CSC373 Notes

Jenci Wei

Fall 2022

Contents

1 Divide and Conquer

1.1 Master Theorem

Divide and Conquer Algorithm

- Divide problem of size n into a smaller subproblems of size n/b each
- Recursively solve each subproblem
- Combine the subproblem solutions into the sulution of the original problem
- Runtime:
$$T(n) = aT\left(\frac{n}{b}\right) + cn^d$$
  - Integer $a \geq 1, b > 1, c \geq 0, d \geq 0$
  - $T(n)$ is time on input of size n
  - $aT(n/b)$ is time for the a *recursive calls*
  - cn^d is time for *dividing* the problem and *combining* the subproblem solutions

Master Theorem

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

- The running time does *not* depend on the constant c
- In many algorithms $d = 1$, i.e. dividing and combining takes *linear* time, in which case
$$T(n) = \begin{cases} \Theta(n), & \text{if } a < b \\ \Theta(n \log n), & \text{if } a = b \\ \Theta(n^{\log_b a}), & \text{if } a > b \end{cases}$$

Well-Known Examples

- Merge sort: sorting array of size n
  - Split array into two arrays of size $n/2$ each $\implies a = 2, b = 2$
  - Sort each array recursively
  - Merge the two sorted arrays (in linear time) $\implies d = 1$
  - $a = b^d \implies T(n) = \Theta(n \log n)$
- Binary search: search an item x in a sorted array of size n
  - Compare x with the middle element of the array
  - Return "found", or recursively search first **or** second *half of the array* $\implies a = 1, b = 2, d = 0$
  - $a = b^d \implies T(n) = \Theta(\log n)$

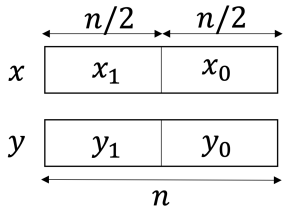
1.2 Karatsuba's Integer Multiplication Algorithm

Overview and Motivation

- Adding two binary numbers of length n takes $\Theta(n)$ time
- Multiplying two binary numbers of length n takes $\Theta(n^2)$ time by performing n additions
- We want to do better than $\Theta(n^2)$

Karatsuba's Algorithm

- Assume n is a power of 2



- Then

$$x = x_1 \cdot 2^{n/2} + x_0$$
$$y = y_1 \cdot 2^{n/2} + y_0$$

- We can rewrite the multiplication as

$$x \cdot y = \underbrace{x_1 \cdot y_1}\cdot 2^n + (\underbrace{x_1 \cdot y_0} + \underbrace{x_0 \cdot y_1}) \cdot 2^{n/2} + \underbrace{x_0 \cdot y_0} \qquad \text{Notice the 4 multiplications}$$
$$= \underbrace{x_1 \cdot y_1}\cdot 2^n + [\underbrace{(x_1 + x_0) \cdot (y_1 + y_0)} - \underbrace{x_1 \cdot y_1} - \underbrace{x_0 \cdot y_0}] \cdot 2^{n/2} + \underbrace{x_0 \cdot y_0} \qquad \text{Notice the 3 multiplications}$$

- $T(n) = 3 \cdot T(n/2) + c \cdot n$ and so $a = 3, b = 2, d = 1$
- $a > b^d \implies T(n) = \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.585}\right)$
- Minor issue: a carry may increase length of $x_1 + x_0$ and $y_1 + y_0$ to $n/2 + 1$
  - But that's okay, since we can decompose the $n/2+1$ by $n/2+1$ multiplication to a 1 by 1, two 1 by $n/2$, and a $n/2$ by $n/2$ multiplication

Pseudocode

```
MULT(x, y):
    n <- max{length(x), length(y)}
    Put 0s into the left of the shortest of x and y so both have length n
    x0 <- rightmost n/2 bits of x
    x1 <- leftmost n/2 bits of x
    y0 <- rightmost n/2 bits of y
    y1 <- leftmost n/2 bits of y
    s <- Add(x0, x1)
    t <- Add(y0, y1)
    a <- Mult(x0, y0)
    b <- Mult(x1, y1)
    c <- Mult(s, t)
    u <- Subtract(c, Add(a, b))
    Put n zeros to the right of b
    Put n/2 zeros to the right of u
    Return Add(b, Add(u, a))
```

Lifting the Assumption

- When n is not a power of 2, we can embed each vector in a vector whose dimension is the next power of 2
- In the worst case, this doubles the size of the problem n to $2n$
- And so $T(n) = \Theta\left((2n)^{\log_2 3}\right) = \Theta\left(2^{\log_2 3} n^{\log_2 3}\right) = \Theta\left(3n^{1.585}\right) = \Theta\left(n^{1.585}\right)$

State of the Art

- 1962: Karatsuba $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$
- 1971: Schönhage-Strassen $\Theta(n \cdot \log n \cdot \log(\log n))$
- 2019: Harvey and van der Hoeven $\Theta(n \log n)$

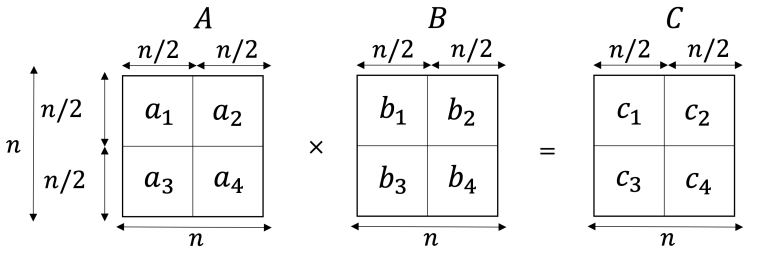
1.3 Strassen's Matrix Multiplication Algorithm

Overview and Motivation

- Let A and B be two $n \times n$ matrices, assume n is a power of 2
- Want to compute $C = A \cdot B$
- Simple algorithm running time is $\Theta(n^3)$ by iterating through each row of A, each column of B, and each pair of elements to multiply
- We want to do better than $\Theta(n^3)$

Strassen's Algorithm

- Simple divide and conquer requires 8 subproblems, each with half the size



- We can regroup the terms and achieve 7 subproblems

- $m_1 = (a_2 - a_4) \cdot (b_3 + b_4)$
- $m_2 = (a_1 + a_4) \cdot (b_1 + b_4)$
- $m_3 = (a_1 - a_3) \cdot (b_1 + b_2)$
- $m_4 = (a_1 + a_2) \cdot b_4$
- $m_5 = a_1 \cdot (b_2 - b_4)$
- $m_6 = a_4 \cdot (b_3 - b_1)$
- $m_7 = (a_3 + a_4) \cdot b_1$

$c_1 = a_1 \cdot b_1 + a_2 \cdot b_3$
$c_2 = a_1 \cdot b_2 + a_2 \cdot b_4$
$c_3 = a_3 \cdot b_1 + a_4 \cdot b_3$
$c_4 = a_3 \cdot b_2 + a_4 \cdot b_4$

instead of computing c_1, c_2, c_3, c_4 like this...

- $c_1 = m_1 + m_2 - m_4 + m_6$
- $c_2 = m_4 + m_5$
- $c_3 = m_6 + m_7$
- $c_4 = m_2 - m_3 + m_5 + m_7$

Check it at home (when bored ☺)

- $T(n) = 7 \cdot T(n/2) + c \cdot n^2$ and so $a = 7, b = 2, d = 2$
- $a > b^d \implies T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.81}\right)$

Lifting the Assumption

- When n is not a power of 2, we can embed each matrix in a matrix whose dimension is the next power of 2
- In the worst case, this doubles the size of the problem n to $2n$
- And so $T(n) = \Theta\left((2n)^{\log_2 7}\right) = \Theta\left(2^{\log_2 7} n^{\log_2 7}\right) = \Theta\left(7n^{2.81}\right) = \Theta\left(n^{2.81}\right)$

1.4 Order Statistics

Overview and Motivation

- Let $A = a_1, a_2, \ldots, a_n$ (unsorted)
- Finding the average/max/min takes linear time
- We want to find *median* in linear time
- Trivial algorithm: sort A, then get the median $\implies \mathcal{O}(n \log n)$
- More general problem: `Select(A, k)`
  - Input: $A = a_1, a_2, \ldots, a_n$ where $1 \leq k \leq n$
  - Output: kth smallest element of A (i.e. element of *rank* k)
  - Assume that the a_is are distinct for simplicity

Pseudocode for `Select(A, k)`

```
1       if |A| = 1 then return A[1]
2       else
3           s <- arbitrary element of A (s is the 'splitter')
4           A- <- sublist of A with elements < s
5           A+ <- sublist of A with elements > s
6           s_rank <- |A-| + 1
7           if k < s_rank then return Select(A-, k)
8           if k = s_rank then return s
9           if k > s_rank then return Select(A+, k - s_rank)
```

- $T(n) = T[\max(|A^-|, |A^+|)] + cn$
- If the splitter s is the ith smallest in A:

$$T(n) = T[\max(i - 1, n - i)] + cn$$

- Worst choice for s: the min/max of A

$$T(n) = T(n - 1) + cn = \Theta(n^2)$$

- Best choice for s: the median of A

$$T(n) = T\left(\frac{n}{2}\right) + cn = \Theta(n)$$

- Notice for any $b > 1$ (even 1.01):

$$T(n) = T\left(\frac{n}{b}\right) + cn \implies a = 1, b > 1, d = 1 \implies a < b^d \implies T(n) = \Theta(n)$$

Select Splitter s uniformly at random

- Definition: s is a *good splitter* if
  - s is greater than $1/4$ elements of A, and
  - s is smaller than $1/4$ elements of A
- A good splitter reduces input size from n to $\leq 3/4n$
- Half of the elements are good splitters
- $P(\text{choosing a good splitter}) = p = 1/2$
- $P(\text{choosing a bad splitter}) = 1 - p = 1/2$

Expected Running Time

- The expected number of trials (splitter selections) till obtaining a good splitter is 2
- Assume that we start from phase 0 with input size n; whenever we get a good splitter we progress to the next phase, and the input size is $\leq 3/4$ of the previous phase
- Phase j: input size $\leq (3/4)^j n$
- Random variable y_j: number of recursive calls in phase j
  - Notice that $E[y_j] = 2 \quad \forall j$
- Random variable x_j: number of steps by all recursive calls in phase j
  - Total number of steps by `Select(A, k)` is $x = x_0 + x_1 + \cdots$

$$\begin{aligned} x_j &= \text{Number of recursive calls in phase } j \times \text{Number of steps in each recursive call in phase } j \\ &\leq y_j \times c\left(\frac{3}{4}\right)^j n \\ E[x_j] &\leq E[y_j] \times c\left(\frac{3}{4}\right)^j n \\ &\leq 2c\left(\frac{3}{4}\right)^j n \qquad \text{Since } E[y_j] = 2 \end{aligned}$$

- Want to find $E[x] = E[x_0] + E[x_1] + \cdots$

$$\begin{aligned} E[x] &= \sum_j E[x_j] \\ &\leq \sum_j^{\infty} 2c\left(\frac{3}{4}\right)^j n \\ &= 2cn \sum_j^{\infty} \left(\frac{3}{4}\right)^j \\ &= 2cn \cdot \frac{1}{1 - \frac{3}{4}} \\ &= 8cn \\ &= \mathcal{O}(n) \end{aligned}$$

Deterministic Algorithm for `Select(A, k)`

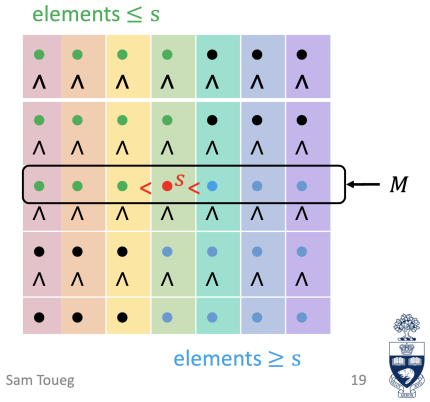
- `Select(A, k)`

```
1       if |A| <= 5 then sort A and return kth smallest
2       find a good splitter s (deterministically)
3       A- <- sublist of A with elements < s
4       A+ <- sublist of A with elements > s
5       s_rank <- |A-| + 1
6       if k < s_rank then return Select(A-, k)
7       if k = s_rank then return s
8       if k > s_rank then return Select(A+, k - s_rank)
```

- Find a good splitter s

```
1           Partition n elements of A into n / 5 groups of size 5 each
2           Find median of each group by sorting the group
3           M <- list of these medians
4           s <- Select(M, ceil(|M| / 2)) # find median of M
```

- s is a good splitter
  - It is greater than $1/4$ elements of A
  - It is smaller than $1/4$ elements of A



- Deterministic `Select(A, k)`

```
1           if |A| <= 5 then sort A and return kth smallest
2           partition n elements of A into n / 5 groups of size 5 each
3           find median of each group by sorting the group
4           M <- list of these medians
5           s <- Select(M, ceil(|M| / 2)) # find median of M
6           A- <- sublist of A with elements < s
7           A+ <- sublist of A with elements > s
8           s_rank <- |A-| + 1
9           if k < s_rank then return Select(A-, k)
10          if k = s_rank then return s
11          if k > s_rank then return Select(A+, k - s_rank)
```

  - Lines 2-4: $\mathcal{O}(n)$

  - Line 5: $T\left(\lceil \frac{n}{5} \rceil\right)$
  - Lines 6-8: $\mathcal{O}(n)$
  - Lines 9-11: $T\left(\lfloor \frac{3n}{4} \rfloor\right)$
- Worst case running time is $T(n) = T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\lfloor \frac{3n}{4} \rfloor\right) + cn$
- This is $\mathcal{O}(n)$ (prove by complete induction)

Group Size

- With groups of 5, total size of subproblems is $n/5 + 3n/4 = 19n/20 < n$
  - $n/5$ is selecting the median of $n/5$ medians
- With groups of size x:
  - Total size of subproblems is $n/x + 3n/4$, which we want to be $< n$
  - Holds if $x > 4$

1.5 Closest Pair of Points

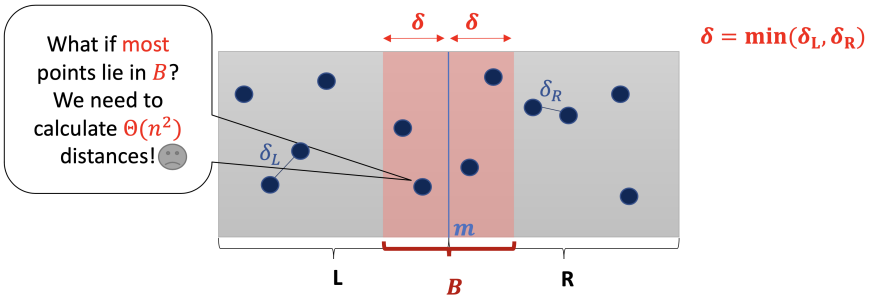
Overview and Motivation

- Input: set P of n points on the plane
- Output: Closest pair of points $p, q \in P$, i.e. $d(p, q) \leq d(p', q'), \forall p', q' \in P$
- Trivial solution: compute the distance between *every* pair $p, q \in P$, which is $\mathcal{O}(n^2)$

1-Dimensional Case

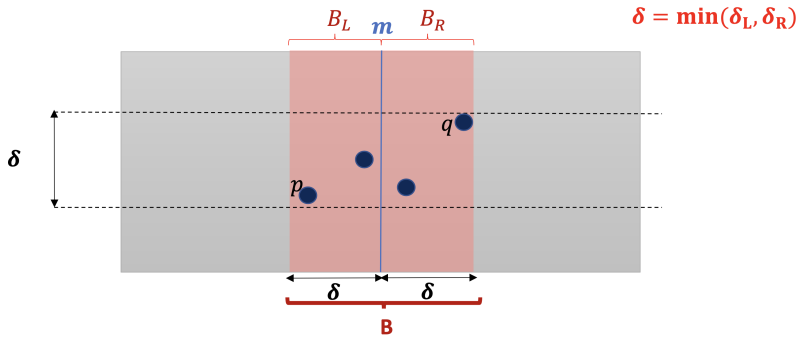
- Divide: axis into two halves
- Conquer: recursively find closest pair in each half
- Combine: find the distance between the pair that spans two halfs
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c = \mathcal{O}(n)$ by Master Theorem

2-Dimensional Case



- Divide: points in roughly equal halves by drawing a vertical line to m
- Conquer: recursively find the closest pair in each half, whose distances are δ_L, δ_R, respectively
- Combine: find the closest pair $(p, q), p \in L, q \in R$

Let $p = (x_p, y_p) \in B_L$ and $q = (x_q, y_q) \in B_R$ with $y_p \leq y_q$. If $d(p, q) < \delta$, then there are at most *six* other points $(x, y) \in B$ such that $y_p \leq y \leq y_q$



Proof. Let $S_L = \{p' = (x, y) : p' \neq p \in B_L \wedge y_p \leq y \leq y_q\}$ and $S_R = \{q' = (x, y) : q' \neq q \in B_R \wedge y_p \leq y \leq y_q\}$. Assume by contradiction that $|S_L| + |S_R| \geq 7$. Without loss, we assume that $|S_L| \geq 4$. The $\delta \times \delta$ square inside B_L contains at least $4 + 1 = 5$ points. By Pigeonhole Principle, one of the four $\frac{\delta}{2} \times \frac{\delta}{2}$ squares has at least 2 points, denoted u, v. Consequently,

$$d(u, v) \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{1}{\sqrt{2}}\delta < \delta$$

However, $u, v \in L$ and $\delta = \min(\delta_L, \delta_R)$, so $\delta \leq d(u, v)$, which is a contradiction.

□

- When the algorithm scans the points of B from bottom to top to find the pairs $p \in B_L, q \in B_R$ with $d(p, q) < \delta$, for each point $r \in B$ it only needs to look at the next 7 nodes up from $r \in B$

Pseudocode

```
ClosestPair(P)
    # P is a set of (2+) points on the plane, given by their x- and y-coordinates
    Px := the list of points in P, sorted by x-coordinate
    Py := the list of points in P, sorted by y-coordinate
    return RCP(Px, Py)

RCP(Px, Py)
    # Px, Py are lists of the same set of (2+) points, sorted by x- and y-coordinate, resp.
    if |Px| <= 3 then calculate all pairwise distances and return closest pair
    else
        Lx := first half of Px # O(n)
        Rx := second half of Px # O(n)
        m := (max x-coord in Lx + min x-coord in Rx) / 2 # O(1)
        Ly := sublist of Py consisting of points in Lx # O(n)
        Ry := sublist of Py consisting of points in Rx # O(n)
        (pL, qL) := RCP(Lx, Ly) # T(n/2)
        (pR, qR) := RCP(Rx, Ry) # T(n/2)
        delta := min(d(pL, qL), d(pR, qR)) # O(1)
        if d(pL, qL) = delta then (p*, q*) := (pL, qL) else (p*, q*) := (pR, qR) # O(1)
        B := sublist of Py consisting of points whose x-coord are within delta of m # O(n)
        for each p in B (in order of appearance in B) do # O(n)
            for each of the next (up to) 7 points q after p in B do # O(1)
                if d(p, q) < d(p*, q*) then (p*, q*) := (p, q)
        return (p*, q*) # O(1)
```

- $T(n) = 2 \cdot T(n/2) + n \cdot c = \mathcal{O}(n \log n)$ by Master Theorem

2 Greedy Algorithms

2.1 Intro

Optimization problem: given an input, find an output (a "solution") that maximizes/minimizes an objective function f under some constraints

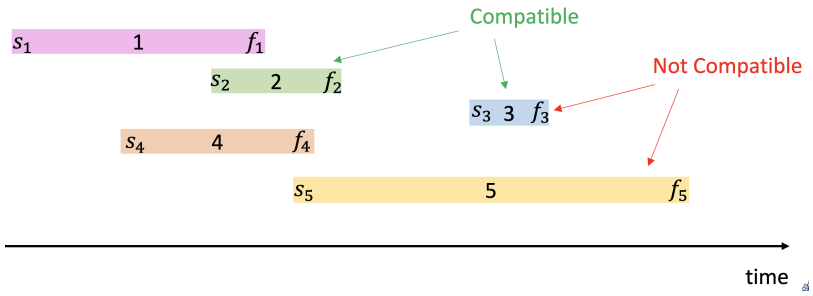
Greedy algorithm (a.k.a. "Myopic" algorithm):

- Build the solution *incrementally* in stages
- At each stage, *extend* the solution
  - *Greedily*: in a way that maximizes the *immediate* benefit of f
  - *Irrevocably*: no backtracking/regretting/changing previous decisions
- Gives optimal solutions for some problems (e.g. Prim's and Kruskal's MST algorithms)
- In general, it is not so good, since it optimizes the "next move" without seeing the big picture

2.2 Interval Scheduling

Problem

- Input: n intervals ("jobs"), interval j starts at time s_j and finishes at time f_j
- Output: maximum-size set of intervals that do not overlap (i.e. are "compatible")



Interval Scheduling Algorithms

- *Brute force*: try every subset of n intervals and take the largest set of compatible intervals (i.e. the largest "feasible set") $\implies \mathcal{O}(2^n)$
- *Greedy algorithm* (general scheme):

```
sort intervals in some order
A <- empty set # set of compatible intervals
for each interval i in sorted order do
    if i is compatible with intervals in A then
        A <- A union {i}
return A
```

- Sorting by earliest finish time (EFT) works (i.e. increasing order of f_j)
- Intuition: taking the interval that finishes earliest first gives more space to the remaining intervals

Greedy algorithm with EFT is optimal.

