CSE 3500 Master Notes

Lecture 1

Tues. Aug. 27, 2019

Algorithms

- Finite → Must terminate
- Complete → Always give a solution when exists
- Correct → Always give a correct solution

ideally, the alg. should also be effective, work in reasonable time

Formulate the Problem

- Use mathematical precision → ask a concrete question

Design an algorithm for the problem

- Use known strategies

Analyze the algorithm

- Prove termination
- Prove completeness
- 3.20Prove correctness
- Evaluate Complexity (running time)

Introduction to **fundamental design techniques** using 5 representative problems

- Interval scheduling
- Weighted Interval Scheduling
- Bipartite Matching
- Independent Set
- Competitive Facility Location

Graph - consists of a pair of sets G = (V, E)

- A collection of V nodes (vertices(
- A collection "E" of edges

Edge - joins two nodes

- An edge e ⊆ E is rep. as 2-element subset of V: e = [u,v], for some u,v ⊆ V, where u & v are called the ends "e"

- Graphs can be directed, unidirectional pointers
- or Undirected, or go in both directions

Interval Scheduling

We dispose of one resources have multiple requests

Request - Use the resource during a given time

CSE 3500 Master Notes

Rule - No 2 overlapping requests can be accepted
Goal - Maximize the # of accepted requests

ex.): we have "n" requests labeled 1, … n
start & finish times (s_i, f_i)

2 requests are **compatible** if they do not overlap

A set of request is **compatible** if all pairs of request are **compatible**

*can never be compatible with yourself

A greedy approach to solve the interval scheduling problem is…

- Sort the request w.r.t. (with respect to) finishing time
- Pick the request with earliest finishing time
- Discard all request in compatible w/ it
- Repeated until the set of request is empty

1. Greedy Method - make decisions right away, decisions are IRREVOCABLE, made one at a time. Gives an **Optimal** solution
***Explanation (not formal proof)**

- The request picked & all requests discarded in any given step are incompatible
- At most one request out of this set can be in the optimal soln.
- exactly 1 of them is in the greedy soln.

∴ soln. **OPTIMAL**

Weighted Interval Scheduling

requests are associated with WEIGHTS → V1

- Goal: find a compatible subset that maximizes the total weight
- compatible means NO OVERLAP

Generally, there is no simple greedy rule that solves the problem → use dynamic programming instead
2. Dynamic Programming - Make decision one at a time, examine the decision sequence → see if an optimal decision sequence contains optimal decision sub-sequences
3. Bipartite Matching - bipartite graphs that connect only between the two groups not between members of the same group
Matching - A subset of edges such that no node is connected by more than one edge
Perfect Matching - A matching where every node appears exactly once

The bipartite matching problem models objects being assigned to other objects
ex.):
- dance partners → men, women
- students → projects

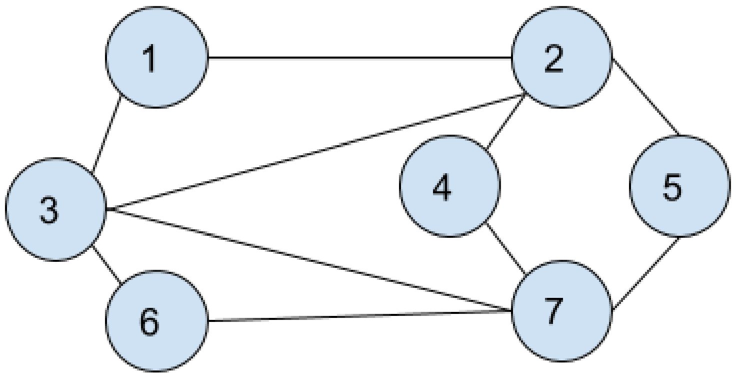
Problem: Given an arbitrary graph, find a matching of maximize size
- If the two sets (x,y) have the same size |x| = |y| = n
- A perfect matching exists iff (if and only if) the matching of size "n"
- **Idea:** build up matchings of increasing size inductively, backtracking along the way

Independent Set

The independent set prob. is more general: interval scheduling & bipartite matching are special cases.

- Given graph G = (V, E), S ⊆ V is independent if no 2 nodes in S are connected by edges
- **Goal:** Find max. ind. set

ex.):



- max. ind. set: **{1, 4, 5,6}**
- when finding max. ind. set, must prove that it is correct

Lecture 2

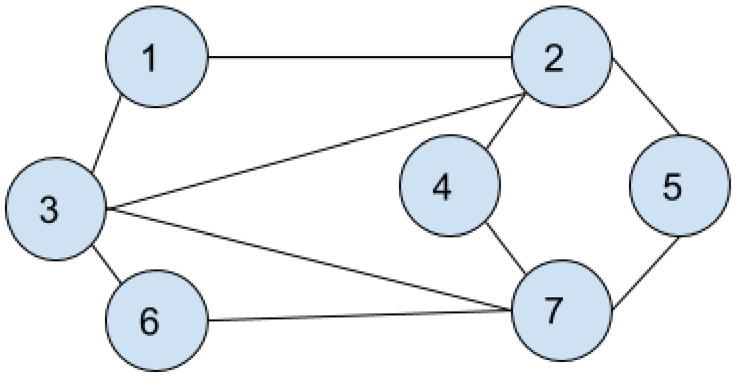
Thurs. Aug. 30, 2019

Independent Set

The independent set prob. is more general: interval scheduling & bipartite matching are special cases.

- Given graph G = (V, E), S ⊆ V is independent if no 2 nodes in S are connected by edges
- **Goal:** Find max. ind. set

ex.):



- max. ind. set: **{1, 4, 5,6}**
- when finding max. ind. set, must prove that it is correct

One Instance

"Proof" of maximality:

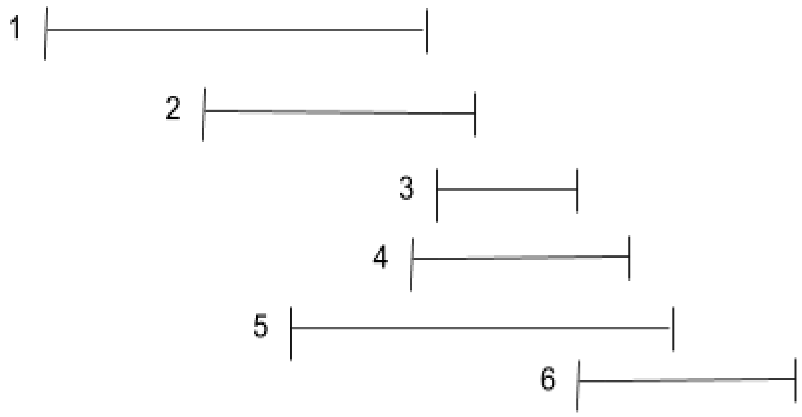
- Only one node from the subset {1, 2, 3}
- Only one node from the subset {3, 6, 7}
- Nodes 4 & 5 left

→ max 4 nodes in any ind. set

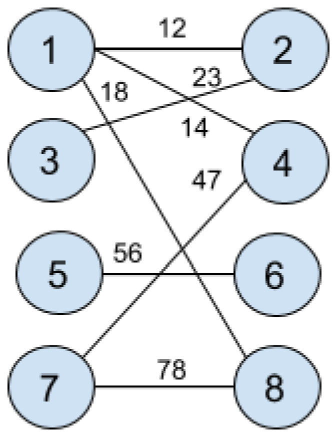
Interval scheduling as Independent Set

How can we transform the interval scheduling problem into a graph?

- **Nodes -**
- **Edges -**



- Using "Greedy" Algorithm allows use to choose the most optimal solution: max. ind. set

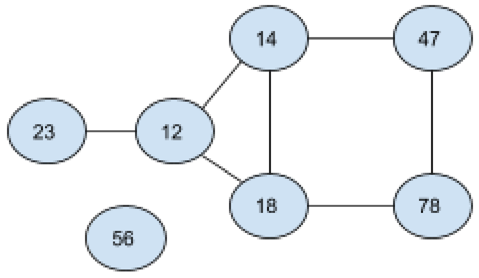


Bipartite Matching as Independent Set
How can we transform the bipartite matching prob. into graph?
The other way around:

- Nodes: edges
- Edges: between edges sharing a node

Perfect matching exists (of size 4) → gives optimal solution
Equivalent Problems - Solve one and you have solved the other (prob.)

Graph of edges (vs. graph of nodes above)



Maximum ind. set:

- {14, 23, 56, 78}
- {18, 23, 47, 56}

Special Cases of Independent Set

We have shown that any **interval scheduling** or **bipartite matching** problem can be solved as an **ind. set** prob.

Is the opposite? → unfortunately not!

Not all graphs can be interpreted as **interval scheduling** or **bipartite matching** problem

- General cases were not proved/talked on, will revisit
- All **interval scheduling & bipartite matching** can be solved as graphs

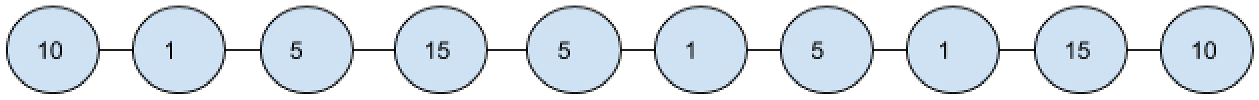
NP-Complete Problems

No efficient algorithm is known for the independent set problem
Brute-force approach: Try all subset & keep the best, might be the best we can do!
NP-Complete Problems

- No efficient solution
- Verifiable in polynomial time
- All problems are equivalent

Competitive Facility Location
Two-Player Game - Given zones with weights, players take turns in picking zones such that chosen zones at all times form an independent set. The players want to maximize their total gain
Question - Given a threshold value: Is there a strategy for player 2 that guarantees a gain larger than the given value?

ex.): Franchising, say coffee shops, cannot (or should not) open shops adjacent to a shop that has been already set up



- P2 has target 20 → guaranteed win
- P2 has target 25 → no win

It is hard to find a general algorithm that will work for all instances

- unlike before, the answer cannot be easily verified:
- all possible choices for both players must be tested
- The competitive facility location belongs to a class of problems called PSPACE (polynomial space)

Stable Matching Problem

- companies propose internships
- students want internships
- everyone has a wishlist

Algorithm: Gale-Shapley -

- companies get to choose
- 1st companies get to choose favorite students
- next comp. get to pick "their" fav. student if they don't have an internship already
- if they already have one, and are asked, they can choose to switch or stay w/ orig.
- If switched, that company that had already selected intern switched out, GET TO GO AGAIN

Notes:

- Could 2 students work together to block a 3 person from their desired internship
- Outcome depends on algorithm?

HW:

- 3 students, 3 companies
- Show algorithm for internship matching,
- Want 1 case that show without adaptation, and 1 case where students get their 1st choice after adaptation
- Outcome depends on algorithm?
- due Weds. Night (presumed 11:59pm)

Lecture 3

Tues. Sept. 3, 2019

An Introduction to Analysis of Algorithms

Brute Force Search

Problems are of a discrete nature → **combinatorics**

Brute Force Search - Generate the space of all possible candidates for the soln. & perform an exhaustive search

● Find an element in a list:	linear
● FCompare all pairs of elements	quadratic
● Check all subsets:	factorial

Brute Force Search can be used as a baseline when benchmarking algorithms

Computational Tractability

An algorithm is **efficient** if, when implemented, it runs quickly on real input instances

- quality of the implementation?
- What is a real instance?
- Does the algorithm

An algorithm is efficient if it achieves qualitatively better worse-case performance, at any analytical level, than brute-force search

- Vague: What does "qualitatively better" really mean?

Worst-Case Running Time

We count the # of primitive Steps

- **One line of high-level code**
- **one assignment**
- **one assembly instruction**

Worse Case: Longest possible running time the algorithm could have over all input of size "n"

- how does this scale w/ n

Computational Tractability - an algorithm is efficient if it has a polynomial running time

Scaling with Growing Input size

Increase input size by one

CSE 3500 Master Notes

Input Size	n	n + 1
linear	n	n + 1
quadratic	n^2	$(n + 1)^2 = n^2 + 2n + 1$
polynomial	n^d	$(n + 1)^d = n^d$ ...
exponential	2^n	$2^n + 1$

Increase input size by a factor two

Asymptotic Upper Bound

T(n) >= 0 - The worst case running time of a certain algorithm mon an input of size n
Big O Notation - T(n) is O(f(n)) if, for some n_0 >= 0 and constant c > 0, for all n >= n_0 we have T(n) <= c * f(n)
Ex.): $3n^2 + n + 2$ is $O(n^2)$

Asymptotic Lower Bound

Big Ω Notation - T(n) is Ω(f(n)), for some n >= 0 and constant d > 0, for all n >= n_0 we have T(n) >= d * f(n)

Asymptotic Tight Bound

Big Theta Notation - T(n) is Theta(f(n) if T(n) is both O(f(n)) and omega(f(n))

- A tight bound can sometimes be found using limits:

2.1 Ratio of Functions - let f & g be functions such that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

for some constant c >) , then f(n) is Theta(g(n))

Limit of a Sequence

Limit of a Sequence - If for every real number epsilon > 0, there is a natural number n_0 > 0, such that for all n >= n_0 we have

Proof of 2.1
Take $\varepsilon = \frac{c}{2}$, then for all n >= n_0 we have

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \quad \Leftrightarrow \quad \left|\frac{f(n)}{g(n)} - c\right| < \frac{c}{2}$$

$$-\frac{c}{2} < \frac{f(n)}{g(n)} - c < \frac{c}{2} \quad \Leftrightarrow \quad \frac{c}{2} * g(n) < f(n) < \frac{3c}{2} * g(n)$$

Transitivity

- If a func. "f" is bounded by a func. "g" that is in turn bounded by "h", then "h" is a bound for "f"
- This goes for upper as well as lower bounds and can be written:

2.2 Transitivity

Proof of 2.2

- f = O(g): $\exists$c, n_0 such that $\forall$ n >= n_0 we have g(n) <= c * h(n)
- g = O(h): $\exists$c', n_0' such that $\forall$ n >= n_0' we have g(n) <= c' * h(n)
- Consider n >= max (n_0 , n_0')
- now f(n) <= g(n) <= c * c' * h(n)
- Thus,
  - \thereexists C, N such that \forall n >= N we have f(n) <= C * h(n)
- meaning that f = O(n)

Sum of Functions

If a bound applies to 2 functions, then it applies to their sum
2.4 Sum of Two Functions - Suppose that f and g are functions such that f = O(h) and g = O(h). Then f + g = O(h)

The statement can be generalized to
2.5 Sum of Functions -

Proof of 2.4

- f = O(g): $\exists$c, n_0 such that $\forall$ n >= n_0 we have g(n) <= c * h(n)
- g = O(h): $\exists$c', n_0' such that $\forall$ n >= n_0' we have g(n) <= c' * h(n)
- Consider n >= max (n_0 , n_0')
- now f(n) + g(n) <= c * h(n) + c' * h(n) = (c " c') * h(n)
- Thus,
  - \thereexists C, N such that \forall n...
- meaning that f + g = O(h)

Asymptotically Tight Bound for a Sum

Consider an algorithm that consists of 2 parts , of which one is considerably slower
→ We want to show that the total running time is comparable to the running time of the slower part

2.6 Asymptotically Tight Bound for a Sum - Suppose that f & g are funcs. s.t. g = O(f). Then f + g = Theta(f)

Proof of 2.6

- It is obvious that f + g = Omega(f) since
  - \forall n >= 0 we have f(n) + g(n) >= f(n)
- We need to show that f + g = O9f)

Asymptotically Tight Bound for a Sum

The asymptotic growth of a polynomial is given by its highest-order term

2.7 Asymptotically Tight Bound for a Sum - Let $f(n) = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d$ with a_d > 0 then f = O(n^d)

- We consider f(n) >= 0, so we must have a_d > 0.
- For i < d, we can have a_i < 0
- Consider running time that is positive, because time cannot be negative

Proof of 2.7

- An upper bound for each term in the sum is given by
- \forall n >= 1 we have $a_i n^i$ <= $|a_i|$ n^i <= $|a^i|$ n^d for i <= d
- This signifies that $a_i n^i$ = O(n^d)

Asymptotically Bounds for Logarithms

Remember that x = $\log_b$ n is the number such that b^x = n

- Logarithms are slowly growing functions:
  - in particular, the logarithm of any base always grows slower than any polynomial

2.8 Asymptotic bounds for Logarithms - For every b > 1, and every x > 0, we have $\log_b$ n = O(n^x)

- Translate logarithms between different basis using the identity

$$log_a\ n = \frac{log_b\ n}{log_b\ a}$$

→ Use any base!

- will not be using any base but "e"
- Rewrite the logarithm using the natural logarithm

$$log_e\ n = \frac{log_b\ n}{log_b\ e} \quad \Leftrightarrow \quad log_b\ n = log_b\ e * log_e\ n = log_b\ e * ln\ n$$

- Write as ratio of 2 functions

$$\frac{f(n)}{g(n)} = \frac{log_b\ n}{n^x} = \frac{log_b\ e * ln\ n}{n^x} = log_b\ e * \frac{ln\ n}{n^x}$$

- Use L'Hopital's rule to compute the limit as n tends to inf

$$\lim_{n \to \infty} \frac{ln\ n}{n^x} = \lim_{n \to \infty} \frac{(ln\ n)'}{(n^x)'}$$

Every exponential grows faster than every polynomial

2.9 Asymptotic Bounds for Exponentials - for every r > 1, and every d > 0, we have n^d = O(r^n)

- all exponentials do not have the same growth rate, but we note that we always have

- Function Growth rate slow → fast: logarithms → polynomials → exponentials
- Once more, write as the ratio of 2 functions

$$\frac{f(n)}{g(n)} = \frac{n^d}{r^n} = \frac{e^{\ln n^d}}{e^{\ln r^n}} = \frac{e^{d \ln n}}{e^{n \ln r}} = e^{d \ln n - n \ln r}$$

Consider the limit as n tends to inifnity

$$\lim_{n \to \infty} e^{d \ln n - n \ln r} = 0$$

According to the def. of limits, we now have $n^d = O(r^n)$

Lecture 4

Thurs. Sept. 5, 2019

Common Running Times

Many Problems have common properties

- # of candidates for a soln.
    - all elements
    - all pairs
    - all subsets of given size
    - all subsets
- Similar strategy
    - recursive calls
    - (nested) iterations
- Space of all possible solns. → cost for brute-force
- recognizing the computational complexity can help us understand the problem

Linear Time

An algorithm that runs in linear time O(n)

→ single pass

→ constant per element

We will see 2 ex.):

- Find the max of “n” numbers
- merge 2 sorted lists

Find the Maximum

- Initialize the max. value
- Go thru the list
    - Update the max. if a bigger value found

Pseudocode:

```
function findMax( a[1 … n] )
        max = a[i]
        for i = 2 to n
                if a[i] > max
                        max = a[i]
                end if
        end for
end function
```

Merging of Sorted Lists

- Given - 2 sorted lists
- Problem - Merge the 2 lists: the resulting list must be sorted
- **First Idea**
    - concatenate the lists
    - sort the results
- **Second Idea**
    - use the fact that list are sorted to speed up the algorithm

O(n logn) Time

O(nlogn) time is a very common running time:
→ Split a problem in 2 equally size problems
→ Solve recusively
→ Combine the solns. in linear time

- Most well known ex.): **Mergesort**

Mergesort - A Walk-through

Mergesort is an efficient divide & conquer algorithm

- Divide - cursive calls
- Conquer - merge

Recursive Calls

```
function Mergesort(a[1 .. n])
    if (n =1 )
        return a
    else
        a1 = mergeSort(a[1 .. n/2])
        a2 = mergeSort(a[n/2+1 .. n])
    end if
end function
```

- depth = k
- n = 2^k
- Recursive algorithm: depth: “log n”
- Merging in linear time: “n:
- Complexity O(n logn)

CSE 3500 Master Notes

Closest Points

```
min = 0
for i = 0 to n-1
    for j = 0 to n-1
        dist =
        if dist < min
            min = dist
    end if
end for
```

Finding Disjoint Subsets

- Given - n sets S1, S2, …, S2
- Problem - Are there 2 disjoint subsets (no elements in common)
- **Algorithm**
    - Check all pairs of subsets
    - For each pair, check all elements

```
function findDisjointSubsets()
    for each setS_i
        for each other set S_j
            for each element p of S_i
                determine if p is in S_j
            end for
            if no element in common
                return found
            end if
        end for
    end for
end function
```

Finding Independent Subsets of Size “k”

The number of subsets of size “k” is

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{n\,(n-1)\,(n-2)*...*(n-k-1)}{k\,(k-1)\,(k-2)} \leq \frac{n^k}{k!}$$

The number of pairs in a subset is

→ but k is a constant!

- Number of subsets: (n k) = $O(n^k)$
- Cost for a subset: (k 2) = $O(k^2)$ Both → $O(n^k)$

Binary Search

```
function BinarySearch (A[1 .. n], val, lo, hi)
    if hi < lo
        return not_found
    end if
    mid = (low + high) / 2
    if A[mid] > val
        return BinarySearch(A, val, low, mid-1)
    else if {A[mid < value}
        return BinarySearch(A, val, mid+1, hi)
    else
        return mid
    end if
end function
```

Lecture 5

Tues. Sept. 10, 2019

Priority Queues

Priority Queue - an abstract data type

- Queue - waiting in a line: FIFO
- stack - Pile of plates: FIFO
- Priority Queue - Emergency Room

Applications include

- Bandwidth management → manage limited resources
- Dijkstra's Algorithm → Find the shortest path in a graph
- Huffman coding → Prefix code for lossless compression

Operations:

● Addition of elements ● deletion of elements ● selection of element with smallest key	All of O(log n)

Unlike more basic data structures:

→ complex handling follows each operation

What can we hope for from a priority queue?

2.11 Sorting Using Priority Queues - A sequence of O(n) priority queue operations can be used to sort a set of "n" numbers.

- Sorting is O(n log n): each operation on a priority queue should not take more than O(log n)

Proof of 2.11

1. Set up a priority queue and insert each number with its value as key
2. Extracting the minimum values, one at a time, yields a sorted list
3. There are n elements, so O(n) operations are needed

If **insertion** and **removing the minimum** can be done in O(log n) time → we have a new sorting algorithm!

Implementing a Priority Queue

Try the usual ideas:

Unsorted array w/ pointer

add	- O(1)
find min	- O(1)
delete	- O(n)

Sorted Array

add	- O(n)
find min	- O(1)
delete	- O(n)

Sorted doubly linked list

add	- O(n)
find min	- O(1)
delete	- O(1)

→ at least one operation in any option is O(n)

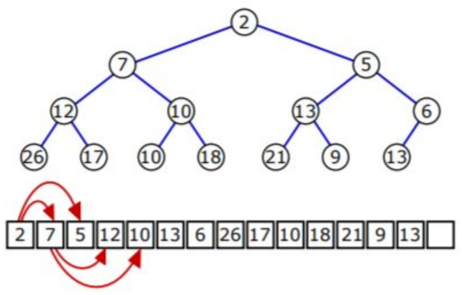
The Heap

Use a new data structure to implement the priority queue → a heap

- a heap can be represented by a **balanced binary tree**

Heap Order - For every element "v", at a node"i", the element "w" at "i's" parent node satisfies key(w) <= key(v)

Binary Trees as Heaps



- Concept - a binary tree
- Implementation - Array → Lay out nodes in breadth-first order

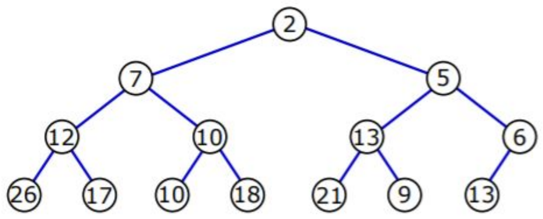
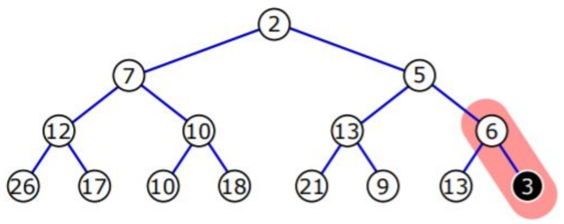
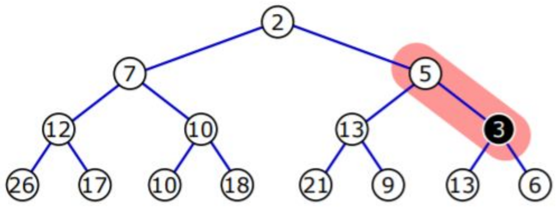
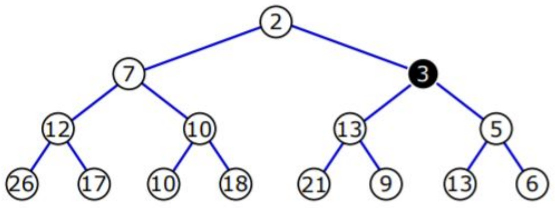
The heap is stored as an array H indexed from 1 to n.
Node w/ index "i" has parent & children according to

- Parent: $j = floor[\frac{i}{2}]$
- Left child: $j = 2i$

- Right child: $j = 2i + 1$

The heap must at all times respect
→ The heap property
→ The balanced binary tree property
→ Will need to restore heap property when violated
Add New Element
Adding an element to an array is O(1), must also conserve heap

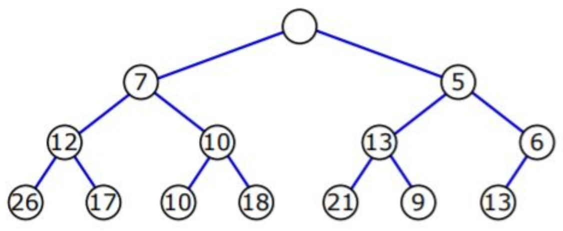
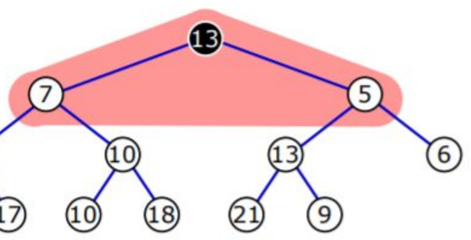
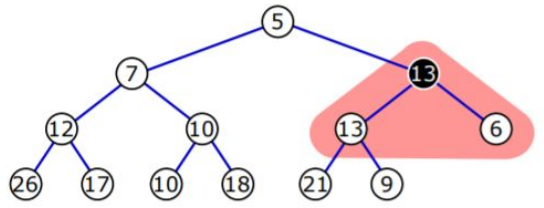
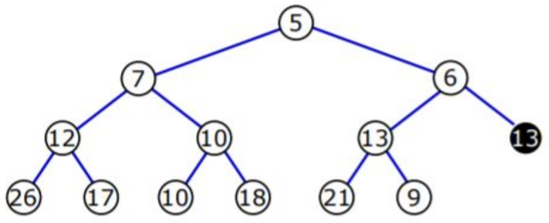
- add to the end → respect balanced tree
- restore the heap property → move small element that are too small upwards

	• Heap property is **verified**
	• insert new element at the end • heap property is **violated** • swap with parent
	• heap property is **violated** • swap with parent
	• heap property is **restored**

Identifying the minimum element is O(1), but we must also restore the heap:

Algorithm:

- pop the root element → not a tree
- move the last element to the root → violates heap property
- restore the heap property → move to big element downwards

	• min. element has been popped • heap is not a binary tree
	• Place last element at the end root • heap property is **violated** • swap with smallest child
	• heap property is **violated** • swap with smallest child
	• heap property is **restored**

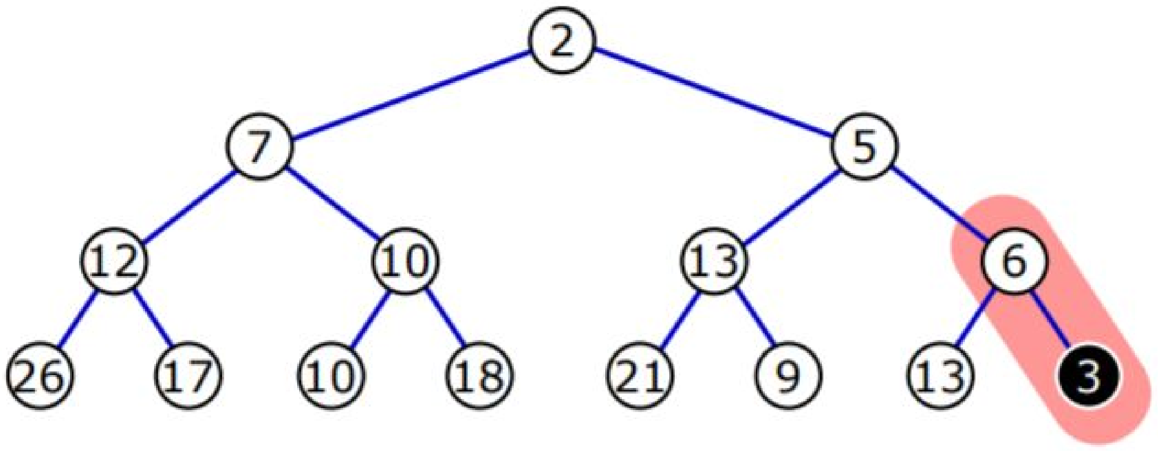
Heapify Up

```
function HeapifyUp(H, i)
    if i > 1
        j = parent(i)
        if key(H[i]) < key(H[j])
            swap(H[i], H[j])
            HeapifyUp(H, j)
        end if
    end if
end function
```

Almost a Heap - Too Small - H is almost a heap with the key of element “i” too small if there is a value (alpha) >= key (i) such that setting key(i) = (alpha) would make H a heap

2.12 HeapifyUp - The function “HeapifyUp(H, i)” restores the heap property in O(log i) time, assuming that H is almost a heap with the key of “i” too small. Using “HeapifyUp”, we can insert a new element in a heap with “n” elements in O(log n) time.

