Analysis and Design of Algorithms

Session 8.

Maestría en Sistemas Computacionales.

Luis Fernando Gutiérrez Preciado.

What will we see today?

- Greedy Algorithms
  - Knapsack Problem
- Graphs
  - Introduction to Graphs
  - Types of Graphs
  - Graph Representation
  - Depth-First Search
  - Breadth-First Search

Greedy Algorithms

- Used to solve optimization problems:

1. Execute all pending tasks following a sequence that minimizes the total time.
2. Find the fastest/cheapest path from A to B.
3. Insert valuable objects into a backpack so that the value of the inserted objects is maximized without exceeding the backpack's capacity.
4. Find a route that visits all cities and returns to the starting point in the shortest possible time.
5. Connect all points in a network using the least amount of cable.

Greedy Algorithms

- They are characterized by making the best local decision without a global view of the problem.
  - Therefore, they do not guarantee global optimization.
- They are easier to program than their competitors.
- When they work (when they deliver an optimal solution), they outperform their competitors because they are more efficient.
  - This means that there are problems that cannot be optimally solved by Greedy Algorithms.
- We will focus on the cases where greedy algorithms work effectively.

Why don't they always work?

- Let's consider a simple example: the change-making problem.
- “Given a set M of coins with different denominations and a monetary amount C to be given as change, provide the exact amount with the minimum possible number of coins”.
- According to the denominations in our monetary system (1, 2, 5, 10, 20), the greedy algorithm always provides an optimal solution:
  - At each step, it chooses the coin with the highest possible denomination such that the sum of the chosen coins does not exceed the change.

Why don't they always work?

- Unfortunately, real-world computable problems are not that simple.
- To add to the challenge, here are the available coins: M = {1, 1, 1, 1, 4, 4, 4, 6, 6, 10, 10}, and the change C = 9.
- A greedy algorithm would give me the solution: {6, 1, 1, 1}.
  - It chooses the largest available coin (local information) that will always be in the solution and does not consider other combinations.
  - Every discarded coin will never be reconsidered.
- However, the best solution is: {4, 4, 1}.
- $|\{4, 4, 1\}| = 3 < 4 = |\{6, 1, 1, 1\}|$

General Algorithm

```
Set/List GreedyAlgorithm(Set/List C) {
    let S ← ∅  {The solution will be stored in S}
    While C ≠ ∅ y  Solution(S), do:
        Let x ← Select(C)
        C ← C \ {x}
        If Feasible(S ∪ {x}) then S ← S ∪ {x}
    End-While
    If Solution(S) return S
    Else return ∅ {No solutions}
}
```

What problems will we solve?

- Of the exemplified optimization problems, there are three that can be solved by a greedy algorithm:
  1. The knapsack problem if we can split the objects.
  2. Connecting all points in a network using the least amount of cable.
  3. Finding the fastest/cheapest path from A to B.
- Therefore, these are the ones we will solve (in that order).

Knapsack Problem

- There are N available objects and a knapsack.
- Each object k weighs w_k and has a value v_k, where w_k, $v_k > 0$.
- The knapsack can carry a weight of no more than W.
- In this version, we can put a fraction f_k of object k in the knapsack, where: $0 \leq f_k \leq 1$.
- The problem is formally defined as follows:
- Maximize $\sum_{k=1}^{N} f_k v_k$ where: $\sum_{k=1}^{N} f_k w_k \leq W$

Knapsack problem

```
List Knapsack(List-of-items C, Max-weight W) {
     List-of-items S ← ∅
     weight ← 0
     While weight < W, do:
          Let k ← Select(C) {Return the index}
          Let x ← C[k] {Get the element}
          C ← C \ {x}
          If weight + x.weight ≤ W       {It still fits, take the whole item}
                    x.fraction = 1.0
                    weight ← weight + x.weight
          Else {We've exceeded W, take the maximum fraction of the item}
                    x.fraction = (W – weight) / x.weight
                    weight = W
          S ← S ∪ {x}
     End-While
     Return S
}
```

Knapsack problem

- Which element do we select?
  1. The most valuable one?
  2. The lightest one?
  3. The one with the highest value-to-weight ratio?

