Peter Subacz – CS534: Artificial Intelligence – Homework 2 – 9/26/19

Contents

Chapter 3, Ex 3.13 2
Chapter 3, Ex 3.16 4
Chapter 3, Ex. 3.26 7
Problem 4: Chapter 4, Ex. 4.3 9
Problem 5: Chapter 4, Ex. 4.5 12
Python Code 14
Problem 4A 14
Problem 4B 21

Problem 1: Chapter 3, Ex 3.13

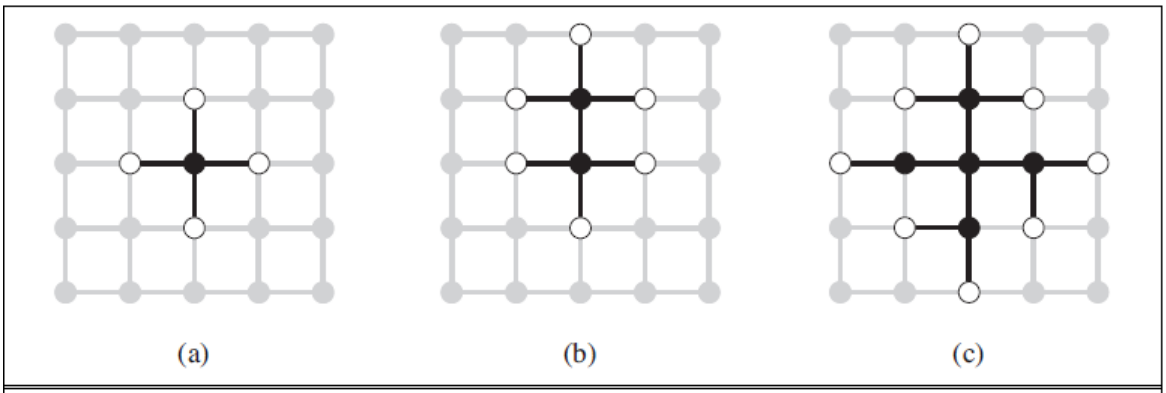


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Prove that GRAPH-SEARCH satisfies the graph separation property illustrated in Figure 3.9. (Hint: Begin by showing that the property holds at the start, then show that if it holds before an iteration of the algorithm, it holds afterwards.) Describe a search algorithm that violates the property.

As states on page 77, the GRAPH-SEACH augments the TREE-SEARCH algorithm by including Open list (Frontier set) and Closed List (Explored Set). GRAPH-SEACH can be defined using the pseudocode:

```
function GRAPH-SEARCH( problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

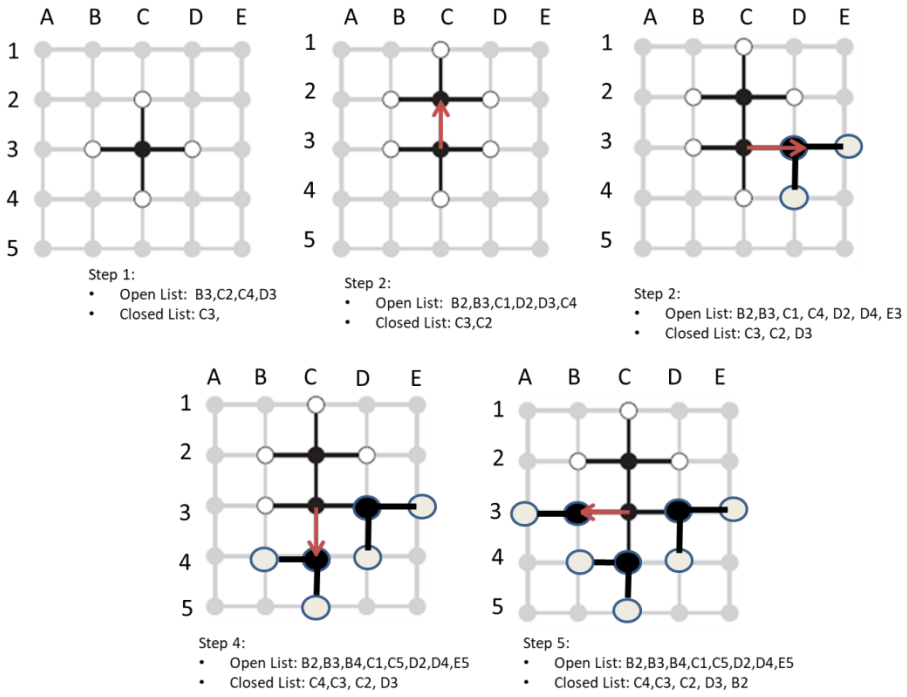
As states in the pseudo code, during a GRAPH-SEACH initialization the frontier node is populated with an initial condition and the explored set is initialized to empty. The open list contains a list of node that whose space state has not been previously marked explored, while the closed list holds nodes that are marked explored. The looping portion of GRAPH-SEARCH check to see if the frontier list is empty and if not chooses a leaf node based on the search strategy and checks for a goal state. If the goal states was not found. The node is expanded and moved in the explored list. Newly connected nodes are nodes are checked to see if they already exist in the frontier or the explored list and, if not are added to the frontier list. The search algorithm repeats until a solution is found or until node exhaustion (returns search failure.

