$directPath$: An Expert Path Planning Framework to Handle Environment Knowledge

Romain JACOB
Vehicle Dynamics and Control Lab
University of California BERKELEY, CA, USA
From : ÉCOLE NORMALE SUPRIEURE de Cachan, FRANCE
Email: rjacob@ens-cachan.fr

J. Karl HEDRICK
Professor of Mechanical Engineering
University of California BERKELEY, CA, USA
Email : khedrick@me.berkeley.edu

***Abstract*—The main purpose of this paper is to present a new path planning framework, called $directPath$, aimed to handle *a priori* environment knowledge efficiently. Our algorithm is an expert system designed to specifically exploit that knowledge, and then eventually collaborate with other frameworks in order to improve overall path planning performances.**

It performs a forward search with partial heuristic. It is iterative and can return a partial but safe path when terminated early. Moreover, unlike classical search algorithms, its design allows a quick detection of unfeasible problems.

In this paper, we present the framework structure and illustrate its performances and properties by simulations. We will show that $directPath$ can complete the planning task very quickly, with only minor optimality loss.

I. INTRODUCTION

Since the 1980s, a tremendous amount of work has been done on real-time path planning for autonomous mobile robots. The classic formulation for a path planning problem is the following : Given an initial position and orientation of an agent and a goal position, a path planner will generate a continuous free path starting at the initial position and orientation terminating at the goal position, if such path exists, and report a failure otherwise [1].

Two main approaches are used to deal with this classic path planning problem. There are referred to as global and local approaches.

Global approaches use a mapping of the entire accessible environment to solve the path planning problem. That insures the property of global convergence. The path planning problem will be solved, if feasible. However, building and eventually updating a global map is a heavy computational burden to bear for a robot. That is one of the main challenges for Simultaneous Localization And Mapping (SLAM), which is currently heavily studied [2]–[4].

Local sensor-based path planning frameworks, on the other hand, use latest sensor data to plan future actions in a reactive/adaptive way. Most commonly used approaches include potential field methods (also known as virtual forces [5] [6]), behavior-based systems [7] [8] and fuzzy logic approaches [9] [10].

Although they are usually much simpler to implement than global ones, local planners may get trapped in a local minimum and subsequently follow a diverging path or a loop, while attempting to escape from it. Thus, it is not realistic to rely only on local planners. As a result, researchers are commonly combining different approaches to improve overall performances [11].

Similarly, the $directPath$ algorithm we present here can be coupled with classical frameworks. Indeed, these aim to be useful in real application cases, where we often have only partial or no prior knowledge of the environment. As a result, it led us toward planning frameworks which do not benefit from such *a priori* knowledge when there is some, or at least not efficiently.

Our algorithm is an expert system designed to specifically exploit that knowledge, and then eventually collaborate with other frameworks to optimize overall performances. It aims to quickly detect unfeasible problems and to reduce computation time.

II. PROBLEM FORMULATION

We are considering a standard path planning problem in a known environment. In $\mathbb{R}^2$, given a starting point S, a goal point G, and a set of obstacles $\mathbb{O}$, find the minimum length path $\mathbb{P}$, string of points that connects S and G without intersecting with any obstacle. Obstacles are closed polygons. They can be either convex or non-convex.

To solve this problem, we present a new framework, called $directPath$, which efficiently handles *a priori* environment knowledge. It is able to complete the path planning task on its own, or can be used as a support to other frameworks, in a hybrid manner.

The main notations and functions used in $directPath$ are listed below :

s	Current starting point, for a given iteration ;
g	Current goal point, for a given iteration ;
$G_{(c)cw}$	Subgoal defined by (counter)clockwise wall-following — Refer to Section III-A2 ;
$\overrightarrow{AB}$	Straight-line segment from point A to point B ;
$\overline{AB}$	Length of segment $\overrightarrow{AB}$;
$\mathbb{P} = \{P_1, \ ... \ , P_n\}$	Path, string of points, vertices of the path from P_1 to P_n, composed of straight-lines segments $\overrightarrow{P_iP_{i+1}}$;
$\mathbb{O} = \{O_1, \ ... \ , O_m\}$	Set of obstacles in the environment ;

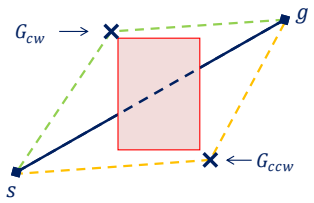

Figure 1. Only two ways to go around an obstacle.