Understanding 2.12



- Swapping propagates the problem → does not multiply it
- Setting both keys to 6 would give a heap → proves that there is only limited damage to the heap

Proof of 2.12

We prove the statement by induction on “i”:

- If “i = 1”, H is a heap.
- Suppose i > 1,
    - v = H[i], j = parent(i), w = H[j], alpha = key(w)
    - Swap elements “v”, and “w” in O(1) time
    - If we set key(v) = alpha, then H would be a heap
    - Therefore, H is either a heap or almost a heap with the key of H[j] too small
- **By the induction hypothesis**, calling the function recursively will produce a haep
- The process follows the tree up from element “i”, so it is O(i)
- To add a new element, we insert it at the last position and use “HeapifyUp” to restore the heap property

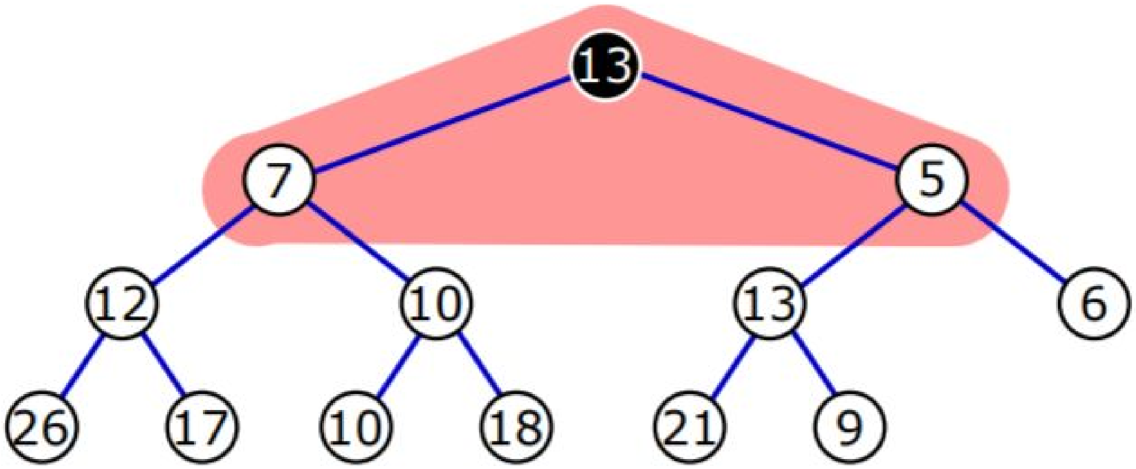
HeapifyDown

```
function HeapifyDown(H, i)
    n = length(H)
    if 2i > n
        done!
        else if 2i < n
                j = child with smallest key
        else
                j = n
        end if
        if key(H[j]) < key(H[i])
                swap(H[i], H[j])
                HeapifyDown(H, j)
        end if
end function
```

Almost a Heap - Too Big - H is almost a heap with the key of element "i" too big if there is a value (beta) <= key(i) such that setting key(i) = (beta) would make H a heap

2.12 HeapifyDown - The function HeapifyDown(H, i) restores the heap property in O(log n) time, assuming that H is almost a heap with the key of "i' too big. Using HeapifyUp or HeapifyDown, we can remove an element in a heap with "n" elements in O(log n) time.

Understanding 2.13



- Must choose smallest child → or we would create a new violation
- The swapping propagates the problem → it does not multiply it
- Setting both keys to 5 would give a heap → proves that there is only limited damage to the heap

Proof of 2.13

We prove the statement by reverse induction of "i":

- If 2i > n, H is a heap.
- Suppose i > 1,
    - v = H[i], j is the smallest child of "i", w = H[j], (Beta) = key(w)

- Swap elements “v” & “w” in O(1) time
- If we set key(v) = (beta), then H would be a heap
- Therefore, H is either a heap or almost a heap with the key of H[j] too big

- Now j >= 2i, so by the induction hypothesis, calling the function recursively will produce a heap
- The process follows the tree down from element “i”, so it is O(n)
- To remove an element, we replaced it with the element from the last position and use *HeapifyUp* and *HeapifyDown* to restore the heap property.

Lecture 6

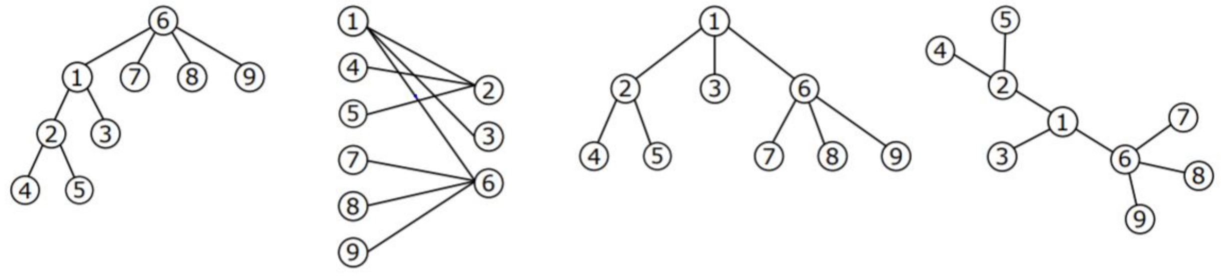
Thurs. Sept. 12, 2019

Graphs

Graphs -

- **Nodes**
    - Physical objects: people, cities, computers
    - Abstract ideas: internship
- **Edges**
    - Physical objects: railways, roads
    - Abstract ideas: hyperlinks, acquaintances

Ex. of Graphs - What do these graphs have in common?



They are identical → same **nodes** and **edges**

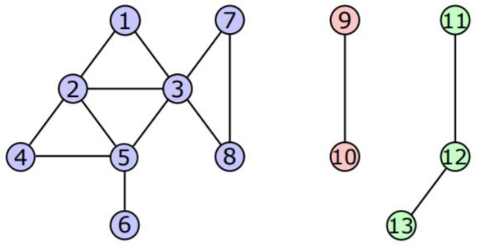
- They way graphs are drawn indicates what they represent
    - dif. hierarchical structures
    - connections between 2 sets
    - simple connections between nodes

Types of Graphs

- Directed Graph - The edges have directions

- Weighted Graph - A eight or a cost applies to each edge
- Connected Graph - There is a path between any 2 nodes
- Bipartite graph - The nodes belong to 2 sets, no edges within the sets
- Tree - A graph with no cycles
- Planar Graph - A graph that can be drawn in the plane without edges crossing each other

Connectivity & Graph Traversal



The following graph consists of 3 connected components

It is easy to see whether a graph is connected
Problem - a computer cannot "see" → we need vocabulary to analyze the graph

Paths

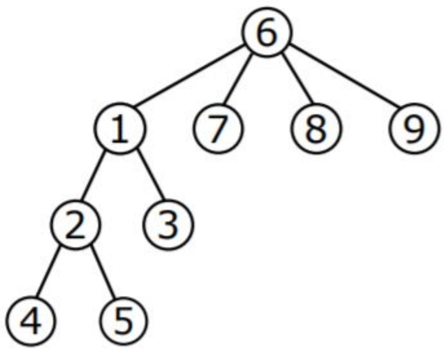
Path - A path is an undirected graph G is a sequences P of nodes { v_1, v_2, … , n_k, } such that all consecutive pairs in the sequence are joined by edges in G. P is called a path from v_1 to v_k
Simple Path - A path is simple if all its nodes are distinct
Cycle - A cycle is path from v_1 to v_k, where the first "k - 1" nodes are distinct and v_1 = v_k

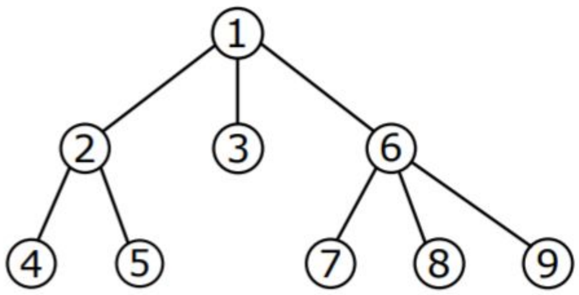
Connectivity

Connected Graph - An undirected graph is connected if, for every pair of nodes in the graph, there is a path between them,
Distance between Nodes - The distance between 2 nodes is the min. # of edges in a path between them.

- In a **weighted graph**, we can redefine the distance between 2 nodes as the min. sum of the weights of the edges in a path between them
  → find the shortest path

Trees





Tree - An undirected graph is a tree if it is connected and does not contain a cycle
- if we remove any edge from a tree, it becomes disconnected

3.1 Number of nodes in a tree - A n-node tree has exactly n-1 edges
- an even more power statement says that

3.2 Number of nodes in a tree - Let G be an undirected graph on “n” nodes. Any two of the following statements imply the third
- G is connected
- G does not contain a cycle
- G has “n - 1” edges

Now we can recognize a tree using 2 of these three Criteria

Hierarchical Structures using Trees

Hierarchy - One node is distinguished from the others → the **root “r”**
Ancestor and Parent - An **ancestor** of the node “v” is a node that is on the path “r” to “v”. The node that directly precedes “v” is called the **parent**
Descendant and Child - The node “v” is a **descendant** of a node “w” if “w” is an ancestor of “v”. The node “v” is called the **child** of “w” if “w” is the parent of “v”.
Leaf - A node is a leaf if it has no children

Connectivity

A fundamental question in a graph is whether it is connected or not and in particular if 2 given nodes belong to the same component:
s-t-connectivity - In a graph G w/ 2 particular nodes “s” and “t”, we have s-t-connectivity if there is a path in G from “s” to “t”.
Application: Finding a way out of a maze
- 2 natural approaches to this problem:
  - Breadth-First Search
  - Depth-First Search

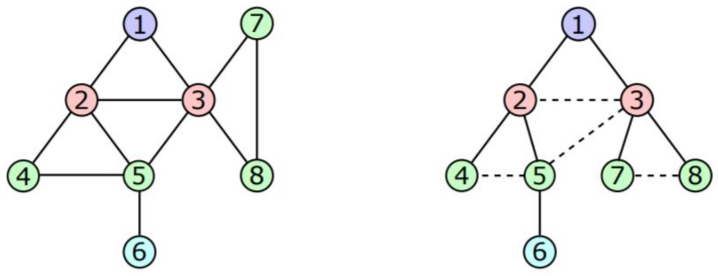
Breadth-First Search

- Start at s
- explore the neighbors
- explore their neighbors
- continue outwards
- stop when t found or when all nodes are visited

defining **layers** created by the algorithm
- L0 - Node S
- L1 - Set of neighbors of s
- L2 - Set of neighbors of L1
- Li+1 - Set of nodes that do not belong to an earlier layers and that have an edge to a node in Li

Breadth-First Search Tree

3.3 Distance and Connectivity - For each j >= 1 layer, Lj produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if "t" is in one of the layers



- The graph to the right represents the layers of BFS on the graph to the left

BFS on G produces a tree T:

- rooted at s
- contains all nodes connected to S
- contains a subset of the edges in G

NOTE: the edges from G missing in T all seem to connect nodes in the same or adjacent layers.

3.4 Breadth-First Search Tree - Let "T" be a breadth-first search tree, let "x" and "y" be nodes in T belonging to layers Li and Lj respectively, and let (x, y) be an edge in G. Then "i" and "j" differ by at most 1

Proof by contradiction:

- Suppose that i < j - 1
- Consider the point in the BFS algorithm where neighbors of "x" where visited
- All nodes discovered from "x" must belong to layer Li+1, or earlier
- Since "y" is a neighbor, it must be discovered now.
- Thus "y" belongs to Li+1, or earlier: contradiction

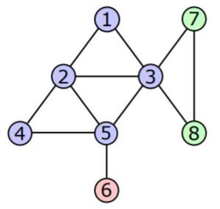
Connected Component

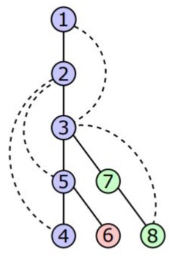
- BFS produces the connected component to which "s" belongs → not the only way:

```
function createConnecetedComponent(s)
    set R = { s }
    while (there is an edge from s ending in a node v outside R)
        add v to R
    end while
end function
```

3.5 Connected Component - The set R produced by this algorithm is precisely the connected component of G containing S

Depth-First Search





Another approach:

- Follow the first edge out from s
- Continue until we reach a dead end: no new neighbors
- backtrack to a node with unexplored neighbors
- Resume the search from there

Recursive Depth-First Search

```
function DFS(u)
    mark u as explored
    add u to R
    for each edge (u, V) from u
        if v is not marked explored
            DFS(v)
        end if
    end for
end function
```

Depth-First Search Tree

3.6 - For a given call to DFS(y), all nodes that are marked explored between the invocation and the end of this recursive call are descendants of “u” in “T”.
3.7 - Let “t” be a depth-first search tree, let “x” and “y” be nodes in “T”, and let (x, y) be an edge of G that is not an edge of “T”. then one of “x” or “y” is an ancestor of the other.

Lecture 7

Tues. Sept. 17, 2019

Representing Graphs

- We think of graphs as nodes linked to each other by edges

→ implementation: objects with references to other objects

- A graph G = (V, E) has:
    - n = |V| edges
    - m = |E| edges
- What is the time complexity for traversing the graph?
- To answer, we will introduce other ways to represent the graph:
    - adjacency matrix
    - adjacency list

CSE 3500 Master Notes

Adjacency Matrix

Consider a graph G = (V, E) with “n” nodes V = {1, 2, …, n}
Adjacency Matrix - The adjacency matrix is a “n x n” matrix A such that

$$A[u,\ v]\ =\ \{\ 1\ \textit{if}\ \ (u,v)\ \textit{is an edge in}\ E \qquad ,\ 0\ \textit{if not}$$

- If the graph is undirected, the adjacency matrix is symmetric

Features:

- Intuitive representation
- Verifying if a given edge is in G takes O(1) time

Drawbacks:

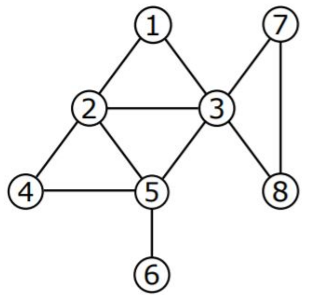
- The representation takes Theta(n^2) space
    - Most graphs have fewer than n^2 edges
      → most compact representation needed
- Visiting all neighbors of a node takes Theta(n) time
    - Important step in many graphs algorithms
      → most efficient methods needed

We need a representation for sparse graphs
→ adjacency list

Adjacency List

Adjacency List - The adjacency list has “n” records. The entry for each node “v” contains a list of all nodes that “v” have edges to

- An edge (u, v) yields 2 entries:
    - u is on the list for entry v
    - v is on the list for entry u



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

1	2	3			
2	1	3	4	5	
3	1	2	5	7	8
4	2	5			
5	2	3	4	6	
6	5				
7	3	8			
8	3	7			

Adjacency List - Space Requirement

- The adjacency list requires an array of length n = O(n)
- Each edge in the graph appears twice inthe adjacency list
  → The total length of the lists is 2m = O(m)

3.10 Space Requirements - The adjacency matrix representation of a graph requires O(n^2) space, whereas the adjacency list representation requires O(m + n) space.

Sum of Degrees in a Graph

- We define the degree of a node n_v to be the number of incident edges it has

3.9 Sum of Degrees - The sum of the degrees of the nodes in a graph is given by

$$\sum_{v \in V} n_v = 2m$$

Proof: Each edge (u, v) contributes exactly twice:

- Once in the quantity n_u
- Once in the quantity n_v
- Sum is the total of the contributions of all edges, so it is equal to 2m

Graph Traversal

- If the adjacency list requires O(m + n) space → can we visit all nodes in O(m + n) time?
- The adjacency list provides a natural way of traversing graphs
- If an algorithm is currently looking at node “u”, it can:
    - Read the list of neighbors in O(n_u) time
      → constant time per neighbor
    - Move to an encountered neighbor “v” in O(1) time
      → visit all its neighbors

Queues and Stacks

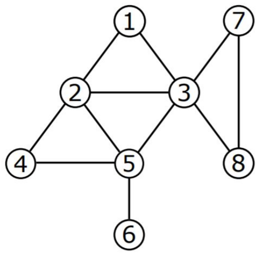
- In which order do we process nodes that we wish to explore?
- There are 2 standard Abstract Data Structures (ADT) that handle this:
- **Queue** → Extract elements in First-In, First-Out (FIFO) order
- **Stack** → Extract elements in First-In, Last-Out (FILO) order
- We will see that our choice will give either a BFS search or a DFS search

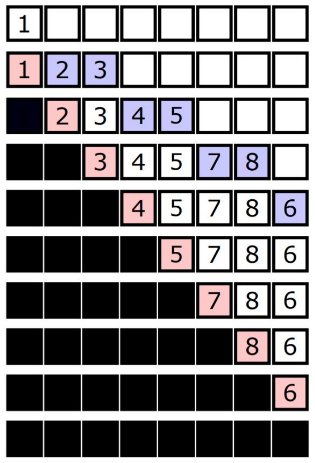
Breadth-First Search using a Queue

```
function DFS(s)
    Q = Queue(s)
    while Q.isempty() == false
        u = Q.dequeue()
        for all edges (u,v) incident to u
            if Discovered[v] == false
                Discovered[v] == true
                Q.enqueue(v)
                end if
            end for
        end while
```

3.11 BFS in O(m + n) time - The proposed implementation of the BFS algorithm runs in O(m + n) time, if the graph is given by the adjacency list

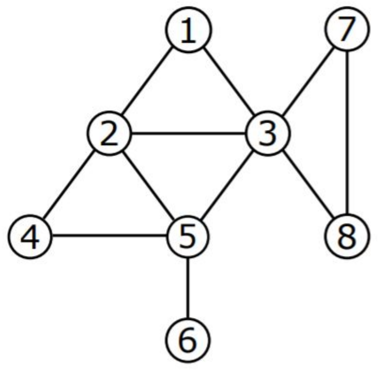
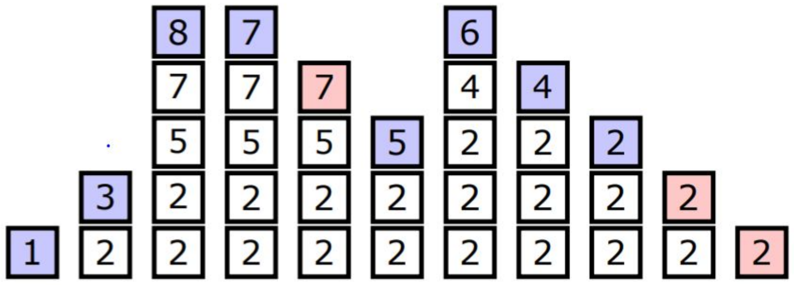
- Once a node v is taken out of the queue and is processed the time spent in the for-loop is O(n_v)
- The total over all nodes is $O(\sum_{v \in V} n_v)$
- We know that (3.9) that $\sum_{v \in V} n_v = 2m$, so that total time over all nodes is O(m)
- In addition to this, we also need O(n) to handle the array “Discovered”
- The total time spent is now O(m + n)





Depth-First Search Using a Stack

```
function DFS(s)
    S = stack(s)
    while S.isEmpty() == false
        u = S.pop()
        if Explored[u] == false
            Explored[u] = true
            for all edges (u,v) incident to u
                if Explored[v] == false
                    S.push(v)
                end if
            end for
        end if
    end while
```

3.12 DFS using a Stack - The proposed algorithm implements a depth-first search

3.13 DFS in O(m + n) time - The proposed implementation of the DFS algorithm runs in O(m + n) time, if the graphs is given by the adjacency list

Trees and Layers

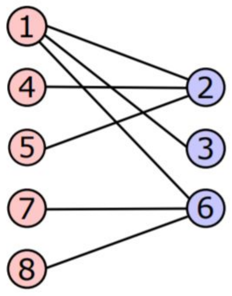
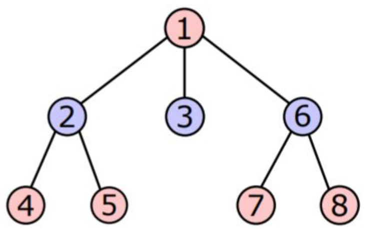
- The previous version of the BFS algorithm produced
    - The BFS tree
    - The layers
- The previous version of the DFS algorithm produced
    - The DFS tree
- The new algorithms do not produce any of those → How can we adapt them?

Lecture 8 - Testing Bipartiteness, Directed Graphs

Thurs. Sept. 19, 2019

Bipartite Graph

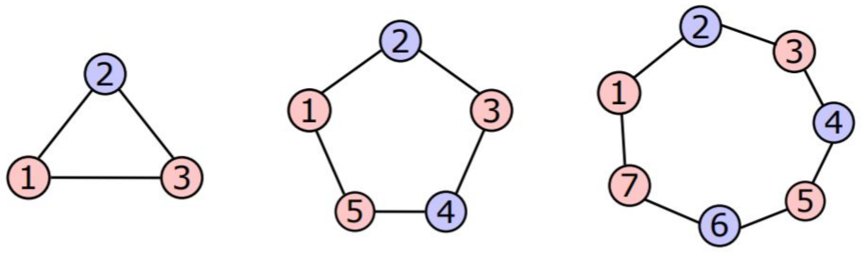
Bipartite Graph - A graph G = (V, E) is bipartite if its set of nodes V can be partitioned into two sets X and Y in such a way that every edge in E has one end in X and one end in Y.

- We see that all trees must be bipartite → What more can be said about bipartiteness?

Odd Cycles

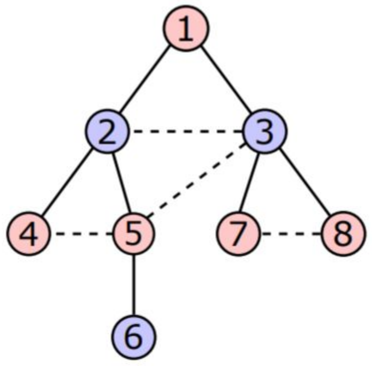
3.14 Odd Cycles - If a graph G is bipartite, it cannot contain an odd cycle



Proof:

- Consider a cycle C of odd length → nodes numbered 1, 2, 3, …, 2k , 2k +1
- Color odd nodes red and even nodes blue
- The first and the last nodes are both red → C cannot be partitioned into two sets

Testing Bipartiteness



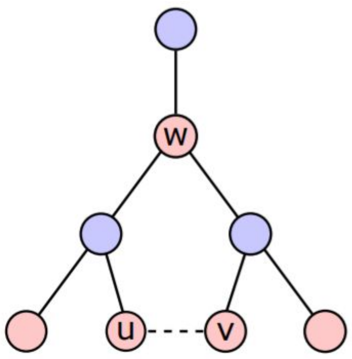
- Pick any node and color it red
- Perform a BFS and color the layers: blue, red, …
- Consider the edges in G but not in T
  → Between adjacent layers: ok
  → Within the same layer: no bipartiteness!
- Connection between same layer edges & odd cycles?

Bipartiteness and Odd Cycles

3.15 Bipartiteness and Odd Cycles - Let G be a connected graph and let L_1, L_2 … be the layers produced by the BFS on G starting at node s. Then exactly one of the two following must hold

- There is no node in G joining two nodes of the same layer. In this case, G is a bipartite graph
- There is an edge in G joining two nodes of the same layer. In this case, G contains an odd cycle and cannot be bipartite

Proof of (3.15)



- In the first case, all edges are between adjacent layers
  → All edges go between nodes of different color
  → The graph is bipartite
- Consider an edge between "u" & "v" in layer "i"
- Consider their first common ancestor "w" in layer "j"
- "u", "v", and "w" belong to a cycle
  → Cycle length 2(i - j) + 1 → odd

- In the second case, there is an edge between nodes "u" and "v" in the same layer
  → The graph contains an odd cycle
  → The graph cannot be bipartite

Directed Graphs

- Consider a directed graph G = (V, E) with "n" nodes, V = {1, 2, …., n}

Adjacency List - The adjacency list has n records. The entry for each node "v" contains **two lists** with …

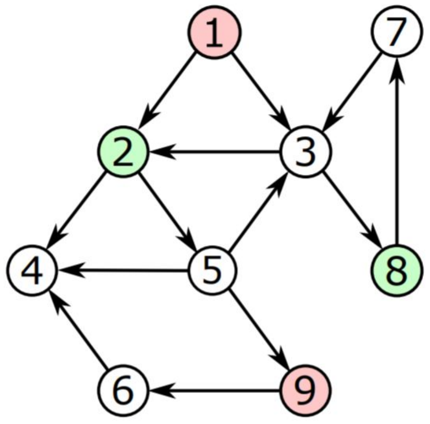
- all nodes that "v" have edges to
- all nodes that have edges to "v"

- An algorithm looking at a given node u can read the edges
  - that are directly reachable from "u"
  - from which "u" is directly reachable

Strong Connectivity

Mutually Reachable - 2 node "u" and "v" in a directed graph are mutually reachable if there is a path from "u" to "v" and from "v" to "u"
Strong Connectivity - A directed graph is strongly connected if, for every pair of nodes "u" and "v", there is a path from "u" to "v" and a path from "v" to "u"

- A directed graph is **strongly connected** if every pair of nodes is **mutually reachable**



Example: Mutually Reachable

- Nodes 2 & 8 are mutually reachable
- Nodes 1 & 9 are not mutually reachable
  → there is a path from 1 to 9 but not the other way around

Mutually Reachable - Transitivity

3.16 Transitivity - If "u" and "v" are mutually reachable and "v" and "w" are mutually reachable, then "u" and "w" are mutually reachable
Proof:

- "u" and "v" are mutually reachable → there is a path from "u" and "v"
- "v" and "w" are mutually reachable → there is a path from "v" and "w"
- Now, there is a path from "u" to "w"
- With the same reasoning, there is a path from "w" to "u" → "u" and "w" are mutually reachable

Determine Strong Connectivity

Reversed Graph - Define the reversed graph G^{rev} of a directed graph G to be the graph by reversing the direction of every graph

- Test if a directed graph is strongly connected:
  - Pick any node "s"
  - Run a BFS on G starting at "s"
  - Run a BFS on G^{rev} starting at "s"
  - If any of the 2 searches fails to reach all nodes → G is not strongly connected

BFS: The algorithm runs in O(m + n) time

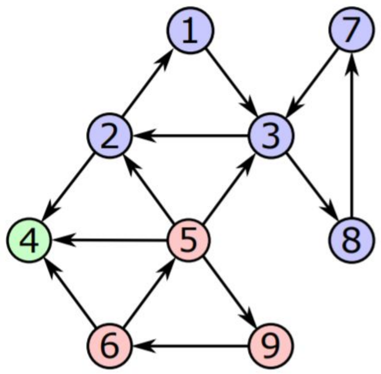
Strong Components

Strong Component - The strong component containing a node "s" in a directed graph is the set of all nodes "v" such that "s" and 'v" are mutually reachable

- Find the strong component containing s:
  - Run a BFS on G starting at "s"
  - Run a BFS on G^{rev} starting at "s"
  - Take the set S of all nodes found in both searches
    → S is the strong component containing "s"

BFS: The algorithm runs in O(m + n) time

3.17 Strong Components for Two Nodes - For any two nodes "s" and "t" in a directed graph , their strong components are either identical or disjoint



- 3 strong components:
  - 1, 2, 3, 7, 8
  - 4
  - 5, 6, 9

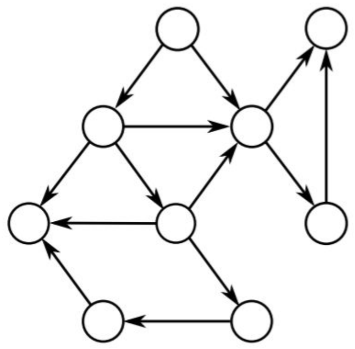
Proof of (3.17)

- Consider two mutually reachable nodes "s" and 't"
  - for any node "v", if "s" and "v" are mutually reachable → "v" and "t" are also mutually reachable by (3.16)
  - for any node "v", if "t" and "v" are mutually reachable → "v" and "s" are also mutually reachable by (3.16)

→ The strong components containing "s" and "t" are identical

- Consider two not mutually reachable nodes "s" and "t"
  - if any node "v" belonged to the strong components of "s" and "t"
    → "s" and "v" would be mutually reachable
    → "v" and "t" would be mutually reachable
    → "s" and "t" would be mutually reachable by (3.16)

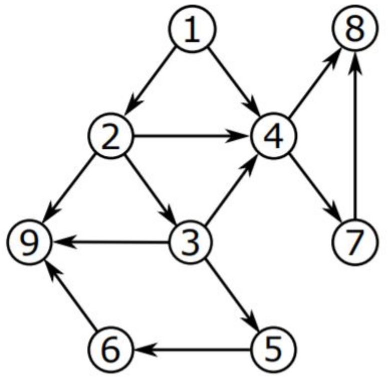
→ The strong components containing "s" and "t" are disjoint



Directed Acyclic Graphs

DAG - A directed with no cycles is called a Directed Acyclic Graph - DAG

- A DAG can represent dependencies or precedence relations → ex.): prerequisites for course



Topological Ordering

Topological Ordering - For a directed graph, a topological ordering is an ordering {v1, v2, …, vn} such that for every edge (v_i, v_j) we have i < j.

