Lecture 3

Shortest Path Algorithms

Shortest Path Problems

Find a shortest path between two specified vertices of a digraph:

SHORTEST s-t-PATH PROBLEM

Instance: A digraph $D = (V, A)$, lengths $\ell : A \mapsto \mathbb{R}$, and two vertices $s, t \in V$.

Task: Find an s-t–path of minimum length.

The **length** of a directed path is the sum of the lengths of arcs in the path.

Problem variant:
Given a source s, find the shortest paths to **all** other nodes in the graph.

Assumption:
The digraph contains a directed path from node s to every other node in the graph.

Negative Cycles and Non-Negative Lengths

Assumption:
The digraph does not contain a **negative cycle**
(i.e. a directed cycle of negative length).

Why do we exclude negative cycles?
Why don't we allow arbitrary arc lengths?

Answer: with arbitrary lengths, the problem becomes NP-hard.

If all lengths are -1, then the s-t–paths of length $1 - |V|$ are precisely the *Hamiltonian* s-t–paths.

(A path is *Hamiltonian* if it visits each vertex of the graph exactly once.)

Undirected Graphs

- Shortest paths in undirected graphs is more difficult unless the edge lengths are non-negative.
- Undirected edges of non-negative lengths can be replaced equivalently by a pair of oppositely directed edges of the same length.
- This reduction does not work for edges of negative length, as this would introduce negative cycles.
- Shortest paths in undirected graphs with negative lengths (but no negative cycles) can be solved in polynomial time via matching methods. (We will come back to this problem when we study matching problems.)

Shortest Path Trees

A **shortest-path tree** is an arborescence rooted at some source node s with the property that the unique path from s to any node is a shortest path to that node.

The existence of a shortest path tree relies on the following observations.

Fact 3.1 *If the path $v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_{k+1}$ is a shortest path from v_1 to v_{k+1}, then every subpath $v_i, e_i, v_{i+1}, \ldots, v_j$ with $1 \leq i < j \leq k+1$ is a shortest path from v_i to v_j.*

Lemma 3.2 *Let the vector $d \in \mathbb{R}^{|V|}$ represent the shortest path distances from a source node s in a digraph $D = (V, A)$ with lengths $\ell : A \mapsto \mathbb{R}_+$. Then a directed path P from s to v is a shortest path if and only if $d(w) = d(v) + \ell_a$ for every arc $a = (v, w) \in P$.*

Label-Setting vs. Label-Correcting Algorithms

- There are two groups of algorithms: label setting and label correcting.
- Both assign tentative distance labels to nodes at each step.
- These distance labels are upper bounds of the shortest path distances.
- Label-setting methods designate one label as permanent (optimal) at each iteration.
- Label-correcting methods consider all labels as temporary until the final step, when they all become permanent.
- Label-setting algorithms are applicable only to directed acyclic graphs (dags) or to graphs with non-negative arc lengths.

Dijkstra's Algorithm

Input: A digraph $D = (V, A)$, lengths $A \mapsto \mathbb{R}_+$ and a vertex $s \in V$.
Output: Shortest paths from s to all $v \in V$ and their lengths.
More precisely, $d(v)$ is the length of a shortest s-v–path;
`pred`(v) denotes the predecessor of v in a shortest path.

1. set $R := \emptyset$; set $d(v) := +\infty$ for all $v \in V$;
   set $d(s) := 0$ and `pred`(s) := `NULL`;
2. WHILE $|R| < n$ DO {
3. select $v \in V \setminus R$ such that $d(v) = \min\{d(w) \,|\, w \in V \setminus R\}$;
4. set $R := R \cup \{v\}$;
5. FOR EACH $w \in V \setminus R$ with $(v, w) \in A$ DO {
6. IF $d(w) > d(v) + \ell(v, w)$ THEN
   $d(w) := d(v) + \ell(v, w)$ and `pred`$(w) := v$;
7. } // end for each
8. } // end while

Dijkstra's Algorithm (continued)

Theorem 3.3 *Dijkstra's algorithm works correctly. Its running time is $O(n^2)$.*

Proof: Correctness: By induction. Details left out here.

Running time:

Node selections: The algorithm performs this operation n times and each such operation requires that it scans each temporarily labeled node. The total node selection time is $n + (n-1) + (n-2) + \cdots + 1 \in O(n^2)$.