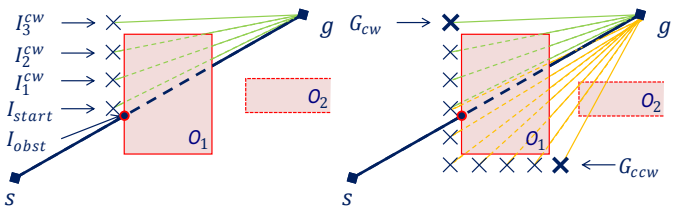

Figure 2. Subgoals definition method : (a) $wallFollow(I_{start}, O_1, cw)$; (b) SUB_g = { G_{cw} ; G_{ccw} }.

$getIntersection(A, B, \mathbb{O})$	Returns a pair $\mathbb{I} = (I, O)$, where • I is the closest of the intersection points between $\overrightarrow{AB}$ and $\mathbb{O}$ from point A, • O is the associated obstacle, if any. Returns $\emptyset$ otherwise ;
$defineSubGoals(s, g, \mathbb{I})$	Returns the set of subgoal options : SUB_g = { G_{cw} ; G_{ccw} } ;
$chooseSubGoal(s, g, \text{SUB}_g)$	Returns chosen subgoal : sub_g ;
$update(s, g, \mathbb{P}, \mathbb{I}, sub_g)$	Returns s and g for next iteration and extends the current path $\mathbb{P}$;
$closestNeigh(A, O)$	Returns the closest safe position around obstacle O from point A ;
$wallFollow(A, O, dir)$	Returns next boundary position of obstacle O, from point A. Direction dir can be clockwise (cw) or counterclockwise (ccw) ;
$isValid(A)$	Returns 1 if A is a walkable state. Returns 0 otherwise.

III. Framework description

This paper aims to present a path planning framework which efficiently benefits from environment knowledge and that can quickly detect unfeasible problems.

We will discuss that $directPath$ can either solve the planning task on its own (Section IV) or be coupled with other path planning frameworks, in a hybrid manner, to improve overall path planning performances (Section V).

A. Framework structure

The $directPath$ framework is built on a simple fact. When one wants to go in a direction and encounters an obstacle, there are only two options. It can go around the obstacle clockwise, or counterclockwise. As illustrated in Figure 1, up to two subgoals can be found, from which the obstacle is no longer on the way. Ultimately, the path planning task can be reduced to a succession of choices between subgoal options.

$directPath$ uses that idea to produce a valid path. First, it tries to go straight to the current goal, g. For any obstacle on its way, subgoal options are generated and a choice is made. The chosen subgoal becomes the new current goal to reach. The previous one is saved and will be retrieved once the subgoal will be reached. The algorithm iterates until the initial goal, G, is reached. Algorithm 1 is $directPath$ pseudo-code.

In the following, we describe in more details the main functions called in $directPath$.

Algorithm 1
directPath$(S, G, \mathbb{O})$

```
1:  s ← S ; g ← G ; P = {S, G} ;                    // Initialization
2:  while s ≠ G do
3:      I ← getIntersection(s, g, O) ;
4:      if I ≠ ∅ then
5:          SUB_g ← defineSubGoals(s, g, I) ;
6:          if SUB_g = ∅ then
7:              return P = ∅ ;                      // No valid path can be found
8:          else
9:              sub_g ← chooseSubGoal(s, g, SUB_g) ;
10:         end if
11:     end if
12:     (s, g, P) ← update(s, g, P, I, sub_g) ;
13: end while
14: return P
```

