

Graphs

- Bipartite \leftrightarrow does not contain an odd-length cycle
 - BFS: no edge joins 2 nodes of same layer
- Strongly connected: $O(m+n)$ time
 - BFS from start, then BFS from end
- Directed acyclic graph – no directed cycles
 - Has node with no incoming edges
 - Has topological ordering

To compute a topological ordering of G :
 Find a node v with no incoming edges and order it first
 Delete v from G
 Recursively compute a topological ordering of $G - \{v\}$
 Append this order after v

Greedy

- Interval scheduling of compatible jobs
 - $n \log n$: Greedy by finish time
- Interval partitioning – min depth schedule
 - $n \log n$: Greedy by start time, depth +1 if not incompatible

Sort intervals by starting time so that $s_1 \leq s_2 \leq \dots \leq s_n$.
 $d = 0$
 for $j = 1$ to n :
 if (lecture j is compatible with some classroom k)
 schedule lecture j in classroom k
 else
 allocate a new classroom $d + 1$
 schedule lecture j in classroom $d + 1$
 $d = d + 1$

- Minimise lateness – earliest deadline first, no idle time
- Caching – optimal offline = farthest in future
 - Reduced schedule = inserts item only when requested
- Shortest path in graph
 - Dijkstra – grow explored nodes with shortest path dist
- Minimum spanning tree
 - Kruskal's – add edges, ascending cost, to T without cycles

Kruskal(G, c):
 Sort edges weights so that $c_1 \leq c_2 \leq \dots \leq c_m$.
 $T = \emptyset$
 foreach ($u \in V$): make a set containing singleton u

 for $i = 1$ to m :
 $(u, v) = e_i$
 if (u and v are in different sets):
 $T = T \cup \{e_i\}$
 merge the sets containing u and v
 return T

- Reverse-delete – Delete edge, descending cost without disconnecting T
- Prim's – add nodes to T , edge has only 1 endpoint in T

Prim(G, c): $O(n^2)$ with array, $O(m \log n)$ binary heap
 foreach ($v \in V$) $a[v] = \infty$
 Initialize an empty priority queue Q
 foreach ($v \in V$) insert v onto Q
 Initialize set of explored nodes $S = \emptyset$

 while (Q is not empty):
 $u =$ delete min element from Q
 $S = S \cup \{u\}$
 foreach (edge $e = (u, v)$ incident to u):
 if ($(v \notin S)$ and ($c_e < a[v]$))
 decrease priority $a[v]$ to c_e

- Cycles and cuts
 - Cycle property – max cost edge in cycle \notin MST
 - Cut property – min cost edge with 1 endpoint in $S \in$ MST

Divide and conquer

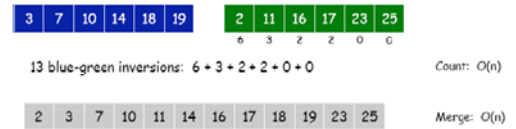
- Mergesort – $O(n \log n)$
 - Proof by telescoping:

$$\begin{aligned} \frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\ &= \frac{T(n/2)}{n/2} + 1 \\ &= \frac{T(n/4)}{n/4} + 1 + 1 \\ &\dots \\ &= \frac{T(n/n)}{n/n} + \underbrace{1 + \dots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

- Proof by induction:

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n(\log_2(2n) - 1) + 2n \\ &= 2n \log_2(2n) \end{aligned}$$

- Counting inversions – $O(n \log n)$; brute force $O(n^2)$ checks all pairs



$$T(n) < T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

- Closest pair of points – halve points, find closest pair of each half

Closest-Pair(p_1, \dots, p_n):
 Compute separation line L such that half the points are on one side and half on the other side. $O(n \log n)$

$\delta_1 =$ Closest-Pair(left half)
 $\delta_2 =$ Closest-Pair(right half)
 $\delta = \min(\delta_1, \delta_2)$

Delete all points further than δ from separation line L
 Sort remaining points by y -coordinate.

Scan points in y -order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ , update δ .

return δ .

