**CS3642-01 Programming Assignment #2 (Fall 2023)**

**Due: October 13, 2023 (11:30PM)**

**To implement Breadth-First Search (BFS), Uniform-Cost Search (UCS), Best-First Search (BFS), and A\* Algorithm to solve the following 8-puzzle problem (i.e. find the goal):**

**8-puzzle Problem:**

**The 8-puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Start with a random state (cannot be fixed). The goal state is listed below.**

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **8** |  | **4** |
| **7** | **6** | **5** |

**The program is to change the initial configuration into the goal configuration. A solution to the problem is an appropriate sequence of moves. You must write your own codes for the algorithms. Make sure your submission meets all of the requirements and free of plagiarism.**

**Your program should be able to address any initial configuration and provide a table of statistics below in your PDF file.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Average number of nodes visited** (you need repeat each algorithm several times with  different initial configuration) | **Average run time in your program** | **Your comment on these algorithms** |
| **Breadth First Search** |  |  |  |
| **Uniform Cost Search** |  |  |  |
| **Best First Search** |  |  |  |
| **A\*using Nilsson’s sequence score** |  |  |  |

You may write your code in a contemporary language of your choice; typical languages would include C/C++, Python, Java, Ada, Pascal, Smalltalk, Lisp, and Prolog. A GUI interface is preferred.

1. Submit a PDF file of your well-commented source program, your design and your printed outputs (screen shots). Please include your codes in your PDF file. It is plagiarism to take any codes from the website or others. Try to understand the algorithm and implement the algorithm by your own. You must have all the information required in your PDF file.
2. Provide a video presentation of your programming assignment in MP3, YouTube, or any media.
3. Please upload items 1) and 2) above separately to D2L.
4. Restriction: No zipped files.

Adding the following two sections (I and II) at the beginning of your PDF including your code and outputs.

1. Your Information:

|  |  |
| --- | --- |
| // Course: | Artificial Intelligence |
| // Student name: | Raehyeong Lee |
| // Student ID: | 000996758 |
| // Assignment #: | Assignmnet 2 |
| // Due Date: | October 13, 2023 |

// Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_(Your signature assures that everything is your own work. Required.)

// Score: \_\_\_\_\_\_\_\_\_\_\_\_\_\_(Note: Score will be posted on D2L)

1. The statistics table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Average number of nodes visited** (you need repeat each algorithm several times with  different initial configuration) | **Average run time in your experiments** | **Your comment on these algorithms** |
| **Breadth First Search** | **20+23+3+25+13+14=16.3** | **6** | **This must be the largest but the results was not working in right way.** |
| **Uniform Cost Search** | **20+23+3+25+13+14=16.3** | **6** | **It seems UCS and A\* algorithms are not operating properly. The result of the algorithms is always same with BFS above.** |
| **Best First Search** | **60+63+3+75+95+16=52** | **6** | **This algorithm must be shortest average but, in my program, it has the biggest average number of nodes moving.** |
| **A\*using Nilsson’s sequence score** | **20+23+3+25+13+14=16.3** | **6** | **It seems UCS and A\* algorithms are not operating properly. The result of the algorithms is always same with BFS above.** |

**1st I/O**

Initial State:

0 6 3

1 5 4

2 8 7

Breadth-First Search Result:

Number of moves: 20

Uniform-Cost Search Result:

Number of moves: 20

Best-First Search Result:

Number of moves: 60

A\* Algorithm Result:

Number of moves: 20

**2nd I/O**

Initial State:

1 6 7

0 5 3

2 8 4

Breadth-First Search Result:

Number of moves: 23

Uniform-Cost Search Result:

Number of moves: 23

Best-First Search Result:

Number of moves: 63

A\* Algorithm Result:

Number of moves: 23

**3rd I/O**

Initial State:

8 1 7

3 5 0

2 4 6

Breadth-First Search Result:

Number of moves: 25

Uniform-Cost Search Result:

Number of moves: 25

Best-First Search Result:

Number of moves: 75

A\* Algorithm Result:

Number of moves: 25

**4th I/O**

Initial State:

1 2 3

8 4 5

7 0 6

Breadth-First Search Result:

Number of moves: 3

Uniform-Cost Search Result:

Number of moves: 3

Best-First Search Result:

Number of moves: 3

A\* Algorithm Result:

Number of moves: 3

**5th I/O**

Initial State:

1 3 6

8 4 5

7 0 2

Breadth-First Search Result:

Number of moves: 13

Uniform-Cost Search Result:

Number of moves: 13

Best-First Search Result:

Number of moves: 95

A\* Algorithm Result:

Number of moves: 13

**6th I/O**

Initial State:

2 5 3

1 6 4

0 8 7

Breadth-First Search Result:

Number of moves: 14

Uniform-Cost Search Result:

Number of moves: 14

Best-First Search Result:

Number of moves: 16

A\* Algorithm Result:

Number of moves: 14

**Program Codes**

import heapq

import copy

#goal state

goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

#up, down, left, right

movements = [(-1, 0), (1, 0), (0, -1), (0, 1)]

#find the empty tile

def find\_empty\_tile(state):

    for i in range(3):

        for j in range(3):

            if state[i][j] == 0:

                return i, j

#check if a state is valid

def is\_valid(state):

    return all(0 <= state[i][j] < 9 for i in range(3) for j in range(3))

#print the puzzle state

def print\_state(state):

    for row in state:

        print(" ".join(map(str, row)))

    print()

def bfs(initial\_state):

    # Create a queue for BFS

    queue = [(initial\_state, [])]

    visited = set()

    while queue:

        state, path = queue.pop(0)

        if state == goal\_state:

            return path

        empty\_i, empty\_j = find\_empty\_tile(state)

        for move\_i, move\_j in movements:

            new\_i, new\_j = empty\_i + move\_i, empty\_j + move\_j

            if 0 <= new\_i < 3 and 0 <= new\_j < 3:

                new\_state = copy.deepcopy(state)

                new\_state[empty\_i][empty\_j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[empty\_i][empty\_j]

                if tuple(map(tuple, new\_state)) not in visited and is\_valid(new\_state):

                    queue.append((new\_state, path + [(new\_i, new\_j)]))

                    visited.add(tuple(map(tuple, new\_state)))

def ucs(initial\_state):

    priority\_queue = [(0, initial\_state, [])]

    visited = set()

    while priority\_queue:

        cost, state, path = heapq.heappop(priority\_queue)

        if state == goal\_state:

            return path

        empty\_i, empty\_j = find\_empty\_tile(state)

        for move\_i, move\_j in movements:

            new\_i, new\_j = empty\_i + move\_i, empty\_j + move\_j

            if 0 <= new\_i < 3 and 0 <= new\_j < 3:

                new\_state = copy.deepcopy(state)

                new\_state[empty\_i][empty\_j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[empty\_i][empty\_j]

                if tuple(map(tuple, new\_state)) not in visited and is\_valid(new\_state):

                    new\_cost = cost + 1

                    heapq.heappush(priority\_queue, (new\_cost, new\_state, path + [(new\_i, new\_j)]))

                    visited.add(tuple(map(tuple, new\_state)))

def heuristic(state):

    total\_distance = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != 0:

                target\_i, target\_j = divmod(state[i][j] - 1, 3)

                total\_distance += abs(target\_i - i) + abs(target\_j - j)

    return total\_distance

def best\_first\_search(initial\_state, heuristic):

    priority\_queue = [(heuristic(initial\_state), initial\_state, [])]

    visited = set()

    while priority\_queue:

        \_, state, path = heapq.heappop(priority\_queue)

        if state == goal\_state:

            return path

        empty\_i, empty\_j = find\_empty\_tile(state)

        for move\_i, move\_j in movements:

            new\_i, new\_j = empty\_i + move\_i, empty\_j + move\_j

            if 0 <= new\_i < 3 and 0 <= new\_j < 3:

                new\_state = copy.deepcopy(state)

                new\_state[empty\_i][empty\_j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[empty\_i][empty\_j]

                if tuple(map(tuple, new\_state)) not in visited and is\_valid(new\_state):

                    heapq.heappush(priority\_queue, (heuristic(new\_state), new\_state, path + [(new\_i, new\_j)]))

                    visited.add(tuple(map(tuple, new\_state)))

