**Dec Making Spring ’24**

**HW 3**

**Dept. of Mechanical Engineering**

**2019-17973 Seung Seok Oh**

**1. Approach**

In this problem, “Dubin’s car” is defined as a system that follows

where and

1-1. Low level planner

The A\* algorithm is an efficient algorithm for finding shortest paths, but it has the disadvantage that it can only find paths that follow discretize-defined nodes on the grid. An algorithm developed to improve this is Hybrid A\*, published in 2008 by D.Dologov et al[[1]](#footnote-1). While traditional A\* uses neighboring nodes to expand a node in the traversal graph, this algorithm samples a few reachable points at a node, taking into account its dynamics, and then includes the grid where the points are located in the graph.

Using Hybrid A\*, I implemented an algorithm to find the shortest path in the presence of obstacles, and some examples are shown below.

|  |  |
| --- | --- |
|  |  |

Figure 1 Result of hybrid A\* algorithm with obstacles

1-2. High level planner

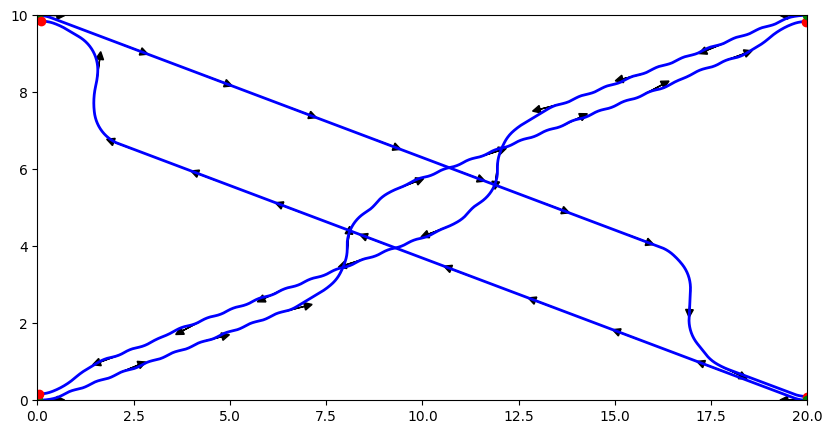
For Multi Agent Motion Planning, such as this problem, we need to evaluate the paths for conflicts and consider replanning if they are not. One of the algorithms to solve this problem is Conflict based search (CBS), published in 2012 by G.Sharon et al[[2]](#footnote-2). Traditional CBS assumes a constraint in the form of

when a conflict occurs and applies it to one of the two agents to repeat the replanning.

Inspired by this, I used an algorithm that simply assumes the existence of a constraint of the form and attempts to cascade replanning in order to reduce computational time. To be specific, when conflict is occurred at and , constraint circle with radius 0.5 is added in the environment, and every grid cells that contain this circle was considered as an static obstacle.

1-3 Result

Result of calculated path is shown as below.

****

It can be guaranteed that there is no collision (close within Euclidean distance 1) between any cars, because distance was checked in every timestep and if it is below 1, constraint was given

|  |  |  |
| --- | --- | --- |
| **T=1(s)** | **T=10(s)** | **T=10.5(s)** |
| **T=11(s)** | **T=12(s)** | **T=23.4(s)** |

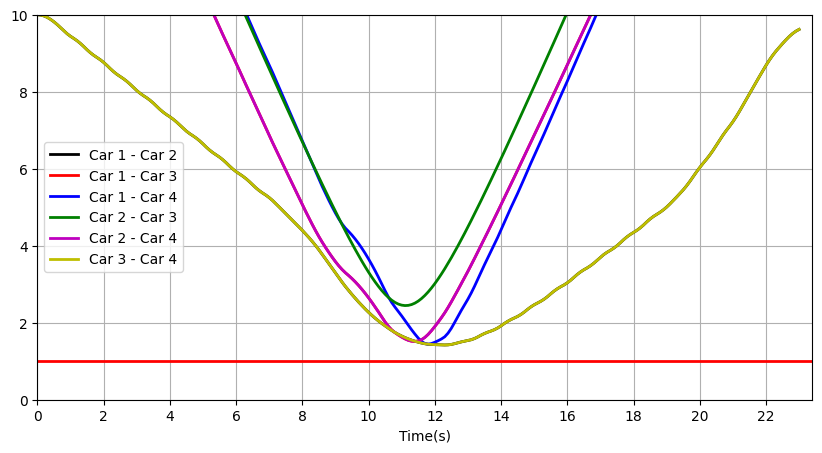
****

Figure 2 Distance between cars

**2. Time Optimization**

In Dubins’ car system, where the velocity is constant and set to 1 except at the goal, optimizing the path length directly correlates to optimizing the time taken to traverse the path.

