Multiagent path planning

Tudor Avarvarei(4822242) and Victor Guillet (4488636)

November 22, 2022

1 IDENTIFYING PERFORMANCE INDICATORS

This section focus on explaining the data collection and statistical analysis process adopted throughout this project. First the key performance indicators selected are described in subsection 1.1. The data sampling method is then explained is in subsection 1.2, and finally an overview of the post-processing process is provided in subsection 1.3.

1.1 Performance indicators

In order to analyse the performance of the constructed models with the scope of comparing them impartially, it is necessary to define some indicators that can be calculated for all of them. These indicators should offer some relevant information about the results and computational efficiency and should be performed both for the system and the agent. For this reason, the following performance indicators were considered by the team:

- **Cost difference per agent:** This indicator is used to determine the average contribution of each agents to the difference between the ideal path (with collisions disabled) and the actual one determined by the algorithm (with collision enabled). This allows for effectively analysing the impact of the collisions on the quality of the solution generated by each method for each agent. This furthermore allow for effectively evaluating the quality of the solution found and comparing performance with other runs and algorithms as the indicator is normalised both with respect to path length and agent count. This metric is computed through first gathering the cumulated ideal cost and actual cost of each run. The cost difference per agent is then calculated using Equation 1.

$$\text{cost diff per agent} = \frac{\sum \text{cost} - \sum \text{ideal cost}}{\text{nb. agents}} \tag{1}$$

- **Efficiency:** The aim of this indicator is to calculate the efficiency of the algorithms by relating the CPU time cost to the solution quality. This indicator is obtained through first measuring the CPU time taken to find a solution, normalising it with the solution cost and agent count, and finally multiplying it with the cost difference per agent (Equation 2). 10 (arbitrarily chosen) is then raised to the power of the negative of the above value.

$$\text{efficiency} = 10^{-(\frac{\text{run time}}{\text{cost} \times \text{nb. agents}} \times \text{cost diff per agent})} \tag{2}$$

The normalised run time and cost difference per agent are multiplied to better reflect the efficiency of the algorithm. An algorithm is considered more efficient the lower its run time is, and the closer the solution is to the ideal one (so the smaller the cost difference per agent). Thus during the analysis of the results, the run time parameter will be shown and analysed. Accordingly, a high efficiency value implies a low run time and a solution close to the optimal one. It should be noted here that the equation adopted for this efficiency calculation could be improved, as the resulting value does not scale evenly with an improvement in run time vs an improvement in solution quality. This could be improved through normalising the two components or making them unit-less. It was however not done here due to time constraints.

Other scenario properties could have been considered, notably:

- **Obstacle density** (the ratio of non-obstacle tiles to obstacle tiles)
- **Layout density** (how clustered/spread out are the obstacles)
- **Starts/goals location density** (how close/far apart are the starts/goals respectively)
- **Operation count** (how many lines of codes are executed)

1.2 Data sampling method

Upon having selected the key performance indicators, a standard data sampling script and format was setup, ensuring consistency of the data collected across runs and methods. Six datapoints were selected for sampling after each run. Those are:

- **Test name:** The reference of the map used, value could be map_1, map_2, map_3
- **Start type:** The start/goal generation method. A value of 0 represent a random initial distribution of both start and goals, a value of 1 is for starts on the left and goals on the right, and 2 is alternating starts and goals left and right.
- **Cost:** The final total cost of the solution found
- **Ideal cost:** The total cost of the ideal solution, with agents collisions disabled
- **Run time:** The total run time of the experiment
- **Nb. of agents:** The agent count involved in the run

The rational behind the selection of the above metrics is as follows. The test name and start/goal generation type are necessary (along with the algorithm used) to properly label each run. The cost and ideal cost are necessary to evaluate the solution quality. The run time is needed to evaluate the algorithms' efficiency, and finally the agent count is necessary to normalise the data collected to make it comparable across runs.

The runner was then configured as follows. A range of agent counts (from 2 to 10 for all algorithms) was selected. The only exception for that being for CBS, for which a range 2 to 6 as it proved more computationally expensive (more on that in subsection 3.3). A run bundle was then defined to be all permutation of the agent range, the three maps, and the three starts/goal generation types, resulting in a bundle containing $9 \times 3 \times 3 = 81$ runs ($5 \times 3 \times 3 = 45$ for CBS). Each bundle was then run repeatedly.

Through the run, the coefficient of variation was computed using Equation 3 and monitored, and the data collection was terminated once it had been considered as stable enough (based on visual inspection of the trend line of the coefficient of stability over a rolling window of 100).

$$c_v = \frac{\sigma(o)}{\mu(o)} \tag{3}$$

1.3 Data post-processing

Upon having collected all the data, a script was developed for performing an analysis of the results (Statistical_significance_analysis.ipynb), and the following data points were then derived. First, the two performance metrics were derived for each run, using the formulas described above. The data was then bucketed according to various categories (agent count, map type, start/goal generation type), and a number of metrics were derived, including the mean of each bucket and their standard deviation. The correlation between the various performance indicators and other data points were then derived (notably the correlation between performance indicators and the agent count), and finally the distributions of the buckets' means and standard deviation was plotted. Finally, a local sensitivity analysis of the means with respect to the agent count, the map type, and the start type was performed. It can be noted here that given only one discrete/continuous variable could be varied (the agent count), the heatmap plots are less meaningful than they could have been, as some categories had to be used instead.

All the results are described in each algorithm's respective results analysis and statistical significance analysis sections. A comparison of the algorithms is then performed in section 5.

2 PRIORITIZED PLANNING WITH A*

In this section, the focus will be on the implementation of prioritized planning including the A* method. The A* method is used to find the most optimal path for an agent from its starting location to its goal. Then, based on the collisions of the agents, constraints are imposed on the agents; from agent 1 (who gets no constraint) to the last one (who had to find the best route based on the other agent's routes). This was the starting point and what was added to the code will be presented in subsection 2.1. Further, the section will focus on the results and their analysis. Thus in subsection 2.2 a brief analysis of the results will be made including how the varied number of agents, their initial locations and goal locations influence the performance of the model. Lastly, the evaluation of the results including the statistical significance analysis is presented in subsection 2.3

2.1 Model addition

The code received had most of the organisation logic included. In order to complete the logic, the A* method had to be developed from scratch along with the constraint table creation and addition and the checking procedure of the constraints for each of the agent's path creations.

Firstly, the A* function will be presented in Listing 1 and explained. The A* function receives the existing map (including walls), starting and goal location of the agent, the agent number and its constraints. With this, a dictionary for saving the studied and finished paths and an open list with open paths that have not been studied fully are created. Furthermore, the variable `earliest_goal_timestep` is created that saves the earliest time step of reaching the goal between all the agents. Next, the heuristic values of the

starting location and the root path are created. The root path is pushed and then constructed until the open list finishes to study all the nodes (in which case it means that no paths were found). In this while loop first, if the goal is reached then the path is outputted. Then, for each of the 5 possibilities (go up, right, left or down or wait), a new location is generated and validated (checked if it either gets out of the map or hits an obstacle). If not, a new child node is been created that checks if the new position is constrained. If it is not constrained, the node is checked in the closed list and if it exists there, it is compared with the existing code. If it is better than the existing node, the node is replaced and pushed back to the open list. If the child node is not in the closed list, it will be pushed there as well as in the open list.

```
def a_star(my_map, start_loc, goal_loc, h_values, agent, constraints):
    open_list = []
    closed_list_dict = dict()

    if constraint_table:
        earliest_goal_timestep = max(list(constraint_table.keys()))
    else:
        earliest_goal_timestep = 0

    # Current heuristics value given location
    h_value = h_values[start_loc]

    # -> Setup current node root
    root = {
        'loc': start_loc,
        'g_val': 0,
        'h_val': h_value,
        'parent': None,
        'timestep': 0
    }
    push_node(open_list, root)
    closed_list_dict[(root['loc'])] = root

    while len(open_list) > 0:

        curr = pop_node(open_list)

        if curr['loc'] == goal_loc and curr['timestep'] >=
        earliest_goal_timestep:
            return get_path(curr)

        # -> Check for child for all four directions (up, right, down,
        # left, wait)
        for dir in range(5):
            child_loc = move(curr['loc'], dir)

            # -> Check if new location is not outside the map
            if child_loc[0] < 0 or child_loc[1] < 0 or child_loc[0] >=
            len(my_map) or child_loc[1] >= len(my_map[0]):
                continue

            # -> Check if new location has obstacle
            if my_map[child_loc[0]][child_loc[1]]:
                continue
```

```
45
46              # -> Create new child node
47              child = {
48                  'loc': child_loc,
49                  'g_val': curr['g_val'] + 1,
50                  'h_val': h_values[child_loc],
51                  'parent': curr,
52                  "timestep": curr["timestep"] + 1      # New timestep
53              }
54
55              # -> Check child clash with constraints
56              if is_constrained(child, constraint_table):
57                  continue
58
59              # -> Check if child node is in closed list
60              if (child['loc'], child["timestep"]) in closed_list_dict:
61                  # -> Get existing node
62                  existing_node = closed_list_dict[(child['loc'],
63                  child["timestep"])]
64
65                  # -> Check if existing node is better than new node
66                  if compare_nodes(child, existing_node):
67                      # -> Replace existing node with new node
68                      closed_list_dict[(child['loc'], child["timestep"])] = child
69
70                      # -> Push node to open list
71                      push_node(open_list, child)
72
73              else:
74                  # -> Add child node to closed list dict
75                  closed_list_dict[(child['loc'], child["timestep"])] = child
76
77                  # -> Push node to open list
78                  push_node(open_list, child)
79
80      return None
```

Listing 1: A* method function

Next for the building of the constraints table, a separate function has been used, shown in Listing 2. This function receives all the constraints of an agent and it will output a dictionary with them ordered after the time step in ascending order.

```
1  def build_constraint_table(constraints, agent):
2      constraints_table = dict()
3
4      for constraint in constraints:
5          if constraint["agent"] == agent:
6              if constraint["timestep"] not in constraints_table:
7                  constraints_table[constraint["timestep"]] = []
8
9              constraints_table[constraint["timestep"]].append(constraint)
10
11     return constraints_table
```

Listing 2: Constraint table function

Moreover, a function has been created that checks if the next move of the agent contains a constraint that hinders it from moving there. The function receives a child that has just done a move and a list of constraints to check through them in both space and time. Finally, the function should output a boolean variable that confirms if the move is constrained or not. Firstly, the function checks if the studied timestep has constraints and if it is not the case, constraints, when agents reached their goal, will be imposed (such that when one agent reaches the goal, it does not physically disappear from the map). Further, for every such constraint, it will be checked if either the vertex or, later, the edge constraints are violated. If that is the case, then the child is constrained.

```
def is_constrained(child, constraint_table):
    # -> If there are not constraints
    if constraint_table == {}:
        return False

    # -> If timestep has constraints
    if child["timestep"] in constraint_table.keys():
        timestep_constraints = constraint_table[child["timestep"]]
    else:
        timestep_constraints =
        constraint_table[max(list(constraint_table.keys()))]

    # ... for every constraint
    for constraint in timestep_constraints:
        # if constraint is position constraint
        if isinstance(constraint["loc"], tuple):
            if constraint["loc"] == child["loc"]:
                return True    # -> Child is constrained

        # if constraint is edge constraint
        elif isinstance(constraint["loc"], list):
            if child["parent"]["loc"] == constraint["loc"][0] and
            child["loc"] == constraint["loc"][1]:
                return True    # -> Child is constrained

    return False    # -> Child is not constrained
```

