LAB 05 Uninformed & Informed Search

1.1 Uniform Cost Search

Instead of expanding the shallowest node as in BFS, **uniform-cost search** expands the node n with the *lowest path cost* g(n). This is done by storing the frontier as a priority queue ordered by g. The algorithm is shown below. At each stage, the path that has the lowest cost so far is extended. In this way, the path that is generated is likely to be the path with the lowest overall cost.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 0-1. Uniform-cost search on a graph use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered.

In **informed search** strategy we use problem-specific knowledge beyond the definition of the problem itself so that we can find solutions more efficiently than can an uninformed strategy.

Informed Search

Informed search is a type of search algorithm used in Artificial Intelligence (AI) that utilizes heuristic information to guide the search towards the goal state more efficiently.

A heuristic is a rule or method used to solve problems more quickly than standard methods. In informed search, the heuristic function estimates how close a state is to the goal state, and this estimate guides the search algorithm to prioritize the states that are more likely to lead to the goal state. In contrast to uninformed search algorithms that only explore the search space without using any domain-specific knowledge, informed search algorithms utilize domain-specific knowledge to search the most promising

paths towards the goal state. This results in faster and more efficient search processes that can find solutions to problems in less time and with less computational resources.

Examples of informed search algorithms include A* search and best-first search. These algorithms use heuristics to guide the search towards the most promising paths while avoiding exploring unlikely paths.

1.2 Greedy Best First Search

The Greedy Best-First Search (GBFS) algorithm is a search algorithm that expands the node that appears to be closest to the goal based on a heuristic function. It is a modification of the Breadth-First Search (BFS) algorithm, in which the next node to be expanded is chosen based on a heuristic estimate of the distance to the goal.

Here are the steps for the Greedy Best-First Search algorithm:

Initialize a priority queue with the start node as the only element.

While the priority queue is not empty:

1. Dequeue the node with the lowest heuristic value.
2. If the dequeued node is the goal node, return the path to it.
3. Otherwise, expand the dequeued node by adding its neighbors to the priority queue.
4. For each neighbor, calculate the heuristic value as an estimate of the distance to the goal.
5. Add the neighbor to the priority queue with the calculated heuristic value as the priority.

If the priority queue is empty and the goal node has not been found, return failure.

Here's the Python code for the Greedy Best-First Search algorithm:

```
vfrom queue import PriorityQueue

def greedy_bfs(start, goal, h):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            break

        for next in neighbors(current):
```

```
            new_cost = cost_so_far[current] + cost(current, next)
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = h(next, goal)
                frontier.put(next, priority)
                came_from[next] = current

    return came_from, cost_so_far
```

Explanation of above code

- In this code, start is the starting node, goal is the goal node, and h is the heuristic function. The neighbors function returns a list of the neighbors of a given node, and the cost function returns the cost of traveling from one node to a neighboring node.
- The frontier is a priority queue that is used to store the nodes that need to be expanded. The came_from dictionary is used to keep track of the path from the start node to the current node. The cost_so_far dictionary is used to keep track of the cost of traveling from the start node to the current node.
- The algorithm runs in a loop until the frontier is empty or the goal node is found. In each iteration of the loop, the algorithm dequeues the node with the lowest heuristic value from the priority queue. If the dequeued node is the goal node, the algorithm returns the path to it. Otherwise, the algorithm expands the dequeued node by adding its neighbors to the priority queue. For each neighbor, the algorithm calculates the heuristic value as an estimate of the distance to the goal, and adds the neighbor to the priority queue with the calculated heuristic value as the priority.
- If the frontier is empty and the goal node has not been found, the algorithm returns failure.

Also in Best-first search algorithm a node is selected for expansion based on an **evaluation function**, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

```
Function best ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else
          {
               add_to_front_of_queue (successors (state));
               sort (queue);
          }
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

1.3 A* Search

A* search is a popular algorithm used in artificial intelligence for finding the shortest path between two points. It combines the heuristic estimation of the remaining cost to the goal and the actual cost from the start to the current node to determine which path to take.

Here are the steps for the A* search algorithm:

- Initialize the starting node with a cost of 0 and add it to the open set.
- While the open set is not empty, select the node with the lowest f-score (f = g + h) and remove it from the open set.
- If the current node is the goal node, we have found the shortest path. Return the path from the start to the goal node.
- Generate the successors of the current node and calculate their g and h values.

- For each successor node, check if it is in the closed set or not. If it is, skip it. Otherwise, add it to the open set.

- If a node is already in the open set, check if the new path to that node is better than the old one. If it is, update the node's g and h values and set its parent to the current node.

- Add the current node to the closed set.

```
from queue import PriorityQueue

def astar(start_node, goal_node, heuristic_func, neighbor_func):
    # Initialize the open set and closed set
    open_set = PriorityQueue()
    open_set.put((0, start_node))
    closed_set = set()
    # Initialize the g and h values of the start node
    g = {start_node: 0}
    h = {start_node: heuristic_func(start_node, goal_node)}

    while not open_set.empty():
        # Get the node with the lowest f-score
        _, current_node = open_set.get()

        # Check if we have reached the goal node
        if current_node == goal_node:
            path = []
            while current_node is not None:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]

        # Generate the successors of the current node
        for neighbor in neighbor_func(current_node):
            # Calculate the tentative g and h values of the neighbor node
            tentative_g = g[current_node] + neighbor.cost
            tentative_h = heuristic_func(neighbor, goal_node)

            # Check if the neighbor node is already in the closed set
            if neighbor in closed_set:
                continue
```

