Homework 1

Part 1 Navigating Bugs

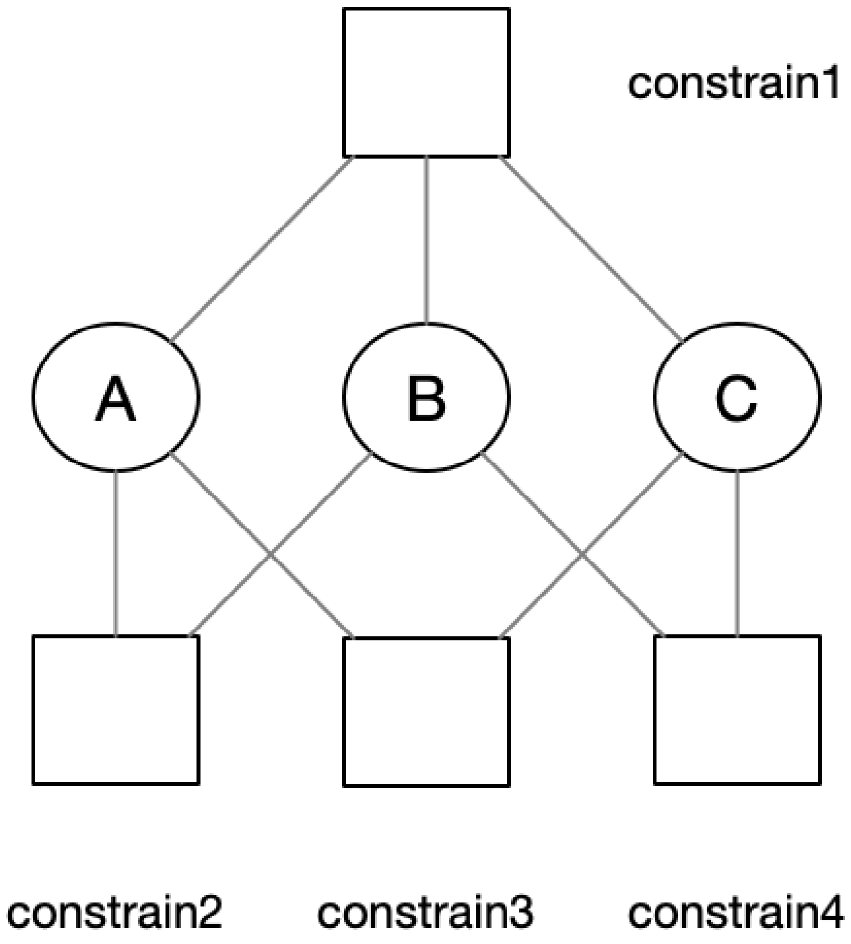
1. a) Fully observable. We have access to all the information about the environment to complete the task.
   b) Deterministic. For every time step, we control the insects so we can excactly know the next state of the enviroment, so the next state of the environment is perfectly predictable.
   c) Episodic. Because we don't need memory of past actions to determine the next best action of the insects.
   d) Static. For each time step, we deal with data source that don't change to make our next step, the data sources don't change frequently over time.
   e) Discrete. The insects to move in discrete steps, there is a finite set of possibilities that can drive the final outcome of the task
   f) Multi-agent/single agent. If there are multiple insects, it is multi-agent. If there is only one insect, it is single-agent.
2. a) The insects' locations.
   b) The size of the state space should be the number of the squares MxN. We can use the Manhattan distance between the insect and the goal as our non-trivial admissible heuristic.
3. a) The insects' locations, Booleans indicating goals.
   b) The size should be the number of the combinations. The size of the state space should be $(M * N)^K * 2^K$. Use the sum of the Manhattan distances between each insect and its goal as the non-trivial admissible heuristic.
4. a) No.
   b) The combination of insect' location and the time step it's at.
   c) $M * N * t$. We can use the Manhattan distance between the insect and the goal as our non-trivial admissible heuristic.
5. a) It becomes episodic because we need to know the insect's action in the privious time step to determine the next best action it can take.
   b) The insects' locations, the direction and the number of steps it moves in the latest time step.
   c)Suppose the insect's most steps number in one direction is v, then the size would be $M * N * 4 * v$. We can use the Manhattan distance between the insect and the goal as our non-trivial admissible heuristic.
6. a)No.
   b)The location of the insect and the time steps it spends in the pesticide squares.
   c) The size of the state space should be $M * N * (L + 1)$.We can use the Manhattan distance between the insect and the goal as our non-trivial admissible heuristic.

