DFS-Matroids-MST

Pierre Aboulker - pierreaboulker@gmail.com

Program of the day:

- Depth First Search (DFS)
  - Implementation
  - Application 1: compute a topological ordering
  - Application 2: compute the strong connected components
- Minimum Spanning tree (Kruskal and Primm Algorithm)
- Priority queue
- Matroids and greedy algorithm

1 - Graph Basics

Basics

All along the course, particularly for complexity analysis,

- n is the number of **vertices**,
- m is the number of **edges**.

An algorithm going in time $O(n + m)$ is said to be linear.

Basics

All along the course, particularly for complexity analysis,

- n is the number of **vertices**,
- m is the number of **edges**.

An algorithm going in time $O(n+m)$ is said to be linear.

- K_n denotes the complete graph on n vertices.
- $G = (U, V, E)$ a bipartite graph, means U and V is the partition.
- $K_{a,b}$ denotes the complete bipartite graph with parts of size a and b.

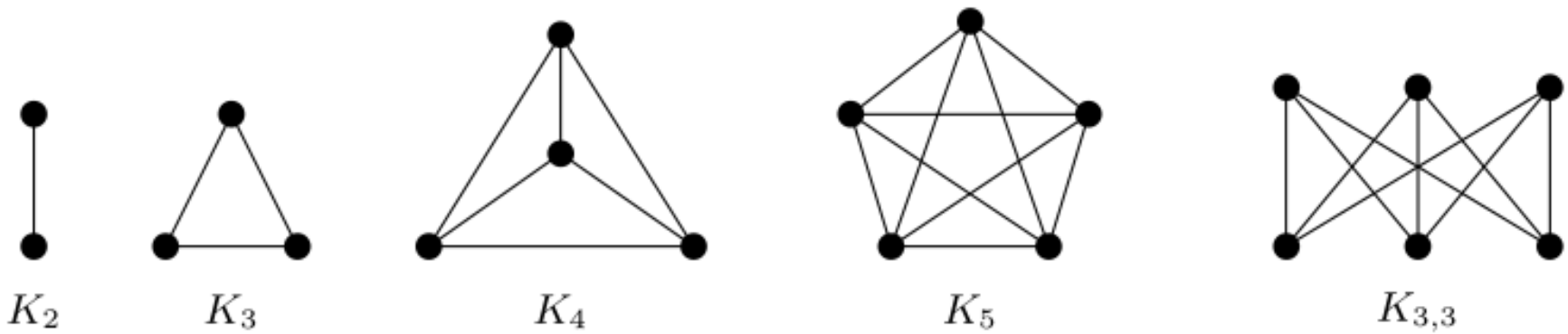


figure from *graphs, network and algorithms*, 4^{th} edition, Dieter Jungnickel

Subgraph containments

Let G and H be two graphs. We say that H is a:

- subgraph of G if H can be obtained from G by deleting vertices and edges
- induced subgraph of G if H can be obtained from G by deleting vertices.
- subdivision of G if H can be obtained from G by subdividing some edges.
- minor of G if H can be obtained from G by deleting vertices, edges and contracting edges.

Only subgraph and induced subgraph will be met in this course.

Graph parameters

- $\delta(G)$: minimum degree.
- $\Delta(G)$: maximum degree.
- $\omega(G)$: clique number.
- $\alpha(G)$: size of a maximum independent set.
- $\chi(G)$: chromatic number.
- $\tau(G)$: vertex cover.
- $\kappa(G)$: vertex connectivity.
- $\lambda(G)$: edge connectivity.
- $\mu(G)$: size of a maximum matching (disjoint edges).
- Girth: length of a shortest cycle of G.
- $\chi'(G)$: chromatic index (edge coloring).
- $tw(G)$: treewidth, measure how much a graph looks like a tree.
- $rkw(G)$, $cw(G)$, $tww(G)$, several notions of *width* exist, they mesure how complicated a graph is.

Some graphs problems

- Path. Is there a path between s and t?
- Shortest path. What is the shortest path between s and t?
- Cycle. Is there a cycle in the graph?
- Hamiltonian cycle. Is there a cycle that uses each vertex
- Connectivity. Is the graph connected?
- MST. What is the best way to connect all of the vertices?
- k-connectivity. Is there a set of at most k vertices whose removal disconnects the graph?
- Vertex cover. Find a minimum set of vertices S such that $G \setminus S$ is edgeless.
- Feed Back Vertex Set. Find a minimum set of vertices S such that $G \setminus S$ is a forest.

Questions. Which of these problems are easy? difficult? intractable?

2 - Graph Search

Generic Graph Search

- Searching a graph means: visit the vertices following the edges of the graph.
- It works the same for non-oriented and oriented graphs.
- Searching a graph often gives you some information on the structure of the graphs.
- A search also output a "search tree".
- A search also output an ordering on the vertices.

What do you mean by "generic" search?

Algorithm 1 GENERIC-SEARCH($G = (V, E)$, s)

1: Initially s is explored, the rest is unexplored
2: **while** Possible **do**
3: Choose an edge uv such that u is explored and v is unexplored
4: Mark v as explored

In this generic search, there is a lot of choices when it comes to decide the next edge to follow. There is several ways to break the ties, each giving different kind of informations on the graph.

Breadth First Search (BFS)

- The **distance** $dist(u, v)$ between two vertices u and v is the length of a shortest path linking u and v. It is ∞ if no such path exists[1].
- BFS starts at a vertex s, then explores all vertices at distance 1 from s, then all vertices at distance 2, then at distance 3 etc etc.
- For the implementation we'll use an adjacency matrix. $Adj[u]$ denotes the list of neighbors of u.

[1] it is the case if and only u and v are in distinct connected components

A few words about BFS

- INPUT: a graph $G = (V, E)$ and a source vertex s.
- GOAL: visit every vertex reachable from s.

Functionning:

- BFS visits all vertices at distance 1 from s, then vertices at distance 2, then at distance 3 etc etc
- The algorithm works on both directed and undirected graphs.

By-product:

- It computes the distances from s to all other vertices.
- It produces a BFS tree rooted at s that contains all reachable vertices:
  - For every u reachable from s, the unique su-path of the BFS tree corresponds to a shortest su-path of G.
  - The BFS-tree is a spanning tree of the connected component containing u (sometime denoted by C_u)

Pseudo-code for BFS: use a *queue*

- An unexplored vertex is white,
- An explored vertex is gray or black.
- It is black if all its neighbors have been explored.

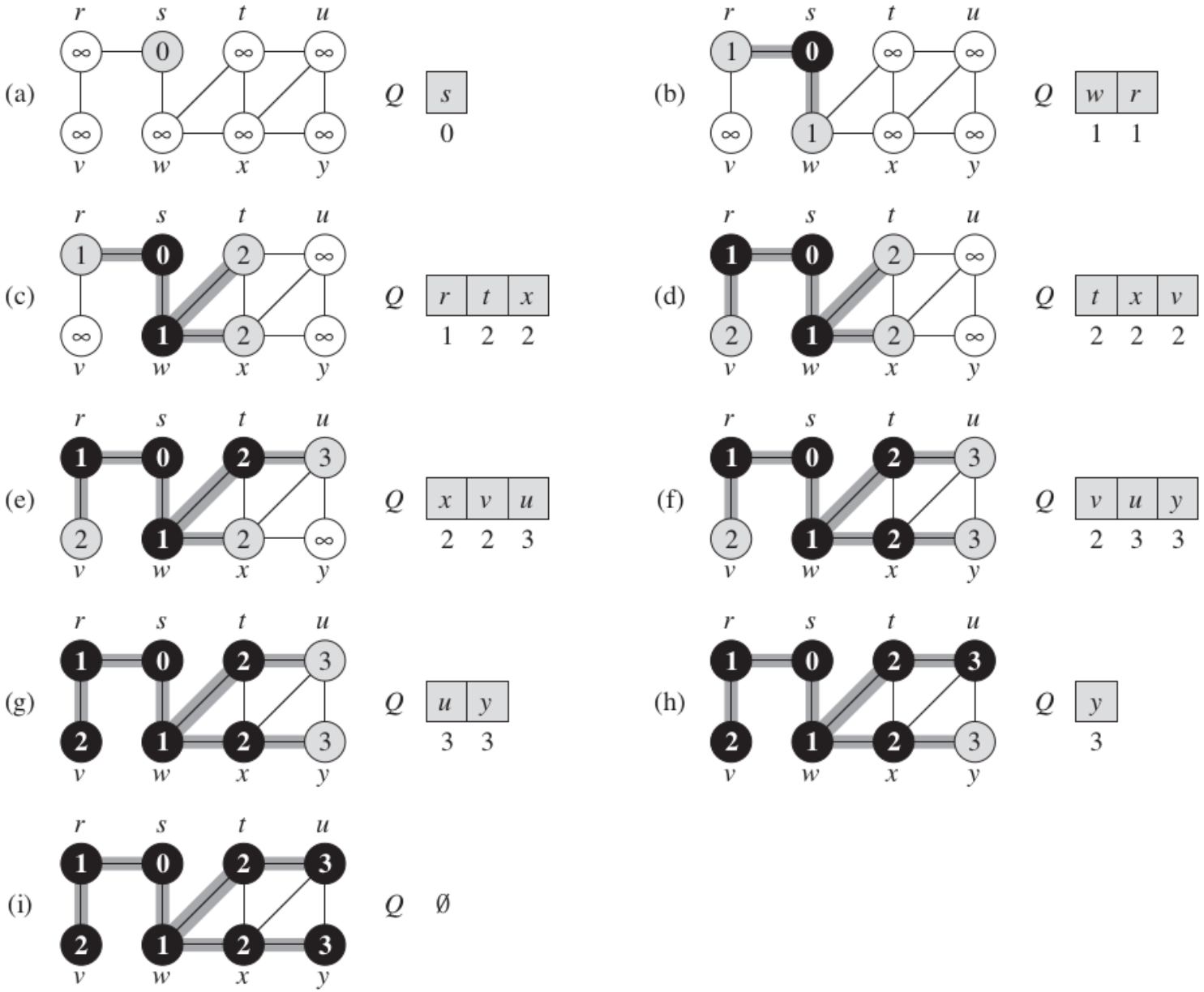
Pseudo-code for BFS: use a *queue*

- An unexplored vertex is white,
- An explored vertex is gray or black.
- It is black if all its neighbors have been explored.

Algorithm 3 BFS($G = (V, E)$, start vertex s)

```
1:  for each u ∈ V do
2:      u.color ← White                ▷ At the beginning all vertices are unexplored
3:  s.color ← Gray                     ▷ Except for s
4:  ENQUEUE(Q, s)                      ▷ Queue initialized with s
5:  while Q ≠ ∅ do
6:      u = DEQUEUE(Q)
7:      for each v ∈ Adj[u] do
8:          if v.color == White then
9:              v.color ← Gray
10:             ENQUEUE(Q, v)
11:     u.color ← Black
```

BFS Execution (pics from CLRS)



Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like *.color* in BFS.

- There is no one best way to implement vertex and edge attributes.

Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like *.color* in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like *.color* in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like *.color* in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

Two possible solutions:

1. Represent vertex attributes in an additional arrays, like: $color[1, \ldots, n]$.
   In this case, "$u.color$" would actually be stored in the array entry $color[u]$.

Attributes and their implementations

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. Like *.color* in BFS.

- There is no one best way to implement vertex and edge attributes.
- For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph.

Two possible solutions:

1. Represent vertex attributes in an additional arrays, like: $color[1, \ldots, n]$. In this case, "$u.color$" would actually be stored in the array entry $color[u]$.
2. In an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a Vertex class.

First observations on BFS

Algorithm 4 BFS(G, start vertex s)

```
1:  s.color ← Gray
2:  for each u ∈ V \ {s} do
3:      u.color ← White
4:  Q ← s                                   ▷ Queue initialized with s
5:  while Q ≠ ∅ do
6:      u =DEQUEUE(Q)
7:      for each v ∈ Adj[u] do
8:          if v.color == White then
9:              v.color ← Gray
10:             ENQUEUE(Q, v)
11:     u.color ← Black
```

Observations:

- Only unexplored vertex can enter the Queue
- All vertices in the Queue are gray.
- u is blackened $\Rightarrow$ all neighbors of u are explored (gray or black)

Coorectness and running time

Correctness: a vertex v is explored if and only if v is reachable from s.

Proof: it is a special case of the generic search.

Running time: $O(n_s + m_s)$ where n_s and m_s are respectively the number of vertices and the number of edges of the connected component containing s.

Reason: Each edge is looked twice ($\Sigma_{v \in V} d(v) = 2m$).

BFS Application: shortest paths

Goal: compute $dist(s, v)$ for every vertex v.

- $u.dist$ is the distance from s to u.

Algorithm 5 BFS(G, start vertex s)

```
1:  s.color ← Gray, s.dist ← 0
2:  for each u ∈ V \ {s} do
3:      u.color ← White, u.dist = ∞
4:  Q ← s
5:  while Q ≠ ∅ do
6:      u =DEQUEUE(Q)
7:      for each v ∈ Adj[u]  do
8:          if v.color == White then
9:              v.color ← Gray
10:             v.dist ← u.dist + 1
11:             ENQUEUE(Q, v)
12:     u.color ← Black
```

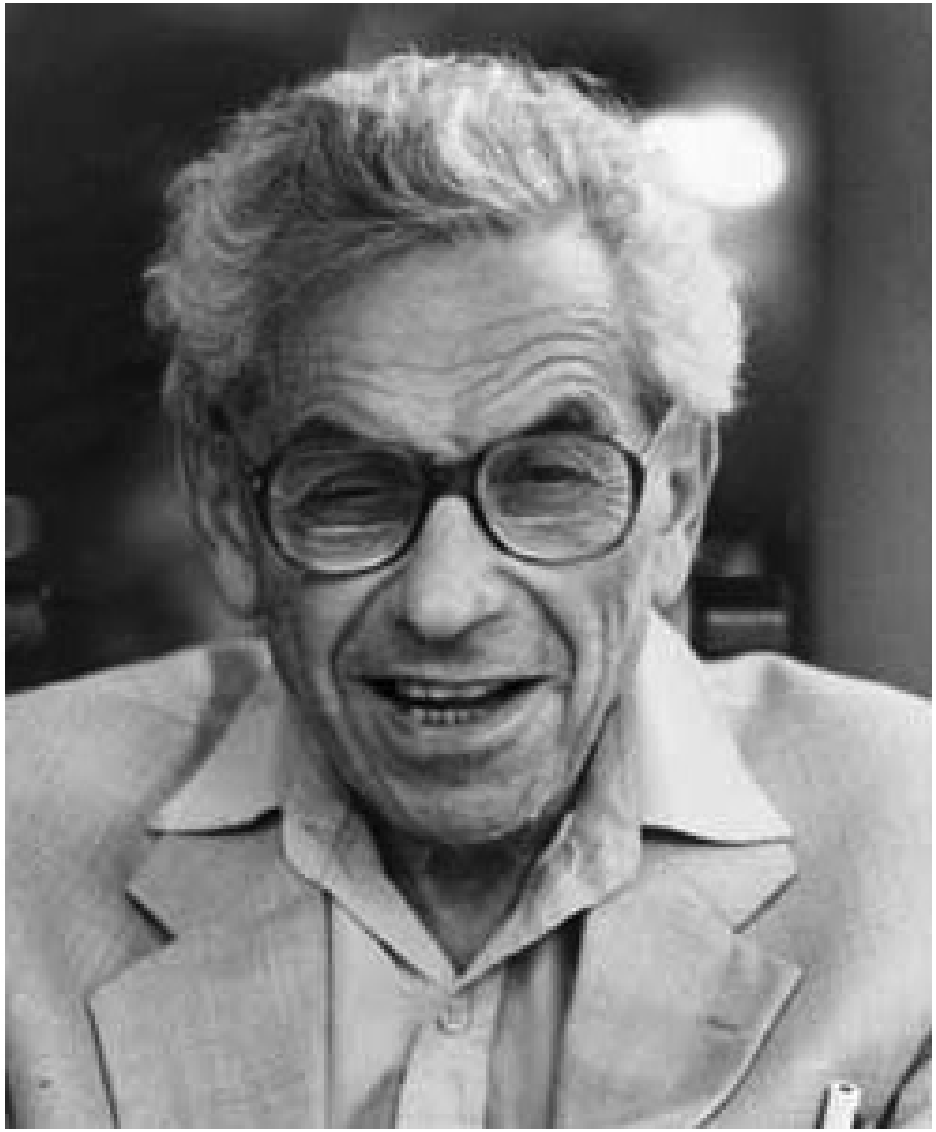
BFS Application: connected components

Def: two vertices u and v are connected if there is a path linking them.

Goal: Preprocess graph to answer queries of the form: "is v connected to w?" in constant time.

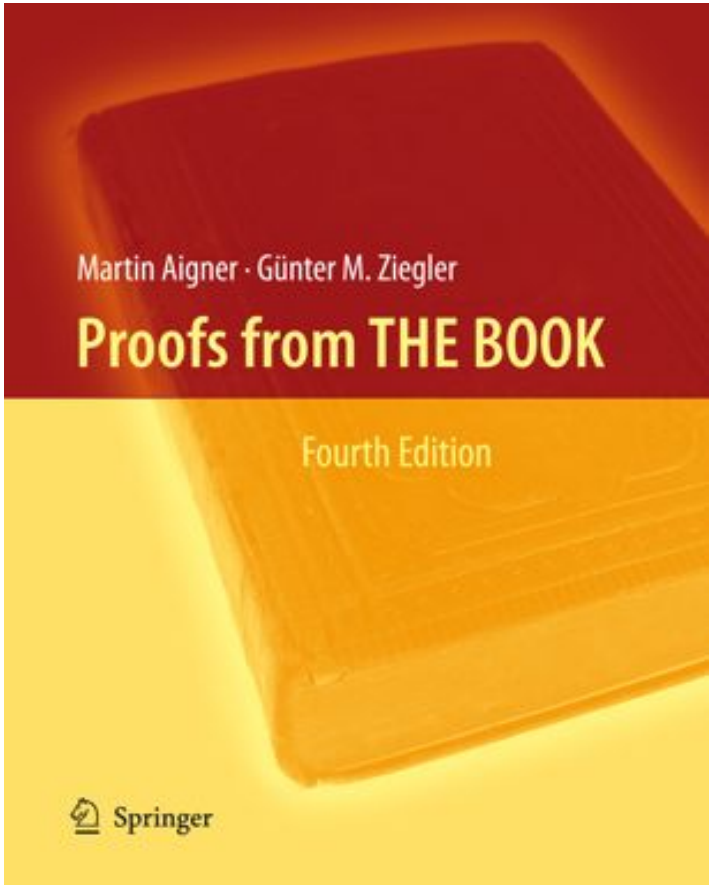
Solution: use BFS (or any search) as a subrootine.

Paul Erdős (1913-1996)



Paul Erdős and Terry Tao

Proofs from the book



"You don't have to believe in God, but you should believe in The Book."

Paul Erdős

BFS Application 2: Erdős Number

Paul Erdős wrote more than 1500 articles.
He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don't have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc

BFS Application 2: Erdős Number

Paul Erdős wrote more than 1500 articles.
He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don't have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc

Modelisation: A graph $G = (V, E)$ where:

- V is the set of science researchers.
- Two researchers are adjacent if they wrote a paper together.
- Erdős number is the distance from Erdős.

BFS Application 2: Erdős Number

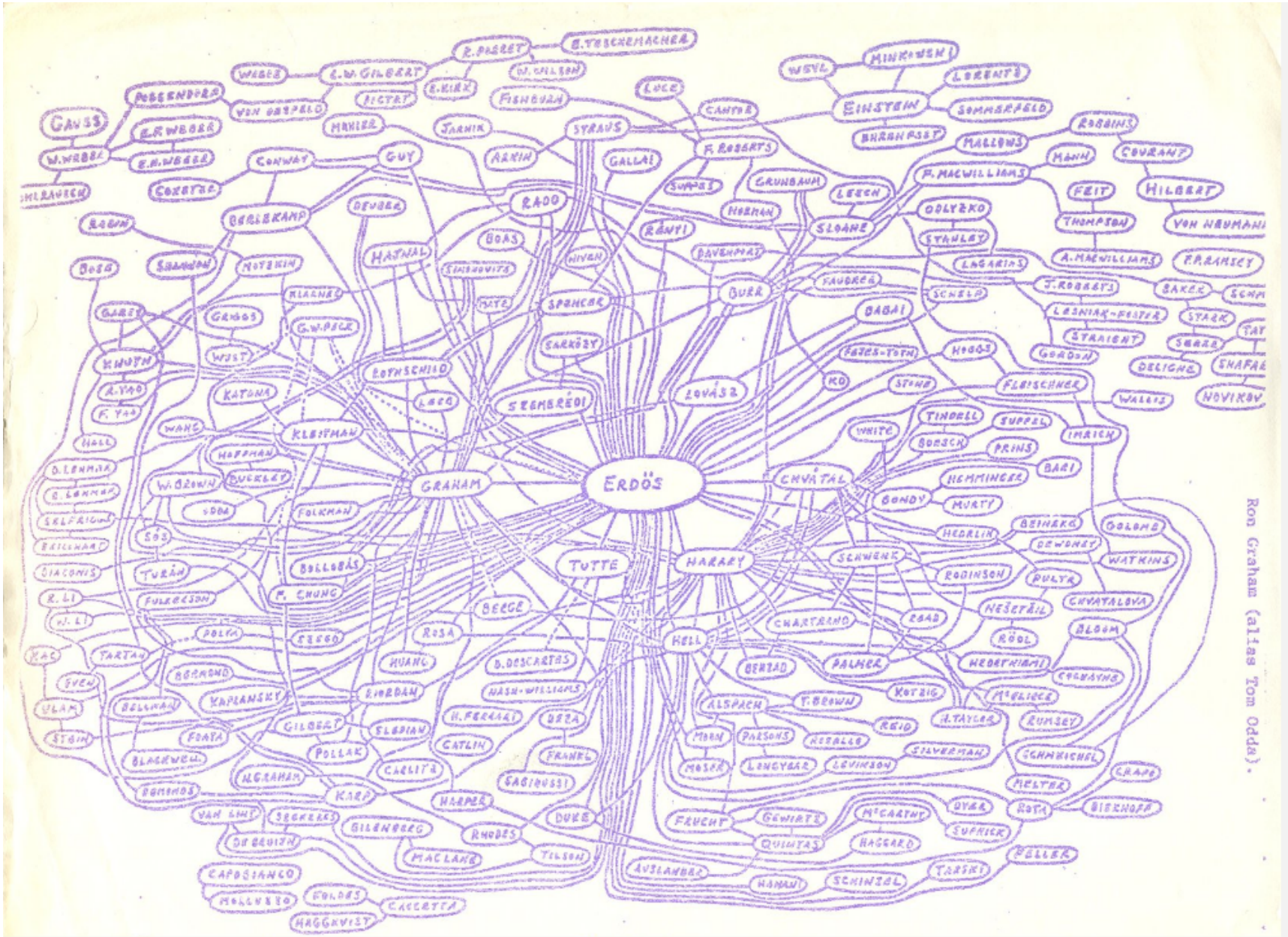
Paul Erdős wrote more than 1500 articles.
He had many collaborators which drived people to define the Erdős number:

- The Erdős number of Erdős is 0.
- If you have a paper with Erdős you have Erdős number 1.
- If you don't have a paper with Erdős but you have a paper with someone that has a paper with Erdős, then you have Erdős number 2.
- etc etc

Modelisation: A graph $G = (V, E)$ where:

- V is the set of science researchers.
- Two researchers are adjacent if they wrote a paper together.
- Erdős number is the distance from Erdős.

