CS 4310 Algorithms: IV. Greedy Methods

E. de Doncker[1]

[1] Dept. of Computer Science, WMU, U.S.A.

February 2024

Outline

Outline

Outline

Outline

Outline

Outline

Greedy meta-algorithm

A typical greedy algorithm optimizes (minimizes or maximizes) an objective function (representing a cost or a gain) under a set of constraints. A solution that satisfies the constraints is called a feasible solution. An optimal solution is a feasible solution that optimizes the objective function.
A general “meta-” or umbrella greedy algorithm is given below [1].

Greedy meta-algorithm

Meta-algorithm *Greedy* (A, n) { // returns *solution* for inputs A and size n
    solution $= \emptyset$; // Initialize *solution*
    for ($i = 1; i \leq n; i{+}{+}$) { // Construct *solution*
        x = *Choose* (A);
        if (*Feasible* (*solution*, x)) // Check if x leads to feasible solution
            solution = *Union* (*solution*, x); // Add x to solution
    }
    return *solution*;
}

Greedy knapsack problem

Given the knapsack capacity M, number of items n, with p_i and w_i the (positive) profit and weight, respectively, per unit of item i, the problem is to determine the fractions x_i of the items to fill the knapsack so that the total profit is maximized. That is [1],

Greedy knapsack problem

maximize $\sum_{i=1}^{n} p_i x_i$ *objective function*, subject to the *constraints* $\sum_{i=1}^{n} w_i x_i \leq M$

and $0 \leq x_i \leq 1$, for $1 \leq i \leq n$

Example : $M = 20$, $n = 3$, profits $\mathbf{p} = (50, 30, 20)$, weigths $\mathbf{w} = (10, 15, 5)$.

Strategy: add items to the knapsack in non-increasing order of profit per (unit of) weight. The ratios for the example are: $(50/10,\ 30/15,\ 20/5) = (5, 2, 4)$. The first and third object are put in completely, leaving room for one third of the second object. The total profit is $50 + 20 + 30/3 = 80$.

Algorithm *GreedyKnapsack*( ) [1]

Algorithm *GreedyKnapsack* (M, p, w, n, x) { // returns optimal fractions x
// for inputs M (knapsack capacity), p (profits), w (weights), and size n,
// assuming $p[1]/w[1] \geq p[2]/w[2] \geq \ldots \geq p[n]/w[n]$
// for positive profits and weights.
  for $(i = 1; i \leq n; i{+}{+})\ x[i] = 0;$ // Initialize
  $U = M;$ // U is remaining capacity
  for $(i = 1; i \leq n; i{+}{+})$ {
    if $(w[i] > U)$ break;
    $x[i] = 1;$
    $U -= w[i];$
  }
  if $(i \leq n)\ x[i] = U/w[i];$
}

Not including sorting of the ratios, the time complexity is linear in n.
Including sorting of the ratios, the time complexity is thus $\mathcal{O}(n \log n)$.

It can be proved that the strategy of adding items to the knapsack in non-increasing order of the profit by unit of weight ratios, yields an optimal solution [1].

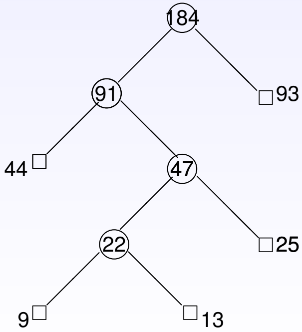
Optimal merge patterns

The objective is to find an optimal merge pattern for multiple sorted files that need to be merged, using a minimum number of record moves. A two-way binary merge tree is constructed.
As an example, consider files x_1, x_2, x_3, x_4, x_5 of sizes $9, 44, 25, 93, 13$. If merges are done by: (1) merging x_1, x_2 using up to $9 + 44 = 53$ moves; (2) merging the previous result with x_3 in $53 + 25 = 78$ moves; (3) merging the previous result with x_4 in $78 + 93 = 171$ moves; (4) merging the previous result with x_5 in $171 + 13 = 184$ moves; then the total number of moves is $53 + 78 + 171 + 184 = 486$.

