Summary of COMP523 Advanced Algorithm

May 24, 2023

Chapter 1

Symmetry Notation

1.1 Asymptotic Notation

Asymptotic notation is a way of describing the limiting behavior of a function when the argument tends towards a particular value or infinity. In computer science, asymptotic notation is frequently used to describe the running time or space usage of an algorithm.

- O-notation: $f(n) = O(g(n))$ if there exist constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.
- Ω-notation: $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.
- Θ-notation: $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.
- o-notation: $f(n) = o(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
- ω-notation: $f(n) = \omega(g(n))$ if for any constant $c > 0$, there exists a constant n_0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

1.2 Comparing Functions

1.2.1 Transitivity

- $f(n) = O(g(n))$ and $g(n) = O(h(n))$ implies $f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ implies $f(n) = \Omega(h(n))$.
- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ implies $f(n) = \Theta(h(n))$.

For example, $n^2 = O(n^3)$ and $n^3 = O(n^4)$ implies $n^2 = O(n^4)$.

1.2.2 Reflexivity

- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.
- $f(n) = \Theta(f(n))$.

For example, $n^2 = O(n^2)$.

1.2.3 Symmetry

- $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
- $f(n) = \Omega(g(n))$ implies $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ implies $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ implies $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ implies $n^3 = \Omega(n^2)$.

1.2.4 Transpose Symmetry

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.
- $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.

For example, $n^2 = O(n^3)$ if and only if $n^3 = \Omega(n^2)$.

1.2.5 sum and maximum

$$f_1(n) + f_2(n) + \cdots + f_k(n) = \Theta(\max(f_1(n), f_2(n), \ldots, f_k(n)))$$

where k is a constant positive integer.
Let $f_j(n) = j$, $k = n$, then

$$f_1(n) + f_2(n) + \cdots + f_k(n) = n(n+1)/2 = \Theta(n^2)$$

1.2.6 Running time hierarchy

- logarithmic: $O(\log n)$
- linear: $O(n)$
- $n \log n$: $O(n \log n)$
- quadratic: $O(n^2)$
- polynomial: $O(n^k)$
- exponential: $O(c^n)$
- constant: $O(1)$
- superconstant: $\omega(1)$
- sublinear: $o(n)$
- superlinear: $\omega(n)$
- superpolynomial: $\omega(n^k)$
- subexponential: $o(c^n)$

1.3 Expect of algorithms

Correctness: An algorithm is correct if it halts with the correct output for every input instance.
Termination: An algorithm is terminating if it halts for every input instance.
Efficiency: An algorithm is efficient if it halts with the correct output for every input instance and runs in polynomial time.

Chapter 2

Recursion and Divide and Conquer techniques

2.1 Finding Majority in array

The pesudocode of the algorithm is shown in Algorithm 2.1.

Algorithm 1 Finding Majority in array

```
1:  procedure MAJORITY(A)
2:      n ← length of A
3:      if n = 0 then
4:          return −1
5:      end if
6:      if n = 1 then
7:          return A[1]
8:      end if
9:      if n1 and n is odd then
10:
11:     end if
12:     Array B of size n/2
13:     set j=0
14:     for i = 1 to n/2 do
15:         if A[2i − 1] = A[2i] then
16:             B[j] ← A[2i − 1]
17:             j ← j + 1
18:         end if
19:     end for
20:     m ← MAJORITY(B)
21:     count ← 0
22:     for i = 1 to n do
23:         if A[i] = m then
24:             count ← count + 1
25:         end if
26:     end for
27:     if count > n/2 then
28:         return m
29:     else
30:         return −1
31:     end if
32: end procedure
```

Correctness:
Lemma: If A has a majority element, then the majority element of A is also the majority element of B.
Base case: $n = 1$, the majority element is $A[1]$.
Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.
Induction step: If A has a majority element, then the majority element of A is also the majority element of B.
Case 1 (A has a majority element m): Then by the lemma, it is also the majority element of B. Then m appears more than $k/2$ times in B. Then m appears more than $(k + 1)/2$ times in A.
Case 2 (A has no majority element): Then B has no majority element. Then A has no majority element.
Proof the lemma:
proof by contradiction. Assume that A has a majority element m and B has a majority element m', but $m \neq m'$.
Let x be the numbers of occurrence of m in A.
Let y be the numbers of occurrence of m' in B.
Then $2y$ times from pairs that are represented in B by a value different from m', and $x - 2y$ times, since each occurrence of m in A that is not paired with another occurrence of m in A is paired with an occurrence of m' in B.
In total, this gives $2y + x - 2y = x$ occurrences of m in A, which is a contradiction.
Running time:
Recursive formula for the running time:

$$T(n) \leq T(n/2) + cn$$

where c is a constant.
The solution to the recurrence is $T(n) = O(n)$.

2.2 Searching in logarithmic time

Searching faster with BinarySearch.
It is a particular case of the divide-and-conquer paradigm.
Input: A sorted array A of n elements and a value x.
Output: An index i such that $A[i] = x$ or the special value -1 if x does not appear in A.
Pseudocode is shown in Algorithm 2.2.

Algorithm 2 BinarySearch

```
1:  procedure BINARYSEARCH(x, i, j)
2:      if i = j then
3:          if A[i] = x then
4:              return i
5:          else
6:              return −1
7:          end if
8:      else
9:          if x = A[⌊(i + j)/2⌋] then
10:             return ⌊(i + j)/2⌋
11:         else if x < A[⌊(i + j)/2⌋] then
12:             return BINARYSEARCH(x, i, ⌊(i + j)/2⌋)
13:         else
14:             return BINARYSEARCH(x, ⌊(i + j)/2⌋ + 1, j)
15:         end if
16:     end if
17: end procedure
```

Running time:
The number of comparisons performed by BinarySearch is:

$$T(n) \leq T(n/2) + 4$$

Keep calculate:

$$\begin{aligned}
T(n) &\leq T(n/2) + 4 \\
&\leq T(n/4) + 4 + 4 \\
&\leq T(n/8) + 4 + 4 + 4 \\
&\leq T(n/2^k) + 4k \\
&\leq T(n/2^{\log(n-1)}) + 4\log(n-1) \\
&= T(2) + 4(\log n - 1) \\
&\leq 4\log n - 4 \\
&= 4\log n
\end{aligned}$$

proof $T(n) \leq 4\log n$:
Base case: $n = 1, T(1) = 0 \leq 4\log 1 = 0$.
Induction hypothesis: Assume that the lemma is true for $n = k$, we will prove that the lemma is true for $n = k + 1$.
Induction step: $T(k+1) \leq 4\log(k+1)$.

$$\begin{aligned}
T(k+1) &\leq T(k/2) + 4 \\
&\leq 4\log(k/2) + 4 \\
&= 4\log k - 4 + 4 \\
&= 4\log k \\
&\leq 4\log(k+1)
\end{aligned}$$

Memory usage:
The memory usage of BinarySearch is:

$$M(n) = O(\log n)$$

Comparing BinarySearch and LinearSearch:

$$\begin{aligned}
T_{\text{BinarySearch}}(n) &= O(\log n) \\
T_{\text{LinearSearch}}(n) &= O(n) \\
T_{\text{BinarySearch}}(n) = O(\log n) &< O(n) = T_{\text{LinearSearch}}(n) \\
M_{\text{BinarySearch}}(n) = O(\log n) &< O(1) = M_{\text{LinearSearch}}(n)
\end{aligned}$$

2.3 Running time of Divide and Conquer algorithms

The Master Theorem:
Suppose that $T(n)$ satisfies the recurrence:

$$T(n) \leq aT(n/b) + cn^d$$

where $a \geq 1$, $b > 1$, $c > 0$ and $d \geq 0$ are constants.
Then $T(n)$ has the following asymptotic bounds:

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

This theorem is useful for solving recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ and $f(n)$ is an asymptotically positive function.
Example:

$$T(n) = 8T(n/2) + 100n^2$$

$a = 8$, $b = 2$, $f(n) = 100n^2$, $d = 2$, $\log_b a = \log_2 8 = 3$.
$d = 2 < \log_b a = 3$, so $T(n) = O(n^{\log_b a}) = O(n^3)$.

2.4 Finding piar of points closest to each other

Input: A set P of n points in the plane.
Output: The pair of points in P that are closest to each other.
Pseudocode is shown in Algorithm 2.4. **Running time**:

Algorithm 3 ClosestPair

```
1: procedure ClosestPair(P_1, ..., P_n)
2:     Construct P_x and P_y. P_x is sorted by x-coordinate, P_y is sorted by y-coordinate.
3:     return ClosestPairRec(P_x, P_y)
4: end procedure
```

Algorithm 4 ClosestPairRec

```
1: procedure ClosestPairRec(P_x, P_y)
2:     if |P_x| = |P_y| ≤ 3 then
3:         For each pair of points (P_i, P_j), compute d(P_i, P_j)
4:         return the pair of points with the smallest distance
5:     end if
6:     Construct Q_x, Q_y, R_x and R_y.
7:     (l_1, l_2) = ClosestPairRec(Q_x, Q_y)
8:     (r_1, r_2) = ClosestPairRec(R_x, R_y)
9:     δ = min{d(l_1, l_2), d(r_1, r_2)}
10:    x* = the largest x-coordinate in Q_x
11:    L = (x, y) : x = x*
12:    S = {p ∈ P : p ∈ L and p is within δ of L}
13:    Construct S_v
14:    for p ∈ S do
15:        Let q be the point in S_v closest to p
16:        if d(p, q) < δ then
17:            δ = d(p, q)
18:            (s_1, s_2) = (p, q)
19:        end if
20:    end for
21:    if d(s_1, s_2) < min{d(l_1, l_2), d(r_1, r_2)} then
22:        return (s_1, s_2)
23:    end if
24:    if d(l_1, l_2) < d(r_1, r_2) then
25:        return (l_1, l_2)
26:    else
27:        return (r_1, r_2)
28:    end if
29: end procedure
```

$T(n) \leq 2T(n/2) + O(n \log n) = O(n \log n)$
Example:

Chapter 3

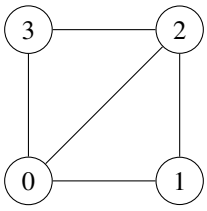
Graph Algorithms

3.1 Graph Definitions

Graph: A graph G consists of a set V of vertices and a set E of edges, where each edge is associated with a pair of vertices.
Directed Graph: A directed graph G consists of a set V of vertices and a set E of directed edges, where each directed edge is associated with an ordered pair of vertices.
Undirected Graph: An undirected graph G consists of a set V of vertices and a set E of undirected edges, where each undirected edge is associated with an unordered pair of vertices.
Neighbours of a vertex v: Set of vertices that are connected to v by an edge.
Degree of a vertex v: number of neighbours of v, denoted by $deg(v)$.
Path: A sequence of (non-repeating) nodes with consecutive nodes being connected by an edge.
length = node count - 1 = edge count.
Distance between two nodes: The number of edges in the shortest path between the two nodes.
Graph diameter: The maximum distance between any two nodes in the graph.

Lines,cycles,trees and cliques:
Line: A graph with n vertices and $n-1$ edges.
Cycle: A graph with n vertices and n edges.
cliques: A graph with n vertices and $n(n-1)/2$ edges.
Tree: A graph with n vertices and $n-1$ edges.

Graph representations:
Adjacency matrix: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge between i and j, and $A_{ij} = 0$ otherwise.
examples of adjacency matrices:
Given the following graph:

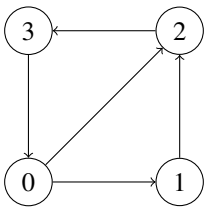


The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency matrix for directed graphs: A $n \times n$ matrix A where $A_{ij} = 1$ if there is an edge from i to j, and $A_{ij} = 0$ otherwise.

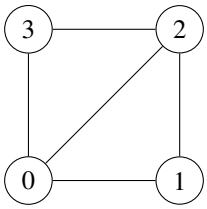
examples of adjacency matrices for directed graphs:
Given the following graph:



The adjacency matrix is:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

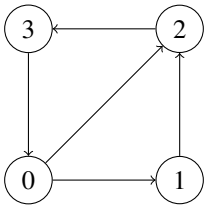
Adjacency list: A list of lists, where the ith list contains the neighbours of vertex i.
Given the following graph:



The adjacency list is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & \\ 0 & 1 & 3 \\ 0 & 2 & \end{bmatrix}$$

Adjacency list for directed graphs: A list of lists, where the ith list contains the neighbours of vertex i.
Given the following graph:



The adjacency list is:

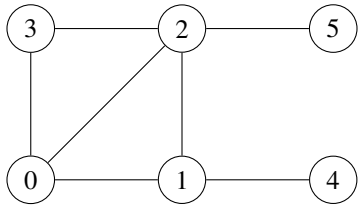
$$\begin{bmatrix} 1 & 2 \\ 2 & \\ 3 & \\ 0 & \end{bmatrix}$$

Adjacency matrix vs adjacency list:

Adjacency matrix	Adjacency list
$O(1)$ to check if there is an edge between i and j	$O(min(deg(i), deg(j)))$ to check if there is an edge between i and j
$O(n)$ to find the neighbours of i	$O(deg(j))$ to find the neighbours of i
$O(n^2)$ space	$O(n + m)$ space

3.2 Depth-first search

Depth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.
example of depth-first search:



The depth-first search sequence is:

$$0, 1, 2, 3, 5, 4$$

Depth-first search algorithm:

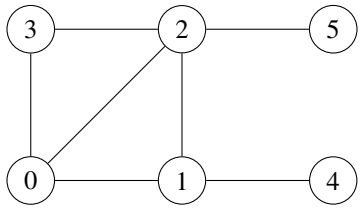
Algorithm 5 Depth-first search algorithm

```
1:  procedure DFS(G, v)
2:      for e ∈ V do
3:          if e is unexplored then
4:              u = head of e
5:              if u is unexplored then
6:                  e is a tree edge
7:                  DFS(G, u)
8:              else
9:                  e is a back edge
10:             end if
11:         end if
12:     end for
13: end procedure
```

Running time of depth-first search: $O(n + m)$

3.3 Breadth-first search

Breadth-first search: A graph search algorithm that explores the neighbours of a vertex before exploring the neighbours of its neighbours.
exaqmple of breadth-first search:



The breadth-first search sequence starting from vertex 0 is 0, 1, 2, 3, 4, 5.
Breadth-first search algorithm:

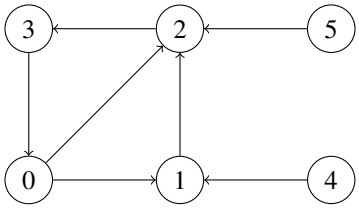
Algorithm 6 Breadth-first search algorithm

```
1:  procedure BFS(G, s)
2:      initial empty list L
3:      L ← s
4:      i ← 0
5:      while L[i] ≠ ∅ do
6:          L_{i+1} ← emptylist
7:          for v ∈ L[i] do
8:              for edges (e) incident to v do
9:                  if e is unexplored then
10:                     w ← the other end of e
11:                     if w is unexplored then
12:                         label e as a tree edge
13:                         add w to L_{i+1}
14:                     else
15:                         label e as a cross edge
16:                     end if
17:                 end if
18:             end for
19:         end for
20:         i ← i + 1
21:     end while
22: end procedure
```

Running time of breadth-first search: $O(n + m)$

3.4 Strong Connectivity

Directed graph: A graph where the edges have a direction.
Examples:



DFS and BFS on directed graphs:
Very similar to undirected graphs, except that we only consider edges that go out of a vertex.
Running time is $O(n + m)$
For example graph above the DFS sequence is 0, 1, 2, 3.
The BFS sequence is 0, 1, 2, 3.

