MULTI-AGENT PATH FINDING PROJECT

Seminar in Robotics - 236824

Nitzan Madar

29/4/2020

Table of Content

1 Task 1: Implementing Space-Time A*

1.1 Searching in the Space-Time Domain

1.1.1 The key/value pair was added to the variables "`root`" and "`child`" by using the next lines –

```
root = { 'loc': start_loc,
         'g_val': 0,
         'h_val': h_value,
         'parent': None,
         'timestep': 0 }
```

the "`root`" node need to be initialize for start location, g value zero, h value as computed at the function (using Dijkstra algorithm), no parent and time step zero.

```
child = { 'loc': child_loc,
          'g_val': curr [ 'g_val' ] + 1,
          'h_val': h_value [ child_loc ],
          'parent': curr,
          'timestep': curr ['timestep'] + 1 }
```

any "child" node is under the loop and have value as written above, where `curr` is popped node from the open list and `child_loc` found by the move and auxiliary function.

1.1.2 Here, we asked to change the `close_list` key to support not only the location but also the time step. I did it by simply change the key to the asked tuple. For example, after pushing the root node it will be using from closed list by calling:

```
closed_list[(root['loc'], root['timestep'])]
```

1.1.3 Adding the "wait" action, is simply done by adjusting the given "move" function, which contains x and y direction move (down (0, -1); up (1,0); right (1,0), left (-1,0)), and I added also the wait direction (0, 0) and update the loop to run 5 time for each move instead of 4 (under A*).

```
def move(loc, dir):
    directions = [(0, -1), (1, 0), (0, 1), (-1, 0), (0, 0)]  #(0,0) is WAIT
"DIRECTION"
    return loc[0] + directions[dir][0], loc[1] + directions[dir][1]
```

1.2 Handling Vertex Constraints

Implement "`build_constraint_table`" and "`is_constrained`" functions.

`build_constraint_table(constraints, agent):` this function first filter the constraints by the given agent and then put in into dictionary (sorted by time step).

```
def build_constraint_table(constraints, agent):
    ##############################
    # TODO 1.2/1.3: Return a table that contains the list of constraints of
    #               the given agent for each time step. The table can be used
    #               for a more efficient constraint violation check in the
    #               is_constrained function.
    #               constrain example: {'agent': 2, 'loc': [(3,4)], 'timestep': 5}

    filtered_agents_constraints = list(filter(lambda con: con['agent'] is
agent, constraints))

    timestep_loc_list = list(map(lambda con: (con['timestep'], (con['loc'],
con['positive'])), filtered_agents_constraints))
    const_table = {}
    for timestep, loc_pos in timestep_loc_list:
        if timestep in const_table:
            const_table[timestep].append(loc_pos)
        else:
            const_table[timestep] = [loc_pos]

    return const_table
```

`is_constrained(curr_loc, next_loc, next_time, constraint_table):` this function changed few times while building the project. The main idea is to check if a move violates any given constraint.
The first function was just and simple check, then I had to update it by using "`INF`" time to marked "goal constraint" (when agent finish his path, the goal location is a constraint for any other agents from time after he finished until the end of the process), and the last change was to support positive and negative constraint (see 4.1).

In this section, we just asked to check for vertex collision, which means checking only if location and time step is violate the constraints table.

1.3 Adding Edge Constraints

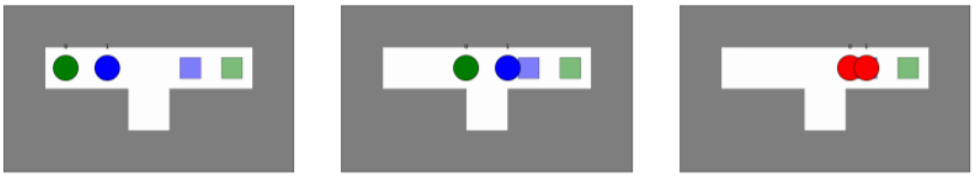
As the same as 1.2, but here we asked to check for edge collisions. Edge constraints have a list of 2 tuples – which stands for edge between 2 location (`curr_location` and `next_location`), and now we had to check if other agent uses that edge at the forbidden time step.

1.4 Handling Goal Constraints

We had to support a constraints after agent find his path to the goal location (in other words, agent a_i who starts at s_i and finish at g_i at time step t won't be affected by (negative) constraint at $T > t$). So, in the A* implementation a check for that problem was needed – for any time step after the agent finish his path, I checked if his goal violate the constraints table (my "`no_goal_constraints`" function do it, it also changed to check positive constraints – see (4) disjoint splitting section).

