ACRL Homework 3 Report

Controlling a Simple Differential Drive Car Through a 2D Cluttered World

Po-Nien (Thomas) Lin and Caleb Harris

Introduction

A route planning algorithm A* [1] is implemented to generate an optimal path from the start position of the car all the way to the designated goal location. Then, DDP is used as a controller to navigate the waypoints throughout the journey towards goal.

A* Algorithm

In our simulation, the optimal routes for the car are generated through A* algorithm, also, these routes will be updated according to the detected state of bridges (or gates), which represented by the small gray squares in each map. Figure 1 demonstrates the examples of optimal route (blue dots) from the start (green dot) to the goal (red dot) at the beginning of the simulation or when the bridge state is updated. Beside A* algorithm, some other auxiliary codes are implemented to fine tune the route mainly keep them away from the walls and have better curvature around the corner of walls. Following Table 1 only presented the pseudo code of A* algorithm, the complete route generation code can be found in *aStart.m*.

Pseudo code

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
        remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor):
        remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
        set g(neighbor) to cost
        add neighbor to OPEN
        set priority queue rank to g(neighbor) + h(neighbor)
        set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

Table 1. The pseudo code of the A* algorithm [1]

Examples of Generated Routes of Different Maps and Bridge States

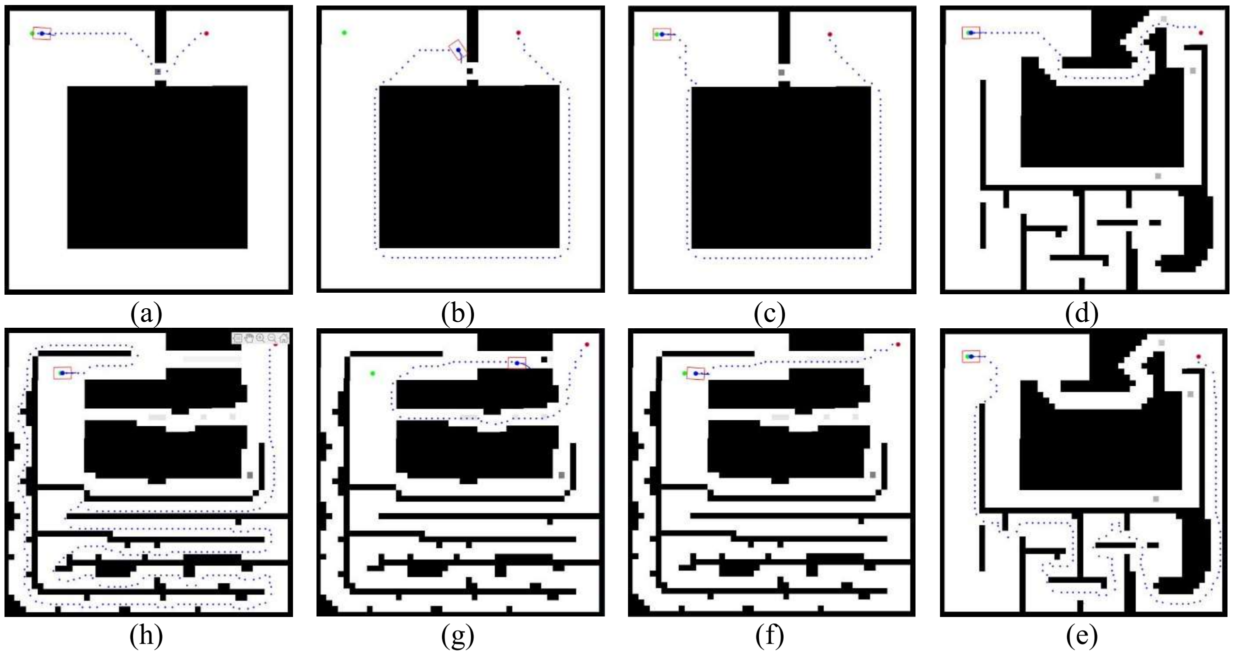

Figure 1. Examples of generated route by A* algorithm are presented. (a) – (b) demonstrate the how the route been updated when closed gate is detected in map 1; (a) before closed gate detected; (b) an updated route is generated when the gate is closed; (c) is the long route when assuming the gate is closed. (d) – (e) shows one short route and one long route for map 2. (f) – (g) also demonstrate the update of the shortest route when a closed gate is detected. (h) shows a long route generated in map 3.

