The Path-Planning Algorithm A*

System and Device Programming - Project Quer 1 - 2021/22

Giuseppe D’Andrea s303378

Mattia De Rosa s303379

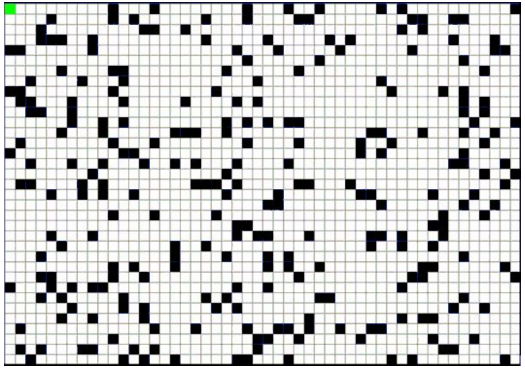
Project Objectives

- Implementation of sequential A*
- Implementation of two parallel versions of A*
- Comparing the performance of the different versions

A* Path Finding Algorithm

A* is the most famous “path finding algorithm” and can be used to find the shortest path from a source to a destination in a weighted graph.

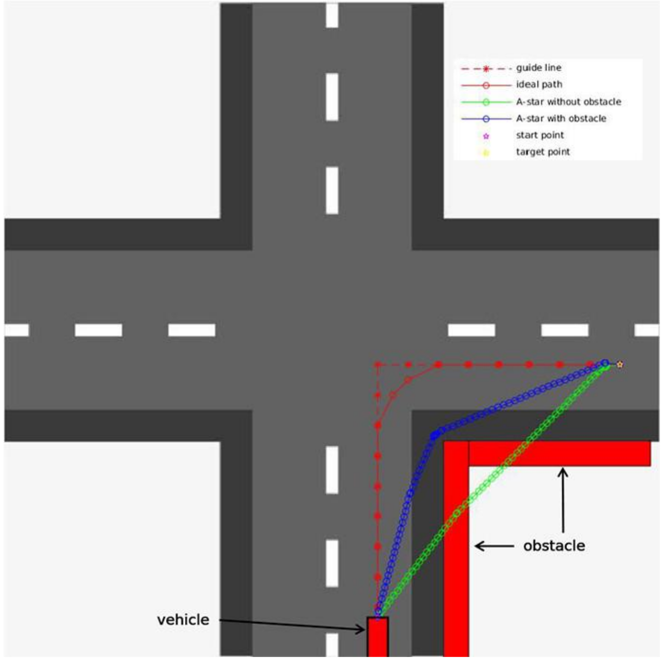
- Graph traversal and path finding algorithm
- Extension of Dijkstra’s algorithm
- Uses heuristics to improve performance



Worst-case performance O(|E|)

Applications

- Videogames
- Network routing protocols (RIP, OSPF, BGP)
- Maps
- Autonomous vehicles



*An improved A-Star based path planning algorithm for autonomous land vehicles - https://journals.sagepub.com/doi/full/10.1177/1729881420962263

Parallelization challenges

- Use of shared resources
- Termination condition
- Nodes re-expansion
- Path reconstruction

Sequential A*

A* components

- Graph G
- Source (s) and destination (d) vertices
- Heuristic function H
- openSet of vertices to be expanded
- closedSet of vertices already explored
- costToCome map of the cost to reach the node

A* iteration

Loop until openSet is empty

1. Take the element (n) with the lowest f from the openSet
   a. f(n) = g(n) + h(n)
   b. g(n) is the cost to come
   c. h(n) is cost to go (heuristic function)
2. Add all neighbors of node n to the openSet if they are not in the closedSet
3. Move n to the closedSet

Heuristic Function

An heuristic function can be

- Admissible

  If it never overestimates the goal $h(v) \le b(v, d)$ with $b(v, d)$ best path between v and d.

- Consistent

  If it follows the triangle inequality $h(u) \le w(u, v) + h(v)$

If the heuristic is admissible and consistent then the first time we reach the destination we have found the shortest path

Hash Distributed A*

HDA* main concepts

- Vertex ownership

  Every vertex is assigned to a processor using an hash function

- Multiple openSets

  When expanding a vertex the neighbors are assigned to the owner processor

HDA* issues

- It is impossible to know the order in which operations are completed.
- A higher cost path may be processed before the lowest cost one reaches the openSet.
- We may need to expand a node multiple times.
- This makes detecting a termination condition more difficult.