Keeping the list of open and closed nodes allows the algorithm to remember which nodes have been explored and avoids loopy paths. This is the core of the separation property being shown in Figure 3.9. The open list contains a set of nodes that are always unexplored and the closed list contains a set of nodes that have been explored. Always picking from the open nodes list ensures that the node and generated

child nodes have not been explored. The figure below shoes the search tree implementing a GRAPH-SEACH algorithm. Each step of the algorithm expends a single node and moved the expanded node to the closed list. Using the search strategy specified in figure 2.9 and the GRAPH-SEARCH methodology defined above:

- The 1st step of the algorithm is initialized in the center node C3 and is expanded. The node C# is added is the closed list and nodes B3, C2, C4, D3 are added to the open list.
- In the 2nd step, the node C2 is expanded and moved to the closed list. Since the closed set only contains nodes C2 and C3, The nodes B2 C1, D2 are added to the open set.
- In the 3rd step of the algorithm explores node D3 from the parent node C3. Nodes E3 and D4 are added to the frontier list. Node D2 is already in the frontier list with a parent node pointing to node C2, therefore, D2 is not added to the frontier list again.
- The 4th step explores node C4 and adds nodes C5 and B4 to the frontier list. Node D4 already exists with a parent node pointing to D3 and is not added to the frontier list.

This process repeats until n the frontier list is empty and has been illustrated in the figure below



A search algorithm that violates this property would be the TREE-SEARCH. As nodes are explored and generated, all new child nodes are added to the frontier list. This causes the each node to be repeated and violates the separation principle.

Problem 2: Chapter 3, Ex 3.16

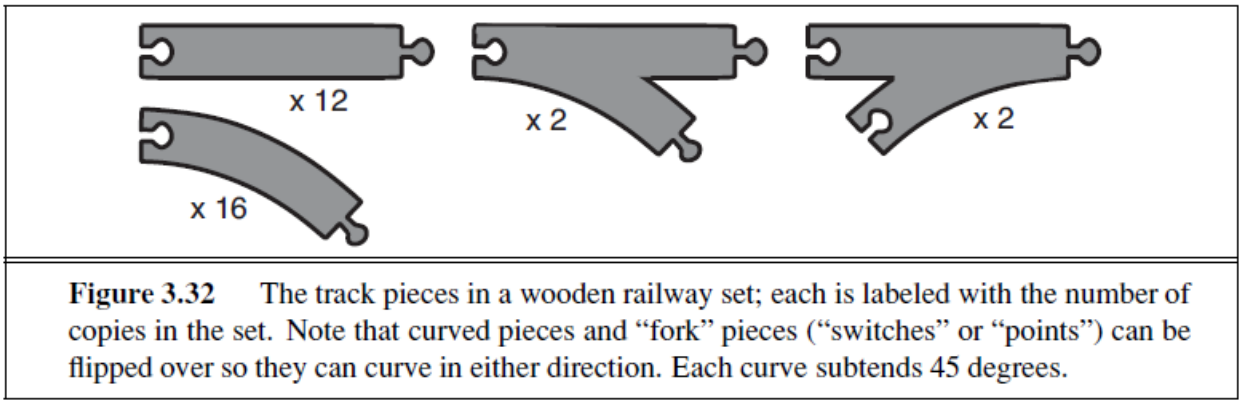


Figure 3.32 The track pieces in a wooden railway set; each is labeled with the number of copies in the set. Note that curved pieces and "fork" pieces ("switches" or "points") can be flipped over so they can curve in either direction. Each curve subtends 45 degrees.

A basic wooden railway set contains the pieces shown in Figure 3.32. The task is to connect these pieces into a railway that has no overlapping tracks and no loose ends where a train could run off onto the floor.

A. Suppose that the pieces fit together exactly with no slack. Give a precise formulation of the task as a search problem.

Each track has at least 1 peg and 1 hole that allow for tracks to be interconnected. Forked tracks have

Track	Holes	Pegs
Straight	1	1
Curved	1	1
Fork 1	1	2
Fork 2	2	1

- States: Any arrangement of the 32 train track piece. The nodes of the graph contain the following states: Track attributes*, list of expanded tracks (expanded list)**, and tracks left to place (frontier list).
- Initial state: 1 of the 32 train tracks have been placed on the board
- Actions:
  - Place train track – Connect a track to an existing peg or hole (forks and curved pieces can be flipped)
- Transition model: Adds new frontier nodes and marks the current node as expanded list.
- Goal test: Check for rail connectivity, check for no overlaying paths, check for track continuity
- Path cost: 1 per action

The total depth of the tree is 32 as each placement action uses a track and removes it from the list. Therefore the as algorithm moves deeper into the tree there will be (32-n) pieces left to be placed at each level, At the 32 second level, there are 0 pieces left.

The branching factor is calculated by: $r = 1 + \left(Pegs_{Open} + Holes_{Open}\right)$

* Track attributes include: type of track, orientation of track, parent track (initial track is null). This information is generated when new frontier nodes are connected to the graph.

