Parallel A* project

August 23, 2022

Lorenzo Ippolito, Mattia Rosso, Fabio Mirto

Abstract

...Here the abstract...

1 Introduction: about the A* algorithm

A* is a graph-traversal and path-search algorithm. It is used in many contexts of computer science and not only. It can be considered as a general case of the Dijkstra algorithm and it achieves better performaces with respect to it. It is a Greedy-best-first-search algorithm that uses an heuristic function to guide itself. What it does is combining:

- Dijkstra approach: favore nodes closed to the starting point(source)
- Greedy-best-first-search approach: favore nodes closed to the final point(destination)

Using the standard terminology we have:

- $g(n)$: exact cost of moving from source to n
- $h(n)$: heuristic estimated cost of moving from a node n(source included) to the destination
- $f(n) = g(n) + h(n)$: in this way we are able to combine the actual cost with the estimated one

At each (main loop) iteration the node n that has the minimum $f(n)$ is examinated.

1.1 Heuristic design

Premises We are going to work with weighted oriented graph where V is the set on nodes/vertices and E is the set of edges of the form (x, y) to indicate that an oriented edge from x to y exists and it has weight $cost(x, y)$.

Heuristic properties The heurstic function represents the acutal core of the A* algorithm. It represents a prior-knowledge that we have about the cost of the path from every node (source included) to the destination.

- If we have not this prior information($h(n) = 0\ \forall n \in V$) we are turning the A* algorithm into Dijkstra (this is why A* can be considered as a more general case of Dijkstra algorithm) but we always have the guarantee of finding the shortest path.
- Admissible Heuristic: if $h(n) < cost(n, dest)\ \forall n \in V$ (so the we never over-estimate the distance to get to the destination from a node n) A* will always find the shortest path and the heuristic function is called *admissible*. The more inaccurate is the estimation the more nodes A* will need to expand (with the upper bound of expanding every nodes in the graph if $h(n) = 0$).
- Consistent Heuristic: if $h(x) \leq cost(x, y) + h(y)$ for every edge (x, y) (so the triangular inequality is satisfied) A* has the guarantee to find an optimal path without processing any node more than once.

Corner cases

- Dijkstra: As already discussed if $h(n) = 0$ for every node in the graph A* turns into the Dijkstra algorithm.
- Ideal: We would obtain a perferct behaviour in case $h(n)$ is exactly equal to the cost of moving from n to the destination (A* will only expand the nodes on the best path to get to the destination).
- Full greedy-best-first search: if $h(n) \gg g(n)$ than only $h(n)$ plays a role and A* turns into a completly greedy-best-first search algorithm.

2 A* project application

Problem definition Given a wide range of fields where the A* algorithm can be applied we have choosed the one of optimal path searching in geographical areas where the goal is to find the minimum distance path from a node source to a node destination.

Notation We work with a weighted oriented graph G that is made of nodes $n \in V$ that represents road-realated points of interest and edges $(x, y) \in E$ represent the unidirectional connections among these points. Each edge (x, y) is associated to a weight that is the great-circle distance between x and y measured in meters.

Benchmark We will exploit the DIMACS benchmark to make robust estimates of the designed algortithms. Starting from the FIPS system format files provided we have adopted them (as better explained in section ...) to provide to the algorithms a file containing information structured as:

- Nodes: each node n is defined as $(index, longitude, latitude)$ where $index$ is a natural progressive number starting from 0 used to univocally identify the node and $(longitude, latitude)$ are the geographical coordinates of the node.
- Edges: each edge (x, y) is defined as $(x, y, weight)$ that represent a unidirectional connection from x to y (a road) with length $weight$ (great-circle distance from x to y).

2.1 Heuristic function: the great-circle distance

As previoulsly discussed the A* needs an admssible and consistent heuristic to properly work and this function is tipically

*correspondence: email@institution.edu

Publication doi

Preprint doi

problem-specific. Given the type of problem we are going to apply A* to we are going to use a measure of geographical distance that extends the concept of heuclidean distance between two points: the great circle distance (that is the shortest distance over the earth surface measured along the earth surface itself). We

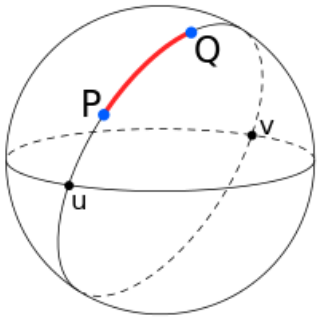

Figure 1: Great circle distance from P to Q

will employee the Haversine formula to compute the distance from node (ϕ_1, λ_1) and (ϕ_1, λ_1) where ϕ is the latitude and λ is the longitude:

$$d = R \cdot c$$
$$\text{where } c = 2 \cdot atan2(\sqrt{a}, \sqrt{1-a})$$
$$a = sin^2\left(\frac{\Delta\phi}{2}\right) + cos(\phi_1) \cdot cos(\phi_2) \cdot sin^2\left(\frac{\Delta\lambda}{2}\right)$$

