Homework 1 – Search

TA Jonghwan Hyeon (hyeon0145@kaist.ac.kr)
TA Zae Myung Kim (zaemyung@kaist.ac.kr)

Date assigned: 14 September 2015
Date due: 23 September 2015

Submit one zip file containing your report and program codes via KLMS (naming format of the zip file: <student#>_<name>.zip, for example "20150724_john.zip"). Late submissions will not be accepted. (Write your name and student ID in your report.) This homework should be done individually.

1. Blind search to solve 8-puzzle problems (50 points)

In this assignment, we are going to implement and experiment with various search algorithms that we learned in the lectures to solve the 8-puzzle problem. A search algorithm finds a path from the initial state to the goal state by searching through the space of states that are created by applying one of the four actions (Left, Right, Up or Down) to the initial state and so on. We regard solutions with shorter path lengths better than the longer ones.

In Problem 1, we consider four blind search methods:

- Depth-first search
- Breadth-first search
- Depth-limited search
- Iterative deepening search

Through experiments, we will examine the limitation and characteristics of each search method. In order to reduce your burden on coding, we have provided the following python codes in ***problem1.zip***:

- problem.py
- fringe.py
- search.py
- run.py

You should write codes where "pass" statements are written. There is no need to modify the other parts of the codes.

problem.py

This file implements the 8-puzzle problem. You do NOT have to modify this file. The major variables and functions in this program are:

- EightPuzzle.State(tiles, blank_location): This class represents a state of the 8-puzzle problem.
  - __init__(tiles, blank_location): This method initializes a state of the 8-puzzle problem.
  - tiles: This 2-dimensinal list stores the tiles of the state.
  - blank_location: This tuple stores the location of blank tile of the state.
  - You can check an equality between two states using the equality (==) operator.
- EightPuzzle: This class represent a configuration of the 8-puzzle problem.
  - __init__(initial_state, goal_state, allowed_max_depth): This method initializes a configuration of the 8-puzzle problem.
  - initial_state: This property stores the initial state of the 8-puzzle configuration.
  - goal_state: This property stores the goal state of the 8-puzzle configuration.
  - goal_test(state): This method checks whether the state is the goal state.
  - successor_function(node): This method generates the legal states that result from applying the four actions.
  - expand(node): This method expands the node by applying the successor function and returns them.

fringe.py

This program file provides you with the skeleton codes that you can use to implement your own fringe classes. The major variables and functions in this program are:

- DepthFirstSearchFringe: This class represents a fringe for the depth-first search algorithm.
  - __init__(elements): This method initializes a queue with the given elements.
  - is_empty(): This method returns true only if there are no more elements in the queue.
  - front(): This method returns the first element of this queue.
  - remove_front(): This method returns front() and remove it from the queue.
  - insert(element): This method inserts an element into the queue in the last-in, first-out order.
  - insert_all(elements): This method insert a set of elements into the queue.

The BreadthFirstSearchFringe class in this program is almost identical to the DepthFirstSeachFringe except the following variables and functions

- BreadthFirstSearchFringe: This class represents a fringe for the breadth-first search

algorithm. You need to program and execute this class.

- insert(element): This method inserts an element into the queue in the first-in, first-out order.
- heuristic_function(node): This method calculates and returns a heuristic value of the node.

search.py

This program file provides the skeleton codes to assist you with implementing your own search functions. The major variables and functions in this program are:

- tree_search(problem, fringe): This function needs to run the TREE-SEARCH algorithm. This function should return a solution (an instance of list) if found a solution. Otherwise, return 'failure' (an instance of str). You need to program and execute this function.
- depth_limited_search(problem, limit): This function needs to run the DEPTH-LIMITED-SEARCH algorithm. This function should return a solution (an instance of list) if found a solution within the depth limit. If exceeded the depth limit, return 'cutoff' (an instance of str). Otherwise, return 'failure' (an instance of str). You need to program and execute this function.
- iterative_deepening_search(problem): This function needs to run the ITERATIVE-DEEPENING- SEARCH algorithm. This function should return a solution (an instance of list) if found a solution. Otherwise, return 'failure' (an instance of str). You need to program and execute this function.

