Graph

Dense: |E|≈|V|^2

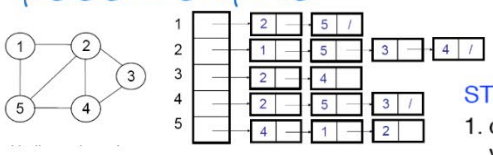
Sparse: |E|≈|V|

Representation

Adjacency matrix (prefered when dense)

- $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$
- Time to determine if (u, v) ∈ E:
  - Θ(1)
- Time to list all vertices adjacent to u:
  - Θ(V)

Adjacency list (prefered when sparse)



- Time to determine if (u, v) ∈ E:
  - O(degree(u))
- Time to list all vertices adjacent to u:
  - Θ(degree(u))

Traversal

BFS O(|V|+|E|)

calculates a shortest-path distance from the source node to all other nodes

```
BFS(G,s) {
    for each u in V {
        color[u] = white
        d[u]     = infinity
        pred[u]  = null
    }
    color[s] = gray
    d[s] = 0
    Q = {s}
    while (Q is nonempty) {
        u = Q.Dequeue()
        for each v in Adj[u] {
            if (color[v] == white) {
                color[v] = gray
                d[v]     = d[u]+1
                pred[v]  = u
                Q.Enqueue(v)
            }
        }
        color[u] = black
    }
}
```

DFS O(|V|+|E|)

```
DFS(G) {
    for each u in V {
        color[u] = white;
        pred[u]  = null;
    }
    time = 0;
    for each u in V
        if (color[u] == white)
            DFSVisit(u);
}

DFSVisit(u) {
    color[u] = gray;
    d[u] = ++time;
    for each v in Adj(u) do
        if (color[v] == white) {
            pred[v] = u;
            DFSVisit(v);
        }
    color[u] = black;
    f[u] = ++time;
}
```

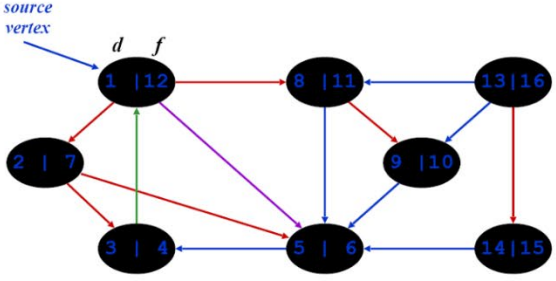
Parenthesis Theorem

Theorem: In any DFS of directed or undirected graph, for any two vertices u and v, one of the following three conditions holds:

1. intervals [d(u), f(u)] and [d(v), f(v)] are disjoint;
2. [d(v), f(v)] entirely inside [d(u), f(u)], i.e. d(u) < d(v) < f(v) < f(u);
3. [d(u), f(u)] entirely inside [d(v), f(v)], i.e. d(v) < d(u) < f(u) < f(v).

White-path Theorem

Theorem: Vertex v is a descendant of u in a DFS tree **if and only if** at time d(u) that u was discovered, vertex v can be reached from u along a path consisting entirely of white vertices.



Tree edges *Back edges* *Forward edges* *Cross edges*

Topological Sort Θ(V+E)

1. call DFS(G) to compute finishing times *f(v)* for each vertex *v*
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Strongly connected components

Kosaraju's algorithm

STRONGLY-CONNECTED-COMPONENTS(G)

1. call DFS(G) to compute finishing times f(u) for each vertex u
2. compute G^T (reversing all edges of G)
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing f(u) (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Dynamic Programming

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0                          Θ(n^2)
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Longest Common Subsequence

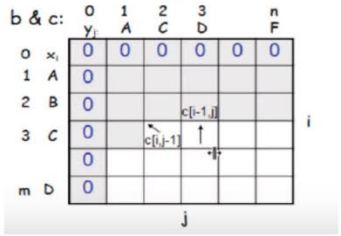
X = ACCGGGTTACCGTTT AAAACCCGGGTAACCT (Size = N)

Y = CCAGGACCAGGGACCGTTTTCCAGCCTTAAACCA (Size = M)

Define:
C[i][j] -- Length of LCS of sequence X[1..i] and Y[1..j]
Thus: C[i][0] == 0 for all i
C[0][j] == 0 for all j
Goal: Find C[N][M]

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\ , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\ , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\ . \end{cases}$$

recnstruct the common subsequence



A matrix b[i,j]
For a subproblem [i,j], tell us what choice was made to obtain the optimal value

```
if xi = yi
    b[i,j]="↖"
else, if c[i-1, j] >= c[i,j-1]
    b[i,j]="↑"
else
    b[i,j]="←"
```

Greedy Algorithm

Activity Selection

Early Finish Greedy

- Select the activity with the earliest finish time
- Eliminate the activities that could not be scheduled
- Repeat!

