

Department: ENCS - Computer Systems Engineering - هندسة أنظمة الحاسوب

ENCS3340 – Artificial Intelligence

**Search Algorithms for Route Navigation**

1191590 – Tala Dweikat

1190102 – Mazen Batrawi

Instructor: Dr. Yazan Abu Farha

Section: 4

Date: 15 – 05 – 2022

Table of contents

[Assumptions 1](#_Toc103717443)

[Program Implementation 2](#_Toc103717444)

[LANGUAGE USED 2](#_Toc103717445)

[MAIN CLASS 2](#_Toc103717446)

[FILE IMPLEMENTATION 2](#_Toc103717447)

[INTERFACE 2](#_Toc103717448)

[BFS Algorithm 3](#_Toc103717449)

[GENERAL INFORMATION ABOUT THE ALGORITHM 3](#_Toc103717450)

[ALGORITHM IMPLEMENTATION 3](#_Toc103717451)

[Uniform Cost Algorithm 4](#_Toc103717452)

[GENERAL INFORMATION ABOUT THE ALGORITHM 4](#_Toc103717453)

[ALGORITHM IMPLEMENTATION 4](#_Toc103717454)

[A\* Searching Algorithm 6](#_Toc103717455)

[GENERAL INFORMATION ABOUT THE ALGORITHM 6](#_Toc103717456)

[ALGORITHM IMPLEMENTATION 6](#_Toc103717457)

[File Implementation 8](#_Toc103717458)

[Home Page UI 9](#_Toc103717460)

[Breadth First Search Run 10](#_Toc103717462)

[Breadth First Search Information Page 11](#_Toc103717464)

[Uniform Cost Run 12](#_Toc103717466)

[Uniform Cost Information Page 13](#_Toc103717468)

[A\* Search Run 14](#_Toc103717470)

[- First method for A\* search 14](#_Toc103717471)

[- Second method for A\* search 15](#_Toc103717473)

[A\* Search Information Page 16](#_Toc103717475)

**Table of figures**

[Figure 1: Input file. 8](#_Toc103548755)

[Figure 2: Home Page. 9](#_Toc103548757)

[Figure 3: BFS Run Results. 10](#_Toc103548759)

[Figure 4: Information About BFS Page. 11](#_Toc103548760)

[Figure 5: Uniform Cost Results. 12](#_Toc103548762)

[Figure 6: Information About Uniform Cost Page. 13](#_Toc103548763)

[Figure 7: First A\* Implementation Run. 14](#_Toc103548765)

[Figure 8: Second A\* Implementation Run. 15](#_Toc103548766)

[Figure 9: Information About A\* Page. 16](#_Toc103548767)

# Assumptions

Before we begin, the following assumptions were taken:

1. The number of cities is defined in the code in the class Main.java, which is denoted by the variable N, you can change it depending on the number of the cities in the input file. If the number is not known, you can assume a large number like 100 or 200.
2. The input file name is nodes.txt
3. The package follows the following directory: src/sample. Adj.get(node).get(i) means we are accessing the ith child of the node (node). Adj.get(node).get(i).get(1) means we are accessing the car distance between the node and its ith child, Adj.get(node).get(i).get(2) for the walking distance, Adj.get(node).get(i).get(0) for the aerial distance.
4. Uniform cost and BFS were implemented to calculate the car distance, it can be changed to any distance using the criteria given above. Adj.get(node).get(i).get(0) is the id of the ith ­child of node (node).
5. A\* searching algorithm was implemented in two methods:

* First method: path cost = walking distance, heuristic = straight line distance.
* Second method: path cost = car distance, heuristic = walking distance estimated from the first method.

# Program Implementation

## **LANGUAGE USED**

* Java was used to implement the program using IntelliJ, and Scene Builder was used for creating the user interface with the help of FXML.

## **MAIN CLASS**

* We created the main class (Main.java) which is responsible for taking the input from the user and running the algorithms and reading the file.