- Delete points further than δ from separation line
- Sort points by y -coord, then find min distance between each and next 11 neighbours
- If $\text{dist} < \delta$, update $\delta - O(n \log^2 n)$

Dynamic programming

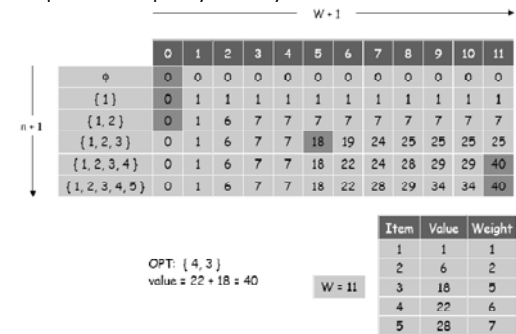
- Weighted interval scheduling (Uses memorisation)
 - Calculate from 1 to n , storing intermediate results

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$
 Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

 Compute $p(1), p(2), \dots, p(n)$
 for $j = 1$ to n :
 $M[j] =$ empty (global array)
 $M[j] = 0$

 M-Comp-Opt(j)
 if ($M[j]$ is empty):
 $M[j] = \max(w_j + M\text{-Comp-Opt}(p(j)), M\text{-Comp-Opt}(j-1))$
 return $M[j]$

- Knapsack – fill up n -by- W array



- For $i = 1$ to n , for $w = 1$ to W
- Maximise total value for up to n objects, up to weight W

Input: $n, w_1, \dots, w_n, v_1, \dots, v_n$
 for $w = 0$ to W :
 $M[0, w] = 0$

 for $i = 1$ to n :
 for $w = 1$ to W :
 if ($w_i > w$):
 $M[i, w] = M[i-1, w]$
 else:
 $M[i, w] = \max\{M[i-1, w], v_i + M[i-1, w-w_i]\}$
 return $M[n, W]$

- Sequence alignment – find alignment of minimum cost
 - For $i \& j$, increment i, j , or both \rightarrow gap, mismatch or match

Sequence-Alignment($m, n, x_1x_2\dots x_m, y_1y_2\dots y_n, \delta, \alpha$):
 for $i = 0$ to m :
 $M[0, i] = i\delta$
 for $j = 0$ to n :
 $M[j, 0] = j\delta$

 for $i = 1$ to m :
 for $j = 1$ to n :
 $M[i, j] = \min(\alpha[x_i, y_j] + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1])$
 return $M[m, n]$

• Shortest path – Bellman-Ford

```

Push-Based-Shortest-Path(G, s, t):
  foreach node v ∈ V:
    M[v] = ∞; successor[v] = φ
    M[t] = 0
  for i = 1 to n-1:
    foreach node w ∈ V:
      if M[w] updated in past iteration:
        for node v s.t. (v, w) ∈ E:
          if M[v] > M[w] + cvw:
            M[v] = M[w] + cvw
            successor[v] = w
    if no M[w] value changed in iteration i, stop.
    
```

- Negative cycles – detect in O(mn)
 - Add new node, cost 0 connect to all others
 - Check OPT(n, v) = OPT(n - 1, v)

Network flow

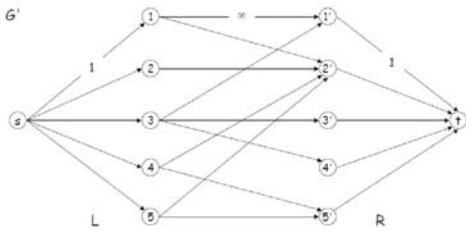
• Ford-Fulkerson augmenting paths max flow algorithm

```

Augment(f, c, P):
  b = bottleneck(P)
  foreach e ∈ P:
    if (e ∈ E) f(e) = f(e) + b
    else f(er) = f(e) - b
  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
    
```

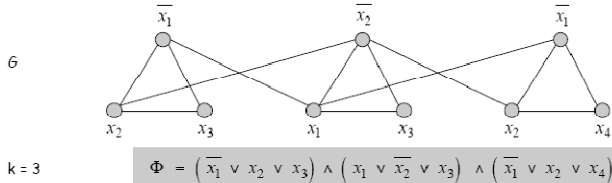
- Theorem: max flow = min cut
- Good augmenting paths – sufficiently large capacity, fewest edges
- Theorem: if capacities are integers, then all flows in max flow integral
- Bipartite matching max flow formulation