1) $getIntersection(s, g, \mathbb{O})$: This function takes two points s and g, and a set of obstacles $\mathbb{O}$ as inputs. It checks whether or not $\overrightarrow{sg}$ intersects with any obstacles of $\mathbb{O}$. It returns a pair $\mathbb{I} = (I_{obst}, O_{ind})$, where I_{obst} is the closest intersection point from s (as illustrated in Figure 2(a)). O_{ind} is the obstacle on the way. If no obstacle lies between s and g, $getIntersection()$ returns $\mathbb{I} = \emptyset$.

2) $defineSubGoals(s, g, \mathbb{I})$: This function takes s, g and $\mathbb{I}$ as inputs and returns the set of subgoal options : SUB_g = { G_{cw} ; G_{ccw} }. Details are shown in Algorithm 2.

Notice that the function takes $\mathbb{I}$ (and not $\mathbb{O}$) as input. It implies that only O_{ind} (the first obstacle on the way) is considered while defining subgoals (e.g., in Figure 2 case, obstacle O_2 is ignored). It means that once one reaches a subgoal, it will be able to go straight from it to g if no other obstacle is on the rest of the way. If there are, the algorithm will iterate to reach g, and ultimately to reach the initial goal, G. Refer to Algorithm 1.

In some cases, the $wallFollow()$ function might fail to find a valid subgoal. For instance, if $wallFollow()$ takes the agent outside of the environment boundaries, or in a non-accessible location (e.g., a too narrow corridor). This is how the algorithm detects unfeasible problems. If both clockwise and counterclockwise wall-followings fail, $directPath$ will return an empty path : the obstacle on the way is unpassable. Refer to lines 5-7 in Algorithm 1.

ALGORITHM 2

defineSubGoals$(s, g, \mathbb{I})$

1: $I_{start} \leftarrow closestNeigh(I_{obst}, O_{ind})$;
2: $I_0^{cw} \leftarrow I_{start}$; $k \leftarrow 0$; // cw wall-following
3: **repeat**
4: $I_{k+1}^{cw} \leftarrow wallFollow(I_k^{cw}, O_{ind}, cw)$; $k \leftarrow k+1$;
5: **until** ($getIntersection(I_k^{cw}, g, \mathbb{O}) = \emptyset$ **or** $/isValid(I_k^{cw})$)
6: **if** $isValid(I_k^{cw})$ **then**
7: $G_{cw} \leftarrow I_k^{cw}$;
8: **else**
9: G_{cw} is undefined ;
10: **end if**
11: $I_0^{ccw} \leftarrow I_{start}$; $k \leftarrow 0$; // ccw wall-following
12: **repeat**
13: $I_{k+1}^{ccw} \leftarrow wallFollow(I_k^{ccw}, O_{ind}, cw)$; $k \leftarrow k+1$;
14: **until** ($getIntersection(I_k^{ccw}, g, \mathbb{O}) = \emptyset$ **or** $/isValid(I_k^{ccw})$)
15: **if** $isValid(I_k^{ccw})$ **then**
16: $G_{ccw} \leftarrow I_k^{ccw}$;
17: **else**
18: G_{ccw} is undefined ;
19: **end if**
20: **return** $\text{SUB}_g = \{ G_{cw} ; G_{ccw} \}$

3) $chooseSubGoal(s, g, \text{SUB}_g)$: This function takes s, g and the set SUB_g as inputs and returns the next subgoal to go to : sub_g. When both G_{cw} and G_{ccw} have been successfully defined, we need to make a choice : Which subgoal should we try to reach before heading again to g ?

