CMPSCI 575/MATH 513

Combinatorics and Graph Theory

Lecture #11: Shortest Paths in Graphs
(Tucker Section 4.1)
David Mix Barrington
30 September 2016

Shortest Paths in Graphs

- Paths in a Weighted Graph
- Dijkstra: Tucker vs. Priority Queue
- A^* Search
- APSP by Matrix Multiplication
- Paths and Matrices over a Semiring
- The Floyd-Warshall Algorithm
- Correctness of Floyd-Warshall

Paths in a Weighted Graph

- If the weights of a weighted graph represent costs, the **cost of a path** is the sum of the edge costs along the path.
- In general there are an exponential number of paths, and we want the one with the minimum cost, called the **shortest path**.
- This has many applications beyond physical distance. Weights might be currencies, with edge weights the cost of converting a sum from one currency to another.

Negative Weights

- Of course, in a non-physical situation, you might *gain* by going from one state to another, which can be modeled by **negative weights** on edges.
- Some of the algorithms we will present still work with negative weights, as long as we don't have a **negative cycle**.
- In that case we may actually not have a shortest path from one vertex to another, if there are infinitely many with increasingly negative costs.

Shortest-Path Algorithms

- It turns out that the best algorithms to find the shortest path from u to v also solve other problems at the same time.
- **Dijkstra's algorithm** (uniform-cost search) will solve the **single-source shortest path** problem, by finding the shortest path from u to each other vertex. If we only care about v, we can stop early.
- We will also present two algorithms to solve the **all-pairs shortest path** problem.

Dijkstra: Tucker vs. PQ

- The idea of Dijkstra's algorithm is to maintain a set S of vertices to which we know the shortest paths. Originally this is just u, and eventually it is all the vertices. (We assume the graph is connected, possibly directed.)
- Tucker presents a somewhat strange version in the book. He starts a counter m at 0, and increments it by ones. At each stage he looks for a node v in S and node x not in S such that $d(u, v) + c(v, x) = m$. Then he adds x to S.

Tucker's Version of Dijkstra

- Here d(u, v) is the path distance found, assumed to be optimal, and e(v, x) is the edge weight.
- We can add x to S because we know that the path from u through v to x is optimal: if there were a shorter path we would have seen it.
- What is strange is that the number of passes is proportional to the cost of the shortest path, which could be very high.

Sensible Version of Dijkstra

- In CS 250, we call the sensible version of Dijkstra's algorithm **uniform-cost search**, and place it in a framework that includes DFS and BFS.
- We keep a priority queue, whose entries are of the form (v, d, x), where v is a node in S, x a node not in S, and d the cost d(u, v) + e(v, x). The priority of the entry is d.
- At each round we pull the entry of minimum priority, and add x to S, remembering v and d.

Dijkstra: Tucker vs. PQ

- When we are done, all the nodes are in S. To find the best path from u to some node y, we look at the predecessor node in the entry we saved for y, then the predecessor of that, and so on until we get back to u.
- For each edge in the graph, we do O(1) operations plus two priority queue operations.
- If we use a heap for the PQ, our total running time is O(e log e), with e the number of edges. This is $O(n^2 \log n)$ for a dense graph.

A^* Search

- Also in CS 250, we usually present an alternate version of UCS called **A^* search**.
- This finds the same result as UCS, but may do it faster with the help of a **heuristic**, an additional function that is a lower bound on the true cost.
- The only change in the code is that the priority of the PQ is a function of both the distance found and the heuristic value.

Semirings, Paths and Matrices

- We normally represent a weighted graph as a matrix M, where the entry $M_{i,j}$ is the label on the edge from i to j. If i = j, we might have $M_{i,i} = 0$, and if there is no edge we have $M_{i,j} = \infty$.
- A solution to the APSP problem is also a matrix N, where $N_{i,j}$ is the distance from i to j along the shortest path.
- The first of our two ways to get from M to N involves **matrix multiplication**, and requires a digression.

Semirings, Paths, and Matrices

- Matrix multiplication is defined in terms of addition and multiplication of entries: If AB = C, then C_{ij} is the sum over all k of $A_{ik}B_{kj}$.
- A semiring is a structure with an "addition" operation and a "multiplication" operation, satisfying various axioms including the distributive law. We can multiply matrices over any semiring.
- Over the correct semiring, multiplication will solve our APSP problem.

