POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione - Dipartimento di Elettronica e Informazione

Corso di Laurea Magistrale in Ingeneria dell'Automazione

PLANNING DYNAMIC TRAJECTORIES WITHIN THE SEARCH BASED PLANNING LIBRARY

Relatore: Prof. Luca Bascetta

Correlatore: Prof. Matteo Matteucci

Tesi di Laurea di

Ming Li

Matr. 761171

Anno accademico 2011-2012

Abstract

Motion planning for autonomous vehicles is an important topic across the world. Nowadays there exist several classes of algorithms for motion planning, such as grid-based algorithms and sampling-based algorithms. A simple and effective method to plan the path for an autonomous robot based on a YAMAHA Grizzly all terrain vehicle, is here presented, accounting for the anti-rollover problem as well. Grid-based algorithms are a good choice.

This thesis presents a few grid-based algorithms and two ways to decide the vehicle commands. A benchmark among different grid-based algorithms and different methods to compute the vehicle commands is shown, together with an analysis of their pros and cons.

Abstract

Il problema della pianificazione del percorso per veicoli autonomi, ovvero privi di conducente umano è un tema di grande interesse in tutto il mondo. Al giorno d'oggi esistono diverse classi di algoritmi per la pianificazione del moto, quali ad esempio quelli grid-based e sampling-based. In questo lavoro viene presentato un metodo semplice ed efficace per la pianificazione del percorso di un veicolo autonomo basato su un ATV YAMAHA Grizzly. Tale metodo è di tipo grid-based e permette di considerare il problema del ribaltamento del veicolo.

Questa tesi illustra alcuni algoritmi grid-based e due modi per decidere l'azione di controllo del veicolo. Sulla base di un problema benchmark verranno analizzati pro e contro di ciascun algoritmo.

Acknowlegement

First of all, I would like to thank Prof. Luca Bascetta for giving this opportunity to start a thesis in all-terrain-vehicle motion planning. His patient gives me a good start, to understard the main issue of this thesis, to have an idea of YAMAHA Grizzly features.

Secondly, thanks to the guidance of Prof. Matteo Matteucci, he is the one who lead me to the world of search-based planner library. He provide motion primitive file for ATV, and gives me instructions on how to get action increment directly on vehicle states.

Thanks Politecnico di milano for giving me this opportunity to study in master program of automation, and thanks to all the professors who give lectures to us.

Contents

List of Tables

List of Figures

1. Introduction

YAMAHA Grizzly 700 is a four-wheel all terrain vehicle. It is 2.06m in length, 1.18 m in width and 1.24m in height. The ground clearance is 0.275m. The total weight is 294kg without human rider. The electric power steering system, also know as the EPS, contains a DC motor, a gear motor with transmission ratio of 34:1, a torque sensor mounted on the steering shaft and an electronic control unit.

Figure 1.1: YAMAHA GRIZZLY 700

The experiment platform is on this ATV. As shown in Figure 1.2, it is equipted with an industiral PC, 2 laser scanners, a increment encoder, a GPS，etc. A steering control system and a throttle control system have already been developed on this platform.

Figure 1.2 Experiment platform

We want to generate feasible paths in an identified environment according to vehicle dynamics. This requires a reasonable trajectory to avoid obstacles, plus a sequence of reasonable speed and steer angle setpoints to the onboard controller. The main issue of this thesis is to present a reasonable motion planner for our ATV. It is able to generate feasible moving trajectories, speed and steer angle setpoints for the control system.

The thesis starts by providing the background on motion planning and graph search methods. Then a few heuristic-based methods are presented to show how they work in path planning. After that is the Search-Based Planning Library, the thesis will show how to use Navxytheta class and motion primitives in SBPL to generate trajectories. NavATV class, which is a 5 degree of freedoms motion planning class, will show how to use state speed and state steer to generate dynamic feasible paths. Last one is the benchmark between different trajectories using Navxytheta and NavATV.

2. Motion Plannning

The state space for motion planning is a set of possible states that could be reacted by the robot. This will be referred to as the configuration space (C_{space}) [1]. $C_{obstacle}$ is the subset of configuration space including obstacles and unobtainable subset of state space. C_{free} is the remaining subset of C_{space}.

Path Planning consists of finding a sequence of actions that transforms some initial state into some desired goal state, which is to find a sequence of states in C_{free}. In the case of just vehicle motion planning, the state variable could be in simple case [position, yaw] for kinematic car model, or in a complex case [position, yaw, speed, steer] for dynamic car model.

Transition cost(action cost) is a measurement of the state transformation. Lower this transition cost, better the state transformation. A path is optimal if the sum of its transition costs from initial state to goal state is minimal across all possible paths in C_{free}. A planning algorithm is complete if it can find a path in finite time when exists. Similarly, a planning algorithm is optimal if it can always find an optimal path.

Motion planning has several applications, such as robot arm navigation, robot design in CAD software, and vehicle navigation without human driver, as well as applications in other fields, such as animating digital characters, video game AI, and the study of biological molecules. Many algorithms are developed to cope with variants of basic problems. For example, adding nonholonomic dynamic constraint for cars, planes and differential drive robots or dealing with uncertainty problems such as motion uncertainty and sensorless planning.

Generally, there are four different categories of motion planning algorithms, the grid-based search method, the geometric algorithm, the potential field method and the sampling-based

algorithm. Grid-based search method has been chosen in our case for its simplicity to construct cost function. Besides, it is convenient to assign action directly on the states.

Grid-Based Search

Grid-based approaches overlay a grid on configuration space, and assume each configuration is identified with a grid cell[3]. At each cell, the robot is allowed to move to adjacent cells as long as the robot action between them is completely contained within configuration space. Many algorithms, like A*, ARA*, ANA, AD* can be used to find a path in grid environments. An example is given in Figure 2.1.

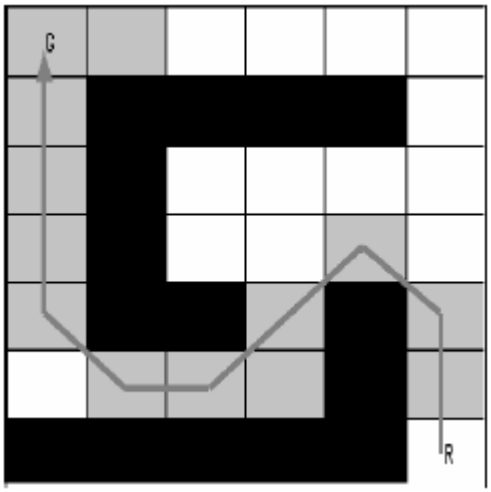


Figure 2.1: Simple example for Grid-based search

These approaches require setting a grid resolution. Search is faster with coarser grids, but the algorithm will fail to find paths through narrow portions of configuration space. Moreover, the number of points on the grid grows exponentially in the configuration space dimension, which makes them inappropriate for high-dimensional problems.

Geometric Algorithm

Visibility graphs may be used to find Euclidean shortest paths among a set of polygonal obstacles in the plane: the shortest path between two obstacles follows straight line segments except at the vertices of the obstacles, where it may turn. So the Euclidean shortest path is the shortest path in a visibility graph that has as its nodes the start and destination points and the vertices of the obstacles[3]. Therefore, the Euclidean shortest path problem may be decomposed into two simpler sub problems: constructing the visibility graph, and applying a shortest path algorithm such as Dijkstra's algorithm to the graph. Figure 2.2 gives an example of the constructed visible map. The start and goal should be “visible” to each other. That is by

connecting all visible vertices, including start and goal points, there should be a series of line segments (including connection on the edge of obstacles) that link start to goal. These paths exist on the perimeter of obstacles.

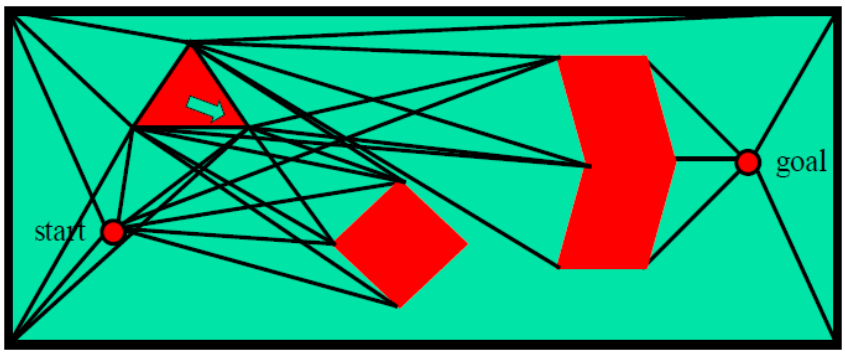


Figure 2.2 Visibility graph

Potential Fields

Another approach is treats the robot's configuration as a point in a potential field that combines attraction to the goal, and repulsion from obstacles. The resulting trajectory, the global minima, is output as the path. This approach has advantages in that the trajectory is produced with little computation. However, potential field method can become trapped in local minima of the potential field, and fail to find a path. Example in Figure 2.3 works like this:

- The goal location generates an attractive potential pulling the robot towards the goal.
- The obstacles generate a repulsive potential pushing the robot far away from the obstacles.
- The negative gradient of the total potential is treated as an artificial force applied to the robot.
- The artificial force controls the robot.

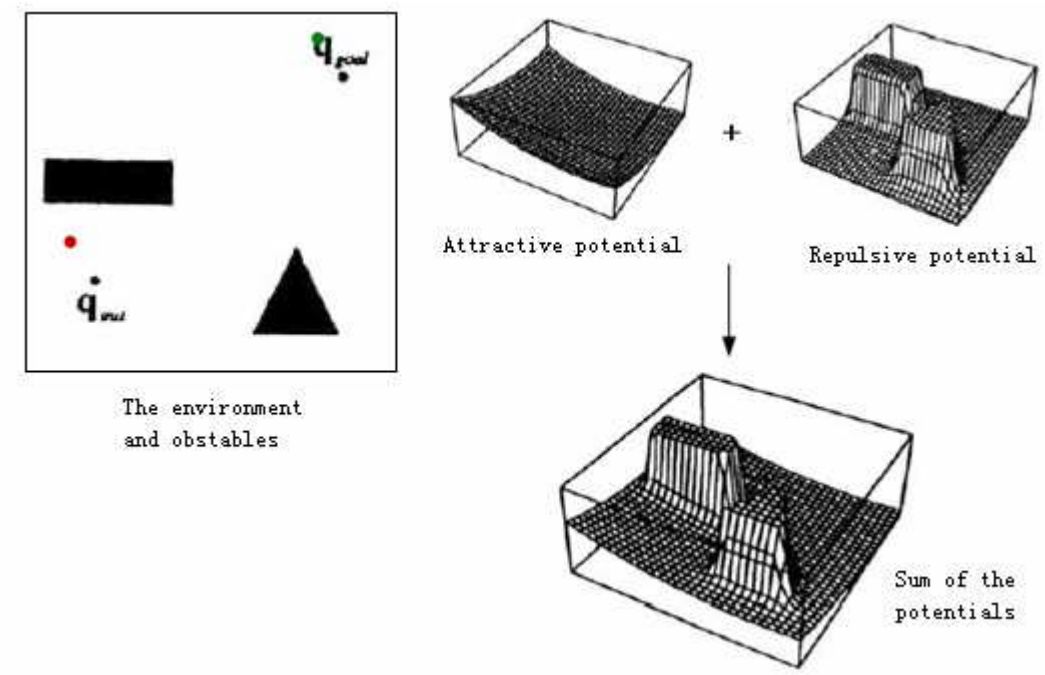


Figure 2.3: Potential field example

Sampling-Based Algorithms

Sampling-based algorithms represent the configuration space with a roadmap of sampled configurations. A basic algorithm samples N configurations in configuration space, and retains those in free space to use as *milestones*. A roadmap is then constructed that connects two milestones P and Q if the line segment PQ is completely in free space. Again, collision detection is used to test inclusion in free space. To find a path that connects start and goal, they are added to the roadmap. If a path in the roadmap links start and goal, the planner succeeds, and returns that path. If not, the reason is not definitive: either there is no path in free space, or the planner did not sample enough milestones.

These algorithms work well for high-dimensional configuration spaces, because unlike combinatorial algorithms, their running time is not (explicitly) exponentially dependent on the dimension of configuration space. They are also (generally) substantially easier to implement. They are probabilistically complete, meaning the probability that they will produce a solution approaches 1 as more time is spent. However, they cannot determine if no solution exists. Like the case in Figure 2.4, it is a Rapidly-Exploring Random Trees method for Dubins car. The Dubins car can move forward, turn both left and right, overall three actions.

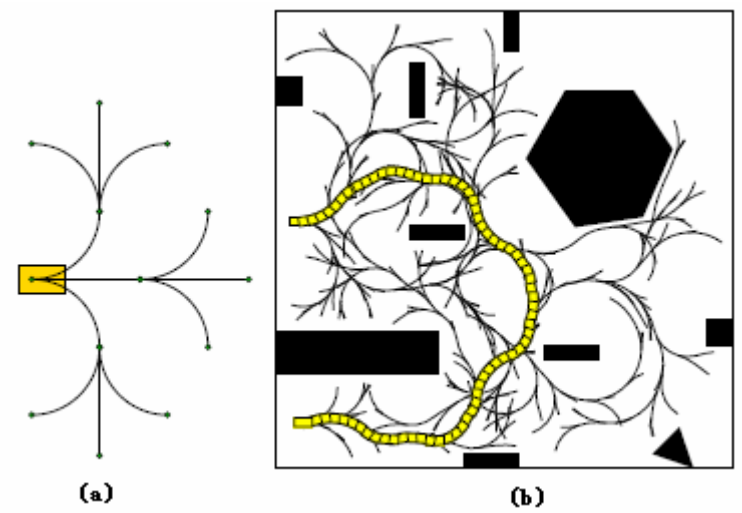


Figure 2.4: Rapidly-Exploring Random Trees method for Dubins car, (a) Two stage expansion, (b) The global tree and generated trajectory

We do not use sampling-based algorithms because it is not possible to guarantee optimal paths.

3. Graph search

3.1 A star

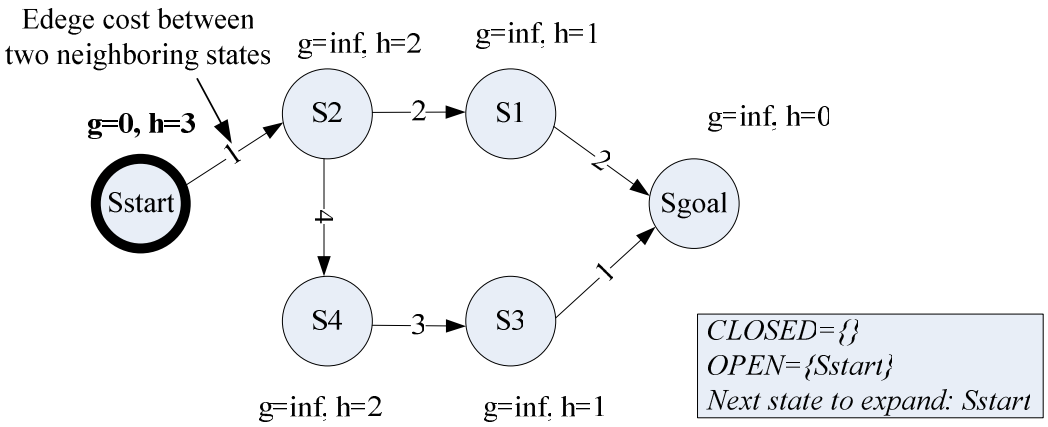
Planning a path for navigation can be regarded as a graph search problem. A* (Hart, Nilsson, & Rafael 1968; Nilsson 1980) algorithm is one of the early popular graph search methods. The method operates guiding its search towards the most promising set of states, return an optimal path, potentially saving a significant amount of computation.

For the forward version of A*, denote *S* the set of states in state space, for example a set of [position, yaw] states for a kinematic car model. Denote in table 3.1 that:

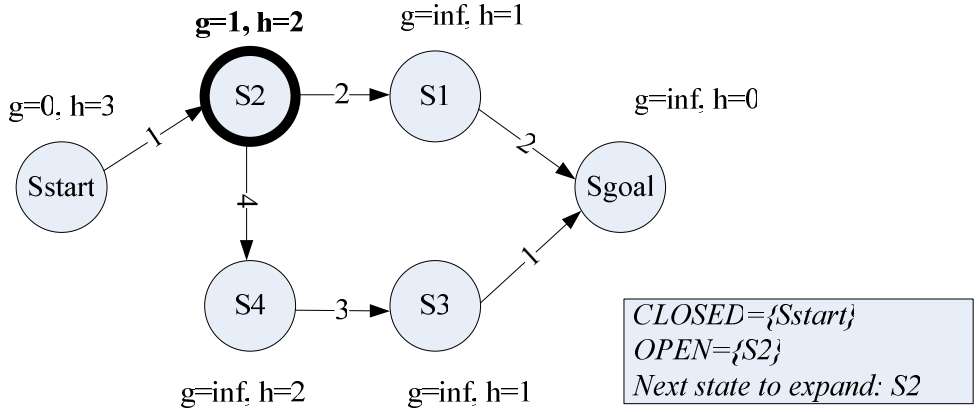
S	The set of states in state space, for exanple a set of [position, yaw] states for a kinematic car model.
$g(s)$	The path cost from the initial state $s_{start} \in S$ to state $s \in S$.
$h(s, s_{goal})$	The heuristic function estimate from state s to s_{goal}, for example the euclidean distance from state s to s_{goal}.
OPEN	A priority queue storing states needed to be expanded in the future. Each element s in this *OPEN list* is sorted according to $(g(s)+h(s, s_{goal}))$, which is the sum of its current path cost from start plus an admissible heuristic function estimate of path cost from s to s_{goal}.
c(s,s')	Edege cost from current state s to its neighboring state s'.

Table 3.1 Specific parameter in A star, forward version.

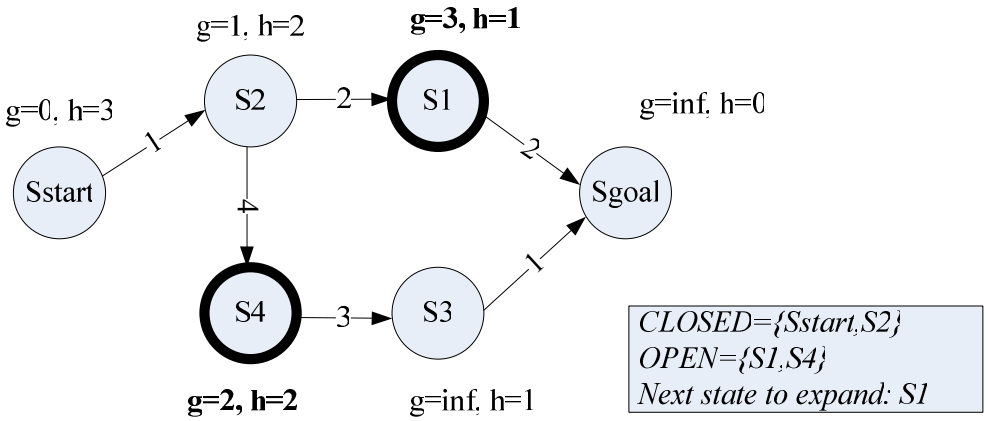
A simple example in Figure 3.1 will show how the algorithm works[12]. Here *CLOSED* stores visited list so far, number in the middle of the arrow represent the edege cost between two neighboring states.