- All edges point “forward”
- Note always Unique
- The topological ordering respects all dependencies → ex.): safe order to perform tasks

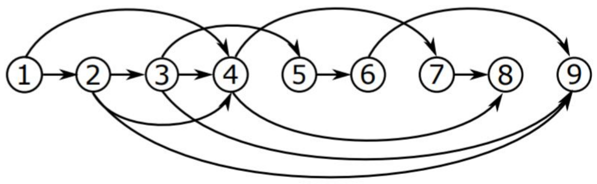
Topological Ordering and DAGs

3.18 Topological Ordering Implies a DAG - If G has a topological ordering, then G is a DAG

Visually:

- Two graphs w/ toplogical ordering are the same

→ No cycles since all edges go from left to right



Proof of (3.18)

Prove by contradiction:

- Suppose that G has a topological ordering {v1, v2, …, vn}
- Let v_i be the lowest-indexed node in C
- et v_j be the node just before $v_i$in C
  → Then there is an edge (v_j, v_i)
  → we have i < j
- Contradicts the topological ordering

The Inverse Statement

- state the inverse
- design an algorithm that compute the topological ordering

3.19 Incoming Edges - In every DAG G, there is na node with no incoming edges

Algorithm - To construct the topological ordering, start with a node v with no incoming edges and proceed recursively on G - {v}.

Proof of (3.19)

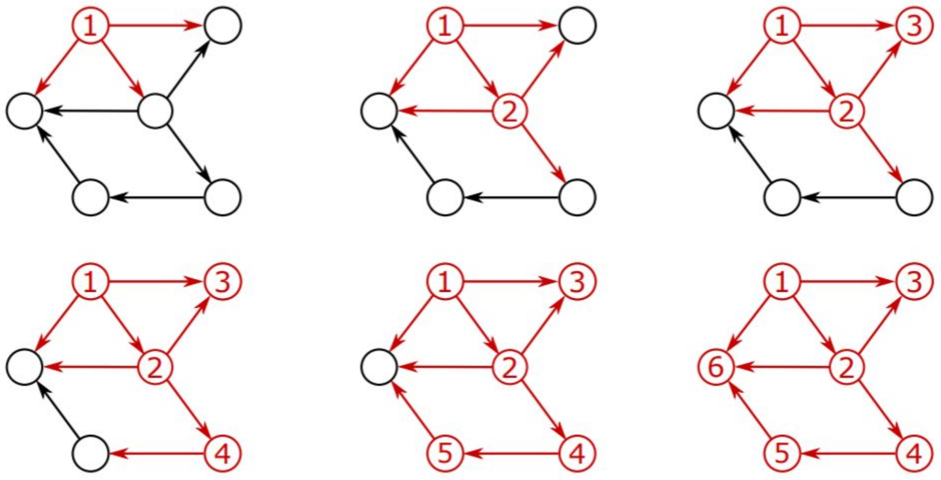
- Suppose that G is a graph where every node has at least one incoming edge

- if we can show that we can find a cycle, this prove the claim
- Pick any node "v" and follow the edges backward → This is possible since all nodes have at least one incoming edge
- Take "n + 1" steps → Now, at least one node has been visited twice
- Then, G has a cycle

DAGs and Topological Ordering

3.20 DAG Implies a Topological Ordering - If G is a DAG, then G has topological ordering

- Use (3.19) to identify a node with no incoming edges:



- Remove the node rom the graph and proceed recursively

Proof of (3.20)

We prove the statement by induction on "n":

- A DAG with 1 or 2 nodes has a topological ordering
- Suppose that the statement is true for a graph of size"n"
- Given a DAG on "n + 1" nodes, there is a node with no incoming edges by (3.19)
- The graph G - {v} is a DAG, since removing a node cannot create a cycle
- By the induction hypothesis, G - {v} has a topological ordering.
- Place "v" before all nodes in this topological ordering
- We have now created a topological with "n_ 1:

Compute a Topological Ordering

```
function computeTO(G)
      find a v with no incoming edges
      order = v
      append computeTO(G - v) to order
      return order
```

- Finding a node with no incoming edges takes O(n) time

- n function calls → Time complexity O(n^2)
- Find nodes with no incoming edges can done in a better way!

Active Nodes

- A node is active if it has not yet been deelted by the algorithm

Keep track of:

- For each node, the number of incoming edges from other active nodes
- The set of nodes with no incoming edges from other active nodes

CTO - Analysis

- We can pick any node "v" with no incoming edges

→ Total cost: O(n)

- We can update the number of incoming edges from other active nodes by subtracting one from each one of the nodes that "v" had edges to

→ Total cost: O(m)

- The time complexity of the algorithm is now O(m + n)

→ due to a clever choice of

  - book keeping
  - update procedure

Lecture 9 - Interval Scheduling & Partitioning

Tues. Sept. 24, 2019

Greedy Algorithms

- **Greedy Algorithm** strategy that:
  - optimizes some measure → many choices possible
  - makes locally optimal choices → optimal w. r. t. (with respect to) some
  - builds an incremental soln. → step-by-step: one choice at a time
- For many probs., the greedy approach will not work → when it does work, it is hard to beat!

Evaluate the Choice of a Greedy Algorithm

- greedy algorithm can be formulated for almost any problem
- out main challenge will be to
  - Find the cases in which they work well
  - analyze the solution

    → prove optimality

    → evaluate the computational complexity

- We will consider the **Interval Scheduling Problem** and the related **interval Partitioning System**

Interval Scheduling - Problem Formulation

- We dispose of one resource and have multiple requests:

request - use the resource during a given time
rule - no two overlapping requests can be accepted
goal - maximize the number of accepted requests

- We have "n" requests labeled 1,... n
    - start and finish times (s_i, f_i)
- Two requests are **compatible** if they do not overlap
- A set of requests is compatible if all pairs of requests are **compatible**

Interval Scheduling - A Greedy Algorithm

- The request will be chosen one at a time → which strategy to adopt when choosing the next interval?
- A greedy algorithm could be described as follows:
    - sort the request with respect to some criterion
    - pick the first available request
        - Discard all requests incompatible with it
    - repeat until the set of requests is empty

Different Choices

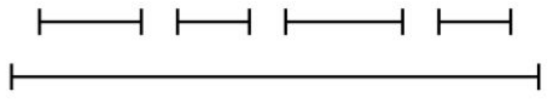
Ideas: Choose the request…

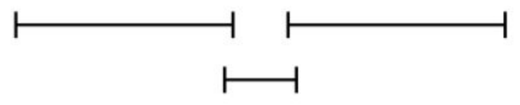
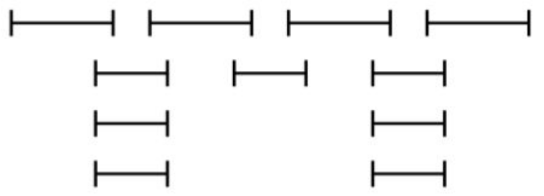
- with the earliest start time → the resource will start being used as soon as possible
- that requires the shortest time → the resource will be block for as short a time as possible
- that has the fewest conflicts → the resource will be block for as short a time as possible

***We will use counter examples to show that these choices do not produce optimal solutions

Counterexamples

The strategies presented will not yield an optimal solution:

Minimal start time → solution: 1 → optimum: 4	

Minimal start time → solution: 1 → optimum: 2	
Minimal start time → solution: 3 → optimum: 4	

Earliest Finish Time

We have already seen a greedy algorithm that gives an optimal soln:

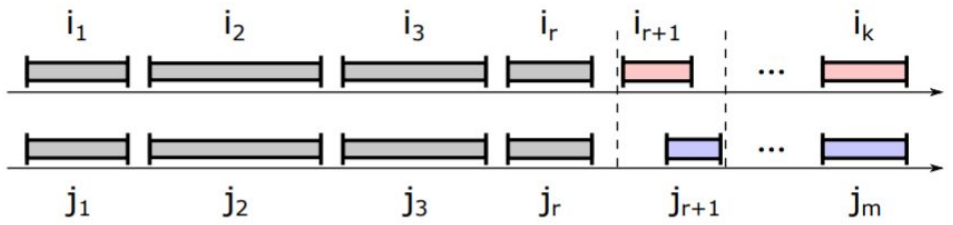
- Sort the request w. r. t. earliest finish time
- Pick the first available request
    - Discard all request incompatible with it
- Repeat until the set of request is empty

How do we prove that this gives an optimal soln.?
→ Start with notin that it gives a solution
4.13 Compatible Set - The set of request chosen by the algorithm is a comptable set

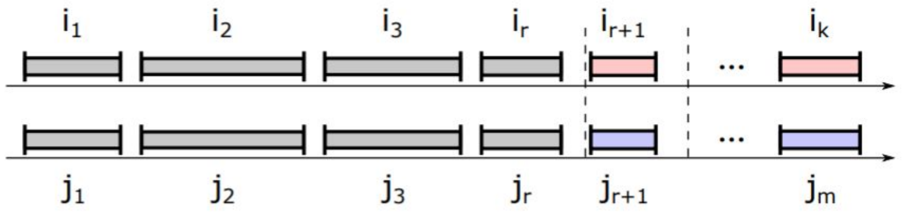
Optimal Solution

4.3 Optimality - The greedy algorithm return an optimal set
Proof: by contradiction

- Assume the greedy soln. is not optimal
    - Let i_1, i_2, … , i_k be request selected by greedy
    - Let j_1, j_2, … , j_k be request in an optimal soln. with
    - $i_1 = j_1$, $i_2 = j_2$, … , $i_r = j_r$ for largest possible "r"



- Request i_{r+1} exists and finishes earlier than j_{r+1} → we can replace j_{r+1} with i_{r+1}



- The soln. is still feasible and optimal → we have a contradiction on the largest value “r”

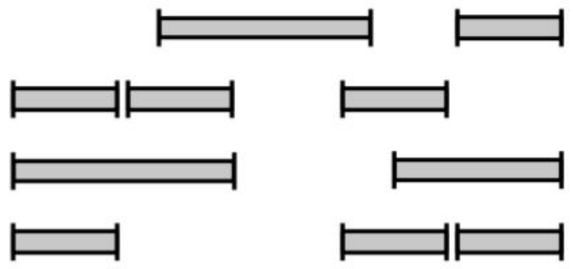
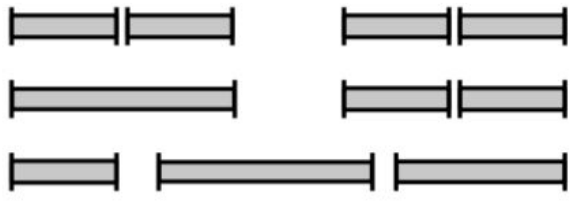
Running Time

- Preparation & data structures:
    - sort the request w. r. t. finish times: O(n log n)
    - Construct an array S[1..n] w/ start times s(i): O(n)
- Algorithm:
    - Pick the 1st request: call its finish time “f”
    - Iterate thru S to find the 1st start time s(i) >= f → pick request “i” & call its finish time “f”
    - cont. the search from “i”
- Only one pass is needed

→ total running time: O(n log n)

Interval Partitioning

Now - Same requests, but more than one resource
Goal - schedule all the request with as few resources as possible

one instance of the problem	
same instance: better solution	

Interval Partitioning - Problem Formulation

- We dispose of multiple resources and have multiple request:

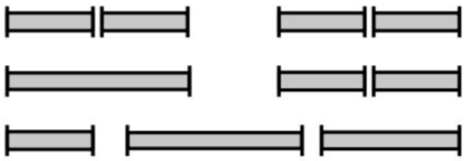
CSE 3500 Master Notes

Request - use a resource during a given time
Rule - no 2 overlapping requests can be accepted for the same resource

- We have "n" requests label 1,... n
  start and finish times (s_i, f_i)
- two requests are compatible if they do not overlap
  $f_i <= s_j$ or $f_j <= s_i$

Optimal Solution

- Can this instance be solved using only 2 resources?



- answer is **no** → can identify 3 mutually incompatible requests

Depth of a Set of Intervals - the max. # of intervals that pas over any single point on the time-line.
Minimal Number of Resources - In any instance of Interval Partitioning, the number of resources needed is at least the depth "d" of the set of intervals.

- If we can design a greedy algorithm that schedules all requests using at most d it must be optimal.

Greedy Approach

Algorithm

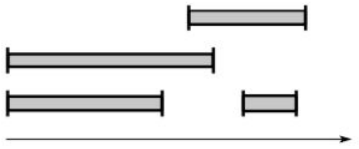
- Consider a number of lectures in some order → which order?
- Assign a lecture to an available classroom if there is one → which one?
- Allocate a new classroom if none available

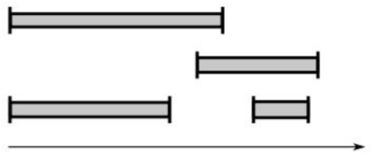
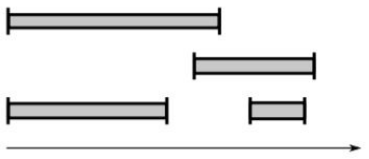
Ideas: Choose the lecture…

- ... with the earliest start time
- ... with the earliest finish time
- ... that requires the shortest time
- ... that has the fewest conflicts

Counterexamples

Some strategies presented will not yield an optimal solution:

Earliest finish time → solution: 3 → optimum: 2	

Earliest finish time → solution: 1 → optimum: 2	
Earliest finish time → solution: 3 → optimum: 4	

Earliest Start Time

We propose a greedy algorithm that gives an optimal solution:

- Sort the requests w.r.t. earliest start time
    - If lecture is compatible w/ some classroom
        - schedule lecture in any such classroom
    - else
        - allocate a new classroom & schedule lecture there

Note: The algorithm never schedules 2 incompatible lectures in the same classroom

Implementation

● Choose according to the following:			
● Lecture	Earliest start time	→	sort lectures
● Classroom	Earliest finish time	→	key in priority queue

Algorithm Design:

- Sort lectures ascending with respect to start time
- Store classrooms in a priority queue
- Determine compatibility between lecture i and classrooms

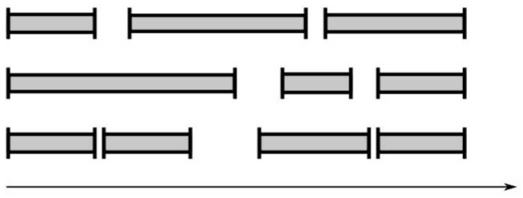
→ compare s_i to key for min classroom in queue

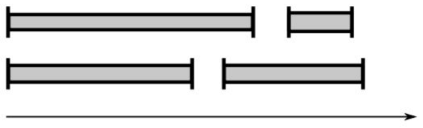
- To add lecture i to classroom k

****** O(log n) for depth of classrooms that can be modeled as balanced binary tree of depth 2^k where the levels produce a "log n" running time to dive through the levels***

→ increase key of classroom k to i

Examples





Running Time

- Algorithm will perform the following operations:
  - Sorting at O(n log n)
  - O(n) priority queue operations at O(log n) each
- Only one pass is needed

→ Total running time: O(n log n)

Optimal Solution

(4.6) Optimality - The proposed algorithm is optimal

Proof:

- Let “d” be the # of classrooms allocated by the algorithm
- Classroom “d” is allocated because we have a lecture “i” that is incompatible w/ all “d - 1” classrooms
- All these “d” lectures each end after “s_i”
- Since the lectures are sorted by start time, all incompatibilities are caused by lectures starting no later than “s_i”
- Tus, we have ‘d” lectures overlapping at time “s_i”
- By 4.4, all schedules use at least “d” classrooms

Lecture 10 - Greedy Algorithms Minimize Lateness

Thurs. Sept. 26, 2019

CSE 3500 Master Notes

Interval Scheduling - Problem Formulation

- We dispose of one resource and have multiple requests:
    - request - required processing time before a deadline
    - rule - no two overlapping requests can be accepted
    - goal - minimize the maximum lateness
- We have "n" requests labeled 1,... n

  duration and deadlines (t_i, d_i)
- Two requests are compatible if they do not overlap

  $f_i <= f_j$ *or* $f_j <= s_i$

Minimize Lateness

Job i

- Requires t_i units of processing time
- is due at time d_i, the deadline
- Starts at s_i and finishes at $f_i = s_i + t_i$
- Has lateness $l_i = \max\{0, f_i - d_i\}$

Goal

- minimize the maximum lateness
- $L = \max_i l_i$

Greedy Approach

- Schedule jobs according to some natural order → which order?

Different Choices

Ideas: Minimize the maximum lateness

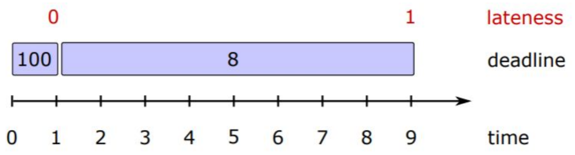
- Processing time t_i → Get short jobs out of the way
- Deadline d_i → Make sure that jobs with early deadlines get done
- Slac $d_i - t_i$ → take both processing time and deadline into account

We will use counter examples to show that two out of three orderings to not produce optimal solutions.

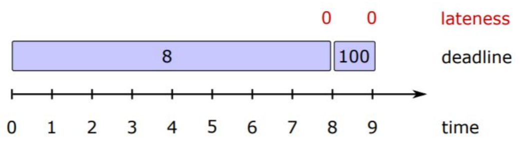
Order by Shortest Processing Time

Counterexample: Order by Shortest Processing Time

- Shortest processing time → maximum lateness 1

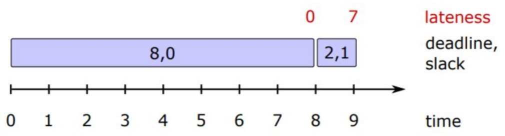


- Optimal schedule → maximum lateness 0

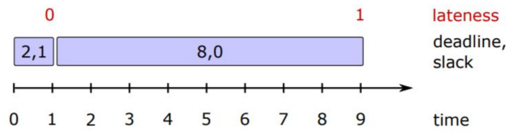


Counterexample: Order by Shortest Slack

- Shortest slack → maximum lateness 7



- Optimal schedule → maximum lateness 1



Order by Deadline

- The optimal schedule in the two previous examples had the jobs sorted by deadline.

Greedy algorithm:

- Sort the requests with respect to earliest deadline
- Set t = 0
- Pick request i
  - Schedule it from time t to t + ti
  - Set t = t + ti
- Repeat until the set of requests is empty

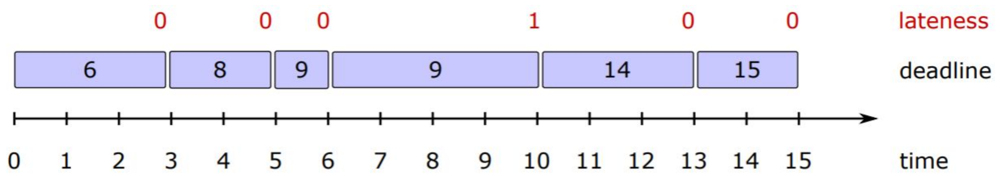
Note: The algorithm produces a scheule with no idle time.

Example: Earliest Deadline First

- Consider a set of jobs with processing times and deadlines

	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

- Order the jobs in ascending order of their deadlines
- Schedule them with no idle time

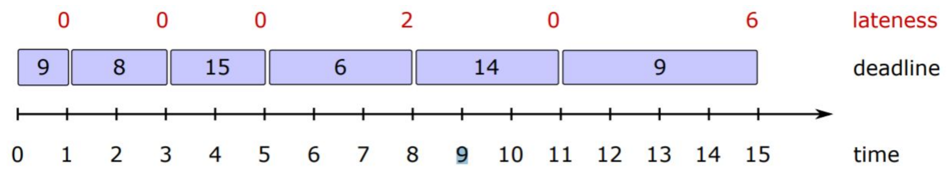


Example: Shortest Processing Time First

- Consider a set of jobs with processing times and deadlines

	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

- Order the jobs in ascending order of their processing times
- Schedule them with no idle time

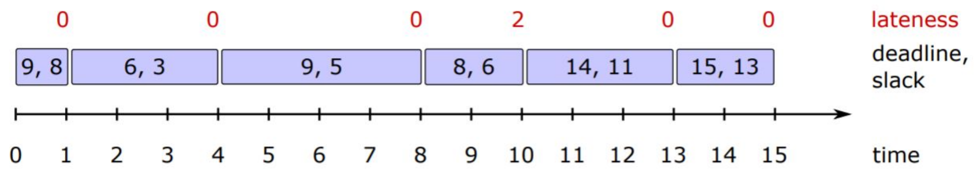


Example: Least Slack First

- Consider a set of jobs with processing times and deadlines

	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

- Order the jobs in ascending order of their processing times
- Schedule them with no idle time

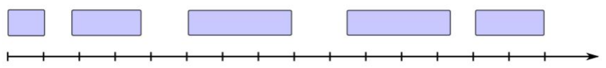


Example: We have seen three examples where the proposed method is the best among the three → Is it optimal?

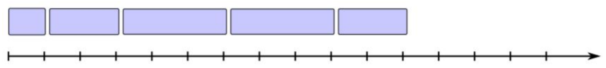
Optimal Schedule with no Idle Time

4.7 Idle Time - There is an optimal schedule with no idle time.

- Consider the following optimal schedule with idle time



- If we remove all idle time by shifting all jobs to the left



- The lateness cannot increase → The new schedule is also optimal.

Equal Maximum Lateness

- Define an inversion as a pair of jobs i and j such that…

dj < dj and i < j

- In our proposed algorithm, we consider jobs ordered by deadline

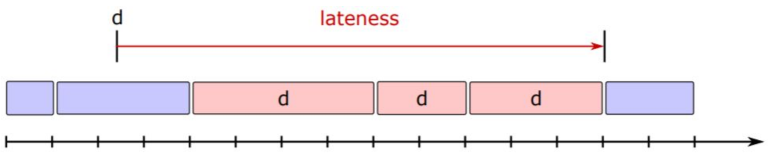
d1 ≤ d2 ≤ . . . ≤ dn

- By construction, the schedule given by our algorithm has no inversions.

(4.8) Equal Maximum Lateness - All schedules with no inversions and no idle time have the same maximum lateness.

Proof:

- No inversions mean that dj < dj and i < j
- The schedule can only differ if several jobs have the same deadline



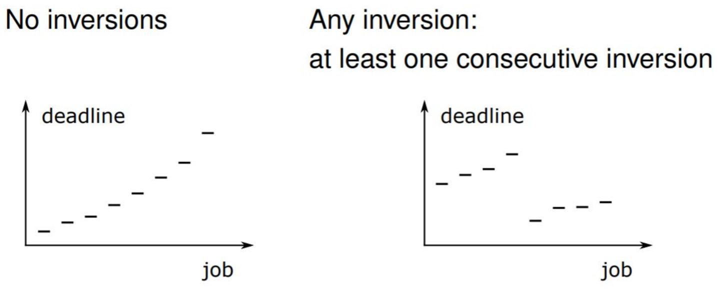
- Jobs with the same deadline are schedules consecutively.
- The maximum lateness of the group is not affected by their internal order.
- The maximum lateness of the schedule is the same.

Optimal Schedule

(4.9) Optimal Schedule - There is an optimal schedule that has no inversions and no idle time. We will show that the following are true:

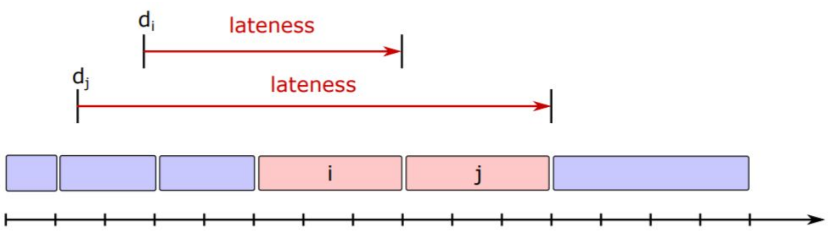
- If the optimal schedule has an inversion, there is a pair of consecutive jobs that is inversed.
- After swapping this pair we get one inversion less.
- The new swapped schedule is still optimal.

Start with any optimal schedule
→ Get rid of idle time
→ Swap consecutive jobs until no inversions remain

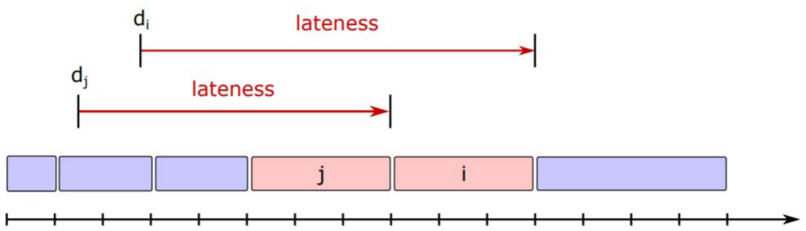


Swapped Schedule

- Consider a consecutive inversion:



- After swapping:



- The lateness for job j cannot increase → consider job i

Lateness

- The lateness of job i may have increased, but:

→ It cannot be more late than job j was before!

- Call f the finish time of job j before the swap. We have

$$dj < di\ .\ f - di < f - dj$$

- The lateness of job i after the swap is smaller than the lateness of job j before.

(4.10) Optimality of the Greedy Algorithm - The schedule produced by the greedy algorithm has optimal maximum lateness.

Lecture 11 - Shortest Paths in a Graph

Tues. Oct. 1, 2019

Shortest Paths in a Graph

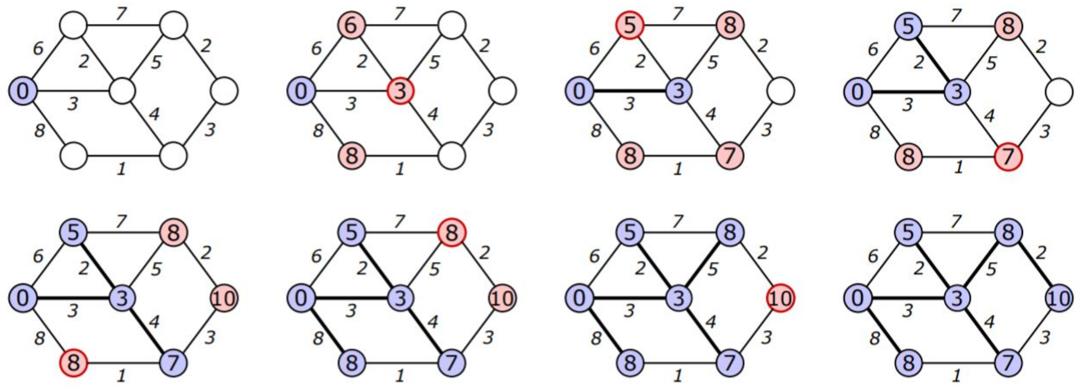
- Weighted graph - Assign a numeric value to each edge
- Length of a path - The sum of the weights
- **setup** - Considered a weighted, directed graph
- **problem** - What is the shortest path from a given start node to each one of the nodes in the graph?
- We will see a greedy algorithm that answers the question → The equivalent of BFS in an unweighted graph

Dijkstra's Algorithm

- In 1959, Edsger Dijkstra proposed the following algorithm:
- Consider a weighted, directed graph G = (V, E):

- We call the set of explored nodes S. The distance from s to a node v is called d(v).

- Set S = s and d(s) = 0
- For all edges (u, v) from a node u ∈ S to a node v ∈ V − S
  → Determine the distance d(v) from s to v, via u
  → Pick the node v with the shortest path from s
  → Add v to S
- Repeat until S = V



- **Blue** The set S of explored nodes
- **Red** Neighbors of S → Active Nodes

(4.14) Correctness of Dijkstra's Algorithm - Consider the set S at any point in the execution of the algorithm. For each u ∈ S, the path Pu is a shortest s-u-path

- The statement implies correctness, since it can be applied when S = V and the algorithm terminates.

Method

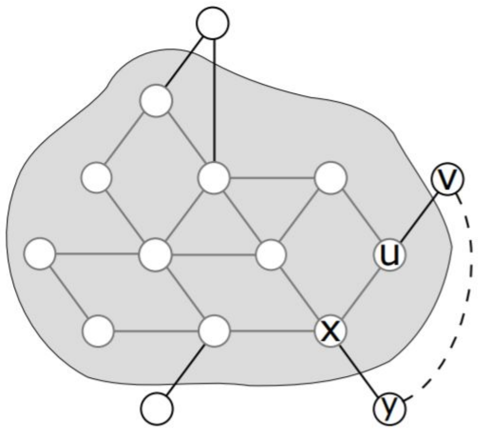
- Show that the algorithm stays ahead of all other solutions:
- In each step, it selects a path to a node v that is shorter than every other path to v.

→ use induction

Proof of (4.14)

- Use of induction on the size of S
- For |S| = 1, we have S = s and d(s) = 0
- Suppose true for |S| = k
  → Grow S to size k + 1 by adding v, via the node u
  → The edge (u, v) is the last edge on the s-v-path Pv
  → By the induction hypothesis: Pu is the shortest s-u-path
  → We now need to show that: Any other s-v-path P is at least as long as Pv

- Consider another s-v path P
- THe path must leave S at some other node x



- By construction, v was selected because it has the lowest distance

- In iteration k + 1, Dijkstra's algorithm considered the node y

→ the node y was rejected in favor of v
- No path Py from s to y is shorter than Pv : Py is a subpath of P

→ P is at least as long that Pv

Implementation of Dijkstra's Algorithm

Naive approach	• O(n) nodes to add • O(m) edges to consider → Running time: O(mn)
Appropriate data structures	• Keep track of d 0 (v) values • Use a priority queue • Update values when needed → Running time: ?

