Assignment lecture W47

TDT4121

Task 1a

Task 1 Approximation

a) Why do we need approximation algorithms?

Task 1a: Solution

Solution:
Because with practical problems, if they happen to be NP-Hard or NP-complete, simply saying that ”there is no solution” isn’t a viable option, waiting for days or years for an algorithm to finish isn’t exactly helpful either. We need some form of solution that solves the problem to a satisfactory level.

Task 1b

b) Give an example of a situation/problem where an approximation solution is needed, that is *not* covered in the book or lectures.

Task 1b: Example solution

- Traveling Salesman: approximate the solution by building a Minimum Spanning Tree (MST), and generate a cycle that visits all nodes in T using depth-first search.

Task 2a

You're acting as a consultant for *Dofoora*, a fictional delivery service in Trondheim. They use couriers on bicycles to deliver food orders from a restaurant to customers placing their orders online.

Here is a basic sort of problem they face. A number of orders n arrive, each with a total weight $w_1, w_2, ..., w_n$. At the restaurant, there is a set of couriers waiting, each of which can deliver a limit of K units of weight (the company doesn't want to get into trouble by a courier getting hurt due to too much weight). A courier can carry several orders, subject to the weight restriction of K. The goal is to minimize the number of couriers that are needed to deliver all the orders. This problem is NP-complete (you don't have to prove this). You can assume that K and each w_i is an integer and that the place of delivery does not matter.

A greedy algorithm you might use for this is the following. Start with a courier who starts picking up orders $1, 2, 3, ...$ until the total weight would exceed the limit K by picking up another order. This courier is now "full" and is sent on its way to deliver the orders. In comes the next courier which repeats the process.

Task 2a

You're acting as a consultant for *Dofoora*, a fictional delivery service in Trondheim. They use couriers on bicycles to deliver food orders from a restaurant to customers placing their orders online.

Here is a basic sort of problem they face. A number of orders n arrive, each with a total weight $w_1, w_2, ..., w_n$. At the restaurant, there is a set of couriers waiting, each of which can deliver a limit of K units of weight (the company doesn't want to get into trouble by a courier getting hurt due to too much weight). A courier can carry several orders, subject to the weight restriction of K. The goal is to minimize the number of couriers that are needed to deliver all the orders. This problem is NP-complete (you don't have to prove this). You can assume that K and each w_i is an integer and that the place of delivery does not matter.

A greedy algorithm you might use for this is the following. Start with a courier who starts picking up orders $1, 2, 3, ...$ until the total weight would exceed the limit K by picking up another order. This courier is now "full" and is sent on its way to deliver the orders. In comes the next courier which repeats the process.

Task 2a

You're acting as a consultant for *Dofoora*, a fictional delivery service in Trondheim. They use couriers on bicycles to deliver food orders from a restaurant to customers placing their orders online.

Here is a basic sort of problem they face. A number of orders n arrive, each with a total weight $w_1, w_2, ..., w_n$. At the restaurant, there is a set of couriers waiting, each of which can deliver a limit of K units of weight (the company doesn't want to get into trouble by a courier getting hurt due to too much weight). A courier can carry several orders, subject to the weight restriction of K. The goal is to minimize the number of couriers that are needed to deliver all the orders. This problem is NP-complete (you don't have to prove this). You can assume that K and each w_i is an integer and that the place of delivery does not matter.

A greedy algorithm you might use for this is the following. Start with a courier who starts picking up orders $1, 2, 3, ...$ until the total weight would exceed the limit K by picking up another order. This courier is now "full" and is sent on its way to deliver the orders. In comes the next courier which repeats the process.

a) Give an example of a set of orders with weights, and a value of K, where this algorithm does not use the minimum amount of couriers.

Task 2a: Solution

Solution:
We let three orders $\{1, 2, 3\}$ have weights $\{w_1, w_2, w_3\} = \{1, 2, 1\}$ and $K = 2$. The greedy algorithm will use three couriers, since the first will pick up order 1, and not have room for order 2 so it heads off. Two more couriers are then needed to pick up orders 2 and 3. However if the first courier had waited until a second courier picked up w_2, then the first courier could have picked up order 3 as well.

Task 2b

b) **Show, however, that the number of couriers used by this algorithm is within a factor of 2 of the minimum possible number of couriers needed (the algorithm will never use more than double of what an optimal solution would have), for any set of weights and any value of K.**

Task 2b: Solution

Solution:
Let $W = \sum_i w_i$. Note that in any solution, each courier holds at most K units of weight, so W/K is the lower bound of the number of couriers needed.

Suppose the number of couriers used by our greedy algorithm is an odd number $m = 2q+1$. Divide the couriers used into consecutive groups of two, for a total of $q+1$ groups. In each group but the last, the total weight of orders must be *strictly* greater than K (else, the second courier in the group would not have any orders to deliver.) Thus, $W > qK$ and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q+1$ couriers, which is within a factor of 2 of $m = 2q + 1$.

Task 3a

Task 3 Largest Subset of Integers

Suppose you are given a set of positive integers $A = \{a_1, a_2, ..., a_n\}$ and a positive integer B. A subset $S \subseteq A$ is called feasible if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B$$

The sum of the numbers in S will be called the total sum of S. You would like to select a feasible subset S of A whose total sum is as large as possible. Example. If $A = \{8, 2, 4\}$ and $B = 11$, then the optimal solution is the subset $S = \{8, 2\}$.

a) Here is a proposed algorithm for this problem.

```
largest_subset_of(A):
    S = {}
    T = 0
    for i=1,2,...,n:
        if T + ai <= B:
            S += {ai}
            T += ai
```

This algorithm does not always give the optimal solution. Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

Task 3a: Solution

Solution:
Let $A = \{1, 10\}$ and $B = 10$, only a_1 will be chosen ($S = \{a_1\}$) when the optimal solution is $S = \{a_2\}$.

Task 3b

b) **Give a polynomial-time approximation algorithm for this problem with the following guarantee: It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$. Your algorithm should have a running time of at most $O(n \log n)$.**

Task 3b: Solution

Solution:
This can be done in $O(n)$ time.

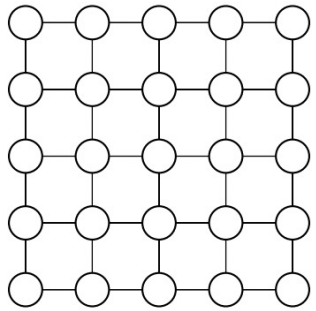
We go through all the numbers in the set in order. If a number is exactly B we can return that number as the optimal set, if the number is greater than B we skip it since it cannot be a part of the optimal set. Otherwise we add the numbers to S until the total T exceeds B. This happens when number a_j is attempted to be added. We at this point have: $\sum_{i=1}^{j} a_i > B$, we also have that $\sum_{i=1}^{j-1} a_i \leq B$ and $a_j < B$. Thus one of the sets $\{a_1, a_2, ..., a_{j-1}\}$ or $\{a_j\}$ must have size of at least $B/2$, and at most B. We have then found a guaranteed subset of total sum at least $B/2$, and can return the larger one of the two.

```
1  S = {}
2  T = 0
3  for i=1,2,...,n:
4      if ai == B:
5          return {ai}
6      if ai > B:
7          continue
8      if T + ai < B:
9          S += {ai}
10         T += ai
11     else:
12         if T > ai:
13             return S
14         else:
15             return {ai}
```

Task 4a

Task 4 Grid Graph

Suppose you are given an $n \times n$ grid graph G as shown below ($n = 5$).



