Introduction to Artificial Intelligence: Final Project Report

Github repository link:
https://github.com/nctu16028/NYCU-2021Spring-Intro_to_AI-final_project

Introduction

Traveling salesman problem is a well-known NP hard problem. It's important to solve it because it's widely applied on real-world problem, for example, for logistic delivery it's important to select a min distance route to minimize the cost.
Since TSP has been proved as a NP hard problem, it has no efficient way to find optimal solution, therefore we decided to find a sub-optimal solution using reinforcement learning approaches.

Related work

1. Dynamic programming

```
def DP_TSP(adj_mat):
  global cnt2
  n = len(adj_mat)
  all_points_set = set(range(n))

  # dp keys: tuple(visted_set, last_point_in_path)
  # dp values: tuple(cost, predecessor)
  dp = {(tuple([0,i]), i): tuple([adj_mat[0][i], 0]) for i in range(1,n)}
  dp[tuple([0]),0]=(0,None)
  queue = [(tuple([0,i]), i) for i in range(1,n)]
  cnt2+=n

  while queue:
    cnt2+=1
    visited, last_node = queue.pop(0)
    prev_dist, predecessor = dp[(visited, last_node)]
    to_visit = all_points_set.difference(set(visited))

    if len(to_visit)==0:
        #back to depot (predecessor no change for retrace)
        dp[(visited,last_node)] = (prev_dist+adj_mat[last_node][0], predecessor)
        continue

    for successor in to_visit:
      new_visited = tuple(sorted(list(visited) + [successor]))
      new_dist = (prev_dist + adj_mat[last_node][successor])

      if (new_visited, successor) not in dp:
        dp[(new_visited, successor)] = (new_dist, last_node)
        queue += [(new_visited, successor)]
      else:
        if new_dist < dp[(new_visited, successor)][0]:
          dp[(new_visited, successor)] = (new_dist, last_node)

  optimal_path, optimal_cost = get_rel(dp, n)
  return optimal_path, optimal_cost
```

We use dynamic programming to improve brute-force by utilizing optimal sub-structures, but it's still an exponential time algorithm.
In this implementation my dp dictionary stored the min cost of visiting all nodes in a given set from depot (the staring node) and a predecessor node to help re-trace the route.

After all the sub-problem were solved, we can find the optimal path by comparing the total cost and choose the min one, which is

min(
    min cost (start from depot visit all the nodes in node set and stop at node i)
    + distance from node i to depot
)
for every node i in the node set

we have to compute all the sub-problems to get to the final result, therefore it's computational inefficiency when problem size grows.
In my test its limit is about 20 nodes.

2. Branch-and -bound

(1) Heuristic

```
def heuristic(adj_mat, i):
    first = second = float('inf')
    n = len(adj_mat)
    for j in range(n):
        if adj_mat[i][j] < first:
            second = first
            first = adj_mat[i][j]
        elif(adj_mat[i][j] < second):
            second = adj_mat[i][j]
    return (first+second)/2
```

We use heuristic function to compute the lower bound of a given path, which is the best possible traveling route cost.
To ensure admissible heuristic, we consider a traveling route, every node in the route has an in-edge and an out-edge, we chose the least and second least edge from its

adjacent edges to be the lower bound of in-edge and out-edge, therefore we can get the lower bound of a traveling route.