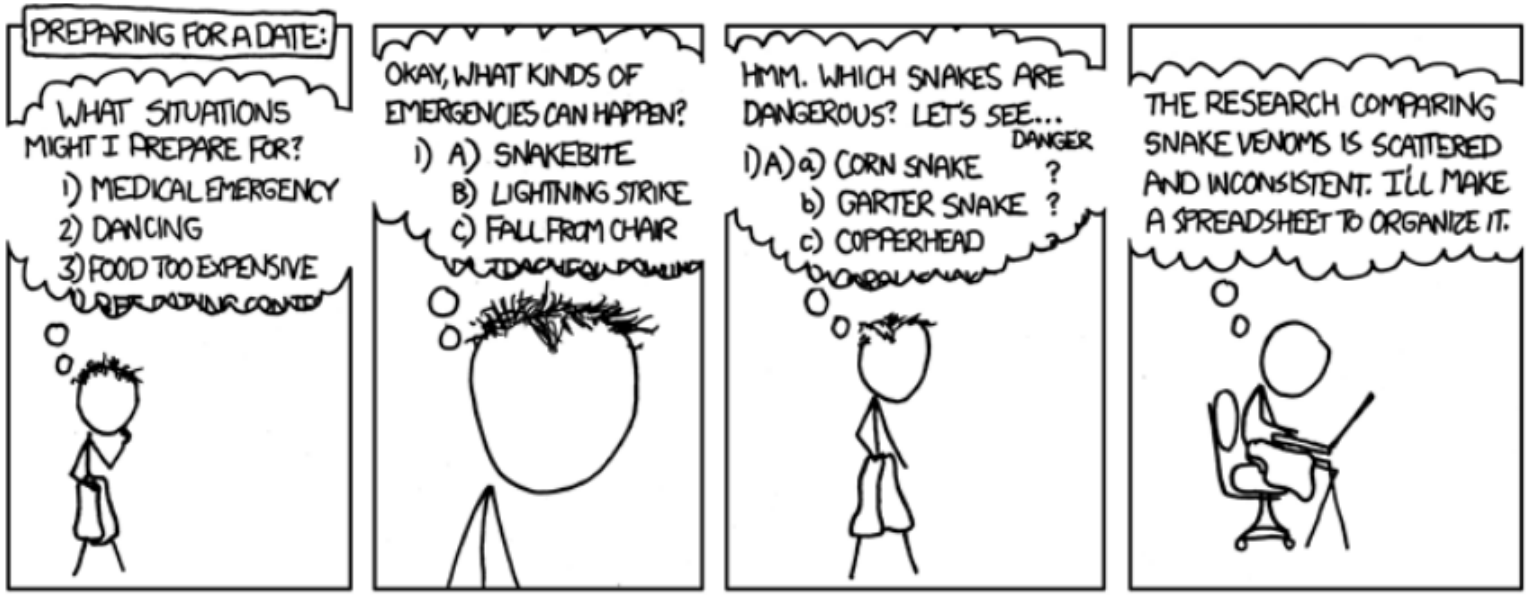
Solution: run *BFS* with source vertex *Erdős*



hand-drawing of part of the Erdös graph by Ron Graham

Depth First Search (DFS)

- It is a special kind of search, so it takes a graph G and a source vertex s as input, and visits all vertices reachable from s.
- DFS is an aggressive search, it backtracks only when no other choice is available.
- It has several applications on directed graphs:
  - Computes topological ordering of Directed Acyclic Graph (DAG)
  - Computes strong connected components of directed graphs
- It does all that in linear time.



xkcd

http://xkcd.com/761/

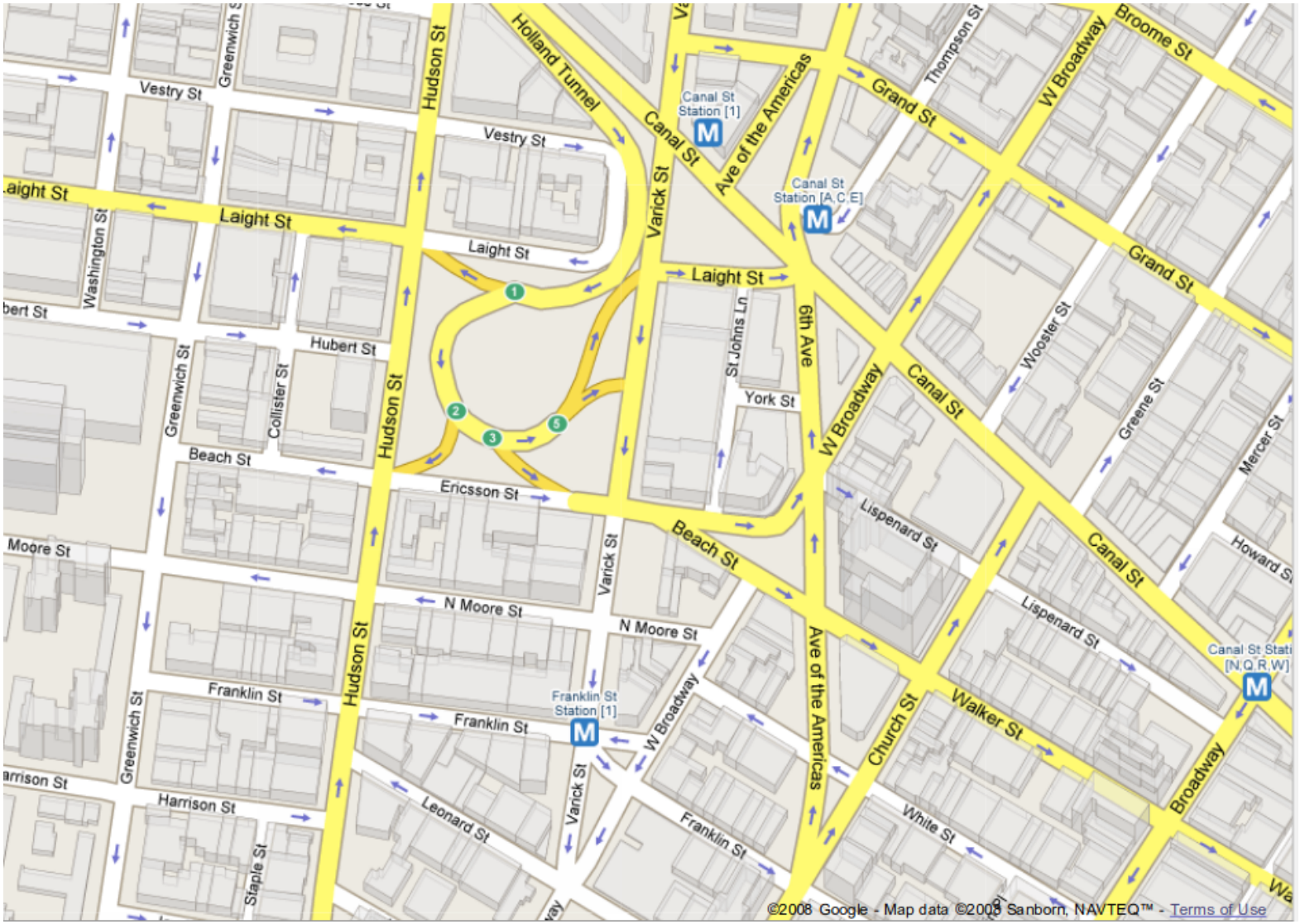
Definitions for directed graphs

- Digraph: like a graph but edges are directed and called arcs.
- A vertex v has two kind of neighbors:
  - Out-neighborhood: $N^+(v)$ out-degree: $d^+(v)$,
  - In-neighborhood: $N^-(v)$ in-degree: $d^-(v)$.
- directed path: path with all arcs in the same direction.
- directed cycle: cycle with all arcs in the same direction.
- u is reachable from v if there is a *directed* path from u to v.

Property: $$\sum_{u \in V} d^+(u) = \sum_{u \in V} d^-(u) = |E|$$

Example of modelisation: road network

Vertices are the intersections, arcs are the one-way street.



Digraphs applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Picture from Kevin Wayne

Some digraphs problems

- Path. Is there a directed path from s to t?
- Induced path. Is there a directed induced path between s and t?
- Shortest path. What is the shortest directed path from s to t?
- Topological sort. Can you order the vertices so that all arcs point right?
- Strong connectivity. Is there a directed path between all pairs of vertices?
- Transitive closure. For every pair of vertices u, v, add the arc from u to v if there is a directed path from u to v.
- PageRank. What is the importance of a web page?

Digraph representation: adjacency list

Maintain vertex-indexed array of lists containing the out-neighbors of each vertex.

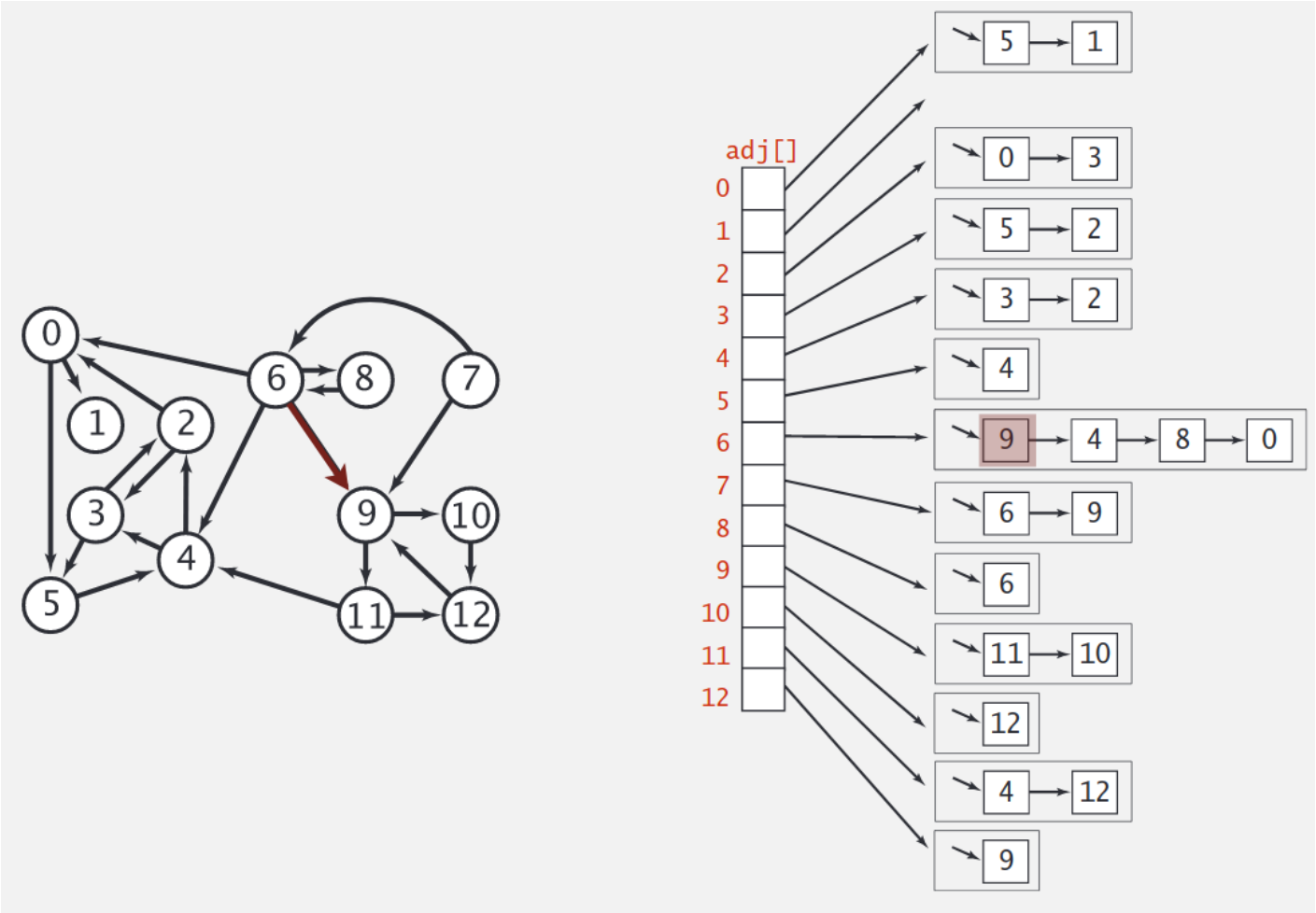


figure by Kevin Wayne

Compare digraph representations

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 †	1	V
adjacency lists	E + V	1	outdegree(v)	outdegree(v)

† disallows parallel edges

Picture from Kevin Wayne

A first Pseudo-code for DFS

DFS is a recursive algorithm.
We use adjacency list, ADJ[u] is the set of out-neighbours of u.

Algorithm 6 DFS(G, start vertex s)

```
1: Mark s as discovered
2: for each v ∈ Adj[u] do                    ▷ For each arc uv
3:     if v is unexplored then
4:         DFS(G, v)
```

We want more

As for BFS, we color the vertices:

- *White* if it is unexplored
- *Gray* if it is discovered: explored but some of its (out)-neighbors are still unexplored
- *Black* if it is finished: all its (out)-neighbors has been discovered.

We want more

As for BFS, we color the vertices:

- *White* if it is unexplored
- *Gray* if it is discovered: explored but some of its (out)-neighbors are still unexplored
- *Black* if it is finished: all its (out)-neighbors has been discovered.

We will also compute:

- $v.d$: time when v has been discovered (grayed), and
- $v.f$: time when v is finished (blackened).
- For every vertex v, we have $1 \leq v.d \leq v.f \leq 2n$

We want more

As for BFS, we color the vertices:

- *White* if it is unexplored
- *Gray* if it is discovered: explored but some of its (out)-neighbors are still unexplored
- *Black* if it is finished: all its (out)-neighbors has been discovered.

We will also compute:

- $v.d$: time when v has been discovered (grayed), and
- $v.f$: time when v is finished (blackened).
- For every vertex v, we have $1 \leq v.d \leq v.f \leq 2n$

As well as a DFS forest:

- As in BFS, whenever DFS discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi \leftarrow u$
- $G_\pi = (V, E_\pi)$ where $E_\pi = \{uu.\pi : u \in V,\ u.\pi \neq Nil\}$

Recursive pseudo-code for DFS

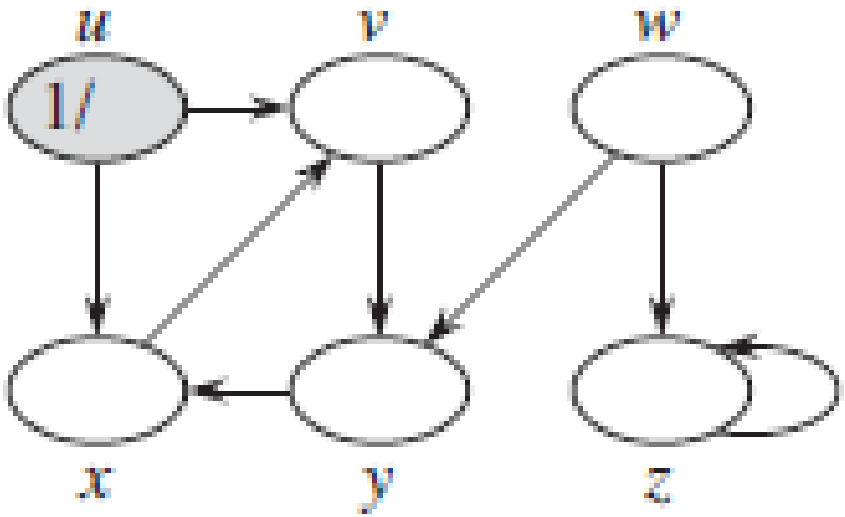
Algorithm 7 DFS(G)

```
1: for each u ∈ V do
2:     u.color ← White, and u.pi ← NIL                    ▷ Initialization
3: time ← 0                          ▷ time is a global variable used for timestamping
4: for Each vertex v ∈ V do
5:     if v.color == White then
6:         DFS-Visit(G, v)
```

Algorithm 8 DFS-Visit(G, v)

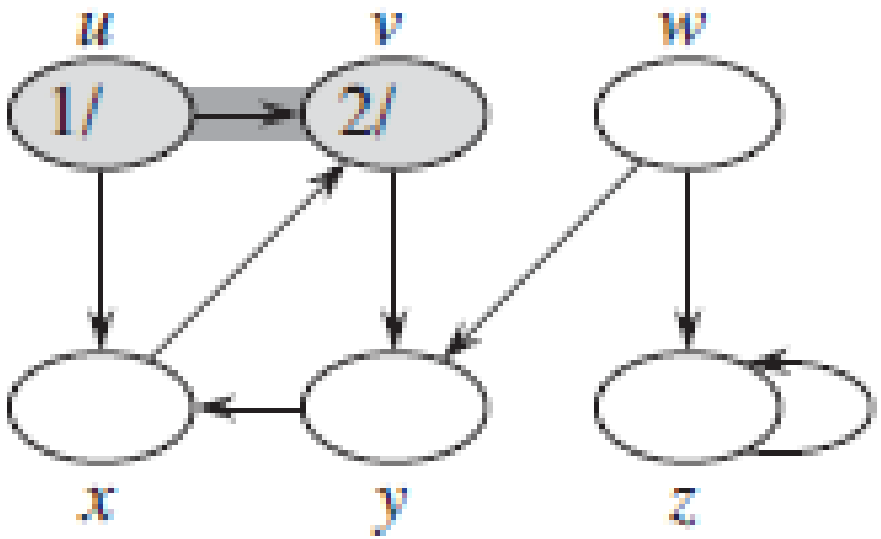
```
1: time ← time + 1
2: v.color ← Gray and v.d ← time
3: for each u ∈ Adj[v] do                                  ▷ For each arc uv
4:     if u.color == White then          ▷ if v has not been discovered yet
5:         u.π ← u
6:         DFS-Visit(G, u)
7: time ← time + 1
8: u.color ← Black and u.f ← time
```

DFS Execution



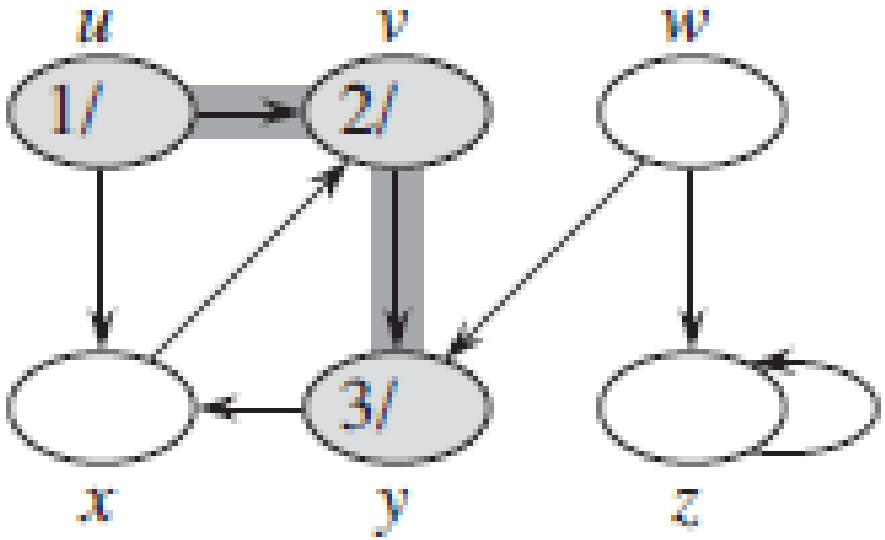
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



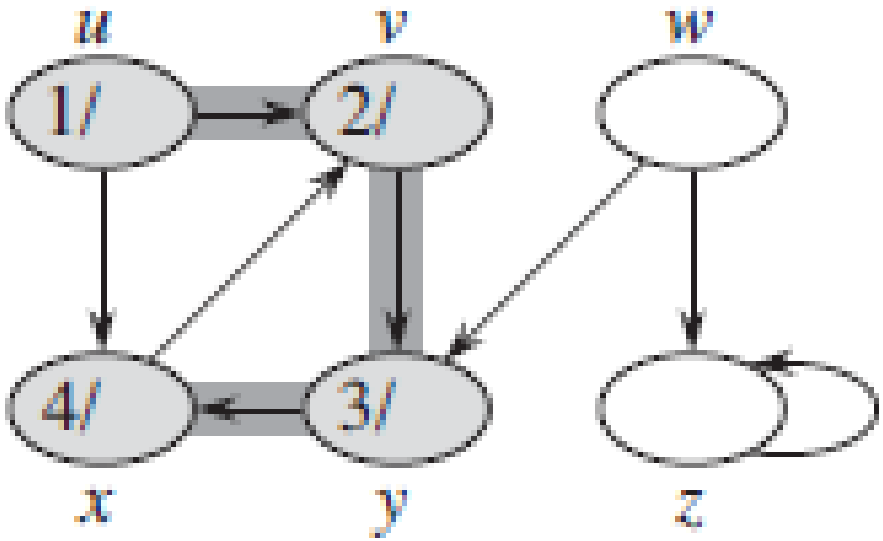
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



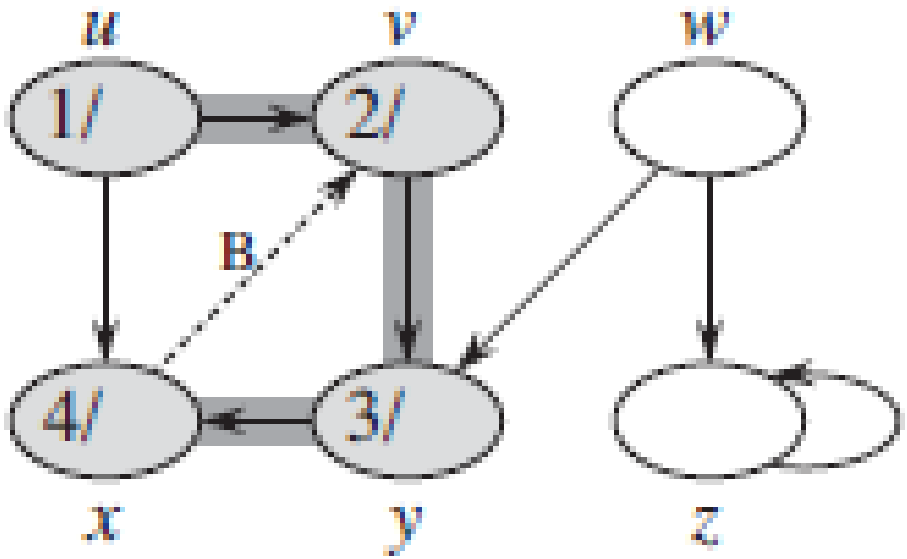
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



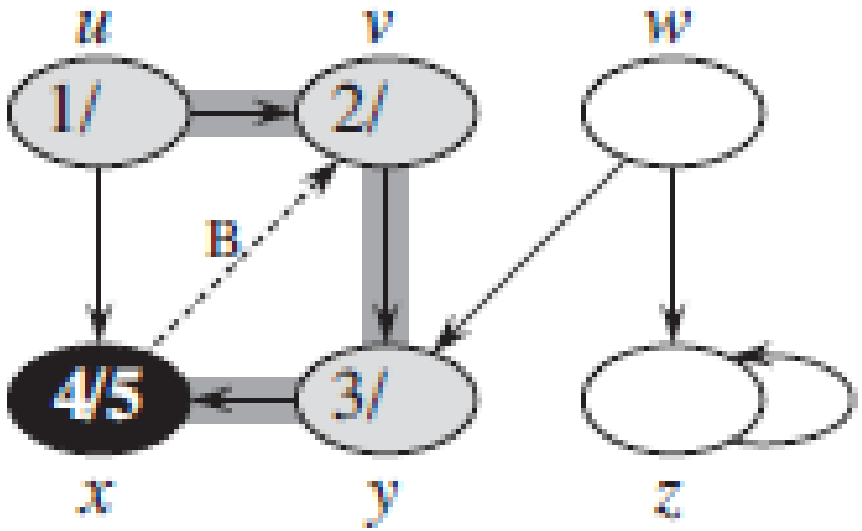
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



