This assignment is about implementing Kruskal's and Prim-Dijkstra's MST finding algorithms and comparing their execution times on random graphs of varying sizes (number of nodes) and density (number of edges). The report is organized as follows: source code, brief description about the experimental setup, comparison plot, graph generation, and data structures.

Source code

The source code for the implementation of Kruskal's and Prim-Dijkstra's MST finding algorithms with varying inputs to generate the comparison plot along with the required data structures is given below:

```
# --------------------------
# Imports
# --------------------------

import random
import time
import heapq
import matplotlib.pyplot as plt
from collections import defaultdict

# --------------------------
# Graph Generator
# --------------------------

def generate_connected_random_graph(n, m):
    """
    Generates a connected undirected weighted graph with n nodes and m
edges.
    Ensures initial connectivity by forming a linear chain, then adds
extra edges randomly.
    """
    assert m >= n - 1
    W = [[0 for _ in range(n)] for _ in range(n)]  # Adjacency matrix
    edges = []  # Edge list: (u, v, weight)

    # Step 1: Linear chain to guarantee connectivity
    for i in range(n - 1):
        w = random.randint(1, 1000)
```

```
        W[i][i + 1] = W[i + 1][i] = w
        edges.append((i, i + 1, w))


    # Step 2: Add remaining random edges
    count = n - 1
    while count < m:
        i, j = random.randint(0, n - 1), random.randint(0, n - 1)
        if i != j and W[i][j] == 0:
            w = random.randint(1, 1000)
            W[i][j] = W[j][i] = w
            edges.append((i, j, w))
            count += 1

    return W, edges


# --------------------------
# Kruskal's Algorithm
# --------------------------

def find(parent, i):
    # Path compression for Union-Find
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]

def union(parent, rank, x, y):
    # Union by rank to optimize merging
    xroot = find(parent, x)
    yroot = find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1


def kruskal(n, edges):
    """
    Kruskal's MST algorithm using Union-Find and sorted edge list.
    Returns total weight of the MST.
    """
```

```
    parent = list(range(n))
    rank = [0] * n
    edges.sort(key=lambda x: x[2])  # Sort edges by weight

    mst_weight = 0
    edge_count = 0

    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            union(parent, rank, u, v)
            mst_weight += w
            edge_count += 1
            if edge_count == n - 1:
                break

    return mst_weight


# --------------------------
# Prim's Algorithm
# --------------------------

def prim(n, W):
    """
    Prim's MST algorithm using binary min-heap (heapq).
    Operates on adjacency matrix W.
    Returns total weight of the MST.
    """
    visited = [False] * n
    min_heap = [(0, 0)]  # (weight, node)
    mst_weight = 0

    while min_heap:
        w, u = heapq.heappop(min_heap)
        if visited[u]:
            continue
        visited[u] = True
        mst_weight += w
        for v in range(n):
            if W[u][v] != 0 and not visited[v]:
                heapq.heappush(min_heap, (W[u][v], v))

    return mst_weight
```

```python
# --------------------------
# Runtime Measurement
# --------------------------

def measure_time(func, *args):
    """
    Measures the runtime of a given function.
    Returns: (result of function, execution time in seconds)
    """
    start = time.time()
    result = func(*args)
    end = time.time()
    return result, (end - start)


# --------------------------
# Running the Experiment
# --------------------------

def run_experiment():
    """
    Runs MST algorithm experiments across different graph sizes and
densities.
    Repeats 5 times for each (n, m/n) to compute average execution
times.
    """
    sizes = [16, 32, 64, 128, 256]
    densities = [1, 2, 4, 8, 16, 32]
    results = {"kruskal": {}, "prim": {}}

    for n in sizes:
        for d in densities:
            m = min(d * n, n * (n - 1) // 2)  # Upper bound: complete
graph
            kruskal_times = []
            prim_times = []

            for _ in range(5):  # Repeat 5 times per configuration
                W, edges = generate_connected_random_graph(n, m)
                _, k_time = measure_time(kruskal, n, edges)
                _, p_time = measure_time(prim, n, W)
                kruskal_times.append(k_time)
                prim_times.append(p_time)

            # Compute average runtime
```

```
            avg_k = sum(kruskal_times) / 5
            avg_p = sum(prim_times) / 5
            results["kruskal"].setdefault(n, []).append((d, avg_k))
            results["prim"].setdefault(n, []).append((d, avg_p))

    return results


# --------------------------
# Plotting Results
# --------------------------

def plot_results(results):
    """
    Plots average execution time vs. density (m/n) for each algorithm
and graph size.
    """
    plt.figure(figsize=(12, 7))
    for algo in results:
        for n in results[algo]:
            xs = [d for d, _ in results[algo][n]]
            ys = [t for _, t in results[algo][n]]
            plt.plot(xs, ys, marker='o', label=f"{algo.capitalize()}
({n})")

    plt.xscale("log", base=2)
    plt.xlabel("Density (m/n)")
    plt.ylabel("Average Execution Time (s)")
    plt.title("Kruskal vs Prim Execution Time on Random Graphs")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()


# --------------------------
# Main Execution
# --------------------------

results = run_experiment()
plot_results(results)
```

Experimental Setup

Processor: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s)

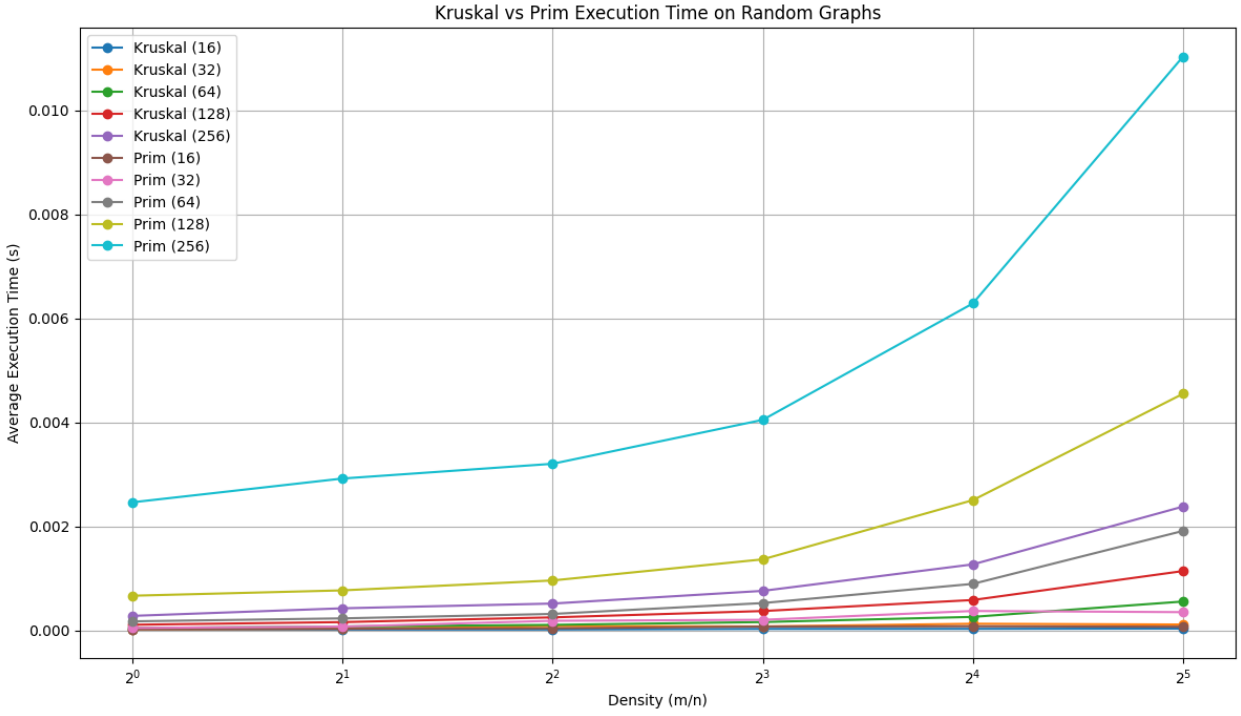
Language: Python

Random Number Generator: random.randint() function using random library in Python.

Comparison Plot

The objective of the plot was to:

- Compare Kruskal's and Prim's MST algorithms.
- Measure their execution times across different graph sizes (n) and densities (m/n).
- Plot average execution time vs. density, with:
    - X-axis = m/n (log scale)
    - Y-axis = execution time in seconds
    - One curve per (n, algorithm) pair



The selected values of n are 16, 32, 64, 128, 256. For each n, the varied m (number of edges) as m = n, 2n, 4n, 8n, 16n, 32n up to complete graph: n(n-1)/2. For each (n, m) pair, 5 random connected graphs were generated. The execution time of both Kruskal's algorithm (using edge list + union-find) and Prim's algorithm (using adjacency matrix + binary heap) were measured and averaged. The plot was done by keeping m/n (density) on the X-axis and execution time on the Y-axis. Each curve is labeled by algorithm and size, e.g., (128, Kruskal).

X-axis (Density m/n):

- Scaled logarithmically with base 2.
- Shows how graph density increases exponentially (1, 2, 4, ..., 32).
- Allows fair visual spacing for rapid growth in edges.

Y-axis (Average Execution Time in seconds):

- Linear scale.
- Plots the actual measured execution time.

The provided plot illustrates the comparative performance of Kruskal's and Prim's minimum spanning tree (MST) algorithms across varying graph sizes and densities. Graph sizes (n) range from 16 to 256, and for each size, edge counts (m) are scaled to reflect increasing density, with the x-axis representing m/n on a logarithmic scale. The y-axis shows the average execution time in seconds, computed from five random connected graphs for each (n, m) pair. Graph generation time was excluded to ensure accurate measurement of algorithm performance.

Each curve corresponds to a fixed n and algorithm, allowing direct visual comparison. The results show that Kruskal's algorithm scales more efficiently with increasing density, particularly in larger graphs, due to its O(E log E) complexity and edge-focused approach. In contrast, Prim's algorithm, implemented with an adjacency matrix and binary heap, demonstrates steeper growth in execution time as both n and m increase which is especially evident for n = 256. It highlights its sensitivity to dense connectivity. Overall, Kruskal outperforms Prim in dense and large graphs, while both perform comparably in smaller or sparser scenarios.

Graph Generation

To evaluate the performance of Minimum Spanning Tree (MST) algorithms such as Kruskal's and Prim's, a consistent and controlled method of graph generation is necessary. The process ensures that the graph is always connected while allowing for randomization in edge weights. The task specifies different strategies depending on the density of the graph, governed by the number of vertices n and edges m.

```
def graph_generation():
    sizes = [16, 32, 64, 128, 256]
    densities = [1, 2, 4, 8, 16, 32]
    results = {"kruskal": {}, "prim": {}}

    for n in sizes:
        for d in densities:
            m = min(d * n, n * (n - 1) // 2)
            kruskal_times = []
            prim_times = []

            for _ in range(5):  # Repeat 5 times per configuration
                W, edges = generate_connected_random_graph(n, m)
                _, k_time = measure_time(kruskal, n, edges)
                _, p_time = measure_time(prim, n, W)
                kruskal_times.append(k_time)
                prim_times.append(p_time)

            # Compute average runtime
            avg_k = sum(kruskal_times) / 5
            avg_p = sum(prim_times) / 5
            results["kruskal"].setdefault(n, []).append((d, avg_k))
            results["prim"].setdefault(n, []).append((d, avg_p))

    return results
```

General Graph Generation Process (For any n and m):

Initialization of the Weight Matrix

The generation begins by initializing an n × n weight matrix W filled with zeros. Each cell W[i][j] represents the weight of an edge from vertex i to vertex j. A value of 0 means that no edge currently exists between those two nodes.

Example: When n = 16, the matrix W is a 16×16 matrix initialized as:

```
W = [[0 for _ in range(n)] for _ in range(n)]
```

This structure allows for efficient lookup and update of weights and will be used to construct both sparse and dense graphs.

Ensuring Connectivity with a Spanning Path

To guarantee that the graph is connected, a deterministic set of n - 1 edges is inserted. These edges form a linear path from node 1 to node n.

Example: For n = 16, the following 15 edges are added:

$$\{1, 2\}, \{2, 3\}, \{3, 4\}, ..., \{15, 16\}$$

In zero-indexed code, this translates to:

```
for i in range(n - 1):
    w = random.randint(1, 1000)
    W[i][i + 1] = w
    W[i + 1][i] = w
```

This step ensures that the graph is fully connected from the start, and no part of the graph is isolated.

Adding Random Edges (When m > n - 1)

Once the base connectivity is established, the remaining m - (n - 1) edges must be added randomly. For each new edge:

- Two random integers i and j are chosen such that $0 \leq i < j < n$.
- A random weight w is generated within the range 1 to 1000.
- The edge {i, j} is only added if it does not already exist in the matrix, i.e., W[i][j] == 0.

This process repeats until exactly m unique edges are present in the graph.

Example: If n = 64 and m = 512 (i.e., density = 8, as m/n = 8), the base path contributes 63 edges. Therefore, 512 - 63 = 449 additional random edges must be generated.

The corresponding code snippet might look like:

```
added = set()
while len(added) < (m - (n - 1)):
    i = random.randint(0, n - 1)
    j = random.randint(0, n - 1)
```

```
    if i != j and W[i][j] == 0:
        w = random.randint(1, 1000)
        W[i][j] = w
        W[j][i] = w
        added.add((min(i, j), max(i, j)))
```

This ensures no duplication and maintains symmetry for the undirected graph.

Generating a Complete Graph Efficiently (m = n(n - 1)/2)

For this special case, the graph must contain every possible pair of nodes, which equates to n(n - 1)/2 edges. Attempting to add edges one by one while checking for duplicates would be highly inefficient.

Instead, the method directly populates the upper triangle of the matrix with unique random weights.

Example: For n = 16, the total number of edges is:

$$\frac{16 \times 15}{2} = 120$$

A total of 120 unique random weights are generated and assigned as follows:

```
weights = random.sample(range(1, 1001), k=(n * (n - 1)) // 2)
idx = 0
for i in range(n):
    for j in range(i + 1, n):
        W[i][j] = weights[idx]
        W[j][i] = weights[idx]
        idx += 1
```

This guarantees full connectivity, symmetric weights, and avoids repeated checking or retries.