A priority queue has the following useful operations:
- `ExtractMin`
- `ChangeKey`

Using a Priority Queue

Implement Dijkstra's algorithm using a priority queue:
- Store the nodes in V with d 0 (v) as the key for v ∈ V

→ Initialize to ∞
- Select the node v to be added to S

→ ExtractMin
- When a node v is added to S: Update its neighbors

→ ChangeKey

Need to keep track of the nodes present in the queue: Use an array

In order to evaluate the running time of the algorithm → Determine how many operations of each type

Running Time Analysis

(4.15) Running Time of Dijkstra's Algorithm - Using a priority queue, Dijkstra's algorithm can be implemented on a graph with n nodes and m edges to run in O(m) time plus the time for n `ExtractMin` and m `ChangeKey` operations.
- Select the node v to be added to S
  - The greedy algorithm picks one node in each step
  - All nodes will be picked exactly once

→ Exactly one ExtractMin operation per node

- When a node v is added to S
    - Visit all nodes w that have edges to v
    - If the new path via v is shorter, update d 0 (w)
      → At most one ChangeKey operation per edge

Heap-Based Priority Queue

Heap-based priority queue → All priority queue operations run in O(log n) time.

- The (maximum) size of the heap is the number of nodes n:
    - `ExtractMin` O(log n)
    - `ChangeKey` O(log n)
- That makes a total of

$$O(m) + n \cdot O(\log n) + m \cdot O(\log n) = O(m \log n)$$

Lecture 12 - Minimum Spanning Trees

Thurs. Oct. 3, 2019

Network Design Problem

- **Context** - Imagine a set of locations. We wish to build a communication network over the locations to the lowest cost possible. Between certain locations, there is a possibility of a direct link and a cost associated.
- **Graph Models** - Consider a connected graph G = (V, E) with positive costs ce associated with the edges
- **Problem** - Find a subset T ⊆ E such that the total cost $\sum_{e \in T} c_e$ is as small as possible
- We are interested in a greedy algorithm that solved the problem → what should be chosen and by which rule?

Minimum Cost Solution

(4.16) Tree - Let T be a minimum-cost solution to the network design problem. Then, (V, T) is a tree.

Proof: We need to show that (V, T) has the two tree properties

- Connected → (V, T) is connected by definition
- Acyclic
    - Suppose that (V, T) contains a cycle C
    - Let e be an edge on the cycle
    - Now, (V, T − {e}) is also connected
    - Then (V, T − {e}) is a cheaper solution to the problem
    - We have a contradiction

→ (V, T) has no cycles

Minimum Spanning Tree

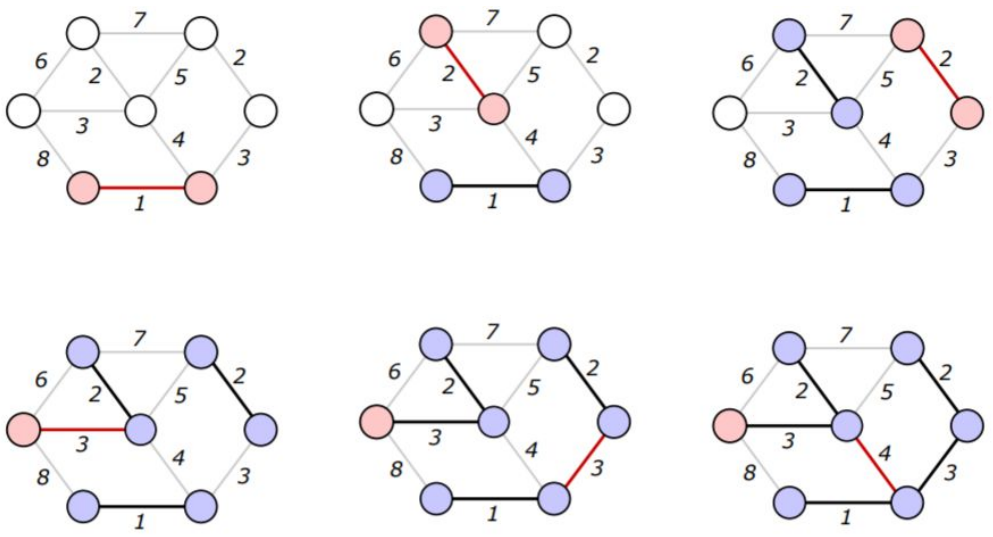
Spanning Tree - A subset T ⊆ E is called a spanning tree of G if (V, T) is a tree.

- There may be exponentially many spanning trees → How do we find a minimum spanning tree? → How do we find a minimum spanning tree?
- What do we pick?
  → Nodes
  → Edges
- In which order?
  → Increasing cost
  → Decreasing cost

Greedy Approaches

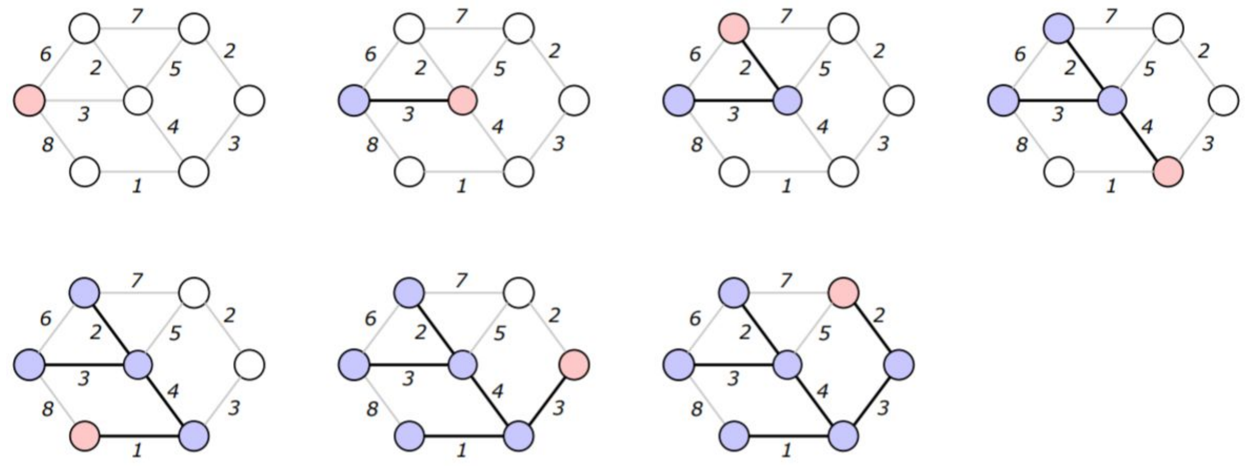
- **Kruskal's Algorithm** - Start without edges and successively insert edges from E in order of increasing cost. Discard edges that create cycles.
- **Prim's Algorithm** - Start with a node s and add the node that can be attached at the lowest cost to the partial tree.
- **Reverse-Delete Algorithm** - Start with the full graph and remove edges in order of decreasing cost. Do not remove edges if this will disconnect the graph.

Kruskal's Algorithm



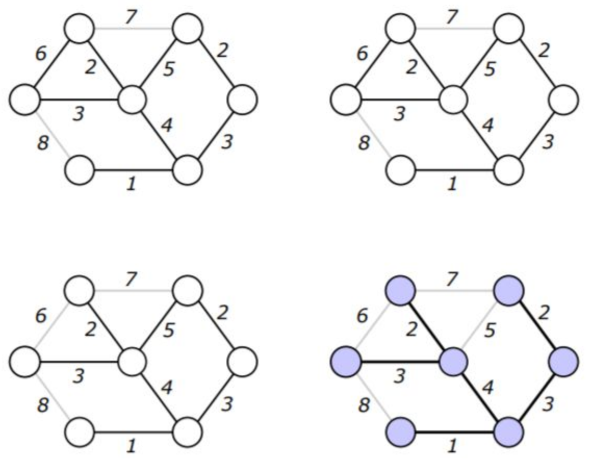
- Add edges successively
- Order after increasing cost
- Do not add edges that would create cycles

Prim's Algorithm



- Add nodes successively
- Pick the node that can be attached to the lowest cost

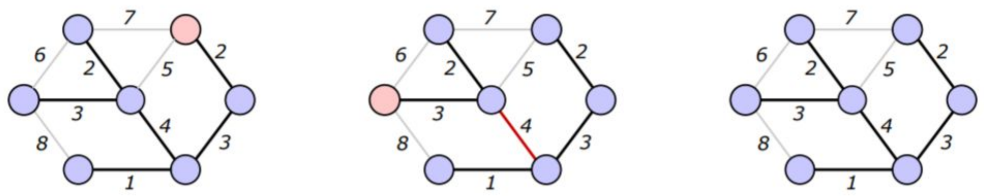
Reverse-Delete Algorithm



- Remove edges successively
- Pick the node that can be attached to the lowest cost
- Do not remove edges that disconnect the graph

Resulting Minimum Spanning Trees

- The result of the 3 greedy algorithms:



- The 3 algorithms are all guaranteed to find the minimum spanning tree of the graph

→ all give optimal solutions!

- This is a property of the problem: robustness

Include Edges

- We need a criterion to decide if an edge must be in the optimal solution

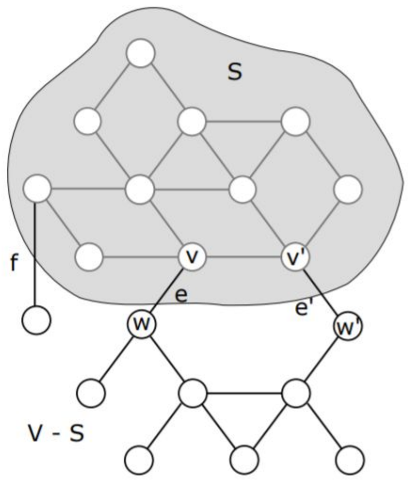
(4.17) Cut Property - Assume that all edge costs are distinct. Let S be any subset of nodes that is neither empty nor equal to all of V and let edge e = (u, v) be the minimum-cost edge with one end in S and the other in v − S. Then every minimum spanning tree contains the edge e.

- We will prove the statement using an exchange argument:
- Start with an optimal solution that does not contain e

→ Show that there is a cheaper MST containing e

Proof of (4.17)

Proof:



- Let T be a minimum spanning tree that does not contain e
- Identify an edge e 0 in T that is more expensive than e such that exchanging e for e 0 results in another spanning tree
- This spanning tree will be cheaper than T
- T is a tree, so there must be a path P from v ∈ S to w ∈ V − S
- Let e 0 = (v 0 , w 0) with v 0 ∈ S to w 0 ∈ V − S be on the path P
- Exchange e for e 0 to obtain T 0 = T − {e 0} ∪ {e}
- We now claim that T 0 is a spanning tree

Show that T' is a spanning tree:

- (V, T 0) is connected, since there is still a path from v to w
- (V, T) is acyclic
- The only cycle in (V, T ∪ {e}) is the one consisting of P and e
- This cycle is removed in (V, T 0) due to the removal of e 0
- e is the cheapest edge between S and V − S, so ce < ce 0
- Now the total cost of T 0 is less than that of T

Optimality of Kruskal's Algorithm

- We can now prove the optimality of Kruskal's and Prim's algorithms

(4.18) Optimality of Kruskal's Algorithm - Kruskal's Algorithm produces a minimum spanning tree of G.
→ Show that each addition of edges is justified by the cut property

Proof of (4.18)

Each addition of an edge is justified:

- Consider an edge e = (v, w) added by Kruskal's algorithm
- Let S be the set of nodes connected to v before the addition of e
- S and V − S are disconnected
- Now, v ∈ S and w ∈ V − S since e does not create a cycle Adding an edge from S to V − S could have been done without creating a cycle
- By (4.17) it must belong to every minimum spanning tree

We must now show that (V, T) is a spanning tree of G:

- By construction, (V, T) does not contain any cycles
- Assume that (V, T) is not connected
    - There must exist a non-empty subset of nodes S such that there is no edge from S to V − S
    - But since G is connected, there is at least one edge between S and V − S
    - The algorithm will add the cheapest of these edges
      → We have reached a contradiction
- Now, (V, T) is an optimal minimum spanning tree

Optimality of Prim's Algorithm

(4.19) Optimality of Prim's Algorithm - Prim's Algorithm produces a minimum spanning tree of G

Proof:

Each addition of an edge is justified:

- Call S the partial spanning tree created starting at s
- By definition, e = (u, v) is the cheapest edge connecting S and V − S
- By the cut property (4.17), the edge e must be in every minimum spanning tree

As before, we can show that (V, T) is a spanning tree of G

The Cycle Property

- We need a criterion to decide if an edge cannot be in the optimal solution

(4.20) Cycle Property - Assume that all edge costs are distinct. Let C be any cycle in G and let edge e = (v, w) be the most expensive edge belonging to C. Then edge e does not belong to any minimum spanning tree of G.

- We will prove the statement using an exchange argument:
- Start with an optimal solution that contains e
  → Show that there is a cheaper MST not containing e

Proof of (4.20)

- Let T be a minimum spanning tree that contains e.
- We need to find a cheaper edge e 0 such that we can exchange e for e 0 and still have a minimum spanning tree
  → How do we find a cheaper edge e' ?
- Remove the edge e from T
  - This partitions T into S containing v and V − S containing w
- The edge that we insert to connect the tree must have one edge in S and one edge in V − S

- Follow the cycle C from v to w:
  - The edges of C form by definition a path from v to w
  - We start in S and end up in V − S
  - There must an edge e 0 from S to V − S
- Now consider T 0 = T − {e} ∪ {e 0}
- Since e is the most expensive edge on the cycle C
  → Exchanging e for e 0 will lower the cost of the tree

As before, (V, T 0) is a cheaper minimum spanning tree.

Optimality of the Reverse-Delete Algorithm

(4.21) Optimality of Prim's Algorithm - The Reverse-Delete Algorithm produces a minimum spanning tree of G
Proof:
Each removal of an edge is justified:

- Consider an edge removed by the algorithm
- At the time it is removed, it belong to a cycle C
- Since it is the first edge removed from the cycle, it must be the most expensive edge on C

Show that the output is a spanning tree:

- (V, T) is connected, since the algorithm never disconnects the graph
- Suppose that the (V, T) contains a cycle

Lecture 13 - Exam 1 Review, Mergesort

Tues. Oct. 8, 2019

Priority Queues - What to Know

- A priority queue can be implemented using a min heap

- The min heap operations
    - ExtractMin()
    - Inset()
    - ChangeValue()

      → Be able to perform them on an instance
- The running time of the min heap operations

  → O(log n)
- A min heap can be represented as an array

  → Be able to interpret and use an array as a min heap

Graph Traversal - What to Know

- A graph can be traverse using
    - Breadth-First Search
    - Depth-First Search

      → Be able to perform them on an instance
- The running time of DFS and BFS

  → O(m + n)
- Recognize the use of BFS and/or DFS in a graph algorithms
    - Testing bipartiteness
    - Identifying connected components

      → Be able to use BFS and DFS as building blocks
- Know the importance of the choice of data structures

  → Recognize traversal implemented using stacks or queues

  → Adjacency lists as a way to represent graphs

Directed Graphs - What to Know

- Connectivity in directed graphs
    - Mutual reachability
    - Strong connectivity
    - Strong components are either identical or disjoint
- Directed Acyclic Graphs and topological ordering
    - Know what they can represent
    - Know the equivalence relation

  → Be able to compute a topological ordering

Scheduling - What to Know

- Greedy Algorithms find optimal solutions

  → Be able to find an optimal solution
- Know which strategy to adopt
    - Interval scheduling: sort by earliest finish time
    - Interval partitioning: sort by earliest start time

- Minimize lateness: sort by earliest deadline

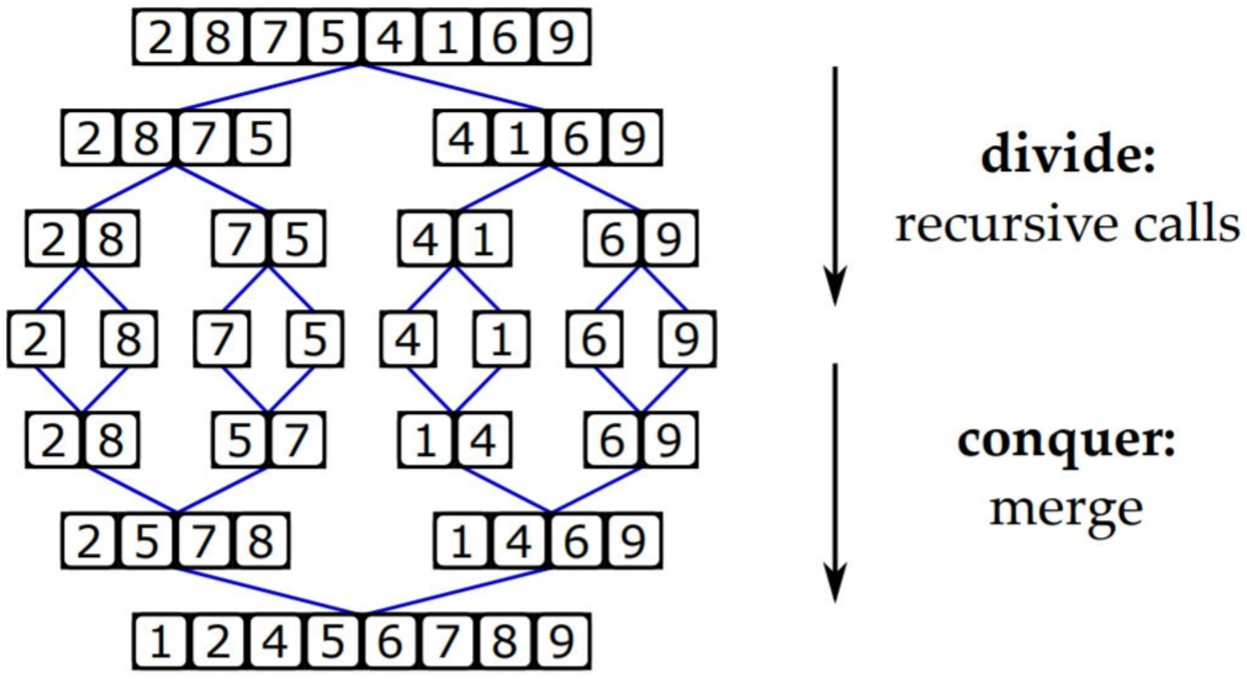
Exam Questions Content

1. True / False questions on certain algorithm aspects
2. Will have a question on being able to take BFS of graph, build traversal tree, and answer questions about the process, certain aspects or properties of algorithm or each graph/tree
3. Question with a few algorithms written in Python not pseudo code
    a. if, not, list
    b. Don't require object oriented thinking, will make, for example, stack functions obviously (like s = stack, s.pop() for stack pop function
4. No proofs

Mergesort

Divide and Conquer - Mergesort

- Merge sort is an efficient Divide and Conquer algorithm



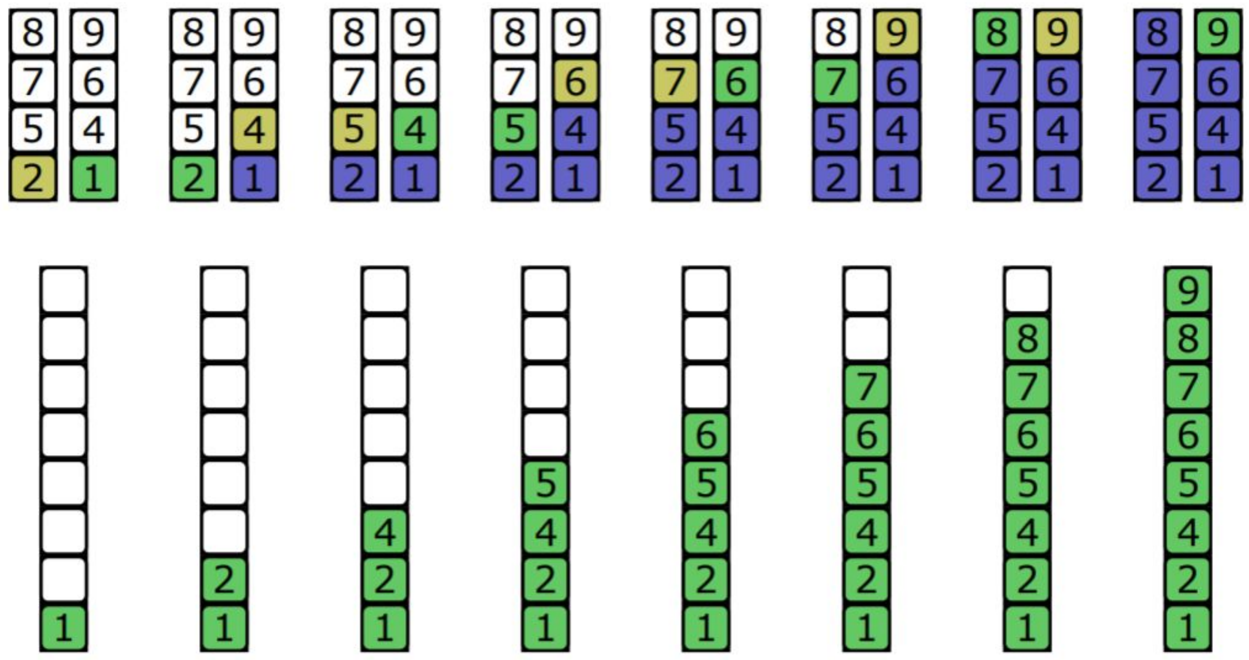
Recursive Implementation

```
function mergeSort(a[1 .. n])
      if (n == 1)
```

```
            return a
        else
            a1 = mergeSort(a[1 .. n /2])
            a2 = mergeSort(a[n/2+1 .. n])
            return merge(a1, a2)
        end if
end function
```

→ How do we **merge** two sorted arrays?

Linear Time Merging



Merging - Pseudocode

```
function merge(a1[1 .. n], a2[1 .. n])
        i, j, k = 1
        while (i ≤ n and j ≤ n)
            if (a1[i] ≤ a2[j])
                a[k++] = a1[i++]
            else
                a[k++] = a2[j++]
            end if
        end while
        while (i ≤ n)
            a[k++] = a1[i++]
```

```
    while (j ≤ n)
        a[k++] = a2[j++]
    return a
end function
```

The Mergesort Algorithm

Mergesort Algorithm

- Sorting is hard
    - Let's a try a smaller problem:
    - Divide until the problem gets easy
      → Sorting a one-element list is easy!
- How do we combine two sorted lists?
  → Merge in linear time

Mergesort is our first example of **Divide and Conquer**

Divide and Conquer Template

Divide and Conquer Template

- Divide the input into two pieces of equal size
  → Solve the 2 subproblems separately using recursion
- Combine the two results in an overall solution
- Spend only linear time on divide and combining

Analyzing a Divide and Conquer Algorithm general involves solving a recurrence relation
→ Bound the running time recursively
→ Express in terms of smaller instances

Analysis of the Algorithm

- Let T(n) be the worse-case running time on inputs of size "n"
- The running time of the algorithm can be expressed as

divide the problem	O(n)
recursive calls	2 * T(n/2)
combine the solutions	O(n)

- To simplify, we will assume that "n" is even.

→ Else: replace 2* T(n/2) with T(ceiling(n/2)) + T(floor(n/2))

Recurrence Relation

5.1 Recurrence Relation

For some constant c,	$T(n) <= 2\ T(n/2) + cn$
when n >= 2 and	$T(2) <= c$

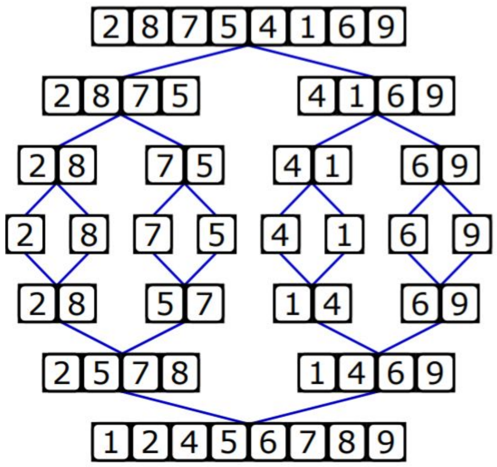
No explicit asymptotic bound
→ T(n) is bounded recursively in terms of smaller instances
Solve the recurrence relation
→ Find an inequality where T(n) appears only on the LHS

Solving Recurrence Relations

There are two approaches to solving a recurrence:

- Unroll the recursion
  - Expand the recursion
  - Find a pattern
  - Sum over all levels of recursion
    → Until the base case is reached
- Guess the solution
  - Useful if the pattern is known but the parameters
  - Substitute the guess into the recurrence relation
    → Prove using induction

Mergesort - Running Time



- Analyze the first levels
  0. One problem of size cn
  1. Two problems of size cn/2
  2. Four problems of size cn/4
- Identify a pattern
  k 2^k problems of size $cn/2^k$
  → each level cost cn
- Sum over all levels
  - Divide by 2 k times to get to 1:
  - $k = \log_2 n$
    → O(n log n)

Total running time of mergesort is O(n log n)

Substitute a Solution

(5.2) A solution to the Recurrence Relation - Any function T(*) satisfying the recurrence relation (5.1) is bounded by O(n log n) when n > 1.
Proof:
We believe that T(n) <= cn logn for n >= 2

- True for n = 2 since c *2 log 2 = 2c

→ We know from (5.1) that T(2) <= c
- Now suppose true for all values , < n
    - Induction hypothesis: T(m) <= cm log m

  → In particular T(n/2) <= c(n/2) log n/2

Substitute a Solution

Now, use T(n/2) <= c(n/2) log n/2

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &\leq 2c \cdot (n/2)\, log\, (n/2) + cn \\ &= cn\, [log\, n - log\, 2] + cn \\ &= cn\, [log\, n - 1] + cn \\ &= cn\, log\, n - cn + cn \\ &= cn\, log\, n \end{aligned}$$

Thus, by induction, the statement holds for all values of n.

(Lecture 14) - Midterm 1

Thurs. Oct. 10, 2019

Lecture 14 - Recurrence Relations

Tues. Oct. 15, 2019

Divide and Conquer Template

We will consider a more general class of algorithms:
Divide-and-Conquer algorithms that follow the template

Divide and Conquer Template
- Divide the input into “q” pieces of size n/2
  → Solve the “q” subproblems separately using recursion
- Combine the “q” results in an overall solution
- Spend only linear time on divide and combining

As for Mergesort, we will define and solve a recurrence relation:
→ Bound the running time recursively
→ Express in terms of smaller instances

Recurrence Relation

- Let T(n) be the worst-case running time on inputs of size n
- The running time of the algorithm can be expressed as
    - divide the problem O(n)
    - recursive calls q * T(n/2)
    - combine the solutions O(n)

(5.3) Recurrence Relation
For some constant c,

$$T(n) <= q\ T(n/2) + cn$$

when n >= 2 and

$$T(2) <= c$$

Solving Recurrence Relations (1)

Recall how we unrolled the recursion to solve the recurrence relation for Mergesort:

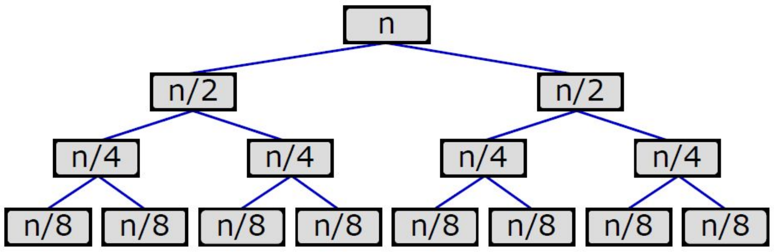
- Expand the recursion
- Find a pattern
- Sum over all levels of recursion

→ Until the base case is reached

We now consider 3 cases:

q = 1 → 1 recursive call
q = 2 → 2 recursive calls
q > 2 → 3 or more recursive calls

Unroll the Recursion: q = 2



Level	Number	Size	Cost	Total
0	1	n	c * n	1 * cn = cn
1	2	n/2	c * n/2	2 * cn/2 = cn
2	2^2	$n/2^2$	$c * n/2^2$	$2^2 * cn/2^2 = cn$
k	2^k	$n/2^k$	$c * n/2^k$	$2^k * cn/2^k = cn$

Solution to the Recurrence Relation

The Mergesort algorithm is defined by:

- Two recursive calls
  → q = 2
- Each subproblem has size n/2
  → T(n/2)
- Divide and combine in linear time
  → O(n)

The running is given by:

- Constant cost at level “k”
  → cn
- Number of levels - Depth of the recursion
  → $\log_2 n$

→ Running time: O(n log n)

The Case of q > 2 Subproblems

Now consider q > 2:

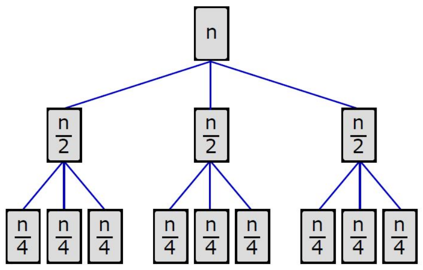
- **Three or more recursive calls**
- Each subproblem has size n/2
- Divide and combine in linear time

Note: The number of levels is still $\log_2 n$

- What is the cost for an arbitrary k?

→ Unroll the recursion starting at the first levels…

Unroll the recursion: q > 2



Level	Number	Size	Cost	Total
0	1	n	c * n	1 * cn = cn
1	q	n/2	c * n/2	q * cn/2 = (q/2) * cn
2	q^2	$n/2^2$	$c * n/2^2$	$q^2 * cn/2^2 = (q/2)^2 * cn$
k	q^k	$n/2^k$	$c * n/2^k$	$q^k * cn/2^k = (q/2)^k * cn$

(2) Solving the Recurrence Relation

the running time is given by:

- Cost at level k
  $\rightarrow (\frac{q}{2})^k \cdot cn$
- Number of levels - Depth of the recursion
  $\rightarrow \log_2 n$
- Total running time
  → Sum over all $\log_2 n$ levels

$$T(n) \leq \sum_{k=0}^{log_2 n-1} (\frac{q}{2})^k cn = cn \sum_{k=0}^{log_2 n-1} (\frac{q}{2})^k$$

  → We recognize a geometric sum!

