ME292B Individual Project: Planning

Zaowei Dai 3039643180

2024.2.29

Abstract

This report presents the iterative enhancements made to the baseline A* algorithm applied to 500 diverse scenarios in the CommonRoad benchmark suite. Initially, A* and Greedy Best-First Search (GBFS) were compared, revealing A*'s robustness in finding optimal paths, albeit at a lower success rate (338 out of 500) than GBFS (361 out of 500). Subsequent modifications involved dynamic weighting of the heuristic function in A*, focusing on its adaptability and computational efficiency. The final implementation demonstrated a notable improvement, successfully solving 378 scenarios, with a reduction in timeouts and computational complexity. The findings underscore the significance of heuristic function calibration in improving path planning algorithms.

1. Introduction

The pursuit of efficient and reliable autonomous vehicle path planning has led to the exploration of various algorithms, with the A* algorithm standing as a cornerstone due to its optimality and completeness. However, when faced with the challenge of planning paths across 500 varied CommonRoad scenarios, the standard A* algorithm's limitations became apparent. This project embarked on enhancing A*'s baseline planner to address its computational intensity and inability to solve all scenarios within a finite time. Through a series of strategic enhancements, this report details the evolutionary steps taken to amplify the algorithm's success rate and efficiency, thereby pushing the boundaries of its application in autonomous vehicle navigation.

2.Selection of Baseline Planner: A*

2.1 Description of A*

A* is a best-first search algorithm that finds the shortest path from a start node to a target node in a weighted graph. It uses a combination of two cost functions.

The total cost function is **f(n) = g(n) + h(n)**

g(n): the exact cost of the path from the start node to any node n.

h(n): a heuristic estimate of the cost from n to the goal. This heuristic is problem-specific and should be chosen so that it never overestimates the actual cost.

Advantage: The A* algorithm is characterized by its high search efficiency and its guarantee to find the shortest path when specific conditions are met.

Disadvantage: If the heuristic function h(n) is estimated too optimistically for some scenarios (or does not take enough consideration of the actual path cost), it may lead A* to spend more time exploring unnecessary paths in the process of finding the global optimal solution, especially in scenarios where the path choices are diverse and complex.

2.2 Description of the Greedy Best-First Search (GBFS)

GBFS is a heuristic-based search algorithm that explores a graph by selecting the node that appears to be closest to the goal, as estimated by a heuristic function h(n). Unlike A*, GBFS only uses the heuristic function h(n) for decision-making and does not consider the cost of the path taken to reach the current node.

The total cost function is **f(n) = h(n)**

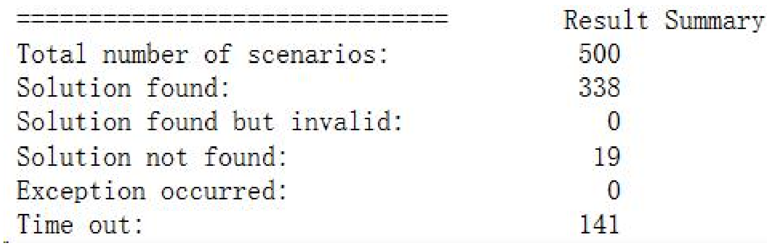
Advantage: GBFS does not take into account the cost of paths already traveled, so it may not find the optimal path, but in some cases the search is faster.

Disadvantage: The GBFS algorithm selects the vertices with the smallest estimated cost h(n) for expansion at each step, ignoring the actual cost of reaching these vertices, and thus is not guaranteed to find the optimal path.

2.3 Comparison between A* and GBFS

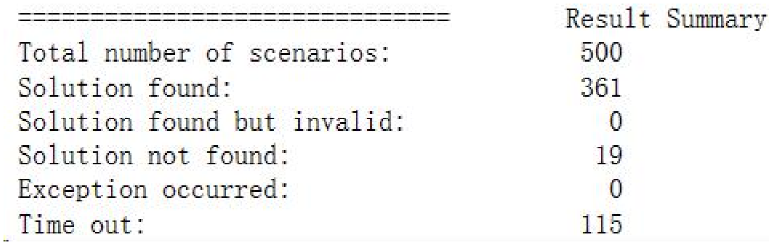
2.3.1 Test results in 500 scenarios

Through Parallel Batch Processing, we get the results as blow:



```
==============================        Result Summary
Total number of scenarios:             500
Solution found:                        338
Solution found but invalid:              0
Solution not found:                     19
Exception occurred:                      0
Time out:                              141
```