Part 2 Comparing Search and Heuristics

1. The order is $A-B-D-G$.
   The path solution is $A-B-D-G$.
2. The order is $A-B-C-D-F$.
   The path solution is $A-C-F$.
3. The order is $A-B-C-D-G$.
   The path solution is $A-B-D-G$.
4. h1, h2, h3 are admissible.
5. h1, h2 are consistent.
6. $A-B-C-D-F-G$
7. $A-C-D-B-G$
8. $A-C-D-G$
9. $A-C-F$

Part2 Train CSP

1.Domain: $\{1,2,3\}$
Constrains:
a. $A \neq B \; B \neq C \; A \neq C$
b. $A \geq B+1 \; or \; A \leq B-2$
c. $A > C$
d. $B \geq C+2 \; or \; B < C$
2.



The constrain1 is the a constrain in the question 1,
constrain2 is b, 3 is c, 4 is d.

3. B=1, C=2
4. B=1, C=1
5. No departure time is possible for B and C.

Programming Portion

```python
"""
COMS W4701 Artificial Intelligence - Programming Homework 1

In this assignment you will implement and compare different search strategies
for solving the n-Puzzle, which is a generalization of the 8 and 15 puzzle to
squares of arbitrary size (we will only test it with 8-puzzles for now).
See Courseworks for detailed instructions.

@author: Haoyu Yan (hy2574)
"""

import time

def state_to_string(state):
    row_strings = [" ".join([str(cell) for cell in row]) for row in state]
    return "\n".join(row_strings)


def swap_cells(state, i1, j1, i2, j2):
    """
    Returns a new state with the cells (i1,j1) and (i2,j2) swapped.
    """
    value1 = state[i1][j1]
    value2 = state[i2][j2]

    new_state = []
    for row in range(len(state)):
        new_row = []
        for column in range(len(state[row])):
            if row == i1 and column == j1:
                new_row.append(value2)
            elif row == i2 and column == j2:
                new_row.append(value1)
            else:
                new_row.append(state[row][column])
        new_state.append(tuple(new_row))
    return tuple(new_state)


def get_successors(state):
    """
    This function returns a list of possible successor states resulting
    from applicable actions.
    The result should be a list containing (Action, state) tuples.
    For example [("Up", ((1, 4, 2),(0, 5, 8),(3, 6, 7))),
                 ("Left",((4, 0, 2),(1, 5, 8),(3, 6, 7)))]
    """
    row_num = len(state[0])
    for i in range(row_num):
        for j in range(row_num):
            if state[i][j] == 0:
                row = i
                column = j
    child_states = []
    def left():
        child_states.append(("Left", swap_cells(state,row,column,row,column+1)))
    def right():
        child_states.append(("Right",swap_cells(state,row,column,row,column-1)))
    def up():
        child_states.append(("Up", swap_cells(state,row,column,row+1,column)))
    def down():
        child_states.append(("Down",swap_cells(state,row,column,row-1,column)))
    if row==0 and column==0:
        left()
        up()
    elif row == 0 and column == row_num-1:
        right()
        up()
    elif row == row_num-1 and column == 0:
        left()
        down()
    elif row ==row_num-1 and column == row_num-1:
        right()
        down()
    elif row==0 and column!=0 and column != row_num-1:
        left()
        right()
        up()
    elif row!=0 and row != row_num-1 and column==row_num-1:
        right()
        up()
        down()
    elif row==row_num-1 and column!=0 and column != row_num-1:
        left()
        right()
        down()
    elif row != 0 and row!=row_num-1 and column == 0:
        left()
        up()
        down()
    else :
        left()
        right()
        up()
        down()
    # YOUR CODE HERE . Hint: Find the "hole" first, then generate each possible
    # successor state by calling the swap_cells method.
    # Exclude actions that are not applicable.
    return child_states


def goal_test(state):
    row_number = len(state[0])
    for i in range(row_number):
        for j in range(row_number):
            if state[i][j]!= i*row_number + j:
                return False
    return True


def bfs(state):#Done
    """
```

```python
    Breadth first search.
    Returns three values: A list of actions, the number of states expanded, and
    the maximum size of the fringe.
    You may want to keep track of three mutable data structures:
    - The fringe of nodes to expand (operating as a queue in BFS)