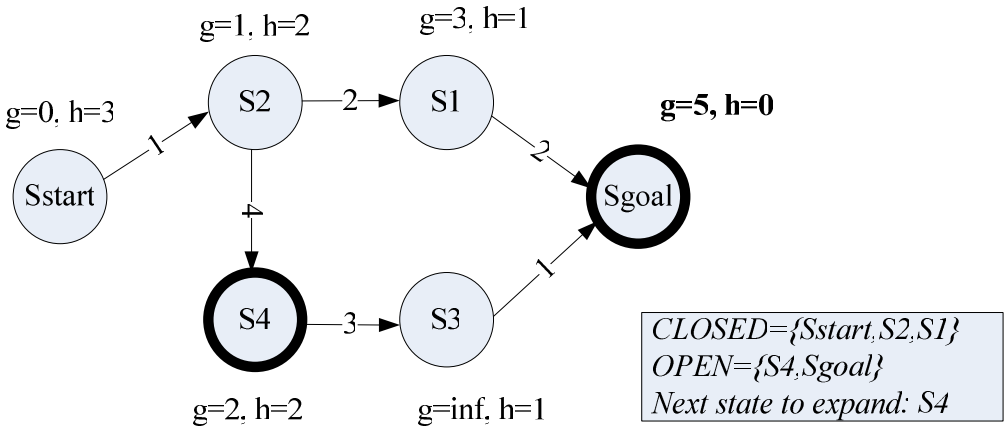
(a) after initialization



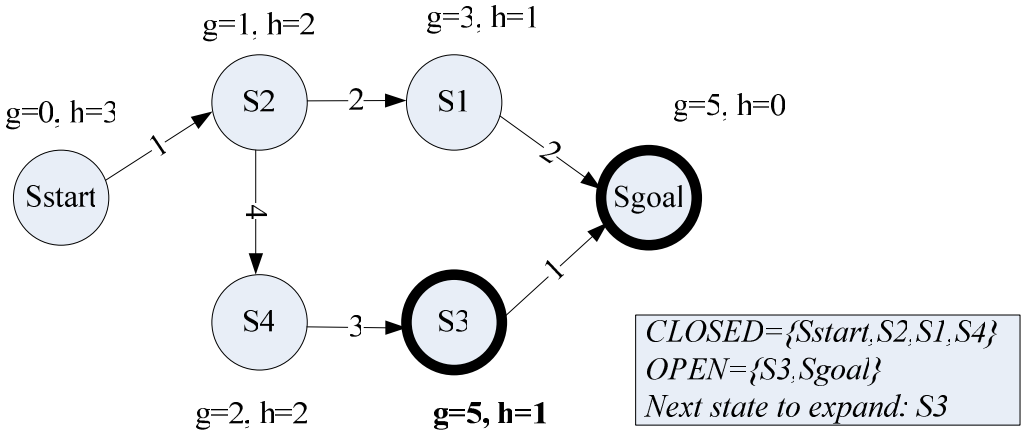
(b) after expansion of Sstart



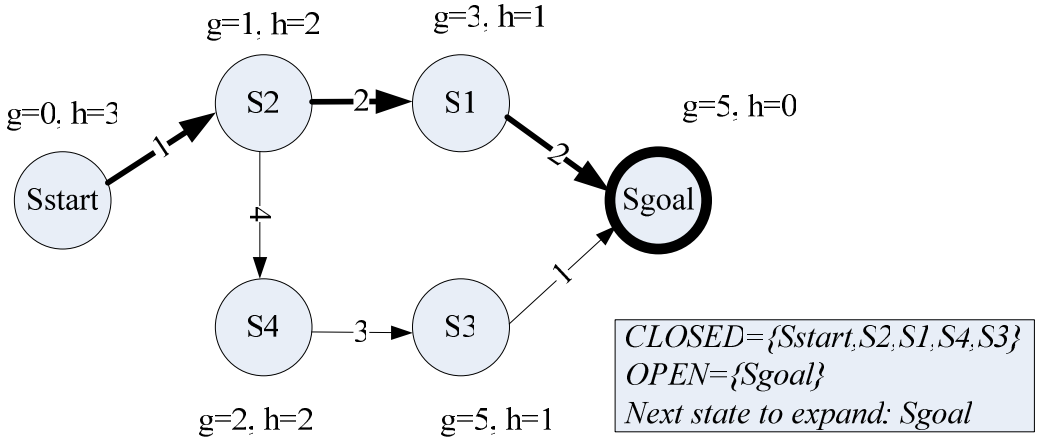
(c) after expansion of S2



(d) after expansion of S1



(e) after expansion of S4



(f) after expansion of S3, Sgoal is reached, solution is given by bold arrows

Figure 3.1 An example of how A* works

The heuristic $h(s, s_{goal})$ typically underestimates the cost of the optimal path from s to s_{goal} and is used to focus the search. The algorithm pops the state *s* at the front of the *OPEN* and updates the cost of all states reachable from this state through a direct edge: if g*(s')<c(s,s')+g(s),* then the cost of *s'* is set to this new, lower value. Also, if the cost of a neighboring state *s'* changes, it is placed on the *OPEN* list.

The algorithm continues popping states off the queue until it pops off the goal state. At this stage, if the heuristic is admissible, i.e. guaranteed to not overestimate the path cost from any state to the goal, then the path cost of s_{goal} is guaranteed to be optimal. The complete algorithm is given in Table 3.2.

ComputeShortestPath()
01. while ($\arg\min_{s \in OPEN}(g(s) + h(s, s_{goal})) \neq s_{goal}$)
02. remove state *s* from the front of *OPEN*;
03. for all $s' \in Succ(s)$
04. if ($g(s') > g(s) + c(s, s')$)
05. $g(s') = g(s) + c(s, s')$;
06. insert s' into *OPEN* with value $(g(s') + h(s', s_{goal}))$;
Main()
07. for all $s \in S$
08. $g(s) = \infty$;
09. $g(s_{start}) = 0$;
10. $OPEN = \varnothing$;
11. insert s_{start} into *OPEN* with value $(g(s_{start}) + h(s_{start}, s_{goal}))$;
12. ComputeShortestPath();

Table 3.2: Complete Algorithm of A*, forward version.

3.2 Anytime Repairing A*

When an agent must react quickly and the planning problem is complex, computing optimal paths as described in the previous sections can be infeasible, due to the sheer number of states required to be processed in order to obtain such paths. In such situations, we must be satisfied with the best solution that can be generated in the time available.

A backward version of Anytime Reparing A*(ARA*) is present here because it is easy to analysis. The planning start from goal state to start state, parameters are given in Table 3.3.

S	The set of states in state space, for exanple a set of [position, yaw] states for a kinematic car model.
$g(s)$	The path cost from current state $s \in S$ to goal state $s_{goal} \in S$.
$h(s_{start}, s)$	The heuristic function estimate from state s_{start} to s.
OPEN	A priority queue storing states needed to be expanded in the future.
CLOSED	All states already expanded once in the current search.
INCONS	All the inconsistent states that are not in *OPEN*. For example, When we are now at state s, due to a cost change associated with a neighboring state $s_{neighbore}$, this state is placed into *INCONS* list rather than reinserted into *CLOSED*.
C(s,s')	Edege cost from current state s to its neighboring state s'.
key(s)	Also known as the *f-value, is given by* $f = g(s) + \varepsilon \cdot h(s_{start}, s)$;
ε	The inflation factor used for inflated heuristics. A* search proceed the expansion in the order $f = (g(s) + h(s_{start}, s)$. Here for ARA* in order to have a quicker expansion, or in a more accurate way, to bias towards states that are closer to the goal. The expansion order is given by $f = (g(s) + \varepsilon \cdot h(s_{start}, s), \varepsilon \geq 1$.

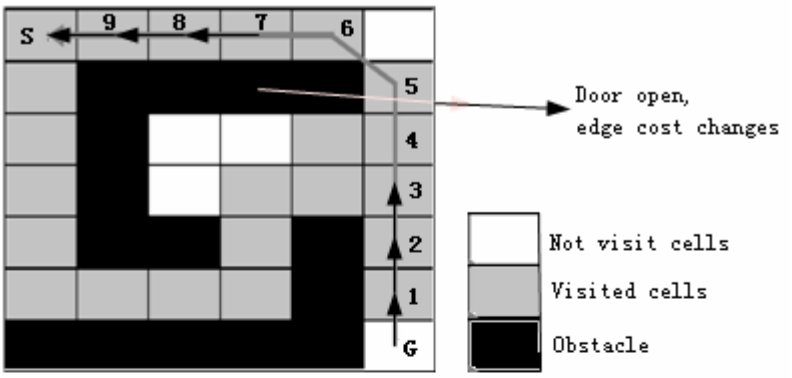
Table 3.3 ARA* pamameters, backward version.

ARA* limits the processing performed during each search by only considering those states whose costs at the previous search may not be valid given the new ε value. It begins by performing an A* search with an initial inflation factor ε_0 , but during this search it only expands each state at most once.

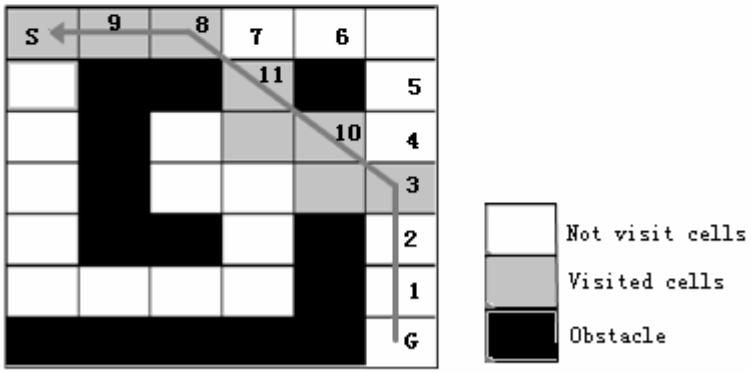
Once a state s has been expanded during a particular search, if it becomes inconsistent due to a cost change associated with a neighboring state, it is placed into the *INCONS* list. When current search terminates, the states in the *INCONS* list are inserted into a fresh priority queue (with new priorities based on the new inflation factor ε) which is used by the next search. This improves the efficiency of each search in two ways. Firstly, by only expanding each state

at most once a solution is reached much more quickly. Secondly, by only reconsidering states from the previous search that were inconsistent, much of the previous search effort can be reused. Thus, when the inflation factor ε is reduced between successive searches, a relatively minor amount of computation is required to generate a new solution.

An easy example in Figure 3.2 will show how the backward ARA* planning algorithms works. This example is a static planning from bottom right goal to the top left start.



(a) When plan to state 3, an edge cost is detected on its predecessors neighboring state 4. So that state 3 is moved into Inconsistent list.



(b) This time with a smaller inflation factor. The planner try its best to improve previous path solution, that is here to go through state 10, 11 and 8.

Figure 3.2 Example of how backward version of ARA* works

Figure 3.2(a)	Start with ε=2, after two steps of planning, a door is opened at the top walls, which means an edge cost change takes place. ARA* plays the same way A* does, but it also add state 3 into the inconsistent list, which is only used for next planning to improve solution trajectory (with a smaller inflation factor).
Figure 3.2(b)	This time, the planner tries to improve the generated path(ε=1). The planner will skip state 1 & 2, starting from state 3, continue updating cost of its predecessors until goal is reached. An improved path is generated, by only considering the inconsistent state and unvisited state(state goal, 1 & 2 are not considered because they are visited and consistent), the efficiency is improved.

Table 3.4 Comment on Figure 3.2

Overall, the ARA* planner will start planning with a big inflation factor $\varepsilon_0 > 1$, works out a sub-optimal path. Repeat improving this sub-optimal path by updating inconsistent states in it, until an optimal path is get, which is planning with $\varepsilon = 1$. A simplified, backwards-searching version of the algorithm is given in Table 3.5.

key(s)	
01.	return $g(s) + \varepsilon \cdot h(s_{start}, s)$;
ImprovePath()	
02.	while ($\min_{s \in OPEN}(key(s) < key(s_{start}))$)
03.	remove s with the smallest key(s) from *OPEN*;
04.	$CLOSED = CLOSED \cup \{s\}$;
05.	for all $s' \in pred(s)$
06.	if s' was not visited before
07.	$g(s') = \infty$;
08.	if $g(s') > c(s', s) + g(s)$
09.	$g(s') = c(s', s) + g(s)$;
10.	if $s' \notin CLOSED$
11.	insert s' into *OPEN* with $key(s')$;
12.	else
13.	insert s' into *INCONS*;

Table 3.5(a): Function key(s) and ImprovePath in ARA*

Main()
14. $g(s_{start}) = \infty; g(s_{goal}) = 0;$
15. $\varepsilon = \varepsilon_0;$
16. $OPEN = CLOSED = INCONS = \varnothing;$
17. insert s_{goal} into $OPEN$ with $key(s_{goal})$;
18. ImprovePath();
19. publish current ε-suboptimal solution;
20. while $\varepsilon > 1$
21. decrease ε;
22. Move states from *INCONS* into *OPEN*;
23. Update the priorities for all $s \in OPEN$ according to $key(s)$;
24. $CLOSED = \varnothing$;
25. ImprovePath();
26. publish current ε-suboptimal solution;

Table 3.5(b): Main loop of ARA*

3.3 D* Lite

D* Lite algorithm is present by Maxim Likhachev and Sven Koenig in 2002. D* Lite maintain least-cost paths between a start state and any number of goal states as the cost of arcs between states change. It can handle increasing or decreasing arc costs and dynamic start states.

S	The set of states in state space, for exanple a set of [position, yaw] states for a kinematic car model.
$g(s)$	The path cost from the initial state $s_{start} \in S$ to state $s \in S$.
$h(s,s')$	Cost estimate of an optimal path from state *s* to *s'*.
c(s,s')	The cost of moving from *s* to *s'* (the arc cost)
rhs(s)	Known as a one-step look ahead cost. Here $succ(s) \in S$ denotes the set of successors. A state is called consistent if and only if its g-value equals its rhs-value, otherwise it is either overconsistent (if $g(s) > rhs(s)$) or underconsistent (if $g(s) < rhs(s)$). $rhs(s) = \begin{cases} 0 & if\ s = s_{goal} \\ \min_{s' \in succ(s)}(c(s,s') + g(s')) & otherwise, \end{cases}$
OPEN	A priority queue holds exactly the inconsistent states, that is where edge cost changes.
key(s)	The priority, or also known as the key value, represent the priority in *OPEN* list. D* Lite expands states from the queue in increasing priority, updating their g-values and the rhs-values of their predecessors, until there is no state in the queue with a key value less than that of the start state. $key(s) = [k_1(s), k_2(s)]$ $= [\min(g(s), rhs(s)) + h(s_{start}, s),$ $\min(g(s), rhs(s))].$ In lexicographic ordering, $key(s)$ is less than priority $key(s')$, that is $key(s) < key(s')$, if and only if $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) < k_2(s')$.

Table 3.6: D* Lite parameters

D* Lite maintains a least-cost path from a start state to a goal state. As with A*, D* Lite uses a heuristic and a priority queue to focus its search and to order its cost updates efficiently. During its generation of an initial solution path, it performs in exactly the same manner as a backwards A* search. If arc costs change after this initial solution has been generated, D* Lite updates the rhs-values of each state immediately and places those states into *OPEN with* corresponding key value. As before, it then expands the states on the queue in order of

increasing priority until there is no state in the queue with a key value less than that of the start state. A simple robot navigation example in Figure 3.3 will show how the planner works:

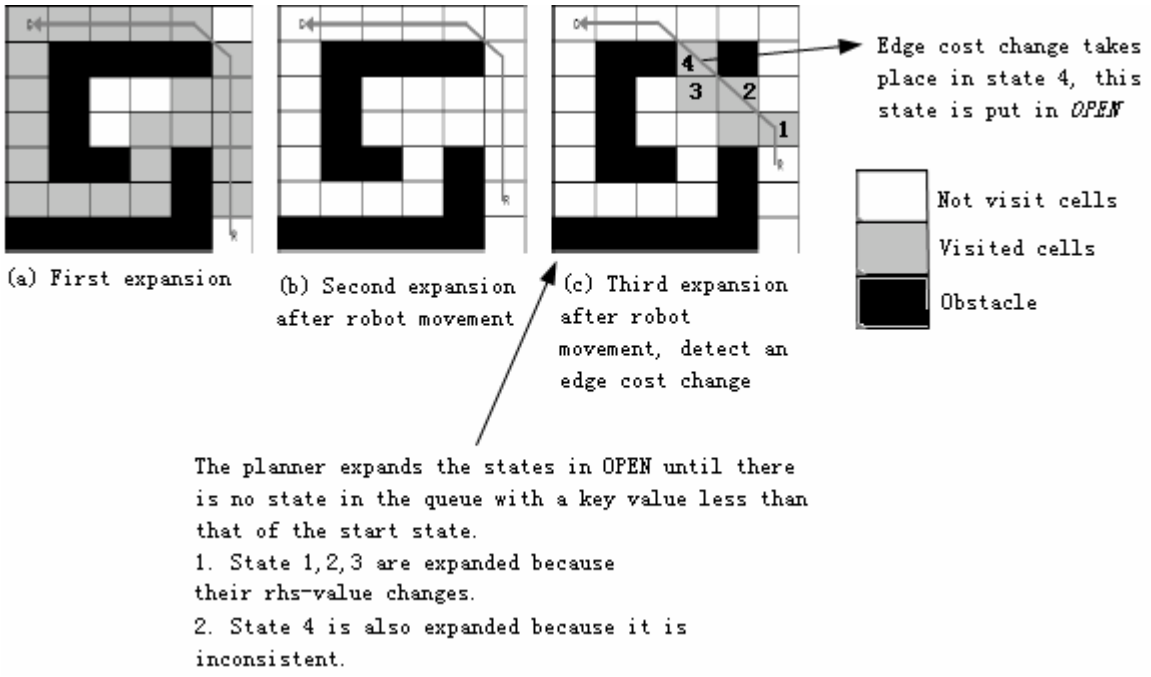


Figure 3.3: Robot navigation example with D* Lite.

D* Lite ensures that states that are along the current path and on the queue are processed in the most efficient order. Combined with the termination condition, this ordering also ensures that a least cost path will have been found from the start state to the goal state when processing is finished. The basic version of the algorithm is given in below in Table 3.7:

key(s)
01. return [$\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))$];
UpdateState(s)
02. if s was not visited before
03. $g(s) = \infty$;
04. if ($s \neq s_{goal}$) $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
05. if ($s \in OPEN$) remove s from *OPEN*;
06. if ($g(s) \neq rhs(s)$) insert s into *OPEN* with *key*(s);
ComputeShortestPath()
07. while $(\min_{s \in OPEN}(key(s)) < key(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start})$.
08. remove state s with the minimum key from *OPEN*;
09. if ($g(s) > rhs(s)$)
10. $g(s) = rhs(s)$;
11. for all $s' \in pred(s)$ UpdateState(s');
12. else
13. $g(s) = \infty$;
14. for all $s' \in pred(s) \cup \{s\}$ UpdateState(s');
Main()
15. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty$;
16. $rhs(s_{goal}) = 0; OPEN = \varnothing$;
17. insert s_{goal} into *OPEN* with key(s_{goal});
18. forever
19. ComputeShortestPath();
20. Wait for changes in edge costs;
21. for all directed edges (u, v) with changed edge costs
22. Update the edge cost $c(u, v)$;
23. UpdateState(u);

Table 3.7: D* Lite algorithm

3.4 Anytime Dynamic A*

In 2005, Maxim Likhachev developed Anytime Dynamic A* (AD*), an algorithm that combines the replanning capability of D* Lite with the anytime performance of ARA* (Likhachev et al. 2005). Parameter details are given in Table 3.8.

S	The set of states in state space, for exanple a set of [position, yaw] states for a kinematic car model.
$g(s)$	The path cost from the initial state $s_{start} \in S$ to state $s \in S$.
$h(s,s')$	Cost estimate of an optimal path from state *s* to *s'*.
c(s,s')	The cost of moving from *s* to *s'* (the arc cost)
rhs(s)	Known as a one-step look ahead cost. Here $succ(s) \in S$ denotes the set of successors. A state is called consistent if and only if its g-value equals its rhs-value, otherwise it is either overconsistent (if $g(s) > rhs(s)$) or underconsistent (if $g(s) < rhs(s)$). $$rhs(s) = \begin{cases} 0 & if\ s = s_{goal} \\ \min_{s' \in succ(s)}(c(s,s') + g(s')) & otherwise, \end{cases}$$
OPEN	A priority queue storing states needed to be expanded in the future.
CLOSED	All states already expanded once in the current search.
INCONS	All the inconsistent states that are not in *OPEN*. Each time ε is decreased, all inconsistent states are moved from *INCONS* to *OPEN* and *CLOSED* is made empty.
key(s)	The priority, or also known as the key value, represent the priority in *OPEN* list. D* Lite expands states from the queue in increasing priority, updating their g-values and the rhs-values of their predecessors, until there is no state in the queue with a key value less than that of the start state. $$\begin{aligned} key(s) &= [k_1(s), k_2(s)] \\ &= [\min(g(s), rhs(s)) + h(s_{start}, s), \\ &\quad \min(g(s), rhs(s))]. \end{aligned}$$ In lexicographic ordering, $key(s)$ is less than priority $key(s')$, that is $key(s) < key(s')$, if and only if $k_1(s) < k_1(s')$ or both $k_1(s) = k_1(s')$ and $k_2(s) < k_2(s')$.