Greedy-choice property:

show that a greedy choice lead to an optiaml solution

- Show there is an optimal solution that begins with a greedy choice (with activity 1, which as the earliest finish time)
- Suppose A ⊆ S in an optimal solution
  - Order the activities in A by finish time. The first activity in A is k
    - If k = 1, the schedule A begins with a greedy choice
    - If k ≠ 1, show that there is an optimal solution B to S that begins with the greedy choice, activity 1
  - Let B = A – {k} ∪ {1}
    - $f_1 \le f_k$ → activities in B are disjoint (compatible)
    - B has the same number of activities as A
    - Thus, B is optimal

Optimal substructure property

show that an optiaml solution must be achieved by an optiaml sub-solution

Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1

- Optimal Substructure
- If A is optimal to S, then $A' = A - \{1\}$ is optimal to $S' = \{i \in S: s_i \ge f_1\}$
- Why?
  - If we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A
    → contradicting the optimality of A

After each greedy choice is made, we are left with an optimization problem of the same form as the original problem

- By induction on the number of choices made, making the greedy choice at every step produces an optimal solution

Knapsack Problem

fractional

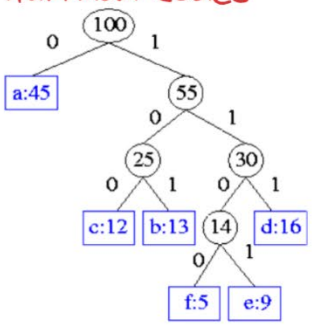
- Compute the value per pound v_i/w_i for each item
- Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
- If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room

0-1 NP-complete

Set of n Items: $\mathcal{S}_n = \{\langle v_1, w_1\rangle, \cdots \langle v_i, w_i\rangle \cdots\cdots \langle v_n, w_n\rangle\}$
Let $\mathcal{B}[k, w]$ be the solution of 0-1 knapsack problem over items from 1 to k in $\mathcal{S}_n$ under weight budget of w

$$\mathcal{B}[k,w] = \begin{cases} \mathcal{B}[k-1,w] & \text{if } w_k > w \\ \max\{\mathcal{B}[k-1,w], B[k-1,w-w_k]+v_k\} & \text{if } w_k \le w \end{cases}$$

Huffman codes



Minimum Spanning Trees

a tree that connects all vertices with minimum weight

Prim's algorithm node-centric

$\Theta(V)$ total:
$Q \leftarrow V$
$key[v] \leftarrow \infty$ for all $v \in V$
$key[s] \leftarrow 0$ for some arbitrary $s \in V$

$|V|$ times:
while $Q \neq \emptyset$
do $u \leftarrow$ EXTRACT-MIN(Q)

$degree(u)$ times:
for each $v \in Adj[u]$
do if $v \in Q$ and $w(u, v) < key[v]$
then $key[v] \leftarrow w(u, v)$
$\pi[v] \leftarrow u$

Handshaking Lemma ⇒ $\Theta(E)$ implicit DECREASE-KEY's.

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Kruskal's algorithm Θ(E·logV) edge-centric

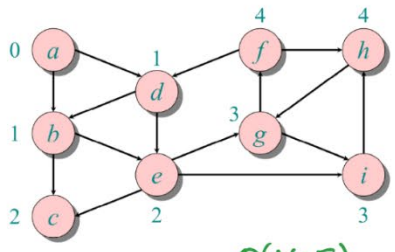
Uses the disjoint-set data structure to avoid loop

```
Kruskal(V, E)

A = ∅
foreach v ∈ V:
   Make-disjoint-set(v)
Sort E by weight increasingly
foreach (v1, v2) ∈ E:
   if Find(v1) ≠ Find(v2):
      A = A ∪ {(v1, v2)}
      Union(v1, v2)
return A
```

