Dimieru Răzvan-Costinel

CEN 2.1B

AI - IA Homework Assignment 2023-2024

Homework Statement

The Traveling Salesman Problem (TSP) task is determining the optimal route through a set of cities, where the objective is to minimize the total distance traveled. Specifically, given a list of cities and the distances between each pair of cities, the goal is to find the route of minimum cost that visits each city exactly once and returns to the origin city. The cost function of the solution must be the minimization of the longest distance between two consecutive cities.

The objective of this homework is to implement and compare different search algorithms to solve the Traveling Salesman Problem (TSP). Specifically, we will implement Depth-First Search (DFS), Uniform Cost Search (UCS), and A* Search. The TSP involves finding the shortest possible route that visits each city exactly once and returns to the origin city.

Pseudocode for Depth-First Search (DFS)

```
function DFS-Agent(start, N, distances) returns an action
  persistent: seq, an action sequence, initially empty
  persistent: state, the current node, initially start
  persistent: goal, a goal state, initially null
  persistent: problem, the TSP problem formulation
  persistent: visited, an array to track visited nodes
  persistent: min_cost, the minimum cost found, initially infinity

  function Update-State(state, percept)
    return percept

  function Formulate-Goal(state)
    return "visit all cities and return to start"

  function Formulate-Problem(state, goal)
    return (start, N, distances)

  function Search(problem) returns a sequence
    function DFS(node, visited, cost)
      if all nodes are visited then
        return cost + distances[node][start]
      min_cost = infinity
      for each neighbor in neighbors(node) do
        if neighbor is not visited then
          visited[neighbor] = true
          new_cost = DFS(neighbor, visited, cost + distances[node][neighbor])
          min_cost = min(min_cost, new_cost)
```

```
        visited[neighbor] = false
      return min_cost

    return DFS(start, visited, 0)

  state ← Update-State(state, start)
  if seq is empty then
    goal ← Formulate-Goal(state)
    problem ← Formulate-Problem(state, goal)
    seq ← Search(problem)
    if seq = failure then
      return null
  action ← First(seq)
  seq ← Rest(seq)
  return action
```

Pseudocode for Uniform Cost Search (UCS)

```
function UCS-Agent(start, N, distances) returns an action
  persistent: seq, an action sequence, initially empty
  persistent: state, the current node, initially start
  persistent: goal, a goal state, initially null
  persistent: problem, the TSP problem formulation
  persistent: min_cost, the minimum cost found, initially infinity

  function Update-State(state, percept)
    return percept
```

```
function Formulate-Goal(state)
  return "visit all cities and return to start"

function Formulate-Problem(state, goal)
  return (start, N, distances)

function Search(problem) returns a sequence
  min_cost = 1000000
  stack = malloc(N * N * sizeof(int*))
  stack_cost = malloc(N * N * sizeof(int))
  stack_size = malloc(N * N * sizeof(int))
  stack_top = 0

  for i = 0 to N-1 do
    if i ≠ start then
      stack[stack_top] = malloc(N * sizeof(int))
      stack[stack_top][0] = i
      stack_cost[stack_top] = distances[start][i]
      stack_size[stack_top++] = 1

  while stack_top > 0 do
    stack_top--
    temp_path = stack[stack_top]
    temp_cost = stack_cost[stack_top]
    temp_size = stack_size[stack_top]

    if temp_size = N and distances[temp_path[temp_size - 1]][start] then
      min_cost = min(min_cost, temp_cost + distances[temp_path[temp_size - 
1]][start])
      free(temp_path)
```

```
        continue

      for i = 0 to N-1 do
        in_path = false
        for j = 0 to temp_size-1 do
          if temp_path[j] = i then
            in_path = true
            break
        if not in_path and distances[temp_path[temp_size - 1]][i] then
          stack[stack_top] = malloc(N * sizeof(int))
          copy(temp_path, temp_path + temp_size, stack[stack_top])
          stack[stack_top][temp_size] = i
          stack_cost[stack_top] = temp_cost + distances[temp_path[temp_size - 1]][i]
          stack_size[stack_top++] = temp_size + 1
      free(temp_path)

    free(stack)
    free(stack_cost)
    free(stack_size)
    return min_cost

  state ← Update-State(state, start)
  if seq is empty then
    goal ← Formulate-Goal(state)
    problem ← Formulate-Problem(state, goal)
    seq ← Search(problem)
    if seq = failure then
      return null
  action ← First(seq)
```

```
  seq ← Rest(seq)
  return action
```