Depending on the situation, this is performed either by the use of a heuristic or by considering previous choices made. For every obstacle, the first subgoal is always picked by $heuristic()$. If more than one subgoal is needed to get pass the same obstacle, then $history()$ is used. Both sub-fuctions are described below :

a) $heuristic(s, g, \text{SUB}_g)$: This function takes s, g and SUB_g as inputs and returns a pair of path lengths $\mathbb{H} = (H_{cw} ; H_{ccw})$. These lengths are estimates of the euclidean length of paths $\{s, G_{(c)cw}, g\}$, which we want to minimize. They are computed as follows :

$$H_{(c)cw} = l_1^{(c)cw} + d^{(c)cw} + l_2^{(c)cw}$$

Where $l_1^{(c)cw} = \overline{sG_{(c)cw}}$ and $l_2^{(c)cw} = \overline{G_{(c)cw}g}$. $d^{(c)cw}$ is presented in the next paragraph.

It turns out that $l_1^{(c)cw}$ is not sufficient to provide a good estimate of the final length of $\{s, G_{(c)cw}\}$. As illustrated in Figure 3, it does not take into account that we have to go around the obstacle to reach $G_{(c)cw}$. $d^{(c)cw}$ is added to the heuristic to solve that problem. It is defined as the maximal distance we get away from $\overrightarrow{sg}$ during the wall-followings performed in $defineSubGoals()$. It is an estimate of the penalty for having to go around the obstacle, instead of going straight to $G_{(c)cw}$. This is illustrated in Figure 3(b).

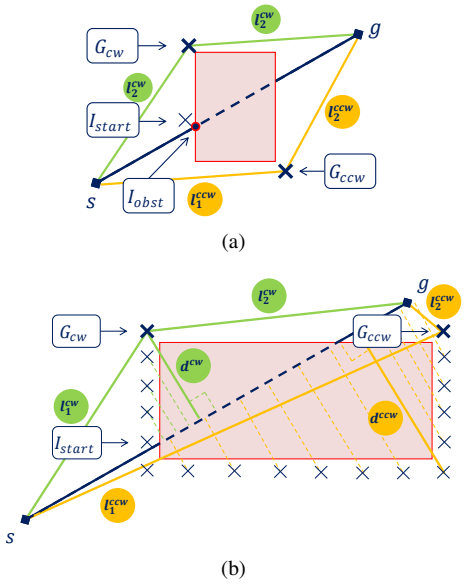

Figure 3. Heuristic definition : (a) Definition of $l_1^{(c)cw}$ and $l_2^{(c)cw}$; (b) Definition of $d^{(c)cw}$.

Simulations have shown this heuristic to be sufficient to pick the *best* subgoal.

b) $history()$: This function takes previous choices made as inputs and returns sub_g. When several subgoals are needed to get pass the same obstacle, the algorithm makes choices consistent with the previous ones.

For example, consider Figure 3(b) case. If we were to choose G_{ccw} (which is not *optimal* since $H_{cw} < H_{ccw}$), we would need to pick another subgoal afterwards to reach it. Then $directPath$ would keep getting pass the obstacle counterclockwise, to provide consistency to the choosing strategy. This avoids to get trapped in a local minimum, which, in our case, would be a loop, a cyclic choices of the same subgoals.

In a maze, one will ultimately get out by always turning in the same direction. Similarly, our strategy assure to get pass any obstacle, if it can be.

Consequences of this approach in term of completeness and optimality will be further discussed in CONCLUSIONS (Section VI).

4) $update(s, g, \mathbb{P}, \mathbb{I}, sub_g)$: This function takes s, g, $\mathbb{P}$, $\mathbb{I}$ and sub_g as inputs. It returns s and g for the next iteration (**while** loop line 2 of ALGORITHM 1). Eventually, it extends the current path $\mathbb{P}$.

