Theory of Computing

Lecture 5

MAS 714

Hartmut Klauck

DFS

- Procedure:
  1. For all v:
     - π(v)=NIL, d(v)=0, f(v)=0
  2. Enter s into the stack S, set TIME=1, d(s)=TIME
  3. While S is not empty
     a) v=peek(S)
     b) Find the first neighbor w of v with d(w)=0:
        – push(w,S) , π(w)=v, TIME=TIME+1, d(w)=TIME
     c) If there is no such w: pop(S), TIME=TIME+1, f(v)=TIME

DFS

- The array d(v) holds the time we first visit a vertex.
- The array f(v) holds the time when all neighbors of v have been processed
- “discovery” and “finish”
- In particular, when d(v)=0 then v has not been found yet

Simple Observations

- Vertices are given d(v) numbers between 1 and 2n
- Each vertex is put on the stack once, and receives the f(v) number once all neighbors are visited
- Running time is O(n+m)

A recursive DFS

- Stacks are there to roll out recursion
- Consider the procedure in a recursive way!
- Furthermore we can start DFS from all unvisited vertices to traverse the whole graph, not just the vertices reachable from s

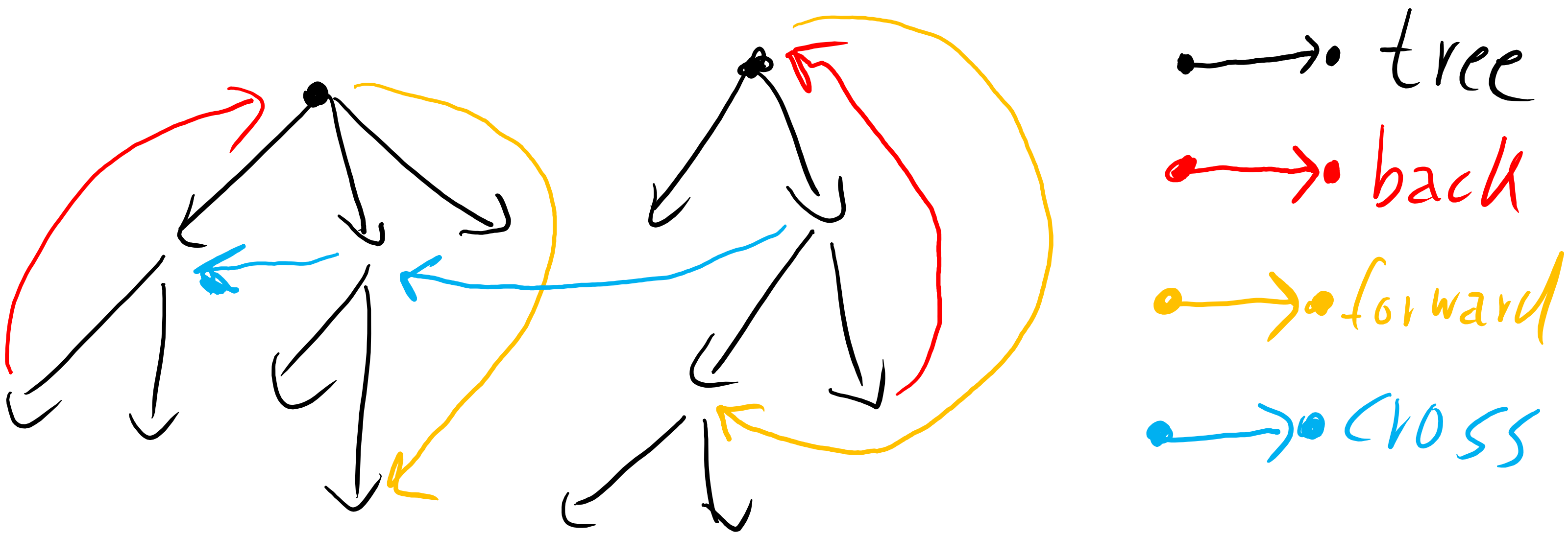
- We also want to label edges
  - The edges in (π(v),v) form trees: tree edges
  - We can label all other edges as
    - back edges
    - cross edges
    - forward edges

