

Lecture Notes by

Paulette Lieby

NICTA, Tower A, 7 London Circuit

6267 6232

paulette.lieby@anu.edu.au

Graphs

- vertices and edges or arcs (for undirected or directed graphs resp.): $G(V, E)$ has vertex set $V(G)$ and edge set $E(G)$
- successors and predecessors in directed graphs
- notation: an edge (u, v) in an undirected graph, $[u, v]$ in a directed graph
- connected graphs, simple graphs, loops
- concept of isolated vertex, paths, cycles, path length, degree
- cyclic and acyclic graphs
- trees

- What is a graph
- Exploring a graph: DFS and BFS
- Minimal spanning trees
- (Single source) Shortest path algorithm

Representation of Graphs

- as an adjacency matrix
- as an adjacency list

Representation of Graphs: Comparison

$$|V(G)| = n, |E(G)| = m.$$

	matrix	list
storage (in bits)	n^2	$32 * n + 64 * m$
edge existence	$O(1)$	$O(m/n)$
find all neighbours	$O(n)$	$O(m/n)$
find all successors	$O(n)$	$O(m/n)$
find all predecessors	$O(n)$	$O(m)$

- times are averages
- successor, predecessor: applies in the directed case only

Depth First Search (DFS)

- visiting each vertex and its neighbours
- going as far/deep from the initial vertex as we can:
 - going forward if possible
 - otherwise backtracking
- each vertex is visited **once** only

DFS: The Algorithm (G connected)

mark all vertices of G as NOT VISITED

DFS(u, G)

mark u as VISITED

for all neighbours v of u **do**

if v NOT VISITED

then

DFS(v, G)

end-if

end-for

DFS: Backtracking

- not being able to go “further” down the DFS tree:
 - can’t go to another neighbour from the current vertex
- once all neighbours of the current vertex u have been visited, backtrack to the parent of u and visit one of its remaining unvisited neighbours

The DFS tree

- why does DFS result in a tree?
- by convention, the DFS tree is drawn from left to right in the order of discovery of the vertices
- the terms ancestor and descendant refer to the vertices and their relationship in the DFS tree

DFS: Running Time

- **DFS** performed on each vertex exactly once
- **DFS**(u) takes $O(m_u)$
- in total, this gives $O(m)$

DFS Enhanced

- Adding discovery $d[u]$ and finishing $f[u]$ times to each vertex u
- discovery time: the 1st time a vertex is encountered
- finishing time: the last time

$t_d = 0, t_f = 0$

mark all vertices of G as NOT VISITED

DFS(u, G)

mark u as VISITED

$d[u] = t_d, t_d = t_d + 1$

for all neighbours v of u **do**

if v NOT VISITED

then

DFS(v, G)

end-if

end-for

$f[u] = t_f, t_f = t_f + 1$

Properties

- u ancestor of v in the DFS tree iff $d[u] < d[v]$ and $f[u] > f[v]$

DFS Tree Traversal

- preorder:
 - 1. visit node
 - 2. traverse left subtree
 - 3. traverse right subtree
- traversal order given by discovery time
- Q: what about postorder?

Edge Classification

- tree edges: (u, v) ($[u, v]$) is a tree edge if it is an edge in the DFS tree
 - (u, v) ($[u, v]$) is a tree edge iff $d[u] = d[v] - 1$
- back edges: (u, v) ($[u, v]$) is a back edge if v is an ancestor of u
 - if (u, v) ($[u, v]$) is a back edge then $d[u] > d[v]$

Edge Classification cont.

- forward edges: (u, v) ($[u, v]$) is a forward edge if it is not a tree edge and if u is an ancestor of v
 - $[u, v]$ is a forward edge iff $d[u] < d[v] - 1$
- cross edges: (u, v) ($[u, v]$) is a cross edge if v is neither an ancestor nor a descendant of u
 - $[u, v]$ is a cross edge iff $d[u] > d[v]$ and $[u, v]$ is not a back edge (see Proposition 2)

Edge Classification cont.

- edges are classified in the **order** of their discovery during DFS
- in particular, (u, v) (undirected graph) will be classified according to whichever (u, v) or (v, u) is encountered first

Proposition 2. *In the DFS of a directed graph G , if $[u, v]$ is a cross edge then $d[u] > d[v]$.*

Proof. Let $[u, v]$ be a cross edge in the DFS of G .

Then, by def., u is not an ancestor of v .

