Contents

Section outline

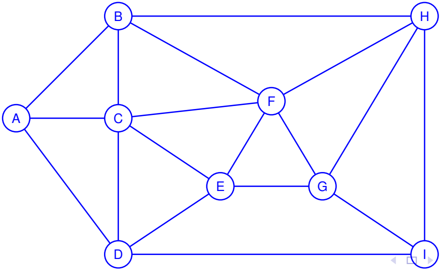
Simple graphs

Definition (Graph)

A (simple, undirected) graph, $G = (V, E)$, consists of a non-empty set V of vertices (or nodes), and a set $E \subseteq V \times V$ of (undirected) edges (or arcs).

Every edge $\langle u, v \rangle \in E$ has two distinct vertices u and v ($u \neq v$) as endpoints and are said to be adjacent in G;

For an undirected graph, $\langle u, v \rangle \in E \Rightarrow \langle v, u \rangle \in E$.



Directed graphs

Definition (Digraph)

A directed graph (digraph), $G = (V, E)$, consists of a non-empty set V of vertices (or nodes), and a set $E \subseteq V \times V$ of directed edges (or arcs).

Every edge $\langle u, v \rangle \in E$ has a start (tail) vertex u and an end (head) vertex v.

- A graph $G = (V, E)$ is a set V together with a binary symmetric relation E on V.
- A directed graph $G = (V, E)$ is a set V together with a binary relation E on V.
- A multigraph is permitted to have multiple edges between pairs of vertices.

Degree and neighbourhood of a vertex

Definition (Degree)

The degree of a vertex v in a undirected graph is the number of edges incident with it. The degree of the vertex v is denoted by $\deg(v)$.

Definition (Nbd)

The neighborhood (neighbor set) of a vertex v in a undirected graph, denoted $N(v)$ is the set of vertices adjacent to v.

Definition (In-degree/Out-degree)

The in-degree of a vertex v, denoted $\deg^{-}(v)$, is the number of edges directed into v. The out-degree of v, denoted $\deg^{+}(v)$, is the number of edges directed out of v.

Note that a self loop at a vertex contributes 1 to both in-degree and out-degree.

Subgraph of a graph

Definition (Subgraph)

A subgraph of a graph $G(V, E)$ is a graph $H(W, F)$, where $W \subseteq V$ and $F \subseteq E$. A subgraph H of G is a proper subgraph of G if $H \neq G$.

Definition (Induced subgraph)

Let $G(V, E)$ be a graph. The subgraph induced by a subset W of the vertex set V is the graph $H(W, F)$, whose edge set F contains an edge in E if and only if both endpoints are in W.

Graph components

Definition (Component of an undirected graph)

A connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

- Components partition a simple undirected graph and induce equivalence classes of nodes that are reachable from each other
- Components in a simple undirected graph are easily identified through depth first traversal
- This definition is not of much use in a digraph; there the notion of strongly connected components is more relevant

Definition (Strongly connected component of a digraph)

A subgraph of a digraph is said to be strongly connected if every vertex is reachable from every other vertex.

Simple properties of graphs

- (Handshaking Lemma): If $G = (V, E)$ is a undirected graph with m edges, then: $\sum_{v \in V} \deg(v) = 2m$
- An undirected graph has an even number of vertices of odd degree.
- For a directed graph $G = (V, E)$, $|E| = \sum_{v \in V} \deg^{-}(v) = \sum_{v \in V} \deg^{+}(v)$
- In every graph there are two vertices of the same degree

Graph representation

Adjacency matrix

- $A[i,j] = A[j,i] = 1$ if and only if there is an edge between v_i and v_j
- A is symmetric for (simple undirected) graphs
- Not symmetric for digraphs
- $|V| \times \frac{|V|}{2}$ enough to store adjacency information
- Entries of A may also indicate weights, absence of edge may be indicated by ∞
- Uneconomic for sparse graphs

Adjacency list

- A vector L of V linked lists indicate the adjacencies of each vertex
- If v_x is adjacent to $\{v_1, v_2, \ldots, v_k\}$, the linked list for $L[v_x]$ points to the linked list with entries $\{v_1, v_2, \ldots, v_k\}$
- Uneconomic for dense graphs

Section outline

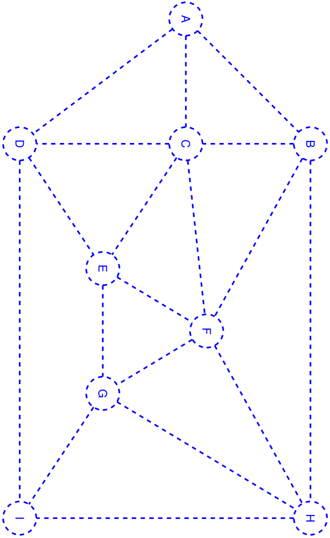
2. **Depth first traversal of a graph**
   - Simple DFTr using a stack
   - DFTr with stack for digraphs and components
   - Cycle detection in a graph
   - Recursive DFTr with pre/postorder labelling
   - Digraph edge classification
   - Properties of previst and postvisit numbers
   - Topological sorting
   - Cycle detection and topological sorting
   - Forming circuit equations using fundamental cycles

Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $\quad v \leftarrow$top(S)
7. $\quad$ if v has an unvisited neighbour u
8. $\quad\quad$ mark u as `in-stack`
9. $\quad\quad$ push u into S
10. $\quad\quad$ add edge $\langle v, u\rangle$ to T
11. $\quad\quad$ pre-number u as d; incr d
12. $\quad$ else
13. $\quad\quad$ mark v as `visited`; pop S
14. $\quad\quad$ post-number v as d; incr d
15. done

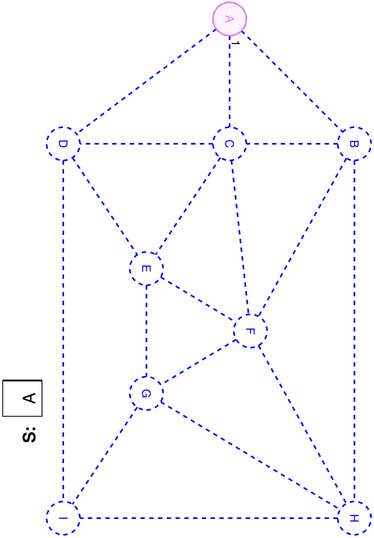
S:



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

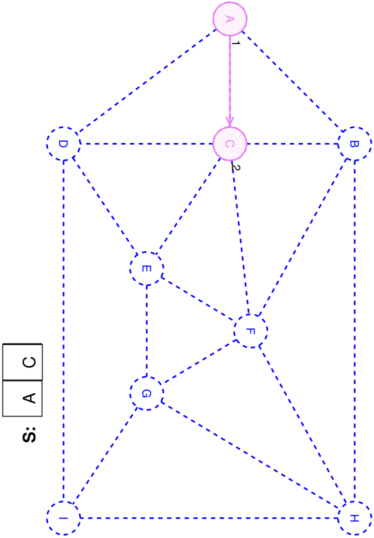
all nodes of G are initially marked `unvisited`
initialise $d \leftarrow 1$; initialise tree T to empty
mark v as `in-stack`; push v into S
pre-number v as d; incr d
while (S is not empty) do
    $v \leftarrow$top(S)
    if v has an unvisited neighbour u
        mark u as `in-stack`
        push u into S
        add edge $\langle v, u \rangle$ to T
        pre-number u as d; incr d
    else
        mark v as `visited`; pop S
        post-number v as d; incr d
done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

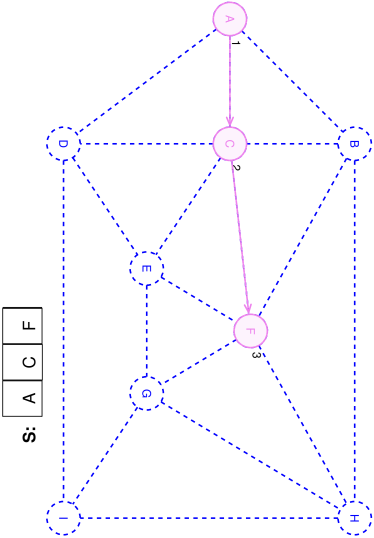
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

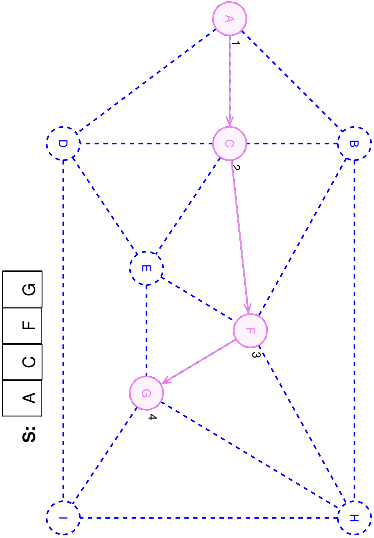
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

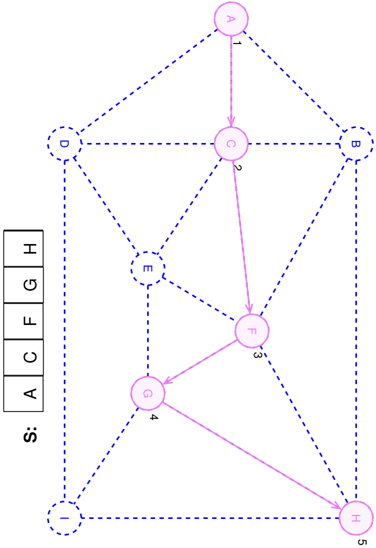
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u\rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

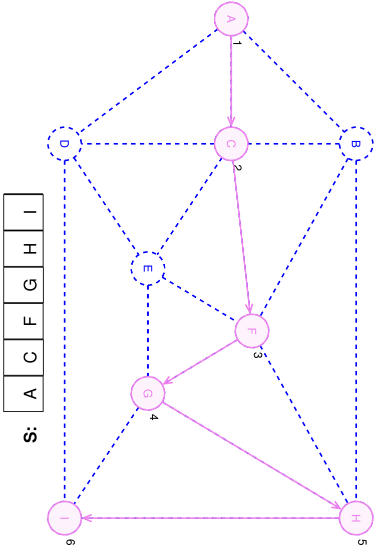
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

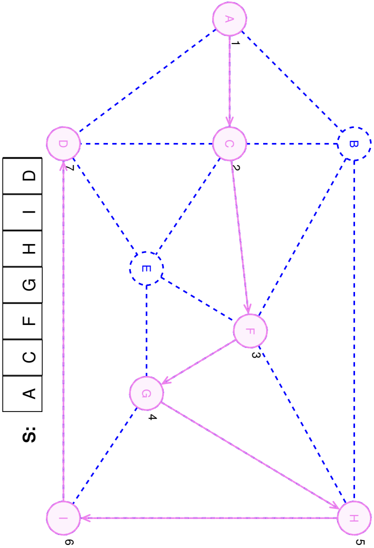
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u\rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

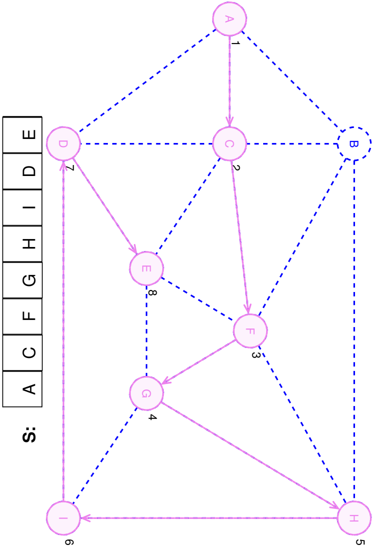
all nodes of G are initially marked `unvisited`
initialise $d \leftarrow 1$; initialise tree T to empty
mark v as `in-stack`; push v into S
pre-number v as d; incr d
while (S is not empty) do
    $v \leftarrow$top(S)
    if v has an unvisited neighbour u
        mark u as `in-stack`
        push u into S
        add edge $\langle v, u \rangle$ to T
        pre-number u as d; incr d
    else
        mark v as `visited`; pop S
        post-number v as d; incr d
done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

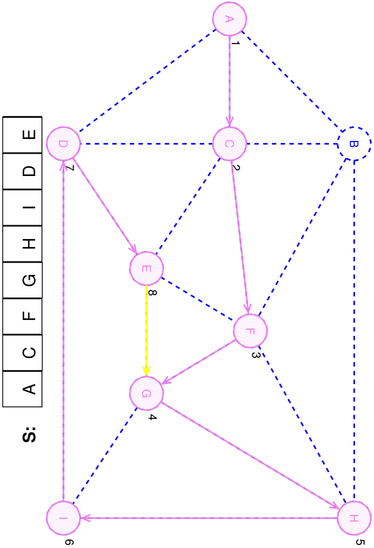
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

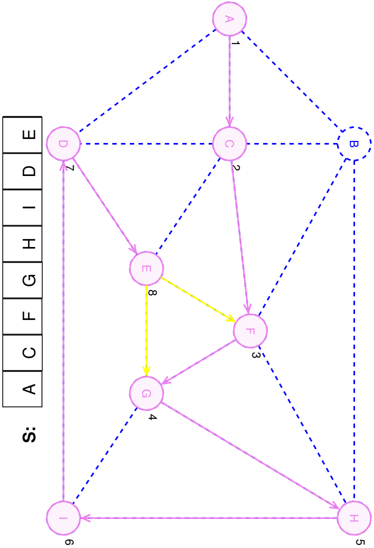
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

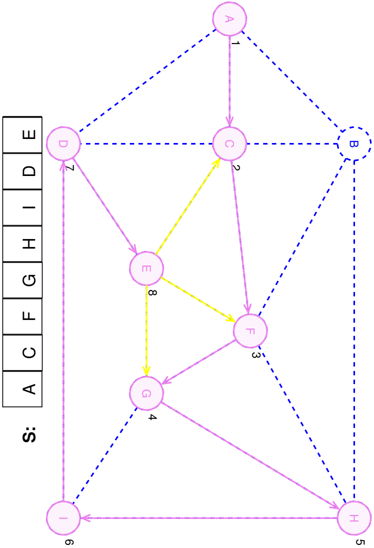
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

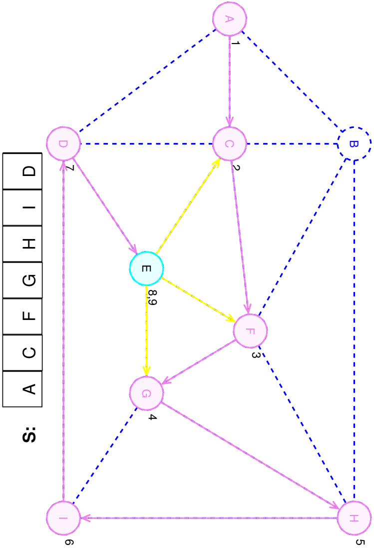
all nodes of G are initially marked `unvisited`
initialise $d \leftarrow 1$; initialise tree T to empty
mark v as `in-stack`; push v into S
pre-number v as d; incr d
while (S is not empty) do
    $v \leftarrow$top(S)
    if v has an unvisited neighbour u
        mark u as `in-stack`
        push u into S
        add edge $\langle v, u \rangle$ to T
        pre-number u as d; incr d
    else
        mark v as `visited`; pop S
        post-number v as d; incr d
done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

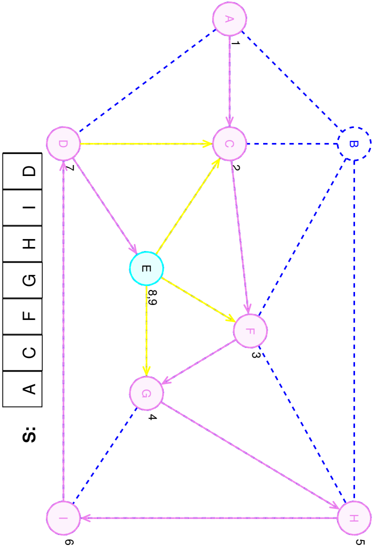
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $\quad v \leftarrow$top(S)
7. $\quad$ if v has an unvisited neighbour u
8. $\quad\quad$ mark u as `in-stack`
9. $\quad\quad$ push u into S
10. $\quad\quad$ add edge $\langle v, u\rangle$ to T
11. $\quad\quad$ pre-number u as d; incr d
12. $\quad$ else
13. $\quad\quad$ mark v as `visited`; pop S
14. $\quad\quad$ post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

