

PARALLEL A* PROJECT

SEPTEMBER 7, 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. It is a Greedy-best-first-search algorithm that uses an heuristic function to guide itself. What it does is combining:

- Dijkstra approach: favoring nodes closed to the starting point(source)
- Greedy-best-first-search approach: favoring 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 to the destination.
- $f(n) = g(n) + h(n)$: combination of the actual cost with the estimated one.

At each iteration the node n that has the minimum $f(n)$ is examined(expanded).

1.1 Heuristic design

Premises We are going to work with weighted oriented graphs 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 indicated as $cost(x, y)$.

Heuristic properties The heuristic function represents the actual 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 expanded nodes 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 perfect 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 completely 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-related 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 algorithms. Starting from the FIPS system format files provided we have adapted them (as better explained in section 3) 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 $(x, y, weight)$ and it represents 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 previously discussed the A* algorithm needs an admissible and consistent heuristic to properly work and this function is typically 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 euclidean distance between two points: the great circle distance (that is the shortest

distance over the earth surface measured along the earth surface itself). We will employ the **Haversine formula** to compute

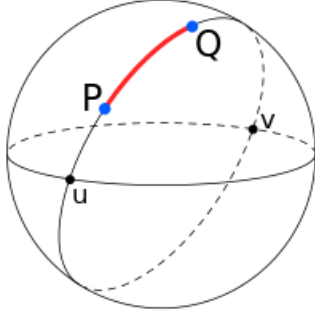


Figure 1: Great circle distance from P to Q

the distance from node (ϕ_1, λ_1) and node (ϕ_2, λ_2) where ϕ is the latitude and λ is the longitude:

$$d = R \cdot c$$

where $c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1) \cdot \cos(\phi_2) \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right)$$

R is the earth radius that we have fixed to $R = 6.371\text{km}$

3 GRAPH FILE INPUT

File input format Now we start discussing the A* algorithm 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 $N[\text{int}]$
- N following lines: nodes appearing as $(\text{index}[\text{int}], \text{longitude}[\text{double}], \text{latitude}[\text{double}])$
- E following lines (with E unknown): edges appearing as $(x[\text{int}], y[\text{int}], \text{weight}[\text{double}])$

Random test graph We have tested the designed algorithms also on a random generated graph that is built starting from these information:

- Which path we want to find: given the couple (source, destination) it is generated a graph of $\max(\text{source}, \text{destination}) + 1$ 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 approaches 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 represent geographic points identified by 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:

Table 1: Test paths for A*

| Nodes | Edges | Source | Dest |
|------------------------|----------|----------|---------|
| Random map | | | |
| 101 | - | 0 | 100 |
| California(BAY) | | | |
| 321270 | 800172 | 321269 | 263446 |
| Florida(FLA) | | | |
| 1070376 | 2712798 | 0 | 103585 |
| Western USA(W) | | | |
| 6262104 | 15248146 | 1523755 | 1953083 |
| Full USA(USA) | | | |
| 23947347 | 58333344 | 14130775 | 810300 |

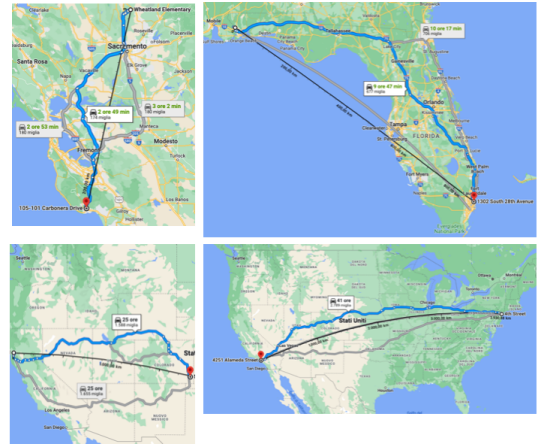


Figure 2: From left to right top to bottom BAY, FLA, W, USA

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 heuristic $h(n)$ for each node (by definition the $h(\text{dest})$ 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 $g(source)$ is clearly 0. We thus keep global data structures f and g to do this.
- The *open set* contains all the nodes that still have to be explored (at the beginning only the source nodes).
- The *closed set* contains all the nodes that have already been visited and don't have to be re-evaluated.

We will also need two additional data structures:

- The *costToCome* table (where $costToCome[i]$ contains the current best cost to reach node i) that is initialized to $DOUBLE_MAX$.
- The *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 Priority 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]
```

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 Results

We are now going to run the sequential A* algorithm. We can realize that from table 2 (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.

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 have any prior knowledge about the distances between the nodes ($h(n) = 0 \forall n \in V$).

Algorithm 1: Sequential A*

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