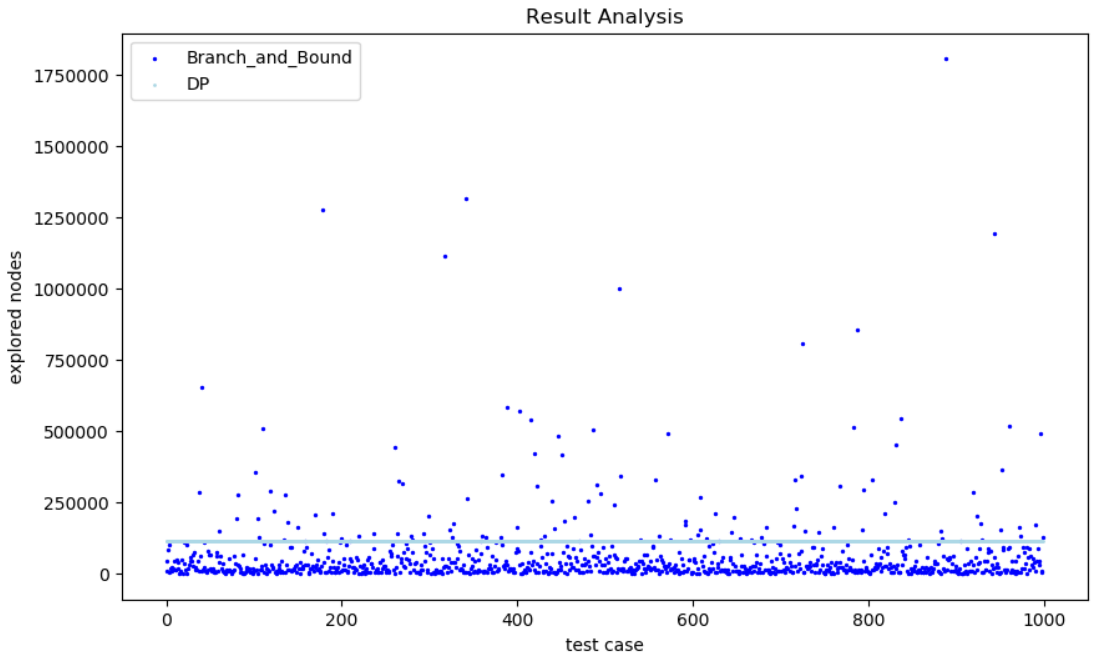
(2) Implementation

```
def BranchAndBound_TSP(adj_mat, src=0):
    global cnt1
    optimal_tour = []
    n = len(adj_mat)
    u = Node()
    PQ = PriorityQueue()
    v = Node(level=0, path=[0], bound=0)
    min_length = float('inf')
    # Compute initial lower bound
    for i in range(n):
        v.bound+=heuristic(adj_mat,i)
    cnt1+=1
    PQ.put(v)
    while not PQ.empty():
        cnt1+=1
        v = PQ.get()
        if v.bound < min_length:
            u.level = v.level + 1
            last = v.path[-1]
            for i in filter(lambda x: x not in v.path, range(1, n)):
                u.path = v.path[:]
                u.path.append(i)

                if u.level == n - 2:
                    l = set(range(1, n)) - set(u.path)
                    u.path.append(list(l)[0])
                    # putting the depot at last
                    u.path.append(0)
                    _len = get_cost(adj_mat, u)
                    if _len < min_length:
                        min_length = _len
                        optimal_tour = u.path[:]

                else:
                    #compute lower bound of successor node u and add it into PQ if it's decent
                    u.bound = v.bound - heuristic(adj_mat,last) + adj_mat[last][i]
                    if u.bound < min_length:
                        PQ.put(u)

                u = Node(level=u.level)

    return optimal_tour, min_length
```

This method is ‘smarter’ we compute heuristic lower bound for every possible route. We explore the most likely node in state space tree each time, like what we do in A* search. Every time we reach a leaf node of state space tree, we get a possible solution and update the optimal route if its cost is smaller than current min.
Noted that, we prune the unfinished nodes if its heuristic lower bound is greater than optimal cost so far. This can avoid exploring a lot of impossible nodes, but the method will sometimes degenerate to brute-force approach if the impossible nodes cannot be prune early.

3. Comparison

We ran both DP and branch and bound approach on random 15 vertices test case for 1000 times and below is the chart of number of state space tree nodes explored during the search for both approaches.



The experiment shows DP having fixed explored nodes in every test case, since it has to solve every sub-problem. For most of the test cases, branch-and-bound does better than DP, but sometimes much worse than DP. It depends on how good the pruning is.

4. Problem:

Even though branch-and-bound method works better than DP in average cases, it is struggle to solve big-scale application problems for instance, a logistic routing problem that has more than 100 customers nodes.

It's hard to apply real world problems and for many cases we only need a sub-optimal solution, so we decided to find a sub-optimal solution using reinforcement learning approaches.

Methodology

Reinforcement learning consists of lots of types. Here we focus on a basic form, Q-learning, in which we maintain a table that records a Q-value for each state taking each action. During the process, we try a path that goes through all the nodes and update the Q-table for many iterations. We define a state ***s*** to be a set of all visited nodes during each iteration. Then, an action ***a*** is the next node to go to. After an action is applied, the new node is appended to the list of visited nodes. If we reach the state that all the nodes have been searched, an iteration is finished, which means the whole Q-table has been updated once.

```python
class TSPsolver:
    def __init__(self, graph, dists):
        self.graph = graph  # represented by a list of nodes
        self.edges = dists  # costs of edges
        self.visited = [graph[0]]   # A "state" is represented by a list of visited nodes
        self.epsilon = 1    # the probability of "exploration" rather than "exploitation"
        self.alpha = 0.9    # the learning rate
        self.gamma = 0.9    # the discount factor
        self.q_values = [[0 for i in range(len(graph) + 1)] for j in range(2 ** (len(graph)) + 1)]
```

```python
if __name__ == '__main__':
    dists = genRandomGraph(15)
    testGraph = list(range(len(dists)))
    print("Starting Graph:", testGraph)

    bestSolution = []
    shortest = float('inf')
    myAgent = TSPsolver(testGraph, dists)
    num_trials = 50
    for i in range(num_trials):
        nextAction = True
        myAgent.visited = [testGraph[0]]
        myAgent.epsilon = 1 / math.sqrt(i + 1)
        while(nextAction != []):
            nextAction = myAgent.getAction()
            if(nextAction != []):
                myAgent.update(nextAction)

        newCost = utils.pathCost(myAgent.visited, myAgent.edges)
        if newCost < shortest:
            shortest = newCost
            bestSolution = myAgent.visited
            bestTrial = i + 1

        if (i+1) % (num_trials / 10) == 0:
            print("After", i+1, "trials:", myAgent.visited, newCost)

    print("Best solution at trial", bestTrial, ":", bestSolution, shortest)
```

