Robin Mehta
Artificial Intelligence, Homework 2

Problem 1

Part 2

Breadth-First-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	71	7	25	7	17	8
2	34	2	15	10	30	11
3	234	7	83	31	77	32
4	331	8	100	27	60	28
5	189	5	79	44	104	45
Total	859	29	302			

Depth-First-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	25	6	10	9	21	10
2	25	5	11	10	30	11
3	101	21	38	35	89	36
4	262	28	82	41	90	42
5	165	18	69	52	128	53
Total	578	78	210			

Uniform-Cost-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	54	8	19	7	17	8
2	34	2	15	10	30	11
3	164	11	61	31	77	32
4	281	22	84	27	60	28
5	195	7	82	44	104	45
Total	728	50	261			

A*-Search: Manhattan Heuristic

Robin Mehta
Artificial Intelligence, Homework 2

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	39	8	14	7	17	8
2	29	2	13	10	30	11
3	150	11	56	35	89	36
4	109	20	39	27	60	28
5	128	10	54	44	104	45
Total	455	51	176			

A*-Search: Euclidean Heuristic

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	31	7	12	11	27	12
2	29	2	13	10	30	11
3	149	13	56	35	89	36
4	119	21	42	27	60	28
5	126	10	53	44	104	45
Total	454	53	176			

Iterative-Deepening-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	25	6	10	9	21	10
2	25	5	11	10	30	11
3	101	21	38	35	89	36
4	262	28	82	41	90	42
5	165	18	69	52	128	53
Total	578	78	210			

2. Which Search Algorithm

a) Fewest total nodes: A* Euclidean Heuristic search, just beat A* Manhattan Heuristic Search by 1 node.
b) Stored fewest nodes at one time: Breadth First Search
c) A* Search Algorithms

3. The informed search algorithms did much better than the uninformed because they had a better understanding of their environments, and were able to more optimally choose successor nodes based on this knowledge. This let the algorithm choose optimal paths before choosing sub-optimal paths. Having no knowledge of an environment creates the necessity to check all possibilities for successor nodes, which can inefficient either with time or space complexity.

Part 4:

1. Local search does not do well in 8-puzzle, as only 10-15 puzzles were solved out of 25000 generated. 8-puzzle creates scenarios where it might be beneficial to make a move away from the goal in order to find a solution – otherwise, it can get stuck at any local maxima. 8-queens works well with local search because it can continue placing queens until it finds a goal state. In 8-puzzle, trying to improve the search space configuration is tricky and often plateaus without any major improvement.

2. Simulated annealing can be better than local-search because it can probabilistically infer successor moves based on the search space, which can use a function to provide more information than just the current configuration. Random Restart Hill Climbing would also work better because once it reaches a local maxima, unlike a regular hill-climbing algorithm, random restart allows for a new starting point, rather than staying stuck.

Problem 2

Part 1:

1.
a) A B C D E F
b) A D C F E B
c) A D C F E B (assuming d = 1, until algorithm stops at d = 3
d) A D B C E F
e) A C F E B D
f) A C E F B D

2.
a) Creating a 4x4x2 boolean array to mark states visited creates 32 representable states.
Example: Visit(true, 1, 2) marks that 1 cannibal and 2 missionaries on the right (goal) side of the bank has occurred so far.
b) 32 representable states
c) Initial state: Visit(true, 0, 0), indicating that there are 0 cannibals and 0 missionaries on the left side of the bank at the start.
d) Visit(true, 3, 3), indicating the goal state of 3 cannibals and 3 missionaries on the right side of the bank.

e) Potential successors of an arbitrary state will include all possibilities of (false, cannibal+1, or missionary+1). A transition will be defined by marking one of these potential successor states as true.

f)
Visit(true, 0, 0)
Visit(true, 2, 0)
Visit(true, 1, 0)
Visit(true, 3, 0)
Visit(true, 2, 0)
Visit(true, 2, 2)
Visit(true, 1, 1)
Visit(true, 1, 3)
Visit(true, 0, 3)
Visit(true, 2, 3)
Visit(true, 3, 3)

3.
Let k(n) denote the least expensive path from a (starting node) to the goal node.
Prove: Every consistent heuristic is admissible, i.e. h(n) <= k(n).

Base Case: if starting point is the goal, then h(n) = 0 <= k(n).