If $d[u] < d[v]$ then, after **DFS**(u) terminates, v must be in the subtree rooted at u . Contradiction, so $d[u] > d[v]$. □

Proposition 1. *In the DFS of an undirected graph G , every edge of G is either a tree edge or a back edge.*

Proof. Let (u, v) be an edge of G . Assume wlog that $d[u] < d[v]$.

(u, v) can be explored in two directions: from u to v in which case it is a tree edge, or from v to u , in which case it is a back edge. □

DFS: The Algorithm (G arbitrary)

DFS(G)

mark all vertices as NOT VISITED

while not all vertices are VISITED **do**

 choose a NOT VISITED vertex u and mark it as VISITED

DFS(u, G)

end-while

DFS for G Arbitrary

- running time: $O(n + m)$
- results in a forest

Applications of DFS

- finding cycles in graphs (how, in an undirected graph?, complexity?)
- finding connected components (how?)
- reachability problem (how?)
- topological sort
- finding connected components in undirected graph

Finding Cycles in Directed Graphs

- if $[u, v]$ is a tree edge, then $f[u] > f[v]$
- if $[u, v]$ is a forward edge, then $f[u] > f[v]$
- if $[u, v]$ is a cross edge, then $f[u] > f[v]$
- if $[u, v]$ is a back edge, then $f[u] < f[v]$
- a cycle in a directed graph can only be formed by a back edge (why?)
- to find cycles in a directed graph:
 - there is a cycle iff there is an edge $[u, v]$ such that $f[u] < f[v]$

Topological Sort using DFS

- only defined for acyclic directed graphs
- node comes before all nodes to which it has edges
 - a canonical application of topological sorting is in scheduling a sequence of jobs
- simple: the reverse of the postorder traversal of DFS
- why: all edges $[u, v]$ have $f[u] > f[v]$

Breath First Search (BFS)

- for each node, first visit **all** its neighbours
- each vertex is visited **once** only
- uses a FIFO queue

- BFS results in a tree (why?)

BFS: Running Time

- initialisation takes $O(n)$
- total of queue operations takes $O(n)$
- scanning the edges takes $O(m)$
- in total, this gives $O(n + m)$

BFS: The Algorithm (G connected)

BFS(u, G)

$Q = \emptyset$

mark all vertices of G as NOT VISITED

mark u as VISITED

ENQUEUE(Q, u)

while Q not empty **do**

$v =$ DEQUEUE(Q)

for all neighbours w of v **do**

if w NOT VISITED

then

 mark w as VISITED

 ENQUEUE(Q, w)

end-if

end-for

end-while

BFS Enhanced

- storing the length of the path from source u to each discovered vertex w
- how would we store the shortest path from source to each vertex?

```

BFS( $u, G$ )
   $Q = \emptyset$ 
  mark all vertices of  $G$  as NOT VISITED
  mark  $u$  as VISITED
   $d[u] = 0$ 
  ENQUEUE( $Q, u$ )
  while  $Q$  not empty do
     $v =$  DEQUEUE( $Q$ )
    for all neighbours  $w$  of  $v$  do
      if  $w$  NOT VISITED
      then
        mark  $w$  as VISITED
         $d[w] = d[v] + 1$ 
        ENQUEUE( $Q, w$ )
      end-if
    end-for
  end-while

```

29

Proof cont. If any of v_1, \dots, v_i has any non visited neighbours w they will be enqueued with $d[w] = d[v_1] + 1 = d[v_i] + 1 = d[v_{i+1}]$ and the claim holds for the same value of i .

Assume that at the time of dequeuing v_{i+1} the last vertex in Q is v_s . Note that $d[v_{i+1}] = d[v_r] = \dots = d[v_s]$. If any of v_{i+1}, \dots, v_s has any non visited neighbours w they will be enqueued with $d[w] = d[v_{i+1}] + 1$ and the claim holds with $i = s$. If none of v_{i+1}, \dots, v_s has a non visited neighbour the claim still holds for $i = s$. \square

Claim. Let v_1, v_2, \dots, v_r be the sequence of vertices in the queue Q at any time of BFS, in order of addition to Q . Then, for some $1 \leq i < r$ we have $d[v_1] = \dots = d[v_i]$, $d[v_{i+1}] = \dots = d[v_r]$, and $d[v_i] + 1 = d[v_{i+1}]$. If $i = r$ then $d[v_1] = \dots = d[v_r]$.