## **FILE IMPLEMENTATION**

* The file follows the following style: City1 City2 CarDist WalkDist AirDist; which means that there is an edge between City1 and City2, and the distances are specified for this edge, the edge is bidirectional. An id was given for each city and was stored in a hash map. To store the graph, we used an adjacency list, where it is initialized with empty lists with the number of the cities. If city1 and city2 has an edge between them, then the adjacency list at the index of the id of the first city will have the information about this edge, and city2 will also have the information because the edge is bidirectional.
* Example:

if at index 1 of the adjacency list we had the following structure: {{2, 100, 90, 80}, {3, 50, 45, 40}}, this means that city 1 is connected to city 2 and city 3, with the specified distances for each edge, and city 2 and city 3 will have the same information about city 1 (Bidirectional).

## **INTERFACE**

* The main interface has a map with radio buttons on it, each for a city, the user selects a radio button and then presses the button for selecting the start to save the start city, and makes the same operation for the goal city. Then he selects the algorithm he would like to run and the program takes care of the rest. The Main class calls the class where the algorithms were implemented (Algorithms.java), and sends the needed information for the specified algorithm’s function. After running the algorithm, the function returns the information from the run. Then the Main class shows the results of running the algorithm, i.e., start and end cities, the distance, the path, and some information about the algorithm.

# BFS Algorithm

## **GENERAL INFORMATION ABOUT THE ALGORITHM**

* BFS algorithm is an uninformed searching algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.
* Idea: Expand shallowest unexpanded node.
* Uses a FIFO queue.
* Properties:

1) Complete for finite branching factor b

2) Optimal if cost = 1 per step

3) Space complexity: O(b^d)

4) Time complexity: O(b^d)

## **ALGORITHM IMPLEMENTATION**

* We used a queue (linked list) for running the breadth first search algorithm, because it follows the principle of FIFO (first in first out).
* BFS starts by initializing the parent of every city to -1 (has no parent) and the cost for every city to infinity and visited for each city to false (visited means generated not expanded).

* The queue stores a pair of integers, the second in the pair is the node we are at, and the first in the pair is the current cost to reach the node we are at. So, we initialize the cost of the start node with 0 and we mark it as visited, and we add it to the queue. Then we traverse the graph by expanding the node we are at. We loop its children (neighbor cities), if the neighbor is not visited, then we update the cost of the child to the new cost and we add it to the queue and we mark the child as visited, and we update the parent of the neighbor to the node we came from. We keep looping until the queue is empty.
* Two arrays are returned for the Main class, cost array and parent array. We use the parent array to find the path with the help of a stack.
* The Main class calls the class that is responsible for showing the path and the cost and displays them for the user.

# Uniform Cost Algorithm

## **GENERAL INFORMATION ABOUT THE ALGORITHM**

* Uniform cost algorithm is an uninformed searching algorithm used to search for shortest path between initial and final states (with no calculations for the heuristic).
* Uses a priority queue.
* Idea: Expands least-cost unexpanded node.
* Equivalent to breadth-first if step costs all equal.
* Properties:

1. Complete, if step cost is greater than some positive constant ε (we don’t want infinite sequences of steps that have a finite total cost)
2. Optimal, since it expands the least cost solution.
3. Space complexity: O(b^(C\*/ ε)).
4. Time complexity: Number of nodes with path cost ≤ cost of optimal solution (C\*), O(b^(C\*/ ε)). This can be greater than O(bd): the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps.