3.4.1 Connectivity

Weak connectivity:If we ignore the direction for all edges, there would be a pah from any vertex to any other vertex.
Strong Connectivity:For every two nodes u and v, there is a path from u to v and a path from v to u.

3.4.2 Mutual Reachability

Two nodes u and v are mutually reachable if there is a path from u to v and a path from v to u.
Strong connectivity:For every pair of nodes u and v, these two nodes are mutually reachable.
Transitivity:If u is mutually reachable with v and v is mutually reachable with w, then u is mutually reachable with w.

3.4.3 Testing strong connectivity

Algorithm 7 Testing strong connectivity

```
1:  procedure TESTSTRONGCONNECTIVITY(G)
2:      define G^R to be the graph with the same vertices as G but with all edges reversed
3:      Select a node s in G
4:      BFS(G, s),BFS(G^R, s)
5:      for each node v do
6:          if v is unexplored in either BFS then
7:              return False
8:          end if
9:      end for
10:     return True
11: end procedure
```

3.5 Testing bipartiteness

Bipartite graph: A graph $G = (V, E)$ is bipartite if any only if the vertices can be partitioned into two sets V_1 and V_2 such that every edge has one end in V_1 and the other end in V_2.
A Graph $G = (V, E)$ is bipartite if and only if it has no odd cycles.(odd cycle: a cycle with odd number of edges)
Testing bipartiteness:
Given a graph $G = (V, E)$, we want to test if G is bipartite.
Given a graph $G = (V, E)$, decide if it is 2-colourable.
Given a graph $G = (V, E)$, decide if it has an odd cycle.
Colouring the nodes It is quite familiar with BFS:

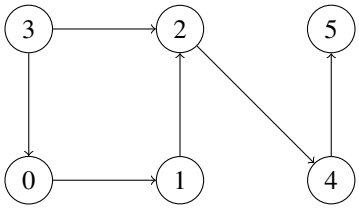
Algorithm 8 Colouring the nodes

```
1: procedure COLOURING(G, s)
2:     initial empty list L
3:     initial empty list C
4:     L ← s
5:     C[s] ← red
6:     i ← 0
7:     while L[i] ≠ ∅ do
8:         L_{i+1} ← emptylist
9:         for v ∈ L[i] do
10:            for edges (e) incident to v do
11:                if e is unexplored then
12:                    w ← the other end of e
13:                    if w is unexplored then
14:                        label e as a tree edge
15:                        add w to L_{i+1}
16:                        if i + 1 is odd then
17:                            C[w] ← green
18:                        else
19:                            C[w] ← red
20:                        end if
21:                    else
22:                        label e as a cross edge
23:                        if C[v] = C[w] then
24:                            return False
25:                        end if
26:                    end if
27:                end if
28:            end for
29:        end for
30:        i ← i + 1
31:    end while
32:    for e(v, w) ∈ G do
33:        if C[v] = C[w] then
34:            return False
35:        end if
36:    end for
37:    return True
38: end procedure
```

Running time of colouring the nodes: $O(n+m)$
Correctness of colouring the nodes:
Proof by contradiction.
Suppose that G is not bipartite.
Then G has an odd cycle.
Suppose to the contrary that the algorithm return True.
That means that the algorithm did not detect the odd cycle.

3.6 DAGs and Topological Ordering

DAG: A directed acyclic graph (DAG) is a directed graph with no directed cycles.
examples of DAGs:



Topological ordering:Given a graph$G = (V, E)$, a topological ordering of G is an ordering of the nodes $u_1, u_2, \ldots, u_n$ such that for every edge (u_i, u_j), we have $i < j$.
Intutively, a topological ordering is an ordering of the nodes such that every edge goes from left to right.
example of topological ordering based on given graph above:

$$3, 0, 1, 2, 4, 5$$

Topological ordering implies DAG:

- If G has a topological ordering, then G is a DAG.
- Suppose by contradiction that G has a topological ordering $u_1, u_2, \ldots, u_n$ but G also has a cycle C.
- Let u_j be the smallest element of C in the topological ordering.
- Let u_i be its predecessor in C.
- u_i must appear before u_j in the topological ordering.
- This contradicts the fact that u_j is the smallest element of C in the topological ordering.

DAG implies topological ordering:
Proof by induction: Base case: If G has one or two nodes, then G has a topological ordering.
Induction steps: Assume that a DAG up to k nodes has a topological ordering(induction hypothesis). we will prove that a DAG with $k + 1$ nodes has a topological ordering.

- By our lemma, there is at least one source node in G, and let u be the node.
- Put u at the beginning of the topological ordering.
- Consider the graph G', obtained by G by removing u and its incident edges.
- G' is a DAG with k nodes.
- It has a topological ordering $u_1, u_2, \ldots, u_k$ by the induction hypothesis.
- Append this ordering to u to get a topological ordering of G.

Here is the algorithm:

Algorithm 9 Topological Sorting

```
1: procedure TOPOLOGICALSORTING(G)
2:     find a source vertex u
3:     set u as the first element of the topological ordering
4:     G' ← G with u and its incident edges removed
5:     L ← TOPOLOGICALSORTING(G')
6:     append L to u
7: end procedure
```

Running time of the algorithm is $O(n^2)$
Modified Topological Sorting:
Running time of the algorithm is $O(n + m)$

Algorithm 10 Modified Topological Sorting

```
1: procedure MODIFIEDTOPOLOGICALSORTING(G)
2:     L ← emptylist
3:     S ← set of all source vertices
4:     while S ≠ ∅ do
5:         remove a vertex u from S
6:         append u to L
7:         for each edge (u, v) do
8:             remove edge (u, v) from G
9:             if v is a source vertex then
10:                add v to S
11:            end if
12:        end for
13:    end while
14:    if G has edges then
15:        return G has a cycle
16:    else
17:        return L
18:    end if
19: end procedure
```

3.7 Finding strongly connected components

connected components: A connected component of an undirected graph is subgraph of the graph where any two nodes are connected by a path.
strongly connected components: A strongly connected component of a directed graph is a subgraph of the graph where any two nodes are mutually reachable.(mutually reachable: there is a path from u to v and a path from v to u)
Finding strongly connected components:
Kosaraju's algorithm:

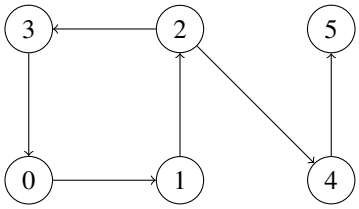
Algorithm 11 Kosaraju's algorithm

```
1: procedure KOSARAJU(G)
2:     Initialise stack S
3:     Select a arbitrary node s
4:     DFS_tree=DFS(G, s)
5:     S ← nodes in DFS_tree
6:     G^R ← nodes in order of S
7:     DFS(G^R, s)
8:     return the nodes in the DFS tree
9: end procedure
```

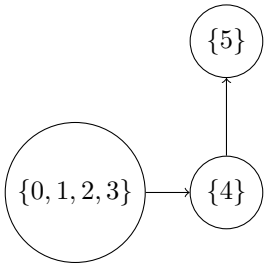
Running time of Kosaraju's algorithm: $O(n+m)$
Correctness of Kosaraju's algorithm:

- Define a meta-graph of G, called $G^{SCC} = (V^{SCC}, E^{SCC})$.
- Supposed that G has strongly connected components (SCCs) $C_1, C_2, \ldots, C_k$, for some k.
- $V^{SCC} = \{C_1, C_2, \ldots, C_k\}$ contains some of the SCCs of G.
- There is an edge (C_i, C_j) in E^{SCC} if G contains a directed edge (x, y) such that $x \in C_i$ and $y \in C_j$, crossing different components.

Examples:



The SCCs are $\{0, 1, 2, 3\}$ and $\{4, 5\}$.
The meta-graph is:



Chapter 4

Greedy Algorithms

The greedy approach:

- The goal is to find a global solution to a problem.
- The solution will be built up in small consecutive steps.
- For each step, we choose the best option available to us at that moment.

4.1 Interval Scheduling

Interval Scheduling:
A set of requests$R = \{1, 2, \ldots, n\}$.

- Each request i has a start time s_i and a finish time f_i.
- Alternative view: every request is an interval $[s_i, f_i]$.

Two requests i and j are compatible if $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap.
Goal: Find a maximum-size subset of compatible requests.
Example:

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.

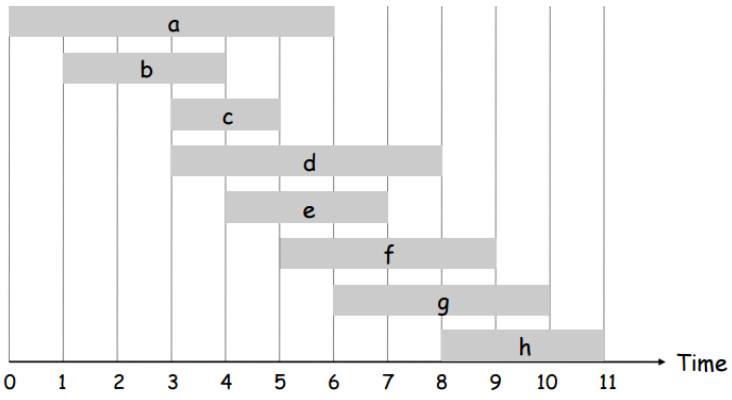


Figure 4.1: Interval Scheduling

Interval Scheduling Algorithm:

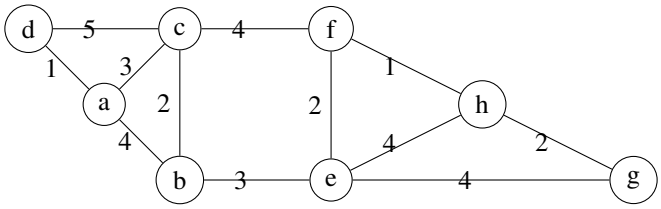
Algorithm 12 Interval Scheduling Algorithm

```
1: procedure IntervalScheduling([s_1, f_1], [s_2, f_2], ..., [s_n, f_n])
2:     R is the set of requests
3:     A ← ∅
4:     while R ≠ ∅ do
5:         select a request i in R with the smallest finishing time
6:         add i to A
7:         remove all requests from R that are incompatible with i
8:     end while
9:     return A
10: end procedure
```

Running time of Interval Scheduling Algorithm: $O(n \log n)$
Correctness of Interval Scheduling Algorithm: Since the algorithm always selects the request with the smallest finishing time, it is clear that the algorithm will always select a compatible request.
Arguing optimality:

4.2 Minimum Spanning Trees

Consider a connected graph $G = (V, E)$, such that each edge $e = (v, w)$ of E, there is an associated cost c_e.
Goal: Find a spanning tree T of E so that the graph $G' = (V, T)$ has minimum cost.
Example:



Greedy approach 1:

- Start with an empty set of edges T.
- Repeat until T forms a spanning tree:
  - Select an edge e of minimum cost.
  - If $T \cup \{e\}$ does not contain a cycle, then add e to T.

krukals algorithm:

Algorithm 13 Krukals algorithm

```
1: procedure Krukals(G)
2:     T ← ∅
3:     while T is not a spanning tree do
4:         select an edge e of minimum cost
5:         if T ∪ {e} does not contain a cycle then
6:             add e to T
7:         end if
8:     end while
9:     return T
10: end procedure
```

Running time of Krukals algorithm: $O(m \log n)$
Greedy approach 2:

- Start with an empty set of edges T.
- Start with a node s.
  - Add an edge $e = (s, v)$ of minimum cost to T.
- Repeat until T forms a spanning tree:

Prims algorithm:

Algorithm 14 Prims algorithm

```
1: procedure PRIMS(G)
2:     T ← ∅
3:     s ← an arbitrary node
4:     while T is not a spanning tree do
5:         add an edge e = (s, v) of minimum cost to T
6:         s ← v
7:     end while
8:     return T
9: end procedure
```

Running time of Prims algorithm: $O(m \log n)$
minimum spanning tree of example graph:
the minimum spanning tree sequence is d, a, c, b, e, f, h, g.

Greedy approach 3:

- Start with the full graph $G = (V, E)$.
- Delete an edge from G
  - the edge of maximum cost
- Repeat until G forms a spanning tree:

Reverse-delete algorithm:

Algorithm 15 Reverse-delete algorithm

```
1: procedure REVERSEDELETE(G)
2:     T ← G
3:     while T is not a spanning tree do
4:         delete an edge e of maximum cost from T
5:     end while
6:     return T
7: end procedure
```

For when two edges have the same cost, use distinct labels to distinguish them.

Optimal with Priorty Queue:
Add PQ to Prim's algorithm.

Algorithm 16 Optimal with Priorty Queue

```
1: procedure OPTIMAL(G)
2:     T ← ∅
3:     s ← an arbitrary node
4:     PQ ← empty priority queue
5:     for each node v do
6:         add v to PQ with key ∞
7:     end for
8:     decrease key of s to 0
9:     while PQ is not empty do
10:        v ← node with minimum key in PQ
11:        add an edge e = (s, v) of minimum cost to T
12:        s ← v
13:        for each edge e = (v, w) incident to v do
14:            if w is in PQ then
15:                decrease key of w to c_e
16:            end if
17:        end for
18:    end while
19:    return T
20: end procedure
```

Running time of Optimal with Priorty Queue: $O(m \log n)$

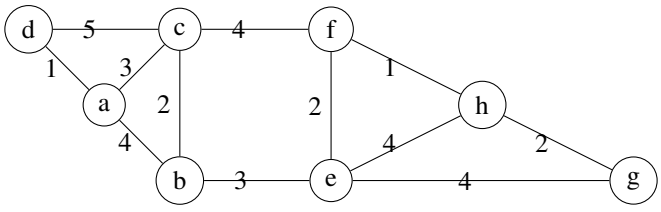
4.3 Clustering

- a collection of n objects
- they have different degrees of similarity
- we want to organise them into coherent groups
- there is a notion of distance between objects

Definition:

- Given a set U of n elements, a $k-$clustering of U is a partition of U into non-empty subsets $C_1, C_2, \ldots, C_k$.
- The spacing of a $k-$clustering is the minimum distance between any pair of points in different clusters.

Goal:Among all possible $k-$clusterings, find one with minimum spacing.
Example:



Greedy approach:

- Pick two objects p_i and p_j with minimum distance $d(p_i, p_j)$.
- Connect them with an edge $e = (p_i, p_j)$.