** The expanded track list is held in the frontiers for each possible state.

B. Identify a suitable uninformed search algorithm for this task and explain your choice.

If the loopy paths of the graph are pruned and each that can be flipped is flipped, the number of unique actions is defined by 12 straight path pieces, $16 * 2 = 32$ curved path pieces, and $4 * 2 = 8$ forked path pieces. The totals number of possible state generated for each action is 52. If all 4 forked tracks are initially used the $(Pegs_{Open} + Holes_{Open}) = 3 + 3 = 6$ branches at the 4th level of the graph.

The tree depth is therefore has a maximum tree depth is 52 with a worst case branching factor of 6. The shallowest solution exists at depth of 32. Therefore using the figure 3.21 as a guide where *branching* = 6, *depth of shallowest solution* = *l* = 32, *maximum depth of tree* = 54, therefore $b^d = 6^{32} = 8 * 10^{24}$ and $b^m = 6^{52} = 3 * 10^{40}$.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

For each algorithm the Time and space complexity is given in the table below:

Criterion	Breadth First	Uniform Cost*	Depth First	Depth Limited	Iterative Deeping**	Bidirectional
Time	$8 * 10^{24}$	$8 * 10^{24}$	$3 * 10^{40}$	$8 * 10^{24}$	$8 * 10^{24}$	$4 * 10^{24}$
Space	$8 * 10^{24}$	$8 * 10^{24}$	312	192	192	$4 * 10^{24}$

Depth limited search would be the uninformed algorithm of choice to limit the search to the for the solution. DLS has the smallest time and space requirement for a single direction search. The bidirectional searching has a lower time complexity by a higher space complexity.

*Step cost is assumed equal for all placement steps there as per page 83 uniform cost = breadth first

** Shallowest possible solution it at 32 and therefore Depth Limited = Iterative Deeping to find the shallowest possible solution.

C. Explain why removing any one of the "fork" pieces makes the problem unsolvable. Give an upper bound on the total size of the state space defined by your formulation. (Hint: think about the

maximum branching factor for the construction process and the maximum depth, ignoring the problem of overlapping pieces and loose ends. Begin by pretending that every piece is unique.)

As stated above, the branching factor is calculated by:$b = \left(Pegs_{Open} + Holes_{Open}\right)$. If a single forked piece is removed the new branching factor becomes $b = \left(Pegs_{Open} + Holes_{Open}\right) = 5$. No matter which forked piece is removed. An inherent assumption of the problem is that ration of pegs to holes is required to be 1 in order to close all possible loops. By deduction, removing one of the forked pieces reduces this ratio as there is more peg or holes depending on the piece removed. Completing a track loop reduces the branching factor by 2 for each loop complete. Therefore, with 1 loop complete the branching factor becomes 3 and with 2 loops complete the branching factor is 1. There are no open holes or pegs for the last track to connect to and the problem becomes unsolvable.

Problem 3: Chapter 3, Ex. 3.26

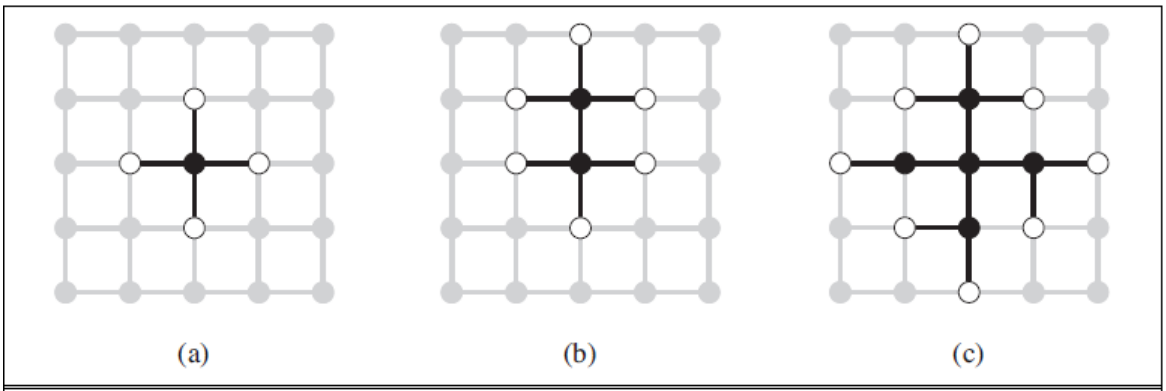


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Consider the unbounded version of the regular 2D grid shown in Figure 3.9. The start state is at the origin, (0, 0), and the goal state is at (x, y).