The Geometric Sum

Geometric Sum - For r =/= 1 the sum of the N first terms of a geometric series is

$$\sum_{k=0}^{N-1} r^k = \frac{r^N - 1}{r - 1}$$

We have:

- $N = \log_2 n$
- $r = q/2$

$$T(n) \leq cn \frac{r^{log_2 n} - 1}{r - 1} \leq cn \frac{r^{log_2 n}}{r - 1}$$

- The constants does not change the asymptotic behavior

$$T(n) \leq \frac{c}{r-1} \cdot nr^{log_2 n}$$

A Logarithm Identity

A Logarithm Identity - For a > 1, and b > 1 we have $a^{log\, b} = b^{log\, a}$

- To see why this is true, we take the logarithms of both sides:

LHS → $log(a^{log\, b}) = log\, b \cdot log\, a$

RHS → $log(b^{log\, a}) = log\, a \cdot log\, b$

Now, we have

$$r^{log_2 n} = n^{log_2 r} = n^{log_2 (q/2)} = n^{log_2 q - 1}$$

$$T(n) \leq \frac{c}{r-1} \cdot nr^{log_2 n} = \frac{c}{r-1} \cdot n \cdot n^{log_2 q - 1} = \frac{c}{r-1} \cdot n^{log_2 q}$$

An Upper Bound for q > 2

(5.4) Asymptotic Upper Bound - Any function T(*) satisfying (5.3) with q > 2 is bounded by $O(n^{\log 2\ q})$.

The running time is more than linear, since $\log_2 q > 1$ for q > 2

- q = 3
    - $\log_2 3 = 1.5850 \approx 1.59$ $\rightarrow O(n^{1.59})$
- q = 4
    - $\log_2 4 = 2$ $\rightarrow O(n^2)$

→ Why can't we use q = 2 and get O(n)?

The Case of q = 1 Subproblems

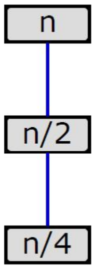
Now consider q = 1:

- **One recursive call**
- Each subproblem has size n/2
- Divide and combine in linear time

Note: The number of levels is still $\log_2 n$

- What is the cost for an arbitrary level k?

→ Unroll the recursion starting at the first levels...



Unroll the Recursion: q = 1

Level	Number	Size	Cost	Total
0	1	n	c * n	1 * cn = cn
1	q	n/2	c * n/2	cn/2 = (1/2) * cn
2	q^2	$n/2^2$	$c * n/2^2$	$cn/2^2 = (1/2)^2 * cn$
k	q^k	$n/2^k$	$c * n/2^k$	$cn/2^k = (1/2)^k * cn$

(3) Solving the Recurrence Relation

The running time is given by:

- Cost at level k
  $\rightarrow (\frac{1}{2})^k \cdot cn$
- Number of levels - Depth of the recursion
  $\rightarrow \log_2 n$
- Total running time
  → Sum over all $\log_2 n$ levels

$$T(n) \leq \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k \, cn = cn \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k$$

→ A **convergent** geometric sum!

An Upper Bound for q = 1

The sum is convergent, so the running time is bound by

$$T(n) \leq \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k \leq 2 \cdot cn = O(n)$$

(5.5) Asymptotic Upper Bound - Any function T(*) satisfying (5.3) with q = 1 is bounded by O(n).

The Effect of the Parameter q

Consider the effect of q in recurrences of the form

$$T(n) \leq q\, T(n/2) + O(n)$$

- q = 1
    - Half of the work spent in the first level

  → Linear running time
- q = 2
    - Equal amount of work spent in each level

  → O(n log n) running time
- q > 2
    - More and more work spent in each level

  → Polynomial running time: O(n^d) with d > 1

A Related Recurrence

Now consider a recurrence relation of the form

$$T(n) \leq q\, T(n/2) + O(n^2)$$

Unroll the recursion:

Level	Number	Size	Cost	Total
0	1	n	c * n^2	1 * cn^2 = cn^2
1	q	n/2	c * (n/2)	2 * c * $(n/2)^2$ = $(cn/2)^2$
2	q^2	$n/2^2$	c * $(n/2^2)^2$	2 * c * $(n/2^2)^2$ = $(cn/2^2)^2$
k	q^k	$n/2^k$	c * $(n/2)^{2k}$	2 * c * $(n/2^k)^2$ = $(cn/2^k)^2$

(4) Solving the Recurrence Relation

The running time is given by:

- Cost at level k
  → $(\frac{1}{2})^2$ ·
- Number of levels - Depth of the recursion
  → $\log_2$ n
- Total running time
  → Sum over all $\log_2$ n levels

$$T(n) \leq \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k cn^2 = cn^2 \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k$$

→ Once again, we obtain a **convergent** geometric sum!
We recognize the sum from before, in the case of q = 1

$$T(n) \leq \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k cn^2 = cn^2 \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k$$

- The sum is convergent, so we have

$$T(n) \leq cn^2 \sum_{k=0}^{log_2 n-1} (\tfrac{1}{2})^k \leq 2 \cdot cn^2 = O(n^2)$$

Analogy

$$T(n) \leq 1 \cdot T(n/2) + O(n)$$

- Half of the work spent in the first level

→ Linear running time

$$T(n) \leq 2 \cdot T(n/2) + O(n^2)$$

- Half of the work spent in the first level

→ Quadratic running time
The complete problem with $\log_2$ n levels of recursion
→ Same asymptotic behavior as the first level!

Lecture 15 - Counting Inversions

Thurs. Oct. 17, 2019

Analyze Rankings

- **Context** - Based on your viewing history and how you rated other titles, what will Netflix recommend for you?

- **Approach** - One way is to give recommendations based on what other users with similar tastes have liked
- **Problem** - How can we compare your rankings to those of another user?

→ We will use the Divide-and-Conquer paradigm

Compare Two Rankings

- Label the movies 1, 2, …, "n" according to your ranking
- Another user ranks the same movies a_1, a_2 , … , a_n. The list is a permutation of the numbers from 1 to "n".

- **Goal** - Compare the two rankings
- **Means** - Define a measure of how far from sorted a list is

- A sorted list is a 100% match
- A reversed sorted list is as bad as it gets
- What about everything in between?

Counting Inversions

- **Idea** - Count the # of pairs of elements that are out of order.

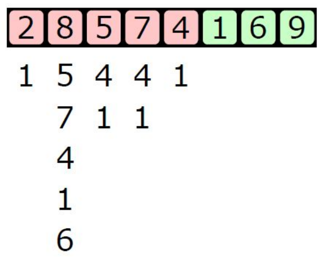
Inversion - Two indices i < j form an inversion if $a_i > a_j$

	Nb of inversions
Sorted list	0
Reversed sorted list	$\binom{n}{2}$

→ Count the # of inversions

Examples

ranking	inversions	nb
$[1,2,3,4]$	$\varnothing$	0
$[4,3,2,1]$	$(4,3), (4,2), (4,1), (3,2), (3,1), (2,1)$	6
$[2,4,1,3]$	$(2,1), (4,1), (4,3)$	3



Divide-and-Conquer Approach

- **Brute-force**
  - Look at all pairs (a_i, a_j) and determine whether they constitute is an inversion:
  - This will take $O(n^2)$ time
- **Claim**
  - The problem can be solved in O(n log n) time using Divide-and-Conquer

Divide-and-Conquer Framework
- Divide the input into two pieces of equal size
- Solve the two subproblems recursively
- Combine the two results
  - Spend only linear time for dividing and recombining

Relation to Mergesort

- **Challenge** - Count all inversion without even looking at them individually
- **Idea** - Use Mergesort and count inversions while performing the sorting.

- If the list has size 1
  → List is orted, no inversions
- Divide the list in two sublists
  → Return sorted sublists and #of inversions
- Merge the two sorted sublists and count the inversion
  → Return sorted list & total # of inversions

Mergesort

The inversion can be between elements:

- In the same sublist
    - Handled in the next recursive call to mergesort

  → Define a new recursive function: sort_and_count
- In different sublists
    - Handled by the function merge

  → Define a new function: merge_and_count

Analogy:

- Mergesort does not sort

  → Merge sorted lists in merge
- sort_and_count does not count, it only sums

  → Inversions are counted in merge_and_count

Merge and Count

We only need to worry about inversion between 2 sublists

→ All inversion within sublists have been handled by the recursive calls!

- L has been split into 2 sublists

  → A and B
- The 2 sublists have been sorted

  → The next step is to merge the sublists

How do we count inversion between sublists while merging?

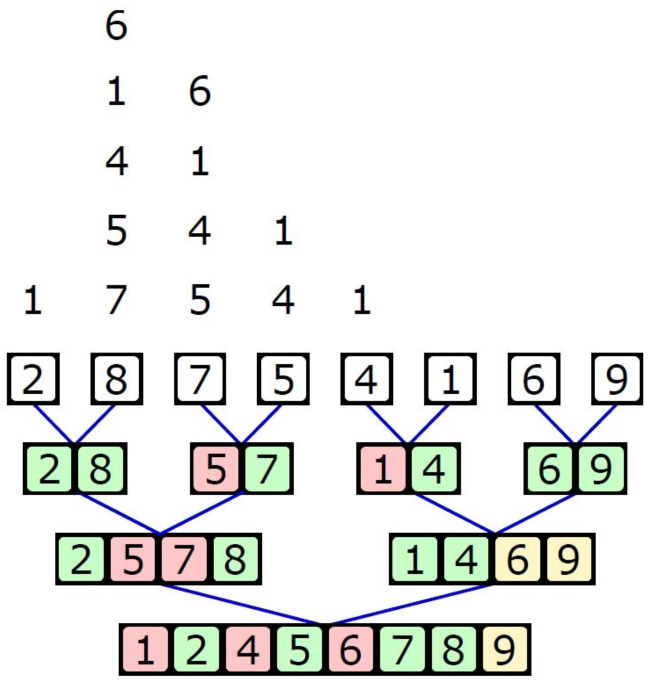
- Every time an element ai from A is selected
    - a_i is smaller than all remaining elements in B
    - a_i comes before all of them in the original list

  → No inversions
- Every time an element bj from B is selected
    - b_j is smaller than all remaining elements in A
    - b_jj comes after all of them in the original list

  → All remaining elements in A constitutes inversions

→ All remaining elements in A constitutes inversions

Example: Merge and Count



Linear Merge and Count

We need to show that merge_and_count runs in linear time.

- Each time an element is selected
- The size of the sublists are known
    - n/2
- The number of elements selected from each one of them are known
    - i or j
- In constant time we can compute the # of inversions
    - n/2 - i or 0

There are "n" elements to be selected:

- The time added for the counting is O(n)
- The total running time is O(n)

Sort and Count

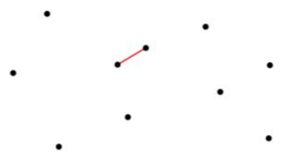
- Since our Merge-and-Count takes O(n) time, the running time of Sort-and-Count satisfies the recurrence
- Therefore, by (5.2)

(5.7) Sort-and-Count - The Sort-and-Count algorithm correctly sorts the input list and counts the number of inversions. It runs in O(n log n) time for a list with n elements.

Lecture 16 - Divide & Conquer: Closest Pair of Points

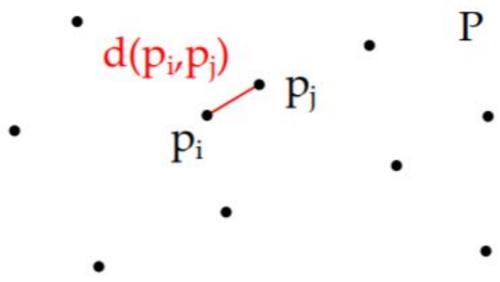
Tues. Oct. 22, 2019

Closest Pair of Points



Context - Computational Geometry, computer graphics, computer vision, geographic information
Problem - Given n points in the plane, find the pair that is the closest together.
Brute-Force - Compute the distances between the $O(n^2)$ pairs of points and return the smallest.

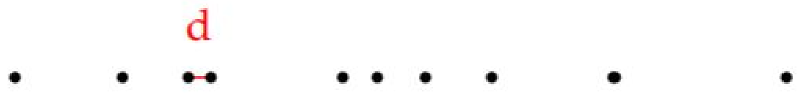
Notation



- Point - p_i has point coordinates (x_i, y_i)
- Set of Points - $P = \{p_1, p_2, \ldots, p_n\}$
- Distance - For $p_i, p_j \in P$, the Euclidean distance is $d(p_i, p_j)$

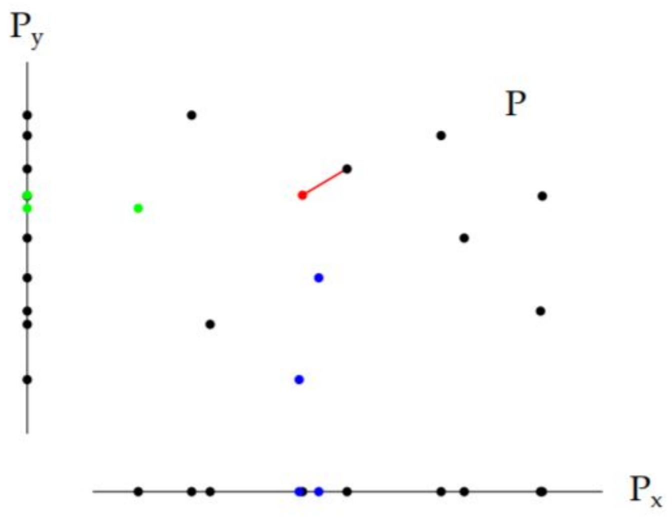
First Approach

- **In 1D** - Sort the points and compute the distance between adjacent points. −→ One of these distances must be the minimum.



- **Idea** - Sort by x or y coordinates and hope that close points are close in the sorted list. → Not necessarily true

Sort by x or y Coordinates



- We can imagine an arbitrary number of points that are closer in the sorted lists than the actual closest point.

Divide-and-Conquer Approach

We will use the Divide-and-Conquer paradigm:

- Aim for a running time of O(n log n)

- Split the set of points in 2 subsets
  - Find the min. dist. in each subset
- Dividing & combining in linear time
  - Compare the minima from the subset

→ What if the minimum distance is between subsets?

Dividing

Prepare the recursion: Sort the points in P

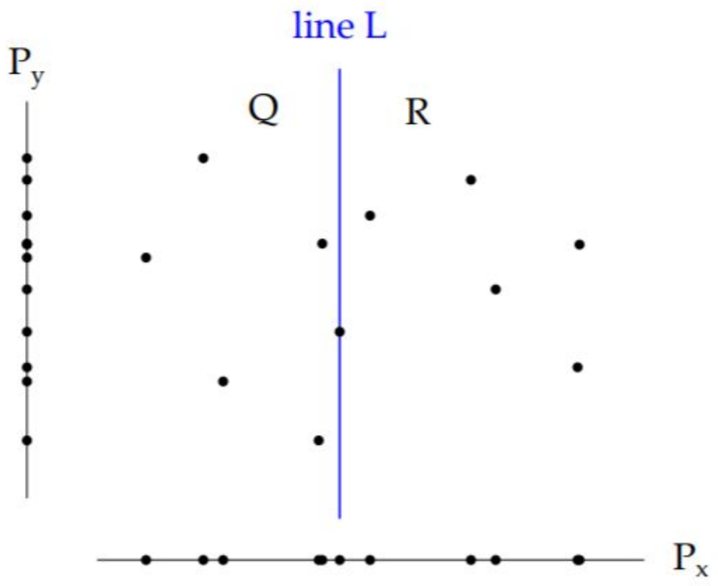
- by x-coordinate → P_x
- by y-coordinate → P_y

For each entry, keep track of the position of that point in other list.

- Q is the set of the first n/2 points in P_x
  - The list Q_x is given by the splitting
    → The list Q_y is found by a single pass through P_y
- R is the set of the last n/2 points in P_x
  - The list R_x is given by the splitting
    → The list R_y is found by a single pass through P_y

→ Dividing can be done in O(n) time

The First Level of Recursion



The Recursion

- Finding the minimum distance within a subset is easy
  → Divide until the problem is trivial:
  A subset with two points
- Finding the minimum between subsets is harder T
  - here are $O(n^2)$ pairs to compute distances between
    → We need a O(n) strategy!

Remember the problem of counting inversions?

Suppose the subproblems are solved:

- In Q
  - Closest points are q_0^* and q_1^*
  - Minimum distance dQ = $d(q_0^*, q_1^*)$
- In R
  - Closest points are r ∗ 0 and r ∗ 1
  - Minimum distance dR = $d(r_0^*, r_1^*)$
- Let δ = min (dQ, dR)

The question is now:

Are there points q ∈ Q and r ∈ R such that d(q, r) < δ

Candidates for the Closest Points

We have three candidates for the minimum distance:

→ dQ = $d(q_0^*, q_1^*)$

→ dR = d(r_0^*, r_1^*)
→ d(q, r)

- Let x^* be the x-coordinate of the rightmost point in Q.
- Let L be the line that separates Q from R L = {(x, y) : x = x^*}

(5.8) - If there exists q ∈ Q and r ∈ R for which d(q, r) < δ, then both q and r must lie within a distance of L.

Proof of (5.8)

- Write $q = (q_x, q_y)$ and $r = (r_x, r_y)$. By definition, we have $q_x \leq x^* \leq r_x$. Now we have know that
  - $x^* - q_x \leq r_x - q_x \leq d(q, r) < \delta$
- and
  - $r - q_x \leq r_x - q_x \leq d(q, r) < \delta$
- So, both q and r have an x-coordinate within of x. This shows that they lie within distance from the line L.

Restriction to a Smaller Set

- Let S be the set of points in P within of L.
- Construct the list Sy by sorting the points in S by increasing y-coordinate
- → Can be done in a single pass through Py : linear time

Now, restate (5.8) in terms of S:

(5.9) - There exist q ∈ Q and r ∈ R for which d(q; r) < δ if and only if there exist s, s' ∈ S for which d(s, s') < δ

- We have restricted the set of possible closest pairs split between the two sets to S

→ But how does this help?

Sparsity of the Points

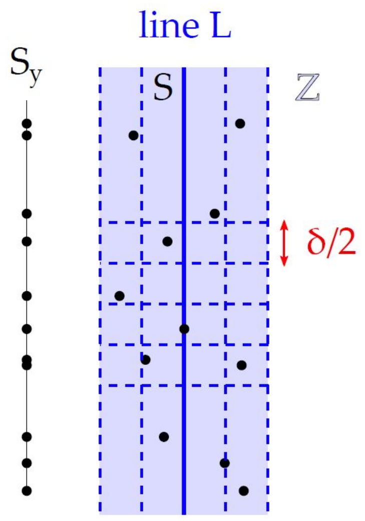
(5.10) Sparsity of Points - If s, s' ∈ S have the property that d(s, s') < δ , then s and s0 are within 15 positions of each other in the sorted list Sy .

The power of this statement is that there is a constant: 15

Strategy:

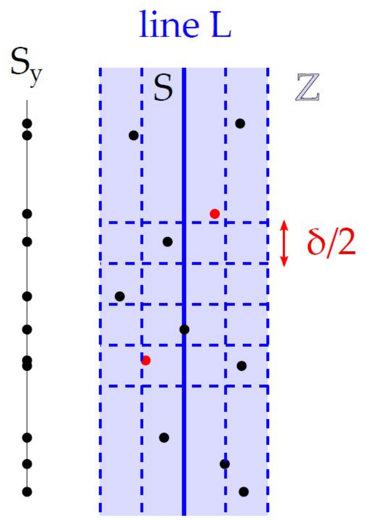
- Compute the distance to the 15 following points
  → The minimum distance in S will be in the list
- Constant # of ops. in a single pass
  → Linear Time

Partitioning of the Plane



- Consider the subset of the plane consisting of all points within distance δ of L
- Partition Z into boxes: squares with a side of δ/2
- Two points within the same box both belong to either Q or R
- But any two points in the same box are at a distance of at most $\frac{\sqrt{2}}{2} \cdot \delta < \delta$
- This contradicts the minimum distance δ

→ Each box can contain at most one point



- Suppose that s, s' ∈ S have the property that d(s, s') < δ
- Suppose that s and s' are at least 16 positions apart in Sy
- There can be at most one point per box
- s and s' must be at least three rows apart:
  → At a distance of at least $\frac{3\delta}{2}$

→ ! Contradiction

Algorithm

Pretation:

- Construct Px and Py
  → O(n log n) time
- Call the recursive function ClosestPair

ClosestPair:

- Base case: $|P| \leq 3$
  → Find minimum by brute-force
- Construct Qx, Qy, Rx, Ry
  → O(n) time
- Recursive calls on Q and R
- Construct the set S
- Return the minimum distance in Q, R or S

Construct the set S:

- $\delta = min(d_Q, d_R)$
- $x^* = max\{x : (x, y) \in Q\}$

- $L = \{(x, y) : x = x^*\}$
- $S = \{p \in P : d(p, L) < \delta\}$
- Construct Sy
  → O(n) time

Find the minimum distance in S:
- For all points s ∈ Sy ,
    - Compute the distance to the next 15 points
  → O(n) time

Analyzing the Algorithm

(5.11) Correctness - The algorithm correctly outputs a closest pair of points in P.

Proof: by construction.

(5.12) Running Time - The running time of the algorithm is O(n log n).

Proof:
The initial sorting takes O(n log n) time. The running time for the recursion satisfies the recurrence (5.1) and therefore, by (5.2) we have the running time O(n log n).

Lecture 17 - Weighted Interval Scheduling

Thurs. Oct. 24, 2019

Weighted Interval Scheduling

- We dispose of one resource and have **weighted** requests

- **request** - use the resource during a given time, each request has an associated value (or weight)
- **rule** - no 2 overlapping requests can be accepted
- **goal** - maximize the total value of the scheduled requests

We have “n” requests labeled 1,... “n”
start and finish times(s_i, f_i)
value v_i
- Two requests are **compatible** if they do not overlap

- Choose the compatible subset S ⊆ {1, ..., n} that maximizes $\sum_{i \in S} v_i$

CSE 3500 Master Notes

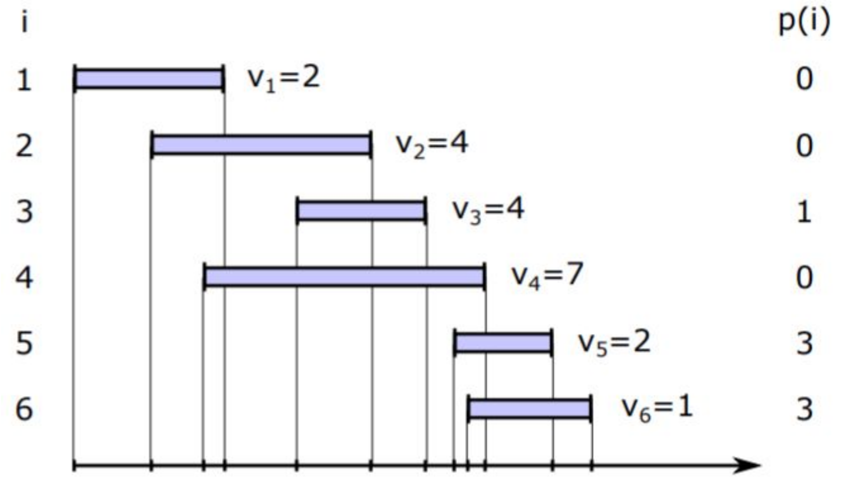
Greedy Algorithms

- Interval Scheduling
  → Sort the request with respect to (w.r.t.) earliest finish time
  → Discard incompatible request & repeat
- interval Partitioning
  → Sort the request w.r.t. earliest start time
  → Allocate a new resource when necessary
- Minimize Lateness
  → Sort the request w.r.t. to earliest deadline
  → Schedule requests w/ no idle time
- Weighted Interval Scheduling
  → No greedy algorithm is known for this problem!

Problem Formulation

- **Sorting** - In order of ascending finish time $f_1 \leq f_2 \leq \ldots \leq f_n$
- **Ordering** - A request "i" come before a request "j" if "$f_i < f_j$"
- **Last disjoint** - We define the function p(j) for an interval "j" as the largest index "i" such that "i" and "j" are disjoint

Example: An Instance of the Problem



Optimal Solution

- Consider an instance of the **Weighted Scheduling Problem** on "n" requests.

→ Consider an optimal solution "O"

- We do not know anything about the solution, but we can state the obvious
- One of the two following must be true
  - The last interval "n" belongs to O

- ○ The last interval “n” does not belong to O

→ We hope to break down the prob. into subprobs.

Subproblems

- The last interval “n” belongs to O
    - No interval between p(n) and “n” can belong to O
      → O includes the optimal soln. to the subprob. consisting of the requests {1, 2, …, n -1}
- The last interval “n” does not belong to O
  → O is equal to the optimal solution to the subproblem consisting of the requests {1, 2, . . . , n − 1}

Else, the selected subset could be exchanged for the optimal solution
→ This contradicts the optimality of O

Recursive Solution

- O_j - Optimal solution to the problem on {1, 2, … ,j }
- OPT(j) Value of the optimal solution O_j
  → define OPT(0) = 0

- If “j” is the optimal solution
  → $O_j = \{j\} \cup O_{p(j)}$
  → $OPT(j) = v_j + OPT(p(j))$
- If “j” is not the optimal solution
  → $O_j = O_{j-1}$
  → OPT(j) = OPT(j - 1)

- We can now express the optimal solution to the prob.

→ In terms of optimal solns. to subproblems

(6.1) Recursive Formulation - $OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$

- We can also formulate a strategy to choose the requests:

(6.2) Recurrence Equation - Request j belongs to an optimal solution on the set {1, 2, . . . , n} if and only i $v_j + OPT(p(j)) >= OPT(j - 1)$

Algorithm

Assume that

- The requests are sorted by ascending finish time

- We have computed p(j) for each j

The following recursive function computes the optimal solution:

```
function OptR(j)
    if j == 0 then
        return 0
    else
        return max(v[j] + OptR(p[j]), OptR(j - 1))
    end if
```

Analyzing the Algorithm

(6.3) Correctness of the Algorithm - `OptR(j)` correctly computes OPT(j) for each j = 1, 2, . . . , n.

Proof: by induction

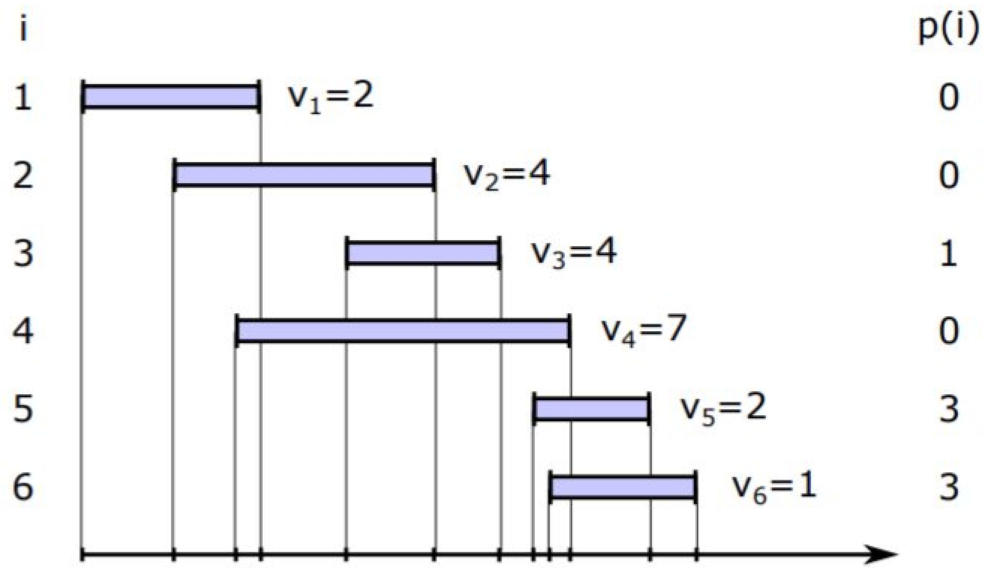
- By definition, OPT(0) = 0
- Take j > 0 and suppose that OptR(i) correctly computes OPT(i) for all i < j.
- By the induction hypothesis:
    - OptR(j-1) = OPT(j − 1)
    - OptR(p[j]) = OPT(p(j))

Now, from (6.1) it follows that

OPT(j) = max(`v[j] = OptR(p[j]), OptR(j-1))`
`= OptR(j)`

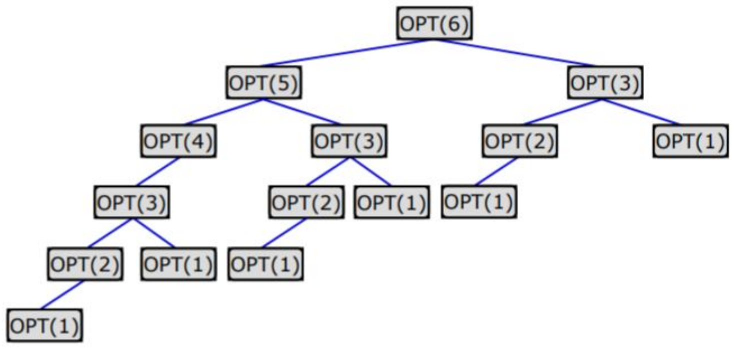
Example: An Instance of the Problem

Recall the instance that we saw earlier



Recursive Calls

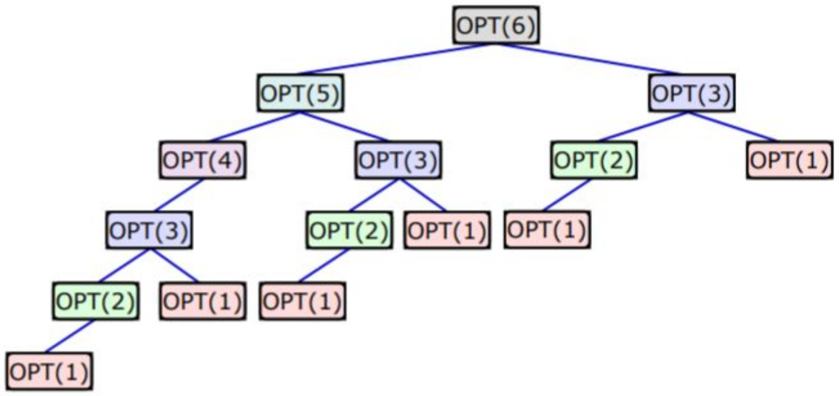
The tree of recursive calls to the function for the instance is



The number of recursive calls grows fast
→ Worse-case: exponentially!