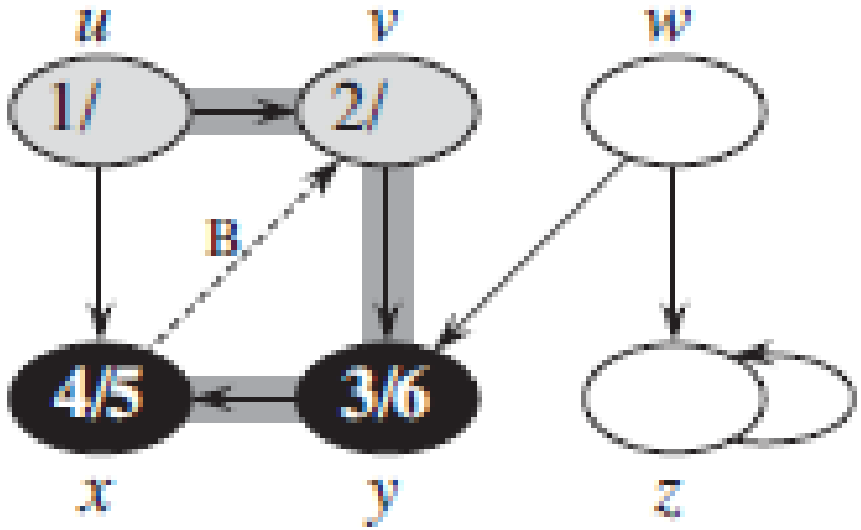
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



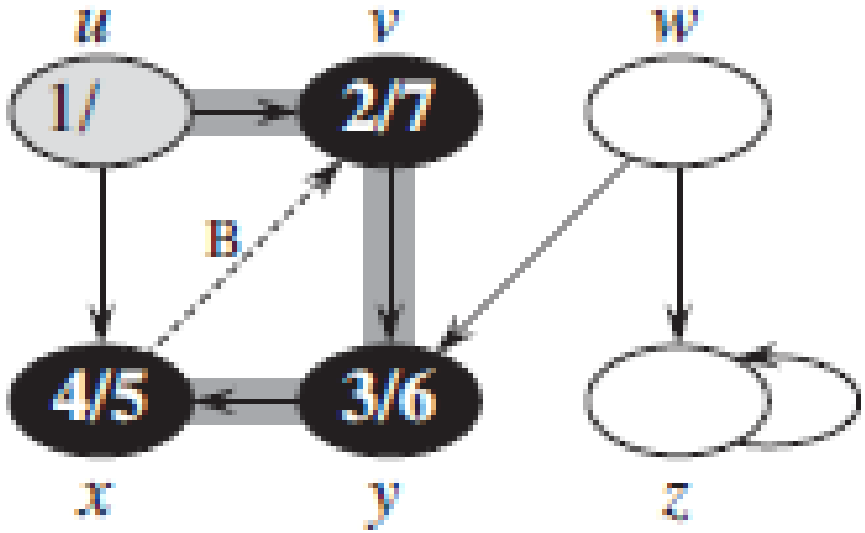
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



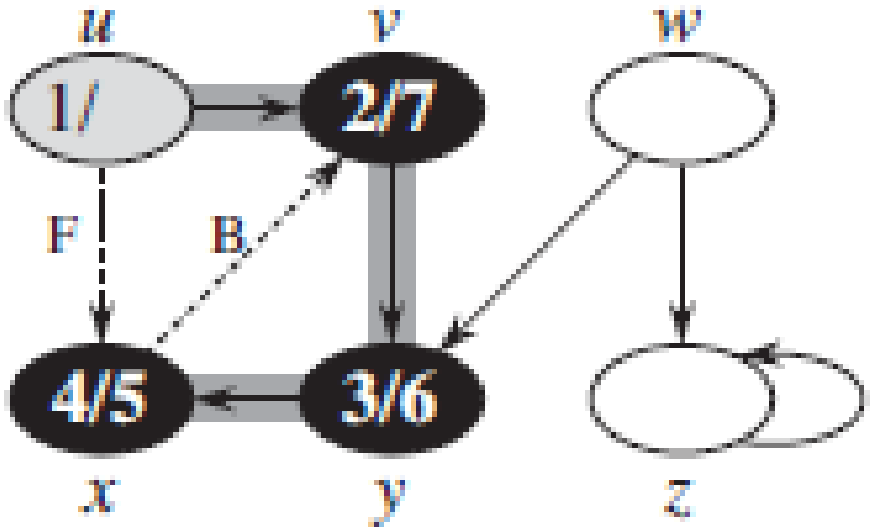
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



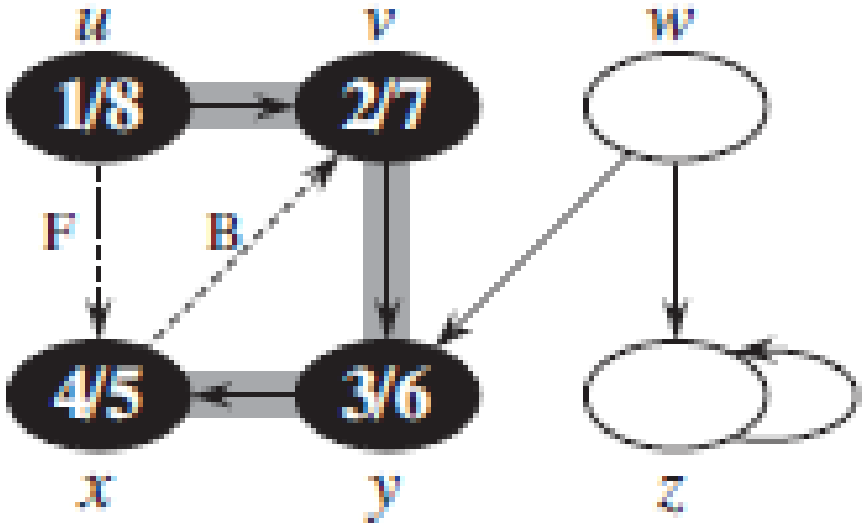
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



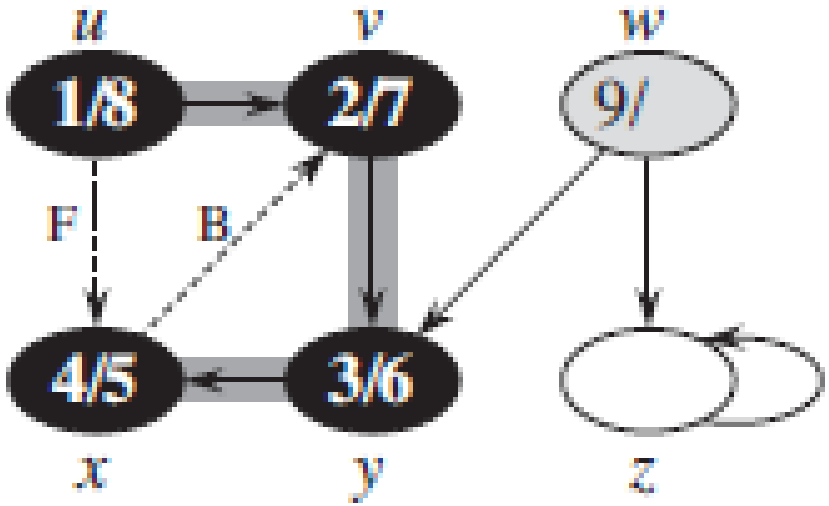
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



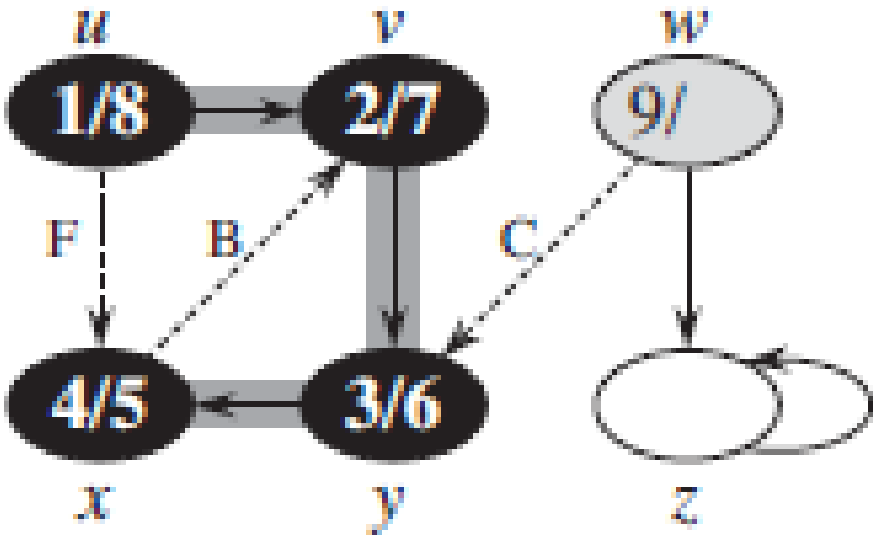
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



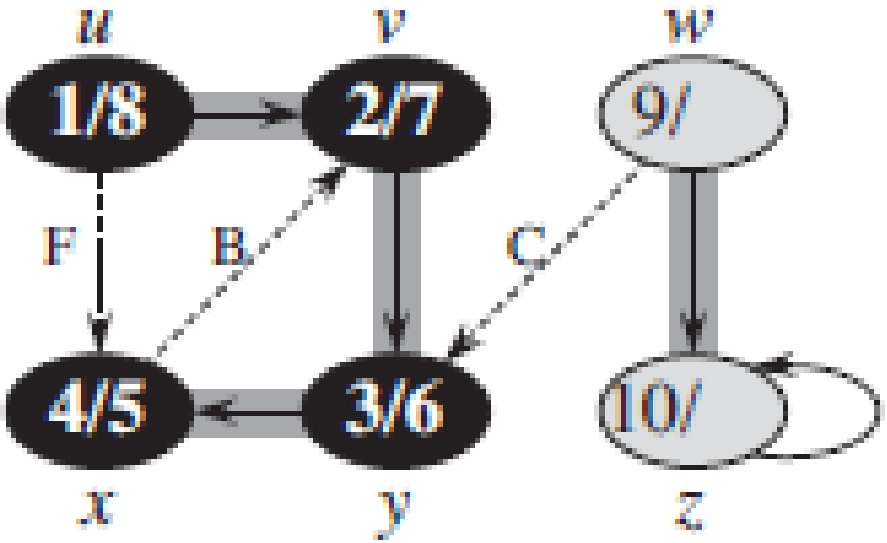
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



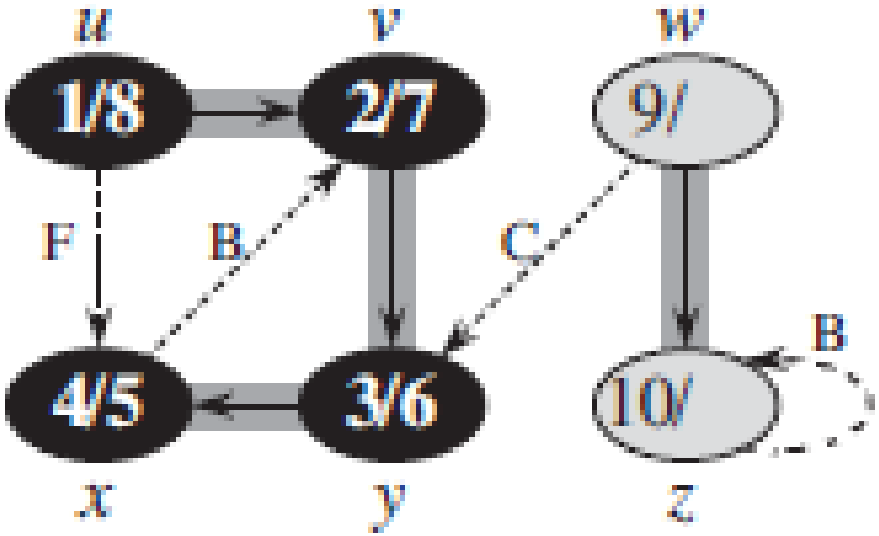
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



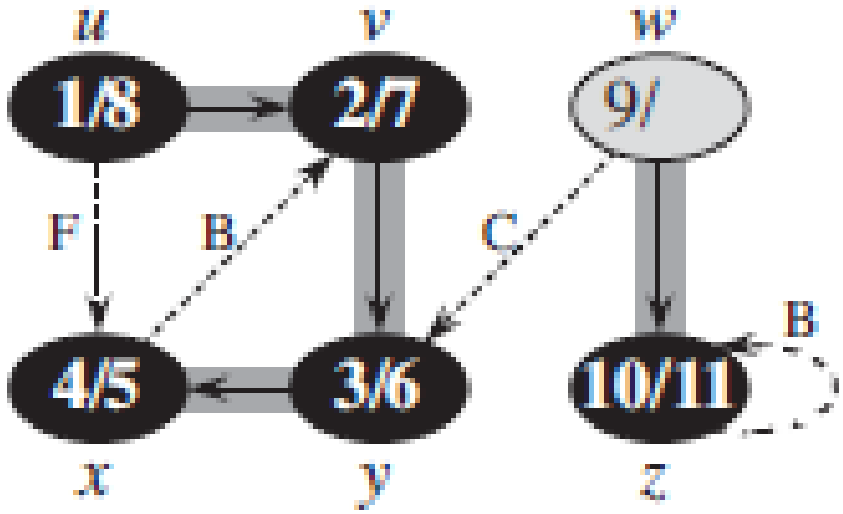
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



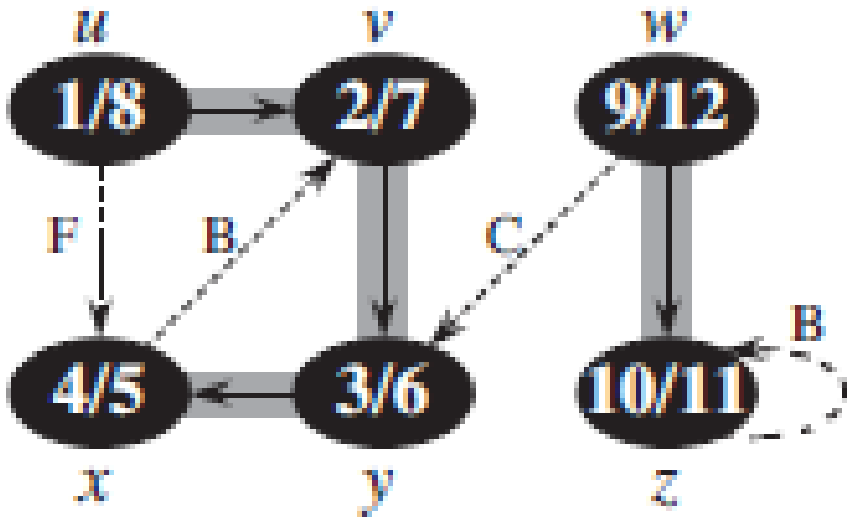
Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution



Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

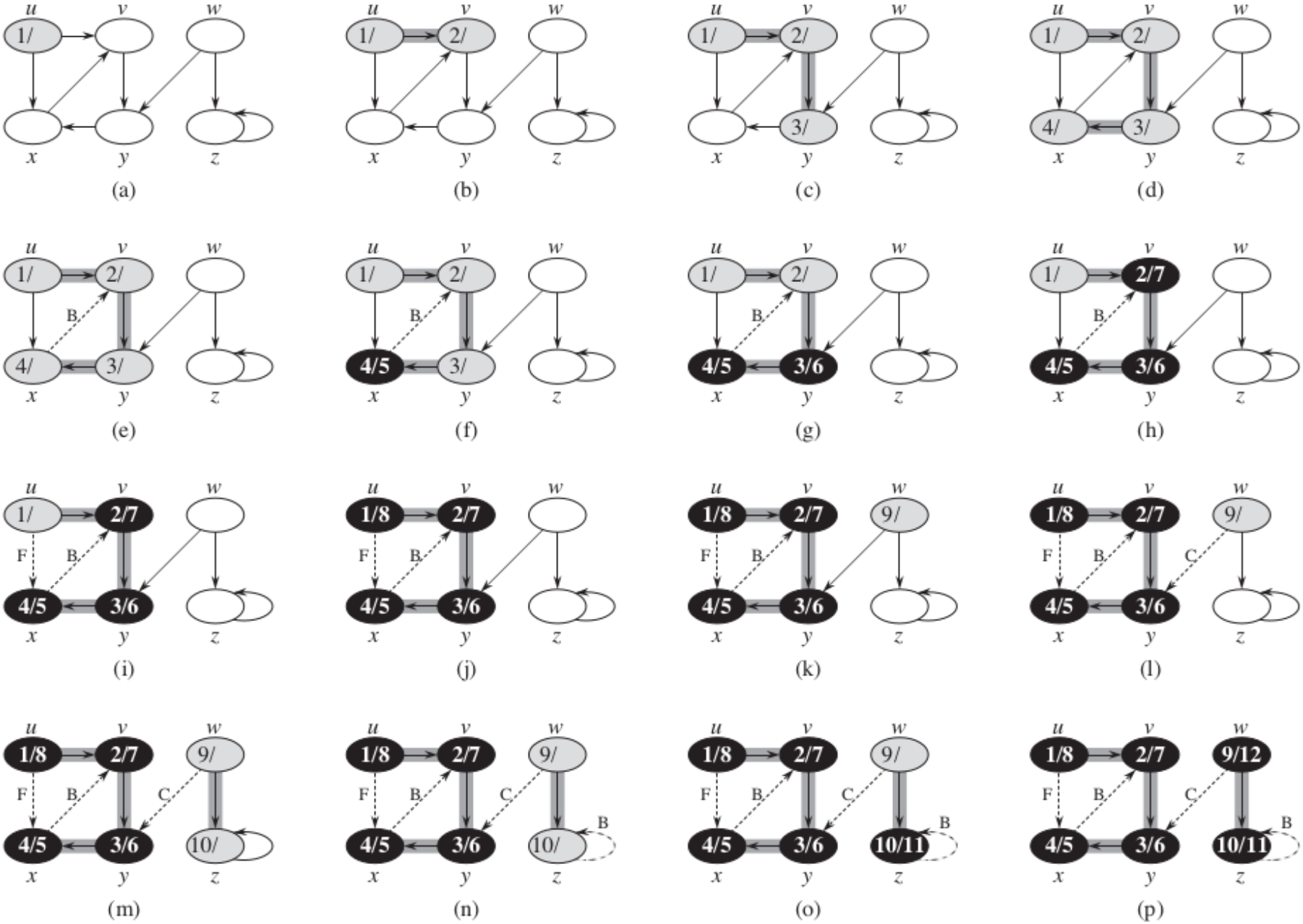
DFS Execution



Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

DFS Execution

Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein



Running time

Algorithm 9 DFS(G)

```
1: for each u ∈ V do
2:     u.color ← White, and u.pi ← NIL
3: time ← 0                          ▷ time is a global variable used for timestamping
4: for each vertex v ∈ V do
5:     if v.color == White then
6:         DFS-Visit(G, v)
```

Algorithm 10 DFS-Visit(G, v)

```
1: time ← time + 1
2: u.d ← time and u.color ← Gray
3: for each v ∈ Adj[u] do                          ▷ For each arc uv
4:     if v.color == White then                    ▷ if v has not been discovered yet
5:         v.π ← u
6:         DFS-Visit(G, v)
7: time ← time + 1
8: u.color ← Black and u.f ← time
```

Running time $O(n+m)$

First properties of DFS

- For all vertices $1 \leq u.d < u.f \leq 2n$.

- At time t, the color of a vertex u is:
  - white if $t < u.d$,
  - gray if $u.d \leq t < u.f$,
  - black if $u.f \leq t$

- G_π is a forest ($u = v.\pi$ if and only if $\mathrm{DFS\text{-}VISIT}(G, v)$ was called during a search of u's adjacency list).

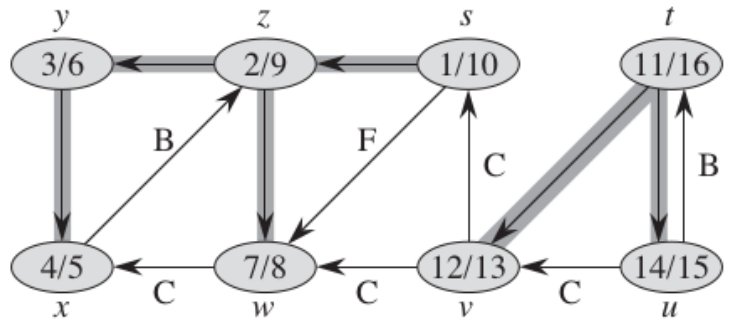
- A vertex v is a descendant of a vertex u in the DFS forest if and only if v is discovered during the time u is gray (i.e. $u.d \leq v.d \leq u.f$).

Theorem (Parenthesis theorem)

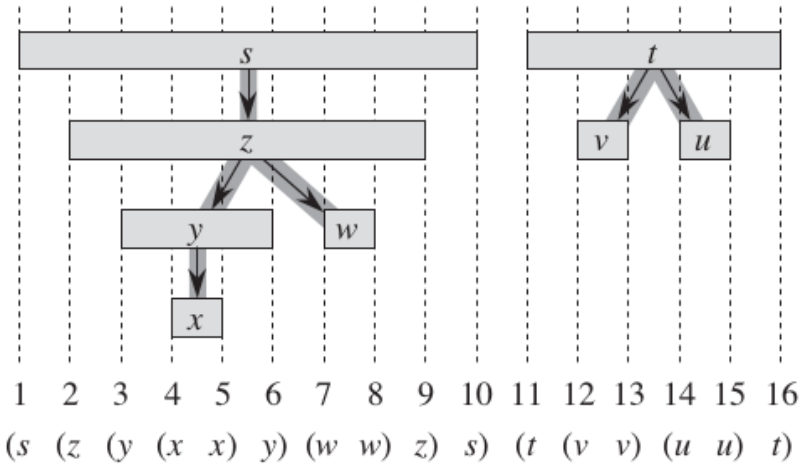
In any DFS of a digraph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint,
2. the interval $[u.d, u.f]$ is contained in $[v.d, v.f]$ (i.e. $v.d < u.d < u.f < v.f$).
3. the interval $[v.d, v.f]$ is contained in $[u.d, u.f]$ (i.e. $u.d < v.d < v.f < u.f$).

(a)



(b)



Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Theorem (Parenthesis theorem)
In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint,
   AND u is not a decendant nor an ancestor of v.
2. the interval $[u.d, u.f]$ is contained in $[v.d, v.f]$ (i.e. $v.d < u.d < u.f < v.f$).
   AND u is a descendant of v.
3. the interval $[v.d, v.f]$ is contained in $[u.d, u.f]$ (i.e. $u.d < v.d < v.f < u.f$).
   AND v is a descendant of u.

Proof We begin with the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$ or not. The first subcase occurs when $v.d < u.f$, so v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.d$, and by inequality (22.2), $u.d < u.f < v.d < v.f$; thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $v.d < u.d$ is similar, with the roles of u and v reversed in the above argument. ■

Descendant in the DFS forest

Corollary: A vertex v is a descendant of a vertex u in the DFS forest **if and only if** $u.d < v.d < v.f < u.f$

Theorem (**White-path theorem**)
In a DFS forest of a digraph $G = (V, E)$, a vertex v is a descendant of a vertex u **if and only if** at the time $u.d$, there is a (u, v)-path made of white vertices.

