Smoother robot control with the variants of A* planning algorithm

Zehui Li zl432@cantab.ac.uk

I. INTRODUCTION

Mobile robot control consists of two steps: firstly, the **planning algorithm** generates a trajectory from the starting point to the destination while avoiding obstacles along the way; secondly, a path following algorithm (**control algorithm**) enables the robot to follow the generated path. In this report, we explore both planning and control sides of the mobile robot control system, aiming to provide smoother robot control.

On the planning side, variants of A* algorithms are evaluated in order to create more predictable paths with a lower cost than sampling algorithms and potential field methods. A* is one of the most used combinatorial planning algorithms, in this report, we increase the performance of basic A* algorithm by applying various techniques, including **beam search**, **dynamic weighting**, and **bidirectional search**.

For the controlling algorithm, I use pure pursuit: PD control algorithm. Furthermore, I try to model the robot as a plant and build the controller in the frequency domain. The performance of controllers is compared against each other using proper metrics, such as rise time, settling time, and steady state error. The code for this project can be accessed through github[1] .

II. BACKGROUND

In this section, we will review planning algorithms and controlling algorithms, followed by a description of the framework we used for simulation and the scope of our research.

A. Planning Algorithms

In general, planning algorithms fall into three categories: sampling, potential field, and combinatorial algorithms. Sampling algorithms create paths by exploring the map in a probabilistic manner, in which only a part of the map is explored; as a result, the created path is not optimal. Example of sampling algorithms are rapidly-exploring random trees [1] (rrt) and rrt* [2]; both of them have the advantage of fast exploration, but the optimality is not guaranteed. Potential field method can efficiently generate paths to the destination for any given starting position, while the generated path may lead the robot to a locally optimal point.

By contrast, combinatorial algorithms can overcome the sub-optimality problem of sampling methods. This report will study one type of combinatorial algorithms - A* algorithm, which guarantees the optimality of generated path with the disadvantage of extra memory cost and higher time complexity. Mobile robot often requires real-time path planning; therefore, A* algorithm needs to be optimized in order to be used. In this report, I apply multiple optimization techniques, such as beam search, dynamic weighting, and bidirectional search, to improve the time complexity of A* algorithm. The performance of optimized A* algorithms is then evaluated in the simulated environment. Simulation is conducted using Python, and the visualization tool is created based on the framework provided in the second assignment.

[1]The instruction on how to use the code is given in the appendix VI. The link for accessing the code is https://github.com/Zehui127/MobileRobot.

B. Path-following Algorithms

With the generated path from the planning algorithm, a controlling algorithm will enforce the robot to move along the trajectory. Common path-following algorithms include **pure pursuit: PD control** and **geometric method**. **Pure pursuit: PD control** adjusts the pose of the robot according to the error between the current position of the robot and the reference position. At the same time, the geometric method computes the required instantaneous curvature for the robot to fit the given trajectory.

In this project, **PD control** is implemented. Furthermore, in order to build a controller in the frequency domain, we simplify the robot as a plant with a simple transfer function, which is then used for creating a controller with the Matlab control system toolbox.

III. IMPLEMENTATION OF EFFICIENT A* PATH PLANNING

In this section, the implementation details of A* algorithms are provided. In particular, three optimization techniques (dynamic weighting, bidirectional search, and beam search) are detailed. The evaluation of Vanilla A* and improved variants of A* will be presented in the next section.

A. Vanilla A*

As algorithm-1 shows, Vanilla A* first builds an **open list**, starting from the initial node, and then it pops the candidate node with the lowest cost according to some heuristics one by one until the destination has been reached. The most important bottleneck is the pop operation and the operation of checking the existence of a node in the open list. If a dictionary is used for the open list, the **checking-existence** operation will have a constant time complexity, but the **pop** operation will take the $O(n)$ time. Alternatively, if a heap is used for the open list, the pop operation will have $O(\log(n))$ time complexity while the complexity checking-existence operation will be linear.