A. *What is the branching factor b in this state space?*

The branching factor is 4 as an agent can only move 4 directions per action

B. *How many distinct states are there at depth k (for k > 0)?*

Each movement of causes the number for nodes in the graph to expand by 4n

- At level 1 there are 1+4 = 5 states
- At level 2 there are 1+4+8 =13 states
- At level 3 there are 1+4+12 = 25 states

Therefore there are at depth K there are $2k^2 + 2k + 1$

C. *What is the maximum number of nodes expanded by breadth-first tree search?*

If all the nodes are expanded on the 3rd level there are 16 nodes and on the 3rd level there are 64 nodes. Therefore the nodes expand maximum possible search rate at depths n is 4^n. The answer lies at position (x,y) which has a L1 norm of |x| +|y|. Therefore the maximum possible nodes are at $4^{(|x|+|y|)}$

D. *What is the maximum number of nodes expanded by breadth-first graph search?*

The graph search uses a expanded list and a frontier list. Only nodes in the frontier list are expanded. Therefore BFS graph search only needs to expand at a rate of $2k^2 + 2k + 1$ where K is the level of the L1 norm.

E. *Is $h = |u - x| + |v - y|$ an admissible heuristic for a state at (u, v)? Explain.*

As per page 94, an admissible heuristic never over estimates the cost to reach the goal. This function minimizes the cost to the goal by the position moved.

F. How many nodes are expanded by A∗ graph search using h?

A^* only expends |x| +|y| because A* is selecting nodes using the L1 norm and the space state is 2D grid.

G. Does h remain admissible if some links are removed?

Yes, Removing link can make the best past to the solution longer therefore H is will still underestimate the cost to the goal

H. Does h remain admissible if some links are added between nonadjacent states?

No, Adding adjacent links to nodes would cause H is overestimating the cost to the goal as the shortest path is no longer the L1 norm.

Problem 4: Chapter 4, Ex. 4.3

In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise 3.30.

Note: Assumed to be implemented in python. The problem is unclear on how or what problem these algorithms are to be implemented in. These problems are solved assuming that TSP map to be used is the Romanian map.

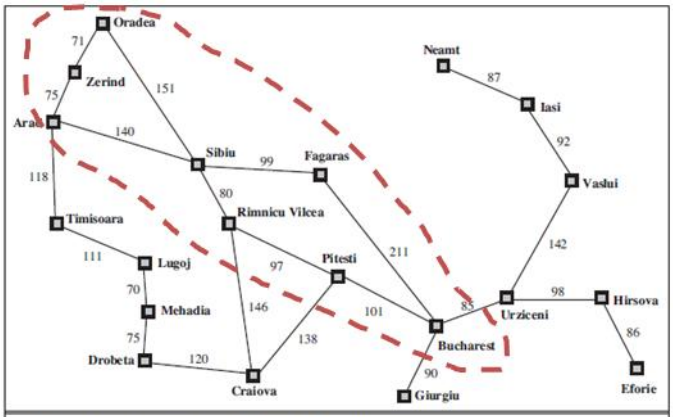


Figure 3.2 A simplified road map of part of Romania.

Exercise 3.30: The traveling salesperson problem (TSP) can be solved with the minimum-spanning tree (MST) heuristic, which estimates the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

These implementations use a trimmed map of Romania for testing as shown above in red. This reduces the possible states to search and speeds up the run time of the algorithm

a. Implement and test a hill-climbing method to solve TSPs.
The TSP problem states "What is the shortest possible route that visits each city and returns to the origin city?"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

See Python implementation attached. This implementation is a greed approach to solve the Romanian map where the agent climbs the hill with the lowest path cost. This can be seen somewhat as A* where the heuristic is 0 and therefore the algorithm is only using the cost function to pick the path forward. This agent implements by searching for nodes with the lowest travel cost and retained the past distance traveled for each node generated. In the original HILL-Climbing algorithm gets suck stuck in a local minimum by traversing between Oradea and

Zerind. Therefore, frontier nodes need to be generated that keep the past cost of the traveled with the future cost to generate a forward searching agent. The goal test for this algorithm is to see if the agent as visited all the cities and is at the origin. The city is presented in a graph format and the agent in unaware of the graph. As the agent explores the graph nodes are created and stored. The agent evaluates the nodes picks the node with the lowest cost. This presents the highest hill an agent can climb. The output of this agent when finished for a search of including the following cities {Oradea, Zerind, Sibiu, Fagaras, Bucharest, Pitesti, Rimnicu Vilcea}:

```
Iteration:  6784
Currently in City:  Oradea Distance traveled:  1001
ALL VISITED
AT ORIGIN
Jobs Done!
Travel Solution:  ['Oradea', 'Zerind', 'Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest',
'Fagaras', 'Sibiu', 'Oradea']
Travel Distance:  1001
```

b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larra˜naga et al. (1999) for some suggestions for representations.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

See Python implementation attached. This implementation uses the partially-mapped crossover path encoding method defined in Larra˜naga et al. (1999). The cities are encoded as follows:

[Oradea ‣ 1, Zerind ‣ 2, Arad ‣ 3, Sibiu ‣ 4, Fagaras ‣ 5, Bucharest ‣ 6, Pitesti ‣7, Rimnicu Vilcea ‣ 8].

1. The initial population this algorithm holds 2n number of cities encoding which is shuffled to represent a random ordering of cities traveled.