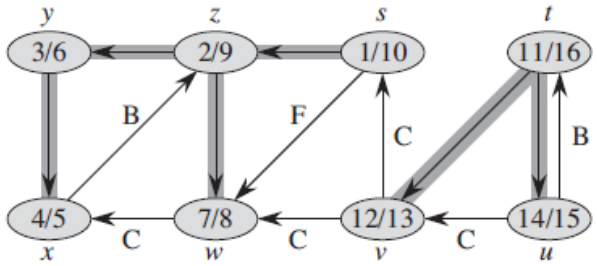
Classification of the arcs

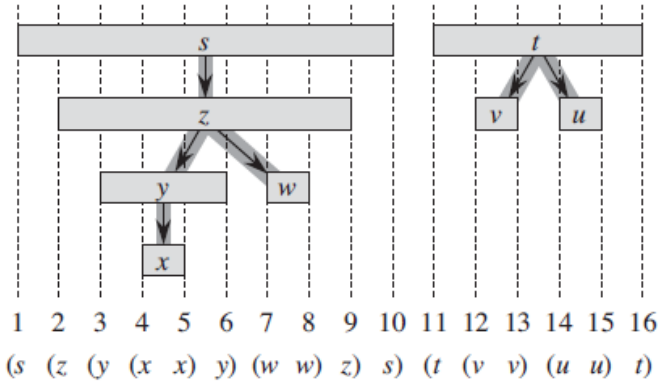
We can define four arc types in terms of the DFS forest G_π produced by a DFS:

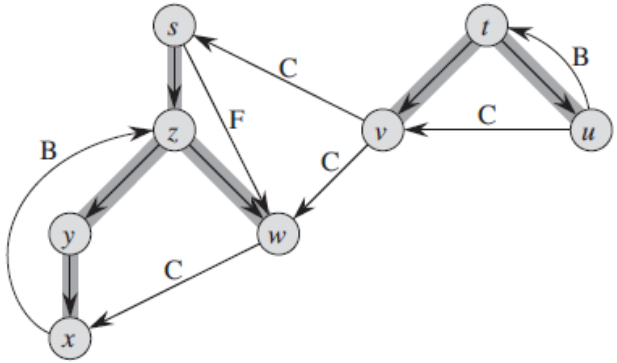
1. Tree arcs are arcs of G_π.
2. Back arcs are arcs uv such that u is a descendant of v.
3. Forward arcs are arcs uv such that u is an ancestor of u.
4. Cross arc are arcs uv such that u is not an ancestor of v and v is not an ancestor of u (i.e. no directed path linking u and v in the DFS tree).

A lot of information are contained in this, for example, a digraph has a directed cycle if and only (any) DFS produces a *back arc*.

Pics from CLRS





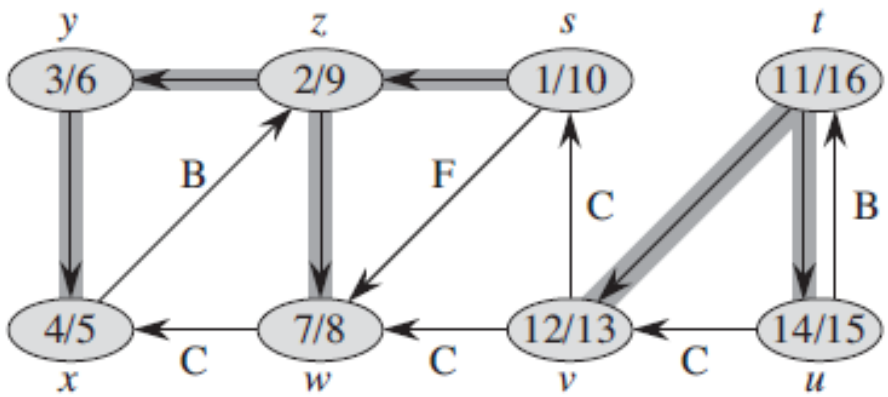


Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Dates and type of arcs

Let ux be an arc.

- if $u.d < x.d < x.f < u.f$, then ux is .
- if $x.d < u.d < u.f < x.f$, then ux is .
- if $x.d < x.f < u.d < u.f$, then ux is .

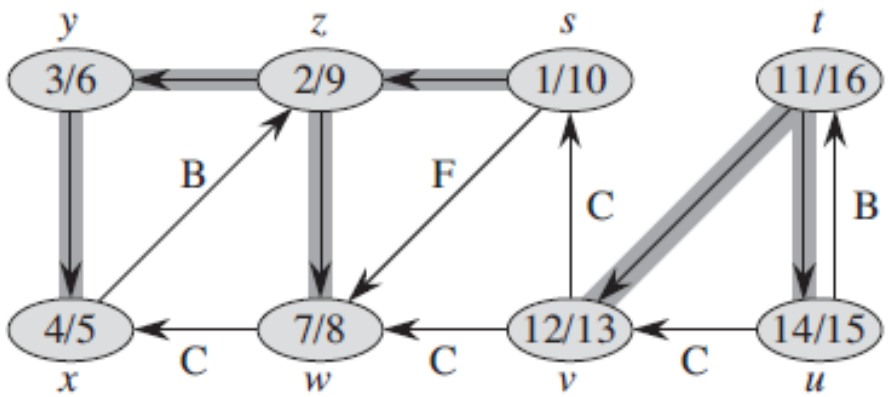


Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Dates and type of arcs

Let ux be an arc.

- if $u.d < x.d < x.f < u.f$, then ux is a tree arc or a forward arc.
- if $x.d < u.d < u.f < x.f$, then ux is .
- if $x.d < x.f < u.d < u.f$, then ux is .

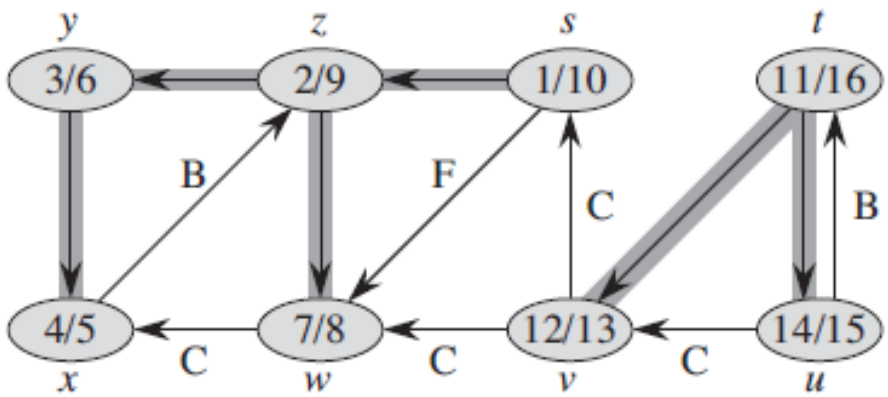


Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Dates and type of arcs

Let ux be an arc.

- if $u.d < x.d < x.f < u.f$, then ux is a tree arc or a forward arc.
- if $x.d < u.d < u.f < x.f$, then ux is a back arc.
- if $x.d < x.f < u.d < u.f$, then ux is .

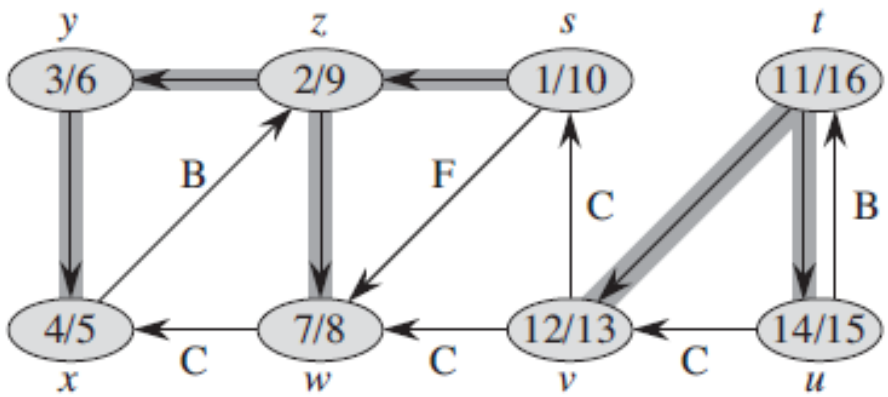


Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Dates and type of arcs

Let ux be an arc.

- if $u.d < x.d < x.f < u.f$, then ux is a tree arc or a forward arc.
- if $x.d < u.d < u.f < x.f$, then ux is a back arc.
- if $x.d < x.f < u.d < u.f$, then ux is a cross arc.



Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Compute the type of the arcs

We can compute the type of each arc when running DFS, with no extra cost.

Assume DFS is running and the arc uv is being scanned. Then the color of v gives us some indication on the type of uv[2]:

- If v is white, then uv is a tree arc (easy to see).
- If v is gray, then uv is a back arc.
- If v is black then
  - uv is a forward arc if $u.d < v.d$
  - uv is a cross arc if $u.d > v.d$.

To see the second point, observe that at any moment, a set of gray vertices induces a path of the DFS forest (a path of *tree arcs*).

To see the third points, observe that when you're blacken, all your descendants and ancestors have been discovered already.

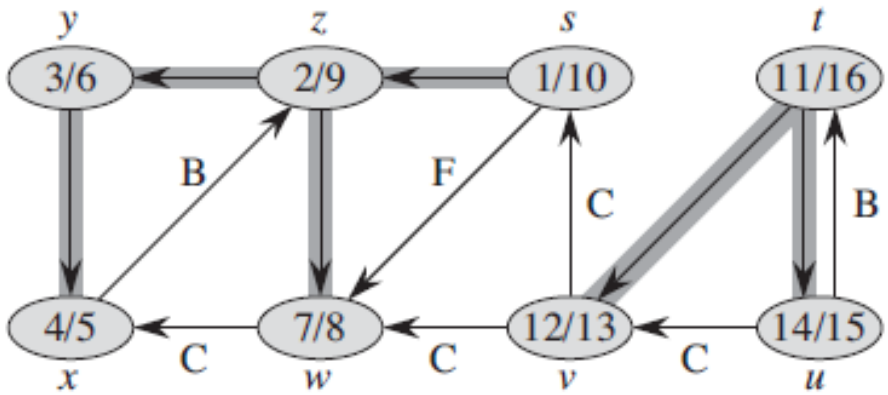
[2] Note that u is gray at this moment

Compute the type of the arcs

We can compute the type of each arc when running DFS, with no extra cost.

Assume DFS is running and the arc uv is being scanned. Then the color of v gives us some indication on the type of uv[2]:

- If v is white, then uv is a tree arc (easy to see).
- If v is gray, then uv is a back arc.
- If v is black then
  - uv is a forward arc if $u.d < v.d$
  - uv is a cross arc if $u.d > v.d$.



Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

[2]Note that u is gray at this moment

First Application of DFS: topological sort

Precedence scheduling

Goal: Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

Digraph model: vertex = tasks, arc = precedence constraint.

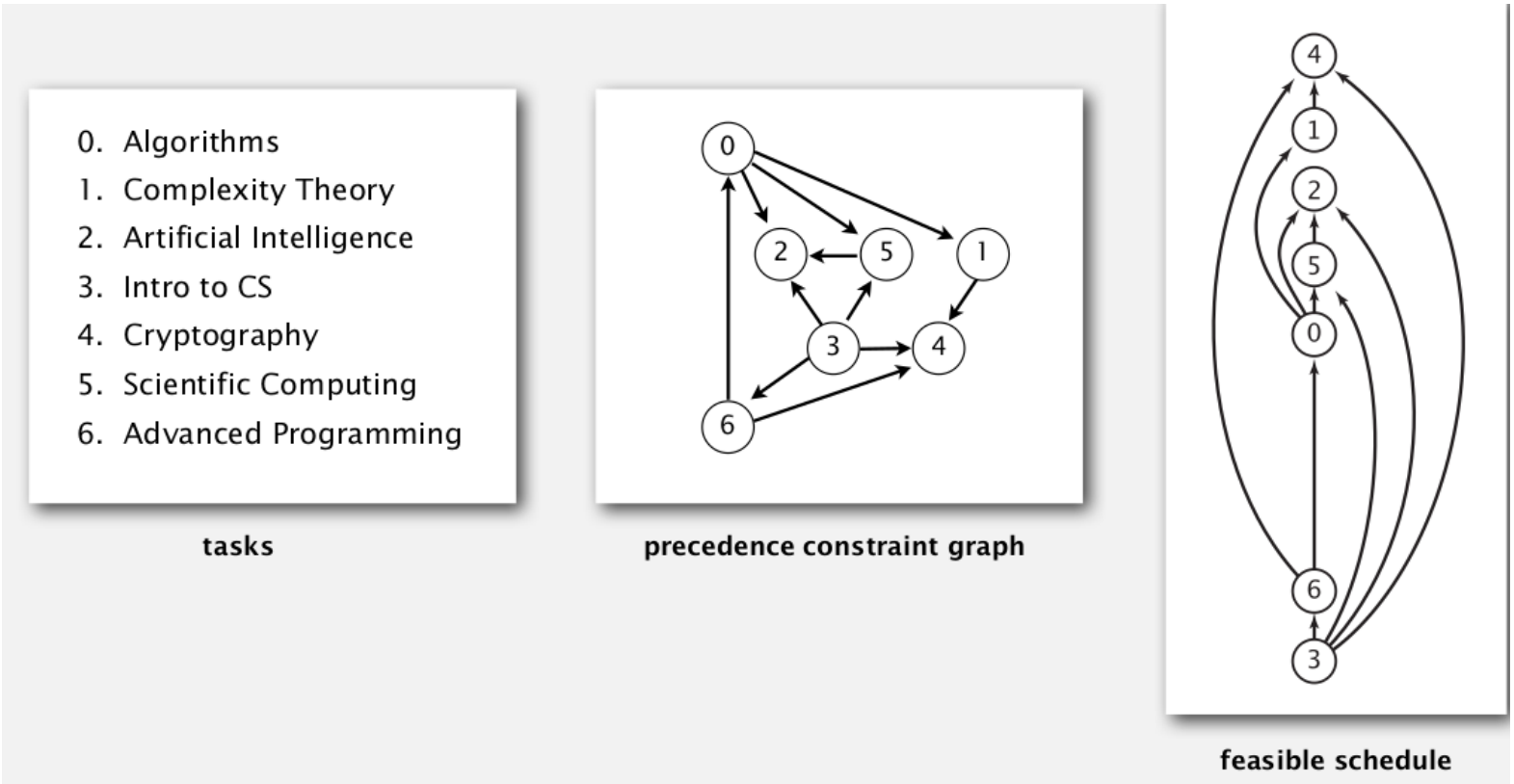


Figure by Kevin Wayne

Definition of a topological ordering

Definition A topological ordering of the vertices of a digraph G is a labeling $f : V(G) \to \{1, \ldots, n\}$ such that: $uv \in E$ only if $f(u) < f(v)$.

Observations:

- It is an ordering of the vertices along a horizontal line so that all directed arcs go from left to right.
- It is not unique (see example on the board)

Motivation: schedule a bunch of tasks when there is precedence constraints among the tasks.

Sinks and DAGs

Sink = vertex of outdegree 0.

DAG = Directed Acyclic Graph.

Sinks and DAGs

Sink = vertex of outdegree 0. DAG = Directed Acyclic Graph.

Observations:

- The last vertex of a topological ordering is a sink.
- Every DAG has a sink (**Idea of the proof:** the last vertex of a maximal directed path is a sink.)
- G admits a topological ordering if and only if G is a DAG.

Sinks and DAGs

Sink = vertex of outdegree 0. DAG = Directed Acyclic Graph.

Observations:

- The last vertex of a topological ordering is a sink.
- Every DAG has a sink (**Idea of the proof:** the last vertex of a maximal directed path is a sink.)
- G admits a topological ordering if and only if G is a DAG.

Algorithm 13 STRAIGHTFORWARD-TOPOLOGICAL-SORT(G)

Find a sink vertex v
$f(v) \leftarrow n$
STRAIGHTFORWARD-TOPOLOGICAL-ORDERING$(G \setminus \{v\})$

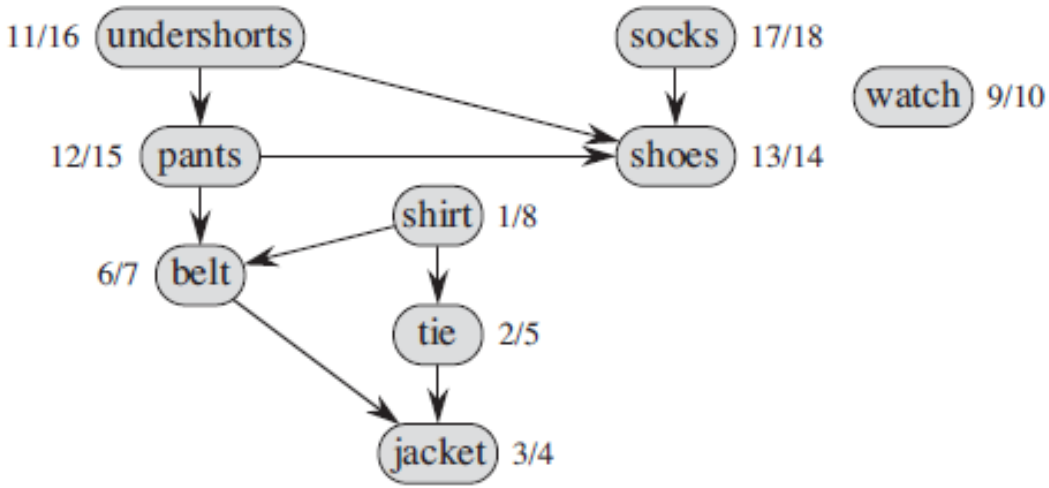
Running time: $O(n^2)$ (we will see in TD how to implement it in linear time).

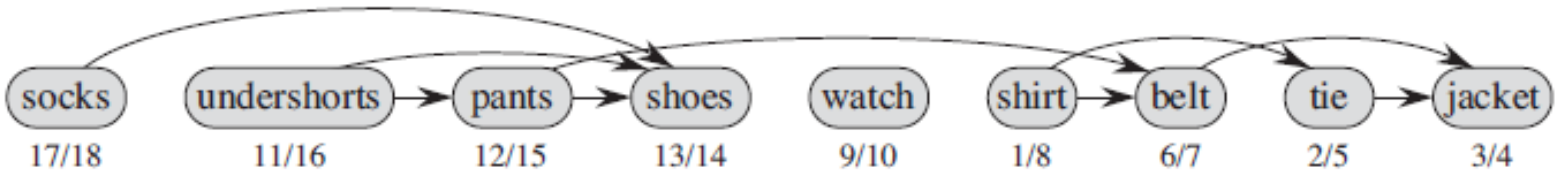
But DFS can do the same thing more efficiently and in a beautiful slick way.

Topological ordering using DFS

Algorithm 14 TOPOLOGICAL-SORT(G)

Call DFS(G)
As each vertex blackens, insert it onto the front of a linked list
return the list





Introduction to algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Running time and correctness

Running time: $O(n+m)$.

Sketch of proof of correctness:

- Assume you are executing the algo and you are at the time where a vertex, say u, is blackened and put in the linked list.
- It is enough to prove that all the out-neighbors of u are already in the linked list
- It reduces to prove that all the out-neighbors of u are black at this time.
- Let v be an out-neighbor of u,
  - it cannot be white (since u is black),
  - and if it is gray then G has a cycle (so we can output: "cycle").

Rigorous proof of correctness

First a Lemma of independent interest:

Lemma 22.11

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Proof $\Rightarrow$: Suppose that a depth-first search produces a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, G contains a path from v to u, and the back edge (u, v) completes a cycle.

$\Leftarrow$: Suppose that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Rigorous proof of correctness

Theorem 22.12

TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

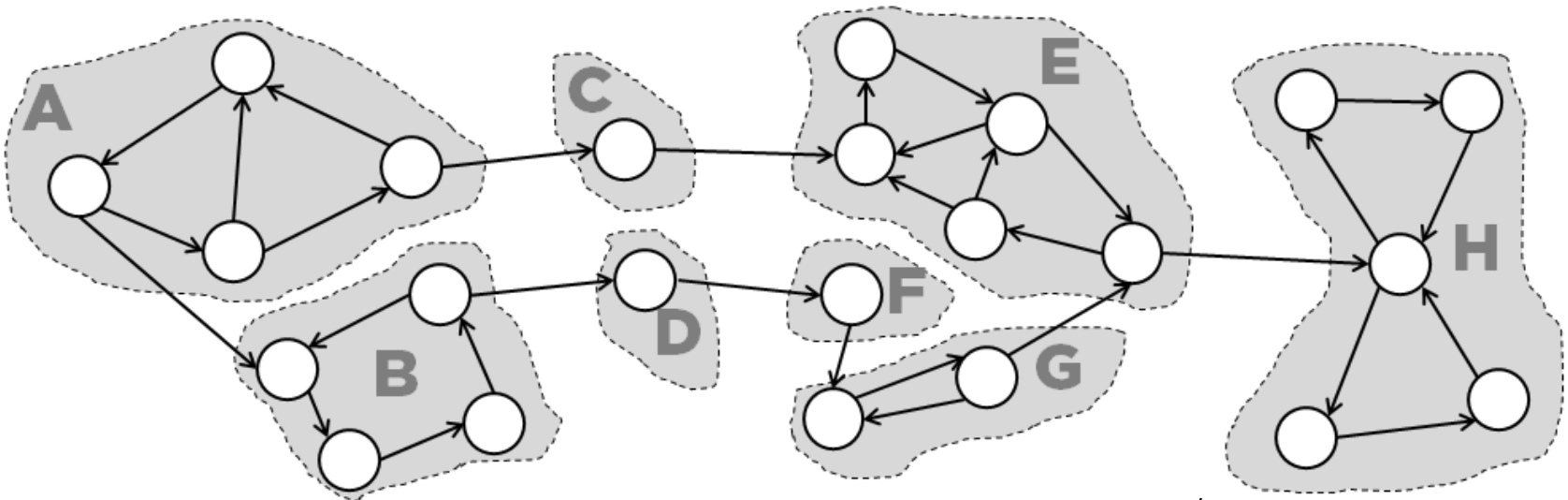
Proof Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting Lemma 22.11. Therefore, v must be either white or black. If v is white, it becomes a descendant of u, and so $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because we are still exploring from u, we have yet to assign a timestamp to $u.f$, and so once we do, we will have $v.f < u.f$ as well. Thus, for any edge (u, v) in the dag, we have $v.f < u.f$, proving the theorem. ■

Pics from Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Second Application of DFS: decompose a directed graph into strong connected components

Definition of strongly connected component

Let G be a directed graph.
A strong connected component (scc) of G is a maximum set of vertices C such that for every pair of vertices $u, v \in C$, there is a directed path from u to v and from v to u[3].



Pics from Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

[3] we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, we say u is strongly connected to v

The Strongly Component Graph

The idea behind this algorithm comes from a key property of the graph $G^{SCC} = (V^{SCC}, E^{SCC})$ obtained from G by shrinking strong connected components:

Let $C_1, \ldots, C_k$ be the strong connected components of G.

Set $V^{SCC} = \{v_1, \ldots, v_k\}$, one vertex for each scc.

And $v_i v_j \in E^{SCC}$ if and only if there is an arc from C_i to C_j in G.