Proof. Suppose for a contradiction that this greedy algorithm is *not* optimal. Say greedy selects intervals $i_1, i_2, \ldots, l_k$ sorted by increasing finish time. Consider an optimal schedule $OPT : j_1, j_2, \ldots, j_m$ also sorted by finish time such that OPT matches the greedy schedule for as long as possible, i.e. $j_1 = i_1, j_2 = i_2, \ldots, j_r = i_r$ for the greatest possible r.

Consider the schedule $S : i_1, i_2, \ldots, i_r, i_{r+1}, j_{r+2}, \ldots, j_m$. S is still feasible since $f_{i_{r+1}} \leq f_{j_{r+1}}$, and so all intervals in S are compatible. S is still optimal since it has the same number of intervals m as OPT. S matches the greedy schedule for one more interval than OPT. This is a contradiction. □

Pseudocode

```
sort intervals in order of increasing finish time
A <- empty set
F <- -inf
for each interval i in sorted order do
    if s[i] >= F then
        A <- A union {i}
        F <- f[i]
return A
```

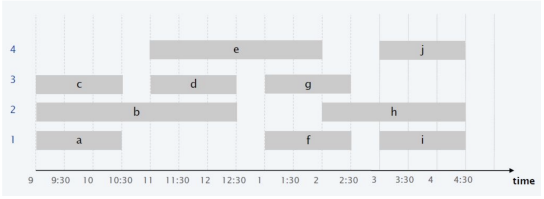
- Running time: $\mathcal{O}(n \log n)$

2.3 Interval Partitioning

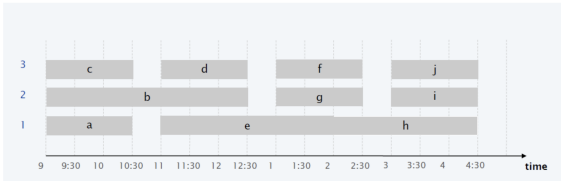
Problem

- Given a set of lecture time intervals (for many courses)
- We want to schedule them into as *few* classrooms as possible

• This schedule uses 4 classrooms for 10 lecture intervals



• This schedule uses only 3 classrooms for the *same* 10 lecture intervals



- Input: n intervals ("jobs"), interval j starts at time s_j and finishes at time f_j
- Output: group intervals into *fewest* partitions such that the intervals in each partition are compatible, i.e. do not overlap

Interval Partitioning Algorithms

- Greedy algorithm (general scheme):

```
sort intervals in some order
for each interval i in sorted order do
    if i is compatible with some existing partition A then
        A <- A union {i}
    else
        insert i into a new partition
return all partitions
```

- Sorting by earliest start time (EST) works (i.e. increasing order of s_j)

Greedy algorithm with EST is optimal.

Proof. Define the *depth* at time t as the number of intervals that contain time t, and the *maximum depth* $d_{\max}$ as the maximum depth over all times. It is clear that the number of partition needed $\geq d_{\max}$. We will show that the EST greedy algorithm creates only $d_{\max}$ partitions so that it is optimal.

Let d be the number of partitions created by the EST greedy algorithm. Partition d was created because there was an interval j that *overlaps* with some *previously scheduled* interval in each of $d-1$ other partitions. Since we sorted by start time, all these $d-1$ intervals *start at/before* s_j. Since interval j overlaps with them, all these $d-1$ intervals *finish after* s_j. So at time s_j, there are d overlapping intervals, i.e. depth $\geq d$, thus $d_{\max} \geq d$. Since all feasible schedules must have at least $d_{\max} \geq d$ partitions, creating d partitions is optimal. □

- We showed that there are d overlapping intervals, therefore the max depth $d_{\max} \geq d$. Since we created d partitions that is the lower bound for d_{max}, it is optimal.

Implementation Details

- Store partitions as a tuple [key, intervals] in a *priority queue*
- Key: finish time of the last interval inserted in the partition
- Intervals: set of intervals in the partition

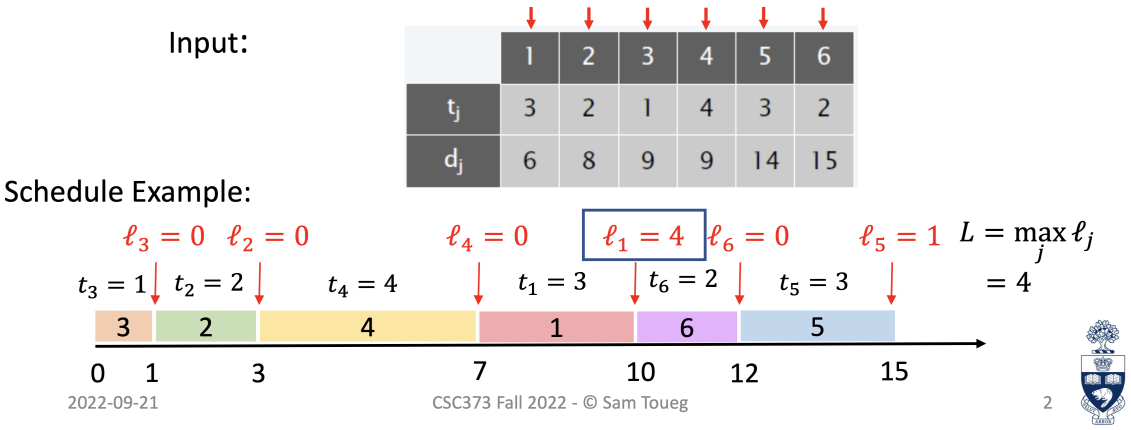
```
1   create an empty min heap H
2   sort all intervals in order of increasing start time # O(n log n)
3   for each interval i in sorted order do # O(n) heap operations, O(n log n) time
4       P <- H.FindMin()
5       if P is Null or s[i] < P.key then # start before earliest finish
6           create a new partition A with A.key = f[i], A.intervals = {i}
7           H.Insert(A)
8       else
9           A <- H.ExtractMin()
10          A.intervals <- A.intervals union {i}
11          A.key <- f[i]
12          H.Insert(A)
13  return H
```

- Running time: $\mathcal{O}(n \log n)$

2.4 Minimize Lateness Scheduling

Problem

- Input: n intervals ("jobs"): $1, 2, \ldots, n$
  - Interval j requires t_j units of time and has deadline d_j
  - If interval j is scheduled to start at time s_j, it will finish at time $f_j = s_j + t_j$
  - Lateness: $l_j = \max(0, f_j - d_j)$
- Output: schedule that *minimizes* the *maximum lateness*, $L = \max_j l_j$

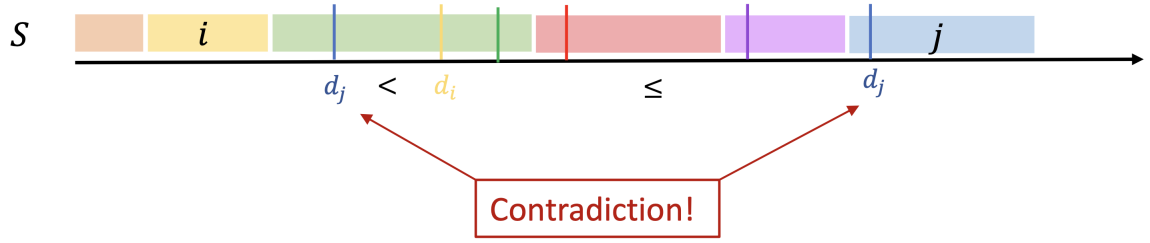


Fact 1: there is an optimal schedule with no gaps

- Brute force algorithm: try all possible permutations of n intervals with no gaps $\implies \mathcal{O}(n!)$
- Greedy algorithm: sort intervals in some order, then schedule all intervals in this order with no gaps
- Sorting in the order of *increasing deadline* d_j works (earliest deadline first, EDF)
- Observation 1: the greedy EDF schedule has *no gaps*
- Define an *inversion* as two intervals i, j such that $d_i > d_j$ but i is scheduled before j
- Observation 2: the greedy EDF schedule has *no inversions*

Lemma 2.1. *If a schedule S with no gaps has an inversion, then S has a pair of inverted intervals that are adjacent*

Proof. Suppose for a contradiction that S has an inversion and that S does not have adjacent intervals that are inverted. Then



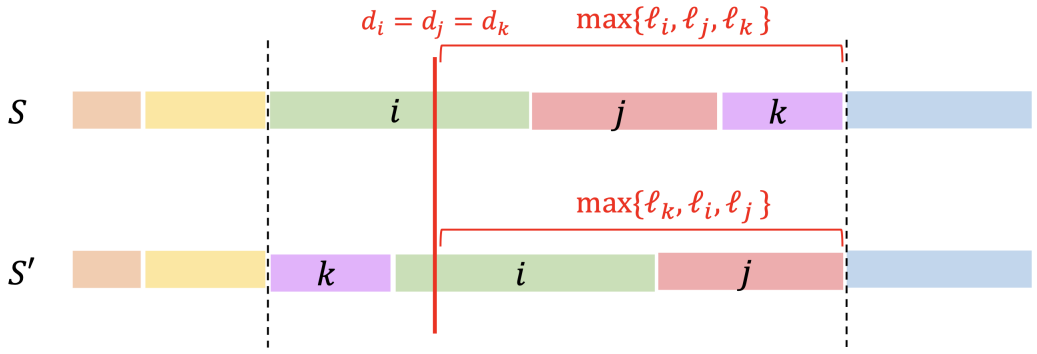
□

Lemma 2.2. *All schedules with no gaps and no inversions have the same lateness.*

Proof. Let S and S' be two *distinct* schedules with no gaps and no inversions. Note that S and S' differ only by the schedule of intervals with the *same deadline*. Consider the intervals with the same deadline:

- They are adjacent in both S and S'
- As a group, they have the same max lateness

The other intervals have the same lateness, so S and S' have the same lateness.



□

Lemma 2.3. *Swapping adjacent inverted intervals does not increase lateness and reduces the number of inversions by 1*

Proof. Let i and j denote two adjacent inverted intervals in schedule S, where i is scheduled before j in S but deadline $d_i > d_j$. By swapping i and j, we get a new schedule S' with 1 fewer inversion. Let l and l' denote lateness before/after swap. It is clear that $l_k = l'_k \quad \forall k \neq i, j$. We claim that $l_j \geq l'_j$ and $l_j \geq l'_i$.

- $l_j \geq l'_j$ because j finishes earlier in S' (i.e. after the swap)
- $l_j = f_j - d_j \geq f'_i - d_i = l'_i$ (i has a later deadline)

$$L = \max\left\{l_i, l_j, \max_{k \neq i,j} l_k\right\} \geq \max\left\{l'_i, l'_j, \max_{k \neq i,j} l'_k\right\} = L'$$

□

The EDF greedy algorithm is optimal

Proof. Suppose for a contradiction that the EDF schedule S is not optimal. Note that S has no gaps and has no inversions. Let S^* be an *optimal* schedule with no gaps that has the *fewest inversions* among all optimal schedules.

- Case 1: S^* has no inversions, then both S and S^* has no gaps and no inversions. By Lemma 2.2, S has the same lateness as S^*, and so S is also optimal, which is a contradiction.
- Case 2: S^* has at least 1 inversion. By Lemma 2.1, S^* has an adjacent inversion (i, j). We swap i and j. By Lemma 2.3, this swap does not increase lateness and reduces the number of inversions by 1. So the new schedule remains optimal and has 1 fewer inversion than S^*, which is a contradiction (since S^* has the fewest inversions).

□

2.5 Huffman Code

Overview and Motivation

- *Alphabet* Γ: a set of n symbols
  - E.g. $\Gamma = \{a, b, \ldots, z\}, n = |\Gamma| = 26$
- *Text*: a string over Γ, e.g. "the"
- *Coding*: maps each symbol $x \in \Gamma$ to a *unique binary string*, called "codeword" (or simply "code") of x
- Two types of symbol coding:
  - *Fixed* length

– *Variable* length

Fixed-Length Code

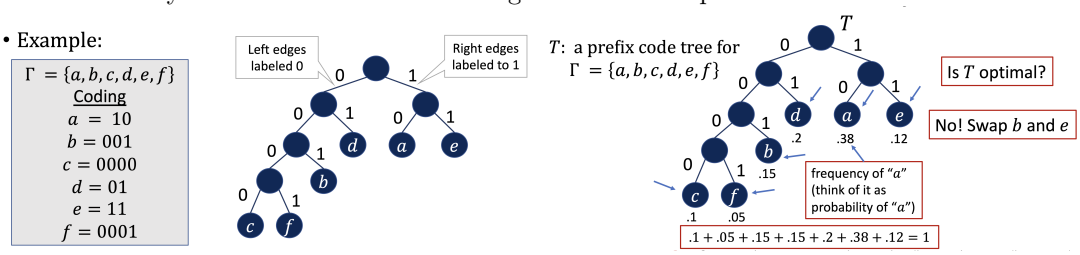
- For n distinct symbols, each codeword has $\lceil \log_2 n \rceil$ bits
  - E.g. ASCII code for 256 distinct characters use 8 bits per character
- Easy to decode
- Not optimal in terms of text coding length
  - E.g. There are much more "e"s than "z"s in English text
  - It would be better to give a shorter code to "e" and a longer code to "z"
- Goal: minimize the length of text coding
- Idea: give shorter codes to frequent text symbols

Variable-Length Code

- Different codewords may have different lengths
- Pro: potential to save space
- Con: harder to decode, can be ambiguous if not done correctly
- **Prefix code**: no codeword is a prefix of any other
  - We can scan from left to right till we find a codeword of some symbol
  - Unambiguous decoding

Prefix Code as a Binary Tree

- Every prefix code can be represented as a *binary tree*
- Symbols of Γ: leaves of the binary tree
- Codeword for symbol x: concatenation of edge labels on the path from the root to leaf x



- *Average depth* of T is $2(0.2 + 0.38 + 0.12) + 3(0.15) + 4(0.1 + 0.05) = 2.45$
- *Average length* of a codeword is 2.45 bits/symbol
- In comparison, a fixed-length code needs 3 bits/symbol (to code 6 symbols)

Prefix Code Problem

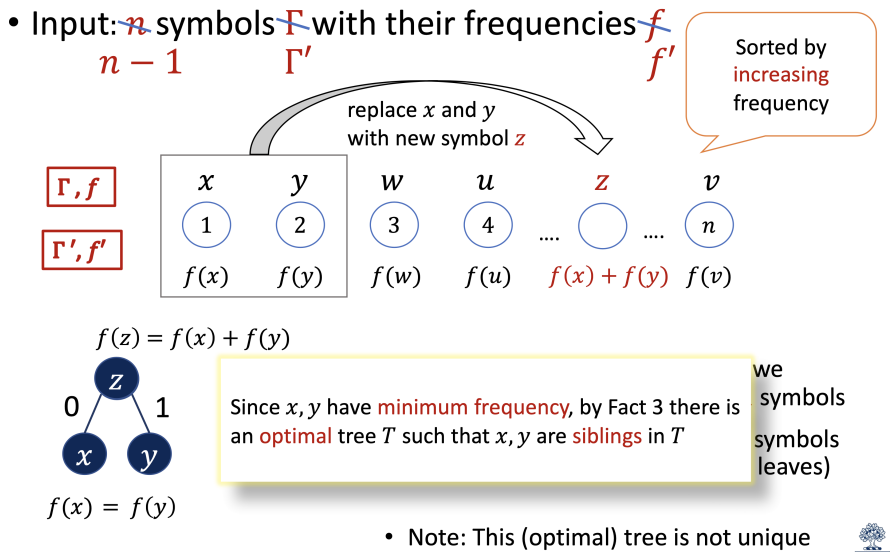
- Input: set of symbols Γ with their frequencies f
  - $\forall x \in \Gamma$, $f(x)$ = frequency of x, where $\sum_{x \in \Gamma} f(x) = 1$

- Output: prefix code tree T that is *optimal* in the sense that the average length of a codeword is minimized for Γ, f
- Weighted average depth of T:
$$AD(T) = \sum_{x \in \Gamma} f(x) \cdot \text{depth}_T(x)$$
where $\text{depth}_T(x)$ is the depth of x in T, as well as the length of code of x in T
- Optimal solution: a tree T with *minimum* $AD(T)$

Facts

1. An optimal tree is a *full* binary tree, i.e. each internal node has 2 children
   - For an internal node with 1 child, we can remove that connection (i.e. remove that extra representation bit)
2. In an optimal tree T, for any two symbols x, y, $f(x) < f(y) \implies \text{depth}_T(x) \geq \text{depth}_T(y)$
3. If x, y are symbols with *minimum frequency*, then there is *optimal* tree T such that x, y are siblings and are at *maximum depth*
   - Implied by Facts 1 and 2

Huffman's Algorithm



- From instance Γ with n symbols, we created an instance Γ' with $n-1$ symbols
- Do this recursively until we get 2 symbols (then we have 1 root with 2 leaves)

Theorem 2.4. *Huffman's Algorithm is optimal.*

Proof. It suffices to show that for any set of symbols Γ and frequencies f, the tree T output by the algorithm has a **minimum average depth** $AD(T)$. We show by induction on the number of symbols n in Γ.

Base case: for $n = 2$ symbols, optimality is obvious since each symbol is at depth 1 (with codes 0 and 1, respectively).

Induction step: for $n > 2$, assume

- (IH) for all Γ with $n-1$ symbols and all f, the algorithm produces an optimal tree.

Let Γ be an alphabet with n symbols, $f(x)$ be a frequency for each $x \in \Gamma$, and H be the tree output by the algorithm on input (Γ, f). We want to show that H is optimal.