Semiring Axioms and Examples

- Addition is commutative, associative, and has an identity element called 0.
- Multiplication is associative and has an identity element called 1.
- $a(b+c) = ab + bc$
- Boolean: {0, 1}, + is $\vee$, × is $\wedge$
- Naturals, integers, reals, complexes, with +, ×
- Languages + is $\cup$, × is language concatenation

The Path-Matrix Theorem

- Let S be any semiring, let G be a graph labeled with entries from S, and let M be the matrix holding these entries.
- The **Path-Matrix Theorem** says that if N is the matrix M^t, where I is the identity matrix for S, then N_{ij} is the "sum", over all paths of t edges from i to j, of the "product" of the costs along the path.
- This is easy to prove by induction on t.

Applications of Path-Matrix

- One simple semiring is the boolean set {0, 1}, where "addition" is OR and "multiplication" is AND. Then the product of edges on paths that exist is 1, and on paths that don't exist is 0. $N_{ij} = 1$ iff there exists a path of length t.
- To find the **transitive closure** of M, we compute the matrix $(M+I)^{n-1}$, where i is the identity matrix.
- Over the semiring of the naturals, N_{ij} is the number of paths of t edges from i to j.

Applications of Path-Matrix

- But what if "addition" is the minimum operation, and "multiplication" is ordinary addition?
- Then N_{ij} is the minimum, over all paths of t edges from i to j, of the total path cost.
- And $(M+I)^{n-1}$ has entries giving the length of the shortest path (with any number of edges) from i to j. (Since we have no negative edge weights, the shortest path is a simple path.)

Applications of Path-Matrix

- Suppose that our semiring is the real numbers from 0 to 1, with ordinary addition and multiplication. Let G be the graph of a **Markov chain**, so that M_{ij} is the probability of going from state i to state j in one time step.
- Then $(M^t)_{ij}$ is the probability of going from i to j in exactly t time steps. The Markov Chain Theorem says that under most circumstances, M^t approaches a constant matrix as $t \to \infty$.

The Floyd-Warshall Algorithm

- Matrix multiplication is simple, but for the boolean and min-plus semirings there is another method that gets us the same result with fewer operations.

```
for (int k=1; k <= n; k++)
    for (int i=1; i <= n; i++)
        for (int j=1; j <= n; j++)
            d[i,j] = d[i,j] + d[i,k]*d[k,j];
```

- In either case we update d[i,j] if we find a better result by combining d[i,k] and d[k,j].

The Floyd-Warshall Algorithm

- Clearly this is $O(n^3)$ time. Warshall proposed this as a means to find the transitive closure of a relation (the boolean case) and Floyd adapted it to shortest paths.

```
for (int k=1; k <= n; k++)
   for (int i=1; i <= n; i++)
      for (int j=1; j <= n; j++)
         d[i,j] = d[i,j] + d[i,k]*d[k,j];
```

- But why does it work?

Correctness of F-W

- After k steps of the outer loop, we claim that d[i,j] represents the cost of the best path from i to j that uses only {1,…,k} as **intermediate vertices**.
- Clearly at the start, a single edge is the best path that uses no intermediate vertices.
- A path using {1,…,k+1} either uses only {1, …,k} or is the concatenation of two paths, one from i to k+1 and one from k+1 to j.

Correctness of F-W

- Our innermost step takes the minimum of the cost of the best path using {1,…,k} and the best two-path combination through k+1. This preserves the invariant, and when we reach the end we have the cost of the best path using any possible intermediate vertices.
- A similar algorithm can be used to calculate the regular expression for the language of a given finite automaton, though in CS 250 and 501 we use "state elimination" instead.

FW versus Multiplication

- As we said, the F-W method takes $O(n^3)$ time, and is certainly easy to code.
- Raising a matrix to the n-1 power involves O(log n) matrix multiplications.
- A matrix multiplication takes $O(n^3)$ operations by the usual method, so we need $O(n^3 \log n)$ for the powering, worse than F-W's time.
- There are faster matrix multiplication algorithms, but they are impractical unless n is huge.