ε	The inflation factor used for inflated heuristics, to bias the states which are closer to the goal. AD* search proceed expansion $key(s) = [\min(g(s), rhs(s)) + \varepsilon \cdot h(s_{start}, s); \min(g(s), rhs(s)))]$.

Table 3.8: AD* parameters

AD* performs a series of searches using decreasing inflation factors ε to generate a series of solutions with improved bounds, as with ARA*. When there are changes in the environment affecting the cost of edges in the graph, locally affected states are placed on the *OPEN* queue to propagate these changes through the rest of the graph, as with D* Lite. States on the queue are then processed until the solution is guaranteed to be ε-suboptimal.

AD* begins by setting the inflation factor ε to a sufficiently high value ε_0, so that an initial, suboptimal plan can be generated quickly. Then, unless changes in edge costs are detected, ε is gradually decreased and the solution is improved until it is guaranteed to be optimal($\varepsilon = 1$). This phase is exactly the same as for ARA*.

When changes in edge costs are detected, there is a chance that the current solution will no longer be ε-suboptimal, i.e. an intermediate state in solution trajectory becomes an obstacle. In such a case, the algorithm increases ε so that a new sub-optimal solution can be produced quickly. Then repeat decreasing ε until the solution becomes optimal.

By incorporating these considerations, AD* is able to handle both changes in edge costs and changes to the inflation factor ε. Table 3.9 gives the complete algorithm.

key(s)
01. if ($g(s) > rhs(s)$)
02. return $[\min(g(s), rhs(s)) + \varepsilon \cdot h(s_{start}, s); \min(g(s), rhs(s)))]$;
03. else
04. return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s)))]$;
UpdateState(s)
05. if *s* was not visited before
06. $g(s) = \infty$;
07. if ($s \neq s_{goal}$) $rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
08. if ($s \in OPEN$) remove s from *OPEN*;
09. if ($g(s) \neq rhs(s)$)
10. if $s \notin CLOSED$
11. insert *s* into *OPEN* with *key(s)*;
12. else
13. insert *s* into *INCONS*;
ComputeOrImprovePath()
14. while $(\min_{s \in OPEN}(key(s)) < key(s_{start})$ **OR** ($rhs(s_{start}) \neq g(s_{start})$)
15. remove state *s* with the minimum key from *OPEN*;
16. if $(g(s) < rhs(s))$
17. $g(s) = rhs(s)$;
18. $CLOSED = CLOSED \cup \{s\}$;
19. for all $s' \in pred(s)$ UpdateState(s');
20. else
21. $g(s) = \infty$;
22. for all $s' \in pred(s) \cup \{s\}$ UpdateState(s');

Table 3.9(a): Key, UpdateState and ImprovePath function

Main()
01. $g(s_{start}) = rhs(s_{start}) = \infty; g(s_{goal}) = \infty;$
02. $rhs(s_{goal}) = 0; \varepsilon = \varepsilon_0;$
03. $OPEN = CLOSED = INCONS = \varnothing;$
04. insert s_{goal} into OPEN with $key(s_{goal})$;
05. ComputeorImprovePath();
06. publish current ε-suboptimal solution;
07. forever
08. if changes in edge costs are detected
09. for all directed edges *(u, v)* with changed edge costs
10. Update the edge cost *c(u, v)*;
11. UpdateState(*u*);
12. if significant edge cost changes were observed
13. increase ε or replan from scratch;
14. else if $\varepsilon > 1$
15. decrease ε;
16. Move states from *INCONS* into *OPEN*;
17. Update the priorities for all $s \in OPEN$ according to *key(s)*;
18. $CLOSED = \varnothing$;
19. ComputeorImprovePath();
20. publish current ε-suboptimal solution;
21. if $\varepsilon = 1$
22. wait for changes in edge costs;

Table 3.9(b): Main loop of AD*

3.5 Anytime Nonparametric A*

Anytime Nonparametric A*, also known as (ANA), is present by Jur van den Berg, Rajat Shah, Arthur Huang and Ken Goldberg. The motivation of this planning algorithm is to develop an Anytime A* algorithm that does not require parameters(do not assign ε and the amount by which ε is decreased for ARA* for example).

S	The set of states in state space, for exanple a set of [position, yaw] states for a kinematic car model.
$g(s)$	The path cost from the initial state $s_{start} \in S$ to state $s \in S$.
G	The cost of the current-best solution, G is set to ∞ in initialization because no solution is found.
e(s)	It is given by $e(s) = \frac{G - g(s)}{h(s)}$, that is the maximal value of e such that $f(s) \leq G$. ANA will expand the state in *OPEN* with maximum e(s).
c(s,s')	Edge cost from current state *s* to its neighboring state *s'*.
OPEN	A priority queue storing states needed to be expanded in the future.

Table 3.10: ANA parameters

Choose to expand the state in *OPEN* with maximum e(s) in ANA means that bias the states which is closest to the goal. This guarantees that ANA will plan quickly get a sub-optimal trajectory and try to improve it with decreasing e(s) until an optimal solution is get.

Moreover, ANA acts more or less the same way as Weighted A* does, but with 2 differences.

1. Each time a state s is expanded, it will try to decrease the g-value(g(*s'*)) of each its successors *s'*. Then, *s* is set to *s'* predecessor such that the solution can be reconstructed once it is found.
2. Moreover, this *s'* is inserted into the *OPEN*. The planner will continue expanding states in *OPEN* until it is empty. That is to prune the goal state and all the updated state *s'*. Details of the complete algorithm is given in Table 3.11.

IMPROVESOLUTION()
1: while $OPEN \neq \emptyset$ do
2: $s \leftarrow \arg\min_{s \in OPEN}\{e(s)\}$
3: $OPEN \leftarrow OPEN \setminus \{s\}$
4: if $e(s) < E$ then
5: $E \leftarrow e(s)$
6: if IsGOAL(s) then
7: $G \leftarrow g(s)$
8: return
9: for each successor *s'* of *s* do
10: if $g(s) + c(s, s') < g(s')$ then
11: $g(s') \leftarrow g(s) + c(s, s')$
12: $pred(s') \leftarrow s$
13: if $g(s') + h(s') < G$ then
14: Insert or update *s'* in *OPEN* with key e(*s'*)
ANA*()
15: $G \leftarrow \infty; E \leftarrow \infty; OPEN \leftarrow \emptyset; \forall s : g(s) \leftarrow \infty; g(s_{start}) \leftarrow 0;$
16: Insert s_{start} into *OPEN* with key e(s_{start})
17: while $OPEN \neq \emptyset$ do
18: IMPROVESOLUTION()
19: Report current *E*-suboptimal solution
20: Update keys $e(s)$ in *OPEN* and prune if $g(s) + h(s) \geq G$

Table 3.11: ANA algorithm

4. Search-Based Planner Library

Search Based Planner Library (SBPL) is a package includes a generic set of motion planners using search based planning. It was developed by Maxim Likhachev at the University of Pennsylvania in collaboration with Willow Garage.

4.1 Environment and Planner class

Environment class and Planner class are two most important parts of it. All their elements inherit from their parent class. Useful detail is given by Table 4.1 and 4.2. For Yamaha grizzly, the most suitable class is Navxytheta class.

Inheritance from basic Environment class	Feature
EnvironmentNAV2D class	2D navigation class. For vehicle motion planning, state is the position(x,y).
EnvironmentNAVTHETA class	3D navigation class. For vehicle motion planning, state is the position(x,y) and yaw(theta)
EnvironmentNAVXYTHETAMLEVLAT class	3D navigation class with multiple level of vehicle configuration, i.e. a vehicle with a car like base and a moving platform on it. State is several layers of position(x,y) and yaw(theta).
EnvironmentRobotARM class	For kinematic robot arm navigation of variable number of degrees of freedom.

Table 4.1: Environment class

Inheritance from basic Planner class	Feature
ADplanner class	Grid search planning class, use Anytime Dynamic A* (AD) algorithm.
ARAplanner class	Grid search planning class, use Anytime Repairing A* (ARA) algorithm.
anaPlanner class	Grid search planning class, use Anytime Nonparametric A* (ANA*) algorithm.

Table 4.2: Planner class

4.2 Motion primitive

Search based planning library uses motion primitives for path planning in a 2D environment, Motion primitives are short, kinematically feasible motions which form the basis of movements that can be performed by the robot platform. Search-based planner generates paths from start to goal by combining a sequence of motion primitives. The result is a smooth kinematically feasible path. A simple example will show how motion primitives work. Let us assume that we are in a simple case in Table 4.3.

Feature	Parameter
Permissive vehicle actions	Run forward and make forward turn 25 degs, both left and right, or run backward.
Yaw(θ) discretisation	Resolution 22.5 degs, which means overall 360/22.5=16 choices of vehicle pointing direction
Action cost	Forward action cost = 1, Making forward turn cost = 5, Moving backward action cost = 10.

Table 4.3: Simple motion primitive feature

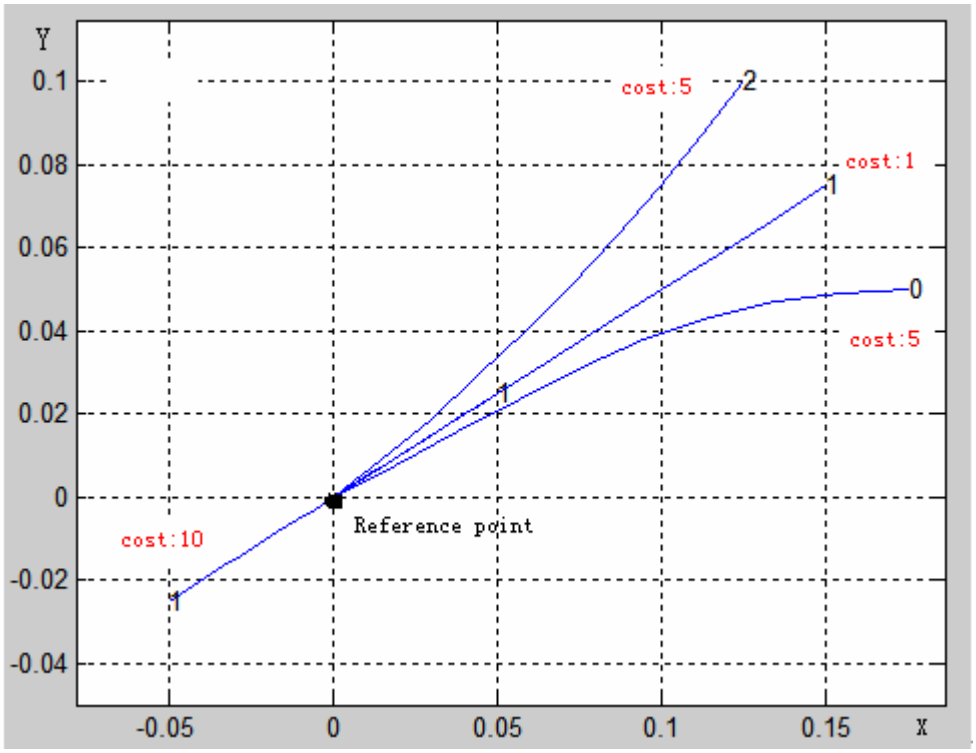


Figure 4.1: Motion primitive with yaw = 22.5 deg

Figure 4.1 is one of the sixteen motion primitives, [θ=22.5 deg, vehicle reference point (0,0)]. When the SBPL planner is working, its goal is to choose between 16 motion primitives and the exact robot action in that motion primitives. Like in Figure 4.2, all 16 motion primitives are included.

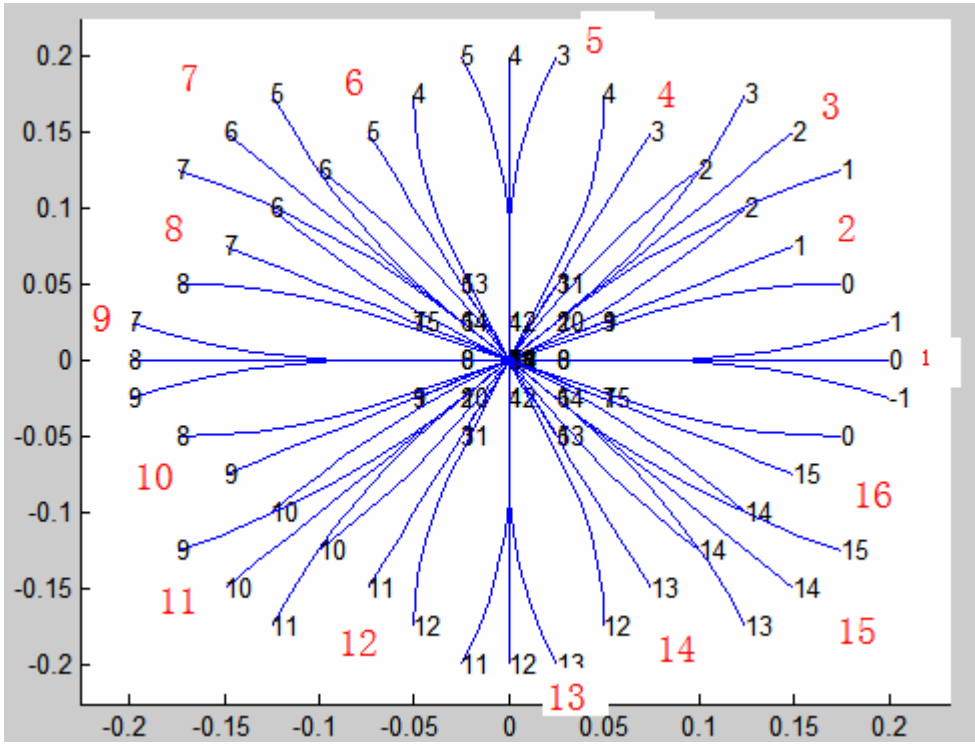


Figure 4.2: All 16 motion primitives

A path solution is a series of these motion primitives. While the planner is working, it will search through all 16 motion primitives from the current state, check obstacle collision problem, try its best to find trajectory with minimal sum action cost(optimal solution). More details is in Figure 4.3, grey objects are obstacles. Solid series of arrows are the path solution, dashed ones represents the actions the planner has tried.

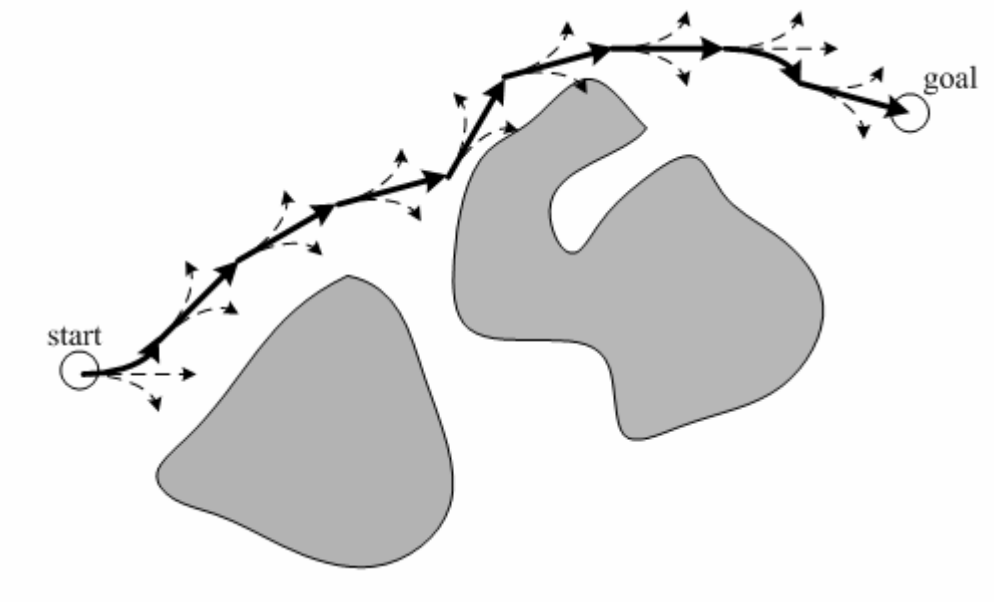


Figure 4.3: Solution composition

5. Navxytheta class

5.1 Grid environment

Navxytheta class uses discretized cells to describe environment. In our case, obstacle threshold is 1, cell_size is 0.1m. So that for a 100m×100m environment, a 2 dimensional matrix of size 1000x1000 is used to store map data. For each element, 0 means free space, 1 means obstacle. An example is given:

$$\begin{bmatrix} 0\,0\,0\,0\,1 & \cdots & 0\,0\,0 \\ 0\,0\,0\,1\,1 & \cdots & 0\,0\,0 \\ \vdots & & \vdots \\ 0\,0\,1\,1\,1 & \cdots & 1\,1\,1 \end{bmatrix}^{1000\times1000}$$

The defination of a roughness map is more or less the same. It has the same size of environment map. For each cell, there is an integer number to describe how much is the roughness, higher this number, higher is the roughness. An example is given below:

$$\begin{bmatrix} 0\,1\,3\,9\,9 & \cdots & 3\,2\,1 \\ 0\,3\,0\,1\,1 & \cdots & 2\,2\,2 \\ \vdots & & \vdots \\ 0\,1\,8\,4\,0 & \cdots & 5\,5\,5 \end{bmatrix}^{1000\times1000}$$

5.2 Motion Primitive for ATV

Motion primitive for Yamaha ATV is composed of all the possible action the vehicle can perform during a specific time period. Parameter details are in Table 5.1, Figure 5.1 and 5.2 are the footprints for yaw = 0 degs and 90 degs seperately.

Motion primitives Feature	Parameter
Allowable vehicle actions	Running both forward and backward. Turnning angle [30, 20, 10, 5] degs, both left and right.
Speed	Zeros speed and forward 2m/s.
Time discretization	0.5s
Yaw(θ) discretization	Resolution 22.5 degs, 360/22.5=16 choices of vehicle pointing direction.
Primitive action cost	Forward moving cost = 1, Making forward turn cost = 2, Moving backward cost = 5.

Table 5.1: Motion primitive parameter for ATV

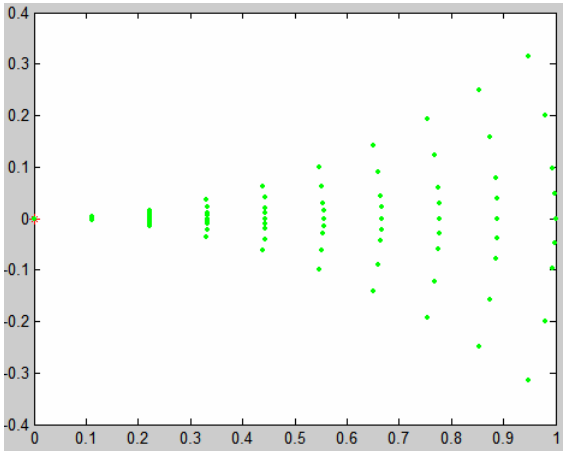


Figure 5.1: Base motion primitive

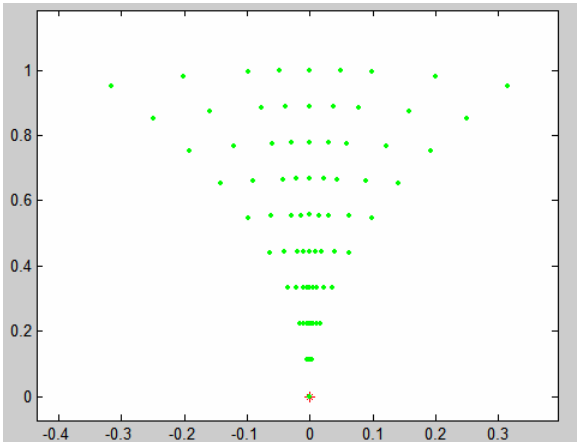


Figure 5.2: Motion primitive with yaw=90 deg

5.3 Action for motion primitive

For example, when we are planning trajectory with forward ARA* planner in Navxytheta library. The planning algorithm needs a reasonable way to assign successors to the planner. Navxytheta class provide this kind of method. It is to choose among pregenerated motion primitives, check their validity of source, target and intermediate states. Then pass on the set of reasonable actions to the planner. Details is given in Figure 5.3.