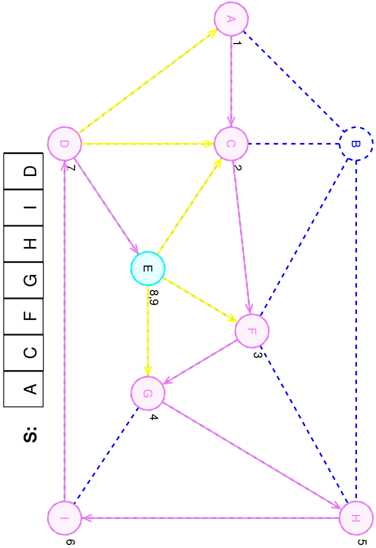
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $\quad v \leftarrow \text{top}(S)$
7. $\quad$ if v has an unvisited neighbour u
8. $\quad\quad$ mark u as `in-stack`
9. $\quad\quad$ push u into S
10. $\quad\quad$ add edge $\langle v, u \rangle$ to T
11. $\quad\quad$ pre-number u as d; incr d
12. $\quad$ else
13. $\quad\quad$ mark v as `visited`; pop S
14. $\quad\quad$ post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

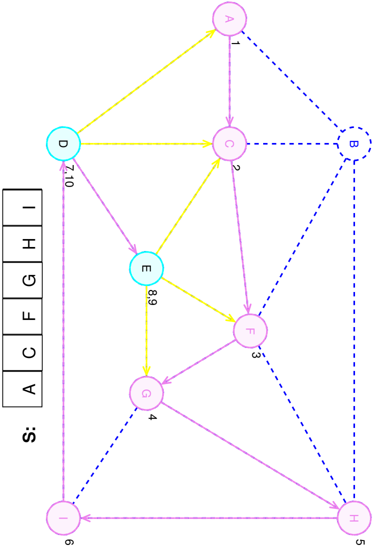
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

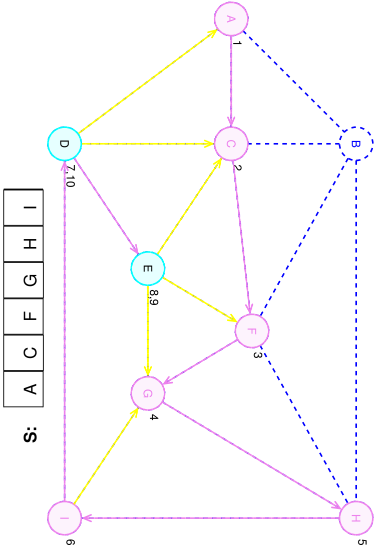
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $\quad v \leftarrow$top(S)
7. $\quad$ if v has an unvisited neighbour u
8. $\quad\quad$ mark u as `in-stack`
9. $\quad\quad$ push u into S
10. $\quad\quad$ add edge $\langle v, u \rangle$ to T
11. $\quad\quad$ pre-number u as d; incr d
12. $\quad$ else
13. $\quad\quad$ mark v as `visited`; pop S
14. $\quad\quad$ post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

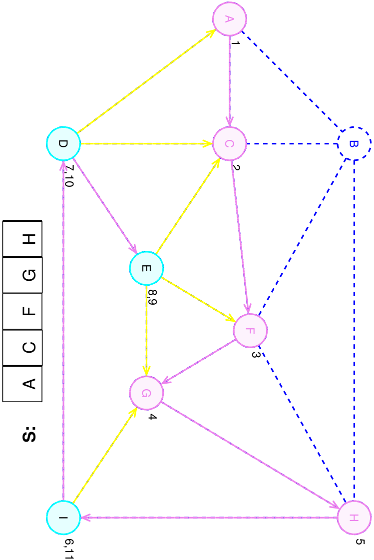
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

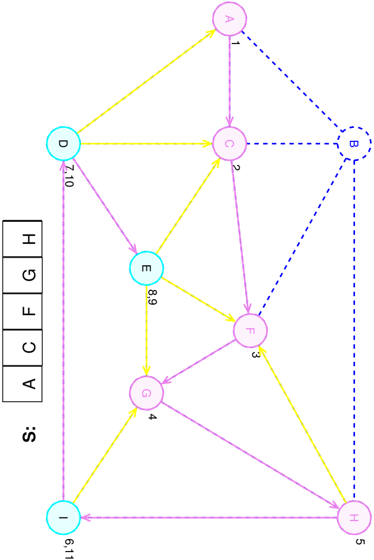
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $\quad v \leftarrow$top(S)
7. $\quad$ if v has an unvisited neighbour u
8. $\quad\quad$ mark u as `in-stack`
9. $\quad\quad$ push u into S
10. $\quad\quad$ add edge $\langle v, u \rangle$ to T
11. $\quad\quad$ pre-number u as d; incr d
12. $\quad$ else
13. $\quad\quad$ mark v as `visited`; pop S
14. $\quad\quad$ post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

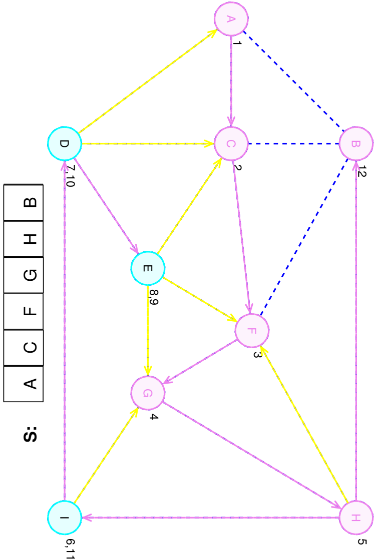
all nodes of G are initially marked `unvisited`
initialise $d \leftarrow 1$; initialise tree T to empty
mark v as `in-stack`; push v into S
pre-number v as d; incr d
while (S is not empty) do
    $v \leftarrow$top(S)
    if v has an unvisited neighbour u
        mark u as `in-stack`
        push u into S
        add edge $\langle v, u \rangle$ to T
        pre-number u as d; incr d
    else
        mark v as `visited`; pop S
        post-number v as d; incr d
done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

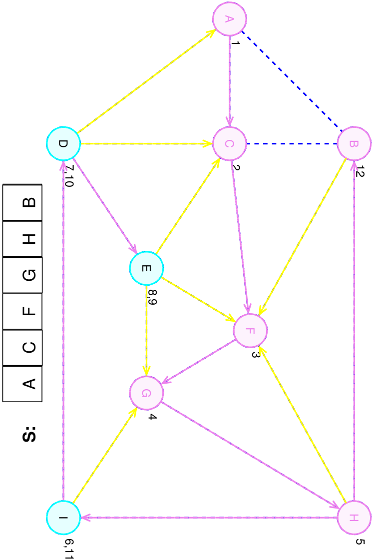
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u\rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

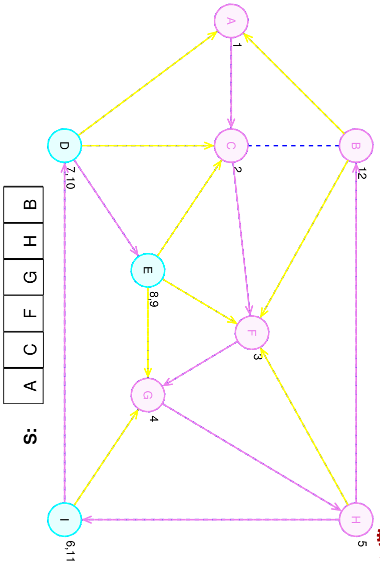
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

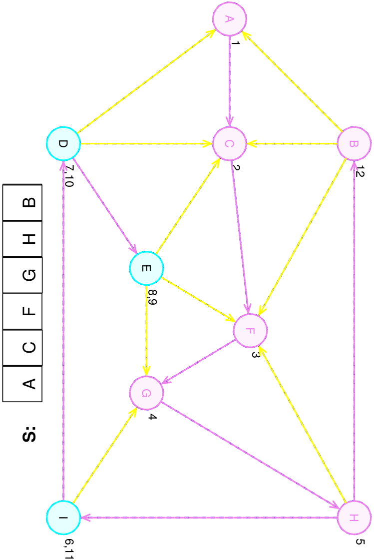
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

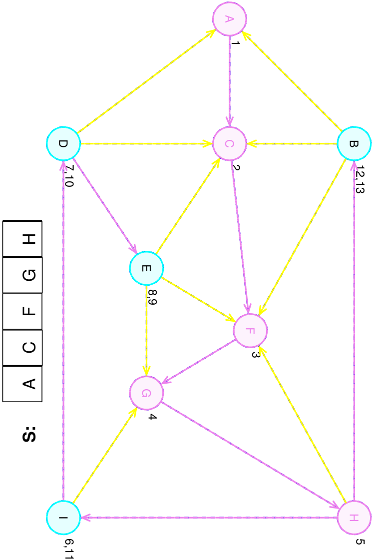
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

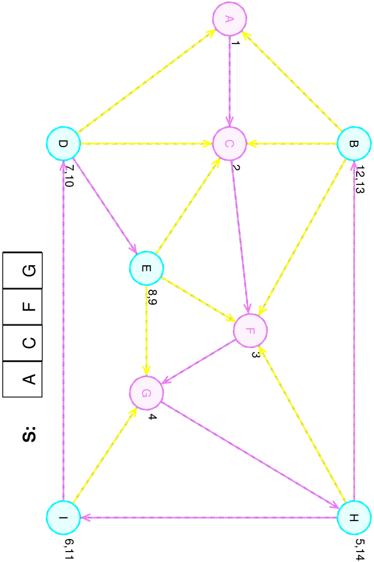
all nodes of G are initially marked `unvisited`
initialise $d \leftarrow 1$; initialise tree T to empty
mark v as `in-stack`; push v into S
pre-number v as d; incr d
while (S is not empty) do
    $v \leftarrow$top(S)
    if v has an unvisited neighbour u
        mark u as `in-stack`
        push u into S
        add edge $\langle v, u \rangle$ to T
        pre-number u as d; incr d
    else
        mark v as `visited`; pop S
        post-number v as d; incr d
done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

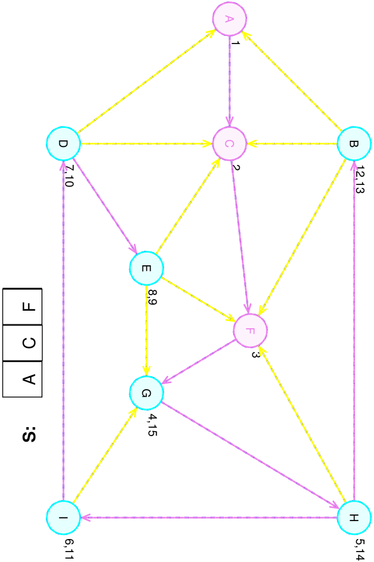
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u\rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

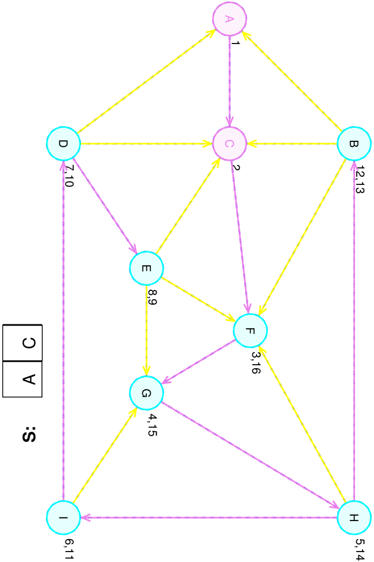
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u \rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

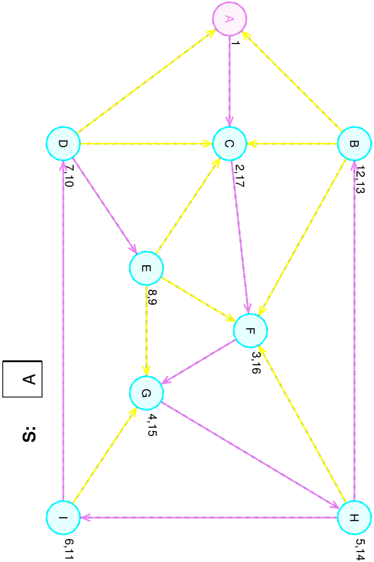
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u \rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

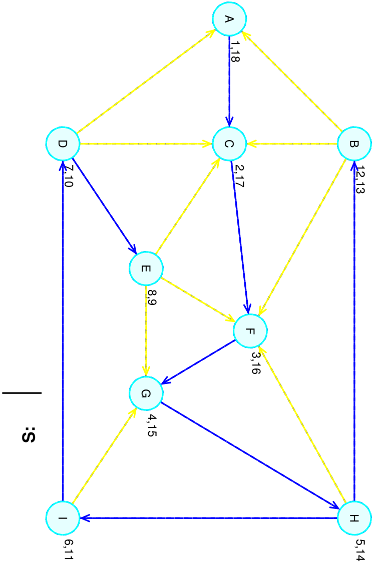
1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7. if v has an unvisited neighbour u
8. mark u as `in-stack`
9. push u into S
10. add edge $\langle v, u\rangle$ to T
11. pre-number u as d; incr d
12. else
13. mark v as `visited`; pop S
14. post-number v as d; incr d
15. done



Simple DFTr using a stack

Starting at v of $G(V, E)$; using stack (or list) S

1. all nodes of G are initially marked `unvisited`
2. initialise $d \leftarrow 1$; initialise tree T to empty
3. mark v as `in-stack`; push v into S
4. pre-number v as d; incr d
5. while (S is not empty) do
6. $v \leftarrow$top(S)
7.   if v has an unvisited neighbour u
8.   mark u as `in-stack`
9.   push u into S
10.   add edge $\langle v, u\rangle$ to T
11.   pre-number u as d; incr d
12.   else
13.   mark v as `visited`; pop S
14.   post-number v as d; incr d
15. done



DFTr with stack for digraphs and components

Starting at v of $G(V, E)$ (any node for a (simple undirected) graph); stack S (or a list S) is used

all nodes of G are initially marked `unvisited`
initialise $c \leftarrow 0$, $d \leftarrow 1$; initialise tree T to empty
repeat
    mark v as `in-stack`; push v into S
    pre-number v as d; incr d
    while (S is not empty) do
        $v \leftarrow$top(S)
        if v has an unvisited neighbour u
            mark u as `in-stack`; push u into S
            add edge $\langle v, u \rangle$ to T
            pre-number u as d; incr d
        else
            mark v as `visited`; pop S; post-number v as d; incr d
    done
    incr(c); choose v as any `unvisited` vertex, if any
until (no vertex is `unvisited`)

DFTr with stack for digraphs and components

Starting at v of $G(V, E)$ (any node for a (simple undirected) graph); stack S (or a list S) is used

```
all nodes of G are initially marked unvisited
initialise c ← 0, d ← 1; initialise tree T to empty
repeat
    mark v as in-stack; push v into S
    pre-number v as d; incr d
    while (S is not empty) do
        v ←top(S)
        if v has an unvisited neighbour u
            mark u as in-stack; push u into S
            add edge ⟨v, u⟩ to T
            pre-number u as d; incr d
        else
            mark v as visited; pop S; post-number v as d; incr d
    done
    incr(c); choose v as any unvisited vertex, if any
until (no vertex is unvisited)
```

- T is the depth first spanning tree of G, if G is simple and undirected
- If G is a digraph, T may have multiple roots
- For simple undirected graphs, the numbering of nodes so obtained is the depth first pre/post numbering
- The repeat-until loop is needed for digraphs and (simple undirected) graphs with multiple components whose count is c, on termination

Cycle detection in a graph

- The depth first traversal of a graph can be used to identify presence of possible cycles in graph
- Augment line 13/13 of the algorithm, with:
  - if v has an `in-stack` neighbour u, mark edge $\langle v, u\rangle$ as a *back edge*
- Every fundamental cycle in a graph is associated with a back edge
- Any spanning tree of a (connected undirected) graph of $|V|$ vertices will have $|V| - 1$ edges
- A (connected undirected) graph with $|V|$ vertices and $|E|$ edges will have $|E| - |V| + 1$ fundamental cycles
- A set of fundamental cycles in a (connected undirected) graph is detected by way of depth first traversal in $O(|E|)$ time
- Fundamental cycles may be composed to obtain other cycles which are not directly identified by the traversal

