CZ3005 Artificial Intelligence
FS5 Lab #1 Report

Prepared by:
Joceline Yi U1920057J
Siow Kee Tat Keith U1922767C
Tey Chin Yi U1920268L

Task 1: Solving a relaxed version of the NYC instance where we do not have energy constraints.

Algorithm used: Dijkstra's shortest path algorithm

This algorithm finds distance from source node to all valid nodes in the graph. Using a priority queue, the nodes are being weighed by their distance from source upon each node expansion. The node with the smallest distance from source will always expand first. If the new distance from source is smaller than the stored distance from source, the distance of the child node will be updated. After the comparison, the child node will be inserted into the priority queue. Node expansion will stop when the priority queue is empty.

Algorithm Analysis

Time	Space	Optimal	Complete
O(ElogV)	$O(V)$	Yes	Yes

Pseudocode

```
function dijkstra_algorithm(source, goal, energy_constraint) returns a solution
  visited_node ← an empty set
  distance ← {source:0}
  parent_vertex ← {source:None}
  priorityQueue ← a priority queue ordered by distance_from_source, with node as the
                  only element
  priorityQueue ← INSERT((distance_from_source, node)) for source

  loop do
     if EMPTY?(priorityQueue) then return
     distance_from_source, node ← POP(priorityQueue)
       if current not in visited_node then
         visited ← INSERT(current)
       if current is goal then
         break
     for each neighbour in neighbours do
       old_cost ← distance[neighbour]
       new_cost ← distance[current] + dists[current,neighbour]
       if new_cost , old_cost then
         priorityQueue ← INSERT((new_cost, neighbour))
         distance[neighbour] ← new_cost
         parent_vertex[neighbour] ← current
      if goal not in parent_vertex.keys then
        return None
      else then
        return parent_vertex, distance
```

Output

The findings of our search algorithm are shown below, with the shortest path found to have a distance of 148648.

```
Shortest path:

1 -> 1363 -> 1358 -> 1357 -> 1356 -> 1276 -> 1273 -> 1277 -> 1269 -> 1267 -> 1268 -> 1284 -> 1283 -> 1282 -> 1255 -> 1253 ->
1260 -> 1259 -> 1249 -> 1246 -> 963 -> 964 -> 962 -> 1002 -> 952 -> 1000 -> 998 -> 994 -> 995 -> 996 -> 987 -> 988 -> 979 ->
980 -> 969 -> 977 -> 989 -> 990 -> 991 -> 2369 -> 2366 -> 2340 -> 2338 -> 2339 -> 2333 -> 2334 -> 2329 -> 2029 -> 2027 -> 201
9 -> 2022 -> 2000 -> 1996 -> 1997 -> 1993 -> 1992 -> 1989 -> 1984 -> 2001 -> 1900 -> 1875 -> 1874 -> 1965 -> 1963 -> 1964 ->
1923 -> 1944 -> 1945 -> 1938 -> 1937 -> 1939 -> 1935 -> 1931 -> 1934 -> 1673 -> 1675 -> 1674 -> 1837 -> 1671 -> 1828 -> 1825
-> 1817 -> 1815 -> 1634 -> 1814 -> 1813 -> 1632 -> 1631 -> 1742 -> 1741 -> 1740 -> 1739 -> 1591 -> 1689 -> 1585 -> 1584 -> 16
88 -> 1579 -> 1679 -> 1677 -> 104 -> 5680 -> 5418 -> 5431 -> 5425 -> 5424 -> 5422 -> 5413 -> 5412 -> 5411 -> 66 -> 5392 -> 53
91 -> 5388 -> 5291 -> 5278 -> 5289 -> 5290 -> 5283 -> 5284 -> 5280 -> 50
Shortest distance: 148648.63722140007
Total energy cost:  294853
Time taken:     0.10970544815063477 seconds
```

Task 2: Implement an uninformed search algorithm to solve the NYC instance.

