AStar assignment — A routing problem

Lluís Alsedà

October 25, 2018

Introduction

The assignment consists in computing 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 by using an AStar algorithm. To this end one has to implement the AStar algorithm (compulsory in form of a function — not inside the main code) and compute and write the optimal path.

As the reference starting node for *Basílica de Santa Maria del Mar* (Plaça de Santa Maria) in Barcelona we will take the node with key (`@id`): `240949599` while the goal node close to *Giralda* (Calle Mateos Gago) in Sevilla will be the node with key (`@id`): `195977239`.

For reference: My implementation (run in a very quick computer with a lot of memory) takes 20.51 CPU seconds to read and store the file and create the edges. The AStar computation itself takes 3.98 CPU seconds and the solution length is 959655.33 meters. It is shorter than the one found by Google maps since we are minimizing distances instead of time.

The map — The file

The map is contained in the file `spain.csv` that you have to download. This file has been created from `spain.osm` (written in *XML*) with the help of the *awk* program to get a format more friendly and easier to read than the original XML format.

The file `spain.csv` has the character '`|`' as field separator and it has three types of fields: `node`, `way` and `relation`. The structure of each field is:

```
node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon
way|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|membernode|membernode|membernode|...
relation|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|rel_type|type;@id;@role|...
```

Nodes specify a single point in the map. *The list of nodes is sorted with respect to keys (`@id`).* A way is a list of nodes (between 2 and 2000) and specify the relations between them (edges in the graph). Relations are mainly for drawing the maps and are irrelevant to our problem.

The `@id` field has maximum length 10 chars. For comparison speed I recommend to store it as `unsigned long`.

The `@oneway` field takes two values: `empty` or `oneway`. If the pair of nodes `A|B` appears in the nodes list of the `way`, then in the graph always there is an edge from `A` to `B`. If the value of `@oneway` is empty (`twoways`) then additionally, in the graph there is an edge from `B` to `A`.

Warning: Unfortunately, the file is not consistent. There are ways with less than two nodes (that have to be discarded) and there are nodes in ways that do not appear in the list of nodes. They have to be omitted and the process of assigning edges to every pair of consecutive nodes in a way must be restarted.

A look into the file:

```
### Format: relation|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|relation_type|membertype;@id;@role|...
### Format: way|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|member nodes|...
### Format: node|@id|@name|@place|@highway|@route|@ref|@oneway|@maxspeed|node_lat|node_lon
node|171773|||||||38.6048094|-0.0489952
node|171774|||||||38.6061981|-0.0496867
node|171775|||||||38.6067166|-0.0498342
node|171776|||||||38.6026304|-0.0497205
node|171933|||traffic_signals|||||40.4200658|-3.7016652
node|171946|||traffic_signals|||||40.4212535|-3.6844535
node|171948|||||||40.4202342|-3.6877944
 ................................
 ................................
 ................................
way|407780|Oviedo-Porceyo||secondary||AS-266|oneway||493404488|479386
way|2497138|pit lane||raceway|||||10920305|944276809|944276820|332679427|944276824|944276827|10920306|944276832|944276835|10
way|2497139|Circuito Permanente de Jerez||raceway|||||10920021|944276570|944276572|332679445|10920054|10920020|944276577|944
way|2497152|Curva Ayrton Senna||raceway|||||10920082|944276785|10920047|944276797|944276799|332679423|10920083
way|3318645|||secondary||PM-801|||248467102|1342058664|1342058670|248467106|248467637
way|3318733|Carretera a SA Caleta||tertiary|||||300064224|300064128|300064130|300064131|300064132|300064133|300064134|300064
way|3318735|||tertiary||PM-801|||1341702065|16348842|1341702088|16348843|1341702089|16348844|16348845|1341702090|1341702093|
way|3986513|gazteluko bidea||track|||||20822627|20822628|20822629|20822630|20822631|20822632|20822633|20822634|20822635|2082
way|3996189|M-40||motorway||M-40|oneway||23002322|23002198|23002236|23001985|23001944|23001220|23001195|359852835|23001124|2
```

Some data about the file:

- The file has 3 comment lines at the beginning, that have to be discarded.
- Longest line: 79857 chars
- Maximum number of fields: 5306
- Maximum width of `@name`: 184 chars
- Number of nodes: 23895681
- Number of ways: 1417363
- Number of relations: 25394533

Note: The best way of reading the file is to read it line by line as a string with the function `getline` and then split the line at the places with a '`|`' with a "tokenizer" such as `strsep` (do not use `strtok` since it does not count as valid the empty fields; in other words '`||||||||`' is interpreted as a single '`|`' thus destroying, for instance, the important characteristic that a node has 9 fields before latitude and longitude). In this framework the fields are strings. The numeric ones must be effectively converted to numbers with the standard functions `strtoul` and `atof`.

Storing the nodes and the graph

Due to the concrete values of the `@id`'s of nodes and the jumps between consecutive `@id`'s (in the file), it is not feasible to use the `@id`'s as indexes in the vectors. To check it let us list a little bit of data on node `@id`'s:

- First node `@id`: 171773
- Last node `@id`: 1543488258
- Jumps between consecutive `@id`'s of length larger than 1000000 (all of them happen only once): 4050482, 2139397, 2098843, 2068243, 1849564, 1657410, 1489548, 1433947, 1142856, 1119317, 1066818, 1059153, 1031763.
- Number of jumps between consecutive `@id`'s of length between 100000 and 1000000: 1396 (with a jump average of 229287.1).

There is no enough memory (in the universe?) for that. Thus, I *strongly recommend to store the nodes in a vector of node structures and refer to the nodes internally in the program by the index in this vector* (not by the `@id`). This is way of naming nodes allows a quicker way of finding them. The price to pay for this is that at reading/storing time the arrows in the graph (specified in the `way` commands) have to be converted from `@id` numbering to vector index numbering.

I am using the following structure although everything (specially the adjacency relations can be implemented in other ways):

```
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;
} node;
```

The number of nodes is the dimension of this vector. To determine this number for any data file it may be necessary to do a first reading of the file to compute this number (although it is given above for the file `spain.csv`).

Neighbours — setting the arrows between nodes

An information that may help in deciding the storage model for the neighbours: the maximum valence in the graph is 16 (maximum number of nodes connected to a given one) and the average valence is 1.99. At reading time, to save `realloc`'s it might be useful to have an additional vector `unsigned short nsuccdim;` such that `nsuccdim[i]` contains actual dimension of `node[i].sucessors` (and it is initialized to zero).

Concerning the conversion of the arrows from `@id` numbering to vector index numbering: When processing the `way`'s one has two keys `A|B` and has to search in the vector of nodes for the indexes (say `a|b`)

of the nodes with those keys to establish the edges between them (`node[a].sucessors[nsucc] = b`). This is a painful process where brute force does not work. For example, the loading of the map of Catalonia takes more than two hours by using this strategy. Thus, some efficient search method as binary search must be used. You may want to read from page 33 to page 47 (specially pages 39 and 41) of my notes (in Catalan) *Estructures i tipus de dades en C* that you can find in
`http://mat.uab.cat/~alseda/MatDoc/C_Estructures.pdf`

Memory models and queues for AStar

For the AStar part, additionally I am using the following memory types:

```
typedef char Queue;
enum whichQueue {NONE, OPEN, CLOSED};

typedef struct {
  double g, h;
  unsigned long parent;
  Queue whq;
} AStarStatus;
```

The variable `whq` together with the `whichQueue` enumeration codes are extremely useful to put nodes in the `OPEN` or in the `CLOSED` list and to answer questions like *is the node* `nodes[index]` *in the* `OPEN` *list?* or *is the node* `nodes[index]` *in the* `CLOSED` *list?* with great simplicity. However, to find the element of the `OPEN` list with minimal cost `f` using this mechanism is prohibitive. For this I recommend to create a queue based on a linked list with an insertion function that always maintains the list sorted according to the cost value `f` from minimum to maximum. Then, the the element of the `OPEN` list with minimal cost `f` is simply the first one.