Recursive DFTr with pre/postorder labelling

Recursive DF traversal

DFTrav(v)

mark v
preVisit(v)
foreach vertex w adjacent to v
    if w is unmarked
        introduce edge $\langle v, w \rangle$ in T
        DFTrav(w)
postVisit(w)

DF traversal for a directed graph

DFTravAll(v)

initCount
forall vertices v
    unmark v
forall vertices v
    if v is unmarked
        DFTrav(v)

Labelling functions

preVisit(v)

v.pre $\leftarrow$ *count*
count $\leftarrow$ *count* $+ 1$

postVisit(v)

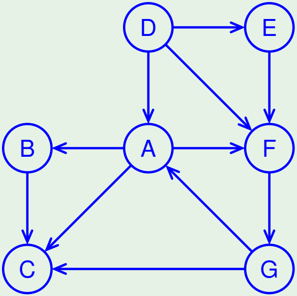
v.post $\leftarrow$ *count*
count $\leftarrow$ *count* $+ 1$

initCount

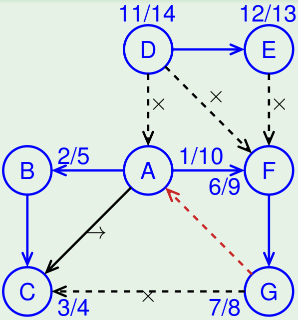
count $\leftarrow 0$

Digraph edge classification

Example (DF traversal of a digraph with pre and post numbering)

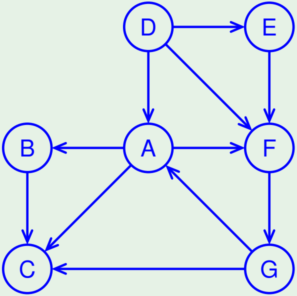


- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique

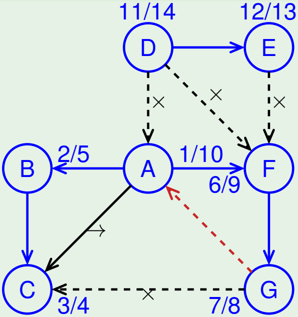


Digraph edge classification

Example (DF traversal of a digraph with pre and post numbering)



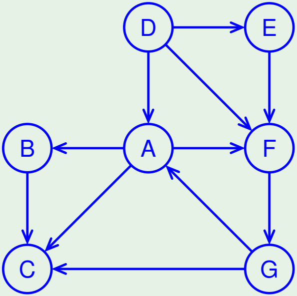
- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique



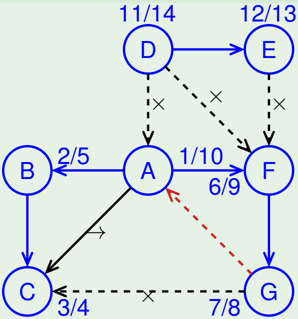
Tree edges those edges present in the DF traversal forest

Digraph edge classification

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique

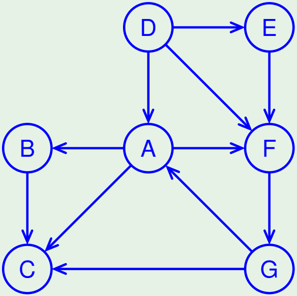


Tree edges those edges present in the DF traversal forest

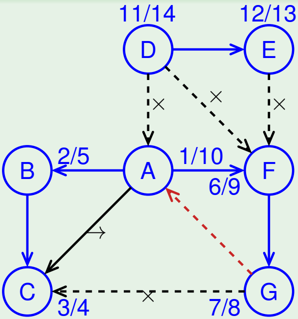
Forward edges $\langle u, v\rangle$ where v is a proper descendent of u in the tree

Digraph edge classification

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique



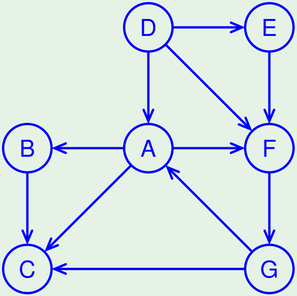
Tree edges those edges present in the DF traversal forest

Forward edges $\langle u, v \rangle$ where v is a proper descendent of u in the tree

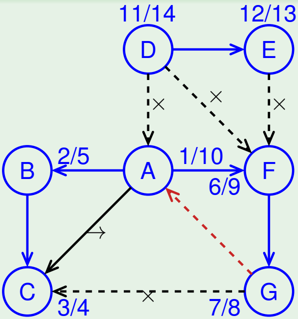
Back edges $\langle u, v \rangle$ where v is an ancestor of u in the tree

Digraph edge classification

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique



Tree edges those edges present in the DF traversal forest

Forward edges $\langle u, v\rangle$ where v is a proper descendent of u in the tree

Back edges $\langle u, v\rangle$ where v is an ancestor of u in the tree

Cross edges $\langle u, v\rangle$ where u, v are neither ancestors nor descendents of one another

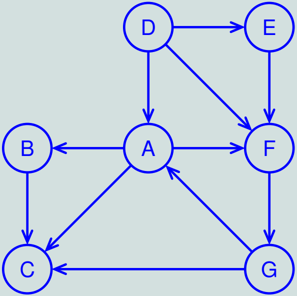
Properties of pre/postvisit numbers

Theorem (Parenthesis structure of previsit and postvisit numbers)

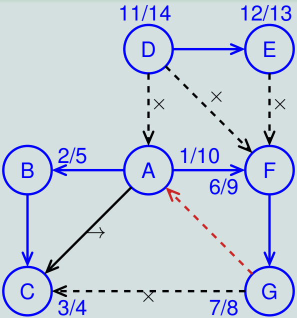
Let $G(V, E)$ be a digraph, F any of its DF traversal forest and $u, v \in V$ with previsit and postvisit numbers as $u.r, u.s, v.r, v.s$ respectively:

- *u is a descendent of v in F if and only if $[u.r, u.s]$ is a subinterval of $[v.r, v.s]$, so that $v.r < u.r < u.s < v.s$*

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique

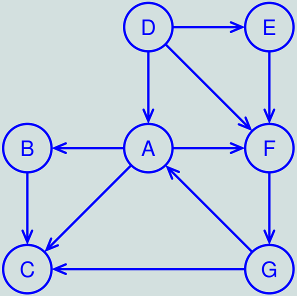


Properties of pre/postvisit numbers (contd.)

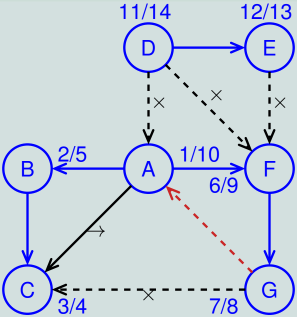
Theorem (Parenthesis structure of previsit and postvisit numbers)

- *u is unrelated to v in F if and only if $[u.r, u.s]$ and $[v.r, v.s]$ are disjoint intervals, so that $u.s < v.r$ or $v.s < u.r$*
- *$u.r < v.r < u.s < v.s$ and $v.r < u.r < v.s < u.s$ are not possible*

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique

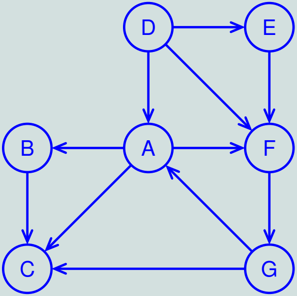


Properties of pre/postvisit numbers (contd.)

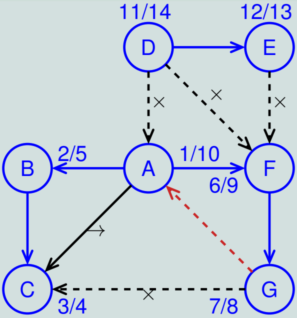
Theorem (Parenthesis structure of previsit and postvisit numbers)

- *Tree edges, forward edges, and cross edges all go from a vertex of higher postvisit number to a vertex of lower postvisit number*
- ***Back edges go from a vertex of lower postvisit number to a vertex of higher postvisit number***

Example (DF traversal of a digraph with pre and post numbering)



- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique

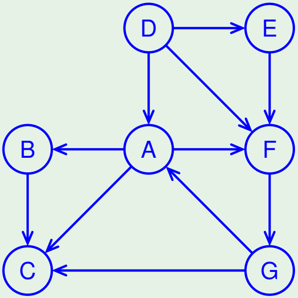


Topological sorting

Definition (Topological sorting)

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge $\langle u, v\rangle$ from vertex u to vertex v, u comes before v in the ordering.

Example (A graph and its topological sort)



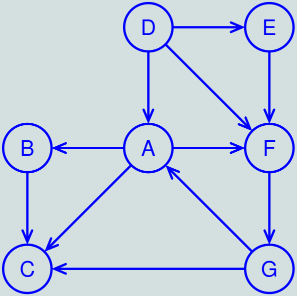
- D, E, A, B, F, G, C
- D, A, B, E, F, G, C

Cycle detection and topological sorting

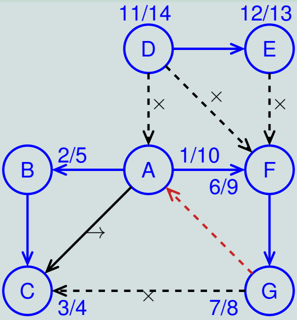
Theorem (Parenthesis structure of previsit and postvisit numbers)

- ***A digraph is acyclic if and only if it is free of back edges***
- ***Vertices ordered by their postvisit numbers form a reverse topological sort***

Example (DF traversal of a digraph with pre and post numbering)



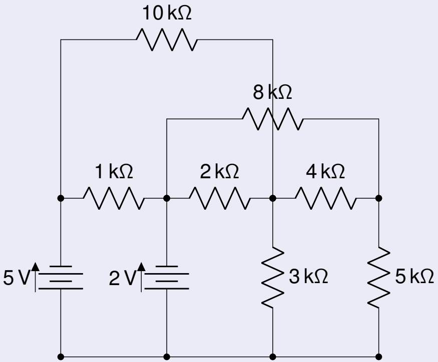
- Start DF traversal at vertex A
- Continue traversal from vertex D
- Previst and postvisit labels shown
- Traversal is not unique



Forming circuit equations using fundamental cycles

Equations from circuit

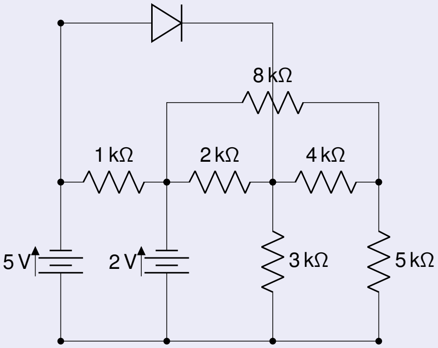
1. Develop a suitable data structure to represent such a circuit having only resistances and voltage sources.
2. How can you identify the fundamental cycles to form the circuit equations using branch currents?
3. Illustrate the working of your scheme on this circuit.



Forming circuit equations using fundamental cycles (contd.)

Equations from circuit

1. Develop a suitable data structure to represent such a circuit having only resistances and voltage sources.
2. How can you identify the fundamental cycles to form the circuit equations using branch currents?
3. Illustrate the working of your scheme on this circuit.



Section outline

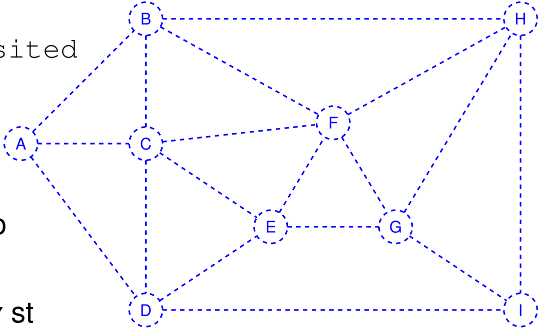
3 **Breadth first traversal**

- Breadth first traversal of a graph
- Breadth first traversal with numbering

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7.   foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9.   mark v as `visited`
10.   enqueue v into S
11.   add edge $\langle u, v \rangle$ to T
12. done

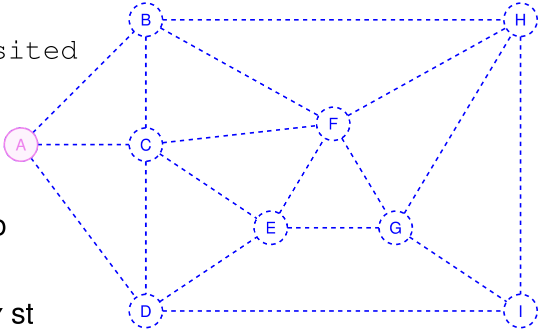


S:

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7.   foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9.   mark v as `visited`
10.   enqueue v into S
11.   add edge $\langle u, v \rangle$ to T
12. done

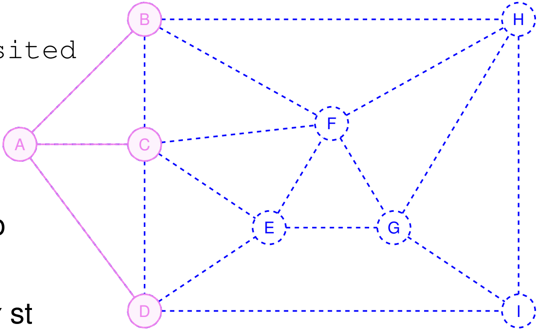


S: $\boxed{\text{A}}$

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $\quad u \leftarrow$dequeue(S)
7. $\quad$ foreach `unvisited` v st
8. $\qquad \langle u, v \rangle \in E$
9. $\qquad$ mark v as `visited`
10. $\qquad$ enqueue v into S
11. $\qquad$ add edge $\langle u, v \rangle$ to T
12. done



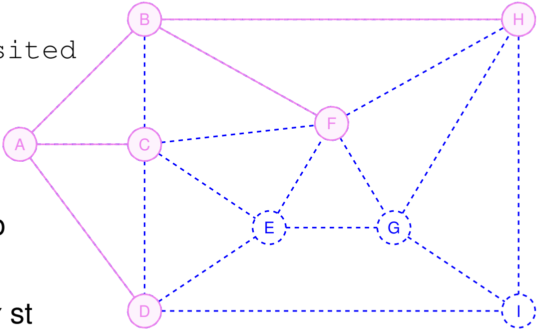
S:

B	C	D

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done



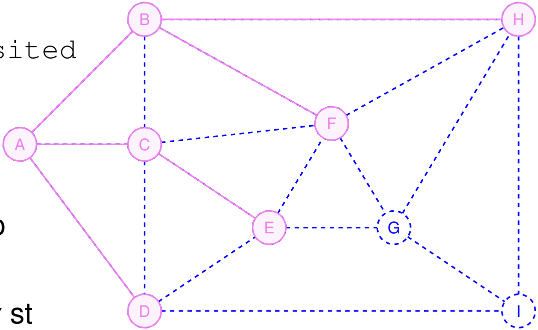
S:

C	D	F	H

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7.   foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9.   mark v as `visited`
10.   enqueue v into S
11.   add edge $\langle u, v \rangle$ to T
12. done



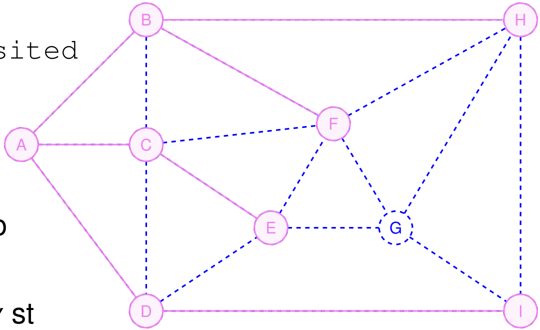
S:

D	F	H	E

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done



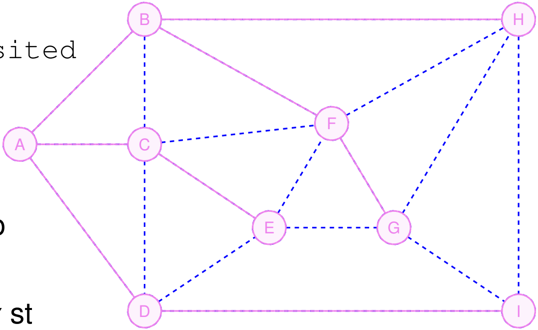
S:

F	H	E	I

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7.   foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9.   mark v as `visited`
10.   enqueue v into S
11.   add edge $\langle u, v \rangle$ to T
12. done

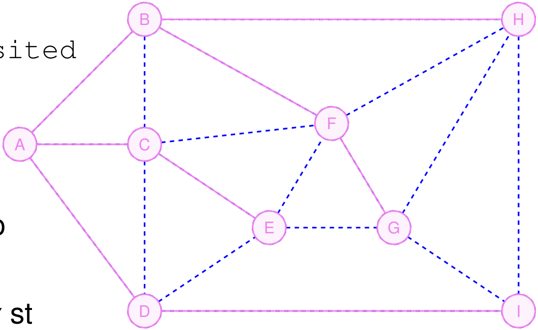


S: | H | E | I | G |

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done



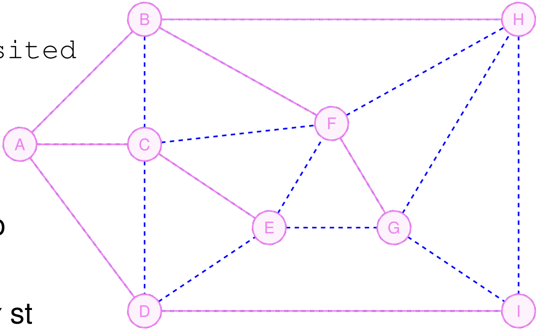
S:

E	I	G

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done

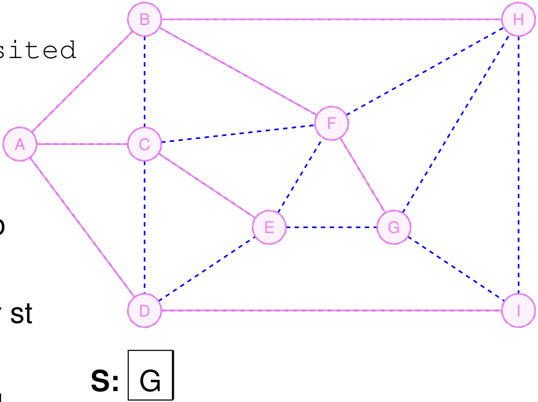


S: | I | G |

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

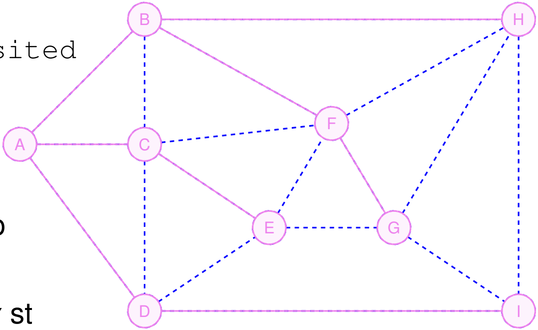
1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done



Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done

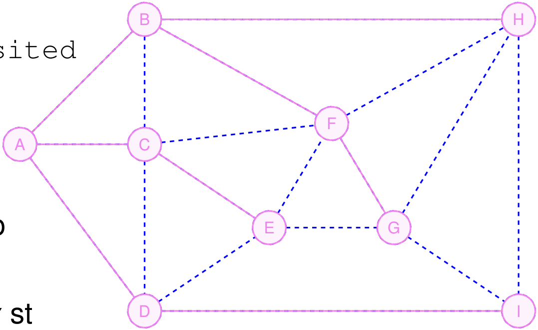


S:

Breadth first traversal of a graph

Start at any $v \in G(V, E)$

1. mark all $v \in V$ as `unvisited`
2. initialise tree T to empty
3. mark v as `visited`
4. enqueue v into S
5. while (S is not empty) do
6. $u \leftarrow$dequeue(S)
7. foreach `unvisited` v st
8. $\langle u, v \rangle \in E$
9. mark v as `visited`
10. enqueue v into S
11. add edge $\langle u, v \rangle$ to T
12. done



S:

Depth of a node in the BFS tree is independent of the order in which children of nodes are enqueued

Breadth first traversal with numbering

Start at any $v \in V$; queue S (or a list S) is used

1. mark all nodes of G as `unvisited`
2. initialise $\boxed{j \leftarrow 1,\ b_j \leftarrow 1}$; initialise tree T to empty
3. repeat
4.   mark v as `visited`; enqueue v into S // at the end
5.   number v as b_j; incr b_j
6.   while (S is not empty) do
7. $u \leftarrow$dequeue(S) // remove u from front of S
8.   foreach unvisited neighbour v of u
9.   mark v as `visited`; enqueue v into S
              // insert v at end of S
10.   add edge $\langle u, v \rangle$ to T
11.   number v as b_j; incr b_j
12.   done
13. $\boxed{\text{incr}(j);\ b_j \leftarrow 1}$; choose `unvisited` vertex v, if any
14. until (no vertex is `unvisited`)

Breadth first traversal with numbering

Start at any $v \in V$; queue S (or a list S) is used

1. mark all nodes of G as `unvisited`
2. initialise $\boxed{j \leftarrow 1,\ b_j \leftarrow 1}$; initialise tree T to empty
3. repeat
4. mark v as `visited`; enqueue v into S // at the end
5. number v as b_j; incr b_j
6. while (S is not empty) do
7. $u \leftarrow$dequeue(S) // remove u from front of S
8. foreach unvisited neighbour v of u
9. mark v as `visited`; enqueue v into S
   // insert v at end of S
10. add edge $\langle u, v \rangle$ to T
11. number v as b_j; incr b_j
12. done
13. $\boxed{\text{incr}(j);\ b_j \leftarrow 1}$; choose `unvisited` vertex v, if any
14. until (no vertex is `unvisited`)

- T is the breadth first spanning tree of G, if G is simple and undirected
- If G is a digraph, T may have multiple roots
- For simple undirected graphs, the numbering of nodes so obtained is the breadth first numbering
- The repeat-until loop is needed for graphs with multiple components
- works trivially for single component undirected graphs

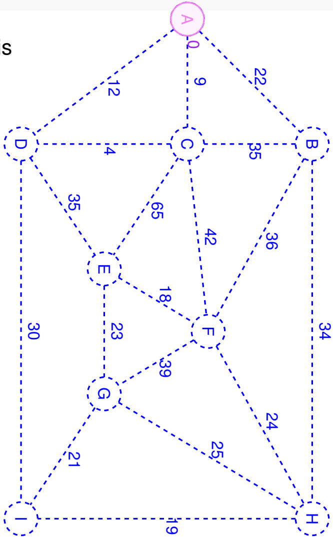
Section outline

4. **Shortest paths from a given source**
   - Shortest paths in graphs
   - Dijkstra's algorithm
   - Correctness proof of Dijkstra's algorithm
   - Complexity of Dijkstra's algorithm
   - Shortest distance between each pair of vertices

Shortest paths in graphs

- A graph may have positive and negative weights associated with its edges
- The length of a walk is the sum of the weights of the edges as they appear in the walk
- Presence of a circuit of negative weight creates a problem for identifying shortest paths
- If all edges of $G(V, E)$ have the same positive (unit) weight, the breadth first traversal, starting from some vertex $u \in V$ yields the shortest paths from u to all other vertices
- Different algorithm can be devised to find shortest paths with edges having arbitrary weights, but free of circuits of negative weight

Dijkstra's algorithm



Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

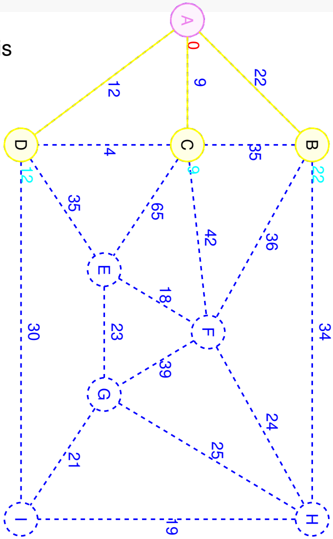
```
all nodes of G are initially marked unvisited
set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
  while (Q_H is not empty) do
    ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
    if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
    foreach neighbour x of v which is not visited
      if (x is unvisited)
        mark x as in-queue
        enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
      else // ⟨_, x, q⟩ is in-queue
        if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
    endfor
  done
```

Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

```
all nodes of G are initially marked unvisited
set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
   while (Q_H is not empty) do
      ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
      if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
      foreach neighbour x of v which is not visited
         if (x is unvisited)
            mark x as in-queue
            enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
         else // ⟨_, x, q⟩ is in-queue
            if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
      endfor
   done
```



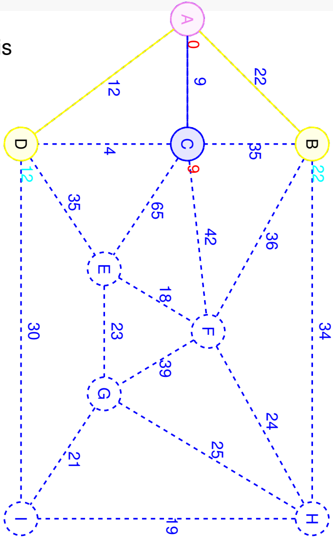
Invariants:

1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

```
all nodes of G are initially marked unvisited
set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
  while (Q_H is not empty) do
    ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
    if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
    foreach neighbour x of v which is not visited
      if (x is unvisited)
        mark x as in-queue
        enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
      else // ⟨_, x, q⟩ is in-queue
        if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
    endfor
  done
```

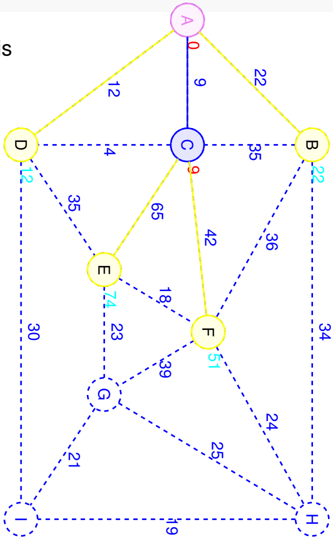


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0 \rangle$ into Q_H
3. while (Q_H is not empty) do
4. $\langle u, v, \ell \rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5. if ($u \neq v$) add edge $\langle u, v \rangle$ to T; label v with ℓ
6. foreach neighbour x of v which is not `visited`
7. if (x is `unvisited`)
8. mark x as `in-queue`
9. enqueue $\langle v, x, \ell + w(v, x) \rangle$ into Q_H
10. else // $\langle \_, x, q \rangle$ is `in-queue`
11. if ($q' [\leftarrow \ell + w(v, x)] < q$) $\langle, x, q \rangle \leftarrow \langle v, x, q' \rangle$
12. endfor
13. done

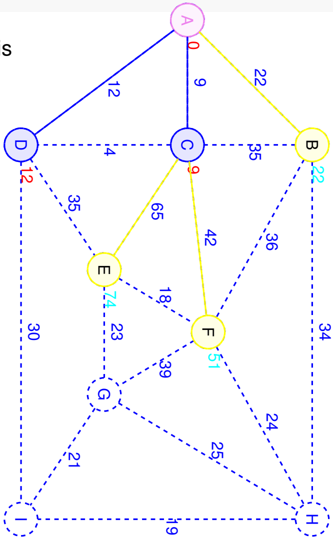


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

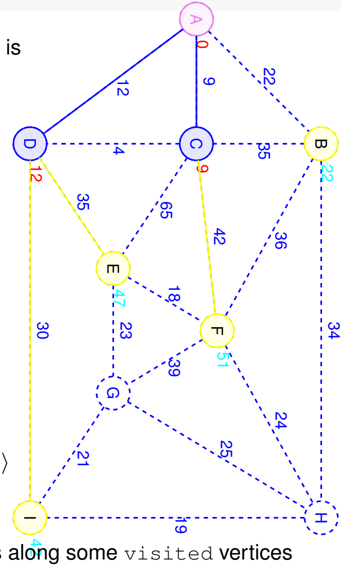
Start at v of $G(V,E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

all nodes of G are initially marked `unvisited`
set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
  while (Q_H is not empty) do
    $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
    if $(u \neq v)$ add edge $\langle u, v\rangle$ to T; label v with ℓ
    foreach neighbour x of v which is not `visited`
      if (x is `unvisited`)
        mark x as `in-queue`
        enqueue $\langle v, x, \ell + w(v,x)\rangle$ into Q_H
      else // $\langle \_, x, q\rangle$ is `in-queue`
        if $(q'[\leftarrow \ell + w(v,x)] < q)$ $\langle , x, q\rangle \leftarrow \langle v, x, q'\rangle$
    endfor
  done



Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)



Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

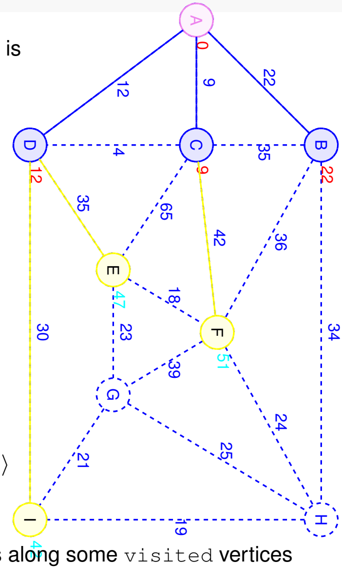
all nodes of G are initially marked `unvisited`
set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
  while (Q_H is not empty) do
    $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
    if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
    foreach neighbour x of v which is not `visited`
      if (x is `unvisited`)
        mark x as `in-queue`
        enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
      else // $\langle \_, x, q\rangle$ is `in-queue`
        if $(q'[\leftarrow \ell + w(v, x)] < q)$ $\langle, x, q\rangle \leftarrow \langle v, x, q'\rangle$
    endfor
  done

Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
3. while (Q_H is not empty) do
4. $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5. if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
6. foreach neighbour x of v which is not `visited`
7. if (x is `unvisited`)
8. mark x as `in-queue`
9. enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
10. else // $\langle \_, x, q\rangle$ is `in-queue`
11. if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle , x, q\rangle \leftarrow \langle v, x, q'\rangle$
12. endfor
13. done

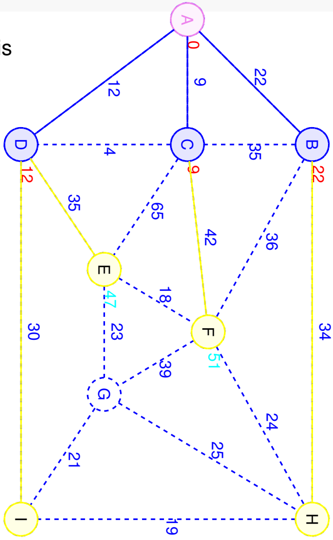


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

```
all nodes of G are initially marked unvisited
set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
  while (Q_H is not empty) do
    ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
    if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
    foreach neighbour x of v which is not visited
      if (x is unvisited)
        mark x as in-queue
        enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
      else // ⟨_, x, q⟩ is in-queue
        if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
    endfor
  done
```

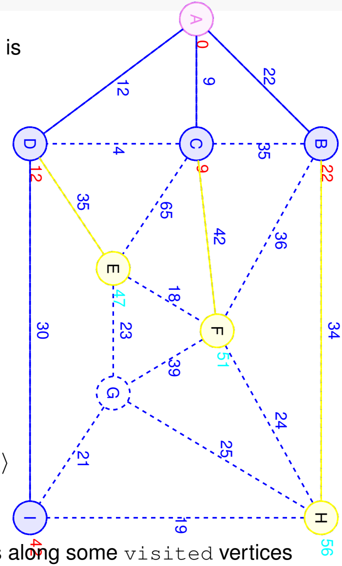


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

all nodes of G are initially marked `unvisited`
set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
  while (Q_H is not empty) do
    $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
    if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
    foreach neighbour x of v which is not `visited`
      if (x is `unvisited`)
        mark x as `in-queue`
        enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
      else // $\langle \_, x, q\rangle$ is `in-queue`
        if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle, x, q\rangle \leftarrow \langle v, x, q'\rangle$
    endfor
  done