Inductive Step: if the starting point to the goal consists of i steps, the best path from n to n' (successor node) has i-1 steps. This gives us a proof by consistency: h(n') <= k(n').
h(n) <= c(n, a, n') + h(n') <= c(n, a, n') + k(n') = k(n).

4. Local Search in general

a) Two possible advantages of local search over graph / tree search are that it is very space efficient, only saving 1 node in memory at a time, and always improving, which allows it to often solve large, continuous problems (in addition to optimization problems). Two possible disadvantages of local search vs. graph / tree search are that when you hit a maxima or plateau, there's no better successor. Additionally, there's no clear answer on how often to restart or try to "repair" by choosing successor moves randomly (Lecture, Slide 13).

b) Hill-climbing search is a loop that continually move in direction of increasing value, and terminates when reaching a peak. Genetic algorithms start with k randomly generated states, represent each state as a string, and rate each state using an objective function (Lecture 6, Slide 21). Genetic algorithms are different than Hill-climbing algorithms, because they have an uphill tendency but not expectation, they explore randomly (mutations occur), and there is an exchange of information across parallel search threads.

c) Simulated annealing search picks a random move and accepts with a probability if it improves. Random restart searches independently, without passing information about probability.

5.

Robin Mehta
Artificial Intelligence, Homework 2

a)

```
if L:
	pole = NudgeRight()
else:
	pole = NudgeLeft()
# if initial state is V, do nothing.
```

b)

```
while(pole != vertical):
	# Establishes a new state for the pole after being nudged.
	randomNudge = randInt(0, 1)
	if randomNudge == 0:
		pole = NudgeLeft
	else:
		pole = NudgeRight
```

Problem 3: Adversarial Search

Minimax: minimize the possible loss for a worst-case scenario.

B(4)
C(3)
D(7)
A(7)
The leaf nodes are the same.

If we are using alpha-beta pruning left to right, we can skip child-nodes of D, which are: K, L, and M.

Apply minimax to the tree with chance nodes:

B(10)
C(20)
Chance node: 15

D(30)
E(20)
Chance node: 22

Robin Mehta
Artificial Intelligence, Homework 2

A(22)

Problem 1, Part 1: Appended code

robin_mehta_search.py

```python
from sets import Set
from collections import deque
import heapq

# A class that wraps the heapq functionality
# into a convenient min heap container
# which always maintains the smallest element of
# a set of data, with O(log(n)) insertion and removal
# This heap does not support node cost updating, so just
# insert duplicate nodes with different path costs if necessary
class MinHeap:
    def __init__(self):
        self.container = []

    def push(self, obj):
        heapq.heappush(self.container, obj)

    # If obj is not in heap, push it into the heap
    # If obj is already in heap, but with a higher cost,
    #   update the heap so that obj has the lower cost. Otherwise,
    #   do nothing.
    def push_or_update(self, obj):
        if obj in self.container:
            index = self.container.index(obj)
            if obj.totalcost < self.container[index].totalcost:
                self.container[index] = self.container[-1]
                self.container.pop()
                heapq.heapify(self.container)
                heapq.heappush(self.container, obj)
        else:
            heapq.heappush(self.container, obj)

    def pop(self):
        return heapq.heappop(self.container)

    def top(self):
        return self.container[0]

    # So that you can call the python len() function on this object
    def __len__(self):
        return len(self.container)

# An implementation of the node class
# You may re-implement this if you wish
# But then be sure to re-implement the extract_path function
class node:
    node_count = 0
```