## **ALGORITHM IMPLEMENTATION**

* This algorithm is implemented using a priority queue sorted by the less cost, where the least cost comes first in the priority queue so we expand it first.
* We initialize the parent and the cost for each city with -1 and infinity respectively.
* The priority queue stores a pair of integers, the second in the pair is the node we are at, and the first in the pair is the current cost to reach the node we are at.
* We set the cost of the start city to 0 and we add it to the priority queue. Then we traverse the graph by expanding the node we are at.
* We loop its children (neighbor cities), if the cost of the neighbor is greater than the cost of reaching the neighbor from the current city (current cost for reaching the neighbor < cost[neighbor])
* Then, we update the cost of the child to the new cost and we add it to the priority queue, and we update the parent of the neighbor to the node we came from. We keep looping until the priority queue is empty.
* Two arrays are returned for the Main class, cost array and parent array.
* We use the parent array to find the path with the help of a stack.
* The Main class calls the class that is responsible for showing the path and the cost and displays them for the user.

# A\* Searching Algorithm

## **GENERAL INFORMATION ABOUT THE ALGORITHM**

* A\* algorithm is an informed searching algorithm used to search for shortest path between initial and final states.
* Uses a priority queue.
* Idea: avoids expanding paths that are already expensive.
* The evaluation function f(n) is the estimated total cost of the path through node n to the goal:  
   f(n) = g(n) + h(n)
* g(n): cost so far to reach n (path cost)
* h(n): estimated cost from n to goal (heuristic)

Properties:

1) Complete for finite number of nodes

2) Optimal if h(n) is admissible

3) Space complexity: exponential

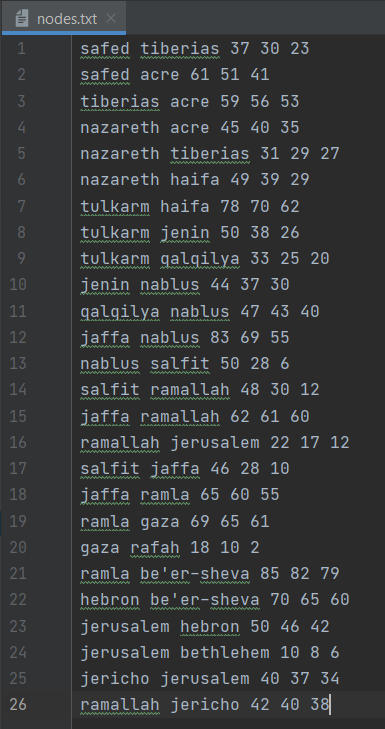
4) Time complexity: exponential

## **ALGORITHM IMPLEMENTATION**

* This algorithm was implemented using a priority queue like the uniform cost algorithm, but the priority queue stores a triple not a pair.
* The first in the triple is the current cost, the second is the node we are at, and the third is the heuristic of the node.
* A city can be visited from multiple other cities, so, the heuristic differs depending on the node we came from.
* The priority queue is initialized with the neighbors of the start node.
* Their parent is set to the start node and their cost is set to the edge cost plus their heuristic because it is guaranteed that at the first iteration, all of the costs will be less than infinity.
* The heuristic is subtracted when adding a new neighbor and a new heuristic is added. If we are at the goal city, we don’t add the heuristic because it is 0 for it.
* We loop until the priority queue is empty, if the cost of the neighbor is greater than the cost of reaching the neighbor from the current city (current cost for reaching the neighbor < cost[neighbor]), then we update the cost of the child to the new cost and we add it to the priority queue, and we update the parent of the neighbor to the node we came from, and we add the new heuristic with it.
* An additional array is returned from this function which is the path cost of the walking distance without the heuristic added to it, so we can use it in the second implementation of A\* as a heuristic, so the function returns a triple, three arrays, cost array, parent array, next heuristic array.
* We use the parent array to find the path with the help of a stack.
* The Main class calls the class that is responsible for showing the path and the cost and displays them for the user, and then proceeds to the second A\* implementation, which is implemented as the first one but with changing the edge cost and the heuristic.
* Note that each city has a fixed heuristic and the heuristic of the goal is 0 in the second implementation because we are obliged to use the heuristic that came from the first implementation.

# File Implementation

* Here is the input file that we have read for our project. It was explained in program implementation section in the first page of the report; each line contains an edge, each edge connects 2 cities and specifies the car, walk, and aerial distances respectively.