Let us assume the current path is :

$$\mathbb{P} = \{S,\ wp_1,\ \ldots\ \overbrace{wp_k}^{s},\ \overbrace{wp_{k+1}}^{g},\ wp_{k+2},\ \ldots\ , G\}$$

a) If $\mathbb{I} = \emptyset$: No obstacle lies between s and g. We only have to increment current starting point s and goal point g.

$$\begin{cases} s & \longleftarrow & wp_{k+1} \\ g & \longleftarrow & wp_{k+2} \end{cases}$$

$$\Rightarrow \mathbb{P} = \{S,\ wp_1,\ \dots\ wp_k, \overbrace{wp_{k+1}}^{s},\ \overbrace{wp_{k+2}}^{g},\ \dots\ , G\}$$

b) If $\mathbb{I} \neq \emptyset$ *:* An obstacle is on the way and a subgoal $sub_g \in \{G_{cw}\ ;\ G_{ccw}\}$ has been returned by $chooseSubGoal()$. In that case, the current starting point s remains, sub_g is set as new current goal point and the path is extended.

$$\begin{cases} s & \longleftarrow & wp_k \\ g & \longleftarrow & sub_g \end{cases}$$

$$\Rightarrow \mathbb{P} = \{S,\ wp_1,\ \dots\ \overbrace{wp_k}^{s},\ \overbrace{sub_g}^{g},\ wp_{k+1},\ wp_{k+2},\ \dots\ , G\}$$

B. Overview of framework properties

In the last section, we described the main functions called in $directPath$. Here, we highlight the main properties of that framework.

Most importantly, if the planning task is unfeasible, $directPath$ terminates as soon as the unpassable element (e.g., an absolute barrier or a too narrow corridor) is encountered. If $defineSubGoals()$ returns an empty set, the algorithm stops and returns an empty path without wasting any more time.

Moreover, the algorithm plans in a forward manner. Once first segments of the path are returned, they will be on the final path. Then, even if the rest of the path is computing, the agent can start tracking the beginning of the path, without any efficiency loss.

Finally, since it is completely built on geometry, $directPath$ is deterministic. This avoids problems that might occur with biased random methods, in which randomness sometimes produces unexpected long computation times.

In the next section, we present an empirical evaluation of some properties and performances of $directPath$.

IV. Empirical Evaluation

Simulations have been performed to illustrate and evaluate two key performances of the framework : computation speed and path efficiency. We used a basic scenario, described in the following, to compare $directPath$ with a standard $A^\star$ framework [12].

A. Simulation scenario

Given a 2 dimensional squared grid world (later referred to as a field) and set of obstacles $\mathbb{O}$, an agent has to find its way between a starting point S and goal point and G. The field is a room-like environment. The agent cannot go out of the boundaries. Allowed actions are cardinal and diagonal moves.

In this scenario, agent shape is not considered : it is limited to a simple dot, which gives a 2 dimensional problem (X and Y coordinates, the agent orientation can be omitted due to that assumption). This model is sufficient since obstacle boundaries can be arbitrary expended to compensate for an agent of nonzero size [13]. Moreover, the dimensional loss doesn't matter since we mainly want to compare performances of $directPath$ and $A^\star$.

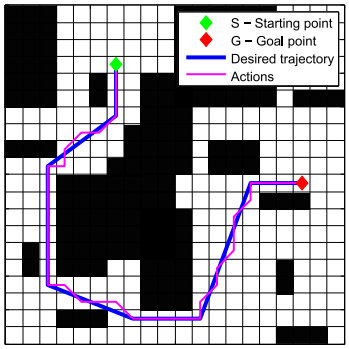

Figure 4. Example of scenario : $N = 250$ and $h_{val} \approx 0.35$

Among others, we focus on two important parameters : the number of walkable states in the field, N (the size of the state space), and the *heuristic value* of a scenario, later referred to as h_{val}. It is defined as the ratio of the euclidean distance and the length of the shortest path between S and G :

$$h_{val} = \frac{\overline{SG}}{length\ of\ the\ optimal\ path}$$

Therefore, the smaller the h_{val}, the harder the scenario is to solve. If $h_{val} = 1$, there is no obstacle between S and G.

Fields were extracted from an open repository of room maps, mazes and random maps [14]. From 150 different maps, we kept parts of different size (e.g., 20 by 20, 100 by 100, ...) to get a wide range of N values. S and G were picked randomly. Path lengths are measured with diagonal moves having cost $\sqrt{2}$ and cardinal ones having cost 1.

