CMPSCI 250: Introduction to Computation

Lecture #18: Search Algorithms for Graphs
David Mix Barrington
9 April 2013

Search Algorithms For Graphs

- Finding Paths in Graphs
- Matrix-Based Algorithms
- Depth-First Search
- Breadth-First Search
- Uniform-Cost Search (Dijkstra’s Algorithm)
- Using a Heuristic to Improve Search
- A* Search

Exam Review

- Question 1, Translations: Lots of “$\forall x:(x \neq a) \wedge F(a, x)$” instead of “$\rightarrow$”. Many didn’t translate “exactly one” correctly in (c).
- Question 2, graph from statements: Most got the graph right (if they didn’t I graded (b) and (c) based on their graph). Most also had the definitions right, though some said “cycle” instead of “odd cycle” in (b).
- Question 3, quantifier proof: Some people didn’t realize that with Statement II no longer assumed, you don’t know whether Cardie and Duncan are friends. That is your case breakdown -- if they are friends the third dog can be Ace, if they are not the third dog can be Biscuit. Lots of people broke into cases based on who the third dog is, when that is something you control.

More Exam Review

- Question 4, binary search: Many people made no reference to guessing or the target in the inductive part, and some even didn't in the base case. You need to say why one guess halves your range, observe that the halved range can be searched in k guesses by the IH, then conclude that the whole range can be searched in k+1 guesses total.
- Question 5, rook game: Most people described the winning strategy (which is the same as for two-pile Nim), but there weren't that many good inductions. If you use my SIH, then the base case is (i, 0) or (0, j) for any i and j. The inductive case depends on the $i = j = k+1$ case, where P1 must move to a place where P2 wins by the SIH. *Then* you can say P1 wins with $i \neq j$.
- Question 6, NYC graph: Most got the first induction. For strong connectivity, you had to argue both cases, odd and even. For the counting, I got only a few good arguments for the answer (if $k = 4m$, $f(k) = 2^{m-1}$, else $k = 0$). I gave five points for telling me how to compute the answer by matrix multiplication.

Finding Paths In Graphs

- The basic **search problem** in a directed graph is to input two vertices u and v and determine whether there is a (directed) path from u to v. We can consider the search problem in an undirected graph to be a special case, looking at the directed graph we get by replacing each edge by an arc in each direction.

- In Discussion #9 and #10 we considered the related **shortest-path problem**, where we have a directed graph labeled with a non-negative number called a **distance** or **cost** for each edge. The **length** of a path is the sum of the distances of its edges, and the **distance** from u to v is the length of the shortest path from u to v.

- It's another problem to actually **navigate**, to travel from u to v along the shortest path. If for every vertex x we have g(x), the distance from x to v, we can decide where to go next from any vertex u. We look at d(u, x) + g(x) for each x, and take the x for which this sum is smallest. The sum represents the cost of taking the edge from u to x, then the best path from x to v.

iClicker Question #1: The Triangle Inequality

- If d(x, y) represents the distance from x to y, part of what we mean by "distance" is normally the fact "**∀x:∀y:∀z: d(x, z) ≤ d(x, y) + d(y, z)**", called the **Triangle Inequality**. Which of these is a correct argument that the Triangle Inequality always holds for our distance function in directed graphs?
- (a) Because d(x, y) and d(y, z) are both non-negative, so is their sum.
- (b) In a directed graph, d(x, z) may not be equal to d(z, x).
- (c) If the graph is not strongly connected, d(x, z) may be infinite (undefined).
- (d) One way to go from x to z is to take the best path from x to y, then the best path from y to z. The shortest path has to be no longer than this path.

Matrix-Based Algorithms

- In Lecture #16 we proved the Path-Matrix Theorem, which relates matrix multiplication to properties of paths in a labeled directed graph.
- If A is a matrix whose (i, j) entry gives the label of the edge from i to j, then A^t is a matrix whose (i, j) entry gives the “sum”, over all t-step paths from i to j, of the “product” of the labels on the path. With ordinary integer “sum” and “product”, A^t_{ij} gives the number of t-step paths from i to j.
- If the labels are boolean (1 for an edge, 0 for no edge), “sum” means OR, and “product” means AND, then $A^t_{ij} = 1$ iff there is *at least one* t-step path. The matrix $(A + I)^t$ tells us whether there is a path of *t or fewer* steps.
- In Discussion #9 we redefined “sum” to be minimum and “product” to be sum, so that A^{n-1}_{ij} gave the shortest-path distance from i to j.