Termination condition

- A processor can't terminate when his own openSet is empty
  - It may receive more work from another processor
- Solution #1: send a path reached message
  - Only guarantees we reach the destination, but the best path may not be the first found
- Solution #2: terminate only when all open sets are empty
  - Explores many more nodes than needed
  - Requires synchronization (ex: barriers)
- Use both to prune nodes in the open set if they have an higher weight than the current best path

HDA* implementations

HDA* Shared Memory

- openSets and costToCome in shared memory
- Locks to protect from concurrent accesses
- Path reconstruction can be done by a single thread

HDA* Message Passing

- openSets and costToCome in local memory of each thread
- Message queues to send/receive work to/from other threads
- All threads must work together to reconstruct the path

HDA* Termination Condition

- Barriers to synchronize the check of the termination condition
- Every thread checks if its openSet is empty
- If every thread finished its work the search terminates

```
// termination condition
if (openSet.empty()) {
    barrier.arrive_and_wait();
    // set finished flag
    finished[threadId] = openSet.empty();
    barrier.arrive_and_wait();

    // check if all threads finished working, otherwise continue
    if (has_finished()) {
        break;
    } else {
        continue;
    }
}
```

HDA* Synchronization

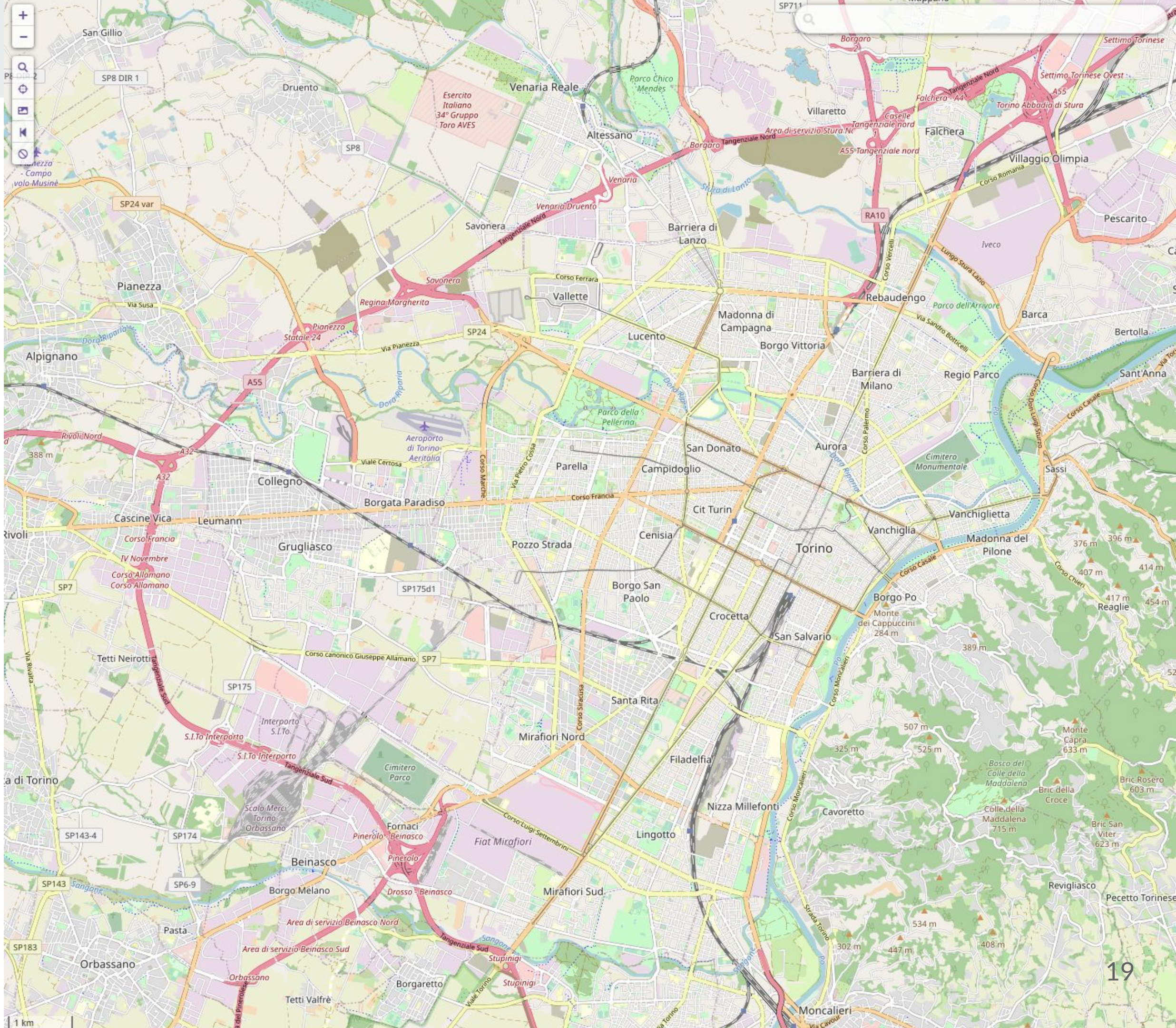
- std::barrier is used in both versions to check for the termination condition
- std::mutex is used to protect resources in the shared memory version
- std::counting_semaphore is used for path reconstruction in the message passing version