Line 11, Column 15 Tab Size: 4 Python

Robin Mehta
Artificial Intelligence, Homework 2

robin_mehta_search.py UNREGISTERED

robin_mehta_search.py ×

```python
    node_count = 0
    def __init__(self, parent, position, pathcost=0, heuristic=0):
        self.idnum = node.node_count
        node.node_count += 1
        self.position = position
        self.pathcost = pathcost
        self.heuristic = heuristic
        self.totalcost = pathcost+heuristic
        self.parent = parent
        if parent is None:
            self.depth = 0
        else:
            self.depth = parent.depth+1

    # Less-than implemented so nodes can be put in MinHeap
    def __lt__(self, other):
        return self.totalcost < other.totalcost

    # Equality implemented so you can test whether nodes
    # are in a Set()
    # Nodes are equivalent if their positions are equal
    def __eq__(self, other):
        return (self.position == other.position)

    # Hash implemented so nodes can be put in Set()
    def __hash__(self):
        return hash(self.position)

# A dummy heuristic function that always returns 0
# Useful for searches that don't have a heuristic
def zero_heuristic(curpos, endpos):
    return 0

# Manhattan distance between curpos and endpos (tuples)
def manhattan(curpos, endpos):
    return abs(endpos[0]-curpos[0])+abs(endpos[1]-curpos[1])

# Euclidean distance between curpos and endpos (tuples)
def euclidean(curpos, endpos):
    return ((curpos[0]-endpos[0])**2+(curpos[1]-endpos[1])**2)**(0.5)

# A function that takes the current node, the grid,
# The goal position (endpos), and possibly the heuristic function, and returns a
def get_successors(curnode, grid, endpos, heuristic=zero_heuristic):
    # You may want to create several versions of this function

    successors = []
```

Line 11, Column 15 Tab Size: 4 Python

Robin Mehta
Artificial Intelligence, Homework 2

robin_mehta_search.py ×

```
    if (curnode.position[0] < len(grid) and curnode.position[1] < len(grid[0])):
        #currnode exists

        if ((curnode.position[0] - 1) < len(grid) and (curnode.position[1]) < len(
            #north exists
            if (grid[curnode.position[0] - 1][curnode.position[1]]): # not a wall
                i = curnode.position[0] - 1
                j = curnode.position[1]
                northPos = (i,j)
                northNode = node(curnode, northPos, 1 + heuristic(curnode.positio
                successors.append(northNode)

        if ((curnode.position[0]) < len(grid) and (curnode.position[1] + 1) < len(
            #east exists
            if (grid[curnode.position[0]][curnode.position[1] + 1]): # not a wall
                i = curnode.position[0]
                j = curnode.position[1] + 1
                eastPos = (i,j)
                eastNode = node(curnode, eastPos, 2 + heuristic(curnode.position,
                successors.append(eastNode)

        if ((curnode.position[0] + 1) < len(grid) and (curnode.position[1]) < len(
            #south exists
            if (grid[curnode.position[0] + 1][curnode.position[1]]): # not a wall
                i = curnode.position[0] + 1
                j = curnode.position[1]
                southPos = (i,j)
                southNode = node(curnode, southPos, 3 + heuristic(curnode.positio
                successors.append(southNode)

        if ((curnode.position[0]) < len(grid) and (curnode.position[1] - 1) < len(
            #west exists
            if (grid[curnode.position[0]][curnode.position[1] - 1]): # not a wall
                i = curnode.position[0]
                j = curnode.position[1] - 1
                westPos = (i,j)
                westNode = node(curnode, westPos, 4 + heuristic(curnode.position,
                successors.append(westNode)

    return successors

# A function that tests whether a node is a goal node
def goal_test(curnode, endpos):
    return (curnode.position == endpos)

# extract_path takes a goal node and returns a path
# (a list of tuples) from the initial state to the goal
```

Line 11, Column 15 Tab Size: 4 Python

robin_mehta_search.py

```python
# If you change the node class, re-implement this function
def extract_path(curnode):
    path = []
    while curnode is not None:
        path.append(curnode.position)
        curnode = curnode.parent
    return path[::-1]

# For all of the following search functions,
# "grid" is a python list of lists of integers describing the board
# Index the grid via grid[rowindex][columnindex]
# startpos is a tuple of the agent's starting position
# endpos is the location of the batter
# Do not change the arguments or return values of these functions
def depth_first_search(grid, startpos, endpos):
    # Perform depth-first search
    # Goal test at generation
    # Stack based search
    goalnode = node(None, endpos)
    total_nodes_generated = 0
    max_nodes_stored = 0
    num_iters = 0
    depth_of_goal = 0
    total_cost_of_goal = 0

    frontier = deque()
    startNode = node(None, startpos)
    frontier.append(startNode)
    exploredSet = Set()

    while (len(frontier) > 0):
        next_node = frontier.pop() # POP FROM SAME SIDE, TO IMMITATE STACK
        exploredSet.add(next_node)
        num_iters = num_iters + 1

        if (goal_test(next_node, endpos)):
            depth_of_goal = next_node.depth
            total_nodes_generated = next_node.node_count
            goalnode = next_node

            count = 0
            paths = extract_path(next_node)
            prevTuple = (0, 0) #safety initialization
            for nextTuple in paths:
                if (count < len(paths) and count > 0):
                    if (prevTuple[0] < nextTuple[0]):
                        #south
```