```
g[i] ← DOUBLE_MAX ∀i ∈ V;
f[i] ← DOUBLE_MAX ∀i ∈ V;
h[i] ← h(i, d) ∀i ∈ V;
costToCome[i] ← DOUBLE_MAX ∀i ∈ V;
parentVertex[i] ← -1 ∀i ∈ V;
f[s] ← h[s];
g[s] ← 0;
openSet := {(s, f[s])};
while !openSet.EMPTY() do
    a ← openSet.POP();
    if a == d then
        pathFound ← true;
        reconstructPath();
    end
    if a ∈ closedSet then
        CONTINUE
    end
    closedSet.PUSH(a)
    foreach neighbor b of a do
        if b ∈ closedSet then
            CONTINUE
        end
        wt ← weight(a, b);
        tentativeScore ← g[a] + wt;
        if tentativeScore is less than g[b] then
            parentVertex[b] ← a;
            costToCome[b] ← wt;
            g[b] ← tentativeScore;
            f[b] ← g[b] + h[b];
            openSet.PUSH((b, f[b]));
        end
    end
end
```

Table 2: Sequential algorithms performance

| | File Size | Reading | A* | Total | Reading Impact |
|------------|-----------|----------|----------|----------|----------------|
| RND | 2876B | 0.0011s | 0.0519s | 0.0530s | 2.1% |
| BAY | 20.51MB | 0.9538s | 0.2197s | 1.1735s | 81.3% |
| FLA | 69.09MB | 3.1551s | 0.7174s | 3.8725s | 81.5% |
| W | 394.26MB | 18.3065s | 2.5890s | 20.8955s | 87.6% |
| USA | 1292.40MB | 56.9942s | 13.6716s | 70.6658s | 80.6% |

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 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 4 clearly show in blue the nodes expanded by the sequential A* algorithm while the red show the nodes expanded by the Dijkstra algorithm. In figure 5 we can notice that the execution time is not necessary much better for A*. This could depend both on the path we are looking

Table 3: Expanded nodes in different maps

| | Dijkstra | Sequential A* |
|------------|---------------------|--------------------|
| RND | 15 of 101 | 13 of 101 |
| BAY | 318725 of 321270 | 157137 of 321270 |
| FLA | 996956 of 1070376 | 592480 of 1070376 |
| W | 5470394 of 1070376 | 1600083 of 1070376 |
| USA | 16676528 of 1070376 | 8998767 of 1070376 |

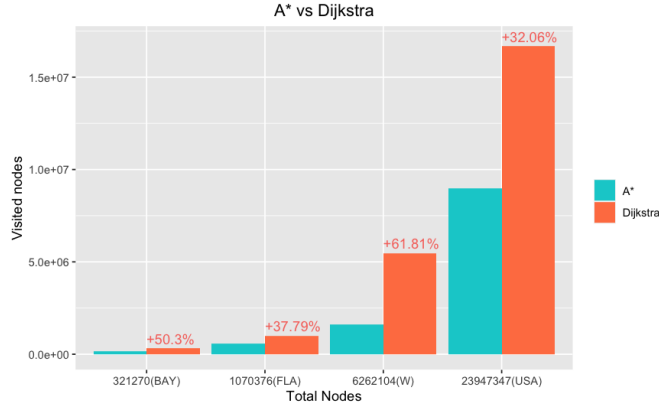


Figure 3: Expanded nodes: A* vs Dijkstra

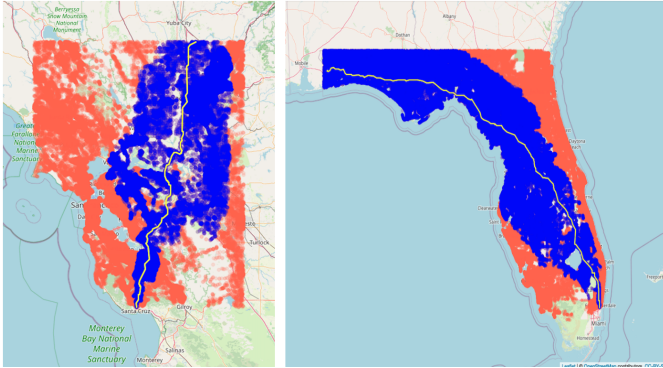


Figure 4: Test paths on BAY(left) and FLA(right)

for and on the precision of the heuristic function with respect to the cost of the optimal path. From figure 6 we realize that considering CPU and Memory usage A* needs more memory because of the higher number of data structures allocated but the less number of nodes expanded makes the CPU usage inferior w.r.t Dijkstra algorithm.

6 PARALLEL READING OF THE INPUT FILE

As we have seen in section 5 the time spent by the algorithm to read the input graph (the input binary file) used by the A* algorithm is very high w.r.t. the total amount of time spent. This is the reason why we have decided to inspect three techniques of parallelization of the reading phase to speed it up. The input file, as already discussed, is divided in two different sections (nodes and edges) so, in general, we need to take care of which section a given thread is working on because different data structures of the graph need to be loaded in the two sections (the symbol