On the heuristic function `h`

It is very important the choice of the heuristic distance (the concrete result strongly depends on this). Assuming that you choose as heuristic the shortest straight distance on the earth surface between two points, there are different ways of computing this distance (it is not well defined). Most common distances are Haversine formula (great-circle distance between two points), Spherical Law of cosines and others. I recommend you to consider carefully the distance you adopt. To this end, you may want to look at
`http://www.movable-type.co.uk/scripts/latlong.html`

Saving time when dealing with the graph

An efficient approach to the problem (similar to what happens in GPS industry) is to write two programs:

- One that reads the file and computes the graph with binary search and stores the graph in a *binary file* with arrows between nodes already determined. This file is thus very quick to read. To avoid a big number of `alloc`'s I recommend to store all names of all nodes in a single vector and all successors of all nodes in another single vector, and store these vectors in the binary file in one piece.

- A second program reads the (formatted) binary file, thus getting the map already in graph form, asks for the start and goal nodes and performs AStar algorithm. After reading the (big) vectors containing all names and successors, each node can set pointers to the parts of the vectors corresponding to this node. The reading of the binary file takes 0.76 CPU seconds in my implementation.

The specific code for writing and reading this binary file is the following:

Writing:

```
void ExitError(const char *miss, int errcode) {
    fprintf (stderr, "\nERROR: %s.\nStopping...\n\n", miss); exit(errcode);
}

int main (int argc, char *argv[]){
  FILE *fin;
  unsigned long nnodes = 23895681UL;
  char name[257];

  node *nodes;
  if((nodes = (node *) malloc(nnodes*sizeof(node))) == NULL)
            ExitError("when allocating memory for the nodes vector", 5);
```

```
/* Computing the total number of successors */
unsigned long ntotnsucc=0UL;
for(i=0; i < nnodes; i++) ntotnsucc += nodes[i].nsucc;

/* Setting the name of the binary file */
strcpy(name, argv[1]); strcpy(strrchr(name, '.'), ".bin");
if ((fin = fopen (name, "wb")) == NULL)
         ExitError("the output binary data file cannot be opened", 31);

/* Global data --- header */
if( fwrite(&nnodes, sizeof(unsigned long), 1, fin) +
    fwrite(&ntotnsucc, sizeof(unsigned long), 1, fin) != 2 )
         ExitError("when initializing the output binary data file", 32);

/* Writing all nodes */
if( fwrite(nodes, sizeof(node), nnodes, fin) != nnodes )
         ExitError("when writing nodes to the output binary data file", 32);

/* Writing sucessors 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 )
         ExitError("when writing edges to the output binary data file", 32);
}

fclose(fin);
```

Reading:

```
void ExitError(const char *miss, int errcode) {
   fprintf (stderr, "\nERROR: %s.\nStopping...\n\n", miss); exit(errcode);
}

int main (int argc, char *argv[]){
  FILE *fin;
  unsigned long nnodes;
  node *nodes;
```

```
if ((fin = fopen (argv[1], "r")) == NULL)
         ExitError("the data file does not exist or cannot be opened", 11);

/* Global data −−− header */
if( fread(&nnodes, sizeof(unsigned long), 1, fin) +
    fread(&ntotnsucc, sizeof(unsigned long), 1, fin) != 2 )
         ExitError("when reading the header of the binary data file", 12);

/* getting memory for all data */
if((nodes = (node *) malloc(nnodes*sizeof(node))) == NULL)
         ExitError("when allocating memory for the nodes vector", 13);
if( (
    allsuccessors = (unsigned long *) malloc(ntotnsucc*sizeof(unsigned long))
    ) == NULL)
         ExitError("when allocating memory for the edges vector", 15);

/* Reading all data from file */
if( fread(nodes, sizeof(node), nnodes, fin) != nnodes )
         ExitError("when reading nodes from the binary data file", 17);
if(fread(allsuccessors, sizeof(unsigned long), ntotnsucc, fin) != ntotnsucc)
         ExitError("when reading sucessors from the binary data file", 18);
fclose(fin);

/* Setting pointers to successors */
for(i=0; i < nnodes; i++) if(nodes[i].nsucc) {
      nodes[i].successors = allsuccessors; allsuccessors += nodes[i].nsucc;
}
```