To improve the efficiency of vanilla A*, I implement a specialized data structure used for the **open list**, which combine the advantage of heap and dictionary, resulting in a constant time complexity in the checking-existence operation and $O(\log(n))$ complexity in the pop operation. In the following part of this section, we will further improve the vanilla A* by reducing the number of neighbours that A* requires to explore before the termination.

Algorithm 1: Vanilla A* Algorithm

Input: start position p_{start}, destination p_{dest}, map m
$list_{open}$ = [];
$list_{close}$ = [];
add p_{start} to $list_{open}$;
while $list_{open}$ *not empty* **do**
  $p_{current} = list_{open}$.pop();
  add $p_{current}$ to $list_{close}$;
  $list_{nodes} = neighbours(node_{current})$;
  for $i \leftarrow 0$ **to** 7 **do**
    $p_{nei} = list_{nodes}(i)$;
    if $p_{nei} = p_{dest}$ **then**
      return p_{start}, p_{nei};
    end
    if $p_{nei} \in list_{close}$ **then**
      continue;
    end
    `/* Dist() is the heuristics used for the cost */`
    $g = cost(p_{current}) + Dist(p_{current}, p_{nei})$;
    $h = Dist(p_{nei}, p_{dest})$;
    $p_{nei}.cost() = g + h$;
    if $p_{nei} \notin list_{open}$ **then**
      add p_{nei} to $list_{open}$;
    else
      if $p_{nei}.cost() < list_{open}.cost(p_{nei})$ **then**
        update($p_{nei}, list_{open}$);
      end
    end
  end
end

B. Dynamic Weighting

Dynamic weighting [3] is a simple strategy to reduce the size of searching space by modifying the cost function. Instead of having a cost function as in the equation-1, a weight term is added to h, resulting in a new formulation for the cost function in equation-2.

$$cost = g + h \tag{1}$$

$$cost = g + w * h \tag{2}$$

where w is the weight and $w >= 1$. w is set to a relatively large value at the beginning of the iteration, and then w decays through the process of exploration, where the decay function is defined in equation-3.

$$w_t = 1 + \sigma - \sigma * d_t/N \tag{3}$$

where d_t is the current depth of the search and N is the upper bound of the search depth. σ is a hyper-parameter controlling the value of w.

Dynamic weighting can efficiently reduce the number of points needed to be explored. The intuition of dynamic weighting for A* is that at the beginning of exploration, the cost for each position is dominated by the heuristic term (the distance between the current position and the destination) while at the later stage the cost return to the normal cost function. In this manner, nodes with a higher probability to lead to the destination is prioritized at the beginning of the search.

C. Bidirectional Search

When the goal is given in a search problem, Bidirectional search (BS) can be used to improve the searching speed. In the path planning setting, the goal is often given in advance, and therefore, BS can be naturally applied. To use A* with BS, two A* searching run in parallel; one searches from the start position to the destination while the other starts from the destination and move towards the starting point. When one A* algorithm pops a position $p_{overlap}$ which have already appeared in the close list of the other, the algorithm terminates. Finally, we can reconstruct the solution from $p_{overlap}$.

By using the BS technique, the size of the searching tree can be reduced significantly. In [4], the theoretical boundary of the bidirectional search is analyzed. The reason why bidirectional is effective is that the A* search will generate a searching tree until the goal position is reached. By conducting the BS technique, the large searching tree is broke down into two smaller searching trees. The total number of nodes decreases. In the best case, the time complexity of the algorithm can be reduced from $O(b^n)$ to $O(b^{n/2})$, where b is the branching factor, and n is the depth of search.

When it comes to the implementation, we can conveniently create two separate open lists and close lists, updating two sets of variables simultaneously within each iteration to simulate the parallel behaviour.

D. Beam Search

Beam search is also a generalized technique used by many searching algorithms. When beam search is applied to A*, it improves the efficiency of A* by restricting the size of the open list. To be more specific, beam search only keeps k elements with the smallest cost in the open list, and all the other nodes with the higher cost is abandoned. By doing so, the algorithm will always expand nodes which are close to the target position.