Repeated Calls

Note: Repeated calls to the same subproblems:
→ There are only “n + 1” problems to solve!



Idea - Remember the results of already computed solutions

A Memoized Recursive Algorithm

The technique of storing previously computed values
→ Memoization
A memoized version of the recursive function:

```
function OptM(j)
    if j == 0 then
        return 0
    else if M[j] not Empty then
        return M[j]
    else
        M[j] = max(v[j] + OptR(p[j]), OptR(j-1))
        return M[j]
    end if
```

Running Time of the Memoized Algorithm

(6.4) Running Time - The running time of OptM(n) is O(n), assuming that the requests are sorted by ascending finish times.

Proof of (6.4)

We will use the array M to show the progress of the algorithm:
Each call to the recursive function

- Takes constant time plus the time for the recursive calls
- Generates two recursive calls
- Fills in one value in M

But there are only n + 1 entries in M
→ There can be at most n + 1 calls to the function!
The running time of the algorithm is now O(n), as stated.

Computing a Solution

- **Note** - We have only computed the value of the optimal solution.
- **Next** - Use this to retrieve the intervals included in the optimal solution.

We will use the values of the solutions to the subproblems.
→ The array M
Note that we know from (6.2) how the optimal choice was made in each step:

$$\mathbf{j \in O \text{ if and only if } v_j + OPT(p(j)) \geq OPT(j-1)}$$

→ Trace back through M to find the solution

A Recursive Algorithm

The following recursive algorithm uses the array M to compute the solution:

```
function FindSolution(j)
    if j == 0 then
        return ∅
    else
        if v[j] + M[p[j]] >= M[j-1] then
            return FindSolution(p[j]) ∪ j
        else
            return FindSolution(j-1)
        end if
    end if
```

Analyzing the Algorithm

(6.5) Running Time - Given the array M of the optimal values of the subproblems, FindSolution returns an optimal solution in O(n) time.

Proof:

- The function makes only one recursive call
  → to a smaller instance of the problem

→ It makes a total of O(n) calls
  → each call only takes constant time

The running time is therefore O(n).

Solving the Problem

- We solved the problem by formulating a solution to the problem recursively

→ Based on solutions to subproblems

- We realized that the same subproblems were solved repeatedly

→ Store the solutions to solved subproblems

- We ended up with a linear time solution

→ Solve backwards using recursion

Idea - Reverse the problem
→ Solve it the other way around using iteration

An Iterative Approach

The following iterative algorithm fills the array M starting from 0:

```
function OptI(j)
    M[0] = 0
    for i from 1 to n
        M[j] = max(v[j] + M[p[j], M[j-1]])
    end for
```

Once we get to n, we are done!

We can prove that the algorithm has O(n) running time
→ In each step one entry of M is filled in

Dynamic Programming

In order to use **Dynamic Programming**
→ The problem must be based on subproblems.

The subproblems must in turn have certain properties:

- There is only a polynomial number of subproblems
- The subproblems can be ordered from smallest to largest
- The solution to the original problem can be easily computed from the solutions to the subproblems

→ Recurrence relation

Lecture 18 - Dynamic Programming: Segmented Least Squares

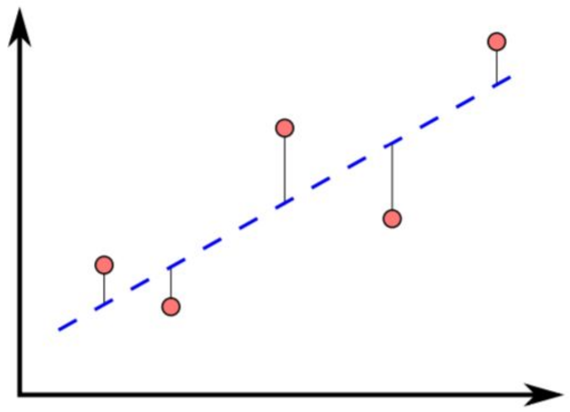
Tues. Oct. 29, 2019

Estimation from Noisy Data

Data - We have a set P of "n" points

The points are denoted as $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Suppose $x_1 < x_2 < \ldots < x_n$



Goal - Find the line that best fits the data

Least Square Estimation

Line - The line L is defined by q = ax + bn

Error - The error of L w.r.t (with respect to) P is the sum of the squared distances from the points to the line

$$e(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

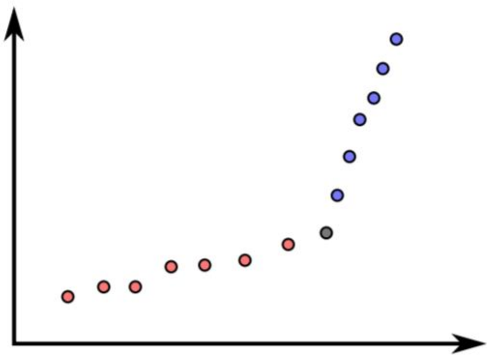
Best Fit - The best line of fit in the least squares sense is the line L that minimizes e(L, P)

- The soln. to the prob. is given by

$$a = \frac{n\sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2} \quad \text{and} \quad b = \frac{\sum_i y_i - a\sum_i x_i}{n}$$

Fitting Two Lines

- What if the pts. is best described not by one line but by two?

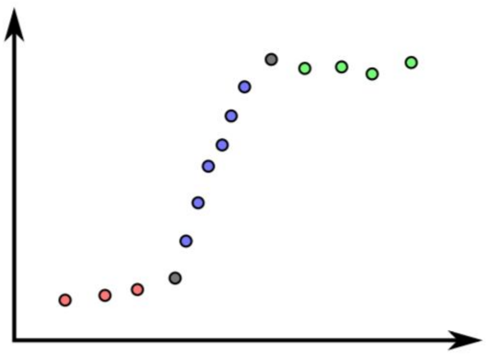


Idea - Split the set in two & est. 2 lines:
Choose the segmentation that minimizes the sum of the 2 errors

$$e(L_1, P) + e(L_2, P)$$

Fitting an Arbitrary Set of Lines

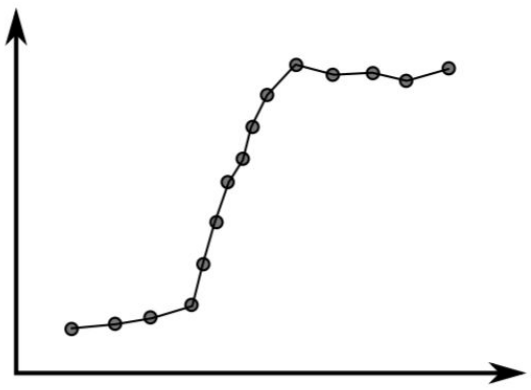
- But, what if it is 3 line and not 2?



Problem - We have hard-coded the # of lines into the algorithm
New Approach - Fit an arbitrary set of lines: Chose the segmentation that minimizes the sum of the errors

Trivial Solution

Problem - The trivial solution of “n” lines between “n” points has error 0



→ We need a better problem formulation!

Segmented Least Squares

Data - Consider the set P of “n” points

The points are denoted $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$

Suppose $x_1 < x_2 < \ldots < x_n$

Approach:

- Partition P into some number of segments S
  - A segment S is a subset of P that represents a contiguous set of x-coordinates: $\{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\}$ for some indices $i <= j$
- For each segment S
  - Compute the line that minimizes the least squares error

→ How do we choose the optimal partition?

Optimal Partition

Penalty of a Partition - The penalty of a partition is defined to be the sum of the following terms
i. The number of segments multiplied with a given constant $C > 0$
ii. For each segment, the error of the optimal line for that segment

Goal - Find the partition of minimum penalty

- Increase the # of segments: Segments penalty ↗ Line errors ↘
- Decreases the # of segments: Segments penalty ↘ Line errors ↗

Dynamic Programming

There is an exponential number of possible partitions of P.

To solve the prob. using Dynamic Programming we need

- A formulation of the orig. prob. using subproblems

→ A polynomial # of subprobs.

- A formulation of the solns. to the subprobs.

→ A formulation of the solns. to the subprobs.

- Solns. to the subprobs.

→ A soln. to the orig. prob.

Subproblems

Observation - The last point p_n belongs to single parents in the optimal partition
The last segment starts at some point p_i
$\{p_i, p_{i+1}, \dots, p_{n-1}, p_n\}$

- If the index "i" is known:
- Remove the points that belong to the last segment
    - $p_i, p_{i+1}, \dots, p_{n-1}, p_n$
- Solve the problem on the remaining
    - $p_1, p_2, \dots, p_{i-2}, p_{i-1}$

OPT(i)	The optimal solution for the points $p_1, \dots, p_{i-1}$
$e_{i,j}$	The minimum error of any line on $p_i, p_{i+1}, \dots, p_j$

Optimal Solution

(6.6) Value of the Optimal Solution - If the last segment of the optimal partition is $p_i, \dots, p_n$. then the value of the optimal solution is

$$\mathrm{OPT}(n) = e_{i,n} + C + \mathrm{OPT}(i-1)$$

Now consider the subproblem of the points $p_1, \dots, p_j$:

- We need to find the best way to define the final segment $p_1, \dots, p_j$
  → At cost $e_{i,j} + C$
- Solve the smaller subproblem on $p_1, \dots, p_{i-1}$
  → At cost OPT(i - 1)

→ We have our recurrence!

Recurrence

(6.7) Recurrence - For the subproblem on the points $p_1, \dots, p_j$,

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} \left(e_{i,j} + C + \mathrm{OPT}(i-1)\right)$$

and the segment $p_1, \dots, p_j$ is used in an optimum solution for the subproblem if and only if the minimum is obtained using index "i"

Preparation Compute the least squares errors $e_{i,j}$ for all segments $p_1, \dots, p_j$

→ Use the recurrence to build up the solutions OPT(i) in order of increasing i

Algorithm

```
function Segmented_Least_Squares(n)
    Allocate M[0 ... n]
    M[0] = 0
    for all pairs i ≤ j
        compute ei,j for the segment pi , . . . , pj
    end for
    for j = 1 to n
        compute M[j] = OPT(j) using (6.7)
    end for
    return M[n]
end function
```

Correctness of the Algorithm

The correctness of the algorithm can be proved directly by induction → The induction step is given by (6.7)

Optimal Partition

- The algorithm only computes the optimal value
  → It does not give the optimal solution
- Use the same strategy as for Weighted Interval Scheduling
  → Trace back through M to compute the solution

```
function Find_Partition(j)
    if j = 0
        return ∅
    else
        find an i that minimizes ei,j + C+ M[i-1]
        return {pi , ..., pj} ∪ Find_Partition(i-1)
    end if
end function
```

Analyzing the Algorithm

The running time of the algorithm has two parts:

- Computing the least squares errors $e_{i,j}$
  - There are $O(n^2)$ pairs
  - The least square error for a line takes O(n) time
    → A total of $O(n^3)$ time

- Computing the values of M
    - There are O(n) values to compute
    - Each M[j] takes O(n) time
      → A total of $O(n^2)$ time

Computing the Least Squares Errors

The running time is

- Dominated by the cost to compute the least squares errors
- Only $O(n^2)$ once the least errors are computed

Idea - Use an incremental strategy to reduce the time

- First compute $e_{i,j}$ for
    - All pairs of points: j - i = 1
    - All groups of three points j - 1 = 2
- Now add a constant number of terms to each one of the sums in the formulas for "a" and"b"
  → Use the sums from $e_{i,j-1}$ to compute $e_{i,j}$ in constant time

Analyzing the Algorithm

Optimizing the computation of the $e_{i,j}$, the optimal value can be computed in $O(n^2)$ time
→ What about finding the optimal solution?

```
function Find_Partition(j)
    if j = 0
        return ∅
    else
        find an i that minimizes ei,j + C+ M[i-1]
        return {pi , ..., pj} ∪ Find_Partition(i-1)
    end if
end function
```

- At instance n
  → Compare "n - 1" diff. penalties
- One recursive call to a smaller instance
  → A total of at most "n" calls

→ Running time $O(n^2)$

Lecture 19 - Dynamic Programming: Subset Sums & Knapsacks

Thurs. Oct. 31, 2019

Yet Another Scheduling Problem

We dispose of one **resource** and have multiple requests:

- **request** - require processing time
- **rule** - the resource is available during a given time W
- **goal** - maximize the total use of the resource

We have n request labeled 1, … n
durations w_i

Selection a subset of the request such that $\sum_{i \in S} w_i \leq W$

$\sum_{i \in S} w_i$ is maximal

Subset Sums & Knapsacks

- **Context** - Imagine a knapsack with a set of items with associated weights w_i
- **Goal** - Pack as much weight as possible in the knapsack without exceeding the capacity → the subset sum problem

- **Variant** - Imagine a knapsack with a capacity W and a set of items with associated weight w_i and values v_i
- **Goal** - Pack as much value as possible in the knapsack without exceeding the capacity → The Knapsack Problem

Greedy Attempt

The problem reminds us of other scheduling problems
→ Look for a Greedy Algorithm!
Idea - Sort the items and select in this order until the total weight exceeds W.

- Sort by decreasing weight
  - Counterexample: $w_1 = \frac{w}{2} + 1,\ w_2 = \frac{w}{2},\ w_3 = \frac{w}{2}$
  - Solution: w_1
  - Optimal Solution: $w_2 \textit{ and } w_3$
- Sort by increasing weight
  - Counterexample: $w_2 = \frac{w}{2}\ ,\ w_2 = \frac{w}{2},\ w_3 = 1$
  - Solution: w_2
  - Optimal solution: $w_2 \textit{ and } w_3$

Dynamic Programming

There is no known greedy algorithm that solves the problem.
→ Let's go for Dynamic Programming!
Once again: To use Dynamic Programming we need

- A formulation of the original problem using subproblems
  → A polynomial number of subproblems

- A formulation of the solutions to the subproblems
  → Use a recurrence
- Solutions to the subproblems
  → A solution to the original problem

A First Attempt

- O_i Optimal solution to the problem on {1, 2, , … i}
- OPT(i) Value of the optimal solution O_i
  → define OPT(0) = 0

Consider the last item n: We have 2 possibilities

- $n \notin S$
  - OPT(n) = OPT(n - 1)
    → Refer to a solution to a smaller instance of the problem
- $n \in S$
  - OPT(n) = w_n + … ?

  → The available capacity has changed!

A Larger Set of Subproblems

When an item is selected
→ The available capacity decreases by its weight
O_i, w Optimal solution to the problem on {1, 2, … , i}
OPT(i, w) Value of the optimal solution $O_{i,w}$
→ define OPT(0, w) = 0 for all w

- We can now write

$$\text{OPT}(i, w) = \max_S \sum_{j \in S} w_j$$

- for all subsets $S \subseteq \{1, \ldots, i\}$ that satisfy

$$\sum_{j \in S} w_j \leq w$$

In Search of a Recurrence

Consider the last item n: We have the 2 possibilities

- $n \notin S$
  - We can ignore the item
    → OPT(n, w) = OPT(n - 1, w)
- $n \in S$
  - Add the weight and remove it from capacity

→ $OPT(n, w) = w_n + OPT(n - 1, W - w_n)$

How do we know if “n is in the optimal solution?

- If $w_n > W$
  → Item “n” doesn’t fit: $n \notin S$
- If $w_n \leq W$
  → Choose the possibility that maximizes the value

Recurrence

The strategy can be resumed by the following recurrence:

(6.8) Recurrence - If w < wi then OPT(i, w) = OPT(i − 1, w). Otherwise
OPT(i, w) = max (OPT(i − 1, w), wi + OPT(i − 1, w − wi))

- We can imagine a 2D table that contains the values OPT(i, w)

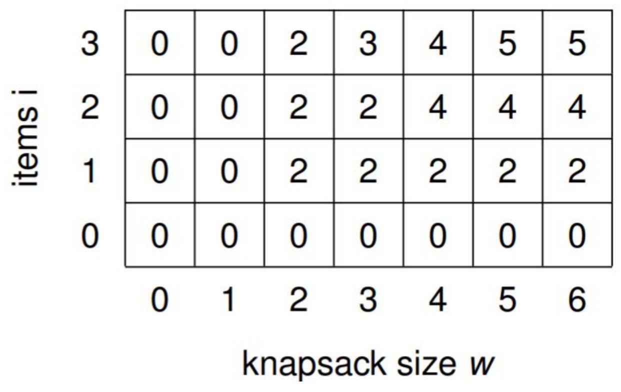
→ Design an iterative algorithm that computes the values: Each value is computed at most once!

Subset Sum Algorithm

```
function SubsetSum(n, W)
    allocate M[1...n, 1...W]
    initialize M[0,w] = 0 for each w = 1 to W
    for i = 1 to n
        for w = 1 to W
            if wi[i] > w
                M[i,w] = M[i-1,w]
            else
                M[i,w] = max(M[i-1,w], wi[i]+M[i-1,w-wi[i]])
            end if
        end for
    end for
    return M[n,W]
end function
```

An Instance of the problem

- Knapsack
  - W = 6
- Items
  - $w_1 = 2$
  - $w_2 = 2$
  - $w_3 = 3$

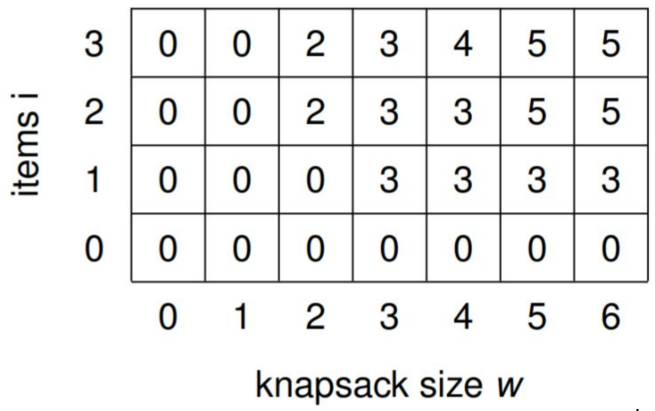


items i							
3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

knapsack size *w*

The Same Instance of the problem

- Knapsack
  - W = 6
- Items
  - $w_1 = 3$
  - $w_2 = 2$
  - $w_3 = 2$



items i							
3	0	0	2	3	4	5	5
2	0	0	2	3	3	5	5
1	0	0	0	3	3	3	3
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

knapsack size *w*

Analyzing the Algorithm

(6.9) Running Time - The `SubsetSum(n, W)` algorithm correctly computes the optimal value of the problem and runs in O(nW) time.

- **Worst Case -** All subproblems must be solved!
- **As before -** The correctness of the algorithm can be proved using induction

(6.10) Optimal Set - Given a table M of the optimal values of the subproblems, the optimal set S can be found in O(n) time.

- For each one of the"n" items, we must decide to include it or not.

the Knapsack problem

- Now assume that each item i has a **weight** w_i and a **value** v_i:
- In the Knapsack Problem we define the optimal value as

$$\mathrm{OPT}(i, w) = \max_{S} \sum_{j \in S} v_j$$

- for all subsets S ⊆ {1, . . . , i} that satisfy

$$\sum_{j \in S} w_j \le w$$

(6.11) Recurrence - If $w < w_i$ then OPT(i, w) = OPT(i − 1, w). Otherwise OPT(i, w) = max (OPT(i − 1, w), v_i + OPT(i − 1, w − w_i))

- The problem can be solved using (almost) the same algorithm.

(6.12) Running Time - The Knapsack Problem can be solved in O(nW) time.

(Lecture 20) - Exam 2

Tues. Nov. 5, 2019

Lecture 20 - Shortest Paths in a Graph

Thurs. Nov. 7, 2019

Shortest Paths in a Graph

- Weighted graph - Assigned a numeric value to each edge → Negative weights allowed
- Length of a path - The sum of the weights of all edges in the path

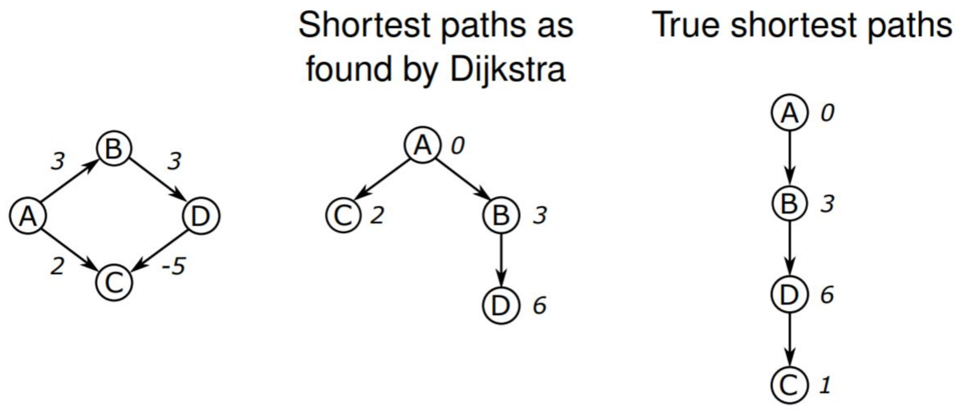
setup - Consider a weighted, directed graph.
problem - What is the shortest path from a given start node to each one of the nodes in the graph?

- Dijkstra's algorithm fails when negative weights are allowed

→ Use Dynamic Programming instead

Dijkstra's Algorithm - Negative Weights

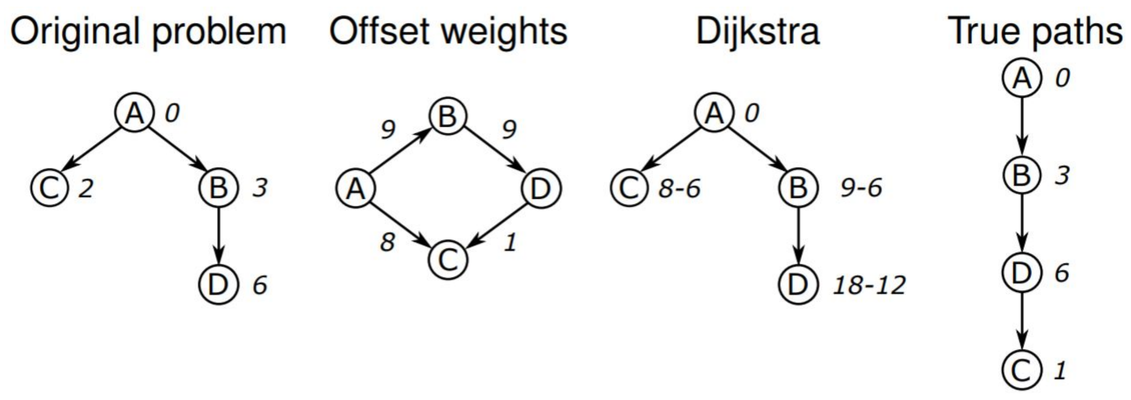
- Consider an instance of the Shortest Paths problem:



Greedy - Dijkstra's algorithm never questions the choices made.

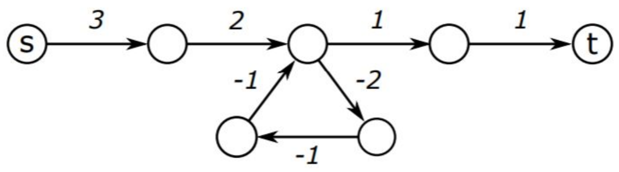
Dijkstra's Algorithm - Offset

Problem - Dijkstra's algorithm is not guaranteed to find an optimal solution if we allow negative weights
Idea - Introduce an offset: Add a constant value to each one of the weights, so that all weights become positive



Negative Cycles

Consider the following graph

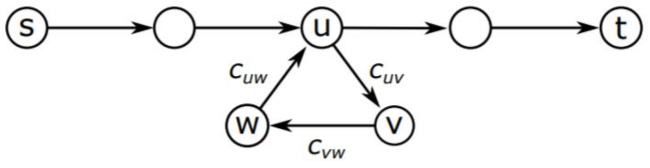


Negative cycle - we can find paths from “s” to “t” of arbitrarily negative cost
→ Consider only graphs with no negative cycles

Dynamic Programming

We start by trying to delimit the problem:

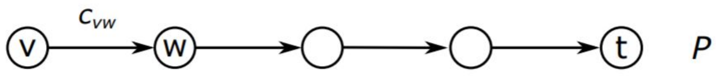
(6.22) Maximum Number of Edges - If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e. does not repeat nodes), and hence has at most n − 1 edges.



- if a node u is repeated, simply remove the cycle to obtain a path with no greater cost and fewer edges.

Subproblems

- OPT(i, v) - Minimum cost of the path from “v” to “t” using at most “i” edges
- P - Optimal path representing OPT(i, v)



- If the path P uses at most “i - 1” edges

    - $OPT(i, v) = OPT(i - 1, v)$
      → Refer to a solution to a smaller instance of the problem
- If the path P uses “i” edges and the first edge is (v, w)
    - $OPT(i, v) = c_{vw} + OPT(i - 1, w)$
      → Choose the node “w” that minimizes the total cost

Recurrence

(6.23) Recurrence - If i > 0 then

$$OPT(i, v) = \min(OPT(i - 1, v), \min(w \in V) (cvw + OPT(i - 1, w)))$$

- Our original problem is to compute OPT(n - 1, s)

→ We have (n - 1) x n subproblems

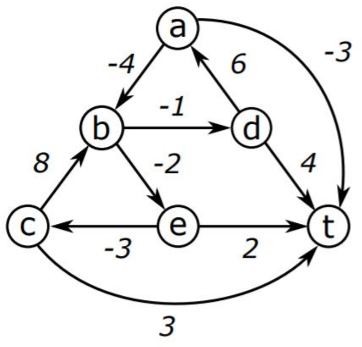
Initialize:

- OPT(i, t) = 0 for all path lengths i
- OPT(0, v) = ∞ for all nodes v

Bellman=Ford Algorithm

```
function ShortestPath(G, s, t)
    allocate M[0...n - 1: V]
    initialize M[0,t] = 0
    initialize M[0,v] = ∞ for each v ∈ V
    for i = 1 to n - 1
        for v ∈ V
            for w ∈ V
                m[w] = c[v,w] + M[i-1,w])
            end for
            M[i,v] = min(M[i-1,v], m)
        end for
    end for
    return M[n-1,s]
end function
```

An Instance of the Problem



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Running Time of the Bellman-Ford Algorithm

(6.24) Running Time - The `ShortestPath` algorithm correctly computes the minimum cost of the shortest path from s to t in any graph with no negative cycles. It runs in O(n^3) time.

- Correctness
  → Follows by induction on (6.23)
- Running time
  - The 2D array M has n^2 entries
  - Each entry takes "n" time to compute

  → A total running time of O(n^3)

A Better Estimate of the Running Time

- Consider G = (V, E) with n = |V| and m = |E|

- In the algorithm, we iterated over all nodes to compute

$$\min_{w \in V} \left(c_{vw} + \text{OPT}(i-1, w) \right)$$

- However, we only need to consider the nodes "w" such that (v, w) ∈ E

→ There are n_v such nodes
- The time to compute one entry M[i,v]
  → O(n_v)
- The total running time of the algorithm is now
  → $\mathcal{O}\left(n \sum_{v \in V} n_v \right)$
- n_v denotes the number of edges leaving "v"
- Each edge leavers exactly one of the nodes in "V"

- Each edges is counted exactly once in the expression:

$$\sum_{v \in V} n_v = m$$

- Now the running time is given by

$$\mathcal{O}\left(n \sum_{v \in V} n_v\right) = \mathcal{O}(mn)$$

Worse-case - The graph is dense, with m = n(n - 1) → Running time O(mn)

(6.25) Running Time - The `ShortestPath` algorithm can be implemented in O(mn) time

- Compare with the greedy algorithm: Dijkstra

→ Running time O(n log m)

Improving the Space Requirements

- The Bellman-Ford algorithm uses a 2D array M[i,v]
- Only values from the last iteration are used to compute "i"

→ Replace with a 1D array M[v]

- Previously, we have used the arrays of optimal values to construct the solution

→ How can we construct the shortest paths?