According to the Q-learning algorithm, the Q-table is updated with the formula:

$$Q^*(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s_{next}, a') - Q(s,a))$$

, where ***r*** is the reward the agent gets after taking action ***a*** from state ***s***, γ is the discount factor, and α plays the role of learning rate. In my implementation, the reward is defined as the negation of distance the agent takes, since we want to minimize the total distance. It is like the concept of "punishment", and the goal of the agent is to avoid more punishment. By the way, the part $r + \gamma \max_{a'} Q(s_{next}, a')$ is called "target Q-value", which is the optimal value the agent can get after taking

action ***a*** from state ***s***, and $Q(s,a)$ is the predicted value, so the part inside the parentheses is analogous to the concept of "gradient" in other kinds of machine learning algorithms.

```
def update(self, action):
    '''Optimizing Q-table'''
    alpha = self.alpha
    nextState = self.getNextState(action)
    reward = 0 - self.edges[self.visited[-1]][action]
    target_Q = reward + self.gamma * self.getValue(nextState)
    model_Q = self.getQvalue(self.visited, action)
    new_Q = model_Q + alpha * (target_Q - model_Q)
    self.setQvalue(self.visited, action, new_Q)
    self.visited = nextState
```

In reinforcement learning, there is a technique called ε-greedy, in which a threshold ε is maintained. The threshold tells the agent whether to "exploit" or to "explore". Initially, it is set to 1, and may be tuned down after iterations. Every time the agent is going to choose an action, it chooses a random number between 0 and 1. If the number is smaller than ε, the agent does the exploration, that is to choose an action randomly; otherwise, the agent does the exploitation, that is to follow the Q-table to determine a best policy. Imagine that the agent always exploits the Q-table, then it may be stuck in some locally optimal solutions. Therefore, the existence of ε is to avoid such situations.

```
def getAction(self):
    '''Choose a possible next node'''
    rand = random.random()
    possibilities = utils.possibleNextNodes(self.graph, self.visited)
    if possibilities == []:
        # No possible new nodes
        return []
    else:
        if (rand > self.epsilon):
            # Taking best policy (exploitation)
            return self.getPolicy(self.visited)
        else:
            # Taking random (exploration)
            return random.choice(possibilities)
```

Experiments

15 nodes are generated randomly, and the distance of each pair is between 0 and 20. After 50 iterations, the results are shown in the following.

```python
def genRandomGraph(numPoints):
    matrix=[]
    for i in range(numPoints):
        row=[]
        for j in range(numPoints):
            if i==j: row.append(float('-inf'))
            elif i<j: row.append(random.uniform(0, 20))
            else: row.append(matrix[j][i])
        matrix.append(row)
    return matrix


if __name__ == '__main__':
    dists = genRandomGraph(15)
    testGraph = list(range(len(dists)))
    print("Starting Graph:", testGraph)

    bestSolution = []
    shortest = float('inf')
    myAgent = TSPsolver(testGraph, dists)
    num_trials = 50
    for i in range(num_trials):
```

