A* Assignment

BELFADIL Anas & PEYMAN Mohammad

January 7, 2020

Abstract

In this assignment we are going to implement the A* algorithm to try to find the shortest path (according to distance) from *Basílica de Santa Maria del Mar* (Plaça de Santa Maria) in Barcelona to the *Giralda* (Calle Mateos Gago) in Sevilla.

We are going to test 3 variations of the A*: the first, with the heuristic function equal to zero, this is the special case of the *Dijkstra* algorithm, the second, using the *Haversine* distance as the heuristic function, and finally using the *Spherical Law of Cosines*.

We are then going to compare the results of: the CPU time used, the distance of the path found and the number of nodes explored by each of the variations.

1 The A* Algorithm:

1.1 Description:

A* is an informed search algorithm, or a best-first search, formulated in terms of weighted graphs it means that: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance traveled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied. At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes :

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

- where n is the next node on the path
- *g(n)* is the cost of the path from the start node to n
- *h(n)* is a heuristic function that estimates the cost of the cheapest path from n to the goal.

A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended. The heuristic function h is problem-specific. If the heuristic function is **admissible**, meaning that it never overestimates the actual cost to get to the goal, A* is guaranteed to return a **least-cost path** from start to goal. If moreover h is **monotone** i.e the cost estimated *h(n)* from the node n to the goal node is greater or equal to the cost from n to the goal node through any node n' children of n :

$$h(n) \leq c(n, n') + h(n') \quad \forall n, n' \quad | \quad n' \textit{ is a child of } n$$

Then A* doesn't expend a node more than once, and therefore the algorithm is **optimally efficient** in this case.

1.2 Pseudo-code:

The goal node is denoted by **node_goal** and the start node is denoted by **node_start**.

We maintain two lists: **OPEN** and **CLOSE**:

- **OPEN** consists on nodes that have been visited but not expanded (meaning that successors have not been explored yet). This is the list of pending tasks.
- **CLOSE** consists on nodes that have been visited and expanded (successors have been explored already and included in the open list, if this was the case).

*PS: In this assignment we are using monotone heuristic functions, therefor once a node is in the **CLOSE** list it won't get out from it.*

Algorithm 1 A* Algorithm pseudo-code

```
1: Put node_start in the OPEN list with f (node_start) = h(node_start) (initialization)
2: while the OPEN list is not empty {
3:    Take from the open list the node node_current with the lowest
4:       f (node_current) = g(node_current) + h(node_current)
5:    if node_current is node_goal we have found the solution; break
6:    Generate each state node_successor that come after node_current
7:    for each node_successor of node_current {
8:       Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9:       if node_successor is in the OPEN list {
10:         if g(node_successor) ≤ successor_current_cost continue (to line 17 )
11:         else {
12:         Add node_successor to the OPEN list
13:         Set h(node_successor) to be the heuristic distance to node_goal
14:      }
15:      Set g(node_successor) = successor_current_cost
16:      Set the parent of node_successor to node_current
17:   }
18:   Add node_current to the CLOSED list
19: }
20: if (node_current != node_goal) exit with error (the OPEN list is empty)
```

1.3 The cost function:

We have two terms to calculate for the cost function f at each node visited by the algorithm n : $f(n) = g(n) + h(n)$ where:

- $g(n)$ is the cost from the start node to n, it is therefor the sum of the weights of the edges constituting the path from the start node to n. We are going to use the Haversine distance to calculate it.
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. We are going to try 3 different heuristic functions for this :

1. The zero function, this is the special case of the Dijkstra

2. The Haversine distance from n to the goal node

3. The Spherical Law of Cosines

1.3.1 The Haversine distance:

The Haversine distance is the great-circle distance d between two points on a sphere given their longitudes λ_1, λ_2 and latitudes φ_1, φ_2, it is given by:

$$d = 2R \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Where:
R is the radius of the sphere.

