Pruning

04 November 2020 20:40

The preceding algorithms can be improved by taking into account multiple paths to a node. We consider two pruning strategies. The simplest strategy is to prune cycles; if the goal is to find a least cost path, there is no use considering paths with cycles. The other strategy is only ever to consider one path to a node, and to prune other paths to that node.

Cycle Pruning

- The search can prune a path that ends in a node already on the path, without removing an optimal solution
- In depth-first methods, checking for cycles can be done in a constant time in path length
- For other methods checking for cycles can be done in linear time in path length

A simple method of pruning the search while guaranteeing that a solution will be found in a finite graph, is to ensure that the algorithm does not consider neighbours that are already on the path from the start.

Cycle pruning checks whether the last node on the path already appears earlier on the path from the start node to that node. Paths where the end node is already in the path are not added to the frontier, or are discarded when removed from the frontier.

The complexity of cycle pruning depends on which search method is used. For DFS, overhead can be constant if a hash function is used that sets a bit when the node is in the path. Alternatively, for methods that have exponential space, cycle pruning takes time linear in the length of the path being searched. These algorithms cannot do better than simply searching up the initial path being considered, checking to ensure that they do not add a node that already appears in the path.

Multiple Path Pruning & A*

- prune a path to a node if we've already found a path to that node
- This is done using a closed list of nodes at the end of expanded nodes
- When a path is selected, if the end node is in the closed list then the path is discarded, otherwise the node is added to the closed list and the algorithm proceeds as before..

Multiple-path pruning is implemented by maintaining an explored set (**closed list**) of nodes that are at the end of paths that have been expanded. The explored set is initially empty. When a path is selected, if the node at the end is already in the closed list then we can discard the path. Otherwise, we add the node at the end to the closed list and the algorithm proceeds.

This does not guarantee that we don't discard the least cost path. Something more sophisticated would need to be done in order to ensure the optimal solution is found. To ensure that the search algorithm can still find a lowest-cost path to a goal, we can do one of the following:

1. Make sure that the first path found to any node is a lowest-code path to that node, then prune all subsequent paths found.
2. If the algorithm finds a lower-cost path, remove all paths that used the higher-cost path to the node
3. Whenever the search finds a lower-cost path to a node than a path to that node already found, it could incorporate a new initial section on the paths that have extended the initial path.

- Suppose path p' to n' was selected, but there is a lower-cost path to n'. Suppose this lower-cost path is via path p on the frontier
- Suppose path p ends at node n
- p' was selected before p (i.e. $f(p') \leq (p)$), so: $cost(p') + h(p') \leq cost(p) + h(p)$
- Suppose $cost(n, n')$ is the actual cost of a path from n to n'. The path to n' via p is lower cost than via p' so: $cost(p) + cost(n, n') < cost(p')$
- From these equations: $cost(n, n') < cost(p') - cost(p) \leq h(p) - h(p') = h(n) - h(n')$
- We can ensure this doesn't occur if $|h(n) - h(n')| \leq cost(n, n')$.

Problem: What if a subsequent path to n is shorter than the first path to n?

1. Ensure this doesn't happen by making sure that the shortest path to a node found first
2. Remove all paths from the frontier that use the longer path

A* does not guarantee that when a path to a node is selected for the first time it is the lowest cost path to that node. Note that the admissibility theorem guarantees this for every path to a goal node but not for every path to any node. Whether it holds for all nodes depends on the properties of the heuristic function.

Consistent Heuristics

A non-negative function h(n) on node n that satisfies the constraint

$h(n) \leq cost(n, n') + h(n')$ for any two nodes n' and n, where $cost(n, n')$ is the cost of the least-cost path from n to n'.

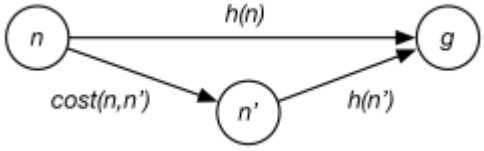
We can guarantee that a heuristic is consistent if it satisfies the monotone restriction

$h(n) \leq cost(n, n') + h(n')$ for any arc $\langle n,n' \rangle$

It is easier to check the monotone restriction as it only depends on the arcs, whereas consistency depends on all pairs of nodes.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if h(m) - h(n) <= cost(m,n) for every arc <m,n>
- If h satisfies the monotone restriction it is consistent meaning h(n) <= cost(n,n') + h(n') for any two nodes n and n'
- A* with a consistent heuristic and multiple path pruning always finds the shortest path to a goal
- This is a strengthening of the admissibility criterion



Consistency and the monotone restriction can be understood in terms of the triangle inequality which specifies that the length of any side of a triangle cannot be greater than the sum of lengths of the other two sides.

With a consistent heuristic, multiple-path pruning can never prevent A search from finding an optimal solution.*

This can be proved using proof by contradiction (see end of 3.7.1)

A* **search** in practice includes multiple-path pruning; if A* is used without multiple-path pruning, the lack of pruning should be made explicit. It is up to the designer of a heuristic function to ensure that the heuristic is consistent, and so an optimal path will be found.

Multiple-path pruning is **preferred over cycle pruning for breadth-first methods** where virtually all of the nodes considered have to be stored anyway.

Depth-first search does not have to store all of the nodes at the end of paths already expanded; storing them in order to implement multiple-path pruning makes depth-first search exponential in space. For this reason, **cycle pruning is preferred over multiple-path pruning for depth-first search**.