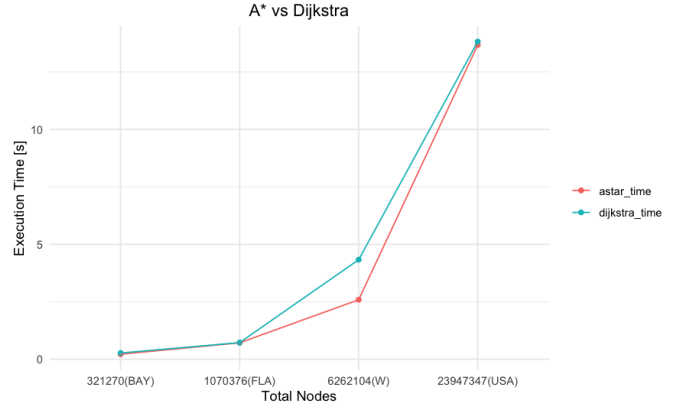


Figure 5: Execution time: A* vs Dijkstra

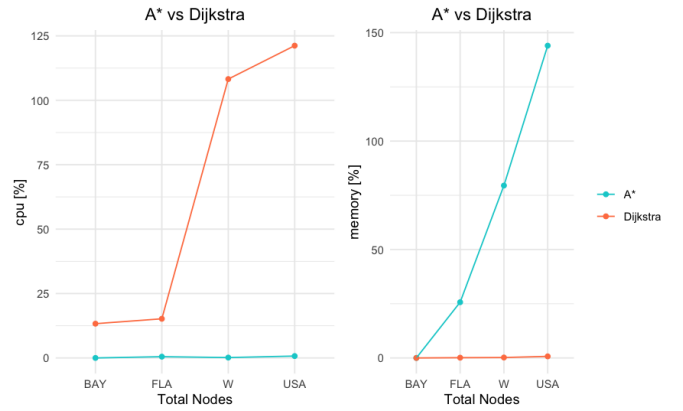


Figure 6: Exploited resources: A* vs Dijkstra

table when reading a *node line* and the linked list when reading a *edge line*).

6.1 Parallel Read: approach 1

6.2 Parallel Read: approach 2

(Explanation)

Results on FLA As we can notice from figure 7...

6.3 Parallel Read: approach 3

This is the approach is characterized by:

- Letting threads differentiate among *nodes section* and *edges section* to be able to read them together.
- Using a (NP, NT) mechanism to read each section of the input file (where NP is the number of partitions the section is divided in and NT is the number of threads that have to read all the partitions).

The (NP, NT) mechanism We need to provide as input:

- NP – *nodes*: the number of partitions of the *nodes section*

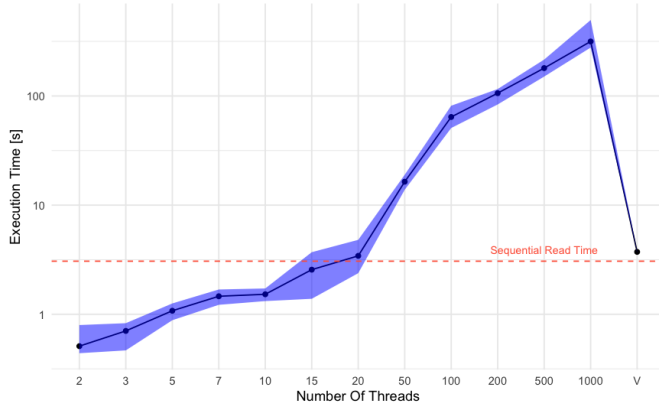


Figure 7: Performance of approach 2 for different number of threads on FLA

- $NP - edges$: the number of partitions of the *edges* section
- $NT - nodes$: the number of threads that have to read the *nodes* section
- $NT - edges$: the number of threads that have to read the *edges* section

If the number of threads is equal to the number of partitions we have a simpler mechanism in which each thread will read only one partition and then terminates. As we can see in the example in figure 8 threads will iterate over the partitions of the sections that have been statically allocated to them. This means that there is not a real contention of the reading phase since the partitions are not overlapping and each partition will be read by one and only one thread (actually, the loading of the graph data structures after reading each line must be done in mutual exclusion so we will need a lock both for the *nodes-threads* and for the *edges-threads*). Each thread terminates when it has no

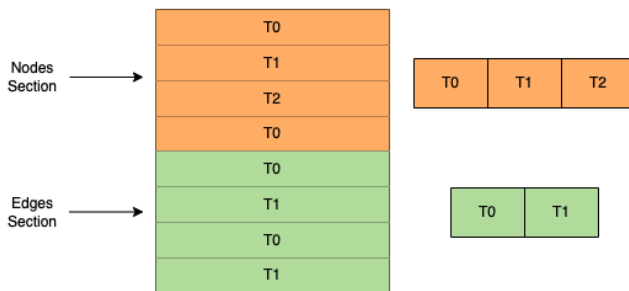


Figure 8: Example of parallel read - approach 3

more partitions to read.

Results on FLA We can realize from figure 9 that the number of partitions (setted equal both for *nodes* section and *edges* section) has not a high impact. What has more impact is the number of threads, an increasing number of threads gives increasing bad performances while with a small number of threads we are able

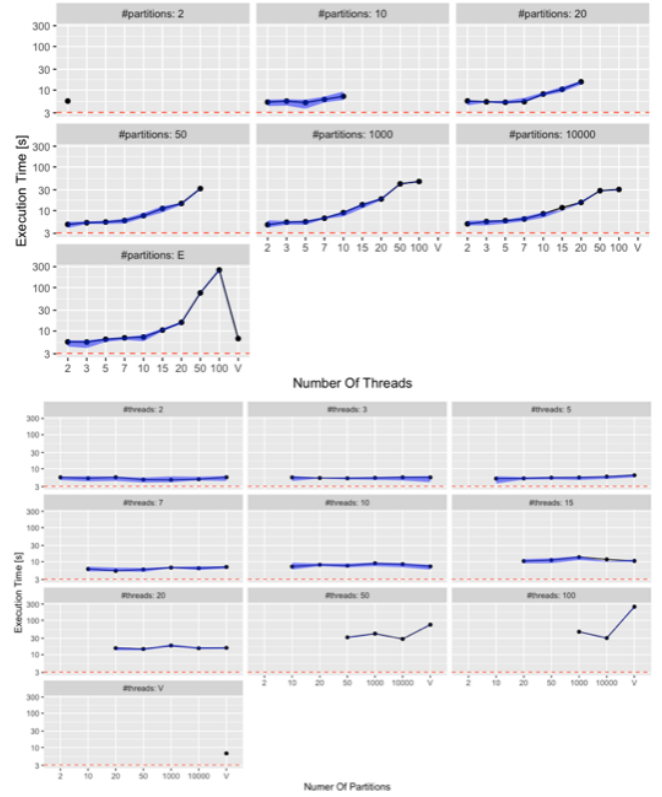


Figure 9: Performance of approach 3 for different number of threads and partitions on FLA