    - A set of closed nodes already expanded
    - A mapping (dictionary) from a given node to its parent and associated action
    """
    states_expanded = 0
    max_fringe = 0

    fringe = []
    closed = set()
    parents = {}

    fringe.append(state)
    closed = set()
    solution = None
    result = None
    while fringe:
        node = fringe.pop(0)
        if node not in closed:
            states_expanded +=1
            closed.add(node)
            if goal_test(node):
                solution = []
                result = node
                break
            for child_node in get_successors(node):
                if child_node[1] not in closed:
                    fringe.append(child_node[1])
                    parents[child_node[1]] = [node, child_node[0]]
                    max_fringe = max(len(fringe), max_fringe)

    if solution!=None:
        while(result!=state):
            solution.insert(0,parents[result][1])
            result = parents[result][0]


    #  return solution, states_expanded, max_fringe
    return solution, states_expanded, max_fringe # No solution found


def dfs(state):
    """
    Depth first search.
    Returns three values: A list of actions, the number of states expanded, and
    the maximum size of the fringe.
    You may want to keep track of three mutable data structures:
    - The fringe of nodes to expand (operating as a stack in DFS)
    - A set of closed nodes already expanded
    - A mapping (dictionary) from a given node to its parent and associated action
    """
    states_expanded = 0
    max_fringe = 0
    fringe = []
    closed = set()
    parents = {}
    fringe.append(state)
    closed = set()
    solution = None
    result = None
    while fringe:
        node = fringe.pop()
        if node not in closed:
            states_expanded +=1
            closed.add(node)
            if goal_test(node):
                solution = []
                result = node
                break
            for child_node in get_successors(node):
                if child_node[1] not in closed:
                    fringe.append(child_node[1])
                    parents[child_node[1]] = [node, child_node[0]]
                    max_fringe = max(len(fringe), max_fringe)

    if solution!=None:
        while(result!=state):
            solution.insert(0,parents[result][1])
            result = parents[result][0]


    #  return solution, states_expanded, max_fringe

    return solution, states_expanded, max_fringe # No solution found


def misplaced_heuristic(state):
    """
    Returns the number of misplaced tiles.
    """
    miss_num = 0
    row_num = len(state[0])
    for i in range(row_num):
        for j in range(row_num):
            if state[i][j] != i*row_num +j:
                miss_num += 1
    return miss_num # replace this


def manhattan_heuristic(state):
    """
    For each misplaced tile, compute the Manhattan distance between the current
    position and the goal position. Then return the sum of all distances.
    """
    dist_sum = 0
    row_num = len(state[0])
    for i in range(row_num):
        for j in range(row_num):
```

```python
            dist_sum += abs(state[i][j]-(i*row_num +j))
    return dist_sum # replace this


def best_first(state, heuristic):
    """
    Best first search.
    Returns three values: A list of actions, the number of states expanded, and
    the maximum size of the fringe.
    You may want to keep track of three mutable data structures:
    - The fringe of nodes to expand (operating as a priority queue in greedy search)
    - A set of closed nodes already expanded
    - A mapping (dictionary) from a given node to its parent and associated action
    """
    # You may want to use these functions to maintain a priority queue
    from heapq import heappush
    from heapq import heappop

    states_expanded = 0
    max_fringe = 0

    fringe = []
    closed = set()
    parents = {}
    solution = None
    result = None
    cost =  heuristic(state)
    heappush(fringe,(cost,state))
    while fringe:
        node = heappop(fringe)[1:][0]
        if node not in closed:
            closed.add(node)
            states_expanded+=1