1.3.2 The Spherical Law of Cosines:

The distance d is given by the Spherical Law of Cosines as:

$$d = \left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)cos\left(\frac{\lambda_2 - \lambda_1}{2}\right)\right)R$$

2 Implementation in C:

2.1 A* for solving a routing problem:

We want to use the A* algorithm to find an optimal path (according to distance) from *Basílica de Santa Maria del Mar* (Plaça de Santa Maria) in Barcelona to the *Giralda* (Calle Mateos Gago) in Sevilla. For this we have a map in the form of a file structured in the following manner:

node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon

way|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|membernode|membernode|...

relation|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|rel_type|type;@id;@role|...

Where the lines starting with node describe the nodes, each line is for one node, it contains its *id, name, longitude, latitude...*

The lines starting with way contains information about the ways between nodes, the most important information they contain are:

- The oneway on the 7^{th} field, which indicate if the way is unidirectional.
- The list of nodemembers of the way, adjacent nodes on this list are connected.

First, we are going to transform this map file to a weighted graph suitable for the A* algorithm and save it as a binary file that can be read very quickly. Then, we develop the A* program that reads the binary file, performs the A* and outputs a file containing the optimal path it found.

2.2 C program for saving the map as a graph:

The goal here is to save the map as a graph in a form that could easily answer to the A* algorithm needs. If we examine the A* pseudo code we can see that the information it most frequently needs from the graph is: give me the list of successors of the *node_current.*

For this reason we are going to use the following data structure, it contains all the information we need to know about a node and can answer easily to the A* needs:

```
/* Data structure for the nodes */
typedef struct{
unsigned long id; // Node identification
char *name;
double lat, lon; // Node position
unsigned short nsucc; // Number of node successors; i. e. length of successors
unsigned long *successors; // List of successors
} node;
```

A sorted array of structure **nodes* is going to be used to store all the nodes.

```
node *nodes;
```

Since the nodes id's are not consecutive, we need a function that could locate a node, given its id, in the sorted array of structures. For this purpose we used the following binary search function:

```
/* Function for binary search */
unsigned long imin, imax, imid;
long bi_search(node *Vector, unsigned long Vector_length, unsigned long target){
  imin = 0; imax = Vector_length-1;
  while (imax >= imin){
    imid = (imax + imin)*0.5;
    if (Vector[imid].id == target){
      return imid;
    }
    else if (Vector[imid].id < target){
      imin = imid + 1;
    }
    else {
      imax = imid - 1;
    }
  }
  return -1;
}
```

We also use this unction for writing the graph in a binary file:

```
/* Function for writing the graph in a binary file*/
void writing(node *nodes){
  FILE *fin;
  int i;
  unsigned long total_nsucc=0UL;
  for(i=0;i<nnodes; i++) total_nsucc+=nodes[i].nsucc;
  if((fin=fopen("binary.bin","wb"))==NULL)
  printf("the output binary data file cannot be opened\n");
  /* Global data---header */
   if((fwrite(&nnodes,sizeof(unsigned long),1,fin)+
    fwrite(&total_nsucc,sizeof(unsigned long),1,fin))!=2)
    printf("when initializing the output binary data file\n");
   /* Writting all nodes */
   if( fwrite(nodes,sizeof(node),nnodes,fin)!=nnodes)
   printf ("when writing nodes to the output binary data file\n");
  //WRITING SUCCESORS IN BLOCKS
  for (i=0;i<nnodes; i++)
    if(nodes[i].nsucc){
      if(fwrite(nodes[i].successors,sizeof(unsigned long),
      nodes[i].nsucc,fin)!=nodes[i].nsucc)
      printf("when writing edges to the output binary data file\n");
    }
fclose(fin);
}
```

Inside the main function, we go through the file line by line, and in each line field by field. From the lines starting with node, we get the id, name, longitude and latitude for the nodes. And, from the lines starting with way, we can get the successors for each node.