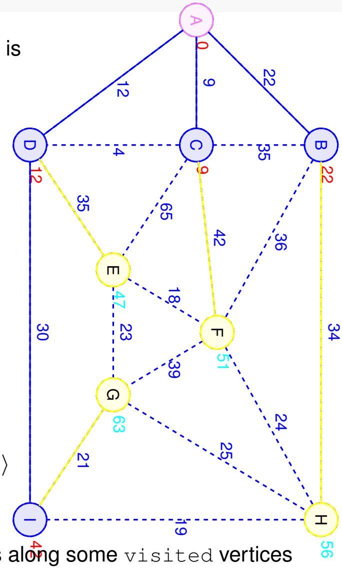


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

```
all nodes of G are initially marked unvisited
set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
  while (Q_H is not empty) do
    ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
    if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
    foreach neighbour x of v which is not visited
      if (x is unvisited)
        mark x as in-queue
        enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
      else // ⟨_, x, q⟩ is in-queue
        if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
    endfor
  done
```



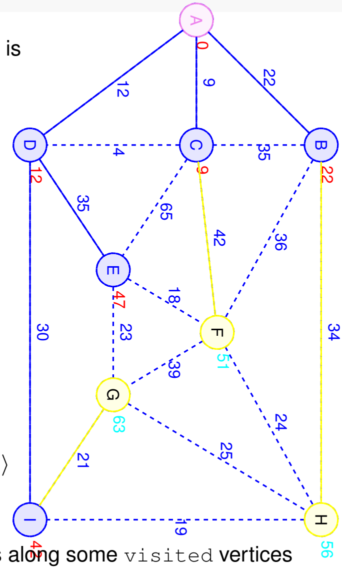
Invariants:

1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V,E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

- all nodes of G are initially marked `unvisited`
- set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
- while (Q_H is not empty) do
  - $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
  - if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
  - foreach neighbour x of v which is not `visited`
    - if (x is `unvisited`)
      - mark x as `in-queue`
      - enqueue $\langle v, x, \ell + w(v,x)\rangle$ into Q_H
    - else // $\langle \_, x, q\rangle$ is `in-queue`
      - if ($q' [\leftarrow \ell + w(v,x)] < q$) $\langle , x, q\rangle \leftarrow \langle v, x, q'\rangle$
  - endfor
- done

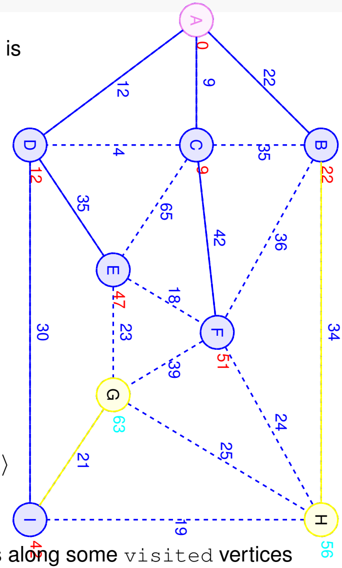


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
3.   while (Q_H is not empty) do
4. $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5.   if $(u \neq v)$ add edge $\langle u, v\rangle$ to T; label v with ℓ
6.   foreach neighbour x of v which is not `visited`
7.   if (x is `unvisited`)
8.   mark x as `in-queue`
9.   enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
10.   else // $\langle \_, x, q\rangle$ is `in-queue`
11.   if $(q'[\leftarrow \ell + w(v, x)] < q)$ $\langle, x, q\rangle \leftarrow \langle v, x, q'\rangle$
12.   endfor
13.   done

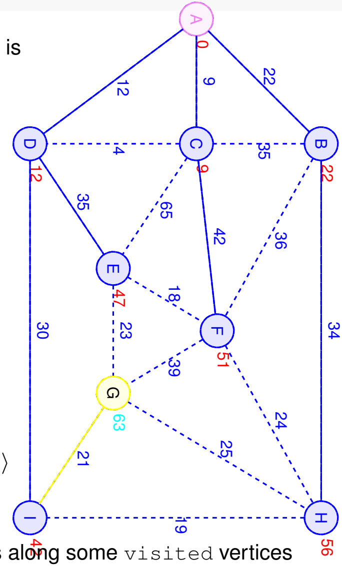


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

all nodes of G are initially marked `unvisited`
set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
  while (Q_H is not empty) do
    $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
    if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
    foreach neighbour x of v which is not `visited`
      if (x is `unvisited`)
        mark x as `in-queue`
        enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
      else // $\langle \_, x, q\rangle$ is `in-queue`
        if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle , x, q\rangle \leftarrow \langle v, x, q'\rangle$
    endfor
  done

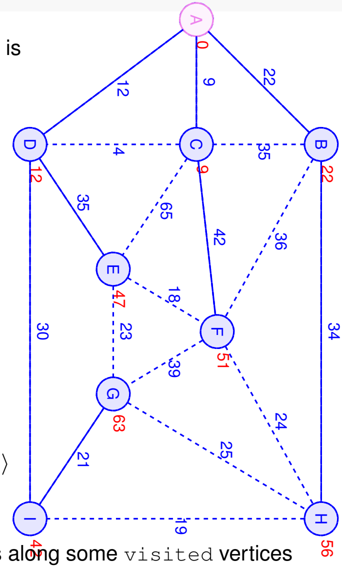


Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Dijkstra's algorithm (contd.)

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

all nodes of G are initially marked `unvisited`
set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
  while (Q_H is not empty) do
    $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
    if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
    foreach neighbour x of v which is not `visited`
      if (x is `unvisited`)
        mark x as `in-queue`
        enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
      else // $\langle \_, x, q\rangle$ is `in-queue`
        if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle, x, q\rangle \leftarrow \langle v, x, q'\rangle$
    endfor
  done



Dijkstra's algorithm terminates with the shortest path tree rooted at A

Invariants:
1. `in-queue` vertices are on shortest paths along some `visited` vertices
2. `visited` vertices are on shortest paths along some `visited` vertices

Correctness proof of Dijkstra's algorithm

Dijkstra's algorithm satisfies the stated invariants.

- Invariants are satisfied after the first node v is picked
- Let claim hold for vertices $X = \{v = x_1, x_2, \ldots, x_k\}$ picked up in L4
- The next vertex x_{k+1} picked up in L4 is the lowest cost `in-queue` vertex; so invariant-2 is not violated

Correctness proof of Dijkstra's algorithm

Dijkstra's algorithm satisfies the stated invariants.

- Invariants are satisfied after the first node v is picked
- Let claim hold for vertices $X = \{v = x_1, x_2, \ldots, x_k\}$ picked up in L4
- The next vertex x_{k+1} picked up in L4 is the lowest cost `in-queue` vertex; so invariant-2 is not violated
- Let z be any vertex in Q_H after x_{k+1} is chosen; let z be at the end of the edges $\langle v_{z_1}, z\rangle$, $\langle v_{z_2}, z\rangle$, $\ldots$, $\langle v_{z_j}, z\rangle$ so that the vertices in $Y = \{v_{z_1}, v_{z_2}, \ldots, v_{z_j}\} \subseteq X$ are all `visited` vertices; further, as the edge costs are always non-negative, the cost of any `in-queue` vertex is no less than the cost of any `visited` vertex
- The distance of z from v gets updated (relaxed) in L11

$\therefore$ Distance of any vertex z in Q_H is the shortest from v through a subset of the nodes in X; so invariant-1 is not violated

$\square$

Correctness proof of Dijkstra's algorithm (contd.)

- Termination of the algorithm is evident as a vertex is dequeued from Q_H in each iteration and marked `visited`
- Note that such a vertex also gets added to the shortest path tree as a successor to the vertex along which the shortest path to it from v was identified

Exercise: Dijkstra's algorithm with negative edge weights

- Check how Dijkstra's algorithm works when cost of edge $\langle B, F\rangle$ is -100 instead of 36
- In particular, following the steps of the algorithm, check whether any of the invariants get violated
- Next, check whether, the restriction of non-negative edge weights solves this problem

Complexity of Dijkstra's algorithm

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0 \rangle$ into Q_H
3. while (Q_H is not empty) do
4. $\langle u, v, \ell \rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5. if ($u \neq v$) add edge $\langle u, v \rangle$ to T; label v with ℓ
6. foreach neighbour x of v which is not `visited`
7. if (x is `unvisited`)
8. mark x as `in-queue`
9. enqueue $\langle v, x, \ell + w(v, x) \rangle$ into Q_H
10. else // $\langle \_, x, q \rangle$ is `in-queue`
11. if $(q' [\leftarrow \ell + w(v, x)] < q)$ $\langle , x, q \rangle \leftarrow \langle v, x, q' \rangle$
12. endfor
13. done

Complexity of Dijkstra's algorithm

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
3.   while (Q_H is not empty) do
4. $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5.   if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
6.   foreach neighbour x of v which is not `visited`
7.   if (x is `unvisited`)
8.   mark x as `in-queue`
9.   enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
10.   else // $\langle \_, x, q\rangle$ is `in-queue`
11.   if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle, x, q\rangle \leftarrow \langle v, x, q'\rangle$
12.   endfor
13.   done

- Edge weights assumed non-negative, except possibly those having v as the head;
- L4, requiring deletion from heap, iterates $|V|$ times, contributing $|V| \lg |V|$
- L9 or L11 happens $O(|E|)$ times, contributing $|E| \lg |V|$ for either key insertion or decreasing key cost at $O(\lg |V|)$ time (with binary heaps)
- Overall time complexity is $O((|V| + |E|) \lg |V|)$ (with binary heaps)

Complexity of Dijkstra's algorithm

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

```
1  all nodes of G are initially marked unvisited
2  set tree T to v; enqueue ⟨v, v, 0⟩ into Q_H
3    while (Q_H is not empty) do
4      ⟨u, v, ℓ⟩ ←dequeue(Q_H); mark v as visited
5      if (u ≠ v) add edge ⟨u, v⟩ to T; label v with ℓ
6      foreach neighbour x of v which is not visited
7        if (x is unvisited)
8          mark x as in-queue
9          enqueue ⟨v, x, ℓ + w(v, x)⟩ into Q_H
10       else // ⟨_, x, q⟩ is in-queue
11         if (q′[← ℓ + w(v, x)] < q) ⟨, x, q⟩ ← ⟨v, x, q′⟩
12     endfor
13   done
```

- Edge weights assumed non-negative, except possibly those having v as the head;
- L4, requiring deletion from heap, iterates $|V|$ times, contributing $|V| \lg |V|$
- L9 or L11 happens $O(|E|)$ times, contributing $|E| \lg |V|$ for either key insertion or decreasing key cost at $O(\lg |V|)$ time (with binary heaps)
- Overall time complexity is $O((|V| + |E|) \lg |V|)$ (with binary heaps)
- Fibonacci heaps allow insert key and decrease key in $O(1)$ time, reducing the time complexity to $O(|V| \lg |V| + |E|)$
- T is the shortest path (spanning) tree

Complexity of Dijkstra's algorithm

Start at v of $G(V, E)$; priority queue Q_H (or a min-heap Q_H) is used; node markings: `unvisited`, `visited`, `in-queue`

1. all nodes of G are initially marked `unvisited`
2. set tree T to v; enqueue $\langle v, v, 0\rangle$ into Q_H
3.   while (Q_H is not empty) do
4. $\langle u, v, \ell\rangle \leftarrow$dequeue($Q_H$); mark v as `visited`
5.   if ($u \neq v$) add edge $\langle u, v\rangle$ to T; label v with ℓ
6.   foreach neighbour x of v which is not `visited`
7.   if (x is `unvisited`)
8.   mark x as `in-queue`
9.   enqueue $\langle v, x, \ell + w(v, x)\rangle$ into Q_H
10.   else // $\langle \_, x, q\rangle$ is `in-queue`
11.   if ($q'[\leftarrow \ell + w(v, x)] < q$) $\langle , x, q\rangle \leftarrow \langle v, x, q'\rangle$
12.   endfor
13.   done

- Edge weights assumed non-negative, except possibly those having v as the head;
- L4, requiring deletion from heap, iterates $|V|$ times, contributing $|V| \lg |V|$
- L9 or L11 happens $O(|E|)$ times, contributing $|E| \lg |V|$ for either key insertion or decreasing key cost at $O(\lg |V|)$ time (with binary heaps)
- Overall time complexity is $O((|V| + |E|) \lg |V|)$ (with binary heaps)
- Fibonacci heaps allow insert key and decrease key in $O(1)$ time, reducing the time complexity to $O(|V| \lg |V| + |E|)$
- T is the shortest path (spanning) tree

Need to prove: Dijkstra's algorithm finds minimal cost paths to the vertices in the ascending order

Shortest distance between each pair of vertices

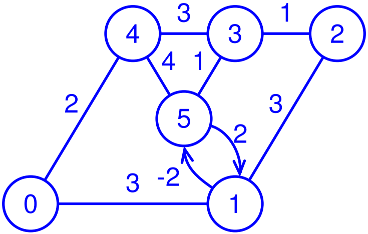
- Dijkstra's algorithm can be run for each vertex in the graph
- Running time will be $O(|V|(|V|+|E|)\lg|V|)$ (with binary heaps), $O(|V|(|V|\lg|V|+|E|))$ with Fibonacci heaps
- Also, Dijkstra's algorithm fails with negative edge weights
- Another formulation builds the optimal solution through optimal solution of overlapping sub-problems – **dynamic programming**

Section outline

5. **All shortest paths (Floyd-Warshall)**
   - Floyd-Warshall algorithm
   - Floyd-Warshall example
   - Distance between each pair of vertices through another vertex
   - General dynamic programming step for Floyd-Warshall algorithm
   - Floyd-Warshall's algorithm
   - Widest path problem
   - Other FW problems

Floyd-Warshall algorithm

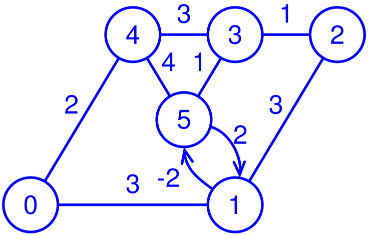
- We are given a weighted digraph, edge weights may be negative



- Initially we know the shortest distance between each pair of vertices going through 0 other vertices
- If there is no edge between two vertices, the initial cost ∞
- The distance of a vertex from itself is 0
- Otherwise, it is the usual directed edge cost between two vertices
- These costs are denoted as $d^0_{i,j}$

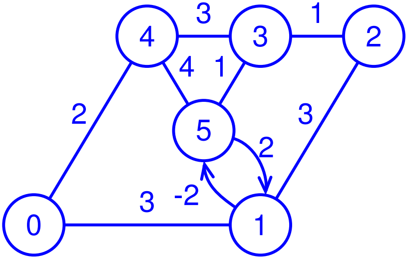
Floyd-Warshall algorithm

- We are given a weighted digraph, edge weights may be negative



- Initially we know the shortest distance between each pair of vertices going through 0 other vertices
- If there is no edge between two vertices, the initial cost ∞
- The distance of a vertex from itself is 0
- Otherwise, it is the usual directed edge cost between two vertices
- These costs are denoted as $d^0_{i,j}$
- At this stage the predecessor of a vertex j in the path $\pi^0_{i,j}$ from v_i to v_j is: if $d^0_{i,j} = \infty$ then $\pi^0_{i,j} = \text{NIL}$, otherwise $\pi^0_{i,j} = i$ (for v_i)

Floyd-Warshall example



E	0	1	2	3	4	5
0	$\langle 0, 0 \rangle$	$\langle 3, 0 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 2, 0 \rangle$	$\langle \infty, \phi \rangle$
1	$\langle 3, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle -2, 1 \rangle$
2	$\langle \infty, \phi \rangle$	$\langle 3, 2 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$
3	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 3 \rangle$	$\langle 0, 3 \rangle$	$\langle 3, 3 \rangle$	$\langle 1, 3 \rangle$
4	$\langle 2, 4 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 3, 4 \rangle$	$\langle 0, 4 \rangle$	$\langle 4, 4 \rangle$
5	$\langle \infty, \phi \rangle$	$\langle 2, 5 \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 5 \rangle$	$\langle 4, 5 \rangle$	$\langle 0, 5 \rangle$

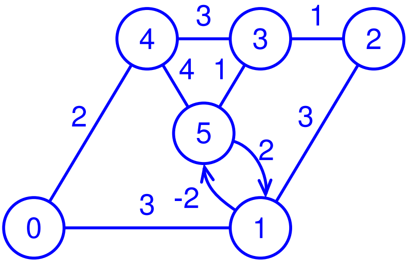
Distance between each pair of vertices through another vertex

- Next, find the shortest distance between each pair of vertices possibly going through vertex v_1
- If there is no improvement is going through v_1, the direct edge (if present) is used
- After this step the resulting cost $d^1_{i,j}$ is that of possibly going through v_1
- Thus, $d^1_{i,j} = \min\left(d^0_{i,j}, d^0_{i,1} + d^0_{1,j}\right)$

Distance between each pair of vertices through another vertex

- Next, find the shortest distance between each pair of vertices possibly going through vertex v_1
- If there is no improvement is going through v_1, the direct edge (if present) is used
- After this step the resulting cost $d^1_{i,j}$ is that of possibly going through v_1
- Thus, $d^1_{i,j} = \min\left(d^0_{i,j}, d^0_{i,1} + d^0_{1,j}\right)$
- With no improvement, path $\pi^1_{i,j} \leftarrow \pi^0_{i,j}$ (same as before)
- For improvement through v_1, in the path $\pi^1_{i,j}$, v_j will have the same predecessor that it has in the path $\pi^0_{1,j}$
- So, in case of improvement through v_1, $\pi^1_{i,j} \leftarrow \pi^0_{1,j}$

Floyd-Warshall example (contd.)