Edge classification

- **Lemma:** the edges $(\pi(v),v)$ form a tree
- **Definition:**
  - Edges going down along a path in a tree (but not tree edge) are *forward edges*
  - Edges going up along a path in a tree are *back edges*
  - Edges across paths/tree are *cross edges*
- A vertex v is a *descendant* of u if there is a path of tree edges from u to v
- Observation: descendants are discovered after their “ancestors” but finish before them

Example: edge labeling

- Tree edges, Back edges, Forward edges, Cross edges



Recursive DFS

- DFS(G):
  1. TIME=0 (global variable)
  2. For all v: π(v)=NIL, d(v)=0, f(v)=0
  3. For all v: if d(v)=0 then DFS(G,v)
- DFS(G,v)
  1. TIME=TIME+1, d(v)=TIME
  2. For all neighbors w of v:
     1. If d(w)=0 then (v,w) is tree edge, DFS(G,w)
     2. If d(w)$\neq$0 and f(w)$\neq$0 then cross edge or forward edge
     3. If d(w)$\neq$0 and f(w)=0 then back edge
  3. TIME=TIME+1, f(v)=TIME

Recursive DFS

- How to decide if forward or cross edge?
  - Assume, (v,w) is an edge and $f(w)$ is not 0
  - If $d(w)>d(v)$ then forward edge
  - If $d(v)>d(w)$ then cross edge

Application 1: Topological Sorting

- A DAG is a directed acyclic graph
  - A partial order on vertices
- A topological sorting of a DAG is a numbering of vertices s.t. all edges go from smaller to larger vertices
  - A total order that is consistent with the partial order

DAGs

- Lemma: G is a DAG iff there are no back edges
  - Proof:
    - If there is a back edge, then there is a cycle
    - The other direction: suppose there is a cycle c
    - Let u be the first vertex discovered on c
    - v is u's predecessor on c
    - Then v is a descendant of u, i.e., d(v)>d(u) and f(v)<f(u)
    - When edge (v,u) is processed: f(u)=0, d(v)>d(u)
    - Thus (v,u) is a back edge

Topological Sorting

- Algorithm:
  - Output the vertices in the reverse order of the f(v) as the topological sorting of G

- I.e., put the v into a list when they finish in DFS, so that the last finished vertex is first in list

Topological Sorting

- We need to prove correctness
  - Certainly we provide a total ordering of vertices
  - Now assume vertex i is smaller than j in the ordering
  - I.e., i finished after j
  - Need to show: there is no path from j to i
  - Proof:
    - j finished means all descendants of j are finished
    - Hence i is not a descendant of j (otherwise i finishes first)
    - If j is a descendant of i then a path from j to i must contain a back edge (but those do not exist in a DAG)
    - If j is not a descendant of i then a path from j to i contains a cross edge, but then $f(i) < f(j)$

Topological Sorting

- Hence we can compute a topological sorting in linear time

Application 2: Strongly connected components

- Definition:
  - A strongly connected component of a graph G is a maximal set of vertices V’ such that for each pair of vertices v,w in V’ there is a path from v to w
- Note: in undirected graphs this corresponds to connected components, but here we have one-way roads
- Strongly connected components (viewed as vertices) form a DAG inside a graph G

Strongly connected components

- Algorithm
  - Use DFS(G) to compute finish times f(v) for all v
  - Compute G^T [Transposed graph: edges (u,v) replaced by (v,u)]
  - Run DFS(G^T), but in the DFS procedure go over vertices in order of decreasing f(v) from the first DFS
  - Vertices in a tree generated by DFS(G^T) form a strongly connected component

Strongly connected components

- Time: O(n+m)
- We skip the correctness proof
- Note the usefulness of the f(v) numbers computed in the first DFS

Shortest paths in weighted graphs

- We are given a graph G (adjacency list with weights W(u,v))
- No edge means $W(u,v)=\infty$
- We look for shortest paths from start vertex s to all other vertices
- Length of a path is the sum of edge weights
- Distance $\delta(u,v)$ is the minimum path length on any path u to v

Variants

- Single-Source Shortest-Path (SSSP):
  Shortest paths from s to all v
- All-Pairs Shortest-Path (APSP):
  Shortest paths between all u,v

- We will now consider SSSP
- Solved in unweighted graphs by BFS
- Convention: we don‘t allow negative weight cycles