#Nilsson's sequence score

def nilsson\_sequence\_score(state):

    score = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != 0:

                target\_i, target\_j = divmod(state[i][j] - 1, 3)

                if (i, j) != (target\_i, target\_j):

                    score += 2

                    if (i, j) != (0, 0) and state[i][j] == goal\_state[i][j]:

                        score -= 2

                    elif (i, j) != (0, 1) and state[i][j] == goal\_state[i][j]:

                        score -= 2

    return score

def a\_star\_nilsson(initial\_state):

    priority\_queue = [(nilsson\_sequence\_score(initial\_state) + 0, initial\_state, [])]

    visited = set()

    while priority\_queue:

        \_, state, path = heapq.heappop(priority\_queue)

        if state == goal\_state:

            return path

        empty\_i, empty\_j = find\_empty\_tile(state)

        for move\_i, move\_j in movements:

            new\_i, new\_j = empty\_i + move\_i, empty\_j + move\_j

            if 0 <= new\_i < 3 and 0 <= new\_j < 3:

                new\_state = copy.deepcopy(state)

                new\_state[empty\_i][empty\_j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[empty\_i][empty\_j]

                if tuple(map(tuple, new\_state)) not in visited and is\_valid(new\_state):

                    new\_cost = len(path) + 1

                    heapq.heappush(priority\_queue, (new\_cost + nilsson\_sequence\_score(new\_state), new\_state, path + [(new\_i, new\_j)]))

                    visited.add(tuple(map(tuple, new\_state)))

#initial state

initial\_state = [[2, 5, 3], [1, 6, 4], [0, 8, 7]]

print("Initial State:")

print\_state(initial\_state)

print("Breadth-First Search Result:")

bfs\_result = bfs(initial\_state)

print("Number of moves:", len(bfs\_result))

for move in bfs\_result:

    print(move)

print()

print("Uniform-Cost Search Result:")

ucs\_result = ucs(initial\_state)

print("Number of moves:", len(ucs\_result))

for move in ucs\_result:

    print(move)

print()

print("Best-First Search Result:")

best\_first\_search\_result = best\_first\_search(initial\_state, heuristic)

print("Number of moves:", len(best\_first\_search\_result))

for move in best\_first\_search\_result:

    print(move)

print()

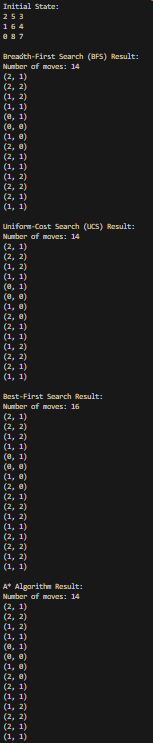
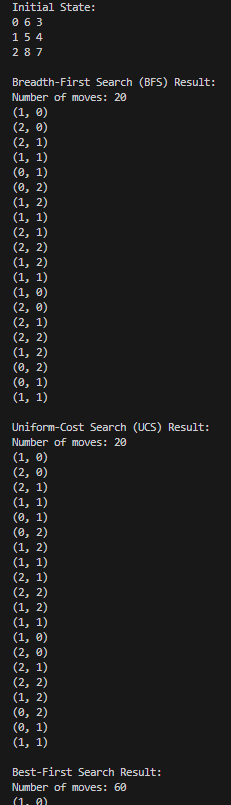
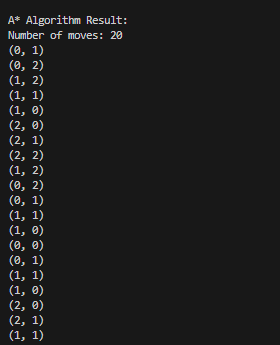
print("A\* Algorithm Result:")

a\_star\_result = a\_star\_nilsson(initial\_state)

print("Number of moves:", len(a\_star\_result))

for move in a\_star\_result:

    print(move)

**Screenshots of Output examples**