# CMPSCI 250: Introduction to Computation

Lecture #26: Uniform-Cost and A* Search

David Mix Barrington

30 March 2012

# Uniform-Cost and A* Search

- The All-Pairs and Single-Source Shortest Path Problems
- Priority Queues
- The Uniform-Cost Search Algorithm
- Proving Uniform-Cost Search Correct
- Using a Heuristic to Improve Search
- The A* Search Algorithm
- Examples of A* Heuristics

# Shortest Path Problems

- Today we look at the problem of finding **shortest paths** in a **labelled graph**.
- Let G be a graph (directed or undirected) where every edge is labelled with a non-negative **cost**. The simplest example is where nodes represent places and edge labels represent distances. Every path in the graph has its **total cost**, which is the sum of the labels of the edges on it.
- We could write down all the edge costs in a **single-step distance matrix**, where the entry in row u and column v is 0 if u = v, the edge cost if (u, v) is an edge, and infinity otherwise. The **all pairs shortest path problem** is to take this matrix and produce the **best-path distance matrix**, where the (u, v) entry is the length of the shortest path from u to v, or infinity if there is no path.
- This can be solved with the right sort of matrix multiplications, as you may see in CMPSCI 311. But a matrix multiplication takes $O(n^3)$ steps, which is prohibitively bad if n is, say, $10^6$.

# The Single-Source Shortest Path Problem

- A GPS navigation program, for example, is usually asked to find the best path from u to v. It turns out that the simplest algorithm to solve this problem also gives the best path from u to *all* other vertices (or at least those that are closer to u than v is). We call this the **single-source shortest path** problem.

- Our algorithm will be a variant of the generic search algorithm for directed graphs. Since we will be able to recognize previously seen nodes, it will process each vertex once and each edge once. Its running time will be O(e), where e is the number of edges. Most graphs on which you would run this algorithm are **sparse**, meaning that they have many fewer than the $O(n^2)$ edges of a complete graph. (Most nodes have only a few neighbors.) An O(e) running time is thus much better than anything that deals with all $O(n^2)$ entries of a matrix.

- Our algorithm will also use only O(n) space, as opposed to $O(n^2)$ for a matrix.

## Priority Queues

- For DFS we kept the open list as a stack, and for BFS we kept it as a queue. The simple idea for our new algorithm is to keep it as a **priority queue**.
- In a priority queue, each item stored has a **priority**, and the basic operations are to **insert** a new item and to **remove** the item with **minimum priority**. (We choose to refer to the item we want most as "minimum" priority.)
- In CMPSCI 187 we saw how to implement a priority queue with a **heap**, so that with n elements in the queue we could carry out either insertions or removals in O(log n) time, maintaining the properties of the queue.
- In Java the priority can be given by the `compareTo` method of the item's class, or by a separate `Comparator` object. In our algorithm the priority of a node will be its best-path distance from the source node.

# The Uniform-Cost Search Algorithm

- Our **uniform-cost search** algorithm is simply the generic search where the open list is kept as a priority queue that returns the node that is closest to the start node.

- We begin with the start node s in the queue.  We take s out, look at all the nodes that have edges from the start node, and insert the endpoints of those edges into the queue, marked with the length of those edges.

- In general when we take a node x off the queue, marked with its distance from the start, then (assuming it is not the goal) we look at *its* neighbors, compute the distance from s through x to each neighbor, and insert those neighbors into the queue with those distances.  There may be multiple entries in the queue for the same node -- if so we ignore all but the one closest to s.

- When the goal node g comes *off* the queue, we declare victory and report the distance from s to g from the priority of g in the queue.

# Proving Uniform-Cost Search Correct

- From our results for general search, we know that we will terminate and report victory if and only if a path to the goal node exists. The only question is whether the distance we find is really the minimum distance possible.
- Here is the important invariant. When any node x comes *off* the queue, its priority is the length of the shortest path from s to x. We prove this by strong induction on the number of nodes that have come off the queue -- P(n) will be the statement “the distance for the n’th node off the queue is correct”.
- For the base case of n = 1, the first node off is s with priority 0, and 0 is the correct distance from s to s.
- Now assume Q(n), that all the nodes already off the queue have the correct distance, and we will prove P(n+1), that the next node x’s distance is also correct.