Depth-First Search

- In CMPSCI 187 (maybe) and CMPSCI 311, we deal with several other search algorithms that take advantage of a the graph being input as an adjacency list instead of as a matrix. All these methods share a common structure.

- Beginning with a start node, we **explore neighbors** of each new node we find, meaning that we place a record of that node on an open list. When we are done with a node we put in on a closed list. We advance the search by taking a node off the open list and exploring its (non-closed) neighbors.

- In **depth-first search (DFS)** the open list is a **stack**, and we do a **backtrack search** of all vertices reachable from the start node. If we see a node already on the open list, we don't put it on again. We'll find a path if there is one, but it may not be the shortest path.

- If we first find vertex v via the edge from u, the edge (u, v) is part of the **DFS tree** for the search. This tree has many other uses you'll see in CMPSCI 311.

Breadth-First Search

- Breadth-first search (BFS) uses the same general search strategy as DFS, but places the vertices on a **queue** rather than a stack. Again, a vertex already on the queue is not put on again when seen again.
- BFS begins by putting all the vertices at distance 1 from the start on the queue, and so they come off first. As they do, we put on the nodes of distance 2, and when they come off we put on the nodes of distance 3, and so on. We are guaranteed to find any reachable node by the shortest path (in terms of number of arcs, not any distance given by labels).
- Again, if we find v via the arc from u, that arc become part of the **BFS tree**.
- We can use BFS to test whether a simple graph is bipartite. Once we have assigned a level to each node, we can see whether any edge goes from level k to itself, rather than to level k+1. (Why couldn't it go to another level?)

iClicker Question #2: Levels in a BFS Tree

- If we do a BFS of an undirected graph, an edge (x, y) becomes part of the BFS tree if we first reach y via that edge from x. There may be other edges in the graph, but they can only connect nodes at level k to level k-1, k, or k+1. **What is the reason for this?** (Let k be the level of x, and (x, y) be the edge.)

- (a) The level of y can't be > k+1 because we'll see it from x and put it at k+1 if it isn't seen already. It can't be < k+1 because then we would have seen x from y and put it a level < k.

- (b) An edge to another level would create a cycle in the original graph.

- (c) All non-tree edges from level k must be to level k as well.

- (d) If the graph is bipartite, edges from odd levels have to go to even levels and vice versa.

Uniform-Cost Search (Dijkstra’s Algorithm)

- In Section 10.6, Rosen presents Dijkstra’s Algorithm, which is a solution to the single-source shortest path problem. Given a directed graph with non-negative distance labels, with a start node s, it gives the distance from s to each other vertex in the graph.

- We can present the same algorithm as a variant of our general search technique, using a **priority queue** for the open list. A node x goes onto the open list in a record with a label giving the distance from s to x according to the path we have just found. We don’t mark x as closed when the record goes on, because we might find a shorter path later.

- When we get a record off the open list, the priority queue gives us the one closest to s. We can prove that this record gives the shortest path to its node.

- This is called uniform-cost search because it dequeues all nodes at one distance x before it dequeues any nodes at a larger distance.

iClicker Question #3: Negative Edge Lengths

- We specified that the distance label on every edge had to be a non-negative number. Which of these is not a problem for uniform-cost search if edge labels were allowed to be negative?
- (a) A priority queue cannot hold items with negative priorities.
- (b) We cannot safely close a node when it comes off the priority queue, because there might still be a better path using negative weights.
- (c) There might not even be a shortest path, if you can get from s to g while using a negative-weight cycle.
- (d) Trick question, all three of these are serious problems.

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.

iClicker Problem #4: Manhattan Streets Again

- Suppose I have a directed graph in the form of a grid, with streets alternating one-way east and one-way west, and avenues alternating one-way north and one-way south. I want to use A* search to find the minimum number of **driving blocks** from my start intersection s to my goal intersection g, obeying the one-way restrictions. Which of these would **not** be a valid heuristic?
- (a) The Euclidean (“crow-flies”) distance to g.
- (b) The pedestrian distance to g, following streets and avenues but not obeying the one-way restriction.
- (c) A distance of one block to g for every intersection except g.
- (d) Trick question, (a), (b), and (c) are all valid.

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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Figure from en.wikipedia.org "Fifteen puzzle"