$W = 100$	1	2	3	4	5
v	20	30	66	40	60
w	10	20	30	40	50
v / w	2.0	1.5	2.2	1.0	1.2

KnapSack problem

$W = 100$	1	2	3	4	5
v	20	30	66	40	60
w	10	20	30	40	50
v / w	2.0	1.5	2.2	1.0	1.2

Select	1	2	3	4	5	Value
Max v_k	0	0	1	0.5	1	146
Min w_k	1	1	1	1	0	156
Max v_k / w_k	1	1	1	0	0.8	164

Greedy Algorithms Applied to Problems Represented as Graphs

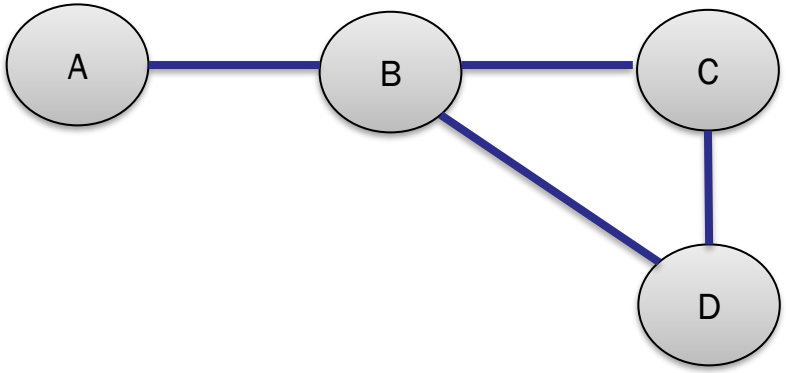
- Graphs

Graphs

- Many computational problems can be formulated in terms of objects and connections between them. For example:
  - Given a map of **airline routes** in a country, what is the fastest way to travel from one city to another? What is the cheapest way?
  - Given an **electrical circuit** composed of transistors, resistors, and capacitors, are all elements connected, and will it function correctly?
  - Given a set of **tasks to be performed**, where some tasks are activated upon the completion of others, when does each task start?
  - In a **social network**, are there groups of users who are not connected to any other contact in the network? Is there a user who can be known by another user who is not their contact?

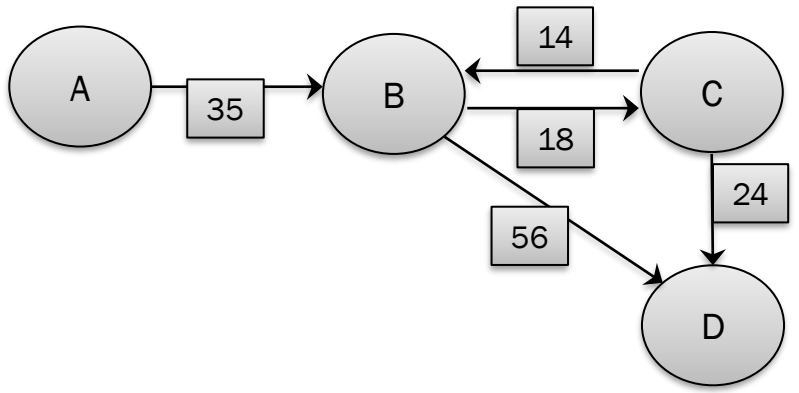
Graphs

- From a programming perspective, a graph is a data structure composed of nodes interconnected by *edges*.
- Each node is identified by a value.
- The following graph has four nodes {A, B, C, D} and four edges {(A, B), (B, C), (B, D), (C, D)}.
  - Note that (A, B) = (B, A).



Types of Graphs:

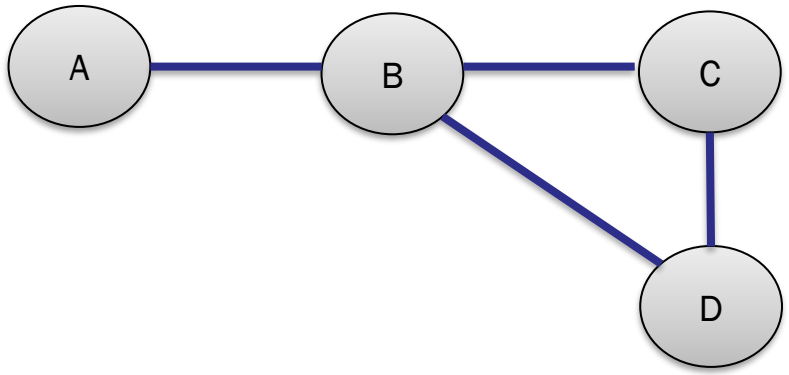
- **Weighted graphs**, where values or weights are assigned to edges (distance, cost, time, etc.).
- **Directed graphs**, where edges have a direction, i.e., (A, B) does not imply (B, A).
- **Weighted directed graphs**, which combine the two characteristics mentioned above.