The part of the code that does this :

```
/* Get the first line of the file. */
line_size = getline(&line_buf, &line_buf_size, fp);
/* nodenum variable for given the order number of the node */
unsigned long nodenum = 0;
while (line_size >= 0) {
/* Go line by line and get nodes lines*/
if (line_buf[0] == 'n') {
    /* Go through the line field by field and get the node info*/
    int field = 0;
    char *token;
    while ((token = strsep(&line_buf, "|")) != NULL) {
      if (field == 1) {
        /*We get the node id from the first field*/
        nodes[nodenum].id = strtoul(token, NULL, 10);
      }
      else if (field == 2) {
        /*From the second field we get the node name*/
        nodes[nodenum].name = token;
      }
      else if (field == 9) {
        /*From the 9th field we get the node latitude*/
        nodes[nodenum].lat = atof(token);
      }
      else if (field == 10) {
        /*From the 10th field we get the node longitude*/
        nodes[nodenum].lon = atof(token);
      }
      field += 1;
    }
    nodenum += 1;
  }


  /* Go line by line and get number of neighbours*/
else if (line_buf[0] == 'w')
{
    /* Go through the line field by field and count the neighbours*/
    int field = 0;
    char *token;
    char *oneway;
    unsigned long token0;
    long index;
    long index0;

    while ((token = strsep(&line_buf, "|")) != NULL) {
      if (field == 7) {
        oneway = token;
      }
      else if (field > 8) {
        index = bi_search(nodes, nnodes, strtoul(token, NULL, 10));
        if (field == 9) index0 = index;
        /* Add the node in the field to the succesors of the node in field-1 */
        else if (index != -1 && index0 != -1) {
          nodes[index0].nsucc += 1;
          nodes[index0].successors = realloc(nodes[index0].successors,
          sizeof(unsigned long)*nodes[index0].nsucc);
          nodes[index0].successors[nodes[index0].nsucc-1] = strtoul(token, NULL, 10);
          /* If the way is not "oneway" add the node in field-1 to the successors of
              the node in field */
          if ((strcmp(oneway,"oneway") != 0) && (field > 9)) {
            nodes[index].nsucc += 1;
            nodes[index].successors = realloc(nodes[index].successors,
            sizeof(unsigned long)*nodes[index].nsucc);
            nodes[index].successors[nodes[index].nsucc-1] = token0;
          }
```

```
        }
        token0 = strtoul(token, NULL, 10);
        index0 = index;
      }
      field += 1;
    }
  }
/* Increment the line count */
line_count++;

/* Get the next line */
line_size = getline(&line_buf, &line_buf_size, fp);
}
```

The total time for constructing the graph is around 49s and it takes 3s for writing it to the binary file.

2.3 A* program in C:

The constants we are going to use inside this program are:

```
#define PI 3.14159265359
#define R 6.3781e6 //earth radius in meters
```

Besides the node data structure, we are going to use AStartStatus data structure for the node status, it contains the values of g and h, the parent id and the status indicating if the node is in the open list, the closed list or neither.

For easy referencing this data will be organized in an array like the nodes array of structure i.e. the node in the i^{th} position in the *nodes* array will have it's status in the i^{th} position in the array *Status*.

```
/* Data structure for the status of the node */
typedef struct {
  double g, h;
  unsigned long parent;
  Queue whq;
} AStarStatus;
```

We are also going to use a linked list to store the open list of nodes in a sorted way according to their cost function values.

```
/* linked list to store the open list nodes in a sorted way */
struct linked_list{
  unsigned long index; // internal id of the node
  double f;
  struct linked_list * next;
};
```

To manage this linked list we need two function:

- One for doing a sorted insert in the list:

```
/* function to insert a new_node in a list in a sorted way */
void sortedInsert(open_list ** head_ref, open_list * new_node)
{
  open_list * current;
  /* Special case for the head end */
  if (*head_ref == NULL || (*head_ref)->f >= new_node->f) {
    new_node->next = *head_ref;
    *head_ref = new_node;
  }
  else{
    /* Locate the node before the point of insertion */
    current = *head_ref;
    while (current->next!=NULL &&
           current->next->f < new_node->f)
```

```
    {
        current = current->next;
    }
    new_node->next = current->next;
    current->next = new_node;
  }
}
```

- And one for deleting the nodes that move to the closed list:

```
/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */
void deleteNode(open_list **head_ref, unsigned long key)
{
  // Store head node
  open_list* temp = *head_ref, *prev;

  // If head node itself holds the key to be deleted
  if (temp != NULL && temp->index == key) {
      *head_ref = temp->next;   // Changed head
      free(temp);               // free old head
      return;
  }

  // Search for the key to be deleted, keep track of the
  // previous node as we need to change 'prev->next'
  while (temp != NULL && temp->index != key) {
      prev = temp;
      temp = temp->next;
  }

  // If key was not present in linked list
  if (temp == NULL) return;

  // Unlink the node from linked list
  prev->next = temp->next;

  free(temp);  // Free memory
}
```