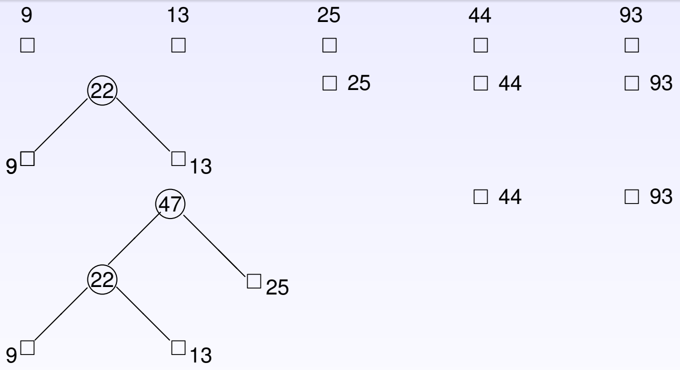
The optimal strategy is to combine the least two at each step: (1) merge x_1 and x_5 in $9 + 13 = 22$ moves, (2) merge the result of length 22 with x_3 in $22 + 25 = 47$ moves, (3) merge the result of length 47 with x_2 in $44 + 47 = 91$ moves, (4) merge the result of length 91 with x_4 in $91 + 93 = 184$ moves; then the total number of moves is $22 + 47 + 91 + 184 = 344$.

Optimal merge patterns - Example

The given date are represented by the leaf (square) nodes, called external nodes. The nodes combining children nodes are internal nodes, labeled by the sum of the keys of the children nodes. The key value (weight) for the i-th external node is‘q_i and its distance from the root is d_i. The weighted external path length is defined as $\sum_{i=1}^{n} q_i d_i$. We have $\sum_{i=1}^{n} q_i d_i = (9+13) \times 4 + 25 \times 3 + 44 \times 2 + 93 \times 1 = 344$.



Optimal merge patterns - Example



. . .

Optimal merge patterns - Algorithm

An outline of the algorithm can be given as follows (see also [2, 1]):
• Initialize the tree by forming a list of n one-node binary trees with weights $q_i,\ i = 1, \ldots, n$.
• Repeat until only one tree remains:
– Find two subtrees of minimum weight
– Merge into one subtree (create a new parent node and attach the selected subtrees as left and right child (in any order))
– Set the new node weight to the sum of the weights of the children
– Insert new subtree into the list

Pseudocode of the algorithm is given in [1]. It is shown that the algorithm generates the optimal two-way merge tree. The strategy minimizes the weighted external path length $\sum_{i=1}^{n} q_i d_i$, where d_i is the distance of the external node with weight q_i.

Indeed, this corresponds to placing external nodes with smaller weigths farther from the root (at larger depths d_i). Using a minheap as a priority queue data structure for selecting the next tree with least weight efficiently, the time as a function of n is $\mathcal{O}(n \log n)$. Using a linked list requires total time $\mathcal{O}(n^2)$.

Huffman trees and codes

The general objective is file compression, in order to minimize the file size and thus the transmission time for sending it. For compressing a text, or sending a message in the English language, all alphabet symbols and punctuation symbols in English need to be encoded. It is thus beneficial to give shorter encodings to symbols that occur more frequently.
A Huffman tree is constructed using an algorithm as outlined for optimal merge patterns, where the weights q_i represent frequencies. The basic principle is that more frequent symbols occur at shorter distances from the root, and less frequent symbols are deeper in the tree. The algorithm minimizes the weighted external path length $\sum_{i=1}^{n} q_i d_i$ and produces an optimal encoding tree.
Using a heap priority queue the algorithm time is $\mathcal{O}(n \log n)$.
To construct the Huffman code as a binary string for each symbol, descend the tree from the root to the symbol, while assigning '1' to each right branch and '0' to each left branch.

An example from [2] is based on the relative frequencies of the letters in the English alphabet (occurring in a text), as given in the table below. Its decoding tree (Huffman tree) is also given.

Huffman trees and codesi - Example

Letter	Frequency	Huffman code
A	0.073	1011
B	0.009	000100
C	0.030	01011
D	0.044	0000
E	0.130	100
F	0.028	01010
G	0.016	011110
H	0.035	10100
I	0.074	1100
J	0.002	011100001
K	0.003	01110010
L	0.035	10101
M	0.025	00011

Letter	Frequency	Huffman code
N	0.078	1111
O	0.074	1101
P	0.027	01000
Q	0.003	01110001
R	0.077	1110
S	0.063	0110
T	0.093	001
U	0.027	01001
V	0.013	000101
W	0.016	011101
X	0.005	01110011
Y	0.019	011111
Z	0.001	011100000

Figure: Frequencies and Huffman encoding of English alphabet symbols [2]