Shortest Path

BFS Unweighted graphs



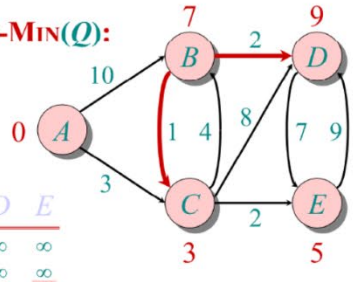
O(V+E)

while $Q \neq \emptyset$
do $u \leftarrow$ DEQUEUE(Q)
for each $v \in Adj[u]$
do if $d[v] = \infty$
then $d[v] \leftarrow d[u] + 1$
ENQUEUE(Q, v)

Dijkstra's algorithm

nonnegative edges

"D" ← EXTRACT-MIN(Q):



Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

S: { A, C, E, B, D }

$|V|$ times:
while $Q \neq \emptyset$
do $u \leftarrow$ EXTRACT-MIN(Q)
$S \leftarrow S \cup \{u\}$

$degree(u)$ times:
for each $v \in Adj[u]$
do if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$

Handshaking Lemma ⇒ $\Theta(E)$ implicit DECREASE-KEY's.

Time = $\Theta(V)\cdot T_{\text{EXTRACT-MIN}} + \Theta(E)\cdot T_{\text{DECREASE-KEY}}$

Q	$T_{\text{EXTRACT-MIN}}$	$T_{\text{DECREASE-KEY}}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap	$O(\lg V)$ amortized	$O(1)$ amortized	$O(E + V \lg V)$ worst case

Bellman-Ford O(V·E)

detect negative-weight cycle

```
d[s] ← 0
for each v ∈ V − {s}                 } initialization
    do d[v] ← ∞

for i ← 1 to |V| − 1
    do for each edge (u, v) ∈ E
        do if d[v] > d[u] + w(u, v)          } relaxation step
              then d[v] ← d[u] + w(u, v)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
          then report that a negative-weight cycle exists
```

in DAG, shortest path well defined

Topological sort + one pass Bellman-Ford

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

Floyd-Warshall Algorithm Θ(V^3)

```
FLOYD-WARSHALL(W)
1  n = W.rows                          All-pairs
2  D^(0) = W
3  for k = 1 to n                      dynamic programming
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
8  return D^(n)
```

Extract Shortest Paths

Predecessor matrices

- Initialization:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty , \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty . \end{cases}$$

- Update:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

- Termination: $pred(i, j) \triangleq \pi_{ij} = \pi_{ij}^{(n)}$

```
Path(i, j)
{
    if (pred(i, j) = i )        single edge
        output (i, j);
    else                        compute the two parts of the path
    {
        Path(i, pred[i, j]);
        Path(pred[i, j], j);
    }
}
```

Loop Invariants

Proving loop invariants works like induction

- Initialization (base case):
  - It is true prior to the first iteration of the loop
- Maintenance (inductive step):
  - If it is true before an iteration of the loop, it remains true before the next iteration
- Termination:
  - When the loop terminates, the invariant gives us a useful property that shows the algorithm is correct
  - Stop the induction when the loop terminates

Transitive Closure

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

Recurrence:

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) .$$

NP Problems

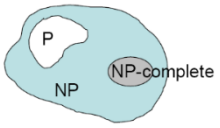
- **Class P** consists of (decision) problems that are solvable in polynomial time
- Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$
- **Nondeterministic algorithm** = two stage procedure:
- Nondeterministic ("guessing") stage:
  - generate randomly an arbitrary string that can be thought of as a candidate solution ("certificate")
- Deterministic ("verification") stage:
  - take the certificate and the instance to the problem and returns YES if the certificate represents a solution
- **NP algorithms (Nondeterministic polynomial)**
  - verification stage is polynomial
- **Class NP** consists of problems that could be solved by NP algorithms
  - i.e., verifiable in polynomial time

NP-complete

- A problem B is **NP-complete** if:

  (1) B ∈ **NP**

  (2) A $\leq_p$ B for all A ∈ **NP**



- If B satisfies only property (2) we say that B is **NP-hard**

Other NP-Complete Problems

- Knapsack
- 3-Partition: given n integers, can you divide them into triples of equal sum?
- Traveling Salesman Problem: shortest path that visits all vertices of a given graph — decision version: is minimum weight ≤ x?
- Longest common subsequence of k strings
- Shortest paths amidst obstacles in 3D

- **3-SAT: NP Complete**
- **2-SAT: P**

If A $\leq_p$ B and B ∈ P, then A ∈ P

If A $\leq_p$ B and A ∉ P, then B ∉ P

If A $\leq_p$ B and A is NP-Complete, B is NP-Hard. In addition, if B ∈ NP ⇒ B is NP-Complete