# Figure 1: Input file.

# Home Page UI

* The main UI is shown in figure 2. The user selects a radio button and presses the button labeled with “Select Start City”, and selects another radio button (Can be the same as the start one) and presses the button labeled with “Select Goal City”. Then, the user chooses a searching algorithm and presses Run.
* After pressing run, the user will get to one of the three algorithms interfaces to get the results.



# Figure 2: Home Page.

# Breadth First Search Run

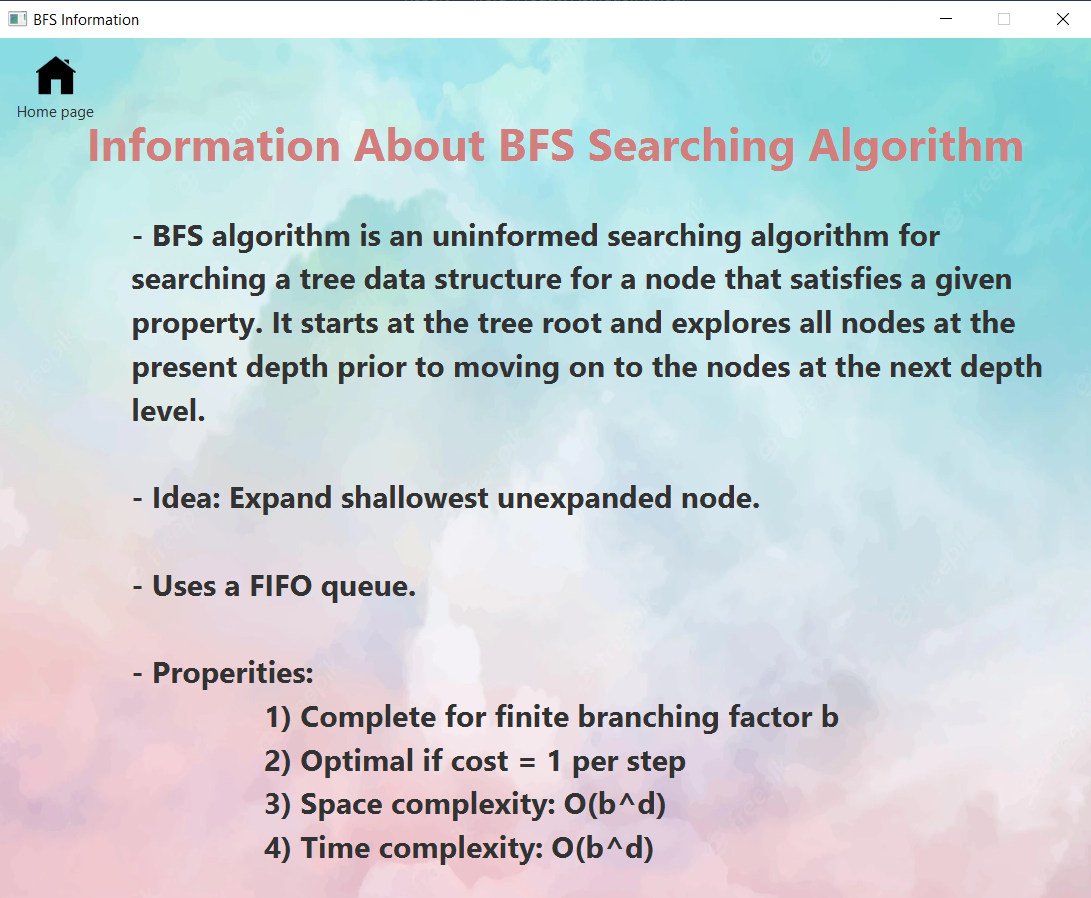
* If the user selects Safed and Ramallah for the search (start, goal), and chooses BFS algorithm for the execution, then he will get the following page, displaying information about the start and goal cities, the path between them, and the total distance.
* A button that is shown in the bottom of the page leads to another page for the algorithm, the information page.