To ease the results analysis, we regrouped data points into buckets. Each bucket represents an interval of N and h_{val} (e.g., $1.10^4 < N < 5.10^5$ and $0.4 < h_{val} < 0.5$) and contains at least 10 data points.

B. Framework evaluation

We evaluate $directPath$ performances through two key features : the average computation time to solve the path planning problem, and the efficiency of the returned path, later referred to as η. Simulation results are allocated into N and h_{val} buckets. Figures show mean values over the buckets.

Our framework is compared with a basic $A^\star$ algorithm, which uses euclidean distance to the goal as heuristic, no tie-breaking rule and no embedded way of using *a priori* information about obstacles shape and location.

Simulations were run with an Intel Core i7-3740QM, 2.70GHz CPU and 8GB RAM.

1) Computation time : Computation time results are presented in Figure 5.

The blue lines, associated to the left-hand Y-axis, represent the mean computation time for $directPath$ to solve the problem. Red bars represent standard deviation over each bucket. The green doted-lines, associated to the right-hand Y-axis, represent the time ratio between $directPath$ and $A^\star$.

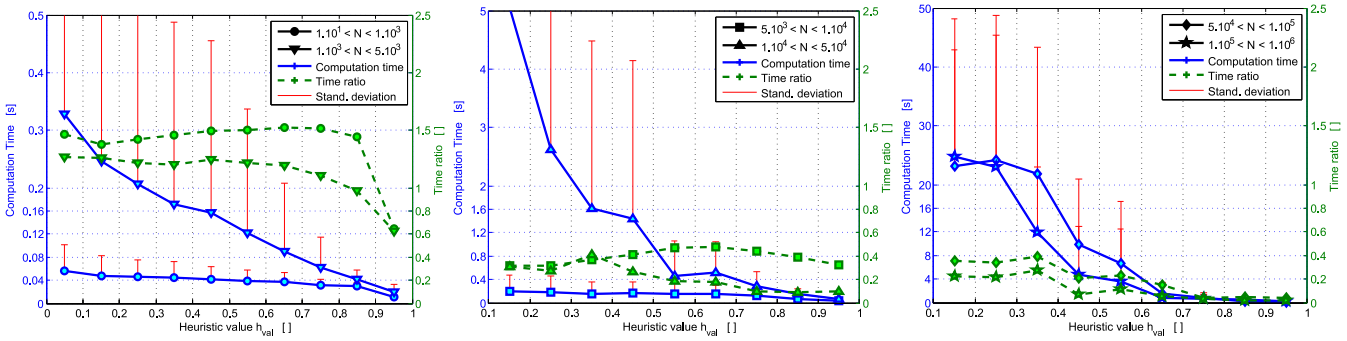

Figure 5. Computation time for different h_{val} and six classes of state space size N

When the ratio is smaller than one, $directPath$ is faster than $A^\star$.

From these results, we can say that $directPath$ completes fairy quickly. For instance, it takes no more than 0.4 second to find a path in fields of 10000 states, even for the most difficult scenario (when h_{val} is very small).

Standard deviation is higher for difficult scenario. This is mainly because we do not consider the optimal path length of a scenario while allocating it in a bucket. In most difficult scenario, path lengths can vary a lot, and so the computation time. Optimal path length would be a pertinent third parameter for further performances evaluation.

Naturally, both algorithms require more time when difficulty increases (when N grows and h_{val} drops). However, time ratio illustrates how the computation time of both algorithms evolves with scenario challenge. It shows that $directPath$ performs better, compared to $A^\star$, when N grows (up to 20 times faster when $N > 10^5$) ; but this benefit diminishes with h_{val}.

2) Path efficiency η : Another key feature of a path planning framework is the optimality of the paths it returns — with respect to a given cost function. In our case, we want to minimize the total length of the path. We define the path efficiency η as :

$$\eta = \frac{length\ of\ the\ optimal\ path}{length\ of\ returned\ path}$$

The optimal path length is obtained by using $A^\star$, which is proven to be optimal [12]. So η provides a good metrix for optimality.