1.5 Optional: Designing Constraints

Here we asked to design minimal constraints set by hand that allows the algorithm to find collision-free paths with minimal sum of path lengths for the next problem:



It's easy to see that we need to enforce agent 1 (blue) to go down at time step 2 to get any solution, and without any "wait" it will be also an optimal solution. assuming the algorithm take him to (1,3) at time step 1, we need add 3 constraints that cancel the 3 other options (move left, move right or wait at (1,3)), so 3 constraints are needed to cancel each options:

```
constraints = [{'agent' : 1, 'loc' : [(1,2)], 'timestep' : 2},
               {'agent' : 1, 'loc' : [(1,3)], 'timestep' : 2},
               {'agent' : 1, 'loc' : [(1,4)], 'timestep' : 2}]
```

2 Task 2: Implementing Prioritized Planning

2.1 Adding Vertex Constraints

Here implementation of 2 loops is needed, one to iterate over the path (cell in path) of the current agent and one to add vertex constraint for all future agent (i+1 to N, I is the current agent in the main loop, N stands for number of agents). In addition, a variable t is added which consistent to time step of the cell in path. At every loop we added a vertex constraint as the above:

```
{'agent': future_agent, 'loc': [cell], 'timestep': t}
```

2.2 Adding Edge Constraints

Using the same loops as discussed in 2.1, under the condition $t < T_{max} - 1$ (where T_{max} stands for current path length), we added the relevant edge constraint:

```
{'agent': future_agent, 'loc': [path[t+1], cell],
'timestep': t+1}
```

```
for i in range(self.num_of_agents):  # Find path for each agent

    path = a_star(self.my_map, self.starts[i], self.goals[i],
self.heuristics[i],
                  i, constraints)
    if path is None:
        raise BaseException('No solutions')
    result.append(path)

    ##############################
   # TODO 2: Add constraints here
   #        Useful variables:
   #           * path contains the solution path of the current (i'th) agent, e.g., [(1,1),(1,2),(1,3)]
   #           * self.num_of_agents has the number of total agents
   #           * constraints: array of constraints to consider for future A* searches
   ##############################
    # max_path_lengths = max(len(path), max_path_lengths)

    t = 0 # time variable for current agent
    for future_agent in range(i+1, self.num_of_agents): # other agent
        for cell in path:
            #vertex contrains
            constraints.append({'agent': future_agent, 'loc': [cell],
'timestep': t, 'positive': False})

            #edge constrains
            if t<len(path)-1:
                constraints.append({'agent': future_agent, 'loc':
[path[t+1],cell], 'timestep': t+1,'positive': False})
            t = t + 1 #one step

        constraints.append({'agent': future_agent, 'loc':(len(path)-
1,[path[-1]]),'timestep': INF, 'positive': False})
        t = 0 # for the next agent
```

2.3 Optional: Adding Additional Constraints

Here we added constraints that solve the problem that agent can move on top of other agents that have already reached their goal locations.

```
constraints.append({'agent': future_agent, 'loc':(len(path)-1,[path[-
1]]),'timestep': INF, 'positive': False})
```

INF marked final location and use it when checking constraint with `is_constraints`.

```
def is_constrained(curr_loc, next_loc, next_time, constraint_table):
    ##############################
    # todo Task 1.2/1.3: Check if a move from curr_loc to next_loc at time step next_time violates
    #             any given constraint. For efficiency the constraints are indexed in a constraint_table
    #             by time step, see build_constraint_table.
     if next_time not in constraint_table: # no constraint at the timestep
         if INF in constraint_table: # someone on goal - inf constraint
             # print(constraint_table[INF])
             # print((((next_time, [next_loc]), False)))
             for con in constraint_table[INF]:
                 if con[0][0] <= next_time and con[0][1] == [next_loc] and
con [1]==False:
                     return True

         else: # next time no in constraints table and there is no INF constraint
             return False


     else:
         return is_negative_constraint(curr_loc, next_loc, next_time,
constraint_table) or \
                is_positive_constraint(curr_loc, next_loc, next_time,
constraint_table)
```

where `is_negative_constraint` and `is_positive_constraint` are:

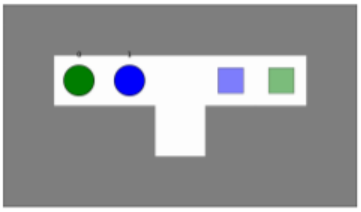
```
def is_negative_constraint(curr_loc, next_loc, next_time,
constraint_table):
    return (([next_loc], False) in constraint_table[next_time]) or \
           (([curr_loc, next_loc], False) in constraint_table[next_time])


def is_positive_constraint(curr_loc, next_loc, next_time,
constraint_table):
    positive_in_timestep = list(filter(lambda con: con[1] is True,
constraint_table[next_time]))
    if len(positive_in_timestep) > 0:
        return ([next_loc], True) not in positive_in_timestep and
([curr_loc, next_loc], True) not in positive_in_timestep
```

2.4 Optional: Addressing Failures

In class we discussed a tight runtime bound as a function of previous paths (in homework 2 I use it) and map's size. To keep it simple I change it to upper bound $\mathcal{O}(4 \cdot M \cdot N)$ to avoid another input of previous paths, here N and M are the map's dimensions.