Differential Drive Controller

In order to successfully navigate the A* waypoints a controller was required. Initially, a series of controllers were investigated. A PID controller was tested, but proved unsuccessful. Since the differential-drive system is unable to stop (as there is a nominal control input), the PID controller proved unsuccessful. Next, a linear controller was attempted. Linearizing the system dynamics proved rather difficult as the control input resulted in a highly nonlinear response. In the end, DDP, or Differential Dynamic Programming, was selected as it has shown success in cases with nonlinear dynamics that are difficult to linearize, such as the inverted pendulum problem.

DDP is described in detail in [2], therefore there is no equations described in this report. However, the key benefits of DPP are described here. DDP utilizes local dynamic programming methods and a quadratic approximation of the Q function. A pseudo-hamilton is formed from the second-order expansion of the Q-function. Also, the optimal control is defined as a perturbation from a nominal control seen here:

$$\delta u^{*}(\delta x) = k + K\delta x$$

The optimal control input is then found by line search:

$$\hat{u}_i = u_i + \alpha k_i + K_i(\hat{x}_i - x_i)$$

For this case, processing time was a critical factor. Therefore, a first order finite-difference was used providing much faster processing times.

After successful implementation of this controller, it was discovered that the controls must be limited. Two methods were tested: hard bounds on the control after the line search approximation and a squashing function. Ultimately, the squashing function was used to limit the control during both the backward and forward pass. The squashing function is described in [2] and seen here:

$$s(u) = (b_{max} - b_{min})/2 * \tanh(u) + (b_{max} + b_{min})/2$$

For our case, the controls are bounded from [-1, 1]. Thus, the squashing function is just tanh(u).

A range of cost functions were examined, but ultimately a tracking control was implemented for the next three waypoints. The cost function is then described by:

$$J_{\text{lqr}} = \bar{x}_N^T Q_N \bar{x}_N + \sum_{k=0}^{N-1} \bar{x}_k^T Q \bar{x}_k + \bar{u}_k^T R \bar{u}_k$$

A table of the final parameters used is seen below:

Horizon	18
Iterations	50
Dt	1.0
Qf	Diag(1, 1, 0)
R	0.01
Q	Diag(100,100,10)
Alpha	0.05

Though these final parameters showed promising results, a decision was made to use a dynamic horizon approach to the DDP algorithm. This was because of the range of A* solution distances that came from the tight corridors and wall padding calculations. The final adaptive parameters are described in the code below, where it is a based off the optimal ratio of horizon to waypoints multiplied by a percentage of the distance to the furthest point.

```
params.horizon = ceil(point_dis * 0.90 * (20/3));
params.iterations = ceil(point_dis * 0.85 * (100/3));
```

In combination with the controller, a set of logic functions were required to decide when to bypass a waypoint or when to determine when the waypoint was successfully met. A 0.1 space circle around the waypoints had to be met, unless one of the next 3 waypoints in the list was closer. Therefore, certain scenarios can be seen where a set of waypoints is skipped since the DDP controller's horizon is not long enough to reach the waypoints.

Results and Discussions

Map	**Sample1**	**Success**	**Processing Time [s]**	**Sim Time**
1	**1**	**1**	**90.44**	**992**
1	**2**	**1**	**70.24**	**698**

2	**[1:20]**	**0**	**43**	**378**
3	**2**	**1**	**247.24**	**1373**
3	**34**	**1**	**41.3**	**243**

Map 1

The planner and controller can successfully navigate Map 1 with and without the obstacle in the path. Thought, the second sample ended up having to backup for a period of time then circle around to get to the goal. The main reason for this is not optimizing the parameters for that specific scenario. The most robust parameters would successfully complete both maps, but were sometimes limited in the performance by those parameters.

Overall

- **100% Successfully completed**
- **80 s average processing time**

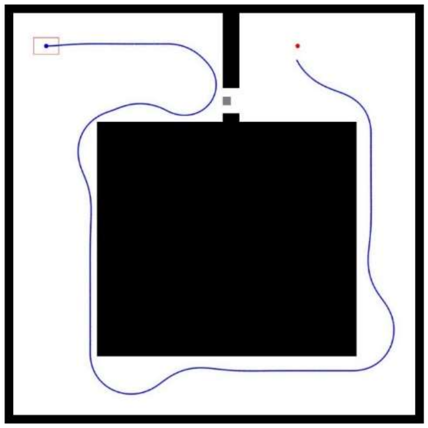

Figure 1: Map 1, Sample 1 Successful Path

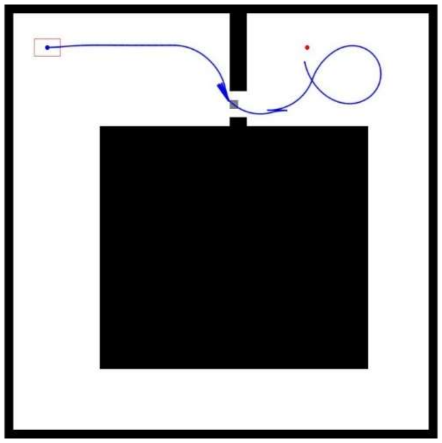

Figure 2: Map 1, Sample 2 Successful Path

Map 2

Map 2 was unsuccessful. This was because of the top corridor and sharp turns required. The easiest path to navigate was the second available path, however we did not want to force the algorithm to choose the "easy" path. Therefore it always attempted the top path, yet always failed.