- Continue like this until we have k clusters.
- If the edge e under consideration connects two object p_i and p_j already in the same cluster, then discard e.

kruskals algorithm:

Algorithm 17 kruskals algorithm for clustering

Require: A graph $G = (V, E)$
Ensure: A minimum spanning tree of G with k clusters

```
1:  procedure KRUSKAL(G, k)
2:      T ← ∅
3:      C ← {{v} | v ∈ V}                                   ▷ Initial clusters
4:      Sort edges in E in increasing order of weight
5:      for {u, v} ∈ E do
6:          if C contains k clusters then
7:              break
8:          end if
9:          if clusters containing u and v are different in C then
10:             T ← T ∪ {{u, v}}
11:             merge clusters containing u and v in C
12:         end if
13:     end for
14:     return T
15: end procedure
```

For Given example, the result of divide them into 3 clusters is:

$$\{a, b, c, d\}, \{e, f, h\}, \{g\}$$

Chapter 5

Dynamic Programming

The paradigm of dynamic programming: Given a problem P, define a sequence of subproblems, with the following properties:

- The subproblems are ordered from the simplest to the largest
- The largest problem is our original problem P
- The optimal solution of a subproblem can be structured from the optimal solutions of smaller subproblems.

Solve the subproblems from the smallest to the largest. When you solve a subproblem, store the solution and use it to solve larger subproblems.

5.1 Weighted Interval Scheduling

- A set of requests $R = \{1, 2, \ldots, n\}$.
  - Request i has a start time s_i and a finish time f_i, and a value v_i.
  - Alternative view: every request is an interval $[s_i, f_i]$ associated with a value v_i.
- Two requests i and j are compatible if $[s_i, f_i]$ and $[s_j, f_j]$ do not overlap.

build up a solution:

1. let O the optimal solution
2. O contains an optimal solution O' of the subproblem $R' = \{1, 2, \ldots, i-1\}$
3. in order to find O, it suffices to look at smaller problems and find $O(1, 2, \ldots, j)$ for some j
4. Let O_j be a shorthand for $O(1, 2, \ldots, j)$ and let $OPT(j)$ be its total value.
5. Define $OPT(0) = 0$
6. Then $O = O_n$ with value $OPT(n)$
7. $OPT(j)$ can be computed from $OPT(j-1)$
8. $OPT(j) = \max\{OPT_{p_j} + v_j, OPT(j-1)\}$

Algorithm 18 ComputeOPT

```
1: procedure COMPUTEOPT(j)
2:     if j = 0 then
3:         return 0
4:     else
5:         return max{COMPUTEOPT(p_j) + v(j), COMPUTEOPT(j − 1)}
6:     end if
7: end procedure
```

Correctness: ComputeOPT(j) correctly computes $OPT(j)$ for all $j = 0, 1, \ldots, n$.
Proof by induction:
Base case: $OPT(0) = 0$ by definition.
Inductive step: Assume that it is true for all $i < j$.(Induction hypothesis)
return $\max\{\text{COMPUTEOPT}(p_j) + v(j), \text{COMPUTEOPT}(j-1)\}$
Running time: $\Omega(2^n)$
Memoization:

- Compute ComputeOPT(j) for all $j = 0, 1, \ldots, n$.
- Store it in an accessible place to use again later.
- Keep an array $M[0, \ldots, n]$.
  - initially $M[j] = \text{EMPTY}$ for all $j = 0, 1, \ldots, n$.
  - when ComputeOPT(j) is called, $M[j] =$ ComputeOPT(j).

Algorithm 19 M-ComputeOPT

procedure M-COMPUTEOPT(j)
  if $j = 0$ **then**
    return 0
  else if $M[j]$ is not empty **then**
    return $M[j]$
  else
    $M[j] \leftarrow \max\{\text{M-COMPUTEOPT}(p_j) + v(j), \text{M-COMPUTEOPT}(j-1)\}$
    return $M[j]$
  end if
end procedure

Running time: $O(n \log n)$

Algorithm 20 Find-Solution

procedure FIND-SOLUTION(j)
  if $j = 0$ **then**
    return $\emptyset$
  else
    if then$v(j) + \text{M-COMPUTEOPT}(p_j) > \text{M-COMPUTEOPT}(j-1)$
      return $\{j\} \cup \text{FIND-SOLUTION}(p_j)$
    else
      return $\text{FIND-SOLUTION}(j-1)$
    end if
  end if
end procedure

Dynamic Programming vs Divide and Conquer:

Dynamic Programming:

- DP is an optimisation techniques and is only applicable to problems that have optimal substructure.
- DP splits the problem into parts, finds solutions to the parts and joins them.(The parts are not significantly smaller than the original problem and are overlapping.)
- In DP, the subproblems dependency can be represented by a directed acyclic graph.

Divide and Conquer:

- DC is not normally used for optimisation problems.
- DC splits the problem into parts, finds solutions to the parts and joins them.(The parts are significantly smaller than the original problem and are non-overlapping.)
- In DC, the subproblems dependency can be represented by a tree.

5.2 Subset Sum

Problem Description:

- Given a set of n items $1, 2, \ldots, n$
- Each item i has a non-negative weight w_i.
- Given a bound W.
- **Goal**: select a subset S of items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximised.

Dynamic Programming: To find the optimal value of $OPT(n)$, we need

- the optimal value of $OPT(n-1)$ if item n is not selected.
- the optimal value of the solution on input 1,2,...,n-1 with weight bound $W - w_n$.

subproblems:

- Assumptions:
  - W is an integer
  - Every w_i is an integer
- subproblem for each $i = 0, 1, \ldots, n$ and each integer $0 \leq w \leq W$.
- Let $OPT(i, w)$ be the optimal value of the solution on subset $1, 2, \ldots, i$ with weight bound w.

Algorithm 21 SubsetSum

```
procedure SUBSETSUM(n, w)
    Array M[0, ..., n, 0, ..., W]
    M[0, w] = 0 for each w = 0, 1, ..., W
    for i = 1 to n do
        for w = 0 to W do
            if w_i > w then
                M[i, w] = M[i − 1, w]
            else
                M[i, w] = max{M[i − 1, w], M[i − 1, w − w_i] + w_i}
            end if
        end for
    end for
    return M[n, W]
end procedure
```

Running time: $O(nW)$

5.3 knapSack

Problem Description:

- Given a set of n items $1, 2, \ldots, n$
- Each item i has a non-negative weight w_i and a non-negative value v_i.
- Given a bound W.
- **Goal**: select a subset S of items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximised.

the fractional knapsack problem:

- Given a set of n items $1, 2, \ldots, n$
- Each item i has a non-negative weight w_i and a non-negative value v_i.
- Given a bound W.
- **Goal**: select a fraction x_i of each item i such that $\sum_{i \in S} w_i x_i \leq W$ and $\sum_{i \in S} v_i x_i$ is maximised.

The 0/1 knapsack problem: Solution for 0/1 knapsack problem:

Algorithm 22 0/1 knapsack in dynamic programming

```
procedure 0/1 KNAPSACK(n, W)
    Array M[0, ..., n, 0, ..., W]
    M[0, w] = 0 for each w = 0, 1, ..., W
    for i = 1 to n do
        for w = 0 to W do
            if w_i > w then
                M[i, w] = M[i − 1, w]
            else
                M[i, w] = max{M[i − 1, w], M[i − 1, w − w_i] + v_i}
            end if
        end for
    end for
    return M[n, W]
end procedure
```

Chapter 6

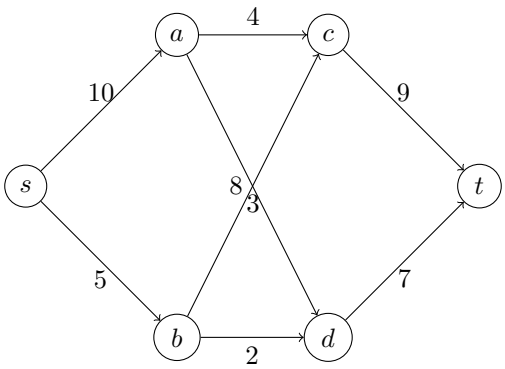
Network Flow

6.1 Network Flow Definitions

Flow network: A flow network is a directed graph $G = (V, E)$ with the following properties:

- Each edge $(u, v) \in E$ has a non-negative capacity c_e.
- There is a single source s in V.
- There is a single sink t in V.
- All other nodes in $V - \{s, t\}$ are called intermediate nodes.

example:



Further definitions:

- The source s has no incoming edges.
- The sink t has no outgoing edges.
- There is at least one edge incident to each node.
- All capacities are integers.

Flow: An $(s - t)$ flow is a function $f : E \to \mathbb{R}^+$, mapping each edge e to a non-negative real number $f(e)$.
A feasible flow must satisfy the following conditions:

- Capacity: For each edge $e \in E$, $0 \leq f(e) \leq c_e$.
- Flow conservation: for each node $v \in V - \{s, t\}$, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

The source s generates flow, and the sink t absorbs flow.
Value of a flow f, denoted $val(f)$, is the total amount of flow generated by the source s:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

Generally, define $f^{out}(v)$ and $f^{in}(v)$ for the flow going out of(resp. going into) node v.
Similarly, define $f^{out}(S)$ and $f^{in}(S)$ for sets of nodes S.

6.2 Maximum Flow Problem

The maximum flow problem: Given a flow network $G = (V, E)$, find a flow of maximum possible value.
algorithm for maximum flow:
Idea: push flow forward on edges with leftover capacity, push flow backward on edges that are already carrying flow.
The residual graph G_f:
The residual graph G_f of G(also called the flow network) is defined as follows:

- The node set V_f of G_f is the same as the node set V.
- For each edge $(u, v) \in E$ which $f(e) < c_e$, there are $c_e - f(e)$ "leftover" units of capacity.
  - We will call this number the **residual capacity** of edge e.
  - We will call the edge e a forward edge.
- For each edge $(u, v) \in E$ with $f(e) > 0$, there is an edge $e' = (v, u)$ in E_f with a capacity of $f(e)$.We will call the edge e' a backward edge.

Working with residual graphs:

- Find an $(s - t)$ path P in G_f. This is called an **augmenting path**.
- Define the bottleneck of P,
  - Denoted bottleneck(P, f)
  - to be the minimum residual capacity of any edge in P.
- Define the augmentation of flow f into flow f'
  - Denoted augment(f, P).

Augmenting the flow:
Feasibility of capacity:

Algorithm 23 Augmenting the flow

```
1:  procedure AUGMENT(f, P)
2:      b ← bottleneck(P, f)
3:      for each edge e = (u, v) ∈ P do
4:          if e is a forward edge then
5:              f(e) ← f(e) + b
6:          else
7:              f(e) ← f(e) − b
8:          end if
9:      end for
10:     return f
11: end procedure
```

consider an arbitrary edge $e = (u, v) \in P$. Suppose that e is a forward edge.

$$0 \le f(e) \le f'(e) = f(e) + b \le f(e) + (c_e - f(e)) = c_e$$

Suppose that e is a backward edge.

$$c_e \geq f(e) \geq f'(e) = f(e) - b \geq f(e) - (f(e) - 0) = 0$$

The Ford-Fulkerson algorithm:

Algorithm 24 Max-flow algorithm

```
1: procedure MAX-FLOW(G, s, t)
2:     f(e) ← 0 for all edges e ∈ E
3:     while there exists an (s − t) path P in G_f do
4:         f ← augment(f, P)
5:         f' ← update(f)
6:         G_f ← update(G_f, f)
7:     end while
8:     return f
9: end procedure
```

Running time of Ford-Fulkerson algorithm: $O(mC)$, where C is the maximum capacity of any edge in the network.

6.3 Min Cut theorem

A cut C is a partition of the nodes of G into two sets S and T such that $s \in S$ and $t \in T$.
The capacity of a cut $C = (S, T)$ of a cut C is the sum of the capacities of the edges "out of" S: these are edges (u, v) such that $u \in S$ and $v \in T$.

The min-cut theorem: In every flow network, the value of the maximum flow is equal to the capacity of the minimum cut.
A series of facts:
Fact 1: Let f be any $(s - t)$ flow and let (S, T) be any cut. Then $v(f) = f^{out}(S) - f^{in}(S)$.

1. By definition, $v(f) = f^{out}(s)$.
2. By definition $f^{in}(s) = 0$.
3. Hence, $v(f) = f^{out}(s) - f^{in}(s)$.
4. For every other node $v \neq s, t$, we have $f^{out}(v) = f^{in}(v)$.
5. Therefore, $v(f) = \sum_{v \in S}(f^{out}(v) - f^{in}(v))$.
6. rewrite as $v(f) = \sum_{v \in S}(f^{out}(v) - f^{in}(v)) = \sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e) = f^{out}(S) - f^{in}(S)$.

Fact 2: Let f be any $(s - t)$ flow and let (S, T) be any $(s - t)$ cut. Then $v(f) = f^{out}(T) - f^{in}(T)$.
Fact 3: Let f be any $(s - t)$ flow and let (S, T) be any $(s - t)$ cut. Then $v(f) \leq c(S, T)$.

$$\begin{aligned} v(f) &= f^{out}(S) - f^{in}(S) \\ &\leq f^{out}(S) \\ &= \sum_{e \text{ out of } S} f(e) \\ &\leq \sum_{e \text{ out of } S} c(e) \\ &= c(S, T) \end{aligned}$$

Fact 4: Let f be any $(s-t)$ flow in G such that the residual graph G_f contains no augmenting paths. Then there exists an $(s-t)$ cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
Proving fact 4: In the residual graph G_f, identify all nodes that are reachable from the source s. Let S^* be the set of these nodes, and let $T^* = V - S^*$.

1. $s \in S^*$ and $t \in T^*$.
2. No edge of G_f crosses from S^* to T^*.
3. Every edge of G_f crosses from T^* to S^*.
4. $f^{out}(S^*) = v(f)$.
5. $f^{in}(S^*) = 0$.

$$\begin{aligned} v(f) &= f^{out}(S^*) - f^{in}(S^*) \\ &= \sum_{e \text{ out of } S^*} f(e) - \sum_{e \text{ into } S^*} f(e) \\ &= \sum_{e \text{ out of } S^*} c(e) - 0 \qquad\qquad = c(S^*, T^*) \end{aligned}$$

Fact 5: If all capacities are integers, then there exists a maximum flow f for which $f(e)$ is an integer for every edge e.

6.4 Choosing Better Augmenting Paths

The Edmonds-Karp algorithm:

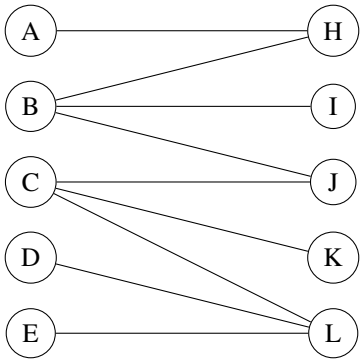
Algorithm 25 Edmonds-Karp algorithm

```
1:  procedure EDMONDS-KARP(G, s, t)
2:      f(e) ← 0 for all edges e ∈ E
3:      while there exists an (s − t) path P in G_f do
4:          P is a shortest (s − t) path
5:          f ← augment(f, P)
6:          f' ← update(f)
7:          G_f ← update(G_f, f)
8:      end while
9:      return f
10: end procedure
```

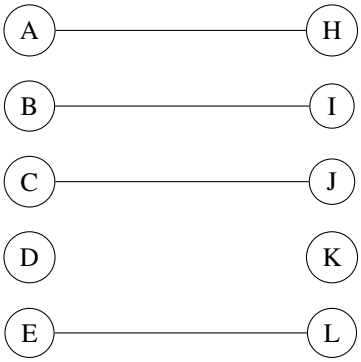
Running time of Edmonds-Karp algorithm: $O(nm^2)$, where n is the number of nodes and m is the number of edges in the network.
The shortest path can be found in $O(m)$ time using BFS.