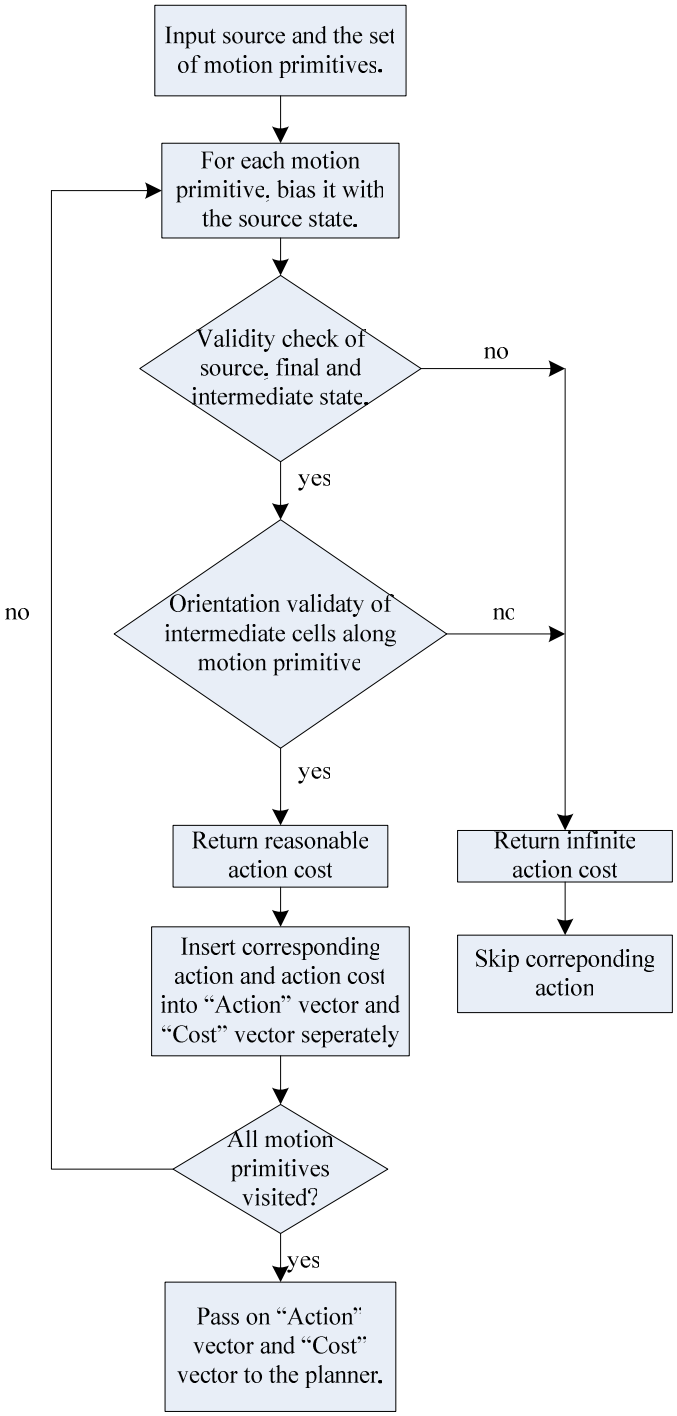


Figure 5.3: Flow diagram to assign actions

5.4 Main loop for Navxytheta

Main loop for atv path planning with motion primitives is given by Figure 5.4. The inflation factor epsilon is used for increasing planning speed, bigger the epsilon, faster the planning speed. An optimal solution is given if epsilon is 1, in that case the solution cost is minimum.

Commonly the planning is started with a big epsilon, the planner itself is going to decrease epsilon, trying its best to reduce the solution cost until it can be decreased no more.

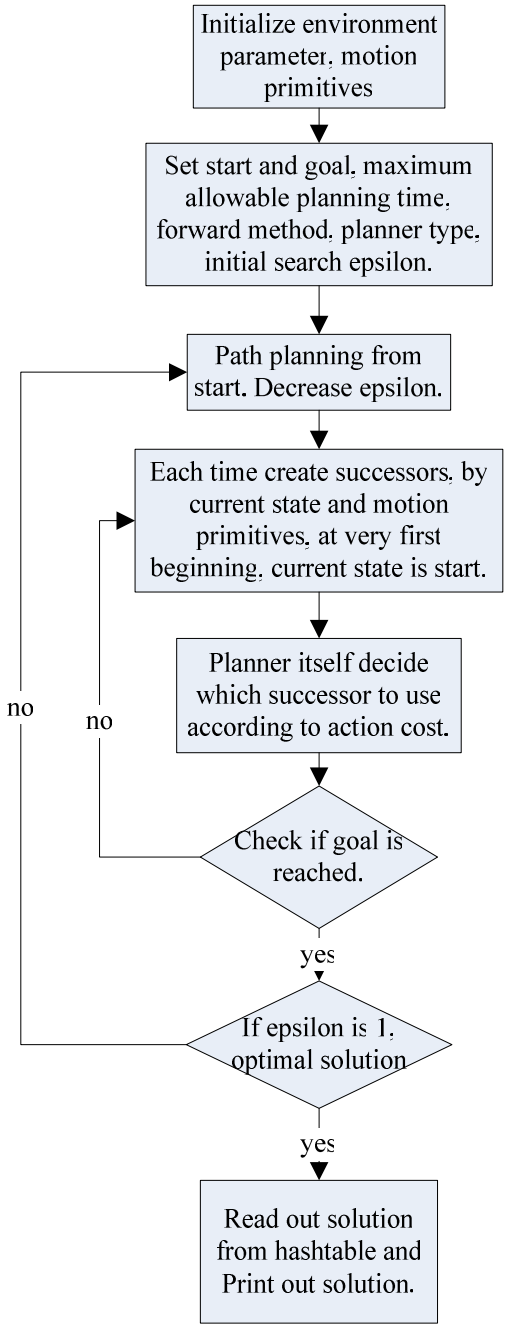


Figure 5.4: Main loop for Navxytheta, forward planning direction

6. NavATV class

NavATV(NavXXX) class comes from modified Navxytheta class. Navxytheta class can only generate kinematic feasible trajectory, which is not enough for YAMAHA Grizzly. What we want is not only the trajectory but the state speed and state steer for vehicle control. Moreover, to cope with anti-roll over problem, dynamic constraint is needed to limit vehicle movements, i.e. limit turning angle while running at high speed. A dynamic car model and a suitable way to assign vehicle speed and steering angle are important.

6.1 Dynamic Car Model

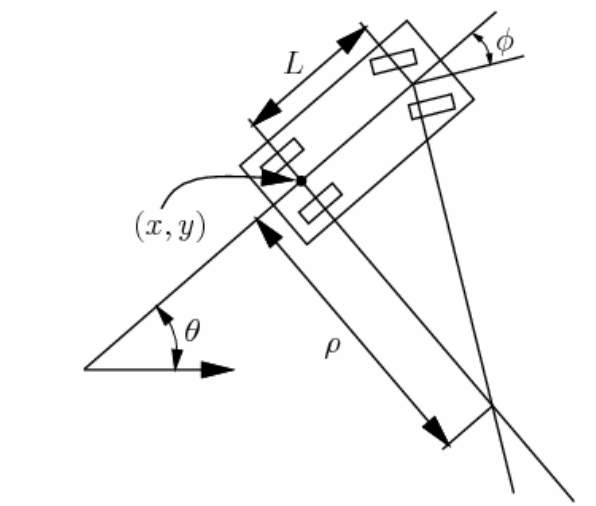


Figure 6.1: Dynamic Car model

Parameter:	Feature:
s	Axial speed
ϕ	Steering angle
L	Distance between front and rear axis
θ	Yaw
w	The distance traveled by the car
ρ	The radius of the turning circle

Table 6.1: Dynamic Car Features

A configuration for dynamic car is denoted by $q = (x, y, \theta)$ [1]. In a small time interval Δt, the car is moving approximately in the direction that the rear wheels are pointing. When Δt tends to zero, it implies that $dy/dx = \tan\theta$. Since $dy/dx = \dot{y}/\dot{x}$, and $\tan\theta = \sin\theta/\cos\theta$, We can get a Pfaffian constraint function $-\dot{x}\sin\theta + \dot{y}\cos\theta = 0$. The constraint is satisfied if $\dot{x} = s \cdot \cos\theta$ and $\dot{y} = s \cdot \sin\theta$, which is the decomposition of axial speed along x and y axis.

The next task is to derive dynamic equation for $\dot{\theta}$. w means the distance traveled by the car(integral of speed). As shown in Figure 6.1, If in the case steering angle is fixed, we can come out that $dw = \rho \cdot d\theta$. And from trigonometry $\rho = L/\tan\phi$, it is clear that $d\theta = \frac{\tan\phi}{L} dw$. By assigning s and ϕ as control variables u_s and u_ϕ. Dynamic equation for x, y and theta is given by:

$$\dot{x} = u_s \bullet \cos\theta$$
$$\dot{y} = u_s \bullet \sin\theta$$
$$\dot{\theta} = \frac{u_s}{L}\tan u_\phi$$

All this lead to our final dynamic equations. Denote *t* current time instant, *t+1* future time instant, Δt the time interval. The complete equation is given below.

$$s_{t+1} = s_t + u_s \cdot \Delta t$$
$$\phi_{t+1} = \phi_t + u_\phi \cdot \Delta t$$
$$\theta_{t+1} = \theta_t + \frac{s_{t+1}}{L}\tan\phi_{t+1} \cdot \Delta t$$
$$x_{t+1} = x_t + s_{t+1} \cdot \cos\theta_{t+1} \cdot \Delta t$$
$$y_{t+1} = y_t + s_{t+1} \cdot \sin\theta_{t+1} \cdot \Delta t$$

Unluckily, SBPL provides nothing to assign u_s and u_ϕ. Mofication on Environment Navxytheta is necessary. Besides, once a new dynamic constraint is added, a new set of motion primitives has to be generated for this certain case. Generally speaking, this is not reasonable for practice. One way is to make motion primitive “online”. Let the planner decide future actions and intermediate footprint based on actual vehicle state. Our ATV requires state

to be $[x_t, y_t, \theta_t, s_t, \phi_t]'$, let the planner decide the set of suitable increment $\{[\Delta s, \Delta\phi]'\}$. By a slight modification on previous equation, future state can be achieved by following.

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \\ s_{t+1} \\ \phi_{t+1} \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \\ \theta_t \\ s_t \\ \phi_t \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \Delta s \\ \Delta\phi \end{bmatrix}$$

Where $\Delta x = s_{t+1} \cdot \cos\theta_{t+1} \cdot \Delta t$, $\Delta y = s_{t+1} \cdot \sin\theta_{t+1} \cdot \Delta t$ and $\Delta\theta = s_{t+1} / L \cdot \tan\phi_{t+1}$.

6.2 Increment generation

6.2.1 Assign speed and steering increment

According to Yamaha Grizzly on road performance in Table 6.2, and also leave some margin for off road behavior to cope with high roughness. Since $[\Delta s, \Delta\phi]' = [\dot{s}, \dot{\phi}]' \cdot \Delta t$, we can assign our set of $\dot{s}$ and $\dot{\phi}$, the set of $\{[\Delta s, \Delta\phi]'\}$ is easily get. Detail is in Table 6.3.

Yamaha Grizzly	Feature
Allowable movement	Forward, forward turnning. No backward running
Maximum on road speed	107Km/h, that is 29.7m/s.
Steering on road	Steer 55 degs in 2 secs. For a linear steer process, the steering is 0.48rad/s
Speed up on road	Speed up from zero speed to 60 km/h in 5.1s. For a linear speed up case, the acceleration speed is 3.27m/s.

Table 6.2 YAMAHA Grizzly on road performance

Vehicle behavior constraint for SBPL
The set of possible $\dot{s}$ = [-0.8, -0.6, -0.3, 0, 0.3, 0.6, 1.2](m/s).
The set of possible $\dot{\phi}$ = [-15, -10, -5, 0, 5, 10, 15](degs/s).
Maximum permissible speed = 3m/s.
Minimum speed = 0.
Maximum permissible steer = 30 degs.
Integration time Δt =0.5s.
The set of possible Δs =[-0.4, -0.3, -0.15, 0, 0.15, 0.3, 0.6](m).
The set of possible $\Delta\phi$ =[-7.5, -5, -2.5, 0, 2.5, 5, 7.5](degs).

Table 6.3 Vehicle behavior constraint for SBPL in our experiment

6.2.2 ATV model and state discretize

Yamaha Grizzly can be modelled by a rectangular shape Arkerman model. Since we are in 2D case, there is no need to consider ATV height. The center of gravity lies in the middle of the rear axial. Its length is 1.8 meter and width is 0.9 meter. The gravity center is the vehicle reference point, in vehicle body collision check, this reference point is very important.

In Navxytheta library, position x and y are already discretized to guarantee a finite dimensional state space. Here the same thing has to carried out for yaw, speed and steer. Speed discretize can be achieved by setting the maximum speed limit and the minimum speed limit. For yaw, experiment results shows that 20 degs of resolution is perfect to balance planning speed and accuracy. Steer resolution is set to 2 degs to garantee precision. Up till now, we get a 5D finite dimensional state space.

6.3 Action for direct increment

Now the action is based on actual and future vehicle states. Here I am going to present how shortest path and smoothest path comes.

Shortest path is the combination of a series of actions with minimum total increment on displacement. By define the action cost function as the “euclidean distance” of the increment, SBPL planner will automatically get a path with minumum sum of increments. The smoothest path means that minimum speed and steer angle difference along the trajectory. But since speed and steer have the different unit, normalize speed difference and normalized steer difference is used here instead. Details are given by Figure 6.2.

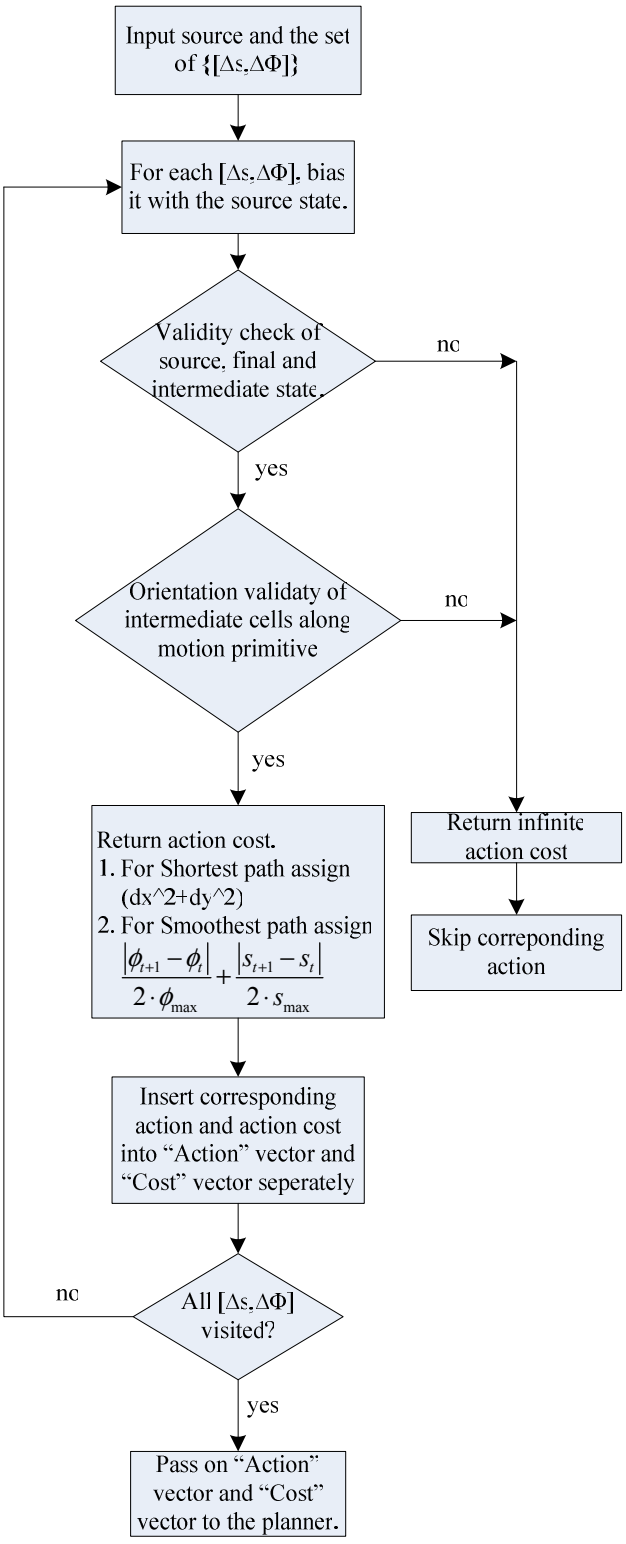


Figure 6.2: Action assignment for shortest path and smoothest path

6.4 Mainloop for NavATV

This is loop for forward atv path planning with direct increment method, detail is given by Figure 6.4. Also here the inflation factor epsilon is for speed planning. Commonly the planning is started with a big epsilon, the planner itself is going to decrease epsilon, trying its best to reduce the solution cost until it can be decreased no more. And this time epsilon is equal to 1.

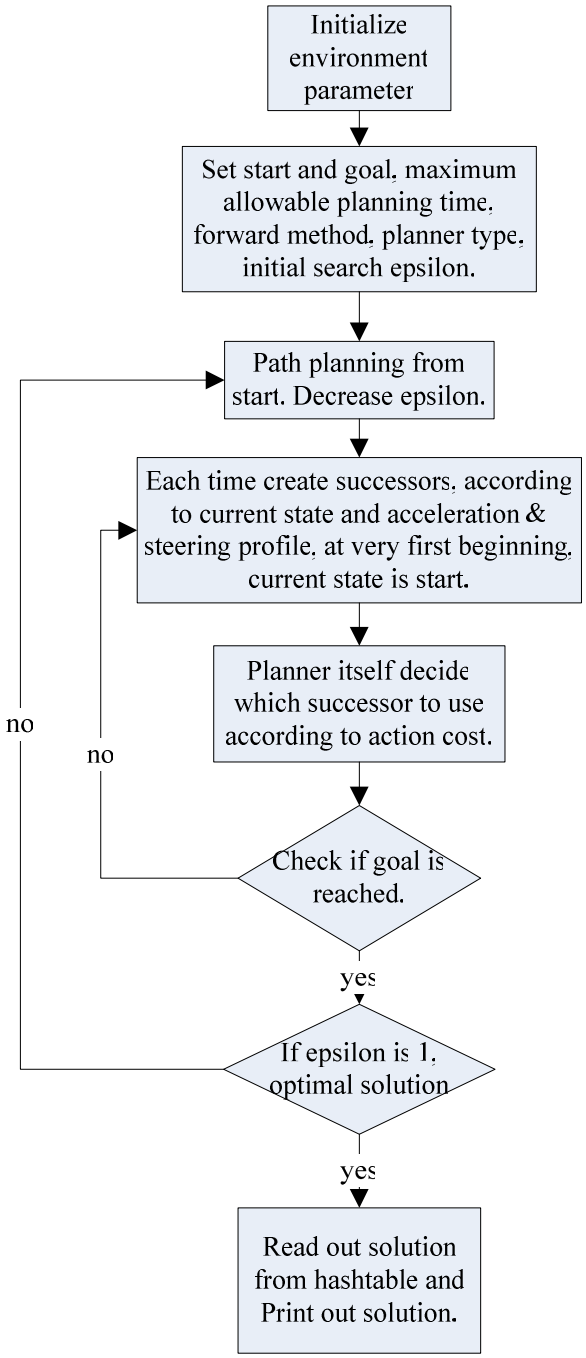


Figure 6.4 Main loop for EnvATV

7. Benchmark

This part is a benchmark of optimal trajectory between different kinds of paths and different planners. The hardware platform is CPU T6400 2.2Ghz, 4G of ram, NVIDIA GT220M 1G graphics. Maximum allowable time for planning is 300s. Constraints are given below:

Constraint	Motion Primitive	Direct Increment(shortest path & smoothest path).
Speed limit	Two speeds, zero speed and 2m/s	Maximum speed 3m/s, minumum speed 0.
Steer limit	30 degs both left and right.	30 degs both left and right.

Table 7.1 Constraint for benchmark

7.1 Navigate along the obstable

7.1.1 Motion Primitives, shortest path and smoothest path benchmark

This section is the performance analysis along the obstacle. The environment map is $100\times100m^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,80) and goal position (90,85). Detail is given below.