Associated with each node v is a weight w_v, which is a nonnegative integer. You may assume that the weights of all nodes are distinct. Your goal is to choose an independent set S of nodes of the grid, so that the sum of the weights of the nodes in S is as large as possible. (The sum of the weights of the nodes in S will be called its total weight.)

Consider the following greedy algorithm for this problem:

```
The "heaviest-first" greedy algorithm:
    Start with S equal to the empty set
    while some node remains in G
        Pick a node v of maximum weight
        Add v to S
        Delete v and its neighbors from G
    Endwhile
    return S
```

a) Let S be the independent set returned by the "$heaviest - first$" greedy algorithm, and let T be any other independent set in G. Show that, for each node $v \in T$, either $v \in S$, or there is a node $v' \in S$ so that $w_v \leq w_{v'}$ and (v, v') is an edge of G.

Task 4a: Solution

Solution:
If $v \notin S$, it must have never been chosen by the greedy algorithm. This means that it was deleted in some iteration by the selection of node v'. By the definition of the selection rule, this node v' must be both a neighour of v, and have at least as much weight as v.

Task 4b

b) **Show that the "$heaviest - first$" greedy algorithm returns an independent set of total weight at least $1/4$ times the maximum total weight of any independent set in the grid graph G.**

Task 4b: Solution

Solution:
Consider any other independent set T. For each node $v \in T$, we *charge* it to a node in S as follows. If $v \in S$, then we charge v to itself. Otherwise, by (a), v is a neighbor of some node $v' \in S$ whose weight is at least as large. We charge v to v'.

Now if v is charged to itself, then no other node is charged to v, since S and T are independent sets. Otherwise, at most four neighboring nodes of no greater weight are charged to v. Either way, the total weight of all nodes charged to v is at most $4w_v$. Since these charges account for the total weight of T, it follows that the total weight of nodes in T is at most four times the total weight of nodes in S, making the total weight of nodes in S at least $\frac{1}{4}w_T$

Task 5

Task 5 ★ *Optional*

a) Greedy Balance

Some friends of yours are working with a system that performs real-time scheduling of jobs on multiple servers, and they've come to you for help in getting around an unfortunate piece of legacy code that can't be changed.

Here's the situation. When a batch of jobs arrives, the system allocates them to servers using the simple Greedy-Balance Algorithm from Section 11.1 in the book, which provides an approximation to within a factor of 2. The algorithm is shown below:

```
Greedy-Balance:
    Start with no jobs assigned
    Ti=0
    Ai={} for all machines Mi
    For j=1,...,n
        Let Mi be a machine that achieves the minimum Tk
        Assign job j to machine Mi
        Set Ai += {j}
        Set Ti += tj
    EndFor
```

In the decade and a half since this part of the system was written, the hardware has gotten faster to the point where, on the instances that the system needs to deal with, your friends find that it's generally possible to compute an optimal solution.

The difficulty is that the people in charge of the system's internals won't let them change the portion of the software that implements the Greedy-Balance Algorithm so as to replace it with one that finds the optimal solution. (Basically, this portion of the code has to interact with so many other parts of the system that it's not worth the risk of something going wrong if it's replaced.)

After grumbling about this for a while, your friends come up with an alternate idea. Suppose they could write a little piece of code that takes the description of the jobs, computes an optimal solution (since they're able to do this on the instances that arise in practice), and then feeds the jobs to the Greedy-Balance Algorithm in an order that will cause it to allocate them optimally. In other words, they're hoping to be able to reorder the input in such a way that when Greedy-Balance encounters the input in this order, it produces an optimal solution.

So their question to you is simply the following: Is this always possible? Their conjecture is:

> For every instance of the load balancing problem from Section 11.1, there exists an order of the jobs so that when Greedy-Balance processes the jobs in this order, it produces an assignment of jobs to machines with the minimum possible makespan.

Decide whether you think this conjecture is true or false, and give either a proof or a counterexample.

Task 5: Solution

Solution:
This is true.

Consider the assignment of jobs to machines in an arbitrary optimal solution, and an order of jobs arbitrarily on each machine. We say that the *base height* of a job j is the total time requirement of all jobs that precede it on its assigned machine.

We order all jobs by their base heights, and we feed them to the Greedy-Balance algorithm in this order.

We claim the following by induction on r. After the first r jobs have been processed by Greedy-Balance, the set of machine loads is the same as the set of machine loads if we consider the assignment of these r jobs made by the optimal solution.

This is clearly true for $r = 1$, since one machine will have load t_1 and all others will have load 0. Now suppose it is true up to some r, with loads $T_1, ..., T_m$, and consider job $r+1$. Because we have sorted jobs by their base height, job $r+1$ comes from the machine that, in the optimal solution, has load $min_i T_i$. By the definition of Greedy-Balance, this is the machine on which job $r+1$ will be placed, giving it a load of $t_{r+1} + min_i T_i$. This completes the induction step.

Small summary of some relevant themes

Analysis of algorithms

- When we analyse we are mainly looking at **Use of resources** and **accuracy**
- Accuracy is often proved by induction
- We often want to describe use of resources with asymptotic notation

Analysis of algorithms – Input and runtime

- Most often we are interested in the runtime as a function of the input size. It can be either the number of elements in the input or the number of bits the input needs to be represented.

- Sometimes it is not only the input size but also the characteristics of the input that govern how long the algorithm needs. In this case, it is often useful to talk about best- and worst case runtime

Analysis of algorithms – Input and runtime example

- Insertion type consists of two loops. The outermost runs (n-1) times, the innermost runs a maximum of (n-2) times.