R is the earth radius that we have fixed to $R = 6.371km$

3 GRAPH FILE INPUT

File input format Now we start discussing the A* alogorithm implementation and to do that we need to specify which types of files we will need to provide to the algorithm to load the graph of interest. Each file has the format:

- First line: the number of nodes $N[int]$
- N following lines: nodes appearing as $(index[int], longitude[double], latidue[double])$
- E following lines (with E unknown): edges appearing as $(x[int], y[int], weight[double])$

Random test graph We have tested the designed algorithms also a random generated graph that is build starting from:

- Which path we want to find: given the couple (source, destination) it is generated a graph of $max(source, destination) + 1$ nodes.
- How many paths at most have to be generated from source to destination
- The maximum length of these paths (that will be randomly chosen for each path)

In this way we have ad-hoc files to stress the algorithm having the guarantee that more than one path exists from source to destination (other more standart approach exist for random graph generation but we have implemented this custom one for this reason). To be consistent with benchmark files also these random graphs represents geographic points with longitude and latitude.

DIMACS benchmark The benchmark files we have used come from the DIMACS benchmark. Here each geographic map is described by:

- *.co* file: a file containing the coordinates of the nodes following the FIPS system notation
- *.gr* file: a file containing the edges and the relative weight(distance) expressed in meters

The generation of a file consistent with the format described above happens by merging these files into a new one (in binary format). One of the challenge we are going to undertake is the one of parallelizing the reading of these huge files (that we will show having an high impact in terms of execution time over the overall A* algorithm).

Test paths To analyze the performance of the different versions of A* algorithms we have used these paths: The paths on

Table 1: Test paths for A*

Nodes	Edges	Source	Dest
Random map			
101	-	0	100
California(BAY) map			
321270	800172	321269	263446
Florida(FLA) map			
1070376	2712798	0	103585

benchmark graphs BAY and FLA have been choosen ad-hoc to traverse the entire map with a reasonable long path.

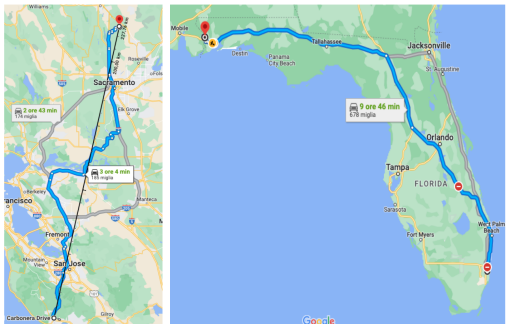

Figure 2: Test paths on BAY(left) and FLA(right)

4 SEQUENTIAL A* ALGORITHM

Sequential A* algorithm is the one we will start with to see how the algorithm works and performs. The first step consists of a pre-computation of:

- The heurstic $h(n)$ for each node (by defintion the $h(dest)$ will be 0) computed through the Haversine formula. We thus keep a global data structure `h` to do this.
- The initial values of $f(n)$ and $g(n)$ that will be set to $DOUBLE\_MAX$ for each node except for the source node that will have $f(source) = h(source)$ because

$g(soruce)$ is clearly 0. We thus keep global data structures `f` and `g` to do this. $f(source) = g(source) + h(source) = 0 + h(source) = h(source)$.

We will also need two additional global data structures:

- The global *costToCome* table (where `costToCome[i]` contains the current best cost to reach node `i`) that is initialized to *DOUBLE_MAX*.
- The global *parentVertex* table (where `parentVertex[i]` contains the parent node of node `i` according to the current best path found to reach the destination) that is initialized with -1.

The outer loop is based on the nodes extraction from a *open set*, the one containing the nodes that have still to be explored (it is implemented as a Priory Queue where the priority is associated to the $f(n)$ of the nodes). At each iteration the node a with minimum $f(a)$ is extracted from the *open set* and its neighbors are expanded: the inner loop is repeated once for each neighbor b of the extracted node a(b is neighbor of a if the edge (a, b) exists). A tentative score is computed for each node b as $g(b) = g(a) + weight(a, b)$. If $g(b)$ is less than current `g[b]` these data structures are updated:

```
g[b] = g[a] + weight(a, b)
costToCome[b] = g[b]
parentVertex[b] = a
f[b] = g[b] + h[b]
```

The final step of this inner loop is checking wheter the node b has already been added to the *open set* or not. In case the *open set* doesn't contain it we add it (with priority $f(n)$ just computed). Since the heuristic function we have chosen is both admissible and consistent we have the guarantee that the first time a node is extracted from the *open set* we have found a best path to it (this is why when the destination node is extracted we can terminate having found the best path). So despite the node may be added to the *open set* more than once (when discovered as a neighbor of different nodes) it will be only expanded once.