For the distance we use:

- Haversine:

```
double Haversine(node *init, node *prev)
{
  // Haversine formula
  double phi1= init->lat/180*PI;
  double phi2= prev->lat/180*PI;
  double lambda1=init->lon/180*PI;
  double lambda2=prev->lon/180*PI;
  double dif_lat= fabs(phi1-phi2);
  double dif_lon= fabs(lambda1-lambda2);

  double a = sin(dif_lat/2)*sin(dif_lat/2)+cos(phi1)*cos(phi2)*sin(dif_lon/2)*sin(
      dif_lon/2);
  double c= 2*atan2(sqrt(a),sqrt(1-a));

  return c*R;
}
```

- And Cosines:

```
double Cosines (node * init, node * prev)
```

```
{
  // Spherical Law of Cosines
  double phi1= init->lat/180*PI;
  double phi2= prev->lat/180*PI;
  double lambda1=init->lon/180*PI;
  double lambda2=prev->lon/180*PI;
  double dif_lon= fabs(lambda1-lambda2);

  return acos( sin(phi1)*sin(phi2)+cos(phi1)*cos(phi2)*cos(dif_lon))*R;

}
```

Reading from the binary file took around 1s which a lot better than constructing the graph every time before running the A* which takes 49s as mentioned before, the function for reading the file is:

```
void reading()
{
  int i;
  static unsigned long * allsuccessors;
  unsigned long total_nsucc=0UL;

  FILE * fin;

  fin=fopen("binary.bin","rb");

//GLOBAL DATA HEADER
  if ((fread(&nnodes,sizeof(unsigned long),1,fin)+
  fread(&total_nsucc,sizeof(unsigned long),1,fin))!=2)
  printf("when reading the header of the binary data file\n");
  // GETTING MEMORY FOR ALL THE DATA
  if ((nodes= (node *) malloc(sizeof(node)*nnodes))==NULL)
  printf("when allocating memory for the nodes vector\n");
  if ((allsuccessors= (unsigned long*)
  malloc(sizeof(unsigned long)*total_nsucc))==NULL)
  printf("when allocating memory for the edges vector\n");
//READING ALL DATA FROM FILE
  if (fread(nodes,sizeof(node),nnodes,fin)!=nnodes)
  printf("when reading nodes from the binary data file\n");
  if (fread(allsuccessors,sizeof(unsigned long),total_nsucc,fin)!=total_nsucc)
  printf("when reading successors from the binary data file\n");

  fclose(fin);
//SETTING POINTERS TO SUCCESSORS
  for (i=0;i<nnodes; i++) if (nodes[i].nsucc){
    nodes[i].successors=allsuccessors;
    allsuccessors+=nodes[i].nsucc;
  }
}
```