Figure 1: Test result of original A* algorithm



```
==============================        Result Summary
Total number of scenarios:             500
Solution found:                        361
Solution found but invalid:              0
Solution not found:                     19
Exception occurred:                      0
Time out:                              115
```

Figure 2: Test result of original GBFS algorithm

From the figure above, we can see that the **original A*** could only find **338** solutions out of 500 scenarios and there are **141** cases of time out. However, the original GBFS could find **361** solution more than A* with just **115** cases of time out, which could show that:

Theoretically, A* is guaranteed to find the optimal path. But in the In the case of limited search efficiency and computation time, GBFS's straightforward approach, focusing only on the heuristic to prioritize nodes, can make it faster in scenarios where the path cost (g(n)) calculation in A* might introduce significant computational overhead. This efficiency could enable GBFS to explore more paths in a given time, potentially finding solutions where A* might not within the same computational budget.

So I'm wondering if I can improve on baseline planner A*, so that the algorithm can guarantee to find the optimal path as much as possible, but also speed up the computation to avoid too much time out.

3. Preliminary improvement based on A*:

3.1 Heuristic function with weighting factors

From the above comparison between A* and GBFS, it is clear that A* loses: in trying to minimize the total cost f(n), it may explore more nodes, which means that it requires more computational resources. With limited resources, A* may not be able to find a solution because it times out or reaches the limit of its computational power.

A*: f(n) = g(n) + h(n)

GBFS: f(n) = h(n)

From the perspective of the cost function, A* needs to consider not only the estimated cost of the current node to the goal point h(n), but also weigh the actual cost g(n) of each node, resulting of not going directly in the direction of the fastest goal point. Differently, since GBFS only uses to consider the estimated cost h(n) from the current node to the target node, it has a faster search speed.

Therefore, we can add a weight w(w>1) in front of the heuristic function in A*, which can increase the weight of the heuristic information h(n) in the actual cost f(n) of the whole path:

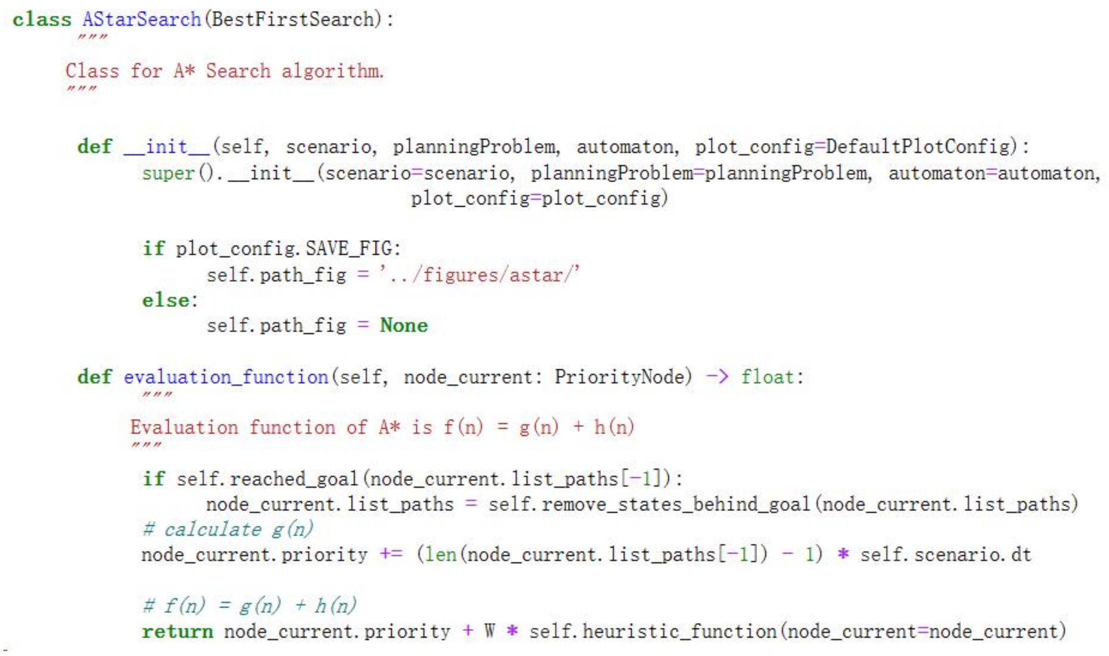
A* with weighted heuristic function: f(n) = g(n) + w * h(n), w > 1