Algorithm used: Uniform cost search (UCS) with energy constraints

Instead of using a First-In-First-Out (FIFO) queue used in breadth-first search algorithms, a priority queue is used instead. In the UCS algorithm, we consider the cost of the edges (in this case, the distance of the path as well as the energy cost) and the expansion of nodes occurs with the least path cost, *g*. The *UCS()* method implements the search algorithm and generates the shortest path by inserting the nodes that have the least path cost into the priority queue. The algorithm then returns the first minimum path that meets the energy constraint.

Pseudocode

```
function UCS_search(source, goal, energy_constraint) returns a solution
  visited_node ← an empty set
  priorityQueue ← a priority queue order by distance_from_curent with cost_from_current,
                  current, path as the only three elements
  priorityQueue ← INSERT((distance_from_current, cost_from_current, source, path)) for source
  total_cost ← 0
  total_distance ← 0
  min_cost ← 100000000
  min_distance ← 100000000
  min_path ← an empty set

  loop do
     if EMPTY?(priorityQueue) then return
     distance_from_current, cost_from_current, current , current_path ← POP(priorityQueue)
       if current not in visited_node then
         visited ← INSERT(current)
       if current is goal then
         if CALCULATE_COST(current_path) > energy_constraint then
           visited_node ← REMOVE(current)
         else
            if min_distance >= CALCULATE_DISTANCE(current_path) then
              min_path = current_path
              min_distance = distance_from_current
            visited ← REMOVE(current)
       else
         neighbours ← graph.current
         for each neighbour in neighbours do
           neighbour_distance ← distance[current,neighbour]
           neighbour_cost ← cost[current,neighbour]
           neighbour_path ← current_path
           neightbour_path ← INSERT(neighbour)
           if cost_to_neighbour + neighbour_cost <= energy_constraint then
             priorityQueue ← INSERT((neighbour_distance + distance_from_current,
                                     neighbour_cost + cost_from_current, neighbour, neighbour_path))

  return min_path
```

Algorithm Analysis

Time	Space	Optimal	Complete
$O(b^d)$	$O(b^d)$	Yes	Yes

where b denotes the maximum branching factor of the search tree and d denotes the depth of the proposed least cost solution.

Output

The findings of our search algorithm are shown below, with the shortest path found to have a distance of 150784 and a total energy cost of 287931, which satisfies the energy constraint given in the problem statement.

```
Shortest path:

1 -> 1363 -> 1358 -> 1357 -> 1356 -> 1276 -> 1273 -> 1277 -> 1269 -> 1267 -> 1268 -> 1284 -> 1283 -> 1282 -> 1255 -> 1253 ->
1260 -> 1259 -> 1249 -> 1246 -> 963 -> 964 -> 962 -> 1002 -> 952 -> 1000 -> 998 -> 994 -> 995 -> 996 -> 987 -> 986 -> 979 ->
980 -> 969 -> 977 -> 989 -> 990 -> 991 -> 2369 -> 2366 -> 2340 -> 2338 -> 2339 -> 2333 -> 2334 -> 2329 -> 2029 -> 2027 -> 201
9 -> 2022 -> 2000 -> 1996 -> 1997 -> 1993 -> 1992 -> 1989 -> 1984 -> 2001 -> 1900 -> 1875 -> 1874 -> 1965 -> 1963 -> 1964 ->
1923 -> 1944 -> 1945 -> 1938 -> 1937 -> 1939 -> 1935 -> 1931 -> 1934 -> 1673 -> 1675 -> 1674 -> 1837 -> 1671 -> 1828 -> 1825
-> 1817 -> 1815 -> 1634 -> 1814 -> 1813 -> 1632 -> 1631 -> 1742 -> 1741 -> 1740 -> 1739 -> 1591 -> 1689 -> 1585 -> 1584 -> 16
88 -> 1579 -> 1679 -> 1677 -> 104 -> 5680 -> 5418 -> 5431 -> 5425 -> 5429 -> 5426 -> 5428 -> 5434 -> 5435 -> 5433 -> 5436 ->
5398 -> 5404 -> 5402 -> 5396 -> 5395 -> 5292 -> 5282 -> 5283 -> 5284 -> 5280 -> 50
Shortest distance:  150784.60722193593
Total energy cost:  287931
Time taken:     1.4351942539215088 seconds
```

Task 3: Develop an A* search algorithm to solve the NYC instance.