- The running time for all possible runs of insertion sort will therefore be in O(n^2)

```
def insertionsort(A, len):
    for j in range(1,len):
        key = A[j]
        i = j - 1
        while(i >= 0 and A[i] > key):
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Analasys of algorithms – Recursive runtime

- In some cases, it is convenient to express the runtime as a recursive expression

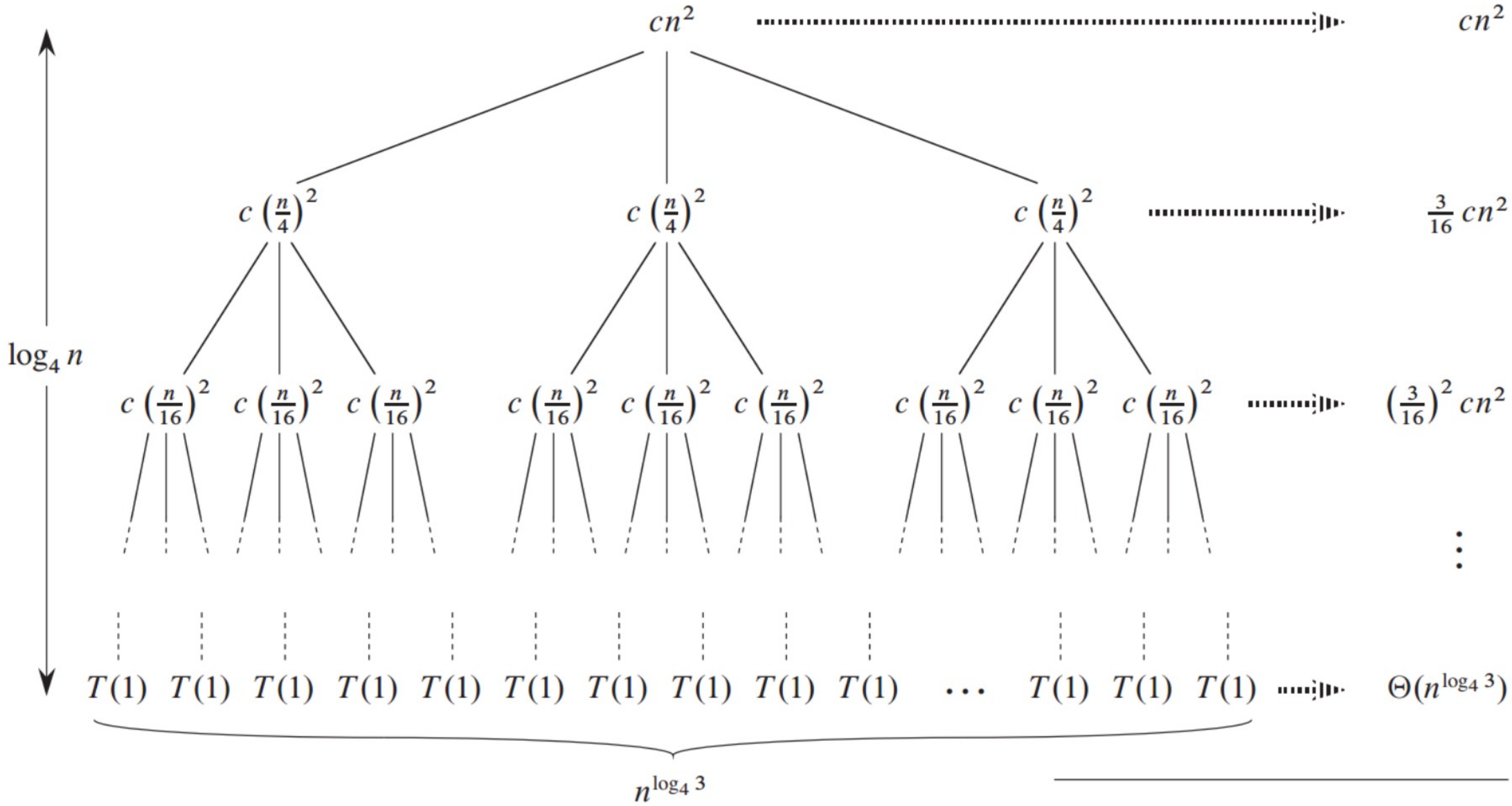
$$T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$$

- In order to be able to use these in practice, we want to translate it into an expression depending only on n

- This can be done using the iteration method, recurrence trees, or the substitution method

Recurrence trees

- Plot each recursive call as nodes
- Can be used as a guess for the substitution method, or as a solution in itself if you are careful
- We look at the tree of $T(n) = 3T(n/4) + cn^2$



$T(n) = 3T(n/4) + cn^2$

Total: $\Theta(n^2)$

Proof by induction (substitution)

- Step 1: guess a solution
- Step 2 assume that it holds for all $m < n$:
- Step 3: insert and substitute
- Step 4: show that it holds for one base case (arbitrarily chosen n)

Claim. If T(n) satisfies this recurrence, then T(n) = n $\log_2$ n.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2\,T(n/2) + n & \text{if } n > 1 \end{cases}$$