Optimizing Generation for Dense Graphs

For dense graphs, where m is large (e.g., m = 992 for n = 64), generating and validating random edges becomes inefficient due to high collision probability in the weight matrix. Instead, it is more practical to begin with a complete graph and delete random edges until the count reaches m.

Procedure:

- Generate a complete graph as above.
- Create a list of all edge pairs excluding the deterministic path {i, i+1} for all i.
- Randomly delete edges from the list while ensuring that none of the critical path edges are removed.

Example: If n = 64, the complete graph contains 2016 edges. If the target m = 1024, then 2016 - 1024 = 992 edges must be deleted.

```
critical_edges = {(i, i + 1) for i in range(n - 1)}
all_edges = {(i, j) for i in range(n) for j in range(i + 1, n)}
deletable = list(all_edges - critical_edges)
to_delete = random.sample(deletable, len(all_edges) - m)

for (i, j) in to_delete:
    W[i][j] = 0
    W[j][i] = 0
```

This ensures that:

- The graph remains connected via the preserved path.
- Edge count matches the desired m.
- Randomness is preserved for the remaining structure.

Data Structures

Kruskal's Algorithm

Kruskal's algorithm builds a minimum spanning tree (MST) by greedily choosing the smallest weight edge at each step, ensuring that no cycles are formed. To achieve this, it uses a min-heap to prioritize edges and a union-find data structure to manage disjoint sets (connected components).

A min-heap to store all the edges prioritized by their weights

A min-heap is used to prioritize edges based on weight. All m edges are inserted into a heap before the MST construction begins. An array H[1..m] is used to hold edge records with fields i, j, and w, representing an edge between nodes i and j with weight w.

Example: For a graph with n = 64 and m = 512, the heap may contain:

H = [{'i': 3, 'j': 9, 'w': 104}, {'i': 12, 'j': 29, 'w': 37}, ...]

Timing Rule: The construction of the heap (inserting all edges and heapifying) must be included in the timing, but gathering edges from the matrix should not be.

N trees representing n connected components in the beginning

The algorithm starts with n disjoint sets (trees), one for each node. It maintains a forest that merges components as edges are added.

Data Structures:

- parent[1..n]: Keeps track of the root (or parent) of each node. Initially, parent[i] = i.
- rank[1..n]: Stores the rank (approximate height) of each tree. Helps to optimize union operations.

Find Operation (C-Find):

- Returns the representative (root) of a node's set.
- Uses path compression to speed up repeated lookups.
- Example: Find(29) might return 12 if node 29 is part of the tree rooted at node 12.

```
def find(parent, i):
    # Path compression for Union-Find
    if parent[i] != i:
        parent[i] = find(parent, parent[i])
    return parent[i]
```

Union Operation (W-Union):

- Merges two sets. Uses rank to attach the smaller tree under the root of the larger one.
- Example: If rank[12] = 3 and rank[29] = 1, then the root of 29 is updated to point to 12.

```
def union(parent, rank, x, y):
    # Union by rank to optimize merging
    xroot = find(parent, x)
    yroot = find(parent, y)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
```

```
        parent[yroot] = xroot
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
```

Avoiding Cycles:

- Before adding an edge {i, j}, Kruskal checks if Find(i) != Find(j).
- If true, the edge is safe to add; otherwise, it would form a cycle.

A variable TotalWeight which will accumulate the weight of the edges in the spanning forest as it is formed

Two additional constructs are used to track the MST. One is **TotalWeight**, a scalar that keeps the cumulative sum of the weights of edges added to the MST.

TotalWeight = 0

An array T [1..n - 1] implementing the set of edges in the spanning tree

T[1..n-1]: An array storing the edges included in the MST. This helps verify correctness.

Example: After completion,

T = [(3, 9), (12, 29), ..., (55, 62)]

Prim's Algorithm

Prim's algorithm incrementally grows the MST from a starting node by always connecting the closest non-MST node. It relies on a min-heap for nodes, an adjacency list for fast lookups, and a NEAR array to track nearest connections.

The array NEAR[1..n]

Prim's algorithm maintains an array NEAR[1..n] where:

- NEAR[i] stores the minimum known cost to reach node i from the growing MST.
- It is initialized to ∞ for all nodes except the starting node.

Example for n = 64:

```
NEAR = [0, inf, inf, ..., inf]  # Start from node 0
```

Min-Heap to Store Nodes

Unlike Kruskal's edge-based heap, Prim's heap stores nodes prioritized by their current closest edge cost.

Structure:

```
heap = [(0, 0)]  # (cost, node)
```

Operations:

- Extract-Min: Removes the closest node to the current MST.
- Decrease-Key: Updates a node's cost if a shorter path is found.

Example:

- Node 27 discovered with new cost 12 → heap.push((12, 27)).
- Later, if an edge with weight 9 to node 27 is found → update heap with heap.decrease_key(27, 9).

Adjacency List Representation

Prim's algorithm needs fast access to neighbors, which is achieved via adjacency lists.

Construction:

```
adj = [[] for _ in range(n)]
adj[3].append((9, 104))  # Edge from 3 to 9 with weight 104
adj[9].append((3, 104))
```

Timing Rule: Building the adjacency list is not included in the timing. Only the MST construction operations should be timed.

Usage:

- After extracting node u from the heap, its neighbors are scanned via adj[u].
- Each neighbor v is considered for addition or update in the heap.

Like Kruskal, Prim also uses:

- **TotalWeight**: Sum of selected edge weights.
- T[1..n-1]: Array of MST edges, e.g., [(0, 3), (3, 9), ..., (55, 62)].

Graph Representation (for Both Algorithms)

Both Kruskal and Prim can work off an adjacency list or adjacency matrix:

- Kruskal: Can scan upper/lower triangle of matrix W[i][j] to gather edges.
- Prim: Requires adjacency list for fast neighbor access.

For n = 64, m = 512:

- The maximum number of unique undirected edges is 2016.
- An average degree is 2m/n = 16, implying moderate density.

Edge Scanning for Kruskal:

- Can be done once at the beginning to populate heap.
- Should not be included in timing.

2 Chapter 9: Sorting in Linear Time

2.1 Counting Sort

(a) Rank each element

i. Count how many times i occurs

ii. Prefix sum on counter array to find how many items $\leq i$

(b) Place at its location.

Start from right, place item in the output array, decrease its corresponding count.

$W(n) = O(n)$ if range is $O(n)$.

Stable sorts – counting sort, mergesort, Insertion sort, Bucket sort.

Nonstable sorts – Quicksort, heapsort.

2.2 Bucket Sort

k Buckets.

Algorithm:

1. hash items among buckets
2. sort the buckets
3. Combine buckets

Let there be k buckets, n items

1. distribution $O(n)$
2. sort the buckets
   $w(n) = O(n \log n)$
   $A(n) = O(k\frac{n}{k} \log \frac{n}{k}) = O(n \log(n/k))$
3. combine buckets $O(n)$.

Thus, bucket sort is

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

$$A(n) = O(n \log \frac{n}{k})$$

If k is constant,

$$A(n) = O(n \log n - n \log k)$$
$$= O(n \log n)$$

$A(n) = O(n)$ if $k = n/20$, $A(n) = O(n)$

Good when item distribution is known so that bucket get equitable number of keys.

Space Usage

worst-case: each bucket should have space for n key (any allocation)

$\Rightarrow$ total $= O(nk)$

Thus, as k increases, average space increases but so does the space requirement.

If linked allocation is used

Space needed $= O(k) + O(n) = O(n + k) = O(n)$

However sorting each bucket using quicksort, mergesort, and heapsort will be difficult which require array representation.

If insertion sort is used to sort linked list, (buckets),

$$A(n) = O(\frac{n^2}{k^2}) * k = O(\frac{n^2}{k}) = O(n) \text{ for k} = \text{O(n)}.$$

2.3 Radix sort

for $i \leftarrow 1$ to d

stable sort A on digit i.

(a) counting sort can be used

(b) bucket sort can also be used

Finding largest: $n-1$ comparisons

Finding second largest: .

- Two scans: $(n-1) + (n-2) = 2n-3$ comparisons

- Divide & Conquer:
  $W(n) = 2W(n/2) + 2$
  $= 2 + 2(2 + 2W(n/2^2))$
  $= 2 + 2^2 + 2^2 W(n/2^2)$
  $= 2 + 2^2 + 2^2(2 + 2W(n/2^3))$

$= 2 + 2^2 + 2^3 + 2^3 W(n/2^3)$

Let $n = 2^k$,

We know, $W(2) = 1 \Rightarrow W(n/2^{k-1}) = 1$

$W(n) = 2 + 2^2 + 2^3 + \cdots + 2^{k-1} + 2^{k-1} W(n/2^{k-1})$

$= 2^1 + 2^2 + \cdots + 2^{k-1} + 2^{k-1}$

$= (2^k - 2) + 2^{k-1}$

$= n - 2 + n/2$

$= 3n/2 - 2$

- Using Tournament Tree

  To build a tournament tree requires $n - 1$ comparisons.

  $W(n) = n - 1$ for max

  $W(n) = n - 1 + (\log_2 n - 1)$ for 2max

2.1 Selection of kth smallest/largest

1. Sorting based

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

2. Tournament tree based

   $W(n) = O(n + k \log n) = O(n)$ for $k \leq \frac{n}{\log n}$ or $k \geq \frac{n}{\log n}$

2.2 A Good Av. Case Algorithm

Divide & conquer

Selection$(A[p,r],k)$:

$j =$ Partition2(A,p,r)

if $k < j$

return Selection$(A[p..(j-1)],k)$

else if k = j **then return** $L[j]$

else $\{k > j\}$

return Selection$(A[j+1..r],k-j)$

$W(n) = n - 1 + W(n-1) = O(n^2)$

$A(n) \approx n - 1 + A(n/2)$

$= \ (n-1) + (n/2 - 1) + (n/4 - 1) + \cdots + 1$

$< 2n \in O(n)$

(gross simplification)

2.3 Worst-case $O(n)$ algorithm

To improve $W(n)$, we must ensure a good split point.

Selection'$(A[p, r], k)$

1. Divide A in $\frac{n}{r}$ sublist ($r = 5, 7$, etc.), $n = r - p + 1$.
2. Find median of each of the $\frac{n}{r}$ sublists.
3. Recursively find median of these $\frac{n}{r}$ medians.
4. Use median of medians (MM) as the pivot in the previous algorithm for selection:

   Let MM be at index i. Swap$(A[p], A[i])$

$j = \text{Partition2}(A, p, r)$

5. Choose the appropriate partition for further search:

if $k < j$

return Selection'$(A[p..(j-1)], k)$

else if k=j **then return** $L[j]$

else $\{k > j\}$ **return** Selection'$(A[j+1..r], k-j)$

$T(n) \leq cn$ (Steps 1, 2, 4: for finding n/r medians and for partitioning the array based on pivot chosen as median of medians)

$+ T(n/5)$ (for step 3, recursively finding median of $n/5$ medians)

$+ T(3n/4)$ (for recursive call to larger partition)

To Prove: Let

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right)$$

for $n \geq 5$. Then

$$T(n) \leq 20cn.$$

Choose c large enough such that $T(n) \leq cn$ for $n \leq 24$ (for list of size less than 5, there is no recursive call).

Basis: For $n \leq 24$, by choice of c, $T(n) \leq cn \leq 20cn$.

Hypothesis: Assume for $k \geq 24$, $T(k) \leq 20ck$.

Induction:

To show that $T(k+1) \leq 20c(k+1)$.

$$T(k+1) \leq c(k+1) + T\left(\frac{k+1}{5}\right)$$

$$
\begin{aligned}
T\left(\frac{3(k+1)}{4}\right) & \\
& \leq c(k+1) \ + \ 20\left(\frac{k+1}{5}\right)c \\
& \quad 20\left(\frac{3(k+1)}{4}\right)c \\
& = (k+1)(c+4c+15c) \\
& = 20(k+1)c
\end{aligned}
$$

How to make quicksort $O(n \log n)$? Use selection algo to find the median.

Use median as partitioning element in Quicksort.

Complexity $T(n) = cn + 2T(n/2) = O(n \log n)$

(MATROID)

STRATEGY GREEDY

```
Solution ← ∅
for i ← 1 to n do
    SELECT the next input x.
    If {x} ∪ Solution is
    FEASIBLE then
        Solution ← COMBINE (Solution, x)
```

Select appropriately finds the next input to be considered.

A feasible solution satisfies the constraints required for the output.

Combine enlarges the current solution to include a new input.

Strategy Divide-and-Conquer.

Divide problem P into smaller problems $P_1, P_2, \ldots, P_k$.

Solve problems $P_1, \ldots, P_k$ to obtain solutions $S_1, S_2, \ldots, S_k$

Combine solution $S_1, S_2, \ldots, S_k$ to get the final solution.

Subproblems $P_1, P_2, \ldots, P_k$ are solved **recursively** using divide-and-conquer.

2 STRATEGY DIVIDE-AND-CONQUER

- Divide Problem P into smaller problem $P_1, P_2, \ldots, P_k$.
- Solve problems $P_1, P_2, \ldots, P_k$ to obtain solutions $S_1, S_2, \ldots, S_k$
- Combine solution $S_1, S_2, \ldots, S_k$ to get the final solution.

Subproblems $P_1, P_2, \ldots, P_k$ are solved recursively using divide-and-conquer.

Examples: Quicksort and mergesort.

3 STRATEGY GREEDY

Solution $\leftarrow \Phi$

for $i \leftarrow 1$ to n do

SELECT the next input x.
If $\{x\} \cup$ Solution is **FEASIBLE** then
solution $\leftarrow$ **COMBINE**(Solution, x)

- **SELECT** appropriately finds the next input to be considered.
- A **FEASIBLE** solution satisfies the constraints required for the output.
- **COMBINE** enlarges the current solution to include a new input.