2. A population is randomly selected and evaluated using the graph edges. The population is decoded to generate a path of cities where a cost is calculated as follows:
    a. Neighboring cities have n path costs presented by the distance/time required to move to a city
    b. Repeated cities have cost of 0 as the agent is not moving between cities. That is, there is no path cost for staying in the same city.
    c. Non traversable such that the path cannot be followed with a graph (next city in the path is not a neighbor or the current city) the cost is set to -1
    d. The path cost is squashed using a fitness functions use the smallest cost as to evaluate a function. The equation is $n^{\frac{1}{cost}}$ where n is the number of cities and the cost is the total distance to traverse child population. The goal is to minimize *n* as possible with the perfect child giving n. The purpose of this is encoding a quick reference of traversable paths if the fitness function is < 1 then the path is bad and will be culled (the result is less than 1). If fitness is > 1 then the path is good but in might be suboptimal. In this implementation a path equal to 1 is not possible because an agent will have to be repeat cities
3. Child nodes are produced by taken the inner portions of the encoding parent encoding and swapping them.
4. Mutations occur on 0.25 chance to introduce a random city in the encoding. A higher mutation rate may lower the total iteration but the result is more mutation is stable paths resulting in worse encodings.
5. Populations are culled based on off a fitness function when a population reaches a high enough number. As states above the population is culled based on the lowest fitness number *2 (the highest path cost of a population). The highest path cost is returned and is reused as a survivors bias to continually improve a solution.

The output for when an agent is finished looks as below:

Iteration: 2999

Jobs Done!

Possible paths:

[[['Oradea', 'Zerind', 'Zerind', 'Zerind', 'Arad', 'Arad', 'Zerind', 'Oradea'], 268],

[['Oradea', 'Zerind', 'Zerind', 'Zerind', 'Arad', 'Arad', 'Zerind', 'Oradea'], 268]]

Problem 5: Chapter 4, Ex. 4.5

The AND-OR-GRAPH-SEARCH algorithm in Figure 4.11 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were to store every visited state and check against that list. (See BREADTH-FIRST-SEARCH in Figure 3.11 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (Hint: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels, as defined in Section 4.3.3, to avoid having multiple copies of subplans.

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

AND-OR-GRAPH-SEARCH searches though a graph where the OR actions can generate multiple states that are introduced by the environment. These are called AND nodes. The algorithm first examines the OR-SEARCH for a goal evaluation and if the state already exists within the known path. Then the algorithm looks act each action and generates an AND-SEARCH generates new states to be evaluated by the function. This builds AND-OR-GRAPH set of nodes that can be reference when search to see if a new state has already been encountered. Therefore, the information that should be stored in the each node would be similar to information stored within the nodes of A*:

- Expanded node list – List of encountered states in a path which point to other nodes, these can be counted as solved states.
- Frontier node list – list of open or failed states in a path which point to failures or open states which have not been expended.

When new state is generated:

1. If the current states test true during a goal test, mark the node s solved and moved to the expended node list
2. If the current state does not test true during a goal test and has not failed, then the node is still being explored and move to the frontier node list

3. If the states is an OR-node and the child node are solved states, update the node as solved. The subsequent actions are the new plan for this node.
4. IF the state is an and node, its child nodes are solved, update the nodes as solved

When the algorithm runs through each node, the states is checked in the expanded nodes list the parent node will point to the node in the expanded states. This can generate cyclic solutions that point back within it. The labels used by the algorithm will assist in reducing repeated actions and node states. The labels that can be used need to be unique to ensure that nodes in the expanded node list are not ambiguous, that is an action from a unique state cannot lead to redundant unique states. This is what is being demonstrated in Figure 4.12 of the text.

Python Code

Problem 4A

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 27 21:15:34 2019

@author: psubacz
"""

""" A Python Class
A simple Python graph class, demonstrating the essential
facts and functionalities of graphs.
"""
import sys

class Node:
    '''
    Simple node class for the agent to keep track of self in graph
    '''
    def __init__(self, parent = None, node = None, dist = 0,all_visited = False):
        self.parent = parent
        self.node = node
        self.h = 0
        self.dist = dist
        self.all_visited = all_visited

    def get_parent(self):
        return self.parent

    def get_node(self):
        return self.node
```

```
    def get_dist(self):
        return self.dist

    def get_all_visited(self):
        return self.all_visited

    def set_all_visited(self):
        self.all_visited = True

class Graph(object):
    '''
    Graph class source: https://www.python-course.eu/graphs_python.php
    '''
    def __init__(self, graph_dict=None):
        """ initializes a graph object
            If no dictionary or None is given,
            an empty dictionary will be used
        """
        if graph_dict == None:
            graph_dict = {}
        self.__graph_dict = graph_dict

    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()
```

```
def add_vertex(self, vertex):
    """ If the vertex "vertex" is not in
        self.__graph_dict, a key "vertex" with an empty
        list as a value is added to the dictionary.
        Otherwise nothing has to be done.
    """
    if vertex not in self.__graph_dict:
        self.__graph_dict[vertex] = []

def add_edge(self, edge):
    """ assumes that edge is of type set, tuple or list;
        between two vertices can be multiple edges!
