EECS 477: Introduction to algorithms.

Lecture 11

Prof. Igor Guskov
guskov@eecs.umich.edu

October 10, 2002

Lecture outline: greedy algorithms

- Graph traversals
- Dijkstra
- Making change: greedy!
- MST: Prim and Kruskal

Graph traversals

- *Graph traversal*: an algorithm that visits all the vertices/edges of a graph in some order
- While traversing we may do some work – augmenting the traversal
  - For instance, find a shortest path from A to B
- The complexity of a traversal depends on graph representation (what are those?)
- DFS and BFS

Breadth-first search

```
void BFS(G, start) {
    unmark_all_vertices();
    queue q; q.push(start);
    mark(start); d[start] = 0;
    while(!q.empty()) {
        u = q.pop_front();
        for( v adjacent to u ) {
            if(!is_marked(v)) {
                d[v] = d[u] + 1;
                mark(v);
                q.push(v);
            }
        }
    }
}
```

BFS

- BFS find shortest paths with respect to the hop distance
- BFS uses a queue, DFS uses a stack, otherwise the same
- BFS complexity $O(V + E)$

Edge-weighted graphs

- Given a directed graph $G = (N, A)$, N - nodes, A - directed edges (arrows)
- Each edge has non-negative length $L : A \to \mathbf{R}^+$
- One node is *source* node
- Find the length of the shortest path from the source to each of the nodes
- Use ∞ for convenience when not connected
- This should be similar to BFS. Except need to prioritize more carefully and we use a priority queue and add the vertex with minimal distance

Dijkstra algorithm

```
void Dijkstra(LengthFn& L) {
    vector<float> D(n); // n nodes
    set<int> C = {1,2, ..., n-1};
    for(i=1; i<n; ++i)
        D[i] = L(1,i)
    for(i=0; i<n-2; ++i) {
        v = find_min( heap on D );
        C.remove(v);
        for_all( w from C )
            D[w] = min( D[w], D[v+L(v,w)] );
    }
}
```

Dijkstra algorithm

- Proof that Dijkstra works: define $S = N \setminus C$
- Thm: all vertices will be annotated with their shortest path lengths
- Loop invariant:
  - true before we start
  - induction shows that it holds in the loop
  - after the loop is finished it gives us the correctness of the algorithm
- A. `D[i]` holds shortest path length for $i \in S$
  B. `D[i]` holds shortest *special* path length for $i \notin S$

Dijkstra algorithm

- Fact: added v is the length of the minimal shortest special path among vertices not in S
- Assume A and B, first prove A, then prove B
- A is proven by contradiction: if `D[v]` is not the shortest path, then there is a shorter *non-special* path, and its first non-S vertex would have to be chosen instead of v.
- Thus: after a vertex is chosen its `D` value becomes true shortest path length
- B is shown: the only special paths need updating that would be going via v.

Making change

- Given a set of coin values: $1, 5, 10, 25, 50, 100, \ldots$
- Need to pay a given amount A using smallest number of coins
- Algorithm: add a highest-valued coin such that the total does not exceed A
- This works for the particular problem above. It's hard to prove that it works though.

Greedy algorithms

- Problem:
  - Optimize cost (w.r.t. objective function)
  - Solutions are composed from components (candidates)
  - A *validity check* function
  - A *feasibility check* function (checks whether the partial solution can be completed to a valid solution)
- Greedy algorithm
  - goes step by step
  - maintains partial solution and possible extensions
  - *selects* the best possible extensions (how?)

Greedy verbatim

```
Greedy() {
    initialize_partial_solution();
    while(there are extensions) {
        do {
            choose the best extension;
            check its validity;
        while(it's not valid);
        add that extension to get a new partial solution
        see if the problem is solved, if so return;
    }
}
```

Minimum Spanning Trees

- or simply MSTs
- Now we consider *connected undirected* graph $G = (V, E)$
- Again edge length assignment $L : E \to \mathbf{R}^{+}$
- Find subset of edges T with minimal *total* cost
- That's got to be a tree (since a cycle can drop any of its edges and still stay connected)
- So we call the result an MST
- Towns and telephone network

Minimum Spanning Trees

- Candidates: edges
- Valid solutions: spanning trees
- Objective: minimum total length
- Feasible partial solutions: subgraphs with no cycles
- Selection function: depends on the algorithm

Promising

- Define: a feasible(?) set of edges is *promising* if it can be extended(?) to an optimal(?) solution
- **Lemma:** Let $B \subset N$, $B \neq N$. If T is a promising set of edges whose vertices are fully in B, and if e is the shortest edge that *leaves* B, then $T \cup \{e\}$ is promising again.
- **Proof:** Let U be a MST that contains T. Now suppose that e is not in U. Then adding e to U will create a single cycle (why?). Then since e leaves B there will be another edge e' leaving(rather entering?) B whose length is not less than that of e. Then we form $U' = U \cup \{e\} \setminus \{e'\}$ is an even better MST.

Approaches

- Sort edges, start with smallest, add when still acyclic, grow it. That is Kruskal's algorithm and that works!
- Runtime asymptotics is $\Theta(|E| \log |E|) = \Theta(|E| \log |V|)$.
- Start with a root, add branches of minimal length that do not cycle. That is Prim's algorithm and it works as well.
- Runtime asymptotics is $O(|E| \log |V|)$.

Kruskal's algorithm

```
void Kruskal(G=(V,E), length, vector& T) {
    sort E by increasing length;
    initialize union-find structure;
    E::iterator ei;
    do {
        e = (*(ei++));
        comp1 = find(e.first_vertex);
        comp2 = find(e.second_vertex);
        if(comp1!=comp2) {
            merge(comp1, comp2);
            T.push_back(e);
        }
    } while( T.size()==V.size()-1 );
}
```

Prim's algorithm

```
void Prim(G=(V,E), length, vector& T) {
    for all verts key[v] = infinity;
    key[root] = 0;
    initialize heap on key;
    while(heap not empty()) {
        u = extract-min-heap();
        for(v in adj[u]) {
            if(v in heap and length(u,v)<key[v]) {
                parent[v] = u;
                key[v] = length(u,v);
            }
        }
    }
}
```