Examples: Max finding, Selection Sort, and Kruskal's Smallest Edge First algorithm for Minimum Spanning Tree.

4 STRATEGY DYNAMIC PROGRAMMING

- Fibonacci Numbers:

$$F_n = F_{n-1} + F_{n-2}$$

$F_1 = F_0 = 1$.

— Recursive solution requires exponential time: has **overlapping subproblems.**

— **Bottom-up** iterative solution is linear – **compute once, store, and use many times.**

4.1 Matrix Sequence Multiplication

- **eg. 1:**

  $A_{30x1}\ X\ B_{1x40}\ X\ C_{40x10}\ X\ D_{10x25}\ X\ E_{25x1}$

  — Left to right evaluation requires more than 12K multiplications.

  — $(A\ X\ (\ (B\ X\ C)\ X\ (D\ X\ E)\ )\ )$ needs only 690 multiplications (minimum needed).

  — **Greedy Algorithm: Largest Common Dimension First**

- **eg. 2:**

  $A_{1x2}\ X\ B_{2x3}\ X\ C_{3x4}\ X\ D_{4x5}\ X\ E_{5x6}$

  — Largest Common Dimension First imposes following order:

  $(A\ X\ (\ B\ X\ (C\ X\ (D\ X\ E)\ )\ )\ )$

  which needs 240 multiplications.

— Best order:

$$((((A\ X\ B)\ X\ C)\ X\ D)\ X\ E)$$

which needs 68 multiplications.

— **Another Greedy Algorithm: Smallest Common Dimension First** but did not work for eg. 1.

4.2 Divide and Conquer Solution

Input: $\begin{matrix} A_1 & * & A_2 & \ldots & * & A_n \\ d_0 * d_1 & & d_1 * d_2 & \ldots & & d_{n-1} * d_n \end{matrix}$

Output: A paranthesization of the input sequence resulting in minimum number of multiplications needed to multiply the n matrices.

- **Subgoal:** Ignore Structure of Output (order of parenthesization), focus on obtaining a numerical solution (minimum number of multiplications)
- Define **M[i,j]** = the minimum number of multiplications needed to compute

$$A_i * A_{i+1} * \cdots * A_j$$

for $i \leq j \leq n$

- Subgoal is to obtain $M[1, n]$.

e.g. For,

$A1_{30x1}\ X\ A2_{1x40}\ X\ A3_{40x10}\ X\ A4_{10x25}\ X\ A5_{25x1}$

$M[1,1] = 0, M[1,2] = 1200, M[1,5] = 690$

4.3 Recursive Formulation of $M[i, j]$

- A1 X A2 = (A1) X (A2)

  — Partition at k=1: Subproblems (A1) and (A2)

  — cost of (A1) is $M[1, 1]$ and that of (A2) is $M[2, 2]$

  — cost of combining (A1) and (A2) into one is $d_0 * d_1 * d2$.

  — $M[1, 2] = M[1, 1] + M[2, 2] + d_0 * d_1 * d2$.

  — $M[1, 2] = 0 + 0 + 1200 = 1200$.

- A2 X A3 X A4 = (A2) X (A3 X A4) *(k=2)*
  Or, = (A2 X A3) X A4 *(k=3)*.

  — $k = 2$: cost= $M[2, 2] + M[3, 4] + d1 * d2 * d4$

  — $k = 3$: cost= $M[2, 3] + M[4, 4] + d1 * d3 * d4$

  — $M[2, 4] = min(M[2, 2] + M[3, 4] + d1 * d2 * d4, M[2, 3] + M[4, 4] + d1 * d3 * d4)$

In short, $M[2,4] =$
$\min_{2 \leq k \leq 3} (M[2,k] + M[k,4] + d_1 d_k d_4)$

- A2 X A3 X A4 X A5
  = (A2) X (A3 X A4 X A5) *(k=2)*
  Or, = (A2 X A3) X (A4 X A5) *(k=3)*
  Or, = (A2 X A3 X A4) X (A5) *(k=4)*

  $M[2,5] = (M[2,2] + M[3,5] + d_1 d_2 d_5, M[2,3] +$
  $M[4,5] + d_1 d_3 d_5, M[2,4] + M[5,5] + d_1 d_4 d_5)$

- In general, by factoring $(A_i * A_{i+1} * \cdots * A_j)$ at kth index position into $(A_i * A_{i+1} * \cdots * A_k)$ and $(A_{k+1} * \cdots * A_j)$ need $M[i,k] + M[k+1,j]$ multiplications and creates matrices of dimensions $d_{i-1} * d_k$ and $d_k * d_j$. These two matrices need additional $d_{i-1} * d_k * d_j$ multiplications to combine.

4.4 Recursive Formula and Time Taken

- Recursively,
  $M[i,j] =$

$$\min_{i \leq k \leq j-1} \left(M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right)$$

$$M(i,i) = 0$$

- **Optimal Substructure**

- We can recursively solve for
  $M[1,n] =$
  $\min_{1 \le k \le n-1} (M[1,k] + M[k+1,n] + d_0 d_k d_n)$
  $= min[M[1,1] + M[2,n] + d_0 d_1 d_n,$
  $M[1,2] + M[3,n] + d_0 d_2 d_n,$
  $M[1,3] + M[4,n] + d_0 d_3 d_n,$
  $\vdots$
  $M[1,n-1] + M[n,n] + d_0 d_{n-1} d_n$

- Time Complexity:

$$\begin{aligned} T_n = n \; & + T_1 + T_{n-1} \\ & + T_2 + T_{n-2} \\ & + T_3 + T_{n-3} \\ & \vdots \\ & + T_{n-2} + T_2 \\ & + T_{n-1} + T_1 \end{aligned}$$

$$T_n = n + 2T_1 + 2T_2 + \cdots + 2T_{n-1} \quad (1)$$
$$T_{n-1} = n - 1 + 2T_1 + 2T_2 + \cdots + 2T_{n-2} \quad (2)$$

Subtracting (I)-(II) yields

$$\begin{aligned} T_n - T_{n-1} &= 1 + 2T_{n-1} \\ T_n &= 1 + 3T_{n-1} \\ &= 1 + 3(1 + 3T_{n-2}) \\ T_n &= 1 + 3 + 3^2 + 3^3 + \cdots + 3^{n-1}T_1 \\ &= 1 + 3 + 3^2 + 3^3 + \cdots + 3^{n-2} \end{aligned}$$

- Recursive Solution is exponential time $\Omega(3^{n-2})$
- Space $O(n)$ stack depth.
- **Overlapping subproblems:** e.g. Recursion tree for $M[1,4]$.
  26 recursive calls for just 10 subproblems
  $M[1,1], M[2,2], M[3,3], M4,4], M[1,2], M[2,3], M$

So we turn to dynamic Programming,

- the same formulation
- approach the problem bottom-to-top
- find a suitable table to store the sub-solutions.

How many sub-solutions do we have?

$$
\begin{array}{ll}
M[1,1], M[2,2], M[3,3] \cdots, M[n,n] & n \\
M[1,2], M[2,3], \cdots, M[n-1,n] & n-1 \\
M[1,3], M[2,4], \cdots, M[n-2,n] & n-2 \\
\vdots & \\
M[1,n-1], M[2,n] & 2 \\
M[1,n] & 1
\end{array}
$$

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

$\Rightarrow$ we need $O(n^2)$ space

MATRIX ORDER

M, FACTOR : MATRIX

for i ← 1 to n do M[i,i] ← 0
{* main diagonal *}

for diagonal ← 1 to n-1 do
  for i ← 1 to n-diagonal do
    j ← i + diagonal
    $M[i,j] = \min_{i \le k \le j-1} [M[i,k] + M[k+1,j] + d_{i-1} d_k d_j]$
    factor[i,j] = k that
      gave the min value for
      M[i,j].
  endfor
endfor

$A_1 \times A_2 \times A_3 \times A_4 \times A_5$

$30 \times 1 \quad 1 \times 40 \quad 40 \times 10 \quad 10 \times 25 \quad 25 \times 1$

	1	2	3	4	5
1	0	1200 (1)	700 (1)	1400 (1)	690 (1)
2		0	400 (2)	650 (3)	660 (3)
3			0	10000 (3)	650 (3)
4				0	250 (4)
5					0

$$M[1,2] = \min_{1 \le k \le 1} \left[M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right]$$

$$= \min \left[M[1,1] + M[2,2] + d_0 d_1 d_2 \right]$$

$$= 0 + 0 + 30 \times 1 \times 40$$

$$= 1200$$

$$M[1,3] = \min_{1 \le k \le 3-1} \left[M[i,k] + M[k+1,j] + d_{i-1} d_k d_j \right]$$

$$= \min \left[M[1,1] + M[2,3] + d_0 d_1 d_3, \; M[1,2] + M[3,3] + d_0 d_2 d_3 \right]$$

$$= \min \left[0 + 400 + 30 \times 1 \times 10, \; 1200 + 0 + 30 \times 40 \times 10 \right]$$

$$= \min [700, 12000 + 1200] = 700$$

4.5 Matrix Parenthesization Order

M,Factor: Matrix

for $i \leftarrow 1$ to n do $M[i,i] \leftarrow 0$
  /* main diagonal*/

for diagonal $\leftarrow 1$ to $n-1$ do

  for $i \leftarrow 1$ to $n-$diagonal do

    $j = i +$ diagonal

    $M[i,j] = \min_{i \leq k \leq j-1}(M[i,k]+M[k+1,j]$
      $+d_{i-1}d_k d_j)$

    Factor$[i,j] = k$ that gave the minimum value
      for $M[i,j]$.

  endfor

endfor

4.6 Work out

$A1_{30x1}\ X\ A2_{1x40}\ X\ A3_{40x10}\ X\ A4_{10x25}\ X\ A5_{25x1}$

$M[1,2] = \min_{1 \le k \le 1}[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$

$= min[M[1,1] + M[2,2] + d_0 d_1 d_2]$

$= 0 + 0 + 30 * 1 * 40$

$= 1200$

$M[1,3] = \min_{1 \le k \le 3-1}[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$

$= min[M[1,1] + M[2,3] + d_0 d_1 d_3,$

$M[1,2] + M[3,3] + d_0 d_2 d_3]$

$= min[0 + 400 + 30 * 1 * 10,$

$1200 + 0 + 30 * 40 * 10]$

$= min[700, 12000 + 1200]$

$= 700$

$M[2,4] = min[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$
$2 \leq k \leq 3$
$= min[M[2,2] + M[3,4] + d_1 d_2 d_4,$
$M[2,3] + M[4,4] + d_1 d_3 d_4]$
$= min[0 + 10000 + 1 * 40 * 25, 400 + 0 + 1 * 10 * 25]$
$= min[10100, 650] = 650$

$M[3,5] = min[M[i,k] + M[k+1,j] + d_{i-1}d_k d_j]$
$3 \leq k \leq 4$
$= min[M[3,3] + M[4,5] + d_2 d_3 d_5,$
$M[3,4] + M[5,5] + d_2 d_4 d_5]$
$= min[0 + 250 + 40 * 10 * 1, 10000 + 0 + 40 * 25 * 1]$
$= min[650, -] = 650$

$M[1,4] = min[M[1,1] + M[2,4] + d_0 d_1 d_4,$
$M[1,2] + M[3,4] + d_0 d_2 d_4,$
$M[1,3] + M[4,4] + d_0 d_3 d_4]$
$= min[0 + 650 + 30 * 1 * 25,$
$1200 + 10000 + 30 * 40 * 25,$
$700 + 0 + 30 * 10 * 25]$
$= min[1400, -, -] = 1400$

5 DYNAMIC PROGRAMMING REQUIREMENTS

Requirements: **a)** Optimal Substructure

b) Overlapping subproblem

Steps: **1)** Characterize the structure of an optimal solution

2) Formulate a recursive solution

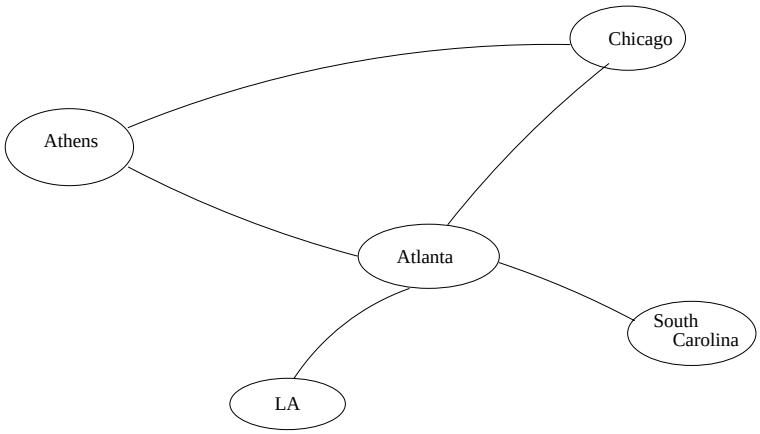
3) Compute the value of *an* opt. solution bottom-up. (get value rather than the structure)

4) Construct an optimal solution (structure) from computed infomation.

Memoization: Top-down, compute and store first time, reuse subsequent times.

2 GRAPHS

Consists of a set of nodes or point some of which are connected by edges or lines.



A (hypothetical) graph of nonstop airline flights.

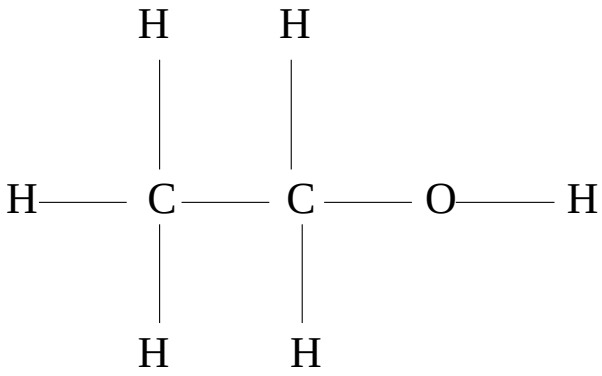
Molecule alcohol: CH_3CH_2OH

Def. A graph $G = (V, E)$ which V is the set of nodes,

$$V = \{v_1, v_2, \ldots, v_n\}$$

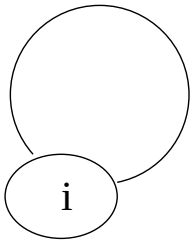
and E is the set of edges,

$$E = \{\{v_i, v_j\} | v_i \in V, v_j \in V\}$$



note:.