Simulation results are presented in Figure 6.

As expected, the efficiency of $directPath$ decreases with the problem challenge. We mentioned in Section III that suboptimality mainly comes from the decision making process. So naturally, the more decisions $directPath$ makes, the more likely it is to return a suboptimal path.

However, Figure 6 shows a fairly small loss of optimality (most of the time below 5%). The average efficiency of returned paths remains close to 1, which correspond to the optimal path returned by the $A^\star$ algorithm.

We can also note that η seems higher when h_{val} is very small. This is because smaller h_{val} scenario mainly come from

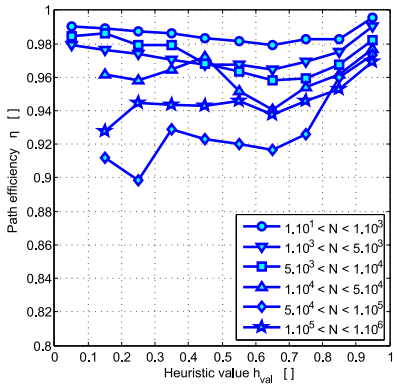

Figure 6. Path efficiency η for different h_{val} and six classes of state space size N

mazes, in which there are in general only few options to reach the goal. Ultimately, if there were only one way to get to the goal, $directPath$ would necessarily return the optimal path, since no suboptimal decisions can lead to the goal.

V. Possible hybrid use of $directPath$

Simulations presented in the last section allow us to say that $directPath$ can provide valid paths quickly, with only small optimality loss. In this section, we point out some possible benefit for a hybrid use of $directPath$ with other classical frameworks.

A. Quick detection of unfeasible problems

As presented in Section III, a key property of $directPath$ design is its ability to quickly detect unfeasible problems.

This can be very useful when the state space size grows. Indeed, for search algorithms like $A^\star$, time to detect unfeasibility is at least of the order of the number of reachable states from the starting point. Which can be huge.

B. Hybrid use with $A^\star$

Consider using $directPath$ to obtain a succession of subgoals from S to G. Then, instead of asking $A^\star$ to produce the complete path, subdivide the planning task, asking for a path to the first subgoal, and iterate until the initial goal G is reached.

This is to say that we use a new (inconsistent) heuristic in a $A^\star$ framework. It would be the euclidean distance to next subgoal. It can also be considered as a scale reduction approach.

That would allow to deal with higher dimensional problems. For instance $directPath$ would return directions for X and Y coordinates of the agent while $A^\star$ would handle its shape and orientation. However, it would inherit the suboptimality of $directPath$.

C. Hybrid use with a potential field approach

Consider using the original path returned by $directPath$ as a guide to avoid local minimum in a potential field framework.

That could be very interesting in applications with complex motion constrains (e.g., kinematics plus dynamics) and in dynamic environments, both aspects well-handled by potential field approaches.

These claims are no proof for a general benefit of using $directPath$ in hybrid frameworks. However, it seems fair to say that further studies on this subject are worth being considered.

VI. Conclusions

$directPath$ aims to efficiently benefit from *a priori* environment knowledge. Therefore, speed is a key feature of the algorithm. We showed in Section IV that the path planning task is completed fairly quickly. Moreover, $directPath$ can quickly detect unfeasible problems. The order of $directPath$ completion time is $O(n \times l)$, where n is the number of obstacles and l the order of their perimeter. This has not be discussed here because of space limitation. Eventually, this would make it suitable for on-line planning.