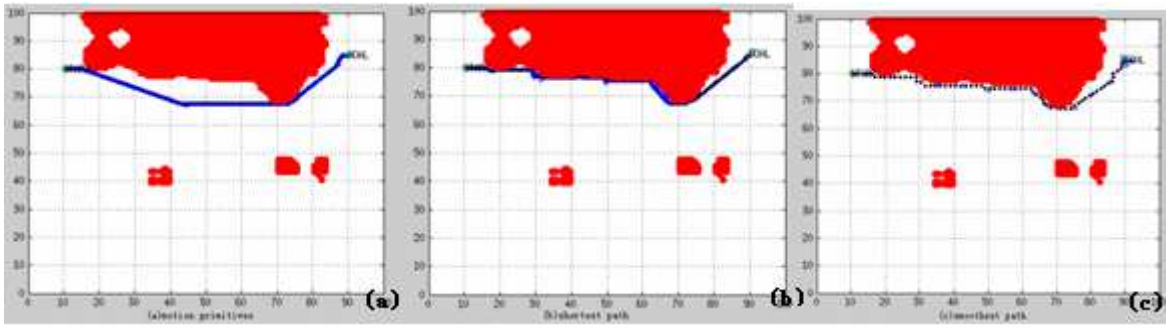


Figure 7.1: (a) Motion primitives trajectory, (b) Shortest path trajectory, (c) Smoothest path trajectory

	Travel Distance
Motion primitive	90.365m
Shortest path	95.78m
Smoothest path	107.86m

Table 7.2: Travel Distance of different trajectories, notice that motion primitive path is shorter but unfeasible with respect to vehicle dynamics

7.1.2 Benchmark of shortest path between AD, ARA*, ANA planners

The environment map is $100\times100m^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,80) and goal position (90,85). Detail is given below.

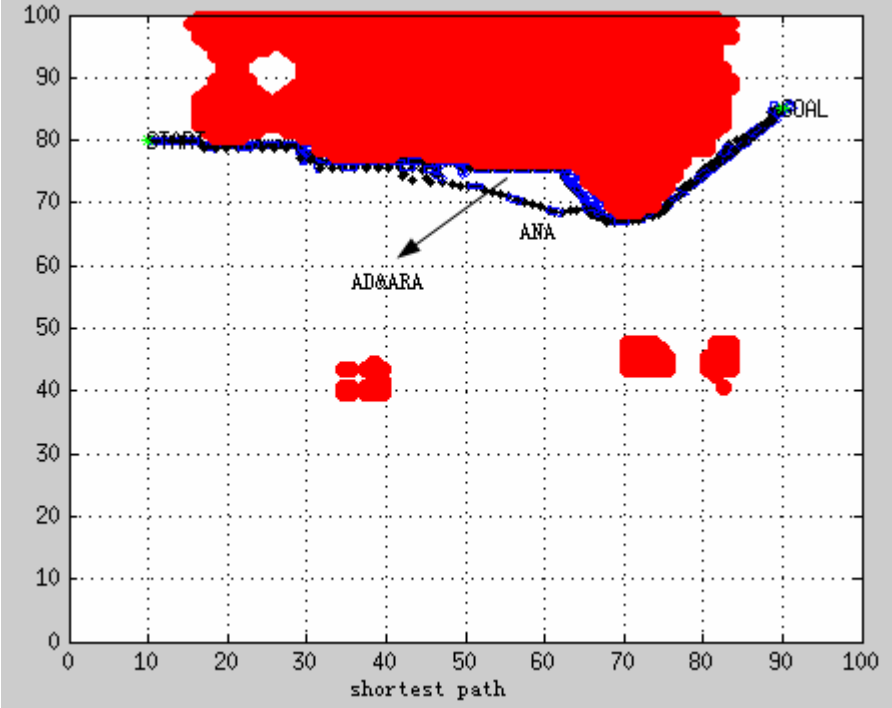


Figure 7.2: Footprint of shortest paths of AD, ARA* and ANA planners

	Shortest path travel distance	Shortest path travel time
AD	95.78m	164.5s
ARA*	95.78m	164.5s
ANA	115.14m	59.5s

Table 7.3: Shortest path details of AD, ARA* and ANA

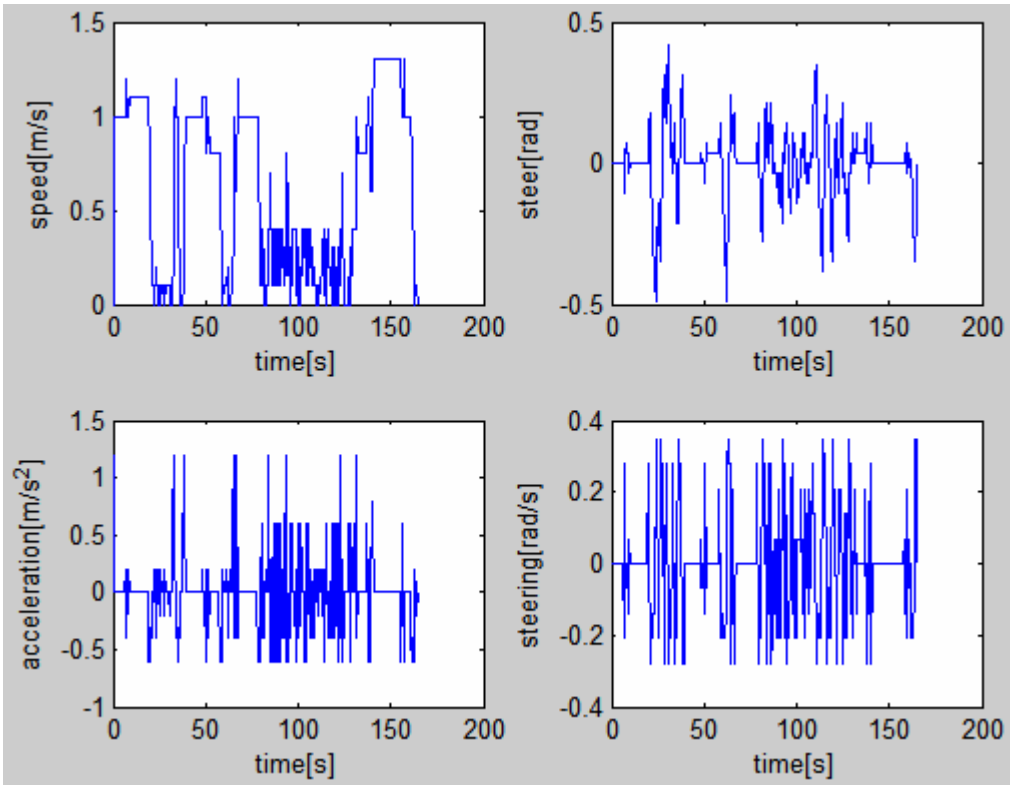


Figure 7.3 (a): AD planner shortest path states

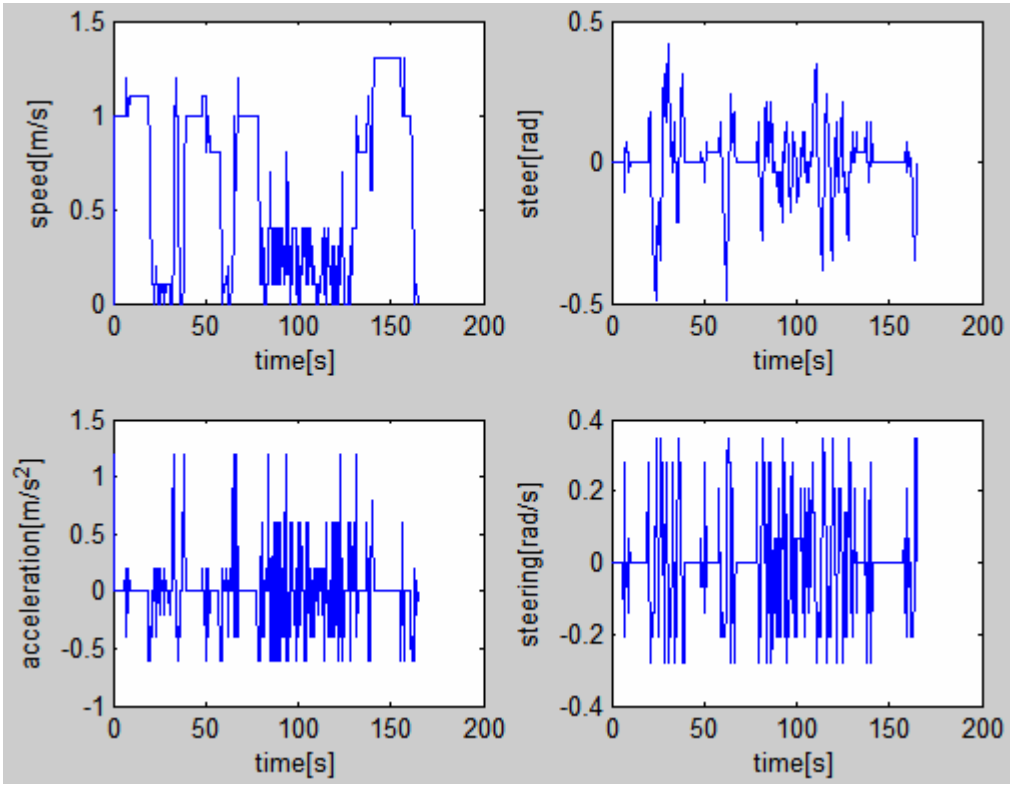


Figure 7.3 (b): ARA* planner shortest path states

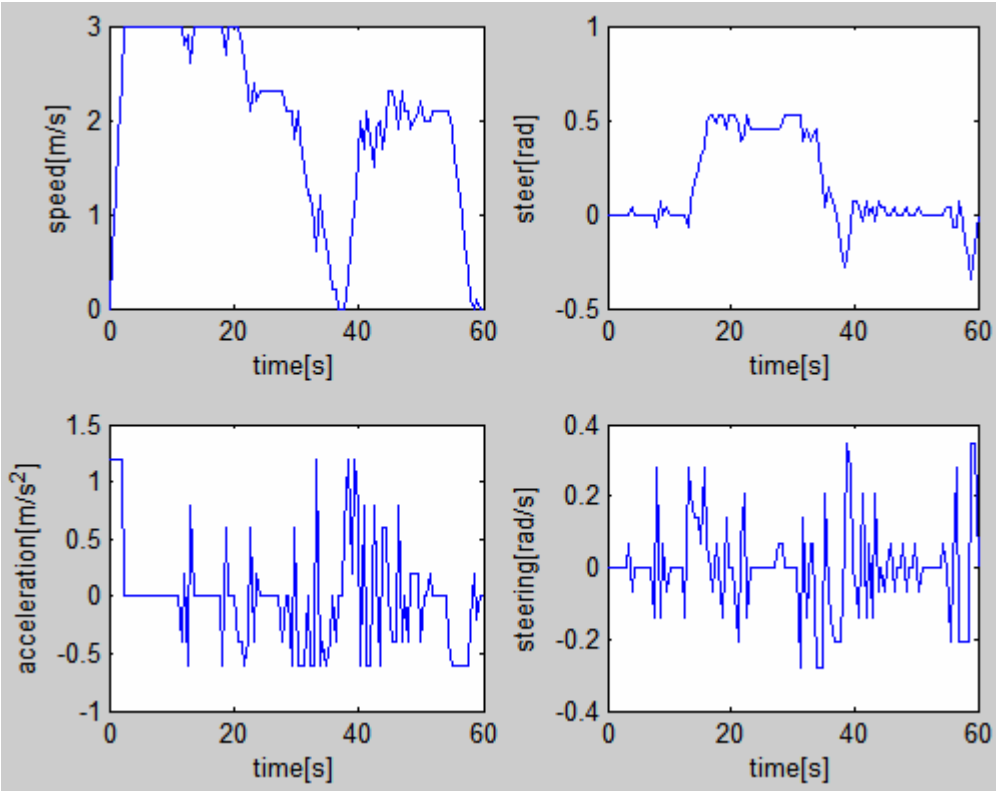


Figure 7.3 (c): ANA planner shortest path states

7.1.3 Benchmark of smoothest path between AD, ARA*, ANA planners

The environment map is $100\times100m^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,80) and goal position (90,85). Detail is given below.

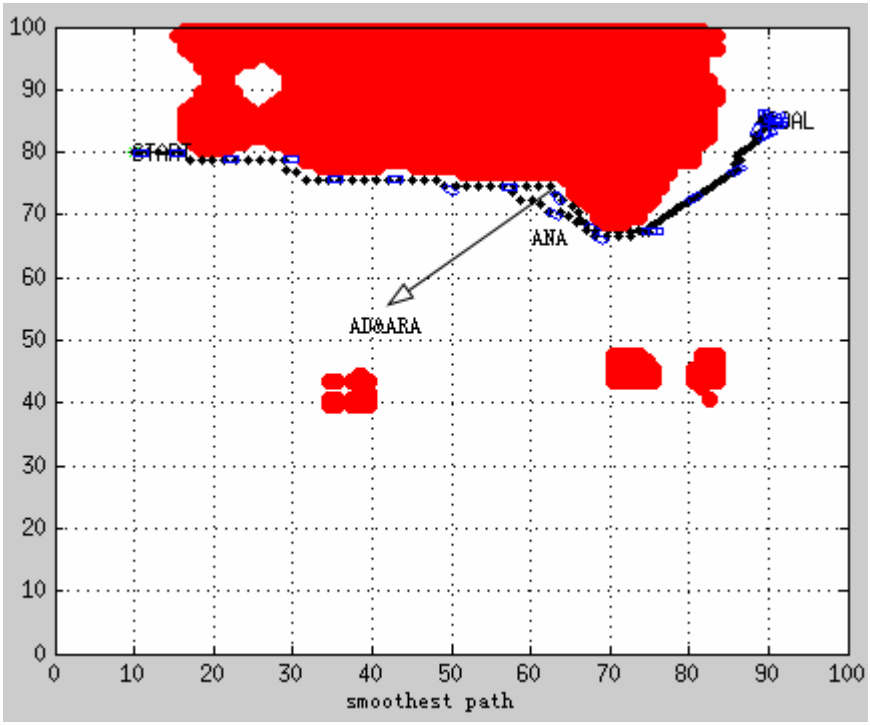


Figure 7.4: Footprint of smoothest paths of AD, ARA* and ANA planners

	Smoothest path travel distance	Smoothest path travel time
AD	107.86m	43s
ARA*	107.86m	43s
ANA	108.07m	47.5s

Table 7.4: Smoothest path details of AD, ARA* and ANA

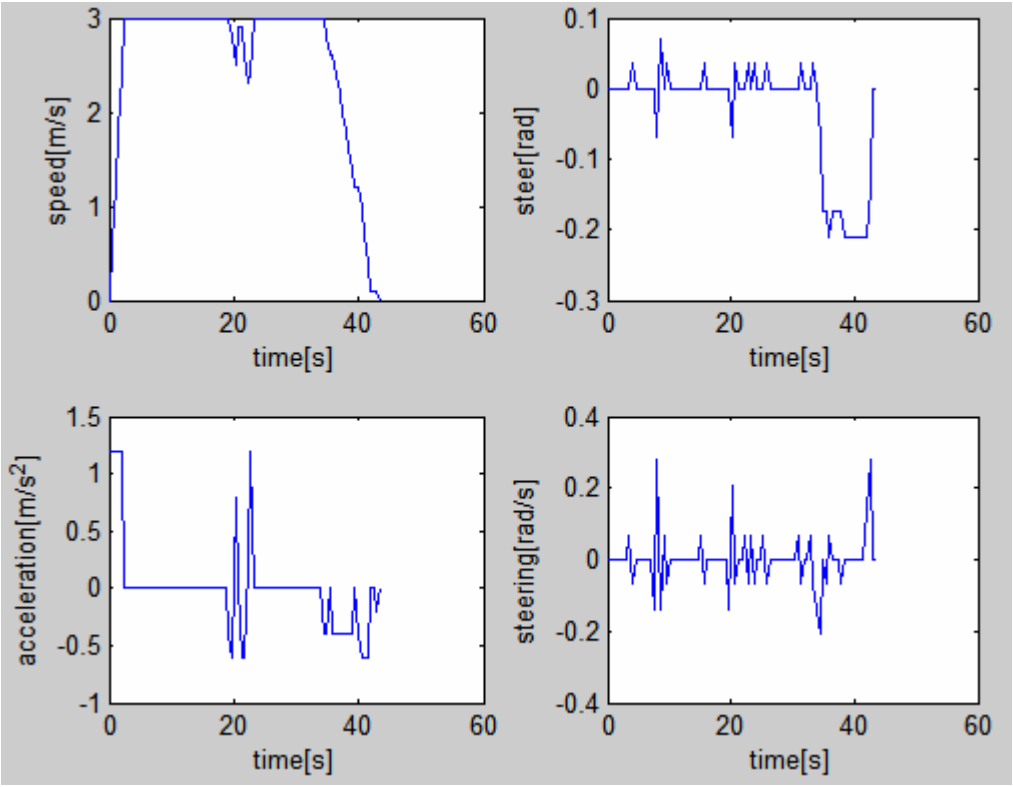


Figure 7.5 (a): AD planner smoothest path states

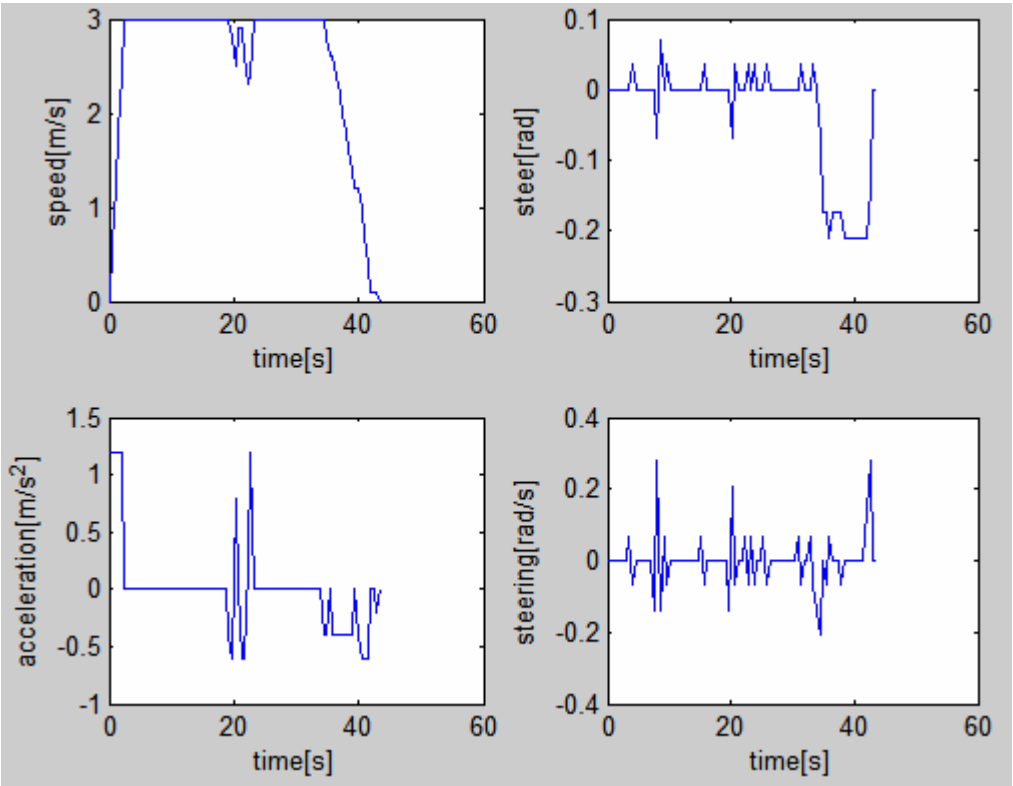


Figure 7.5 (b): ARA* planner smoothest path states

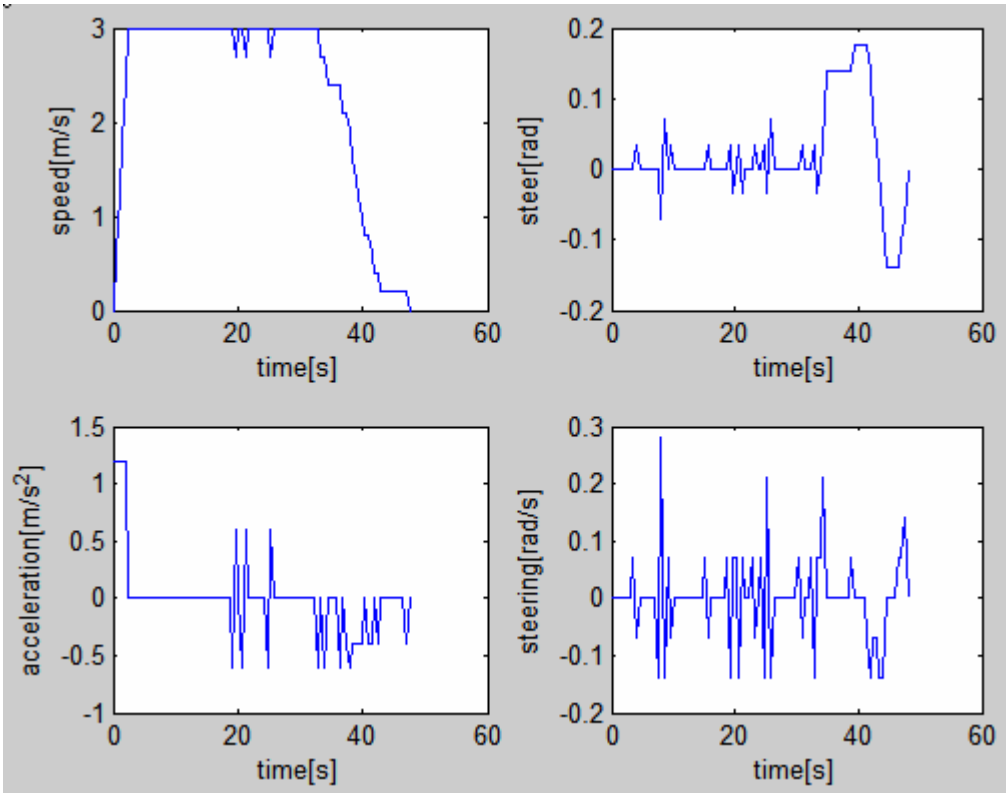


Figure 7.5 (c): ANA planner smoothest path states

7.2 Navigate through the obstacle

7.2.1 Motion Primitives, shortest path and smoothest path benchmark

This section is the performance analysis of a passing through the obstacle. The environment map is $100\times100m^2$, cell resolution is 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,10) and goal position (90,90). Detail is given below.