Note on storing paths

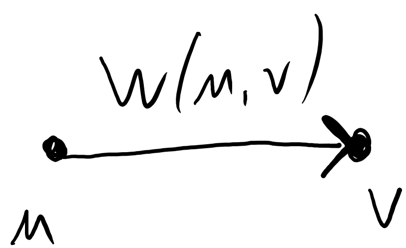
- Again we store predecessors $\pi(v)$, starting at NIL
- In the end they will form a shortest path tree

Setup

- We will use estimates d(v), starting from d(s)=0 and d(v)=∞ for all other v (Pessimism!)
- Improve estimates until tight

Relaxing an edge

- Basic Operation:
  - Relax(u,v,W)
    - if d(v)>d(u)+W(u,v)
      then d(v):=d(u)+W(u,v); π(v):=u
- I.e., if we find a better estimate we go for it



$$d(u) + w(u,v) < d(v)$$

Properties of Relaxing

- Every sequence of relax operations satisfies:
  1. d(v) $\geq \delta$(s,v) at all time
  2. Vertices with δ(s,v)=∞ always have d(v)=∞
  3. If s$\rightarrow$ u$\rightarrow$ v is a shortest path and (u,v) an edge and d(u)=δ(s,u).
     Relaxing (u,v) gives d(v)=δ(s,v)
     - d(v)$\leq$ d(u)+W(u,v) after relaxing
       =δ(s,u)+W(u,v)
       =δ(s,v) by the minimality of partial shortest paths

Observation

- Consider a shortest path p from v_1 to v_k
  - All subpaths p' of p are also shortest!
  - E.g. $p'=v_4 \rightarrow \ldots \rightarrow v_{k-34}$ is a shortest path
- Rough Idea: We should find shortest paths with fewer edges earlier
  - Starting with 1 edge (shortest edge from v)
  - Caution: cannot find shortest paths strictly in order of number of edges, just try to find all subpaths of a shortest path p before p
- Reason:
  - Never need to reconsider our decisions
  - `Greedy Algorithm'

Dijkstra‘s Algorithm

- Solves SSSP
- Condition: $W(u,v) \geq 0$ for all edges
- Idea: store vertices so that we can choose a vertex with minimal distance estimate
- Choose v with minimal d(v), relax all edges
- Until all v are processed
- v that has been processed will never be processed again

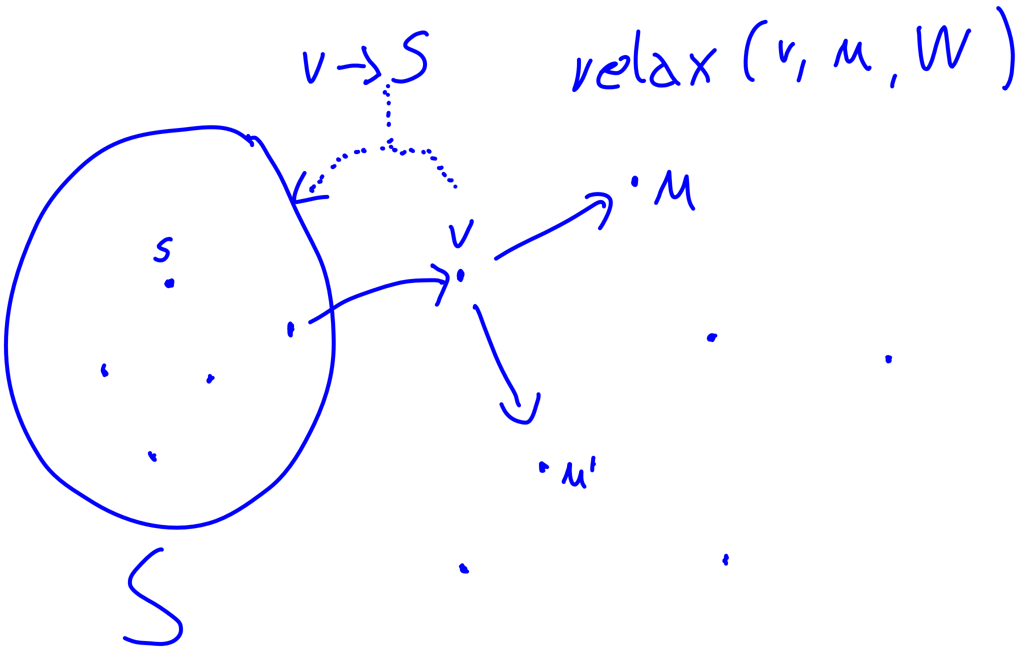
Data Structure: Priority Queue

- Store n vertices and their distance estimate d(v)
- Operations:
  - ExtractMin: Get the vertex with minimum d(v)
  - DecreaseKey(v,x): replace d(v) with a smaller value
  - Initialize
  - Insert(v)
  - Test for empty