To implement beam search, I use a sorted list as the internal data structure for the open list. The insertion time complexity can be reduced to $O(\log(k))$ with the **bisect** method; updating operation has a time complexity of $O(k)$ while the **pop** operation still has a constant time complexity.

One disadvantage of beam search in A* is that it may lead to local optimality if k is too small. Therefore, the value of k needs to be tuned to maintain the optimality of A*.

IV. Path-following Algorithms

This section describes the PD-control path-following algorithm and the process of building a controller in the frequency domain.

A. Pure Pursuit: PD control

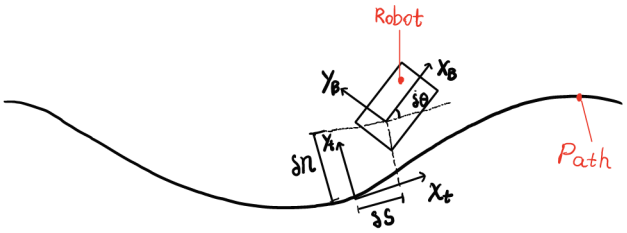

Fig. 1. The errors between the robot and the given trajectory comes from three aspects. (X_t, Y_t) is the reference location while (X_B, Y_B) is the position of the robot. $\delta s, \delta n$, and $\delta\theta$ are the along-track, cross-track, and rotational errors.

Figure-1 shows the setting of Path-following problem. At time t, the error between the robot and the generated path can be measured by three errors. The error along the moving direction is called along-track δs; the error orthogonal to the movement direction is called δn while the difference between the moving direction is called rotational error $\delta\theta$.

PD controller simply use these three types of errors to correct the movement of the robot. A gain matrix G is first defined in equation-4.

$$G = \begin{bmatrix} g_s & 0 & 0 \\ 0 & g_n & g_\theta \end{bmatrix} \tag{4}$$

Then final forward velocity u and the rotational velocity ω can be defined as equation-5.

$$\begin{bmatrix} u \\ \omega \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{x_t}^2 + \dot{y_t}^2} \\ \dot{\theta_t} \end{bmatrix} + G \begin{bmatrix} \delta s \\ \delta n \\ \delta\theta \end{bmatrix} \tag{5}$$

This controller is a PD controller because $\delta\theta$ is related to the derivative of δn and δs.

B. Build Controller in the Frequency Domain

To model the path following controller in the frequency domain is very challenging because the system is not a linear time-invariant (LTI) system. Instead of working with equation-5, I model the robot's behaviour in the simulation environment under the framework of assignment 2. As figure-2 and figure-3 shows, the position of the robot B is decided by the holonomic point P. The controller takes the error E as the input, and then it outputs the holonomic point's velocity V_p; following that the plant will output the position of the robot.

The open loop transfer function is given in equation-6.

$$V_p(t) = m\frac{\mathrm{d}^2 B(t)}{\mathrm{d}t^2} + c\frac{\mathrm{d}B(t)}{\mathrm{d}t} + kB(t) \tag{6}$$

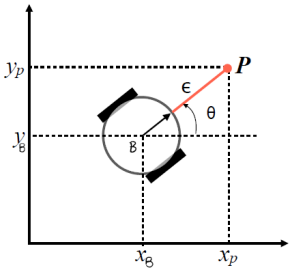

Fig. 2. The position of a differential drive is linked to a holonomic point P.

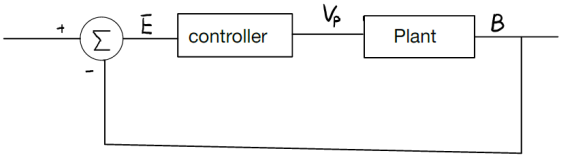

Fig. 3. The simplifies control loop of the robot.The plant simply outputs the position of the robot according to the given velocity of the holonomic point.