Lecture 21 - The Maximum-Flow Problem

Tues. Nov. 12, 2019

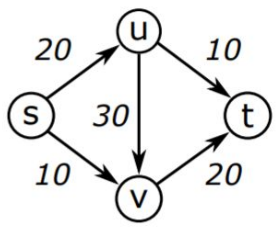
Flow Networks

A flow network is a directed graph

G = (V, E)

with the following features

- Capacity
  - Each edge e has a capacity denoted $c_e > 0$
- Source
  - Single source node s ∈ V
- Sink
  - Single sink node t ∈ V



Additional Assumptions

We will make a few assumptions, in order to simplify:

- Source

→ No edges enter the source
- Sink
  → No edges leave the sink
- Capacity
  → All capacities are integers

→ Simplifies the reasoning
Preserves the essentials of the problem

Flow

- An s-t-flow is a real function “f” that maps each edge “e” to a non-negative real number
- The value f(e) represents the flow carried by edge “e”
- A flow “f” must satisfy the following two properties

- Capacity conditions
  - For each $e \in E$, we have $0 \leq f(e) \leq c_e$
- Conservation conditions
  - For each node v other than s and t, we have

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

Notations

Define

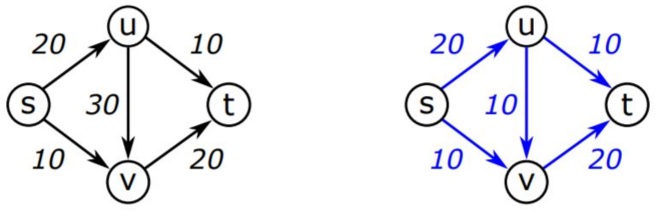
$$f^{out}(v) = \sum_{e \text{ out of } v} f(e)$$

$$f^{in}(v) = \sum_{e \text{ into } v} f(e)$$

- The source generates flow $f^{out}(s)$
- The sink absorbs flow $f^{in}(t)$
- The value of the flow $v(f) = f^{out}(s) = f^{in}(t)$

The Maximum Flow Problem

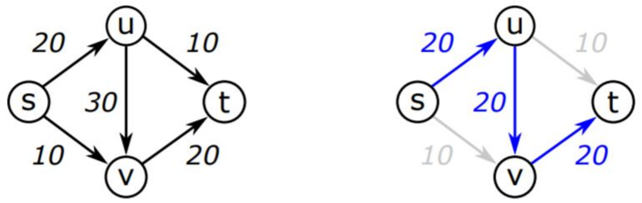
- **Goal** - Find the maximum flow in a network
- **Rules** - The flow must respect the capacity and conservation conditions



→ No known Dynamic Programming approach
→ No known Greedy Algorithm

A First Approach

Idea - Find a s-t-path and push a flow up to the capacity limits, respecting the two conditions.



Problem - Maximum is not achieved and there is no available s-t-path to push more flow.

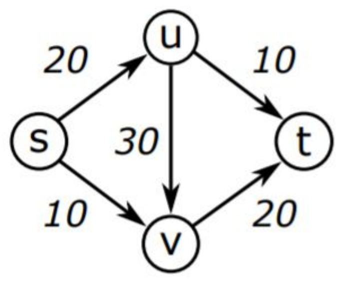
Residual Graph

Given a flow network G and a flow "f" on G, we define the following graph G_f on G with respect to f as follows:

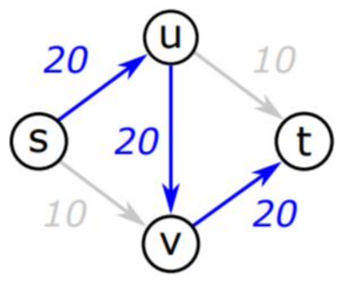
- G_f has the same nodes as G
- For an edge e = (u, v) in G with f(e) < ce
  → Include e in G_f with capacity ce − f(e)
  forward edges
- For an edge e = (u, v) in G with f(e) > 0
  → Include e 0 = (v, u) in G_f with capacity f(e)
  backward edges

We refer to the capacities of G_f as residual capacities

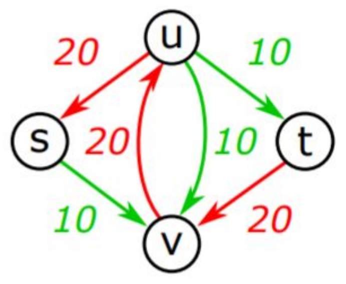
- For an edge e = (u, v) in G with f(e) < ce
  → Include e in Gf with capacity ce − f(e)
- For an edge e = (u, v) in G with f(e) > 0
  → Include e 0 = (v, u) in G_f with capacity f(e)



flow network



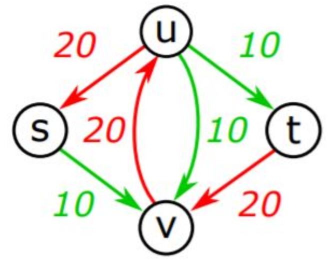
flow on G



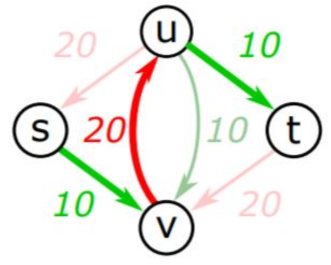
residual graph G_f
forward, backward

Augmenting Paths

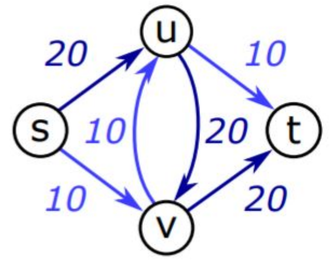
- **Idea** - Find an s-t-path in the residual graph and push an additional flow within the limits of the capacity
- **Path** - Let P be a simple s-t-path in G_f .
- **Bottleneck** - The minimum residual capacity of any edge on P.



residual graph G_f
forward, backward



s-t-path P
bottleneck: 10



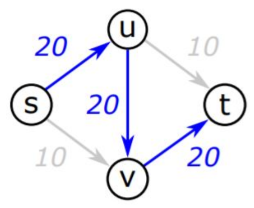
original flow: 20
additional flow: 10

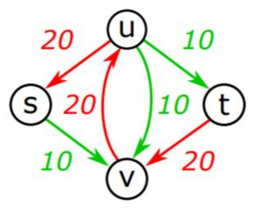
Augment Algorithm

```
function augment(f, P)
      Let b = bottleneck(P, f)
      for each edge (u, v) ∈ P
            if e = (u, v) is a forward edge then
                  increase f(e) in G by b
            else e = (u, v) is a backward edge then
                  Let e’ = (v, u)
                  decrease f(e’) in G by b
            end if
      end for
```

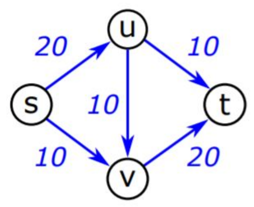
```
        return f
end function
```

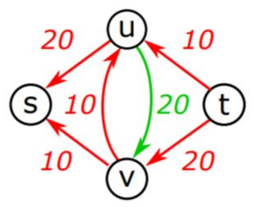
Augmenting Paths





flow f: 10 residual graph G_f $\longrightarrow$ $(s, v), (v, u), (u, t)$





flow f': 30 residual graph $G_{f'}$ $\longrightarrow$ No s-t-path

Correctness of Augment

(7.1) Correctness of Augment - f' is a flow in G.

We must verify that f' satisfies the 2 conditions:

- Capacity
- Conservation

The flow has changed only on edges on P.

By definition of the capacities of the residual edges:

- On forward edges
  → we avoid increasing the flow above the capacity
- On backward edges
  → We avoid decreasing the flow below zero

Proof of (7.1)

Suppose that e = (u, v) is a forward edge on P

- Its residual capacity is c_e - f(e)
- Then bottleneck(P, f) ≤ c_e − f(e)

We increase the flow on e, so

$$0 \le f(e) \le f 0 (e)$$

and

$$f'(e) = f(e) + \text{bottleneck}(P,f) \le f(e) + (c_e - f(e)) = c_e$$

We have showed that

$$0 \le f'(e) \le c_e$$

Therefore, the capacity condition holds on e.

Suppose that e' = (u, v) is a backward edge on P that comes from the edge e [(v, u) in G

- Its residual capacity is f(e)
- Then bottleneck(P, f) ≤ f(e)

We decrease the flow on e, so

$$c_e \ge f(e) \le f'(e)$$

and

$$f'(e) = f(e) - \text{bottleneck}(P, f) \ge f(e) - f(e) = 0$$

We have showed that

$$0 \le f'(e) \le ce$$

Therefore, the capacity condition holds on e.

- We have shown that the capacity condition holds for all edges on the path P.
  → Also need to show that the conservation condition holds for all nodes.

Maximum Flow Algorithm

```
function Maximum_Flow(f,P)
    Initialize f(e) = 0 for all e in G
    while there is an s-t-path in the residual graph Gf
        Let P be a simple s-t-path in Gf
        f' = augment(f,P)
        Update f to be f'
        Update Gf to be Gf'
    end while
    return f
end function
```

- The algorithm is called the Ford-Fulkerson algorithm after its developers

→ Simple algorithm but does it work?

Analyzing the Algorithm

(7.2) Integer Capacities - At every intermediate stage of the Ford-Fulkerson Algorithm, the flow values {f(e)} and the residual capacities are integers

Proof:

- The statement is true initially
- Suppose the statement is true at iteration “j”
    - Since all residual capacities in G_f are integers
      → bottleneck(P, f) found in iteration “j + 1” is an integer
      → The new flow f’ will have integer values

→ The new residual capacities will have integer values

(7.3) Strictly Increasing Flow - Let f be a flow in G and let P be a simple s-t-path in G_f . Then

$$v(f 0) = v(f) + \text{bottleneck}(P, f)$$

and since bottleneck(P,f) > 0, we have

$$v(f 0) > v(f)$$

We defined the total flow as

$$v(f) = f^{out}(s)$$

We need to show that the flow on edges incident to “s” increase

Proof of (7.3)

The first edge e on P is an edge out of s in G_f.

- G has no edges entering s
  → e must be a forward edge
- The path P is simple and does not return to s.
  → This is the only edge incident to s where the flow changes
- The flow on e increases by bottleneck(P,f) > 0
  → The total new flow f 0 exceeds f by bottleneck(P,f)

Termination of the Algorithm

An upper bound for the maximum flow is given by

$$C = \sum_{e \text{ out of } s} c_e$$

(7.4) - Suppose that all capacities in the flow network are integers. Then the Ford-Fulkerson Algorithm terminates in at most C iterations of the while loop.

Proof of (7.4)

- No flow in G can have a value greater then C
  → Due to the capacity condition on edges from s
- By (7.2), the value of the flow increases in each iteration
- By (7.1), the value of the flow takes only integer values
  → Increases by at least 1 in each iteration
- C is an upper bound for the maximum flow
  The while loop can run for at most C iterations

Running Time of the Algorithm

(7.5) Running Time of the Ford-Fulkerson Algorithm - Suppose that all capacities in the flow network G are integers. Then the Ford-Fulkerson Algorithm can be implemented to run in O(mC) time.

Proof of (7.5)

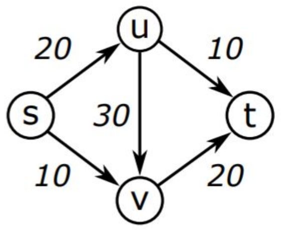
- We now that from (7.4) that the maximum number of iterations is C
- The residual graph G_f has at most 2m edges.
- We can find an s-t-path in G_f using BFS or DFS in O(m + n) time.
- Augment takes O(n) time since there are at most n − 1 edges on the path.
- We can update the residual graph within the augment function at O(n) time.

Lecture 22 - The Maximum-Flow Problem

Thurs. Nov. 14, 2019

Flow Networks

A flow network is a directed graph



with the following features

- Capacity
  - Each edge "e" has a capacity denoted c_e => 0
- Source
  - Single source node ∈ V
- Sink
  - Single sink node t ∈ V

We have yet to prove the **optimality** of the Ford-Fulkerson Algorithm

Optimality of the Ford-Fulkerson Algorithm

- **Goal** - Show that the flow returned by the
- **Idea** - Find an upper bound for the flow and show that this bound is reached

We already have a weak upper bound

$$v(f) \leq C \quad \text{where} \quad C = \sum_{e \text{ out of } s} c_e$$

We will introduce the notion of cut to formulate a tighter bound

Cuts

- **Cut** - A cut is a partition of V in two sets A and B, so that s ∈ A and t ∈ B
- **Capacity** - The capacity of a cut (A;B) is the sum of the capacities of the edges from A to B

$$c(A, B) = \sum_{e \text{ out of } A} c_e$$

Intuitively, the capacity of a cut is an upper bound for the flow.
Define:

- $f^{out}(S) = \sum_{e \text{ out of } S} f(e) \quad \text{for } S \subseteq V$
- $f^{in}(S) = \sum_{e \text{ into } S} f(e) \qquad \text{for } S \subseteq V$

Flow from A to B

(7.6) Flow from A to B - Let f be any s-t-flow and (A, B) any s-t-cut. Then

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

This is not only an upper bound but also a way to measure the flow.

If A = {s}, then

- $f^{in}(A) = f^{in}(s) = 0$ No edges enter the source
- $f^{out}(A) = f^{out}(s) = v(f)$ By definition of the value of the flow

Proof of (7.6)

- The source has no entering edges, by definition
  → $v(f) = f^{out}(s) - f^{in}(s)$
- For all internal nodes v, by the conservation condition
  → $f^{out}(v) - f^{in}(v) = 0$

Now we have v(f) = $\sum_{v \in A} \left(f^{out}(v) - f^{in}(v)\right)$.

- The edge e has both ends in A
  → f(e) appears twice in the sum and cancels out
- The edge e has its tail in A and its head in B
  → Contribution: $+\sum_{e \text{ out of } A} f(e) = f^{out}(A)$
- The edge e has its end in A and its tail in B

→ Contribution: $-\sum_{e \text{ into } A} f(e) = f^{in}(A)$

An Alternative Statement

(A, B) is a cut → The edges out of A are precisely the edges into B:

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

(7.7) Flow from A to B - Let f by any s-t-flow and (A, B) any s-t-cut. Then

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

If A = V - {t} and B = {t}, then

● $f^{out}(B) = f^{out}(t) = 0$	No edges leave the sink
● $f^{in}(B) = f^{in}(t) = v(f)$	Alternative definition

Flows and Cuts

(7.8) Upper Bound - Let f by any s-t-flow and (A,B) any s-t-cut. Then v(f) ≤ c (A, B).

Proof:

$v(f) = f^{out}(A) - f^{in}(A)$	by (7.6)
$\leq f^{out}(A)$	since $f^{in}(A) > 0$
$= \sum_{e \text{ out of } A} f(e)$	
$\leq \sum_{e \text{ out of } A} f(e)$	by capacity conditions
$= c(A,B)$	

Why Are We Doing This Again?

Let f be any s-t-flow and (A, B) any s-t-cut. Then

● $v(f) = f^{out}(A) - f^{in}(A)$	(7.6)
● $v(f) \leq c(A, B)$	(7.8)

The equality seems to make (7.6) a stronger statement.
However, the LHS of (7.8) is independent of any particular flow.

→ If we find any s-t-cut in G with a capacity of c*:
No s-t-flow f in G can have a value v(f) higher than c*.

→ If we find any s-t-flow in G with a value of v*:
No s-t-cut (A;B) in G can have a capacity c lower than v*.

Analyzing the Ford Fulkerson Algorithm

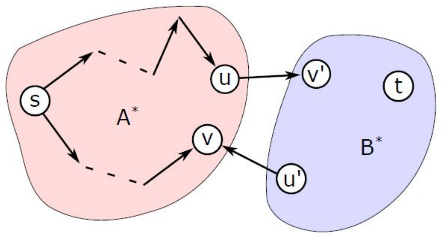
Denote the flow returned by the Ford-Fulkerson Algorithm f:

- Show that v(f) = $\max_f$ v(f)
- Show that there is a cut with capacity equal to v(f)

(7.9) Maximum Flow Equals Minimum Cut - If f is an s-t-flow such that there is no s-t-path in the residual graph Gf , then there is an s-t-cut (A*, B*) in G for which v(f) = c(A*, B*). Consequently, f has the maximum value of any flow in G, and (A*, B*) has the minimum capacity of any s-t-cut in G.

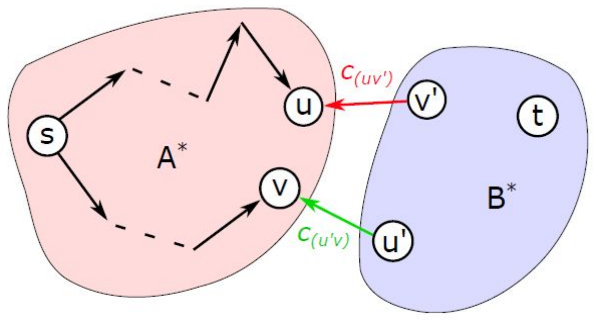
Proof of (7.9)

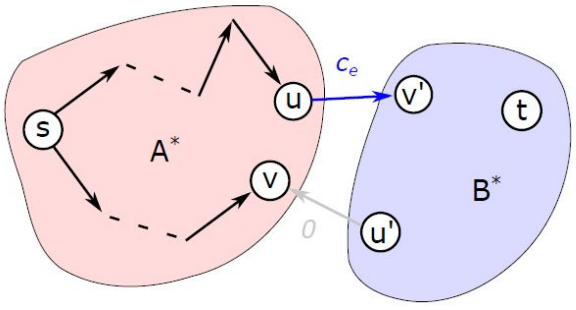
- Let A* denote all nodes v in G such that there is an s-v-path in G_f
- Let B* = V - A*



(A*, B*) is a cut since

- V = A* U B*
- s ∈ A* by construction
- t ∉ A* since there is no s-t-path in G_f → t ∈ B*

 s u v' t A* B* v u' $c_{(uv')}$ $c_{(u'v)}$ residual graph Gf	• e = (u, v') saturated → f(e) = c_e Or there would be a forward edge (u, v') in Gf

 graph G	• e' = (u', v) unused → f(e') = 0 Or there would be a backward edge (v, u') in Gf

- All edges out of A* are saturated
- All edges into A* are unused

By (7.6), we can now write

v(f) = fout(A*) - f in (A*)

$$= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e)$$

$$= \sum_{e \text{ out of } A^*} c_e - 0$$

= c(A*, B*)

Optimality of the Ford-Fulkerson Algorithm

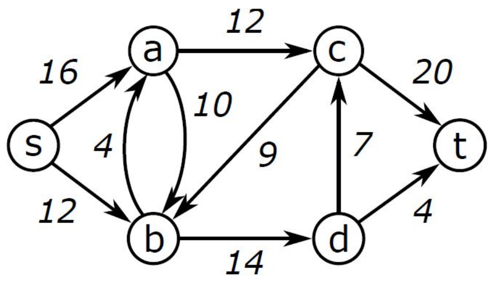
Termination of the Ford-Fulkerson Algorithm:
→ There is no s-t-path in the residual graph.

(7.10) Optimality of the Ford-Fulkerson Algorithm - The flow f returned by the Ford-Fulkerson Algorithm is a maximum flow.

(7.11) Computation of a Minimum Cut - Given a flow of maximum value, we can compute an s-t-cut of minimum capacity in O(m) time.

Construct the minimum cut by a BFD or a DFS in the residual graph, starting at s. All nodes reachable from "s" to A*.

One Instance of the Problem



Lecture 23 - NP & Computational Intractability: Polynomial-Time Reductions

---

Tues. Nov. 19, 2019

Computational Intractability

Computational Tractability - An algorithm is efficient if it has polynomial running time

- So far, we have focused on problems that can be solved by efficient algorithms:
- Out next challenge is to deal with problems that we cannot solve, that maybe cannot be solved, by efficient algorithms

→ Classify problems according to difficulty

Computationally Hard Problems

- **Goal** - Explore the space of computationally hard problems
- **Idea** - Compare the relative difficulty of problems

Problem X is at least as hard as Problem Y

Black Box - Suppose we have a black box capable of solving X. If we write down the input for an instance of X, then the black box will return the correct answer in a single step.

Polynomial-Time Reductions

Polynomial-Time Reductions - If an arbitrary instance of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X. Then

$$Y \leq_P X$$

- Y is polynomial-time reducible to X
- X is at least as hard as Y

Note: We have given up on the polynomial time: We accept to spend polynomial time on the reduction

Polynomial-Time Solution

- If a polynomial-time algorithm that solves problem X exists

→ Replace the black box with this algorithm.

- A polynomial number of steps, plus a polynomial number of calls to a polynomial algorithm

→ We have a polynomial algorithm that solves Y.

(8.1) Existence of a Polynomial-Time Solution - Suppose Y $\leq_P$ X. If X can be solved in polynomial time, the Y can be solved in polynomial time.

No Polynomial-Time Solution

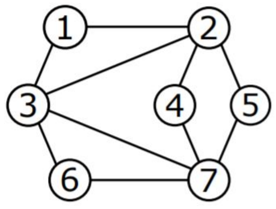
We will have more use for the contrapositive:

(8.2) Non-Existence of a Polynomial-Time Solution - Suppose Y $\leq_P$ X. If Y cannot be solved in polynomial time, the X cannot be solved in polynomial time.

- We can reduce Y to X in polynomial time
- There is no polynomial time solution to Y

If we did solve X in polynomial time, we would also solve Y in polynomial time
→ Contradiction!

Independent Set



Given a graph G = (V, E), a set S ⊆ V is independent if no two nodes in S are connected by edges.

- **Goal** - Find a maximum independent set
- **Solution** - S = {1, 4, 5, 6}

We will formulate problems as questions with yes/no answers:

Independent Set - Given a graph G and a number k, does G contain an independent set of size at least k?

Optimization vs Decision

The problem can be formulated in two versions:

- *Optimization*

→ Find the maximum size of an independent set.

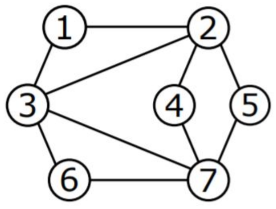
- *Decision*

→ Decide, yes or no, whether there is an independent set of size at least k?

If we can solve the decision problem in polynomial time, we can solve the optimization problem in polynomial time

→ Use binary search to find the maximum value of k: Solve the decision problem O(log n) times

Vertex Cover



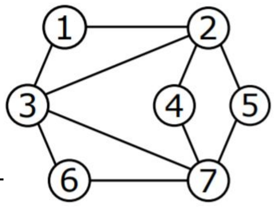
Given a graph G = (V, E), S ⊆ V is a vertex cover if every edge e ∈ E has at least one edge in S.

- **Goal** - Find a minimum vertex cover
- **Solution** - S = {2, 3, 7}

We will formulate problems as questions with yes/no answers:

Vertex - Given a graph G and a number k, does G contain a vertex cover of size at most k?

Independent Set vs Vertex Cover



- **Equivalence** - The two problems are essentially the same.

Max Independent Set	{1, 4, 5, 6}
Min Vertex Cover	{2, 3, 7}

(8.3) Independent Set and Vertex Cover - Let G = (V, E) be a graph.
Then S is an independent set if and only if its complement V − S is a vertex cover.

Proof of (8.3)

Suppose S is an independent set:

- Consider an arbitrary edge e = (u, v)
- We cannot have u ∈ S and v ∈ S
- Then u ∈ V − S or v ∈ V − S
- All edges have at least one edge in v − S

→ V − S is a vertex cover

Suppose V − S is a vertex cover:

- Consider any two nodes u, v ∈ S
- If there was an edge e = (u, v) then neither end of e would be in V − S
- Contradicts the fact that V − S is a vertex cover
- No two nodes in S are joined by an edge

→ S is an independent set

Equally Hard Problems

(8.4) - Independent Set $\leq_P$ Vertex Cover

If we have a black box that solves Vertex Cover, we can decide whether G has an independent set of size k by asking for a vertex cover of size at most n − k.

(8.5) - Vertex Cover $\leq_P$ Independent Set

Set Cover

Given a set U of elements, a collection S1, . . . , Sm of subsets of U and a number k, does there exist a collection of at most k of these sets, whose union is equal to all of U?

- The white circles represent the set U
- The colored blocks represent the sets S1, . . . , S7
- Smallest set cover: pink blocks → k = 3

What is the connection between Vertex Cover and Set Cover?

Vertex Cover vs Set Cover

- **Vertex Cover** - Cover edges of a graph using sets of edges incident to vertices
- **Set Cover** - Cover an arbitrary set using arbitrary subsets

→ Vertex Cover is a special case of Set Cove

Formally, this can be written

(8.6) - Vertex Cover $\leq_P$ Set Cover

Proof of (8.6)

- **Vertex Cover** - Consider an instance on G = (V, E): Cover the edges in E
- **Set Cover** - Choose the ground set to be U = E Si consists of all edges incident to vertex i ∈ V

We have two equivalent problems:

S can be covered if and only if E can be covered

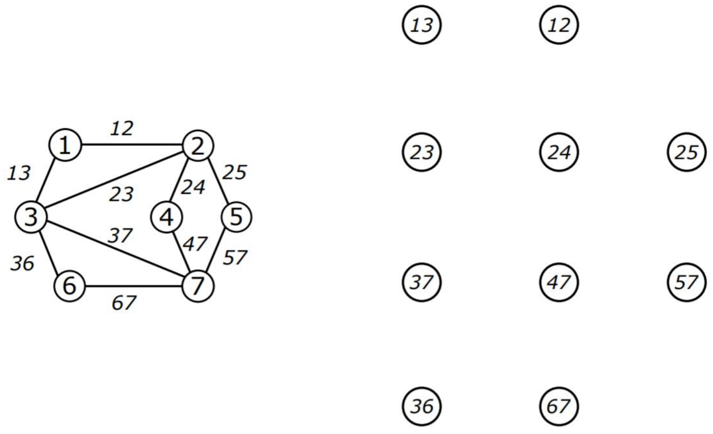
- Suppose there is a set cover of size at most k

→ Show that there must be a vertex cover of size at most k

- Suppose there is a vertex cover of size at most k

→ Show that there must be a set cover of size at most k

Understanding (8.6)



Proof of (8.6)

- U The set E of all edges in G
- S_i All edges incident to vertex i

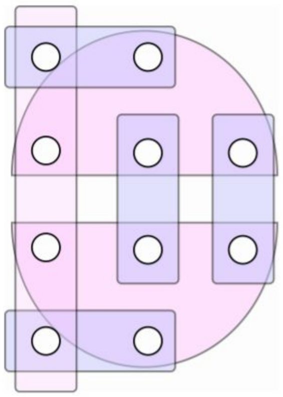
- Suppose $S_{i1}, \ldots, S_{il}$ are $l \le k$ sets that cover U
  - We defined U = E, so all edges in G are covered
  - Every edge in G is incident to one of the vertices i1, . . . , i l

→ The set {i1, . . . , i l} is a vertex cover in G of size at most k

- Suppose the set {i1, . . . , i l} is a vertex cover in G of size at most k

→ Then Si1 , . . . , Si l are l ≤ k sets that cover U

Set Packing



Given a set U of elements, a collection S1, . . . , Sm of subsets of U and a number k, does there exist a collection of at least k of these sets, with the property that no two of them overlap?

- The white circles represent the set U
- The colored blocks represent the sets S1, . . . , S7
- Largest set packing: purple blocks

→ k = 3

What is the connection between Independent Set and Set Packing?

- Independent Set and Vertex Cover are identical problems
- Vertex Cover can be generalized to Set Cover

→ Independent Set can be generalized to Set Packing

(8.7) - Independent Set $\le_P$ Set Packing

Lecture 24 - NP & Computational Intractability: The Satisfiability Problem

---

Tues. Nov. 19, 2019

Polynomial-Time Reduction

Polynomial-Time Reducible - If an arbitrary instance of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X. Then

$$Y \le_P X$$

- A reduction is an algorithm that transforms an arbitrary instance of a problem to an instance of another problem.
- The reduction algorithm does not attempt to solve the problem.

CSE 3500 Master Notes

A General Framework

- Every problem that we will encounter has similar elements: Selection

→ Something is being selected

- Requirement

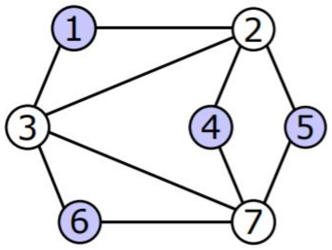
→ Something is forcing us to select a sufficient number of such things

- Restrictions

→ Something is limiting our ability to select those things

Reduction - The role of a reduction is to find a mapping between these similar elements to each other.

Independent Set



- Selection

→ We select nodes from a graph

- Requirement

→ We want to select at least k nodes

- Restrictions

→ No two selected nodes can be adjacent

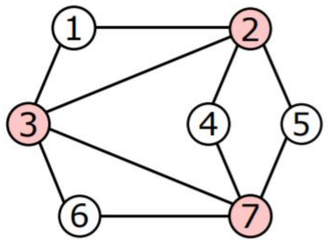
- Largest independent set

→ S = {1, 4, 5, 6}

- Decision problem with k = 4

→ Answer: Yes

Vertex Cover



- Selection

→ We select nodes from a graph

- Requirement

→ We want to select at most k nodes

- Restrictions

→ All edges in the graph must be incident to at least one of the selected nodes

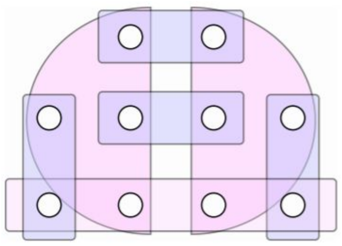
- Smallest vertex cover

→ S = {2, 3, 7}

- Decision problem with k = 3

→ Answer: Yes

Set Packing



- Selection

→ We select subsets

- Requirement

→ We want to select at least k subsets

- Restrictions

→ No two selected subsets can overlap

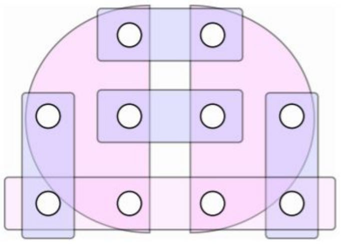
- Smallest set cover

→ the three pink blocks

- Decision problem with k = 3

→ Answer: Yes

Set Cover



- Selection

→ We select subsets

- Requirement

→ We want to select at most k subsets

- Restrictions

→ The union of the selected subsets must be equal to the ground set

- Smallest set cover

→ the three pink blocks

- Decision problem with k = 3

→ Answer: Yes

Boolean Clauses

Suppose that X is a set of n Boolean variables $x_1, \ldots, x_n$.