0	0	1	2	3	4	5
0	$\langle 0, 0 \rangle$	$\langle 3, 0 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 2, 0 \rangle$	$\langle \infty, \phi \rangle$
1	$\langle 3, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle \infty, \phi \rangle$	$\langle 5, 0 \rangle$	$\langle -2, 1 \rangle$
2	$\langle \infty, \phi \rangle$	$\langle 3, 2 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$
3	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 3 \rangle$	$\langle 0, 3 \rangle$	$\langle 3, 3 \rangle$	$\langle 1, 3 \rangle$
4	$\langle 2, 4 \rangle$	$\langle 5, 0 \rangle$	$\langle \infty, \phi \rangle$	$\langle 3, 4 \rangle$	$\langle 0, 4 \rangle$	$\langle 4, 4 \rangle$
5	$\langle \infty, \phi \rangle$	$\langle 2, 5 \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 5 \rangle$	$\langle 4, 5 \rangle$	$\langle 0, 5 \rangle$

General dynamic programming step for Floyd-Warshall algorithm

- In step k, we can find the shortest distance between each pair of vertices possibly going through v_k, using the paths possibly going through vertices in $\{v_1, \ldots, v_{k-1}\}$

General dynamic programming step for Floyd-Warshall algorithm

- In step k, we can find the shortest distance between each pair of vertices possibly going through v_k, using the paths possibly going through vertices in $\{v_1, \ldots, v_{k-1}\}$
- For cost reduction going through v_k, $d_{i,j}^{k} \leftarrow d_{i,k}^{k-1} + d_{k,j}^{k-1}$, otherwise, $d_{i,j}^{k} \leftarrow d_{i,j}^{k-1}$

General dynamic programming step for Floyd-Warshall algorithm

- In step k, we can find the shortest distance between each pair of vertices possibly going through v_k, using the paths possibly going through vertices in $\{v_1, \ldots, v_{k-1}\}$
- For cost reduction going through v_k, $d_{i,j}^k \leftarrow d_{i,k}^{k-1} + d_{k,j}^{k-1}$, otherwise, $d_{i,j}^k \leftarrow d_{i,j}^{k-1}$

That way, solutions of possibilities (sub-problems) of going through vertices in $\{v_1, \ldots, v_{k-1}\}$ is used to compute the benefit of going through v_k – this is the essence of *dynamic programming*

General dynamic programming step for Floyd-Warshall algorithm

- In step k, we can find the shortest distance between each pair of vertices possibly going through v_k, using the paths possibly going through vertices in $\{v_1, \ldots, v_{k-1}\}$
- For cost reduction going through v_k, $d_{i,j}^k \leftarrow d_{i,k}^{k-1} + d_{k,j}^{k-1}$, otherwise, $d_{i,j}^k \leftarrow d_{i,j}^{k-1}$

That way, solutions of possibilities (sub-problems) of going through vertices in $\{v_1, \ldots, v_{k-1}\}$ is used to compute the benefit of going through v_k – this is the essence of *dynamic programming*

- Thus, new cost is $\min\left(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\right)$

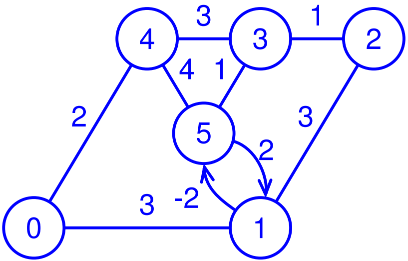
General dynamic programming step for Floyd-Warshall algorithm

- In step k, we can find the shortest distance between each pair of vertices possibly going through v_k, using the paths possibly going through vertices in $\{v_1, \ldots, v_{k-1}\}$
- For cost reduction going through v_k, $d^k_{i,j} \leftarrow d^{k-1}_{i,k} + d^{k-1}_{k,j}$, otherwise, $d^k_{i,j} \leftarrow d^{k-1}_{i,j}$

That way, solutions of possibilities (sub-problems) of going through vertices in $\{v_1, \ldots, v_{k-1}\}$ is used to compute the benefit of going through v_k – this is the essence of *dynamic programming*

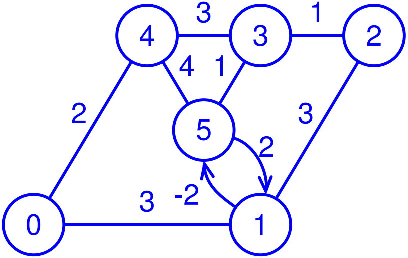
- Thus, new cost is $\min\left(d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j}\right)$
- For improvement through v_k, $\pi^k_{i,j} \leftarrow \pi^{k-1}_{k,j}$, otherwise $\pi^k_{i,j} \leftarrow \pi^{k-1}_{i,j}$

Floyd-Warshall example (contd.)



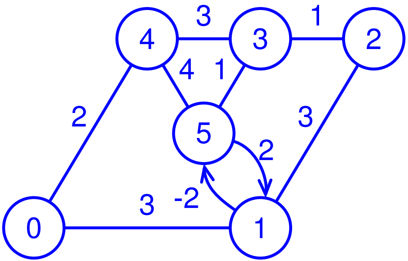
1	0	1	2	3	4	5
0	$\langle 0, 0\rangle$	$\langle 3, 0\rangle$	$\langle 6, 1\rangle$	$\langle \infty, \phi\rangle$	$\langle 2, 0\rangle$	$\langle 1, 1\rangle$
1	$\langle 3, 1\rangle$	$\langle 0, 1\rangle$	$\langle 3, 1\rangle$	$\langle \infty, \phi\rangle$	$\langle 5, 0\rangle$	$\langle -2, 1\rangle$
2	$\langle 6, 1\rangle$	$\langle 3, 2\rangle$	$\langle 0, 2\rangle$	$\langle 1, 2\rangle$	$\langle 8, 0\rangle$	$\langle 1, 1\rangle$
3	$\langle \infty, \phi\rangle$	$\langle \infty, \phi\rangle$	$\langle 1, 3\rangle$	$\langle 0, 3\rangle$	$\langle 3, 3\rangle$	$\langle 1, 3\rangle$
4	$\langle 2, 4\rangle$	$\langle 5, 0\rangle$	$\langle 8, 1\rangle$	$\langle 3, 4\rangle$	$\langle 0, 4\rangle$	$\langle 3, 1\rangle$
5	$\langle 5, 1\rangle$	$\langle 2, 5\rangle$	$\langle 5, 1\rangle$	$\langle 1, 5\rangle$	$\langle 4, 5\rangle$	$\langle 0, 5\rangle$

Floyd-Warshall example (contd.)



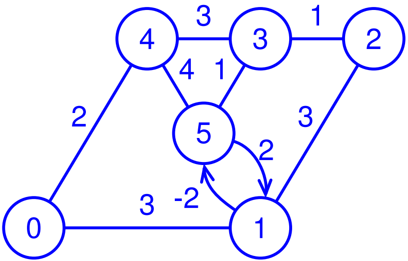
2	0	1	2	3	4	5
0	$\langle 0, 0 \rangle$	$\langle 3, 0 \rangle$	$\langle 6, 1 \rangle$	$\langle 7, 2 \rangle$	$\langle 2, 0 \rangle$	$\langle 1, 1 \rangle$
1	$\langle 3, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 5, 0 \rangle$	$\langle -2, 1 \rangle$
2	$\langle 6, 1 \rangle$	$\langle 3, 2 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle 8, 0 \rangle$	$\langle 1, 1 \rangle$
3	$\langle 7, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 1, 3 \rangle$	$\langle 0, 3 \rangle$	$\langle 3, 3 \rangle$	$\langle 1, 3 \rangle$
4	$\langle 2, 4 \rangle$	$\langle 5, 0 \rangle$	$\langle 8, 1 \rangle$	$\langle 3, 4 \rangle$	$\langle 0, 4 \rangle$	$\langle 3, 1 \rangle$
5	$\langle 5, 1 \rangle$	$\langle 2, 5 \rangle$	$\langle 5, 1 \rangle$	$\langle 1, 5 \rangle$	$\langle 4, 5 \rangle$	$\langle 0, 5 \rangle$

Floyd-Warshall example (contd.)



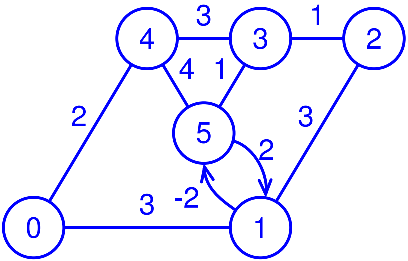
3	0	1	2	3	4	5
0	$\langle 0, 0\rangle$	$\langle 3, 0\rangle$	$\langle 6, 1\rangle$	$\langle 7, 2\rangle$	$\langle 2, 0\rangle$	$\langle 1, 1\rangle$
1	$\langle 3, 1\rangle$	$\langle 0, 1\rangle$	$\langle 3, 1\rangle$	$\langle 4, 2\rangle$	$\langle 5, 0\rangle$	$\langle -2, 1\rangle$
2	$\langle 6, 1\rangle$	$\langle 3, 2\rangle$	$\langle 0, 2\rangle$	$\langle 1, 2\rangle$	$\langle 4, 3\rangle$	$\langle 1, 1\rangle$
3	$\langle 7, 1\rangle$	$\langle 4, 2\rangle$	$\langle 1, 3\rangle$	$\langle 0, 3\rangle$	$\langle 3, 3\rangle$	$\langle 1, 3\rangle$
4	$\langle 2, 4\rangle$	$\langle 5, 0\rangle$	$\langle 4, 3\rangle$	$\langle 3, 4\rangle$	$\langle 0, 4\rangle$	$\langle 3, 1\rangle$
5	$\langle 5, 1\rangle$	$\langle 2, 5\rangle$	$\langle 2, 3\rangle$	$\langle 1, 5\rangle$	$\langle 4, 5\rangle$	$\langle 0, 5\rangle$

Floyd-Warshall example (contd.)



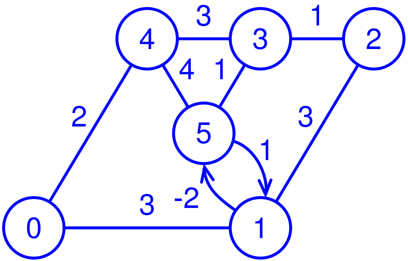
4	0	1	2	3	4	5
0	$\langle 0, 0 \rangle$	$\langle 3, 0 \rangle$	$\langle 6, 1 \rangle$	$\langle 5, 4 \rangle$	$\langle 2, 0 \rangle$	$\langle 1, 1 \rangle$
1	$\langle 3, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle 4, 2 \rangle$	$\langle 5, 0 \rangle$	$\langle -2, 1 \rangle$
2	$\langle 6, 1 \rangle$	$\langle 3, 2 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle 4, 3 \rangle$	$\langle 1, 1 \rangle$
3	$\langle 5, 4 \rangle$	$\langle 4, 2 \rangle$	$\langle 1, 3 \rangle$	$\langle 0, 3 \rangle$	$\langle 3, 3 \rangle$	$\langle 1, 3 \rangle$
4	$\langle 2, 4 \rangle$	$\langle 5, 0 \rangle$	$\langle 4, 3 \rangle$	$\langle 3, 4 \rangle$	$\langle 0, 4 \rangle$	$\langle 3, 1 \rangle$
5	$\langle 5, 1 \rangle$	$\langle 2, 5 \rangle$	$\langle 2, 3 \rangle$	$\langle 1, 5 \rangle$	$\langle 4, 5 \rangle$	$\langle 0, 5 \rangle$

Floyd-Warshall example (contd.)



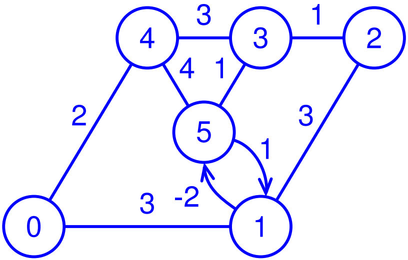
5	0	1	2	3	4	5
0	$\langle 0, 0\rangle$	$\langle 3, 0\rangle$	$\langle 3, 3\rangle$	$\langle 2, 5\rangle$	$\langle 2, 0\rangle$	$\langle 1, 1\rangle$
1	$\langle 3, 1\rangle$	$\langle 0, 1\rangle$	$\langle 0, 3\rangle$	$\langle -1, 5\rangle$	$\langle 2, 5\rangle$	$\langle -2, 1\rangle$
2	$\langle 6, 1\rangle$	$\langle 3, 2\rangle$	$\langle 0, 2\rangle$	$\langle 1, 2\rangle$	$\langle 4, 3\rangle$	$\langle 1, 1\rangle$
3	$\langle 5, 4\rangle$	$\langle 3, 5\rangle$	$\langle 1, 3\rangle$	$\langle 0, 3\rangle$	$\langle 3, 3\rangle$	$\langle 1, 3\rangle$
4	$\langle 2, 4\rangle$	$\langle 5, 0\rangle$	$\langle 4, 3\rangle$	$\langle 3, 4\rangle$	$\langle 0, 4\rangle$	$\langle 3, 1\rangle$
5	$\langle 5, 1\rangle$	$\langle 2, 5\rangle$	$\langle 2, 3\rangle$	$\langle 1, 5\rangle$	$\langle 4, 5\rangle$	$\langle 0, 5\rangle$

Faulty Floyd-Warshall example with -ve cost cycle



0	0	1	2	3	4	5
0	$\langle 0, 0 \rangle$	$\langle 3, 0 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 2, 0 \rangle$	$\langle \infty, \phi \rangle$
1	$\langle 3, 1 \rangle$	$\langle 0, 1 \rangle$	$\langle 3, 1 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle -2, 1 \rangle$
2	$\langle \infty, \phi \rangle$	$\langle 3, 2 \rangle$	$\langle 0, 2 \rangle$	$\langle 1, 2 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$
3	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 3 \rangle$	$\langle 0, 3 \rangle$	$\langle 3, 3 \rangle$	$\langle 1, 3 \rangle$
4	$\langle 2, 4 \rangle$	$\langle \infty, \phi \rangle$	$\langle \infty, \phi \rangle$	$\langle 3, 4 \rangle$	$\langle 0, 4 \rangle$	$\langle 4, 4 \rangle$
5	$\langle \infty, \phi \rangle$	$\langle 1, 5 \rangle$	$\langle \infty, \phi \rangle$	$\langle 1, 5 \rangle$	$\langle 4, 5 \rangle$	$\langle 0, 5 \rangle$

Faulty Floyd-Warshall example (contd.)