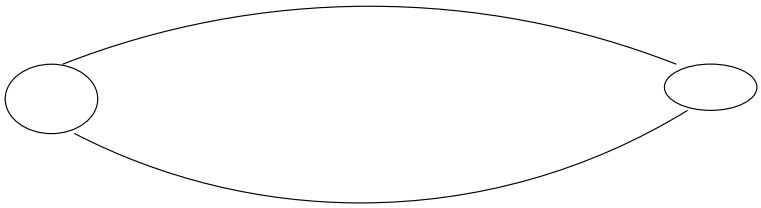
1. each edge is a set of pair of vertexes. No order implied.
2. $v\{v_i, v_j\}$, i could be equal to j.
   self-loop



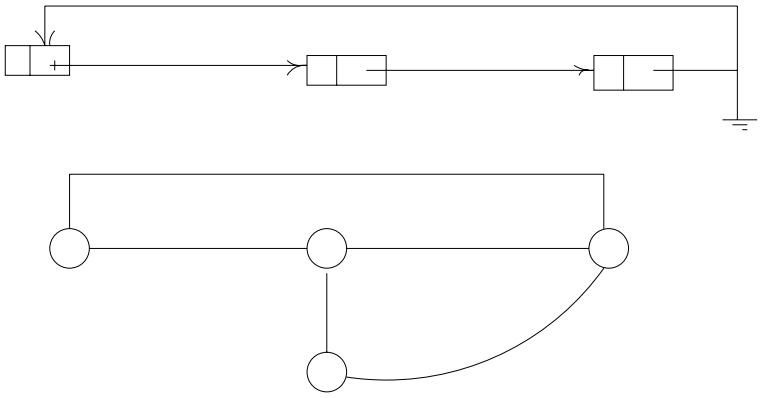
self loop

3. One also allows multiple edge between two node in a general graph

4. Cardinality of V, $|V| = n$, number of nodes, $|E| = m$, number of edges.



3 DIGRAPH



eg. A flow chart of a program

def. A graph whose edges are directed is a digraph, directed graph.
def. A digraph $G_2(V, E)$ when V is the set of vertex $V = \{v_1, v_2, \ldots, v_n\}$ and E is the set of directed edges on every $E = \{(v_i, v_j) such that v_i, v_j \in V\}$

Note

For the purpose of this chapter, we assume that V, the set of nodes, is nonempty & finite and that there no self loops or multiple edges in a graph or digraph.

Question

1. which route is cheapest?
2. which route is fastest?
3. if a node or a computer goes down in a computer network, does it get disconected?
4. is there a loop in a flowchart?

3.1 Subgraph

$V' \subseteq V, E' \subseteq E$
Induced Subgraph by V'
$G' = (V', E'), E' = \{u, v | u, v \in V'\}$

Complete graph
$E = \{\{v_i, v_j | 1 \leq j \leq n\}$ eg. if (v, w) is an edge then v & w are adjacent to each other and v and w are said to be incident with the edge (v, w).

def A path from v to w is a sequencs of vertex $v_0, V_1, v_2, \ldots,$ such that $v_0 = v, v_k = w$& $\{V_i, v_j\} \in E$.
(Books defenition is not good)
If $v_0 = v, v_k = w$& all $v_0, v_1, \ldots, v_k$ are distinct, then the length of the path is k. v_0 is a path of length 0.

Connected Graph:

Cycle is path $v_0v_1v_2 \ldots v_k$ such that $v_0 = v_k$
A graph without any cycle is called acylic.

Tree: Acyclic connected graph

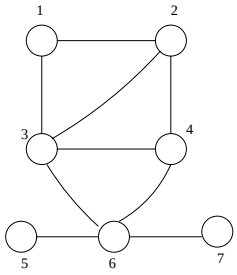
Rooted tree has a designated root vertex establishing parent & child relationship
number of edge in a tree is $n - 1$
Could be proved by induction
A connected component of a graph G is a maximal connected subgraph of G

Weighted Graph
$G = (V, E, w)$
$W : E \leftarrow 2+$
$w(e)$ weight of e (Capacity or distance)

3.2 Representation of a graph

3.2.1 Adjacency Matrix

$A = (a_{ij})_{n*n}$
$a_{ij} = 1$ if $v_i, v_j \in E$
0 else
for $1 \leq i, j \leq n$

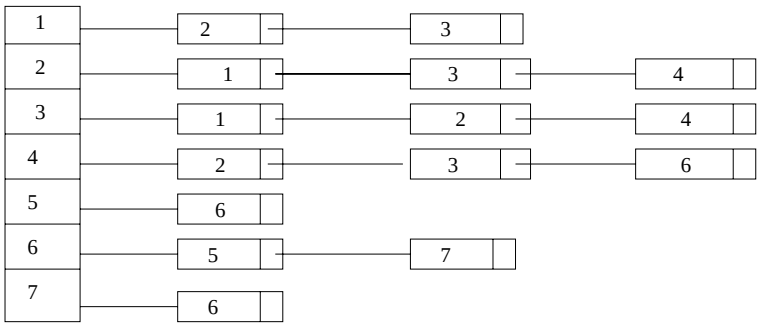


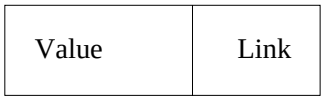
	1	2	3	4	5	6	7
1	0	1	0	1	0	0	0
2	1	0	1	1	0	0	0
3	1	1	0	1	0	0	0
4	0	1	1	0	0	1	0
5	0	0	0	0	0	1	0
6	0	0	1	1	0	1	1
7	0	0	0	0	0	1	0

Space usage $\Leftarrow O(n^2)$

3.2.2 Adjacency list

On any $A[1 \ldots n]$ of linked list, $A[j]$ is the linked list of all the vertices adjacent to v_j.



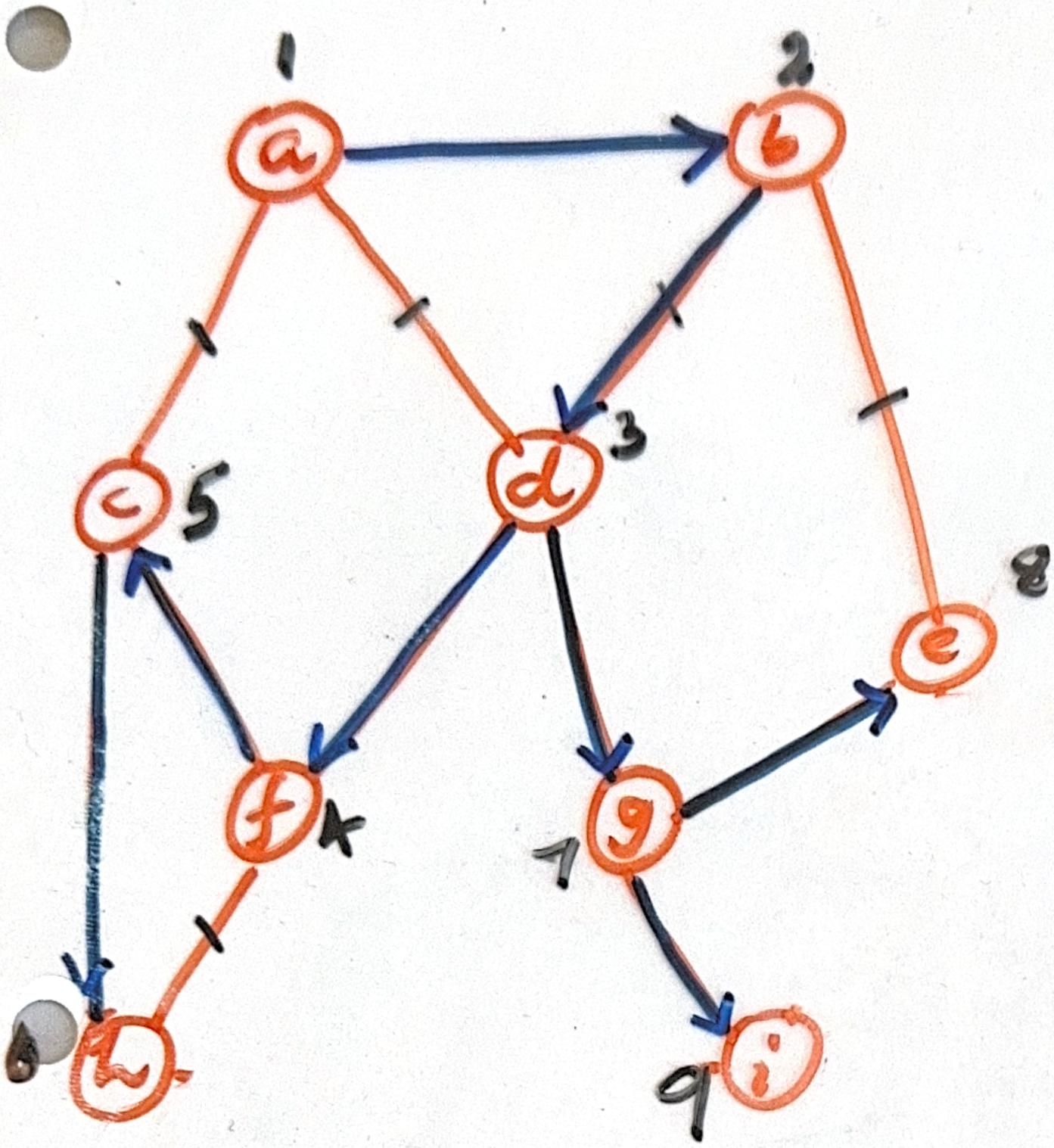


Space usage $\Leftarrow O(n) + O(m) = O(m + n)$
$= O(n^2)$ if m is $O(n^2)$

3.3 Weighted Graph Represatation

Adjacency Matrix $a_{ij} = W(v_i, v_j)$ if $\{v_i v_j\} \in E$
$= 0$ otherwise
Adjacent link

Traversing Graphs



→ Tree Edge

⊣ Back Edge

DFS (v) : Depth-First Search

mark v

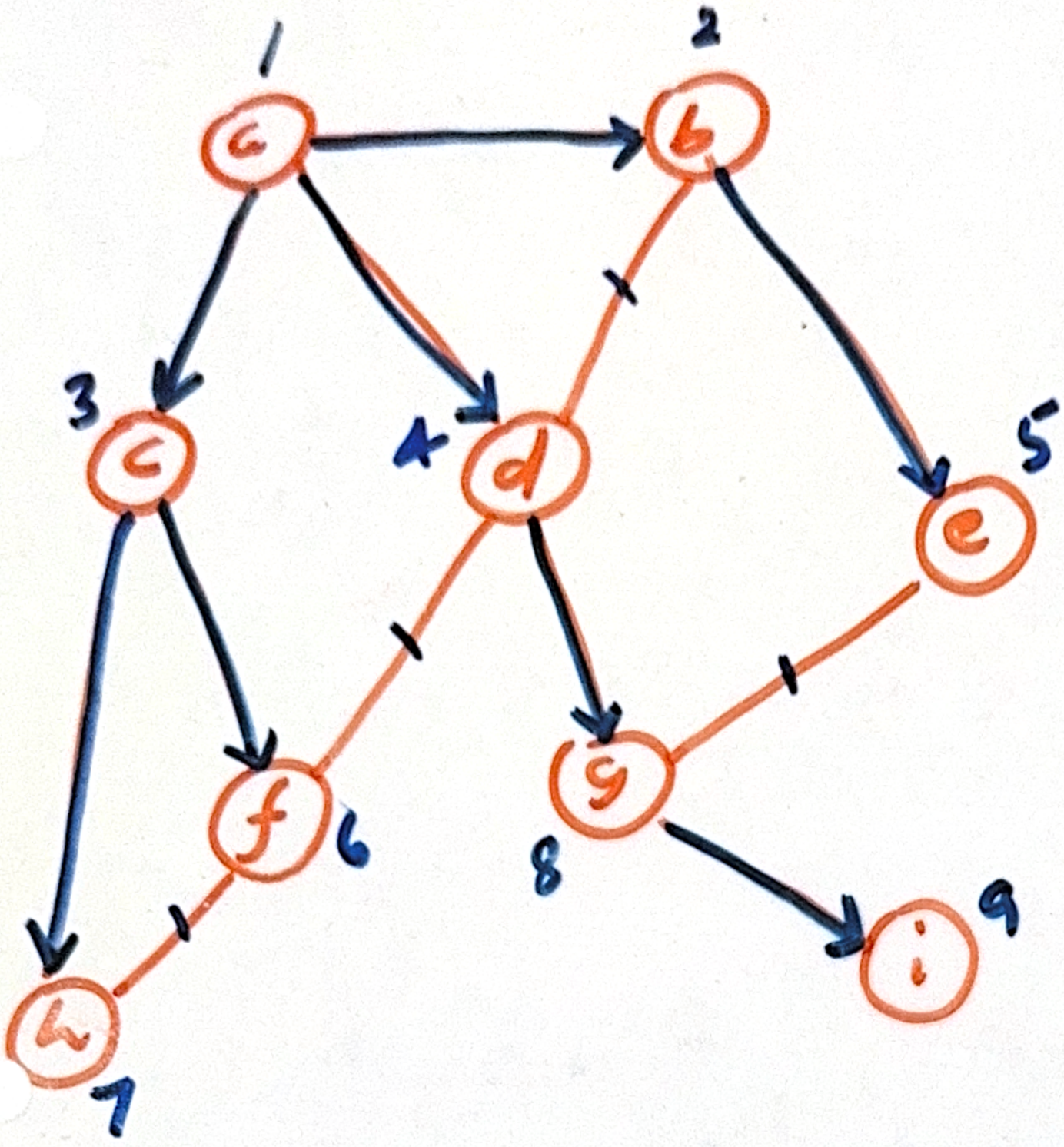
while there is an unmarked node

w adjacent to node v

DFS(w)

end

[Connected Components]