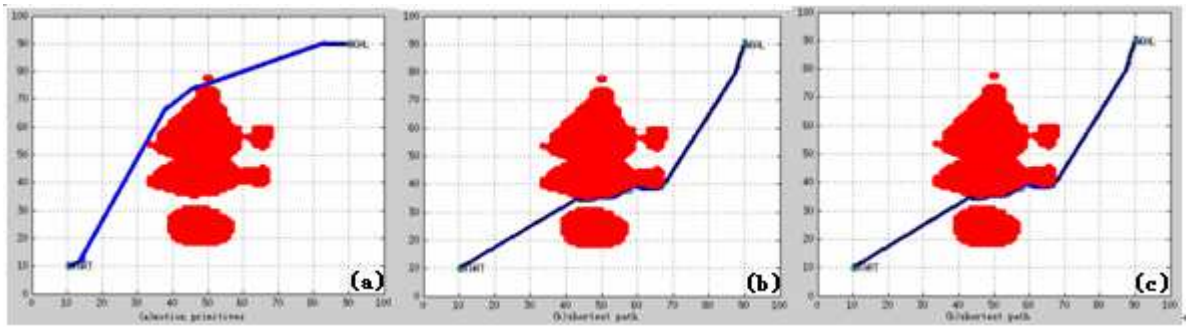


Figure 7.6: (a) motion primitives trajectory, (b) Shortest path trajectory, (c) Smoothest path trajectory

	Travel Distance
Motion primitive	122.64m
Shortest path	124.1m
Smoothest path	139.62m

Table 7.5: Travel Distance of different trajectories, notice that motion primitive path is shorter but unfeasible with respect to vehicle dynamics

7.2.2 Benchmark of shortest path between AD, ARA*, ANA planners

The environment map is $100\times100m^2$, cell resolution is 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,10) and goal position (90,90). Detail is given below.

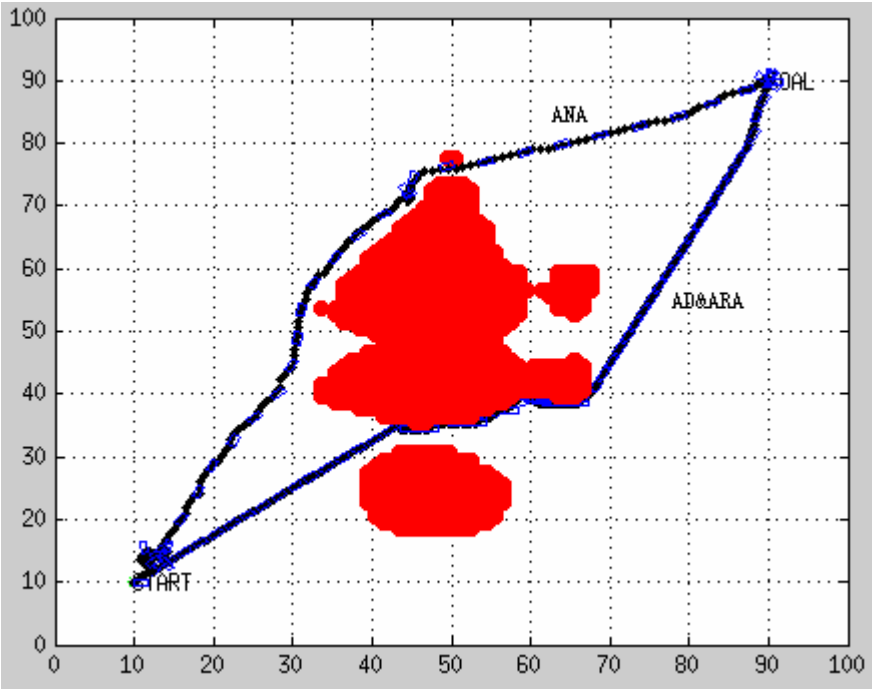


Figure 7.7: Footprint of shortest paths of AD, ARA* and ANA planners

	Shortest path travel distance	Shortest path travel time
AD	124.1m	161s
ARA*	124.1m	161s
ANA	170.11m	96s

Table 7.6: Shortest path details of AD, ARA* and ANA

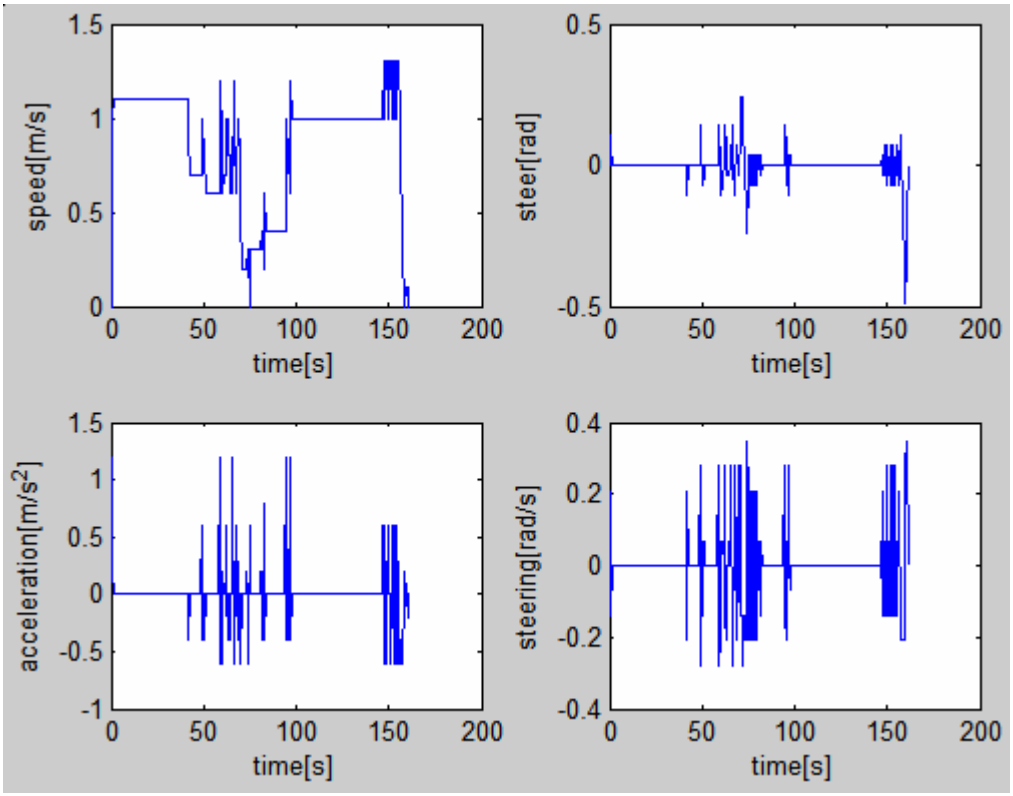


Figure 7.8 (a): AD planner shortest path states

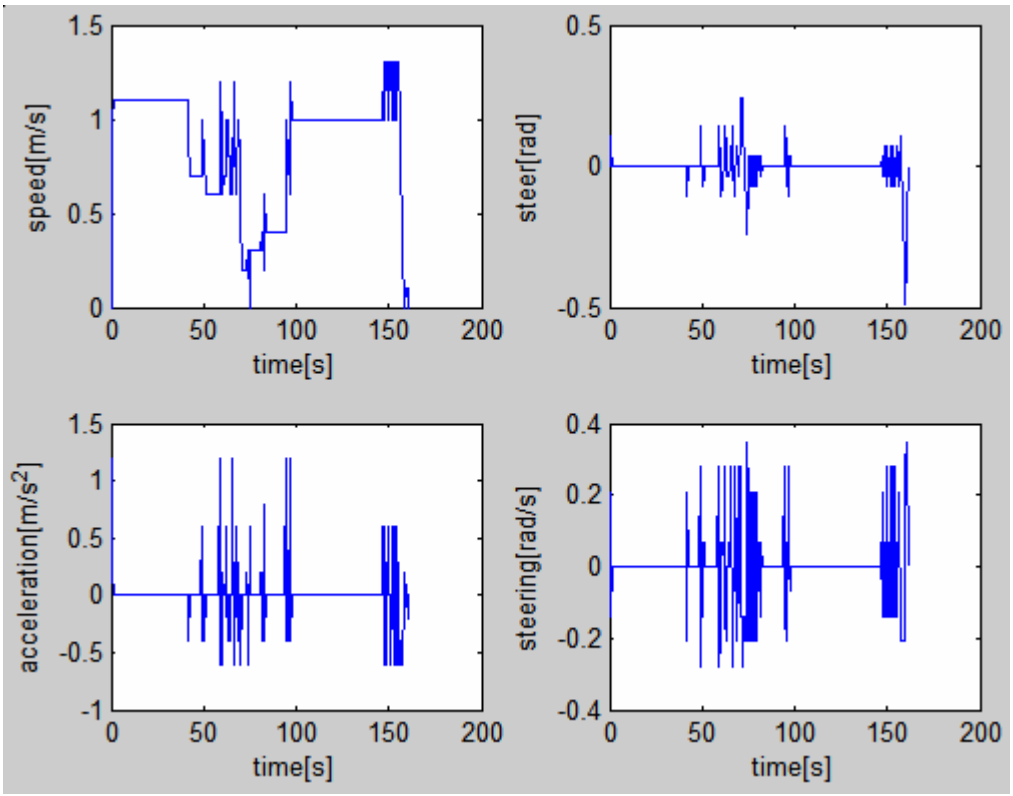


Figure 7.8 (b): ARA* planner shortest path states

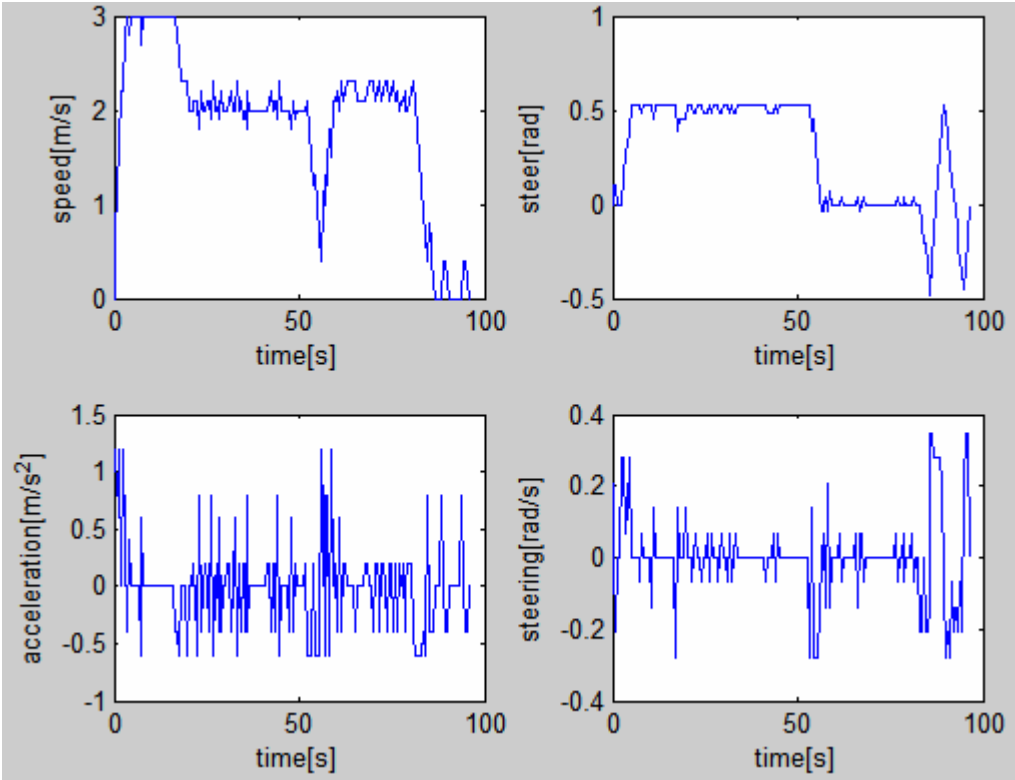


Figure 7.8 (c): ANA planner shortest path states

7.2.3 Benchmark of smoothest path between AD, ARA*, ANA planners

The environment map is $100\times100m^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,80) and goal position (90,85). Detail is given below.

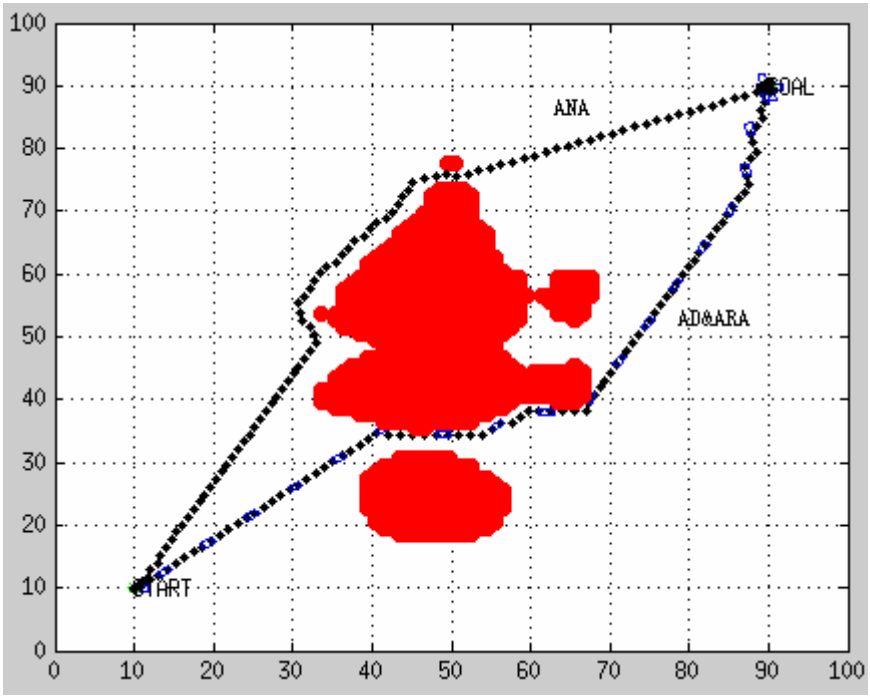


Figure 7.9: Footprint of smoothest paths of AD, ARA* and ANA planners

	Smoothest path travel distance	Smoothest path travel time
AD	139.62m	55s
ARA*	139.62m	55s
ANA	137.46m	53.5s