Experimental Results

Politecnico di Torino

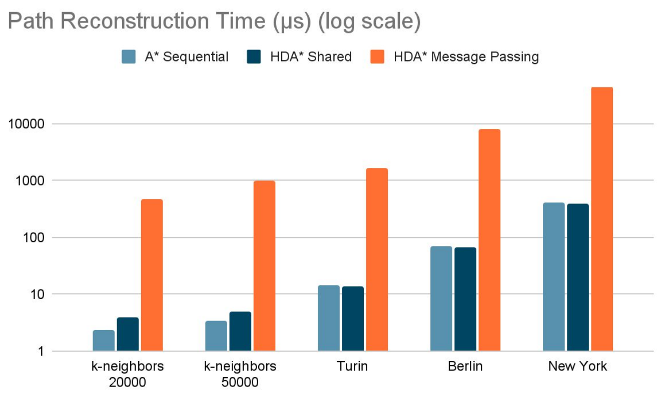
Input graphs

- K-neighbors graphs
  - 20,000 nodes
  - 50,000 nodes
- City maps exported using OpenStreetMap API
  - Turin: 100K nodes
  - Berlin: 350K nodes
  - New York: 4M nodes



Path reconstruction

- Sequential and shared memory equivalent performance
- Message passing slower than the others



Path reconstruction time (ms)	k-neighbors 20000	k-neighbors 50000	Turin	Berlin	New York
A* Sequential	0.002390	0.003493	0.014338	0.069426	0.403660
HDA* Shared	0.003957	0.005057	0.013706	0.067179	0.399261
HDA* Message Passing	0.476632	1.019622	1.674454	8.170810	44.614082

Results by type of Graph

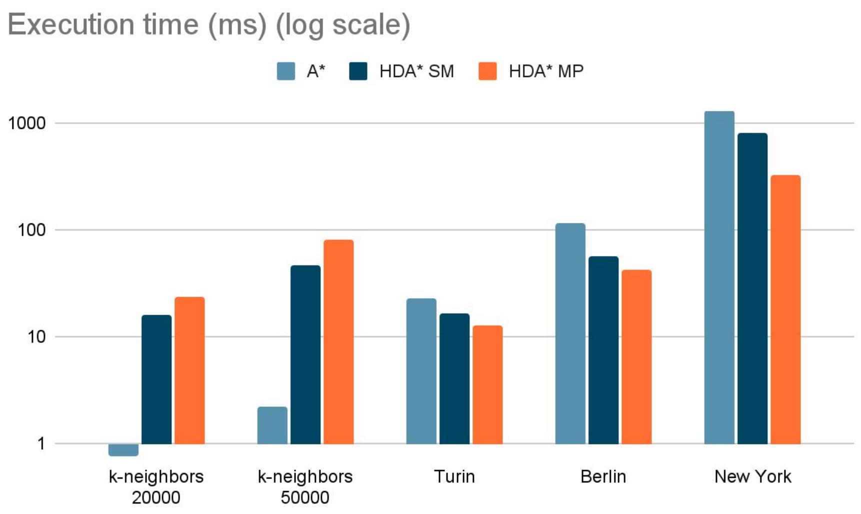
Graph differences:

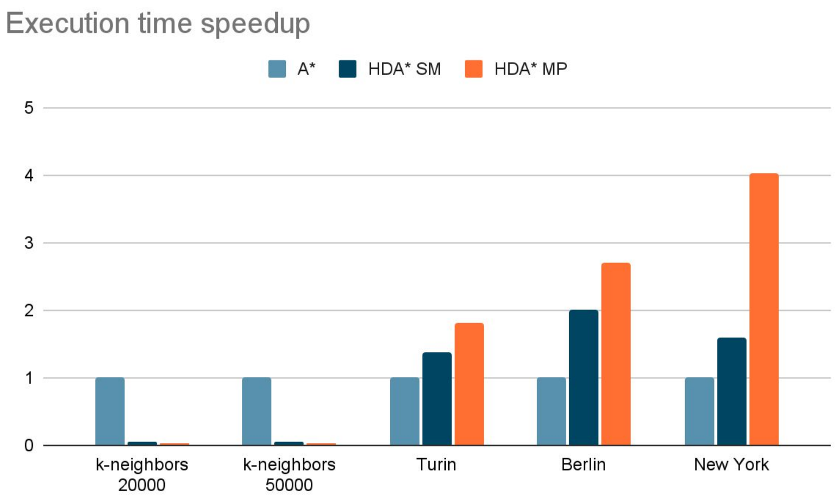
- More edges per node in the k-neighbors graphs
- Weight of edges in the 2 types of graphs

Graph info	k-neighbors 20000	k-neighbors 50000	Turin	Berlin	New York
Nodes	20,000	50,000	95,228	371,857	3,946,582
Edges	78,400	157,0551	105,173	394,062	4,189,184
Edges/Nodes	3.92	31.4	1.1	1.06	1.06

Nodes explored	k-neighbors 20000	k-neighbors 50000	Turin	Berlin	New York
A*	263	409	34,105	137,729	1,028,596
HDA* SM	11,357	18,621	67,363	250,513	3,748,860
HDA* MP	14,499	23,976	59,701	203,157	1,512,746

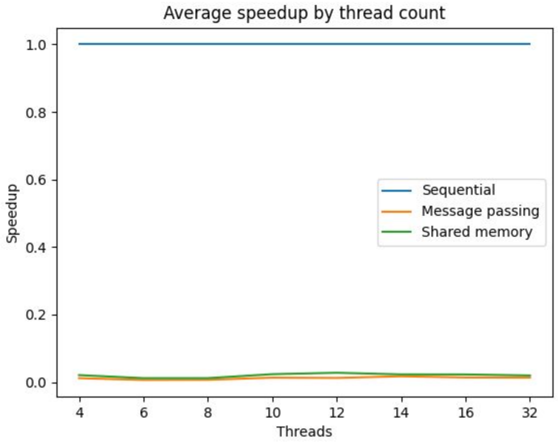
Results by type of Graph



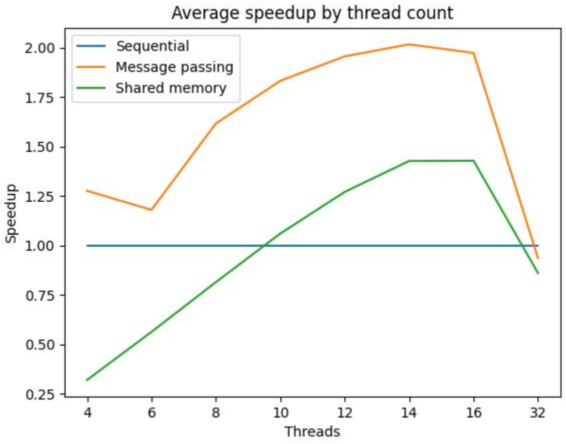


Speedup by thread count

Speedup increases with thread count until we reach the number of logical threads of the processor