Distance updates: Each arc is visited at most once. A single update costs $O(1)$ for a total of $O(m)$. □

This time bound is best possible for dense graphs with $m \in \Omega(n^2)$.

Soon we will see how it can be improved for sparse graphs.

Speed-Up Technique I: Early Termination

- Suppose that we search for a shortest path from s to some specified node t.
- We can stop the execution of the Dijkstra algorithm as soon as node t becomes permanently labeled (even though some nodes are still temporarily labeled).
- This technique leads to significant savings in practice, but does not change the worst case complexity.

Speed-Up Technique II: Bidirectional Dijkstra

- Apply simultaneously a forward search from s and a backward search from t.
- Denote by R^f the set of permanently labeled nodes in the forward search, and by R^b the set of permanently labeled nodes in the backward search.
- The algorithm alternatively selects a smallest temporarily label from $V \setminus R^f$ or from $V \setminus R^b$ and designates it as permanent until both the forward and the backward search have permanently labeled the same node, say node w (i.e., $R^f \cap R^b = \{w\}$).
- At this point, let $P(v)$ denote the shortest path from s to node $v \in R^f$ and let $P'(u)$ denote the shortest path from node $u \in R^b$ to t.
- The shortest path from s to t is either the path $P(w) \cup P'(w)$ or a path $P(v) \cup (v,u) \cup P'(u)$ for some arc $(v,u) \in A$.
- Bidirectional Dijkstra search has asymptotically the same worst case performance as ordinary Dijkstra, but is often faster in practice.

Priority Queues

A **priority queue** is a data structure which allows us to perform the following operations on a collection H of objects, each with an associated real number, called its **key**:

- create-pq(H): create an empty priority queue H.
- insert(H, x): insert the element x into H.
- find-min(H, x): find and return an object x of minimum key in H.
- delete-min(H, x): delete an object x of minimum key from H.
- decrease-key(H, x, y): decrease the key of an object x in H to the new value y .

Priority Queue Implementation of Dijkstra's Algorithm

Input and Output: as before (Slide 34)

1. create-pq(H); // use distance values $d(\cdot)$ as keys
   set $d(v) := +\infty$ for all $v \in V$;
   set $d(s) := 0$ and `pred`$(s) :=$ `NULL`;
2. insert(H, s);
3. WHILE $H \neq \emptyset$ DO {
4. find-min(H, v); delete-min(H, v);
5. FOR EACH $(v,w) \in A$ DO {
6. IF $d(w) > d(v) + \ell(v,w)$ THEN {
   IF $d(w) == +\infty$ THEN
   $d(w) := d(v) + \ell(v,w)$, `pred`$(w) := v$ and insert(H, w);
   ELSE $d(w) := d(v) + \ell(v,w)$, `pred`$(w) := v$
   and decrease-key($H, w, d(v) + \ell(v,w)$);
7. } // end if
8. } // end for each
9. } // end while

Binary Heaps

- A **binary heap** is a natural data structure to implement a priority queue.
- It stores the heap elements as a binary tree and maintains the **heap property**: For every tree node, the key value is not larger than that of its parent.
- It requires
  - $O(\log n)$ time to perform insert, decrease-key and delete-min;
  - $O(1)$ for create-pq and find-min.
- Consequently, a binary heap implementation of Dijkstra's algorithm requires $O(m \log n)$ time.
- Note that this is slower than the original implementation for very dense graphs ($m \in \Omega(n^2)$), but faster when $m \in O(n^2/\log n)$.

d-heap Implementation

- For any $d \geq 2$, the d-**heap** data structure uses d-ary trees to maintain the elements of the priority queue.
- It requires
  - $O(\log_d n)$ time to perform insert, decrease-key;
  - $O(d \log_d n)$ time to perform delete-min;
  - $O(1)$ for create-pq and find-min.
- Consequently, a d-heap implementation of Dijkstra's algorithm requires $O(m \log_d n + nd \log_d n)$ time.
- The optimal choice (worst case) of d is $d = \max\{2, \lceil \frac{m}{n} \rceil\}$.
- For very sparse graphs, i.e. $m \in O(n)$, the running time is $O(n \log n)$.
- For non-sparse graphs with $m \in \Omega(n^{1+\varepsilon})$ for some $\varepsilon > 0$, the running time is $O(m)$, because

$$O(m \log_d n) = O(\frac{m \log n}{\log d}) = O(\frac{m \log n}{\log n^\varepsilon}) = O(\frac{m}{\varepsilon}) = O(m).$$

