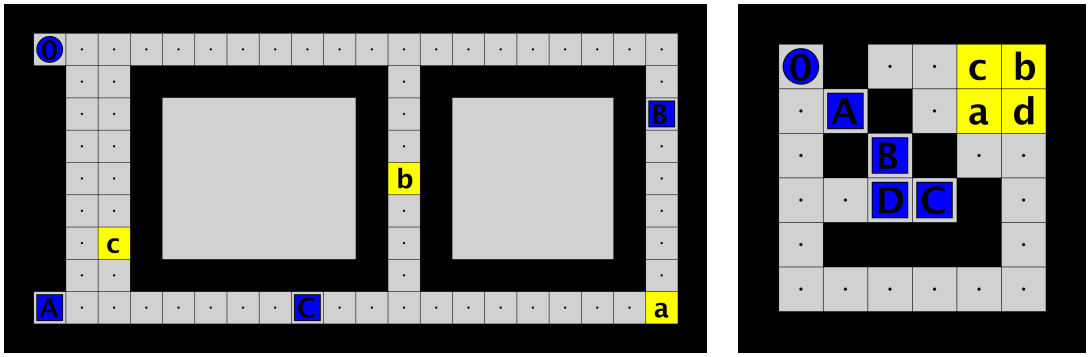

Figure 1: The (relatively simple) levels `SAFirefly.lvl` (left) and `SACrunch.lvl` (right).

```
#domain
hospital
#levelname
SAsoko1_08
#colors
blue: 0, A
#initial
++++++++++
+0A      +
++++++++++
#goal
++++++++++
+        A+
++++++++++
#end
```

Figure 2: `SAsoko1_08.lvl`

```
#domain
hospital
#levelname
SAsoko2_08
#colors
blue: 0, A
#initial
++++++++++
+0A      +
+        +
+        +
+        +
+        +
+        +
+        +
++++++++++
#goal
++++++++++
+        A+
+         +
+         +
+         +
+         +
+         +
+         +
++++++++++
#end
```

Figure 3: `SAsoko2_08.lvl`

```
#domain
hospital
#levelname
SAsoko3_08
#colors
blue: 0, A
#initial
++++++++
+0A     +
+ A     +
+ A     +
+ A     +
+ A     +
+ A     +
++++++++
#goal
++++++++
+      A+
+      A+
+      A+
+      A+
+      A+
+      A+
++++++++
#end
```

Figure 4: `SAsoko3_08.lvl`

Exercise 5 (State space growth)

Consider levels of the form shown in Figure 2–4. We let the *size* of such a level be the width of the level excluding walls. The levels `SAsoko1_08.lvl`, `SAsoko2_08.lvl` and `SAsoko3_08.lvl` all have size 8. In `searchclient.zip` we have provided you with levels of different sizes. What are the largest levels of each type that your BFS client can currently handle? In this exercise, we try to look a bit deeper into why there is such a big difference between the biggest size of level that can be handled of the three types.

1. Consider a level of the form in Figure 2. The length of a shortest solution is $\Theta(n)$ where n is the size of the level, since such levels can always be solved in $n-2$ steps (making $n-2$ pushes).[1] What is the size of the state space (number of possible states), also given in Θ-notation as a function of the size n of the level? *Report your answer as well as the calculations/reasoning leading to it.*

2. Consider a square level of the form in Figure 3. Since the level is square, the size n now denotes both the width and the height of the level, excluding walls. The length of a shortest solution is clearly still $\Theta(n)$. However, the size of the state space is bigger. What is now the size of the state space in Θ-notation? *Report your answer as well as the calculations/reasoning leading to it.*

3. Consider a square level of the form in Figure 4. Now also the length of a shortest solution is higher. Compute the length of a shortest solution as well as the size of the state space, both in Θ-notation.[2] Is the size of the state space polynomial in n, that is, is it in $O(n^k)$ for some constant k? *Report your answers as well as the calculations/reasoning leading to the answers.*

4. Run your BFS client on the different sizes of `SAsoko1`, `SAsoko2` and `SAsoko3` provided in `searchclient.zip`. Relate your findings to your answers to the previous 3 questions.