6.5 Modeling with Network Flows

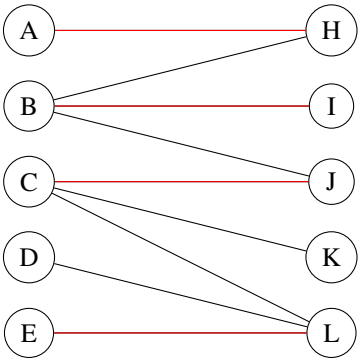
Bipartite graphs: A graph $G = (V, E)$ is bipartite if any only if it can be partitioned into two sets A and B such that every edge has one endpoint in A and one endpoint in B.
Maximum bipartite matching:
Matching: A subset M of edges E such that each node $v \in V$ appears in at most one edge $e \in M$.
Maximum matching: A matching with maximum cardinality.
examples of bipartite graphs:



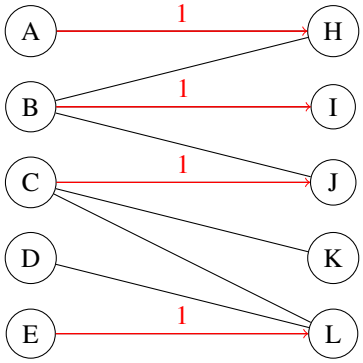
example of maximum bipartite matching:

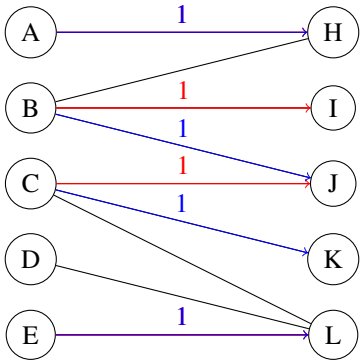


exapmle of maximal bipartite matching:



From matchings to flows:





Maximum Flow and Maximum Matching
The size of the maximum matching is equal to the value of the maximum flow.

The edges of M are the edges that carry flow form A to B in the residual network.
Running time: $O(mn)$

Baseball Elimination

- Given a set S of teams
- For each team x in S, the current number of wins w_x
- For teams x and y in S, they still have to play g_{xy} games against each other
- Given a designated team z
- Can z still win the turnament?

From Baseball Elimination to flows

- For each pair of teams x and y, create a vertex v_{xy}
- For each team x, create a vertex v_x
- For each pair of teams x and y, create an edge (s, v_{xy}) with capacity g_{xy}
- For each team x, create an edge (v_x, t) with capacity $w_z + g_{xz} - w_x$
- For each pair of teams x and y, create an edge (v_{xy}, v_x) with infinite capacity
- For each pair of teams x and y, create an edge (v_{xy}, v_y) with infinite capacity

Open pit mining

- Given a set S of blocks
- For each block x in S, the value v_x of the ore in the block
- For each block x in S, the cost c_x of mining the block
- For each block x in S, the set N_x of blocks that are neighbors of x
- Given a designated block z
- What is the maximum value of ore that can be mined?

From open pit mining to flows

- For each block x in S, create a vertex v_x
- For each block x in S, create an edge (s, v_x) with capacity v_x
- For each block x in S, create an edge (v_x, t) with capacity c_x
- For each block x in S, create an edge (v_x, v_y) with infinite capacity for each block y in N_x

Chapter 7

NP-Completeness

7.1 NP-Completeness

Polynomial time reduction

- Given a problem A to solve
- Reduce solving A to solving B
- Assume there is an algorithm ALG^B that solves B at cost $O(1)$
- Construct an algorithm ALG^A that solves A, which uses ALG^B as a subroutine
- If ALG^A runs in polynomial time, then this is a polynomial time reduction

How to work with reductions
Positive: Assume that I want to solve problem A and I know how to solve problem B.

I can try come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for A.
Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves A.

If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

B is "at least as hard to solve as" A, because if I could solve B, I could also solve A.
Types of reductions

- Turing reduction: $A \leq_T B$
  - A reduction which solves A using (potentially many) calls to an oracle for B
  - As known as Cook reduction
- Many-one reduction: $A \leq_m B$
  - A reduction which converts instances of A to instances of B
  - Also known as Karp reduction

Problem classification
Problems in P:

Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, weighted interval scheduling, etc.
Problems in NP:

subset sum, knapSack, weighted interval scheduling, Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, etc.
NP-hardness
A problem B is NP-hard if for every problem A in NP, $A \leq^p B$.

If every problem in NP is polynomial time reducible to B, then this captures the fact that B is at least as hard as any problem in NP.
3 SAT

- A CNF formula with m clauses and k literals.

$$\varphi = (x_1 \vee \neg x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \cdots \wedge (x_3 \vee x_8 \vee x_{12})$$

- each clause has exactly three literals
- Truth assignment: A value in $\{0, 1\}$ for each variable x_i
- Satisfying assignment: A truth assignment which makes the formula evaluate to 1
- Computational problem 3 SAT: Decide if the input formula φ has a satisfying assignment.

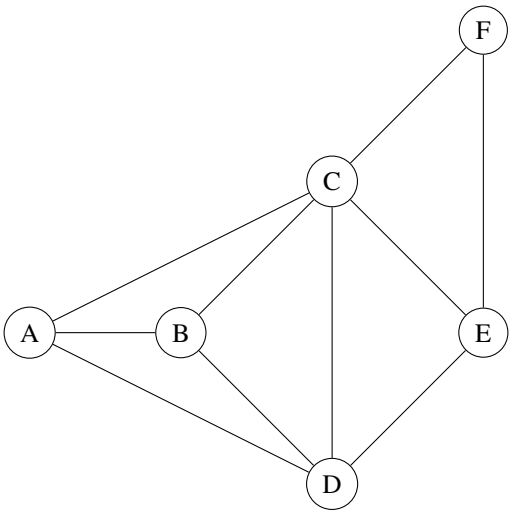
3 SAT is NP-complete

- 3 SAT is in NP
  - Given a truth assignment, we can check in polynomial time if it is satisfying
- 3 SAT is NP-hard
  - Given a CNF formula φ, we can construct a polynomial time reduction to 3 SAT

Proving NP-completeness Suppose that you are given a problem A and you want to prove that A is NP-complete.
First, prove that A is in NP.
Usually by observing that a solution is efficiently verifiable.
Then prove that A is NP-hard.
construct a polynomial time reduction from a known NP-complete problem P.

7.2 NP-completeness of the vertex cover problem

Vertex cover Definition: A vertex cover C of a graph $G = (V, E)$ is a subset of vertices $C \subseteq V$ such that for every edge e in E, at least one of the endpoints of e is in C.
Definition: A minimum vertex cover is a vertex cover of smallest possible size.
Input: A graph $G = (V, E)$
Output: A minimum vertex cover
Example



the vertex cover $\{A, C, E\}$ is not a minimum vertex cover
the minimum vertex cover is $\{A, E\}$.

Vertex cover is NP-hard: construct a polynomial time reduction from 3 SAT to vertex cover.

Let φ be a 3 CNF formula with m clauses and d variables.

Construct in polynomial time an instance $< G, k >$ of vertex cover, with $k = d + 2m$.

if φ is satisfiable, then G has a vertex cover of size at most k

Let $(y_1, y_2, \ldots, y_d)$ in $\{0,1\}^d$ be a satisfying assignment for φ

For the nodes on the top: If $y_i = 1$. Include node x_i in the vertex cover C, otherwise include node $\neg x_i$ in C.

For the nodes on the bottom: in each triangle, choose a node x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.

if φ is not satisfiable, then G has no vertex cover of size at most k

Let C be a vertex cover of size $k = d + 2m$ in G.

Since it is a vertex cover, it must include at least two out of three nodes in each "clause gadget" at the bottom.

this means that at most d nodes can be picked on the top.

To satisfy the edges at the top, in each "variable gadget", at least one node must be picked.

7.3 Further reductions in NP

Form optimization to decision

Given an optimization problem P, introcude a threshold k.

The decision version P_d becomes: Given an instance of P and the threshold k as input, is there a solution to P with value at most k?

If P solved in polynomial time, then P_d is also solved in polynomial time.

If P_d solved in polynomial time, then P is also solved in polynomial time.

NP-complete problems

Independent Set in graph G: A set of nodes in the graph, such that there is no edge between any two nodes in the set.

Maximum Independent Set: Given a graph G, find an independent set of maximum size.

Maximum Independent Set(desision version): Given a graph G and a threshold k, is there an independent set of size at least k?

Set Packing: Given a set U and a collection of subsets $S_1, S_2, \ldots, S_m$ of U and a number k, does there exist a collection of at least k of these subsets that no two of them intersect?

Set Cover: Given a set U and a collection of subsets $S_1, S_2, \ldots, S_m$ of U and a number k, does there exist a collection of at most k of these sets whose union is U?

3-Dimensional Matching: Given three disjoint sets X, Y, Z each of size n, and a collection of triples $T \subseteq X \times Y \times Z$, does there exist a set of n triples in T, so that each element of $X \cup Y \cup Z$ appears in exactly one of these triples?

K-Colouring of a graph G: A function $f : V \to \{1, 2, \ldots, k\}$ so that for every edge (u, v) in E, $f(u) \neq f(v)$.

3-Colouring: Given a graph G, can we colour the nodes of G using 3 colours so that no two adjacent nodes have the same colour?

Hamiltionian cycle in a directed graph G: A cycle in a directed graph that visits every node exactly once.

Hamiltionian path in a directed graph G: A path in a directed graph that contains every node exactly once.

Hamiltionian Cycle: Given a directed graph G, does G contain a Hamiltionian cycle?

Hamiltionian Path: Given a directed graph G, does G contain a Hamiltionian path?

Traveling Salesman: Given a complete graph G with edge weights, and a threshold k, is there a tour of G with total weight at most k?

A texonomy of NP-complete problems

Packing problems: Independent Set, Set Packing

Covering problems: Vertex Cover, Set Cover

Partitioning problems: 3-Dimensional Matching, Graph Colouring

Sqequcen problems: Hamiltionian Cycle, Hamiltionian Path, Traveling Salesman

Numerical problems: Subset Sum, Knapsack

Constraint satisfaction problems: 3 SAT

Chapter 8

Liner Programming

Problem: A company makes two products, X and Y. Each product requires time on two machines, $AandB$.

- Each unit of X requires 50 minutes on machine A and 30 minutes on machine B.
- Each unit of Y requires 24 minutes on machine A and 33 minutes on machine B.
- At the start of the week, there are 30 units of X and 90 units of Y in stock.
- The available processing time on machine A is 40 hours, and on machine B is 35 hours.
- The demand for X in this week is 75 units, and for Y is 95 units.

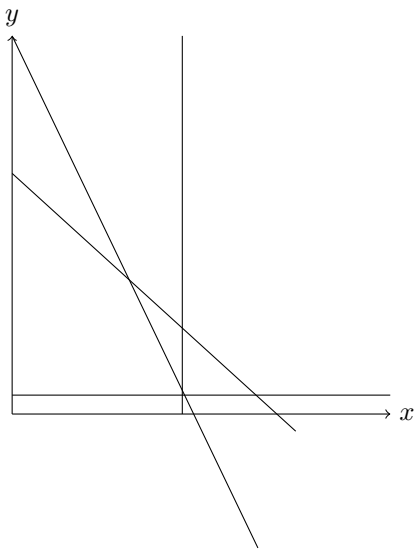
Goal: Maximise the combine sum of units of X and Y in stock at the end of the week.

$$maximise(x + 30 - 75) + (y + 90 - 95) = x + y - 50$$

Base in the Description, we can get the following equations:

$$\begin{aligned} \text{maximize } & x + y - 50 \\ \text{subject to } & 50x + 24y \leq 2400 \\ & 30x + 33y \leq 2100 \\ & x \leq 45 \\ & y \leq 5 \end{aligned}$$

Solving the linear program



8.1 Linear Programming

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \text{ for } j = 1, 2, \ldots, n \end{aligned}$$

matrix form

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}^T \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

Solving the linear program

- To find the optimal solution, it suffices to examine the conrners of the feasible region.
- These are the intersection points of the lines defined by the constraints.
- This is the Simplex method.
- Otehr algorithm for solving LPs: Ellipsoid method, Interior point method.

Integer Linear Programming

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \text{ for } j = 1, 2, \ldots, n \\ & x_j \in \mathbb{Z} \text{ for } j = 1, 2, \ldots, n \end{aligned}$$

Solving the integer linear program
The corners are not necessarily integer solutions. it does not suffices to look at the corners.
WE can exhaustively search all possible integer solutions.
Generally, this is NP-hard.

1. Linear Programming can be solved in polynomial time.
2. Integer Linear Programming Generally cannot be solved in polynomial time.(unless P=NP)

8.2 Duality

- Suppose that we have a linear program, which we call the primal.
- We will construct another linear program, which we call the dual.
- The variables of the primal become the constraints of the dual, and vice versa.
- Maximisation becomes minimisation.
- The two linear programs will have a very important connection.

Primal

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \text{ for } j = 1, 2, \ldots, n \end{aligned}$$

Dual

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} b_i y_i \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \geq c_j \text{ for } j = 1, 2, \ldots, n \\ & y_i \geq 0 \text{ for } i = 1, 2, \ldots, m \end{aligned}$$

Weak Duality
Let $\mathbf{x}$ be any feasible solution to the primal, and let $\mathbf{y}$ be any feasible solution to the dual. Then

$$value(\mathbf{x}) \leq value(\mathbf{y})$$

Strong Duality
Let $\mathbf{x}$ be any feasible solution to the primal, and let $\mathbf{y}$ be any feasible solution to the dual. if

$$value(\mathbf{x}) = value(\mathbf{y})$$

then $\mathbf{x}$ and $\mathbf{y}$ are optimal solutions to the primal and dual respectively.
LP-relaxation
An LP-relaxation of an integer linear program is a linear program which is identical to the integer linear program, except all the integrality constraints have been removed, or replaced with non-negativity constraints.
Example of Minimum Cut as ILP and LP-relaxation

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v) \in E} c_{uv} d_{uv} \\ \text{subject to} \quad & d_{uv} - z_u + z_v \geq 0 \text{ for each } (u, v) \in E, u \neq s, v \neq t \\ & d_{su} + z_u \geq 0 \text{ for each} (s, u) \in E \\ & d_{ut} - z_u \geq 0 \text{ for each} (u, t) \in E \\ & d_{uv} \geq 0 \text{ for each } (u, v) \in E \\ & z_u \in \{0, 1\} \text{ for each } u \in V - \{s, t\} \end{aligned}$$

Chapter 9

Approximation Algorithms

9.1 Approximation Algorithms

Why Approximation Algorithms?

- For some problems, there is no known polynomial time algorithm.
- We can design an approximation algorithm
  - It runs in polynomial time
  - Compute a solution that is guaranteed to be close to the optimal solution

Methods of approximation algorithms

- Greedy algorithms
- Linear programming and rounding
- Pricing method
- Dynamic programming on round inputs

Application: Load Balancing

- We have a set of m identical machines $M_1, M_2, \ldots, M_m$.
- We have a set of n jobs, with job j having processing time t_j.
- We want to assign each job to some machine.
- Let $A(i)$ be the set of jobs assigned to machine i
- The load of machine i is $T(i) = \sum_{j \in A(i)} t_j$
- The goal is to minimize the makespan,i.e, $T = \max_i T(i)$

The load balancing problem is NP-hard.
Greedy Algorithm

- Pick any job
- Assign it to the machine with the smallest load so fraction
- Remove it from the pile of jobs

Arguing about the optimal

- we want to prove that T os not far from T^*
- We will show that T is not far from something which is samller than T^*

- Then it is certainly not far from T^*
- Fundamental idea: Bounding the optimal from below (for minimisation problems) and from above (for maximisation problems).