A* Search Pseudocode*

```
function heuristic(path, path_size, N, distances):
  h = 0
  in_path = array of size N initialized to false

  for each city in path:
    in_path[city] = true

  for i from 0 to N-1:
    if not in_path[i]:
      min_cost = infinity
      for j from 0 to N-1:
        if i ≠ j and distances[i][j] < min_cost:
          min_cost = distances[i][j]
      h += min_cost

  return h

function aStarSearch(start, N, distances):
  min_cost = infinity

  structure Node:
    city
```

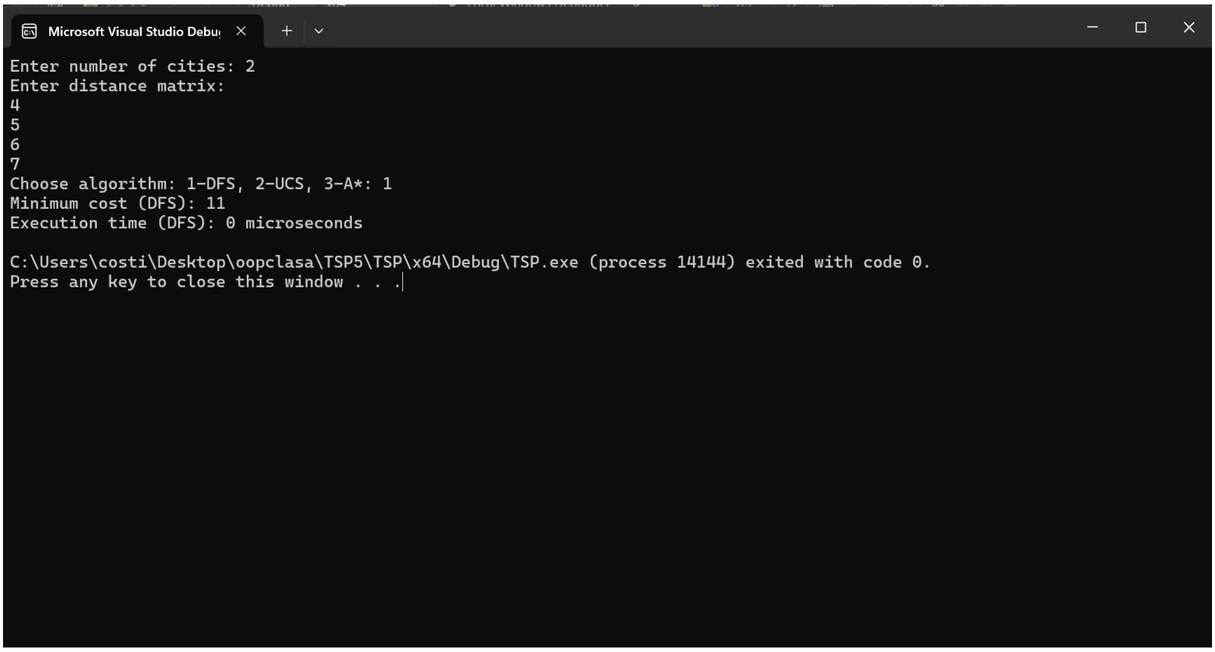
```
    cost
    heuristic
    path array of size 20
    path_size

    Node():
        city = 0
        cost = 0
        heuristic = 0
        path = array of size 20 filled with zeros
        path_size = 0

    Node(c, cst, h, p, p_size):
        city = c
        cost = cst
        heuristic = h
        path = copy of p
        path_size = p_size

stack = array of Nodes of size N * N
stack_top = 0

initial = Node()
initial.city = start
initial.heuristic = heuristic(initial.path, initial.path_size, N, distances)
initial.path[0] = start
initial.path_size = 1
stack[stack_top] = initial
stack_top++
```