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
      result ← RECURSIVE-DLS(successor, problem, limit)
      if result = cutoff then cutoff-occurred? ← true
      else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

run.py

This program file contains five test cases to test your program. You can run and check your program by executing this program. You must NOT modify this file.

Tasks

Using the codes provided, compete the following tasks:

A. Implement BreadthFirstSearchFringe class in ***fringe.py*** by referring to DepthFirstSearchFringe class. (5 points)

B. Implement tree_search function in ***search.py*** by referring to TREE-SEARCH. (15 points)

C. Implement depth_limited_search function in ***search.py*** by referring to DEPTH-LIMITED-SEARCH. (10 points)

D. Implement iterative_deepening_search function in ***search.py*** by referring to ITERATIVE-DEEPENING-SEARCH. (5 points)

E. Fill in the result table for each search method by running the ***run.py***. If the execution exceeds the allowed maximum depth, just write 'exceeded' in the 'Execution Time' column. (5 points)

Search name: <name of search method>				
Test Case	Execution Time	Number of Generated Nodes	Max Depth	Action Sequence
1				
2				
3				
4				
5				

F. If depth-first search exceeds the allowed maximum depth, explain in detail why depth-first search method is not complete. (5 points)

G. If the result of depth-limited search is not optimal, explain in detail why. (5 points)

2. Heuristic search to solve 8-puzzle problems (50 points)

In Problem 2, we consider two heuristic search methods:

- Greedy best-first search
- A* search

Through experiments, we will examine the limitation and characteristics of each search method. In order to reduce your burden on coding, we have provided the following python codes in ***problem2.zip***:

- problem.py
- fringe.py
- search.py
- run.py

You should write codes where "pass" statements are written. There is no need to modify the other parts of the codes.

problem.py

This program file is identical to the ***problem.py*** in the problem 1. You do NOT have to modify this file.

fringe.py

This program file provides you with the skeleton codes that you can use to implement your own fringe classes. The variables and functions in this program are almost identical to those in ***fringe.py*** in the problem 1 except the following functions:

- calculate_manhattan_distance(location1, location2): This function calculates and returns a Manhattan distance between location1 and location2. You need to program and execute this function.
- calculate_total_manhattan_distance(state1, state2): This function calculates and returns a total Manhattan distance between state1 and state2. You need to program and execute this function.
- GreedyBestFirstSearchFringe: This class represents a fringe for the greedy best-first search algorithm. You need to program and execute this class.
  - __init__(goal_state, elements): This method initializes a queue with the given elements and stores goal state to calculate heuristic values.
  - insert(element): This method inserts an element into the queue according to a priority.
  - heuristic_function(node): This method calculates and returns a heuristic value of the

node. Use “total Manhattan distance” as a heuristic function.

- AStarSearchFringe(goal_state, elements): This class represents a fringe for the A* search algorithm. You need to program and execute this class.
  - __init__(goal_state, elements): This method initializes a queue with the given elements and stores goal state to calculate heuristic values.
  - insert(element): This method inserts an element into the queue according to a priority.
  - heuristic_function(node): This method calculates and returns a heuristic value of the node. Use “total Manhattan distance” as a heuristic function.

search.py

This program file is identical to the ***serach.py*** in the problem 1. You must use your implementation of ***search.py*** in the problem 1 again.

run.py

This program file contains five test cases to test your program. You can run and check your program by executing this program. You must NOT modify this file.

Tasks

A. Implement calculate_manhattan_distance function in ***fringe.py***.(5 points)
B. Implement calculate_total_manhattan_distance function in ***fringe.py***.(5 points)
C. Implement GreedyBestFirstSearchFringe class in ***fringe.py***.(15 points)
D. Implement AStarSearchFringe class in ***fringe.py***. (15 points)
E. Fill in the result table for each search method by running the ***run.py***. If the execution exceeds the allowed maximum depth, just write ‘exceeded’ in the ‘Execution Time’ column. (5 points)

Search name: <name of search method>				
Test Case	Execution Time	Number of Generated Nodes	Max Depth	Action Sequence
1				
2				
3				
4				
5				

F. If greedy best-first search exceeds the allowed maximum depth, explain in detail why greedy best-first search method is not complete. (5 points)