$O(m+n)$

+ cross edges

BFS (v) : Breadth-First SEARCH

mark v

Q = Qv

Q

v

while Q is nonempty do

x = deleteFront (Q)

x

for each unmarked node

w adjacent to x do

{ mark w

Q = Qw

end

4 TRAVERSING GRAPHS

DFS(v): Depth-First Search
mark v
while there is an unmarked node w adjacent to node v DFS(w)
end

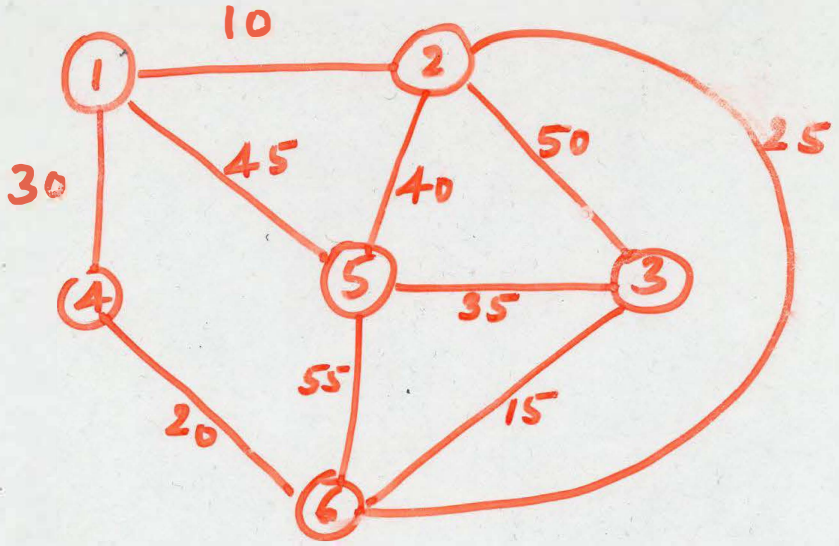
connected components
$O(m+n)$

BFS(v): Breadth-First Search
mark v
Q=Qv
while Q is nonempty do

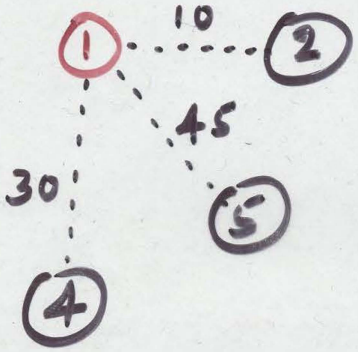
x = delete Front(Q)

for each unmarked node w adjacent to x do mark w
Q=Qw

PRIM-DIJKSTRA'S MST ALGORITHM



PARTIAL MST

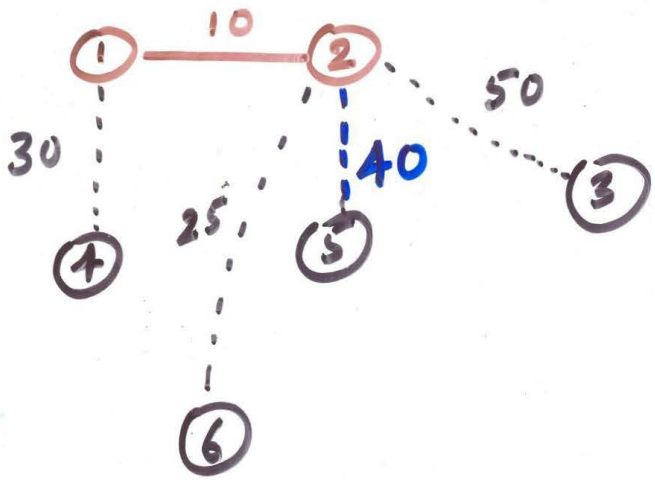


Fringe Vertices

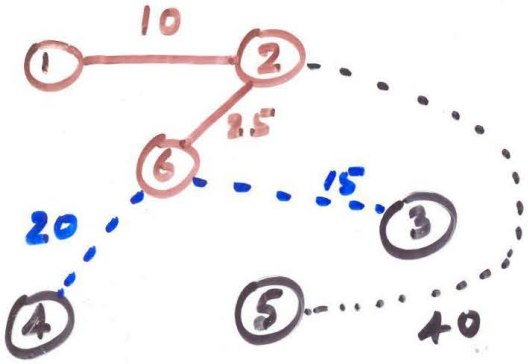
3 unseen vertices

6

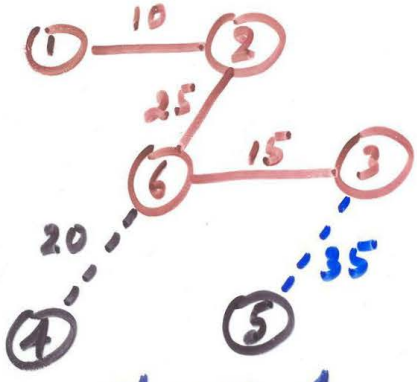
Choose node 2



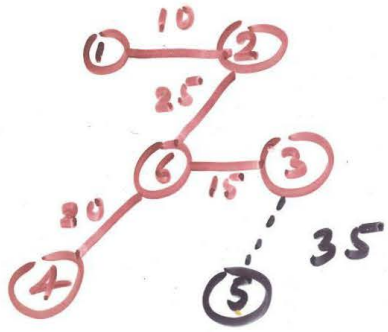
choose node 6



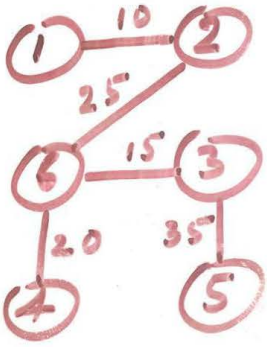
choose node 3



choose 4



choose 5



MST
W = 105

PRIM-DIJKSTRA MST ALGORITHM

Input: $G = (V, E, w)$

Output: $T = (V_T, E_T)$

$E_T = \emptyset$

Select a vertex $v \in V$ and move v to V_T: $V_T = \{v\}$, $V = V - \{v\}$

For $i := 1$ to $n-1$ do

$\Theta(n^2)$ Let $\{v, w\}$ be an edge such that $v \in V_T$, $w \in V$, and for all such edges, $\{v, w\}$ has the minimum weight.

$V_T = V_T \cup \{w\}$

$E_T = E_T \cup \{\{v, w\}\}$

$V = V - \{w\}$

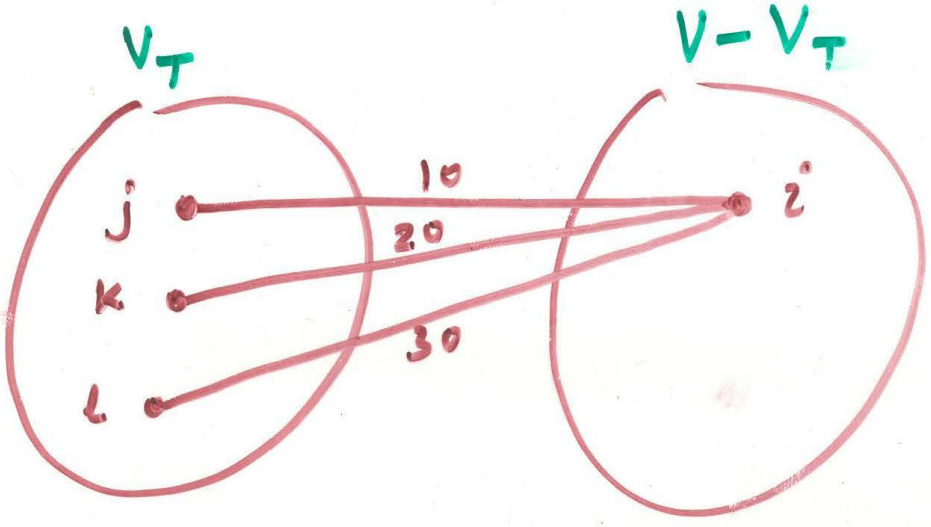
endfor

$W(n) = O(n^3)$

DATA STRUCTURE FOR PRIM'S ALGORITHM

NEAR[1..n]

$$NEAR[i] = \begin{cases} 0 & \text{if } i \in V_T \\ j & \text{if } w(i,j) \text{ is minimum among all } j \in V_T \end{cases}$$



NEAR[i] = j

NEAR[j] = 0

NEAR[l] = 0

MODIFIED PRIM'S ALGORITHM

Input $G = (V, E, w)$, a connected weighted graph

Output $T = (V_T, E_T)$, a MST.

1. $E_T = \emptyset$
2. NEAR[1] = 0 /* $V_T = \{1\}$ */
   NEAR[2..n] = 1 /* $V = V - \{1\}$ */
3. FOR i = 1 to n-1 do

   $O(n)$ $O(\log n)$ FIND j such that NEAR[j] ≠ 0 AND w(j, NEAR[j]) is min.

   NEAR[j] = 0 /* $V_T = V_T \cup \{j\}$ */

   $E_T = E_T \cup \{\{j, NEAR[j]\}\}$.

   /* update NEAR[1..n] */

   for k = 1 to n do

   $O(n \log n)$ DECREASE-KEY if NEAR[k] ≠ 0 AND w(k, NEAR(k)) > w(k, j) then NEAR[k] = j ←

   endfor

endfor.

$w(n) = O(n^2)$

Thm Let $G=(V,E,w)$ be a weighted connected graph and $T=(V,E_T)$ be a MST of G. Let $T'=(V',E')$ is a subtree of T. If $\{x,y\}$ is the minimum weight edge such that

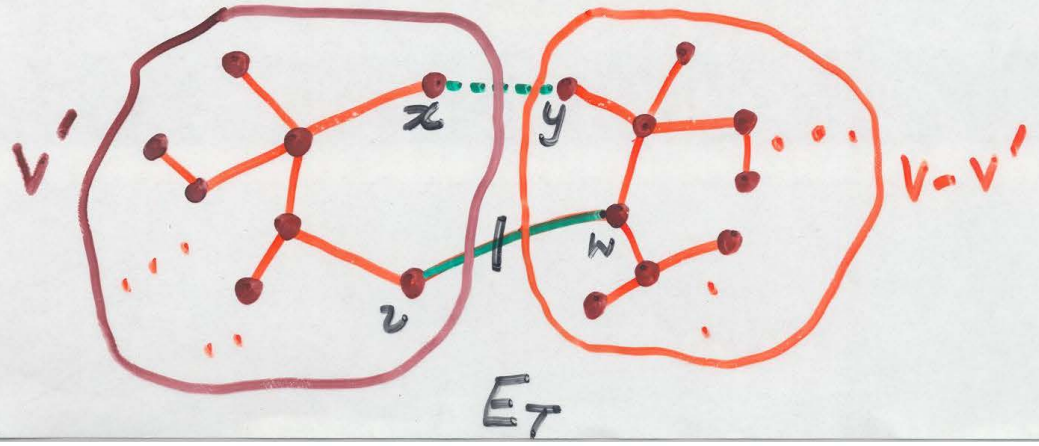
$x \in V'$ and $y \in V-V'$

then $T''=(V' \cup \{y\}, E' \cup \{\{x,y\}\})$ is a subtree of a MST of G.

Proof

1. If $\{x,y\} \in E_T$, done.
2. Let $\{x,y\} \notin E_T$.

   Then $E_T \cup \{\{x,y\}\}$ has a cycle:



By the choice of $\{x, y\}$

$$w(\{x, y\}) \leq w(\{v, w\})$$

Consider $E_T \cup \{\{x, y\}\} - \{\{v, w\}\}$

Its weight is no more than the weight of T and it is a spanning tree.

$\Rightarrow E' \cup \{\{x, y\}\}$ is a subtree of a MST of G.

□

Kruskal's Algorithm

input: $G = (V, E, W)$, a connected graph

output: $T = (V, E_T)$, a MST of G.

$T \leftarrow \emptyset$

while $|T| < n-1$ do

Let $\{v, w\}$ be the least cost edge in E.

$E \leftarrow E - \{\{v, w\}\}$

if $\{v, w\}$ does not create a cycle in T

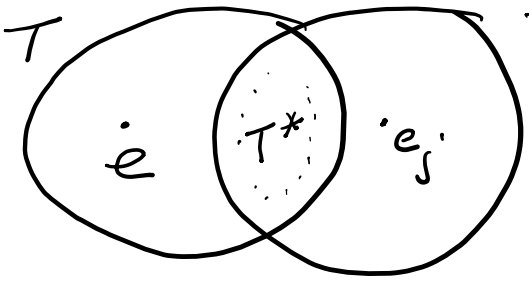
then add $\{v, w\}$ to T

end while

Kruskal's MST Algorithm
Proof of Correctness

Th: Let T be the spanning tree for G generated by kruskal's algorithm. Let T' be a minimum cost spanning tree for G. Show that $W(T) \leq W(T')$. and T is a MST.

Proof:



Let $e \in T - T'$ of min weight.

$\{e\} \cup T'$ has a cycle.

Let $e_j \in T' - T$ in this cycle.

$W(e_j) \geq W(e)$ else $e_j \in T$ by kruskal's

{Why? e_j would have been considered before e for inclusion into a subset $T^* \subseteq T \cap T'$

$\Rightarrow \{e_j\} \cup T^*$ would not form cycle because $T^* \subset T'$.

$\Rightarrow W(T') \geq W(\underbrace{T' - \{e_j\} \cup \{e\}}_{T''}) \geq W(T)$

$\Rightarrow T$ is a MST. $\square$

Repeat for each $e \in T - T'$ $\Rightarrow$ T' becomes T

Kruskal's Algorithm (Refined)

OPERATIONS

1. UNION(i,j), of sets i & j, contains elements of sets i and j.
2. FIND(v) = i iff v ∈ set i.

a) Construct a min-heap E of edges in E.

b) Each v ∈ V forms singleton set by itself, such that FIND(v) = v.

c) $E_T = \emptyset$ {Tree is empty}

d) while $|E_T| < n-1$ do

log m Delete the root edge {v,w} from min-heap E and restore heap E.