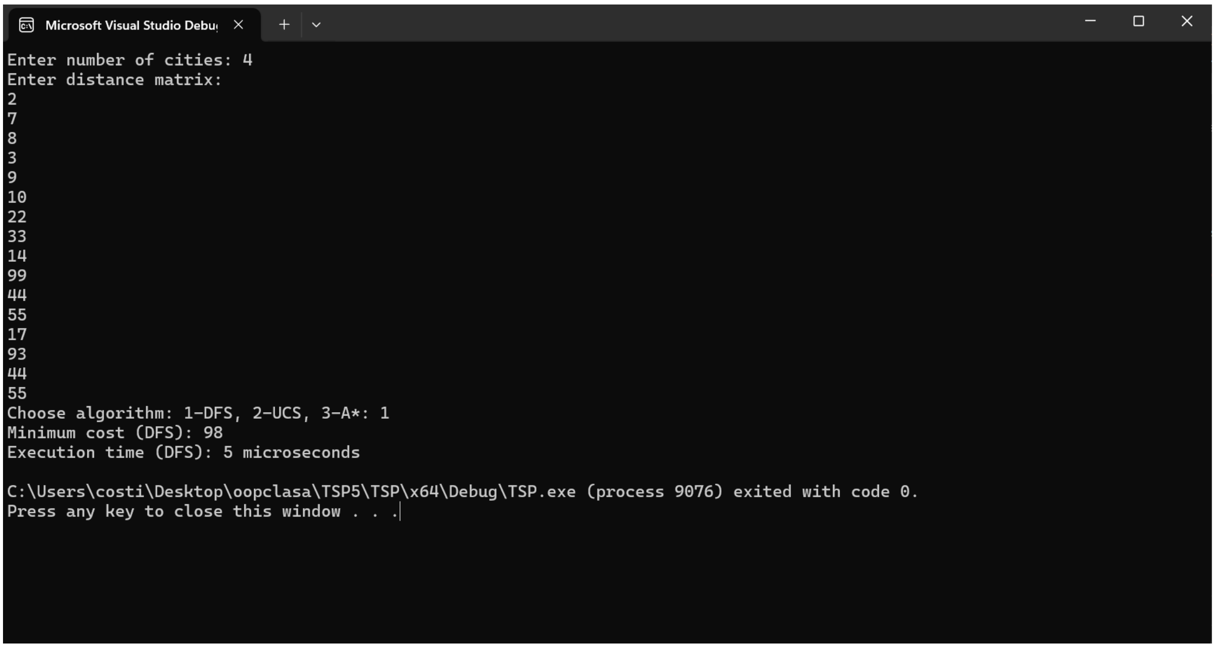
```
  while stack_top > 0:
    node = stack[stack_top - 1]
    stack_top--

    if node.path_size == N and distances[node.city][start]:
      min_cost = min(min_cost, node.cost + distances[node.city][start])
      continue

    for i from 0 to N-1:
      in_path = false
      for each city in node.path:
        if city == i:
          in_path = true
          break

      if not in_path and distances[node.city][i]:
        new_node = Node()
        new_node.city = i
        new_node.cost = node.cost + distances[node.city][i]
        new_node.path_size = node.path_size + 1
        new_node.path = copy of node.path
        new_node.path[new_node.path_size] = i
        new_node.heuristic = heuristic(new_node.path, new_node.path_size, N,
distances)
        stack[stack_top] = new_node
        stack_top++

  return min_cost
```

The application is designed to solve the Traveling Salesman Problem (TSP) using three different strategies: Depth-First Search (DFS), Least-Cost Search (UCS), and A* Search. Each strategy is implemented as a separate module, allowing users to compare the performance of these strategies in solving the TSP.