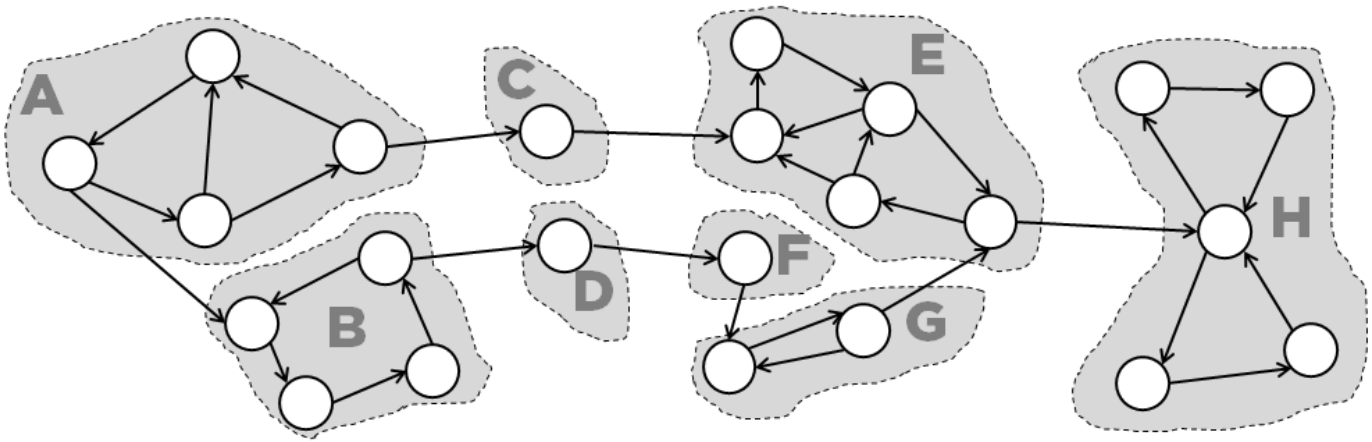


Figure from Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Key property of G^{SCC}

Key property: G^{SCC} is a DAG.

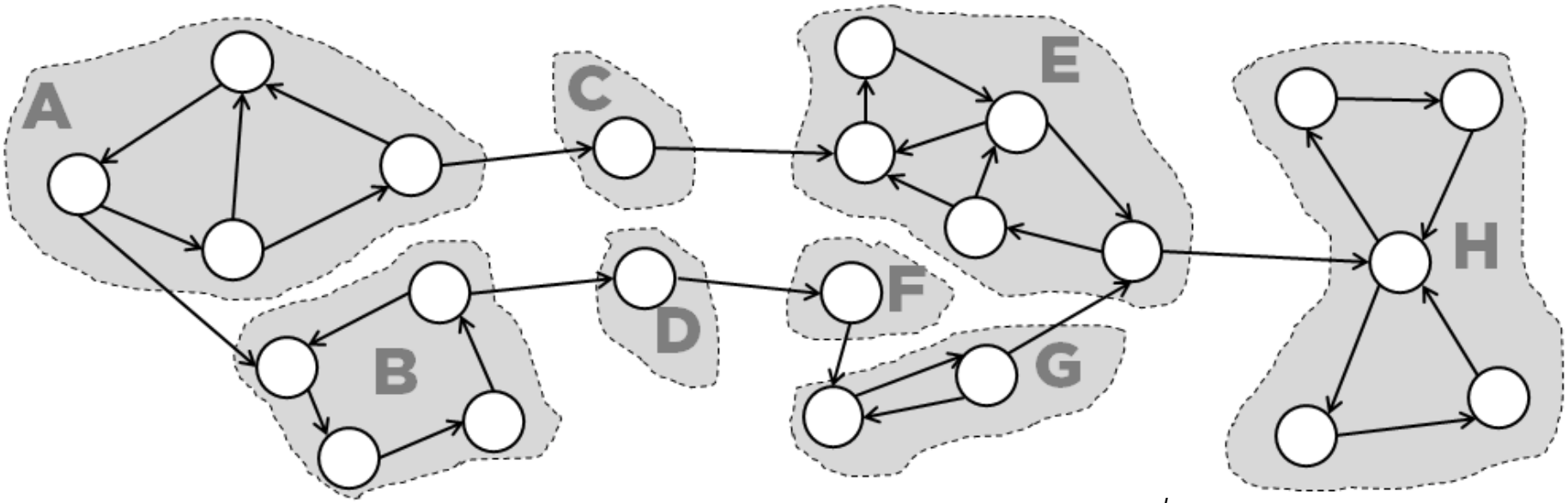
It is an easy corolary of the following Lemma:

Lemma: Let C and C' be two scc of a directed graph G. Let $u, v \in C$ and $u', v' \in C'$. Assume moreover that there is a path from u to u'. Then there cannot be a path from v' to v.

Proof: If G contains a path $v' \rightsquigarrow v$, then it contains paths and $v' \rightsquigarrow v \rightsquigarrow u$ and a path $u \rightsquigarrow u' \rightsquigarrow v'$. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components.

Intuition why DFS can compute scc

Idea: if we are lucky, the strong connected components correspond to the connected components of the DFS forest.



Pics from Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

Definitions

We extend the *discovery* and *finishing times* from vertices to sets of vertices: if S is a set of vertices, we write

- $d(S) = \min\{u.d : u \in S\}$ (first time a vertex of S is discovered)
- $f(S) = \max\{u.f : u \in S\}$ (time when all vertices of S are black).

Lemma: Let C and C' be two scc and let $u \in C$ and $v \in C'$ such that $uv \in E$. Then $f(C) > f(C')$.

Lemma: Let C and C' be two scc and let $u \in C$ and $v \in C'$ such that $uv \in E$. Then $f(C) > f(C')$.

Proof We consider two cases, depending on which strongly connected component, C or C', had the first discovered vertex during the depth-first search.

If $d(C) < d(C')$, let x be the first vertex discovered in C. At time $x.d$, all vertices in C and C' are white. At that time, G contains a path from x to each vertex in C consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C'$, there is also a path in G at time $x.d$ from x to w consisting only of white vertices: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary 22.8, x has the latest finishing time of any of its descendants, and so $x.f = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let y be the first vertex discovered in C'. At time $y.d$, all vertices in C' are white and G contains a path from y to each vertex in C' consisting only of white vertices. By the white-path theorem, all vertices in C' become descendants of y in the depth-first tree, and by Corollary 22.8, $y.f = f(C')$. At time $y.d$, all vertices in C are white. Since there is an edge (u, v) from C to C', Lemma 22.13 implies that there cannot be a path from C' to C. Hence, no vertex in C is reachable from y. At time $y.f$, therefore, all vertices in C are still white. Thus, for any vertex $w \in C$, we have $w.f > y.f$, which implies that $f(C) > f(C')$. ■

Algorithm 15 DFS(G)

```
1: for each u ∈ V do
2:     u.color ← White, and u.pi ← NIL
3: time ← 0                          ▷ time is a global variable used for timestamping
4: for each vertex v ∈ V do
5:     if v.color == White then
6:         DFS-Visit(G, v)
```

Algorithm 16 DFS-Visit(G, v)

```
1: time ← time + 1
2: u.d ← time and u.color ← Gray
3: for each v ∈ Adj[u] do                          ▷ For each arc uv
4:     if v.color == White then                    ▷ if v has not been discovered yet
5:         v.π ← u
6:         DFS-Visit(G, v)
7: time ← time + 1
8: u.color ← Black and u.f ← time
```

The reverse of a directed graph

Let $G = (V, E)$ be a directed graph.
We define the reverse of G as $G^R = (V, E^R)$ where $E^R = \{uv : vu \in E\}$ (reverse all arcs)

The reverse of a directed graph

Let $G = (V, E)$ be a directed graph.
We define the reverse of G as $G^R = (V, E^R)$ where $E^R = \{uv : vu \in E\}$ (reverse all arcs)

Question: How are the scc of graph G and G^R related?.

The reverse of a directed graph

Let $G = (V, E)$ be a directed graph.
We define the reverse of G as $G^R = (V, E^R)$ where $E^R = \{uv : vu \in E\}$ (reverse all arcs)

Question: How are the scc of graph G and G^R related?.

Answer: G and G^R have the same scc!

The reverse of a directed graph

Let $G = (V, E)$ be a directed graph.
We define the reverse of G as $G^R = (V, E^R)$ where $E^R = \{uv : vu \in E\}$ (reverse all arcs)

Question: How are the scc of graph G and G^R related?.

Answer: G and G^R have the same scc!

Exercise: How can you compute G^R efficiently?

Kosaraju's two pass algorithm

The following linear-time algorithm mysteriously computes the scc using two depth-first searches, one on G and one on G^R.

Algorithm 17 $\mathrm{SCC}(G)$

1: Call $\mathrm{DFS}(G)$ to compute the *finished* time $u.f$ of each vertex
2: Compute G^R
3: Call $\mathrm{DFS}(G^R)$, but in the main loop of DFS, consider the vertices in order of decreasing $u.f$
4: Output the vertices of each tree in the DFS forest computed by $\mathrm{DFS}(G^R)$ as the set of strongly connected components

The main loop of DFS corresponds to line 4:

1: **for** each $u \in V$ **do**
2: $\quad u.color \leftarrow White$, and $u.pi \leftarrow NIL$
3: $time \leftarrow 0$
4: **for** each vertex $v \in V$ **do**
5: $\quad$ **if** $v.color == White$ **then**
6: $\quad\quad \mathrm{DFS\text{-}Visit}(G, v)$

Execution of SCC

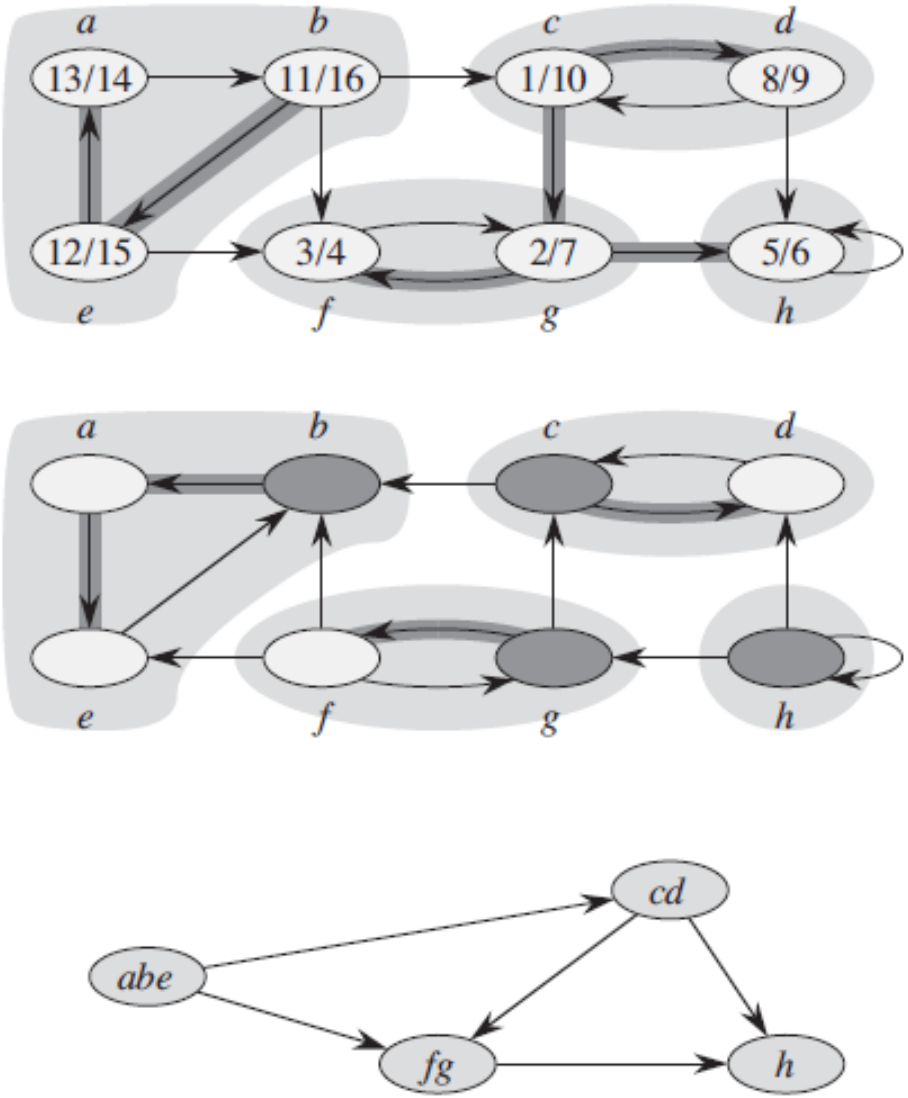


Figure from Introduction to algorithms, 3^{rd} edition, Cormen, Leiserson, Rivest, Stein

By the previous Lemma, SCC output the strongly connected components of the input graph.

3 - Minimum Weighted Spanning Tree (MST)

Trees

A tree is a graph that is:

- connected and
- acyclic.

A forest is an acyclic graph (i.e. its connected components are trees).

Trees

A tree is a graph that is:

- connected and
- acyclic.

A forest is an acyclic graph (i.e. its connected components are trees).

A leaf of a tree is a vertex of degree 1 in T.

Properties:

- Every tree T contains at least 2 leaves.
- Every tree T satifies $|V(T)| = |E(T)| - 1$ (and if F is a forest, $|V(T)| = |E(T)| - cc(F)$ where $cc(F)$ is the number of connected components of F).
- Every subgraph of a tree is a forest.

Proof: Let P be a longest path of T. The two extremities of P have degree 1. To prove the second point do an induction on the number of vertices: delete a leaf, the result graph is still a tree, apply induction, conclude.

Five Trees' characterizations

Tree's characterizations:

Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).

Five Trees' characterizations

Tree's characterizations:

Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).
2. Any two vertices in G are linked by a unique path.

Five Trees' characterizations

Tree's characterizations:
Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).
2. Any two vertices in G are linked by a unique path.
3. G is connected, and $|E| = |V| - 1$ ($m = n - 1$).

Five Trees' characterizations

Tree's characterizations:
Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).
2. Any two vertices in G are linked by a unique path.
3. G is connected, and $|E| = |V| - 1$ ($m = n - 1$).
4. G is acyclic, and $|E| = |V| - 1$ ($m = n - 1$).

Five Trees' characterizations

Tree's characterizations:
Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).
2. Any two vertices in G are linked by a unique path.
3. G is connected, and $|E| = |V| - 1$ ($m = n - 1$).
4. G is acyclic, and $|E| = |V| - 1$ ($m = n - 1$).
5. G is connected, but for every edge $e = uv$, $G - \{uv\}$ has exactly two connected components, one containing u the other v.

Five Trees' characterizations

Tree's characterizations:
Let $G = (V, E)$ be a graph. The following are equivalent:

1. G is a tree (i.e. connected acyclic graph).
2. Any two vertices in G are linked by a unique path.
3. G is connected, and $|E| = |V| - 1$ ($m = n - 1$).
4. G is acyclic, and $|E| = |V| - 1$ ($m = n - 1$).
5. G is connected, but for every edge $e = uv$, $G - \{uv\}$ has exactly two connected components, one containing u the other v.
6. G is acyclic, but if any edge is added to G, the resulting graph contains a unique cycle (going through this added edge).

For a proof see Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein, Theorem B.2

Definitions

- Let $G = (V, E)$ be a graph.
- Let $\omega : E \to \mathbb{R}$ be a a weight function on the edges of G.
- Then $G = (V, E, \omega)$ is called an edge-weighted graph.

Definitions

- Let $G = (V, E)$ be a graph.
- Let $\omega : E \to \mathbb{R}$ be a a weight function on the edges of G.
- Then $G = (V, E, \omega)$ is called an edge-weighted graph.

- Spanning tree: subgraph of G that is a tree and contains all vertices of G.
- The weight of a tree T is the sum of the weights of its edges:

$$w(T) = \sum_{uv \in T} w(uv)$$

Problem (Minimum Weighted Spanning Tree (MST))

Input : An edge-weighted graph $G = (V, E, \omega)$.
Output : A spanning tree of minimum weight.

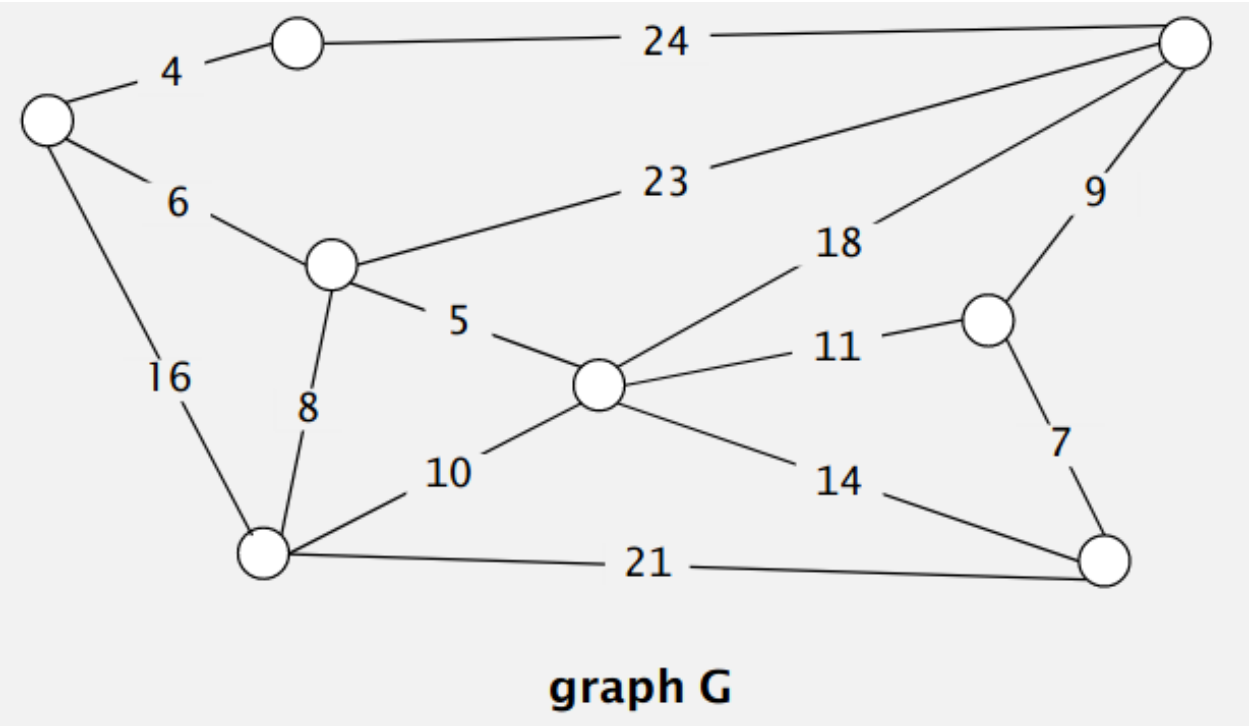


figure by Kevin Wayne

Problem (Minimum Weighted Spanning Tree (MST))

Input : An edge-weighted graph $G = (V, E, \omega)$.
Output : A spanning tree of minimum weight.

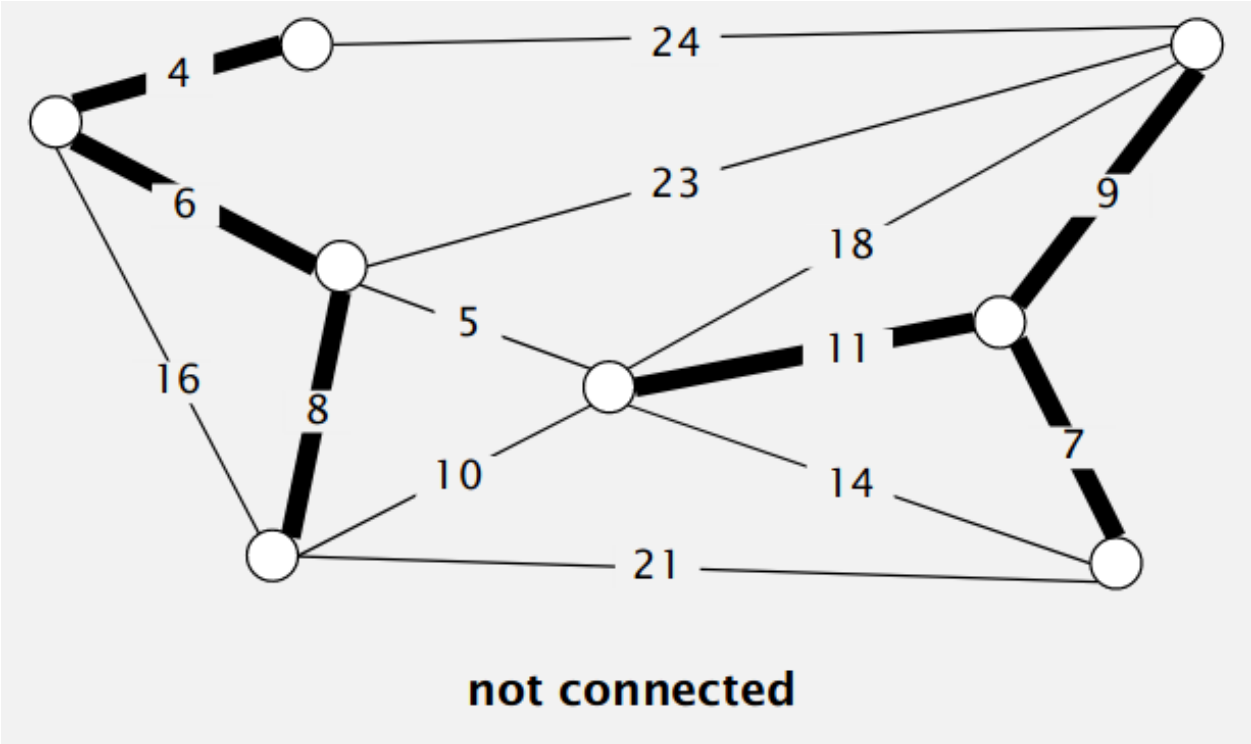


figure by Kevin Wayne

Problem (Minimum Weighted Spanning Tree (MST))

Input : An edge-weighted graph $G = (V, E, \omega)$.
Output : A spanning tree of minimum weight.