The A* function for the case when we use Haversine distance as the heuristic function is as follows:

```
int AStar (unsigned long start, unsigned long goal)
{
  node *current , *successor;

  int i, nelements=0;
  open_list *succ;
  double successor_current_cost;


  unsigned long start_index = bi_search(nodes, nnodes, start);
  unsigned long goal_index = bi_search(nodes, nnodes, goal);

  /* Allocating memory for the status of the nodes */
  Status = (AStarStatus *) calloc(nnodes, sizeof(AStarStatus));
```

```
Open = (open_list *) calloc(1, sizeof(open_list));
succ = (open_list *) malloc(1* sizeof(open_list));

/* Put node_start in the OPEN list */
Open->index = start_index;

/* The number of elements in the open list now is nelements = 1 */
nelements += 1;

/* Change the Queue status for node start to open */
Status->whq = 1;

/* Set g and h for the start node */
Status->g = 0;
Status->h =  Haversine(&nodes[start_index], &nodes[goal_index]);

/* f(start) = h(start) */
Open->f = Status->h;


while (nelements>0){
  current = &nodes[Open->index]; //takes first node in the open list: the one with
      lowest f distance
  unsigned long current_index = bi_search(nodes, nnodes, current->id); //find its
      position in the nodes array

  if ( current->id == goal) return 0; //if node_current is node_goal we have found
      the solution; break
  if (current->nsucc > 0) {
    /*Generate each state node_successor that come after node_current*/
    for ( i = 0; i < (current->nsucc); i++){
      unsigned long successor_index = bi_search(nodes, nnodes, *(current->successors
          +i));
      successor = &nodes[successor_index];
      successor_current_cost = Status[current_index].g + Haversine(successor,
          current);
      if ( Status[successor_index].whq == 1){
          if ( Status[successor_index].g <= successor_current_cost) continue;
        }

      else if ( Status[successor_index].whq == 2){
          if ( Status[successor_index].g <= successor_current_cost) continue;
          Status[successor_index].whq = 1;
          succ = (open_list *) malloc(1* sizeof(open_list));
          succ->index = successor_index;
          succ->f = successor_current_cost + Haversine(successor, &nodes[goal_index
              ]);
          nelements += 1;
          sortedInsert(&Open, succ);
      }

      else{
        Status[successor_index].whq = 1;
        succ = (open_list *) malloc(1* sizeof(open_list));
        succ->index = successor_index;

        succ->f = successor_current_cost + Haversine(successor, &nodes[goal_index]);

        sortedInsert(&Open, succ);
        nelements += 1;
        Status[successor_index].h = Haversine(successor, &nodes[goal_index]);
      }

      Status[successor_index].g = successor_current_cost;
      Status[successor_index].parent = current_index;
```

```
      }
    }

  Status[current_index].whq = 2;
  deleteNode (&Open, current_index);
  nelements -= 1;
  }

  return 1;
}
```

Finally in the main function we combine everything to solve the routing problem, we calculate the time it takes for reading the binary file, for executing the A* function and the total time, we also calculate the number of nodes visited and we print the solution path to the output file.

```
int main ()
{
  unsigned long start = 240949599; //771979683  240949599
  unsigned long goal = 195977239; //429854583  195977239
  time_t ttime, ttime1;
  ttime = clock();
  int counter=0;

  reading();
  printf("Total time spent reading : %fs \n",((clock()-(double)ttime))/CLOCKS_PER_SEC)
      ;

  //unsigned long start_index = bi_search(nodes, nnodes, start);
  unsigned long goal_index = bi_search(nodes, nnodes, goal);

  /* Allocating memory for the status of the nodes */
  Status = (AStarStatus *) malloc(nnodes * sizeof(AStarStatus));
  Open = (open_list *) malloc(1* sizeof(open_list));

  ttime1 = clock();
  AStar(start, goal);
  printf("Total time spent in A* : %fs \n",((clock()-(double)ttime1))/CLOCKS_PER_SEC);

  AStarStatus* curr;
  long n;

  FILE * output;
  output=fopen("output_a_star.txt","w");

  printf("Iteration |  Node  Id  | Distance  |    Lat    |   Long \n");
  fprintf(output, "Iteration |  Node  Id  | Distance  |    Lat    |   Long \n");

  n=0;
  curr = &Status[goal_index];
  unsigned long id = goal_index;
  while (curr->parent != 0)
  {
    printf("     %4.1d | %10.lu | %6.2f | %2.6f | %2.6f \n", n, nodes[id].id, curr->g,
        nodes[id].lat, nodes[id].lon);
    fprintf(output,"     %4.1d | %10.lu | %6.2f | %2.6f | %2.6f \n", n, nodes[id].id,
       curr->g, nodes[id].lat, nodes[id].lon);
    id = curr->parent;
    curr = &Status[curr->parent];
    n++;
  }

  printf("     %4.1d | %10.lu | %6.2f | %2.6f | %2.6f \n", n, nodes[id].id, curr->g,
      nodes[id].lat, nodes[id].lon);
  fprintf(output,"     %4.1d | %10.lu | %6.2f | %2.6f | %2.6f \n", n, nodes[id].id,
      curr->g, nodes[id].lat, nodes[id].lon);
```

```
  for (int i=0; i<nnodes; i++){
    if (Status[i].whq != 0) counter++;
  }

  printf("Total number of visited nodes : %d \n", counter);
  printf("Total time spent : %fs ",((clock()-(double)ttime))/CLOCKS_PER_SEC);
  return 0;
}
```