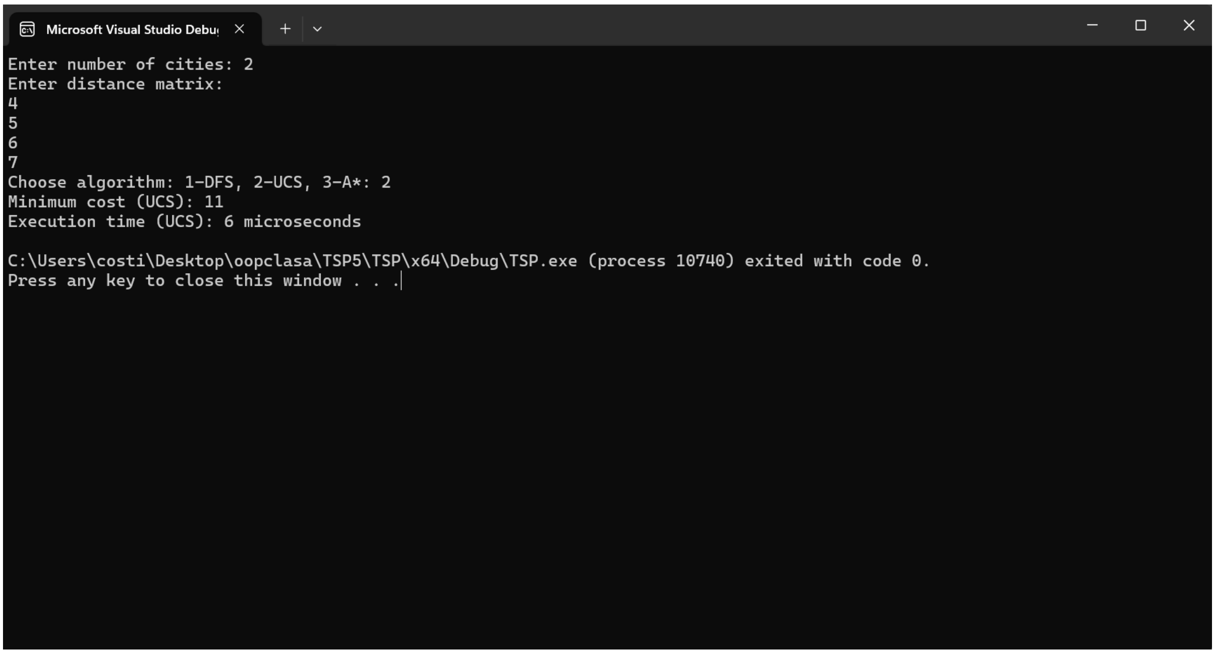
For the input, I used two matrices, one 2x2 and the other 5x5, with random values. The same matrices were used for all three algorithms. Each alogithm displays the same cost value but there is a problem with the execution time, it takes too long, for A*.

DFS:



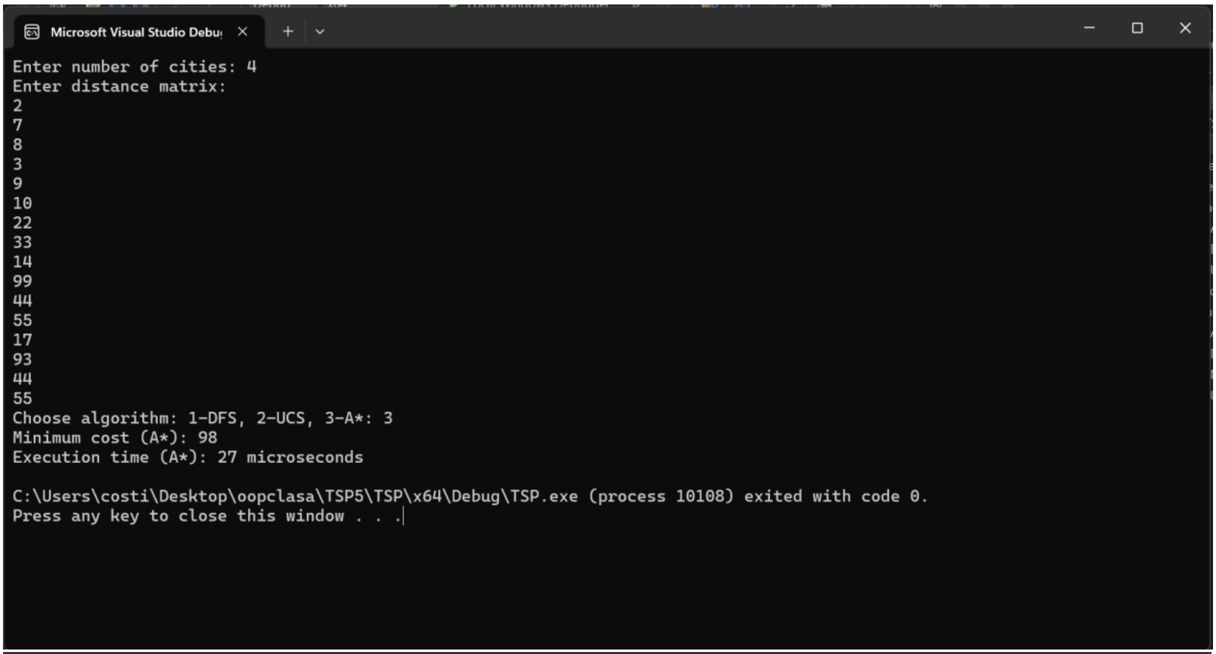

Microsoft Visual Studio Debu
Enter number of cities: 4
Enter distance matrix:
2
7
8
3
9
10
22
33
14
99
44
55
17
93
44
55
Choose algorithm: 1-DFS, 2-UCS, 3-A*: 1
Minimum cost (DFS): 98
Execution time (DFS): 5 microseconds
C:\Users\costi\Desktop\oopclasa\TSP5\TSP\x64\Debug\TSP.exe (process 9076) exited with code 0.
Press any key to close this window . . .