4.1 Pseudocode

Algorithm 1: Sequential A*

Data: Graph G(V,E), Source s, Destination d, Heurstic h
Result: Best path from Source to Destination and relative cost

$g[i] \leftarrow DOUBLE\_MAX \ \forall i \in V$;
$f[i] \leftarrow DOUBLE\_MAX \ \forall i \in V$;
$h[i] \leftarrow h(i, d) \ \forall i \in V$;
$costToCome[i] \leftarrow 1 \ \forall i \in V$;
$parentVertex[i] \leftarrow 1 \ \forall i \in V$;
$f[s] \leftarrow h[s]$;
$g[s] \leftarrow 0$;
$openSet := \{(s, f[s])\}$;
while $!openSet.EMPTY()$ **do**
  $a \leftarrow openSet.POP()$;
  if $a == d$ **then**
    $pathFound \leftarrow true$;
    reconstructPath();
  end
  foreach *neighbor b of a* **do**
    $wt \leftarrow weight(a, b)$;
    $tentativeScore \leftarrow g[a] + wt$;
    if *tentativeScore is less than* $g[b]$ **then**
      $parentVertex[b] \leftarrow a$;
      $costToCome[b] \leftarrow wt$;
      $g[b] \leftarrow tentativeScore$;
      $f[b] \leftarrow g[b] + h[b]$;
      if $!openSet.CONTAINS(b)$ **then**
        $openSet.PUSH((b, f[b]))$;
      end
    end
  end
end

4.2 Results

We are now going to run the sequential A* algorithm. We can realize that (despite for the random graph that is too small to appreciate this result) the reading time has a very high impact on the overall execution time of the algorithm and in section 6 we will investigate three different techniques for parallelizing the reading of the file.

Table 2: Sequential algorithms performance

	File Size	Reading	A*	Total	Reading Impact
RND	2876B	0.0011s	0.0862s	1.1714s	1.2872%
BAY	20.51MB	0.9365s	0.2349s	1.1714s	79.9477%
FLA	69.09MB	3.0728s	0.5893s	3.6621s	83.9075%

Table 3: Expanded nodes vs total nodes for Dijkstra and A*

Expanded nodes	Dijkstra	Sequential A*
RND	15 of 101	13 of 101
BAY	318725 of 321270	156950 of 321270
FLA	996956 of 1070376	591926 of 1070376

5 A* AND DIJKSTRA: A COMPARISON

As already mentioned the Dijkstra algorithm can be considered as a particular case of A* where we don't any prior knowledge about the distances of the nodes (heuristic $h(n) = 0 \ \forall n \in V$). We have also seen that the more precise is the heuristic function we provide the less nodes the algorithm will expand to get to the destination obtaining the best path. To investigate this point we have run Dijkstra algorithm on the same graph (here we only consider BAY and FLA) comparing the number of expanded nodes by the two algorithms:

The number of expanded nodes is clearly much higher when Dijkstra algorithm is used and the picture 3 cleary show in blue the nodes expanded by the sequntial A* algorithm while the red show the nodes expanded by the Dijkstra algorithm. In the BAY map Dijkstra expands 50% nodes more than A* while in FLA it is less evident since Dijkstra visits 38% of nodes more than A*.

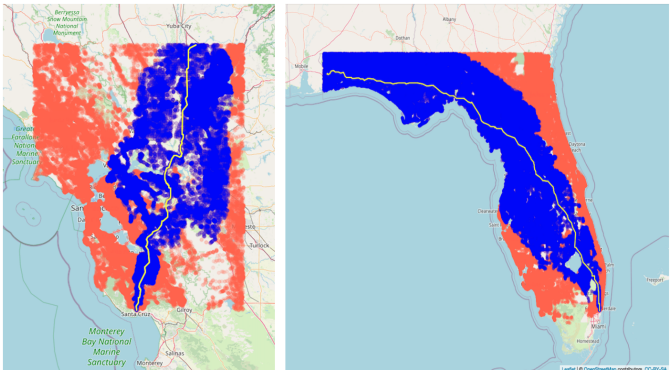

Figure 3: Test paths on BAY(left) and FLA(right)

6 PARALLEL READING OF THE INPUT FILE

(Motivation of parallel reading)

6.1 Parallel Read: approach 1

(Explanation)

6.2 Parallel Read: approach 2

(Explanation)

6.3 Parallel Read: approach 3

(Explanation)

6.4 Results

(Plots of parallel reading vs sequential reading)

7 PARALLEL A*: TWO EXAMINATED APPROACHES

(Explanation)

*7.1 First Attempt In Parallelizing A**

(Explanation + Pseudocode?)

*7.2 HDA**

(Introduction + map with colored points)

7.2.1 Message Passing Model

(Explanation + Pseudocode?)

7.2.2 Shared Address Space Model

Barrier(SAS-B) (Explanation + Pseudocode?)

Barrier(SAS-SF) (Explanation + Pseudocode?)

7.3 Results

(Plots with the comparison of all the models with sequential A* on random, BAY, FLA)

8 COMPLETE RESULTS

(Tables with numbers)

9 FINAL CONSIDERATIONS

(Comments)

10 DIMACS BENCHMARK

(More detailed explanation of the input format of the benchmarks)

11 FUTURE WORKS

(Possible improvements)

REFERENCES