2.5 Optional: Showing that Prioritized Planning is Incomplete and Suboptimal

In this question I had to show a case that illustrate that prioritized planning is not complete and suboptimal. I picked the first task (Design a MAPF instance for which prioritized planning does not find an (optimal or suboptimal) collision-free solution for a given ordering of the agents), which means showing prioritized planning is not complete. To keep it simply let use " instances/exp2_3.txt".



Here, assuming that agent 1 has the highest priority, block agent 0 to get to his goal at location (1,5), but as we know there is a solution as discussed before. We can generalize that for any case that higher priority agents are finish paths and blocking all paths to the goal of some other agent.

3 Task 3: Implementing Conflict-Based Search (CBS)

3.1 Detecting Collisions

The implemented functions `detect_collision` get as an input 2 paths, and first found which path's time is bigger (T_{max}) and which is smaller (T_{min}). Then, compare every pairs of location in a loop from 1 to T_{max} to detect vertex collision and under the condition that current time step is smaller than $T_{min} - 1$ edge collision are detected (switching positions).

In addition, `detect_collisions` is implemented, which is simply 2 loop on agent number index (i) and other future index (j, starts at i+1) , and use the `detect_collision` function to find the first collision if any.

```
def detect_collision(path1, path2):
    ##############################
    # TODO Task 3.1: Return the first collision that occurs between two robot paths (or None if there is no collision)
    #         There are two types of collisions: vertex collision and edge collision.
    #         A vertex collision occurs if both robots occupy the same location at the same timestep
    #         An edge collision occurs if the robots swap their location at the same timestep.
    #         You should use "get_location(path, t)" to get the location of a robot at time t.
    Tmin = min(len(path1),len(path2))
    Tmax = max(len(path1),len(path2))
    for t in range(Tmax):
        # when t is out of path range the get_location bring the final position
        L1 = get_location(path1, t)
        L2 = get_location(path2, t)
        # vertex:
        if L1 == L2:
            return {'loc': [L1], 'timestep':t}

        # edge:
        if t < Tmin - 1:
            next1 = get_location(path1, t + 1)
            next2 = get_location(path2, t + 1)
            if L1 == next2 and L2 == next1: #switch positions, edge collision
                # PAY ATTENTION! the output is according to path1!
                return {'loc': [L1, next1], 'timestep': t + 1}

    return None     # no collisions


def detect_collisions(paths):
    ##############################
    # TODO Task 3.1: Return a list of first collisions between all robot pairs.
    #         A collision can be represented as dictionary that contains the id of the two robots, the vertex or edge
    #         causing the collision, and the timestep at which the collision occurred.
    #         You should use your detect_collision function to find a collision between two robots.

    collisions = []
    for i in range(len(paths)):
        for j in range(i + 1, len(paths)): #next agents
            first_coll = detect_collision(paths[i],paths[j])
            if first_coll is not None:
                collisions.append(
                    {'a1': i, 'a2': j, 'loc': first_coll['loc'],
'timestep': first_coll['timestep']})

    return collisions
```

3.2 Converting Collisions to Constraints

The `standard_splitting` function implementation, which simply returns a list of 2 constraints to resolve the given collision. So, this function is nothing more than one line:

```
def standard_splitting(collision):
    ##############################
    # todo Task 3.2: Return a list of (two) constraints to resolve the given collision
    #          Vertex collision: the first constraint prevents the first agent to be at the specified location at the
    #                            specified timestep, and the second constraint prevents the second agent to be at the
    #                            specified location at the specified timestep.
    #          Edge collision: the first constraint prevents the first agent to traverse the specified edge at the
    #                          specified timestep, and the second constraint prevents the second agent to traverse the
    #                          specified edge at the specified timestep

    return [{'agent': collision['a1'], 'loc': collision['loc'], 'timestep':
collision['timestep'], 'positive': False},
            {'agent': collision['a2'], 'loc': collision['loc'][::-1],
'timestep': collision['timestep'], 'positive': False}]
```

notice that in case of edge collision the constraint of agent 2 needs to be at the opposite direction (detection notation is according to agent 1), and then

`collision['loc'][::-1],`

will solve that problem and will not change anything for vertex collision (because vertex collision is one location).