Overall

- **0% Successfully completed**
- **40 s average processing time**

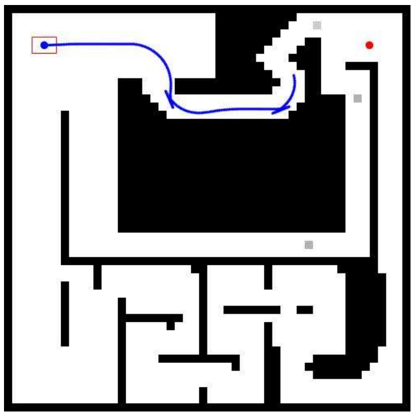

Figure 3: Map 2 Trajectory that was never successful

Map 3

Map 3 was also successful in a majority of samples. The figures below show successful navigation of two paths in the environment.

Overall

- **70% Successfully completed**
- **120 s average processing time**

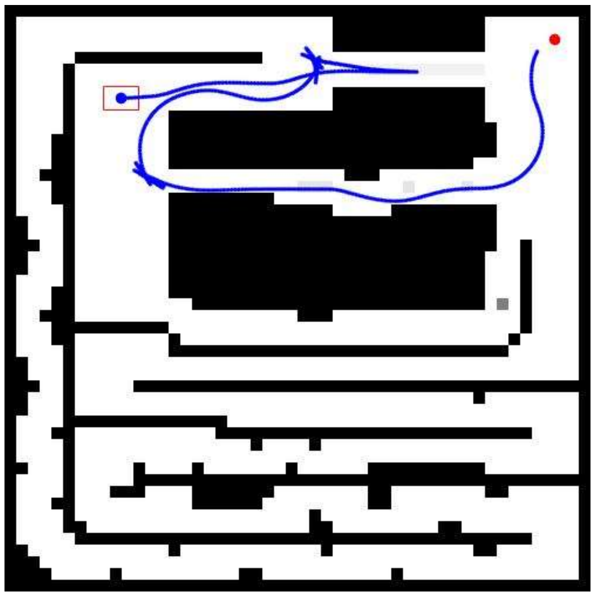

Figure 4: Map 3, Sample 2 Successful Path

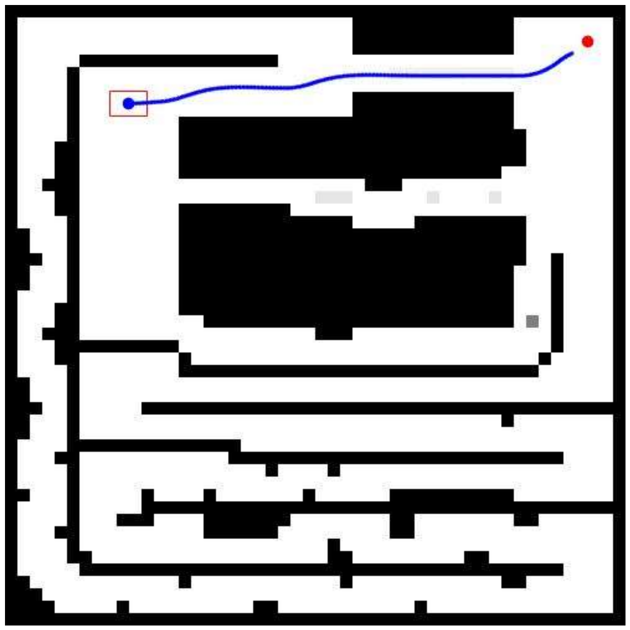

Figure 5: Map 3, Sample 34 Successful Path

Appendix

- The .mat files of the results can be found in the /results folder.
- The ddp code can be found in the /ddp folder.
- The astar algorithm is detailed in the astar.m file.
- The parameter updates and trajectory following can be found in run_sim.m.

References

[1] A* Algorithm, website: http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html

[2] Tassa, Mansard, Todorov, “Control-Limited Differential Dynamic Programming,” 2014.