In this case, the increase in weight of h(n) makes the A* algorithm somewhat closer to the behavior of GBFS. The improved A* will focus more on moving directly towards the goal rather than carefully weighing the cost of each step. This may allow the algorithm to find the goal after exploring fewer nodes, thus speeding up the search in some scenarios.

3.2 Test results in 500 scenarios

In the code, we add the w variable in front of h(n).

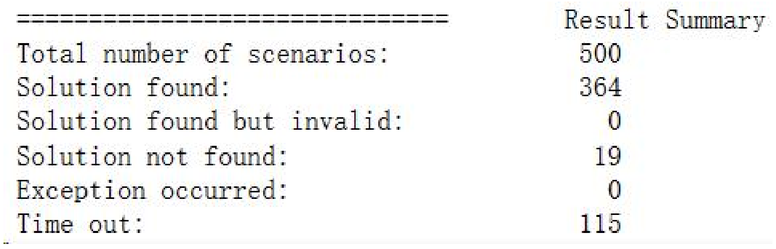
Then through Parallel Batch Processing, we tested at different w's values as below:



```python
class AStarSearch(BestFirstSearch):
    """
    Class for A* Search algorithm.
    """

    def __init__(self, scenario, planningProblem, automaton, plot_config=DefaultPlotConfig):
        super().__init__(scenario=scenario, planningProblem=planningProblem, automaton=automaton,
                         plot_config=plot_config)

        if plot_config.SAVE_FIG:
            self.path_fig = '../figures/astar/'
        else:
            self.path_fig = None

    def evaluation_function(self, node_current: PriorityNode) -> float:
        """
        Evaluation function of A* is f(n) = g(n) + h(n)
        """
        if self.reached_goal(node_current.list_paths[-1]):
            node_current.list_paths = self.remove_states_behind_goal(node_current.list_paths)
        # calculate g(n)
        node_current.priority += (len(node_current.list_paths[-1]) - 1) * self.scenario.dt

        # f(n) = g(n) + h(n)
        return node_current.priority + W * self.heuristic_function(node_current=node_current)
```

Figure 3: Code of A* with weighted h(n)

Take w = 4 as an example:



```
================================        Result Summary
Total number of scenarios:               500
Solution found:                          364
Solution found but invalid:                0
Solution not found:                       19
Exception occurred:                        0
Time out:                                115
```

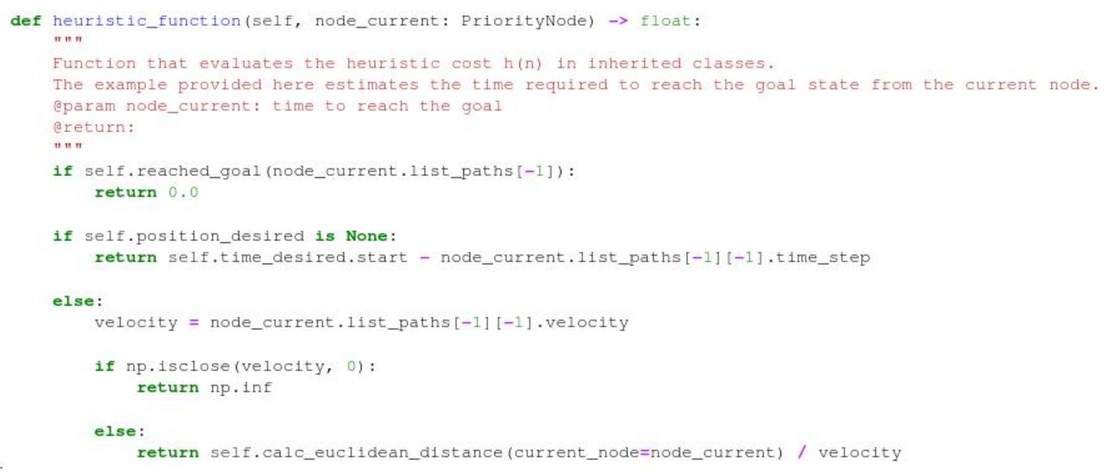
Figure 4: Test result of A* algorithm with weighted heuristic function (w = 4)