# Figure 3: BFS Run Results.

# Breadth First Search Information Page

* When the user presses the information button, the following page will be displayed with some information about the BFS algorithm.
* The user can return to the home page by pressing the home page icon in the top left corner.



# Figure 4: Information about BFS Page.

# Uniform Cost Run

* If the user selects the same cities but with the uniform cost algorithm, he will get the following results.
* A button that is shown in the bottom of the page leads to another page for the algorithm, the information page.

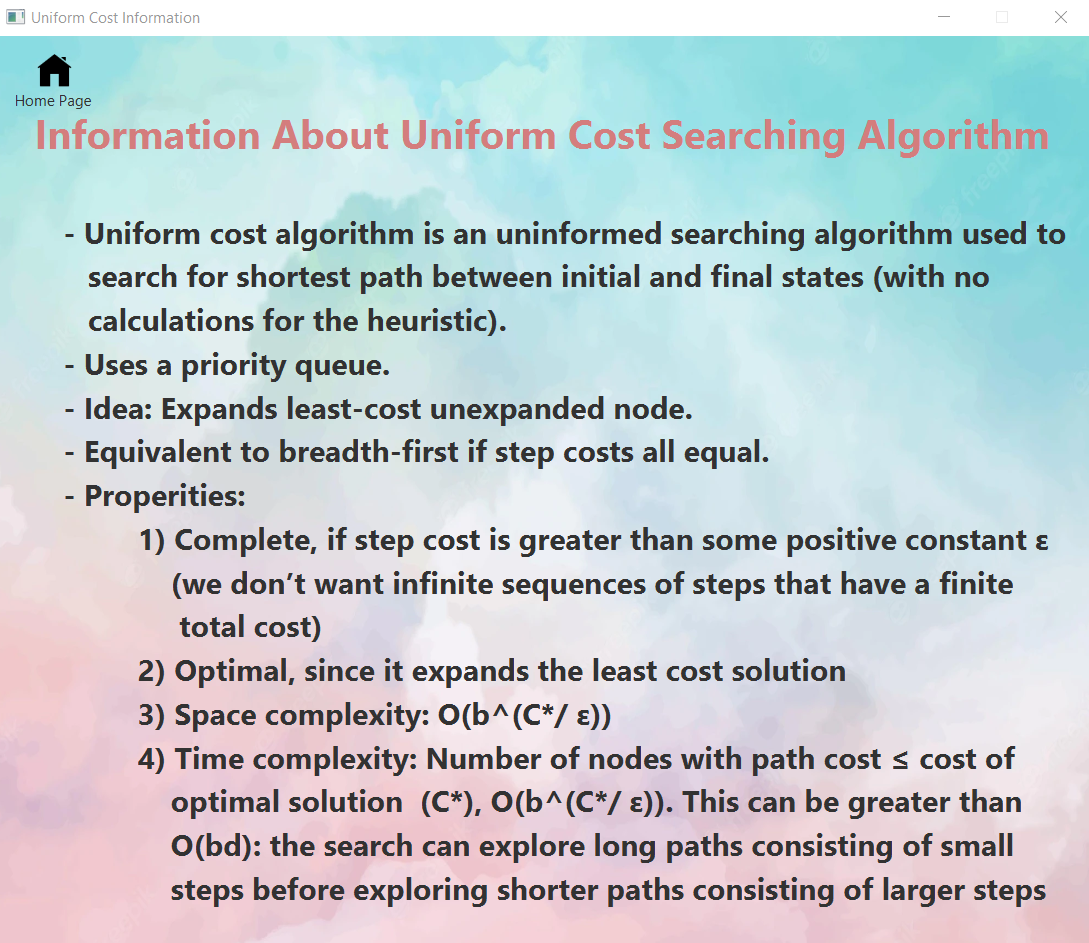


# Figure 5: Uniform Cost Results.

* ***Note: The paths are different because the uniform cost algorithm is optimal but the BFS is not.***

# Uniform Cost Information Page

* Then, when the user presses the information button, the following page that will be shown in the figure below will be displayed with some information about the Uniform Cost algorithm.
* The user can return to the home page by pressing the home page icon in the top left corner.