Dijkstra‘s Algorithm

- Initialize $\pi(v)$=NIL for all v and d(s)=0, d(v)=∞ otherwise
- S=$\emptyset$ set of vertices processed
- Insert all vertices into Q (priority queue)
- While Q$\neq\emptyset$
  - v:=ExtractMin(Q)
  - S:=S $\cup$ {v}
  - For all neighbors u of v: relax(v,u,W)
    - relax uses DecreaseKey

Dijkstras Algorithmus



d (v) minimal

Things to do:

1. Prove correctness
2. Implement Priority Queues
3. Analyze the running time

3) Running Time

- n ExtractMin Operations and m DecreaseKey Operations
  - n vertices, m edges

- Their time depends on the implementation of the Priority Queue

2)

- There is a Priority Queue with the following running times:
  - Amortized time of ExtractMin is O(log n), i.e. the total time for (any) n operations is O(n log n)
  - Amortized time of DecreaseKey is O(1), i.e. total time for m operations O(m)
- With this the running time of Dijkstra is O(m + n log n)

Simple Implementation

- Store d(v) in an array
  - DecreaseKey in time O(1)
  - ExtractMin in time O(n)
- Total running time: $O(n^2)$ for ExtractMin and O(m) for DecreaseKey
- This is good enough if G is given as adjacency matrix

1) Correctness

- First some observations
  1. $d(v) \geq \delta(s,v)$ at all times (proof by induction)
  2. Hence: vertices with $\delta(s,v)=\infty$ always have $d(v)=\infty$
  3. Let $s \rightarrow v \rightarrow u$ be a shortest path with final edge (v,u) and let $d(v)=\delta(s,v)$ (at some time). Relaxing (v,u) gives $d(u)=\delta(s,u)$
  4. If at any time $d(v)=\infty$ for all $v \in V-S$, then all remaining vertices are not reachable from s

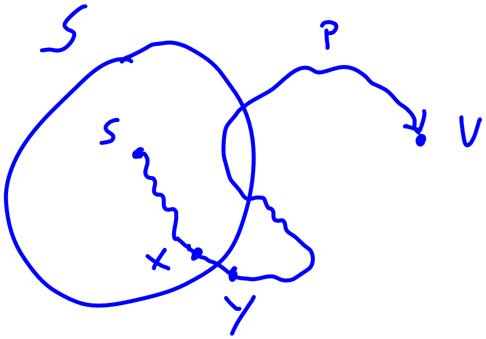
Correctness

- **Theorem:** Dijkstra‘s Algorithm terminates with $d(v)=\delta(s,v)$ for all v.
- Proof by induction over the time a vertex is taken from the priority queue and put into S
  - Invariant: For all $v \in S$ we have $d(v)=\delta(s,v)$
  - In the beginning this is true since $S=\emptyset$

Correctness

- Have to show that the next chosen vertex has $d(v) = \delta(s,v)$
- For s this is trivially true
- For $v \neq s$:
    - Only vertices with finite d(v) are chosen, or all remaining vertices are not reachable (and have correct d(v))
    - There must be a (shortest) path s to v
    - Let p be such a path, and y the first vertex outside S on p, x its predecessor

Correctness



- **Claim:** $d(y)=\delta(s,y)$, when v is chosen from Q
- Then: $d(v) \leq d(y)=\delta(s,y) \leq \delta(s,v)$. QED
- Proof of Claim:
  - $d(x)=\delta(s,x)$ by induction hypothesis
  - edge (x,y) has been relaxed, so $d(y)=\delta(s,y)$

Correctness

- Hence d(v) is correct for all vertices in S
- In the end S=V, all distances are correct

- Still need to show: the predecessor tree computed is also correct