Algorithm used: A* search algorithm with energy constraints

For task 3, we used A* search algorithm which is a combination of Greedy search with Uniform-Cost Search. Instead of comparing just g(n) cost of path to node n, we compare f(n) instead, which includes heuristic cost.

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

where f(n) is the estimated total cost of path through node n to goal state,

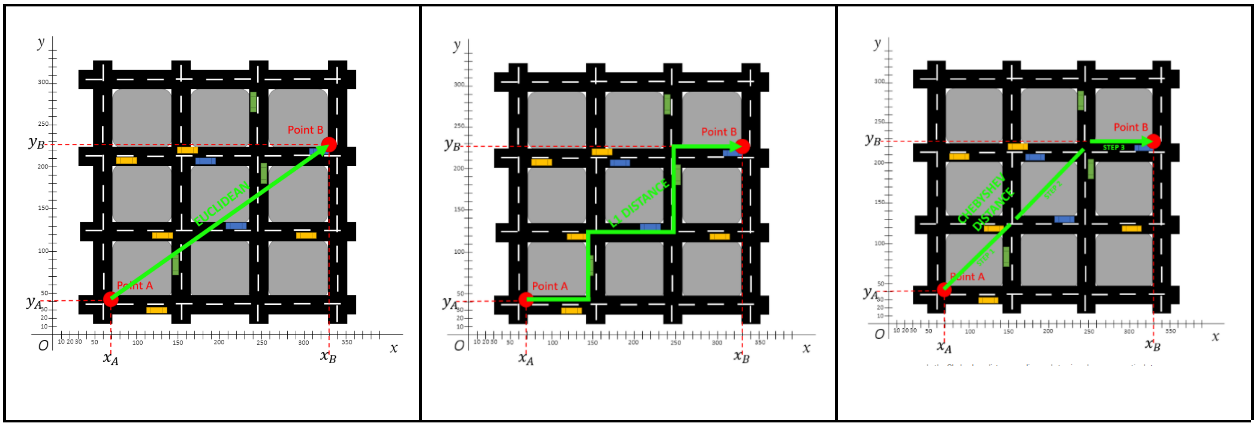
g(n) is the cost of path to node n and h(n) is the heuristic cost of node n

Since we are dealing with low dimensional data (coordinates), we decided to go with Euclidean Distance since it is highly intuitive and simple to implement. (Too complex heuristic may not necessarily be effective)

h(n) = euclidean distance between node n and goal node (calculated using Pythagorean theorem)

There are 3 different methods when trying to calculate an approximation for heuristic cost h(n):

Euclidean Distance	L1 Distance	Chebyshev Distance



Pseudo-code

```
function A*_search(source, goal, energy_constraint) returns a solution
  visited_node ← an empty set
  priorityQueue ← a priority queue order by estimated_total_cost of path f(n),
                with node and path as only two elements
  priorityQueue ← INSERT((current_distance, source, path)) for source
  min_distance ← 100000000
  min_path ← an empty set
  target_x, target_y ← coords[target]

  loop do
     if EMPTY?(priorityQueue) then return
     estimated_total_cost, current, current_path ← POP(priorityQueue)
     current_distance ← CALCULATE_DISTANCE(current_path)
     current_cost ← CALCULATE_COST(current_path)
       if current not in visited_node then
         visited ← INSERT(current)
       if current is goal then
         if current_cost > energy_constraint then
           visited_node ← REMOVE(current)
         else
            if min_distance >= current_distance then
              min_path = current_path
              min_distance = current_distance
            visited ← REMOVE(current)
       else
         neighbours ← graph.current
         for each neighbour in neighbours do
           neighbour_distance ← distance[current,neighbour]
           neighbour_cost ← cost[current,neighbour]
           neighbour_path ← current_path
           neightbour_path ← INSERT(neighbour)
           distance_to_neighbour ← current_distance + neighbour_distance
           cost_to_neighbour ← current_cost + neighbour_cost
           neighbour_x, neighbour_y ← coords[neighbour]
           euclidean_distance ← SQRT((neighbour_x-target_x)**2-(neighbour_y-target_y)**2)
           if cost_to_neighbour <= energy_constraint then
             priorityQueue ← INSERT((distance_to_neighbour+euclidean_distance, neighbour, neighbour_path))

  return min_path
```