O(1) O(log n) if FIND(v) ≠ FIND(w) {$E_T \cup \{\{v,w\}\}$ has no cycle}

then

O(1) $E_T = E_T \cup \{\{v,w\}\}$

/* now, combine the components of E_T joined by {v,w} into one */

O(1) UNION (FIND(v), FIND(w))

end if

end while O(m log m)

1. UNION (1, 2)

2. UNION (2, 3)

⋮

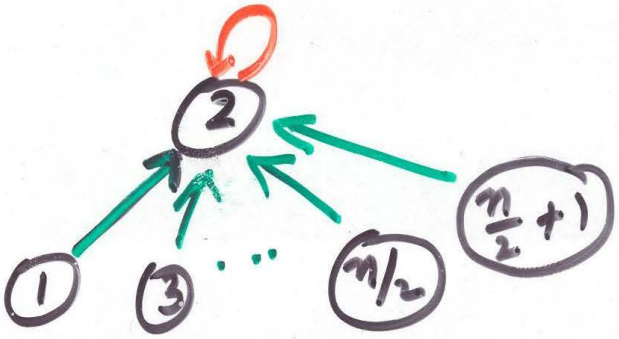
$n/2$. UNION ($n/2$, $n/2+1$)

$\frac{n}{2}+1$ FIND (1)
FIND (1)

⋮

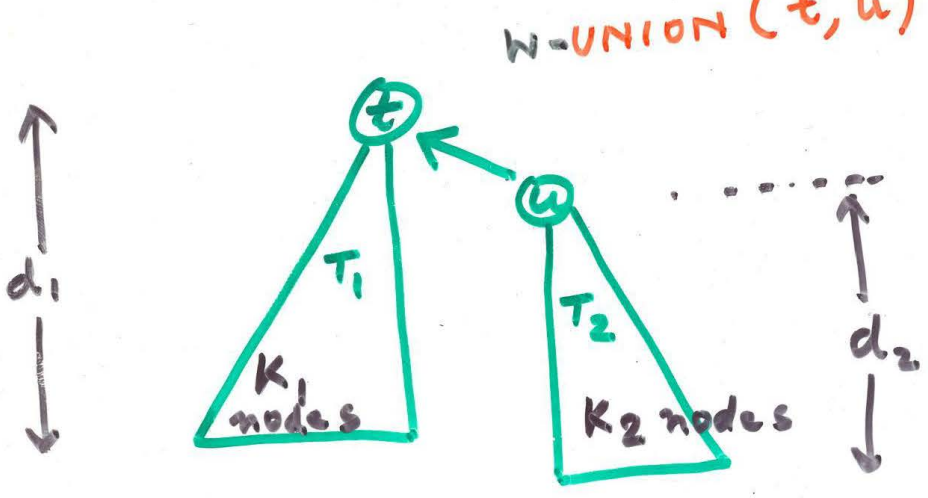
n FIND (1)

TREE WITH UNWEIGTED UNION



TREE WITH WEIGTED UNION

Lemma With W-UNION, ANY TREE THAT HAS K NODES HAS DEPTH AT MOST $\lfloor \lg k \rfloor$.



Let $k_1 \geq k_2$, $k = k_1 + k_2$

$d = d_1$ if $d_1 > d_2$

$= d_2 + 1$ if $d_1 \leq d_2$

BASIS: $K = 1$ $\lfloor \lg 1 \rfloor = 0$

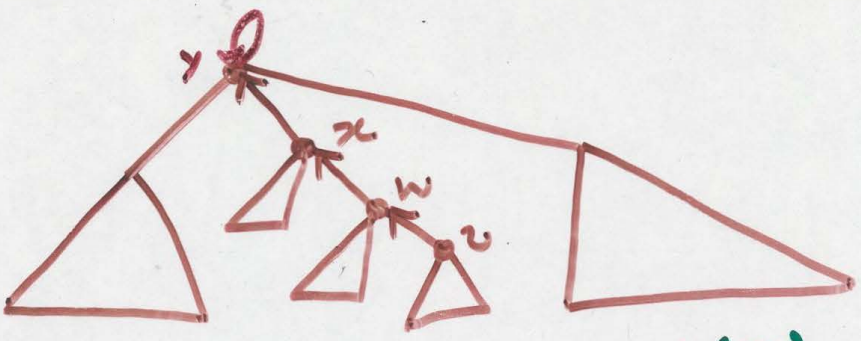
HYPOTHESIS: for $k' < k$, depth $\leq \lfloor \lg k' \rfloor$

INDUCTION

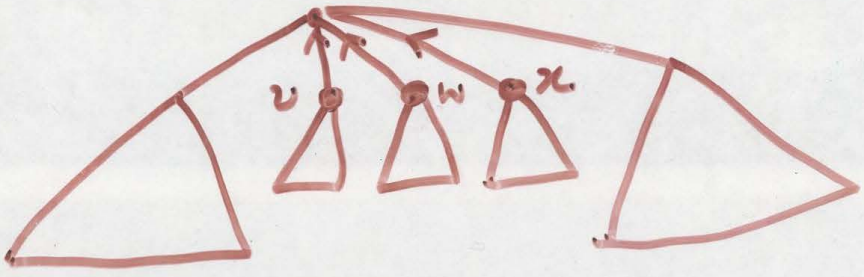
I. $d = d_1 \leq \lfloor \lg k_1 \rfloor \leq \lfloor \lg (k_1 + k_2) \rfloor = \lfloor \lg k \rfloor$

II. $d = d_2 + 1 \leq \lfloor \lg k_2 \rfloor + 1 = \lfloor \lg 2k_2 \rfloor \leq \lfloor \lg (k_1 + k_2) \rfloor$

Lemma A UNION-FIND PROGRAM OF SIZE n DOES $\Theta(n \lg n)$ link OPERATIONS IN THE WORST CASE IF THE WEIGTED UNION IS USED.



BEFORE C-FIND(v)
COMPRESSING-FIND(v)



AFTER C-FIND(v)

Lemma THE NUMBER OF LINK OPERATIONS DONE BY A UNION-FIND PROGRAM OF LENGTH n IMPLEMENTED WITH W-UNION AND C-FIND IS $O(n G(n))$.

Tarjan & Hopcroft

$G(n) = \log^* n$

$= \text{SMALLEST } i \text{ SUCH THAT}$

$\log^{(i)} n \leq 1$

where $\log^{(i)} n = \log(\log^{(i-1)} n)$

and $\log^{(0)} n = n$

$G(65536) = \log^*(2^{16}) = 4$

$\log 2^{16} = 16, \quad \log 16 = 4, \quad \log 4 = 2, \quad \log 2 = 1$

$G(2^{65536}) = 5. \Rightarrow G(n) \leq 5$ for all reasonable n.

[pp. 498-508 ch 22]

COMPARISON

m	$O(n)$	$O(\frac{n^2}{\log n})$	$O(n^2$
Kruskal's	$O(n \log n)$	$O(n^2)$	$O(n^2 \lg n)$
Prim's	$O(n^2)$	$O(n^2)$	$O(n^2$

Kruskal's
$O(m \log m)$
$= O(m \log n)$

PRIM's
$O(n^2)$

Time Complexity of Prim's

- with heap $O(n \log n + m \log n)$
- with Fibonacci Heap
  $O(n \log n + m)$

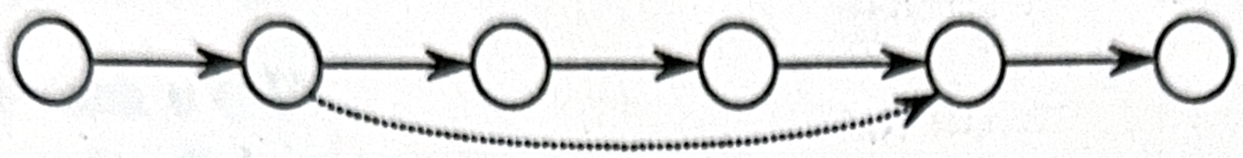
Shortest paths — from Chapter 25 introduction and Section 25.1

- *Generalization of breadth-first-search (from last lecture) to handle weighted graphs.*
- Directed graph $G = (V, E)$, weight function $w : E \to \mathbf{R}$
  (BFS — $w(e) = 1 \ \forall e$)
- e.g. street map (with distances on roads between intersections)
- Weight of path $p = v_0 \to v_1 \to \cdots \to v_k$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .$$

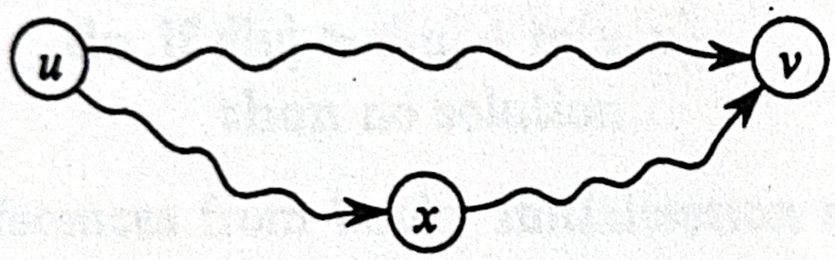
- "Shortest" path = path of minimum weight

- **Optimal substructure**
  (Thus, will see greedy and dynamic-programming algorithms.)
  Theorem: Subpaths of shortest paths are shortest paths.
  ⟨⟨Lemma 25.1, with both statement and proof in English rather than in mathematics.⟩⟩
  Proof: By "cut and paste"



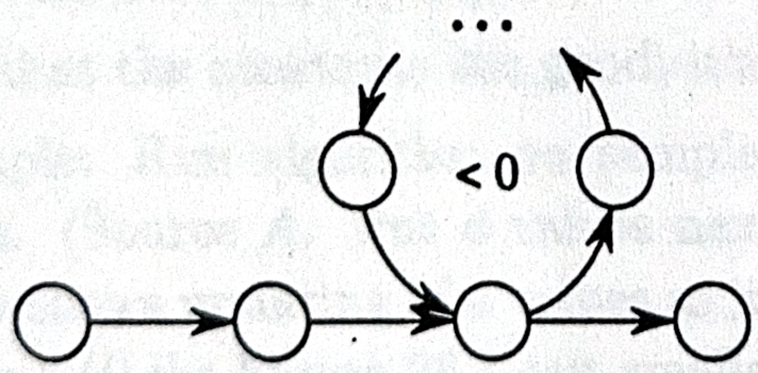
  If some subpath were not a shortest path, could substitute the shorter subpath and create a shorter total path.

- Def: $\delta(u, v) =$ weight of a shortest path from u to v
- **Triangle inequality**
  Theorem: $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$
  ⟨⟨Generalization of Lemma 25.3⟩⟩
  Proof:



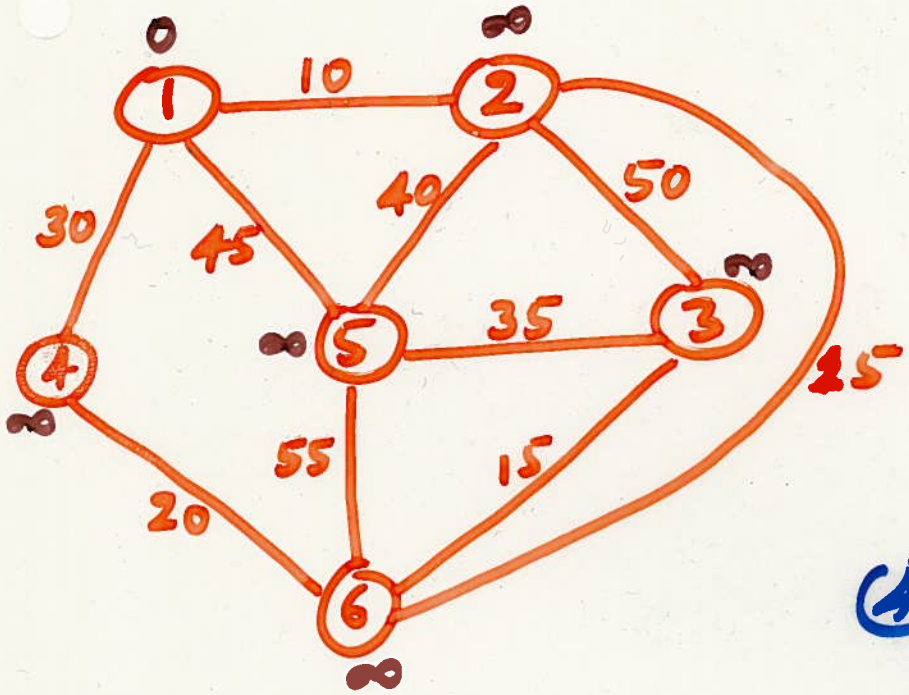
  Shortest path $u \rightsquigarrow v$ is no longer than any other path — in particular, the path that takes the shortest path $u \rightsquigarrow x$, then the shortest path $x \rightsquigarrow v$.

- **Well-definedness**
  Negative-weight cycle in graph ⇒ some shortest paths may not exist.
  Argument: Can always get a shorter path by going around the cycle again.

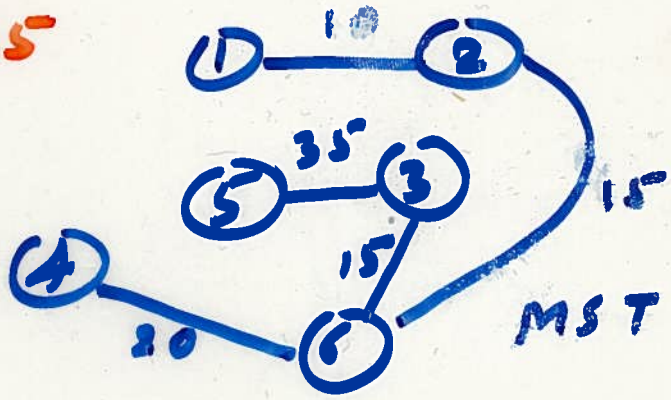


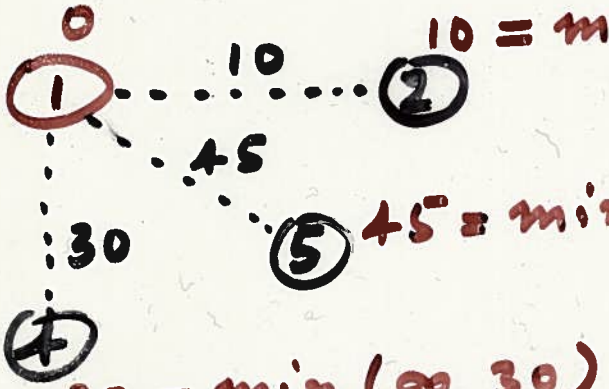
- (Above were high points of theory — rest of theory is in book.)