↑ assumes n is a power of 2

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: T(n) = n $\log_2$ n.
- Goal: show that T(2n) = 2n $\log_2$ (2n).

recurrence ↓

$$T(2n) = 2\,T(n) + 2n$$

inductive hypothesis →

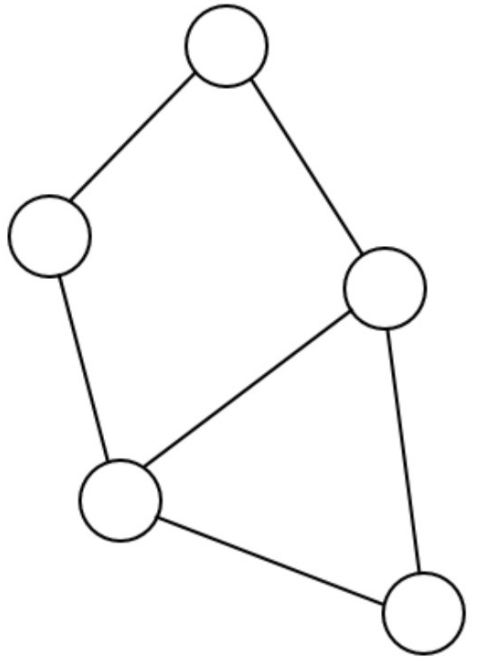
$$= 2\,n \log_2 n + 2n$$

$$= 2\,n\,(\log_2 (2n) - 1) + 2n$$

$$= 2\,n \log_2 (2n). \quad \blacksquare$$

Graph traversal

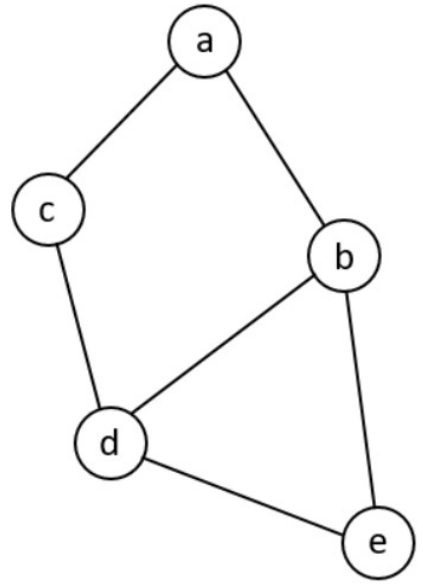
What is a graph?



- Collection of nodes and edges
- Directed or undirected edges
- Can be represented by neighbor matrices or neighbor lists

Neighbor matrix

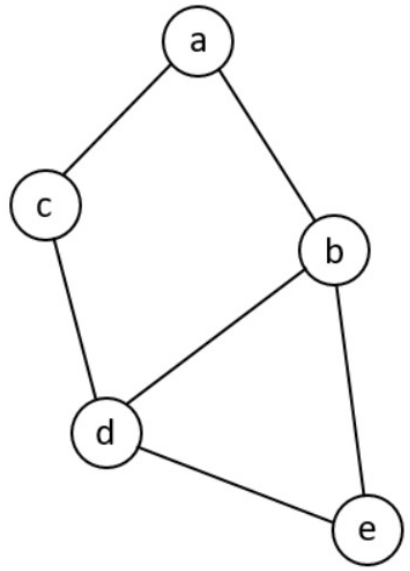
- Read from row to column
- Quickly look up a specific edge
- Less practical for traversing



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	0	1	1
c	1	0	0	1	0
d	0	1	1	0	1
e	0	1	0	1	0

Neighbor lists

- Each node has a list of neighbors
- No more information than necessary, good for traversal
- Checking if an edge exists requires a linear search



a → [b, c]

b → [a, d, e]

c → [a, d]

d → [c, b, e]

e → [d, b]

Graph traversal

- We want to search through the nodes in a graph
- Common: We start in a node, and choose new nodes to traverse among the neighbors
- Which nodes do we choose?
- How do we keep track of which nodes have already been traversed?

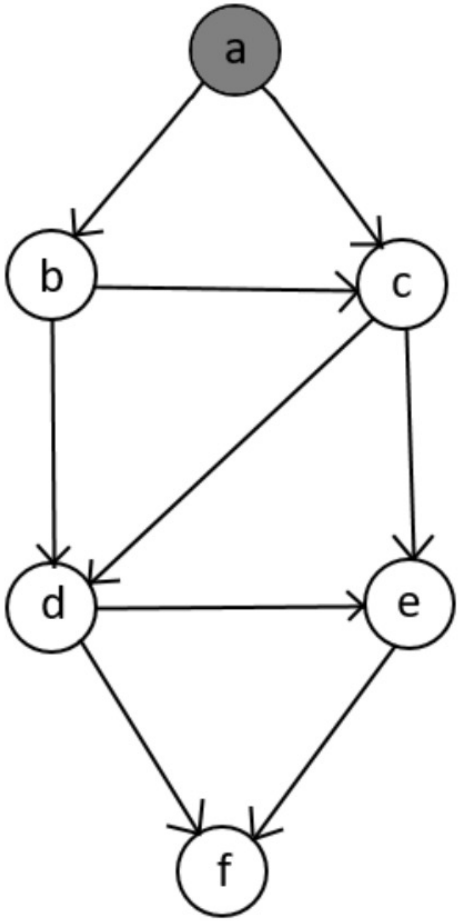
