Course Name: Analysis and Design of Algorithms

Topics to be covered

- Greedy Method
  - General Method
  - Minimum Spanning Tree
  - Kruskal's Algorithm

Overview of Greedy Method

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Greedy Strategy

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it’s always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

Activity-selection Problem

- Input: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.

Example:

Activities in each line are compatible.

Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- **Subproblems**: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max_{i<k<j}\{c[i,k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Recursive Algorithm

Recursive-Activity-Selector (s, f, i, j)

1. $m \leftarrow i+1$
2. **while** $m < j$ and $s_m < f_i$
3. **do** $m \leftarrow m+1$
4. **if** $m < j$
5. **then return** $\{a_m\} \cup$ Recursive-Activity-Selector(s, f, m, j)
6. else return ϕ

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: $\Theta(n)$

Straightforward to convert the algorithm to an iterative one. See the text.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

Elements of Greedy Algorithms

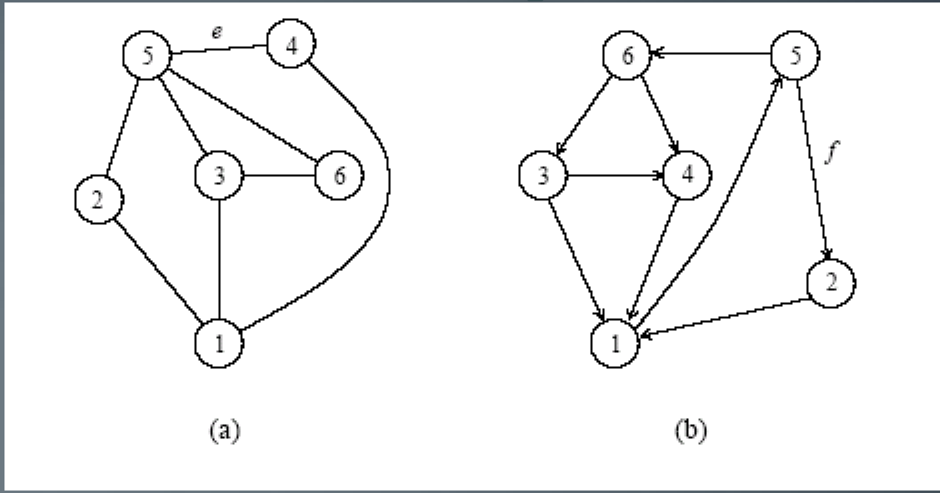
- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Minimum Spanning Trees

Definitions and Representation

- An *undirected graph* G is a pair (V,E), where V is a finite set of points called *vertices* and E is a finite set of *edges*.
- An edge $e \in E$ is an unordered pair (u,v), where $u,v \in V$.
- In a directed graph, the edge e is an ordered pair (u,v). An edge (u,v) is *incident from* vertex u and is *incident to* vertex v.
- A *path* from a vertex v to a vertex u is a sequence $<v_0,v_1,v_2,\ldots,v_k>$ of vertices where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for $I = 0, 1, \ldots, k-1$.
- The length of a path is defined as the number of edges in the path.

Definitions and Representation



a) An undirected graph and (b) a directed graph.

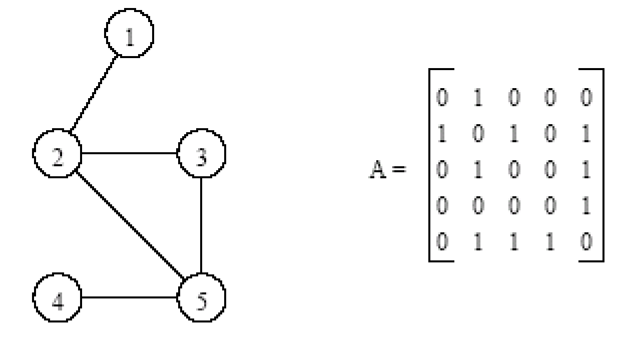
Definitions and Representation

- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A *forest* is an acyclic graph, and a *tree* is a connected acyclic graph.
- A graph that has weights associated with each edge is called a *weighted graph*.

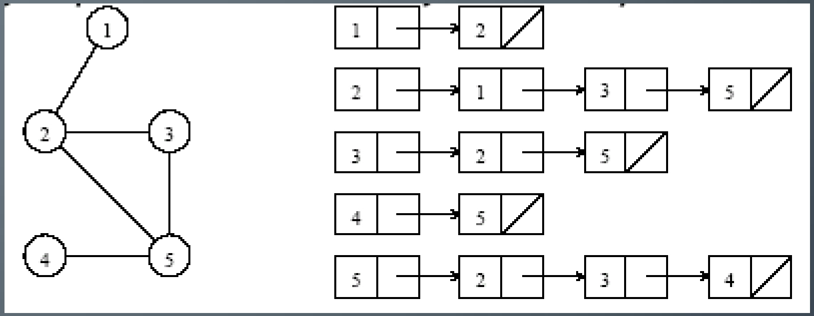
Definitions and Representation

- Graphs can be represented by their adjacency matrix or an edge (or vertex) list.
- Adjacency matrices have a value $a_{i,j} = 1$ if nodes *i* and *j* share an edge; 0 otherwise. In case of a weighted graph, $a_{i,j} = w_{i,j}$, the weight of the edge.
- The *adjacency list* representation of a graph $G = (V,E)$ consists of an array $Adj[1..|V|]$ of lists. Each list $Adj[v]$ is a list of all vertices adjacent to *v*.
- For a grapn with *n* nodes, adjacency matrices take $\Theta(n^2)$ space and adjacency list takes $\Theta(|E|)$ space.

Definitions and Representation



An undirected graph and its adjacency matrix representation.

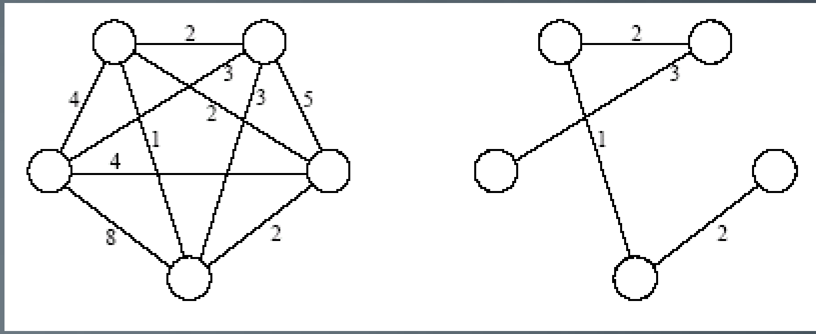


An undirected graph and its adjacency list representation.

Minimum Spanning Tree

- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.
- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.

Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

Kruskal's Algorithm

- Starts with each vertex in its own component.
- Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
- Scans the set of edges in monotonically increasing order by weight.
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.