Consider the total processing time of all jobs.(the sum of all t_j).
one of the m machines must be allocated at least $\frac{1}{m}$ of the total work.
we have that $T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j$
Every job must be scheduled on some machine.
the makespan is certainly at least the largest processing time t_j of any job.
we have that $T^* \geq \max_j t_j$

The Proof

Consider the final job j assigned to machine M_i by the greedy algorithm. Consider the time when this assignment is made.
This was the smallest load among all machines.
Every other machine had load at least $T_i - t_j$ at this time.
Summing up over all machines, we get

$$\sum_k T_k \geq m(T_i - t_j) \Rightarrow T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

So we got the first lower bound.

$$T^* \geq T_i - t_j (\text{first lower bound})$$

Consider a job j assigned to machine M_i by the greedy algorithm. Consider the time when this assignment took place.
The load of machine j was $T_i - t_j$.
This was before we added the job.
After we add the job, the load is $T_i - t_j + t_j$.
Obviously, $t_j \leq \max_k t_k \leq T^*$.
Since T^* is the optimal makespan, it must be at least as large as the load of the machine with the largest load.
So we got the second lower bound.

$$t_j \leq T^* (\text{second lower bound})$$

Combining the two lower bounds, we get

$$T_i \leq 2T^* (\text{since j was the final job})$$

Approximation Ratio

Consider a minimization problem P and an objective obj.
Here: Load Balancing on identical machines, and makespan.

- Consider an approximation algorithm A.
- Consider an input x to P.
- Let $obj(A(x))$ be the value of the objective function on the output of A.

The approximation ratio of A is defined as

$$\max_{x \in \text{all inputs}} \frac{obj(A(x))}{obj(x)}$$

- In order to prove an upper bound on the approximation ratio, we have to somehow argue about all inputs.
- In order to prove a lower bound on the approximation ratio, we have to argue about a specific input.

For maximum problems, we define the approximation ratio as

$$\max_x \frac{obj(x)}{obj(A(x))}$$

For minimization problems, we define the approximation ratio as

$$\max_x \frac{obj(A(x))}{obj(x)}$$

Convention, to have approximation ratio always ≥ 1.
A better greedy algorithm for load balancing

- Sort the jobs in decreasing order of processing times.
- Pick a job and according to this order.
- Assign it to the machine with the smallest load so far.
- Remove it from the pile of jobs.

Algorithm 26 Greedy Algorithm for Load Balancing

1: Sort the jobs in decreasing order of processing times.
2: $T \leftarrow 0$
3: **for** $j = 1$ to n **do**
4: Assign job j to the machine with the smallest load so far.
5: $T \leftarrow \max(T, T(i))$
6: **end for**
7: **return** T

Consider a job j that was assigned to machine M_i by the greedy algorithm. Consider the time when this assignment took place.

The load of machine j was $T_i - t_j$.
This was before we added the job.
After we add the final job, the load is $T_i - t_j + t_j$.
we established that $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$.
Now we have the third lower bound.

$$t_j \leq \frac{1}{2}T^* \text{(third lower bound)}$$

Combining the three lower bounds, we get

$$T_i \leq \frac{3}{2}T^*$$
$$\Rightarrow T \leq \frac{3}{2}T^*$$

The approximation ratio of the greedy algorithm actually is $\frac{4}{3}$ with a better analysis.
For the loading balancing problem on identical machines, there is a Polynomial time approximation scheme (PTAS).

An algorithm which, given an input and a parameter ϵ, runs in polynomial time and produces an outcome which is $1 + \epsilon$ far from the optimal.

Inapproximability

A PTAS (or an FPTAS, more on that later) is the best approximation we can hope for, for an NP-hard problem.
Often it is not possible to get that close.

Inapproximability α of a problem P:

There is no polynomial time algorithm that achieves an approximation ratio better than α.

9.2 The traveling salesman problem

Problem definition:
n cities, c_{ij} is the cost of traveling from city i to city j.
$c_{ij} = c_{ji} \geq 0, c_{ii} = 0$
Feasible solution: A tour that visits every city exactly once and returns to the starting city.
Optimal solution: A feasible tour with minimum total cost.
Graph interpretation
Complete graph on n vertices.
c_{ij} is the weight of the edge (i, j).
Tour: Hamiltonian cycle in the graph.
Goal: A Hamiltonian cycle with minimum total weight.
Mtetric TSP problem: The cost satisfy the triangle inequality.

$$c_{ik} \leq c_{ij} + c_{jk}$$

The Nears Neighbor Algorithm

- Start with a greedy step
- Start with an edge (i, j) of minimum weight.
- Create a cycle(i, j, i).

Generally, Assume that we have already visited a set of cities S. We find a city $i \in S$ and a city $j \notin S$ such that the cost of the edge (i, j) is minimum possible. We include j in S and insert it between i and the city that follows i in the cycle.
Consider the tour $(u_1, u_2, \ldots, u_n)$ and remove the final that brings us back to u_1. We will have a spanning tree. So the cost of the tour is at least the cost of the spanning tree.
Lemma: cost(optimal tour) $\geq$ cost(minimum spanning tree).
We can use Prim Algorithm to find the minimum spanning tree.
Let $F = \{(u_2, v_2), (u_3, v_3), \ldots, (u_n, v_n)\}$ be the set of edges in the minimum spanning tree.

$$\text{Hence} OPT \geq \sum_l c_{u_l, v_l}$$

The cost of the algorithm is at most $2\sum_l c_{u_l,v_l}$.
The cost of the tour between first two nodes is $2c_{u_1,v_1}$.
When inserting the city j, the cost increases by $c_{i,j} + c_{j,k} - c_{i,k} \leq c_{i,j} + c_{i,k} - c_{i,k} = 2c_{i,j}$.
A different perspective

- Start by finding a minimum spanning tree.
- Duplicate each edge of the tree.
- Do a traversal following the edges of the tree.
- Turn this into a tour by shortcutting.

Eulerian Graph
There is a permutation of the edegs $(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_0)$. If this exists, we can make a traversal of the edegs, visiting each edge exactly once.
Property: A graph is Eulerian if and only if it is connected and every vertex has even degree.
Some Facts:
Fact 1: the sum of its degrees is even.
Fact 2: The total number of even-degree nodes is an even number.
Fact 3: The total number of odd-degree nodes is an even number.
Fact 4: The number of odd-degree nodes is even.
Constructing an Eulerian graph:
Start from the minimum spanning tree T.
Consider the set of odd-degree nodes O.
Find a minimum weight perfect matching of O. (This can be done in polynomial time.)

Add the edges of the matching to T.
The resulting graph is Eulerian.
Equivalently: The cost of the perfect matching is at most $OPT/2$.

Bunding the cost of the perfect matching
Claim: There is partial tour containing the cities in O that has cost at most OPT.
Consider the optimal tour.(on the whole graph)
Consider a part $(i, u_1, u_2, \ldots, u_l, j)$ of the tour where the cities $u_1, u_2, \ldots, u_l$ are in O.
"Shortcut" from i to j directly, without an increase in cost.
The cheapest of those two has cost at most $OPT/2$.

The approximation landscape of TSP
The $\frac{3}{2}$-approximation algorithm is due to Christofides (1976).
Inapproximability: 1.0045.

9.3 Primal-Dual Method

Design a polynomial time apporximation algorithm for the weighted vertex cover problem.

The Pricing Method
Consider a vertex cover S.
Every vertex i has a cost, which is equal to w_i.
If $i \in S$, then the edges that "use it" have to pay for the cost.
Each edge e in the graph pays a price p_e.

Given a vertex cover i, we never ask the edges to pay more than the cost of i.

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

Lemma: Let S be any vertex cover and let p_e be any non-negative fair prices. Then ,it holds that:

$$\sum_{e \in E} p_e \leq w(S)$$

Proof of Lemma:

1. By fairness, we have that: $\sum_{e=(i,j)} p_e \leq w_i$
2. Adding up over all nodes, we have
$$\sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in S} w_i = w(S)$$
3. Since S is a vertex cover, each edge contributes at least one term p_e to expression.
4. From this, we get that
$$\sum_{e \in E} p_e \leq \sum_{i \in S} \sum_{e=(i,j)} p_e$$

Terminology: We will say that a node i is "tight" if $\sum_{e=(i,j)} p_e = w_i$.

Algorithm 27 Vertex-Cover-Approximation(G,w)

```
p_e ← 0 for all e ∈ E
while there is an edge e = (i, j) such that neither i nor j is tight do
    p_e ← (w_i + w_j)/2
end while
S ← {i : i is tight}
return S
```

second lemma: The set S and the prices p returned by the Vertex-Cover-Approximation algorithm satisfy:

$$w(S) \leq 2 \sum_{e \in E} p_e$$

Proof of second lemma:

1. Consider a node i in S.
   - This means that i is tight.
   - This means that $\sum_{e=(i,j)} p_e = w_i$.
2. Summing up over all nodes in S, we get that
$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e$$
3. An edge $e = (i, j)$ can be included in the sum at most twice.
4. From this, we get that
$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e$$

Correctness:

1. First,S is a vertex cover.
2. Suppose that it is not.
3. Then there is an edge $e = (i, j)$ that is not covered by S.
4. This means that neither i nor j is tight.
5. The algorithm did not terminate, because of "while there is an edge $e = (i, j)$ such that neither i nor j is tight".

Approximation Ratio:

- Let S^* be the mimimum weight vertex cover.
- Recall that S is the vertex cover returned by the algorithm.
- we have:
$$\sum_{e \in E} p_e \leq w(S^*) \text{ and } w(S) \leq 2 \sum_{e \in E} p_e$$
- Combining the two, we get that $w(S) \leq 2w(S^*)$.

Vertex Cover as an ILP:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} w_i x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0, 1\} \quad \forall i \in V \\ & x_i \geq 0 \quad \forall i \in V \end{aligned}$$

Vertex Cover LP-relaxation:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} w_i x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & 0 \leq x_i \quad \forall i \in V \end{aligned}$$

The Dual

$$\begin{aligned} \text{maximize} \quad & \sum_{(i,j)\in E} y_{ij} \\ \text{subject to} \quad & \sum_{(i,j)\in E} y_{ij} \leq w_i \quad \forall i \in V \\ & y_{ij} \geq 0 \quad \forall (i,j) \in E \end{aligned} \qquad \begin{aligned} \text{maximize} \quad & \sum_{e\in E} p_e \\ \text{subject to} \quad & \sum_{e=(i,j)} p_e \leq w_i \quad \forall i \in V \\ & p_e \geq 0 \quad \forall e \in E \end{aligned}$$

Primal-dual method:

- We start with an infeasible integral solution x to the primal and a feasible fractional solution y to the dual.
- We increase the value of some y_j until some constraint becomes tight.
  - We obtain a better feasible fractional solution y to the dual.
  - We increase the corresending variable x_i of the primal to obtain a still feasible integral solution x to the primal, which however violates fewer constraints.
- We end up with a feasible integral solution x to the primal and a feasible fractional solution y to the dual.
- We compare the two solutions.

9.4 Rounding Linear Programs

In last section, we designed an algorithm for the vertex cover problem, which used the following LP-relaxation:

$$\begin{aligned} \text{minimize} \quad & \sum_{i\in V} w_i x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & 0 \leq x_i \quad \forall i \in V \\ & x_i \in \{0,1\} \quad \forall i \in V \end{aligned}$$

Solving the LP-relaxation:

- We can solve the LP-relaxation in polynomial time, to find an optimal solution.
- The optimal solution is a "fractional" vertex cover, where variables can take values between 0 and 1.
- Is the value of the "fractional" vertex cover, smaller or larger than the value of the mimimum weight vertex cover?

We do not know the optimal, so using a lower bound for the optimal.
Now we have a bound:

$$\text{optimal fractional VC weight} \leq \text{optimal VC weight} \leq \text{algorithm VC weight}$$

Rounding the LP-relaxation:
We round the fractional solution to an integral solution.

- We pick a variable x_i and round it to 0 or 1.
- If we set everything to 0, it is not a vertex cover.
- If we set everything to 1, we "pay" too much.
- We set variable x_i to 1 if $x_i \geq 1/2$ and to 0 otherwise.

Feasibility: Claim: The solution that we obtain is a vertex cover. This is, the round solution is a feasible solution to the ILP. Easy Proof: Assume that it is not.

- Then there is an edge $e = (i, j)$ such that both x_i and x_j are 0.

- This means that in the LP-relaxation, we have that $x_i < 1/2$ and $x_j < 1/2$.
- But then the constraint $x_i + x_j \geq 1$ would be violated, and this would not be a feasible solution to the LP-relaxation.

Approximation Ratio:

- Claim: The vertex cover obtained from rounding has weight at most twice that of the minimum weight vertex cover.
- Intuition is easy: We included all the vertices for which the LP-relaxation "paid" at least 1/2, and we rounded them to 1, so we "paid" at most twice the optimal.
- One line proof: Let x^* be the optimal solution to the LP-relaxation, then

$$\sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S)$$

The LP-relaxation and round algorithm for vertex cover has approximation ratio of 2. We already know that 2 was possible, from the Pricing method algorithm. In this case, the ILP-relaxation and round algorithm seems conceptually simpler. In other cases, rounding the solution will not be straightforward.

Limitations of the techniques

The LP-relaxation and round techniques cannot provide approximation ratios better than the integrality gap of the ILP-LP fomulation.

Caution: The integrality gap is a quantity has to do with the formulation of the ILP-LP, not the problem.

- An integrality gap of α does not mean that it is not possible to design an algorithm with approximation ratio better than α.
- actually, it does not even mean that LP-relaxation and round cannot give you an algorithm with approximation ratio better than α.
- It means that with this formulation of the ILP-LP, α is the best approximation ratio that you can get.

Inapproximability of VC

- We saw two different algorithms, both with an approximation ratio of 2.
- Could we hope to design a better algorithm?
- The answer to this question depends on our definition of "hope".
- **Known fact 1**: It is impossible to design an algorithm with approximation ratio better than 1.363, unless $P = NP$.
- **Known fact 2**: It is impossible to design an algorithm with approximation ratio better than 2, unless Unique Games is an NP hard problem.
- Both facts are quite involved to prove.

Easier Inapproximability

Definition: A problem P is strongly NP-hard, when it has a polynomial time reduction from a strongly NP-hard problem.

For a strongly NP-hard problem

- There is no Fully Polynomial Time Approximation Scheme (FPTAS).
- There is no pseudopolynomial time algorithm that solves it exactly.

  Equivalently, if it remains NP-complete even when all of its number parameters are bounded by a polynomial in the length of the input.