Listing 3: Check if the next move is constrained function

Lastly, until now, only the A* algorithm was built. In order to arrange multiagent path planning, constraints had to be imposed. In order to do this, the first agent in the list will find a path. Based on his path, it will impose constraints (vertex and edge) for the other agents such that it does not intersect his path. Then, the second agent will find his path based on the already existent constraints and his path will impose new constraints. And this procedure will be repeated till all the agents reach their goal. In the end, the constraint addition procedure was implemented as shown in Listing 4.

```
for j in range(i + 1, self.num_of_agents):
    # -> Reset timestep
    timestep = 1

    for k, step in enumerate(path[1:]):
        constraints.append({'agent': j, 'loc': step, 'timestep': timestep})
        constraints.append({'agent': j, 'loc': [step, path[k]], 'timestep': timestep})
        timestep += 1
```

Listing 4: Constraints addition procedure for prioritized planning

2.2 Results analysis

This section will focus on the basic results analysis of the prioritized model using only the A* method for pathfinding. In order to perform this analysis, 2 different aspects of the problem are varied. Firstly, the number of agents will be altered and then the generation of the starting and goal positions of the agents will be changed.

The number of agents varied from 2 agents to 10. But before analysing the results, it is important to understand how path planning with A* works. As it was explained before, the planning is done by giving full priority to the first agents (priority is derived from the order in which they are processed), resulting in the constraints growing gradually as the agents paths are solved for, first for agent 0 till the very last agent. Thus, it exists the possibility that solutions might not be found as one agent with higher priority will have goals that block an agent with lower priority to reach its goal. In other words, if agent 1 needs to reach its goal in a channel behind agent 0, this method will fail to provide a valid solution. Thus it was observed that increasing the number of agents proved in the end not to be beneficial as this method rarely found solutions with no collisions. The solutions found proved to be quite variable, with some solutions occasionally matching the optimal solution. Having more agents (and implicitly more goals), results in the chance of blocking an agent with a lower priority (later in the list) being higher. As such the quality of the solution is heavily dependent on the priority order. Regarding the efficiency of the results, it was found that having very few agents prove more efficient as the chance of collision is low and the A* method has to solve for fewer paths. The efficiency of the method gradually drops as more path are added to the constraints, resulting in a more complex scenario.

Secondly, the starting points (and their goals placed on the opposite side of the map) of the agents were changed as well. One case is that all starting locations are placed in the left-most 2 columns while their goals are in the right-most 2 columns. The second case uses the same extremities of the map but it switches half of the agent's goal locations with their starting locations. In the end, it was found that the second case proved to be better as the problem with blocking the agent is half mitigated because the goals are placed in both areas now. However, the second case required the agent to meet in the middle of the board which led to some traffic jams solved, in the end, in an inefficient manner Lastly, the method was also tested by placing the starting and goal locations randomly on the whole map. In this case, the planning method could solve the models even with more agents as the goals had fewer chances of blocking the goal.

To summarise this method proved to be very efficient (computationally wise) when having a low number of agents with a low possibility of clashing. Also, placing their starting and goal locations as sparsely as possible is preferable for using this method. To understand better these conclusions, the next section will detail the results using some statistical analysis.

2.3 Statistical significance analysis

This section will present the statistical analysis performed for the prioritized planning with the help of the A* algorithm. As it was mentioned before, this section will analyse the algorithm based on 2 performance indicators (the cost difference per agent and the efficiency parameter). Moreover, the analysis will present the data in such a way that the influence of the number of agents involved, the map chosen, and on the start type (random, one-sided or double-sided) on the performance indicators are clearly shown.

Firstly, the influence of the number of agents on the performance indicators will be presented (to check the values, check the Table 1 and Table 19). In order to understand better the data (and to see the correlation more clearly), the mean of the cost difference per agent count was plotted in Figure 1 and the standard deviation of the same data was plotted in Figure 2. As it can be seen in these plots, the mean and standard deviation of cost difference is almost linearly (slightly exponentially) correlated with the number of agents. This conclusion is confirmed by the correlation coefficient shown in Table 2 where a clear positive correlation between the cost difference and the number of agents, close to 1, can be observed. These results make sense and are as expected given the concept of the algorithm. As the number of agents increase, the difference between the ideal cost and the actual cost also increases because the model becomes more complex (so harder to be solved) and more constraints appear (which forces solution off the normal path). Similarly, as the model becomes more complex the standard deviation also increases linearly with the number of agents as confirmed by the same correlation factor in Table 2. This is simply the result of having a higher mean for cost difference and the randomness of the starting and goal locations which translates in higher variance around the mean.

Next, the influence of the number of agents on the run time will be analysed where already it can be seen that the algorithm solves the problem in very short time. Firstly, the mean of these results were plotted in Figure 1 where a clear upwards exponential trend can be observed (confirmed again by the correlation factor in Table 2). This behaviour can be explained by the complexity of the model which also increases as the number of agents increases. In other words, the constraints increase linearly which means that for each agent that has to be checked with A*, exponentially more constraints must be checked to make a path which leads to exponential increase in run time. Regarding the standard deviation, the data was plotted in Figure 4 and exactly the same exponential behaviour can be observed. The resembling of the plots can be explained using the same reasons as in the cost difference resembling case regarding the strong correlation between mean and standard deviation values.

Lastly, for the number of agents data set, the efficiency was also calculated as explained in subsection 1.1. The data can be seen in Table 1 where this algorithm reaches extremely high efficiencies. However, there is a slight decrease in efficiency as the number of agents increase. This can be explained using the fact that, by increasing the number of agents, the run time and the cost difference increase which translates further into lower efficiency. Thus, for a high efficiency, it is favourable to have low number of agents.

Table 1: Mean of the performance indicators as a function of the number of agents for A* planning

Number of agents	cost difference per agent	run time	efficiency
2	0.913	0.001	0.999897
3	1.560	0.002	0.999591
4	2.126	0.004	0.998995
5	2.532	0.007	0.998161
6	2.926	0.010	0.996967
7	3.243	0.014	0.995498
8	3.570	0.019	0.993550
9	3.856	0.024	0.991218
10	4.139	0.030	0.988479

Figure 1: Mean of the cost difference as a function of the number of agents for A* planning

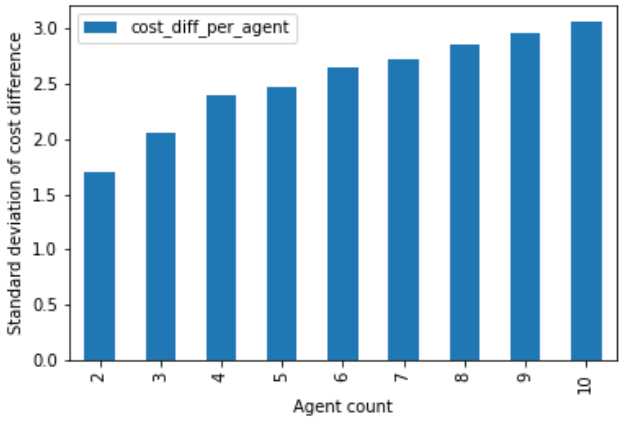

Figure 2: Standard deviation of the cost difference as a function of the number of agents for A* planning

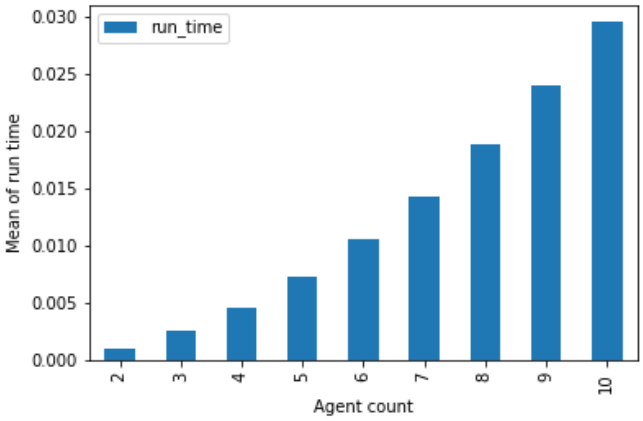

Figure 3: Mean of the run time as a function of the number of agents for A* planning

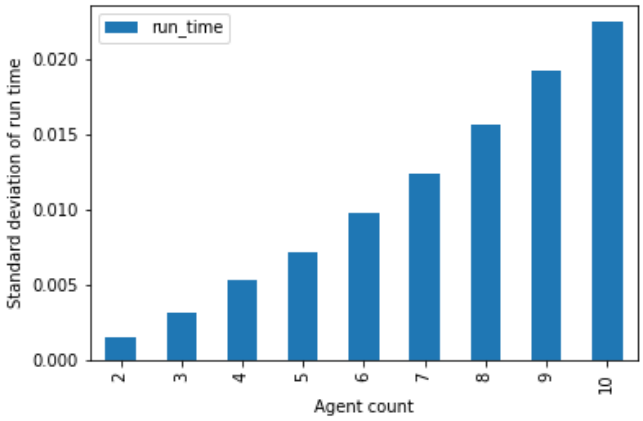

Figure 4: Standard deviation of the run time as a function of the number of agents

Table 2: Correlation of the performance indicators with respect to the number of agents for A* planning

Peformance indicators	Correlation mean	Correlation standard deviation
cost difference per agent	0.989	0.964
run time	0.982	0.992

Following the analysis of the number of agents, the map type and start type influence will be shown. As the dataset contains 3 points, the table with results will be shown while the plots can be found in the appendix as Figures 19-22. Firstly, the map type influence will be analysed. By looking at the means in Table 3 and the standard deviation in Table 4, it can be seen that the values do not vary greatly depending on the map type. Thus it can be reached a conclusion that the maps provided do not influence the results

too much. This can be explained using the fact that for A*, if the map becomes more complex, the cost and the ideal cost will increase proportionally. However, a slightly lower value can be seen for map 2. This does not have a clear justification and for a clear conclusion more data gathering might be necessary. Regarding the standard deviation the similarity to mean trend is still followed as the results do not vary a lot. The only interesting factor that could be learned is that the variance of map 1 is the smallest. This could be explained as, being the map with the most space, the chance of having collisions is always lower than for the other maps. Thus this translates into lower cost difference variance.