to obtain better results w.r.t the first approach but worse result w.r.t the sequential reading. The reason of this behaviour could be due to the fact that there is not a completely parallel work of the threads since the access to Graph's data structure has to be done in mutual exclusion so we have benefits in dividing the threads in *edges* threads and *nodes* threads (something that didn't happen in the first approach) but the mutual exclusion access to the Graph makes the overhead due to threads creation and management not being able to obtain good performances

6.4 Final results

For the sake of completeness we have also test the most promising reading approach on all the maps (BAY, FLA, W, USA) and results are showed in table 4 where the speed-up with respect to the sequential reading time is evident.

Table 4: RA2 results against sequential reading

| | RA2 (2 threads) | Sequential | Speed-Up |
|------------|--------------------|------------|----------|
| BAY | 0.1936s | 0.9366s | 79.3% |
| FLA | 0.5103s | 3.0650s | 83.4% |
| W | 14.1492s | 56.4445s | 74.9% |
| USA | 14.1492s | 56.4445s | 74.9% |

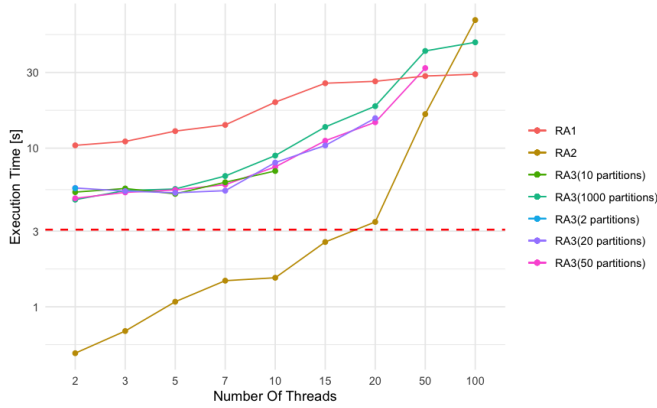


Figure 10: RA(Read Approach) 1,2,3 compared with sequential reading on FLA

7 PARALLEL A*: TWO EXAMINATED APPROACHES

The goal of the project was to find one or more parallel versions of the A* algorithm and showing their performances w.r.t. the sequential version. We have chosen two approaches to face the problem of parallelizing the A* algorithm: a first approach (known in literature as HDA*) that puts in action a complex way of parallelizing the algorithm by defining a hash-based work distribution strategy and a second approach that makes use of the parallelism in order to make the work of finding the shortest path from *source* to *dest* split between only two threads where one looks for the path from *source* to *dest* and the other one looks for the path from *dest* to *source* in the reversed graph (the PNBA* algorithm).

7.1 HDA*

The Hash-Distributed-A* (HDA*) algorithm works in a completely different way w.r.t. the first attempt algorithm. It is based on the fact that each thread is *owner* of a specific set of nodes of the Graph: given a node n it is defined a hash function $f: f(n) = t$ where $t \in \{1..N\}$ with N the number of threads. When a thread extracts from the *open set* (expands) a node all its neighbors are added to the *open set* of the owner thread of the expanded node. One important fact is that HDA* doesn't provide the same guarantees of the sequential algorithm:

- In sequential A* if it's provided an heuristic function that is both *admissible* and *consistent* we have the guarantee that each node will be only expanded once and that the first time we expand that node we have found a shortest path to it.
- In HDA* we loose these guarantees: since we don't know in which order nodes will be processed it could happen that a longer path to *dest* is found before the shortest one so a node could be opened more than once and expanding the *dest* node doesn't mean that we have terminated.

Hash Function The way how the threads divide among themselves the work to be done happens using a *hash function*. The hash function that we have employed is simply