# Figure 6: Information about Uniform Cost Page.

# A\* Search Run

If the user selects to go, for example, from Qalqilya to Hebron using A\* algorithm, he will get 2 pages, each page represents a method:

## **First method for A\* search**

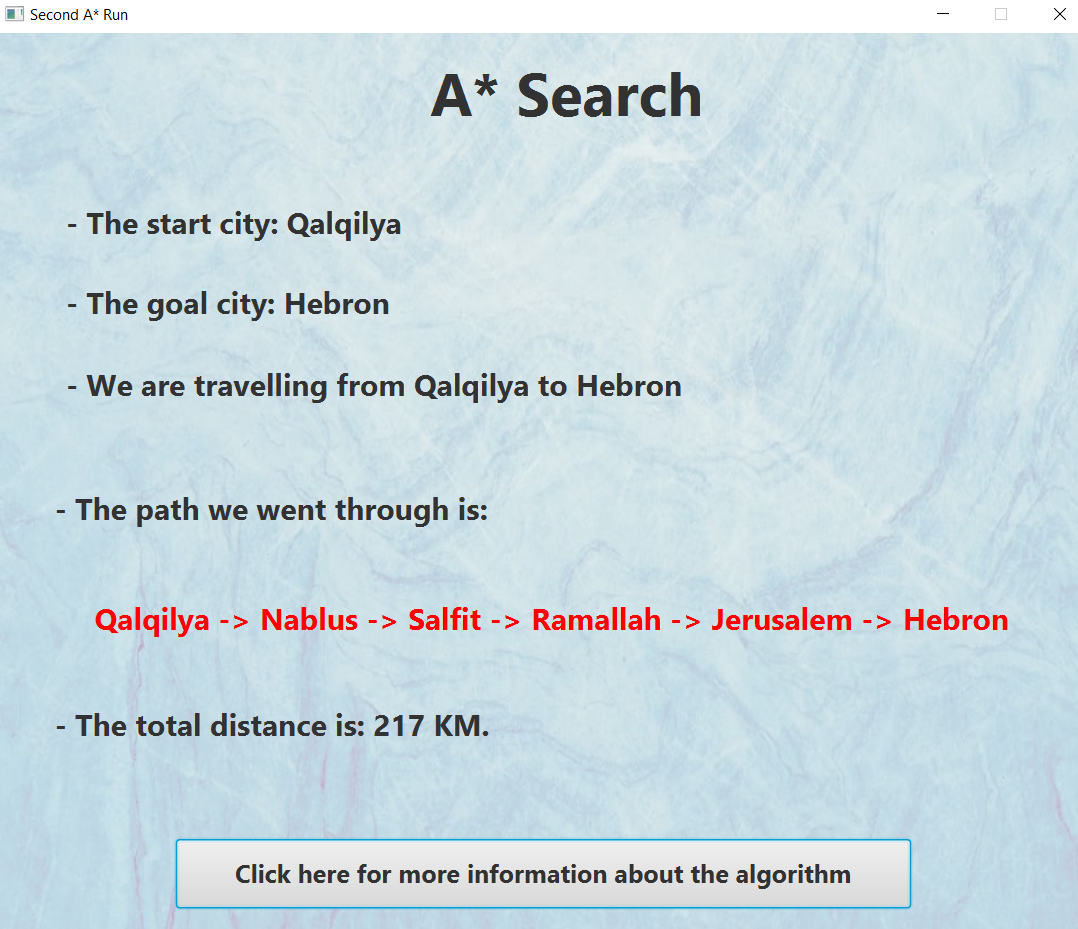
* For the 1st method: the path cost is the walking distance and the heuristic is the aerial distance.
* A button that is shown in the bottom of the page leads to another page for the algorithm, which is the second method of the A\*search.