6	0	1	2	3	4	5
0	⟨0, 0⟩	⟨2, 5⟩	⟨3, 3⟩	⟨2, 5⟩	⟨2, 0⟩	⟨0, 1⟩
1	⟨2, 1⟩	⟨−1, 5⟩	⟨0, 3⟩	⟨−1, 5⟩	⟨2, 5⟩	⟨−3, 1⟩
2	⟨5, 1⟩	⟨2, 5⟩	⟨0, 2⟩	⟨1, 2⟩	⟨4, 3⟩	⟨0, 1⟩
3	⟨5, 4⟩	⟨2, 5⟩	⟨1, 3⟩	⟨0, 3⟩	⟨3, 3⟩	⟨0, 1⟩
4	⟨2, 4⟩	⟨4, 5⟩	⟨4, 3⟩	⟨3, 4⟩	⟨0, 4⟩	⟨2, 1⟩
5	⟨3, 1⟩	⟨0, 5⟩	⟨1, 3⟩	⟨0, 5⟩	⟨3, 5⟩	⟨−2, 1⟩

Note the presence of negative cost entries in the diagonal elements

Floyd-Warshall's algorithm

$d^0_{i,j} = D_{i,j}$ if v_j is adjacent to v_i, otherwise ∞

$\pi^0_{i,j} = i$ if v_j is adjacent to v_i, otherwise NIL

for $k = 1$ to n do

  for $i = 1$ to n do

    for $j = 1$ to n do

      if $\left(d\left(\leftarrow d^{k-1}_{i,k} + d^{k-1}_{k,j}\right) < d^{k-1}_{i,j}\right)$

        $d^k_{i,j} \leftarrow d$; $\pi^k_{i,j} \leftarrow \pi^{k-1}_{k,j}$

      else

        $d^k_{i,j} \leftarrow d^{k-1}_{i,j}$; $\pi^k_{i,j} \leftarrow \pi^{k-1}_{i,j}$

    endfor

  endfor

endfor

- Time complexity: $\Theta(n^3)$
- Space complexity: $\Theta(n^2)$

Floyd-Warshall's algorithm

1. $d^0_{i,j} = D_{i,j}$ if v_j is adjacent to v_i, otherwise ∞
2. $\pi^0_{i,j} = i$ if v_j is adjacent to v_i, otherwise NIL
3. for $k = 1$ to n do
4.   for $i = 1$ to n do
5.   for $j = 1$ to n do
6.   if $\left(d \left(\leftarrow d^{k-1}_{i,k} + d^{k-1}_{k,j}\right) < d^{k-1}_{i,j}\right)$
7. $d^k_{i,j} \leftarrow d$; $\pi^k_{i,j} \leftarrow \pi^{k-1}_{k,j}$
8.   else
9. $d^k_{i,j} \leftarrow d^{k-1}_{i,j}$; $\pi^k_{i,j} \leftarrow \pi^{k-1}_{i,j}$
10.   endfor
11.   endfor
12. endfor

- Time complexity: $\Theta(n^3)$
- Space complexity: $\Theta(n^2)$
- Works with negative edge weights
- $d^k_{i,j} < 0$ in the presence of negative weight cycles

Widest path problem

Definition (All widest paths)

- Finding a path between all pairs of vertices in a weighted graph to maximise the weight of the minimum weight edge in the path
- Also known as the **bottleneck shortest path problem** or the **maximum capacity path problem**

$d_{i,j}^0 = D_{i,j}$ if $\langle v_i, v_j \rangle \in E$, otherwise 0

$\pi_{i,j}^0 = i$ if $\langle v_i, v_j \rangle \in E$, otherwise NIL

for $k = 1$ to n do

    for $i = 1$ to n do

        for $j = 1$ to n do

            if $\left(d \left(\leftarrow \min\left(d_{i,k}^{k-1}, d_{k,j}^{k-1}\right)\right) > d_{i,j}^{k-1}\right)$

                $d_{i,j}^k \leftarrow d$; $\pi_{i,j}^k \leftarrow \pi_{k,j}^{k-1}$

            else

                $d_{i,j}^k \leftarrow d_{i,j}^{k-1}$; $\pi_{i,j}^k \leftarrow \pi_{i,j}^{k-1}$

        endfor

    endfor

endfor

Other FW problems

Definition (All narrowest paths)

Finding a path between all pairs of vertices in a weighted graph to minimise the weight of the maximum weight edge in the path

Definition (All safest paths)

Given a graph with direct survival probabilities between vertices, find the least dangerous path between pairs of vertices

Definition (All reachable paths)

Given a 0/1 weighted graph for direct connectivity between vertices, find the reachable pairs of vertices

Section outline

6 **Minimum cost spanning tree**
- Spanning tree variety
- Kruskal's algorithm
- Analysis of Kruskal's algorithm
- Optimality of Kruskal's algorithm
- Prim's algorithm
- Analysis of Prim's algorithm
- Optimality of Prim's algorithm

Spanning tree variety

Some of the spanning trees seen so far:

- Spanning tree from a depth first traversal
- Spanning tree from a breadth first traversal
- Shortest path spanning tree from Dijkstra's single source shortest paths algorithm

Spanning tree variety

Some of the spanning trees seen so far:

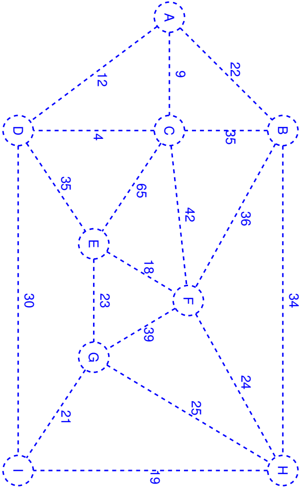
- Spanning tree from a depth first traversal
- Spanning tree from a breadth first traversal
- Shortest path spanning tree from Dijkstra's single source shortest paths algorithm
- We now seek to find a spanning tree that minimises the sum total of the edge costs – called the minimum cost spanning tree or just minimum spanning tree (MST)
- Kruskal's, Prim's, Boruvka's and hybrid Boruvka-Prim algorithms for building MSTs of connected undirect graphs are considered

Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

    do
4. extract min cost edge $e \in H$
5. if (e does not cause

    a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

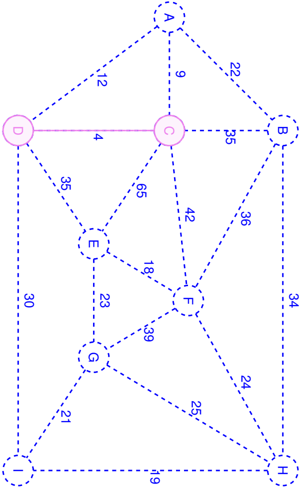


Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

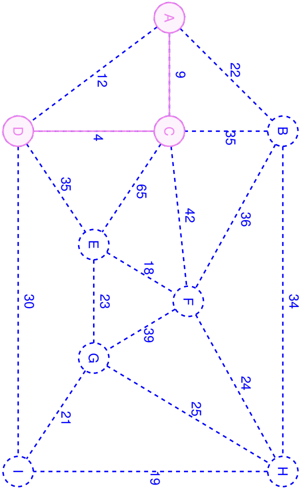


Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

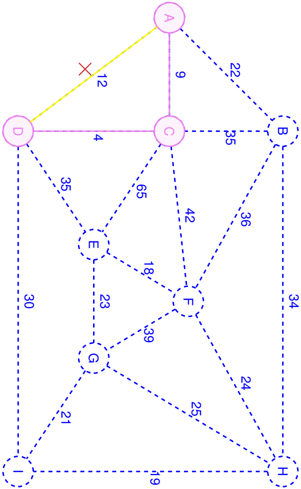
   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

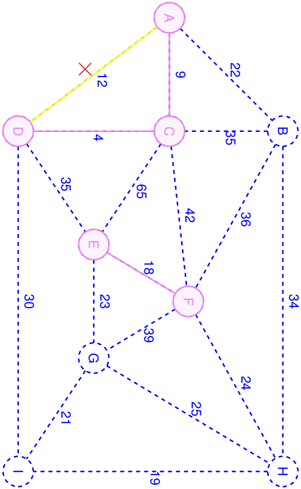


Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

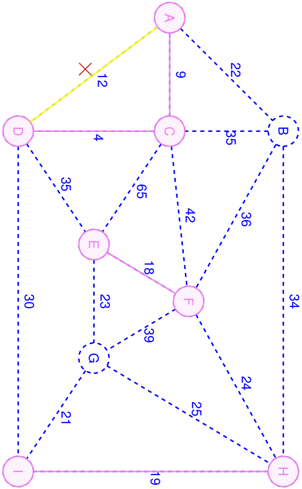


Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

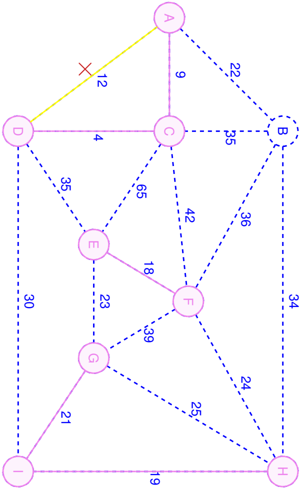
   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



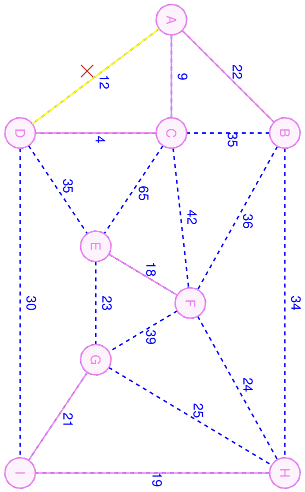
Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

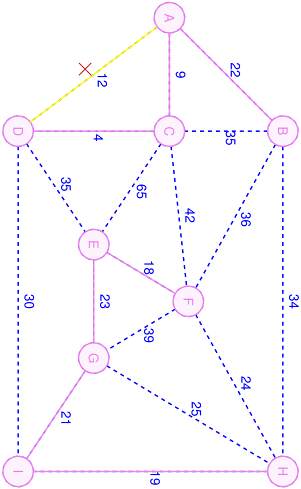


Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while ($H \neq \phi \wedge c < |V| - 1$)

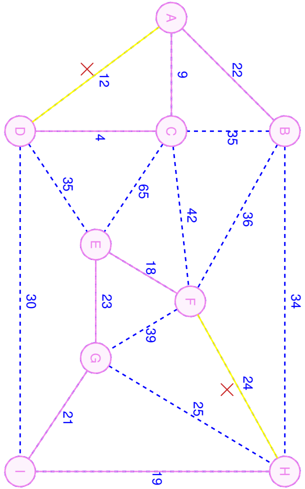
   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



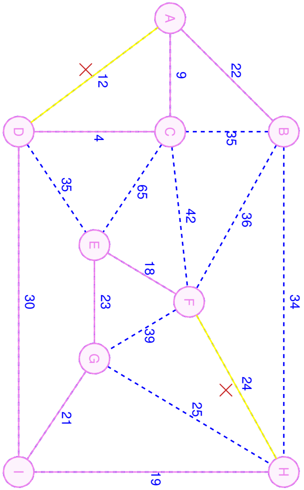
Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done



Analysis of Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$

   do
4. extract min cost edge $e \in H$
5. if (e does not cause

   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

Analysis of Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while $(H \neq \phi \wedge c < |V| - 1)$ do
4. extract min cost edge $e \in H$
5. if (e does not cause a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

- Heap creation time is $O(|E|)$ in L1
- All edges may be removed from H in L4
- Total time to execute L4 is $O(|E| \lg |E|)$ ($\in$ $O(|E| \lg |V|)$, as $|E| \in O(|V|^2)$)

Analysis of Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while ($H \neq \phi \wedge c < |V| - 1$) do
4. extract min cost edge $e \in H$
5. if (e does not cause a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

- Heap creation time is $O(|E|)$ in L1
- All edges may be removed from H in L4
- Total time to execute L4 is $O(|E| \lg |E|)$ ($\in$ $O(|E| \lg |V|)$, as $|E| \in O(|V|^2)$)
- Cycle creation may be checked (at L5) by starting a DF traversal at one vertex of the edge e and monitoring whether the other end of e is reached

  Each check will take $O(|V| - 1)$ time

Analysis of Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while ($H \neq \phi \wedge c < |V| - 1$)
   do
4. extract min cost edge $e \in H$
5. if (e does not cause
   a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

- Heap creation time is $O(|E|)$ in L1
- All edges may be removed from H in L4
- Total time to execute L4 is $O(|E| \lg |E|)$ ($\in$ $O(|E| \lg |V|)$, as $|E| \in O(|V|^2)$)
- Cycle creation may be checked (at L5) by starting a DF traversal at one vertex of the edge e and monitoring whether the other end of e is reached

  Each check will take $O(|V| - 1)$ time
- This may have to be done for each edge leading to an aggregate cost of $O(|E|(|V| - 1))$ time

Analysis of Kruskal's algorithm

1. create a min heap H of the edges
2. initialise $T \leftarrow \phi$; edge count $c \leftarrow 0$
3. while ($H \neq \phi \wedge c < |V| - 1$) do
4. extract min cost edge $e \in H$
5. if (e does not cause a cycle in T)
6. add e to T; $c \leftarrow c + 1$
7. done

- Heap creation time is $O(|E|)$ in L1
- All edges may be removed from H in L4
- Total time to execute L4 is $O(|E| \lg |E|)$ ($\in$ $O(|E| \lg |V|)$, as $|E| \in O(|V|^2)$)
- Cycle creation may be checked (at L5) by starting a DF traversal at one vertex of the edge e and monitoring whether the other end of e is reached

  Each check will take $O(|V| - 1)$ time
- This may have to be done for each edge leading to an aggregate cost of $O(|E|(|V| - 1))$ time
- Using DSUF, cycle checking is done in $O(|V| \lg^* |V|) \approx O(|V|)$ time

Optimality of Kruskal's algorithm

Let T be MST of $G(V, E)$ found by KMST, and assume (wrongly) that there exists another minimum spanning tree S, such that $W(S) < W(T)$

1. Let e_k be the least cost edge in T but not in S; $\{e_1, \ldots, e_{k-1}\} \subset T, S$
2. Consider adding edge e_k to S
   - This creates a cycle C in S, containing e_k
   - Cycle C contains an edge e not in T

Optimality of Kruskal's algorithm

Let T be MST of $G(V, E)$ found by KMST, and assume (wrongly) that there exists another minimum spanning tree S, such that $W(S) < W(T)$

1. Let e_k be the least cost edge in T but not in S; $\{e_1, \ldots, e_{k-1}\} \subset T, S$
2. Consider adding edge e_k to S
   - This creates a cycle C in S, containing e_k
   - Cycle C contains an edge e not in T, otherwise T has cycle C

Optimality of Kruskal's algorithm

Let T be MST of $G(V, E)$ found by KMST, and assume (wrongly) that there exists another minimum spanning tree S, such that $W(S) < W(T)$

1. Let e_k be the least cost edge in T but not in S; $\{e_1, \ldots, e_{k-1}\} \subset T, S$
2. Consider adding edge e_k to S
   - This creates a cycle C in S, containing e_k
   - Cycle C contains an edge e not in T, otherwise T has cycle C
3. If we drop e and add e_k in S we get a spanning tree S' where
   - cost of $e_k <=$ cost of e, for otherwise, e would have been chosen in preference to e in T by KMST without getting a cycle
   - S' is now one edge closer to T than S

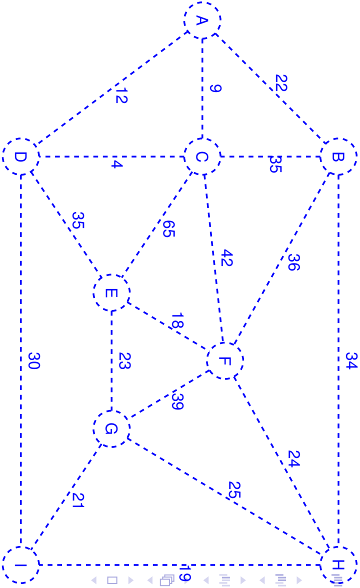
Optimality of Kruskal's algorithm

Let T be MST of $G(V, E)$ found by KMST, and assume (wrongly) that there exists another minimum spanning tree S, such that $W(S) < W(T)$