$$\text{hash}(\text{node_index}, \text{num_threads}) = \text{node_index} \% \text{num_threads}$$

As we will realize together this fully equal work distribution could be not the optimal choice in a road network. As we can

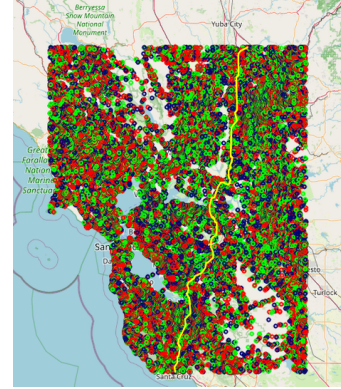


Figure 11: HDA* work distribution among 3 threads in BAY map

see in figure 11 with 3 thread the work distribution is so equal among the 3 threads that the way how each thread works for finding an optimal path to destination could be not the best choice??.

Distributed Termination Condition If in the sequential algorithm expanding the node *dest* triggers the end of the algorithm (despite the fact that the *open set* could be not empty) in HDA* this is not valid anymore: when a thread is working on its *open set* it is expanding nodes putting their neighbors in the *open set* of another thread. When thread t_i has its *open set* empty it could think to have finished its work but this might not be true because another thread t_j could be sending to t_i some nodes that have to be processed in the meanwhile. We have employed two different approaches for the distributed termination condition that are:

- Barrier method (B): when a thread realizes that its *open set* is empty a barrier is hitten and when all the threads have hitten the barrier each one makes a check to confirm (or not) that all the *open sets* of all the threads are still empty. If this is not true it means that there are nodes that have still to be processed and the best path to *dest* found so far could not be the optimal one otherwise all the threads can terminate.
- Sum-Flag method (SF): the idea behind the sum flag method comes from the fact that the Barrier mechanism could be quite expensive. In this termination condition method each thread keeps a binary flag saying whether its *open set* is empty or not. When no more nodes are inside it the flag is set and if $\sum_{i=1}^N \text{flag}[i] = N$ all the threads can correctly terminate.

Duplicate node checking The loose of guarantees explained before has also to do with the *closed set* handling. In particular when a node n is expanded by thread t_i it is done a check whether this node is inside the *closed set* of thread t_i or not. This is not sufficient: we also need to check whether the cost associated to

that node when it was previously added to the *closed set* is less or equal to the cost computed at time n is re-expanded. This is called duplicate checking so, a node n is a duplicate of node m if:

- n is equal to m
- *closedSet* of thread t_i contains the node m
- $g(m) \leq g(n)$

A duplicate node can be discarded.

Working methodology What it differentiates the algorithms we have implemented is not only the distributed termination condition (B or SF) but also the way how the threads communicate with each other. This can be done using a shared address space (SAS) approach (that will need to cope with mutual exclusion) or a message passing (MP) model.