[1] Θ-notation is one of the standard asymptotic complexity measures used in algorithms. If you don't know what Θ-notation is, try to look it up in a textbook on algorithms or on the Internet—and otherwise ask the teacher or teaching assistant for help.

[2] *Hint*: You are allowed to use the binomial coefficient in your Θ-expressions.

For this question, you should include benchmarks of all the provided sizes of `SAsoko1`*,* `SAsoko2` *and* `SAsoko3` *in the style of Table 1, except you're allowed to exclude the column showing memory usage. Also, for each type of level, if you can't solve a level of size n, you don't need to include the benchmarks for levels of size $> n$. Analyse how the growth in solution length and number of generated states observed in your benchmarks relate to the theoretical complexity results from the previous questions.*

The previous exercise shows that even relatively simple problems in AI are completely infeasible to solve with naive, uninformed search methods. We will now turn to 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 6 (Heuristics)

Your next task is to implement an informed search strategy and then provide it with some proper information via the heuristic function. Background reading for this exercise is Section 3.5 in Russell & Norvig. In particular, when referring to $f(n)$, $g(n)$ and $h(n)$ below, they are used in the same way as in Russell & Norvig (and almost all other texts on heuristic search). 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 throws a `NotImplementedException`. 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. Benchmark your heuristic on the levels `MAPF02C.lvl`, `SAFirefly.lvl` and `SACrunch.lvl` and compare the results to DFS.

   For this question, first fill in the relevant lines of Table 2. Then analyse the differences in performance compared to DFS (why is greedy-best first search with the goal count heuristic significantly faster?). Include in your report also your choice of data structure for the frontier and all source code that you have made/modified.

2. Now try to design your own improved heuristic function h' to allow informed search to solve more levels and solve levels faster. Try to make it as efficient as possible, so that it can solve as many levels of possible. Remember that a heuristic function $h'(n)$ should always estimate the length of a solution from the state n to a goal state, while still being cheap to calculate. You may find it useful to do some preprocessing of the initial state in the `Heuristic` constructor.

   When designing your heuristic, it might be worth to do some benchmarks 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

Level	Eval	Heuristic	States Gen	Time/s	Mem Used/MB	Sol length
MAPF02C	Greedy	Goal Count				
SAFirefly	Greedy	Goal Count				
SACrunch	Greedy	Goal Count				
SAD1	A*	h'				
SAD1	Greedy	h'				
SAD2	A*	h'				
SAD2	Greedy	h'				
SAfriendofDFS	A*	h'				
SAfriendofDFS	Greedy	h'				
SAfriendofBFS	A*	h'				
SAfriendofBFS	Greedy	h'				
SAFirefly	A*	h'				
SAFirefly	Greedy	h'				
SACrunch	A*	h'				
SACrunch	Greedy	h'				
SAsoko1_64.lvl	Greedy	h'				
SAsoko2_64.lvl	Greedy	h'				
SAsoko3_64.lvl	Greedy	h'				

Table 2: Benchmarks table for informed search.

uninformed search than $A^\star$, so it is advised to primarily benchmark using greedy-best first search. Doing benchmarks with greedy best-first search also often gives 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).

For this question, you should in the report include: 1) a detailed and mathematically precise specification of the heuristic you have implemented, 2) the source code that you have made/modified, and 3) a description of the reasoning and intuition behind your heuristic, including how it estimates a length of a solution, and how general and precise it is.

3. Benchmark the performance of best-first search with your heuristic by filling in the remaining lines of Table 2. Analyse the improvements over uninformed search by comparing the new benchmarks with your earlier benchmarks. *For this question, you need to: 1) fill in Table 2; 2) analyse the improvements provided by $A^\star$ and greedy best-first search over BFS and DFS.*

Exercise 7 (Multi-Agent Planning)

In Exercise 2, we discovered that uninformed search quickly starts to struggle when the number of agents increase. We will now consider to which extend the same is true for informed search. Consider the level in Figure 5. This level as well as all simplified levels achieved by removing one or more of the agents (and their corresponding boxes) have been made available. For instance,