    """
    edge = set(edge)
    (vertex1, vertex2) = tuple(edge)
    if vertex1 in self.__graph_dict:
        self.__graph_dict[vertex1].append(vertex2)
    else:
        self.__graph_dict[vertex1] = [vertex2]

def __generate_edges(self):
    """ A static method generating the edges of the
        graph "graph". Edges are represented as sets
        with one (a loop back to the vertex) or two
        vertices
    """
    edges = []
    for vertex in self.__graph_dict:
        for neighbour in self.__graph_dict[vertex]:
            if {list(neighbour)[0], vertex} not in edges:
                edges.append({vertex, list(neighbour)[0]})
```

```
        return edges

    def __str__(self):
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

def hueristic():
    pass

def agent_hill_climbing(_map,origin,frontier,expanded = None,max_itar = 30):
    '''
    Implementation of a hill climbing agent to solve a TSP problem
    '''
    #Create Origin Node
    origin_node = Node(parent = None, node = origin)
    #Set Current node to Origin
    current_node = origin_node
    #Create a list of frontier nodes
    frontier_nodes = []

    curr_iter = 0

    #While not all cities have been visited
    while (True):
        all_visited = False
        at_origin = False
```

```
    all_cities = list(_map)

    print('Iteration: ', curr_iter,'\nCurrently in City: ',current_node.get_node(), 'Distance traveled:
',current_node.get_dist())

    #1. Check for gaol conditions - See if all cities have been visited
    #  and if at origin
    cities_visited = [current_node.get_node()]
    solution =  current_node

    #This loop traces the nodes back to the orgin and checks off visited
    #  cities if all cities have been checked then set the goal all_visited
    #  to true
    while (solution.get_parent() != None):
        #Grab the parent node
        solution = solution.get_parent()
        #Add to visited cities list
        cities_visited.append(solution.get_node())
        #Check off the visited city and remove from list
        if (all_cities.count(solution.get_node())>=1):
            all_cities.remove(solution.get_node())
        #if all cities have been checked then set the goal to true
        if (len(all_cities) == 0):
            current_node.set_all_visited()
            all_visited = True
            print('ALL VISITED')

    #Check to see if the agent is at the origin node origin
    if (current_node.get_node() == origin):
        print('AT ORIGIN')
        at_origin = True
```

```
        #If both conditions have been met, then break and return the solution
        if at_origin and all_visited:
            print('Jobs Done!')
            cities_visited.reverse()
            print('Travel Solution: ',cities_visited)
            print('Travel Distance: ',current_node.get_dist())
            break

        #1. Look at neighbor cities
        neighbors = list(_map[current_node.get_node()])

        #2 - Generate nodes at next level
        total_poss_dist = 0

#       print(list(_map[current_node.get_node()]))
        for neighbor in neighbors:
            new_node = Node(parent = current_node, node = list(neighbor)[0],
                            dist = current_node.get_dist()+neighbor[list(neighbor)[0]],
                            all_visited = current_node.get_all_visited())


            frontier_nodes.append(new_node)

            total_poss_dist = total_poss_dist+current_node.get_dist()+neighbor[list(neighbor)[0]]
        sol = []
        #Select the lowest node
        for temp_node in frontier_nodes:
            sol.append(temp_node.get_node())
            if (temp_node.get_dist() < total_poss_dist):
                total_poss_dist = temp_node.get_dist()
                current_node = temp_node
```

```
        frontier_nodes.remove(current_node)

        if (current_node.get_node() == origin):
            at_origin = True

        curr_iter += 1

if __name__ == "__main__":
    romania_map = { 'Oradea' :[{'Zerind':71},{'Sibiu':151}],
                    'Zerind':[{'Oradea':71},{'Arad':75}],
                    'Arad':[{'Zerind':75},{'Sibiu':140},{'Timisoara':118}],
                    'Timisoara':[{'Arad':118},{'Lugoj':111}],
                    'Lugoj':[{'Mehadia':70},{'Timisoara':111}],
                    'Mehadia':[{'Drobeta':75},{'Lugoj':70}],
                    'Drobeta':[{'Mehadia':75},{'Craiova':120}],
                    'Craiova':[{'Drobeta':120},{'Rimnicu Vilcea':146},{'Pitesti':138}],
                    'Rimnicu Vilcea':[{'Craiova':146},{'Pitesti':97},{'Sibiu':80}],
                    'Sibiu':[{'Oradea':151},{'Arad':140},{'Fagaras':99},{'Rimnicu Vilcea':80}],
                    'Fagaras':[{'Sibiu':99},{'Bucharest':211}],
                    'Pitesti':[{'Rimnicu Vilcea':97},{'Craiova':138},{'Bucharest':101}],
                    'Bucharest':[{'Pitesti':101},{'Giurgiu':90},{'Fagaras':211},{'Urziceni':85}],
                    'Giurgiu':[{'Bucharest':90}],