Proof. By induction on time of execution (or number of queue operations). At the beginning, the claim is true for $i = 1 = r$. Assume it is true for some $r > 1$ and $1 \leq i \leq r$, at some time t during execution of BFS. Note that dequeuing does not affect the truth of the claim.

- the distance from u to v is the length (ie number of edges) of the shortest path between u and v ; call it $\delta(u, v)$

Proposition 3. At completion of BFS, $d[v] = \delta(u, v)$ for all $v \in V(G)$.

Proof. By induction. True for u : $d[u] = \delta(u, u) = 0$; Assume it is true for all vertices inserted in the queue before time t and suppose that at time t , v is inserted in Q with $d[v] > \delta(u, v)$.

Minimal Spanning Trees

Proof cont. Then there exists a shortest path v_1, \dots, v_r, v of length $d[v_r] + 1 < d[v]$. By the previous claim, v_r was inserted at a time prior to t in Q , since $d[v_r] < d[v]$. However this implies that when dequeuing v_r , v would eventually have been enqueued with $d[v] = d[v_r] + 1$, contradicting our assumption. \square

- where the edges of the graph G have **weights**
- a **spanning tree** of $G(V, E)$ is a tree $T(V, E)$ where $V(T) = V(G)$ and $E(T) \subseteq E(G)$
- a **minimal** spanning tree of G is a spanning tree of G whose sum of edge weights is as small as that of any spanning tree of G

Finding a Minimal Spanning Tree: Kruskal's Algorithm

- sort the edges by increasing weight
- initially, each vertex forms a tree
- for each edge e in increasing order of weight:
 - if its two end-points are in different trees T_1 and T_2 , accept the edge and merge the two trees
 - else (if its two end-points are in the same tree) reject e

Data Structure for Kruskal's: Tree Node and Tree

- vertex:
 - to which tree a vertex belongs to
 - using the parent-child relationship: **parent**(v)
- tree:
 - identified by its root, say r
 - want to know its height: **height**(r)

- to which tree does vertex u belong to

find_tree(u)

$x = u$

while parent(x) not NIL **do**

$x = \text{parent}(x)$

return x

- why is height updated only when **height**(lower) == **height**(higher)
- why is the height update in this case correct?

- merge two trees by making the root of the lower a child of the root of the higher

merge(x, y)

if height(x) > height(y)

higher = x , lower = y

else

higher = y , lower = x

parent(lower) = higher

if height(lower) == height(higher)

height(higher) = height(higher) + 1

Proposition 4. A tree of height h formed by **merge** has at least 2^h vertices.

Proof. Induction. True for $h = 0$.

Assume true for some $h \geq 0$. The only way to get a tree T of height $h + 1$ is to merge two trees T_1 and T_2 both of height h . Then by IH, T has at least $2^h + 2^h = 2^{h+1}$ vertices. □

Kruskal's: Creating the Sorted List of Edges

- an edge e as a pair u, v of vertices
 - end nodes given by $\text{end1}(e)$ and $\text{end2}(e)$
- a list of edges as an array or a linked list
- list sorted using, say, quicksort

Kruskal's: Initialisation

- all the vertices are made into trees

init()

for all vertices u

parent(u) = NIL

height(u) = 0

E = sorted list of edges

Kruskal's: the Algorithm

- T is the minimal spanning tree

init()

$T = \emptyset$

while not all edges in E processed

e = next edge in E

x = **find_tree**($\text{end1}(e)$)

y = **find_tree**($\text{end2}(e)$)

if $x \neq y$

merge(x, y)

add e to T

Kruskal's: Running Time

- better: **while** $|E(T)| < n - 1$
- **init** (without sorting): $O(n)$
- sorting of edge list: $O(m^2)$ if insertion sort, $O(m \log m)$ average time if quicksort (worst time $O(m^2)$), $O(m \log m)$ worst time if heapsort or mergesort
- going through the edges: $O(m)$
- **find_tree**: $O(\log n)$ (by Prop. 4)
- **merge**: $O(1)$
- total: $O(n + m + m \log m + m \log n) = O(m \log m)$

Proposition 5. *The tree T in Kruskal's is a minimal spanning tree.*

Proof. We assume that no two edges have the same weight. Let K be the tree produced by Kruskal's, and let T be a minimal spanning tree such that $K \neq T$. Let e_1, e_2, \dots, e_m be the sequence of all the edges of G ordered in non-decreasing order of their weights. Let e_i be the first edge not in both trees. Two cases:
(i) e_i is in T but not in K : can only be the case if e_1, e_2, \dots, e_i form some cycle; but then this would be the case in T too.