Breadth First (BFS)

- We give all the nodes a color, and start by coloring all the white ones.
- The nodes are organized into a queue.
- Nodes in the queue are grey
- Nodes that have been dequeued are black

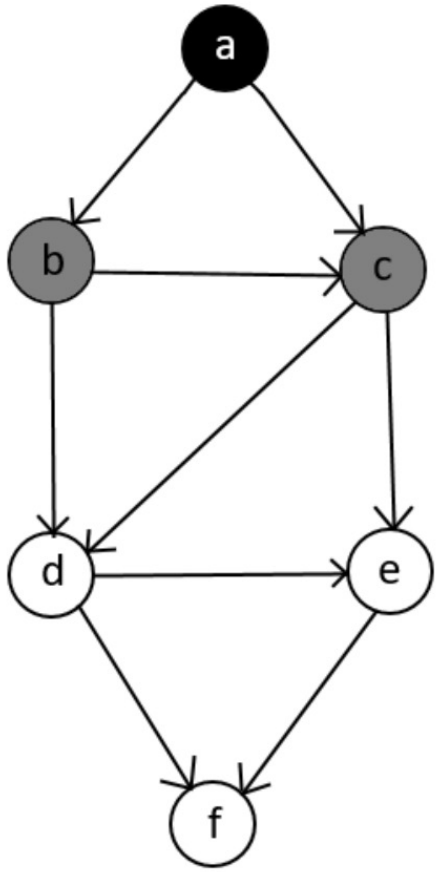
- When all the neighbors have been added to the queue, we take the first one out, and inspect the neighbors of the next one in the queue

Queue = [a] (we put in on the right, and take out on the left)

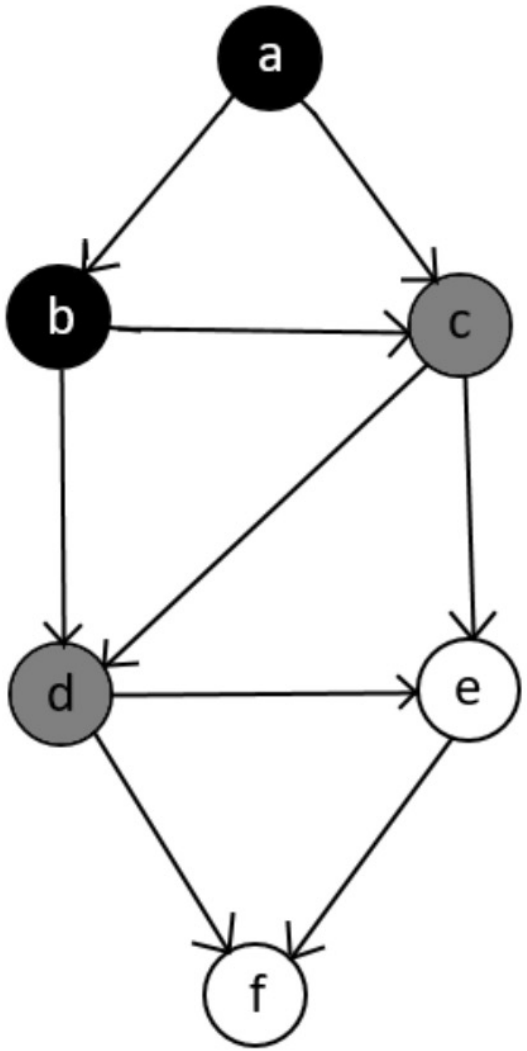
Finished = Ø



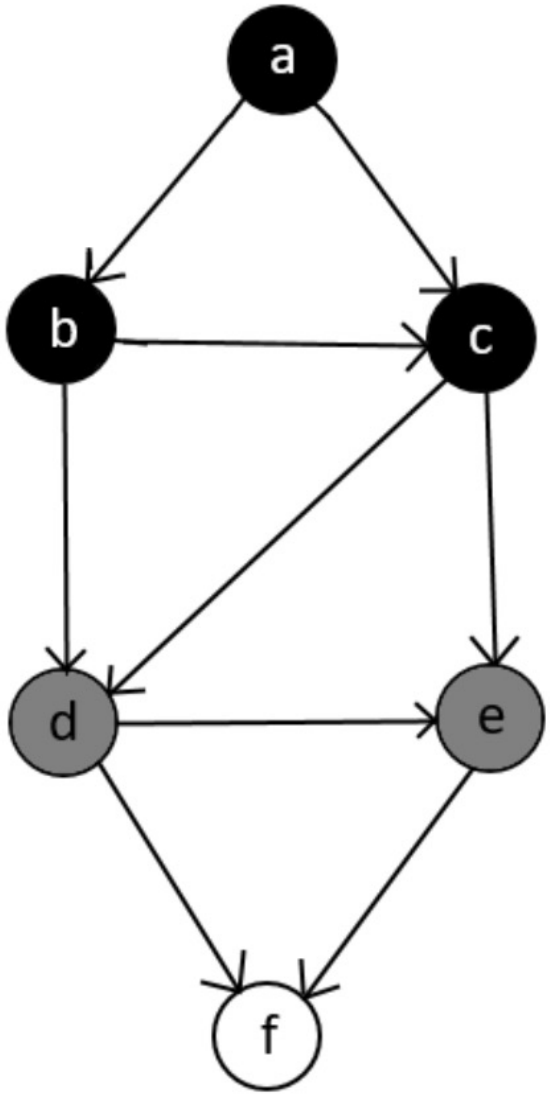
- Queue = [b, c]
- Done = {a}



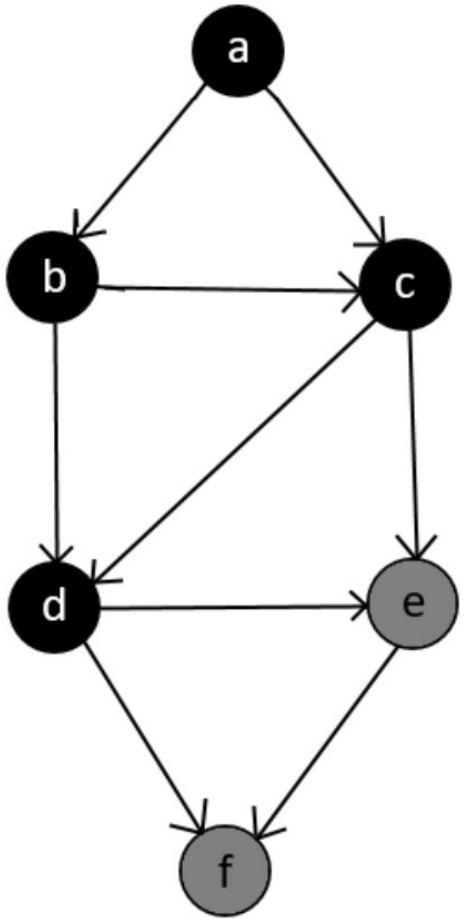
- Queue = [c, d]
- Done = {a, b}



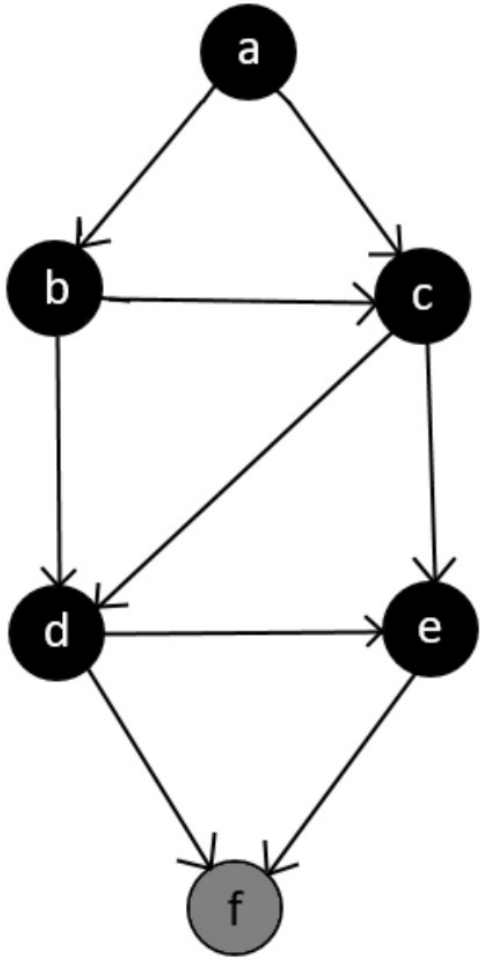
- Queue = [d,e]
- Done = {a, b, c, }



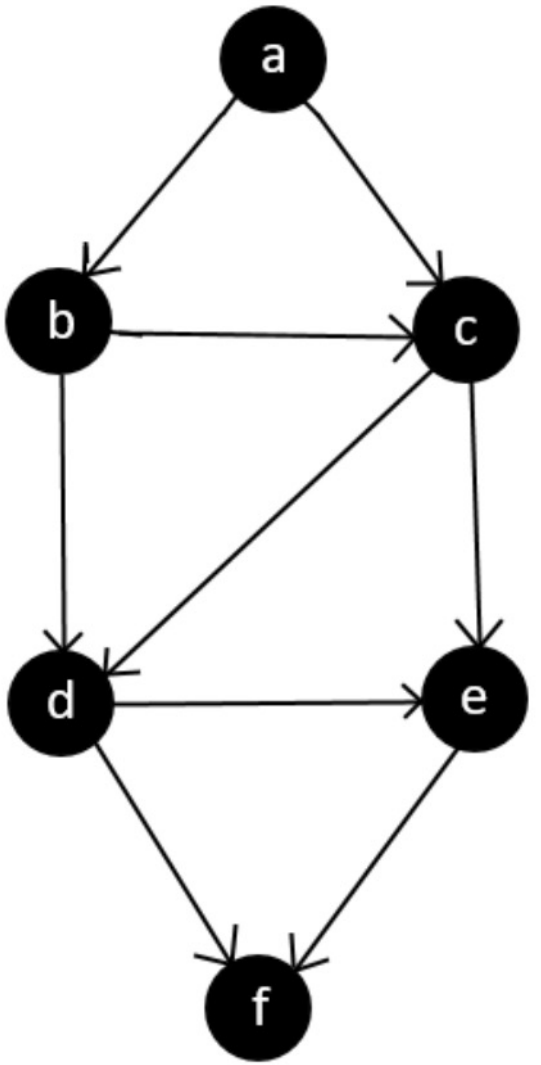
- Queue = [e, f]
- Done = {a, b, c, d}



- Queue = [f]
- Done = {a, b, c, d, e}



- Queue = []
- Done = {a, b, c, d, e, f}



Run time and applications BFS

- O(V + E)
- Possible to save predecessors to keep the traversal tree
- Provides the shortest path from the start node to all other reachable nodes
- Subalgorithm in other algorithms
  - Edmonds-Karp
  - Dijkstra

Greedy algorithms

- Many algorithms consist of making a series of choices. In some cases, making the choice that looks best here and now will provide an optimal solution. The problems this works for have the greed property