```
            # Check if the neighbor node is not in the open set or the new
 path to it is better than the old one
            if neighbor not in g or tentative_g < g[neighbor]:
                # Update the g and h values of the neighbor node
                g[neighbor] = tentative_g
                h[neighbor] = tentative_h
                # Add the neighbor node to the open set with the new f-
score
                f = tentative_g + tentative_h
                open_set.put((f, neighbor))
                # Set the parent of the neighbor node to the current node
                neighbor.parent = current_node

        # Add the current node to the closed set
        closed_set.add(current_node)

    # If we reach here, there is no path from the start node to the goal n
ode
    return None
```

Here's an explanation of the A* search algorithm implemented in Python code:

1. We first import the PriorityQueue class from the queue module, which we will use to store the nodes in the open set based on their f-score.

python

from queue import PriorityQueue

2. We define the astar function, which takes four arguments: start_node, goal_node, heuristic_func, and neighbor_func.

def astar(start_node, goal_node, heuristic_func, neighbor_func):

3. We initialize the open set as a PriorityQueue and add the starting node to it with a priority of 0. We also initialize the closed set as an empty set.

open_set = PriorityQueue()
open_set.put((0, start_node))
closed_set = set()

4. We initialize the g and h values of the starting node to 0 and the heuristic value between the starting and goal nodes, respectively.

```
g = {start_node: 0}
h = {start_node: heuristic_func(start_node, goal_node)}
```

5. We begin the main loop, which continues as long as there are nodes in the open set. In each iteration, we get the node with the lowest f-score from the open set.

```
while not open_set.empty():
    _, current_node = open_set.get()
```

6. We check if we have reached the goal node. If we have, we build and return the path from the starting node to the goal node by tracing back through each node's parent node.

```
if current_node == goal_node:
    path = []
    while current_node is not None:
        path.append(current_node)
        current_node = current_node.parent
    return path[::-1]
```

7. If we haven't reached the goal node, we generate the successor nodes of the current node using the neighbor_func function, and calculate their tentative g and h values based on the current node's g value and the cost of moving from the current node to each neighbor node, as well as the heuristic value between each neighbor node and the goal node.

```
for neighbor in neighbor_func(current_node):
    tentative_g = g[current_node] + neighbor.cost
    tentative_h = heuristic_func(neighbor, goal_node)
```

8. We check if the neighbor node is already in the closed set. If it is, we skip it and move on to the next neighbor node.

```
    if neighbor in closed_set:
        continue
```

9. We check if the neighbor node is not in the open set, or if the new path to it is better than the old one. If it is, we update the neighbor node's g and h values, add it to the open set with its new f-score, set its parent node to the current node, and move on to the next neighbor node.

```
    if neighbor not in g or tentative_g < g[neighbor]:
        g[neighbor] = tentative_g
        h[neighbor] = tentative_h
        f = tentative_g + tentative_h
        open_set.put((f, neighbor))
        neighbor.parent = current_node
```

10. We add the current node to the closed set.

    closed_set.add(current_node)

11. If we reach this point in the loop, it means that there is no path from the starting node to the goal node, so we return None.

    return None

12. Overall, this code implements the A* search algorithm by using a priority queue to store the open set of nodes

The most widely known form of best-first search is called **A*** **search**. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

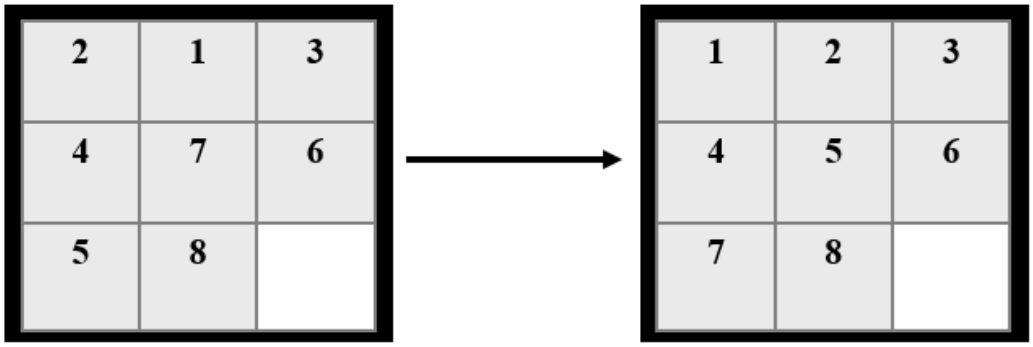
$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n. Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n).

1.4 Lab Tasks

Exercise 1.

Using state representation, goal_test() and operators from previous labs, implement Greedy Best First Search to solve *8-Puzzle* problem. You should write a function *GreedySearch(state)* that accepts any initial state and returns the result.



The first heuristic we consider is to count how many tiles are in the wrong place. We will call this heuristic, $h_1(n)$. An improved heuristic, $h_2(n)$, takes into account how far each tile had to move to get to its correct state. This is achieved by summing the **Manhattan distances** of each tile from its correct position. (Manhattan distance is the sum of the horizontal and vertical moves that need to be made to get from one position to another). Implement both the heuristic to solve the given 8-puzzle problem.

Exercise 2.

Solve above problem using A* Search algorithm.

Exercise 3.

Modify the code of Breadth First Search implemented in LAB 04 to Uniform Cost Search. Display the order in which nodes are added to the explored set, with **start state S** and **goal state G**. Path costs are mentioned on the arcs. Break ties in alphabetical order. What is the total cost of path found using UCS?