SHORTEST PATH



Q: Find a shortest path from 1 to 3



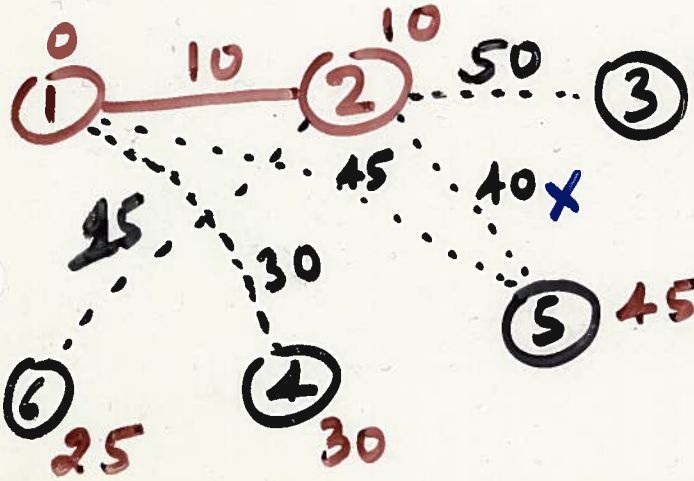


$10 = \min(\infty, 10)$

$45 = \min(\infty, 45)$

$30 = \min(\infty, 30)$

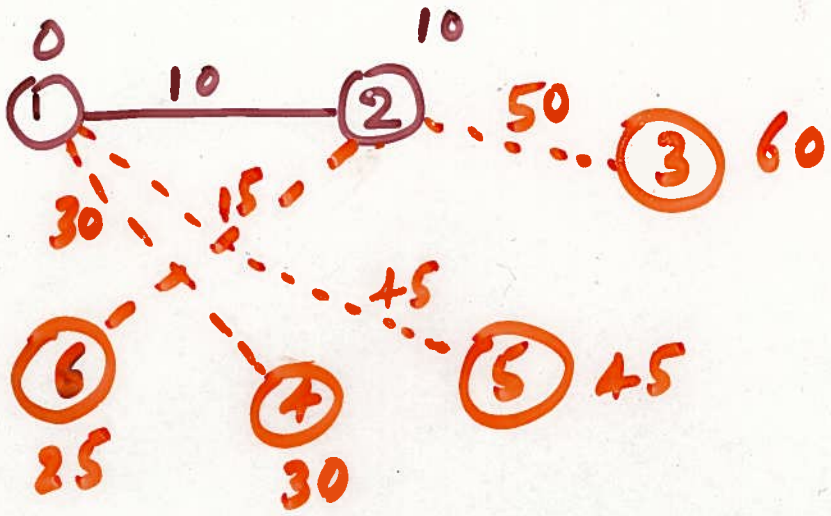
choose 2



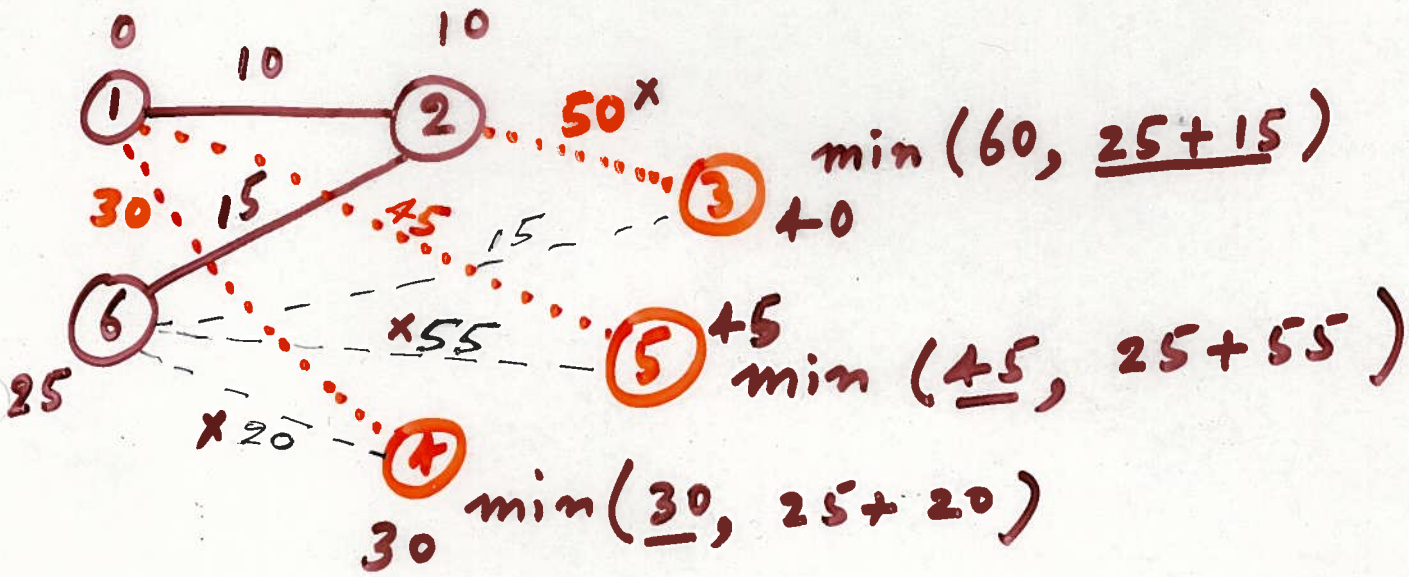
$60 = \min(\infty, \underline{10+50})$

$45 = \min(\underline{45}, 10+40)$

choose 6



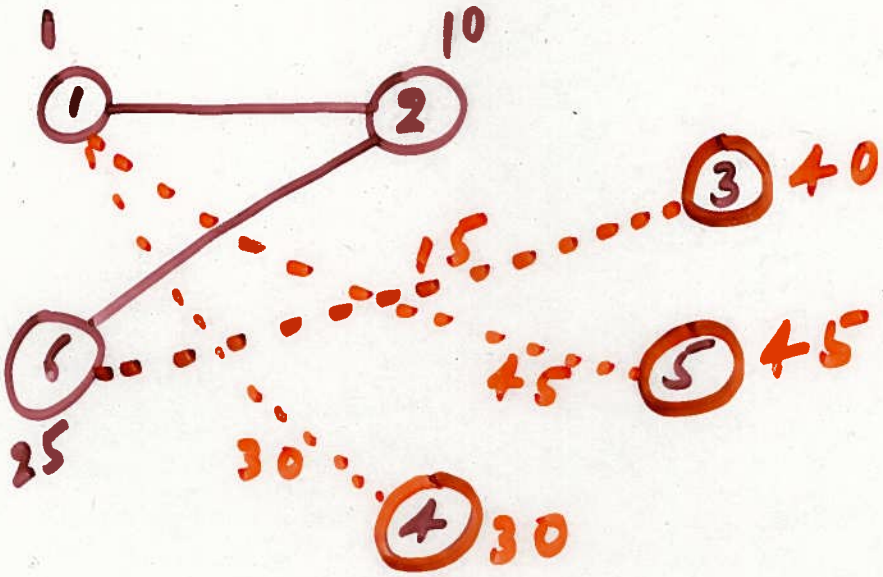
Choose 6



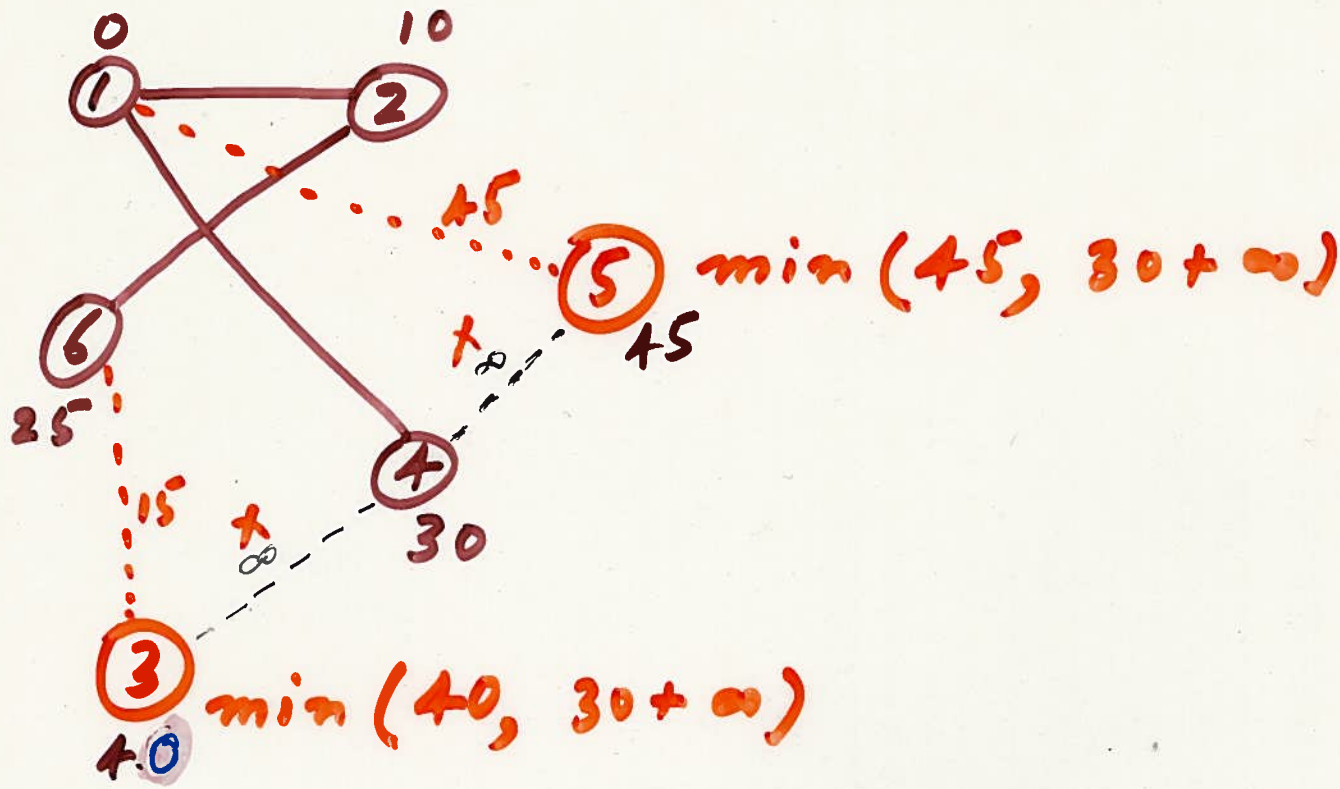
min (60, 25+15)

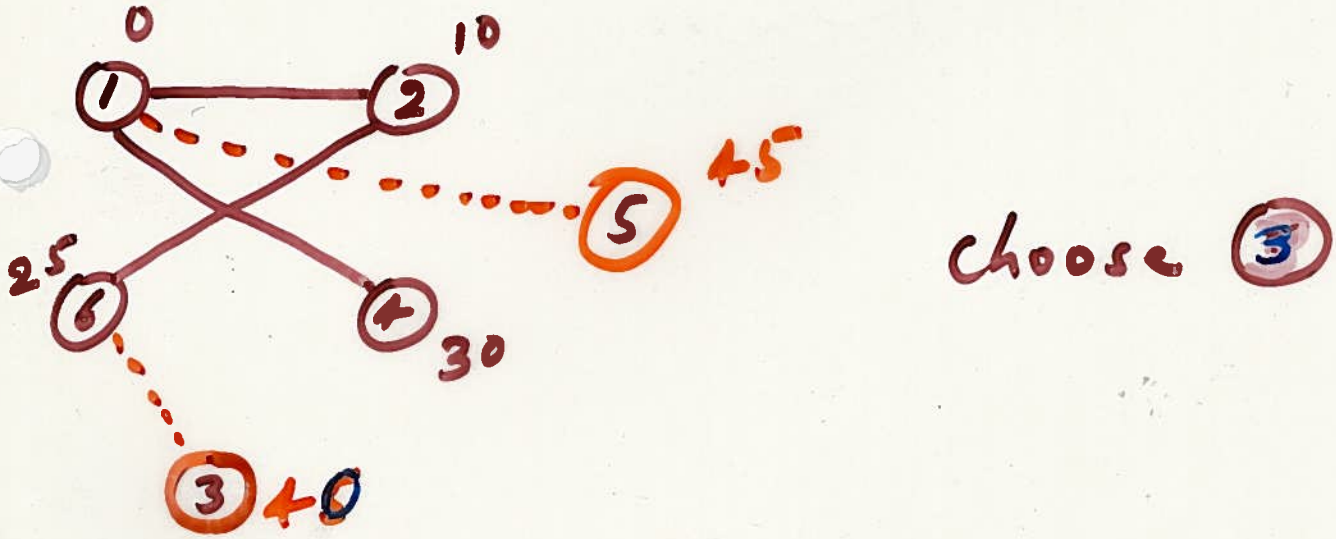
min (45, 25+55)

min(30, 25+20)