Table 3: Mean of the performance indicators as a function of the map type for A* planning

Map type	cost difference per agent	run time
Map 1	2.825	0.014
Map 2	2.569	0.011
Map 3	2.855	0.012

Table 4: Standard deviation of the performance indicators as a function of the map type for A* planning

Map type	cost difference per agent	run time
Map 1	2.481	0.017
Map 2	2.810	0.015
Map 3	2.965	0.015

Next, the start type will be analysed and its influence on the performance indicators. For this, the mean of the results will be presented in Table 5 while the standard deviation in Table 6. Firstly, the mean of the cost difference is the highest for random start which is explained as, compared to the other cases, the agents have the highest possibility into running into each other. This is because the starting and goal locations have no rule on placing on the map. Thus, as opposite to single-sided case where they all will follow same direction, in the random case they can clash in any direction anywhere. Thus this the higher probability in clashing leads to more collisions and higher cost difference. Similarly, as more collisions have to be solved, more computation is needed which also means a higher run time. Following up, the variance of the cost difference per agent is influenced similarly to the mean values using the same trend similarity as before. In other words, this translates to higher variance for random map for both performance indicators.

Table 5: Mean of the performance indicators as a function of the start type for A* planning

Start type	cost difference per agent	run time
Random	5.165	0.020
Single-sided	1.529	0.008
Double-sided	1.515	0.009

Lastly, the local analysis will be done using the contour plots parameters which were explained in subsection 1.3. The results can be seen in Figure 5:

These plots first confirm a number of trends observed above. First the strong correlation present between the agent count and the cost difference per agent. As can be noted on the plots, all tiles located

Table 6: Standard deviation of the performance indicators as a function of the start type for A* planning

Start type	cost difference per agent	run time
Random	3.429	0.022
Single-sided	1.027	0.008
Double-sided	1.034	0.009

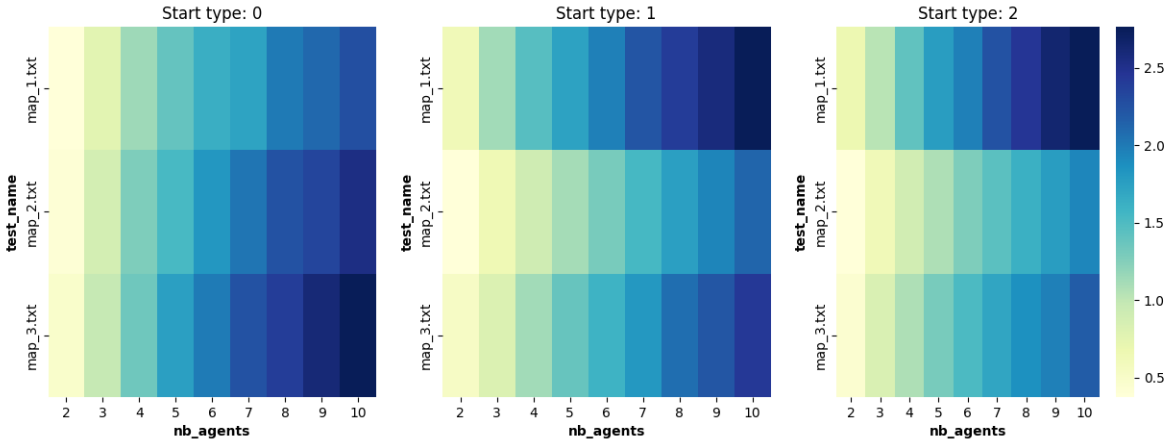

Figure 5: The average cost difference per agent for each map type, agent count, and start type (0 is random, 1 is left to right, and 2 is alternating

on the right hand sides are significantly darker, implying a higher value. The other interesting pattern is the influence of the map type and the start type on the cost difference per agent values. For start type 0 (random starts and goal location), the clear trend was that as the constraints introduced by the environment increased, so did the cost difference. This however is interestingly not the case for the other two start types, which both result in larger costs on map 1 (map 2 being the most performant map overall).

3 PRIORITIZED PLANNING WITH CBS

In this section, the focus will be on the implementation of prioritized planning with the Conflict-Based-Search(CBS) method. The A* method is still used to find the most optimal path for an agent from its starting location to its goal. Then, the CBS method, based on the collisions of the agents, will impose constraints on the agents and will create different nodes to be studied. For each constraint, one node will be created where the constraint is solved by one agent involved in the collision. Then, another node is created where the other agent involved in the collision solves the same constraint. In this way, all the possible routes are studied. This was the starting point and what was added to the code will be presented in subsection 3.1. Further, the section will focus on the results and their analysis. Thus in subsection 3.2 a brief analysis of the results will be made including how the varied number of agents, their initial locations and goal locations influence the performance of the model. Lastly, the evaluation of the results including the statistical significance analysis will be presented in subsection 3.3.

3.1 Model addition

The code received had most of the organisation logic included. In order to complete the logic, the CBS method to create constraints had to be developed from scratch along with some functions to detect the first collision between 2 agents and further split them into constraints.

Firstly, the CBS function will be presented in Listing 5 and explained. The CBS function is part of the CBS Solver class which receives the existing map (including walls) and the starting and goal locations of each agent. The function first finds a root node with the most optimal paths for each agent and it finds the collisions. This root node (all together with collisions) is pushed to the open list array and this is the point where the team's implementation starts. Thus, CBS works as long as the open list has nodes to be studied in which case it selects the first node. In case there are no nodes left, the CBS will return no paths which means that no solution was found. Also, it is worth mentioning that the nodes in the open list are ordered first after the cost of the node and then by the number of collisions. Thus, each time a node is popped, the node with the lowest cost and lowest number of collisions will come out.

If there are no collisions in this node then this is the best node found by the search according to the pre-made order, explained previously. Otherwise, it is necessary to study the node and thus, for each collision, constraints are translated for each agent in the collision. Further, for each constraint (respective for each agent), a new node is created that brings in the old constraints plus the new one and the old paths. Then for the agent implicated in the collision, the path is calculated with the new constraints using the A* method. Lastly, if a path could be found, the new node will be pushed to the open list. Also, it is worth mentioning that before pushing the node, the collisions of the new paths will be studied and the new cost calculated.

The main problem with CBS is that, even though it assures the optimal solution, the time it takes to find this value is usually very high, especially for very complex problems. Thus, the team decided to include some limits during the search such that the process is sped up. Some of the ideas included will be presented below:[1]

- **Ignore the new nodes with more than 7 collisions:** The idea behind this is that if the new node has too many collisions then probably the paths decided to follow are not the most optimal as it has to solve too many problems which will require a lot of time addition.
- **Ignore the new nodes that have 2 more new collisions:** This limit tries to discard the new wrong nodes as if it creates more collisions, it means that it needs to solve more constraints later which will translate in higher times.
- **Discard the nodes that have a cost of 10 higher than the root node cost:** This limit ensures that the CBS will eventually stop after some time as it will stop adding nodes with a cost too high. Moreover, this value is strictly chosen based on the current maps and assure that the solution that needed to be found is not too high.
- **Stop the process if the open list has more than 2000 nodes to study:** This last limit was only imposed for the statistical analysis and it assured that a run does not last longer than 15-20

[1]It is worth mentioning that these limits imposed to CBS are tweaked for the case of multiagent path planning inside the provided maps. Thus it is highly recommended to discard these speeding procedures for other CBS usage.

seconds. The purpose was that if the limit was reached then probably the CBS did not converge fast and a lot more time is needed so a fast stop is required.

```
while len(self.open_list) > 0:
    # Pop node from the list with smallest cost
    current_node = self.pop_node()

    if len(current_node['collisions']) != 0:
        for existing_collision in current_node['collisions']:
            new_constraints = standard_splitting(existing_collision)

            for new_constraint in new_constraints:
                # Add new constraint if it is not already in the list
                new_node_constraints = current_node['constraints'].copy()

                new_node_constraints.append(new_constraint)

                new_node_Q = {
                    'cost': 0,      # Placeholder
                    'constraints': new_node_constraints,
                    'paths': current_node['paths'].copy(),
                    'collisions': []
                }

                # Find the new path of the new constraint's agent
                agent_i = new_constraint['agent']

                path = a_star(
                    my_map=self.my_map,
                    start_loc=self.starts[agent_i],
                    goal_loc=self.goals[agent_i],
                    h_values=self.heuristics[agent_i],
                    agent=agent_i,
                    constraints=new_node_Q['constraints']
                )

                if path is not None:
                    new_node_Q['paths'][agent_i] = path
                    new_node_Q['collisions'] = detect_collisions(new_node_Q['paths'])
                    new_node_Q['cost'] = get_sum_of_cost(new_node_Q['paths'])

                    self.push_node(new_node_Q)

    else:
        return current_node['paths']
```

Listing 5: CBS method function

Next, the function that detects the collisions will be explained and can be seen in Listing 6. This function receives the paths of the agents that it needs to study and will output a list of collisions (where the collisions are dictionaries including data about the 2 agents involved, the location, the timestep and the type of collision (vertex or edge)). The function goes through each possible combination of each 2 paths and will try to detect the first collision that appears between the 2 agents. If there is a collision, the number of the agents is saved and the collision is appended to the list of collisions to be sent further to CBS

```python
def detect_collisions(paths):
    # -> Check collisions between all robot path pairs
    collisions = []

    for i in range(len(paths) - 1):     # ... for every agent
        for j in range(i + 1, len(paths)):      # ... check every other agent
            collision = detect_collision(paths[i], paths[j])
            if collision is not None:
                # -> Add robot ids to collision
                collision['a1'] = i
                collision['a2'] = j

                # -> Add collision to list
                collisions.append(collision)

    return collisions
```

Listing 6: Detect of the collisions function

In order to detect the first collision between 2 agents, a separate function was created (which could be seen in Listing 7) that receives the paths of the 2 agents. In the end, the function should output the collision dictionary including the collision type, location and timestep when it was produced (the agent numbers will be initialised in the previous function). To detect the vertex collision, the maximum length of the 2 paths has to be found to account for the case when one of the agents already reached the goal. Then the location of each agent is checked and if the locations match at one timestep, a vertex collision is found. Otherwise, for detecting the edge collision, the minimum length of the 2 paths has to be found as an edge collision is impossible to be detected when an agent reached its goal. Then if the location of 2 agents matches at 2 different consecutive time steps. In other words, the first agent location at time step t should be the same as the one of the second agent at time step t+1 and vice versa.

```python
def detect_collision(path1, path2):
    # -> Iterate through all timesteps and check for collisions
    path_max = max(len(path1), len(path2))

    for step in range(path_max):
        # -> Check if the robots are at the same location
        if get_location(path1, step) == get_location(path2, step):
            return {'type': 'vertex', 'loc': get_location(path1, step), 'timestep':
    step}

    path_min = min(len(path1), len(path2))

    for step in range(path_min):
        # -> Check if the robots swap their location
        if get_location(path1, step) == get_location(path2, step + 1) and get_location
    (path1, step + 1) == get_location(path2, step):
            return {'type': 'edge', 'loc': [get_location(path1, step), get_location(
    path1, step + 1)], 'timestep': step+1}

    return None
```

Listing 7: Detect of the first collision function

The last function that was used for the creation of the CBS algorithm is a simple splitting function and can be seen in Listing 8. The input of this function is a collision dictionary and the function outputs

a list of 2 constraint dictionaries respective for each agent including the location, the agent name and the time step. The function considers also the 2 types of constraints namely edge and vertex constraints.

```
def standard_splitting(collision):
    if collision['type'] == 'vertex':
        return [{'agent': collision['a1'], 'loc': collision['loc'], 'timestep':
        collision['timestep']},{'agent': collision['a2'],
        'loc':collision['loc'], 'timestep': collision['timestep']}]
    else:  # Edge constraint (the second agent gets the limits switched)
        return [{'agent': collision['a1'], 'loc': collision['loc'], 'timestep':
        collision['timestep']},{'agent': collision['a2'], 'loc':
        [collision['loc'][1], collision['loc'][0]], 'timestep':
        collision['timestep']}]
```

Listing 8: Split collision into constraints function

3.2 Results analysis

This section will focus on the basic results analysis of the prioritized model using the CBS algorithm for pathfinding. In order to perform this analysis, the three parameters investigated previously are similarly to the previous section the agent count, the map type, and the start/goal generation method.