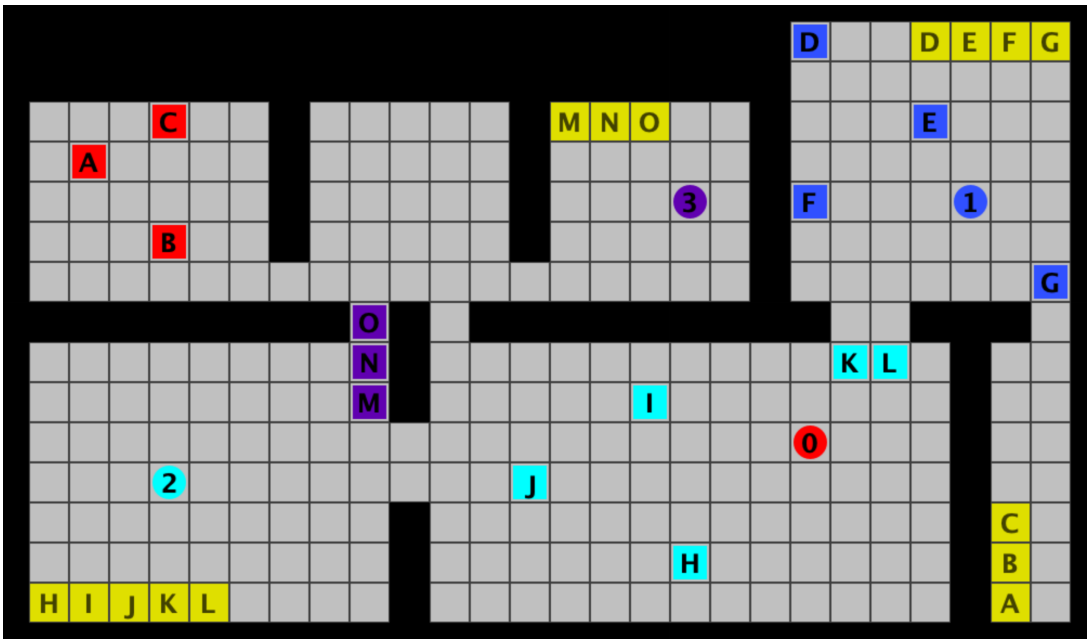

Figure 5: The multi-agent level `MAthomasAppartment.lvl`.

the level containing only the red and the blue agent is named `MAthomasAppartment_redblue.lvl` (we use colours to uniquely identify the relevant agents, as the level format requires the agents to be always numbered from 0).

1. First check how many of the levels you can solve with greedy best-first search and your current heuristic. Ideally you should be able to solve all of the levels with one or two agents, and at least half of the ones with three agents. If not, reconsider your heuristic, and see if you can improve it so that it can. It is not mandatory that you end up with a heuristic that can solve at least half of the three-agent levels, but you should at least try. *For this question, you should: 1) report on any improvements you might have made to the heuristic to make it solve more levels; 2) report on which levels you can solve in the style of the previous benchmarks (if you can solve* ***`redbluecyan`****, you can of course also solve* ***`redblue`****,* ***`redcyan`*** *and* ***`bluecyan`****, so only mention* ***`redbluecyan`*** *in that case).*

2. Discuss how to improve the client so that it would be able to solve the original level `thomasAppartment.lvl` (and similar multi-agent levels). Look at the benchmarks and solutions you found in the previous question to help you identify what makes such levels hard to solve. Then think about what could be done to make it easier. You can suggest any type of improvement or extension to the existing algorithms that you expect would solve the problem. The things that could potentially be relevant to think about includes:

   - The problem can to some extend be split into *subproblems* of the individual agents.
   - However, the subproblems are not completely independent. Some agents rely on other agents to move boxes out of the way.
   - Furthermore, two agents can create a new *conflict* if they plan to use the same cell at the same time (for themselves or for boxes).

If you have time, you are of course very welcome to implement your ideas and check that it works, but this is not mandatory. *For this question, you should: 1) explain as precisely as possibly your ideas for improving the algorithm to be able to handle the level (and similar multi-agent levels); 2) Describe any possible trade-offs involved, e.g. whether your suggested algorithmic modifications might lead to lower solution quality (longer solutions); 3) if you implemented your ideas, please include the relevant code and benchmarks of your results.*