From the result, it can be seen that **improved A*** is able to find **364** solutions out of 500 scenarios after assigning a weight of **w = 4** in front of the heuristic function, which exceeds **original A*** and **GBFS's** results: **338 & 361** solutions out of 500 scenarios. And the case of time out is decreased from **141** to just **115**.

It is successfully demonstrated that giving weight coefficients in front of the heuristic function h(n) can greatly improve the search speed of the algorithm under the premise of guaranteeing the ability to find the optimal path, and finally greatly reduce the occurrence of time out.

4. Further improvement based on A*

4.1 Improvement of the kind of the heuristic function h(n)



```
def heuristic_function(self, node_current: PriorityNode) -> float:
    """
    Function that evaluates the heuristic cost h(n) in inherited classes.
    The example provided here estimates the time required to reach the goal state from the current node.
    @param node_current: time to reach the goal
    @return:
    """
    if self.reached_goal(node_current.list_paths[-1]):
        return 0.0

    if self.position_desired is None:
        return self.time_desired.start - node_current.list_paths[-1][-1].time_step

    else:
        velocity = node_current.list_paths[-1][-1].velocity

        if np.isclose(velocity, 0):
            return np.inf

        else:
            return self.calc_euclidean_distance(current_node=node_current) / velocity
```

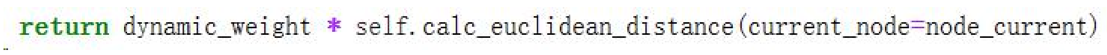
Figure 5: Initial code of h(n) of A*

From the initial heuristic function h(n) as above, we can find that it use the **Euclidean distance from the current node to the target point / speed of the current node** to estimate the **time** needed for the current node to reach the target.

But this way of estimating the cost is shallow-sighted because the speed of the car is changing all the time throughout the search. If the speed of the car changes a lot afterward, then this approach is likely to be inaccurate for the time needed to get from the current node to the target.

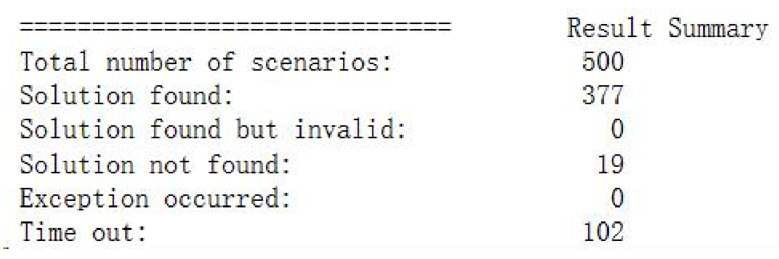
For an example, It is assumed that when the car travels to the next node based on the current minimum time spent path selected by the algorithm, it needs to reduce its speed to a great extent because of obstacles, traffic light moments, or road conditions, thus greatly increasing the time from that path point to the goal, making the selected route not optimal. And this increases the computational complexity and is not conducive to finding the optimal path in finite time.

Therefore, we directly use the **Euclidean distance** for the heuristic function and test at a weighting factor of w = 4:



```
return dynamic_weight * self.calc_euclidean_distance(current_node=node_current)
```

4.2 Test results in 500 scenarios



```
===============================        Result Summary
Total number of scenarios:              500
Solution found:                         377
Solution found but invalid:               0
Solution not found:                      19
Exception occurred:                       0
Time out:                               102
```