                    'Urziceni':[{'Bucharest':85},{'Hirsova':98},{'Vaslui':142}],
                    'Hirsova':[{'Urziceni':98},{'Eforie':86}],
                    'Eforie':[{'Hirsova':86}],
                    'Vaslui':[{'Urziceni':142},{'Iasi':92}],
                    'Iasi':[{'Vaslui':92},{'Neamt':87}],
                    'Neamt':[{'Iasi':87}]}

    romania_map = {'Oradea' :[{'Zerind':71},{'Sibiu':151}],
```

```
                'Zerind':[{'Oradea':71},{'Arad':51}],
                'Arad':[{'Zerind':75},{'Sibiu':140}],
                'Sibiu':[{'Arad':140},{'Oradea':151}]}

    graph = Graph(romania_map)

#    print("Vertices of graph:")
#    print(graph.vertices())
#
#    print("Edges of graph:")
#    print(graph.edges())

    frontiers = graph.vertices()
    agent_hill_climbing(romania_map,'Oradea',frontiers)
```

Problem 4B

```
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 28 16:08:19 2019

@author: psubacz
"""


# -*- coding: utf-8 -*-
"""
Created on Fri Sep 27 21:15:34 2019

@author: psubacz
"""
import random as rd

class Node:
```

```
    '''
    Simple node class for the agent to keep track of self in graph
    '''
    def __init__(self, parent = None, node = None, dist = 0,all_visited = False):
        self.parent = parent
        self.node = node
        self.h = 0
        self.dist = dist
        self.all_visited = all_visited

    def get_parent(self):
        return self.parent

    def get_node(self):
        return self.node

    def get_dist(self):
        return self.dist

    def get_all_visited(self):
        return self.all_visited

    def set_all_visited(self):
        self.all_visited = True

class Graph(object):
    '''
    Graph class source: https://www.python-course.eu/graphs_python.php
    '''
    def __init__(self, graph_dict=None):
        """ initializes a graph object
```

```
        If no dictionary or None is given,
        an empty dictionary will be used
    """
    if graph_dict == None:
        graph_dict = {}
    self.__graph_dict = graph_dict

def vertices(self):
    """ returns the vertices of a graph """
    return list(self.__graph_dict.keys())

def edges(self):
    """ returns the edges of a graph """
    return self.__generate_edges()

def add_vertex(self, vertex):
    """ If the vertex "vertex" is not in
        self.__graph_dict, a key "vertex" with an empty
        list as a value is added to the dictionary.
        Otherwise nothing has to be done.
    """
    if vertex not in self.__graph_dict:
        self.__graph_dict[vertex] = []

def add_edge(self, edge):
    """ assumes that edge is of type set, tuple or list;
        between two vertices can be multiple edges!
    """
    edge = set(edge)
    (vertex1, vertex2) = tuple(edge)
    if vertex1 in self.__graph_dict:
```

```
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            with one (a loop back to the vertex) or two
            vertices
        """
        edges = []
        for vertex in self.__graph_dict:
            for neighbour in self.__graph_dict[vertex]:
                if {list(neighbour)[0], vertex} not in edges:
                    edges.append({vertex, list(neighbour)[0]})
        return edges

    def __str__(self):
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res




def agent_genetic(graph,_map,origin,breakpoint = 10,mutation_rate = 25):
    '''
    Implementation of a genetic agent to solve a TSP problem
```

```
    The cities are incoded as [Oradea ‣ 1, Zerind ‣ 2, Arad ‣ 3, Sibiu ‣ 4,
     Fagaras ‣ 5, Bucharest ‣ 6, Pitesti ‣7, Rimnicu Vilcea ‣ 8]
    '''

    def fitness_fuction(population,dist):
    #    1. What is the total distance traveled
    #    2. Are all cities accounted for
    #    3. Are we at the origin
    #    4. Can we travel to all
#       print('2',population[0])
        org_vrtx = population[0][0]
        vrtx = population[0][1]
#           Assume path is bad
        is_traversable = True
#       print(population[0])
        for vrtx in population[0]:
#           print(vrtx)
            #if we are not moving
            if (graph.vertices()[org_vrtx] != graph.vertices()[vrtx]):
                #If we can move from Point A to point B
                if ((list(graph.edges()[org_vrtx]).count(graph.vertices()[vrtx])>=1)or
                    (list(graph.edges()[vrtx]).count(graph.vertices()[org_vrtx])>=1)):
                    for neighbor in _map[graph.vertices()[org_vrtx]]:
                        if(list(neighbor)[0] == graph.vertices()[vrtx]):
                            population[1] = population[1] + neighbor[graph.vertices()[vrtx]]
                else:
                    is_traversable = True

            elif(graph.vertices()[org_vrtx] == graph.vertices()[vrtx]):
                    pass
```

```
            org_vrtx = vrtx
        if not (is_traversable):
            population[1] = -1
        population[2] = len(graph.vertices())**(1/population[1])