# Figure 7: First A\* Implementation Run.

## **Second method for A\* search**

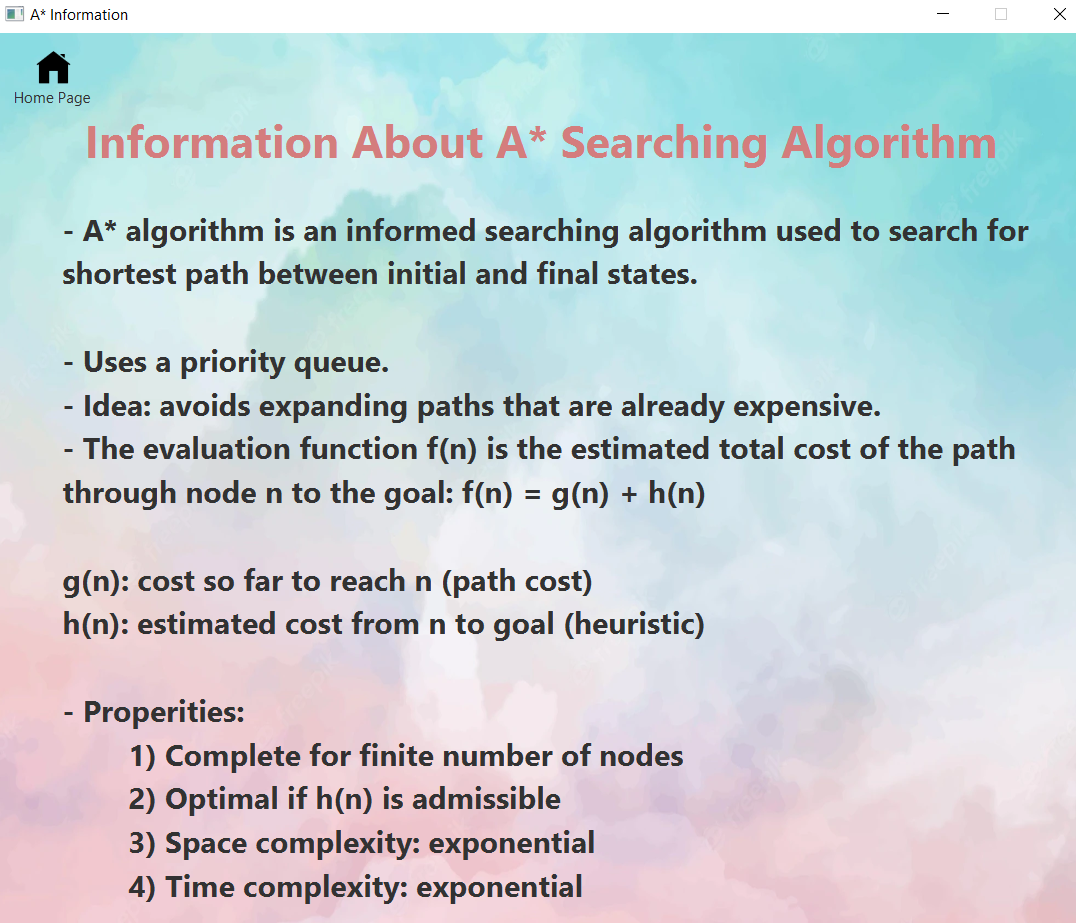
* For the 2nd one: the path cost is the car distance and the heuristic is the walking distance estimated from the 1st implementation.
* A button that is shown in the bottom of the page leads to another page for the algorithm, the information page.



# Figure 8: Second A\* Implementation Run.

# A\* Search Information Page

* When the user presses the information button, the following page that is shown in the figure below will be displayed with some information about the A\* algorithm.
* The user can return to the home page by pressing the home page icon in the top left corner.



# Figure 9: Information about A\* Page.