Chapter 3, Ex 3.13

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.

1. The graph separation property states that every path from the initial state to an unexplored state has to pass through a state in the frontier. Let's start with the initial state. We assume that the property holds at the beginning. That is, the initial node acts as a frontier node to itself that connects to the further successor nodes. So once all the neighbors are recognised for the initial nodes, it then adds to the explored set and the neighbors become the frontiers. Over here the first iteration completes. Now assuming that none of them is the goal node, we need to pass through the frontier to get to any unexplored node. So, between the initial node and the explored node there is always a frontier that it has to pass through. This property will hold true for graph searches because as you pass every iteration to look for the goal state, we have to pass through a frontier to get to an unexplored node.
2. This property can be violated if the algorithm does not explore all the successors of the frontier node before adding it into the explored set. Or even when only some of the successors are added as frontier nodes. This could happen when an algorithm does not need to explore all the nodes to reach its goal state. Hence, in this case the graph separation property is violated.

Chapter 3, Ex 3.16

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.

1. Suppose that the pieces fit together exactly with no slack. Give a precise formulation of the task as a search problem.
    a. Initial state: A puzzle piece selected at random.
    b. Successor function: Adding a piece to the existing structure depending on the end, whether it is an open hole or not. The function could also have the condition of checking if the structure is overlapping. It can terminate that step as soon as the condition comes true.
    c. Goal State: To get a railway track that has no loose ends and no overlapping tracks.
    d. Step Cost: Cost will be one for each railway piece.

2. Identify a suitable uninformed search algorithm for this task and explain your choice.
    a. DFS seems the best uninformed search choice. As all the pieces are at the same depth it will pick one option at a time and start exploring that path till the end because the solution is at the bottom of the tree. Whereas in BFS, the algorithm will explore all the options together. This will lead to unnecessary memory usage and unlike DFS it will terminate only at the leaf nodes layer.
3. Explain why removing any one of the “fork” pieces makes the problem unsolvable.
    a. Removing one of the fork pieces makes the problem unsolvable because there will be two loose ends of one type and only one loose end of the other type. So, it is not possible to reach the goal state as there will be at least one loose end remaining. We will need a fork

piece only to have an ending to the track.

4. Give an upper bound on the total size of the state space defined by your formulation. 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.
    a. The way to do this would be to start off by figuring out that there are only three 3 open pegs at maximum. This would be if there is an open peg with a fork (2 more pegs) added onto the initial one. Depth equals 32 as there are a total of 32 pieces. We can account all the possible combinations of the pieces into this formula (12 + (2*16) + (2*2) + (2*2*2)) = 56 total choices on each peg. Multiply 56*3 = 168 for each of the 3 open pegs. And the end calculating an upper bound you would get 168^32 / (12! * 16! * 2! * 2!). 168^32 is for each level of the depth and the factorials are used because it also considers the pieces that are used twice.

Chapter 3 Ex 3.26

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).

1. What is the branching factor b in this state space?
    a. The branching factor is 4 as there are four neighbours for each node.
2. How many distinct states are there at depth k (for k > 0)?
    a. 4K, where K is the depth and 4 is the number of nodes at each depth.
3. What is the maximum number of nodes expanded by breadth-first tree search?
    a. The maximum number of nodes expanded by breadth-first tree search is the branching factor to the depth ie. b^d. For a given point (x,y) the number of expanded nodes would be b^(x+y). As b is 4 in this case then it would be 4^(x+y).
4. What is the maximum number of nodes expanded by breadth-first graph search?
    a. There are quadratically many states within the square for depth x + y, so the answer is 2(x + y)(x + y + 1) - 1.
5. Is h = |u − x| + |v − y| an admissible heuristic for state (u, v)? Explain.
    a. Yes it is, as this is the Manhattan distance formula. Hence it gives us the path length.
6. How many nodes are expanded by A∗ graph search using h?
    a. All the nodes are candidates for optimal path in a rectangle of (x,y). So, in the worst case all the nodes need to be expanded.
7. Does h remain admissible if some links are removed?
    a. Yes, removing links will just increase the path length as it will lead to detours so h will be more than expected.
8. Does h remain admissible if some links are added between nonadjacent states?
    a. No, because the path length might reduce actual path distance and may go below Manhattan distance.

Chapter 4, Ex 4.3

1. In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise 3.30.
    a. Implement and test a hill-climbing method to solve TSPs. Compare the results with op-timal solutions obtained from the A∗algorithm with the MST heuristic (Exercise 3.30).