7.1.1 Message Passing Model

One way of achieving the communication among threads is message passing. This mechanism is based on Message Queues: when thread t_i expands a node and computes (through the hash function) its owner t_j a message is sent to t_j . Since the message queue is unique for all the threads each thread needs to be able to process only messages directed to it. Each thread maintains all the data structures as private (*open set*, *parentVertex*, *costToCome*) and the only way it has to communicate with other threads is via message passing. The termination condition that has been implemented is the Barrier Method (B).

Message Structure Each message sent from t_i to t_j regarding node n contains:

- Message id: the identifier of the owner thread t_j
- The index of the node n
- The parent node of n (the one expanded from the *open set* of t_i)
- The value of $g(n)$ according to t_i
- The value of $f(n)$ according to t_i

Message Based Path Reconstruction What it makes not trivial the path reconstruction phase in the Message Passing model is that *parentVertex* and *costToCome* data structures are not shared among threads. This means that thread t_i will have inside *parentVertex_i*, once the algorithm is terminated, a consistent value of *parentVertex_i*[n] only for the nodes n it is the owner of. This implies that if we want to know which is the parent node of node n in the final path this information is stored in *parentVertex_i*[n] (where t_i is the parent of n). Path reconstruction needs to be done in a message passing fashion starting from the destination's thread owner t_d . What happens is that t_d sends a message to the owner of *parentVertex_d*[*dest*] (abbreviated as $pV_d[dest]$) that we call t_k . Immediately after t_k will send a message to $pV_k[pV_d[dest]]$ and so on and so forth till we have reached the first node of the path (the one with $pV_x[source] = -1$ where t_x is the owner of *source*).

7.1.2 Shared Address Space Model

This approach applies the explained concepts of HDA* by using a communication method among threads that exploits the shared address space (SAS). There is:

- A global array of *open sets* that here we call A where $A[i]$ contains a pointer to the *open set* of thread t_i and the size of A is N (the number of threads).
- The *parentVertex* and *costToCome* data structures are shared among all the threads.

This approach clearly requires locks so that the operations on the shared data structures can happen in mutual exclusion. In particular we need:

- One lock $L1$ for each *open set* so for each $A[i] \forall i \in \{1..N\}$.
- One lock $L2$ for each node of the graph in order to correctly update *parentVertex* and *costToCome* data structures.

With SAS both the barrier (B) and the sum-flag (SF) distributed termination conditions have been implemented to compare their performances.