Another Proof for Proposition 5

Proof. By induction.

IH: There exists a minimum spanning tree (MST) T that includes the forest K being built so far by Kruskal's.

Base case: true for $K = \emptyset$.

Assume it is true for $|K| = k - 1$, and let $e = (u, v)$ the k th edge added to K .

(i) $e \in T$: there is nothing to show

Proof cont. (ii) e_i is in K but not in T : suppose $e_i = (u, v)$. Let $P = u \rightsquigarrow v$ with $P \subset T$ (because T is a MST). Then $P \cup e_i$ forms a cycle in T :

(a) $w(e_i) > w(e)$ for all $e \in P$; but then we would have $P \subset K$ and $P \cup e_i$ a cycle in K .

(b) let $e_j \in P$ be the edge with largest weight in P ; swap e_j and e_i to obtain a new tree T' with $w(T') < w(T)$.

Therefore $T = K$.

(Note that the last statement is valid since edge weights are assumed distinct).

□

Proof cont. (ii) $e \notin T$: then there is a path $P = u \rightsquigarrow v$ in T which together with e forms a cycle in $T + e$ (why?). Let $e_a \in P$ be an edge which is not in K (there must exist a least one such edge, why?). Then $w(e) \leq w(e_a)$ and $w(T - e_a + e) \leq w(T)$. However T is a MST so that $w(e) = w(e_a)$ and replacing e_a by e gives a MST $T - e_a + e$ including $K + e$.

□

Kruskal's: a Greedy Algorithm

- a **greedy** algorithm: makes the best decision at any point in time
- that is, locally optimal decisions
- in general it may not result in global optimal result
- but, in case of Kruskal's, result is **also** globally optimal

Finding a Minimal Spanning Tree: Prim's Algorithm

- assumption: the weights are positive
- grow a tree starting at one vertex by adding an edge/vertex at each step
- edges are added in order of increasing weight
- uses a heap or a priority queue

Prim's

- uses concept of **graph cut**:
 - a **cut** of a graph G is a partition of its vertex set into two sets S and $V \setminus S$
- let T be the tree Prim's has built so far
- we add a "safe" edge to T if it is a edge that "crosses" the cut, i.e., an edge that connects a vertex in T with a vertex in $G \setminus T$ (why a "safe" edge?)

Binary Heap

- invariant: "top" of the heap is always minimum key (or maximum, depending on chosen order)
- based on a binary heap (recall heapsort)
- operations (when order is non-decreasing)
 - **build_heap**: $O(n)$
 - **extract_top**: $O(1)$
 - **rebuild_heap** or **increase_key** (i.e. **bubble_down**): $O(\log n)$
 - **insert_heap** (i.e. **bubble_up**): $O(\log n)$
 - **decrease_key** (i.e. **bubble_up**): $O(\log n)$

Priority Queue Q for Prim's

operations:

- **build_heap**(Q): $O(n)$
- **extract_min**(Q): $O(\log n)$
Why? : \equiv **extract_top** followed by **rebuild_heap**
- **decrease_key**(Q, u): $O(\log n)$

Prim's: Initialisation

- r is root
- $\pi[v]$ is predecessor of v
- $k[v] = w(v, \pi[v])$ ($w(u, v)$ is weight of edge (u, v))

init(r)

for all nodes v

$k[v] = \infty$

$\pi[v] = \text{NIL}$

$k[r] = 0$

Prim's

init(r)

build_heap(Q)

while $Q \neq \emptyset$

$u = \text{extract_min}(Q)$

for all neighbours v of u

if $v \in Q$ and $w(u, v) < k[v]$

$\pi[v] = u$

$k[v] = w(u, v)$

decrease_key(Q, v)

Why it Works

- invariant: $V \setminus Q$ contains the vertices of the tree being grown (with $k[v]$ recording the current minimum distance of v to $V \setminus Q$)
- the tree is grown by
 - adding the edge of minimum weight (**extract_min**(Q))
 - that will not create a cycle:
 - **extract_min**(Q) identifies an edge (u, v) such that $u \in V \setminus Q$ and $v \in Q$
- same proof by induction as for Kruskal's

Prim's: Running Time

- **init:** $O(n)$
- **build_heap:** $O(n)$
- loop: n **extract_min** plus m **decrease_key**:
 $O((n + m) \log n)$
- total: $O(m \log n)$

- given a vertex u , the problem of finding all the shortest paths $u \rightsquigarrow v$, for all vertices v (G is assumed connected)
- we have already seen this: BFS for unweighted graphs
- for weighted graphs: Dijkstra's algorithm