Robin Mehta
Artificial Intelligence, Homework 2

robin_mehta_search.py

```python
                        total_cost_of_goal += 3
                    elif (prevTuple[0] > nextTuple[0]):
                        #north
                        total_cost_of_goal += 1
                    elif (prevTuple[1] < nextTuple[1]):
                        #east
                        total_cost_of_goal += 2
                    else:
                        #west
                        total_cost_of_goal += 4
                prevTuple = nextTuple
                count = count + 1

            return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

        for unexploredNode in get_successors(next_node, grid, endpos):
            if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier)):
                exploredSet.add(unexploredNode)
                frontier.append(unexploredNode)
                if (len(frontier) > max_nodes_stored):
                    max_nodes_stored = len(frontier)

    return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

def iterative_deepening_search(grid, startpos, endpos):
    # Perform iterative deepening search
    # Goal test at generation
    goalnode = node(None, endpos)
    total_nodes_generated = 0
    max_nodes_stored = 0
    num_iters = 0
    depth_of_goal = 0
    total_cost_of_goal = 0
    d = 1

    frontier = deque()
    exploredSet = Set()
    while (d):
        startNode = node(None, startpos)
        frontier.append(startNode)
        print('outer loop called', d, len(frontier))
        while (len(frontier) > 0):
            print('inner loop called', d)
            next_node = frontier.pop() # POP FROM SAME SIDE, TO IMMITATE STACK
            exploredSet.add(next_node)
            num_iters = num_iters + 1

            if (goal_test(next_node, endpos)):
                depth_of_goal = next_node.depth
```

Robin Mehta
Artificial Intelligence, Homework 2

robin_mehta_search.py

```
            if (goal_test(next_node, endpos)):
                depth_of_goal = next_node.depth
                total_nodes_generated = next_node.node_count
                goalnode = next_node
                count = 0
                paths = extract_path(next_node)
                prevTuple = (0, 0) #safety initialization
                for nextTuple in paths:
                    if (count < len(paths) and count > 0):
                        if (prevTuple[0] < nextTuple[0]):
                            total_cost_of_goal += 3
                        elif (prevTuple[0] > nextTuple[0]):
                            total_cost_of_goal += 1
                        elif (prevTuple[1] < nextTuple[1]):
                            total_cost_of_goal += 2
                        else:
                            total_cost_of_goal += 4
                    prevTuple = nextTuple
                    count = count + 1
                return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

            for unexploredNode in get_successors(next_node, grid, endpos):
                if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier) and (unexploredNode.depth <=

                    print(unexploredNode.position)

                    exploredSet.add(unexploredNode)
                    frontier.append(unexploredNode)
                    if (len(frontier) > max_nodes_stored):
                        max_nodes_stored = len(frontier)

                d = d + 1
        break
            #print(d)
            #d = d + 1
            #break
    return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal


def breadth_first_search(grid, startpos, endpos):
    # Perform breadth-first search
    # Goal test at generation
    # Queue based search
    goalnode = node(None, endpos)
    total_nodes_generated = 0
    max_nodes_stored = 0
    num_iters = 0
    depth_of_goal = 0
```

```python
    depth_of_goal = 0
    total_cost_of_goal = 0

    frontier = deque()
    startNode = node(None, startpos)
    frontier.append(startNode)
    exploredSet = Set()
    while (len(frontier) > 0):
        next_node = frontier.popleft() #POP FROM OPPOSITE SIDE TO IMMITATE QUEUE
        exploredSet.add(next_node)
        num_iters = num_iters + 1

        if (goal_test(next_node, endpos)):
            depth_of_goal = next_node.depth
            total_nodes_generated = next_node.node_count
            goalnode = next_node

            count = 0
            paths = extract_path(next_node)
            prevTuple = (0, 0) #safety initialization
            for nextTuple in paths:
                if (count < len(paths) and count > 0):
                    if (prevTuple[0] < nextTuple[0]):
                        #south
                        total_cost_of_goal += 3
                    elif (prevTuple[0] > nextTuple[0]):
                        #north
                        total_cost_of_goal += 1
                    elif (prevTuple[1] < nextTuple[1]):
                        #east
                        total_cost_of_goal += 2
                    else:
                        #west
                        total_cost_of_goal += 4
                prevTuple = nextTuple
                count = count + 1

            return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

        for unexploredNode in get_successors(next_node, grid, endpos):
            if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier)):
                exploredSet.add(unexploredNode)
                frontier.append(unexploredNode)
                if (len(frontier) > max_nodes_stored):
                    max_nodes_stored = len(frontier)

    return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
```

```python
def uniform_cost_search(grid, startpos, endpos):
    # Perform uniform cost search
    # breadth first search, except with a MinHeap instead of deque.
    # Goal test at expansion
    goalnode = node(None, endpos)
    total_nodes_generated = 0
    max_nodes_stored = 0
    num_iters = 0
    depth_of_goal = 0
    total_cost_of_goal = 0

    frontier = MinHeap()
    startNode = node(None, startpos)
    frontier.push_or_update(startNode)
    exploredSet = Set()
    while (len(frontier) > 0):
        next_node = frontier.pop() #POP FROM OPPOSITE SIDE TO IMMITATE QUEUE
        exploredSet.add(next_node)
        num_iters = num_iters + 1

        if (goal_test(next_node, endpos)):
            depth_of_goal = next_node.depth
            total_nodes_generated = next_node.node_count
            goalnode = next_node

            count = 0
            paths = extract_path(next_node)
            prevTuple = (0, 0) #safety initialization
            for nextTuple in paths:
                if (count < len(paths) and count > 0):
                    if (prevTuple[0] < nextTuple[0]):
                        #south
                        total_cost_of_goal += 3
                    elif (prevTuple[0] > nextTuple[0]):
                        #north
                        total_cost_of_goal += 1
                    elif (prevTuple[1] < nextTuple[1]):
                        #east
                        total_cost_of_goal += 2
                    else:
                        #west
                        total_cost_of_goal += 4
                prevTuple = nextTuple
                count = count + 1

            return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
```

