AStar assignment — A routing problem

Lluís Alsedà

November 21, 2013

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 the 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 takes 158.52 CPU seconds to read and store the file and create the edges. The AStar computation itself takes 5.37 CPU seconds and the solution length is 959655.33 meters. It is shorter than the one found by Google since we are minimizing distances in time.

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

Nodes specify a single point. The list of nodes is sorted with respect to keys. 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. Some data about the file:

- 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

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: 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.

To read the file I recommend to store the nodes in a vector of node structures. 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 nsucc, succdim;   // Node successors: wighted edges
  unsigned long *successors;
  double g, h;               // Node data for AStar
  unsigned long parent;
  bool OPEN, CLOSED;
} node;
```

The number of nodes is the dimension of this vector. To determine it 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`).

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.

Due to the concrete values of the keys of nodes and the jumps between consecutive (in the file) keys, it is not feasible to use the keys as indices in the vectors. There is no enough memory (in the universe?) for that. Then, when processing the `way`'s one has two keys `A|B` and has to search in the vector of nodes for the nodes with those keys to stablish the edges between them. 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 while if done using the strategy proposed below takes less than 7 seconds.

To optimize the search of the node corresponding to a given key I recommend to create a *hash table* with interval `hashinterval`. This consists in creating a vector named `hashtable` and deciding a value `hashinterval` such that `hashtable[i] = k` if and only if the node with key `k` is stored exactly in the position `i*hashinterval` in the vector of nodes. This vector is filled when reading the nodes by

```
n = nnodes/hashinterval;
if(nnodes == n*hashinterval) hashtable[n] = nodes[nnodes].id;
```

where `nnodes` denotes the number of nodes and `nodes` obviously denotes the vector of nodes. Observe that the last node is not stored in `hashtable`.

The number `hashinterval` and the dimension of the `hashtable` vector are computed as follows:

```
hashinterval = sqrt((double) nnodes) + 1UL;
hastabledim  = nnodes/hashinterval + 1;
```

so that `hastabledim*hasinterval > nnodes`.

Then, given a node with key `k` the algorithm to find the position of the corresponding node in the nodes vector consists in

1. Find `i` such that `hastable[i] <= k < hastable[i+1]`
2. Search `nodes[j]` for $j =$ `i*hasinterval`$, \ldots,$ `(i+1)*hasinterval-1` to find the node that has key `k`.

Note: The case `k >= hastable[nnodes/hasinterval]` has to be treated separately since we did not store the largest key. Also, observe that this works because the list of nodes is key-order-preserving. This is crucial!

This algorithm can be implemented as:

```
unsigned long FindNodeIndex(unsigned long id, node *nodvect, unsigned long nnod,
                            unsigned long *hashtable, unsigned long hashinterval){
  register unsigned long i, pos, start, end;

/* The folowing empty loop stops the first time that k < hastable[pos].
 * This implies that pos is the smallest integer such that hastable[pos] <= k */
  for( pos=nnod/hashinterval, start=pos*hashinterval, end=nnod;
       id < hashtable[pos] ;
       pos--, end=start, start -= hashinterval);
  if (id == hashtable[pos]) return start;
  for(i=start+1; i < end ; i++) if(id == nodvect[i].id) return i;
  return ULONG_MAX;
}
```

Explanation: The number of comparisons to find the node with a given key is at most the length of the hash list plus `hashinterval`. In what follows we denote `hashinterval` by x and the number of nodes by N. The length of the hash list is N/x. So. the maximum number of comparisons to perform in a search is $N/x + x$ (and in fact the *average* number of comparisons to perform in a search is $(N/x + x)/2$). Both functions have a minimum at $x = \sqrt{N}$.