Search: Advanced Topics and Conclusion

CPSC 322 Lecture 8

January 24, 2007
Textbook §2.6
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming
A* Search Algorithm

- A* is a mix of lowest-cost-first and Best-First search.
- It treats the frontier as a priority queue ordered by $f(p)$.
- It always selects the node on the frontier with the lowest estimated total distance.
Let’s assume that arc costs are strictly positive.

- **Completeness:** yes.
- **Time complexity:** $O(b^m)$
  - the heuristic could be completely uninformative and the edge costs could all be the same, meaning that $A^*$ does the same thing as BFS
- **Space complexity:** $O(b^m)$
  - like BFS, $A^*$ maintains a frontier which grows with the size of the tree
- **Optimality:** yes.
In fact, we can prove something even stronger about $A^*$: in a sense (given the particular heuristic that is available) no search algorithm could do better!

Optimal Efficiency: Among all optimal algorithms that start from the same start node and use the same heuristic $h$, $A^*$ expands the minimal number of nodes.

- problem: $A^*$ could be unlucky about how it breaks ties.
- So let’s define optimal efficiency as expanding the minimal number of nodes $n$ for which $f(n) \neq f^*$, where $f^*$ is the cost of the shortest path.
Why is $A^*$ optimally efficient?

**Theorem**

$A^*$ is optimally efficient.

- Let $f^*$ be the cost of the shortest path to a goal. Consider any algorithm $A'$ which has the same start node as $A^*$, uses the same heuristic and fails to expand some node $n'$ expanded by $A^*$ for which $\text{cost}(n') + h(n') < f^*$. Assume that $A'$ is optimal.
- Consider a different search problem which is identical to the original and on which $h$ returns the same estimate for each node, except that $n'$ has a child node $n''$ which is a goal node, and the true cost of the path to $n''$ is $f(n')$.
  - that is, the edge from $n'$ to $n''$ has a cost of $h(n')$: the heuristic is exactly right about the cost of getting from $n'$ to a goal.
- $A'$ would behave identically on this new problem.
  - The only difference between the new problem and the original problem is beyond node $n'$, which $A'$ does not expand.
- Cost of the path to $n''$ is lower than cost of the path found by $A'$.
- This violates our assumption that $A'$ is optimal.
|-----------|------------------|-------------|-----------------|-------------------|-----------------------|

Search: Advanced Topics and Conclusion
Branch-and-Bound Search

- A search strategy often not covered in AI, but widely used in practice
- Uses a heuristic function: like $A^*$, can avoid expanding some unnecessary nodes
- Depth-first: modest memory demands
  - in fact, some people see “branch and bound” as a broad family that includes $A^*$
  - these people would use the term “depth-first branch and bound”
Branch-and-Bound Search Algorithm

- Follow exactly the same search path as depth-first search
  - treat the frontier as a stack: expand the most-recently added node first
  - the order in which neighbors are expanded can be governed by some arbitrary node-ordering heuristic
- Keep track of a lower bound and upper bound on solution cost at each node
  - lower bound: $LB(n) = cost(n) + h(n)$
  - upper bound: $UB = cost(n')$, where $n'$ is the best solution found so far.
    - if no solution has been found yet, set the upper bound to $\infty$.
- When a node $n$ is selected for expansion:
  - if $LB(n) \geq UB$, remove $n$ from frontier without expanding it
    - this is called “pruning the search tree” (really!)
  - else expand $n$, adding all of its neighbours to the frontier
Branch-and-Bound Analysis

- **Completeness:** no, for the same reasons that DFS isn’t complete
  - however, for many problems of interest there are no infinite paths and no cycles
  - hence, for many problems B&B is complete

- **Time complexity:** $O(b^m)$
- **Space complexity:** $O(bm)$
  - Branch & Bound has the same space complexity as DFS
  - this is a big improvement over $A^*$!

- **Optimality:** yes.
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming

Search: Advanced Topics and Conclusion
Other $A^*$ Enhancements

The main problem with $A^*$ is that it uses exponential space. Branch and bound was one way around this problem. Are there others?

- Iterative deepening
- Memory-bounded $A^*$
Iterative Deepening

- B & B can still get stuck in cycles
- Search depth-first, but to a fixed depth
  - if you don’t find a solution, increase the depth tolerance and try again
  - of course, depth is measured in $f$ value
- Counter-intuitively, the asymptotic complexity is not changed, even though we visit nodes multiple times
Memory-bounded $A^*$

- Iterative deepening and B & B use a tiny amount of memory
- what if we’ve got more memory to use?
- keep as much of the fringe in memory as we can
- if we have to delete something:
  - delete the oldest paths
  - “back them up” to a common ancestor
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming
Non-heuristic pruning

What can we prune besides nodes that are ruled out by our heuristic?

- Cycles
- Multiple paths to the same node
You can prune a path that ends in a node already on the path. This pruning cannot remove an optimal solution.

Using depth-first methods, with the graph explicitly stored, this can be done in constant time.

For other methods, the cost is linear in path length.
Multiple-Path Pruning

- You can prune a path to node $n$ that you have already found a path to.
- Multiple-path pruning subsumes a cycle check.
- This entails storing all nodes you have found paths to.
Multiple-Path Pruning & Optimal Solutions

**Problem:** what if a subsequent path to $n$ is shorter than the first path to $n$?

- You can remove all paths from the frontier that use the longer path.
- You can change the initial segment of the paths on the frontier to use the shorter path.
- You can ensure this doesn’t happen. You make sure that the shortest path to a node is found first.

- Heuristic function $h$ satisfies the **monotone restriction** if $|h(m) - h(n)| \leq d(m, n)$ for every arc $\langle m, n \rangle$.
- If $h$ satisfies the monotone restriction, A* with multiple path pruning always finds the shortest path to every node
  - otherwise, we have this guarantee only for goals
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming
The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Of course, this presumes an explicit goal node, not a goal test.
- Also, when the graph is dynamically constructed, it can sometimes be impossible to construct the backwards graph.

- **Forward branching factor**: number of arcs out of a node.
- **Backward branching factor**: number of arcs into a node.
- **Search complexity is** $b^n$. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
Bidirectional Search

- You can search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
  - The main problem is making sure the frontiers meet.
  - This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.
Lecture Overview

1. Recap
2. Branch & Bound
3. A* Tricks
4. Other Pruning
5. Backwards Search
6. Dynamic Programming
Dynamic Programming

Idea: for statically stored graphs, build a table of $dist(n)$ the actual distance of the shortest path from node $n$ to a goal. This can be built backwards from the goal:

$$dist(n) = \begin{cases} 
0 & \text{if } is\_goal(n), \\
\min_{\langle n,m \rangle \in A} (|\langle n,m \rangle| + dist(m)) & \text{otherwise.}
\end{cases}$$

This can be used locally to determine what to do.

There are two main problems:

- You need enough space to store the graph.
- The $dist$ function needs to be recomputed for each goal.

Complexity: polynomial in the size of the graph.