Given that the cost function in the hybrid A\* algorithm is defined as the total length of the arc the car moves, the path length derived from this algorithm is guaranteed to be at least the shortest among the considered paths. However, due to the discretization of angular velocities in the car's actions, there may exist paths with shorter lengths that are not considered.

In the context of the Conflict-Based Search (CBS) algorithm, as the conflict tree expands, nodes representing paths with the minimum cost (which is the total path length) are explored first. This means that the algorithm prioritizes finding the shortest possible path. Consequently, the first combination of paths that is found to be collision-free and has the minimum path length becomes the solution. While we cannot claim that the found path is globally optimal, we can expect it to be among the shortest paths considered, as replanning is attempted from the shortest path among those evaluated. To prove optimality, it would be necessary to expand all generated nodes until a collision-free path is found.

**3. Discussion**

1) I had more ideas for better algorithms, but implementation difficulties prevented me from trying them.

One out them was using Dubins’ path (LSL, RSL, LSR, RSR). At first I didn’t use Dubins’ path because optimality of the length cannot be guaranteed. However, when approaching the goal, the generated node closest to the goal may not be feasible to reach due to orientation differences and car dynamics. Therefore, it would be better to check if a Dubins’ path can be generated for every expanded node. Whenever such a path is generated, we can consider it as a path candidate. Although integrating Dubins' paths may introduce additional computational load, selectively applying them when nearing the goal could balance feasibility and optimality.

2) While the CBS algorithm inherently favors path length minimization due to its search strategy, the overall time-optimality of the solution is subject to the resolution and completeness of the underlying pathfinding algorithm, in this case, the hybrid A\* algorithm. Therefore, the time-optimality of the CBS algorithm is closely tied to how well the hybrid A\* algorithm approximates the true shortest path in the discretized action space.

In this context, one of the limitations of my method is that it does not account for the fact that a car may pass through a specific constraint point at a different time than another car. As a result, unnecessary constraints may be imposed, leading to suboptimal pathfinding. If the pathfinding algorithm could consider time-dependent constraints, such as Spatio-Temporal Hybrid A\* Algorithm[[3]](#footnote-3), it would allow for more efficient use of space and time, potentially finding better paths with reduced travel times.

3) While working on this assignment, I realized, as mentioned in the class, that there is always a trade-off between the optimality of motion planning and computational resources. If an algorithm guaranteeing better optimality were implemented, it would likely result in longer computation times.

4. Code

------

**class** HybridAstar:

**def** \_\_init\_\_(self, car, grid, unit\_theta=pi/12, dt=1e-2):

self.car = car

self.grid = grid

self.unit\_theta = unit\_theta

self.dt = dt

self.start = self.car.start\_pos

self.goal = self.car.end\_pos

self.drive\_steps = int(sqrt(2) \* self.grid.cell\_size / self.dt) + 1

self.arc = self.drive\_steps \* self.dt

self.ws = [-1,-0.8,-0.6,-0.4,-0.2,0,0.2,0.4,0.6,0.8,1] # range of angular velocities

self.thetas = get\_discretized\_thetas(self.unit\_theta)

**def** construct\_node(self, pos):

theta = pos[2]

pt = pos[:2]

theta = round\_theta(theta % (2 \* pi), self.thetas)

cell\_id = self.grid.to\_cell\_id(pt)

grid\_pos = cell\_id + [theta]

**return** Node(grid\_pos, pos)

**def** simple\_heuristic(self, pos):

# Heuristic by Manhattan distance and orientation difference with goal

**return** abs(self.goal[0] - pos[0]) + abs(self.goal[1] - pos[1]) + 0.1\*theta\_diff(self.goal[2], pos[2])

**def** get\_children(self, node):

children = []

