

Stable Matching.

```

Initialize each person to be free.
while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
}
    
```

Graphs.

Adjacency matrix: Space $O(n^2)$, Generate $O(n^2)$
 Adjacency list: Space $O(m + n)$, Generate $O(m + n)$

Bipartiteness.

Cannot contain odd length cycle
 No edge join two nodes of the same layer (produced by BFS)

Weighted Interval Scheduling. $O(n \log n)$

```

Input: n, s1, ..., sn, f1, ..., fn, v1, ..., vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n)

for j = 1 to n
    M[j] = empty
    M[j] = 0

M-Compute(j) {
    if (M[j] is empty)
        M[j] = max(wj + M-Compute(j)), M-Compute(j-1)
    return M[j]
}
    
```

Knapsack.

		W + 1 →											
		0	1	2	3	4	5	6	7	8	9	10	11
n + 1 ↓	∅	0	0	0	0	0	0	0	0	0	0	0	0
	{1}	0	1	1	1	1	1	1	1	1	1	1	1
	{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
	{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
	{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
	{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	34	40

OPT: {4, 3}
 value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```

Input: n, w1, ..., wn, v1, ..., vn

for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}
return M[n, W]
    
```

Bellman Ford. Time $O(mn)$, Space $O(m + n)$

```

Handles negative path costs
Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← ∅
    }

    M[s] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
            if (M[w] has been updated in previous iteration) {
                foreach node v such that (v, w) ∈ E {
                    if (M[v] > M[w] + c_vw) {
                        M[v] ← M[w] + c_vw
                        successor[v] ← w
                    }
                }
            }
        }
        If no M[w] value changed in iteration i, stop.
    }
}
    
```

List Scheduling. $O(n \log n)$ using priority queue

```

List-Scheduling(m, n, t1, t2, ..., tn) {
    for i = 1 to m {
        Li ← 0           Loads on machine i
        J(i) ← ∅         jobs assigned to machine i
    }
    for j = 1 to n {
        i = argmin_k Lk   i has smallest load
        J(i) ← J(i) ∪ {j} assign job j to machine i
        Li ← Li + tj      Update load of machine i
    }
}
    
```

Centre Selection.

```

Greedy-Center-Selection(k, n, s1, s2, ..., sn) {
    C = ∅
    repeat k times {
        Select a site si with maximum dist(si, C)
        Add si to C
    }
    return C
}
    
```

Vertex cover.

```

Weighted-Vertex-Cover-Approx(G, w) {
    foreach e in E
        pe = 0

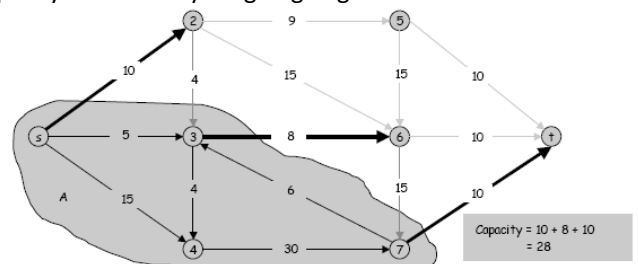
    while (∃ edge i-j such that neither i nor j are tight)
        select such an edge e
        increase pe without violating fairness

    S ← set of all tight nodes
    return S
}
    
```

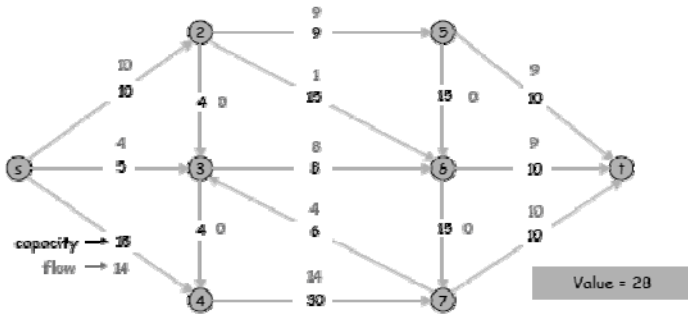
$$\sum_{e \in E} p_e = w_i$$

Flows and Cuts.

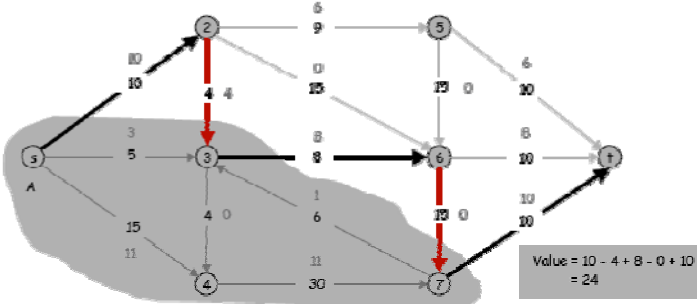
Capacity – count only outgoing edges



Flow – for each edge, use flow \leq capacity. For each node, sum of flow into it is equal to the sum of flow out of it (conservation).



Flow value.



Weak duality. Flow value \leq cut capacity

Ford-Fulkerson for max flow using augmenting paths.

```

Augment(f, c, P) {
  b ← bottleneck(P)
  foreach e ∈ P {
    if (e ∈ E) f(e) ← f(e) + b    forward edge
    else       f(eR) ← f(e) - b    reverse edge
  }
  return f
}
    
```

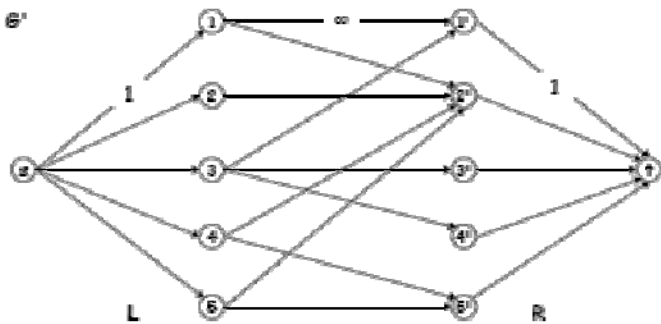
```

Ford-Fulkerson(G, s, t, c) {
  foreach e ∈ E f(e) ← 0
  Gf ← residual graph

  while (there exists augmenting path P) {
    f ← Augment(f, c, P)
    update Gf
  }
  return f
}
    
```

Flow f is a max flow iff there are no augmenting paths.

Bipartite matching and max flow.



Approximation algorithms.

Must sacrifice optimality, poly-time, or arbitrary instances

Prim's algorithm. $O(n^2)$ with array, $O(m \log n)$ with binary heap

```

Prim(G, c) {
  foreach (v ∈ V) a[v] ← ∞
  Initialize an empty priority queue Q
  foreach (v ∈ V) insert v onto Q
  Initialize set of explored nodes S ← ∅

  while (Q is not empty) {
    u ← delete min element from Q
    S ← S ∪ { u }
    foreach (edge e = (u, v) incident to u)
      if ((v ∉ S) and (ce < a[v]))
        decrease priority a[v] to ce
  }
}
    
```

Kruskal's algorithm. $O(m \log n)$ sorting, $O(m \alpha(m, n))$ for U-F

```

Kruskal(G, c) {
  Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
  T ← ∅

  foreach (u ∈ V) make a set containing singleton u

  for i = 1 to m
    (u, v) = ei
    if (u and v are in different sets) {
      T ← T ∪ {ei}
      merge the sets containing u and v
    }
  return T
}
    
```

Dijkstra's algorithm.

For each unexplored node, explicitly maintain

$$\pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$$

Next node to explore = node with minimum $\pi(v)$.

When exploring v, for each incident edge $e = (v, w)$, update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.