3.3 Implementing the High-Level Search

Here, we simply implement the pseudo-code attached in page 7. You can see the notes added to the code that refer to the line in page 7. Important technical issue was to make a shallow copy of property while generation a child.

Basically, as long as the open list in not empty, the function pops the next node, and find solution if the node has no collision. Otherwise, the function chooses the first collision and convert to list of constraints (using `standard_splitting` function, next we will use the disjoint splitting) and add new child to the open list for each constraint. (high-level implementation will add in next section)

3.4 Testing your Implementation

Here we tested our code using the command (running all test instances):

```
python run_experiments.py --instance "instances/test_*" --solver CBS --batch
```

we compare our results to the file attached (optimal results) and check our implementation.

4 Task 4: Implementing CBS with Disjoint Splitting

4.1 Supporting Positive Constraints

Here we added a binary key "positive" to the `single_agent_planner.py` for any constraint dictionary. In addition, to support this change, few changes in the code are added.

4.2 Converting Collisions to Constraints

`disjoint_splitting` function is implemented. First this is to get a random integer 0 or one which define agent ('a1' or 'a2'), and then return list of 2 constraint: one positive and one negative to the random agent (if 'a2' is chosen, we take the collision in the opposite direction using [::-1] as we use at (3.2)).

```
def disjoint_splitting(collision):
    ############################
    # todo Task 4.1: Return a list of (two) constraints to resolve the given collision
    #          Vertex collision: the first constraint enforces one agent to be at the specified location at the
    #                            specified timestep, and the second constraint prevents the same agent to be at the
    #                            same location at the timestep.
    #          Edge collision: the first constraint enforces one agent to traverse the specified edge at the
    #                          specified timestep, and the second constraint prevents the same agent to traverse the
    #                          specified edge at the specified timestep
    #          Choose the agent randomly

    rand01 = random.randint(0, 1)

    if rand01 is 0:
        return [
            {'agent': collision['a1'], 'loc': collision['loc'], 'timestep':
collision['timestep'], 'positive': True},
            {'agent': collision['a1'], 'loc': collision['loc'], 'timestep':
collision['timestep'], 'positive': False}]
    else:
        return [
            {'agent': collision['a2'], 'loc': collision['loc'][::-1],
'timestep': collision['timestep'], 'positive': True},
            {'agent': collision['a2'], 'loc': collision['loc'][::-1],
'timestep': collision['timestep'], 'positive': False}]
```

4.3 Adjusting the High-Level Search

Here we needed to update the function `find_solution` in `cbs.py`. First, line to detect the required splitting method (from command line, see command line example below). The function needs to support positive constraints, I choose to convert any positive constraint to N-1 negative constraints (N is number of agents, for any agent except the positive constraint's agent), because solving negative constraint is already works from previous sections. In addition, the low level of CBS with disjoint splitting might have to find new shortest paths for agents which violate