VC on bipartite graphs
Vertex Cover as an ILP:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & x_i \in \{0,1\} \quad \forall i \in V \\ & x \geq 0 \quad \forall i \in V \end{aligned}$$

Vertex Cover as an LP-relaxation:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \forall (i,j) \in E \\ & x \geq 0 \quad \forall i \in V \end{aligned}$$

The dual of the LP-relaxation:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} y_e \\ \text{subject to} \quad & y_e \leq 1 \quad \forall e \in E \\ & y_e \geq 0 \quad \forall e \in E \end{aligned}$$

The ILP corresponding to the dual:

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} y_e \\ \text{subject to} \quad & \sum_{e=\{i,j\} \in E} y_e \leq 1 \quad \forall i \in V \\ & y_e \in \{0,1\} \quad \forall e \in E \end{aligned}$$

König's Theorem
In a bipartite graph, the size of the maximum matching is equal to the size of the minimum vertex cover.

- König's proof is constructive: It starts from a maximum matching, and constructs a minimum vertex cover, proving that it is minimum.
- Alternative proof based on total unimodularity.

So we have:

- size of the maximum matching = size of the minimum vertex cover
- size of the minimum vertex cover = size of minimum fractional VC.
- size of the maximum fractional matching = size of the minimum fractional VC.
- size of maximum matching = size of minimum VC.

Solve for VC on bipartite graphs:

- Pick a vertex v in the graph.
- Remove it to get graph $G - \{v\}$.
- Property: If v is in the minimum vertex cover, $G - \{v\}$ has a minimum vertex cover of size $k^* - 1$.
- Check if the graph $G - \{v\}$ has a vertex cover of size at most $k^* - 1$.

  Yes: Include v in the vertex cover.

  No: Do not include v in the vertex cover.

  Then move on to the next vertex.

Summing up
Vertex Cover is strongly NP-hard in general graphs.

- In fact, hard to approximate better than 1.363.
- There exist 2-approximate polynomial time algorithms for the problem.

On bipartite graphs, the problem is solvable in polynomial time.

9.5 Dynamic Programming on Rounded Inputs

The 0/1-Knapsack Problem
Given a set of n items $\{1, 2, \ldots, n\}$. Each item i has a non-negative value v_i and a non-negative weight w_i. We are given a bound W on the total weight.
Goal: Select a subset of S of items that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximized.
Dynamic Programming algorithm for 0/1-Knapsack:

Algorithm 28 SubsetSum

$M = [0\dot{n}, 0\dot{W}]$
$M[0, 0] \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  for $w = 0, \ldots, W$ **do**
    if $w_i \leq w$ **then**
      $M[i, w] \leftarrow \max\{M[i-1, w], M[i-1, w-w_i] + v_i\}$
    else
      $M[i, w] \leftarrow M[i-1, w]$
    end if
  end for
end for
return $M[n, W]$

The dynamic programming algorithm for 0/1 knapsack solves knapsack optimally in time polynomial in n and W.

Algorithm 29 Knapsack(n, W)

$M = [0\dot{n}, 0\dot{W}]$
$M[0, 0] \leftarrow 0$
for $i = 1, \ldots, n$ **do**
  for $w = 0, \ldots, W$ **do**
    if $w_i \leq w$ **then**
      $M[i, w] = M[i-1, w]$
    else
      $M[i, w] = \max\{M[i-1, w], M[i-1, w-w_i] + v_i\}$
    end if
  end for
end for
return $M[n, W]$

Another pseudopolynomial time algorithm for 0/1-Knapsack

Algorithm 30 Knapsack(n, W)

```
M = [0...n, 0...V]
M[i, 0] ← 0   ∀i
for i = 1, ..., n do
    S(i) = Σ_{j=1}^{i} v_j
end for
for v = 1, ..., V do
    if v > S(n) then
        M[i, v] = w_i + M[i − 1, v − v_i]
    else
        M[i, v] = min{w_i + M[i − 1, v − v_i], M[i − 1, v]}
    end if
end for
return max{v | M[n, v] ≤ W}
```

Intuition

We will create subproblems based on the values, not the weights.

Each subproblem will be defined by an index i and target value v.

$M[i, v]$ is the smallest knapSack weight W such that it is possible to obtain a solution using a subset of items $\{1, \ldots, i\}$ with total value at least v.

We have subproblems at most $(n^2 v^*)$, where v^* is the maximum value of any item.

Information about knapSack

A pseudo-polynomial time algorithm for solving the problem exactly(actually, a couple of those).

A polynomial time greedy approximation algorithm with approximation ratio 2.

Rounding the values

Use a rounding parameter b.

For each item i, let $\tilde{v}_i = \lceil \frac{v_i}{b} * b \rceil$.

It holds that for each item i, $v_i \leq \tilde{v}_i \leq v_i + b$.

let $\hat{v}_i = \frac{\tilde{v}_i}{b} = \lceil \frac{v_i}{b} \rceil$.

We divide all the values by b and round up.

The algorithm

knapsack-approx($\mathcal{E}$) Set $b = \frac{\epsilon}{2n} \max_i v_i$

Run the DP algorithm for knapsack on values $\hat{v}_i$.

Return the set S of items founds.

Feasibility

The set S is a feasible solution to the knapsack, we didn't mess up the weights.

Running time

The DP algorithm runs in time $O(n^2 v^*)$.

Recall:$v^* = \max_i v_i$.

So here, it runs in time polynomial in n and $\max_i v_i$.

It holds that: $\arg\max_i v_i = \arg\max_i \hat{v}_i$.

So we have :$\max_i \hat{v}_i = \hat{v}_j = \lceil \frac{v_j}{b} \rceil = O(n/\epsilon)$.

The overall running time is $O(n^3/\epsilon)$.

This is polynomial in input parameters and $\frac{1}{\epsilon}$.

Approximation ratio

Let S^* be any feasible solution to the knapsack problem.

We know that $\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \hat{v}_i$.

We have the following:

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} (v_i + b) \leq nb + \sum_{i \in S} v_i$$

Recall: $b = (\frac{\epsilon}{2n}) \max_i v_i$.

Let v_j be the largest value. we have that $v_j = \frac{2nb}{\epsilon}$.

We also have that $v_j = \tilde{v}_j$.

Assumption: Each item fits in the knapsack.

This implies that

$$\sum_{i \in S} \tilde{v}_i \geq \tilde{v}_j = v_j = \frac{2nb}{\epsilon}$$

Finally, from the inequalities of the previous slide, we have that

$$\sum_{i \in S^*} v_i \leq nb + \sum_{i \in S} v_i \leq nb + (2\epsilon^{-1} - 1)nb = (2\epsilon^{-1})nb$$

From this, for $\epsilon \geq 1$, we have that

$$nb \leq \epsilon \sum_{i \in S} v_i$$

Back to the inequalities:

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \tilde{v}_i \leq \sum_{i \in S} \tilde{v}_i \leq \sum_{i \in S} (v_i + b) \leq nb + \sum_{i \in S} v_i \leq (1 + \epsilon) \sum_{i \in S} v_i$$

PTAS(Polynomial Time Approximation Scheme): An approximation algorithm which, given an ϵ, runs in time polynomial in the input parameters and has approximation ratio $1 + \epsilon$.
FPTAS(Fully Polynomial Time Approximation Scheme): An approximation algorithm which, given an ϵ, runs in time polynomial in the input parameters and $\frac{1}{\epsilon}$ and has approximation ratio $1 + \epsilon$.
A PTAS for knapsack.
Consider all possible subsets of items with size at most k.

- There are $O(kn^k)$ of those.
- For each one of those subsets, put those items in the knapsack, and use the greedy algorithm to fill the rest of the knapsack.
- One can prove that this solution is a $1 + 1/k$ approximation in time $O(kn^{k+1})$.
- We can pick $\epsilon = 1/k$, and we have a $1 + \epsilon$ approximation in time $O((1/\epsilon)n^{1/\epsilon})$.
- This is polynomial in n but not in $1/\epsilon$.

Inapproximability
Definition: A problem P is strongly NP-hard, when it has a polynomial time reduction from a strongly NP-hard problem.
For a strongly NP-hard problem P

- There is no Fully Polynomial Time Approximation Scheme(FPTAS)
- There is no pseudo-polynomial time algorithm that solves P exactly.

  Equivalently, if it remains NP-complete even when all of its numerical parameters are bounded by a polynomial in the length of the input.

9.6 Summary

Different techniques(greedy, pricing method aka primal-dual, LP-relaxation and rounding, DP on rounded input, brute-force and greedy, dual fitting, etc.)
Limitations of algorithms(tight instances)
Limitations of techniques(integrality gap)

Chapter 10

Randomised Algorithms

10.1 Probability Recap

Finite Probability Space
Sample space Ω, Event E, Probability measure Pr, sample i, $Pr[E] = \sum_{i \in E} Pr(i)$.
Identifier Selection

- There are n processing in a distributed system.
- The set of identifiers is the set of all k-bit strings.
- Each process chooses an identifier uniformly at random.
- What is the probability that processes 1 and 2 choose the same identifier?

For kn-bit identifiers, the possible choices are 2^{kn}, possible points are 2^{kn}, and same probability for all. All possible output with probability $\frac{1}{2^{kn}}$.
What is the probability that processes 1 and 2 choose the same identifier?
The event E consists of all the points for which the first two coordinates (black and red) are the same.
All values possible for coordinates 3 to n, all values possible for coordinate 2 (red) and then coordinate 1 is fixed (black).

$$Pr[E] = \sum_{i \in E} Pr(i) = \frac{1}{2^{kn}} \cdot 2^{k(n-1)} = \frac{1}{2^k}$$

For 1000 processes, the set of identifiers is the set of all 32-bit strings.
The probability of failure is

$$Pr[F] \leq \sum_{i,j} Pr[E_{i,j}] = \binom{1000}{2} \frac{1}{2^{32}} \leq 0.000125$$

The probability of success is at least $1 - 0.000125 = 0.999875$.
conditional probability
Given that event F has occurred, what is the probability that event E occurs?

$$Pr[E|F] = \frac{Pr[E \cap F]}{Pr[F]}$$

$$Pr[E] = \sum_{i \in E} Pr(i)$$

$$for\ any\ sets F_j where\ E \subseteq \cup_{j=1}^{k} F_j$$

Birthday Problem
The probability that there do not exist any two people that share a birthday is equal to :

$$\frac{1 \times (365-1) \times (365-2) \times \cdots \times (365-n+1)}{365^n}$$

For example if there are 25 people, then this is:

$$\frac{1 \times 364 \times 363 \times \cdots \times 341}{365^{25}} \approx 0.431$$

The probability that there exists at least one pair of people that share a birthday is equal to: $1 - \frac{1 \times 364 \times 363 \times \cdots \times 341}{365^{25}} \approx 0.569$

Independent Events
Two events are independent, if Information about the outcome of one of them does not affect our estimate of the likelihood of the other.
Formally: $Pr[E|F] = Pr[E]$ and $Pr[F|E] = Pr[F]$.
This implies:

$$\frac{Pr[E \cap F]}{Pr[F]} = Pr[E] \Rightarrow Pr[E \cap F] = Pr[E] \cdot Pr[F]$$

In other words, the probability that two independent events happen is the product of the probabilities that each one of them happens.
Generalising:

$$Pr[\bigcap_{i \in I} E_i] = \prod_{i \in I} Pr[E_i]$$

The Union Bound
The probability that A or B or C happens is at most the sum of the probabilities that each one of them happens.

$$Pr[\bigcup_{i=1}^{n} E_i] \leq \sum_{i=1}^{n} Pr[E_i]$$

Suppose that we design an algorithm which will produce the correct output with probability.
We first formulate a set of "bad" events $E_1, E_2, \ldots, E_n$, which will force our algorithm to produce the wrong output.
If none of these events happen, then our algorithm will produce the correct output.
Suppose that F is the event the algorithm fails:

$$Pr[F] \leq Pr[\bigcup_{i=1}^{n} E_i] \leq \sum_{i=1}^{n} Pr[E_i]$$

If we can prove that the sum of probabilities of these events is small, then we can prove that our algorithm with high probability produces the correct output.

Random Variables and Expectation
Random Variable: (Informally) A variable X whose values depend on outcomes of a random experiment.
$Pr[X = j]$: The probability that X takes the value j.
Expectqation of X:

$$E[X] = \sum_{j=1}^{n} j \cdot Pr[X = j]$$

Example: Assume that X takes a value in $\{1, 2, \dot{,} n\}$ with probability $\frac{1}{n}$ each.

$$E[X] = \sum_{j=1}^{n} j \cdot Pr[X = j] = \sum_{j=1}^{n} j \cdot \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Waiting for the first success
We flip a biased coin, where $Pr[H] = p$ and $Pr[T] = 1 - p$.
We flip the coin until we get a head.
Let X be the random variable of the number of flips.
Suppose that we get "heads" on the j-th flip.

We have $Pr[X = j] = p \cdot (1-p)^{j-1}$.
The expection then becomes:

$$E[X] = \sum_{j=1}^{\infty} j \cdot Pr[X = j] = \sum_{j=1}^{\infty} j \cdot p \cdot (1-p)^{j-1} = \frac{1}{1-p} \sum_{j=1}^{\infty} j \cdot (1-p)^{j} = \frac{1}{1-p} \cdot \frac{(1-p)}{p^2} = \frac{1}{p}$$

Linearity of Expectation
Let X and Y be two random variables defined on the same probability space.
Let $X + Y$ be the random variable equal to $X(\omega) + Y(\omega)$ on a point ω of the sample space.
It holds that: $E[X + Y] = E[X] + E[Y]$.
Generally:

$$E[X_1 + X_2 + \cdots + X_n] = \sum_{i=1}^{n} E[X_i]$$

Guessing a card
We have a deck of n cards. We draw a card, and before we see it, we have to guess what it is.
We pick one of the cards uniformly at random from the whole deck. Put the card back if we get it wrong.
Let X be the random variable denoting the number of correct guesses.
Let X_i is 1 if the i-th guess is correct, and 0 otherwise.
Obviously $X = X_1 + X_2 + \cdots + X_n$.
We have $E[X_i] = 0 \cdot Pr[X_i = 0] + 1 \cdot Pr[X_i = 1] = \frac{1}{n}$.
By linearity of expectation, we have:

$$E[X] = E[X_1 + X_2 + \cdots + X_n] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{n} = 1$$

For guessing a card without replacement:
To help us compute X, we define a random variable X_i.
X_i is 1 if the i-th guess is correct, and 0 otherwise.
Obviously $X = X_1 + X_2 + \cdots + X_n$.
We have $E[X_i] = 0 \cdot Pr[X_i = 0] + 1 \cdot Pr[X_i = 1] = \frac{1}{n-i+1}$.
By linearity of expectation, we have:

$$\begin{aligned} E[X] &= E[X_1 + X_2 + \cdots + X_n] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{n-i+1} \\ &= \sum_{i=1}^{n} \frac{1}{i} \\ &= H(n) \\ H(n) &= \Theta(\log n) \end{aligned}$$

10.2 Randomised Global Min Cut

Minimum Cut
A cut C is a partition of the nodes of G into two sets S and T, such that $s \in S$ and $t \in T$.
The capacity $c(S, T)$ of the cut C is the sum of the capacities of all edges "out of S".
These are the edges (u, v) such that $u \in S$ and $v \in T$.
Global Minimum Cut
Given an undirected graph $G = (V, E)$. A cut of G is a partition of the nodes of the graph into two sets A and B. The size of a cut (A, B) is the number of edges with one endpoint in A and the other in B. A global minimum cut is a cut of minimum size.