Our algorithm constructed H by replacing 2 symbols $x, y \in \Gamma$ of minimum frequency with new symbol z:

- $\Gamma' = \Gamma - \{x, y\} \cup \{z\}$ (replace x and y with a symbol z)
- $f'(z) = f(x) + f(y)$ (with frequency $f(x) + f(y)$)
- $f'(\alpha) = f(\alpha), \forall \alpha \in \Gamma - \{x, y\}$
- x and y are siblings (with parent z) in H
- Since H' has $n-1$ symbols, by IH, H' is optimal for (Γ', f')

Notice that

- $AD(H) = C + d \cdot f(x) + d \cdot f(y)$
- $AD(H') = C + (d-1) \cdot (f(x) + f(y))$
- Therefore $AD(H) = AD(H') + f(x) + f(y)$

Since x, y are symbols of Γ of min frequency in f, by Fact 3, there is an optimal tree T for (Γ, f) such that x, y are siblings. Let T' be the tree constructed from T by removing x, y and replacing their parent with symbol z. Then T' is a prefix code tree for (Γ', f'), where (Γ', f') are defined as before. Then $AD(T) = AD(T') + f(x) + f(y)$. Finally,

$$\begin{aligned} AD(H') &\leq AD(T') \qquad \text{Since } H' \text{ is optimal for } (\Gamma', f') \\ AD(H') + f(x) + f(y) &\leq AD(T') + f(x) + f(y) \\ AD(H) &\leq AD(T) \end{aligned}$$

Since T is optimal for (Γ, f), so is H. □

Pseudocode

```
Huffman(n, f) # symbols 1, 2, ..., n with frequencies f(1), f(2), ..., f(n)
    for i = 1 to n do # O(n)
        create a leaf node for symbol i
        H(i) = (i, f(i)) # Min heap array with key f(i)
    BuildMinHeap(H) # O(n)
    for i = n + 1 to 2n - 1 do # n - 1 iterations, O(n log n)
        (x, f(x)) = ExtractMin(H)
        (y, f(y)) = ExtractMin(H)
        create a node labelled i with children x and y
        Insert(H, (i, f(x) + f(y)))
```

- Running time: $\mathcal{O}(n \log n)$

2.6 Dijkstra's Single-Source Shortest Paths Algorithm

Single-Source Shortest Paths Problem

- Input:
  - Directed graph $G = (V, E)$

- Each edge (u, v) has a *non-negative* weight ("length") w_{uv}
- Source node $s \in V$

- Output: length of the shortest path from s to *each* node $t \in V$

Dijkstra's Algorithm

- Let R be a subset of V that includes source s
- A path $s \rightsquigarrow v$ is an *R-path* if it contains only nodes in R except possibly for v
- The algorithm maintains a set of nodes $R \subseteq V$ and for each node v a value $d(v)$, such that
  1. $\forall v \in R$, $d(v)$ is the length of the shortest $s \rightsquigarrow v$ path
  2. $\forall v \in V - R$, $d(v)$ is the length of the shortest $s \rightsquigarrow v$ R-path
- In each iteration, the algorithm adds one more node to R and maintains the above properties
- Let u be the node with minimum d-value in $V - R$
  - We know that $d(u)$ is the length of the shortest $s \rightsquigarrow u$ R-path
  - We claim that $d(u)$ is the length of the shortest $s \rightsquigarrow u$ path

    Proof. Suppose for a contradiction that P is a *shorter* $s \rightsquigarrow u$ path such that $l(P) < d(u)$. Let v be the *first* node of the path P in $V - R$, so that P consists of two parts
    * $P_1 : s \rightsquigarrow v$ is an R-path
    * $P_2 : v \rightsquigarrow u$

    Then

$$\begin{aligned} l(P) &= l(P_1) + l(P_2) \\ &\geq l(P_1) && \text{Since edge weights are non-negative} \\ &\geq d(v) && \text{Since } d(v) \text{ is the length of the shortest } s \rightsquigarrow v \ R\text{-path} \\ &\geq d(u) && \text{Since } u \text{ is the node with minimum } d\text{-value in } V - R \end{aligned}$$

    which is a contradiction. □
- We can move u to set R, without changing $d(u)$, and still satisfy property 1
- We must update the $d(v)$ of each $v \in V - R$. We have two cases:
  - Some shortest $s \rightsquigarrow v$ R-path does not contain u, in this case $d(v)$ does not change
  - All shortest $s \rightsquigarrow v$ R-paths contains u, in this case $d(v) = d(u) + w_{uv}$
- For every $v \in V - R$, we update $d(v)$ as follows:

$$d(v) = \min(d(v), d(u) + w_{uv})$$

Pseudocode (naive implementation)

```
1       R <- {s}
2       d(s) <- 0
3       for each node v != s do # O(n)
4           if (s, v) is an edge then
5               d(v) <- w[s, v]
6           else
7               d(v) <- inf
```

```
        while R != V do # O(n) iterations
            u <- node not in R with minimum d-value # O(n), expensive
            R <- R union {u}
            for each node v s.t. (u, v) is an edge do # O(m) in total
                if d(u) + w[u, v] < d(v) then
                    d(v) <- d(u) + w[u, v]
                    p(v) <- u # predecessor of v
```

- Worst case running time for a graph with n nodes and m edges: $\mathcal{O}(n^2 + m) = \mathcal{O}(n^2)$
- We can keep nodes that are not in R in a min-heap with their d-values as keys

Pseudocode (using a min-heap)

```
        R <- {s}
        d(s) <- 0
        for each node v != s do # O(n)
            if (s, v) is an edge then
                d(v) <- w[s, v]
            else
                d(v) <- inf
        NR <- min heap of all the nodes v != s with their d-value as keys # O(n)
        while R != V do # O(n) iterations
            u <- NR.ExtractMin() # O(log n)
            R <- R union {u}
            for each node v s.t. (u, v) is an edge do # O(m) in total
                if d(u) + w[u, v] < d(v) then
                    d(v) <- d(u) + w[u, v]
                    p(v) <- u # predecessor of v
                    NR.ChangeKey(v, d(v)) # O(m log n) in total
```

- Running time: $\mathcal{O}(n + n \log n + m \log n) = \mathcal{O}(m \log n)$
- Better for sparse graphs, i.e. $m \approx n$
- The naive implementation is better for dense graphs, i.e. $m \approx n^2$

Limitations

- Does not work with negative weights

3 Dynamic Programming

3.1 Introduction

Dynamic programming (DP) works on problems that have an *optimal substructure property*, where optimal solution of a problem can be computed efficiently from the optimal solutions of subproblems

- Break the problem into subproblems, solve each subproblems just *once*, and store their solutions
- When solving a subproblem, another subproblem that we already solved may reoccur
- Instead of recomputing its solution, we can just look up its previously computed solution
- *Memoization*: storing the solutions of subproblems

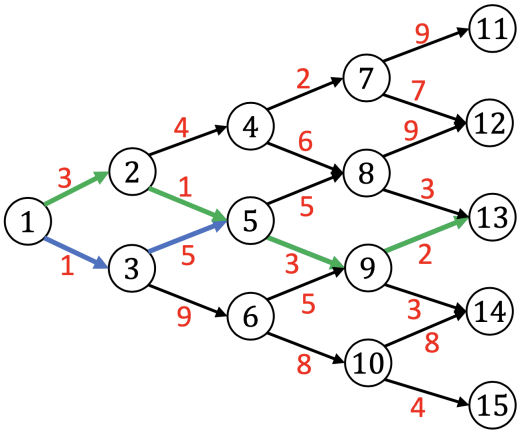
DP is different from Divide and Conquer

- Divide and conquer can be thought as a special case of DP where the subproblems that are solved *never "overlap"*
- No need for memoization to avoid recomputing some previously solved subproblems

3.2 DAG (Direct Acyclic Graph) Shortest Path

Problem

- Find shortest path from the root to *any leaf* in this weighted DAG



DP Idea

- Avoid re-solving the same subproblems multiple times

```
for i <- 15 to 11 do COST(i) <- 0 # leaf nodes
for i <- 10 to 1 do # internal nodes in bottom-up order
    COST(i) <- min(COST(nextup(i)) + costup(i), COST(nextdown(i)) + costdown(i))
```

- Time complexity is *linear* in the size of the problem

3.3 Weighted Interval Scheduling

Problem

- Input: n intervals ("jobs"), interval j starts at time s_j and finishes at time f_j, and has a weight w_j
- Output: subset S of compatible intervals with *maximum total weight* $\sum_{j \in S} w_j$
- If all w_j are equal, then it is the interval scheduling problem (See 2.2)
- If not, we cannot use the greedy approach

The DP Approach

- Convention: intervals are sorted by increasing finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$
- Define $p[j]$ as the largest index $i < j$ such that interval i is compatible with interval j (i.e. interval i finishes before interval j starts)
- Let S be an *optimal* subset of intervals (a "solution") for intervals $\{1, \ldots, n\}$
- Consider the last interval n:
  - Case 1: $n \notin S$, then S is an optimal subset of intervals for intervals $\{1, \ldots, n-1\}$
  - Case 2: $n \in S$, then S is $\{n\} \cup$ an optimal subset of intervals for intervals $\{1, \ldots, p[n]\}$
- Let $OPT(j)$ be the *maximum total weight* of compatible intervals in $\{1, \ldots, j\}$
  - For $j = 0, OPT(0) = 0$
  - For $j > 0$, interval j is either selected or not selected, so $OPT(j) = \max\{OPT(j-1), w_j + OPT(p[j])\}$
- Bellman equation:

$$OPT(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max\{OPT(j-1), w_j + OPT(p[j])\}, & \text{if } j > 0 \end{cases}$$

Dynamic Programming: Bottom-Up

- Find an *order* to compute subproblems so that their solutions are ready when needed

```
BOTTOM-UP(n weighted intervals):
    sort intervals by finish time: f[1] <= f[2] <= ... <= f[n] # O(n log n)
    compute p[1], p[2], ..., p[n] via binary search # O(n log n)
    OPT[0] <- 0
    for j = 1 to n do # O(n)
        OPT[j] <- max(OPT[j - 1], w[j] + OPT[p[j]]) # previously computed values, so O(1)
    return OPT[n]
```

- Time complexity: $\mathcal{O}(n \log n)$

Dynamic Programming: Top-Down

```
TOP-DOWN(n weighted intervals):
    sort intervals by finish time: f[1] <= f[2] <= ... <= f[n] # O(n log n)
    compute p[1], p[2], ..., p[n] via binary search # O(n log n)
    OPT[0] <- 0
    return TD-OPT(n) # O(n)
```

```
TD-OPT(j):
    if OPT[j] is not initialized do
        OPT[j] <- max(TD-OPT(j - 1), w[j] + TD-OPT(p[j]))
    return OPT[j]
```

- `OPT[j]` is computed at most once for each j, so `TD-OPT(n)` takes $\mathcal{O}(n)$ time
- Time complexity: $\mathcal{O}(n \log n)$

Finding the Optimal Subset of Intervals

- Can be done by computing simultaneously
  1. the maximum total weight, and
  2. a subset of intervals that given this weight

```
BOTTOM-UP(n weighted intervals)
    sort intervals by finish time: f[1] <= f[2] <= ... <= f[n]
    compute p[1], p[2], ..., p[n] via binary search
    OPT[0] <- 0; S[0] <- empty set
    for j = 1 to n do
        OPT[j] <- max{OPT[j - 1], w[j] + OPT[p[j]]}
        if OPT[j] = OPT[j - 1] then
            S[j] <- S[j - 1] # OPT does not include interval j
        else
            S[j] <- {j} union S[p[j]] # OPT includes interval j and the intervals
                in S[p[j]]
    return OPT[n], S[n]
```

- We can also compute it after computing the maximum weight array `OPT`

```
OPTIMAL-SET(OPT):
    S <- empty set
    i <- n
    while i != 0 do
        if OPT[i] = OPT[i - 1] then
            i <- i - 1 # OPT does not include interval i
        else
            S <- S union {i} # OPT includes interval i
            i <- p[i] # so it does not include intervals from p[i] + 1 to i - 1
    return S
```

3.4 Edit Distance

Background Information

- Want to know how similar are strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$
- Suppose we can *delete* or *replace* symbols in any string
- How many deletions and replacements does it take to match two strings?

Problem

- Input: strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$

  - Cost $d(a)$ for deleting symbol a
  - Cost $r(a, b)$ for replacing symbol a with b, assuming r is symmetric so $r(a, b) = r(b, a)$
- Output: the minimum *total cost* for matching X and Y

Idea

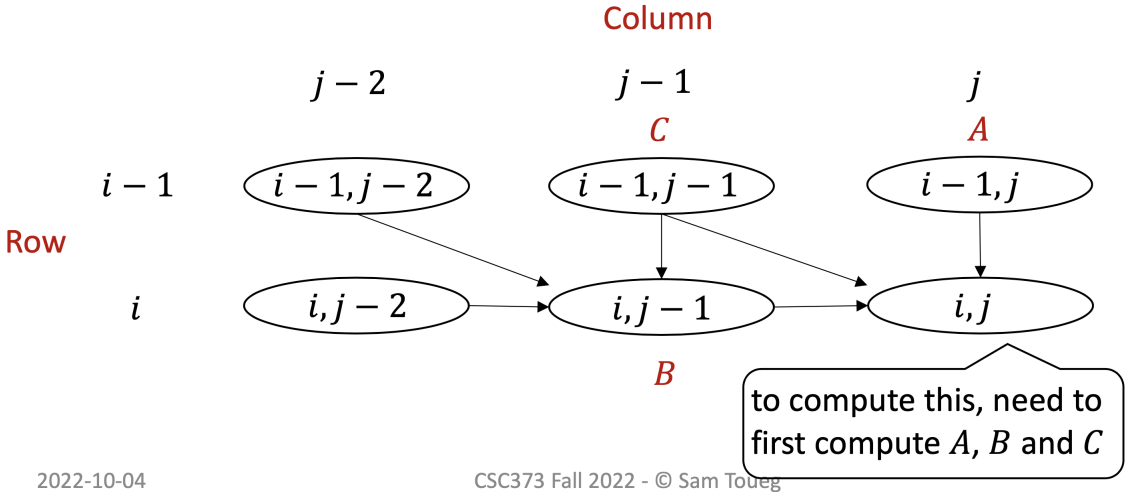
- Let $E(i, j)$ denote edit distance between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$
- Consider the *last* symbols x_i and y_j of X and Y, there are three cases:
  1. Delete x_i, and optimally match $x_1, \ldots, x_{i-1}$ with $y_1, \ldots, y_j$
     - $E(i, j) = d(x_i) + E(i - 1, j)$
  2. Delete y_j, and optimally match $x_1, \ldots, x_i$ with $y_1, \ldots, y_{j-1}$
     - $E(i, j) = d(y_j) + E(i, j - 1)$
  3. Match x_i with y_j, and optimally match $x_1, \ldots, x_{i-1}$ with $y_1, \ldots, y_{j-1}$
     - $E(i, j) = r(x_i, y_j) + E(i - 1, j - 1)$, where $r(x_i, y_j) = 0$ if $x_i = y_j$
- Denote

$$
\begin{aligned}
A &= d(x_i) + E(i - 1, j) \\
B &= d(y_j) + E(i, j - 1) \\
C &= r(x_i, y_j) + E(i - 1, j - 1)
\end{aligned}
$$

- Bellman equation:

$$
E(i, j) = \begin{cases} 0, & \text{if } i = 0 \wedge j = 0 \\ d(x_i) + E(i - 1, 0), & \text{if } i > 0 \wedge j = 0 \\ d(y_j) + E(0, j - 1), & \text{if } i = 0 \wedge j > 0 \\ \min\{A, B, C\}, & \text{if } i > 0 \wedge j > 0 \end{cases}
$$

- Representation:



Algorithm

```
1       EDITDIST(X = x[1], ..., x[m], Y = y[1], ..., y[n], d, r):
2           E(0, 0) = 0
3           for i = 1 to m do
```

```
            E(i, 0) = d(x[i]) + E(i - 1, 0)
        for j = 1 to n do
            E(0, j) = d(y[j]) + E(0, j - 1)
        for i = 1 to m do
            for j = 1 to n do
                A = d(x[i]) + E(i - 1, j)
                B = d(y[j]) + E(i, j - 1)
                C = r(x[i], y[j]) + E(i - 1, j - 1)
                E(i, j) = min{A, B, C}
        return E(m, n)
```

- $\mathcal{O}(mn)$ time, $\mathcal{O}(mn)$ space

Find the optimal edit

```
    Ops = empty set
    i = m; j = n
    while i > 0 and j > 0 do
        if E(i, j) = d(x[i]) + E(i - 1, j) then
            Ops = Ops union {delete x[i]}
            i = i - 1
        else if E(i, j) = d(y[j]) + E(i, j - 1) then
            Ops = Ops union {delete y[j]}
            j = j - 1
        else if E(i, j) = r(x[i], y[j]) + E(i - 1, j - 1) then
            Ops = Ops union {replace x[i] with y[j]}
            i = i - 1; j = j - 1
    if j = 0 then Ops = Ops union {delete x[k] | 1 <= k <= i}
    if i = 0 then Ops = Ops union {delete y[k] | 1 <= k <= j}
```

- Combining with the algorithm to obtain E, still $\mathcal{O}(mn)$ time

3.5 0-1 Knapsack

Overview and Motivation

- Given:
  - n items, item i has value $v_i > 0$ and weight $w_i > 0$
  - Weight capacity C
- Assume: $C, v_i, w_i \in \mathbb{Z}$
- Goal: find a subset S of items with maximum total value and total weight at most C
- 0-1 Knapsack: we can either take a complete item or not take it

Algorithm

- Let S be an *optimal* knapsack for items $1, 2, \ldots, n$ and capacity C
- Two possible cases for the last item n:
  1. $n \notin S$, then S is an optimal knapsack for $1, 2, \ldots, n-1$ and capacity C
  2. $n \in S$, then $S = \{n\} \cup S'$ and S' is an optimal knapsack for $1, 2, \ldots, n-1$ and capacity $C' = C - w_n$
- Subproblems to solve: $K(i, c)$ = value for optimal knapsack for $1, 2, \ldots, i$ and capacity c

- Bellman Equation:

$$K(i,c) = \begin{cases} \max\{K(i-1,c), K(i-1,c-w_i)+v_i\}, & \text{if } i>0 \wedge c \geq w_i \\ K(i-1,c), & \text{if } i>0 \wedge c < w_i \\ 0, & \text{if } i=0 \vee c=0 \end{cases}$$

Pseudocode

```
Knapsack(w[1], ..., w[n], v[1], ..., v[n], C):
    for c = 0 to C do K(0, c) <- 0 # i = 0, no items
    for i = 0 to n do K(i, 0) <- 0 # c = 0, no capacity
    for i = 1 to n do
        for c = 1 to C do
            if c < w[i] then K(i, c) <- K(i - 1, c)
            else K(i, c) <- max{K(i - 1, c), K(i - 1, c - w[i]) + v[i]}
    # Find the optimal set of knapsack items
    S <- empty set; i <- n; c <- C
    while i > 0 and c > 0 do
        if K(i, c) = K(i - 1, c) then i <- i - 1
        else S <- S union {i}; i <- i - 1; c <- c - w[i]
    return S
```

- There are $\mathcal{O}(nC)$ distinct $K(i,c)$ values to compute, where each is computed once and takes constant time to compute
- Total running time is $\mathcal{O}(nC)$

Pseudo-Polynomial Time

- The running time $\mathcal{O}(nC)$ is *not* polynomial time, because C is *exponential* in the number of bits to represent it, i.e. $C = 2^b$ where b is the number of bits
- This is called **pseudo-polynomial**, since it is polynomial in the input *values*
- It would be fully polynomial iff $P = NP$

Different Algorithm for 0-1 Knapsack

- Instead of $C, w_1, \ldots, w_n$ being small integers, we are told that $v_1, \ldots, v_n$ are small integers
- The maximum value that can be achieved with *all* the n items is $V = v_1 + \cdots + v_n$
- Let $K(i,v)$ be the minimum *capacity* needed to pack a total value of at least v from items $1, \ldots, i$, where $1 \leq i \leq n$ and $0 \leq v \leq V$
- We can use DP to compute $K(n,v)$ for all $0 \leq v \leq V$, and find maximum v such that $K(n,v) \leq C$
- To compute $K(i,v)$, consider the last item i
  1. If we don't choose i, then we need capacity $K(i-1,v)$
  2. If we choose i, then we need capacity $w_i + K(i-1, v-v_i)$
- Bellman equation:

$$K(i,v) = \begin{cases} 0, & \text{if } v \leq 0 \\ \infty, & \text{if } v>0 \wedge i=0 \\ \min\{K(i-1,v), w_i + K(i-1,v-v_i)\}, & \text{if } v>0 \wedge i>0 \end{cases}$$

- Running time: $\mathcal{O}(nV)$
- We can choose between $\mathcal{O}(nC)$ and $\mathcal{O}(nV)$