Dijkstra's Algorithm

- uses a heap or priority queue
- the idea of **relaxation**: a big word that means "updating the upper bound on the distance"
- finding the minimum distance from the source to other nodes in the order of minimum distances
- requires non-negative edge weights

- *equivalent to Prim's: difference lies in what is being "relaxed"*

Priority Queue Q for Dijkstra's

operations:

- **build_heap**(Q): $O(n)$
- **extract_min**(Q): $O(\log n)$
Why? : \equiv **extract_top** followed by **rebuild_heap**
- **decrease_key**(Q, u): $O(\log n)$

Dijkstra's: Initialisation

- s is source
- $\pi[v]$ is predecessor of v
- $d[v]$: current shortest distance from s to v

```
init( $s$ )
  for all nodes  $v$ 
     $d[v] = \infty$ 
     $\pi[v] = \text{NIL}$ 
   $d[s] = 0$ 
```

Dijkstra's

- $w(u, v)$ is weight of edge (u, v)

```
init( $s$ )
 $S = \emptyset$ 
build_heap( $Q$ )
while  $Q \neq \emptyset$ 
   $u = \text{extract\_min}(Q)$ 
  add  $u$  to  $S$ 
  for all neighbours  $v$  of  $u$ 
    if  $d[u] + w(u, v) < d[v]$ 
       $d[v] = d[u] + w(u, v)$ 
       $\pi[v] = u$ 
      decrease_key( $Q, v$ )
```

Dijkstra's: Relaxation

```
relax( $u, v$ )
  if  $d[u] + w(u, v) < d[v]$ 
     $d[v] = d[u] + w(u, v)$ 
     $\pi[v] = u$ 
  decrease_key( $Q, v$ )
```

Why it Works

- invariant: $V \setminus Q$ contains the vertices for which the shortest path from the source has already been computed
- $k[v]$ records the current min distance of v to the source

Theorem 1. For a given vertex s and for all vertices u in the graph, Dijkstra's correctly finds the shortest path from s to u .

- Let $\delta(s, u)$ be the (shortest) distance from s to u

Proposition 6. *Subpaths of shortest paths are shortest paths.*

Proof. Use contradiction. Let $s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v . Assume that $s \rightsquigarrow u$ is not a shortest path from s to u . □

Proof (cont.). (i) We show that (a) holds, that is, $d[u] = \delta(s, u)$. Proof by contradiction. Assume that $d[u] \neq \delta(s, u)$. Then there exists a shortest path $s \rightsquigarrow x \rightarrow y \rightsquigarrow u$ from s to u so that the vertices in $s \rightsquigarrow x$ are in S but not y . Note that by (b) there cannot be a shortest special path $s \rightsquigarrow z \rightarrow u$ with length $< d[u]$. Then $d[y] = \delta(s, y)$ by Prop. 6. Further, $d[y] = \delta(s, y) \leq \delta(s, u) < d[u]$; but $d[u] \leq d[y]$ (why?), so contradiction and $d[y] = \delta(s, y) = \delta(s, u) = d[u]$.

- A *special path* from s to u is a path $s \rightsquigarrow x \rightarrow u$ with vertices in $s \rightsquigarrow x$ in S , $u \notin S$, and whose last edge is (x, u)

Proof (of Theorem 1). By induction. IH:

(a) S contains the vertices u with $d[u] = \delta(s, u)$; the shortest path to u consists only of vertices in S .

(b) If $u \notin S$, then $d[u]$ is the length of the shortest special path s to u ; it is ∞ if no such path exists.

Base case: True at initialisation ($S = \{s\}$).

Assume (a) and (b) true at step k and let u be the vertex chosen at step $k + 1$, that is, the vertex with smallest $d[u]$.

Proof cont. (ii) We show that (b) holds. Let v a vertex not in S when u is inserted in S . Is there is no special path from s to v then there is nothing to prove. Likewise when no special path to v contains u . Assume there is a special path to v that contains u . When u is inserted in S the relaxation procedure will ensure that $d[v]$ is the length of the shortest special path to v . □

- Where in the proof of Theorem 1 do we use the fact that the edge weights are non-negative?

Dijkstra's: Running Time

- **init:** $O(n)$
- **build_heap:** $O(n)$
- **loop:** n **extract_min** plus m **decrease_key:**
 $O((n + m) \log n)$
- **total:** $O(m \log n)$