Table 7.7: Smoothest path details of AD, ARA* and ANA

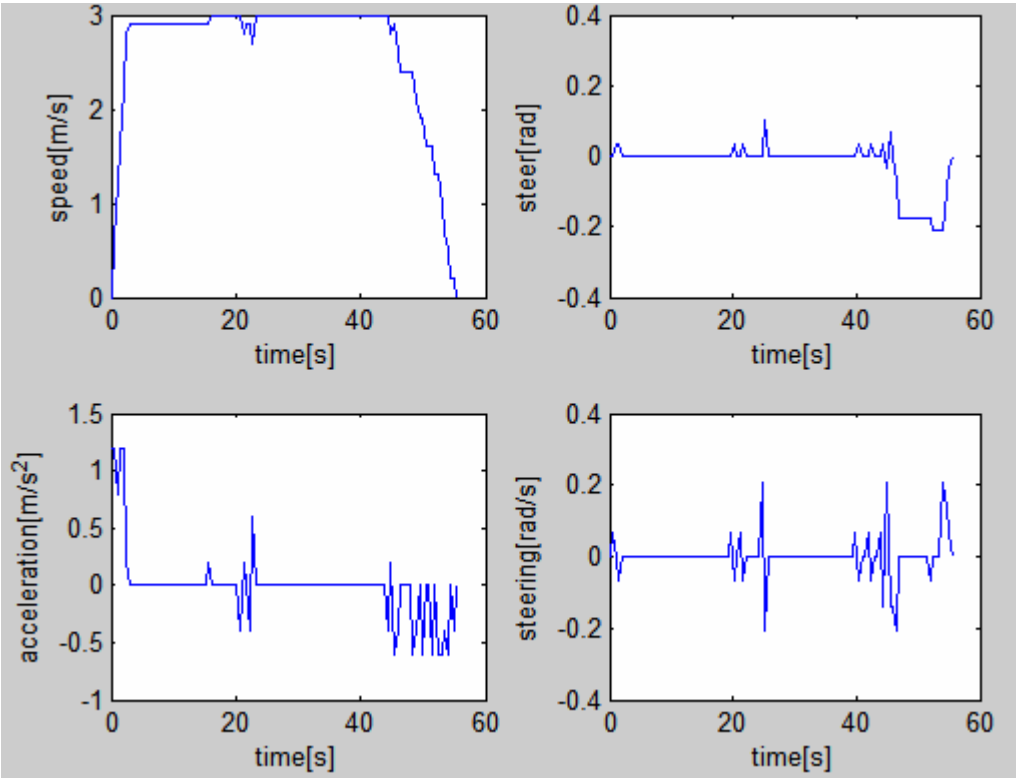


Figure 7.10 (a): AD planner smoothest path states

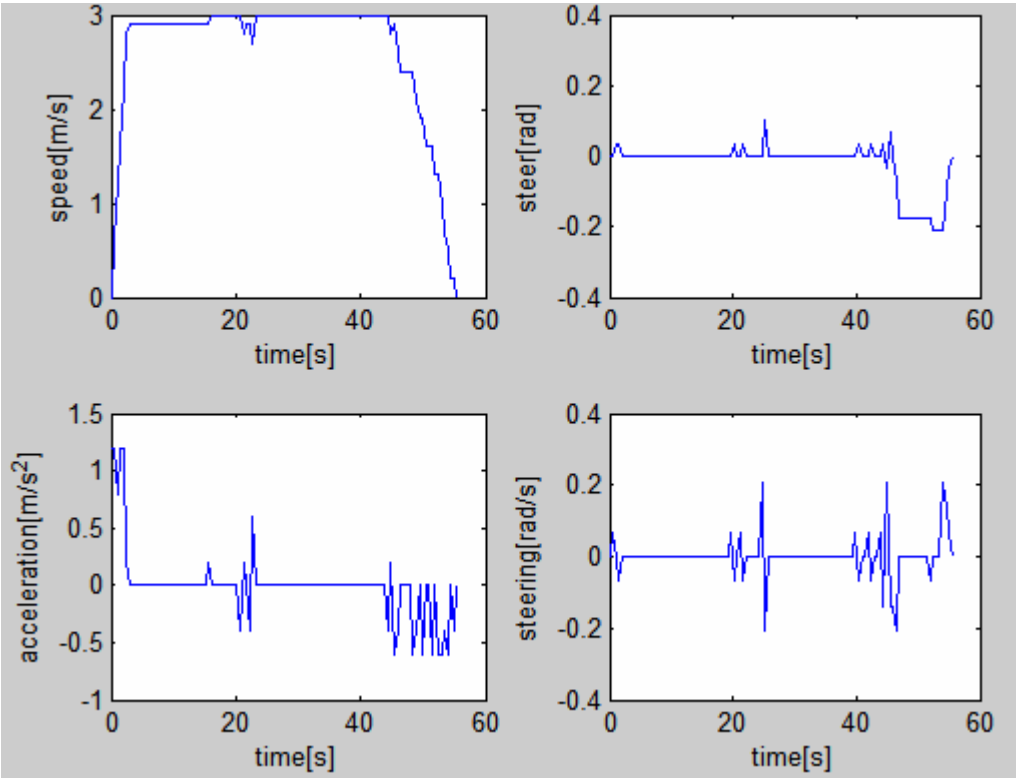


Figure 7.10 (b): ARA* planner smoothest path states

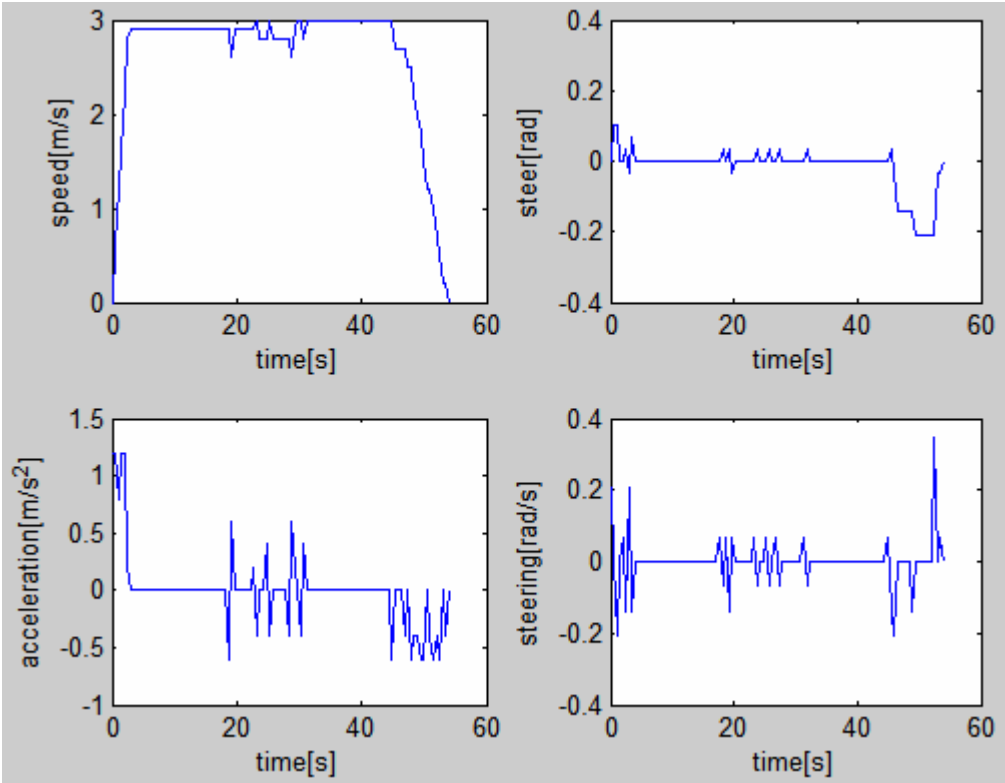


Figure 7.10 (c): ANA planner smoothest path states

8. Benchmark with roughness map

Roughness is used here to multiply the action cost function. It is an amplifier of action cost, which means that, perform the same action on higher roughness cost more. Roughness map is of the same size of the environment, for each grid, there is a one digit decimal integer. Higher is the roughness, larger this integer number.

8.1 Move along the obstacle

This section will present how the roughness affects the trajectory. Take the same environment in section 7.1, and given the following roughness map in Figure 8.1.

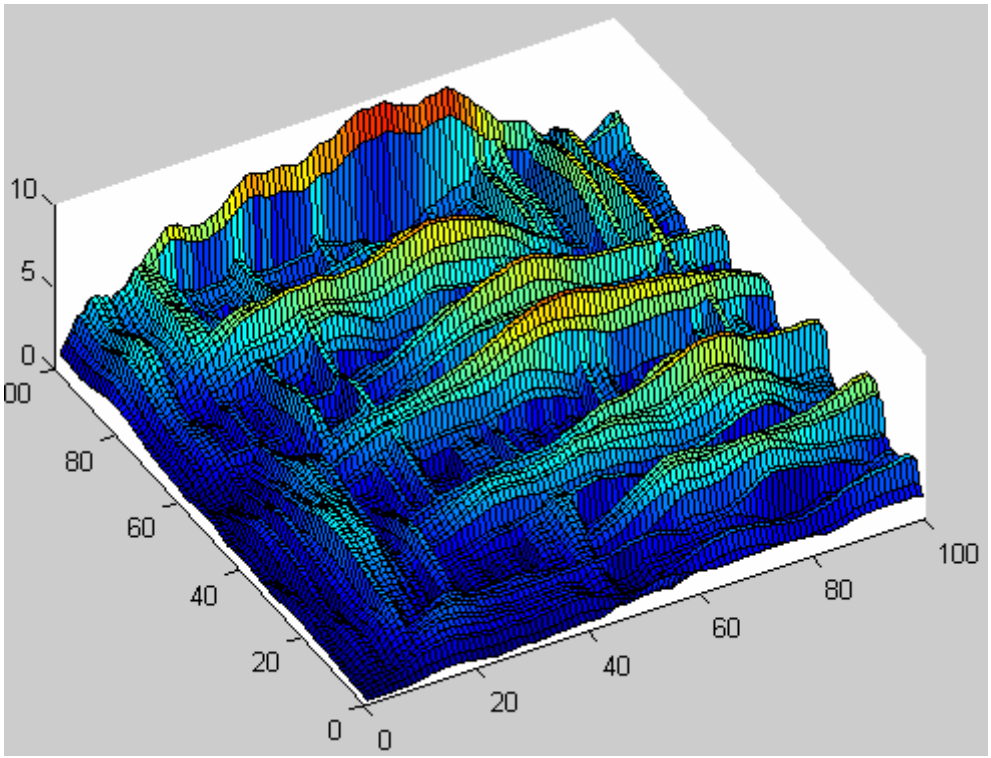


Figure 8.1: Roughness map

The environment map is $100\times100m^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,80) and goal position (90,85). Figure 8.2, 8.3 and 8.4(a) gives the solution footprints, while Figure 8.2, 8.3 and 8.4(b) includes the projection of

roughness on the environment map seperately. Details of the correponding trajectory is given in Table 8.1.

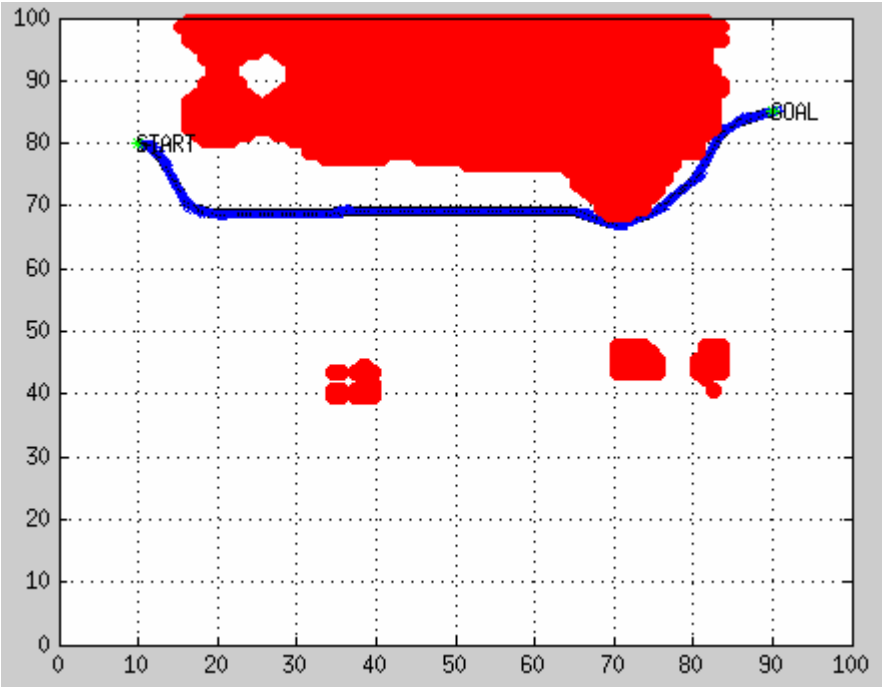


Figure 8.2(a): Motion Primitive trajectory footprint

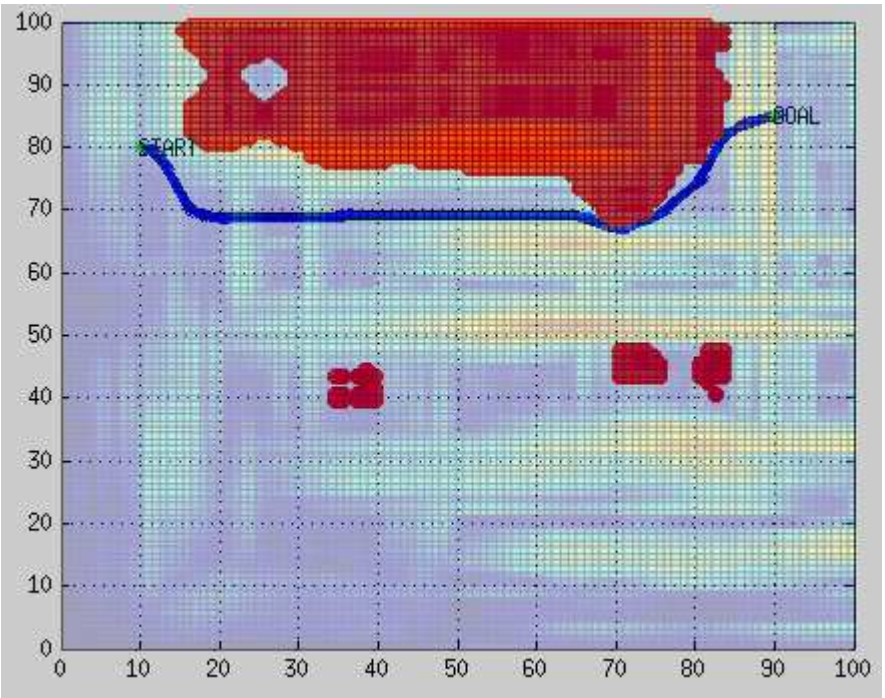


Figure 8.2(b): Roughness projection for motion primitive trajectory

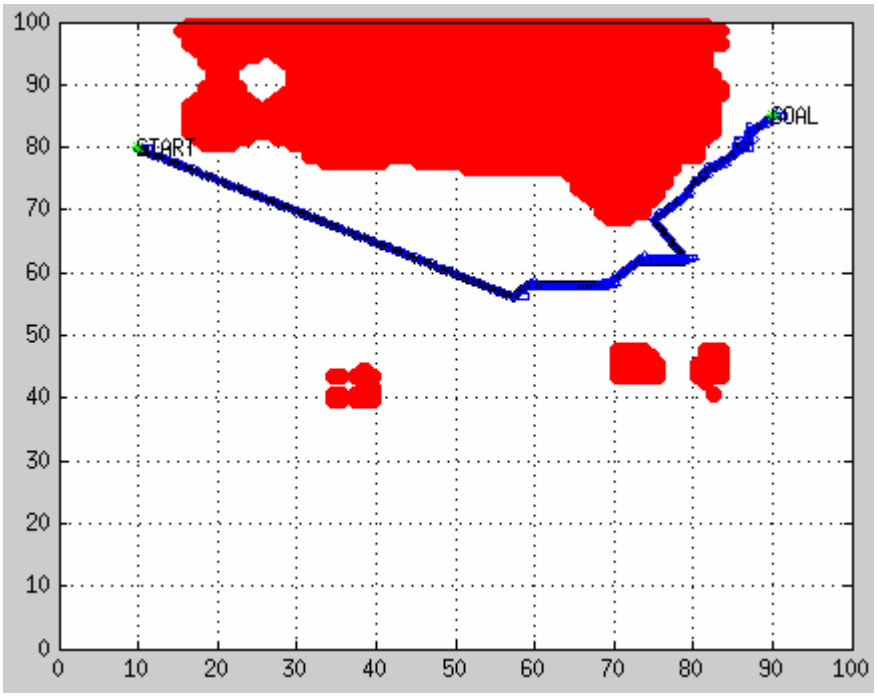


Figure 8.3(a): Shortest path trajectory footprint

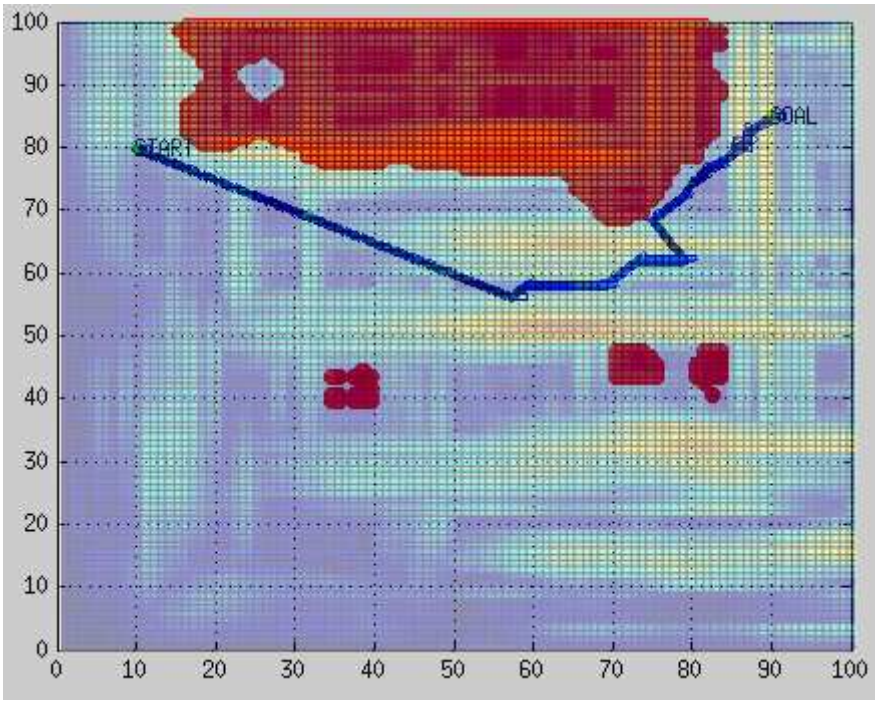


Figure 8.3(b): Roughness projection for shortest path

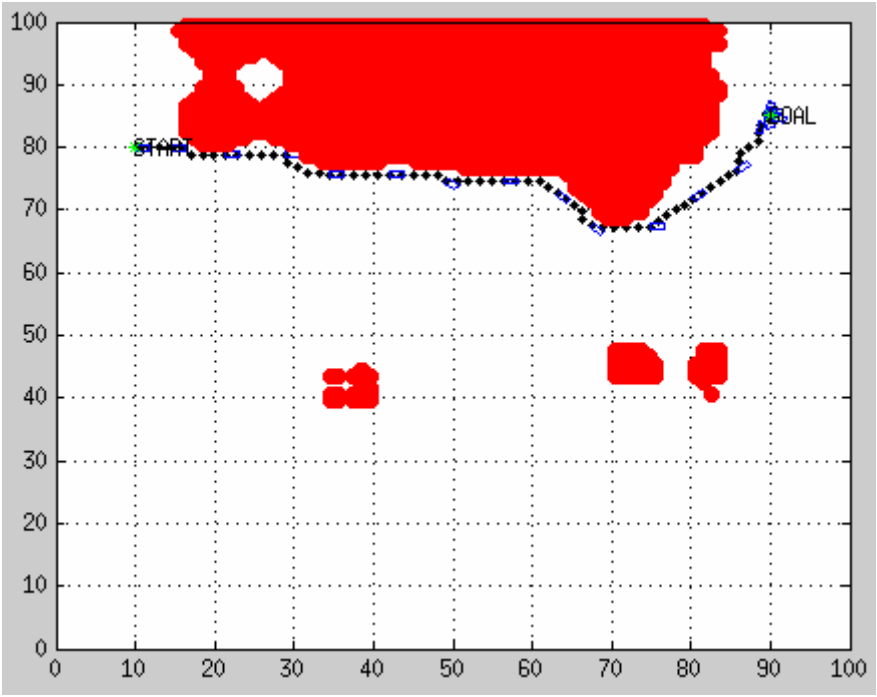


Figure 8.4(a): Smoothest path trajectory footprint

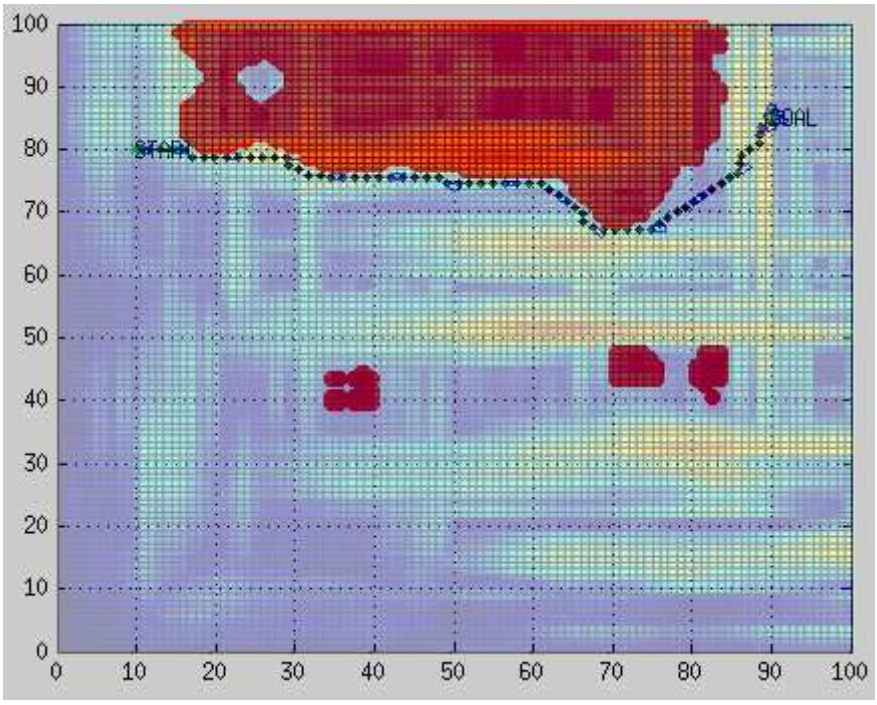


Figure 8.4(b): Roughness projection for smoothest path

Roughness	Travel Distance
Motion primitive	94.742m
Shortest path	108.49m
Smoothest path	105.63m

Table 8.1 Travel distance of different trajectories

8.2 Move through the obstacle

Take the same environment in section 7.2, and given the following roughness map in Figure 8.5.

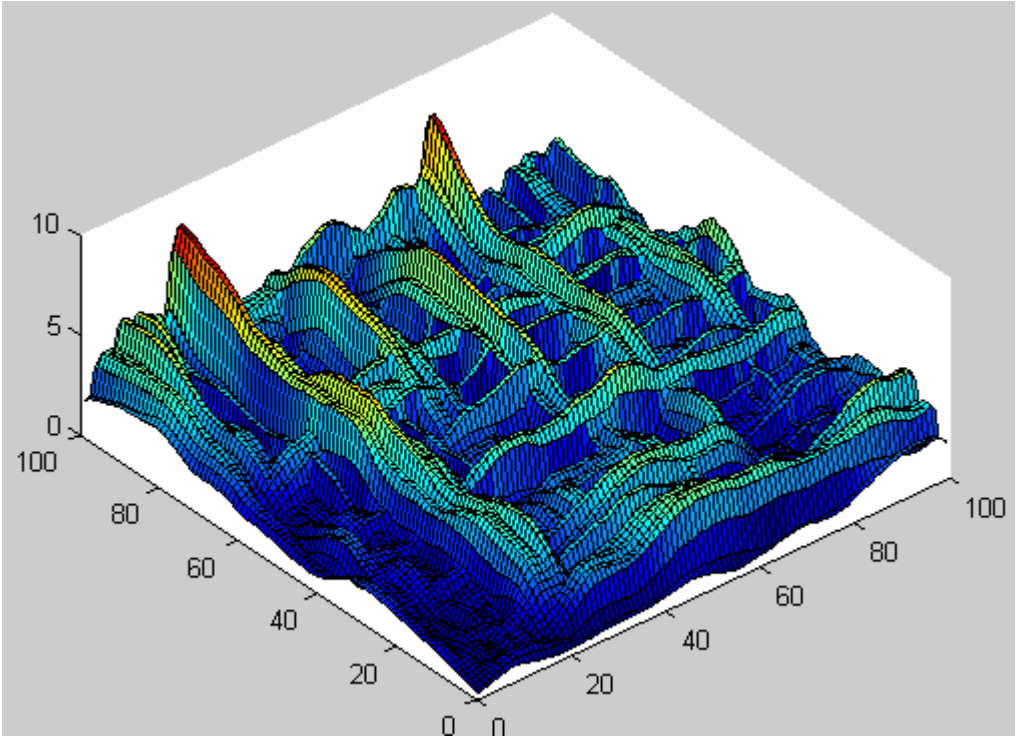


Figure 8.5: Roughness map

The environment map is $100\times100\text{m}^2$, cell resolution 0.1m, Red part is the obstacle, and blue trajectory is the solution. Start position (10,10) and goal position (90,90). Figure 8.6, 8.7 and 8.8(a) gives the solution footprints, while Figure 8.6, 8.7 and 8.8(b) includes the projection of roughness on the environment map seperately. Details of the correponding trajectory is given in Table 8.2.