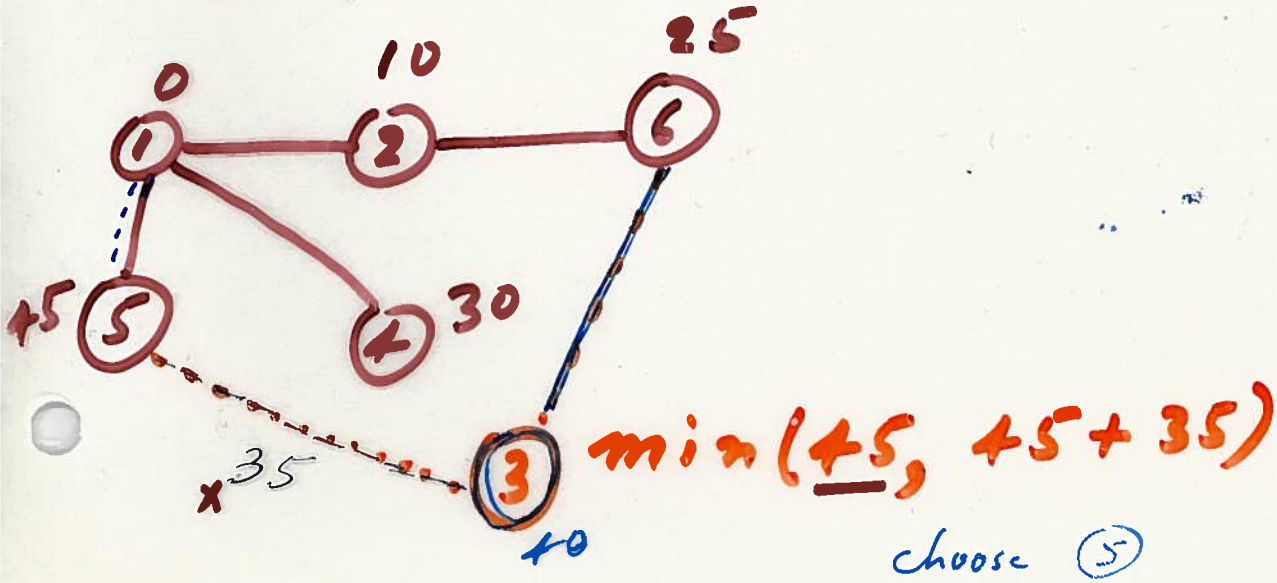


Choose 4

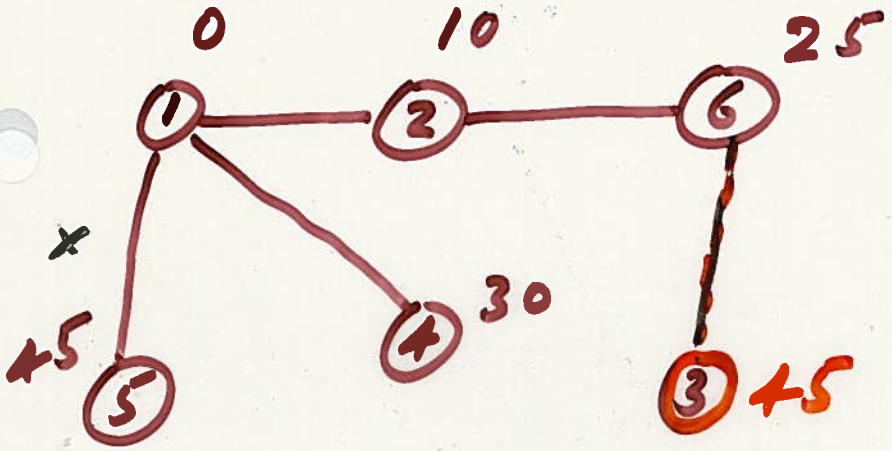




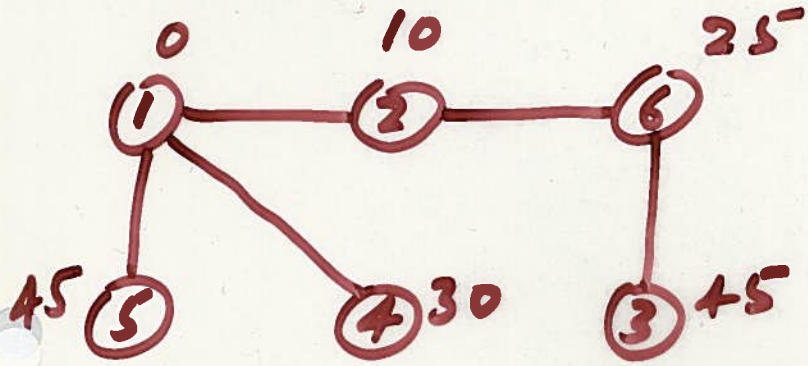
choose (3)



choose (5)



Choose 3



Shortest Path Tree with root node 1

Assumption: No negative weight.

```
DIJKSTRA(G)
    for each v ∈ V
        do d[v] ← ∞
    d[s] ← 0
    S ← ∅
    Q ← V
    while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
           S ← S ∪ {u}
           for each v ∈ Adj[u]
               do if d[v] > d[u] + w(u, v)
                     then d[v] ← d[u] + w(u, v)
```

Q : Priority Queue

* DECREASE KEY (v)

Q	n* Extract-Min + m* Decrease-key	
array	$n * O(n) + m * O(1)$	$= O(n^2)$
heap	$n * \log(n) + m * \log n$	$= O(m \log n)$
fibonacci	$n * \log(n) + m * O(1)$	$= O(n \log n + m)$

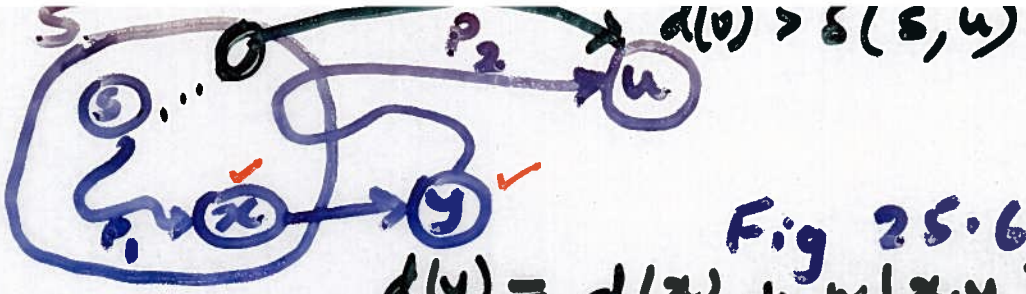


Fig 25.6

Correctness: Prove that whenever u is added to S, $d[u] = \delta(s,u)$
⟨⟨Theorem 25.10⟩⟩

Proof: ⟨⟨Basically same as in book, but derives a different contradiction.⟩⟩

- Figure 25.6
- Note that $\forall v\ d[v] \geq \delta(s,v)$
  (because lemma proved for Bellman-Ford above was just about relaxation, didn't depend on order of relaxing edges)
- Let u be first vertex picked such that $\exists$ shorter path than $d[u]$
  $\Rightarrow d[u] > \delta(s,u)$ — (i)
- Let y be first vertex $\in V - S$ on actual shortest path from s to u
  $\Rightarrow d[y] = \delta(s,y)$ — (ii)
  Because:
  - $d[x]$ is set correctly for y's predecessor $x \in S$ on the shortest path (by choice of u as first choice for which that's not true)
  - when put x into S, relaxed (x,y), giving $d[y]$ correct value,

if $y = u$, we are done.

$$\begin{aligned} d[u] &> \delta(s,u) && \text{— (i)} \\ &= \delta(s,y) + \delta(y,u) && \text{(optimal substructure)} \\ &= d[y] + \delta(y,u) && \text{(ii)} \\ &\geq d[y] && \text{(no negative weights)} \end{aligned}$$

- But $d[u] > d[y]$ $\Rightarrow$ algorithm would have chosen y to process next, not u.
  Contradiction.

Bellman. Ford Algorithm

1. **for** each $v \in V$
   do $d[v] \leftarrow \infty$
   $d[s] \leftarrow 0$

$O(nm)$

2. **for** $i \leftarrow 1$ **to** $|V| - 1$
   do for each edge $(u, v) \in E$
   do if $d[v] > d[u] + w(u, v)$
   then $d[v] \leftarrow d[u] + w(u, v)$

3. **for** each edge $(u, v) \in E$
   do if $d[v] > d[u] + w(u, v)$
   then no solution

Relaxation

Lemma: $d[v] \geq \delta(s, v)$ always.

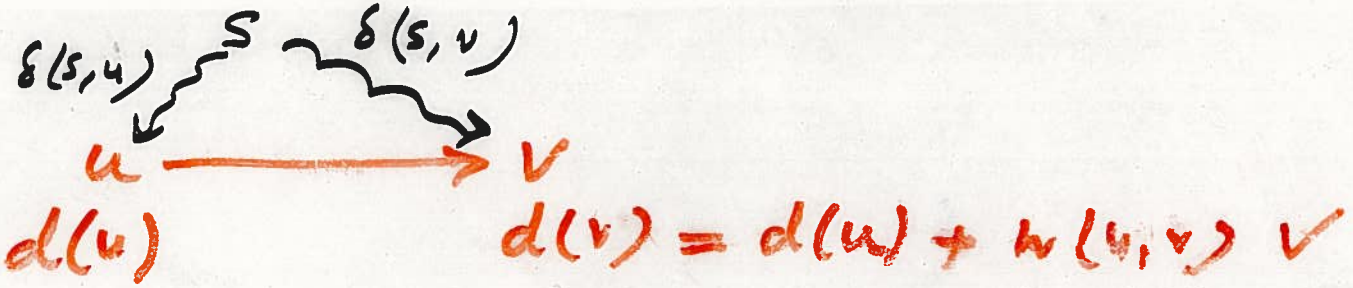
⟨⟨First part of Lemma 25.5, Section 25.1⟩⟩

- Initially true
- Let v be first vertex for which $d[v] < \delta(s, v)$, and let u be vertex that caused $d[v]$ to change:
- $d[v] = d[u] + w(u, v)$ — (i)
- Then

$$\begin{aligned} d[v] &< \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) && \text{(Triangle inequality)} \\ &\leq d[u] + w(u, v) && (v \text{ is first violation}) \end{aligned}$$

$\Rightarrow d[v] < d(u) + w(u,v)$ — (ii)

contradicts $d[v] = d[u] + w(u, v)$ (above).



- Bellman-Ford is correct (after $|V|-1$ passes, all the d values are correct) ⟨⟨Lemma 25.12⟩⟩

  Proof: ⟨⟨Informal version of book's proof.⟩⟩

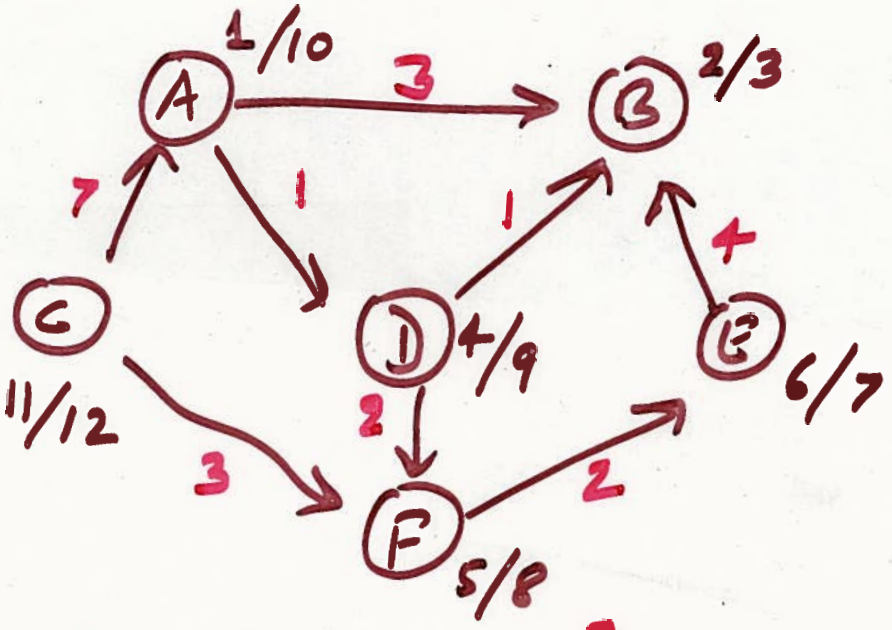
  Let v be a vertex, and consider shortest path from s to v (assuming no neg-weight cycles):

  $s \to v_1 \to v_2 \to v_3 \to v_4 \to v$

  - Initially, $d[s] = 0$ is correct (and doesn't change thereafter — by previous lemma and fact that code never increases d)
  - After 1 pass through edges, $d[v_1]$ is correct (and doesn't change...) ($d[s]$ is correct, and by optimal substructure, shortest distance is $w(s, v_1)$. 1st pass sets $d[v_1] = d[s] + w(s, v_1)$, which is right answer.)
  - After 2 passes through edges, $d[v_2]$ is correct (and doesn't change...)

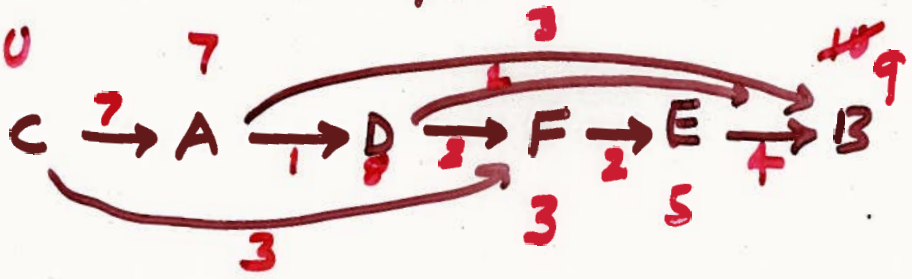
    ⋮

TOPOLOGICAL SORTING



DAG & DFS

(u)
visiting time
finishing time



SHORTEST PATH IN DAGs

1. TOPOLOGICAL SORT USING DFS
2. ONE ITERATION OF BELLMAN-FORD ALGORITHM

$O(n+m)$

1 2 3 4
1 3 2 4