Solving GMC
Theorem: There is a polynomial time algorithm for finding a global minimum cut in an undirected graph G.
Idea: Trun the graph G into a flow network, and find a minimum $s-t$ cut.
Replace every undirected edges with two directed edges, one in the forward direction and one in the backward direction.
Set the capacity of each edge to 1.
Pick two arbitrary nodes s, t in V, and find the minimum $s-t$ cut.
Based on the solving of GMC, a global minimum cut is a cut of minimum size.
Use a randomised algorithm to solve the problem. The algorithm will be faster and simpler. It will produce the correct answer with high probability.
The Contraction Algorithm
Idea:

- Choose an edge of the graph uniformly at random.
- Contract the edge.

  Merge its endpoints (u, v) into a supernode $w = \{u, v\}$.

  Any edge (u, x) is removed.

  Any edge (u, a) or (v, a) is replaced by (w, a).
- When we are left with only two supernodes w_1 and w_2, the corresponding sets of nodes are A and B.

Algorithm 31 Contraction Algorithm

```
1:  procedure CONTRACTION(G)
2:      for each v ∈ V do
3:          S(v) ← {v}
4:      end for
5:      if G has two nodes w_1 and w_2 then
6:          A ← S(w_1)
7:          B ← S(w_2)
8:          return (A, B)
9:      else
10:         choose an edge (u, v) uniformly at random
11:         Let G' be the graph resulting from contracting e, with a new node z_uv replacing u and v
12:         S(z_uv) ← S(u) ∪ S(v)
13:         return CONTRACTION(G')
14:     end if
15: end procedure
```

Analysis of the Contraction Algorithm
Consider a global minimum (A, B) cut of G, and suppose that it has size k. In other words, there is a set F of edges with one endpoint in A and the other in B, such that $|F| = k$. We will prove that the contraction algorithm outputs the cut (A, B) high probability.
Where things can go wrong
If an edge e in F was contracted, when we contract an edge, we irrevocablly decide that its endpoints will be in the same "side" of the cut. For an edge e in F, its endpoints lie in different "sides" of the cut. If we contract e, we can't possibly produce the cut (A, B).
Bounding the error probability
The probability of picking an edge in F:

- F has k edges, pick uniformly at random from $|E|$, so the probability is $\frac{k}{|E|}$.
- To upper bound this quantity, we can lower bound $|E|$.
- Claim: $|E| \geq \frac{nk}{2}$.

The probability of picking an edge in F is at most $\frac{2}{n}$.
After round j

- Suppose that we have gone through j rounds of the algorithm and we have not contracted any edge in F.
- There are $n - j$ super-nodes in the graph G'.
- A cut in G' is also a cut in G.

  The degree of every super-node of G' is at least k.

  $|E_{G'}| \geq \frac{k(n-j)}{2}$.

  The mistake probability is $\frac{k}{|E_{G'}|} = \frac{2}{n-j}$.

Event Mistake: Contract an edge in F.
Event E_j:the algorithm makes a mistake in round j.

$$Pr[E_j] \geq 1 - \frac{2}{n}$$
$$Pr[E_{j+1} \mid E_1 \cap E_2 \cap \cdots \cap E_j] \geq 1 - \frac{2}{n-j}$$

For the probability of not making a mistake in any round:

$$\begin{aligned}
\text{We know } Pr[E_1] &\geq 1 - \frac{2}{n} \\
Pr[E_{j+1} \mid E_1 \cap E_2 \cap \cdots \cap E_j] &\geq 1 - \frac{2}{n-j} \\
Pr[E_1 \cap E_2 \cap \cdots \cap E_{n-2}] &= Pr[E_1] \cdot Pr[E_2 \mid E_1] \cdot \cdots \cdot Pr[E_{n-2} \mid E_1 \cap E_2 \cap \cdots \cap E_{n-3}] \\
&\geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \ldots (1 - \frac{2}{3}) \\
&\geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \cdots \cdot \frac{1}{3} \\
&= \frac{2}{n(n-1)} \\
&= \binom{n}{2}^{-1}
\end{aligned}$$

The contraction algorithm fails to find a global minimum $s - t$ cut with probability at most

$$1 - \frac{1}{\binom{n}{2}} = 1 - \frac{2}{n(n-1)}$$

Suppose that we repeat an experiment multiple times, and each time the probability of success is $p > 0$. If we run the algorithm independently $\binom{n}{2}$ times, the probability of error becomes

$$(1 - \frac{1}{\binom{n}{2}})^{\binom{n}{2}} \leq e^{-1} \approx 0.368$$

If we run the algorithm independently $\binom{n}{2} \ln n$ times, the probability of error becomes

$$(1 - \frac{1}{\binom{n}{2}})^{\binom{n}{2} \ln n} \leq \frac{1}{n}$$

Generally We can run the algorithm independently a number of times, this will reduce the error probability and increase the running time. This is a trade-off between the running time and the error probability.

10.3 Types of Randomised Algs, Max Cut, and SAT

Two main types of randomised algorithms:

- Monte Carlo:The algorithm computes the correct solution with high probability, and the algorithm always terminates

  Example: The global minimum algorithm on multi-graphs

- Las Vegas:The algorithm always computes the correct solution, and its running time is a random variable with bounded expectation.

  Example: Randomised Partition (runs in expected time $O(n)$)

  Example: Randomised Quicksort (runs in expected time $O(n \log n)$)

Markov's inequality: Let X be a non-negative random variable and $a > 0$. Then

$$Pr[X \geq a] \leq \frac{E[X]}{a}$$

From las Vegas to Monte Carlo
Theorem: For any Las Vegas algorithm A, there is a Monte Carlo algorithm B that is correct with probability $\frac{1}{2}$. Proof: Algorithm A has run time X with $\mathbb{E}[x] \leq T$, for some T.
Let algorithm A run for a fixed time $2T$. By Markov's inequality

$$Pr[X \geq 2T] \leq \frac{\mathbb{E}[X]}{2T} \leq \frac{T}{2T} = \frac{1}{2}$$

Thus the probability that by time $2T$, algorithm A has not terminated is at least $\frac{1}{2}$.
Randomised Approximation Algorithms
Use randomisation to design good approximation algorithms.
These algorithms will always terminate, their approximation ratio will be calculated with respect to their expected outcome.
For maximisation problems, the approximation ratio is defined as

$$\max_x \frac{opt(x)}{E[obj(A(x))]}$$

Convention, to have approximation ratios always be ≥ 1.
3 SAT
A CNF formula with m clauses and k literals.

$$\varphi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$$

An AND of ORs
Each clause has three literals.
Truth assignment: A value in $\{0, 1\}$ of each variable x_i.
Satisfying assignment: A truth assignment which makes the formula evaluate to 1.(=true)
Computational problem 3SAT: Decide of the input formula φ is satisfiable.
MAX 3SAT
A CNF formula with m clauses and k literals.

$$\varphi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$$

An AND of ORs
Each clause has three literals.
Truth assignment: A value in $\{0, 1\}$ of each variable x_i.
Satisfying assignment: A truth assignment which makes the formula evaluate to 1.(=true)
Computational problem MAX-3SAT: Find an assignment that satisfies as many clauses of φ as possible.
A 2-approximation for MAX-3SAT
Algorithm: For each variable x_i, set x_i to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$.

Analysis: Let Y_j be a random variable such that $Y_j = 1$ if clause j is satisfied and $Y_j = 0$ otherwise.
Let X be a random variable ,which is equal to the number of satisfied clauses.
By definition, $X = \sum_{j=1}^{m} Y_j$.
we have that

$$\mathbb{E}[X] = \mathbb{E}[\sum_{j=1}^{m} Y_j] = \sum_{j=1}^{m} \mathbb{E}[Y_j] = \sum_{j=1}^{m} Pr[\text{clause j is satisfied}]$$

The probability that each positive literal is set to 0, each negative literal is set to 1. each one of those happens with probability $\frac{1}{2}$, overall, this happens with probability $(\frac{1}{2})^{f(j)}$.
$f(j)$ is the number of literals in clause j.
The probability that clause j is satisfied is

$$1 - (\frac{1}{2})^{f(j)} \geq \frac{1}{2}$$

we have:

$$E[X] = E[\sum_{j=1}^{m} Y_j] = \sum_{j=1}^{m} \mathbb{E}[Y_j] = \sum_{j=1}^{m} Pr[\text{clause j is satisfied}] \geq \frac{m}{2}$$

If we use the trivial upper bound of m on the value of the optimal, we get the 2-approximation.
Analysis of MAX 3SAT
The probability that clause j is satisfied is

$$1 - (\frac{1}{2})^{f(j)} = 1 - (\frac{1}{2})^{3} = \frac{7}{8}$$

we have:

$$E[X] = E[\sum_{j=1}^{m} Y_j] = \sum_{j=1}^{m} \mathbb{E}[Y_j] = \sum_{j=1}^{m} Pr[\text{clause j is satisfied}] = \frac{7}{8}m$$

Minimum Cut vs Maximum Cut
Minimum Cut can be solved in polynomial time.
Always correctly, by flow algorithms.
Almost always correctly, by the contraction algorithm.
Maximum Cut is NP-hard.
MAX-CUT algorithm

- For every vertex $v \in V$ in dependently,
  place $v \in A$ with probability $\frac{1}{2}$
  place $v \in B$ with probability $\frac{1}{2}$.
- We will prove that the approximation ratio of this algorithm for the maximum cut problem is 2.

Analysis
Let X_{ij} be a random variable such that $X_{ij} = 1$ if (i, j) crosses the cut and $X_{ij} = 0$ otherwise.
Let Z be a random variable, which is equal to the number of edges that cross the cut.
By definition, $Z = \sum_{(i,j)\in E} X_{ij}$.
We have that

$$\mathbb{E}[Z] = \mathbb{E}[\sum_{(i,j)\in E} X_{ij}] = \sum_{(i,j)\in E} \mathbb{E}[X_{ij}] = \sum_{(i,j)\in E} Pr[(i,j)\text{crosses the cut}]$$

The probability that (i, j) crosses the cut is $\frac{1}{2}$.

$$E[Z] = E[\sum_{(i,j)\in E} X_{ij}] = \sum_{(i,j)\in E} \mathbb{E}[X_{ij}] = \sum_{(i,j)\in E} Pr[(i,j)\text{crosses the cut}] = \frac{1}{2}m$$

Compare the solution The randomised algorithm works well in expectation. The deterministic algorithm works well always.
Derandomisation

- Sometimes it is possible to "derandomise" a randomised algorithm A_{rand} and obtain a deterministic algorithm A_{det}.

- The performance of A_{det} is the same as the expected performance of A_{rand}.
- We can use randomisation at no extra cost.
- Different mathods for derandomisation.

  Can be very cpmplicated (pseudo-random generators).

  Can be relatively simple (conditional expectations).

10.4 Derandomisation and Randomised Rounding

Algorithm: For each variable x_i, set x_i to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$.
Algorithm: Set variable x_i to 1 or 0 deterministically, and the remining variables to 1 with probability $\frac{1}{2}$ and to 0 with probability $\frac{1}{2}$.
If W is the number of clauses satisfied then we have

$$\begin{aligned}\mathbb{E}[W] &= \mathbb{E}[W|x_1 \leftarrow 1] \cdot Pr[x_1 \leftarrow 1] + \mathbb{E}[W|x_1 \leftarrow 0] \cdot Pr[x_1 \leftarrow 0] \\ &= \frac{1}{2}\mathbb{E}[W|x_1 \leftarrow 1] + \mathbb{E}[W|x_1 \leftarrow 0]\end{aligned}$$

We set x_1 to 1 if $\mathbb{E}[W|x_1 \leftarrow 1] \geq \mathbb{E}[W|x_1 \leftarrow 0]$ and to 0 otherwise.
Generally, if b_1 is piacked to maximise the conditional expectation, then

$$\mathbb{E}[W|x_1 \leftarrow b_1] \geq \mathbb{E}[W]$$

Applying this to all variables
Assume that we have set variables $x_1, \ldots, x_i$ to $b_1, \ldots, b_i$ this way.
We set x_{i+1} to 1 if this holds and to 0 otherwise.

$$\begin{aligned}&\mathbb{E}\left[W \mid x_1 \leftarrow b_1, x_2 \leftarrow b_2, \ldots, x_i \leftarrow b_i, x_{i+1} \leftarrow 1\right] \geq \\ &\mathbb{E}\left[W \mid x_1 \leftarrow b_1, x_2 \leftarrow b_2, \ldots, x_i \leftarrow b_i, x_{i+1} \leftarrow 0\right]\end{aligned}$$

Again, if b_{i+1} is picked to maximise the conditional expectation, then

$$\mathbb{E}[W \mid x_1 \leftarrow b_1, x_2 \leftarrow b_2, \ldots, x_i \leftarrow b_i, x_{i+1} \leftarrow b_{i+1}] \geq \mathbb{E}[W]$$

In the end we have set all variables deterministically.
We have that

$$\mathbb{E}[W \mid x_1 \leftarrow b_1, x_2 \leftarrow b_2, \ldots, x_i \leftarrow b_i, x_{i+1} \leftarrow b_{i+1}] \geq \mathbb{E}[W]$$

We know that $\mathbb{E} \geq \frac{1}{2} \cdot OPT$
We have devised a deterministic 2-approximation algorithm.
Computing the expectations
We have to be able to compute the additional expectations in polynomial time.

$$\begin{aligned}\mathbb{E}\left[W \mid x_1 \leftarrow b_1, \ldots, x_i \leftarrow b_i\right] &= \sum_{j=1}^{m} \mathbb{E}\left[Y_j \mid x_1 \leftarrow b_1, \ldots, x_i \leftarrow b_i\right] \\ &= \sum_{j=1}^{m} \Pr\left[\text{ clause } C_j \text{ is satisfied } \mid x_1 \leftarrow b_1, \ldots, x_i \leftarrow b_i\right]\end{aligned}$$

The probability is

- 1 if the variables already set satisfy the clause.
- $1 - \frac{1}{2^k}$ otherwise, where k is the number of unset variables in the clause.