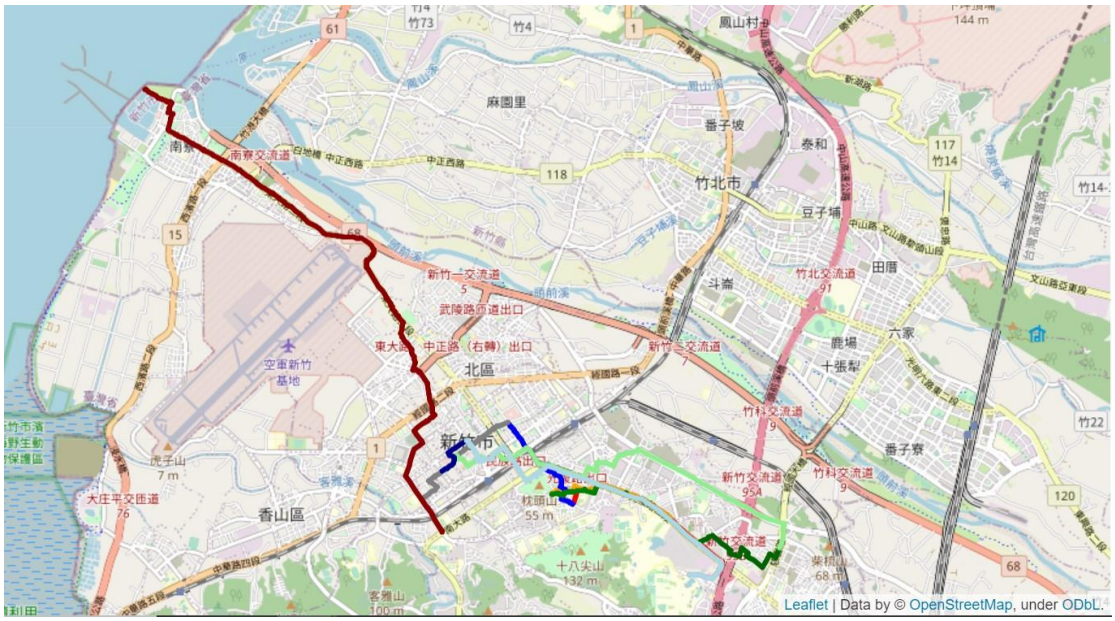
```
D:\course materials and exercises\Artificial Intelligence\project>python tsp.py
Starting Graph: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
After 5 trials: [0, 1, 2, 3, 4, 10, 7, 5, 6, 11, 8, 9, 12, 13, 14, 0] 149.54448244862425
After 10 trials: [0, 1, 2, 3, 9, 4, 5, 11, 6, 7, 8, 10, 13, 14, 12, 0] 132.6394103745989
After 15 trials: [0, 6, 1, 9, 2, 12, 3, 4, 5, 7, 8, 10, 11, 13, 14, 0] 115.40167136351425
After 20 trials: [0, 1, 2, 11, 3, 4, 10, 5, 6, 7, 14, 8, 9, 12, 13, 0] 178.6571646842166
After 25 trials: [0, 1, 7, 2, 3, 4, 12, 5, 6, 8, 9, 10, 11, 13, 14, 0] 103.66362701561437
After 30 trials: [0, 1, 2, 3, 6, 4, 5, 7, 8, 9, 14, 10, 11, 12, 13, 0] 111.92329316907951
After 35 trials: [0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 9, 10, 11, 13, 14, 0] 143.5831461631192
After 40 trials: [0, 12, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 0] 118.6295370479523
After 45 trials: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 0] 129.7059860062962
After 50 trials: [0, 1, 2, 3, 4, 5, 6, 7, 13, 8, 9, 10, 11, 12, 14, 0] 127.92686339636434
Best solution at trial 25 : [0, 1, 7, 2, 3, 4, 12, 5, 6, 8, 9, 10, 11, 13, 14, 0] 103.66362701561437

D:\course materials and exercises\Artificial Intelligence\project>
```

In this result, the list is the path the agent chose, and the number is the total distance the agent travelled. The cost fluctuates with the number of trials, which is the effect of ε-greedy. As you can see, the best solution the agent gave is not that optimal compared to the two approaches mentioned above. I guess it is because the strength of RL algorithm is hard to show if the number of nodes is too low. When there are under 15 nodes, DP and branch-and-bound can still give the optimal solution in a decent time amount. However, if there are many nodes, like real world situations, then it is when RL comes in handy.

Besides, I also try to apply this to a more realistic example. In this experiment, I use

the OpenStreetMap data from HW2 to choose some locations and determine a path that goes through all of them.

```
edge_data = utils.getEdgeData()
nodes = list(edge_data.keys())
target_nodeID = {"交大光復":2270143902, "巨城百貨":1079387396, "市立動物園":426882161, "COSTCO":1737223506, "交大博愛":8263925314,
                 "大遠百":3951170837, "南寮漁港":8513026827, "新竹市政府":415647766, "馬偕醫院":1723037689, "清大":311784014, "清大南
testGraph = list(target_nodeID.values())
num_locs = len(testGraph)
```



Still, the result is under my estimation. I think one of the reasons is the same as the previous experiment. Another possible reason I can come up with is that the street distance data is not enough, so many segments would “collide” with each other, that is, they use the same roads.

Conclusion

TSP is a well-known NP hard problem, but it’s widely applied on real-world problem like logistic delivery. Instead of spending too much time looking for an optimal path, we prefer a solution that may not be perfect but good enough in a shorter time. In this way, reinforcement learning is a great choice, since the RL agent can give a sub-optimal solution in a decent amount of time relatively. Also, under this scheme, the agent “learns while exploring”, which is more elastic compared to other learning algorithms, such as supervised learning.