COMP 3761: Algorithm Analysis and Design

Shidong Shan

BCIT

Overview

- The greedy technique
- Constructing minimum spanning trees
  a. Prim's Algorithm
  b. Kruskal's Algorithm
- Single-source shortest-paths problem
  Dijkstra's algorithm
- Application of greedy technique
  e.g. Scheduling problem

Greedy algorithm

- Typically designed for *optimization problems*
  - several possible legal solutions
  - values associated with solutions
  - want to find a legal solution with the maximum (or minimum) value
- When faced with several choices, make one that is *at this point* the **best** (even though it might eventually lead to a non-optimal solution)

Greedy technique

- Construct a solution to an optimization problem piece by piece through a sequence of choices that are:
  - **feasible**: satisfy the problem's constraints
  - **locally optimal**: best local choice among all feasible choices available on that step
  - **irrevocable**: once made, it cannot be changed on subsequent steps of the algorithm.
- For some problems, yields an optimal solution for every instance
- For most, solution may not be optimal...
- However, it can be useful for fast approximations.

Application of greedy technique

- Optimal solutions:
  - change making for "normal" coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-making problem

- Given unlimited quantities of coins of denominations $d_1 > d_2 > \ldots > d_m$ and a total amount n
- Find the smallest number of coins that add up to amount n.
- For example:
  $d_1 = 25c, \quad d_2 = 10c, \quad d_3 = 5c, \quad d_4 = 1c$, and $n = 48c$
- Greedy solution: 1 quarter, 2 dimes, and 3 pennies.

- The greedy solution to change-making problem is
  - optimal for any amount and “normal” set of denominations
  - may not be optimal for arbitrary coin denominations

Minimum spanning tree (MST)

- **Tree**: a connected graph without cycles (ie. acyclic)
- Tree property: $|E| = |V| - 1$

- **Spanning tree** of a connected graph G:
  a connected acyclic subgraph of G that includes all of G's vertices

- **Minimum spanning tree** of a weighted, connected graph G:
  a spanning tree of G of minimum total weight

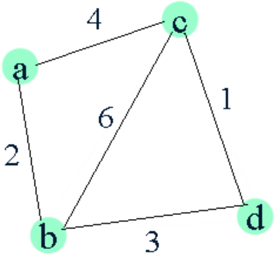
- **Minimum spanning tree problem**:
  the problem of finding a minimum spanning tree for a given weighted connected graph.

Prim's algorithm

- Start with tree T_1 consisting of one (any) vertex
- Grow tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (the "greedy" step)
- Stop when all vertices are included

Application of Prim's algorithm

Given a graph G, find the MST using Prim's Algorithm.



Pseudocode of Prim's algorithm

ALGORITHM *Prim*(G)

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \emptyset$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Notes on Prim's MST algorithm

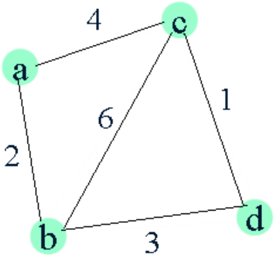
- Proof by induction that this construction actually yields MST
- **fringe vertex**: vertex not in the current tree but adjacent to at least one tree vertex
- Needs priority queue for locating closest fringe vertex
- Efficiency:
  - Given a graph with n vertices and m edges
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$ for adjacency list representation of graph and min-heap implementation of priority queue

Kruskal's MST algorithm

- Sort the edges in nondecreasing order of lengths
- Start with an empty subgraph, scan the sorted list and add the next edge on the list to the current subgraph
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle; if the edge would create a cycle, simply skip it and continue with the next one.

Application of Kruskal's algorithm

Given a graph G, find the MST using Kruskal's Algorithm.



Pseudocode of Kruskal's algorithm

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \ldots \le w(e_{i_{|E|}})$
$E_T \leftarrow \emptyset$; $\quad ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
    $k \leftarrow k + 1$
    if $E_T \cup \{e_{i_k}\}$ is acyclic
        $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

Notes on Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement
- Need to check for cycles at each iteration
- Cycle checking: a cycle is created **if and only if** added edge connects vertices in the same connected component

Single source shortest paths

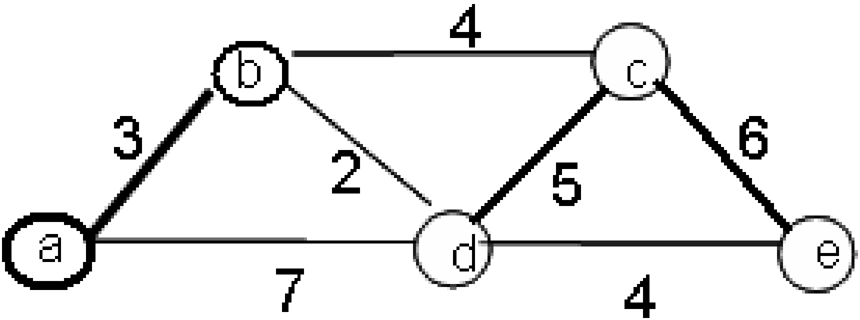
- **Single Source Shortest Paths Problem**:
  Given a weighted connected graph G, find shortest paths from source vertex s to **each** of the other vertices
- **Dijkstra's algorithm**: Similar to Prim's MST algorithm, with a different way of computing numerical labels
- Among vertices not already in the tree, find vertex u with the smallest sum $d_v + w(v, u)$, where
  - v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)
  - d_v is the length of the shortest path from the source to v
  - $w(v, u)$ is the length (weight) of edge from v to u

Operations of Dijkstra's algorithm

1. Start with the source vertex s, label s as $(-, 0)$.
2. Label each vertex u with two labels (v', d), where
   - d: the length of the shortest path from the s to u so far; when u is added to the tree, d is the length of the shortest path from s to u.
   - v': the next-to-last vertex on the shortest path.
3. Identify the next nearest vertex v^* by finding a fringe vertex with the smallest d.

Application of Dijkstra's algorithm

Given a graph G, find the shortest paths from a starting vertex to all other vertices by Dijkstra's algorithm.



Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weights
- Applicable to both undirected and directed graphs
- Efficiency (same as Prim's algorithm)
  - Given a graph with n vertices and m edges
  - $O(n^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(m \log n)$ for graphs represented by adjacency lists and min-heap implementation of priority queue

Scheduling problem

- Given n jobs to execute on a single machine, each job with a start time s_i and a finish time f_i
- Goal: find the maximum number of non-overlapping jobs can be scheduled on the single machine.
- Two jobs i and j are **non-overlapping** if $s_i \geq f_j$ or $s_j \geq f_i$

Greedy Algorithm

//Input: $S = \{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$
//Output: a maximum (size) set of non-overlapping jobs.

Algorithm *GreedySched*(S)
  Sort jobs by finish time, $f_1 \leq f_2 \leq \ldots \leq f_n$
  $SetOfJobs = \phi$
  for $i = 1..n$
    if job i does not overlap any job in $SetOfJobs$
      $SetOfJobs \leftarrow SetOfJobs \cup \{i\}$
  return $SetOfJobs$

Exercises

Section 9.1: 7a, 8

Section 9.2: 1a, 2

Section 9.3: 2a, 3, 4