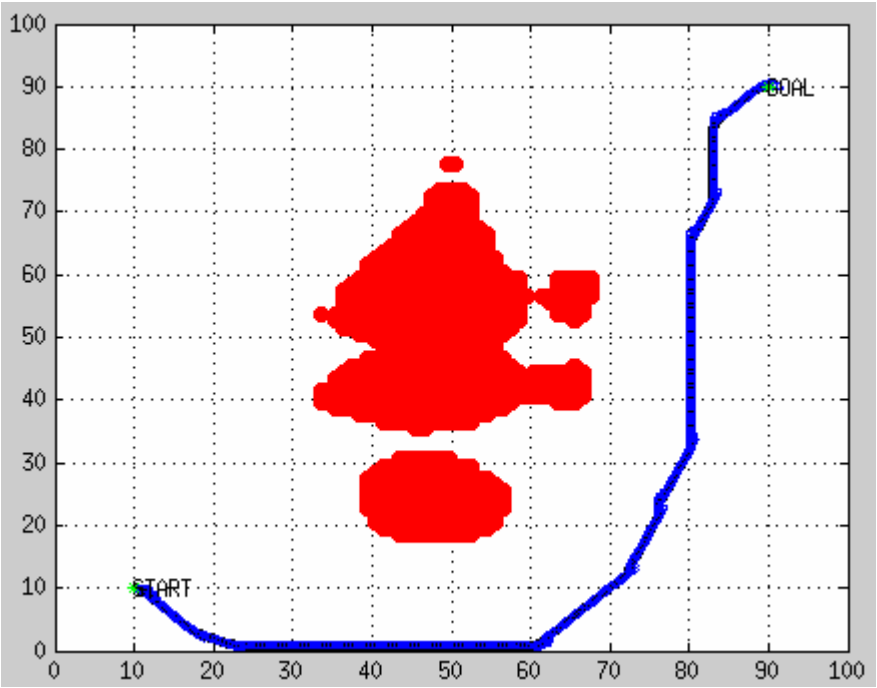


Figure 8.6(a): Motion Primitive trajectory footprint

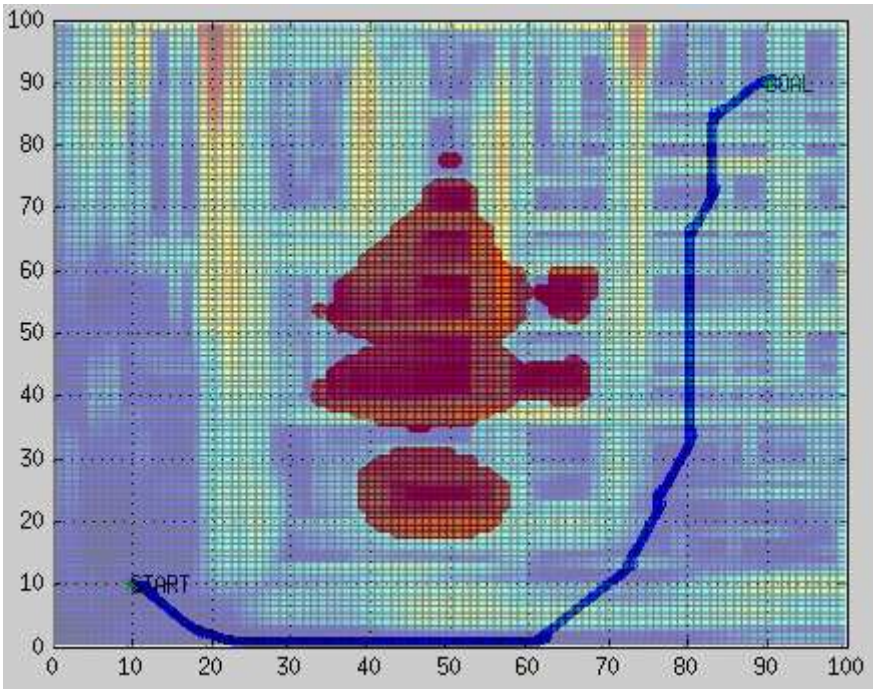


Figure 8.6(b): Roughness projection for motion primitive trajectory

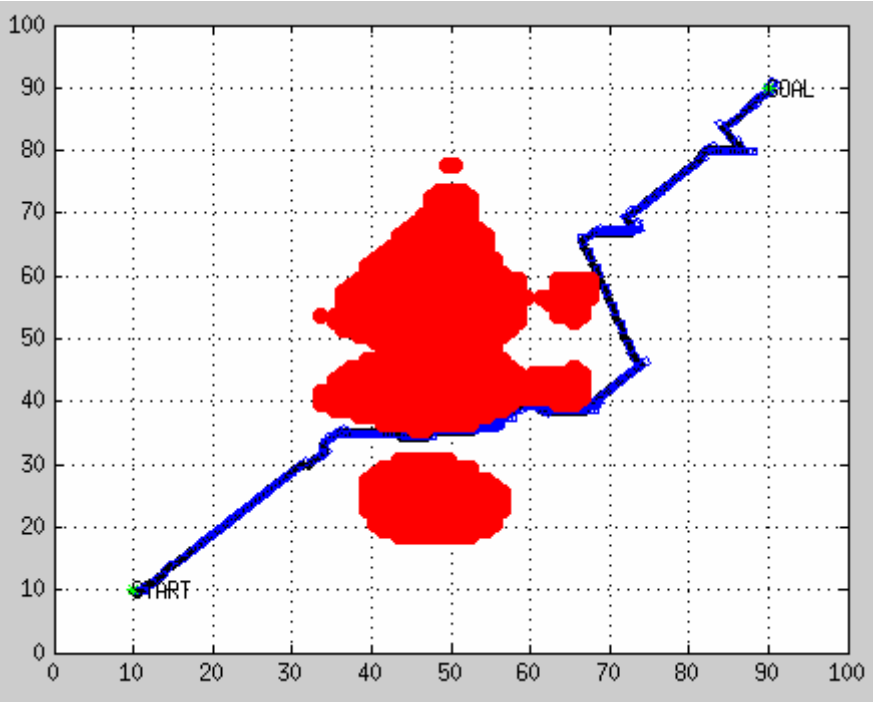


Figure 8.7(a): Shortest path trajectory footprint

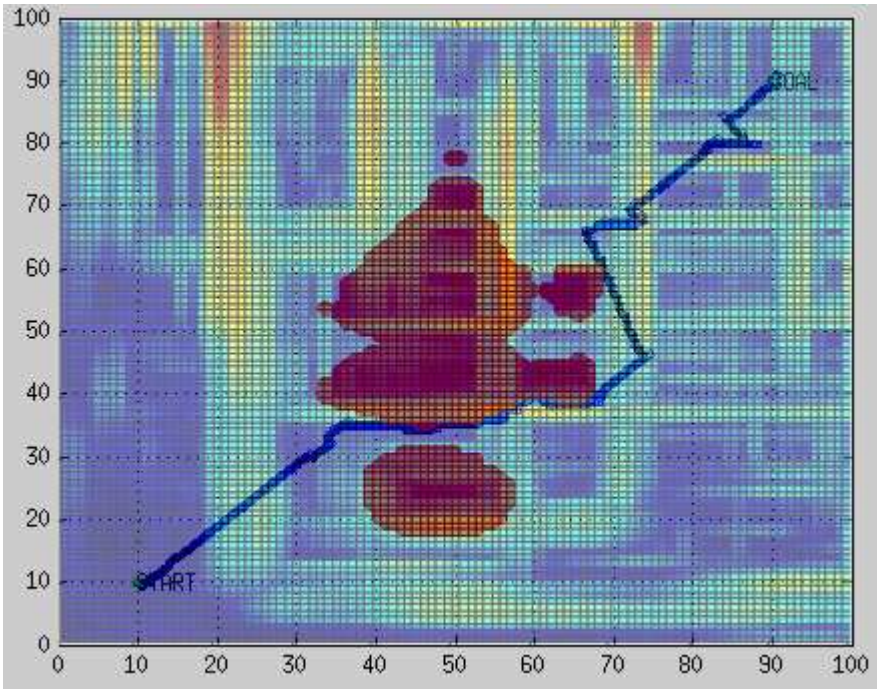


Figure 8.7(b): Roughness projection for shortest path

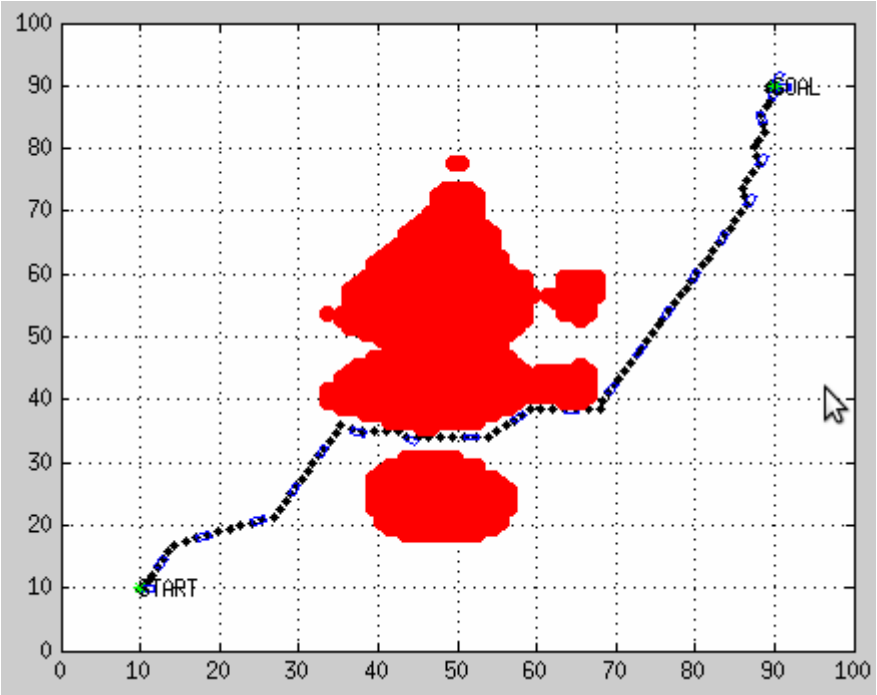


Figure 8.8(a): Smoothest path trajectory footprint

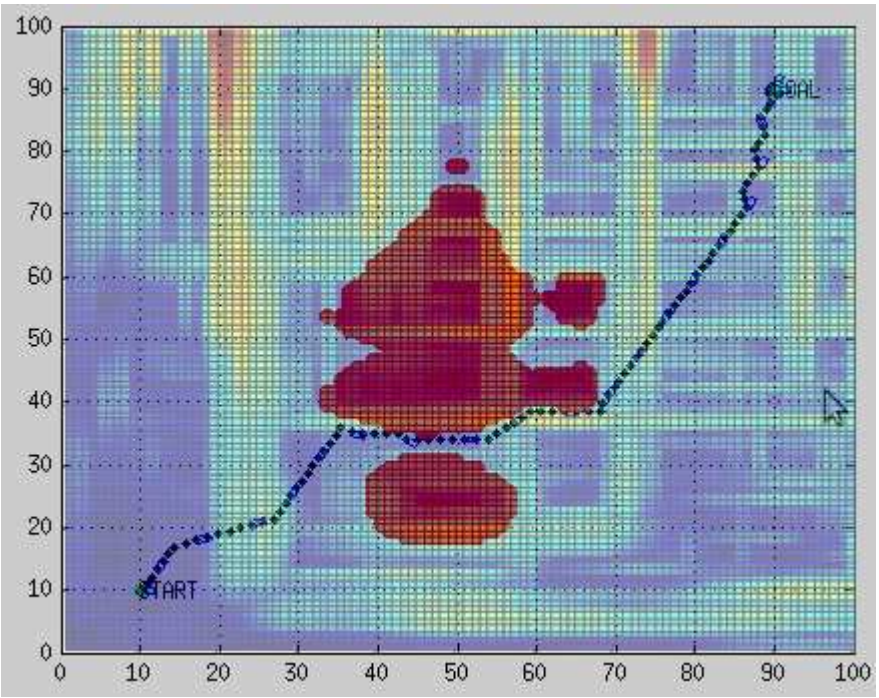


Figure 8.8(b): Roughness projection for smoothest path

Roughness	Travel Distance
Motion primitive	153.45m
Shortest path	147.06m
Smoothest path	141.56m

Table 8.2 Travel distance of different trajectories

Appendix A

Useful Navxytheta class functions:

Here state, state parameter are (x,y,yaw), stateID is the index in hash table for each state.

	Function name	Usage
1	ConvertStateIDPathinto-XYThetaPath(vector stateIDs, vector state parameters)	Iterate in vector stateIDs, search its related state parameter(x,y,theta) in hash table. Assign all of them in "vector state parameters".
2	GetSuccs(source stateID, successors vector, action vector)	Play with motion primitives, get all available successors of source state assigning corresponding actions. It is used in forward version of AD, ARA, ANA algorithms.
3	GetPreds(target stateID, predecessors)	Play with motion primitives, get all available predecessors of target stateassigning corresponding actions. It is used in backward version of AD, ARA, ANA algorithms.
4	GetStartHeuristic(stateID)	Assign the heuristic from state state to current state, return Euclidean distance between them.
5	GetGoalHeuristic(stateID)	Assign heuristic from current state to goal, which is the Euclidean distance also.
6	GetStateFromCoord(state)	Search in hash table by state parameters, return correponding state ID.
7	GetCoordFromState(stateID, state parameter)	Search in hash table by stateID, return correponding state parameter.
8	GetActionCost(source position, action increment)	Calculate action cost, return infinite cost if the action is not valid, collide with obstacle and outside the environment map.
9	InitializeEnv(environment	1. Read in environment data, start and goal state

	configuration file, vehicle shape, motion primitive configuration file)	parameter. Etc. 2. Read in motion primitive configuration file. 3. Read in model describing vehicle shape. 4. Precompute action data using PreComputeAction() function.
10	IsObstacle(position)	Check if position is an obstacle.
11	IsValidCell(position)	Check if current position is obstacle or out of environment map.
12	PreComputeActions()	Running before planning, pre-compute actions, assign their pose and intermediate points on motion primitive configuration data.
13	SetGoal(state parameter)	Create goal state in hashtable with given state parameter.
14	SetStart(state parameter)	Create start state in hash table with given parameter.
15	UpdateCost(position)	It is used to updata map data for a certain cell.

Appendix B

Useful NayATV class functions:

Here state and state parameter are in 5 degs of freedom(x,y,yaw,speed,steer angle), stateID is the index in hash table for the corresponding state.

	Function name	Usage
1	BodyCollisionCheck(position)	Collision problem check between vehicle body and obstacle. It use "position" as vehicle reference point, check 4 corner points, 4 mid points between corners and the geometric mid point.
2	CreateNewHashEntry(state)	Create a new state in hash table with a newly assigned stateID.
3	CreateStartAndGoalState (environment parameter)	Create start state and goal state in hash table, return their stateIDs.
4	GetHashEntry(state)	Return the pointer in hash table to the place where "state" is stored, return NULL if it the state is not in hash table.
5	GetActionCost(source, action)	Return individual price of action, which is roughness*cell cost*shortest path(smoothest path) cost. It will return infinite cost if source is not valid cell, or invalid action intermediate cell also.
6	GetGoalHeuristic(stateID)	Return heuristic from current state to goal, which is the Euclidean distance between them.
7	GetStartHeuristic(stateID)	Return heuristic from start to current state, which is the Euclidean distance between them.
8	GetSuccs(source stateID, succsID vector, cost vector)	Used in forward AD, ARA and ANA plannning algorithms, return all possible successor of source state,

		by assigning reasonable actions(direct increment on actual state). Combine all actions and action costs in "vector succsID" and "vector cost" seperately.
9	GetPreds(target stateID, predsID vector, cost vector)	Used in backward AD, ARA and ANA planning algorithms, return all possible predecessor of target state, by assigning reasonable actions(direct increment on actual state). Combine all action and action costs in "vector predsID" and "vector cost" seperatly.
10	GetStateFromCoord(state parameter)	Search in hash table corresponding index with the same state parameter. If there doesn't exist, create a new element in hash table.
11	IsValidCell(position)	Check if current position locate at an obstacle cell, or it is out of the environment map.
12	InitializeEnv(environment configuration file, vehicle shape configuration file)	Read in environment configuration file, roughness data, map data, start and goal, vehicle shape. Etc.

Bibliography

[1] S.M.LaValle, “Planning Algorithms”. Cambridge University Press, 2006, Available on: *http://planning.cs.uiuc.edu/.*

[2] D.Ferguson, M.Likhachev & A.T.Stentz, “A Guide to Heuristic-based Path Planning”, Proceedings of the *International Workshop on Planning under Uncertainty for Autonomous Systems*, *International Conference on Automated Planning and Scheduling (ICAPS),* June, 2005.

[3] D.Hsu, J.C.Latombe & R.Motwani, “Path Planning in Expansive Configuration Spaces”. *Proc. IEEE Int. Conf. on Robotics and Automation, 1997.*

[4] A* search algorithm, on Wikipedia. *http://en.wikipedia.org/wiki/A*_search_algorithm/.*

[5] Motion planning, on Wikipedia. *http://en.wikipedia.org/wiki/Motion_planning/.*

[6] M.Likhachev, D.Ferguson, G.Gordon, A.Stentz, & S.Thrun, “Anytime Dynamic A*: An Anytime, Replanning Algorithm”. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June, 2005.

[7] S.M.LaValle, “Motion Planning: The Essentials”. *IEEE Robotics & Automation magazine, vol 18, pp.79-89, June 2011.*

[8] S.M.LaValle, “Motion Planning: Wild Frontiers”. *IEEE Robotics & Automation magazine, vol 18, pp.108-118, June 2011.*

[9] SBPL library class, *http://www.ros.org/wiki/sbpl/.*

[10] Jur van den.Berg, Rajat.Shah, Arthur.Huang & Ken Goldberg, “ANA*: Anytime Nonparametric A*”. *Association for the Advancement of Artificial Intelligence: Annual Conference (AAAI)*. San Francisco, CA. Aug 2011.

[11]M.Likhachev, G.Gordon, & S.Thrun, “ARA*: Formal Analysis”. *Tech. Rep. CMUCS-03-148, Carnegie Mellon University, Pittsburgh, PA, 2003.*

[12] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, & Sebastian Thrun, “Anytime Search in Dynamic Graphs”. *Artificial Intelligence, vol.172, pp1613-1643, Sep 2008.*

[13]Sven Koenig, Maxim Likhachev, “D* Lite”, Proceedings of the *AAAI Conference on Artificial Intelligence (AAAI)*. *pp.476-483, 2002*.
[14] Maxim Likhachev, Geoff Gordon & Sebastian Thrun, “ARA*: Anytime A* with Provable Bounds on Sub-Optimality”, In *Advances in Neural Information Processing System 16: Proceedings of the 2003 Conference*, *2004.*
[15] Václav Hlaváč, “Motion Planning Methods” slides, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Cybernetics. Available on “*http://cmp.felk.cvut.cz/~hlavac/TeachPresEn/55IntelligentRobotics/100MotionPlanningMethods.pdf”*.