Representation

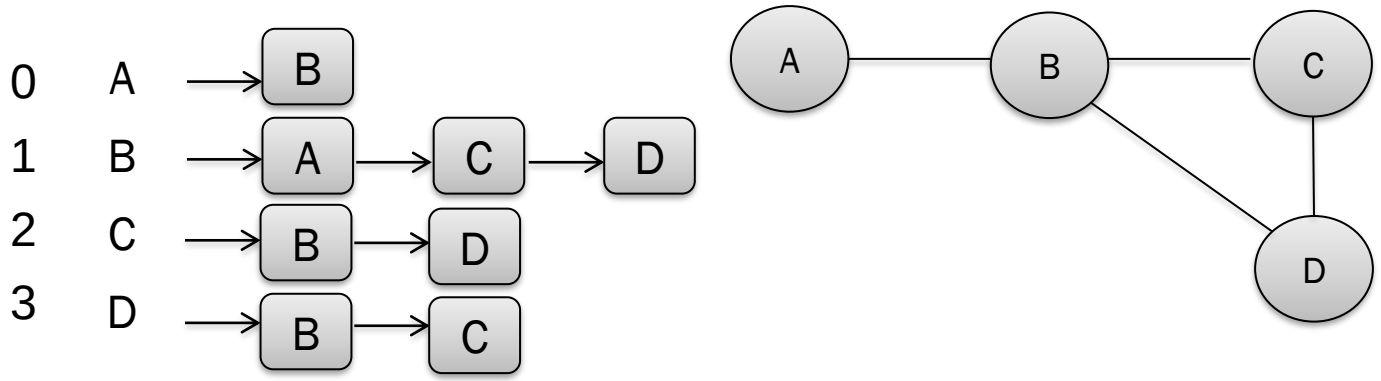
- Adjacency matrices.
  - Easy implementation.
  - Efficient for access.
  - Suitable for complete or dense graphs.
    - Complete graph: the number of edges is the maximum possible: ½ N (N - 1)
    - The number of edges is greater than N log N. (N = number of vertices).
  - For Sparse Graphs, there would be much wastage.
  - Does not allow for growth in the number of vertices.

0	1	0	0
1	0	1	1
0	1	0	1
0	1	1	0



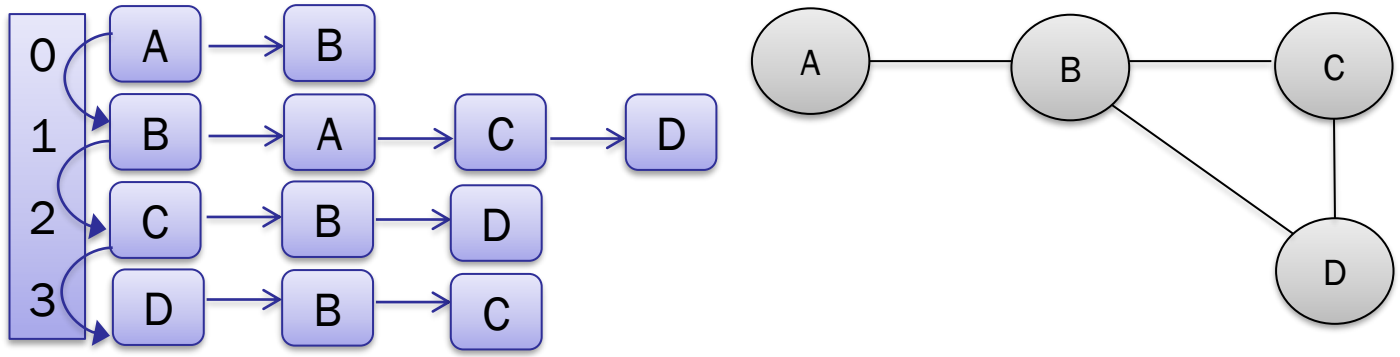
Representation

- Adjacency Lists.
  - Less straightforward implementation.
  - Less memory wastage for Sparse Graphs.
  - Allows for growth in the number of edges.
  - Depth-first traversal has linear complexity.
  - Does not allow for growth in the number of vertices.



Representation

- Dynamic adjacency lists.
  - More complex implementation.
  - Less memory wastage for sparse graphs.
  - Depth-first traversal has linear complexity.
  - Allows for growth in the number of vertices and edges.