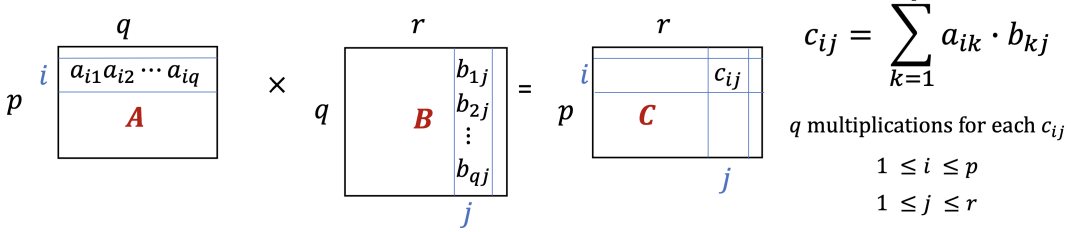
Approximate Solution

- We can find a good *approximate* solutions in polynomial time
- For any $\epsilon > 0$, we can get a knapsack value that is *at least* $(1-\epsilon)V^*$ where V^* is the optimal value, in time $\mathcal{O}\left(\text{poly}\left(n, \log C, \log V, \frac{1}{\epsilon}\right)\right)$
- An approximation algorithm:
  - Approximate all weights and values up to the desired precision
  - Solve knapsack on this approximate input using DP

3.6 Chain Matrix Multiplication

Multiplication of 2 Matrices

- Multiplying a $p \times q$ matrix A by a $q \times r$ matrix B gives a $p \times r$ matrix C



- To compute C, we do prq multiplications

Multiplication of Multiple Matrices

- Matrix multiplication is associative
- Cost of multiplication depends on which one is done first
- Want to find the *best order* to multiply a chain of matrices

Problem

- Input: the dimension $d_{i-1} \times d_i$ of every matrix A_i in $A_1, \ldots, A_n$
- Output: minimum number of multiplications to compute $A_1 \times \cdots \times A_n$

Dynamic Programming Idea

- Let $m(i,j)$ be the minimum number of multiplications to compute $A_i \times \cdots \times A_j$

$$m(i,j) = \begin{cases} 0, & \text{if } i = j \\ \min\limits_{i \le k < j} \{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\}, & \text{if } i < j \end{cases}$$

- Compute bottom-up to avoid recomputing the same $m(i,j)$ subproblems

Pseudocode

```
    for i <- 1 to n do m(i, i) <- 0 # O(n)
    for l <- 1 to n - 1 do # loop by increasing problem size l = j - i
        for i <- 1 to n - l do
            j <- i + l # O(n^2) inner loop iterations
            m(i, j) <- min{m(i, k) + m(k + 1, j) + d[i - 1] * d[k] * d[j]} over i <= k < j
            # Each min is over previously computed values, so takes O(n) time
    return m(1, n)
```

- Computes optimal multiplication in groups of 1, 2, all the way to n, so that we get the optimal solution for the entire chain of matrices
- Time complexity: $\mathcal{O}(n^3)$

3.7 Bellman-Ford's Single-Source Shortest Paths Algorithm

Problem

- Input:
  - Directed graph $G = (V, E)$
  - Each edge (u, v) has a weight ("length") w_{uv}, where $w_{uv} = \infty$ if $(u, v) \notin E$ and $w_{uu} = 0$ for all $u \in V$
  - Source node $s \in V$
- Output: length of a shortest path from s to every node $t \in V$
- When the edge weights are *non-negative*, we can use Dijkstra's algorithm (see 2.6)
- When the edge weights may be negative, Dijkstra's algorithm fails
- If G has a negative-length cycle, the problem is not well defined, since we can traverse the cycle arbitrarily many times to get arbitrary "short" paths

Lemma 3.1. *With no negative cycles, for every node $t \in V$, there is a shortest $s \rightsquigarrow t$ path that is* simple *(i.e. that contains no repeated nodes)*

Proof. Consider a shortest $s \rightsquigarrow t$ path. If it has a repeated node, then it contains a cycle. Remove the cycle, and the path has fewer repeated nodes and is not longer than the original path. Repeat until the path is simple. □

Corollary 3.1.1. *With no negative cycles, for every node $t \in V$, there is a shortest $s \rightsquigarrow t$ path with at most $n - 1$ edges*

Idea

- Consider a shortest $s \rightsquigarrow t$ path P with $t \neq s$
- Consider the node u that immediately precedes t in P, then $P' = P - (u, t)$ must be the shortest $s \rightsquigarrow u$ path
- Define $OPT(i, t)$ as the length of the shortest $s \rightsquigarrow t$ path using *at most* i edges
- If u is the predecessor of t in this path, $OPT(i, t) = OPT(i - 1, u) + w_{ut}$
- Therefore $OPT(i, t) = \min_{u \in V} \{OPT(i - 1, u) + w_{ut}\}$

- Bellman equation:

$$OPT(i,t) = \begin{cases} 0, & \text{if } i = 0 \wedge t = s \\ \infty, & \text{if } i = 0 \wedge t \neq s \\ \min\limits_{u \in V} \{OPT(i-1,u) + w_{ut}\}, & \text{if } i > 0 \end{cases}$$

- By Corollary 3.1.1, $OPT(n-1,t)$ is the length of the shortest path to t

Bellman-Ford Algorithm

```
1       OPT(0, s) <- 0
2       for all nodes t != s do OPT(0, t) <- inf
3       for all i = 1 to n - 1 do # O(n) iterations
4           for all nodes t do # O(m) total cost
5               OPT(i, t) <- min{OPT(i - 1, u) + w[u, t]} over all (u, t) in E or u = t
6       return OPT(n - 1, t) for every t in V
```

- Time complexity: $\mathcal{O}(nm)$
- Space complexity: $\mathcal{O}(n)$ since we don't need any early values of i
- To find the actual shortest paths, for each destination node $t \neq s$, we can keep track of the *current predecessor* of t in the shortest path $s \rightsquigarrow t$
  - Initially `p[t] <- NIL`
  - Update `p[t]` each time a shorter path to t is discovered

Detecting Negative Cycles

- Suppose G may have cycles with *negative length*
- Want to detect this

Lemma 3.2. $\forall t \in V, OPT(k,t) = OPT(k-1,t) \implies \forall t \in V, OPT(k+1,t) = OPT(k,t)$

Proof.

$$\begin{aligned} OPT(k+1,t) &= \min_{u \in V} \{OPT(k,u) + w_{ut}\} \\ &= \min_{u \in V} \{OPT(k-1,u) + w_{ut}\} \\ &= OPT(k,t) \end{aligned}$$

□

Corollary 3.2.1. $\forall t \in V, OPT(n,t) = OPT(n-1,t) \implies \forall t \in V, \forall n' \geq n, OPT(n',t) = OPT(n-1,t)$

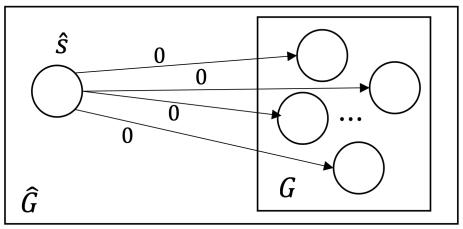
Lemma 3.3. *G has* no *negative length cycle reachable from s if and only if* $\forall t \in V, OPT(n,t) = OPT(n-1,t)$

Proof. "$\implies$" By Corollary 3.1.1, $\forall t \in V$, there is a shortest $s \rightsquigarrow t$ path with at most $n-1$ edges.

"$\impliedby$" Suppose $\forall t \in V, OPT(n,t) = OPT(n-1,t)$. Then by Corollary 3.2.1, $\forall t \in V, \forall n' \geq n, OPT(n',t) = OPT(n-1,t)$. Suppose for a contradiction that G has a negative length cycle reachable from s. Then we can find an $s \rightsquigarrow t$ path to some node t in this cycle, and we can make the length of $s \rightsquigarrow t$ smaller and smaller by repeating this negative cycle. This contradicts to Corollary 3.2.1, therefore G has no negative length cycle reachable from s. □

Detecting Negative Cycles

- Lemma 3.3 enables us to detech whether G has a negative cycle reachable from s
  - Run Bellman-Ford algorithm for n iterations
  - If $\forall t, OPT(n,t) = OPT(n-1,t)$, then no negative cycles are reachable from s
  - If $\exists t, OPT(n,t) \neq OPT(n-1,t)$, then there is a negative cycle reachable from s
- To detect whether G has a negative cycle that may *not* be reachable from s, we add a node $\hat{s}$ that connects to every other node with edge weight 0



detect negative cycles reachable from $\hat{s}$ by running B-F on $\hat{G}$

Lemma 3.4. *Suppose that $OPT(n,t) \neq OPT(n-1,t)$ for some node t. Let P be an $s \rightsquigarrow t$ path of length $OPT(n,t)$ and with at most n edges. Then*

1. *P contains a cycle*
2. *Every cycle on P has negative length*

Proof. Since $OPT(n,t) \neq OPT(n-1,t)$, clearly $OPT(n,t) < OPT(n-1,t)$. Let P be an $s \rightsquigarrow t$ path with length $OPT(n,t)$ and at most n edges. Since $OPT(n,t) < OPT(n-1,t)$, we know that P has exactly n edges and $n+1$ nodes, and that some node v repeats. This means that (1) holds because there is a v to v cycle.

We claim that such cycle has negative length, so that (2) also holds. Suppose for a contradiction that the cycle has length ≥ 0. Then removing the cycle from path P, we get a new path P' such that length of $P \geq$ length of P', and that P' has at most $n-1$ edges. Therefore, $OPT(n,t) \geq$ length of $P' \geq OPT(n-1,t)$, which is a contradiction the first line of the proof. □

3.8 Floyd-Warshall's All-Pairs Shortest Paths Algorithm

Problem

- Input:
  - Directed graph $G = (V, E)$ with no negative-length cycle
  - Edge lengths w_{uv} on each edge (u, v), where $w_{uv} = \infty$ if $(u, v) \notin E$
- Output: length of a shortest path from each node s to each node t
- Edge lengths can be negative, so we cannot use Dijkstra's algorithm
- If we run Bellman-Ford's single-source shortest path algorithm for every source $s \in V$, we get $\mathcal{O}(mn^2)$ time, which is $\mathcal{O}(n^4)$ if G is dense

Idea

- Index V so that $V = \{1, 2, \ldots, n\}, 1 \leq i, j \leq n$
- P is an $i \overset{k}{\rightsquigarrow} j$ path if every *intermediate* node in P is $\leq k$, i.e. P can only use nodes in $\{1, 2, \ldots, k\}$
- Subproblem: $OPT(i, j, k)$ is the length of the shortest (simple) $i \overset{k}{\rightsquigarrow} j$ path

- Consider a *shortest* $i \overset{k}{\rightsquigarrow} j$ path P, there are two possible cases:
  1. k is *not* intermediate node of P, then P is $i \overset{k-1}{\rightsquigarrow} j$ path, therefore $OPT(i,j,k) = OPT(i,j,k-1)$
  2. k is an intermediate node of P, therefore $OPT(i,j,k) = OPT(i,k,k-1) + OPT(k,j,k-1)$
- Bellman equation:

$$OPT(i,j,k) = \begin{cases} 0, & \text{if } k = 0 \wedge i = j \\ w_{ij}, & \text{if } k = 0 \wedge i \neq j \\ \min\{OPT(i,j,k-1), OPT(i,k,k-1) + OPT(k,j,k-1)\}, & \text{if } k \geq 1 \end{cases}$$

- Goal: compute $OPT(i,j,n)$ for every i and j

Floyd-Warshall Algorithm

```
for i = 1 to n do # O(n^2)
    for j = 1 to n do
        if i = j then OPT(i, j, 0) <- 0
        else OPT(i, j, 0) <- w[i, j]
for k = 1 to n do # O(n^3)
    for i = 1 to n do
        for j = 1 to n do
            OPT(i, j, k) <- min{OPT(i, j, k - 1), OPT(i, k, k - 1) + OPT(k, j, k - 1)}
```

- Time complexity: $\mathcal{O}(n^3)$
- Space complexity: $\mathcal{O}(n^2)$ since we don't need any early values of k

Transitive Closure Graph

- The **transitive closure** of a graph $G = (V, E)$ is the graph

$$G^* = (V, E^*), E^* = \{(u,v) | G \text{ has a } u \rightsquigarrow v \text{ path}\}$$

- To compute the transitive closure of $G = (V, E)$:
  - Can use Floyd-Warshall algorithm with edge weights 1
    - $OPT(u,v,n) \neq \infty \iff G$ has a $u \rightsquigarrow v$ path $\iff G^*$ has edge (u,v)
  - Better way: modify Floyd-Warshall algorithm as follows:
    - Subproblems: $OPT(i,j,k) = \begin{cases} 1, & \text{if } G \text{ has a } i \rightsquigarrow j \text{ path} \\ 0, & \text{elsewise} \end{cases}$

$$OPT(i,j,k) = \begin{cases} OPT(i,j,k-1) \vee (OPT(i,k,k-1) \wedge OPT(k,j,k-1)) & \text{if } k > 0 \\ 1 \quad \text{if } k = 0 \wedge (i = j \vee (i,j) \in E) \\ 0 \quad \text{elsewise} \end{cases}$$

Detecting Negative Length cycles

- Can use Floyd-Warshall algorithm to detech negative cycles

Lemma 3.5. *G has a negative cycle if and only if $\exists u \in V, OPT(u,u,n) < 0$*

Proof. Exercise to the reader (this is how it is written on the slides, I don't bother proving it either) □

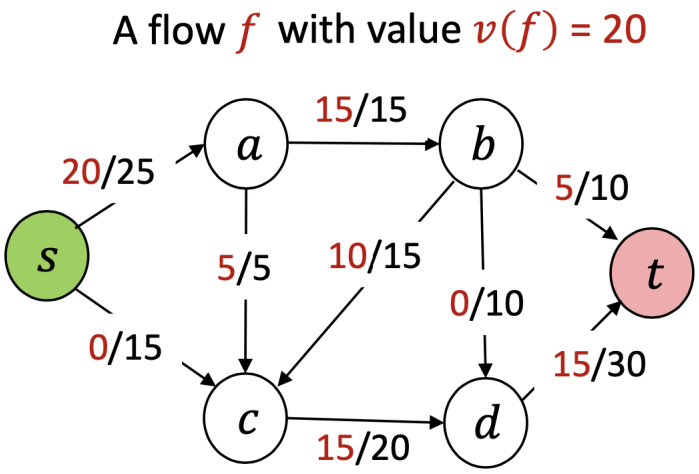
- This lemma can be used to detect negative cycles

4 Network Flow

4.1 Introduction

Network Flow

- Flow Network $\mathcal{F} = (G, s, t, c)$
  - $G = (V, E)$: directed graph
  - $s \in V$: source, no incoming edge
  - $t \in V$: target, no outgoing edge
  - $c : E \to \mathbb{R}^{\geq 0}$: capacity function where $c(u, v)$ is the capacity of the edge $(u, v) \in E$
- Flow $f : E \to R$ respects
  1. *Capacity constraints*: $\forall e \in E, 0 \leq f(e) \leq c(e)$
  2. *Flow conservation*: $\forall v \in V - \{s, t\}, f^{in}(v) = f^{out}(v)$
- Notation
  - $f^{out}(v) = \sum_{e \text{ leaving } v} f(e)$
  - $f^{in}(v) = \sum_{e \text{ into } v} f(e)$
  - *Value* of flow f is $v(f) = f^{out}(s)$



4.2 Ford-Fulkerson Algorithm

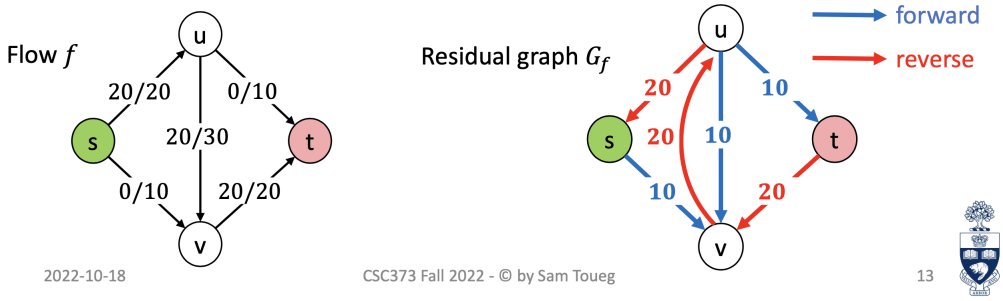
Max Flow Problem

- Input: flow network $\mathcal{F} = (G, s, t, c)$
- Output: flow f of max value, i.e. $\forall$flow $f', v(f) \geq v(f')$

Residual Graph

- Denoted G_f for flow f
- G_f has the same vertices as G

- For each edge $e = (u, v)$ in G, G_f has *at most two edges*:
  1. *Forward edge* $e = (u, v)$ with **residual capacity** $c(e) - f(e) > 0$
     - We can send this much additional flow on e
  2. *Reverse edge* $e^{rev} = (v, u)$ with **residual capacity** $f(e) > 0$
     - The maximum "reverse" flow that we can send on edge e
     - The maximum amount we can reduce the flow on e
- We add a (forward or reverse) edge to G_f only if its **residual capacity** > 0



Augmenting Paths

- Let P be an $s \rightsquigarrow t$ path in the residual graph G_f
- bottleneck(P, f): smallest *residual* capacity among all edges in P
- We can *augment* flow f by sending bottleneck(P, f) units of flow along P
  - For each forward edge $e \in P$, *increase* the flow on e by the bottleneck
  - For each reverse edge $e^{rev} \in P$, *decrease* the flow on e^{rev} by the bottleneck

Lemma 4.1. *The augmented flow f' is better than the original f.*

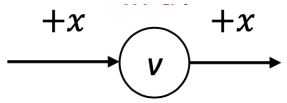
Proof. Note that the first edge (s, u) of P in G_f must be a *forward* edge (because the original graph G has no edge into s). Then $f'(s, u) = f(s, u) + x$ with $x > 0$, therefore $v(f') \geq v(f) + x$ □

Lemma 4.2. *The agumented flow f' is valid.*

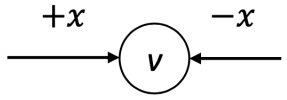
Proof. 1. Capacity constraints. If we increased the flow on e, we did so by at most the *residual capacity* of *forward* edge e in G_f, which is $c(e) - f(e)$. The new flow $f'(e)$ is at most $f(e) + c(e) - f(e) = c(e)$. If we decreased the flow on e, we did so by at most the *residual capacity* of *reverse* edge e^{rev} in G_f, which is $f(e)$. The new flow $f'(e)$ is at least $f(e) - f(e) = 0$.