**for** w **in** self.ws:

pos = node.pos

branch = [pos[:3]]

**for** \_ **in** range(self.drive\_steps):

pos = self.car.step(pos, w, self.dt)

branch.append(pos[:3])

**if** **not** self.car.is\_pos\_safe(pos):

**continue**

child = self.construct\_node(pos)

child.w = w

child.parent = node

child.g = node.g + self.arc

child.g\_ = node.g\_ + self.arc

child.f = child.g + self.simple\_heuristic(child.pos)

children.append([child, branch])

**return** children

**def** backtracking(self, node):

route = []

**while** node.parent:

route.append((node.w, self.drive\_steps))

node = node.parent

**return** list(reversed(route))

**def** search\_path(self, min\_goal\_dist=1e-1,min\_goal\_dtheta=1e-1):

root = self.construct\_node(self.start)

root.g = float(0)

root.g\_ = float(0)

root.f = root.g + self.simple\_heuristic(root.pos)

closed\_ = []

open\_ = [root]

count = 0

**while** open\_:

count += 1

best = min(open\_, key=lambda x: x.f)

open\_.remove(best)

closed\_.append(best)

**if** same\_point(best.pos[:3], self.goal[:3],min\_goal\_dist,min\_goal\_dtheta):

route = self.backtracking(best)

path = self.car.get\_path(self.start, route)

cost = best.g\_

print('Shortest path: {}'.format(round(cost, 2)))

print('Total iterations:', count)

return path, cost

children = self.get\_children(best)

**for** child, branch **in** children:

**if** child in closed\_:

continue

**if** child not in open\_:

best.branches.append(branch)

open\_.append(child)

**elif** child.g < open\_[open\_.index(child)].g:

best.branches.append(branch)

c = open\_[open\_.index(child)]

p = c.parent

for b in p.branches:

if same\_point(b[-1], c.pos[:3],min\_goal\_dist,min\_goal\_dtheta):

p.branches.remove(b)

break

open\_.remove(child)

open\_.append(child)

**return** None, None

# Code Reference: <https://github.com/jhan15/dubins_path_planning>

**class** CBSNode:

**def** \_\_init\_\_(self, constraints=None, paths=None, cost=0):

self.constraints = constraints **if** constraints **else** []

self.paths = paths **if** paths **else** []

self.cost = cost

**def** \_\_lt\_\_(self, other):

**return** self.cost < other.cost

**def** generate\_constraints(conflict):

constraints = []

loc, time, agent1, agent2 = conflict[0], conflict[2], conflict[3], conflict[4]

constraints.append((agent1, loc, time))

constraints.append((agent2, loc, time))

**return** constraints

**def** cbs():

root = CBSNode()

root.paths, root.cost = multi\_car\_hybrid\_astar()

open\_set = []

heapq.heappush(open\_set, root)

**while** open\_set:

node = heapq.heappop(open\_set)

conflict = check\_path\_collision(node.paths)

**if** not conflict:

**return** node.paths, node.cost

constraints = generate\_constraints(conflict)

**for** constraint **in** constraints:

new\_constraints = node.constraints + [constraint]

new\_node = CBSNode(constraints=new\_constraints)

new\_node.paths, new\_node.cost = multi\_car\_hybrid\_astar(conflicts=new\_constraints)

heapq.heappush(open\_set, new\_node)

**return** None

1. Dolgov, Dmitri, et al. "Practical search techniques in path planning for autonomous driving." Ann Arbor 1001.48105 (2008): 18-80. [↑](#footnote-ref-1)
2. Sharon, Guni, et al. "Conflict-based search for optimal multi-agent pathfinding." *Artificial intelligence* 219 (2015): 40-66. [↑](#footnote-ref-2)
3. Wang, Wenjie, and Wooi-Boon Goh. "Multi-robot path planning with the spatio-temporal A\* algorithm and its variants." *Advanced Agent Technology: AAMAS 2011 Workshops, AMPLE, AOSE, ARMS, DOCM 3 AS, ITMAS, Taipei, Taiwan, May 2-6, 2011. Revised Selected Papers 10*. Springer Berlin Heidelberg, 2012. [↑](#footnote-ref-3)