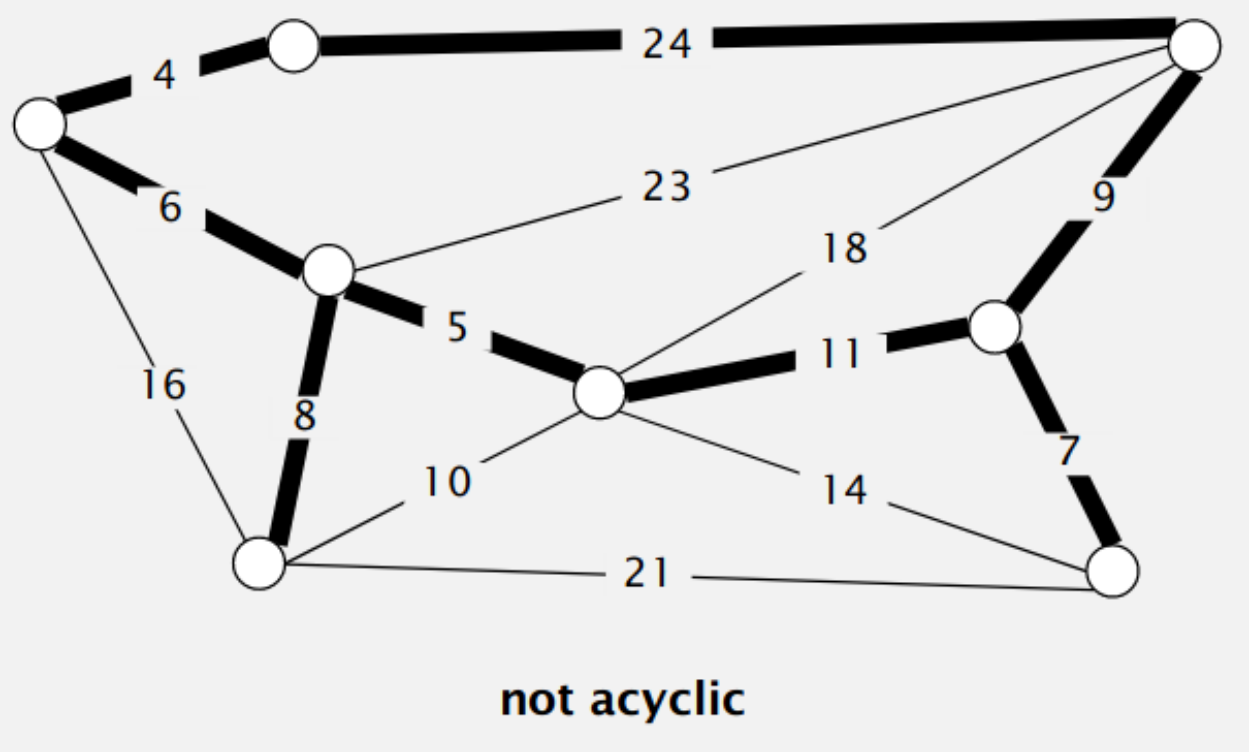


figure by Kevin Wayne

Problem (Minimum Weighted Spanning Tree (MST))

Input : An edge-weighted graph $G = (V, E, \omega)$.
Output : A spanning tree of minimum weight.

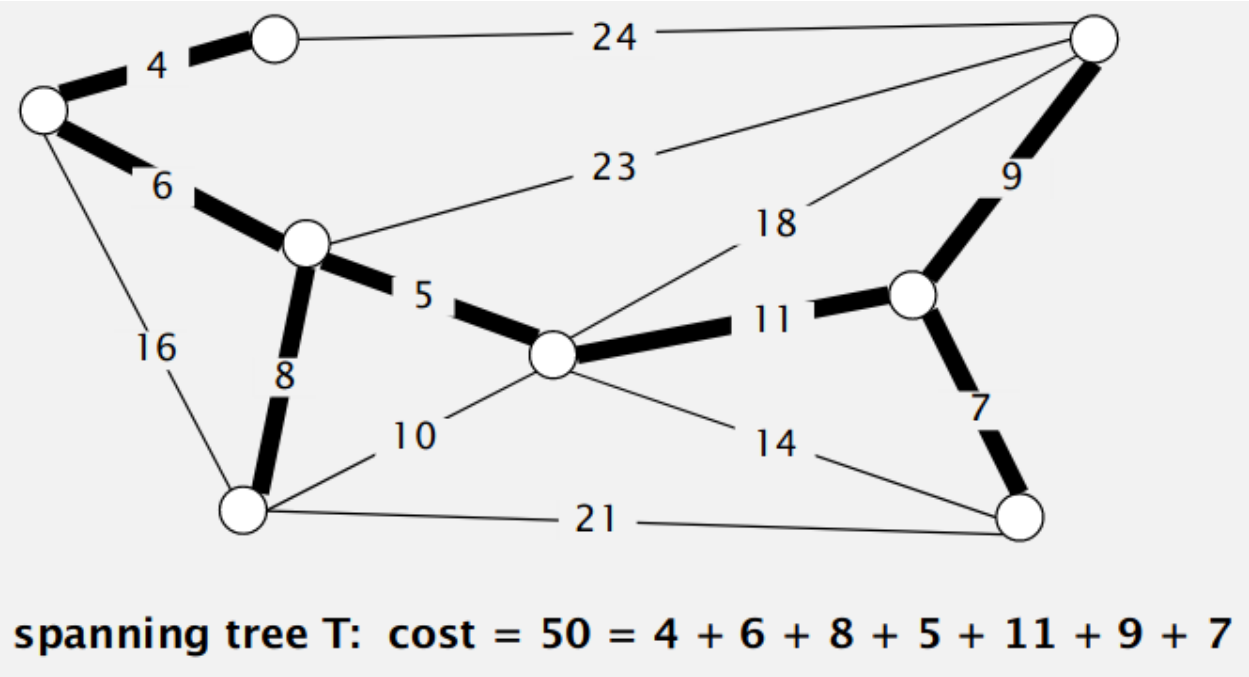


figure by Kevin Wayne

Combinatorial Optimization

In operations research, applied mathematics and theoretical computer science, combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not tractable *according to wkipedia*.

Combinatorial Optimization

In operations research, applied mathematics and theoretical computer science, combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects. In many such problems, exhaustive search is not tractable *according to wkipedia*.

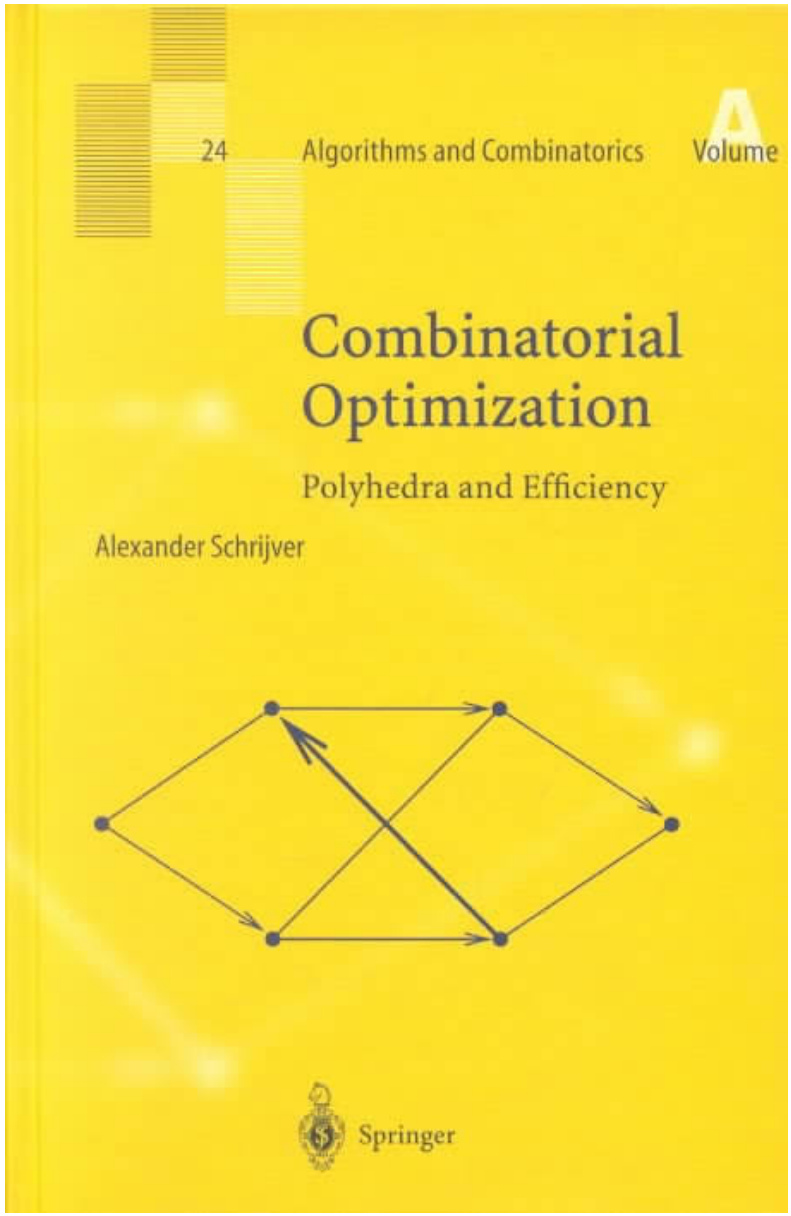
An optimization problem is a problem where you have to maximize (or minimize) a quantity.

Famous combinatorial problems:

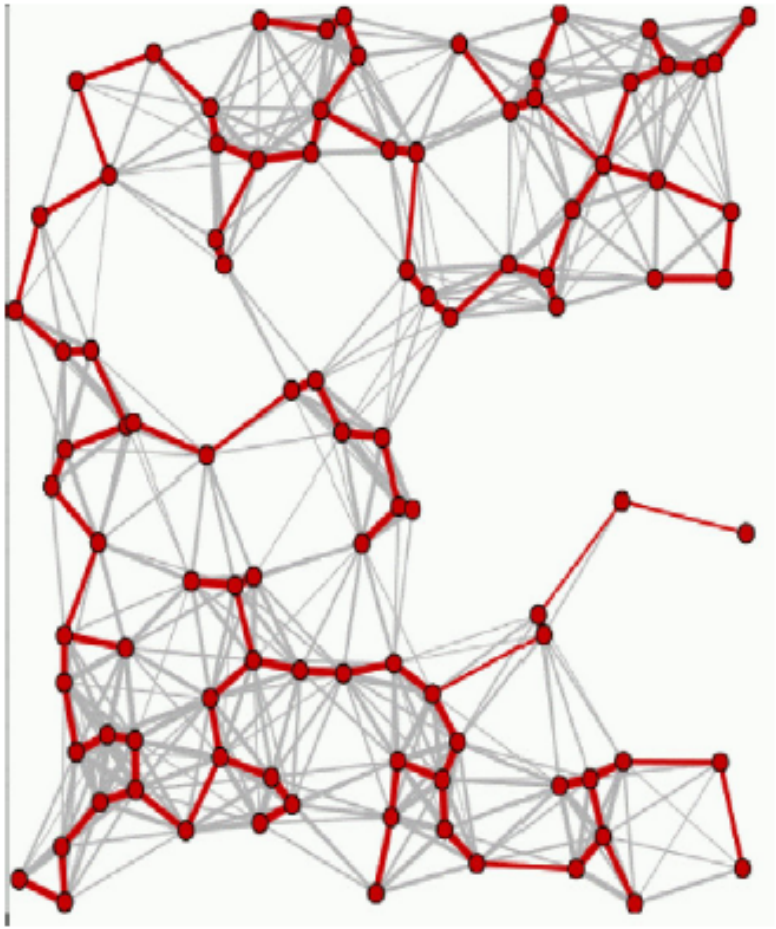
- Minimum Spanning Tree problem ("MST"),
- Knapsack problem.
- Travelling Salesman Problem ("TSP").

Reference: *On the history of combinatorial optimization* by Alexander Schrijver

Ultimate reference for Combinatorial Optimization



Applications of MST



Many applications:

- Electrical, communication, road etc network design.
- Data coding and clustering.
- Approximate NP-complete graph optimisation.
  - Travelling salesman problem: the MST is within a factor of two of the optimal path.
- Image analysis.

http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/

Spanning tree and the cut and paste technic

Spanning tree: subgraph of G that is a tree and contains all vertices of G.

Spanning tree and the cut and paste technic

Spanning tree: subgraph of G that is a tree and contains all vertices of G.

Cut and paste technic

Let $G = (V, E)$ and let T be a spanning tree of G. Let $uv \in E(G) - T$ and let T_{uv} be the unique path linking u and v in T. Then for every edge xy of T_{uv}, $T \setminus \{xy\} \cup \{uv\}$ is a spanning tree of T.

Spanning tree and the cut and paste technic

Spanning tree: subgraph of G that is a tree and contains all vertices of G.

Cut and paste technic

Let $G = (V, E)$ and let T be a spanning tree of G. Let $uv \in E(G) - T$ and let T_{uv} be the unique path linking u and v in T. Then for every edge xy of T_{uv}, $T \setminus \{xy\} \cup \{uv\}$ is a spanning tree of T.

Proof: Since T_{uv} is in the unique path of T linking u and v, removing any edge $xy \in T_{uv}$ from T breaks T into two connected components, one containing u and the other v. Adding uv reconnects the two parts and form a new spanning tree $T \setminus \{xy\} \cup \{uv\}$.

Characterization of minimum spanning tree

Theorem: Let $G = (V, E, \omega)$ be an edge-weighted graph.
A spanning tree T of G is a minimum spanning tree if and only if for every edge $e \in E \setminus T$:

$$\omega(e) \geq \omega(f) \text{ for every edge } f \text{ of the unique cycle of } T \cup \{e\}$$

Proof: straighforwatd with the: Cut and paste technic

Hidden here: a matroid structure

If I give you a graph, how would you find a Minimum Spanning Tree?

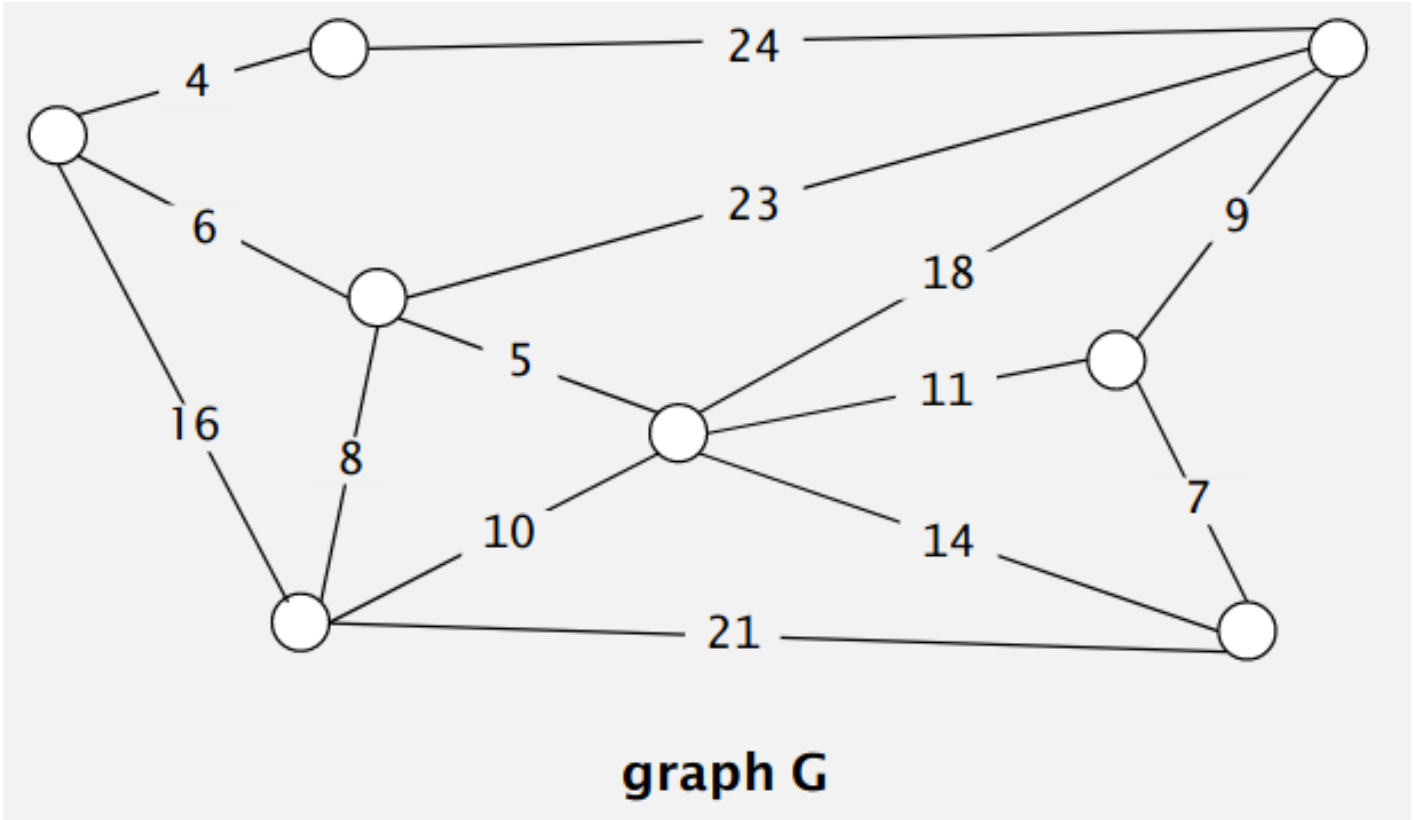


figure by Kevin Wayne

If I give you a graph, how would you find a Minimum Spanning Tree?

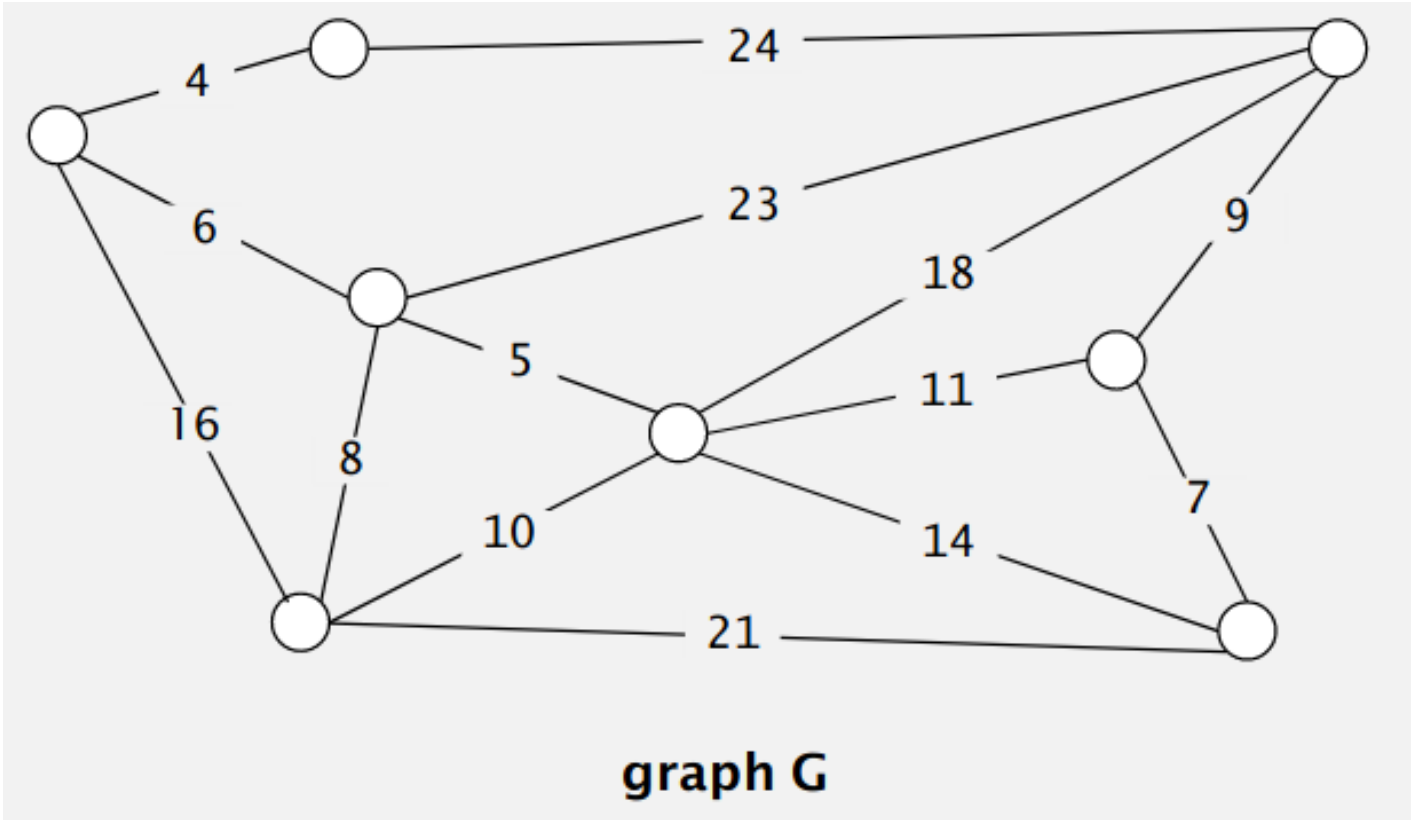


figure by Kevin Wayne

Strategy: greedy, choose one edge at a time, until you reach a solution.

The generic method

- A set of edges A is said to be promising if it can be completed into a minimum spanning tree.

The generic method

- A set of edges A is said to be promising if it can be completed into a minimum spanning tree.

The generic method manages a set of edges A, maintaining the following loop invariant:

Prior to each iteration, A is promising

- At each step, we determine an edge uv that we can add to A without violating this invariant, that is $A \cup \{uv\}$ is still a promising set.
- Such an edge uv is said to be safe with respect to A.

Pseudocode for the generic method

Algorithm 18 GREEDY-MST($G = (V, E, \omega)$)

1: $A = \emptyset$
2: **while** A is not a spanning tree **do**
3: Find an edge uv that is safe for A
4: $A \leftarrow A \cup \{uv\}$
return A

Pseudocode for the generic method

Algorithm 19 GREEDY-MST($G = (V, E, \omega)$)

1: $A = \emptyset$
2: **while** A is not a spanning tree **do**
3: Find an edge uv that is safe for A
4: $A \leftarrow A \cup \{uv\}$
return A

Proof of correctness:

- **Initialization**: After line 1, the set A trivially satisfies the loop invariant.
- **Maintenance**: The loop in lines 2–4 maintains the invariant by adding only safe edges.
- **Termination**: All edges added to A are in a minimum spanning tree, and so the set A returned in line 5 must be a minimum spanning tree.

The tricky part is of course to find a safe edge.

How can we be sure that an edge is safe

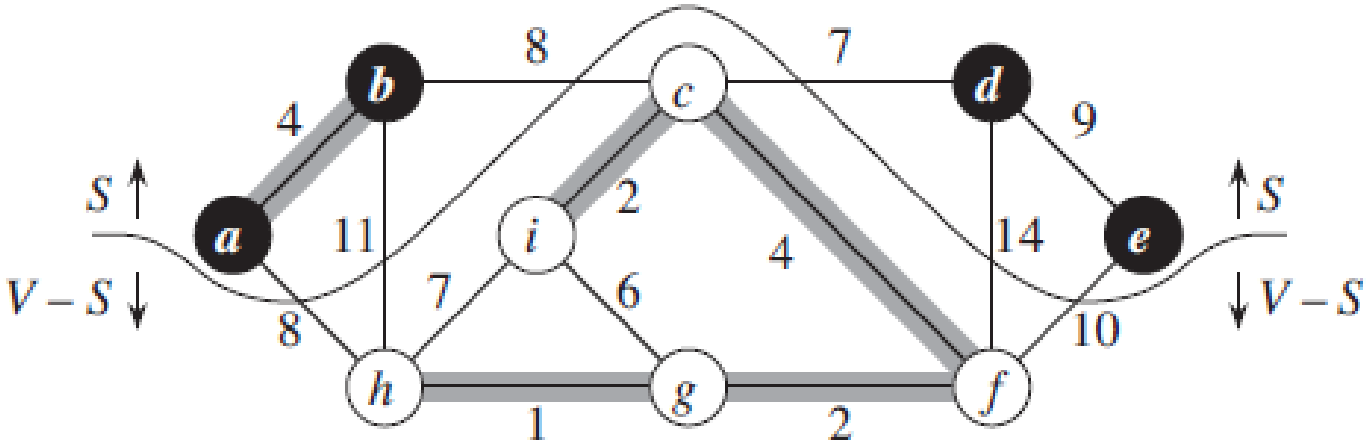
We first need some definitions. Let $G = (V, E)$ be a graph.

- Let $S \subseteq V$, a cut $(S, V - S)$ is a partition of V.
- An edge of the cut (also called crossing edge) is an edge with one end in S and the other in $V - S$.
- We say that a cut respects a set of edges A if no edge of A is a crossing edge of the cut.