2. Flow conservation. Fix $v \in P$. The edge going into v may be a forward/reverse edge, and the edge leaving v may also be a forward/reverse edge. Let x be the change of flow on those edges (i.e. bottleneck). We have 4 cases:

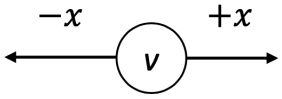
1. Forward/Forward: both $f^{in}(v)$ and $f^{out}(v)$ increase by x
2. Forward/Reversed: we are sending x to v, then v sends x back, therefore $f^{in}(v)$ and $f^{out}(v)$ stay the same
3. Reversed/Forward: we are sending x to v by "pushing back", then send x from v, therefore $f^{in}(v)$ and $f^{out}(v)$ stay the same
4. Reversed/Reversed: both $f^{in}(v)$ and $f^{out}(v)$ decrease by x



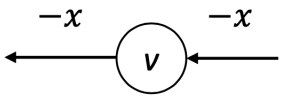
both $f^{in}(v)$ and $f^{out}(v)$ increased by x



no change in $f^{in}(v)$ or $f^{out}(v)$



no change in $f^{in}(v)$ or $f^{out}(v)$



both $f^{in}(v)$ and $f^{out}(v)$ decreased by x

22 - © by Sam Toueg 25

□

Ford-Fulkerson Algorithm

```
1       Ford-Fulkerson(F): # F = (G, s, t, c)
2           for each edge (u, v) of G do
3               f(u, v) = 0 # initial (empty) flow
4           construct G_f # O(m + n)
5           while G_f has an s -> t path do
6               P <- any simple s -> t path in G_f # O(m + n) by BFS/DFS
7               f <- augment(f, P) # O(n), i.e. the length of P
8               update G_f # O(n)
9           return f
```

- Running time of 1 iteration of the while loop: $\mathcal{O}(m+n)$
- Assume that edge capacities are integers
- Observations:
  1. At every step, flows and residual capacities remain integers
  2. Every iteration increases the value of the flow by at least 1
  3. maxflow $\leq C = \sum_{e \text{ leaving } s} c(e)$, therefore after *at most* C iterations, FF algorithm terminates
- Total running time: $\mathcal{O}(C(m+n)) = \mathcal{O}(mC)$
  - Pseudo-polynomial

Improving FF to Polynomial Time

- Pick the augmenting path with the *maximum bottleneck capacity*
  - Using modified Dijkstra's algorithm, giving $\mathcal{O}(m \log n)$ time per iteration
  - Can prove for at most $\mathcal{O}(m \log C)$ iterations
  - Total running time: $\mathcal{O}(m^2 \log n \log C)$

- Pick the *shortest* augmenting path, i.e. one with the *fewest edges* (Edmonds-Karp algorithm)
  - Using BFS, $\mathcal{O}(m)$ time for each iteration
  - Can prove for at most $\mathcal{O}(mn)$ iterations
  - Total running time $\mathcal{O}(m^2 n)$
- Dinic's algorithm (1970): $\mathcal{O}(mn^2)$
- Sleator-Tarjan algorithm (1983): $\mathcal{O}(mn \log n)$
- Newest algorithm (2013): $\mathcal{O}(mn)$

Cuts and Cut Capacities

- Given a network flow $\mathcal{F} = (G, s, t, c)$
- (A, B) is an $s - t$ **cut** of $\mathcal{F}$ if (A, B) is a *partition of the vertices* with $s \in A$ and $t \in B$
- **Capacity** of cut (A, B) is the sum of capacities of all edges *leaving* A
- $\text{cap}(A, B) = \sum_{u \in A, v \in B} c(u, v)$

Lemma 4.3. *For any flow f and any $s - t$ cut (A, B),*

$$v(f) = f^{out}(A) - f^{in}(A)$$

where

$$f^{out}(A) = \textit{ flow from } A \textit{ to } B = \sum_{e: A \to B} f(e)$$
$$f^{in}(A) = \textit{ flow from } B \textit{ to } A = \sum_{e: B \to A} f(e)$$

Proof.

$$\begin{aligned}
v(f) &= \sum_{e \text{ leaving } s} f(e) \\
&= \sum_{u \in A} \left(\sum_{e \text{ leaving } u} f(e) - \sum_{e \text{ entering } u} f(e) \right) \\
&= \sum_{u \in A} \sum_{e \text{ leaving } u} f(e) - \sum_{u \in A} \sum_{e \text{ entering } u} f(e) \\
&= \left(\sum_{e: A \to B} f(e) + \sum_{e: A \to A} f(e) \right) - \left(\sum_{e: B \to A} f(e) + \sum_{e: A \to A} f(e) \right) \\
&= \sum_{e: A \to B} f(e) - \sum_{e: B \to A} f(e)
\end{aligned}$$

□

Lemma 4.4. *For any flow f and any $s - t$ cut (A, B), $v(f) \leq cap(A, B)$*

Proof.

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \qquad \text{By Lemma 4.3} \\ &\leq f^{out}(A) \\ &= \sum_{e:A\to B} f(e) \\ &\leq \sum_{e:A\to B} c(e) \qquad \text{By capacity constraints} \\ &= \text{cap}(A,B) \end{aligned}$$

□

Corollary 4.4.1. *For any flow f and any $s-t$ cut (A,B), if $v(f) = cap(A,B)$, then*

1. *f is a* max flow*, and*
2. *(A,B) is a* min cut*, i.e. a cut with* minimum capacity *among all $s-t$ cuts*

Proof. Let f^* be a max flow, and (A^*, B^*) be a min cut. Let f be any flow, then $v(f) \leq v(f^*)$. Let (A,B) be any $s-t$ cut, then $\text{cap}(A^*,B^*) \leq \text{cap}(A,B)$. By Lemma 4.4, $v(f^*) \leq \text{cap}(A^*,B^*)$. Therefore

$$v(f) \leq v(f^*) \leq \text{cap}(A^*,B^*) \leq \text{cap}(A,B)$$

Knowing that $v(f) = \text{cap}(A,B)$, we have

$$v(f) = v(f^*) = \text{cap}(A^*,B^*) = \text{cap}(A,B)$$

□

Theorem 4.5 (Correctness of Ford-Fulkerson Algorithm)**.** *Ford-Fulkerson algorithm outputs a maximum flow.*

Proof. Let f be the flow returned by FF, and G_f be the corresponding residual graph. Let A be the set of nodes that are reachable from s in G_f, and $B = V - A$. (A,B) is an $s-t$ cut since $s \in A$ and $t \in B$. Note that

1. $\forall (u,v) \in E$ such that $u \in A, v \in B$, we have that $f(u,v) = c(u,v)$
2. $\forall (v,u) \in E$ such that $v \in B, u \in A$, we have that $f(v,u) = 0$

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \qquad \text{By Lemma 4.3} \\ &= \sum_{u\in A, v\in B} f(u,v) - \sum_{v\in B, u\in A} f(v,u) \\ &= \sum_{u\in A, v\in B} c(u,v) - 0 \\ &= \text{cap}(A,B) \end{aligned}$$

By Corollary 4.4.1, f is a max flow. □

- Additionally, (A,B) is a min cut

Theorem 4.6 (Max-Flow Min-Cut)**.** *Value of the max flow equals capacity of the min cut*

Proof. From FF algorithm and correctness proof □

Theorem 4.7 (Integrality)**.** *If all the edge capacities are* integers*, there is an* integral *max flow f such that* $\forall e \in E, f(e) \in \mathbb{Z}$

Proof. From FF algorithm and correctness proof □

- However not all max flows have integer $f(e)$s
- This theorem guarantees the *existence* of one

Min Cut Problem

- Input: flow network $\mathcal{F} = (G, s, t, c)$
- Output: $s-t$ cut (A, B) of min capacity, i.e. $\forall s-t$ cut $(S, T), \text{cap}(A, B) \leq \text{cap}(S, T)$

```
Min-Cut(F):
    f <- Ford-Fulkerson(F)
    compute residual graph G_f
    A <- {all nodes that are reachable from s in G_f}
    B <- V - A
    return (A, B)
```

- Running time: same as Ford-Fulkerson

4.3 Bipartite Matching

Matching

- Let $G = (V, E)$ be an undirected graph
- A **matching** M in G is a subset of edges of G such that no two edges share the same node
- *Maximal* matching: not a subset of another matching
- *Maximum* matching: a matching of maximum size

Bipartite Graph

- Undirected graph $G = (V, E)$ such that
  1. V is partitioned into two sets X and Y (which may have different size)
  2. Every edge in E connects a node in X to a node in Y
- $G = [(X, Y), E]$
- An undirected graph G is bipartite iff G has no odd cycle
- We can determine if G is bipartite in $\mathcal{O}(m+n)$ time using BFS/DFS

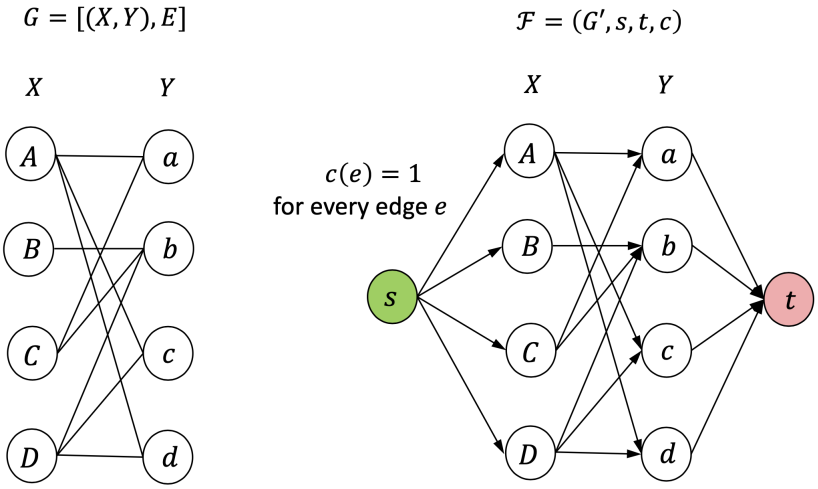
Bipartite Matching Problem

- Input: bipartite graph $G = (V, E)$
- Output: *maximum* matching of G
- Can solve this problem by reducing it to the Max Flow problem

Turning Bipartite Graph into Flow Network

- $G = [(X, Y), E]$

- Direct all edges from X to Y
- Add a new node s and connect it to all nodes in X
- Add a new node t and connect it to all nodes in Y
- Let $c(e) = 1$ for every edge e



Bipartite Matching Algorithm

```
Bipartite-Matching(G = [(X, Y), E]):
    construct flow network F from G # O(m + n)
    f <- MaxFlow(F) # O(mC) = O(mn) since C = O(n)
    M <- {(x, y) | x in X, y in Y, f(x, y) = 1} # O(m)
    return M
```

- By integrality theorem, the flow f found by MaxFlow is an *integer* on each edge
- Since the capacity of each edge is 1, the integer flow f is *either 0 or 1* on each edge

Lemma 4.8. *There exists a matching of size k in G iff there exists an integral flow of value k in $\mathcal{F}$.*

Proof. " $\Longrightarrow$ " Given a matching M in G of size k, we can construct an integral flow f in $\mathcal{F}$ of value k as follows:

$$f_M(e) = \begin{cases} 1, & \text{if } e \text{ is on the path } s \to x \to y \to t \text{ and } (x, y) \in M \\ 0, & \text{elsewise} \end{cases}$$

- f_M satisfies *capacity constraint* because $\forall e \in E, f_M(e) \leq 1 = c(e)$
- f_M satisfies *flow conservation constraint* by exhaustion of possible violations:
  - Flow from source to x but stops at x: impossible by construction of our matching
  - No flow from source to x but flow going out from x: impossible by construction of our matching
  - Flow from source to x, and from x to two other nodes: impossible since it would not be a matching
  - Flow from y to target but no flow into y: impossible by construction of our matching
  - Flow going into y but no flow from y to t: impossible by construction of our matching
  - Flow going from two nodes into y: impossible since it would not be a matching

" $\Longleftarrow$ " Given an integral flow f in $\mathcal{F}$ of value k, we can construct a matching M_f in G of size k as follows:

- Since $f(e)$ is an integer for each e and $c(e) = 1$, $f(e) \in \{0, 1\}$
- Let $M_f = \{(x, y) | x \in X, y \in Y, f(x, y) = 1\}$
- Claim: M_f is a matching in G
- Suppose for a contradiction that it is not, then the possible violations are
  - x matches to two other nodes, which is impossible since it would violate flow conservation constraint
  - Two nodes matches to the same y, which is impossible since it would violate flow conservation constraint
- Claim: $|M_f| = v(f) = k$
- Consider the cut (S, T)
  - $S = \{s\} \cup X$, $T = \{t\} \cup Y$
  - By Lemma 4.3:

$$\begin{aligned} v(f) &= f^{out}(S) - f^{in}(S) \\ &= f^{out}(S) - 0 \\ &= |M_f| \end{aligned}$$

□

- G and $\mathcal{F}$ are constructed together beforehand, this lemma shows that matchings in G and flows in $\mathcal{F}$ are equivalent

Corollary 4.8.1. *Maximum matching in G equals the max flow in $\mathcal{F}$.*

Proof. Follows from Lemma 4.8. □

Graph Matching Problems

- Bipartite graph matching
  - Ford-Fulkerson: $\mathcal{O}(mn)$ (1955)
  - Hopcroft-Karp: $\mathcal{O}(m\sqrt{n})$ (1973)
  - Mucha-Sankowsi: $\mathcal{O}(n^{2.378})$ (2004)
- Nonbipartite graph matching
  - Edmonds: $\mathcal{O}(mn^2)$ (1965)
  - Micali-Vazirani: $\mathcal{O}(m\sqrt{n})$ (1980)

4.4 Minimum Vertex Cover

Vertex Cover

- $G = (V, E)$: undirected graph
- *Vertex cover (VC)*: a subset V' of the nodes of G that "touches" every edge of G, i.e. every edge has at least 1 endpoint in V'
- *Minimum vertex cover*: a VC with minimum size
- For a general graph, finding a min VC is NP-Complete

- For a bipartite graph, a min VC can be found in polynomial time

Lemma 4.9. *For any matching M in G, and any VC C of G, $|M| \leq |C|$.*

Proof. Follows from the facts:

- Every edge in M must be covered by *at least 1* node in C
- Each node in V covers *at most 1* edge of M (because otherwise M would not be a matching)

□

Lemma 4.10. *For any matching M in G, and any VC C of G, if $|M| = |C|$, then*

- *M is a* maximum matching
- *C is a* minimum VC

Proof. Let M be any matching of G and C be any VC of G. Let M^* be a *max matching* of G and C^* be a *min VC* of G. We know that $|M| \leq |M^*|$ and that $|C^*| \leq |C|$. By Lemma 4.9, we have $|M^*| \leq |C^*|$. Therefore,

$$|M| \leq |M^*| \leq |C^*| \leq |C|$$

Knowing that $|M| = |C|$, we have that

$$|M| = |M^*| = |C^*| = |C|$$

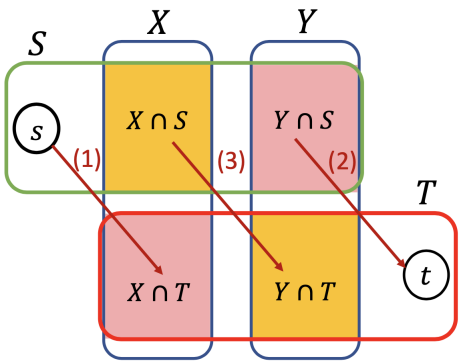
And so M is a max matching and C is a min VC. □

Lemma 4.11. *If $G = [(X, Y), E]$ is a bipartite graph, then*

$$|\textit{max matching of } G| = |\textit{min VC of } G|$$

Proof. Given a bipartite graph G, construct a flow network $\mathcal{F}$ as before. Let f be an integral max flow in $\mathcal{F}$ (which exists by the Integrality Theorem). Let S be nodes reachable from s in G_f, and $T = V - S$. Since (S, T) is a min-cut, we have that $\text{cap}(S, T) = v(f)$. Consider the types of edges in $\mathcal{F}$ from S to T:

1. $s \to X \cap T$
2. $Y \cap S \to t$
3. $X \cap S \to Y \cap T$



The proof finishes with the claims in Lemma 4.13 and Lemma 4.14. □

Lemma 4.12. *There are no edges of type (3) in $\mathcal{F}$.*

Proof. Suppose for a contradiction that there exists (x, y) such that $x \in X \cap S \wedge y \in Y \cap T$. Then there are two cases:

1. $f(x, y) = 0$. Then G_f has edge $x \to y$, therefore y is reachable from s in G_f, consequently $y \in S$, which is a contradiction.
2. $f(x, y) = 1$. Since $x \in S$, x is reachable from s in G_f. Let P be a $s \rightsquigarrow x$ simple path in G_f.
   (a) Subcase $P = (s, x)$, so the edge (s, x) is in G_f. This means $f(s, x) = 0$ because if $f(s, x) = 1$, then edge (s, x) would not be in G_f. Knowing that $f(x, y) = 1$, we have a violation of flow conservation at x.
   (b) Subcase $P = (s, x_1, y_1, x_2, y_2, x_3, \cdots x_k, y_k, x)$. Then all the edges from x_i to y_i are forward, and that all edges from y_i to x_i are reversed. Note that G_f has a reverse edge $y_k \to x$, meaning that $f(x, y_k) = 1$. However, since $f(x, y) = 1$, x has flow towards both y_k and y, which is a violation of flow conservation at x.

□

Lemma 4.13. *$C = (X \cap T) \cup (Y \cap S)$, i.e. vertices in the pink region, is a VC of G.*

Proof. G has 4 types of edges:

1. $X \cap S \to Y \cap S$ (yellow to pink): covered by nodes in $Y \cap S$
2. $X \cap T \to Y \cap T$ (pink to yellow): covered by nodes in $X \cap T$
3. $X \cap T \to Y \cap S$ (pink to pink): covered by nodes in $(X \cap T) \cup (Y \cap S)$
4. $X \cap S \to Y \cap T$ (yellow to yellow): no such edges exist by Lemma 4.12

□

Lemma 4.14. *$C = (X \cap T) \cup (Y \cap S)$, i.e. vertices in the pink regions, is a* min *VC of G.*

Proof.

$$\begin{aligned} |C| &= |X \cap T| + |Y \cap S| && \text{Edges of Type 1 and 2} \\ &= \text{cap}(S, T) && \text{Type 3 edges don't exist by Lemma 4.12} \\ &= v(f) && \text{Because } (S, T) \text{ is a min-cut and } f \text{ is a max-flow} \\ &= |\text{max matching in } G| && \text{By Corollary 4.8.1} \end{aligned}$$

Therefore $|C| = |M|$ for some matching M of G. By Lemma 4.10, C is a min VC. □

Vertex Cover Algorithm for Bipartite Graphs

```
Bipartite-VC(G): # G = [V = (X, Y), E]
    construct flow network F from G # O(m + n)
    f <- MaxFlow(F) # O(mK) = O(mn) since K = O(n)
    S <- nodes reachable from s in G_f # O(n + m)
    T <- V - S # O(n + m)
    VC <- (X intersection T) union (Y intersection S) # O(n)
    return VC
```

- Running time: $\mathcal{O}(mn)$
- Finding a min VC for general graphs is NP-Complete

4.5 Hall's Theorem on Perfect Bipartite Matching

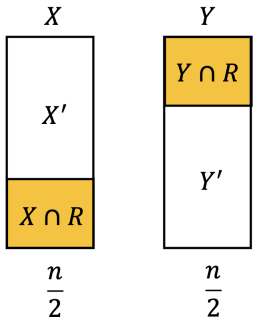
Perfect matching in a bipartite graph $G = ([X, Y], E)$

- **Perfect matching**: every node is matched to another node
- Impossible if there is a subset $X' \subseteq X$ such that $|X'| > |N(X')|$, where $N(X')$ denotes the neighbours of X'
  - More nodes on the left side than the right side

Theorem 4.15 (Hall's)**.** *A bipartite graph $G = ([X, Y], E)$ has* no *perfect matching iff $\exists X' \subseteq X$ such that $|X'| > |N(X')|$*

Proof. " $\Longleftarrow$ " A matching for X' implies that X' has at least $|X'|$ neighbours.

" $\Longrightarrow$ " Assume no perfect matching for G exists. Therefore |max matching in $G| < n/2$. Let R be the min VC of G.