Code analysis

Time	Space	Optimal	Complete
$O(b^d)$	$O(b^d)$	Yes	Yes

where b denotes the maximum branching factor of the search tree and d denotes the depth of the proposed least cost solution.

<u>Output</u>

The findings of our search algorithm are shown below, with the shortest path found to have a distance of 150784 and a total energy cost of 287931, which satisfies the energy constraint given in the problem statement.

```
Shortest path:

1 -> 1363 -> 1358 -> 1357 -> 1356 -> 1276 -> 1273 -> 1277 -> 1269 -> 1267 -> 1268 -> 1284 -> 1283 -> 1282 -> 1255 -> 1253 ->
1260 -> 1259 -> 1249 -> 1246 -> 963 -> 964 -> 962 -> 1002 -> 952 -> 1000 -> 998 -> 994 -> 995 -> 996 -> 987 -> 986 -> 979 ->
980 -> 969 -> 977 -> 989 -> 990 -> 991 -> 2369 -> 2366 -> 2340 -> 2338 -> 2339 -> 2333 -> 2334 -> 2329 -> 2029 -> 2027 -> 201
9 -> 2022 -> 2000 -> 1996 -> 1997 -> 1993 -> 1992 -> 1989 -> 1984 -> 2001 -> 1900 -> 1875 -> 1874 -> 1965 -> 1963 -> 1964 ->
1923 -> 1944 -> 1945 -> 1938 -> 1937 -> 1939 -> 1935 -> 1931 -> 1934 -> 1673 -> 1675 -> 1674 -> 1837 -> 1671 -> 1828 -> 1825
-> 1817 -> 1815 -> 1634 -> 1814 -> 1813 -> 1632 -> 1631 -> 1742 -> 1741 -> 1740 -> 1739 -> 1591 -> 1689 -> 1585 -> 1584 -> 16
88 -> 1579 -> 1679 -> 1677 -> 104 -> 5680 -> 5418 -> 5431 -> 5425 -> 5429 -> 5426 -> 5428 -> 5434 -> 5435 -> 5433 -> 5436 ->
5398 -> 5404 -> 5402 -> 5396 -> 5395 -> 5292 -> 5282 -> 5283 -> 5284 -> 5280 -> 50
Shortest distance:  150784.60722193593
Total energy cost:  287931
Time taken:     2.1938633918762207 seconds
```

Learning points to take away:

Through this lab assignment, we were able to learn, implement and explore three different searching algorithms. Dijkstra's shortest path algorithm, Uniform Cost Search and A* Search. Our biggest takeaway would be implementing UCS and A* search algorithm, since one is uninformed and another is informed search respectively. Although A* is able to find the solution more quickly (according to empirical analysis like time and number of nodes visited), its solution may be incomplete if heuristic functions are ineffective. We learned various heuristic functions that can be used in estimation of distance from the node to the goal/target node. Moreover, with the additional layer of criteria cost being involved, it deepens our understanding of the node traversal process as it is an important step when we want the algorithms to consider both distance and cost of paths.

Contribution	
Task	Contributions
#1 Dijkstra algorithm	Keith, rest - refining and debugging
#2 Uniform Cost Search	Joceline, rest - refining and debugging
#3 A* Search Algorithm	Chin Yi, rest - refining and debugging
Lab Report	Everyone

References:

Geeksforgeeks, 2021, A* Algorithm , https://www.geeksforgeeks.org/a-search-algorithm/

Medium, 2019, Different Types of Distance Metrics used in Machine Learning ,https://medium.com/@kunal_gohrani/different-types-of-distance-metrics-used-in-machine-learning-e9928c5e26c7