Derandomisation using conditional expectations Works for a wide variety of applications as long as

- The variables are set independently
- The conditional expectations can be calculated in polynomial time

Randomised rounding
We formulate the problem as an ILP.
We write the LP-relaxation.
We solve the LP-relaxation.
We round the variables with probabilities that can depend on their values.
MAX SAT as an ILP

$$\begin{aligned}
\text{maximise } & \sum_{j=1}^{m} y_j \\
\text{subject to } & \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i) \geq z_j \text{ for all } C_j = \bigvee_{i \in P_j} x_i \vee \bigvee_{i \in N_j} \bar{x}_i \\
& y_i \in \{0, 1\} \text{ for all } i \in \{1, \ldots, n\} \\
& 0 \leq z_j \leq 1 \text{ for all } j \in \{1, \ldots, m\}
\end{aligned}$$

MAX SAT LP-relaxation

$$\begin{aligned}
\text{maximise } & \sum_{j=1}^{m} z_j \\
\text{subject to } & \sum_{i \in P_j} y_i + \sum_{i \in N_j} (1 - y_i) \geq z_j \text{ for all } C_j = \bigvee_{i \in P_j} x_i \vee \bigvee_{i \in N_j} \bar{x}_i \\
& 0 \leq y_i \leq 1 \text{ for all } i \in \{1, \ldots, n\} \\
& 0 \leq z_j \leq 1 \text{ for all } j \in \{1, \ldots, m\}
\end{aligned}$$

Let (y^*, z^*) be an optimal solution to the LP-relaxation.
Rounding: Set x_i to true independently with probability y_i^*.
Analysis

$$\begin{aligned}
Pr[\text{clause } C_j \text{ is not satisfied}] &= \prod_{i \in P_j} (1 - y_i^*) \prod_{i \in N_j} y_i^* \\
&\leq \left[\frac{1}{\ell_j} \left(\sum_{i \in P_j} (1 - y_i^*) + \sum_{i \in N_j} y_i^* \right) \right]^{\ell_j} \\
&\leq \left[1 - \frac{1}{\ell_j} \left(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j} (1 - y_i^*) \right) \right]^{\ell_j} \\
&\leq \left[1 - \frac{1}{\ell_j} z_j^* \right]^{\ell_j}
\end{aligned}$$

and so we have that

$$Pr[\text{clause } C_j \text{ is satisfied}] \geq 1 - \left[1 - \frac{1}{\ell_j} z_j^* \right]^{\ell_j} \geq 1 - \left[1 - \frac{1}{\ell_j} \right]^{\ell_j} z_j^*$$

Analysis

$$\begin{aligned}
\mathbb{E}[W] &= \sum_{j=1}^{m} Pr[\text{clause } C_j \text{ is satisfied}] \\
&\geq \sum_{j=1}^{m} \left(1 - \left[1 - \frac{1}{\ell_j}\right]^{\ell_j}\right) z_j^* \\
&\geq \min_{\ell \geq 1} \left[1 - \left(1 - \frac{1}{\ell}\right)^{\ell}\right] \sum_{j=1}^{m} z_j^* \\
&\geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^{m} z_j^* \\
&= \left(1 - \frac{1}{e}\right) OPT
\end{aligned}$$

Randomised rounding for MAX-SAT
Our randomised algorithm gives an approximation ratio of $\frac{1}{1-\frac{1}{e}} \approx 1.59$.
This is better than the deterministic 2-approximation algorithm.
This is better than 1.618.
The better of the two
Algorithm 1:

$$Pr[\text{clause } C_j \text{ is satisfied}] \geq \left(1 - (\frac{1}{2})^{\ell_j}\right)$$

Algorithm 2:

$$Pr[\text{clause } C_j \text{ is satisfied}] \geq \left[1 - \left(1 - \frac{1}{\ell_j}\right)^{\ell_j}\right] z_j^*$$

If the clause is long, Algorithm 1 is better.
If the clause is short, Algorithm 2 is better.
Algorithm 3: Choose the better of the two.
Analysis

$$\begin{aligned}
\mathbb{E}[W] &= \mathbb{E}[\max(W_1, W_2)] \\
&\geq \mathbb{E}\left[\frac{1}{2}W_1 + \frac{1}{2}W_2\right] \\
&\geq \frac{1}{2}\mathbb{E}[W_1] + \frac{1}{2}\mathbb{E}[W_2] \\
&\geq \frac{1}{2}\sum_{j=1}^{m}\left[1 - (\frac{1}{2})^{\ell_j}\right] + \frac{1}{2}\sum_{j=1}^{m} z_j^*\left[1 - \left(1 - \frac{1}{\ell_j}\right)^{\ell_j}\right] \\
&\geq \sum_{j=1}^{m} z_j^*\left[\frac{1}{2}\left(1 - \left(1 - \frac{1}{\ell_j}\right)^{\ell_j}\right) + \frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{\ell_j}\right)\right]
\end{aligned}$$

This quantity

$$\frac{1}{2}\left(1 - \left(1 - \frac{1}{\ell_j}\right)^{\ell_j}\right) + \frac{1}{2}\left(1 - \left(\frac{1}{2}\right)^{\ell_j}\right)$$

For $\ell_j = 1$, it evaluates to $\frac{1}{2}$.
For $\ell_j = 2$, it evaluates to $\frac{3}{4}$.

For $\ell_j \geq 3$, we have that

$$\left(1 - (\frac{1}{2})^{\ell_j}\right) \geq \frac{7}{8}$$
$$\left(1 - \left(1 - \frac{1}{\ell_j}\right)^{\ell_j}\right) \geq 1 - \frac{1}{e}$$
$$\frac{1}{2} \cdot \frac{7}{8} + \frac{1}{2}\left(1 - \frac{1}{e}\right) \approx 0.753 \geq \frac{3}{4}$$
$$\mathbb{E}[W] \geq \frac{3}{4}\sum_{j=1}^{m} z_j^* = \frac{3}{4}OPT$$

The "better of the two" algorithm has approximation ratio $\frac{4}{3} \approx 1.33$.
integrality Gap of MAX-SAT
Consider the formula:

$$(X_1 \vee X_2) \wedge (\neg X_1 \vee X_2) \wedge (X_1 \vee \neg X_2) \wedge (\neg X_1 \vee \neg X_2)$$

the optimal integral solution satisfied 3 clauses.
The optimal fractional solution sets $y_1 = y_2 = \frac{1}{2}$ and $z_j = 1 \forall j$ and satisfies 4 clauses.
The integrality gap is at least $\frac{4}{3}$.
Set x_i to true independently with probability $f(y_i^*)$ for some function f such that $1 - 4^{-x} \leq f(x) \leq 4^{x-1}$.
Analysis

$$\begin{aligned}
Pr[\text{clause } C_j \text{ is not satisfied}] &= \prod_{i \in P_j}(1 - f(y_i^*)) \prod_{i \in N_j} f(y_i^*) \\
&\leq \prod_{i \in P_j} 4^{-y_i^*} \prod_{i \in N_j} 4^{y_i^* - 1} \\
&= 4^{-\left(\sum_{i \in P_j} y_i^* + \sum_{i \in N_j}(1 - y_i^*)\right)} \\
&\leq 4^{-z_j^*} \\
Pr[\text{clause } C_j \text{ is satisfied}] &\geq 1 - 4^{-z_j^*} \geq \left(1 - \frac{1}{4}\right) z_j^* = \frac{3}{4} z_j^* \\
\mathbb{E}[W] &= \sum_{j=1}^{m} Pr[\text{clause } C_j \text{ is satisfied}] \\
&\geq \frac{3}{4}\sum_{j=1}^{m} z_j^* \\
&\geq \frac{3}{4}OPT
\end{aligned}$$

The more sophisticated RR algorithm has approximation an approximation ratio of $\frac{4}{3} \approx 1.33$.

Chapter 11

Online Algorithms

11.1 Online Algorithms, Load Ballancing, Paging

Motivating Example
Suppose that you need to take 4 modules in your Masters programme, but you dont't know the difficulty, the content, or the lecturer of each module. You need to make a decision based on limitted information.
Suppose that you have completed your Masters programme successfully and now you are looking for jobs. You have made several applications and you receive an offer from some company. Should you accept it, or should you wait to see if you might get a better offer from another company?
Let's say that you make a series of local (myopic) decisions, based only on information that you have seen so far (and possibly what you expect to see in the future).You can compare the quality of your decisions to that of theclairvoyant.If they are not much worse, then you can convince yourself that you have made good decisions

Suppose that the input of a problem P is given steps.
You have to make a decision in every step.
The goal is to optimaise some objective.
You don't know the length of the input- the input supply might stop at any point.
You will compare against the offline optimal algorithm, which knows the future, and computes the optimal solution on the entire input.
Online algorithms: An algorithm that makes decisions now about events that will happen in the future, without having knowledge of the future events.
Online Load Balancing
- We have a set of m identical machines $\mathrm{M}_1, \ldots, \mathrm{M}_\mathrm{m}$
- We have a set of n jobs, with job j having processing time t.
- The jobs arrive over time, one in each time step.
- We want to assign every job to some machine.
- We will assign a job immediately upon arrival to some machine.
- Let $A(i)$ be the set of jobs assigned to machine i.
- The load of machine i is $T_i = \sum_{j \in A(i)} t_j$
- The goal is to minimise the makespan, i.e., $\mathrm{T} = \max_\mathrm{i} \mathrm{T}_\mathrm{i}$

Competitive ratio
The comptitive ratio of algorithm A is defined as

$$\max_x \frac{obj(A(x))}{opt(x)}$$

Competitive Ratio vs Approximation Ratio
Approximation ratio: The constraint of our algorithm is that it must run in polynomial time. If we didn't have atime constraint, we would obtain the optimal.
Competitive Ratio: The constraint of our algorithm is that it does not know the future part of the input. If we had access to the future part of the input, we would obtain the optimal.
Greedy Algorithm for load balancing

Algorithm 32 Greedy Algorithm for load balancing

1: set $T_i = 0$ and $A(i) = \emptyset$ for all i
2: **for** each job j **do**
3: Let M_i be the machine that achieves the minimum $\min_k T_k$
4: assign j to machine i with minimum load T_i
5: $T_i = T_i + t_j$
6: $A(i) = A(i) \cup \{j\}$
7: **end for**

Greedy Algorithm for online load balancing
• Pick the job that arrives in the current time step.
• Assign it to the machine with the smallest load so far.
• Remove it from the pile of jobs.
The competitive ratio of the greedy algorithm is $2 - 1/m$.
lower bounds: We can show lower bounds on the competitive ratio of any online algorithm, using elementary arguments.
This comes in contrast to approximation algorithms, where Inapproximability results typically require advanced techniques.
terminology:
We will say that the input is given by an adversary, who wishes to maximise the competitive ratio of the algorithm. This is equivalent to considering the worst case input sequence.
Better Algorithms
It is possible to design better online algorithms for the scheduling problem.
For example, for $m = 4$, there is an algorithm with competitive ratio at 1.733 .
Lower bound: For $m = 4$, no online algorithm has competitive ratio better than 1.732 .
For general m, the best possible competitive ratio is between 1.88 and 1.92.
Idea: The Tetris principle - maintain imbalance.

Paging
We have two types of memory, a fast memory (cache) and a slow memory.
The cache has capacity k pages, the slow memory has capacity n pages.
We have a sequence of page requests.
If the page is in the cache, the algorithm returns it at no cost.
If the page is not in the cache, the algorithm "faults" and has to bring it from the cache, paying a cost of 1.
The algorithm must also choose a page in the cache to replace with the page brought from the slow memory.
The online algorithm makes x "faults".
The offline optimal makes $y \leq x$ "faults".
Theorem: The competitive ratio of any online algorithm for paging is at least k. **Lower bound**

- The algorithm "faults" once at every step.
- What about the offline optimal?

  Consider the strategy: "When replacing a page, replace the one that will be requested the furthest in the future".
- Suppose that OPT "faults" on some page p. OPT replaces a page (to bring in p) that will not be requested in the next k-1 steps.

  OPT "faults" once every k steps.

  The competitive ratio is at least k.

Paging Algorithms

- LRU(Least Recently Used): Replace the page that was requested the least recently
- FIFO(First In First Out): Replace the page that has been in the cache the longest
- LIFO(Last In First Out): Replace the page that has been in the cache the shortest

- LFU(Least Frequently Used): Replace the page that was requested the least frequently so far
- MIN(Offline Optimal): Replace the page whose next request happens furthest in the future

Theorem: LRU and FIFO have competitive ratio at most k.

11.2 Marking Algorithm & Randomised Paging

Marking Algorithm:
consider the following algorithm:

- The algorithm proceeds in phases.
- At the beginning of a phase, all the pages are unmarked.
- Whenever a page is requested, it is marked.
- When a "fault" occurs, the algorithm replaces an unmarked page.
- When all pages in the cache are marked, and a request for an unmarked page occurs, the phase ends.

Theorem: The marking algorithm has competitive ratio k.
The algorithm "faults" at most k times in every phase.
Every time it fails, the requested page is marked.
If all pages in the cache are marked and a new page is requested, then the phase changes.
The optimal offline algorithm "faults" at least once in every phase.
The phase ends when $\mathrm{k}+1$ different pages are requested.
The optimal offline algorithm can only keep at most k of those in the cache.
Theorem: LRU and FIFO have competitive ratio at most k.
Proof: LRU and FIFO are marking algorithms.
Corollary: LRU and FIFO are the best online algorithms for paging.
randomisation:
We will use randomisation to achieve a better competitive ratio.
We have to make a distinction, regarding the power of the adversary:
Oblivious Adversary: The adversary fixes an input sequence in advance.
Adaptive Adversary: The adversary can change the input sequence based on the realisations of randomness of the choices of the algorithm.
A Randomised Algorithm A has competitive ratio r if there exists a constant c such that for all inputs x :

$$\mathbb{E}[\mathrm{obj}(A(x))] \leq r^* \operatorname{opt}(x) + c$$

Randomised marking algorithm:

- The algorithm proceeds in phases.
- At the beginning of a phase, all the pages are unmarked.
- Whenever a page is requested, it is marked.
- When a "fault" occurs, the algorithm replaces an unmarked page, chosen uniformly at random.
- When all pages in the cache are marked, and a request for an unmarked page occurs, the phase ends.

Recall the k-th Harmonic number:

$$H_k = \sum_{i=1}^{k} \frac{1}{i}$$

Theorem: The randomised marking algorithm has competitive ratio $2H(n)$ against Oblivious Adversaries.

$$H(k) = O(\log k)$$

Proof;
Assume without loss of generality that RMA makes "faults" on the first request.
Consider phase i.

Let m_i be the number of "new" pages requested in the phase (i.e. pages that were not requested in previous phases).
call the remaining $k - m_i$ pages "old".

Every time a "new" page is requested, we have a "fault".
Every time an "old" page is requested, we may have "fault".

the "fault" occurs if we replaced the "old" page with a "new" one.

Assume that the m_i request for "new" pages come first and the $k - m_i$ requests for "old" pages come last.

The probability that a "fault" occurs is $\frac{m_i}{k}$.

There are $k - m_i - (j-1)$ unmarked pages in cache, this is a random subset of the $k - (j-1)$ old pages not requested so far.
Don't fault with probability $\frac{k-m_i-(j-1)}{k-(j-1)}$, thus the probability of a "fault" is:

$$1 - \frac{k - m_i - (j-1)}{k - (j-1)} = \frac{m_i}{k - (j-1)}$$

The expected number of "faults" in phase i is:

$$m_i + \frac{m_i}{k} + \frac{m_i}{k-1} + \cdots + \frac{m_i}{k - (m_i - 1)} \geq m_i \cdot H(k)$$

Summing over all phases, we get:

$$\mathbb{E}[\text{cost of RMA}] \leq H(k) \sum_{i=1}^{n} m_i$$

about OPT
What is the number of "faults" that the optimal offline algorithm makes?
Let n_{i-1} and n_i be the number of "faults" of OPT in phases $i-1$ and i.
At least $k + m_i$ distinct pages requested in phases i $-$ 1 and i.
Cache has size k, thus we have $n_{i-1} + n_i \geq m_i$.
Summing pairs of phases, we get:

$$OPT \geq \sum_{i=1}^{\frac{n}{2}} m_{2i} \quad \text{and} \quad OPT \geq \sum_{i=0}^{\frac{n}{2}-1} m_{2i+1} \quad \text{thus} \quad OPT \geq \frac{1}{2} \sum_{i=1}^{n} m_i$$

The competitive ratio of RMA is $2H(k)$.