From Lemma 4.11, $|R| =$ |max matching in $G| < n/2$. We also know that $|X'| > |Y \cap R|$ because:

- $|X \cap R| + |X'| = n/2$
- $|X \cap R| + |Y \cap R| = |R| < n/2$

Furthermore $|N(X')| \leq |Y \cap R|$ because

- There is no $X' \leftrightarrow Y'$ edge (otherwise R would not be a vertex cover)
- So $N(X') \subseteq Y \cap R$ and $|N(X')| \leq |Y \cap R|$

Therefore

$$|X'| > |Y \cap R| \geq |N(X')|$$

□

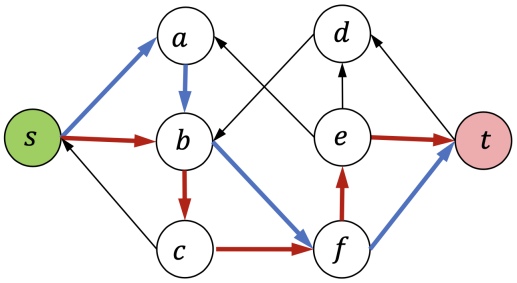
Equivalence of Hall's Theorem

- Bipartite graph $G = ([X, Y], E)$ has a perfect matching iff $\forall X' \subseteq X, |X'| \leq |N(X')|$

4.6 Edge-Disjoint Paths

Definition

- Two $s \rightsquigarrow t$ paths are **edge-disjoint** if they don't share an edge



Problem

- Input: directed graph $G = (V, E)$ where $s, t \in V$
- Output: find a max-size set of edge-disjoint $s \rightsquigarrow t$ paths

Approach

1. From G, construct a flow network $\mathcal{F}$ as follows:
   - Delete edges into s and edges out of t
   - All remaining edges have capacity 1
2. Find integral max flow in $\mathcal{F}$, which equals to the max number of edge-disjoint $s \rightsquigarrow t$ paths
3. Decompose this flow into edge-disjoint paths

Theorem 4.16. *There are k edge-disjoint $s \rightsquigarrow t$ paths in G iff there is an integral flow of value k in $\mathcal{F}$*

Proof. " $\Longrightarrow$ " Let $P_1, \ldots, P_k$ be k edge-disjoint $s \rightsquigarrow t$ paths. Define the following flow f:

- $f(e) = 1$ if $e \in P_1 \cup \cdots \cup P_k$ and 0 otherwise
- f satisfies capacity constraints because $0 \leq f(e) \leq 1$
- f satisfies flow conservation because otherwise there would be more/less paths into a node than out of the node
- s has k outgoing edges (one for each P_i), each with flow 1
- So f is an integral flow with value k

" $\Longleftarrow$ " Let f be an integral flow of value k in the flow network $\mathcal{F}$. Since the capacity of each edge is 1, s must have k outgoing edges with a flow of 1 each. Pick one such edge (s, u). By flow conservation, u must have at least 1 outgoing edge with a flow of 1. Pick one such edge (u, v). We continue this process until we find t, and repeat this for each of the other $k - 1$ edges from s with flow 1. This gives k edge-disjoint $s \rightsquigarrow t$ paths in G. □

Edge-Disjoint Paths Algorithm

```
Edge-Disjoint-Paths(G, s, t):
    construct flow network F from G # O(n + m)
    f <- MaxFlow(F) # FF algo: O(mC) = O(mn) since C = O(n)
    P <- Path-Decomposition(f) # O(n + m)
    return P
```

```
Path-Decomposition(f): # f is an integral flow in F, every edge has capacity 1
    # O(m + n)
    P <- empty set
    while v(f) > 0 do
        u <- s; path <- s
        while u != t do
            v <- any node v such that f(u, v) = 1
            path <- path concatenate v
            f(u, v) <- 0
            u <- v
        P <- P union path
    return P
```

- Running time: $\mathcal{O}(mn)$

5 Linear Programming

5.1 The Linear Programming Problem

Linear Programming (LP) Optimization Problem

- Want to minimize/maximize some *objective function*, which is a *linear* function of $x_1, \ldots, x_n$
- Subject to *constraints* (i.e. equalities or inequalities), which are *linear* combinations of $x_1, \ldots, x_n$
- Can be solved in polynomial time
- Need to be able to write certain problems as LP problems

Example: Diet Problem

- Problem
  - n types of food: $F_1, \ldots, F_n$
  - m types of nutrients $N_1, \ldots, N_m$
  - Each unit of food F_j for $j \in \{1, \ldots, n\}$:
    - Costs c_j
    - Provides a_{ij} units of nutrient N_i for each $i \in \{1, \ldots, m\}$
  - Goal: find the *cheapest* diet that provides the required amount of nutrients
    - Diet: amount of food F_j for each $j \in \{1, \ldots, n\}$
- As an LP problem
  - Variables: $x_1, \ldots, x_n$ where x_j is the amount of food F_j in the diet
  - Objective function: minimize
$$c_1 x_1 + \cdots + c_n x_n$$
  - Constraints:
$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad \forall i \in \{1, \ldots, m\}$$
$$x_1, x_2, \ldots, x_n \geq 0$$

Example: Profit Maximization Problem

- Problem
  - n types of products: $P_1, \ldots, P_n$
  - m types of resources: $R_1, \ldots, R_m$
  - Each unit of P_j for $j \in \{1, \ldots, n\}$:
    - Yields profit c_j
    - Requires a_{ij} units of resource R_i for each $i \in \{1, \ldots, m\}$
  - Quantity of resources available: at most b_i units of resource R_i for each $i \in \{1, \ldots, m\}$
  - Goal: find mix of products to make that *maximizes* total profit
    - Mix: amount of product P_j for each $j \in \{1, \ldots, n\}$
- As an LP problem
  - Variables: $x_1, \ldots, x_n$ where x_j is the amount of product P_j in the mix

- Objective function: maximize

$$c_1x_1 + \cdots + c_nx_n$$

- Constraints:

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \qquad \forall i \in \{1, \ldots, m\}$$
$$x_1, x_2, \ldots, x_n \geq 0$$

Example: Transportation Problem

- Problem
  - Factories $F_1, \ldots, F_k$ (that manufacture the same product)
  - Outlets $O_1, \ldots, O_l$
  - For each $i \in \{1, \ldots, k\}$, factory F_i makes s_i units of product
  - For each $j \in \{1, \ldots, l\}$, outlet O_j wants at least d_j units of product
  - c_{ij} is the cost to ship one unit of product from factory F_i to outlet O_j
  - Goal: minimize the shipping cost to supply every outlet with the amount of product that it wants
- As an LP problem
  - Variables: x_{ij} is the amount of product shipped from factory F_i to outlet O_j
  - Objective funciton: minimize

$$\sum_{i=1}^{k}\sum_{j=1}^{l} c_{ij}x_{ij}$$

  - Constraints:

$$\sum_{j=1}^{l} x_{ij} \leq s_i \qquad \forall i \in \{1, \ldots, k\}$$
$$\sum_{i=1}^{k} x_{ij} \leq d_j \qquad \forall j \in \{1, \ldots, l\}$$
$$x_{ij} \geq 0 \qquad \forall i \in \{1, \ldots, k\}, \forall j \in \{1, \ldots, l\}$$

General Form of Linear Programming

- Variables: $x_1, \ldots, x_n \in \mathbb{R}$
- Objective function: minimize/maximize

$$\sum_{j=1}^{n} c_j x_j$$

  for some given c_js $\in \mathbb{R}$
- Constraints:

$$\sum_{j=1}^{n} a_{ij}x_j \begin{cases} \leq \\ \geq \\ = \end{cases} b_i \qquad \forall i \in \{1, \ldots, m\}$$

  for some given a_{ij}s $\in \mathbb{R}$, b_is $\in \mathbb{R}$, and

$$x_1, x_2, \ldots, x_n \geq 0$$

- Input: $c_j, a_{ij}, b_i \in \mathbb{R}$, $\forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, n\}$
- Output: $x_1, \ldots, x_n \in \mathbb{R}$ that minimizes/maximizes the objectivwe function and satisfy all the constraints

General Form: Matrix Formulation

- Variables: $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$
- Objective function: minimize/maximize $c^\top x$, where $c = (c_1, \ldots, c_n)$
- Constraints: $Ax \begin{cases} \leq \\ \geq \\ = \end{cases} b$, where $A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$ and $b = (b_1, \ldots, b_m)$

5.2 Solving the Linear Programming Problem

Formulate Problem to Have One Type of Constraints

- To get $\leq$ from $\geq$ (or vice versa), multiply the constraint by -1
- To get $\leq$ and $\geq$ from $=$, write the equality in terms of two inequalities

Standard Form

- Standard form of LP:
  - Maximize $c^\top x$
  - Subject to $Ax \leq b$ and $x \geq 0$
- To convert the LP problem to standard form:
  1. Change the objective function to *maximization*
  2. Change $\geq$ and $=$ to $\leq$, and obtain all variables on the left and constants on the right
  3. Restrict all variables to ≥ 0 (i.e. if y is unrestricted, then replace y with $y' - y''$ such taht $y', y'' \geq 0$)
- Example

$$\begin{aligned} &\min\ 4x + 3y - 6z \\ &\text{s.t.} \\ &\quad y - 3z \geq 2x + 2 \\ &\quad 3x + 2y + 5z = 10 \\ &\quad x, z \geq 0 \end{aligned}$$

$$\begin{aligned} &\max\ -4x - 3y + 6z \\ &\text{s.t.} \\ &\quad 2x - y + 3z \leq -2 \\ &\quad 3x + 2y + 5z \leq 10 \\ &\quad -3x - 2y - 5z \leq -10 \\ &\quad x, z \geq 0 \end{aligned}$$

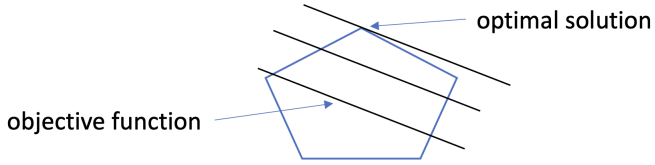
$$\begin{aligned} &\max\ -4x - 3y' + 3y'' + 6z \\ &\text{s.t.} \\ &\quad 2x - y' + y'' + 3z \leq -2 \\ &\quad 3x + 2y' - 2y'' + 5z \leq 10 \\ &\quad -3x - 2y' + 2y'' - 5z \leq -10 \\ &\quad x, z, y', y'' \geq 0 \end{aligned}$$

- If we have non-standard LP constraints, we can break it down into multiple LP problems
  - E.g. a constraint of $\neg(A > 0 \wedge B > 0)$ can be turned into $(A \leq 0) \vee (B \leq 0)$, where we could do 2 LPs and take the best result
  - E.g. a constraint of $C \in \{1, \ldots, 100\}$ can be achieved by doing 100 LPs and take the best result

Optimal Solution

- A **feasible region** of solutions is a *convex* polygon
- If an optimal solution to a LP exists, then *at least one* optimal solution is a *vertex* of the feasible region

- Knowing this, we can only search from the vertices of the feasible region



- Optimal solution may not exist because
  - LP is *infeasible*, i.e. feasible region is empty
    - LP is overconstrained
  - LP is *unbounded*
    - LP is underconstrained

Linear Programming Algorithm/Solvers

- Input: an LP instance (may accept only $\geq$, $\leq$, or $=$ type of constraints)
- Output: one of
  1. Infeasible
  2. Unbounded
  3. Values for $x_1, \ldots, x_n$ that maximize/minimize the objective function and satisfy all constraints

Simplex Algorithm

- Start from any vertex in the feasible region
- Check if there is any vertex neighbour that has a *strictly better value*
  - If yes, go to this neighbour, abd then repeat from here
  - If no, return this vertex as the optimal solution
- Can prove that this local optimum is a global optimum (since the feasible region is *convex*)
- This algorithm works well in practice
- Exponential time in the worst case (i.e. the feasible region can have an exponential number of vertices)

Solving a Linear Programming Problem in Polynomial Time

- Ellipsoid algorithm (Khatchian 1979)
  - Polynomial time in the worst case
  - Slow in practice
- Interior point method (Karmarkar 1984)
  - Polynomial time in the worst case
  - Good in practice

5.3 Variations of Linear Programming

Variations

- LP: solutions are real numbers, i.e. $x_1, \ldots, x_n \in \mathbb{R}$
- Integer LP (ILP): solutions are required to be integers, i.e. $x_1, \ldots, x_n \in \mathbb{Z}$
- 0-1 LP: solutions are required to be binary, i.e. $x_1, \ldots, x_n \in \{0, 1\}$
- ILP and 0-1 LP are *NP-Complete*

Expressing Min Vertex Cover as 0-1 LP

- Problem:
  - Input: an undirected graph $G = (V, E)$
  - Output: minimum size $V' \subseteq V$ such that $\forall (u, v) \in E$, at least o of $u, v \in V'$
- 0-1 LP
  - (Binary) Variables: $x_u, \forall u \in V$ (i.e. $x_u = 1$ iff node u is in the VC)
  - Objective function: minimize
    $$\sum_{u \in V} x_u$$
    (i.e. want a "smallest" VC)
  - Constraints:
    $$\begin{aligned} x_u + x_v &\geq 1 \qquad \forall (u, v) \in E \\ x_u &\in \{0, 1\} \qquad \forall u \in V \end{aligned}$$
    * ILP method: $0 \leq x_u \leq 1, \forall u \in V$, and $x_u \in \mathbb{Z}, \forall u \in V$

    (i.e. every edge must be "covered" by a VC node)

6 NP-Complete Problems

6.1 Clique

Input

- Undirected graph $G = (V, E)$
- A **clique** is a subset of nodes of G that are completely connected by edges of G (i.e. an edge between *every* pair of nodes in the subset)
- Question: does G have a clique of size k?

Brute Force Algorithm

- Generate every subset S of k nodes of V
- Check whether every pair of nodes of S is connected by an edge in E
- Exponential running time

6.2 SAT and 3SAT

CNF Formulas

- Boolean formula in *conjunctive normal form (CNF)*
- E.g.
$$\varphi = (\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_3 \vee \overline{x}_4)$$
- A CNF formula consists of
  - Boolean *variables* $x_1, \ldots, x_n$
  - Their *negations* $\overline{x}_1, \ldots, \overline{x}_n$
  - *Literal* l_i: a variable x_i or its negation $\overline{x}_i$
  - Clause $C = l_1 \vee \cdots \vee l_r$ is a disfunction of literals
  - *CNF formula* $\varphi = C_1 \wedge \cdots \wedge C_m$ is a conjunction of clauses
    - kCNF: each clause havs k literals
    - 3CNF: each clause havs 3 literals

Satisfiability Problems

- A CNF formula φ is **satisfiable** if there is a *truth assignment* to the variables x_i under which φ evaluates to True

SAT Problem

- Input: a CNF formula φ
- Question: is φ satisfiable?

3SAT Problem

- Input: a 3CNF formula φ
- Question: is φ satisfiable?

Brute Force Algorithm

- Try all possible T/F assignments to the n variables x_i of φ
- Takes 2^n time

6.3 Reducing SAT to Clique

SAT is polynomially-reducible to Clique, denoted SAT $\leq_p$ Clique

- We can transform any given CNF formula ϕ (in polynomial time) into a graph and integer (G, k) such that φ is satisfiable *if and only if* G has a clique of size k
- If we can solve Clique in polynomial time, then we can also solve SAT in polynomial time
- If we *cannot* solve SAT in polynomial time, then we *cannot* solve Clique in polynomial time
- The RHS of $\leq_p$ is "at least as hard as" the LHS

Lemma 6.1. *SAT* $\leq_p$ *Clique*

Proof. Given a formula ϕ, we construct the graph $G = (V, E)$ as follows:

- One node for every literal of every clause of ϕ
- No edges between nodes in the same clause
- An edge between every pair of nodes representing literals that are *compatible*

We will show that φ is satisfiable iff G has a clique of size k

Suppose that φ is satisfiable. Then for each clause, there is a literal that evaluates to True. Since those literals are compatible, there is an edge between every pair of such nodes in the graph, which is a clique of size k

Suppose that G has a clique of size k. Then for each pair of clauses, there are compatible literals. This means that φ is satisfiable. □

6.4 Decision Problems

Decision Problems

- Problems with a *yes-or-no answer* to a question
- E.g. Clique, SAT, 3SAT

Decision vs. Optimization Problems

- Many decision problems have an optimization version
- E.g. for Clique, an optimization version would be to maximize k such that G have a clique of size k, another optimization version would be to find the maximum-size clique of G
- Typically, an optimization version is at least as hard to solve as a decision version

6.5 Polynomial-Time Reductions

Polynomial Time Reductions

- Let A and B be two decision problems
- (Karp) $A \leq_p B$ if we can *transform* any instance x of A, in polynomial time, into an instance y of B such that x is a YES instance of A iff y is a YES instance of B
- If $A \leq_p B$ and B can be solved in polynomial time, then A can also be solved in polynomial time
  - We first transform x into an instance y of B with the same answer, then solve the instance y of B and return the YES or NO answer

  - Takes polynomial time

- Suppose that A is already known to be "hard" since A cannot be solved in polynomial time
- To prove that B is also "hard", we can show that $A \leq_p B$
- Alternative definition: (Cook) $A \leq_p B$ if, given an *oracle* (subroutine) for solving B, we can solve A in polynomial time
  - To solve A using the oracle for B in polynomial time, we can call the oracle only a polynomial number oaf times, and do some additional polynomial time computation
  - This definition is a generalization of the other one, where the oracle for B is used *only once*

6.6 Efficient Verifiers

Efficient Verifier For SAT

- A decision problem P has an **efficient verifier** if every YES instance x or P has a short *justification* that can be used to efficiently verify that x is a YES instance of P
- For SAT, if someone claims that φ is satisfiable, then they can give a *specific truth assignment* to the variables of φ that allegedly makes φ True
- A verifier algorithm can then check in polynomial time whether φ is indeed True under the given truth assignment
- This means that SAT has an efficient verifier

Efficient Verifier For Clique

- If someone claims that G has a clique of size k, they can give a *set of nodes* C (which is allegedly a clique of size k in G) as a justification for their claim
- There is a *verifier algorithm* that can check, in polynomial time, whether the given C is indeed a k-clique of G
- The verifier checks that C is a set of k nodes of G, and G has an edge between every pair of nodes in C
- This means that Clique has an efficient verifier

Partition Problem

- Input: a list L of integers
- Question: can we partition L into two lists that sum the same?
- Partition has an efficient verifier: every YES instance L of Partition has a short justification, namely two sublists L_1 and L_2 of L that partition L and sum the same

Efficient Verifier

- A decision problem P has an **efficient verifier** if there is a *polynomial-time verifier algorithm* V such that
  - For all YES instance x of P, there exists a justification y, of polynomial size in the size of x, such that running the verifier V on (x, y) returns YES
  - For all NO instances x of P, running the verifier V on (x, y) returns NO on every y
- A *justification* is also known as the *witness*, *certificate*, or *advice*

- Any decision problem taht can be solved in polynomial time has an efficient verifier, since we can just solve the problem
- Some decision problems may *not* have an efficient verifier
  - E.g. the negation of Clique, i.e. Clique-Freedom
  - Input: graph $G = (V, E)$, and integer k
  - Question: does G *not* have a clique of size k?
  - It is *not known* whether Clique-Freedom has an efficient verifier

6.7 P, NP, and NP-Completeness

P and NP Problems

- Efficient(ly): polynomial-time in the input size
- **NP**: the set of decision problems that have *efficient verifiers*
- **P**: the set of decision problems that can be *solved efficiently*
- Since any problem that can be solved efficiently has an efficient verifier, so $P \subseteq NP$

NP-Hard and NP-Complete Problems

- A problem P is **NP-Hard** if *every* problem P' in NP is p-reducible to P, i.e.

$$\forall P' \in \text{NP}, P' \leq_p P$$

  - P is at least as hard to solve as any problem in NP
- A problem P is **NP-Complete** iff P is in NP and P is NP-Hard
  - P is the hardest among all the problems in NP
- Cook-Levin Theorem states that SAT is NP-Complete
  - If SAT $\in$ P, then P = NP
  - If SAT $\notin$ P, then P $\neq$ NP
- Clique is NP-Complete
  1. Clique is in NP because it has an efficient verifier
  2. Clique is NP-Hard because
     - By Cook-Levin, SAT is NP-Hard, i.e. $\forall P \in \text{NP}, P \leq_p \text{SAT}$
     - We previously proved that SAT $\leq_p$ Clique
     - By chaining the results, we get $\forall P \in \text{NP}, P \leq_p$ Clique

Proving a Problem to be NP-Complete

- First see if P is one of the *known* NP-Complete problems
- If the search fails, then show that:
  1. P is in NP, i.e. it has an efficient verifier
  2. Some *known* NP-Complete problem P' is p-reducible to P

6.8 Independent Set

Problem

- Given an undirected graph $G = (V, E)$
- An **independent set (IS)** is a subset of nodes of G that have *no* edges between them
- Question: does G have an IS of size k?