- Greedy algorithms are often easy to implement, but it can be challenging to know for which problems they give an optimal result. Many problems that can be solved by dynamic programming can also be solved by a greedy algorithm, but not all.

- Important examples from the syllabus are activity selection, the continuous knapsack problem and huffman coding

What does it take for a problem to be solved greedily?

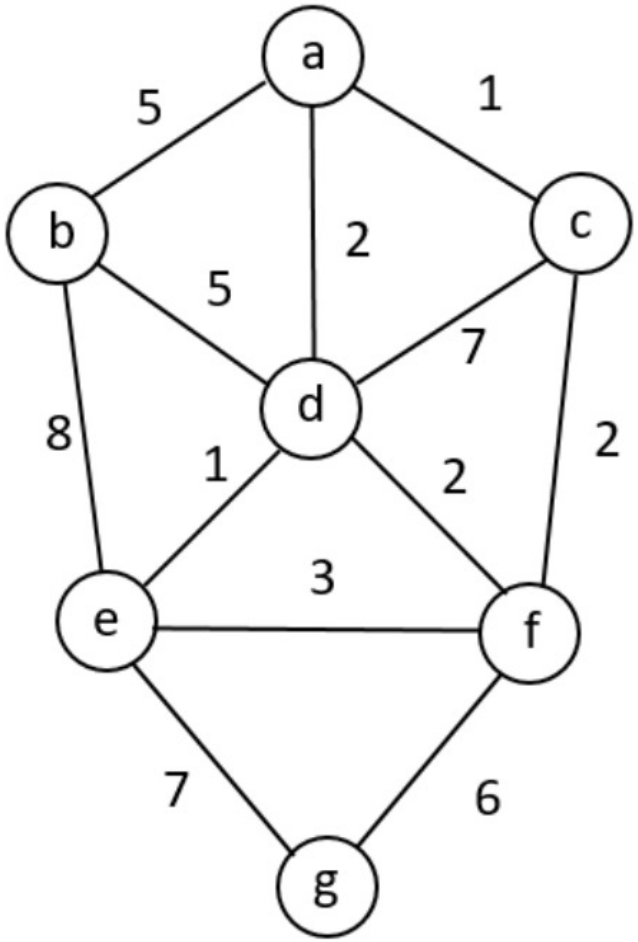
- If a problem is to be optimally solved by a greedy algorithm, it must have an **optimal substructure**. This means that each optimal solution consists of optimal partial solutions.

- We must be guaranteed that the greedy choice gives us an element that belongs in an optimal solution; The greedy choice must choose an element that another optimal algorithm would have also chosen. This is the **greedy-choice property**
- .Each choice must result in only one new subproblem.

Minimal spanning trees

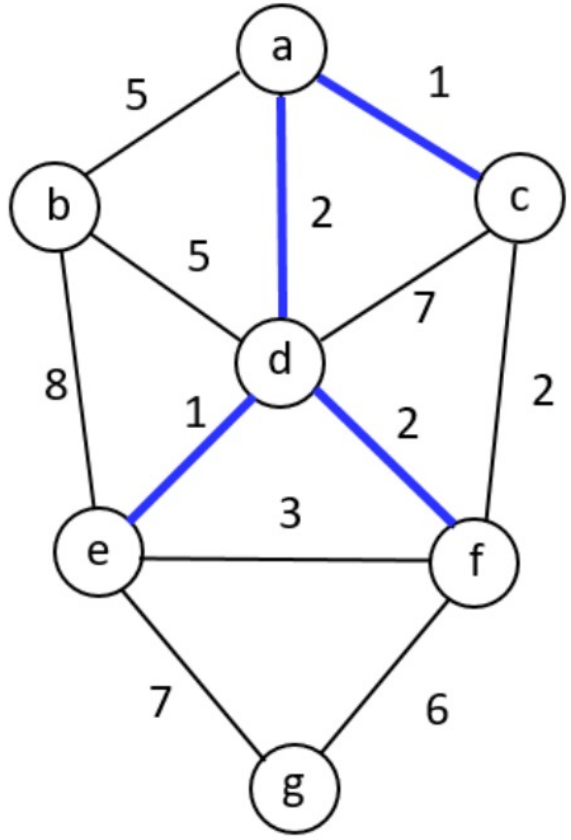
The problem

- We have an undirected graph with n nodes, where each edge has a cost associated with it
- This cost is given by a weight function w(u, v), which gives all edges (u, v) a numerical value
- Goal: Connect all the nodes with the cheapest possible tree