the constraint added. To solve it, function **`paths_violate_constraint`** is added to find list of agents that violate a given constraints table (with the new constraints that added from the positive constraint).

```
def find_solution(self, disjoint=True):

    self.start_time = timer.time()

    root = {'cost': 0, # Line 4 initializte
            'constraints': [], # Line 1
            'paths': [], # Line 2 initializte
            'collisions': []} #Line 3 initializte

    # Line 2
    for i in range(self.num_of_agents):  # Find initial path for each agent
        path = a_star(self.my_map, self.starts[i], self.goals[i], self.heuristics[i], i,
root['constraints'])#, goal_constrains, max_path_lengths)

        if path is None: # if any agent can go to his goal without constarints than there is no solution...
            raise BaseException('No solutions')

        root['paths'].append(path) # add paths of any agent

    root['cost'] = get_sum_of_cost(root['paths']) #Line 4
    root['collisions'] = detect_collisions(root['paths']) #Line 3
    self.push_node(root) # Line 5


    ############################
    # todo Task 3.3: High-Level Search
    #          Repeat the following as long as the open list is not empty:
    #            1. Get the next node from the open list (you can use self.pop_node())
    #            2. If this node has no collision, return solution
    #            3. Otherwise, choose the first collision and convert to a list of constraints (using your
    #               standard_splitting function). Add a new child node to your open list for each constraint
    #          Ensure to create a copy of any objects that your child nodes might inherit

    while len(self.open_list) > 0:
        curr_node = self.pop_node() # Line 7

        if (curr_node['collisions'] == []): # Line 8
            self.print_results(curr_node, disjoint)
            return curr_node['paths'] #Line 9

        collision = curr_node['collisions'][0] #Line 10

        # Line 11
        if disjoint is True:    #one of the features, use the command line
            constraints = disjoint_splitting(collision)
        else:                   # standard splitting
            constraints = standard_splitting(collision)

        for constraint in constraints:
            # Line 13
            new_node = {'cost': 0,
                 'constraints': curr_node['constraints'].copy() + [constraint], #line 14
                'paths': curr_node['paths'].copy(), #line 15
                'collisions': []}

            a_i = constraint["agent"] #line 16

            path = a_star(self.my_map, self.starts[a_i], self.goals[a_i],
self.heuristics[a_i], a_i, new_node['constraints'])  # line 17


            if path is not None and len(path) > 0:

                #######
                new_node["paths"][a_i] = path # Line 19

                no_solution = False
```

```
    # adding negative constraints that implements the positive constraint
            if disjoint is True and constraint['positive'] is True:
                new_constraints = [{'agent': a, 'loc': constraint['loc'][::-1],'timestep':
collision['timestep'], 'positive': False} for a in range(len(new_node['paths'])) if a is not
a_i]

                new_node['constraints'] = new_node['constraints'] + new_constraints

                # find the agents who violate the new constraints list
                violated_agents =
                paths_violate_constraint(new_node['paths'], constraint)

        # for every agent who violate - find a new path using A* function
                for vio in violated_agents:
                    new_path = a_star(self.my_map, self.starts[vio], self.goals[vio],
                                      self.heuristics[vio], vio, new_node['constraints'])

    # maybe some agent have no solution... raise a flag and break the loop
                    if new_path is None or len(new_path) is 0:
                        no_solution = True
                        break

                    else: # new paths for all violated agent found
                        new_node['paths'][vio] = new_path.copy()

            if no_solution:
  # one of the violated agents doesn't have a solution with the new positive constraint
                # do not add child
                continue
            new_node["collisions"] = detect_collisions(new_node["paths"]) # line 20
            new_node["cost"] = get_sum_of_cost(new_node["paths"])#line 21
            self.push_node(new_node) #line 22

    raise BaseException('No solutions') #return "No Solutions", line 23
```

We know that changing the splitting method does not affect the optimality of CBS algorithm (my code proves it). Hence, running this test will solve the instances at the same cost as in the ground truth CSV file attached no matter choosing splitting method. But, by using disjoint method, runtime decrease dramatically.

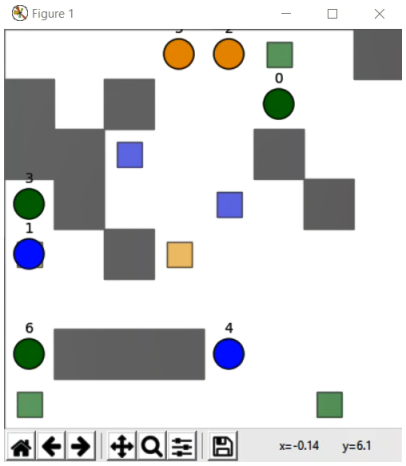
Important

To check our implementation of CBS with disjoint splitting the command line is (running all test instances):

python run_experiments.py --instance "instances/test_*" --disjoint --solver CBS --batch

how much disjoint splitting helps?

In test 47 (see attached image below), I found a significant improvement between standard splitting to disjoint splitting. This maps constraint tree become big because there is a lot of collision while searching fir solution, and disjoint splitting solution has better performance in all the parameter more than 97% (see table below)



	Sum of cost	CPU time	Expanded nodes	Generated nodes
Standard	65	419.47	197429	394857
Disjoint	65	11.19	3638	6099
Improvement	0 (of course, both optimal)	97.3%	98.2%	98.5%

5 Future work

Prioritized search: the algorithm is only searching with the given permutation, and therefore, sometimes it fails to find a solution or find high-cost solution.
If we let the prioritized search use another random permutation (or every possible permutation) we could get a solution with lower cost, and in some cases find any solution (instead of no one). In the other hand, the runtime will increase significantly.

CBS: the algorithm detects the first new collision in any loop. Sometime solving that collision does not solve further collision, and sometime choosing other collision can find better paths (prevent more collision).
As the same as choosing random agent to add constraint, I think that choose random collision as constraint also could help to find better solution (runtime, and expanded nodes), especially at scenario when choosing the first one does not prevent any other.