The number of agents varied from 3 agents to 10. However, as it was stated before, CBS assures that the optimal path is found by exploring the created nodes caused by collisions in the order of cost and number of collisions. As a consequence, this method is much more computationally expensive compared to the previous planning. Thus having 10 agents on the map and finding a solution proved too computationally expensive so in the end, the tests did not include this option. On the other hand, having less than 5 agents would find solutions immediately. Thus for CBS, having more than 7-8 agents in the current maps start to be problematic for the run time but whatsoever the procedure will lead to the most optimal results. Moreover, it is worth mentioning what will the CBS choose when there are multiple optimal paths. The answer for this lies in the A* procedure for pathfinding and more exactly in how the moves to search through were ordered. For this case, the search order moves down, right, up and left and only then wait. As the max function returns the first appearance of maximum values, the CBS will always prefer going down compared to other moves when the same heuristics can be found for more moves. Also, waiting is the last option considered when the heuristics are the same.

Secondly, the starting points (and their goals placed on the opposite side of the map) of the agents were changed as well similar to the prioritized planning with the A* method in 2 cases. Opposite to the previous method, it was found that the first case proved to be computationally better. This is a consequence of having more agents that need to be close to each other (due to meet in the centre at the same time) which leads further to a lot of collisions and accordingly to higher computational times. In the end, CBS learns to make 2 channels dedicated to one direction and 2 channels dedicated to the other direction (which requires a lot of testing without learning models and just trying out nodes till finding the best one). Lastly, the method was also tested by placing the starting and goal locations randomly on the whole map. In this case, the planning method still required a lot of time with more agents as the collisions between agents remained which still requires a lot of time to solve.

To summarise this method proved to always lead to the best result but with a huge cost namely the run time. Thus, to use this method, it is recommended to use it for a small map (where there are few

possibilities to be studied) and preferably with a low number of agents (or with a low risk of collision). Moreover, placing their starting and goal locations such that collisions are avoided (starting spots on one side and the goals on the other one) is preferable for using this method. To understand better these conclusions, the next section will detail the results using some statistical analysis.

3.3 Statistical significance analysis

This section will present the statistical analysis performed for the prioritized planning with the help of CBS algorithm. As it was mentioned before, this section will analyse the algorithm based on 2 performance indicators (the cost difference per agent and the efficiency parameter). Moreover, the analysis will present the data in such a way that the influence of the number of agents involved, the map chosen, and on the start type (random, one-sided or double-sided) on the performance indicators are clearly shown.

Firstly, the influence of the number of agents on the performance indicators will be shown (to check the values, look at the Table 7 and Table 20). In order to understand better the data (and to see the relation between the parameters better), the mean of the cost difference per agent count was plotted in Figure 6 and the standard deviation of the same data was plotted in Figure 7. The first impression about these plots is that, compared to the planning with A*, the relation is not linear anymore. Again, this conclusion is confirmed by the correlation coefficients shown in Table 8 where a positive correlation can be seen, but this time not as close to 1 as before. Firstly, for the correlation mean, a linear relation can be seen (slightly exponential as before). By looking at how the planning with CBS finds the solution, it is known that the optimal solution will come out of it. Thus, the cost difference value shown in the plot comes out from the difference between the ideal cost (with collisions) and the optimal cost (without collisions). As more agents are involved, this difference increases as more collisions are expected and this relation is expected to be linear as it is now. On the other hand, as the model becomes more complex the standard deviation seems to remain level with a huge spike at the end which can be assumed to be an outlier. This outlier is hard to be explained but one possible answer to the problem could be the fact that many tests failed for this number of agents (did not finish). This led further to a high variance in the results where some tests were solved very fast and the other very slow, resulting in this average distribution. However, given the high number of iterations performed this should not have happened and the results are still surprising.

Next, the influence of the number of agents on the run time will be analysed. Firstly, the mean of these results were plotted in Figure 6 where a clear upwards exponential trend can be observed much stronger than in the A* planning case (confirmed again by the correlation factor in Table 8). This behaviour can be explained by the complexity of the model which also increases as the number of agents increases. In other words, the constraints increase linearly and this translates further into 2 more nodes to be studied for each constraint. Thus to solve the problem and find a solution, the computational complexity increases with 2^{nb_agents}. As the run time is proportional to the computational complexity, it can be easily explained why the run time has such a sudden increase. Regarding the standard deviation, the data was plotted in Figure 9 where the results are not as clear as before but seems to just resemble the mean data and be closely related to it. Moreover, there seems to be an outlier as before which could be explained using the same reasons. Lastly, regarding the efficiency, the same behaviour and decreasing trade can be observed as in the A* planning algorithm results. Thus the same conclusion can be drawn. Moreover, the decreasing trend is much higher which can be explained straight-forward by the steep increase in run time.

Following the analysis of the number of agents, the map type and start type influence will be shown.

Table 7: Mean of the performance indicators as a function of the number of agents for A* planning

Number of agents	cost difference per agent	run time	efficiency
2	0.043	0.001	0.999996
3	0.098	0.011	0.999878
4	0.149	0.065	0.998911
5	0.178	0.192	0.996120
6	0.186	0.365	0.992063

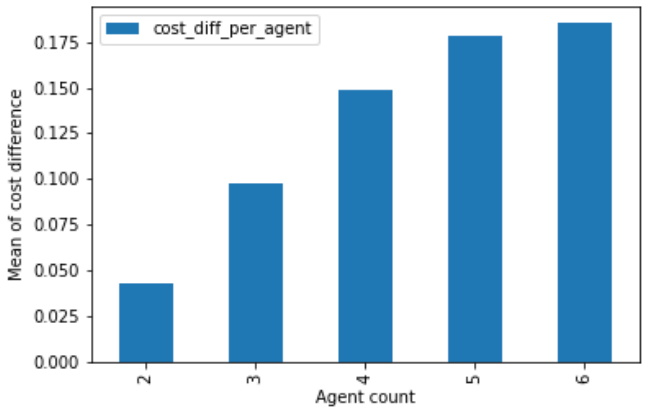

Figure 6: Mean of the cost difference as a function of the number of agents for CBS planning

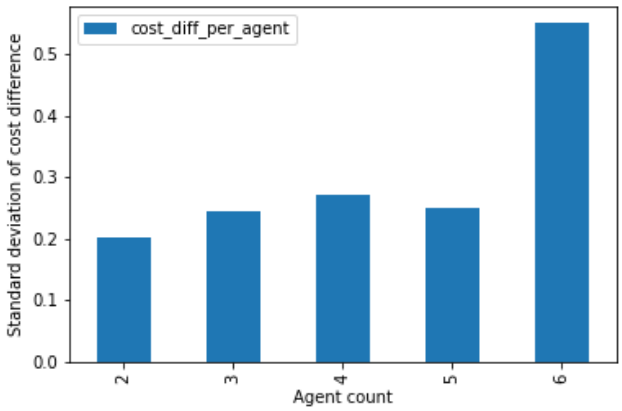

Figure 7: Standard deviation of the cost difference as a function of the number of agents for CBS planning

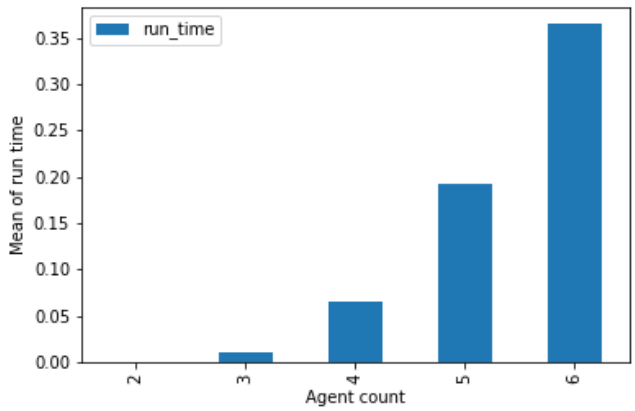

Figure 8: Mean of the run time as a function of the number of agents for CBS planning

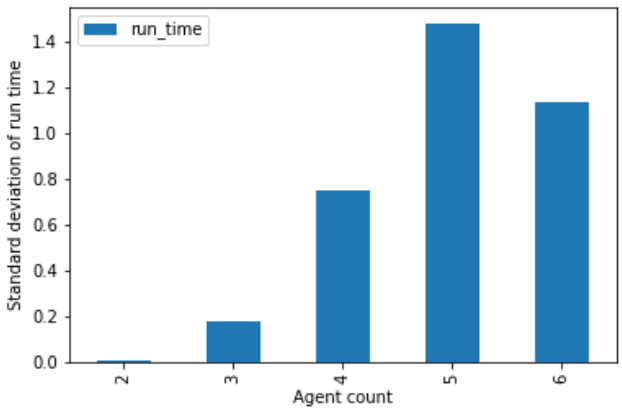

Figure 9: Standard deviation of the run time as a function of the number of agents for CBS planning

As the dataset contains 3 points, the table with results will be shown while the plots can be found in the appendix as Figures 23-26. Firstly, the map type influence will be analysed. By looking at the means in Table 9 and the standard deviation in Table 10, it can be seen that the values increase linearly as the map

Table 8: Correlation of the performance indicators with respect to the number of agents for CBS planning

Peformance indicators	Correlation mean	Correlation standard deviation
cost difference per agent	0.964449	0.792839
run time	0.937471	0.903547

complexity increases. This can be explained as a more complex map leads to more collisions which further translates to a higher difference between the ideal and optimal paths. The relation is much more straight forward as the CBS algorithm calculates the optimal path and the offset from the ideal path depends linearly to the complexity of the map. The same arguments can be applied for the run time linearity trend observed in the table. Regarding the standard deviation some interesting outcomes can be observed as well. Firstly, the values are seemingly uncorrelated to the map type and the standard deviation of the cost difference is overall the same which can be seen as a mirror of the mean values. However, for run time of map 1, there is clearly more variance. This could be explained as, being the least constraining map, it either reaches its goal very fast or there are more constraints to be considered which leads to higher computation times. In other words, the low variance of the last 2 maps is caused by the few possibilities an agent can take due to the high number of obstacles.