Reduction from Clique to Independent Set

- Consider a graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) \mid (u, v) \notin E\}$
- S is a clique of G iff S is an independent set of $\overline{G}$
  - $\forall u, v \in S, (u, v) \in E$
  - $\forall u, v \in S, (u, v) \notin \overline{E}$

Lemma 6.2. *Independent Set is NP-Complete*

Proof. Independent Set is in NP because given an alleged independent set S of k nodes of G, it is easy to verify that S is an IS of size k of G

Clique $\leq_p$ Independent Set so that IS is NP-Hard since we can transform any instance (G, k) to the Clique problem into an instance $(\overline{G}, k)$ of the IS problem with the same answer, and this transformation takes polynomial time in the input size. □

6.9 Vertex Cover

Problem

- Given an undirected graph $G = (V, E)$
- A **vertex cover (VC)** is a subset S of nodes of G that "covers" every edge of G, i.e. every edge of G is incident to at least one node in S

Reduction from Independent Set to Vertex Cover

- S is an IS of G iff $\overline{S} = V - S$ is a VC of G
  - $\forall (u, v) \in E$, at most one of $\{u, v\}$ is in S
  - $\forall (u, v) \in E$, at least one of $\{u, v\}$ is in $\overline{S} = V - S$

Lemma 6.3. *Vertex Cover is NP-Complete*

Proof. Vertex Cover is in NP because given any alleged VC S of k nodes of G, it is easy to verify that S is a VC of size k of G

Independent Set $\leq_p$ Vertex Cover since we can transform any instance (G, k) of the IS problem into an instance $(G, n - k)$ of the VC problem with the same answer, because G has an IS S of size k iff G has a VC $\overline{S} = V - S$ of size $n - k$. □

6.10 Set Cover

Problem

- Input:
  - A universe of elements U
  - A family S of subsets of U
  - An integer k
- Question: do there exist k sets from S whose union is U?

Lemma 6.4. *Set Cover is NP-Complete*

Proof. Set Cover is in NP because given any alleged set cover consisting of k sets of S, it is easy to verify that the nion of these k sets is U.

Vertex Cover $\leq_p$ Set Cover since given any $G = (V, E)$ and k, we can construct an instance of the set cover problem:

- Set $U = E$
- Set $S = \{S_v \mid v \in V$ where S_v is the set of all the edges incident on $v\}$
- G has a vertex cover of size k iff there are k sets in S whose union is U

□

6.11 Integer Linear Programming Feasibility

Problem

- Input: $b \in \mathbb{Q}^m$ and $A \in \mathbb{Q}^{m \times n}$
- Question: does there exist $x \in \{0, 1\}^n$ such that $Ax \leq b$?

Lemma 6.5. *ILP Feasibility is in NP.*

Proof. Given any $x \in \{0, 1\}^n$ that allegedly satisfies $Ax \leq b$, it is easy to verify that this x satisfies $Ax \leq b$. □

Lemma 6.6. *3SAT $\leq_p$ ILP Feasibility.*

Proof. Take any 3CNF formula φ, we can construct an ILP Feasibility instance as follows:

- For each Boolean variable $x_i \in \{\text{False}, \text{True}\}$ in φ, create a Binary variable $x_i \in \{0, 1\}$
- Represent Boolean literals x_i and $\overline{x}_i$ with x_i and $(1 - x_i)$, where
  - $x_i = \text{True}$ and $\overline{x}_i = \text{False}$ iff $x_i = 1$ and $(1 - x_i) = 0$
  - $x_i = \text{False}$ and $\overline{x}_i = \text{True}$ iff $x_i = 0$ and $(1 - x_i) = 1$
- For each clause C of φ, we want at least one of the three literals to be True, so we require that their sum is at least 1

It is easy to check that this is a polynomial reduction, and that the resulting ILP system has a feasible 0-1 solution iff φ is satisfiable. □

6.12 Graph Colouring

Problem

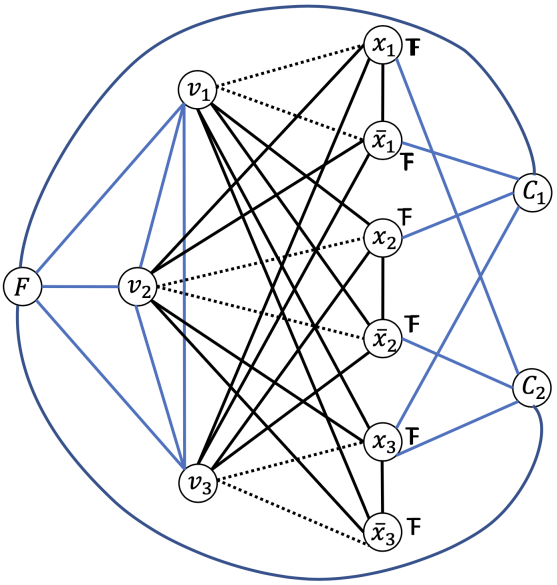
- Input: undirected graph $G = (V, E)$, integer k
- Question: can we colour each node of G using at most k colours such taht no two adjacent nodes have the same colour?

Lemma 6.7. *Colouring is in NP.*

Proof. Given an alleged k-colouring of a graph G, it is easy to verify taht it is indeed a correct k-colouring of G, where we could check that no edge of G has endpoints with the same colour. This can be done in time polynomial in the size of G and k. □

Lemma 6.8. *3SAT $\leq_p$ Colouring.*

Proof. Given any 3CNF formula φ with k variables, we can construct a graph G such that G is $(k+1)$-colourable iff φ is satisfiable. (Example shown below)



1. Construct a node that is coloured "False"
2. For each variable, construct a node of a different colour. Connect all those nodes together, as well as to the "False" node
3. For each variable x, construct two nodes , x and $\overline{x}$, and connect those two
4. For each pair of nodes from (2) and (3), connect them if they don't represent the same variable
5. For each clause, construct a node, and connect it to the nodes from (3) that are *not* literals of the clause
6. Connect the nodes from (5) to the "False" node

If the 3CNF formula is satisfiable due to variable x, then the nodes from (5) can be coloured the same as x from (2). If the 3CNF formula is not satisfiable due to clause C, then the node C from (5) cannot be coloured with any possible colours. □

Graph Colouring Problem

- NP-Complete
- 3-Colouring Problem ($k = 3$) is also NP-Complete
- 2-Colouring Problem ($k = 2$) can be solved in polynomial time
  - G can be coloured using 2 colours iff G is bipartite

6.13 Exact Set Cover

Problem

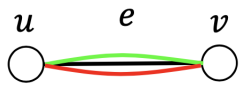
- Input:
  - A universe of elements U
  - A family S of subsets of U
- Question: are there *disjoint* sets in S whose union is U?

Lemma 6.9. *Exact Set Cover is NP-Complete.*

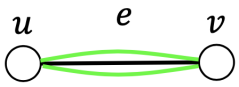
Proof. Given any alleged exact set cover of U, it is easy to verify whether the sets in the set cover are disjoint, and that their union is U.

We will show that Colouring $\leq_p$ Exact Set Cover so that ESC is NP-Hard. Given any instance $G = (V, E)$ and k of the Colouring problem, we can buoild an instance (U, S) of the ESC probem such that G can be coloured with k colours iff there exists disjoint sets in S whose union is U.

- We can transform a *colour conflict* over a shared edge into a *set intersection* by extending the colour of a node to its incident edges



$\{[e, green]\} \cap \{[e, red]\} = \emptyset$



$\{[e, green]\} \cap \{[e, green]\} \neq \emptyset$

- Let $U = V \cup \{[e, i] \mid e \in E \wedge 1 \leq i \leq k\}$
- Let S be the family of subsets of U that contains:
  - $S_{vi} = \{v\} \cup \{[e, i] \mid e \in E \text{ incident to } v\}$ for each $v \in V$ and each colour $1 \leq i \leq k$
  - $T_{ei} = \{[e, i]\}$ for each edge $e \in E$ and each colour i, $1 \leq i \leq k$
- Suppose that G is k-colourable
- The idea is that each S_{vi} contains exactly 1 vertex, as well as its incident edges. In the ESC problem, we need to choose an S_{vi} for each vertex (since we need to cover all vertices), without double-choosing a repeated colour for each edge incident to those vertices. This choice of S_{vi}s will cover all vertices, and some edges
- The remaining edges can be covered by choosing T_{ei}s
- This shows that G is k-colourable implies that S has an exact cover of U

To show the converse:

- Suppose that S has an exact cover of U

- Extract the vertices from subsets S_{vi}s from the exact set cover
- Among those vertices, the pairs that share edges cannot be of the same colour since it would not be allowed by the exact set cover
- This shows that G is k-colourable

□

6.14 Subset Sum

Problem

- Input: set of integers $S = \{w_1, \ldots, w_n\}$, integer W
- Question: is there a subset of S that adds up to exactly W?

Lemma 6.10. *Subset Sum is in NP.*

Proof. Given a subset S' of S that allegedly sums to W, it is easy to verify (in polynomial time in the size of S) whether the elements of S' sum to W. □

Lemma 6.11. *3SAT $\leq_p$ Subset Sum.*

Proof. Given a 3CNF formula φ, we can construct an instance (S, W) of subset sum such that φ is satisfiable iff S has a subset that sums to W. We build a table as follows:

- One column for each variable, and one column for each clause
- One row for each literal l in φ:
  - Has 1 in its variable column
  - Has 1 in the column of every clause where literal l appears
- Each row represents a decimal number in S
- Intuitively: row (i.e. decimal number) selected means that the literal is set to TRUE
- Dummy rows to make the sum in a clause column 4 iff at least one literal is set to TRUE

Example:

$C_1 = \bar{x} \vee y \vee z$
$C_2 = x \vee \bar{y} \vee z$
$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$

	x	y	z	C_1	C_2	C_3
x	1	0	0	0	1	0
¬ x	1	0	0	1	0	1
y	0	1	0	1	0	0
¬ y	0	1	0	0	1	1
z	0	0	1	1	1	0
¬ z	0	0	1	0	0	1
dummies to get clause columns to sum to 4	0	0	0	1	0	0
	0	0	0	2	0	0
	0	0	0	0	1	0
	0	0	0	0	2	0
	0	0	0	0	0	1
	0	0	0	0	0	2
W	1	1	1	4	4	4

There is a truth assignment that satisfies φ iff there is a selection of rows that sums to $1 \cdots 1\ 4 \cdots 4$. □

6.15 Partition

Problem

- Input: multiset of integers S
- Question: can we partition S into two subsets that have the same sum?

Lemma 6.12. *Partition is in NP.*

Proof. Given any alleged partition P and Q of S, we can verify in polynomial time whether it is indeed a partition and the sum of the integers in P and Q is the same. □

Lemma 6.13. *Subset Sum* $\leq_p$ *Partition.*

Proof. Given a set S of integers and an integer k, then

- Compute the sum m of the integers in S
- Build multiset $S' = S \cup \{2k - m\}$
- The sum of the integers in S' is $m + 2k - m = 2k$

We will show that S has a subset that sums to k iff S' can be partitioned into two sets P and Q each summing to k.

Suppose S' can be partitioned into P and Q each summing to k. The integer $2k - m$ added to build S' from S is in P or Q. Say P WLOG, so Q is a subset of S. By assumption, Q sums to k.

Suppose S has a subset that sums to k. Then such subset is also a subset of S'. Since S' sums to $2k$, S' without such subset sums to k. □

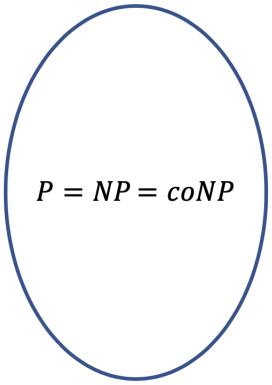
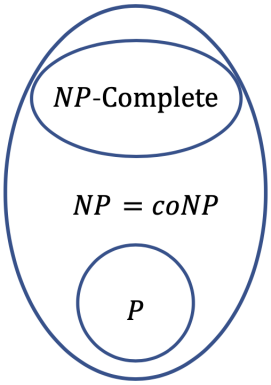
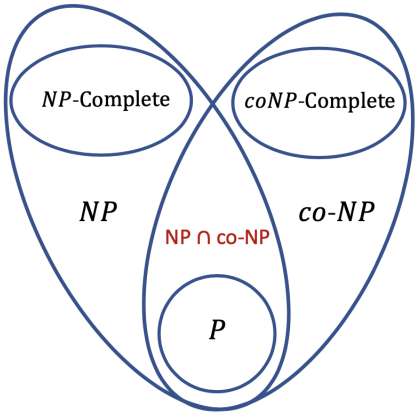
6.16 NP and co-NP

Overview

- **NP** problems have an efficient verifier for YES instances
- **co-NP** problems have an efficient verifier for NO instances
- E.g.
  - SAT $\in$ NP (does there exist a truth assignment satisfying φ?)
  - UNSAT $\in$ co-NP (does there *not* exist a truth assignment satisfying φ?)
  - Clique $\in$ NP (does there exist a clique of size k?)
  - Clique-Freedom $\in$ co-NP (does there *not* exist a clique of size k?)

Open Questions

- NP $\cap$ co-NP is the set of problems that have an efficient verifier for *both* YES and NO instances
- It is clear that P $\subseteq$ NP $\cap$ co-NP
- We don't know whether P $\subset$ NP $\cap$ co-NP, i.e. whether there is a problem that is in NP $\cap$ co-NP but not in P

$P = NP$	$P \neq NP$ $NP = coNP$	$P \neq NP$ $NP \neq coNP$
		
P = NP ?	NP = co-NP ?	P ⊂ NP ∩ co-NP ?

7 Approximation Algorithms

7.1 Intro

Approximation Algorithms for Optimization Problems

- There is a function Cost we want to *minimize*, and a function Profit we want to *maximize*
- Let ALG be an algorithm for the problem, and I an instance of the problem
  - $ALG(I)$ is the solution returned by algorithm ALG on instance I
  - $OPT(I)$ is the optimal solution for instance I
- ALG is a **c-approximation algorithm** iff for every instance I of the problem:

$$\text{Cost}(ALG(I)) \leq c \cdot \text{Cost}(OPT(I)), \qquad \text{or}$$
$$\text{Profit}(ALG(I)) \geq \frac{1}{c} \cdot \text{Profit}(OPT(I))$$

  - E.g. a 2-approximation algorithm has at most *twice* the optimal cost, or at least *half* the optimal profit

7.2 Travelling Salesman Problem

Minimum Spanning Tree (MST)

- A **tree** is a cocnnected undirected graph with no cycles
- Let $G = (V, E)$ be an undirected connected graph
- The **spanning tree** of G is a tree T that connects (i.e. spans) all the nodes of G
  - $T = (V, E')$ such that $E' \subseteq E$
- Let G be *weighted*. The **weight** of a spanning tree T of G is the *total* weight of its edges
- A **minimum spanning tree** of G is a *minimum-weight* spanning tree of G
- Given any weighted graph G with n nodes and m edges, we can find an MST of G in $\mathcal{O}(m \log n)$ time, using Kruskal's or Prim's algorithm

Travelling Salesman Problem (TSP)

- Input: undirected *complete* graph $G = (V, E)$ with nonnegative edge costs (i.e. weights)
- Output: a *tour* of G of *minimum* total edge cost (called TSP tour)
- A **tour** is a cycle that visits *every* node of G *exactly once*, and returns to the starting node
- Brute-force algorithm: find *all* tours of G and select the minimum cost one, takes exponential time
- TSP is NP-hard

Δ-TSP

- Assume the edge costs satisfy the *triangle inequality*, denoted Δ-inequality:
  - For all nodes u, v, w of G: $c(u, v) \leq c(u, w) + c(w, v)$
  - Any indirect route is at least as long as the direct route
- Input: undirected complete graph $G = (V, E)$ with nonnegative edge costs that satisfy the Δ-inequality

- A *tour* of G of *minimum* total edge cost (TSP tour)
- Δ-TSP is still NP-hard
- However we can find an *approximate* solution to Δ-TSP efficiently
  - There is a polynomial-time algorithm that finds a tour of G whose cost is within a fixed constant factor of the optimal (i.e. min-cost) TSP tour

Lemma 7.1. *Let TSP be an* optimal *(i.e. min-cost) TSP tour of G. Let MST be a* minimum spanning tree *of G. Then*

$$cost(MST) \leq cost(TSP)$$

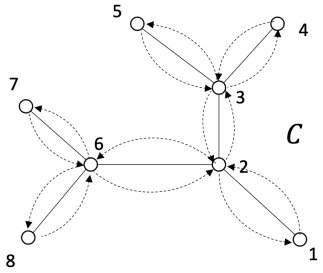
Proof. Consider any TSP of G and remove any edge e of TSP, obtaining a spanning tree.

$$\text{cost(MST)} \leq \text{cost(ST)} \leq \text{cost(TSP)}$$

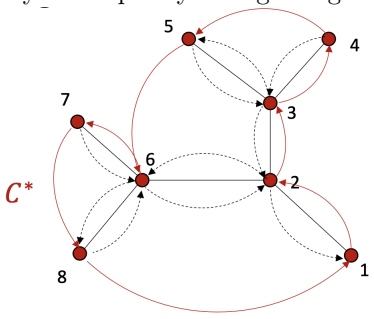
□

Approximation Algorithm for Δ-TSP

1. Find an MST of G
   - Can be done in $\mathcal{O}(m \log n)$ time
2. Do a *full walk* of the MST (i.e. perform DFS)
   - This gives a cycle C of G that uses each edge of the MST *twice*



   - $\text{cost}(C) = 2\text{cost(MST)}$
   - Note that C is a *cycle* of G and is *not* a tour, since C revisits some nodes several times
3. Transform C into a tour C^* by using shortcuts to remove repeated nodes
   - By Δ-inequality of edge weights, $\text{cost}(C^*) \leq \text{cost}(C)$



The cost of the tour C^* found by this algorithm is *at most twice* the cost of a TSP tour of G

$$\text{cost}(C^*) \leq 2\text{cost(MST)} \leq 2\text{cost(TSP)}$$

7.3 Min Vertex Cover and Max Matching

Vertex Cover

- Let $G = (V, E)$ be an undirected graph
- **Vertex cover (VC)**: a subset V' of the nodes of G that "touches" every edge of G, i.e. every edge has at least one endpoint in V'
- **Minimum vertex cover**: a VC with minimum size

Matching

- Let $G = (V, E)$ be an undirected graph
- A **matching** M in G is a subset of edges of G such that no two edges are incident on the same node, i.e. no two edges share a node
- **Maximum matching**: a matching of maximum size

Vertex Cover vs. Matching

- We proved that for any matching M in G, and any VC S of G, $|M| \leq |S|$ (see Lemma 4.9)

Min Vertex Cover

- Input: undirected graph $G = (V, E)$
- Output: a minimum vertex cover S of G
- We already proved that this proble is NP-Complete (see Lemma 6.3)
- Want to find a good *approximation* algorithm

Greedy Edge-Selection Algorithm

- Incrementally build a vertex cover S as follows:
  - Start with $S = \varnothing$
  - While there exists an edge (u, v) that is not yet "covered" by the nodes already in S, add *both* u and v to S

```
1   Greedy-Verrtex-Cover(G): # G = (V, E) undirected
2       S <- {}
3       M <- {}
4       E* <- E
5       while E* != {} do
6           let (u, v) be any edge in E*
7           S <- S union {u, v}
8           M <- M union {(u, v)}
9           delete from E* every edge incident to u and v
10      return S
```

- At the end of this algorithm:
  - S is a vertex cover of G
  - M is a matching of G
  - $|S| = 2|M|$
- This is a 2-approximation algorithm for the minimum vertex cover problem
- Let S^* be the minimum vertex cover. We know that $|M| \leq |S^*|$ by Lemma 4.9, therefore we have that

$$|S| = 2|M| \leq 2|S^*|$$

7.4 Weighted Vertex Cover

Problem

- Input: undirected graph $G = (V, E)$, weights $w : V \to \mathbb{R}^{\geq 0}$
- Output: vertex cover S of minimum *total weight*
- Generalization of the unweighted vertex cover problem, which is already NP-Complete
- The previous algorithm would not work, since it does not take in account of the weights

ILP Formulation

- For each vertex v, create a *bianry* variable $x_v \in \{0, 1\}$, indicating whether v is in the vertex cover
- Then computing the *min weight vertex cover* is equivalent to solving the following ILP problem:

$$\min \sum_v w_v x_v$$

subject to

$$\begin{aligned} x_u + x_v &\geq 1 \quad \forall (u, v) \in E \\ x_v &\in \{0, 1\} \quad \forall v \in V \end{aligned}$$