Fibonacci Heap Implementation

- A **Fibonacci heap** is a special data structure to implement a priority queue (Fredman and Tarjan 1984).
- It requires
  - $O(\log n)$ time to perform delete-min;
  - $O(1)$ amortized time for all other priority queue operations.
- Consequently, the Fibonacci heap implementation of Dijkstra's algorithm requires $O(m + n \log n)$ time.
- This yields the currently best strongly polynomial running time for the shortest path problem in directed graphs with non-negative lengths.

Dial's Implementation

Observation 3.4 *The distance labels that Dijkstra's algorithm designates as permanent are non-decreasing.*

- Assume that the arc lengths are integer-valued and C is an upper bound on the longest arc.
- Dial's implementation of Dijkstra's algorithm stores nodes with finite temporary labels in a sorted fashion: It maintains $nC+1$ so-called **buckets** numbered $0, 1, \ldots nC + 1$.
- Bucket k stores all nodes with temporary distance label equal to k in a doubly linked list. Each node maintains a pointer to the list item in the corresponding bucket list.
- Note that nC is an upper bound on any finitely labeled node.
- Node selection: We scan buckets until we find the first non-empty bucket, say bucket k. Each node in bucket k has a minimum label.

Dial's Implementation (continued)

- Distance update: Whenever we update the distance label of node v from d_1 to d_2, we move node v from bucket d_1 to bucket d_2.
- This version of Dial's algorithm runs in $O(m+nC)$. However, its memory requirements can be prohibitively large.
- The following observation allows us to reduce the number of buckets to $C+1$:

  Observation 3.5 *If $d(v)$ is the distance label that the algorithm designates as permanent at the beginning of an iteration of the while-loop, then at the end of the iteration, $d(w) \leq d(v) + C$ for each finitely and temporarily labeled node.*

  Hence, we can store a temporarily labeled node v with distance label $d(v)$ in the bucket $d(v) \mod (C+1)$. Because of the above observation each bucket will store only nodes with the same distance label at any point of time.

Speed-Up Technique III: Goal-Directed Search

- The idea of goal-directed search is to guide the search towards the target t by incorporating lower bounds $b(w,t)$ for the distance of an arbitrary node w to t.
- Consistency condition for lower bounds $b(w,t)$:
$$\ell(v,w) + b(w,t) \geq b(v,t).$$
- With consistent lower bounds, one can use modified arc lengths
$$\ell'(v,w) := \ell(v,w) - b(v,t) + b(w,t) \geq 0.$$
- In the plane, the Euclidian distance may serve as a lower bound.
- Goal-directed search uses Dijkstra's algorithm with these modified arc lengths.
- Modified distance labels are of the form
$$d'(v) = d(v) + b(v,t) - b(s,t).$$
  Hence, the nearer v is to t, the smaller will be $d'(v)$.

A*-Search

- Goal-directed search can be very effective in practice.
- If $b(v,t)$ are not lower bounds but *estimates* for the true distance $d(v,t)$ from v to t, we cannot apply Dijkstra's algorithm anymore. (If we would, the path found by the search is in general not optimal.)
- However, with such estimates at hand, we can allow multiple scanning of the same node. This is what A*-search does. It requires exponential running time in the worst case.

Speed-Up Technique IV: Preprocessing

- Consider the following scenario: we get a large number of online queries for shortest paths from s_i to t_i in a static digraph.
- Idea: use precomputed information to reduce response time for such online queries.
- For each arc $a \in A$ let $S(a)$ be the set of nodes to which a shortest path starts which uses arc a.
- Let $C(a) \subseteq V$ be a **container** which includes $S(a)$.
- Now it is sufficient to perform Dijkstra's algorithm only on the subgraph induced by the arcs $a \in A$ with the target node t_i included in $C(a)$.
- It is crucial that the containment in a container can be tested efficiently and that the space for storing all containers (one for each arc!) is not too large.
- If the graph is embedded in the plane, the bounding box of $S(a)$ may serve as such a container.
- Empirically, this leads to a substantial speed-up.