Huffman trees and codesi - Example

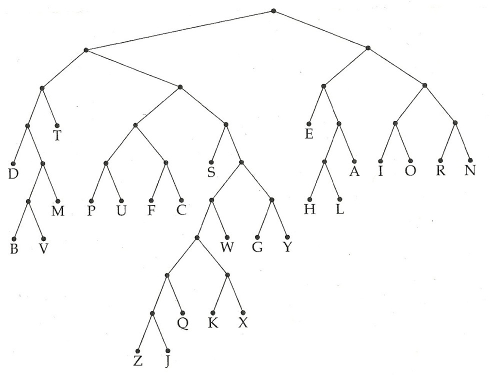


Figure: Huffman (encoding) tree for English alphabet symbols [2]

Graphs - Terminology

Graph $G = (V, E)$: has set of vertices V and set of edges E.
G can be a directed graph (with directed edges), or an undirected graph (with undirected edges).
The presence of an edge (v_i, v_j) indicates that vertex v_j is adjacent to v_i (vertex v_i is adjacent from v_j; for an undirected graph, v_i and v_j are just called adjacent).
A weighted graph has weights on the edges, weight w_{ij} on edge (v_i, v_j).
A path in G consists of a sequence of vertices $v_1, v_2, \ldots, v_k$ (for $k > 1$), such that (v_i, v_{i+1}) is an edge in G for $1 \leq i \leq k-1$. This path is of length $k-1$. A path $v_1, v_2, \ldots, v_k$ where $v_k = v_1$ is a cycle.

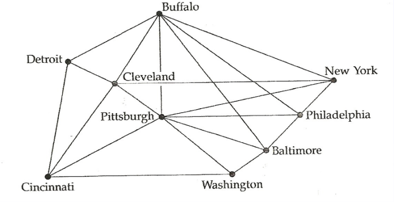


Figure: Undirected graph - Cities example [2]

Graphs - Terminology

Graph representations: *(i)* adjacency matrix; *(ii)* adjacency list
(i) For a graph $G = (V, E)$ with n vertices ($|V| = n$), the adjacency matrix is an $n \times n$ matrix. For a weighted graph, the (i, j) matrix element is the weight $w_{i\,j}$ on edge (v_i, v_j). The adjacency matrix is symmetric for an undirected graph (see below for the Cities example [2]).

	1	2	3	4	5	6	7	8	9	
1		345					97	230	39	Baltimore
2	345			186	252	445	365	217		Buffalo
3				244	265			284	492	Cincinnati
4		186	244		167	507		125		Cleveland
5		252	265	167						Detroit
6		445		507			92	386		New York
7	97	365				92		305		Philadelphia
8	230	217	284	125		386	305		231	Pittsburgh
9	39		492					231		Washington

Figure: Adjacency matrix representation - Cities example [2]

Graphs - Terminology

(ii) For graph $G = (V, E)$ with n vertices and e edges($|V| = n$, $|E| = e$), the adjacency list data structure has n head nodes (with pointers to n lists). The list for head node i (representing vertex v_i) contains a list node for each vertex v_j adjacent to v_i.
There are e list nodes for a directed graph, and $2e$ list nodes for an undirected graph, see figure for the Cities example. In the list for head node i, the list node for adjacent vertex v_j has a field for the vertex number (j), the weight w_{ij} of edge (v_i, v_j), and a next pointer. The list nodes are in no particular order.

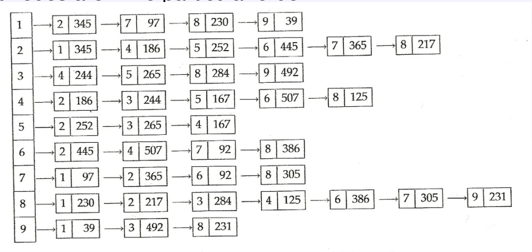


Figure: Adjacency list representation - Cities example [2]

Spanning tree

A spanning tree of an undirected graph $G = (V, E)$ with $|V| = n$ is a tree formed by the n vertices as its nodes and $n - 1$ edges from E.
A connected graph G may have multiple spanning trees. For a weighted graph G, the cost of a spanning tree is the sum of the weights on its edges.
The minimum cost spanning tree is a spanning tree of minimum cost.
The figure below [1] shows a graph (left) and minimum cost spanning tree (right).

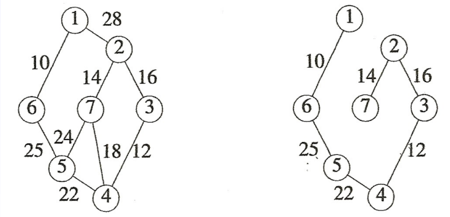


Figure: Graph and minimum cost spanning tree [1]

Kruskal's algorithm - construction

Kruskal's algorithm for minimum cost spanning tree construction is illustrated for the example [1].

