

PARALLEL A-STAR

Salvatore Licata
Lorenzo Ferro
Alessandro Zamparutti

August 2022

1 Introduction

Implementation of the A-Star algorithm both parallel and sequential versions on large and weighted benchmark graphs. Study on performance and comparison between the 2 implementations.

The project implies:

- reading of large weighted benchmark graphs
- run of A-Star parallel, sequential and standard Dijkstra algorithm
- comparison considering computation time and memory usage

2 Reading

Starting from a .map file (standard format for maps representation, using ascii characters to represent walls and passable terrain) the `graph_generator.py` python script is able to create a .txt with the format required by our A-Star implementation.

In the benchmark section we have:

- huge-graphs: graphs with millions of nodes, used for comparisons on large dimensions
- maze: graph with a small number of solutions (few path available from a point to another)
- street-maps: true city grid maps with different size
- small-graphs: generated by us used for testing

The weight of the edges can be randomized by the script in order to raise a level of difficulty in finding a path (all the grids benchmark have only edge with weight 1)

3 Basic structures

The project started implementing all the basic structures needed by the algorithm. All the structures are as generic as possible to encourage reuse and are dynamically allocated and are implemented using lists to have no static limitations.

Each structure have a create function for allocation and a destroy one for deallocation of the memory.

3.1 hash table

Is implemented with multiplicative modular method with golden ratio:

$$golden_ratio = (\sqrt{5} - 1)/2 \quad (1)$$

$$P = 8191 \quad (2)$$

$$hash(key, module) = ((key * golden_ratio) \% P) \% module \quad (3)$$

The structures re-allocates itself when reaches the 3/4 of the maximum capacity, in case of collisions concatenate the elements in a list (keeps in the first place the last inserted item).

3.2 heap

The heap structures uses hash table for faster addressing and is used as a priority queue with the possibility to have MAX or MIN priority as first element

3.3 queue

A generic list with Head and Tail pointers with a generic item. FIFO queue with head extraction and tail insertion

3.4 stack

LIFO implementation with both insertion and extraction on the head

3.5 graph

Generic Graph with possibility to be UNDIRECTED or DIRECTED, have a pointer to generic data (possibility to manipulate passing custom functions eg. 2d data).

Each vertex has a `true_cost` (the cost of the path from start to that vertex) and a `heuristic_cost` (exstimation of the cost from that node to the destination node). The graph also contains a Hash table of all the nodes used to efficiently find a node (a good improvement in the graph generation).

3.6 message queue

Is composed by a vector of queue, as long as the number of threads of the program. The receive and send method are protected by a mutex to avoid race conditions, and receive a Id to send a message to a specific queue. Have a function to check if all is empty used for the terminate detection.

4 A-Star Implementation

The A-Star algorithm is based on the use of a heuristic function which estimates the cost from a point to a destination node.

Both implementations are based on two set, the OPEN and the CLOSE one. The first is implemented using a priority queue (our heap 3.2) and contains nodes that have been discovered but not expanded yet. The other one with a hash table 3.1 and contains node that have been expanded.

All vertex contains a link to a parent vertex which is used to keep trace of the path found by the algorithm, at the end the solution is inserted in a stack 3.4 pushing from the goal node and going backward.

4.1 Sequential

Starting from the pseudo-code taken from (insert bibliography)

```
1 Initialize OPEN to {s_0};
2 while OPEN  $\neq$  0; do
3   Get and remove from OPEN a node n with a smallest f(n);
4   Add n to CLOSED;
5   if n is a goal node then
6     Return solution path from s_0 to n;
7   for every successor n' of n do
8     $g_1$ = g(n) + c(n; n');
9     if n'  $\in$  CLOSED then
10       if g_1 < g(n') then
11         Remove n' from CLOSED and add it to OPEN;
12       else
13         Continue;
14   else
15     if n'  $\notin$  OPEN then
16       Add $n'$ to OPEN;
17     else if $g_1$  $\geq$  g($n'$) then
18       Continue;
19   Set g($n'$) = g_1;
20   Set f($n'$) = g($n'$) + h($n'$);
21   Set parent($n'$) = n;
22 Return failure (no path exists);
```

Where

- s_0 : is the starting node
- $g(n)$: is the best known cost of the path from s_0 to n
- $h(n)$: is the heuristic function
- $f(n)$: is the sum $g(n) + h(n)$ is used in the OPEN set to calculate the priority

4.2 Parallel

Implementation of the decentralized AStar algorithm, where each thread has its own CLOSED and OPEN set, when a thread discover a neighbor node chose the thread that will expand it. When a thread find a solution compare the cost with the global one and if have a lower cost it just copies the cost (the global cost is protected with a lock).