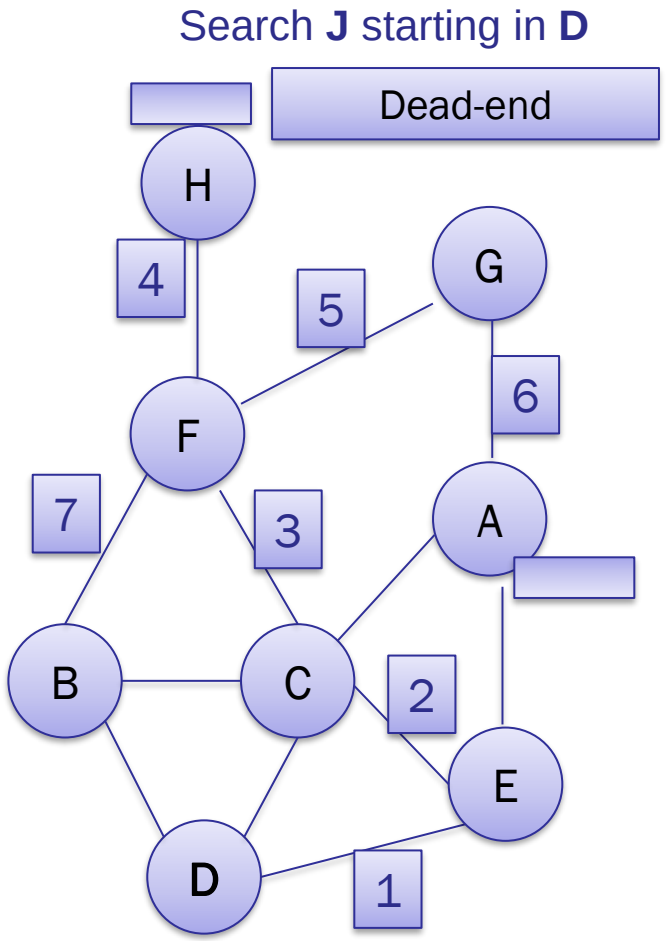
Graph Searches:

- There is an initial node and a key to be found.
- The search can return the data with that key or a simple true or false indicating whether it exists.

- Depth-First Search.
  - Can be implemented using recursion or with stacks (iterative).
- Breadth-first search.
  - Can be implemented using queues (iterative).

Depth-First Search:

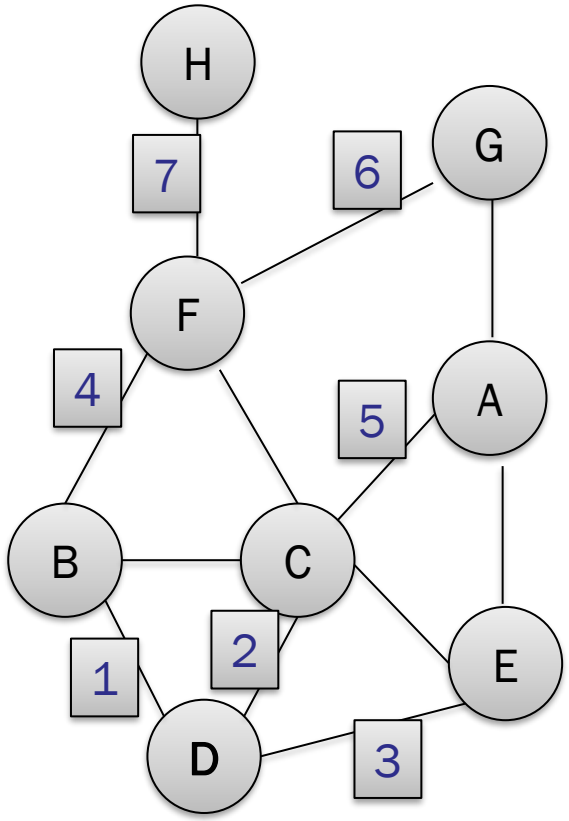
1. Starting from the root node, depth-first search moves to the first unvisited neighbor of the current node as long as it doesn't find the key and doesn't reach a dead-end: no more unvisited neighbors.
2. When it reaches a dead-end, it backtracks: it returns to the second-to-last visited node and processes the next unvisited neighbor.
3. Using a **stack of nodes** to visit, the last added node will be the first one processed



Breadth-First Search:

1. Starting from the root node, breadth-first search processes all neighbors at a distance of 1, then the neighbors of each neighbor (distance = 2), and so on.
2. Using a **queue of nodes** to visit, the first added nodes are the first ones processed.
3. Both types of searches require a list of visited nodes in addition to the queue/stack of nodes to visit.

Search **J** starting in **D**



Exercise

- Implement a method to determine if a key is reachable from another key in an unweighted, undirected, and not necessarily connected graph.
  - Using depth-first and breadth-first search.
  - Using both integer and alphanumeric keys.
- Represent the graph using an adjacency matrix.
- What are the data structures for...
  - Input (graph content)?
  - Processing (visited nodes, nodes to visit)?