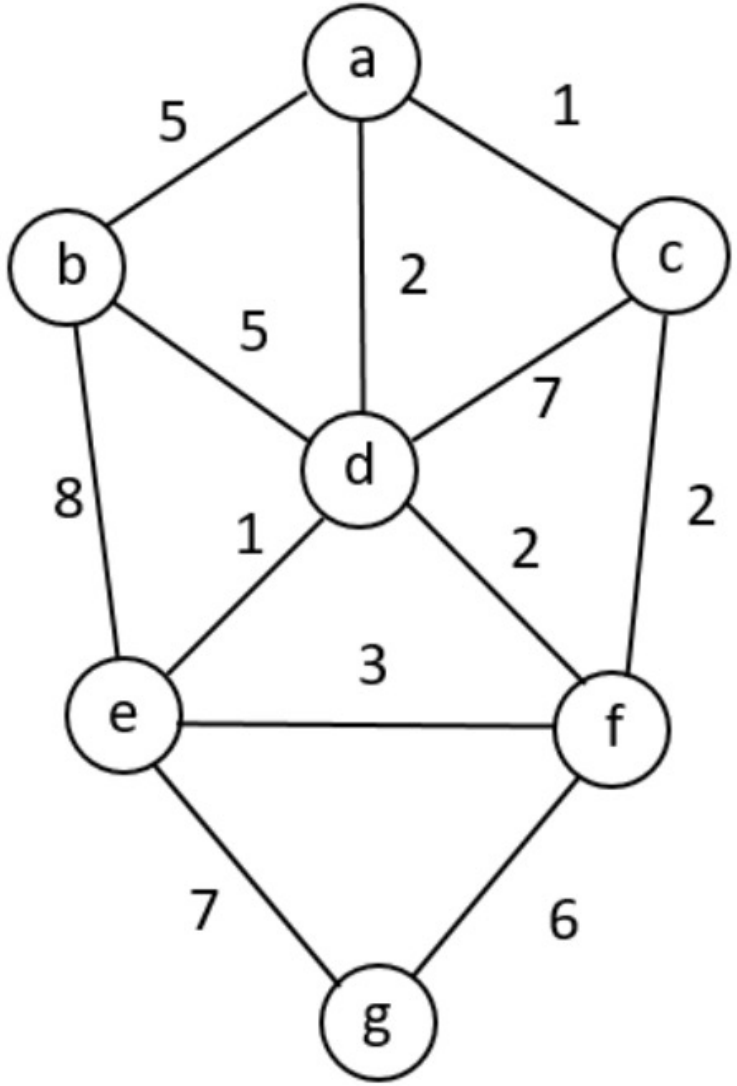
Generic MST

- Greedy solution: pick with the cheapest safe edge
- An edge is safe if it can be added to the solution quantity such that:
  - We still have a tree
  - The result is a subset of a minimal spanning tree
- In the graph on the right, for example (b,d) is safe, while (f, c) is not

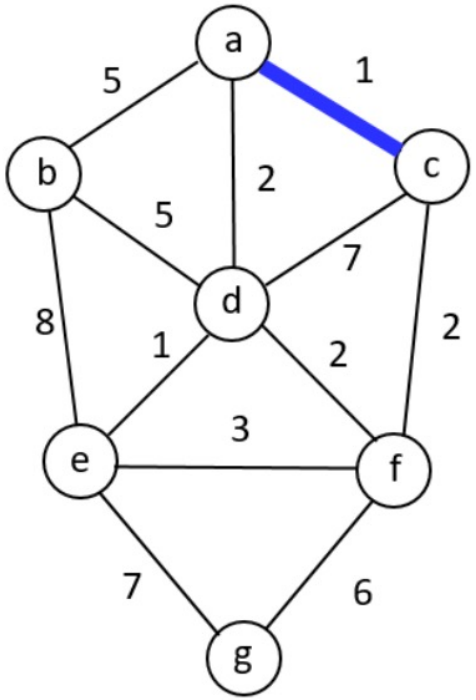


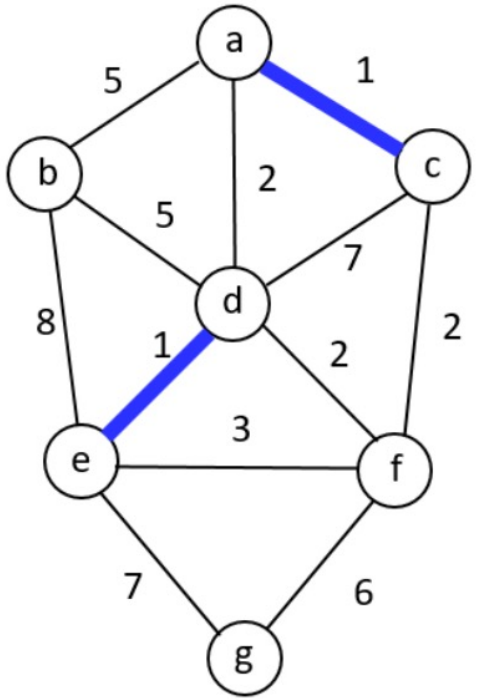
Kruskal's algorithm

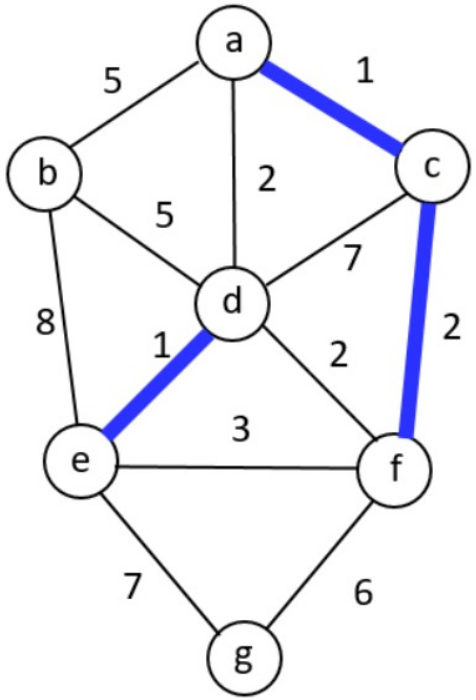
- Finds a safe edge by taking the lightest edge connecting two trees in the forest from already selected edges
- Keep track of the forest using the disjoint set
- When there are no more trees to connect, it means that there is only one left spanning the graph, and this is the minimal spanning tree
- O(E lg V) (with min-heap)



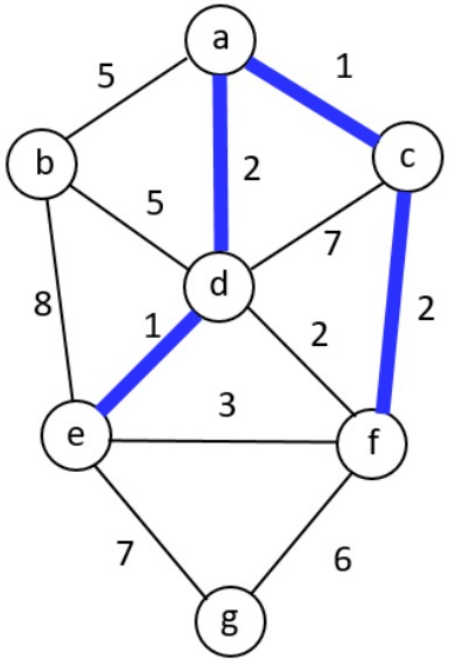
Kruskal's algorithm

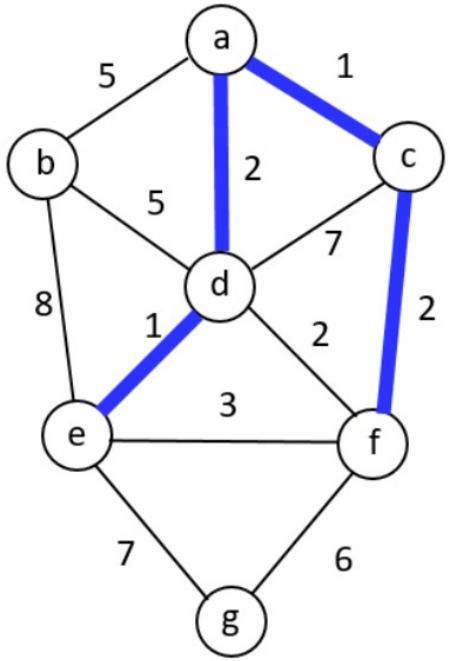


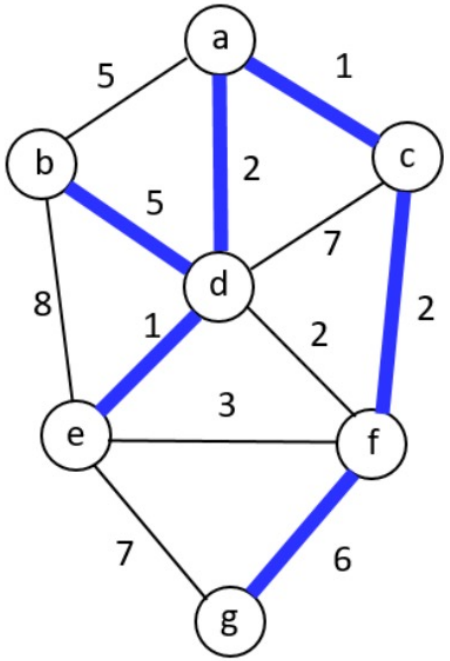




Kruskal's algorithm



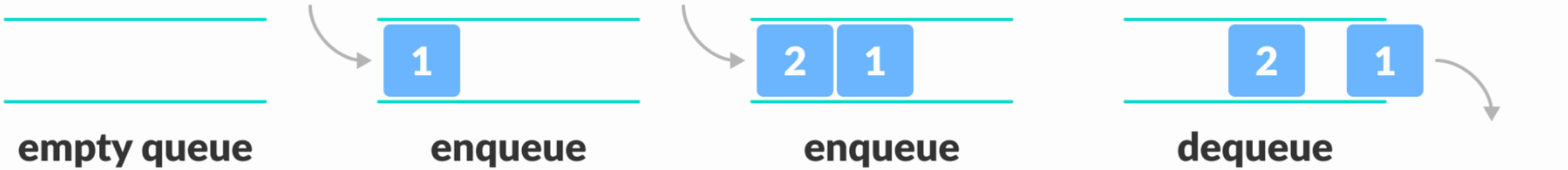




Data structures

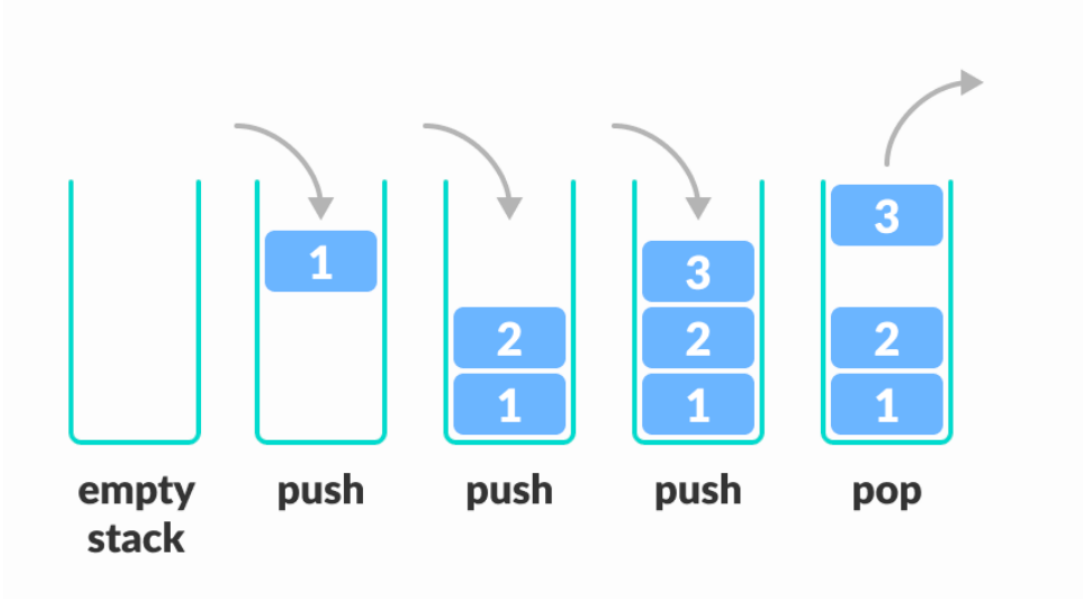
Queue

- It is a data structure that is used to store data in a first-in-first-out (FIFO) manner. The first element added (enqueued) to the queue is the first element to be removed (dequeued)



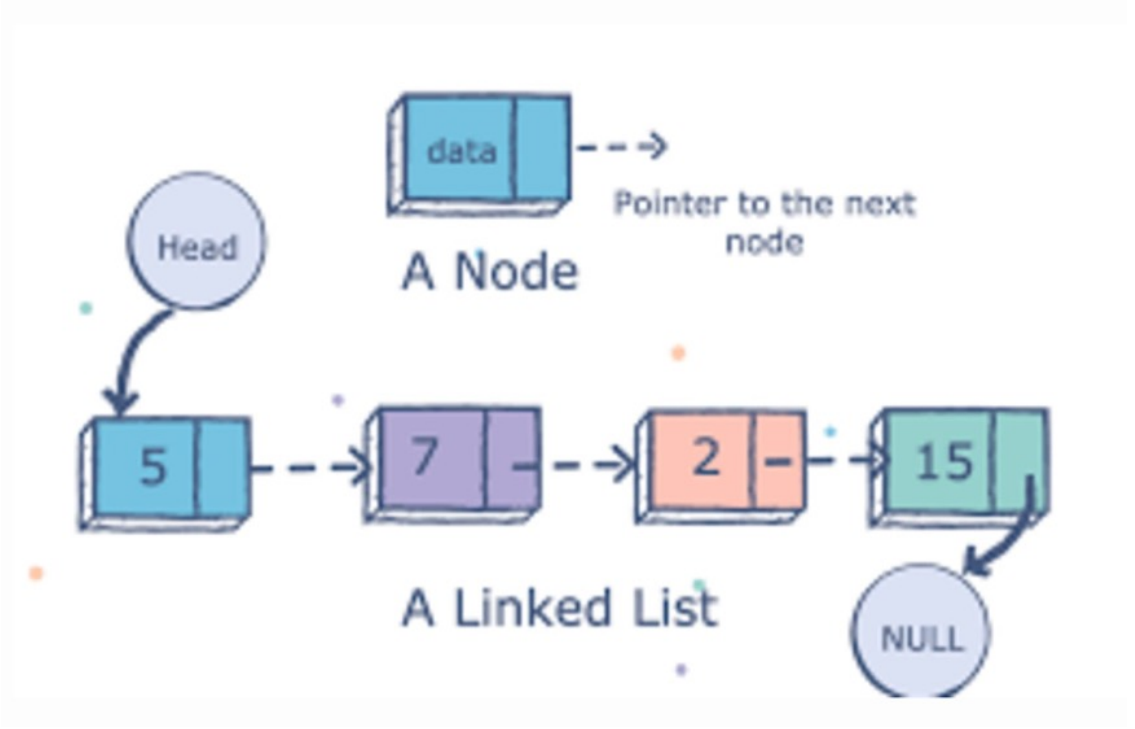