            if goal_test(node):
                solution = []
                result = node
                break
            for child_node in get_successors(node):
                if child_node[1] not in closed:
                    cost = heuristic(child_node[1])
                    heappush(fringe,(cost,child_node[1]))
                    parents[child_node[1]]=[node,child_node[0]]
                    max_fringe = max(len(fringe), max_fringe)
    while(result!=state):
        solution.insert(0,parents[result][1])
        result = parents[result][0]

    return solution, states_expanded, max_fringe # No solution found


def astar(state, heuristic):
    """
    A-star search.
    Returns three values: A list of actions, the number of states expanded, and
    the maximum size of the fringe.
    You may want to keep track of three mutable data structures:
```

```
    - The fringe of nodes to expand (operating as a priority queue in greedy search)
    - A set of closed nodes already expanded
    - A mapping (dictionary) from a given node to its parent and associated action
    """
    # You may want to use these functions to maintain a priority queue
    from heapq import heappush
    from heapq import heappop

    states_expanded = 0
    max_fringe = 0

    fringe = []
    closed = set()
    parents = {}
    costs = {}
    costs[state] = 0
    heappush(fringe,(0,state))
    while fringe:
        node = heappop(fringe)[1:][0]
        if node not in closed:
            closed.add(node)
            states_expanded+=1
            if goal_test(node):
                solution = []
                result = node
                break
            for child_node in get_successors(node):
                if child_node[1] not in closed:
                    cost = heuristic(child_node[1])+costs[node]+1
                    costs[child_node[1]] = costs[node]+1
                    heappush(fringe,(cost,child_node[1]))
                    parents[child_node[1]]=[node,child_node[0]]
                    max_fringe = max(len(fringe), max_fringe)
    while(result!=state):
        solution.insert(0,parents[result][1])
        result = parents[result][0]

    return solution, states_expanded, max_fringe # No solution found


def print_result(solution, states_expanded, max_fringe):
    """
    Helper function to format test output.
    """
    if solution is None:
        print("No solution found.")
    else:
        print("Solution has {} actions.".format(len(solution)))
    print("Total states expanded: {}.".format(states_expanded))
    print("Max fringe size: {}.".format(max_fringe))



if __name__ == "__main__":
```

```python
#Easy test case
# test_state = ((1, 4, 2),
#               (0, 5, 8),
#               (3, 6, 7))

#More difficult test case
test_state = ((7, 2, 4),
              (5, 0, 6),
              (8, 3, 1))

print(state_to_string(test_state))
print()

print("====BFS====")
start = time.time()
solution, states_expanded, max_fringe = bfs(test_state) #
end = time.time()
print_result(solution, states_expanded, max_fringe)
if solution is not None:
    print(solution)
print("Total time: {0:.3f}s".format(end-start))

print()
print("====DFS====")
start = time.time()
solution, states_expanded, max_fringe = dfs(test_state)
end = time.time()
print_result(solution, states_expanded, max_fringe)
print("Total time: {0:.3f}s".format(end-start))

print()
print("====Greedy Best-First (Misplaced Tiles Heuristic)====")
start = time.time()
solution, states_expanded, max_fringe = best_first(test_state, misplaced_heuristic)
end = time.time()
print_result(solution, states_expanded, max_fringe)
print("Total time: {0:.3f}s".format(end-start))

print()
print("====A* (Misplaced Tiles Heuristic)====")
start = time.time()
solution, states_expanded, max_fringe = astar(test_state, misplaced_heuristic)
end = time.time()
print_result(solution, states_expanded, max_fringe)
print("Total time: {0:.3f}s".format(end-start))

print()
print("====A* (Total Manhattan Distance Heuristic)====")
start = time.time()
solution, states_expanded, max_fringe = astar(test_state, manhattan_heuristic)
end = time.time()
print_result(solution, states_expanded, max_fringe)
print("Total time: {0:.3f}s".format(end-start))
```