7.2 Results

These are the troubles found during implementation and testing:

- The SAS-MP variant of HDA* is not scalable. This is due to the fact that we have used Linux Message Queues that have a limited size (few bytes) and when the communication overhead becomes huge the queue gets full causing an impressive slowdown. For the sake of completeness we have decided to report the results on the RND map. (RND map was indeed used almost only to show the SAS-MP performances).
- The SAS HDA* that exploits the Barrier termination condition (SAS-B) is more scalable than SAS-MP but the termination condition is not well-performing in large graphs. This happens because a thread has no way to cut off the barrier if it's receiving work from another thread but it has to wait until all the threads have reached the barrier. Despite the fact that we notice an improvement when the number of threads increases the execution time is too large from map W on.
- The SAS HDA* that exploits the Sum-Flag termination condition (SAS-SF) behaves overall better. Despite the fact on maps BAY, FLA, W it is difficult to notice the improvements when the number of threads increase this is more evident on USA map. Performances are always better compared to the SAS-B algorithm.
- TODO resource considerations (not meaningful on RND map)

HDA* has been proven not to work well in our application. We have tried to explain these results by observing that:

- The hash function we have adopted could not be optimal for this type of problem. It is happening that

instead of helping the algorithm in exploring the nodes towards the destination we are making its work more complex by randomly assigning the work to the different threads. What could be helpful would be choosing another strategy (another hash function) able to divide the work in a more reasonable way among threads (for instance trying to assign near nodes to the same threads maybe by pre-clustering the nodes of the map in a number of clusters equal to the number of threads). This could lead to obtain better performances but we suppose that the sequential algorithm would still be the "best case" instead of the "worst case" in terms of execution time.

- The termination condition is clearly a bottleneck, in particular if the Barrier method is used. This can be less evident with a small maps but we can appreciate it on bigger graphs. Anyway, having more threads improves time execution but has the drawback of consuming more memory and CPU.

7.3 Parallel Bidirectional Search

The second approach we have studied and implemented in order to parallelize the A* algorithm is a parallel version of bidirectional search.

Terminology We will refer to:

- Graph G : the original graph.
- Graph R : the reversed graph of G . The reverse graph has the property of having exactly the same nodes of G and the each edge (x, y) with weight wt has been reversed so that it becomes (y, x) and the weight wt remains the same.