Figure 6: Test result of A* algorithm with euclidean_distance for h(n) and w = 4

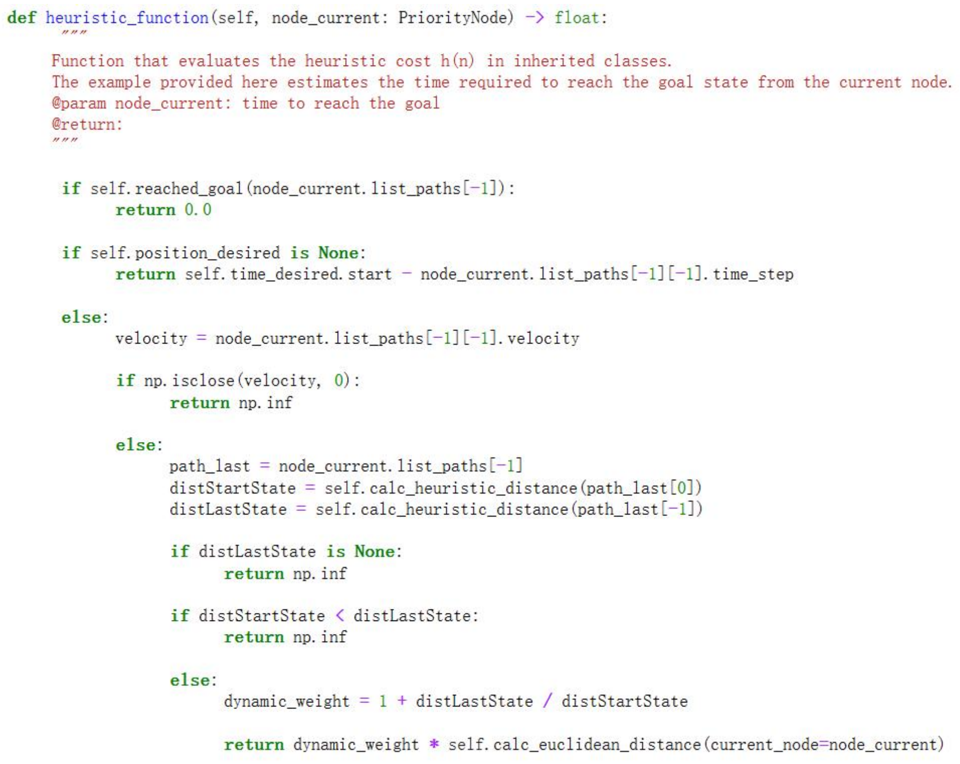
From the result, it can be seen that **further improved A*** is able to find **377** solutions out of 500 scenarios after using euclidean_distance as h(n) with a weight of **w = 4,** which exceeds **improved A*'s** results: **364** solutions out of 500 scenarios. And the case of time out is decreased from **115** to just **102**. And in the challenge board of **CommonRoad**, this algorithm has **390** successful solutions.

5. Final improvement based on A*

5.1 Heuristic function with dynamic weighting factors

In the initial improvement, we used a fixed single-weight w. Although it could guarantee to speed up the algorithm's search at the beginning of the search, it would make the algorithm rely too much on the risk of potentially inaccurate heuristic information when approaching the goal; or when the weight w is too large, the paths around the current node all have similar expenses, thus failing to find a feasible optimal path.

Thus, we can use a dynamic weighting factor in evaluating the heuristic cost of the nodes in the A* algorithm. This dynamic weight is based on the **ratio** of the **heuristic distance from the start state to the current state** to the **heuristic distance from the start state to the last state**. Specifically in code, we will use two functions: **distStartState** & **distLastState** to express the dynamic weight: **1 + distLastState / distStartState.** Here the dynamic weight changes as the path progresses as it depends on the distance already covered and the estimated distance remaining.