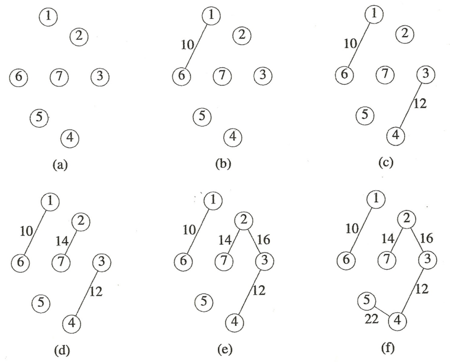


Figure: Kruskal's algorithm for minimum cost spanning tree - construction [1]

Prim's algorithm - construction

Prim's algorithm for minimum cost spanning tree construction is illustrated for the example [1].

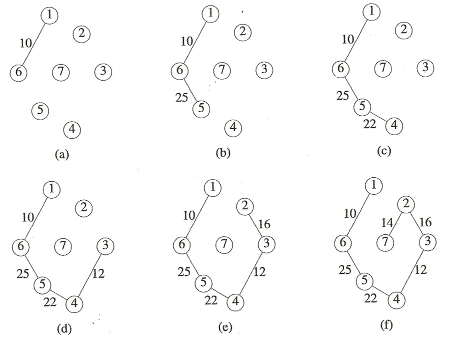


Figure: Prim's algorithm for minimum cost spanning tree - construction [1]

Kruskal's algorithm V1 [1]

```
Algorithm Kruskal1 (G, n, T) { // returns minimum cost spanning tree T for graph G
    T = ∅;   // Initialize T empty
    while ( (T has less than n − 1 edges) and (E ≠ ∅) ) {// Consider next edge to add
        Choose edge (v, w) of lowest cost;
        Delete (v, w) from E;
        if ((v, w) does not create a cycle in T)
            add (v, w) to T;
        else discard (v, w);
    }
}
```

Kruskal's algorithm V2

```
Algorithm Kruskal2 (G, n, T) {   // returns minimum cost spanning tree T for graph G
    T = ∅;   // Initialize T empty (no edges, n one-node subtrees)
    build heap keyed with edge costs;
    while ( (T has less than n − 1 edges) and (E ≠ ∅) ) {// Consider next edge to add
        Choose edge (v, w) of lowest cost;   // delete from heap
        Delete (v, w) from E;
        if ((v, w) does not create a cycle in T)
        // if v and w belong to different subtrees
            add (v, w) to T;   // join (union) the subtrees of v and w
        else discard (v, w);
    }
}
```

Kruskal's algorithm - Analysis

Using a min-heap keyed with the edge costs:
– Building the heap initially requires $\mathcal{O}(e)$ time, where $e = |E|$.
– Within the while loop, with $\mathcal{O}(e)$ iterations:
    delete from heap is done in $\mathcal{O}(\log e)$ time,
    thus $\mathcal{O}(e \log e)$ time throughout the loop.

The accumulated time of lines 5 and 6 throughout the while loop is dominated by the heap processing time by using a suitable data structure for checking whether edge (u, v) would create a cycle.

The union-find data structure is used to perform the following efficiently:

$T_1 =$ subtree containing v;   // *find*(v) in union-find data structure
$T_2 =$ subtree containing w;   // *find*(w) in union-find data structure
if $(T_1 \neq T_2)$ // ((v, w) does not create a cycle in T)
    union(T_1, T_2);   // (add (v, w) to T)

Then the time complexity of Kruskal's algorithm is $\mathcal{O}(e \log e)$.

Prim's algorithm - Analysis

The time complexity of Prim's algorithm as given in [1] is $\mathcal{O}(n^2)$; it is stated that using a red-black tree for the set of nodes that have not yet been included in the tree, yields a version of time complexity $\mathcal{O}((n+e)\log n)$.

Bibliography

S. Sahni E. Horowitz and S. Rajasekeran.
Computer Algorithms/C++.
Computer Science Press, 2nd edition, 1998.
ISBN 0-7167-8315-0.

B. M. E. Moret and H. D. Shapiro.
Algorithms from P to NP, Vol. I - Design and Efficiency.
Benjamin/Cummings Publishing Company, Inc., 1990.
ISBN 0-8053-8008-6.