The safe property

Safe Property:

Let $G = (V, E, \omega)$ and let A be a promising set of edges. Let $(S, V - S)$ be a cut respecting A and let uv be a lightest crossing edge. Then uv is safe.



Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

- The shaded edges form the promising set A.
- The black vertices are in S, the white in $V - S$. Oberve that it respects A.
- dc is a lightest edge of the cut, it is claimed to be safe by the property.

Proof of the safe property

Safe Property:
Let $G = (V, E, \omega)$ and let A be a promising set of edges. Let $(S, V - S)$ be a cut respecting A and let uv be a lightest crossing edge. Then uv is safe.

Proof:

Proof of the safe property

Safe Property:
Let $G = (V, E, \omega)$ and let A be a promising set of edges. Let $(S, V - S)$ be a cut respecting A and let uv be a lightest crossing edge. Then uv is safe.

Proof: Let T be a MST containing A. If T contains uv we are done, so assume it does not. Let T_{uv} be the unique path linking u and v in T. Since uv is a crossing edge of $(S, V - S)$, T_{uv} contains an edge xy of $(S, V - S)$. By the cut and paste technique, $T' = (T \setminus xy) \cup uv$ is a spanning tree of G. By hypothesis, uv is a lightest edge of $(S, V - S)$, so $\omega(uv) \leq \omega(xy)$ and thus:

$$\omega(T') = \omega(T) + \omega(uv) - \omega(xy) \leq \omega(T)$$

So T' is a MST containing uv. □

Two greedy algorithms

Thanks to the safe property, we get that GREEDY-MST computes an MST. Finding the lightest edge of a cut can take $O(m)$ time, which would lead to a $O(nm^2)$-time algorithm.

We are going to see two efficient way to implement GREEDY-MST:

- Kruskal's algorithm: grows a forest whose trees coalesce into one spanning tree
- Prim's algorithm: grows a tree until it becomes a spanning tree

Kruskal's Algorithm

Kruskal's algorithm

Kruskal algorithm is a greedy algorithm that grows a promising forest A:

- Sort the edges by non-decreasing order of weight
- Start with the empty set (or more precisely with $(V, \emptyset)$)
- Add a minimum weighted edge that does not create a cycle.

$\Leftrightarrow$

Add a minimum weighted edge that connects two connected components of the growing forest.

Show an execution

Kruskal's algorithm

Kruskal algorithm is a greedy algorithm that grows a promising forest A:

- Sort the edges by non-decreasing order of weight
- Start with the empty set (or more precisely with $(V, \emptyset)$)
- Add a minimum weighted edge that does not create a cycle.

$\Leftrightarrow$

Add a minimum weighted edge that connects two connected components of the growing forest.

Show an execution

Question: How would you implement it?

Kruskal's algorithm

Kruskal algorithm is a greedy algorithm that grows a promising forest A:

- Sort the edges by non-decreasing order of weight
- Start with the empty set (or more precisely with $(V, \emptyset)$)
- Add a minimum weighted edge that does not create a cycle.

$\Leftrightarrow$

Add a minimum weighted edge that connects two connected components of the growing forest.

Show an execution

Question: How would you implement it?

Answer: Disjoint Set Data Structure is perfect to implement it!

Reminder on disjoint set data structure

- Maintain a collection of disjoint dynamic sets $\mathcal{S} = \{S_1, \ldots, S_k\}$,
- Each set has a representative (or leader)
- It supports three operations:
  - MAKE-SET(x): create a new set containing only x (leader is x).
  - UNION-SET(x,y): union the sets containing x (and y and choose a leader of the new set).
  - FIND(x): return a pointer to the leader of the set containing x.
- Two parameters for the complexity analysis:
  - n: number of elements (= number of MAKE-SET)
  - m: total number of operations

Amortized complexity: $m\alpha(n)$ where α is the inverse Ackerman function.

Pseudo code for Kruskal algorithm

Algorithm 20 MST-Kruskal($G = (V, E, \omega)$)

```
1: A = ∅                                          ▷ A is the growing forest
2: for each vertex u ∈ V do
3:     MAKE-SET(u)
4: Sort the edges by non-decreasing order of weight
5: for each edge uv taken in non-decreasing order do
6:     if FIND(u) ≠FIND(v) then
7:         A = A ∪ {uv}
8:         UNION(u, v)
   return A
```

Pseudo code for Kruskal algorithm

Algorithm 21 MST-Kruskal($G = (V, E, \omega)$)

1: $A = \emptyset$ ▷ A is the growing forest
2: **for** each vertex $u \in V$ **do**
3: MAKE-SET(u)
4: Sort the edges by non-decreasing order of weight
5: **for** each edge uv taken in non-decreasing order **do**
6: **if** FIND(u) $\neq$FIND(v) **then**
7: $A = A \cup \{uv\}$
8: UNION(u, v)
return A

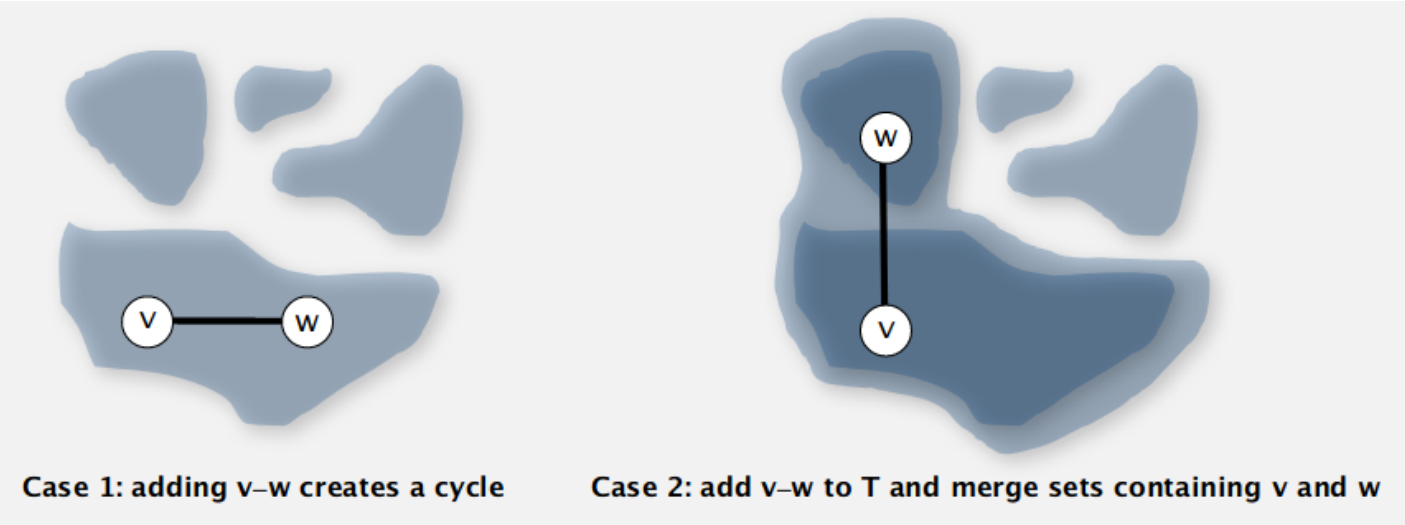


figure by Kevin Wayne

Pseudo code for Kruskal algorithm

Algorithm 22 MST-Kruskal($G = (V, E, \omega)$)

1: $A = \emptyset$ ▷ A is the growing forest
2: **for** each vertex $u \in V$ **do**
3: MAKE-SET(u)
4: Sort the edges by non-decreasing order of weight
5: **for** each edge uv taken in non-decreasing order **do**
6: **if** FIND(u) $\neq$FIND(v) **then**
7: $A = A \cup \{uv\}$
8: UNION(u, v)
return A

Complexity analyse (recall that $|V| = n$ and $|E| = m$):

- ℓ. 4: $O(m \log m) = O(m \log(n))$ because $m = O(n^2)$.
- ℓ. 2-3 and 5-8: $O(m + n)\,\alpha(n)$[4]
  - $O(m)$ FIND-SET and
  - $O(n)$ UNION and MAKE-SET

All together, the sorting wins: $O(m \log(n))$

[4]We implement the *union-find* operations using the disjoint-set-forest implementation with the union-by-rank and path-compression heuristics

Correctness of Kruskal

Loop invariant:

Prior to each iteration, A is promising

Correctness of Kruskal

Loop invariant:

Prior to each iteration, A is promising

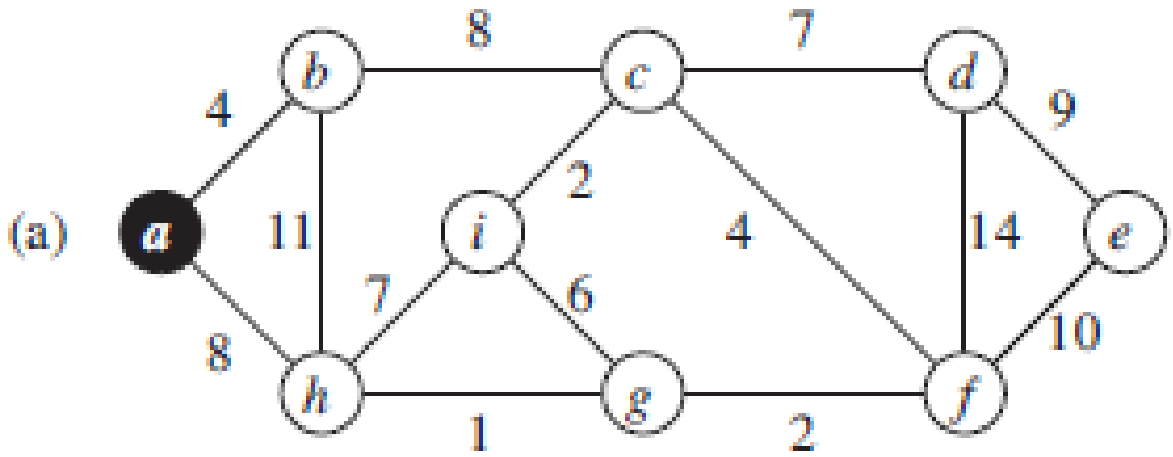
- Assume A is the growing forest and the algorithm tells you to add the edge uv.
- So u and v are in two distinct connected components of A.
- Take a cut $(S, V - S)$ such that:
  - The connected component of A containg u is in S
  - The connected component of A containing v is in $V - S$.
  - Put all the other connected components in S.
- Then $(S, V \setminus S)$ is a cut respecting A (i.e. no crossing edge is in A),
- None of the crossing edge have been scanned yet, so uv must be the ligthest crossing edge.
- So, by the *Safe Property*, uv is safe and thus $A \cup \{uv\}$ is promising.

Prim's Algorithm

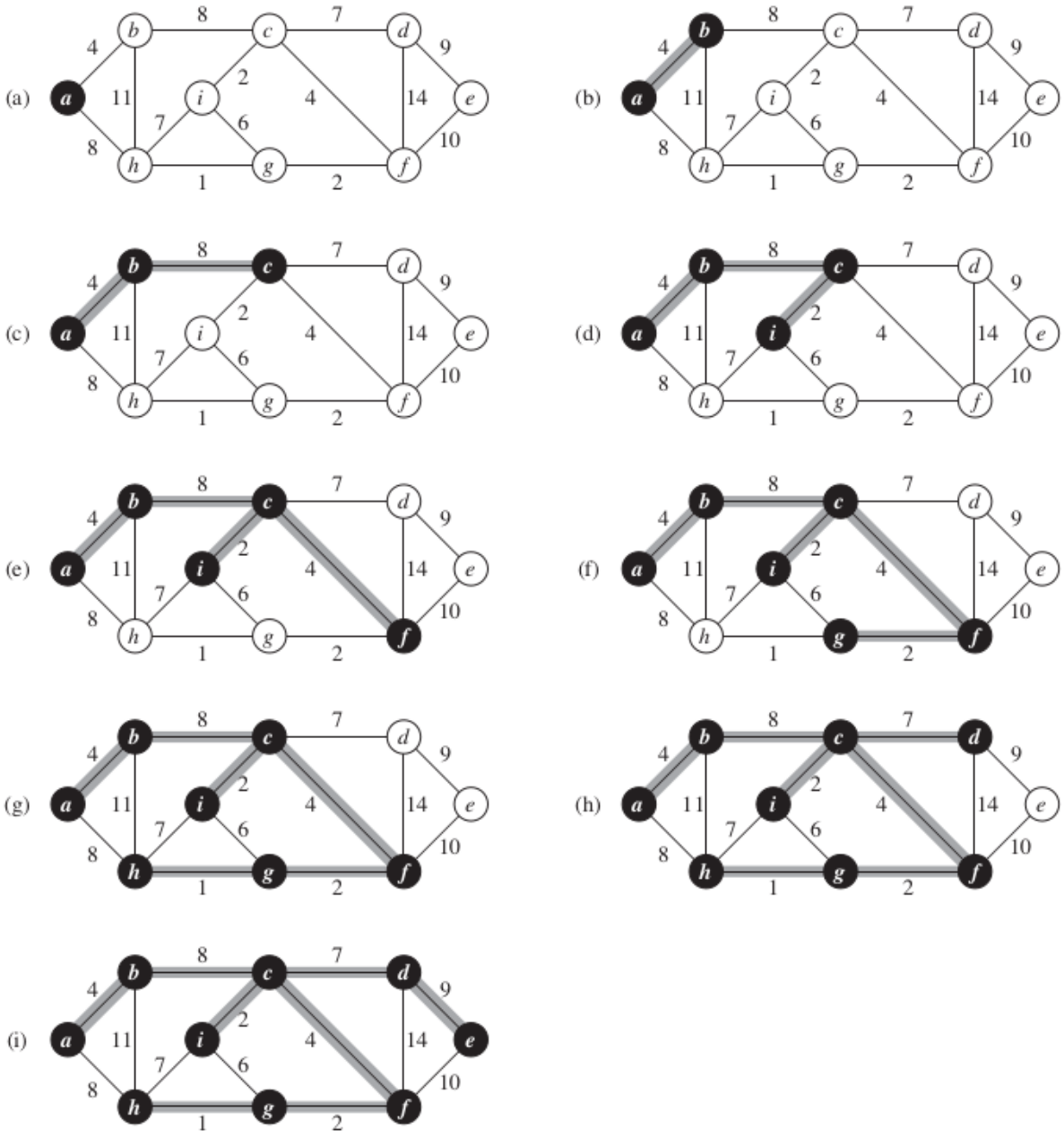
Prim's algorithm

Prim's algorithm is like Kruskal, but instead of growing a forest, it grows a tree A.

- Start with a vertex
- **Repeat:** Add the lightest edge with exactly one extremity in $V(A)$



Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein



Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Priority-Queue

- To find the next edge to add, we use a priority queue.

Priority Queue

Collection. Insert and Delete elements.

The way we choose the next element to delete define distinct type of data structures.

Stack. Delete the element most recently added.

Queue. Delete the element the least recently added.

Priority Queue. Delete the largest (or smallest) item. (Actually it does a bit more).

Priority Queue Applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Picture from Kevin Wayne

Priority queue

A priority queue is a data structure that maintains a set S of elements, each with an associated value called a *key*.

[5] I let you guess the operations supported by a max-priority queue

Priority queue

A priority queue is a data structure that maintains a set S of elements, each with an associated value called a *key*.

There is min-priority queue and max-priority queue.

A min-priority queue supports the following operations[5]:

- INSERT(S, x) inserts the element x into the set S; i.e. $S \leftarrow S \cup \{x\}$.
- MINIMUM(S) returns the element of S with the smallest key.
- EXTRACT-MIN(S) removes and returns the element of S with the smallest key.
- DECREASE-KEY(S, x, k) does $x.key \leftarrow k$ ($k \leq x.key$ is assumed).

Efficient way to implement it: min-heap.

[5] I let you guess the operations supported by a max-priority queue

How to implement a priority queue efficiently

- Store keys in a linked list.
  - ▶ Insert: $\mathcal{O}(1)$
  - ▶ Delete max: $\mathcal{O}(n)$
- Store keys in an ordered array.
  - ▶ Insert: $\mathcal{O}(n)$
  - ▶ Delete max: $\mathcal{O}(1)$

How to implement a priority queue efficiently

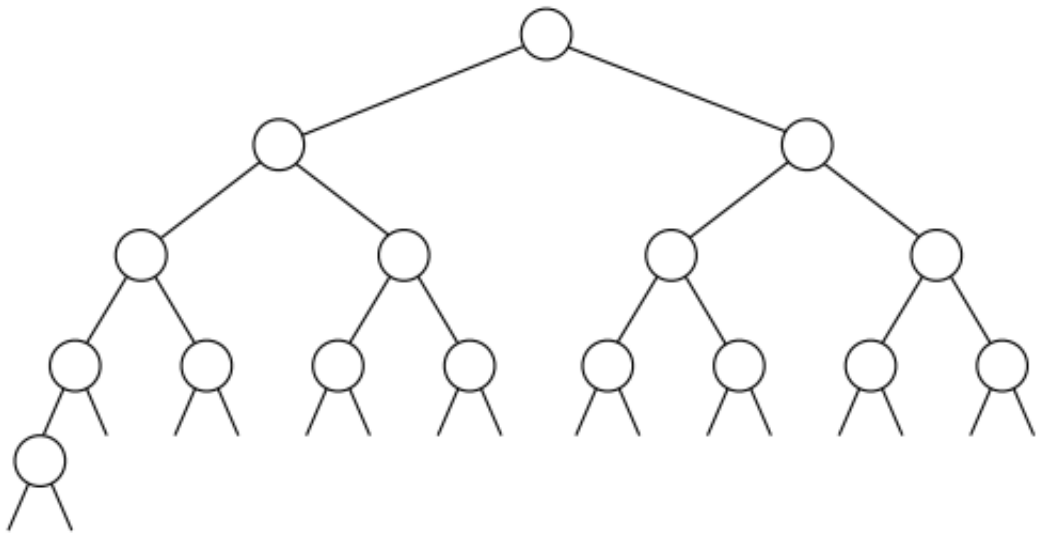
order of growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

Picture from Kevin Wayne

Complete binary tree

Complete binary tree: (except possibly the last one) is completely filled; the last level is filled from left to right.



complete binary tree with n = 16 nodes (height = 4)

Figure from *Kevin Wayne*

Property: heigh of a complete binary tree on n nodes is $\lceil \log_2(n) \rceil$

A wild complete binary tree (of height 4)

Max-heap

Definition of Max-heap:

- It is a complete binary tree.
- Key of a node is larger then keys of its children.
- A max-heap can be implemented in an array:
  $\textsc{Parent}(i) = \lfloor \frac{i}{2} \rfloor$, $\textsc{Left}(i) = 2i$, $\textsc{Right}(i) = 2i + 1$: $O(1)$.

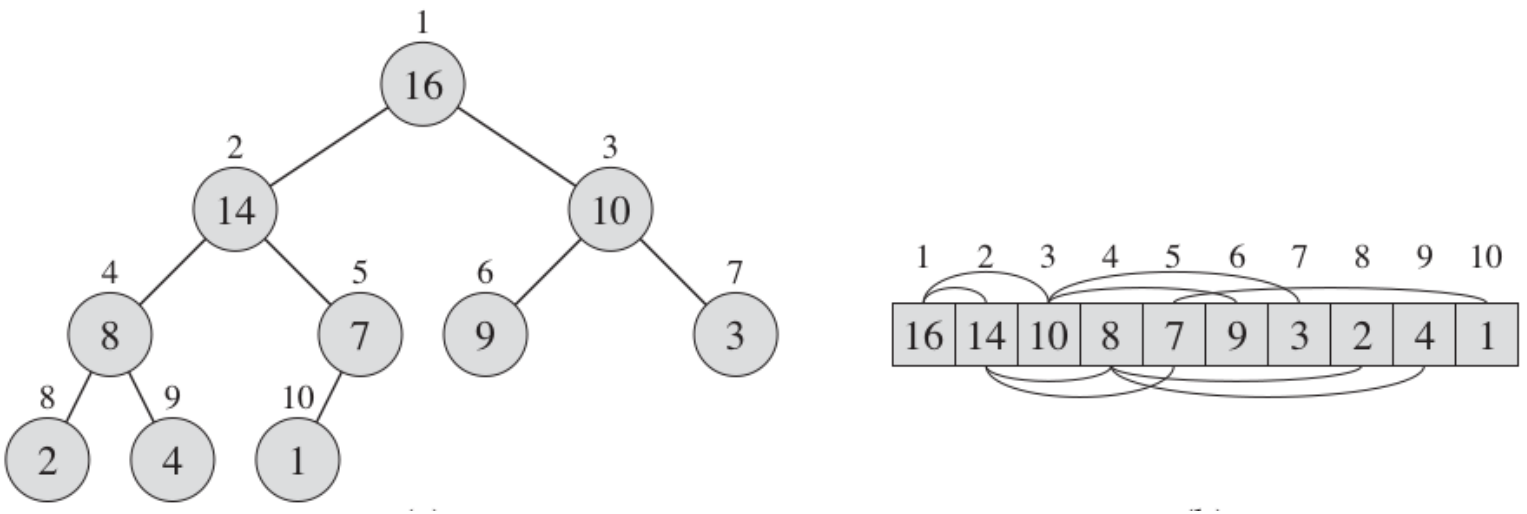


Figure from *Introduction to Algorithms*, 3rd edition, Cormen, Leiserson, Rivest, Stein

Strategy

- INSERT(S, x) inserts the element x into the set S; i.e. $S \leftarrow S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the smallest key.
- EXTRACT-MAX(S) removes and returns the element of S with the smallest key.
- INCREASE-KEY(S, x, k) does $x.key \leftarrow k$ ($k \leq x.key$ is assumed).

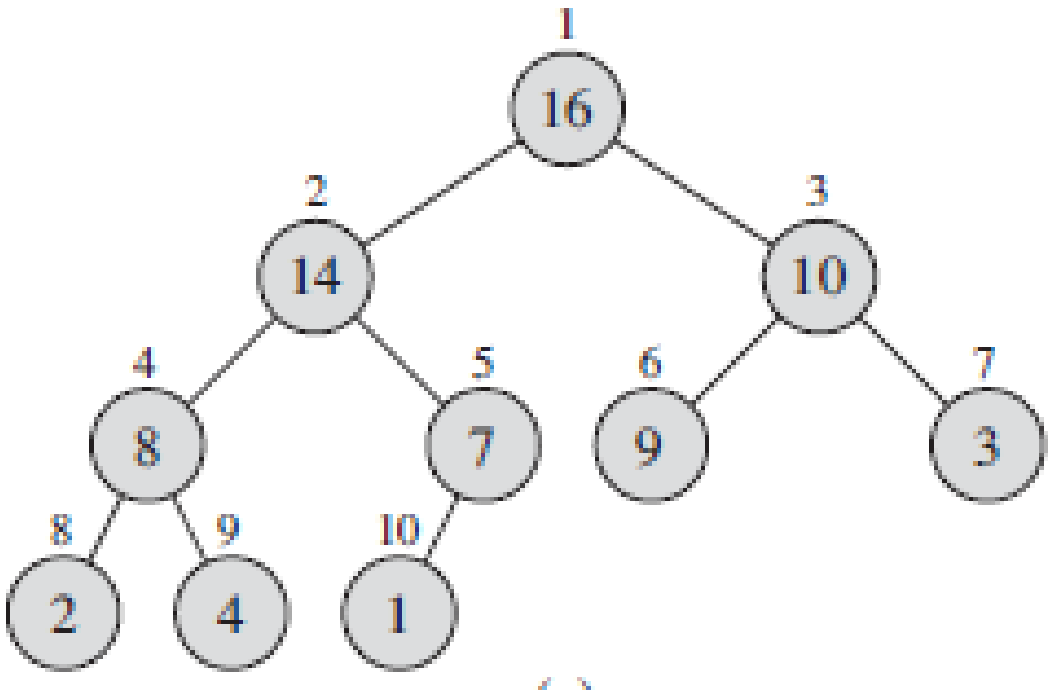


Figure from *Introduction to Algorithms*, 3rd edition, Cormen, Leiserson, Rivest, Stein

Max-Heapify(A,i) is the key procedure to maintain max-heap property.