However, it returns suboptimal paths. As presented in Section III, this comes from both the subgoal definition method ($defineSubGoals()$) and the choosing strategy ($chooseSubGoal()$). Indeed, when subgoals are defined, in order to have simpler sub-problems to solve, we only consider the first obstacle on the way and focus on getting pass this one. That relative myopia can yield to suboptimal decisions. The other main source of suboptimality is the use of $history()$ to choose a subgoal. However, this provides consistency to the choosing strategy of $directPath$. Consistency is commonly used in wall-following based approaches [15] [16] and is known to be a good assurance for completeness (which is the property of solving any feasible problem). It has been proven for similar frameworks [17] [18]. Completeness of $directPath$ remains to be analytically proven. However, during the heavy benchmarking we performed, it always succeeded in complete the planning task, which makes us confident about $directPath$ completeness.

Finally, in Section V, we suggested potential benefits of combining $directPath$ with other frameworks. More generally, it might be worth considering to couple $directPath$ with any framework specially aimed to produce kinematic constrained paths [19] [20].

References

[1] J.-C. Latombe, *ROBOT MOTION PLANNING*. Springer, 1990.

[2] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (slam): Part ii," *IEEE Robotics & Automation Magazine*, vol. 13, no. 3, pp. 108–117, 2006.

[3] M. Wang, W. Wang, J. Xiong, and L. Yan, "A consistent map building method based on surf loop closure detection," in *Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on*. IEEE, 2013, pp. 92–95.

[4] J. Mullane, B.-N. Vo, M. D. Adams, and B.-T. Vo, "A random-finite-set approach to bayesian slam," *Robotics, IEEE Transactions on*, vol. 27, no. 2, pp. 268–282, 2011.

[5] S. S. Ge and Y. Cui, "Dynamic motion planning for mobile robots using potential field method," *Autonomous Robots*, vol. 13, 2002.

[6] Y. Koren and J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*. IEEE, 1991, pp. 1398–1404.

[7] R. C. Arkin, "Motor schema-based mobile robot navigation," *The International journal of robotics research*, vol. 8, 1989.

[8] R. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

[9] H. R. Beom and H. S. Cho, "A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 25, no. 3, pp. 464–477, 1995.

[10] Y. Gong, Y. Liu, and Z. Tang, "Path tracking of unmanned vehicle based on parameters self-tuning fuzzy control," in *Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on*. IEEE, 2013, pp. 52–57.

[11] Z. Fang and L. Zhang, "A multi-objective strategy based on frontier-based approach and fisher information matrix for autonomous exploration," in *Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on*. IEEE, 2013, pp. 96–101.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[13] B. Todor and K. Csorba, "A method for autonomous robot navigation in partly known structured environments," in *Intelligent Systems, 2004. Proceedings. 2004 2nd International IEEE Conference*, vol. 3. IEEE, 2004, pp. 107–112.

[14] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012. [Online]. Available: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[15] M. Katsev, A. Yershova, B. Tovar, R. Ghrist, and S. M. LaValle, "Mapping and pursuit-evasion strategies for a simple wall-following robot," *Robotics, IEEE Transactions on*, vol. 27, no. 1, pp. 113–128, 2011.

[16] X. Yun and K.-C. Tan, "A wall-following method for escaping local minima in potential field based motion planning," in *Advanced Robotics, 1997. ICAR'97. Proceedings., 8th International Conference on*. IEEE, 1997, pp. 421–426.

[17] I. Kamon and E. Rivlin, "Sensory-based motion planning with global proofs," *Robotics and Automation, IEEE Transactions on*, vol. 13, no. 6, pp. 814–822, 1997.

[18] S. S. Ge, X. Lai, and A. A. Mamun, "Boundary following and globally convergent path planning using instant goals," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 35, no. 2, pp. 240–254, 2005.

[19] W. Song and Y. Hong, "Distributed relative attitude formation control of multi-agent systems with directed topology," in *Cyber Technology in Automation, Control and Intelligent Systems (CYBER), 2013 IEEE 3rd Annual International Conference on*. IEEE, 2013, pp. 1–6.

[20] Y. Kuroda and T. Saitoh, "Simultaneous adaptive path planning system for the real world application," in *Robotic Intelligence in Informationally Structured Space, 2009. RIISS'09. IEEE Workshop on*. IEEE, 2009, pp. 46–53.