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

TSP using hill-climbing method

```
import random

def hillClimbing(tsp):
    currentSolution = randSolution(tsp)
    currentDistance = routeDistance(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourDistance = getBestNeighbour(tsp,
neighbours)

    while bestNeighbourDistance < currentDistance:
        currentSolution = bestNeighbour
        currentDistance = bestNeighbourDistance
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourDistance = getBestNeighbour(tsp,
neighbours)

    return currentSolution, currentDistance

def randSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)

    return solution

def routeDistance(tsp, solution):
    route = 0
    for i in range(len(solution)):
        route += tsp[solution[i - 1]][solution[i]]
    return route

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
```

```
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
    bestDistance = routeDistance(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentDistance = routeDistance(tsp, neighbour)
        if currentDistance < bestDistance:
            bestDistance = currentDistance
            bestNeighbour = neighbour
    return bestNeighbour, bestDistance

def randomProblemGenerator(noOfCities):
    tsp = []
    for i in range(noOfCities):
        distances = []
        for j in range(noOfCities):
            if j == i:
                distances.append(0)
            elif j < i:
                distances.append(tsp[j][i])
            else:
                distances.append(random.randint(10, 1000))
        tsp.append(distances)
    return tsp

def main():
    tsp = randomProblemGenerator(20)
    print("Generated random tsp: \n")
    print(*tsp, sep="\n")

print("_____
_____\n")
    solutions = []

    for i in range(10):
        solutions.append(hillClimbing(tsp))
```

```
    print(*solutions, sep="\n")
    print("\nBest solution is: ")
    print(min(solutions, key=lambda x: x[1]))

if __name__ == "__main__":
    main()
```

Output:

```
Generated random tsp:

[0, 810, 541, 706, 234, 220, 143, 367, 421, 805, 296, 484, 301, 477,
567, 27, 724, 527, 607, 924]
[810, 0, 754, 942, 318, 373, 579, 279, 71, 702, 422, 589, 785, 436,
458, 435, 867, 433, 650, 642]
[541, 754, 0, 137, 131, 247, 665, 867, 672, 29, 41, 880, 635, 195, 213,
391, 649, 503, 779, 132]
[706, 942, 137, 0, 85, 46, 922, 181, 320, 157, 472, 625, 812, 111, 634,
15, 302, 632, 805, 737]
[234, 318, 131, 85, 0, 636, 411, 502, 410, 781, 906, 919, 282, 538,
947, 356, 291, 600, 692, 709]
[220, 373, 247, 46, 636, 0, 458, 842, 846, 268, 446, 473, 519, 239,
916, 157, 40, 902, 721, 574]
[143, 579, 665, 922, 411, 458, 0, 339, 799, 573, 254, 615, 813, 925,
760, 561, 245, 195, 86, 118]
[367, 279, 867, 181, 502, 842, 339, 0, 538, 991, 270, 323, 698, 660,
788, 649, 864, 718, 798, 502]
[421, 71, 672, 320, 410, 846, 799, 538, 0, 789, 166, 960, 70, 120, 418,
75, 58, 629, 293, 975]
[805, 702, 29, 157, 781, 268, 573, 991, 789, 0, 693, 135, 855, 575,
721, 707, 466, 656, 875, 372]
[296, 422, 41, 472, 906, 446, 254, 270, 166, 693, 0, 737, 155, 922,
249, 512, 405, 81, 196, 993]
[484, 589, 880, 625, 919, 473, 615, 323, 960, 135, 737, 0, 317, 844,
544, 820, 117, 540, 513, 536]
[301, 785, 635, 812, 282, 519, 813, 698, 70, 855, 155, 317, 0, 830,
605, 177, 18, 774, 980, 567]
[477, 436, 195, 111, 538, 239, 925, 660, 120, 575, 922, 844, 830, 0,
857, 16, 274, 835, 713, 633]
[567, 458, 213, 634, 947, 916, 760, 788, 418, 721, 249, 544, 605, 857,
0, 568, 543, 209, 713, 553]
[27, 435, 391, 15, 356, 157, 561, 649, 75, 707, 512, 820, 177, 16, 568,
0, 252, 849, 797, 147]
[724, 867, 649, 302, 291, 40, 245, 864, 58, 466, 405, 117, 18, 274,
543, 252, 0, 55, 196, 312]
[527, 433, 503, 632, 600, 902, 195, 718, 629, 656, 81, 540, 774, 835,
209, 849, 55, 0, 720, 299]
[607, 650, 779, 805, 692, 721, 86, 798, 293, 875, 196, 513, 980, 713,
713, 797, 196, 720, 0, 803]
```

```
[924, 642, 132, 737, 709, 574, 118, 502, 975, 372, 993, 536, 567, 633,
553, 147, 312, 299, 803, 0]
```

```
([7, 1, 8, 13, 3, 15, 0, 4, 2, 19, 17, 14, 11, 9, 5, 16, 12, 10, 18,
6], 3409)
([0, 7, 11, 9, 2, 4, 1, 8, 12, 10, 14, 17, 19, 6, 18, 16, 5, 3, 13,
15], 2996)
([18, 10, 17, 14, 2, 9, 5, 13, 3, 7, 11, 1, 4, 0, 12, 16, 8, 15, 19,
6], 3794)
([15, 13, 1, 4, 3, 9, 5, 0, 11, 7, 10, 17, 14, 2, 19, 6, 18, 16, 12,
8], 3775)
([5, 16, 17, 1, 8, 12, 4, 7, 11, 9, 2, 19, 14, 10, 18, 6, 0, 15, 13,
3], 3499)
([7, 0, 6, 18, 8, 1, 19, 15, 13, 5, 16, 17, 10, 12, 4, 2, 14, 11, 9,
3], 3978)
([15, 13, 5, 3, 7, 1, 11, 9, 19, 6, 18, 8, 16, 17, 14, 10, 2, 4, 12,
0], 3707)
([5, 3, 13, 15, 0, 4, 1, 14, 17, 19, 6, 18, 16, 8, 12, 11, 7, 10, 2,
9], 3494)
([13, 15, 3, 4, 12, 11, 7, 1, 5, 9, 19, 6, 0, 10, 2, 14, 17, 16, 18,
8], 4014)
([7, 6, 4, 0, 5, 13, 3, 9, 2, 14, 1, 8, 15, 19, 17, 10, 18, 16, 12,
11], 4134)

Best solution is:
([0, 7, 11, 9, 2, 4, 1, 8, 12, 10, 14, 17, 19, 6, 18, 16, 5, 3, 13,
15], 2996)
```