- **Term -** A term is a variable $x_i \in X$ or its negation x¯i
- **Clause -** A clause is a disjunction of distinct terms:

$$t1 \lor t2 \lor \cdots \lor tl$$

A truth assignment for X is a function v : X 7→ {0, 1}

- Each xi is assigned the value 0 or 1 (or true and false).
- The assignment v assigns x¯i the opposite value of xi .

An assignment satisfies a clause if it causes it to evaluate to 1.

- At least one of the terms has been assigned the value 1.

Examples

Consider the clause

$$C = x1 \lor x¯ 2 \lor x¯3 \lor x4$$

What does C evaluate to under the following assignments?

- v1 : x1 = 1, x2 = 1, x3 = 0, x4 = 1

→ 1

- v2 : x1 = 0, x2 = 1, x3 = 1, x4 = 0

→ 0

Satisfiability

An assignment v satisfies a collection of clauses C1, . . . , Ck if it causes the conjunction

$$C1 \wedge C2 \wedge \cdots \wedge Ck$$

to evaluate to 1.

- All clauses must evaluate to 1.
- v is a **satisfying assignment** with respect to C1, . . . , Ck .
- The set of clauses is **satisfiable**.

Examples

Consider

$$(x1 \vee \bar{x}2) \wedge (\bar{x}1 \vee \bar{x}3) \wedge (x2 \vee \bar{x}3)$$

Are the following assignments satisfying?

- v1 : x1 = x2 = x3 = 1

→ No:

- The second clause is not satisfied v2 : x1 = x2 = x3 = 0

→ Yes

Is there a satisfying truth assignment to

$$(x1 \vee \bar{x}2) \wedge (\bar{x}1 \vee x2) \wedge (x2 \vee \bar{x}3)$$

The Satisfiability Problem

The Satisfiability Problem - Given a set of clauses C1, . . . , Ck , over a set of variables X = {x1, . . . , xn}, does there exist a satisfying truth assignment?

- The Satisfiability Problem is often referred to as SAT.
- An equally hard problem, but better for our purposes is:

3-SAT - Given a set of clauses C1, . . . , Ck , each of length 3, over a set of variables X = {x1, . . . , xn}, does there exist a satisfying truth assignment?

3-SAT

We can identify the elements of this problem

- Selection

→ Select which of the n variables are assigned to be true

- Requirement

→ Each one of the k clauses must have at least one true term

- Restrictions

→ If xi is assigned true, then $\bar{x}$i must be false and vice versa

- **Challenge -** Find a polynomial-time reduction from 3-SAT to Independent Set.

From 3-SAT to Independent Set

- **Goal** - Find an algorithm that transforms any instance of 3-SAT to an instance of Independent Set

→ A reduction

We start with the selection:

- 3-SAT

→ Select which of the n variables are assigned to be true

- Independent Set

→ Select which of the n nodes from a graph to pick

- **Idea** - Define a node in G for each term in each one of the clauses.

Requirements:

- 3-SAT

→ Each one of the k clauses must have at least one true term

- Independent Set

→ Select at least k nodes

- **Idea** - Group the nodes into clusters of three, one cluster per clause. We will have k clusters, so one node from each cluster must be in any independent set. Put edges between nodes in the same cluster to make sure that exactly one of the three terms is selected.

Example

Consider the following instance of 3-SAT

$$a \lor \bar{b} \lor c \land (\bar{a} \lor b \lor d)$$

We transform it using the following steps:

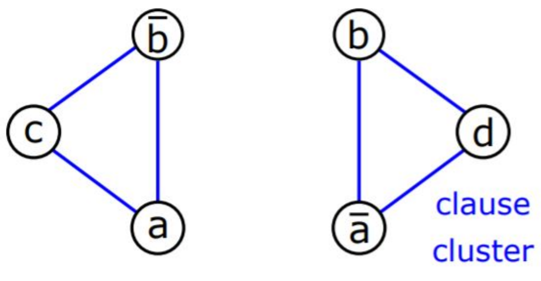
- Selection

→ Define 3k nodes

- Requirement

→ Add edges between nodes in the same cluster

- **Next Step** - enforce restrictions



From 3-SAT to Independent Set

Restrictions:

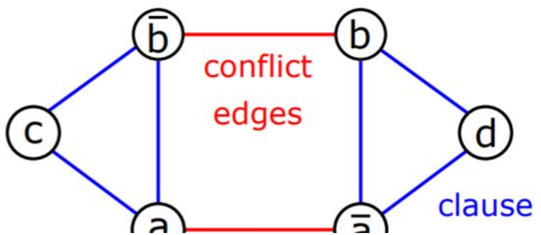
- 3-SAT

→ If x_i is assigned true, then $\bar{x}_i$ must be false and vice versa

- Independent Set

→ No two selected nodes can be adjacent

- **Idea** - Put an edge between two nodes if they correspond to a variable x_i and its negation $\bar{x}_i$



Example

$a \lor \bar{b} \lor c \land (\bar{a} \lor b \lor d)$

- Selection

−→ Define 3k nodes

- Requirement

→ Add edges between nodes in the same cluster.

- Restrictions

→ Add edges between nodes that correspond to a variable and its negation

$(a \lor b \lor c) \land b \lor \bar{c} \lor \bar{d} \land (\bar{a} \lor c \lor d) \land a \lor \bar{b} \lor d$

- a = 1
- c = 1
- $\bar{d}$ = 1

→ d = 0

- b can take any value

Reducing 3-SAT to Independent Set

(8.8) - 3-SAT ≤P Independent Set

- We have constructed an algorithm that transforms a Boolean formula to a graph G.

We need to prove correctness:

F is satisfiable if and only if G has an independent set of size k

Suppose F is satisfiable
→ Show that G has an independent set
Suppose G has an independent set
→ Show that F is satisfiable

Correctness of the Reduction

Suppose that F is satisfiable:

- Each one the clauses of F must have at least one true term

→ Select such a term from each clause

- Call S the set of nodes corresponding to these terms
- There are k clauses

→ S has size k

- No variable and its negation can be in S

→ S is an independent set of size k

Suppose that G has an independent set S of size k:

- We cannot select two nodes from the same cluster
- There are k clusters

→ S has exactly one node from each cluster

- If a node labeled x is in S, then any node labeled $\bar{x}$ cannot also be in S
- There exists a truth assignment such that every term present in S is set to 1

→ The assignment satisfies each one of the clauses in F
→ F is satisfiable

Transitivity of the Reductions

The polynomial-time reducibility is a transitive relation:

(8.9) - If $Z \le_P Y$ and $Y \le_P X$, then $Z \le_P X$.

Proof:

- Given a black box for X, we compose the two algorithms implied by $Z \le_P Y$ and Y $_P X$.
- Replace the black box for Y by a polynomial number of calls to the black box for X together with a polynomial number of simple steps.

The Role of 3-SAT

We have proved

3-SAT ≤P Independent Set ≤P Vertex Cover ≤P Set Cover

We will use that

- 3-SAT $\le_P$ Independent Set
- 3-SAT $\le_P$ Vertex Cover
- 3-SAT $\le_P$ Set Cover

3-SAT is an abstract problem that can be reduced to a large group of problems
→ One of the classifications that we will see is based on polynomial-time reductions.

Example

$$(x1 \lor \bar{x}2 \lor \bar{x}3) \land (\bar{x}1 \lor x2 \lor x3) \land (\bar{x}1 \lor x2 \lor \bar{x}3) \land (x1 \lor \bar{x}2 \lor x3)$$

Lecture 25 - NP & Computational Intractability: Efficient Certification

Tues. Dec. 3, 2019

Decision Problems

Decision Problem - A problem where the answer is yes or no.

Examples:

- Independent Set
  → Does the graph G have an independent set of size k?
- Vertex Cover
  → Does the graph G have a vertex cover of size k?

- Satisfiability
  → Does the set of clauses Φ have a satisfying truth assignment?

Encoding Decision Problems

- **Input -** The input to a problem can be encoded as a binary string s with length |s|
- **Problem -** We identify a decision problem X with the set of strings for which the answer is yes
- **Algorithm -** An algorithm A solves the decision problem X if A(s) = yes $\Leftarrow\Rightarrow$ s ∈ X
- Simplified notation:
  - X(s) = yes −→ s ∈ X
  - X(s) = no −→ s ∈/ X

Polynomial Running Time

Polynomial Running Time - The algorithm A has polynomial running time if there is a polynomial function p(·) such that for every input string s, the algorithm terminates in at most O(p(|s|)) steps.

- **Computationally Tractable** **Polynomial-time algorithm**

The Class P

The Class P - The set of all problems X for which there exists an algorithm A with polynomial running time that solves X is called P.

A more formal definition:
A decision problem X is in P if and only if there is an efficient algorithm A such that for all inputs s,

- if X(s) = yes then A(s) = yes,
- if X(s) = no then A(s) = no.

→ How can we classify the problems that are not in P?

Certification

All problems X ∈ P/ are not equally hard
→ Some problems can at least be checked efficiently

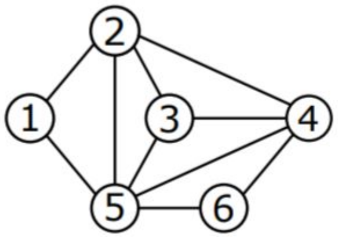
Efficient Certifier - B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input arguments s and t.
- There is a polynomial function p such that for every string s we have s ∈ X if and only if there exists a string t such that |t| ≤ p(|s|) and B(s, t) = yes.

Example: Independent Set

- **X** - A decision problem Does the graph have an independent set of size 3?

- **s** - An instance of the problem A given graph G = (V, E)



- **A** - certificate string: Proof that s ∈ X An independent set of size 3: S = {1, 3, 6} B An algorithm that verifies a solution
- **B** - Check all pairs of nodes if there is an edge between them S ⊆ V, (1, 3) ∈/ E, (1, 6) ∈/ E, (3, 6) ∈/ E → yes

Properties of the Certifier

A certifier must be:

- **Sound** - A certifier is sound if it does not accept any certificate when the answer is no.
  → No false positives
- **Complete** - A certifier is complete if it accepts at least one certificate when the answer is yes.
  → Can be convinced of the answer being yes

A certifier can validate a yes answer, but not a no answer:

- A certificate t such that B(s, t) = yes
  → Proves that s ∈ X
- A certificate t such that B(s, t) = no
  → Does not prove anything!

The Class NP

The Class NP - The set of all problems X for which there exists a certifier B with polynomial running time is called N P.

A more formal definition:
A decision problem X is in P if and only if there is an efficient certifier B such that for all inputs s,

- if X(s) = yes then there is a certificate t such that B(s, t) = yes,
- if X(s) = no then for all certificates t, B(s, t) = no.

NP:

- Nondeterministic Polynomial time
- The original definition was based on nondeterministic Turing machines.

Example: Problems not in NP

- X The Independent Set problem
  - Does the graph have an independent set of size k?

- $\bar{X}$ The complement problem
  - Does the largest independent set of the graph have size smaller than k?

Efficient certificate B

- Polynomial time function B
- Polynomial length certificate t

We need to verify all subsets to see if one of them is an independent set: An exponential number of subsets
→ No efficient certificate!

P and NP

P and NP - P ⊆ N P

Proof:
Consider X ∈ P:

- There is an algorithm A such that A(s) = X(s) for all s
- Need to show that there is an efficient certifier B

Take B(s, t) = A(s) = X(s) for all t:

- B has polynomial running time
- If s ∈ X, there is some t such that B(s, t) = yes
  → Any t!
- If s ∈/ X, then for all t, B(s, t) = no

Examples of Problems in NP

- Independent Set
  - Certificate A set of k vertices
  - Certifier The certifier B verifies that no pair of vertices is joined by an edge
- Set Cover
  - Certificate A list of k sets from the collection
  - Certifier The certifier B checks that the union of these sets is equal to the ground set
- 3-SAT
  - Certificate A truth assignment
  - Certifier The certifier B evaluates the set of clauses using the assignment

P and NP

We do not know any efficient algorithms that can solve these problems
→ We cannot prove that no such algorithm exist!

An Open Question - Is there a problem in N P that does not belong to P? Does
P = NP

The general belief is that *P =/= N P*

Lecture 26 - NP-Complete Problems

Thurs. Dec. 5, 2019

The Hardest Problems in NP

Open Question: P = NP?

→ What are the hardest problems in NP?

NP-Hard - A decision problem X is NP-hard if for every problem Y ∈ NP we have Y $\leq_P$ X.

NP-Complete - A decision problem X is NP-complete if

- X ∈ NP
- X is NP-hard

Polynomial-Time Solutions

(8.12) - Suppose X is a NP-complete problem. Then X is solvable in polynomial time if and only if P = NP.

Proof:

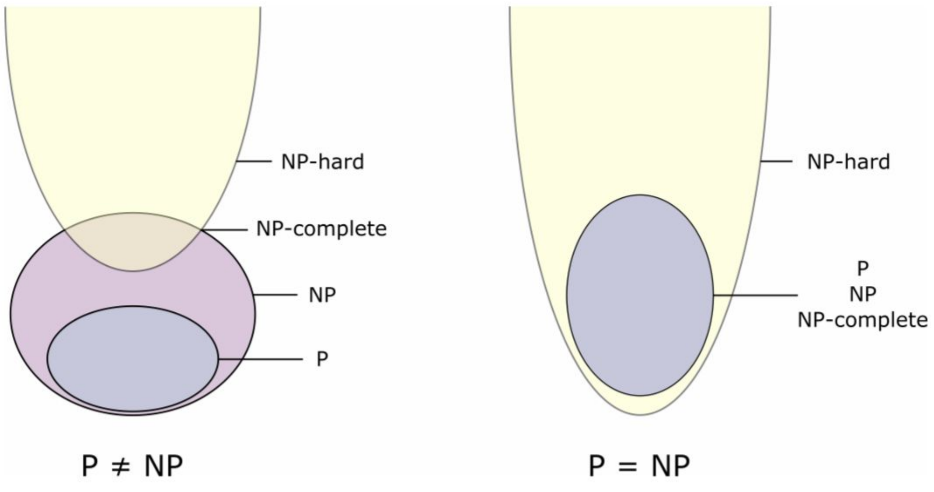
- Assume P = NP

→ X ∈ NP implies X ∈ P and we are done.

- Assume X ∈ P

→ For any other problem Y 2 NP we have Y P X. Then by (8.1) we have Y 2 P, so NP P. From (8.10) we have P NP, which together gives P = NP

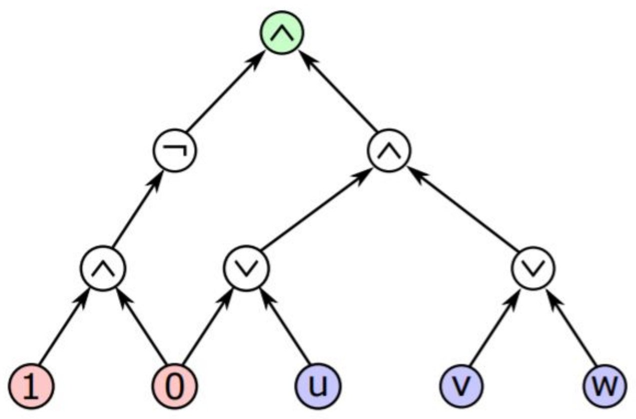
Class Hierarchy



Consequences of (8.12): If there is any problem in NP that cannot be solved in polynomial time, no NP-complete problem can be solved in polynomial time.

Ladner's Theorem - If P =/= NP then NP contains problems that are neither in P nor NP-complete.

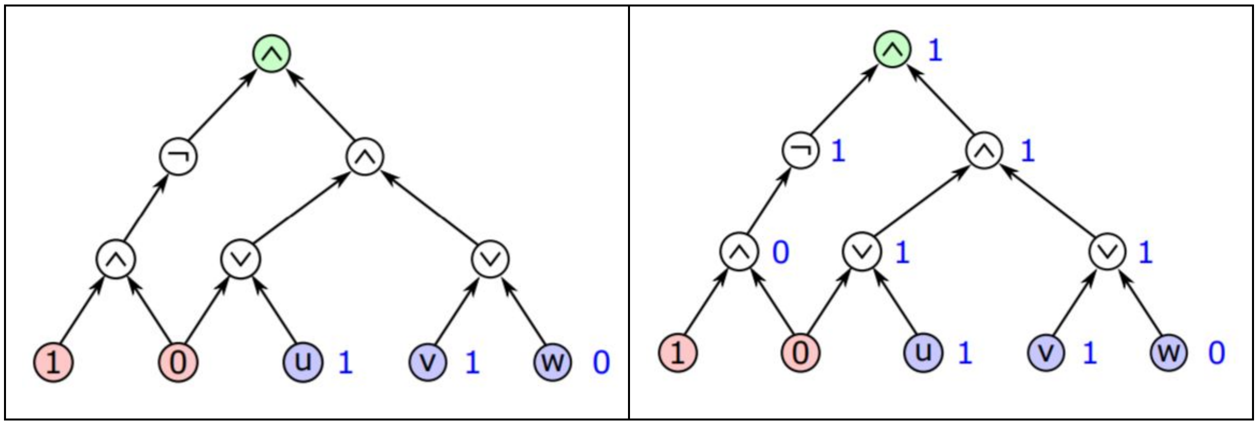
Circuits



- *Sources*
  - ⟶ Constants: $\{0, 1\}$
  - ⟶ Inputs: Variables
- *Operators*
  - $\wedge$ and
  - $\vee$ or
  - $\neg$ not
- *Output*

- The edges carry the value from their start node
- Nodes with an operator carry out the operation
- The value of the circuit is the value of the output node

Example: Circuit



Circuit Satisfiability Problem

- **assignment** - A set of values assigned to the input variables
- **satisfying** - An assignment is satisfying if the output of the circuit is 1.

- **satisfiable** - A circuit is satisfiable if there is a satisfying assignment.

  Circuit Satisfiability Problem - Is a given circuit K satisfiable?

This problem was one of the first known to be NP-complete:

(8.13) Cook-Levin Theorem - Circuit Satisfiability is NP-complete.

NP-Completeness of Circuit Satisfiability

- How can we prove that Circuit Satisfiability is NP-complete?
- Prove that for an arbitrary problem X ∈ NP we have

$$X \leq_P \text{Circuit Satisfiability}$$

- Consider an algorithm
  - **Input A** - fixed number n of bits
  - **Output A** - yes/no answer

→ Can be encoded using a circuit

- We know that X has an efficient certifier B(·, ·)

→ Encode B as a circuit!

- Consider for the problem X
  - **s** - an instance of length n
  - **t** - a certifier of length p(n)
- Convert the certifier to a circuit K with n + p(n) sources
  - **n** - sources with the values of the bits in s
  - **p(n)** - sources with the values of the bits in t
- s ∈ X if and only if there is an assignment to the input bits to K so that the output is 1:

$$X \leq_P \text{Circuit Satisfiability}$$

Example: Independent Set of Size 2

Consider the problem:

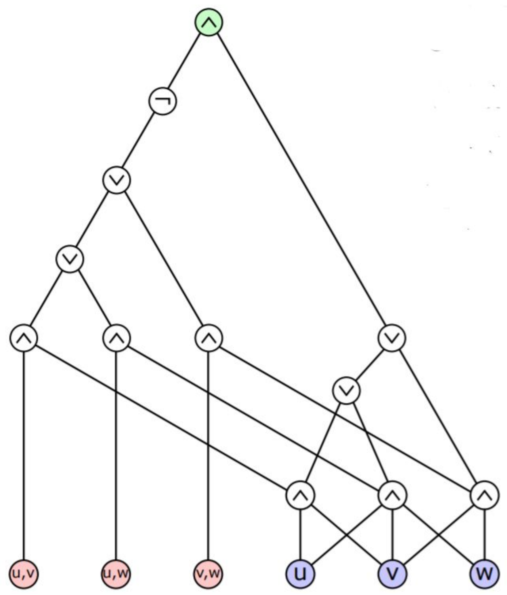
Given a graph G, does it contain an independent set of size 2?

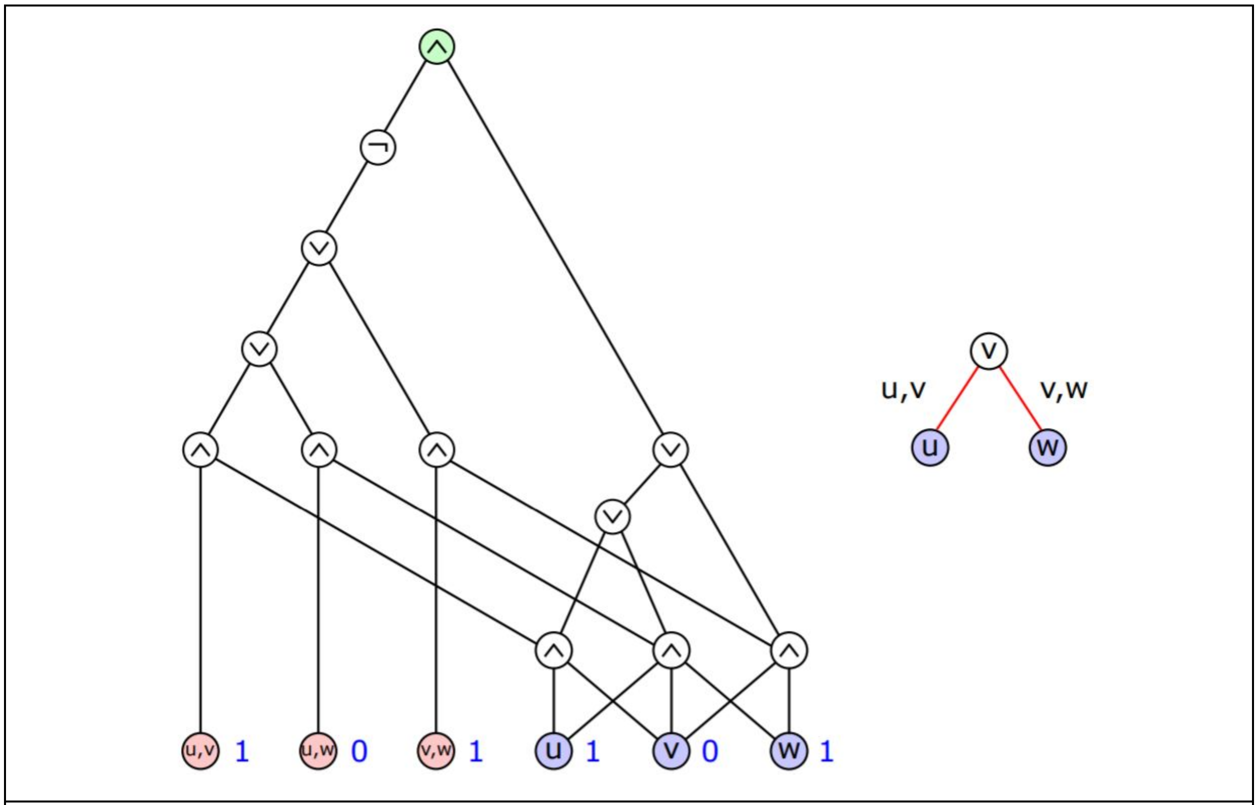
The certifier B must check that for an instance s and a certificate t

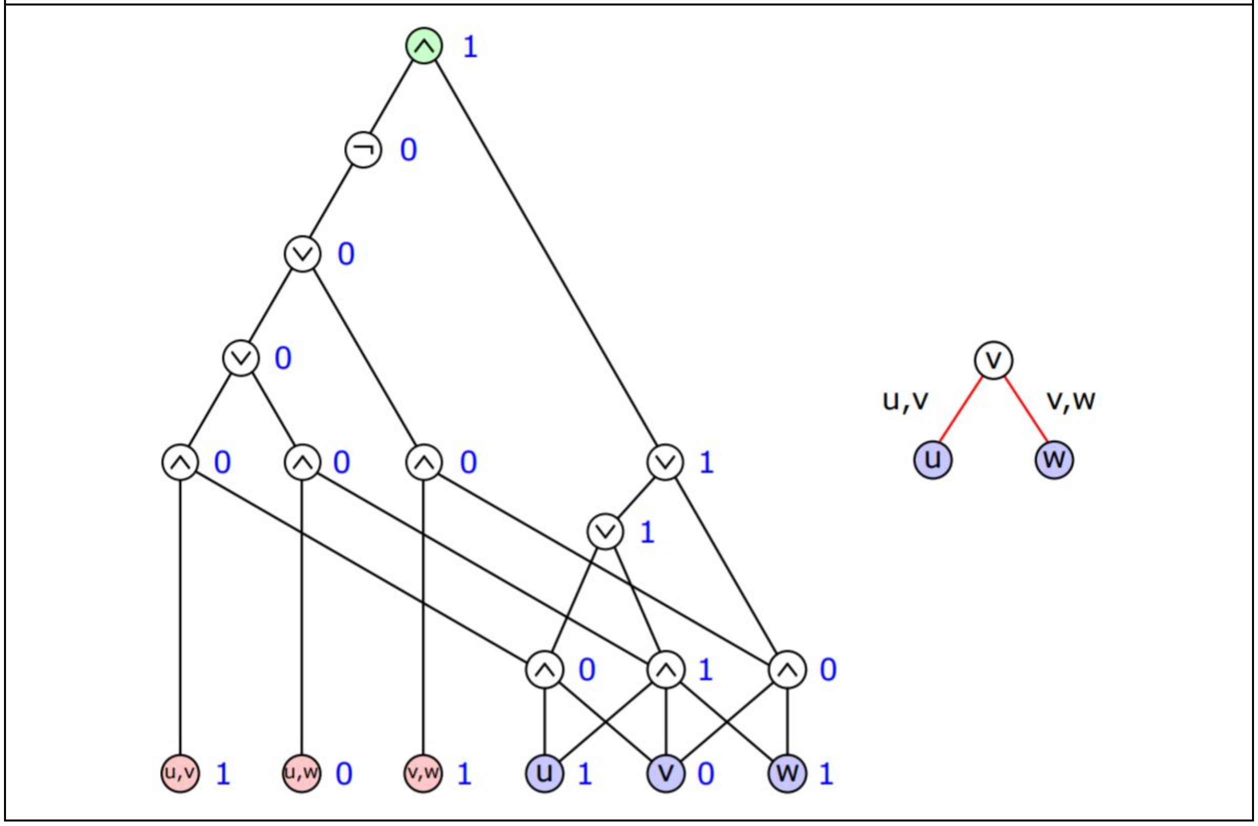
- At least 2 nodes have been chosen for the certificate t
  → The set has size at least 2
- No two nodes in t are joined by an edge in s
  → The set is independent

→ What does the circuit that encodes B look like?

- The circuit encodes the certifier for a graph with 3 nodes
- left The instance: Edges present in the graph
- right The certificate: Nodes present in the set




u,v
1
u,w
0
v,w
1
u
1
v
0
w
1
u,v
v,w


1
0
0
0
0
0
0
1
1
0
1
0
u,v
1
u,w
0
v,w
1
u
1
v
0
w
1
u,v
v,w

Proving Problems NP-Complete

(8.14) - If Y is an NP-complete problem and X is a problem in NP with the property that Y ≤P X, then X is NP-complete.

Proof:
X ∈ NP, so we only need to prove that Z ≤P X for any Z ∈ NP.
Y is NP-complete, so for any Z ∈ NP

Z ≤P Y

By assumption, we have

Y ≤P X

By the transitivity property (8.9) we have

Z ≤P X

3-SAT is NP-Complete

Transform any instance of 3-SAT to Circuit using the following:

- Define the variable xv associated with the node v of the circuit.

If a node v with one entering edge from u is labeled:

- ¬ We need to have xv = x¯u so we add clauses
  - (xv ∨ xu) and (x¯v ∨ x¯u)
- If a node v with two entering edges from u and w is labeled:
  - ∨ We need to have xv = xu ∨ xw so we add clauses
    - (xv ∨ x¯u), (xv ∨ x¯w), (x¯v ∨ xu ∨ xw)
  - ∧ We need to have xv = xu ∧ xw so we add clauses
    - (x¯v ∨ xu), (x¯v ∨ xw), (xv ∨ x¯u ∨ x¯w)
- Finally, add clauses
  - xv or x¯v for every source node
  - xo to enforce the output value 1

NP-Complete Problems

3-SAT - 3-SAT is an NP-complete problem.

NP-Complete Problems - Independent Set, Vertex Cover, Set Cover and Set Packing are NP-complete problems.

Proving Any Problem NP-Complete

Given a new problem X
→ Prove that X is NP-complete

- General strategy: 1
  - Prove that X ∈ NP 2
  - Choose a problem Y known to be NP-complete 3
  - Prove that Y ≤P X

Polynomial Reductions

How do we prove that Y ≤P X?

General strategy:

- Consider an arbitrary instance sY of problem Y
- Show how to construct in polynomial time an instance sX of problem X such that
  - Y(sY) = yes ⇒ X(sX) = yes
  - X(sX) = yes ⇒ Y(sY) = yes

  → sY and sX have the same answer