- Search thread t_G : the thread that perform the search from *source* to *dest* on graph G .
- Search thread t_R : the thread that perform the search from *dest* to *source* on the reversed graph R .

7.4 New Bidirectional A*(NBA*)

The NBA* algorithm is a version of the bidirectional search that uses the data structure M to keep track of the nodes in the middle between the two searcher threads t_G and t_R . M initially contains all the nodes of the graph. The nodes in the search frontiers are the ones that:

- Belongs to M
- Have been labelled: $g_G(n) < \infty$ or $g_R(n) < \infty$

The threads t_G and t_R share a variable L initialized to ∞ that contains the cost of the best path from *source* to *dest*. Other common variables are:

- F_G : lowest f_G value on t_G frontier
- F_R : lowest f_R value on t_R frontier
- Variables F_p, f_p, g_p (with $p \in \{R, G\}$) are written on only one side but read by both sides.

These are the initialization step done by t_G (same for t_R)

- $g_G(\text{source}_G) = 0, F_G(\text{source}_G) = f_G(\text{source}_G)$

At each iteration is extracted a node x such that:

- $x \in M$
- $x : f_G(x) = \min f_G(v) \forall v \in \text{openSet}_G$

The node is removed from M and pruned (not expanded) if $f_G(x) \geq L$ or $g_G(x) + F_R - h_R(x) \geq L$. Otherwise all its successors y are generated. In the first case it is classified as *rejected* while in the other situation it is *stabilized* because $g_G(x)$ won't change anymore. For each y we update:

- $g_G(x)$: $\min(g_G(y), g_G(x) + d_G(x, y))$
- L : $\min(L, g_G(x) + g_G(y))$

The algorithm stops when no more candidates have to be expanded in one of the two sides

7.5 Parallel New Bidirectional A*(PNBA*)

The PNBA* algorithm improves the NBA* algorithm by letting the two threads working in parallel and not in an alternate mode. This requires to cope with mutual exclusion of some data. In

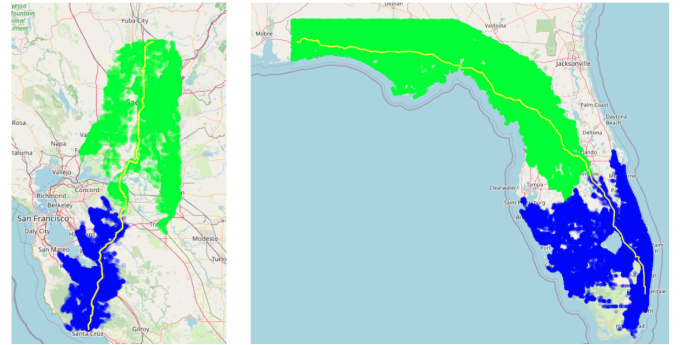


Figure 12: PNBA* work on BAY(left) and FLA(right)

figure 12 we realize that the work distribution is not equal and changes at each iteration (depending on which is the common node found).

8 COMPLETE RESULTS

9 FINAL CONSIDERATIONS

(Comments)

10 DIMACS BENCHMARK

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

11 FUTURE WORKS

(Possible improvements)

REFERENCES

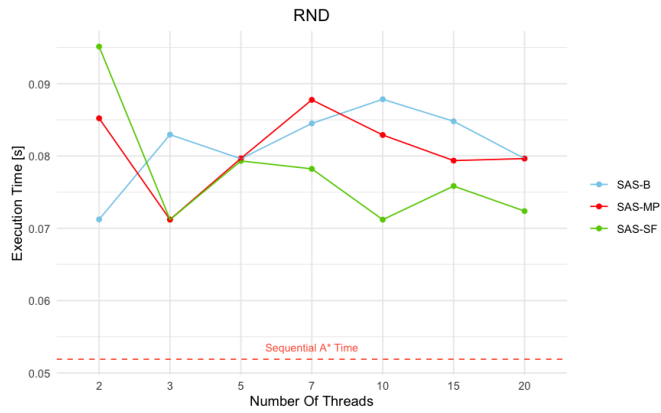


Figure 13: HDA* on RND map (only time performance)

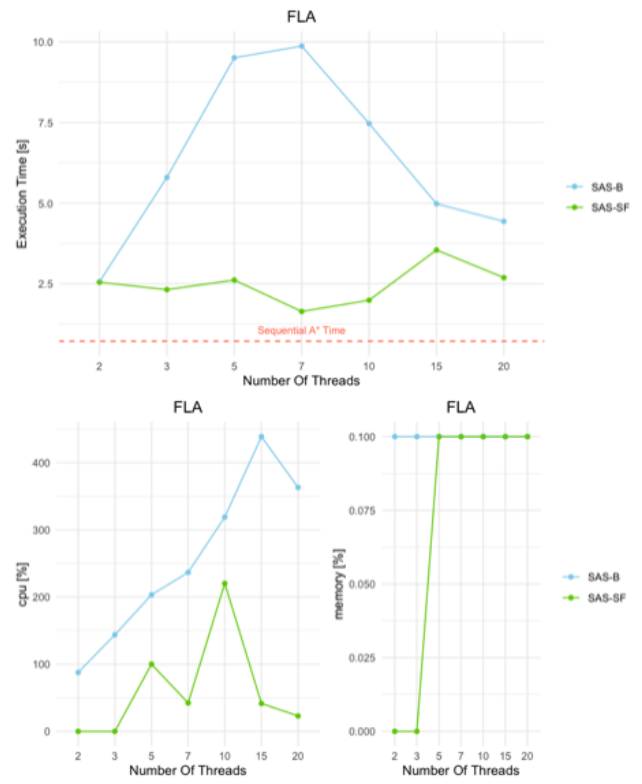


Figure 15: HDA* on FLA map

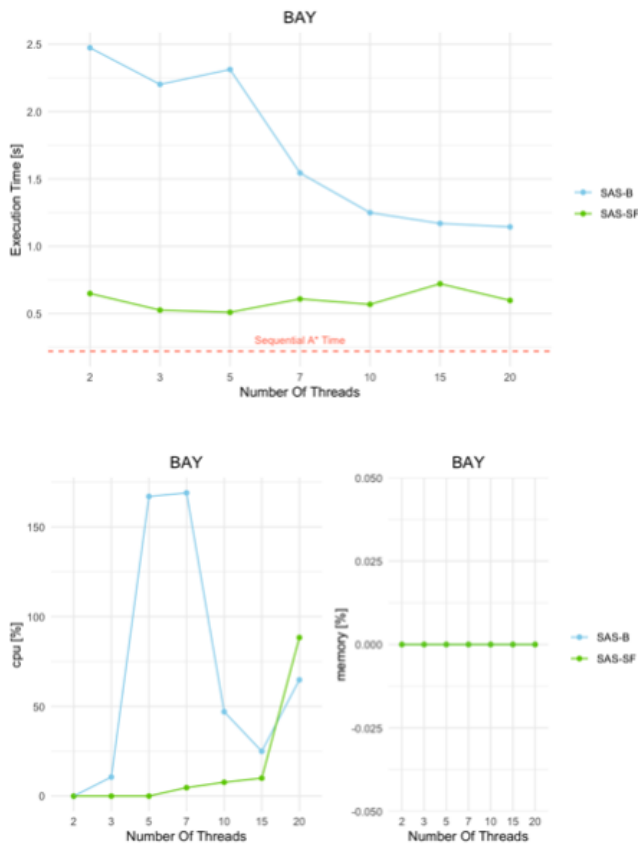


Figure 14: HDA* on BAY map

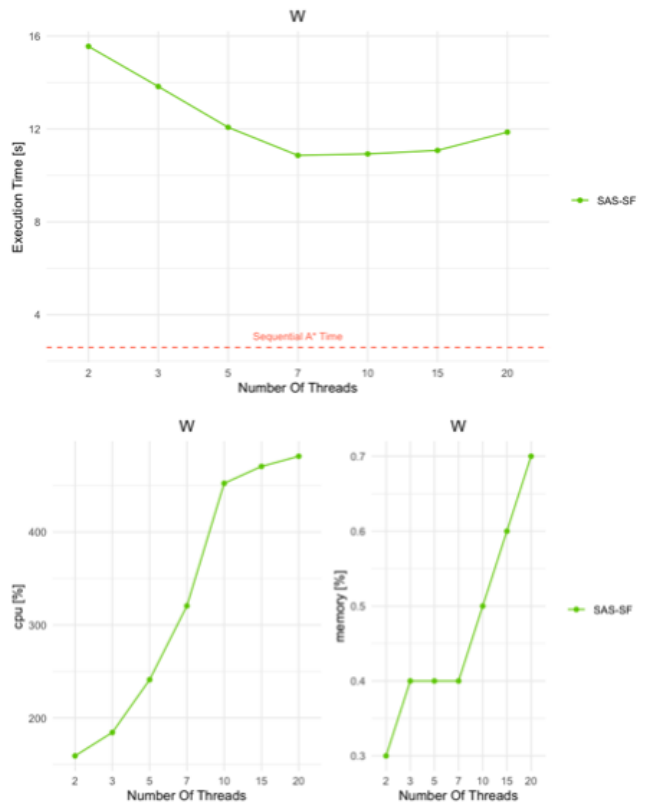


Figure 16: HDA* on W map

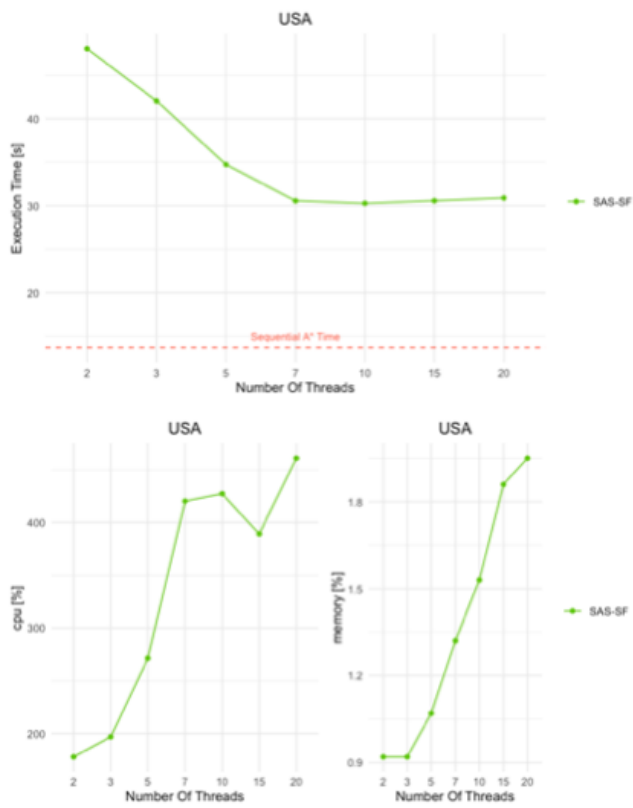


Figure 17: HDA* on USA map

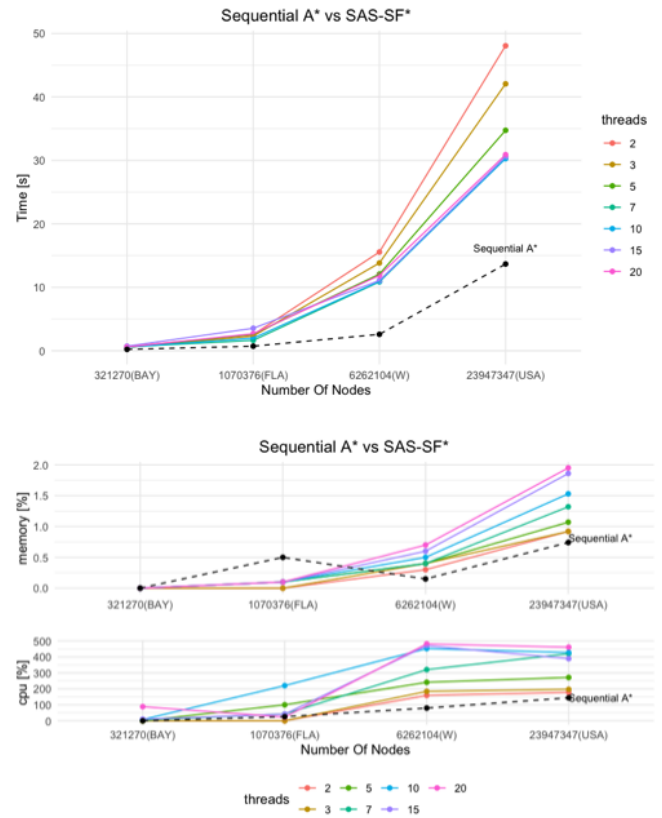


Figure 18: HDA* overall performances

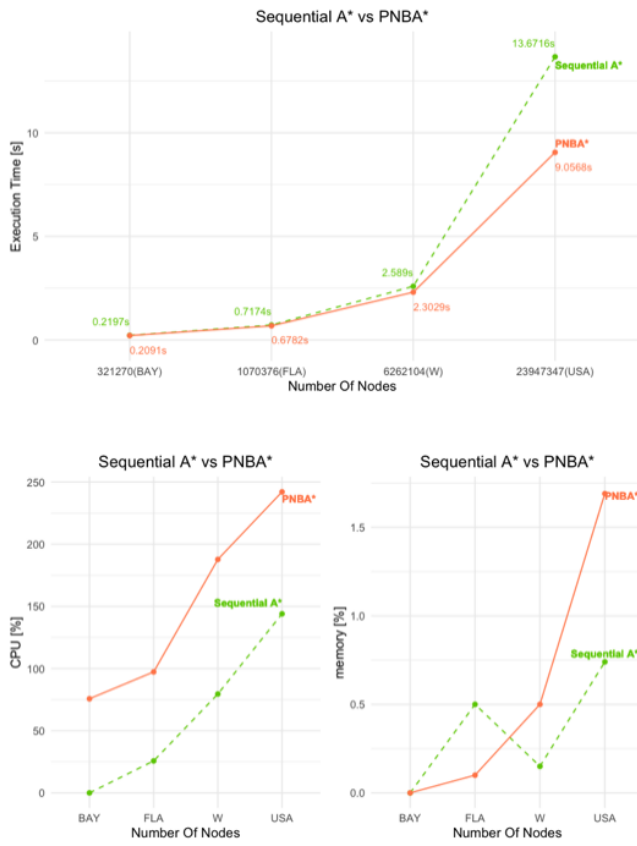


Figure 19: PNBA* overall performances