1. Let e_k be the least cost edge in T but not in S; $\{e_1, \ldots, e_{k-1}\} \subset T, S$
2. Consider adding edge e_k to S
   - This creates a cycle C in S, containing e_k
   - Cycle C contains an edge e not in T, otherwise T has cycle C
3. If we drop e and add e_k in S we get a spanning tree S' where
   - cost of $e_k <=$ cost of e, for otherwise, e would have been chosen in preference to e in T by KMST without getting a cycle
   - S' is now one edge closer to T than S
4. Now $W(S') <= W(S)$; above steps can be reiterated until $S' = T$ to terminate with $W(T) = W(S') <= W(S)$
5. This contradicts our initial assumption, that there can be another MST S with $W(S) < W(T)$
6. So, KMST finds an MST

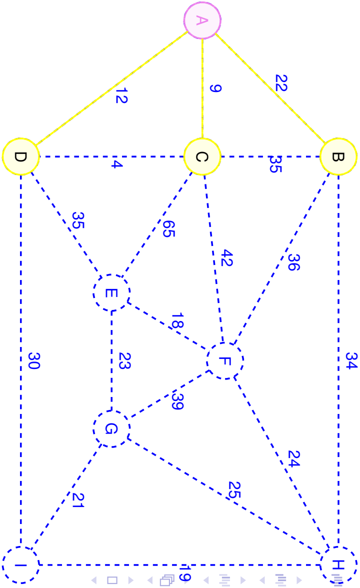
Prim's algorithm

1. initialise $T \leftarrow v, v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



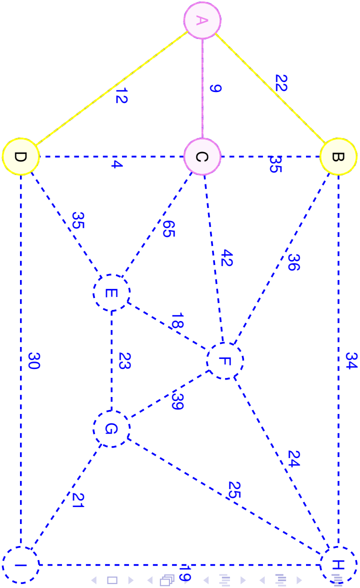
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



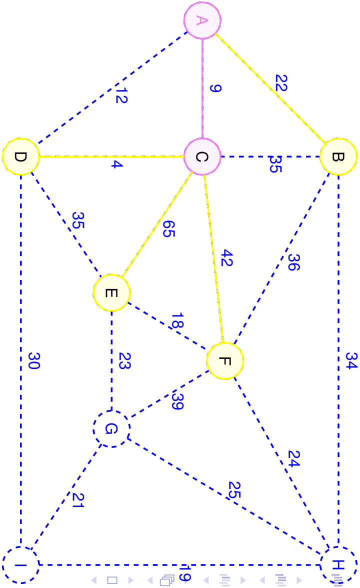
Prim's algorithm

1. initialise $T \leftarrow v, v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



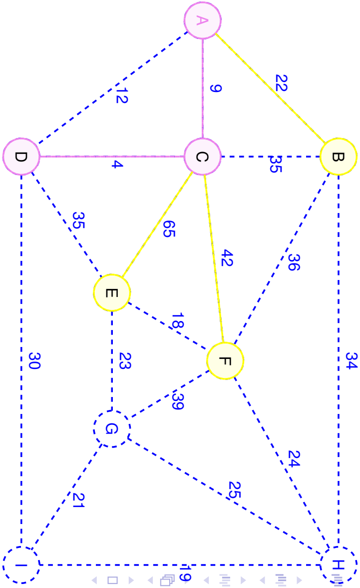
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while ($n_e < |V| - 1$)
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$,
   possibly reduce $C(x)$ in H
   to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$
   ($\equiv \langle x, v \rangle \in E, x \in T, v \notin T$)
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



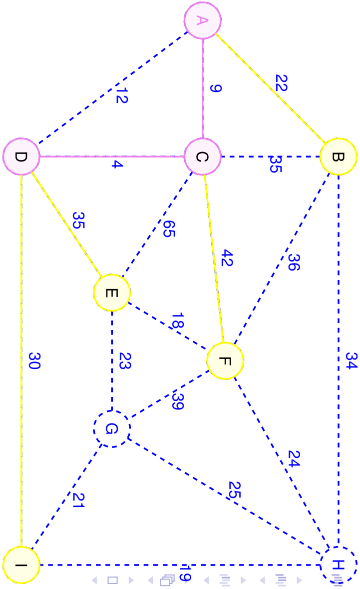
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while ($n_e < |V| - 1$)
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



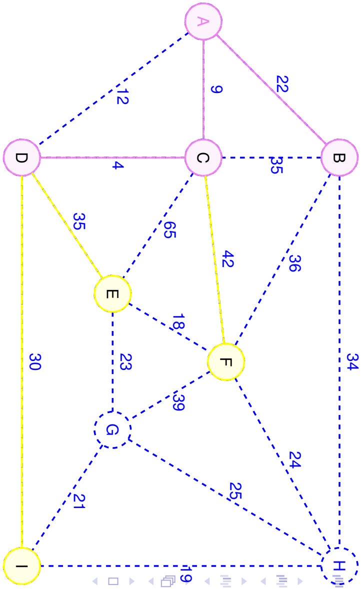
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



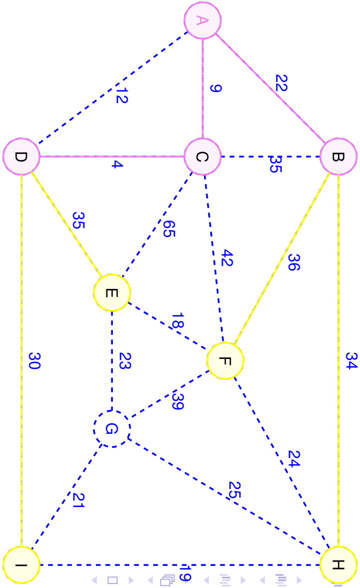
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



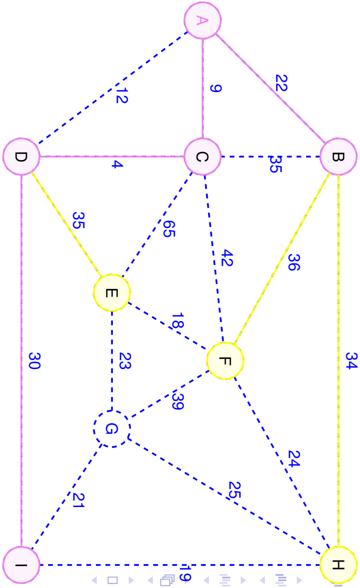
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T,$ *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



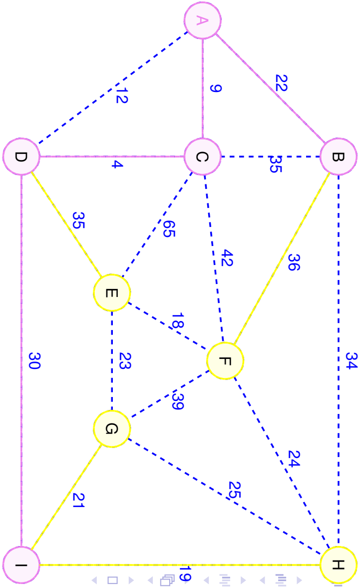
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



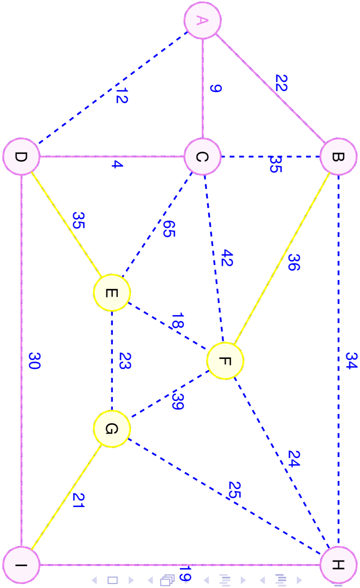
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



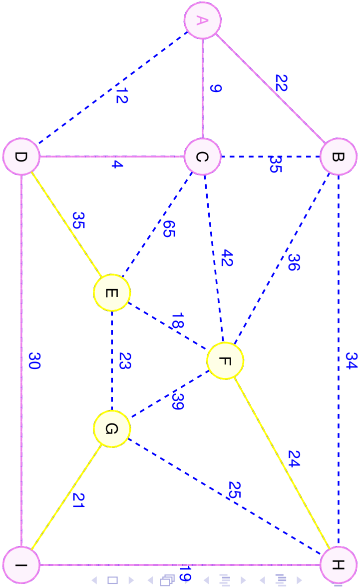
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x |\, \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



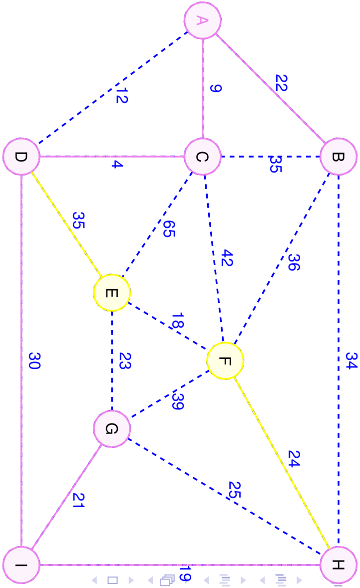
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



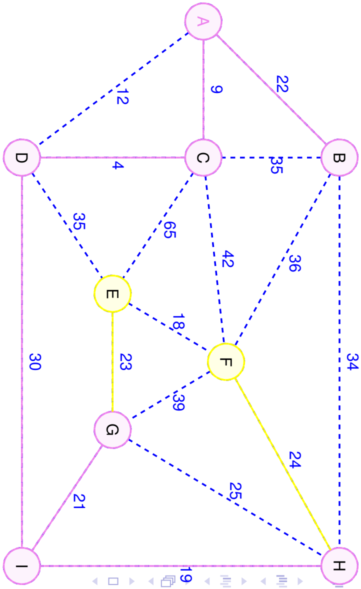
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T,$ *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



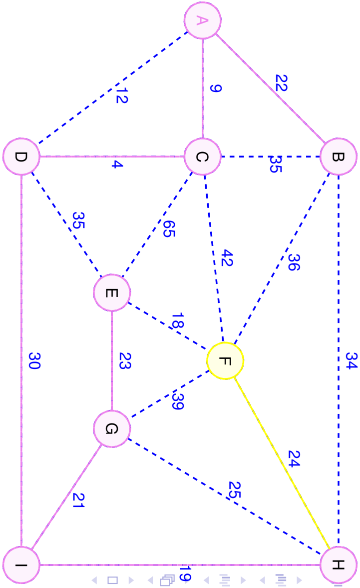
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$,
   possibly reduce $C(x)$ in H
   to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$
   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$,
   possibly reduce $C(x)$ in H
   to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$
   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



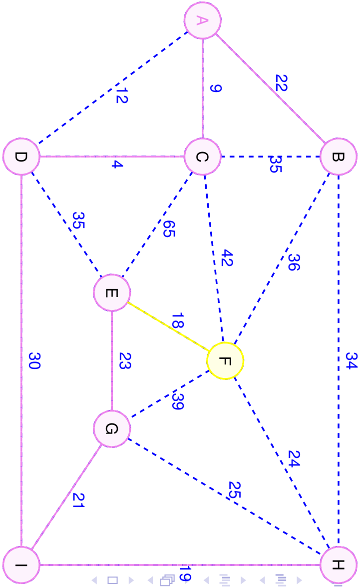
Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x |\, \langle v, x \rangle \in E, v \in T, x \notin T$,

   possibly reduce $C(x)$ in H

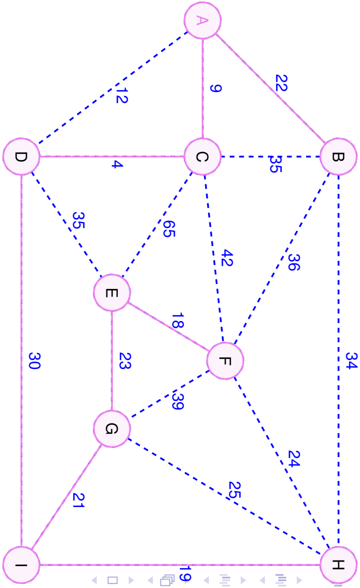
   to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$

   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x |\, \langle v, x \rangle \in E, v \in T, x \notin T,$ *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done



Analysis of Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T,$
   possibly reduce $C(x)$ in H
   to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$
   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done

Analysis of Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T$,

   possibly reduce $C(x)$ in H

   to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$

   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done

- Testing $x \notin T$ in L3 done via marking vertices added to T
- Major contribution is from L3 (put e into H, by way of decrease key value) which can take $O(|E| \lg |V|)$ time (binary heap)

Analysis of Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\forall x | \langle v, x \rangle \in E, v \in T, x \notin T$, *possibly* reduce $C(x)$ in H to $w(\langle v, x \rangle)$
4. extract min cost vertex $v \in H$ $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done

- Testing $x \notin T$ in L3 done via marking vertices added to T
- Major contribution is from L3 (put e into H, by way of decrease key value) which can take $O(|E| \lg |V|)$ time (binary heap)
- L4 also done $|V| - 1$ times contributes $O(|V| \lg |V|)$ time
- Overall time complexity is $O(|V| + |E|) \lg |V|$ (binary heap)

Analysis of Prim's algorithm

1. initialise $T \leftarrow v$, $v \in V$; edge count $n_e \leftarrow 0$, build H for all $x \in V, x \neq v, C(x) = \infty$
2. while $(n_e < |V| - 1)$
3. $\quad \forall x | \langle v, x \rangle \in E, v \in T, x \notin T,$
   possibly reduce $C(x)$ in H
   to $w(\langle v, x \rangle)$
4. $\quad$ extract min cost vertex $v \in H$
   $(\equiv \langle x, v \rangle \in E, x \in T, v \notin T)$
5. $\quad$ add $\langle x, v \rangle$ to T; $n_e \leftarrow n_e + 1$
6. done

- Testing $x \notin T$ in L3 done via marking vertices added to T
- Major contribution is from L3 (put e into H, by way of decrease key value) which can take $O(|E| \lg |V|)$ time (binary heap)
- L4 also done $|V| - 1$ times contributes $O(|V| \lg |V|)$ time
- Overall time complexity is $O(|V| + |E|) \lg |V|$ (binary heap)
- Using a Fibonacci heap, the time complexity is $O(|E| + |V| \lg |V|)$ as decrease key value in a Fibonacci heap takes $O(1)$ time (amortised)

Optimality of Prim's algorithm

The optimality of Prim's algorithm is proven by induction with the induction hypothesis:

There is always an MST having each intermediate spanning tree created by Prim's algorithm

1. Let T_i be the intermediate spanning tree after adding edge $\langle x_i, v_i \rangle$ to T; x_i is a vertex in T_{i-1}
2. A graph can have several spanning trees of the minimum weight; let S_i be an MST tree such that T_i is a subgraph of S_i;
3. S_1 definitely exists (base case is satisfied)
4. Let there be spanning trees $S_1 \supseteq T_1, \ldots, S_k \supseteq T_k$
5. Let S_{k+1} not exist, so let $S'_k = S_k \cup \{\langle x_{k+1}, v_{k+1} \rangle\}$, $x_{k+1} \in T_k$
6. S'_k will have a cycle containing $\langle u, v \rangle \neq \langle x_{k+1}, v_{k+1} \rangle$, $u \in T_k$, $v \notin T_k$
7. Cost of edge $\langle u, v \rangle \geq$ cost of edge $\langle x_{k+1}, v_{k+1} \rangle$, for otherwise PMST would have selected $\langle u, v \rangle$ earlier to $\langle x_{k+1}, v_{k+1} \rangle$
8. Thus, $W(S'_k \setminus \langle u, v \rangle) < W(S_k)$ so $S'_k \equiv S_{k+1}$, contradicting claim