Table 9: Mean of the performance indicators as a function of the map type for CBS planning

Map type	cost difference per agent	run time
Map 1	0.094	0.098
Map 2	0.134	0.114
Map 3	0.152	0.126

Table 10: Standard deviation of the performance indicators as a function of the map type for CBS planning

Map type	cost difference per agent	run time
Map 1	0.234	1.198
Map 2	0.228	0.528
Map 3	0.441	0.768

Next, the start type will be analysed and its influence on the performance indicators. For this, the mean of the results will be presented in Table 5 while the standard deviation in Table 6. Firstly, the mean of run time is the lowest for random start which is explained as, compared to the other cases, the paths it needs to compute are shorter. On the other hand, the cost difference per agent trend is opposite to the run time indicator. This can be explained using the previous argument that for random map, collisions are expected everywhere and in every direction which makes the optimal solution be further from the ideal solution. Similarly, the double-sided starting type have a low cost difference as there are multiple ways to not collide and reach the optimal paths because the starting and goal location are not closely together. Following up, the variance of the cost difference per agent is influenced similarly to the mean values (for the same reasons as before). On the other hand, for the run time, it seems there is higher variance for the random start type. This is straight-forwardly explained using the randomness of the starting and goal locations which lead to either fast or slow solutions.

Lastly, the local analysis will be done using the contour plots parameters which were explained in subsection 1.3. The results can be seen in Figure 10.

Table 11: Mean of the performance indicators as a function of the start type for CBS planning

Start type	cost difference per agent	run time
Random	0.156	0.088
Single-sided	0.137	0.135
Double-sided	0.085	0.117

Table 12: Standard deviation of the performance indicators as a function of the start type for CBS planning

Start type	cost difference per agent	run time
Random	0.463	1.235
Single-sided	0.211	0.660
Double-sided	0.186	0.576

Unlike with prioritized planning, this set of heat map enabled us to gather less information. A trend can be established for start type 0 (random start). Yet again map three proves most constraining, resulting in the highest cost difference per agent. The one consistent pattern across heatmaps though is the correlation between the agent count and the cost difference. This yet again confirms the statistics determined above.

4 INDIVIDUAL PLANNING OF MOVING AGENTS

The approach selected by the group for this task was one primarily leveraging implicit logic to get to their respective goals. The individual planning implementation behaves as follows. Before starting to move on the map, each agent first compute their respective heuristics, providing them with a road map to get to their goal in an optimal scenario with only static obstacles to account for.

```
result = []

# -> Create agent objects with AircraftDistributed class
for i in range(self.num_of_agents):
    # -> Compute heuristics for new agent
    heuristics = compute_heuristics(my_map=deepcopy(self.my_map), goal=self.goals[i])

    # -> Create new agent object
    newAgent = AircraftDistributed(
        my_map=deepcopy(self.my_map),
        start=self.starts[i],
        goal=self.goals[i],
        heuristics=heuristics,
        agent_id=i
    )
    # -> Add agent to list of agents
    self.agents.append(newAgent)
```

Listing 9: Distributed planning initialisation

Upon having solved for this, the agents start taking steps toward their goal. The process is performed in epochs, with each agent taking a single step at the time. At every step, each agent first determines the list of possible actions available to it (tiles with no obstacles or other agents), and their corresponding cost (more details are provided on the cost computation in subsection 4.1). The available actions subset is the same as in CBS, that is at most, agents are allowed to go up, down, left, right, or wait. The agent then

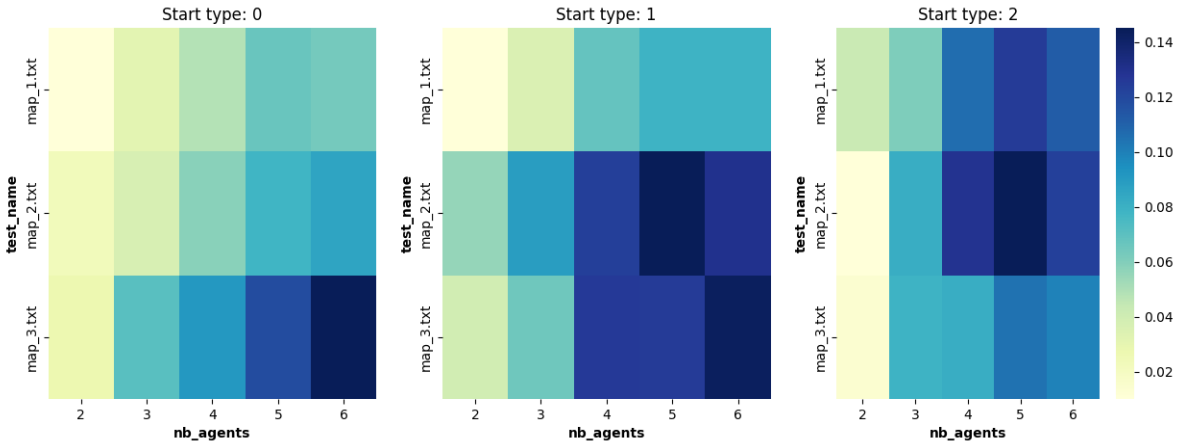

Figure 10: The average cost difference per agent for each map type, agent count, and start type (0 is random, 1 is left to right, and 2 is alternating)

selects the action with the lowest cost, update its internal state (more in subsection 4.1), and move to the new position.

Upon reaching a goal, an agent adds itself to the obstacle map as a permanent obstacle, and trigger a rest of other agent's internal states, and a re-computation of their respective heuristic (taking into account the new permanent obstacle).

This process is repeated until all agents have reached their goals or some agents' path becomes entirely blocked, making their goal unreachable.

```
if agent.at_goal:
    # -> Add agent location as permanent obstacle in my_map
    self.my_map_shadow[agent.loc[0]][agent.loc[1]] = True

    for other_agent in self.agents:
        if not other_agent.at_goal:
            # -> Re-compute heuristics for all agents not at goal
            other_agent.obstacle_map = np.array(self.my_map_shadow)
            other_agent.heuristics = compute_heuristics(my_map=self.my_map_shadow,
    goal=other_agent.goal)

            # -> Reset weights
            other_agent.my_weights = np.ones((len(self.my_map), len(self.my_map[0])))

            # -> Update agent state
            agents_states_dict[other_agent.id] = self.update_agent_state(agent=
    other_agent)
```

Listing 10: Heuristics are re-computed when an agent reaches its goal

4.1 Cost computation

To compute the cost of various actions, two key metrics are initially considered (a third one is added later on to improve implicit coordination). The two initial metrics are the heuristic, multiplied by an

agent-specific "memory weight".

This internal map is initialised with ones everywhere, and is then adjusted as the agent progresses on the map. Every step an agent spends on a tile cause the tile's cost factor to increase, encouraging agents to avoid repeatedly visiting the same tile. This is necessary to ensure agents do not get stuck in dead-ends, and move out of an area if it has been repeatedly visited with no success. It may however be that an agent finds itself in a "temporary dead-end", resulting from other agents temporarily blocking the way for example. To ensure that the area is not permanently ignored, a forgetting factor is also applied at every step, gradually reducing the magnitudes of the memory weight.

The approach selected by the group for this task was one primarily leveraging implicit logic to get to their respective goals. The individual planning implementation behaves as follows. Before starting to move on the map, each agent first compute their respective heuristics, providing them with a road map to get to their goal in an optimal scenario with only static obstacles to account for.

```
result = []

# -> Create agent objects with AircraftDistributed class
for i in range(self.num_of_agents):
    # -> Compute heuristics for new agent
    heuristics = compute_heuristics(my_map=deepcopy(self.my_map), goal=self.goals[i])

    # -> Create new agent object
    newAgent = AircraftDistributed(
        my_map=deepcopy(self.my_map),
        start=self.starts[i],
        goal=self.goals[i],
        heuristics=heuristics,
        agent_id=i
    )

    # -> Add agent to list of agents
    self.agents.append(newAgent)
```

Listing 11: Distributed planning initialisation

Upon having solved for this, the agents start taking steps toward their goal. The process is performed in epochs, with each agent taking a single step at the time. At every step, each agent first determines the list of possible actions available to it (tiles with no obstacles or other agents), and their corresponding cost (more details are provided on the cost computation in subsection 4.1). The available actions subset is the same as in cbs, that is at most, agents are allowed to go up, down, left, right, or wait. The agent then selects the action with the lowest cost, update its internal state (more in subsection 4.1), and move to the new position.

```
 agents_states_dict):
# -> Get available actions
available_actions = self.get_available_actions(agents_location_map=agents_location_map
    )

# -> Compute each action's cost
costs = []

for action in available_actions:
    costs.append(self.heuristics[action] * self.my_weights[action])

# -> Choose action with lowest cost
```

```
action = available_actions[costs.index(min(costs))]

# -> Update weight map
# Forget
self.my_weights = self.my_weights - 0.05
self.my_weights = self.my_weights.clip(min=1)

# Decrease prev loc appeal
self.my_weights[self.loc[0]][self.loc[1]] += 0.3

# -> Update agent's location
self.loc = action

# -> Update agent's path
self.path.append(action)
```

Listing 12: Distributed planning base agent step logic

Upon reaching a goal, an agent adds itself to the obstacle map as a permanent obstacle, and trigger a rest of other agent's internal states, and a re-computation of their respective heuristic (taking into account the new permanent obstacle).

This process is repeated until all agents have reached their goals or some agents' path becomes entirely blocked, making their goal unreachable.

```
if agent.at_goal:
    # -> Add agent location as permanent obstacle in my_map
    self.my_map_shadow[agent.loc[0]][agent.loc[1]] = True

    for other_agent in self.agents:
        if not other_agent.at_goal:
            # -> Re-compute heuristics for all agents not at goal
            other_agent.obstacle_map = np.array(self.my_map_shadow)
            other_agent.heuristics = compute_heuristics(my_map=self.my_map_shadow,
    goal=other_agent.goal)

            # -> Reset weights
            other_agent.my_weights = np.ones((len(self.my_map), len(self.my_map[0])))

            # -> Update agent state
            agents_states_dict[other_agent.id] = self.update_agent_state(agent=
    other_agent)
```

Listing 13: Heuristics are re-computed when an agent reaches its goal

4.2 Coordination between agents

Limiting the computation of cost to the above mentioned factors results in an almost priority-planning behavior in the sense that agents performing their step first during an epoch find themselves at an "advantage" over the ones following them in the processing sequence. To address this, an extra term can be included. The approach adopted for this assignment was that of introducing a repelling force field for each agent. The strength of the field is inversely proportional to the distance between two agents, and inversely proportional to the length of the optimal route between an agent and it's goal. The force fields also have a limited range, ensuring that agents far apart don't affect each other.