        return population

    def random_selection(population):
        dist = 1
        rand = rd.randint(0,len(population)-1)
        population[rand]=fitness_fuction(population[rand],dist)
        return population[rand]

    def reproduce(pop_1,pop_2):
        pop_3 = pop_1
#       print(pop_1[0][1:len(pop_1[0])-1])
        pop_1[0][1:len(pop_1[0])-1]
        pop_2[0][1:len(pop_1[0])-1]
        pop_3[0][1:len(pop_1[0])-1] = pop_2[0][1:len(pop_1[0])-1]
#       print('np',pop_1,pop_2,pop_3)
        return pop_3



    def mutate(child):
        z = rd.randint(1,len(graph.vertices())-2)
        q = rd.randint(0,len(graph.vertices())-1)
        child[0][z] = q
        return child

    def cull_pop(population,surivors_bias=len(graph.vertices())**10):
        survival_bar = len(graph.vertices())**(1/(surivors_bias))
        bias_delta = surivors_bias
```

```
        for i in range(len(population)):
            if (len(population)<=8):
                break
            #randomly pick a pop
            i = rd.randint(0,len(population)-1)
            #We want to maximize this number, the higher this number is the
            #  closer the agent is to a solution. As the survivors_bias shinks,
            #  the minimum check to surivve will require better scores will only get lower
            if (population[i][1]<=surivors_bias):
                delta_h =  population[i][2]-survival_bar
                if (delta_h>=1):
                    new_bias_delta =surivors_bias - population[i][1]
                    if(population[i][1]<surivors_bias):
#                     if(new_bias_delta>bias_delta):
                        bias_delta = population[i][1]
#                   print(delta_h)
                population.pop()
#       print(bias_delta)
        return population,bias_delta

    def init_population(num_population,pop_mult=2):
        population = [[],0,0]
        for i in range(pop_mult):
            for i in range(0,num_population):
                population[0].append(i)
            rd.shuffle(population[0])
        population[0][0] = 0
        population[0][-1] = 0
        return population

    population = [init_population(len(graph.vertices())),init_population(len(graph.vertices()))]
```

```
    x = 0
    while True:
        if x > 1000000:
            return population
        x+=1
#       print('')
        print('Iteration: ',x)
        new_population = []
        for i in range(len(population)):
            pop_1 = random_selection(population)
            pop_2 = random_selection(population)
            child = reproduce(pop_1,pop_2)
            if (rd.random() <= mutation_rate):
                mutate(child)
                child = mutate(child)
            new_population.append(child)
        for pip in new_population:
            population.append(pip)

        if (((2**3)%len(population))>=1):
            population,_ = cull_pop(population,)
#       print(population)
#           break

if __name__ == "__main__":
    romania_map = { 'Oradea' :[{'Zerind':71},{'Sibiu':151}],
                    'Zerind':[{'Oradea':71},{'Arad':75}],
                    'Arad':[{'Zerind':75},{'Sibiu':140},{'Timisoara':118}],
                    'Timisoara':[{'Arad':118},{'Lugoj':111}],
                    'Lugoj':[{'Mehadia':70},{'Timisoara':111}],
                    'Mehadia':[{'Drobeta':75},{'Lugoj':70}],
```

```
                'Drobeta':[{'Mehadia':75},{'Craiova':120}],
                'Craiova':[{'Drobeta':120},{'Rimnicu Vilcea':146},{'Pitesti':138}],
                'Rimnicu Vilcea':[{'Craiova':146},{'Pitesti':97},{'Sibiu':80}],
                'Sibiu':[{'Oradea':151},{'Arad':140},{'Fagaras':99},{'Rimnicu Vilcea':80}],
                'Fagaras':[{'Sibiu':99},{'Bucharest':211}],
                'Pitesti':[{'Rimnicu Vilcea':97},{'Craiova':138},{'Bucharest':101}],
                'Bucharest':[{'Pitesti':101},{'Giurgiu':90},{'Fagaras':211},{'Urziceni':85}],
                'Giurgiu':[{'Bucharest':90}],
                'Urziceni':[{'Bucharest':85},{'Hirsova':98},{'Vaslui':142}],
                'Hirsova':[{'Urziceni':98},{'Eforie':86}],
                'Eforie':[{'Hirsova':86}],
                'Vaslui':[{'Urziceni':142},{'Iasi':92}],
                'Iasi':[{'Vaslui':92},{'Neamt':87}],
                'Neamt':[{'Iasi':87}]}

    romania_map = {'Oradea' :[{'Zerind':71},{'Sibiu':151}],
                'Zerind':[{'Oradea':71},{'Arad':51}],
                'Arad':[{'Zerind':75},{'Sibiu':140}],
                'Sibiu':[{'Oradea':151},{'Arad':140},{'Fagaras':99},{'Rimnicu Vilcea':80}],
                'Fagaras':[{'Sibiu':99},{'Bucharest':211}],
                'Bucharest':[{'Pitesti':101},{'Fagaras':211},{'Urziceni':85}],
                'Pitesti':[{'Rimnicu Vilcea':97},{'Bucharest':101}],
                'Rimnicu Vilcea':[{'Pitesti':97},{'Sibiu':80}]}
#    print(romania_map['Oradea'])
    graph = Graph(romania_map)
#
    pop = agent_genetic(graph,romania_map,'Oradea')
```