Stack

- A stack is a data structure that is used to store data in a last-in-first-out (LIFO) manner. The last element added to the stack is the first element to be removed.



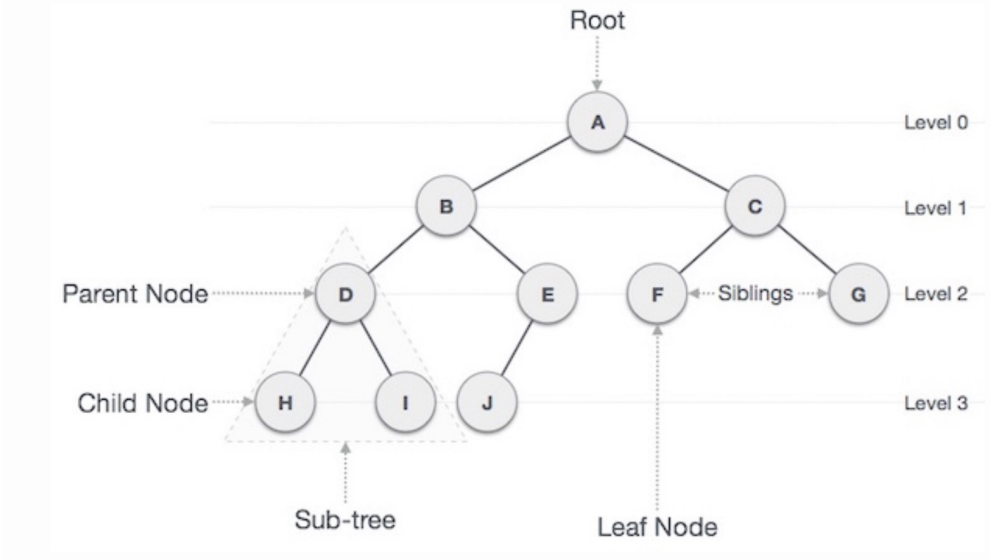
Linked List

- A linked list is a linear data structure that consists of a sequence of nodes, each of which contains a piece of data and a reference to the next node in the sequence.
- The first node in the sequence is called the *head* node, and the last node is called the *tail* node



Tree

- A tree is a graph structure that contains one node as the *root*.
- This root then has an arbitrary number of *children*, which are also nodes.
- The children of the root are called the *branches*.



Dynamic programming

Requirements for DP

- Optimal substructure
- Overlapping subproblems
- How to do it in practice?
  - Memoization
  - Bottom-up problem solving

Divide-and-conquer

- **Divide-and-conquer** method for algorithm design:
- **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
- **Conquer**: conquer recursively to solve the subproblems
- **Combine**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

Divide-and-conquer - Example

- For example, **MergeSort**

```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

- The **subproblems** are **independent**, **all different.**