The resulting behavior is two fold. First, agents close to their goals effectively push back other agents. This essentially allows for giving agents near their goals priority over others. The second key advantage of this method is that groups of agents get priority of individual ones. The implementation adopted allows for these force fields to stack. This results in two agents going one way better being able to repel a single one. This effect is most effective when the group of agent is nearing its goal.

A potential improvement of the above mentioned methods would be to first scale the scope of the first field according to the distance between an agent and its goal (this would ensure that agents really far away from their goals impacts others as little as possible). Another improvement would be to generate directional force field. In the current implementation circular force field were used. Replacing them with more conical ones, repelling only in the direction of the goal could further focus the impact of the field correctly.

```
# -> Get available actions
available_actions = self.get_available_actions(agents_location_map=agents_location_map
    )

# -> Compute each action's cost
costs = []

for action in available_actions:
    costs.append(self.heuristics[action] * self.my_weights[action])

# -> Choose action with lowest cost
action = available_actions[costs.index(min(costs))]

# -> Update weight map
# Forget
self.my_weights = self.my_weights - 0.05
self.my_weights = self.my_weights.clip(min=1)

# Decrease prev loc appeal
self.my_weights[self.loc[0]][self.loc[1]] += 0.3

# -> Update agent's location
self.loc = action

# -> Update agent's path
self.path.append(action)
```

Listing 14: Distributed planning base agent step logic

4.3 Semi-formal specification

A semi-formal specification of the full final algorithm for individual planning can be seen bellow:

```
    # Initialise agents
    loc : start loc
    goal : goal loc
    at_goal : False

Every time an agent must stake a step:
    # Get all available actions, a valid action  is one with no obstacles or agent at the location
    for action in (up, down, left, right, wait) -> {
        valid_action(action, True) -> Ignore action
        valid_action(action, False) -> Accept action
        }

    # Compute the cost of each action (based on own heuristic + own experience map + force fields of agent in a given range)
    cost(action, experience map, other agents loc, other agents' distance to their goals) -> action cost

    # Select action with lowest cost
    min(actions costs) -> best action

    # Update experience map weights
    # "Forget" by decreasing all weight of the experience map by a small fixed amount (with the weights being floored at 1)
    set(experience map, forget_factor) -> Updated experience map

    # Update own location
    set(experience map, prev loc, memory factor) -> Updated experience map

    # Update states based on agents progress

    # Flag itself as at goal
    loc == goal -> {
        set(agent, at_goal(True))

        # Add current position (goal position) to shared map as permanent obstacle.

        Every other agent:
            # Reset own experience map weights to 1
            set(agent, weight map) -> Array of 1s

            # Re-compute new heuristic
            do(agent, compute Heuristic)
        }
```

Listing 15: Distributed planning algorithm pseudo code

4.4 Statistical significance analysis

This section will present the statistical analysis performed for the distributed planning algorithm. As it was mentioned before, this section will analyse the algorithm based on 2 performance indicators (the cost

difference per agent and the efficiency parameter). Moreover, the analysis will present the data in such a way that the influence of the number of agents involved, the map chosen, and on the start type (random, one-sided or double-sided) on the performance indicators are clearly shown.

Firstly, the influence of the number of agents on the performance indicators will be presented (to check the values, check the Table 13 and Table 21 in appendix). In order to understand better the data (and to see the linear relation more clearly), the mean of the cost difference per agent count was plotted in Figure 1 and the standard deviation of the same data was plotted in Figure 2. As it can be seen in these plots, the mean and standard deviation of cost difference is almost linearly (slightly exponentially) correlated with the number of agents. This conclusion is confirmed by the correlation coefficient shown in Table 14 where a clear positive correlation between the cost difference and the number of agents, close to 1, can be observed. This results are as expected given the algorithm's conception. As the number of agents increase, so does the probability for them to get in each other's way. This naturally leads to more conflicts, and more conflict resolution steps accordingly. Those in turn drive up the mean.

Next, the influence of the number of agents on the run time will be analysed. Firstly, the mean of these results were plotted in Figure 1 where a clear upwards exponential trend can be observed (confirmed again by the correlation factor in Table 14). This behaviour can be explained by the complexity of the model which also increases as the number of agents increases. In other words, more agents results in more collisions, and more computations (notably of the force field, and re-computation of all heuristics when an agent gets to its goal), which naturally increases the run time. Regarding the standard deviation, the data was plotted in Figure 4 where a clear outlier can be observed (for 9 agents). This outlier is hard to be explained but could most likely have been caused by the same reasons as in the case of the CBS planning outlier explained before. In short, there could have been a lot of missed tests (where no results is reached). This led further to a high variance in the result where some tests were solved very fast and the other very slow, resulting in this average distribution. Otherwise, similar to the cost difference analysis, this plot should follow the exponential increase of the mean of run time.

Table 13: Mean of the performance indicators as a function of the number of agents for distributed planning

Number of agents	cost difference per agent	run time	efficiency
2	0.948	0.011	0.998858
3	1.313	0.021	0.997028
4	2.038	0.034	0.992831
5	2.552	0.050	0.987270
6	3.418	0.068	0.977607
7	4.141	0.087	0.965866
8	5.024	0.109	0.949805
9	5.993	0.136	0.927891
10	7.000	0.156	0.906668

Following the analysis of the number of agents, the map type and start type influence will be shown. As the dataset contains 3 points, the table with results will be shown while the plots can be found in the appendix as Figures 19-22. Firstly, the map type influence will be analysed. By looking at the means in Table 15 and the standard deviation in Table 16, it can be seen that the values do not vary greatly depending on the map type. However, a slightly higher values can be seen for map 2 and especially for

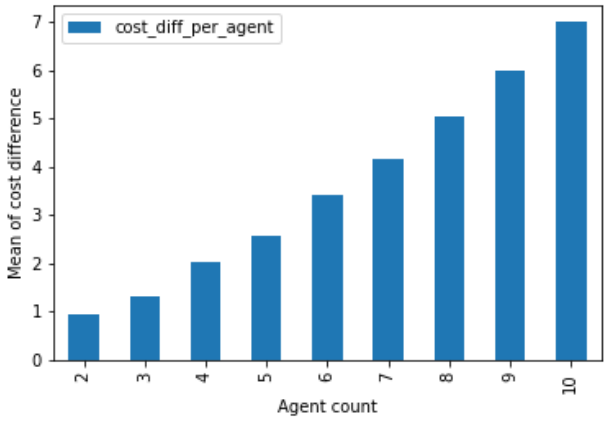

Figure 11: Mean of the cost difference as a function of the number of agents for distributed planning

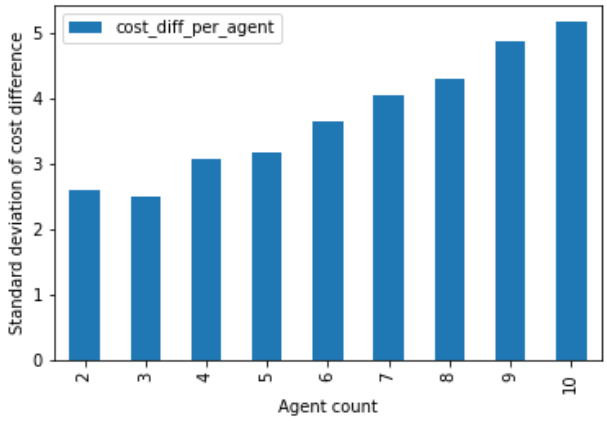

Figure 12: Standard deviation of the cost difference as a function of the number of agents for distributed planning

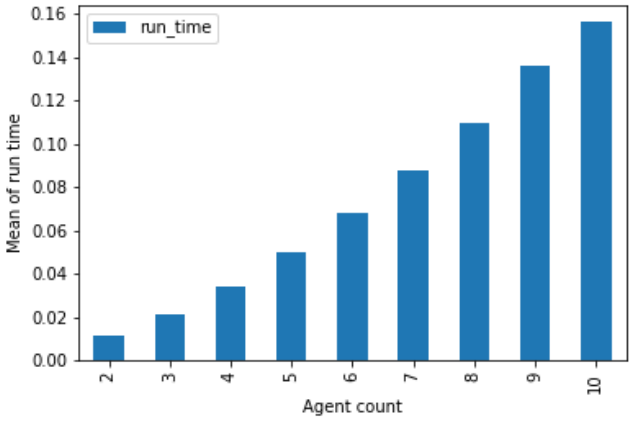

Figure 13: Mean of the run time as a function of the number of agents for distributed planning

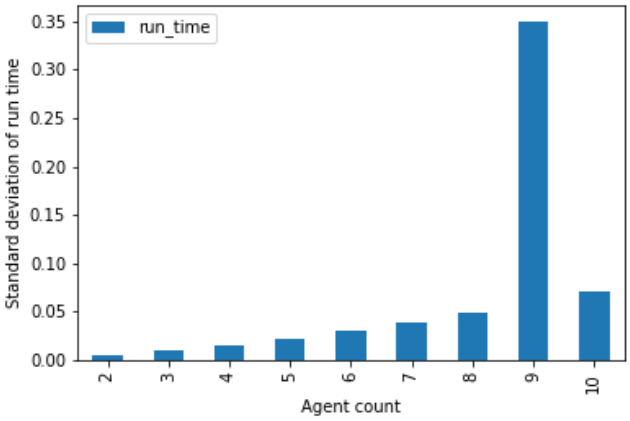

Figure 14: Standard deviation of the run time as a function of the number of agents for distributed planning

Table 14: Correlation of the performance indicators with respect to the number of agents for distributed planning

Performance indicators	Correlation mean	Correlation standard deviation
cost difference per agent	0.964449	0.792839
run time	0.937471	0.903547

map 3. This can be explained as both maps' layout are more prone to resulting in conflicts which will lead to more constraints and thus a higher cost difference and higher run time. Regarding the standard deviation some interesting outcomes can be observed as well. Firstly, the values are seemingly uncorrelated

to the map type and the standard deviation of the cost difference is overall the same which can be seen as a mirror of the mean values. On the other hand, for run time, there is clearly more variance for map 0. This could be explained as, being the least constraining map, it either reaches its goal very fast or there are more constraints to be considered which leads to higher computation times. In other words, the low variance of the last 2 maps is caused by the few possibilities an agent can take due to the high number of obstacles.

Table 15: Mean of the performance indicators as a function of the map type for distributed planning

Map type	cost difference per agent	run time
Map 1	2.970	0.062
Map 2	3.168	0.071
Map 3	3.879	0.072

Table 16: Standard deviation of the performance indicators as a function of the map type for distributed planning

Map type	cost difference per agent	run time
Map 1	3.873	0.189
Map 2	3.667	0.062
Map 3	4.784	0.062

Next, the start type will be analysed and its influence on the performance indicators. For this, the mean of the results will be presented in Table 17 while the standard deviation in Table 18. Firstly, the mean of run time is the lowest for random start which is explained as, compared to the other cases, the paths it needs to compute are shorter (the goals and starts are closer). On the other hand, the cost difference performance indicator per agent is not as straight forward influenced. It seems that higher values are for double-sided start type. This could be explained by the necessity of the agents to meet in the middle which translates further into more constraints and thus more deviation. Following up, it is observed that the variance of the cost difference per agent is influenced similarly to the mean values (for the same reasons as before). On the other hand, the run time has clearly higher variance for random start that can be explained using the high randomness in the placing of the start and goal locations.