# Completing the Correctness Proof

- Look at the priority of x when it comes off the queue. When x was put in the queue, it was given a priority which was the length of a path from s to some node y and by the edge (y, x) to x. The node y has now come off the queue, so by our assumption its priority represents the length of the best path from s to y.

- So we have the best path that goes through y to x. Could there be another shorter path from s to x that does not go through y? Suppose there is, and that its last node before x is z. The distance from s to z is smaller than the priority of x, so z must have come off the queue already and be marked with its correct distance (by the assumption).

- But when z came off, an entry was put into the queue for each of its edges, including the edge (z, x). This couldn't have happened, though, because that entry would have lower priority than our entry for x, and x would already have come off the queue. So we do in fact have the best path through any node.

## Using a Heuristic to Improve Search

- The problem with uniform-cost search is that it searches *all* nodes that are closer to the start node than our goal node. *If* we have some extra information, we can avoid doing this. In a geographical search, you would not look at driving routes from Amherst to Albany that went through Boston.

- In the case of driving routes, we know that the driving distance from x to y *cannot be shorter* than the straight-line ("as the crow flies") distance, though it could be much longer. Such a **lower bound** on the actual distance gives us a **heuristic**, a piece of information that helps guide our search although it does not give us the answer.

- We will still check all paths that have any hope of leading to the actual shortest one. But if an edge takes us far away from the goal according to the heuristic, we will delay taking that entry out of the queue until or unless we find that there is nothing better.

# The A* Search Algorithm

- We assume that we have a heuristic function $h$ such that for any node $x$, $h(x)$ satisfies the **admissibility** rule $0 \leq h(x) \leq d(x, g)$. We also have the technical requirement that $h$ be **consistent**, meaning that if there is an edge from $u$ to $v$ with cost $c(u, v)$, then $h(u) \leq h(v) + c(u, v)$.

- The **A* search algorithm** works exactly like uniform-cost search except for the priority measure in the priority queue. To process the edge $(x, y)$, in the uniform-cost search we let the priority be $d(s, x) + c(x, y)$, the distance from $s$ on the best path to $x$ and then on the edge to $y$. Now our priority is $d(s, x) + c(x, y) + h(y)$, which is our lower bound on the distance from $s$ through $x$ and $y$ to $g$, given by the heuristic's lower bound on $d(y, g)$.

- We still mark each node $x$ coming off the queue with $d(s, x)$, and essentially the same argument shows that this $d(s, x)$ value is the correct one. The priority in the queue can never be greater than the true distance $d(s, g)$.

# Examples of A* Searches

- The smallest possible admissible heuristic is the function that is always zero. In this case A* search becomes exactly the same as uniform-cost search.

- If the heuristic is as large as possible, so that $h(x) = d(x, g)$, the A* search only looks at nodes that are on the shortest path (or *a* shortest path, if there is a tie). This is of course the best possible case for finding the best path quickly.

- In the geographical setting with crow-flies distance as the heuristic, how much the A* search saves depends on how well the air distances approximate the highway distances. You would expect, for example, that the savings would be greater in flat areas than in mountainous ones.

- Whether we can benefit from A* in other circumstances depends again on how accurate the heuristic is as an estimate of the true distance. It helps by pruning the tree of possible paths, eliminating unprofitable branches.

# The 15 Puzzle

- The **15-puzzle** is a 4 × 4 grid of pieces with one missing, and the goal is to put them in a certain arrangement by repeatedly sliding a piece into the hole.

- We can imagine a graph where nodes are positions and edges represent legal moves.

- In order to move from a given position to the goal, each piece must move *at least* the Manhattan distance from its current position to its goal position. The sum of all these Manhattan distances gives us an admissible, consistent heuristic for the actual minimum number of moves to reach the goal. So an A* search will be faster than a uniform-cost search.

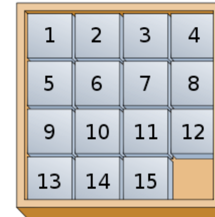


Figure from en.wikipedia.org "Fifteen puzzle"