UCS

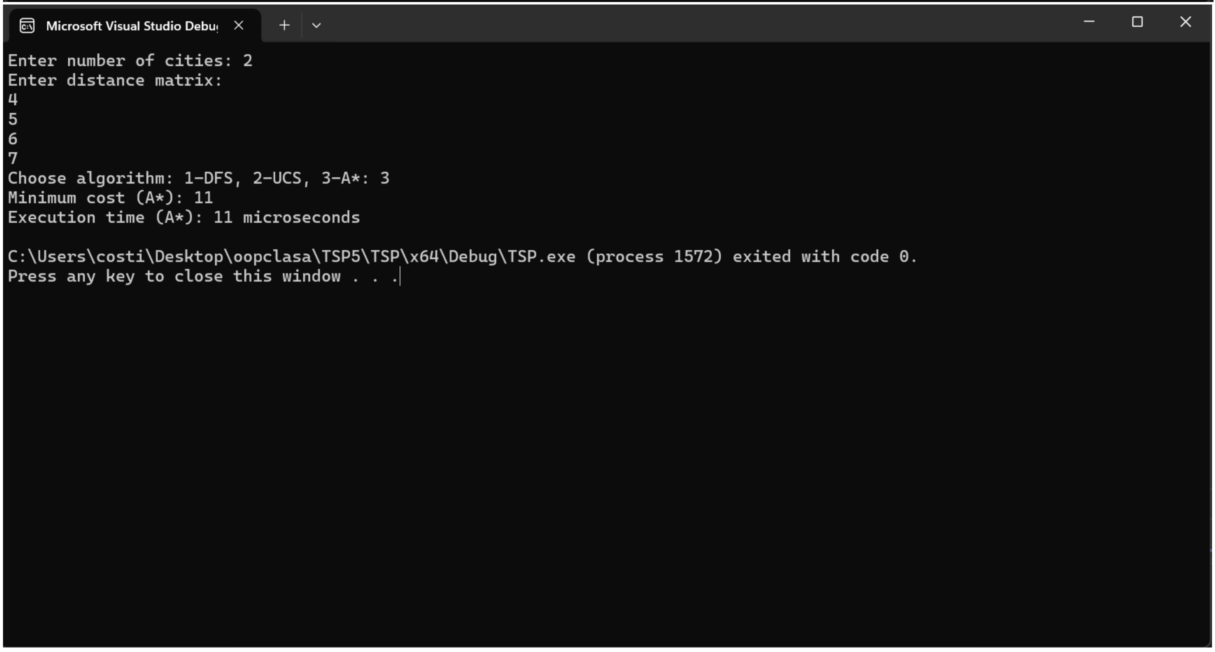

Microsoft Visual Studio Debu
Enter number of cities: 2
Enter distance matrix:
4
5
6
7
Choose algorithm: 1-DFS, 2-UCS, 3-A*: 2
Minimum cost (UCS): 11
Execution time (UCS): 6 microseconds
C:\Users\costi\Desktop\oopclasa\TSP5\TSP\x64\Debug\TSP.exe (process 10740) exited with code 0.
Press any key to close this window . . .

```
Enter number of cities: 4
Enter distance matrix:
2
7
8
3
9
10
22
33
14
99
44
55
17
93
44
55
Choose algorithm: 1-DFS, 2-UCS, 3-A*: 2
Minimum cost (UCS): 98
Execution time (UCS): 23 microseconds

C:\Users\costi\Desktop\oopclasa\TSP5\TSP\x64\Debug\TSP.exe (process 16012) exited with code 0.
Press any key to close this window . . .
```

A*



```
Enter number of cities: 4
Enter distance matrix:
2
7
8
3
9
10
22
33
14
99
44
55
17
93
44
55
Choose algorithm: 1-DFS, 2-UCS, 3-A*: 3
Minimum cost (A*): 98
Execution time (A*): 27 microseconds

C:\Users\costi\Desktop\oopclasa\TSP5\TSP\x64\Debug\TSP.exe (process 10108) exited with code 0.
Press any key to close this window . . .
```



```
Enter number of cities: 2
Enter distance matrix:
4
5
6
7
Choose algorithm: 1-DFS, 2-UCS, 3-A*: 3
Minimum cost (A*): 11
Execution time (A*): 11 microseconds

C:\Users\costi\Desktop\oopclasa\TSP5\TSP\x64\Debug\TSP.exe (process 1572) exited with code 0.
Press any key to close this window . . .
```

The output data format includes the minimum cost required to visit all cities exactly once and return to the origin city. For each strategy (DFS, UCS, A*), the minimum cost is printed to the console and the execution time too.

List of Modules:

1. Depth-First Search (DFS)
2. Least-Cost Search (UCS)
3. A* Search

Module Descriptions:

1. **Depth-First Search (DFS):**
    - This module implements the Depth-First Search algorithm for solving the TSP.
    - Functions:
        - **DFS**: Performs Depth-First Search to find the minimum cost path.
            - Parameters:
                - **start**: The starting city index.
                - **N**: The total number of cities.
                - **distances**: The distance matrix representing distances between cities.
            - Return Value: The minimum cost required to visit all cities and return to the origin.
2. Least-Cost Search (UCS):
    - This module implements the Least-Cost (Uniform Cost) Search algorithm for solving the TSP.
    - Functions:
        - **uniformCostSearch**: Performs Uniform Cost Search to find the minimum cost path.
            - Parameters:
                - **start**: The starting city index.
                - **N**: The total number of cities.
                - **distances**: The distance matrix representing distances between cities.
            - Return Value: The minimum cost required to visit all cities and return to the origin.
3. *A Search:**
    - This module implements the A* Search algorithm for solving the TSP.

- Functions:
    - **aStarSearch**: Performs A* Search to find the minimum cost path.
        - Parameters:
            - **start**: The starting city index.
            - **N**: The total number of cities.
            - **distances**: The distance matrix representing distances between cities.
        - Return Value: The minimum cost required to visit all cities and return to the origin.

In conclusion, I can say that implementing the three algorithms was relatively difficult, and additionally, the algorithms have significant memory issues, with the execution time being poor, especially for A*, which should be the fastest of the three but is currently the slowest. I have failed to make the algorithms work in as short a time as possible.

References

- Russell, S., & Norvig, P. (2009). Artificial Intelligence: A Modern Approach (3rd ed.). Pearson.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.