- Marriage theorem. Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then, G has a perfect matching iff $|N(S)| \geq |S| \forall$ subsets $S \subseteq L$.
- Disjoint paths (no edges in common)
 - Set all edges to unit capacity and find max flow

NP and Intractability

- Reduction by equivalence: Independent Set = V - Vertex Cover
- Reduction special \rightarrow general: Vertex Cover \leq_p Set Cover
 - Set cover: $k = k; U = E; S_v = \{e \in E \mid e \text{ incident to } v\}$
- Reduction, encoding with gadgets. E.g. 3-SAT \leq_p Independent Set



- NP = \exists poly-time certifier; e.g. Composites, SAT \in NP
- P = NP means efficient algorithms for 3-colour, TSP, factoring, SAT
- Problem Y is NP-complete if $X \leq_p Y \forall X \in$ NP
- Circuit-SAT is first NP-complete problem (set inputs to circuit \rightarrow true)
- To show NP-complete: show that $Y \in$ NP, then $X \leq_p Y$ (X NP-complete)
- Co-NP – complements of decision problems in NP
 - E.g. SAT vs tautology, Ham-Cycle vs No-Ham-Cycle

Dealing with intractability

- Small vertex covers – fix subset size k to be small \rightarrow polynomial
- Independent set on trees

```

Independent-Set-In-A-Forest(F):
  S = φ
  while F has at least one edge:
    Let e = (u, v) be an edge such that v is a leaf
    Add v to S
    Delete from F nodes u and v, and all incident edges
  return S
    
```

Computational geometry

- Art gallery problem: $G(n) = \lfloor n/3 \rfloor, G(n) \leq n - 2$

- Upper bound using triangulation: place a guard in each triangle
- Idea: Minimum colour in 3-colouring of all nodes

Approximation algorithms

• LPT list scheduling

```

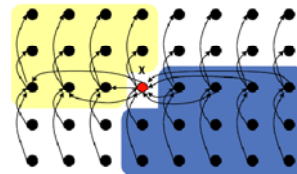
LPT-List-Scheduling(m, n, t1, t2, ..., tn):
  Sort jobs so that t1 ≥ t2 ≥ ... ≥ tn
  for i = 1 to m:
    Li = 0
    J(i) = φ
  for j = 1 to n:
    i = argmink Lk
    J(i) = J(i) ∪ {j}
    Li = Li + tj
    
```

• Pricing method

```

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
    
```

Median algorithm



- Divide elements into groups of 5 elements each
- Find median of the groups by sorting
- Recursively find median of the medians
- Number of steps required is $T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10} + 6) + O(n)$

Randomised quicksort

- Worst case $O(n^2)$, but in practice, $O(n \log n)$

$$\begin{aligned}
 E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
 &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
 &= \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} \\
 &= \sum_{k=2}^n (n+1-k) \frac{2}{k} \\
 &= \left((n+1) \sum_{k=2}^n \frac{2}{k} \right) - 2(n-1) \\
 &= (2n+2) \sum_{k=1}^n \frac{1}{k} - 4n \\
 &= (2n+2)H(n) - 4n \\
 &= 2n \log n + O(n)
 \end{aligned}$$

The proof from Mid-semester

Making use of two properties:

- Cut property: edge e is in every MST when it is the cheapest edge crossing from some set S to the complement $V - S$, and
- Cycle property: edge e is in no MST if it is the most expensive edge on some cycle C,

We can prove that $e = (u, v)$ does not belong to a minimum spanning tree of G if and only if u and v can be joined by a path consisting entirely of edges that are cheaper than e.

First suppose that P is a u-v path consisting entirely of edges cheaper than e. If we add e to P, we get a cycle on which e is the most expensive edge. Thus, by the cycle property, e does not belong to a minimum spanning tree of G.

On the other hand, suppose that u and v cannot be joined by a path consisting entirely of edges cheaper than e, we are able to identify a set S for which e is the cheapest edge with one end in S and the other in $V - S$. By the Cut property e belongs to every minimum spanning tree.