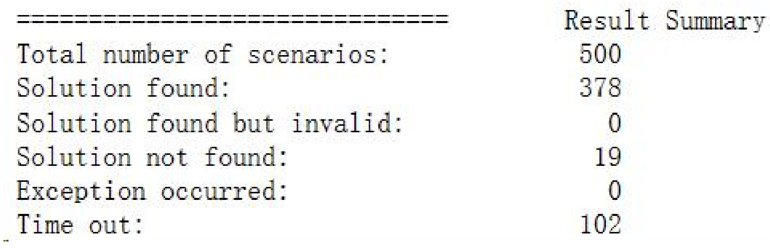


```python
def heuristic_function(self, node_current: PriorityNode) -> float:
    """
    Function that evaluates the heuristic cost h(n) in inherited classes.
    The example provided here estimates the time required to reach the goal state from the current node.
    @param node_current: time to reach the goal
    @return:
    """

    if self.reached_goal(node_current.list_paths[-1]):
        return 0.0

    if self.position_desired is None:
        return self.time_desired.start - node_current.list_paths[-1][-1].time_step

    else:
        velocity = node_current.list_paths[-1][-1].velocity

        if np.isclose(velocity, 0):
            return np.inf

        else:
            path_last = node_current.list_paths[-1]
            distStartState = self.calc_heuristic_distance(path_last[0])
            distLastState = self.calc_heuristic_distance(path_last[-1])

            if distLastState is None:
                return np.inf

            if distStartState < distLastState:
                return np.inf

            else:
                dynamic_weight = 1 + distLastState / distStartState

                return dynamic_weight * self.calc_euclidean_distance(current_node=node_current)
```

Figure 7: Final code of dynamic weighted h(n) of A*

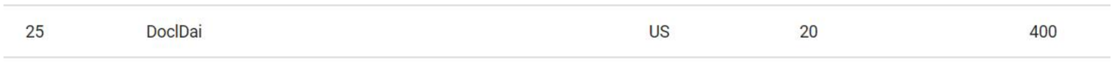
The benefits are: 1. Dynamic weights allow the heuristic function to self-adjust during the search. As the path approaches the goal, the weights may decrease, thus reducing the dependence on the heuristic information. 2.Algorithms can find a balance between finding high-quality paths (i.e., shorter paths) and searching efficiently (i.e., finding solutions quickly). It may be preferable to move quickly in the early stages of the path and more cautiously as the goal is approached to ensure the feasibility of the path.

5.2 Final Test results in 500 scenarios



```
==============================        Result Summary
Total number of scenarios:              500
Solution found:                         378
Solution found but invalid:               0
Solution not found:                      19
Exception occurred:                       0
Time out:                               102
```

Figure 8: Test result of A* algorithm with euclidean_distance for h(n) and dynamic weight



25	DoclDai	US	20	400

Figure 9: CommonRoad Ranking of dynamic_weighted A*

From the result, it can be seen that final **improved A*** is able to find **378** solutions out of 500 scenarios after using euclidean_distance as h(n) with dynamic weight, which exceeds original and improved **A*'s** results: **338**、**364**、**377**. And the case of time out is decreased from **141** to just **102**. And in the challenge board of **CommonRoad**, this algorithm has **400** successful solutions, which is even more than further improved A* with **390** successful solutions.

Overall, the introduction of dynamic weights provides additional flexibility and adaptability to the heuristic function, which may help to improve the performance of the algorithm in complex or uncertain scenarios. Besides, the improvement could reduce the weight of the heuristic information when close to the goal can help the algorithm avoid falling into a local optimum solution, especially in complex scenarios.

6. Conclusion

In summary, the strategic enhancements made to the A* algorithm for the CommonRoad scenarios have led to significant improvements in its problem-solving capabilities. By transitioning from a speed-adjusted heuristic to a purely distance-based heuristic, and introducing dynamic weighting, the algorithm's ability to solve complex scenarios increased, culminating in a success rate of **378** out of 500. This not only demonstrates the value of heuristic function refinement but also highlights the potential for further adaptive methodologies in path planning. With reduced computational demands and improved performance, this work lays a foundation for future advancements in autonomous navigation systems.

Reference:

[1] Xiang, D., Lin, H., Ouyang, J. et al. Combined improved A* and greedy algorithm for path planning of multi-objective mobile robot. Sci Rep 12, 13273 (2022). https://doi.org/10.1038/s41598-022-17684-0

[2] Ji, X., Feng, S., Han, Q. et al. Improvement and Fusion of A* Algorithm and Dynamic Window Approach Considering Complex Environmental Information. Arab J Sci Eng 46, 7445–7459 (2021). https://doi.org/10.1007/s13369-021-05445-6

[3] X. Jing and X. Yang, "Application and Improvement of Heuristic Function in A* Algorithm," 2018 37th Chinese Control Conference (CCC), Wuhan, China, 2018, pp. 2191-2194, doi: 10.23919/ChiCC.2018.8482630.