4.2.1 Pseudo-code

```
1 Initialize OPENp for each thread p;
2 Initialize incumbent.cost =  $\infty$ ;
3 Add s0 to OPENComputeRecipient(s0);
4 In parallel, on each thread p, execute 5-31;
5 while TerminateDetection() do
6   while BUFFERp  $\neq$  0 ; do
7     Get and remove from BUFFERp a triplet (n0; g1; n);
8     if n0  $\in$  CLOSEDp then
9       if g1 < g(n0) then
10         Remove n0 from CLOSEDp and add it to OPENp;
11       else
12         Continue;
13     else
14       if n0  $\in$  OPENp then
15         Add n0 to OPENp;
16       else if g1 < g(n0) then
17         Continue;
18       Set g(n0) = g1;
19       Set f(n0) = g(n0) + h(n0);
20       Set parent(n0) = n;
21   if OPENp = ; or Smallest f(n) value of n  $\in$  OPENp  $\geq$  incumbent.cost then
22     Continue;
23   Get and remove from OPENp a node n with a smallest f(n);
24   Add n to CLOSEDp;
25   if n is a goal node then
26     if path cost from s0 to n < incumbent.cost then
27       incumbent = path from s0 to n;
```

```

28         incumbent.cost = path cost from s0 to n;
29   for every successor n0 of n do
30       Set g1 = g(n) + c(n; n0);
31       Add (n0; g1; n) to BUFFER ComputeRecipient(n);
32 if incumbent.cost = 1 then
33   Return failure (no path exists);
34 else
35   Return solution path from s0 to n;

```

Where

- *BUFFER* : is the structure for thread communication 4.2.3
- *incumbent* : the shared best known path
- *TerminateDetection* : function to check if the path search is completed 4.2.4
- *ComputeRecipient* : function to chose the thread that will expand the node 4.2.2

4.2.2 Compute recipient

A Function to map the nodeId into a thread identifier we have 2 implementations. The basic module computed as $node_id \% number_of_threads$. A implementation of multiplicative hashing:
 $h(node_id) = ((a * node_id + b) \% p) \% number_of_threads$
 where:

- p : prime large number (we used $INT_MAX = 2^{31} - 1$)
- a : random integer $[1, p-1]$
- b : random integer $[0, p-1]$

InsertBiblio?

4.2.3 Thread communication

A message queue 3.6 structure is used to allows threads to communicate: a thread compute the recipient and send a message using the thread destination with the node and cost.

4.2.4 Terminate condition

When a thread has his OPEN set empty or there isn't elements that will improve the cost of the path sets his termination_flag. A thread waits on his condition variable if have no message pending and his termination flag is set, if all other threads are waiting the program terminates instead.

```

1 lock_mutex
2 while termination_flags[this_thread] and
  BUFFER(this_thread) = 0 and not program_terminated
3   if(counter  $\geq$  n_threads - 1 and all buffer empty
4     program_terminated = 1
5     signal all condition variables
6   else
7     counter++
8     wait on cv of this_thread
9     counter--
10 mutex_unlock
11 if(program_terminated)
12   return true
13 else
14   return false

```

Where

- *termination_flags*: array of flags setted by a thread when the OPEN set is empty or doesn't have node with a minor cost then the actual path
- *program_terminated*: flag setted by last thread awake when detects termination of the program
- *counter*: counter of sleeping threads
- *mutex*: one mutex to protect all condition variables
- *BUFFER*: message queue of each thread

5 Summary

Put your summary text here.

References

6 Appendix

Put your appendix text here.