With the Laplace transform, the open loop transfer function can be rewritten as equation-7

$$\frac{B(t)}{V_p(t)} = \frac{1}{ms^2 + cs + k} \tag{7}$$

With a PD controller the closed-loop transfer function follows equation-8.

$$\frac{B(t)}{E(t)} = \frac{k_d s + k_p}{ms^2 + cs + k} \tag{8}$$

where k_d and k_p are the parameters for the PD controller.

Similarly, the close-loop transfer function with the phase-lead controller follows equation-9.

$$\frac{B(t)}{E(t)} = \frac{s - z_0}{(s - p_0)(ms^2 + cs + k)} \tag{9}$$

where z_0 is less than p_0.

V. Evaluation

This section presents the evaluation results of path-planning and path-following algorithms separately with different metrics.

A. Evaluation of Improved A* Path Planning

Four algorithms: **Vanilla A***, A* with **dynamic weighting (DW)**, A* with **bidirectional search (BS)**, and A* with **beam search (BeS)** are evaluated against each other.

To evaluate path-planning algorithms, a map with multiple obstacles is created, where the starting position is located at $(-1.5, -1.5)$ and the final position is at $(1.5, 1.5)$ as figure-4 shows. A* algorithm discretizes the map into grids and expands the searching tree in eight directions. In the experiment, the size of the grid is set to 0.2×0.2; the smaller size of the grid will result in a smoother path.

The evaluation result is shown in **table-I**. Two metrics used for the evaluation are the **running time** t of the

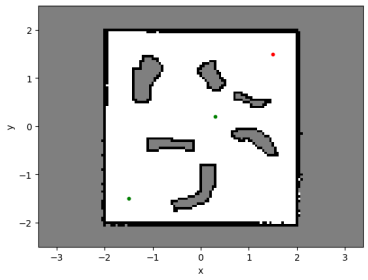

Fig. 4. The raw map where A* algorithms are tested. $(-1.5, -1.5)$ is the starting position while $(1.5, 1.5)$ is the goal position.

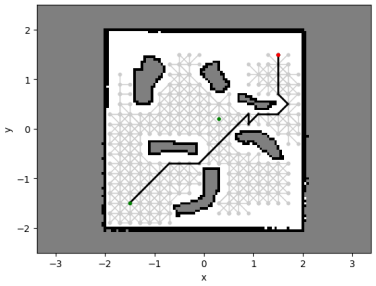

Fig. 5. The searching tree of vanilla A*.

algorithm and the **number of expanded nodes** n. For the time metric, **beam search** achieves the best performance with a running time of 0.95 seconds while the running time of vanilla A* is 1.64 seconds. By contrast, when using the number of nodes as the metric, Bidirectional search has the best performance, outperforming vanilla A* by more than 50 percent. To understand the behaviour of these algorithms, I also study the output of four algorithms qualitatively.

TABLE I

PERFORMANCES OF VARIANTS OF A*

	A*	**DW**	**BeS**	**BS**
Time	1.64	1.27	***0.95***	1.30
Num of Nodes	1160	957	683	***507***

Figure-5 shows the output of vanilla A* algorithm, while Figure-6 and 7 show the output of **beam search** and **bidirectional search**. The grey dots are expanded nodes, and the black line represents the generated path. In comparison to vanilla A*, there are less expanded nodes (grey dots) in the searching tree of BS and BeS.

However, We could see that beam search and bidirectional search improve over vanilla A* in different ways. In bidirectional search, the middle part of the search tree is relatively narrow. This is because the bidirectional search grows two search trees and two trees meet in the middle point. As a result of early termination, two searching trees do not need to expand too much in the middle. When it comes to beam search, the searching tree is uniformly narrower than vanilla A*. This is because beam search enforces the algorithm to expand the nodes which are closer to the destination according to the heuristic. Also, the search tree of the dynamic weighting algorithm is similar to beam search.

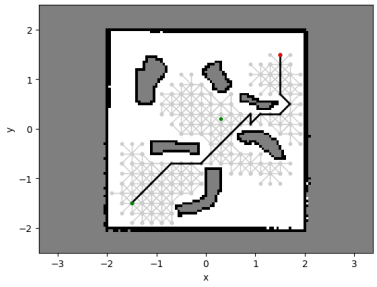

Fig. 6. The searching tree of beam search.