Robin Mehta
Artificial Intelligence, Homework 2

```python

        for unexploredNode in get_successors(next_node, grid, endpos):
            if (unexploredNode not in exploredSet):
                exploredSet.add(unexploredNode)
                frontier.push_or_update(unexploredNode)
                if (len(frontier) > max_nodes_stored):
                    max_nodes_stored = len(frontier)

    return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

def a_star_search(grid, startpos, endpos, heuristic=manhattan):
    # Perform A* search
    # Goal test at expansion
    goalnode = node(None, endpos)
    total_nodes_generated = 0
    max_nodes_stored = 0
    num_iters = 0
    depth_of_goal = 0
    total_cost_of_goal = 0

    frontier = MinHeap()
    startNode = node(None, startpos)
    frontier.push_or_update(startNode)
    exploredSet = Set()
    while (len(frontier) > 0):
        next_node = frontier.pop() #POP FROM OPPOSITE SIDE TO IMMITATE QUEUE
        exploredSet.add(next_node)
        num_iters = num_iters + 1

        if (goal_test(next_node, endpos)):
            depth_of_goal = next_node.depth
            total_nodes_generated = next_node.node_count
            goalnode = next_node

            count = 0
            paths = extract_path(next_node)
            prevTuple = (0, 0) #safety initialization
            for nextTuple in paths:
                if (count < len(paths) and count > 0):
                    if (prevTuple[0] < nextTuple[0]):
                        #south
                        total_cost_of_goal += 3
                    elif (prevTuple[0] > nextTuple[0]):
                        #north
                        total_cost_of_goal += 1
                    elif (prevTuple[1] < nextTuple[1]):
                        #east
                        total_cost_of_goal += 2
                    else:
                        #west
                        total_cost_of_goal += 4
                prevTuple = nextTuple
                count = count + 1

            return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal

        for unexploredNode in get_successors(next_node, grid, endpos, heuristic):
            if (unexploredNode not in exploredSet):
                exploredSet.add(unexploredNode)
                frontier.push_or_update(unexploredNode)
                if (len(frontier) > max_nodes_stored):
                    max_nodes_stored = len(frontier)

    return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
```