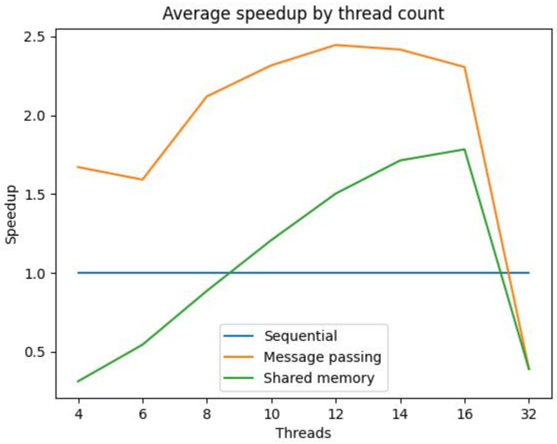
k-neighbors 50000



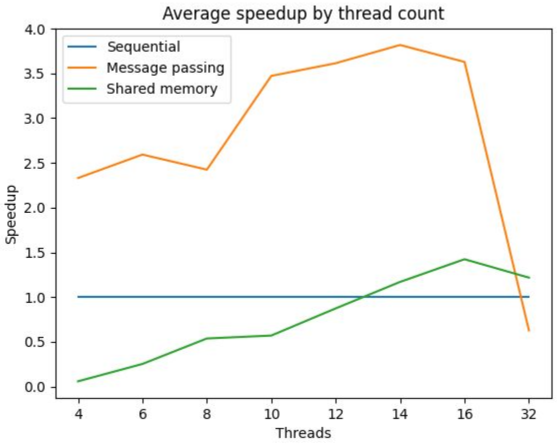
Turin

Politecnico di Torino

Speedup by thread count



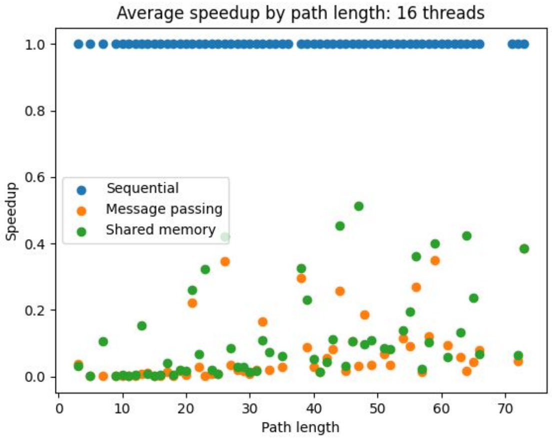
Berlin



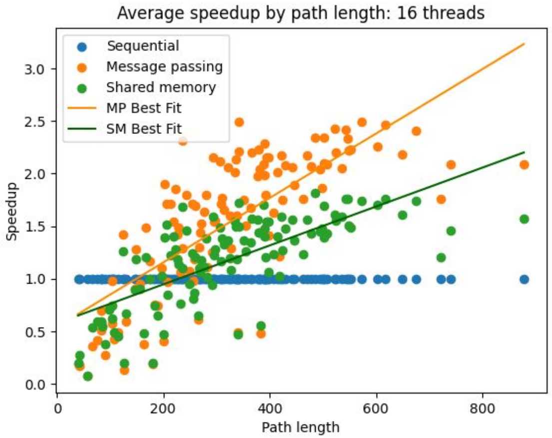
New York

Speedup by path length

Speedup increases on longer and more complex paths



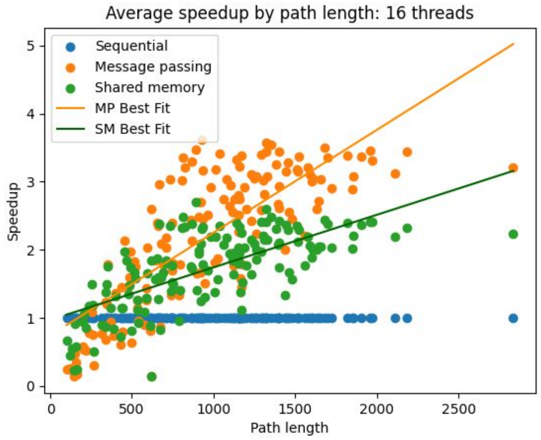
k-neighbors 50000



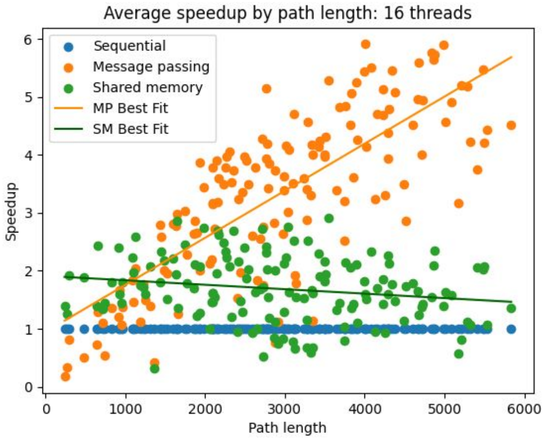
Turin

Speedup by path length

Speedup doesn't increase for very complex graphs in the shared memory implementation



Berlin



New York

Conclusions

- Parallel algorithms works better on big graphs where there is a lot of work to distribute
- Message passing performed generally better than the shared memory implementation
- If the heuristic is admissible, consistent and accurate enough to allow to explore a minimal part of the graph, the sequential algorithm could outperform the parallel

Thank you for your attention!