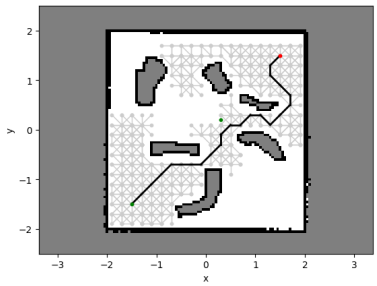

Fig. 7. The searching tree of bidirectional search.

With these observations, we can conclude that three algorithms improve vanilla A* from different perspectives. Noticeably, three algorithms are independent techniques which can be combined. Furthermore, the result of A* which combines three algorithms are shown in figure-8. This combined algorithm has a running time of 0.86 and 348 expanded nodes. In addition, figure-8 use a grid with a smaller size, which shows how reducing the size of the grid can help with generating a smoother path.

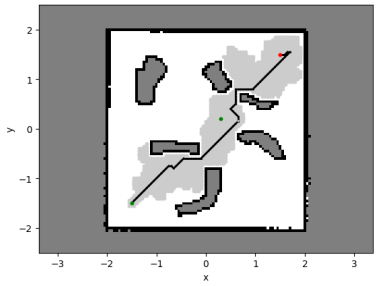

Fig. 8. The searching tree of A* combining three algorithms.

B. Evaluation of the Path-following Algorithms

The evaluation of path-following algorithms is mainly based on three metrics: rise time, settling time, and steady-state error.

- Rise Time: time it takes for the response to rise from 10% to 90% of the steady-state response.
- Settling Time: time it takes for the error to fall within 2% of final steady-state value.
- Steady-state error: the error of the system when it goes to the steady state.

I first use the simulated data from the Gazebo simulator as the shown in figure-9 to estimate the parameters of the open-loop transfer functions (equation-7, and then we can model the plant and tune the parameters of controllers in Matlab.

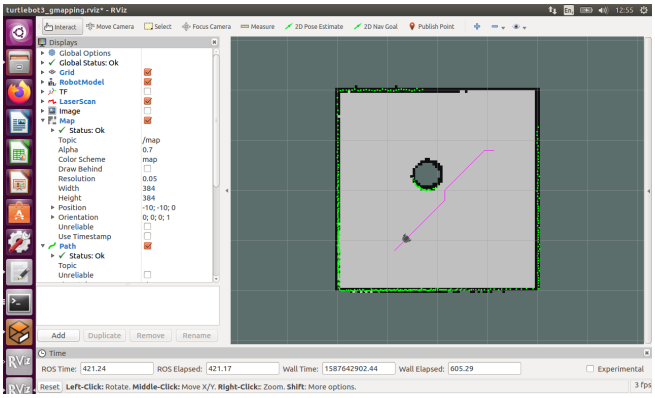

Fig. 9. The simulation of robot with A* path planning algorithm.

With the closed-loop transfer functions defined in section-IV-B, the smoothed change of error along the time without controllers is shown in figure-10. The rise time and settling time of the system are 21.75 and 38.83 seconds, respectively. Also, the steady-state error goes to a fixed positive value instead of zero.

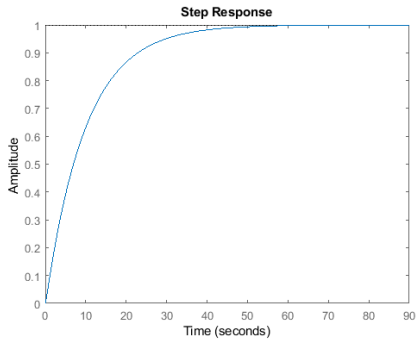

Fig. 10. The change of error along the time without a controller.

