COMP3006 Report

Informed Beam and Simplified Memory Limited A * Search Implementations

Luke Healy - 17086424

Curtin University
Science and Engineering
Perth, Australia
May 2017

Informed Beam Search

Implementation Details

The informed beam search works as follows.

- For each node in the frontier, discover successor nodes in a breadth first fashion.
- Select the best k successors based on the heuristic provided.
- For each node in the new k length frontier, repeat.

This search was quite easy to implement. After getting an ordinary breadth first search working, I simply had to slice the newly generated frontier down to k nodes. I implemented cycle detection by never generating a new successor which is already in the path. The search does not backtrack, as it is breadth first.

An interesting feature of the search is that the beams do not communicate. This means that the same node may exist in more than one place in the frontier, as they produce different paths.

When a solution is found, it can continue searching until it explores the whole search tree. It will print the current state of all beams when a solution is found.

Testing

The logic and correctness of the search was tested on 4 graphs. 3 of which are from the practical sessions, another was engineered to test edge cases. Such edge cases are as follows.

- The same node being reached on the same iteration, but by different paths.
- No solution existing when a k of 1 used, but solutions with k of 2 or higher.

The search was also tested on larger graphs. The sizes range from 50 to 20000 nodes, and 219 to 500000+ edges respectively. As the graphs were randomly generated and quite large, the correctness was not verified, but the solutions returned looked reasonable.

Problems and Bugs

The informed beam search did not present any issues, other than exposing the limitations of Python 2.7's *deepcopy()* function on huge graphs. The recursive copying caused the maximum recursion limmit to be reached. With this increased, the copy simply took far too long. The remedy for this was to implement my own deepcopy in the node class. This fixed the issue well.

Simplified Memory Limited A* Search

Implementation Details

A pseudo code generalisation of my implementation can be found appendix 1. I will discuss the design decisions in the Problems and Bugs section as it is easier to explain from the perspective of problems and errors.

Testing

This search was tested for correctness on the same 4 graphs that the beam search was tested on. It was also given the large generated graphs. All of these graphs but one presented no issues. I also ran the search on 250 randomly generated 100x100 400 node graphs. All searches succeeded. I noticed that some took a lot longer than others.

Problems and Bugs

The SMA* search presented many problems to overcome. They are presented below.

- **Calculating the *f-cost*.** The *f-cost* of a given node is calculated using the following A * formula. $f(n) = g(n) + h(n)$ The actual used *f-cost* is the maximum of this value, and the parents *f-cost*. In summary; $f(n) = max(g(n) + h(n), f(n'))$. This garuantees that the *f-cost* is consistent, even with an inconsistent heuristic. In order to achieve a correct *f-cost*, the cumulative path cost must be available for every node. This cannot be stored in the node, as it may have an alternate path with a different cost. The way in which I'm able to calculate the cost, is be setting a nodes parent when it is generated by the successor function, then use parent traversal to calculate the cost. If a node is generated twice, it's parent is reset and the cost is correct.
- **Duplicate nodes, arriving at the same node from a different path.** As touched on in the previous point, the algorithm must handle successor nodes which already exist in the *open* queue. My implementation effectively overwrites the existing node with the new one, re-allocating the parent, recalculating its cost etc. This is fine as the queue is then sorted and the node is put into it's correct place. If SMA* wants to rediscover the previous path getting to the replaced node, it will do so by default, overwriting it again.
- **Backup of *f-cost* in ancestor nodes.** There was some ambiguity as to how and when a node should back up its best estimate *f-cost*. My implementation backs up the *f-cost* of it's best successor, when it has no more successors to generate. This could be because it has already generated them all, or it is a leaf node. It only backs the *f-cost* up one depth, and does not do it recusively as done in some pseudo code sketches of SMA*. I did try this if course, but it lead to sub-optimal solutions as the *f-cost* of the source node became infinite as soon as a solution was found. This does not happen when I backup non recursively. This does lead to a longer search which I will discuss later.
- **Rolling back nodes.** When rolling back a node, we must ensure that it doesn't exist in any other nodes path. To do this, my implementation only ever rolls back leaf nodes, a leaf node being a node with no successors generated, even if it has children. Without this, my implementation would roll back the source node for example, but it still existed in a solutions path.
- **Successor function and how a child qualifies as valid.** The successor function is absolutely critical for this algorithm. My implementation will only work as long as a generated successor has not been generated this iteration, nor is it in the current path. This sounds obvious, but

took some trial and error. I also tried checking if the successor is already in the *open* queue, and ignoring it if it was. This didn't work as nodes would not be re-explored. I also decided that I must set the parent in this function as well, rather than in the main code. This is so the path cost is always correctly calculated, as previously mentioned.

- **The *open* queue and what it should contain.** The *open* queue of my implementation contains every node that is in the search tree at a given instant. This means the entire path, and as many successors as we can afford. Many pseudo code descriptions have two or three queues. I found it much easier to manage with everything in one queue. S.Russel's pseudo code explanation describes removing the parent from the open queue once all of its children are in memory. I'm not sure how this can work however as you will forget how you got to the current path.
- **Cycle detection.** I avoid cycle detection by using my smart successor function. If the node is in the current path, don't consider it as a successor as its cost must be higher than the first time we saw it in the current path.
- **Time complexity and time to complete.** My implementation finishes nice an quickly (a few seconds) for a small graph (less than 400 nodes and 4000 edges). For some large graphs of 500000+ edges, it also finishes quickly. Some graphs however, cause an extreme amount of "thrashing" where the search must explore and re-explore many paths. This is a known issue of SMA*. (russels paper) On graphs of this nature, the more memory the search has, the *worse* it thrashes. This is because less memory will abandon more paths early as they exceed the maximum depth.
- **Number of iterations.** As touched on previously, the number of iterations that the search performs sometimes increases with the more memory the search has. This is counter intuative, and not usefull, but I believe that it makes sense because with less memory, the search will abandon more paths early.
- **Continuation after solution found.** This was acheived by simply checking the *open* queue at each iteration and storing the path if the goal exists. When the goal becomes the best option, you know you have found all optimal paths. This is an exit condition of the search.
- **When to stop the main loop.** The search gives up looking when the source node has an *f-cost* of infinity. This means that no solution exists shallow enough to find.

Appendix 1

- Graph generator used for testing: `https://github.com/LukeHealy/graph-gen`
- SMA* pseudo code.

```
while source.f != INF do
    best = open[0]
    if goal_test(best) then
        return best
    end if
    successor = best.next_successor()
    if successor != null then
        successor.set_depth()
        if !goal_test(best) and successor.depth == MAX_DEPTH then
            succ.f = INF
        else
            succ.f = max(best.f, pathcost(succ) + succ.h)
        end if
    else
        if best is not a leaf then
            best.f = min(best.successors.f)
            reset best.successors
        else
            best.f = INF
            open.remove(best)
            open.sort()
        end if
    end if
    if best.depth == MAX_DEPTH then
        open.remove(worst node not in path)
    end if
    if successor != null and successor not in open and successor.f != INF then
        open.append(successor)
    end if
    open.sort()
end while
```