Table 17: Mean of the performance indicators as a function of the start type for distributed planning

Start type	cost difference per agent	run time
Random	2.255	0.039
Single-sided	1.675	0.080
Double-sided	5.703	0.090

Lastly, the local analysis will be done using the contour plots parameters which were explained in subsection 1.3. The results can be seen in Figure 15.

This set of heatmaps are overall very insightful, and display a number of notable patterns. The first key one is as noted on previous methods' heatmaps the strong correlation between the agent count and the cost difference per agent. A strong correlation between the start type and the map type can also be noted, with (in the following order) map 1, map 2, and then map 3 proving more and more constraining

Table 18: Standard deviation of the performance indicators as a function of the start type for distributed planning

Start type	cost difference per agent	run time
Random	2.910	0.175
Single-sided	2.500	0.061
Double-sided	5.065	0.067

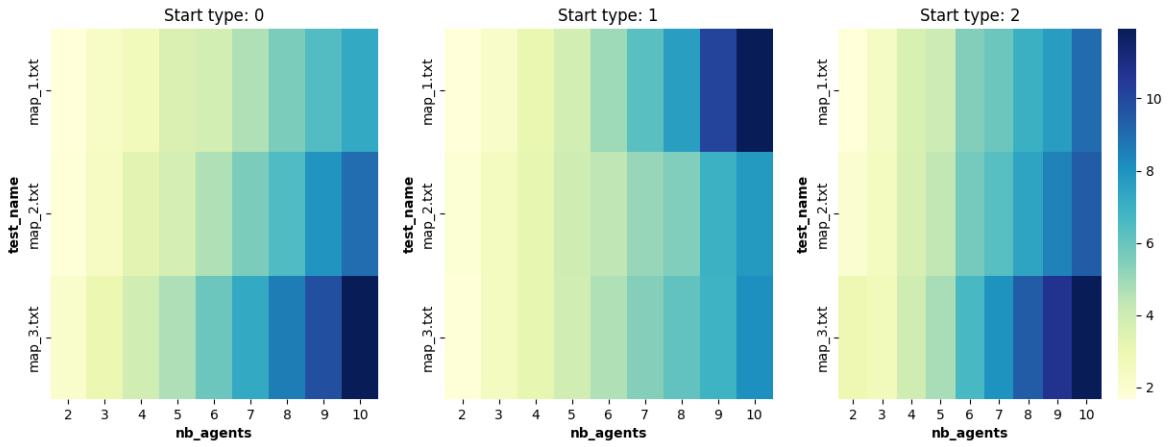

Figure 15: The average cost difference per agent for each map type, agent count, and start type (0 is random, 1 is left to right, and 2 is alternating)

for start type 0 (random). The order for start type 1 (left to right) would be map 2, map 3, and map1. Finally, start type 2 (alternating left and right) seems to follow the same pattern as start 0 (random start).

5 COMPARISON OF THE PLANNING METHODS

Upon having implemented all algorithms and gathered enough data, the last step remaining is comparing their respective performances and properties. To better put in context the three different methods, their respective performance metrics were gathered and plotted. The result can be seen bellow:

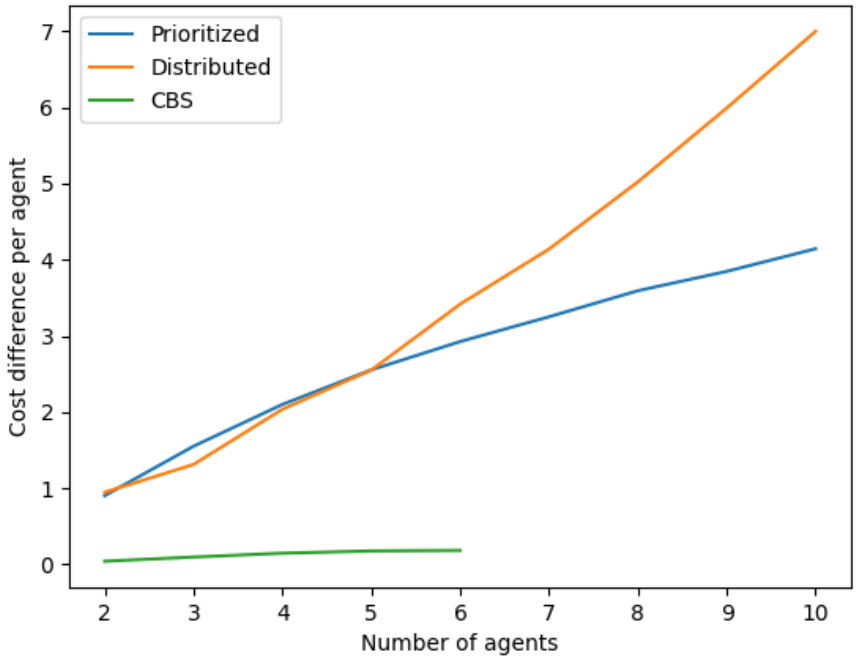

Figure 16: Cost difference per agent as a function of the number of agents for all methods

To start off, it is clear from looking at the cost difference plots that CBS is by far capable of producing the most optimal results. This can be attributed to the fact that the CBS algorithm performs a through search of the entire solution space for the most optimal solution. The slight increase in cost difference can be attributed to the fact that a limit to the number of nodes investigated had to be introduced to restrict computation time (resulting in a sub-optimal solution being returned if found). It can however be assumed that given no limit is put on processing time, CBS would consistently output the optimal solution, with a very small average cost difference close to 0. Comparing the prioritized and distributed methods is more interesting. Initially, the distributed method seems to outperform the prioritized approach. There is however no clear justification for this and we believe further data gathering and inspection would be necessary to confirm this trend. This is further supported by the fact that past 5 agents, distributed performs significantly worst compared to prioritized. This can be attributed to the fact that prioritized, although investigating a smaller subset of the solution state compared to CBS, still perform the planification of all tasks ahead of the run, avoiding inefficient U turns for example through finding the shortest path given its conditions.

The efficiency plot further supports the above mentioned conclusions. Two efficiency plots were generated, to further emphasis each sub components (the cost difference per agent, and the run time per cost per agent). Looking at Figure 17, it is made clear yet again that distributed performances drop the fastest

with the increase in agent count. CBS and prioritized remain on par, with CBS dropping slightly faster due to its significantly worst computational requirements. To get a better grasp of the run time performances, the impact of the run time component was amplified (by squaring it), and is plotted in Figure 18. The resulting plot demonstrates that using this metric, CBS appears to drop off significantly faster, a result of its run time performance scaling poorly with agent count.

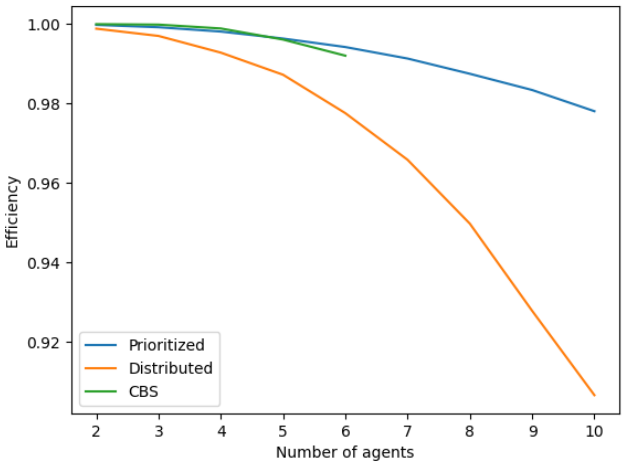

Figure 17: Cost difference per agent as a function of the number of agents for all methods

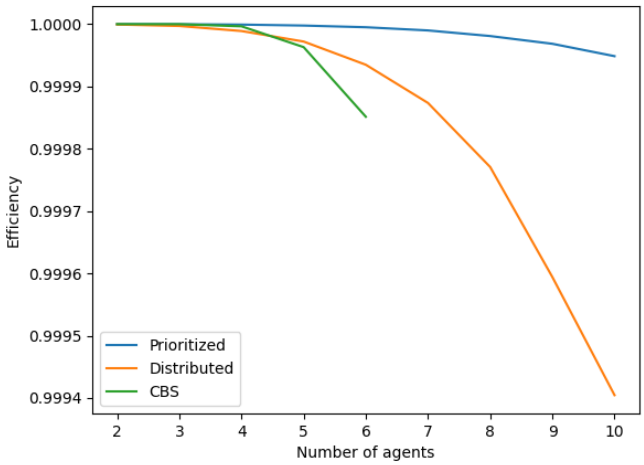

Figure 18: Efficiency as a function of the number of agents for all methods, with the impact of run time amplified

This however is not a fair comparison, as although the results appear comparable, the algorithms have significantly different use-cases, and different properties and limits. While CBS and prioritized both significantly outperform distributed, both perform a full planification of the paths ahead of run time. This however means that in a non-deterministic context (with events such as an agent breaking down), these methods failed entirely, and necessitate a full replanification to find a solution. Distributed on the other hand is much more robust. The lack of pre-planification implies that although the final solution is less optimal, the algorithm is capable of re-routing agents online during a run, and can be ran in a more distributed fashion.

A Appendix

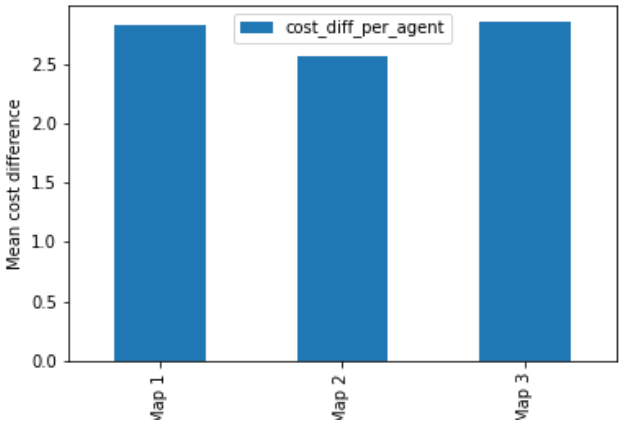

Figure 19: Mean of the cost difference as a function of the map type for A* planning

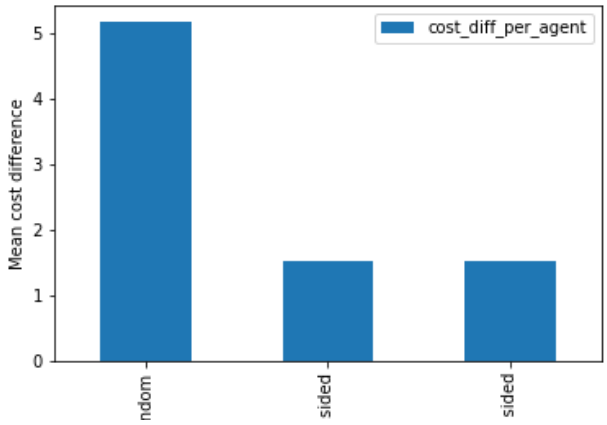

Figure 20: Mean of the cost difference as a function of the start types for A* planning