- However ILP is NP-Hard

LP Relaxation

- Instead of ILP with binary variables, we could solve an LP with real variables:

$$\min \sum_v w_v x_v$$

subject to

$$\begin{aligned} x_u + x_v &\geq 1 \quad \forall (u, v) \in E \\ x_v &\geq 0 \quad \forall v \in V \end{aligned}$$

Rounding LP Solution

- LP minimizes the same objective over a larger feasible space, so optimal LP objective value $\leq$ optimal ILP objective value
- However optimal LP solution x^* is *not* a binary vector
- We want to *round* x^* to some binary vector $\hat{x}$ that is a solution to the ILP without increasing the objective value too much
- Can round as follows: for each $v \in V$:

$$\hat{x}_v = \begin{cases} 0, & \text{if } x_v^* < 0.5 \\ 1, & \text{if } x_v^* \geq 0.5 \end{cases}$$

Lemma 7.2. *$\hat{x}$ is a feasible solution of ILP (i.e. it is a vertex cover)*

Proof. For every edge $(u, v) \in E$, $x_u^* + x_v^* \geq 1$ because x^* is a solution of LP. This means that at least one of x_u^* and x_v^* is at least 0.5. Therefore at least one of $\hat{x}_u$ and $\hat{x}_v$ is 1, so $\hat{x}_u + \hat{x}_v \geq 1$. □

This is a *2-approximation algorithm* for the weighted vertex cover problem

$$\sum_v w_v \hat{x}_v \leq 2 \sum_v w_v x_v^* = 2(\text{LP optimal}) \leq 2(\text{ILP optimal}) = 2(\text{min weight VC})$$

General LP Relaxation Strategy

- Reduce an NP-Complete problem to ILP
  - Maximize $c^\top x$ subject to $Ax \leq b$, $x \in \mathbb{N}$
- Solve the corresponding LP
  - Maximize $c^\top x$ subject to $Ax \leq b$, $x \in \mathbb{R}^{\geq 0}$ (LP relaxation)
  - LP optimal value $\geq$ ILP optimal value
- Let x^* be the LP optimal solution, i.e. $c^\top x^* = \text{LP optimal}$
- *Round* x^* to an *integer solution* $\hat{x}$ such that
  - $\hat{x}$ is still a feasible solution, i.e. $A\hat{x} \leq b$
  - For some approximation factor ρ:

$$c^\top \hat{x} \geq \frac{c^\top x^*}{\rho} = \frac{\text{LP optimal}}{\rho} \geq \frac{\text{ILP optimal}}{\rho}$$

- This gives a ρ-approximation

7.5 Makespan Minimization

Makespan Problem

- Input: m identical machines, n jobs, job j requires processing time t_j
- Output: assign jobs to machines to minimize makespan
- Constraints:
  - Each job must run contiguously on one machine
  - Each machine can process at most one job at a time
- Let $S[i]$ = set of jobs assigned to machine i
- Load on machine i is $\sum_{j \in S[i]} t_j$
- Goal is to minimize the makespan $L = \max_i L_i$
- Even the special case of $m = 2$ is NP-hard, can reduce Partition to Makespan

Greedy Algorithm

- Consider the n jobs in some "nice" sorted order
- Assign each job j to a machine with the *smallest load* so far
- Can be implemented in $\mathcal{O}(n \log m)$ time using priority queue (min heap)

Theorem 7.3 (Graham 1966)**.** *Regardless of the order, greedy gives a 2-approximation.*

Proof. Let L denote the makespan found by the greedy algorithm. Let L^* be the *optimal* makespan. First notice that

$$L^* \geq \max_j t_j$$

since some machine must process the job with the highest processing time. Second notice that

$$L^* \geq \frac{1}{m} \sum_j t_j$$

since the total processing time is $\sum_j t_j$, and that at least one of the m machines must do at least $1/m$ of this work (by pigeonhole principle).

Suppose machine i is the *bottleneck* under greedy (so $L = L_i$). Let j^* be the last job assigned to machine i by greedy. Right before j^* was assigned to i:

- The load on machine i was $L'_i = L_i - t_{j^*}$
- This was the *smallest load* among all the machines at that time, i.e. $L_i - t_{j^*} \leq L'_k$ for all k

After this assignment, the machine loads L'_k can only increase, i.e. $L'_k \leq L_k$ for all k. Therefore

$$\begin{aligned}
&L_i - t_{j^*} \leq L_k \qquad \forall k, 1 \leq k \leq m \\
&m(L_i - t_{j^*}) \leq \sum_k L_k \qquad \text{By summing over all } k \\
&m(L_i - t_{j^*}) \leq \sum_j t_j \qquad \text{Because } \sum_k L_k = \sum_j t_j \\
&L_i - t_{j^*} \leq \frac{1}{m} \sum_j t_j
\end{aligned}$$

Isolating L_i:

$$\begin{aligned}
L_i &\leq t_{j^*} + \frac{1}{m} \sum_j t_j \\
&\leq L^* + L^* \qquad \text{By the two observations at the beginning of the proof} \\
&= 2L^*
\end{aligned}$$

□

- This is not tight. The tight bound is $2 - \frac{1}{m}$

Longest Processing Time (LPT) First Algorithm

- Run the greedy algorithm, but consider jobs in *decreasing order of length*
- $t_1 \geq t_2 \geq \cdots \geq t_n$

Theorem 7.4. *Greedy with longest processing time first gives a* $3/2$ *approximation.*

Proof. First notice that if the *bottleneck* machine has only 1 job, then the solution is optimal, since the optimal solution must assign that job to some machine. Second notice that if there are more than m jobs, then

$$L^* \geq 2t_{m+1} \quad \text{or equivalently} \quad t_{m+1} \leq \frac{L^*}{2}$$

- Consider the first $m+1$ jobs (i.e. in decreasing order of length)
- Each one of these jobs requires at least t_{m+1} time
- Since there are $m+1$ of them and only m machines, by pigeonhole principle, at least two of them will end op on the same machine
- So the final load on that machine is $\geq 2t_{m+1}$

Consider the *bottleneck* machine i (so makespan is $L = L_i$). Let j^* be the *last* job that greedy assigned to machine i. Then there are two cases:

1. Machine i has only one job (i.e. j^*)
   - By the first obserbation, greedy is optimal in this case
2. Machine i has at least 2 jobs
   - Greedy first assigns job $1, 2, \ldots, m$ with lengths $t_1 \geq t_2 \geq \cdots \geq t_m$
   - Since j^* is *not* the first job given to machine i, $j^* \geq m+1$
   - So $t_{j^*} \leq t_{m+1}$
   - By the second observation, $t_{m+1} \leq \frac{L^*}{2}$, so $t_{j^*} \leq \frac{L^*}{2}$
   - We previously proved (for greedy with arbitrary ordering) that $L_i \leq t_{j^*} + L^*$

$$L = L_i \leq t_{j^*} + L^* \leq \frac{L^*}{2} + L^* = 1.5L^*$$

□

- This is not tight. The tight bound is $\frac{4}{3} - \frac{1}{3m}$

7.6 Local Search Paradigm

Local Search

- A heuristic paradigm for solving complex problems
- Idea
  - Start with some solution S
  - While there is a *better* solution S' in the *local neighbourhood* of S, switch to S'
- Need to define
  - What is "better"
  - What is a "local neighbourhood"
- Sometimes local search gives an optimal solution
  - E.g. network flow
  - Start with zero flow
  - Local neighbourhood: set of all flows which can be obtained by augmenting the current flow along a path in the residual graph
  - Better: higher flow value
- Sometimes local search brings us to a local optimum
- Want to bound the ratio between:
  1. The *optimal solution*
  2. The *worst solution* that local search might return

7.7 Max Cut

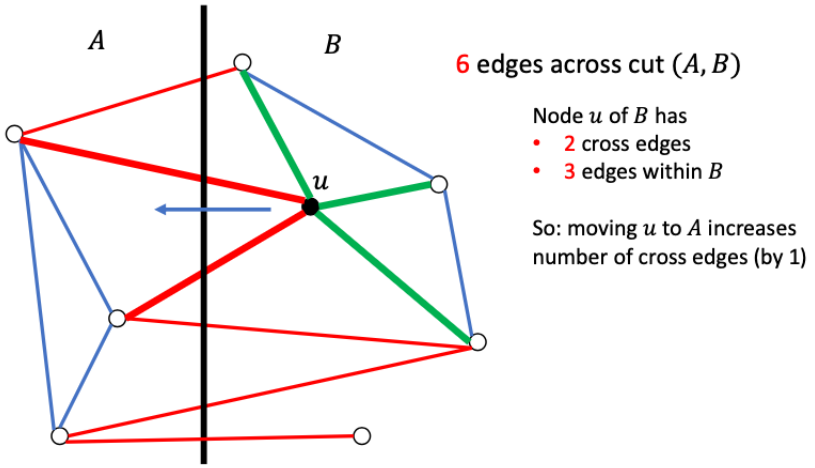
Problem

- Input: an undireccted graph $G = (V, E)$
- Output: a partition (A, B) of V that maximizes the number of edges going across the (A, B) cut, i.e. maximizes $|E'|$ where $E' = \{(u, v) \in E \mid u \in A \wedge v \in B\}$
- This problem is NP-hard

Local Search Algorithm

1. Initialize (A, B) arbitrarily
2. While there is a vertex u such that moving u to the other side increases the number of edges across the cut, then move u to the other side

- Moving u increases the number of edges across the cut iff u has *more* incident edges going *within* its side of the cut than *across* the cut
- Every iteration increases the number of edges across the cut by at least 1, so the algorithm must stop in at most $|E|$ iterations



Lemma 7.5. *At the end of this algorithm, at least half of all edges go across the cut.*

Proof. Let (A, B) be the cut found by the algorithm. Let E be the set of all edges. Let E^c be the set of edges that cross (A, B). We will show that $|E^c| \geq |E|/2$. For each node v, define

- E_v: edges incident to v
- E_v^c: edges incident to v that cross the cut
- E_v^i: edges incident to v that do not cross the cut

Therefore

$$E_v = E_v^c \cup E_v^i$$

Because at the end of the algorithm, v has at least as many edges going *across* the cut as *within* the cut, we have that

$$|E_v^c| \geq |E_v^i|$$

To derive the inequality:

$$
\begin{aligned}
&|E_v| = |E_v^c| + |E_v^i| \\
&|E_v| \leq 2|E_v^c| \\
&\sum_v |E_v| \leq 2 \sum_v |E_v^c| \\
&2|E| \leq 2(2|E^c|) \qquad \text{Since } \sum_v |E_v| = 2|E| \text{ and } \sum_v |E_v^c| = |E^c| \\
&|E| \leq 2|E^c| \\
&|E^c| \geq |E|/2
\end{aligned}
$$

□

7.8 Exact Max k SAT

Problem

- Input: A k-CNF formula $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, each clause C_i has exactly k literals, and each clause C_i has a weight $w_i \geq 0$
- Output: a truth assignment τ *maximizing* the number (or total weight) of clauses satisfied under τ
- Max 2 SAT is NP-hard

Local Search Algorithm

1. Initialize truth assignment τ to the variables x_j of φ arbitrarily
2. While there exists τ' in the *local neighbourhood* of τ such that τ' *increases* the number (or total weight) of clauses that are satisfied, then replace τ with τ'

- Local neighbourhood of τ:
  - $N_d(\tau)$ is the set of all truth assignments obtained by changing the value of *at most* d variables in τ
  - For $d = 1$, this consists of all the truth assignments obtained by changing the value of $\tau(x_j)$ for *one* variable x_j of φ

Theorem 7.6. *The local search algorithm with $d = 1$ gives a $\frac{2}{3}$ approximation to Exact Max 2 SAT.*

Proof. Let τ be a *local optimum* ffound by the local search algorithm. Then τ partitions the set S of all the clauses of φ into:

- S_0: the set of clauses *not* satisfied under τ
- S_1: the set of clauses where *exactly one* literal is true under τ
- S_2: the set of clauses where *both* literals are true under τ
- Let $W(S_0), W(S_1), W(S_2)$ be the corresponding total weights, so τ achieves weight $W(S_1) + W(S_2)$

Knowing that the optimal solution τ^* achieves *at most* $W(S_0) + W(S_1) + W(S_2)$, we will show that

$$W(S_1) + W(S_2) \geq \frac{2}{3} [W(S_0) + W(S_1) + W(S_2)]$$

or equivalently

$$W(S_0) \leq \frac{1}{3} [W(S_0) + W(S_1) + W(S_2)]$$

A clause involves variable j if it contains x_j or $\overline{x}_j$. Define

- A_j: the set of clauses in S_0 involving varible j
- B_j: the set of clauses in S_1 that are True because the literal of variable j is true under τ
- Let $W(A_j), W(B_j)$ be the corresponding total weights

Each clause in S_0 involves 2 variables, so each clause in S_0 appears *exactly twice* in $A_1, A_2, \ldots, A_n$ where n is the number of variables in φ. So the total weight of the clauses in S_0 is *half* the total weight of the clauses in $A_1, A_2, \ldots, A_n$, i.e.

$$2W(S_0) = \sum_j W(A_j)$$

By construction of S_1 and B_j, we know that $S_1 = \bigcup_j B_j$, therefore

$$W(S_1) = \sum_j W(B_j)$$

If we *flip* the truth assignment $\tau(x_j)$ of a variable x_j:

- All the clauses in A_j become True (because they involve x_j but are False under τ)
- All the clauses in B_j become False (because they were True *only* because of $\tau(x_j)$)
- The other clauses in $S_0 \cup S_1 \cup S_2$ do not change truth value
- Therefore the net change in total weight is $W(A_j) - W(B_j)$
- Since the τ that we consider is a *local optimum*, this net change cannot be positive, i.e. $W(A_j) - W(B_j) \leq 0$, and so
$$W(A_j) \leq W(B_j)$$
- Summing over all j gives
$$\sum_j W(A_j) \leq \sum_j W(B_j)$$

Chaining the inequalities:

$$2W(S_0) = \sum_j W(A_j) \leq \sum_j W(B_j) = W(S_1)$$

Further deriving:

$$\begin{aligned} 3W(S_0) &\leq W(S_0) + W(S_1) \\ 3W(S_0) &\leq W(S_0) + W(S_1) + W(S_2) \\ W(S_0) &\leq \frac{1}{3}\left[W(S_0) + W(S_1) + W(S_2)\right] \end{aligned}$$

□

Better Approximation

- We just threw in $W(S_2)$ in our proof, however we could find a way to ensure that $W(S_0) \leq W(S_2)$ to achieve a 3/4 approximation
- We could add an operation of *flipping* all truth assignments of τ, denoted τ^c, which could guarantee that $W(S_0) \leq W(S_2)$
- Neighbourhood $N_1(\tau) \cup \tau^c$ gives a 3/4 approximation for Exact Max 2 SAT

Another Better Approximation

- Notice that S_2 clauses are more "robust" than those in S_1
- We could give more importance to creating S_2 clauses by modiffying the objective function to

$$1.5W(S_1) + 2W(S_2)$$

- While there exists $\tau' \in N_1(\tau)$ that increases $1.5W(S_1) + 2W(S_2)$, we switch to τ'
- This gives a 3/4-approximation to Exact Max 2 SAT

Exact Max k SAT

- The modified local search works for $k \geq 2$
- It gives $\frac{2^k-1}{2^k}$ approximation for Exact Max k SAT
- For $k = 3$, this is a 7/8 approximation for Exact Max 3 SAT
  - Achieving $7/8 + \epsilon$ approximation for any $\epsilon > 0$ is NP-hard

8 Randomized Algorithms

8.1 Exact Max k SAT

Problem

- Input: A k-CNF formula $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, each clause C_i has exactly k literals, and each clause C_i has a weight $w_i \geq 0$
- Output: a truth assignment τ *maximizing* the number (or total weight) of clauses satisfied under τ

Naive Randomized Algorithm

- Set each variable to TRUE with probability 1/2, and FALSE with probability 1/2
- For each cluase C_i:
  - $\Pr[C_i \text{ is not satisfied}] = (1/2)^k$ because C_i has k literals
  - $\Pr[C_i \text{ is satisfied}] = 1 - (1/2)^k = (2^k - 1)/2^k$
- Let W_i be the weight that we get from C_i under random τ
  - W_i is w_i if τ satisfies C_i, and 0 otherwise
  - So $\mathbb{E}[W_i] = w_i \cdot \Pr[C_i \text{ is satisfied}] + 0 \cdot \Pr[C_i \text{ is not satisfied}] = w_i \cdot \frac{2^k-1}{2^k}$
- Let W be the *total* weight that we get from φ under random τ

$$
\begin{aligned}
W &= W_1 + W_2 + \cdots + W_m \\
\mathbb{E}[W] &= \sum_{i=1}^{m} \mathbb{E}[W_i] \qquad \text{By linearity of expectations} \\
&= \sum_{i=1}^{m} w_i \cdot \frac{2^k - 1}{2^k} \\
&= \frac{2^k - 1}{2^k} \sum_{i=1}^{m} w_i \\
&\geq \frac{2^k - 1}{2^k} \cdot OPT
\end{aligned}
$$

Derandomization

- The algorithm is making some random choices, and sometimes they turn out to be good
- We want to make these good choices *deterministically*
- Idea
  - First find a good choice for x_1 where $x_2, \ldots, x_n$ are random
  - Then find a good choice for x_2 where $x_3, \ldots, x_n$ are random
  - Etc.
  - Finally find a good choice for x_n
- Notice that

$$\mathbb{E}[W(\tau)] = \Pr[x_1 = \text{True}] \cdot \mathbb{E}[W(\tau) \mid x_1 = \text{True}] + \Pr[x_1 = \text{False}] \cdot \mathbb{E}[W(\tau) \mid x_1 = \text{False}]$$

  so

$$\mathbb{E}[W(\tau)] = \frac{1}{2} \cdot \mathbb{E}[W(\tau) \mid x_1 = \text{True}] + \frac{1}{2} \cdot \mathbb{E}[W(\tau) \mid x_1 = \text{False}]$$

  therefore *at least one* of $\mathbb{E}[W(\tau) \mid x_1 = \text{True}]$ and $\mathbb{E}[W(\tau) \mid x_1 = \text{False}]$ must be greater than or equal to $\mathbb{E}[W(\tau)]$

- We could compute *both* and take the better of the two, giving an expected total weight *at least as good as* $\mathbb{E}[W(\tau)]$, i.e. set $x_1 = v \in \{\text{True}, \text{False}\}$ such that $\mathbb{E}[W(\tau) \mid x_1 = v] \geq \mathbb{E}[W(\tau) \mid x_1 = \overline{v}]$
- This *deterministic* setting of x_1 (together with the *random* choice for $x_2, \ldots, x_n$) gives an expected total weight at least as good as $\mathbb{E}[W(\tau)]$, so the expected total weight is still $\geq \frac{2^k-1}{2^k} \cdot OPT$
- We do the same for each of $x_2, \ldots, x_n$

```
for i <- 1, ..., n do
    if E[W(tau) | x_1 = v_1, ..., x_i-1 = v_i-1, x_i = T] >= E[W(tau) | x_1 = v_1, ...,
        x_i-1 = v_i-1, x_i = F] then
        v_i <- T
    else
        v_i <- F
    x_i <- v_i
```

- The derandomized algorithm achieves

$$W(\tau) \geq \frac{2^k - 1}{2^k} \cdot OPT$$

- To compute $\mathbb{E}[W(\tau) \mid x_1 = v_1, \ldots, x_{i-1} = v_{i-1}, x_i = \text{True}]$
  - Compute this conditional probability for each clause C_j as follows:
    - Set the values of $x_1, \ldots, x_{i-1}, x_i$ to $v_1, \ldots, v_{i-1}, \text{True}$
    - If C_j is True under this setting, then the corresponding probability is 1
    - If C_j is False under this setting, then the corresponding probability is 0
    - Otherwise C_j must have $l \geq 1$ literals with no truth assignments yet, in which case the corresponding probability that C_j is satisfied is $1 - (1/2)^l$