TSP using Genetic Algorithm:

```
import numpy as np, random, matplotlib.pyplot as plt, operator, pandas
as pd

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
```

```
        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness

def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration

def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
    return route

def initialPopulation(popSize, cityList):
    population = []

    for i in range(0, popSize):
        population.append(createRoute(cityList))
```

```
    return population

def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    return sorted(fitnessResults.items(), key = operator.itemgetter(1),
reverse = True)

def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults

def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool

def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
```

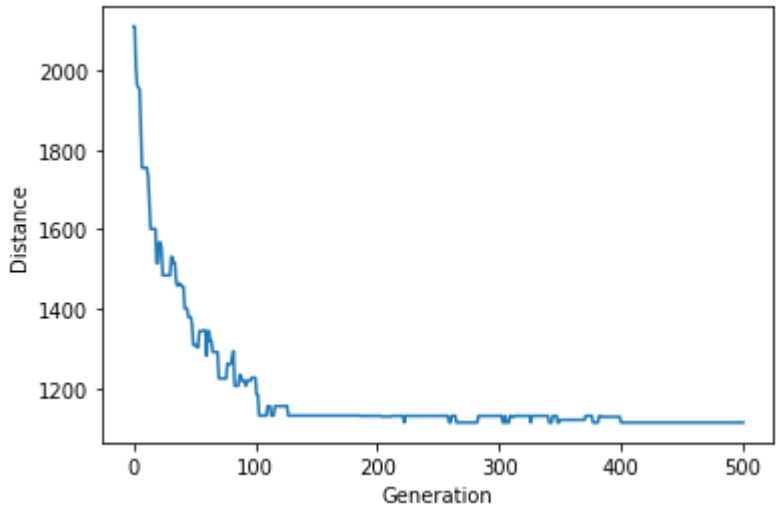
```
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]

    child = childP1 + childP2
    return child

def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
    return children

def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
    return individual

def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop

cityList = []

for i in range(0,25):
```

```
    cityList.append(City(x=int(random.random() * 200),
y=int(random.random() * 200)))

def geneticAlgorithm(population, popSize, eliteSize, mutationRate,
generations):
    pop = initialPopulation(popSize, population)
    progress = []
    progress.append(1 / rankRoutes(pop)[0][1])
    print("Initial distance is: " + str(1 / rankRoutes(pop)[0][1]) +
"\n")

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])

    print("Plotting the progress of the algorithm: ")
    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()

    print("Final generation's distance is: " + str(1 /
rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    print("The best route is: ")
    return bestRoute

geneticAlgorithm(population=cityList, popSize=100, eliteSize=20,
mutationRate=0.01, generations=500)
```

Output:

Initial distance is: 2111.201531714716

Plotting the progress of the algorithm:



Final generation's distance is: 1113.3161855356714
The best route is:
[(168,53),
(140,74),
(134,104),
(179,89),
(195,121),
(167,133),
(176,139),
(182,197),
(72,198),
(18,175),
(37,118),
(47,60),
(1,4),
(6,65),
(27,105),
(73,160),
(112,121),
(103,84),
(106,79),
(103,63),
(95,62),
(92,17),
(109,3),
(134,40),
(188,24)]

Chapter 4, Ex 4.5

The AND-OR-GRAPH-SEARCHalgorithm 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 where to store ever visited state and check against that list. (See BREADTH-FIRST-SEARCHin 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 inSection 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 has previously been solved then return RECALL-SUCCESS(state)
if state has previously failed for a subset of path then return failure
if state is on path then
        RECORD-FAILURE(state, path)
        return failure
for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan != failure then
                RECORD-SUCCESS(state, action plan))
                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])
```

In the AND-OR search, a state returns failure even if a subset of it has failed previously. It depends on the path taken if the solution can be found or not. So, if OR search fails then it has to deal with it carefully.
We can name all new solutions identified, record these labels, and then return the label if these states are visited again to avoid repeating sub-solutions. So, basically we don’t need to go down that path again where we have to find the solution again if the program has already processed that state before. For example, if state[k] where k is some constant has a solution[k] which is already stored then it can directly implement plan[k].