PD controller and phase-lead compensator are used to improve the performance of the system. Figure-11 shows the smoothed error plot with PD controller. PD controller effectively reduces the rise time and settling time to 1.5 seconds and 3.73 seconds. Furthermore, the steady-state error is reduced to less than 0.2. This proves that the PD controller is an effective path-following algorithm.

By contrast, the performance of phase-lead compensator is not as good as PD controller. As figure-12 indicates, the

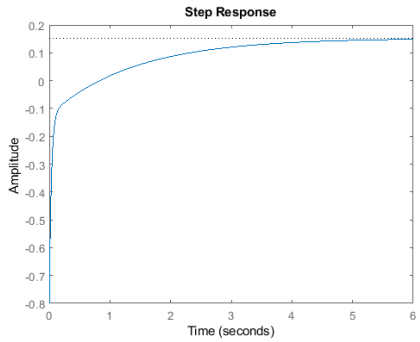

Fig. 11. The change of error along the time with PD controller.

rise time and settling time of the system are 2.17 and 3.75 seconds with a steady-state error of 0.5.

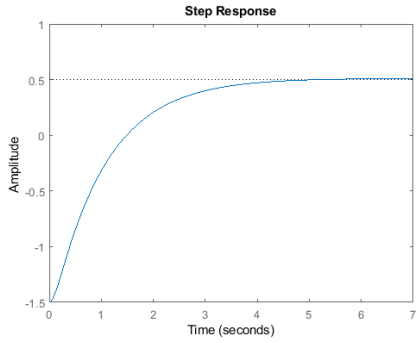

Fig. 12. The change of error along the time with phase-lead compensator.

VI. Discussion and Further Work

In this project, efficient path planning algorithms are implemented by improving upon the vanilla A* algorithm. The evaluation shows that dynamic weighting, beam search, and bidirectional search can boost the performance of A*. This result provides insights into solving the problem of real-time path planning: Beam search and bidirectional search can also be applied to other search-based path-planning algorithms such as rrt and D* search [5]; dynamic weighting can be applied to heuristic path-planning algorithms like rrt*[2]. In addition to three algorithms covered in this report, there are also other techniques which can be used to improve vanilla A*. For example, jump point search (JPS) by [6] can be used in the uniform-cost map, and it can improve the performance of the system by a magnitude of 10. However, the generality of the algorithm is not as good as three techniques used in this project, because JPS requires movement of the object to be symmetric.

Another issue which has not been addressed in this project is the smoothness of the generated path. When discretizing the map, the size of grids will influence the smoothness of the generated path. However, the small size of the grid will be infeasible due to the limit of computational power. As figure-9 shows, the generated path is not as smooth as

potential field method. One solution is to add a filter upon the generated path from A*, and this filter can smooth the path using geometric methods.

APPENDIX

The python code for implementing variants of A* algorithms can be downloaded from the GitHub link. To run the code, simply change the current working directory to **MobileRobot** and run the algorithm with, for example, **python** BeamSearch.py. No external package is required. The parameters, such as grid size can be changed before running.

REFERENCES

[1] LaValle, Steven M. "Rapidly-exploring random trees: A new tool for path planning." (1998).

[2] Khatib, Oussama. "Real-time obstacle avoidance for manipulators and mobile robots." Autonomous robot vehicles. Springer, New York, NY, 1986. 396-404.

[3] Pohl, Ira. "Heuristic search viewed as path finding in a graph." Artificial intelligence 1.3-4 (1970): 193-204.

[4] Kaindl, Hermann, and Gerhard Kainz. "Bidirectional heuristic search reconsidered." Journal of Artificial Intelligence Research 7 (1997): 283-317.

[5] Matsubara, Y., Sakurai, Y. Yoshikawa, M. D-Search: an efficient and exact search algorithm for large distribution sets. Knowl Inf Syst 29, 131–157 (2011). https://doi-org.ezp.lib.cam.ac.uk/10.1007/s10115-010-0336-6

[6] Harabor, Daniel Damir, and Alban Grastien. "Improving jump point search." Twenty-Fourth International Conference on Automated Planning and Scheduling. 2014.