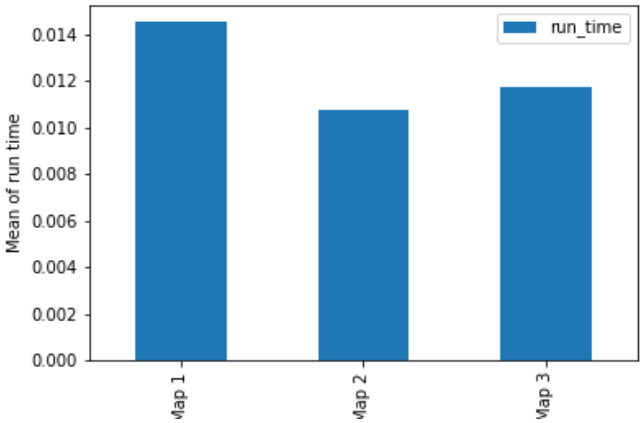

Figure 21: Mean of the run time as a function of the map type

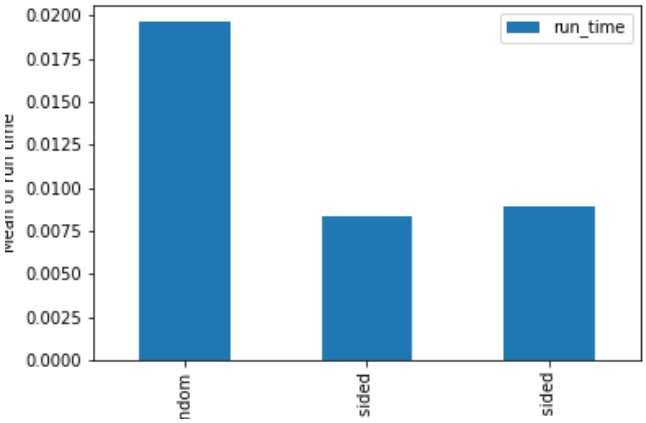

Figure 22: Mean of the run time as a function of the start type

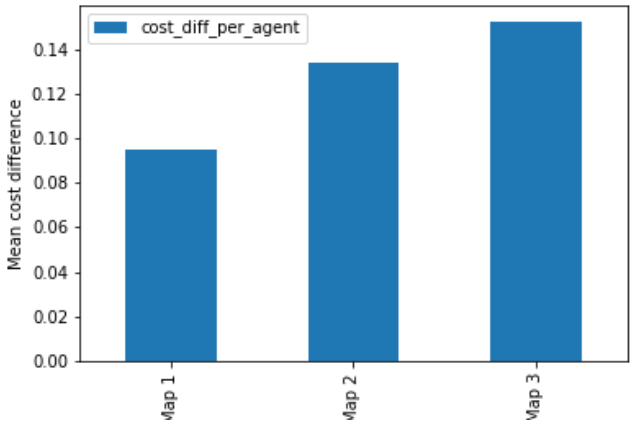

Figure 23: Mean of the cost difference as a function of the map type for CBS planning

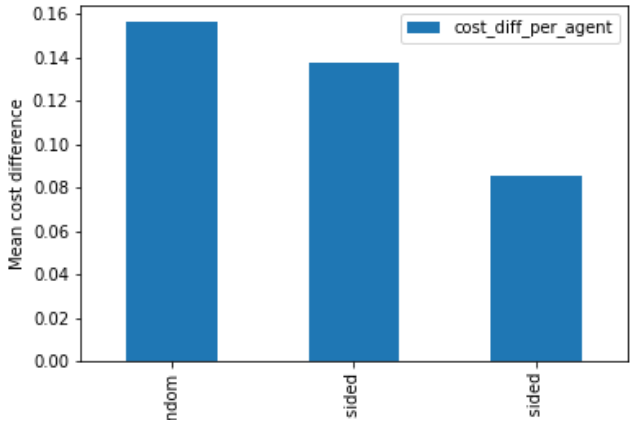

Figure 24: Mean of the cost difference as a function of the start types for CBS planning

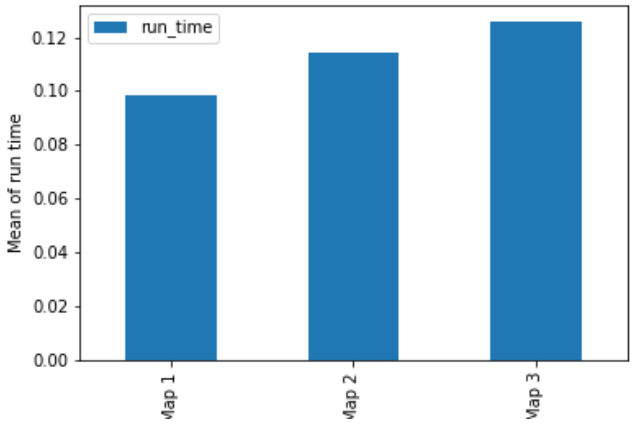

Figure 25: Mean of the run time as a function of the map type for CBS planning

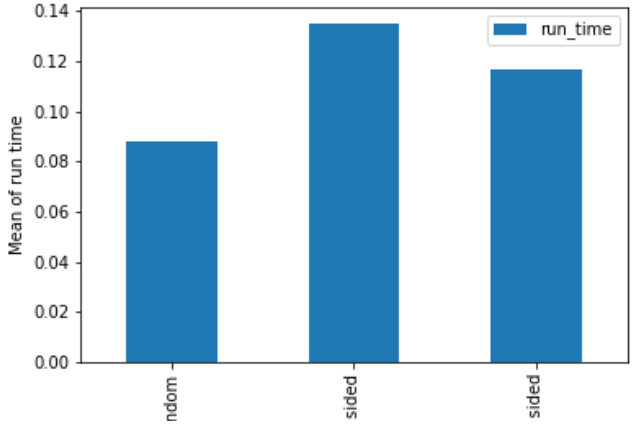

Figure 26: Mean of the run time as a function of the start type for CBS planning

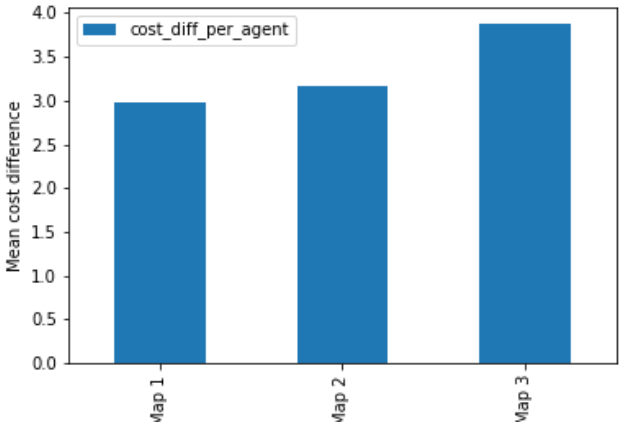

Figure 27: Mean of the cost difference as a function of the map type for distributed planning

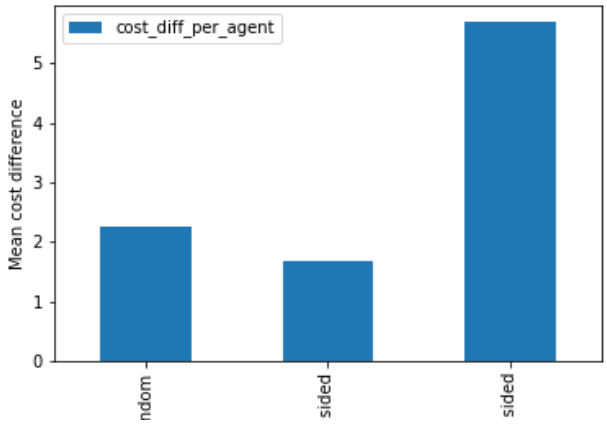

Figure 28: Mean of the cost difference as a function of the start types for distributed planning

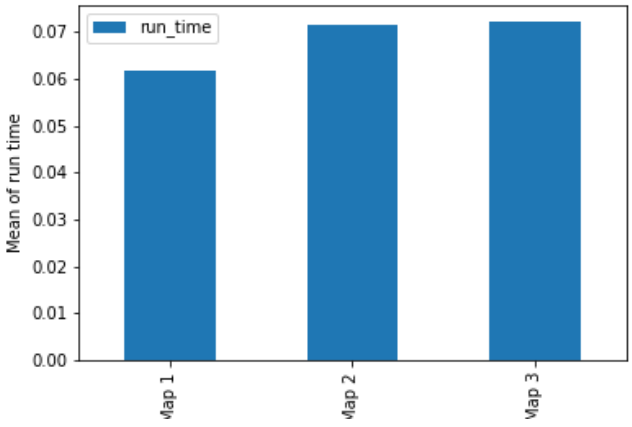

Figure 29: Mean of the run time as a function of the map type for distributed planning

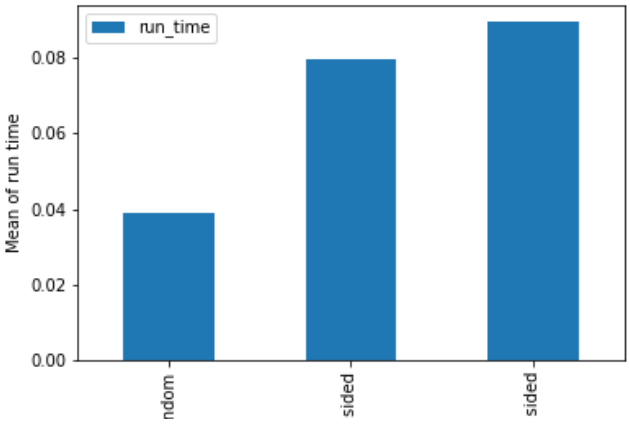

Figure 30: Mean of the run time as a function of the start type for distributed planning

Table 19: Standard deviation of the performance indicators as a function of the number of agents for A* planning

Number of agents	cost difference per agent	run time
2	0.913	0.002
3	1.560	0.003
4	2.126	0.005
5	2.532	0.007
6	2.926	0.009
7	3.243	0.012
8	3.570	0.016
9	3.857	0.019
10	4.139	0.022

Table 20: Standard deviation of the performance indicators as a function of the number of agents for CBS planning

Number of agents	cost difference per agent	run time
2	0.202	0.007
3	0.245	0.177
4	0.272	0.749
5	0.250	1.479
6	0.552	1.136

Table 21: Standard deviation of the performance indicators as a function of the number of agents for distributed planning

Number of agents	cost difference per agent	run time
2	2.600	0.005
3	2.502	0.009
4	3.061	0.015
5	3.164	0.022
6	3.655	0.030
7	4.045	0.039
8	4.301	0.050
9	4.861	0.350
10	5.166	0.072