- **Input**: an complete binary tree A (represented as an array) and an index i.
- It assumes that the trees rooted at the children of i are max-heap.
- But $A[i]$ might be smaller then its children, violating the max-heap property.
- After max-heapify(i), the tree rooted at i is a max-heap.

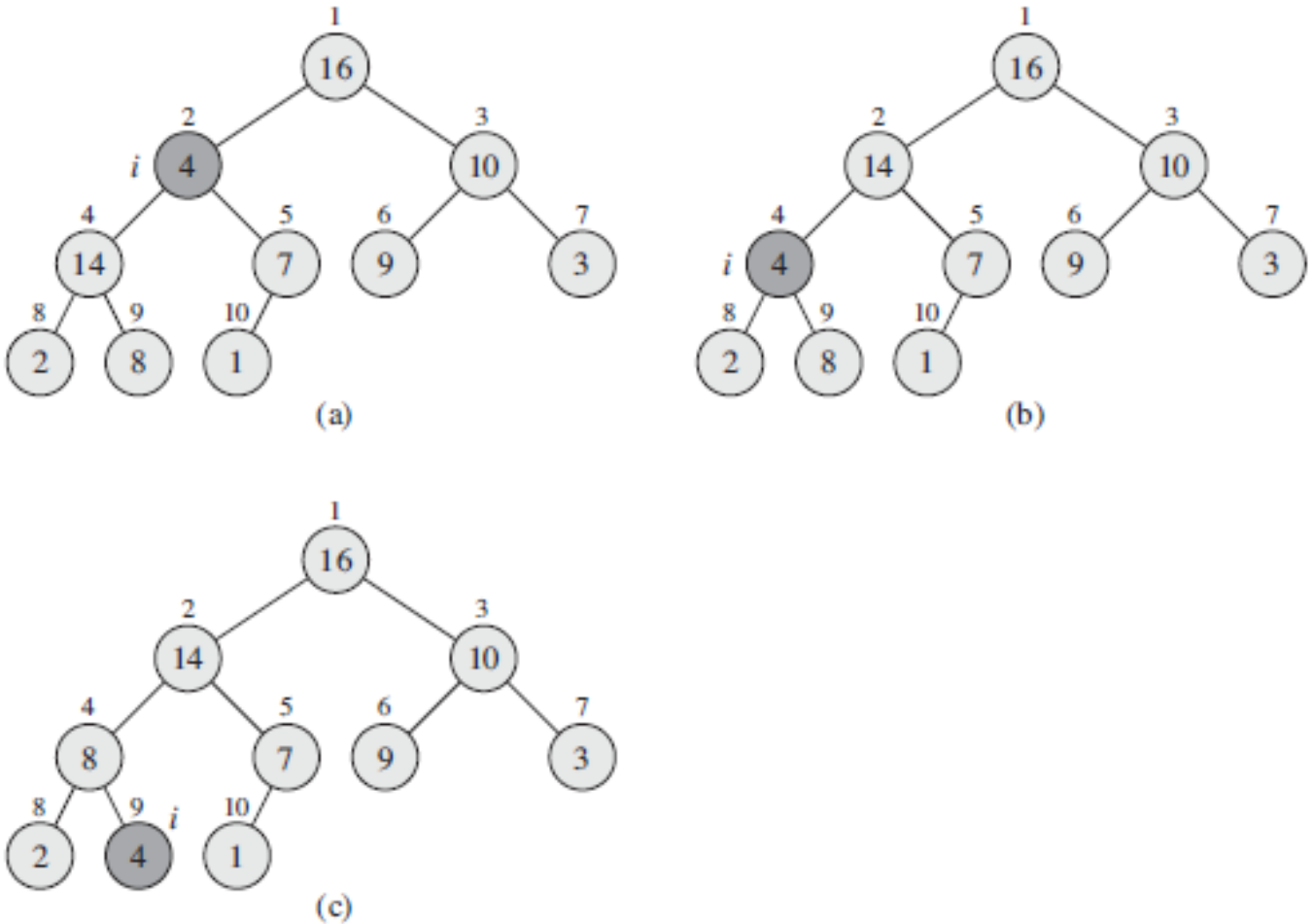


Figure from *Introduction to Algorithms*, 3rd edition, Cormen, Leiserson, Rivest, Stein

Build a heap

A is an array representing a complete binary tree.
A.size is its number of elements.

Algorithm 23 MAX-HEAPIFY(A, i)

```
1: largest = A[i]                    ▷ will contain largest key among i and its children
2: if left(i) > A.size and A[left(i)] > A[i] then
3:     largest = left(i)
4: if right(i) > A.size and A[right(i)] > A[i] then
5:     largest = right(i)
6: if largest ≠ i then
7:     SWAP A[i] and A[largest]
8:     MAX-HEAPIFY(A, LARGEST)
```

It runs in time $O(height(i))$.

Exercise: given an array with n elements, build a heap from it in time $O(n)$.

Extract the maximum

Algorithm 24 EXTRACT-MAX(A)

1: **if** $A.size < 1$ **then**
2: **error**: "heap under flow"
3: $max \leftarrow A[1]$
4: $A[1] \leftarrow A[A.size]$ ▷ put the last element of the heap at the root
5: $A.size \leftarrow A.size - 1$ ▷ decrease the size by 1
6: HEAPIFY($A, 1$) ▷ Bubble down
7: **return** max

It runs in time $O(log(n))$

Insert en element in a heap

Strategy: insert the element at the very end of the tree, then buble-up.

Algorithm 25 INSERT(A, key)

1: $A.size \leftarrow A.size + 1$
2: $A[A.size] \leftarrow -\infty$
3: INCREASE-KEY(A, $A.size$, key)

Algorithm 26 INCREASE-KEY(A, i, key)

1: **if** $A[i] < key$ **then**
2: **error**: "new key is smaller than current key"
3: **while** $i > 1$ and $A[i] > A[parent(i)]$ **do**
4: swap ($A[i]$ and $A[parent(i)]$) ▷ Buble up
5: $i \leftarrow parent(i)$

Runs in time $O(\log(n))$.

Recap

- MAXIMUM(A): $O(1)$
- EXTRACT-MAX(A): $O(\log n)$
- INCREASE-KEY(A, i, key): $O(\log n)$
- INSERT(A, key): $O(\log n)$
- BUILD(A): $O(n)$.

Many ways to implement a priority queue

order-of-growth of running time for priority queue with N items

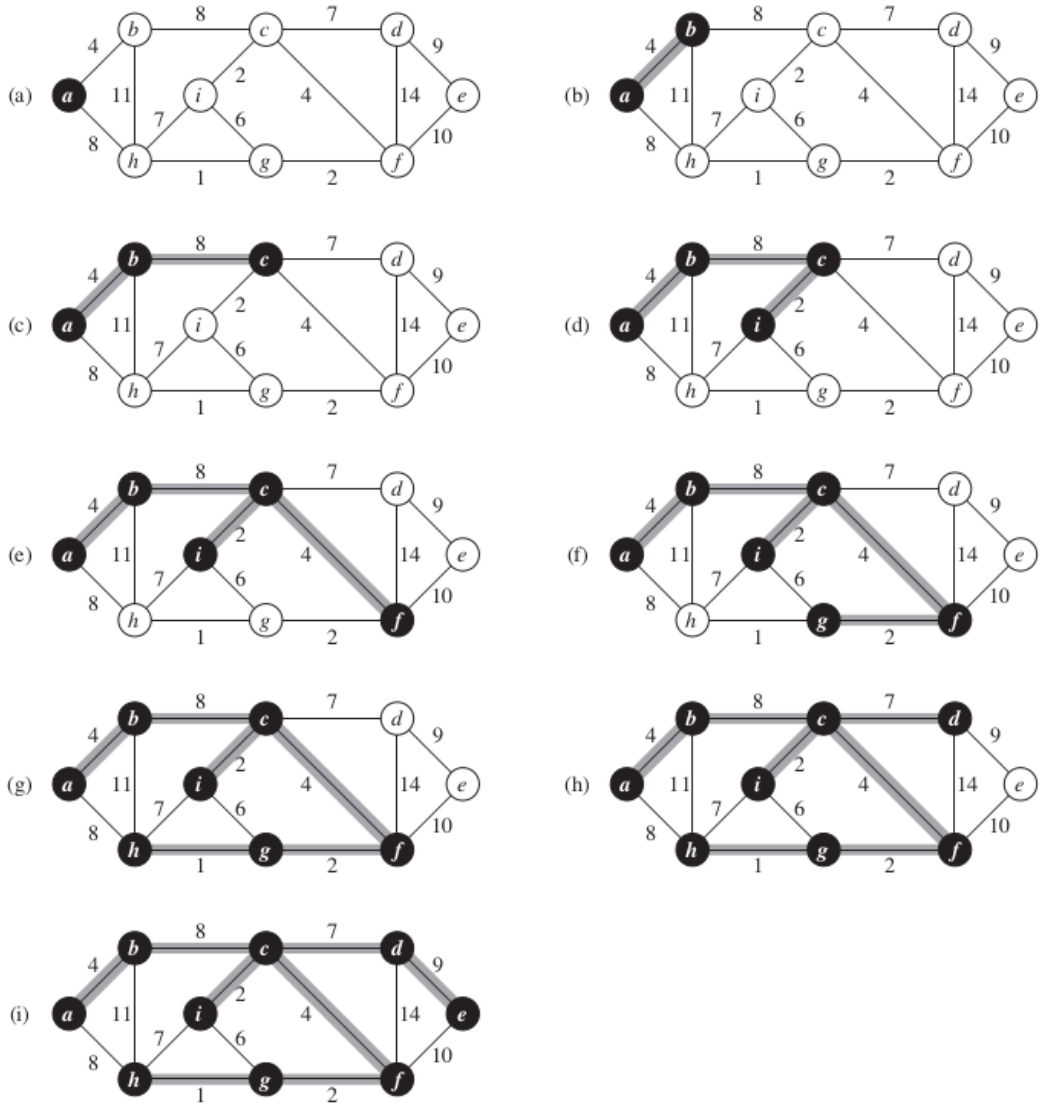
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	log N †	1
Brodal queue	1	log N	1
impossible	1	1	1

← why impossible?

† amortized

figure by Kevin Wayne

Back to Prim's Algorithm



Introduction to Algorithms, 3rd edition, Cormen, Leiserson, Rivest, Stein

Pseudocode of Prim's algorithm

- Q is the priority queue, it contains the set of vertices not yet in the growing tree.
- $V - Q$ is the set of vertices covered by the growing tree
- For each vertex $v \in Q$:
  - $v.key$ is the weight of the lightest edge connecting the growing tree with v.

Pseudocode of Prim's algorithm

- Q is the priority queue, it contains the set of vertices not yet in the growing tree.
- $V - Q$ is the set of vertices covered by the growing tree
- For each vertex $v \in Q$:
  - $v.key$ is the weight of the lightest edge connecting the growing tree with v.

Algorithm 28 MST-PRIM$(G = (V, E, \omega), r)$

```
1:  for each v ∈ V do
2:      v.π = NIL and v.key = +∞
3:  r.key ← 0
4:  Q ← V                                   ▷ Build the priority queue
5:  while Q ≠ ∅ do
6:      u ← EXTRACT-MIN(Q)                  ▷ u is added to the tree
7:      for each v ∈ Adj[u] do              ▷ For each neighbor v of u
8:          if v ∈ Q and v.key > ω(uv) then
9:              v.key ← ω(uv)               ▷ DECREASE-KEY(Q, v, ω(uv))
10:             v.π ← u
```

- The algo maintains $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$

Proof of correctness and running time

Algorithm 29 MST-PRIM($G = (V, E, \omega)$, r)

```
1: for each v ∈ V do
2:     v.π = NIL and v.key = +∞
3: r.key ← 0
4: Q ← V
5: while Q ≠ ∅ do
6:     u ← EXTRACT-MIN(Q)
7:     for each v ∈ Adj[u] do
8:         if v ∈ Q and v.key > ω(uv) then
9:             v.key ← ω(uv)
10:            v.π ← u
```

Proof of correctness: by the *safe property*

Complexity:
- $O(m \log n)$ with a min-heap
- $O(m + n \log n)$ with a Fibonacci heap

Recap

We saw two greedy algorithms to compute MST:

- Kruskal's algorithm:
  - maintains a forest whose trees coalesce into one spanning tree,
  - can be implemented using a disjoint set data structure, runs in $O(m \log n)$.
- Prim's algorithm:
  - Grows a tree (choose the lightest edge that does not create a cycle and maintain the constructing solution connected)
  - Prim's algorithm implemented with a priority queue can run in $O(m + n \log n)$.

4 - Matroid and greedy algorithm

Hereditary set systems and definitions

- Let E be a set of elements and $\mathcal{I} \subseteq 2^E$.
- $(E, \mathcal{I})$ is a set system.
- $(E, \mathcal{I})$ is a hereditary set system if it satisfies:

  (M1) $\emptyset \in \mathcal{I}$

  (M2) If $X \subseteq Y \in \mathcal{I}$, then $X \in \mathcal{I}$ *Hereditary property*

Hereditary set systems and definitions

- Let E be a set of elements and $\mathcal{I} \subseteq 2^E$.
- $(E, \mathcal{I})$ is a set system.
- $(E, \mathcal{I})$ is a hereditary set system if it satisfies:

  (M1) $\emptyset \in \mathcal{I}$

  (M2) If $X \subseteq Y \in \mathcal{I}$, then $X \in \mathcal{I}$ *Hereditary property*
- E is called the ground set.
- Sets in $\mathcal{I}$ are called independent sets.
- Maximal independent sets are called bases.
- For $X \subseteq E$, the rank of X, denoted by $rk(X)$ is the size of a maximum independent set included in X.

$$rk(X) = max\{|Y| \mid Y \subseteq X \text{ and } Y \in \mathcal{I}\}$$

- Sets in $2^E \setminus \mathcal{I}$ are called dependent sets.
- Minimal dependent sets are called circuit.

Picking a heaviest independent set

A positive weight function $c : E \to \mathbb{R}^+$ induced a weight function defined on $\mathcal{I}$:

$$\forall I \in \mathcal{I}, \quad c(I) = \sum_{e \in I} c(e)$$

Problem (Maximization problem for hereditary set system)

Input: *A weighted hereditary set system* $(E, \mathcal{I}, c)$
Task: *Find* $I \in \mathcal{I}$ *such that* $c(I)$ *is maximum.*

- Minimum Spanning Tree: $E = E(G), \quad \mathcal{I} = \{I \subseteq E \mid I \text{ is a forest}\}$.

- TSP: $E = E(G)$, $\mathcal{I} = \{I \subseteq E \mid I \text{ is a subset of edges of a Hamiltonian cycle}\}$.

- Shortest Path Problem:
$E = E(G), \quad \mathcal{I} = \{I \subseteq E \mid I \text{ is a subset of edges of a } s-t\text{-path}\}$.

- Maximum Weighted Matching Problem:
$E = E(G)$ and $\mathcal{I}$ is the set of matching.

- Maximum Weighted Stable Set:
$E = V(G), \quad \mathcal{I} = \{S \subseteq V(G) \mid S \text{ is a stable set}\}$.

- Knapsack Problem:
Given nonnegative numbers c_i, w_i $(1 \leq i \leq n)$ and W, find $S \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in S} w_j \leq W$ and $\sum_{j \in S} c_j$ is maximum.

$E = E(G), \quad \mathcal{I} = \{I \subseteq E \mid \sum_{j \in I} w_j \leq W\}$.

Picking a heaviest independent set

Problem (Maximization problem for hereditary set system)

Input: *A weighted hereditary set system* $(E, \mathcal{I}, c)$
Task: *Find* $I \in \mathcal{I}$ *such that* $c(I)$ *is maximum.*

Greedy algorithm:

- Sort $E = \{e_1, \ldots, e_n\}$ such that $c(e_1) \geq c(e_2) \geq \cdots \geq c(e_n)$
- Set $S = \emptyset$
- For $i = 1, \ldots, n$, if $S \cup \{e_i\} \in \mathcal{I}$, set $S := S \cup e_i$.
- Return S

Picking a heaviest independent set

Problem (Maximization problem for hereditary set system)

Input: *A weighted hereditary set system* $(E, \mathcal{I}, c)$
Task: *Find* $I \in \mathcal{I}$ *such that* $c(I)$ *is maximum.*

Greedy algorithm:

- Sort $E = \{e_1, \ldots, e_n\}$ such that $c(e_1) \geq c(e_2) \geq \cdots \geq c(e_n)$
- Set $S = \emptyset$
- For $i = 1, \ldots, n$, if $S \cup \{e_i\} \in \mathcal{I}$, set $S := S \cup e_i$.
- Return S

Theorem (Rado (1957) - Edmonds (1971))

Greedy algorithm *is optimal* **if and only if** $(E, \mathcal{I})$ *is a matroid*

More formally: Given a hereditary set system $(E, \mathcal{I})$, Greedy algorithm is optimal for every weight function $c : E \to \mathbb{R}^+$ if and only if $(E, \mathcal{I})$ is a matroid.

What on earth is a matroid?

What on earth is a matroid?

A set system $(E, \mathcal{I})$ is a matroid if it satisfies the following three axioms:

(M1) $\emptyset \in \mathcal{I}$;

(M2) If $Y \subseteq X \in \mathcal{I}$, then $Y \in \mathcal{I}$ *Hereditary property*

(M3) If $X, Y \in \mathcal{I}$ and $|X| > |Y|$, then there is $x \in X \setminus Y$ such that $Y \cup x \in \mathcal{I}$ *Exchange property*

Vector matroid and history

• **Vector matroid**: E is the set of collumns of some matrix A over some field, and independent sets are sets of linearly independent columns over F.

This was the first studied matroid (by Hassler Whitney in 1935), who chose the term matroid to say "something like a matrix".

It explains the vocabulary of rank, independent, basis etc

Let $\mathcal{M}$ be a matroid. If there is a matrix A over the field F such that $\mathcal{M}$ is the vector matroid of A over F, then $\mathcal{M}$ is **representable over** F.
There are matroids that are not representable over any field.

Prove that the followings are matroids:

• **Vector matroid**: E is the set of collumns of some matrix A over some field, and independent sets are sets of linearly independent columns.

• **Uniform matroid** $U_{k,n}$: $E = \{1, \ldots, n\}$, $\mathcal{I} = \{X \mid |X| \leq k\}$.

• **Graphic/cycle matroid** $\mathcal{M}(G)$: Let G a graph. $E = E(G)$ and independent sets are forests.

• **Cographic/cocycle matroid** $\mathcal{M}^*(G)$: Let G a graph. $E = E(G)$ and X is independent set if $G \setminus X$ is connected.

• **Disjoint paths matroid**: Let G be a directed graph and $s \in V(G)$. $E = V(G)$ and a set I is independent if there are edge-disjoint paths from s to each vertex in I.

Bases

Lemma: All bases of a matroid have the same size.

Proof: by the Exchange property.

Bases

Lemma: All bases of a matroid have the same size.

Proof: by the Exchange property.

Lemma: Let $(E, \mathcal{I})$ be a matroid and $X \subseteq 2^E$. Then $(X, \{I \subseteq X \mid I \in \mathcal{I}\})$ is a matroid.

Proof: check that the three axioms are satisfied.

Theorem (Rado-Edmonds)

Given a hereditary set system $(E, \mathcal{I})$, GREEDY ALGORITHM *is optimal for every weight function* $c : E \to \mathbb{R}^+$ *if and only if* $(E, \mathcal{I})$ *is a matroid.*

Theorem (Rado-Edmonds)

Given a hereditary set system $(E, \mathcal{I})$, Greedy algorithm is optimal for every weight function $c : E \to \mathbb{R}^+$ if and only if $(E, \mathcal{I})$ is a matroid.

Proof: Let $OPT = \{o_1, \ldots, o_k\}$ be an optimal solution. Note that we may assume that OPT is a basis.
Let $ALG = \{a_1, a_2, \ldots, a_\ell\}$ be the independent set returned by the algorithm. If any other element of E could be added to ALG to obtain a larger independent set, the greedy algorithm would have added it. Thus, ALG is a basis and $k = \ell$.
Assume for contradiction that

$$\sum_{j=1}^{k} c(a_j) < \sum_{i=1}^{k} c(o_i)$$

Now suppose that the elements of ALG and OPT are indexed in order of decreasing weight. Let i be the smallest index such that $c(a_i) < c(o_i)$ and consider

$$ALG_{i-1} = \{a_1, \ldots, a_{i-1}\} \quad \text{and} \quad OPT_i = \{o_1, \ldots, o_{i-1}; o_i\}$$

By the hereditary property, ALG_{i-1} and OPT_i are independent sets. By the exchange property, there is $o_j \in OPT_i$ $(j \leq i)$ such that $o_j \notin ALG_{i-1}$ and $ALG_{i-1} \cup \{o_j\} \in \mathcal{I}$. We have $c(o_j) \geq c(o_i) > c(a_i)$. So the greedy algorithm consider and rejects the heavier element o_j before it considers and accept the lighter element a_i, a contradiction.

Since $(E, \mathcal{I})$ is a hereditary set system that is not a matroid, there exists $X, Y \in \mathcal{I}$ such that $|X| > |Y|$ and for every $x \in X \setminus Y$, $Y \cup \{x\} \notin \mathcal{I}$. Set $m = |Y|$. Define the following weight function:

- For every $y \in Y$, $c(y) = m + 2$,
- For every $x \in X \setminus Y$, $c(x) = m + 1$
- For every $e \in E \setminus (X \cup Y)$, $c(e) = 0$

With these weights, the greedy algorithm will consider and accept every element of Y, then consider and reject every element of X , and finally consider all the other elements. The algorithm returns a set with total weight $m(m+1) = m^2 + m$ while $c(X) \geq (m+1)|X| \geq (m+1)^2 = m^2 + 2m + 1$.

Equivalent of the exchange axiom

Theorem: Let $(E, \mathcal{I})$ be a hereditary set system. Then the following statements are equivalent:

$(M3)$ If $X, Y \in \mathcal{I}$ and $|X| > |Y|$, then there is $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$;

$(M3')$ If $X, Y \in \mathcal{I}$ and $|X| = |Y| + 1$, then there is $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$;

$(M3'')$ For each $X \subseteq E$, all bases of X have the same cardinality.

Proof: exercise

Lower rank and approx

Let $(E, \mathcal{I})$ be a hereditary set system. For $X \subseteq E$, the lower rank $\rho(X)$ of X is the size of a smallest base of X.

$$\rho(X) = min\{|Y| : Y \subseteq X, Y \in \mathcal{I} \text{ and } Y \cup \{x\} \notin \mathcal{I} \text{ for all } x \in X \setminus Y\}$$

Theorem [Jenkyns (1976), Korte and Hausmann (1978)]

Let $(E, \mathcal{I})$ be a hereditary set system. Let ALG be the solution returned by the greedy algorithm and OPT an optimal solution. Then:

$$c(ALG) \geq c(OPT) \cdot q(E, \mathcal{I})$$

Proof: TD