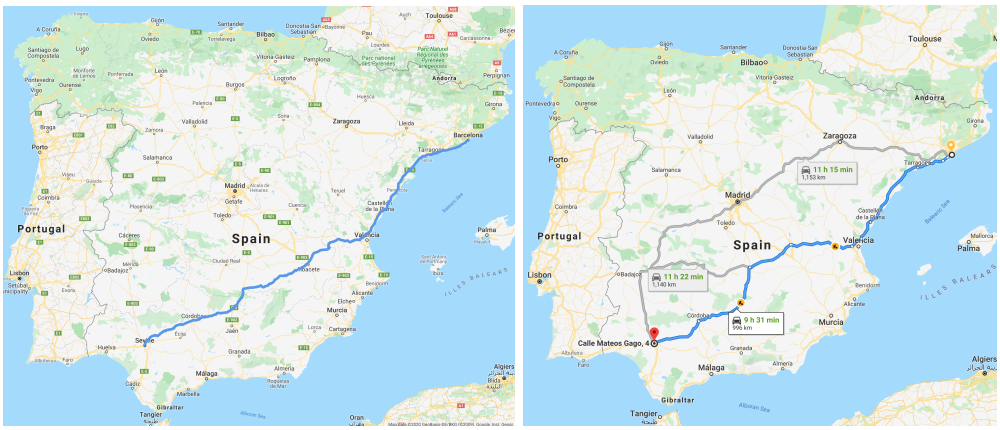
3 Results and conclusions:

System used:

The system used to execute the program is:

- Memory : 8 GiB
- Processor : Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
- OS : Ubuntu 18.04
- GCC : gcc version 7.4.0
- Command for compiling : gcc -Wall -lm -lgcc -lc -c "%f"

Results:



(a) Our solution (optimizing for the distance) (b) Google's solution (optimizing for time)

Figure 1: Comparing our solution and Google solution for the routing problem from *Basílica de Santa Maria del Mar* (Plaça de Santa Maria) in Barcelona to the *Giralda* (Calle Mateos Gago) in Sevilla. Our implementation gives shorter length at around 960 km instead of Google's 996 km.

In the table below we present a comparison between the three variation of A*, in the three we use the Haversine distance to calculate g the cost for traveling from the start to the current node, for the heuristic we use the zero function (this is the Dijkstra algorithm), the Haversine distance and the Spherical Law of Cosines. The metrics for this comparison are the path's length, the A* function time and the number of nodes visited.

Table 1: Summary of the results for the 3 variations of A*

Variation	g	h	path's length	A* time	Number of explored nodes
Dijkstra	Haversine distance	0	959883.54 m	39.6s	11906278
Haversine	Haversine distance	Haversine	959883.54 m	8.3s	2308770
Cosines	Haversine distance	Cosines	959883.54 m	8.6s	2308770

Conclusions:

- The three variations give the same optimal path length.
- Dijkstra takes a lot more time than the other two, this is due to big number of nodes it visits before finding the goal node, this shows the importance of the heuristic function in guiding the exploration towards the goal node.
- When using the Spherical Law of Cosines, A* takes slightly more time to execute although the number of explored nodes is exactly the same, this explained by the fact that the Cosines function takes more time than the Haversine function to calculate the distance from the current to the goal node.

References:

- Wikipedia contributors. (2019, December 17). A* search algorithm. In Wikipedia, The Free Encyclopedia.
- Wikipedia contributors. (2019, August 7). Haversine formula. In Wikipedia, The Free Encyclopedia.
- Wikipedia contributors. (2019, December 16). Earth radius. In Wikipedia, The Free Encyclopedia.
- (Global Edition) Stuart J. Russell, Peter Norvig - Artificial Intelligence_ A Modern Approach, 3rd Edition-Pearson Education (2016)
- (The Addison-Wesley series in artificial intelligence) Judea Pearl - Heuristics_ intelligent search strategies for computer problem solving-Addison-Wesley Pub. Co (1984)