TIN093 — Algorithms Cheat Sheet

Greedy Algorithms

An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution.

Stays ahead Argument

By this we mean that if one measures the greedy algorithm's progress in a step-by-step fashion, one sees that it does better than any other algorithm at each step; it then follows that it produces an optimal solution

Exchange Argument

One considers any possible solution to the problem and gradually transforms it into the solution found by the greedy algorithm without hurting its quality. Again, it will follow that the greedy algorithm must have found a solution that is at least as good as any other solution.

Dynamic Programming

Essentially the opposite of the greedy strategy: one implicitly explores the space of all possible solutions, by carefully decomposing things into a series of *subproblems*, and then building up correct solutions to larger and larger subproblems.

Example.

$$\mathrm{OPT}(j) = \begin{cases} 0 & \text{if } j{=}0 \\ \max(v_j + \mathrm{OPT}(\mathrm{p}(j)),\ \mathrm{OPT}(j{-}1)) & \text{otherwise} \end{cases}$$

Dont forget to use memoization when implementing the pseudocode.

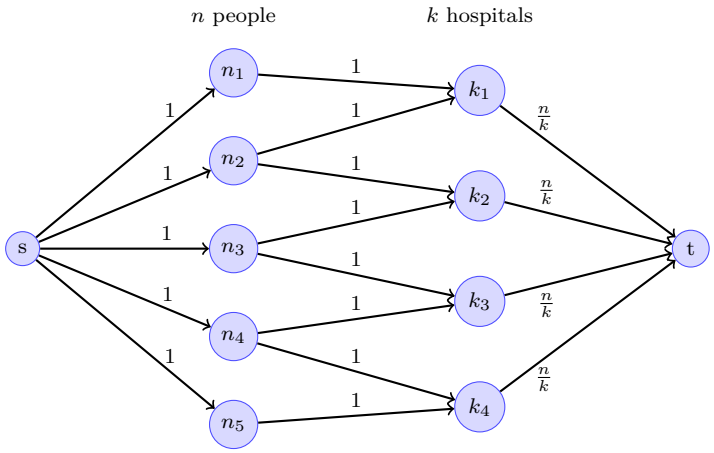
Divide and Conquer

Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these subproblems into an overall solution. In many cases, it can be a simple and powerful method.

Analyzing the running time of a divide and conquer algorithm generally involves solving a recurrence relation that bounds the running time recursively in terms of the running time on smaller instances. We begin the chapter with a general discussion of recurrence relations, illustrating how they arise in the analysis and describing methods for working out upper bounds from them

Example Recurrence (Merge sort)
$T(n){=}2T(\frac{n}{2}){+}c$
Where c is some constant.

Network Flow



- The flow along an edge can not exceed its capacity
- The net flow from u to v must be the opposite of the net flow from v to u. (What goes in, must come out)
- The flow leaving the source s must be equal to the flow arriving at the sink t.

The min cut is equal to the max flow in a flow network.

Ford–Fulkerson: $\mathcal{O}(|E|f)$, f is the maximum flow.
Edmonds–Karp: (a variation of Ford–Fulkerson): $\mathcal{O}(|V||E|^2)$.

NP

$$Y \leq_p X$$

Y is polynomial-time reducible to X, or X is at least as hard as Y (with respect to polynomial time).

Def. Suppose $Y \leq_p X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

NP-Complete problems

To prove that a problem Y is NP-complete you have to:

1. Show that you can verify a solution to Y in polynomial time. ($Y \in$ NP)
2. Give a reduction A from some NP-complete problem X to Y.
3. Prove that $s \in X \Leftrightarrow A(s) \in Y$.

List of problems

Packing	Independent Set
	Set Packing
Covering	Vertex Cover
	Set Cover
Partitioning	3-Dimensional Matching
	Graph Coloring
Sequencing	Hamiltonian Path/Cycle
	Traveling Salesman
Numerical	Subset Sum
Constraint Satisfaction	3-Sat

Graphs

$G(V,E)$

Breadth-First-Search

Perhaps the simplest algorithm for determining $s{-}t$ connectivity is breadth-first search (BFS), in which we explore outward from s in all possible directions, adding nodes one "layer" at a time. Thus we start with s and include all nodes that are joined by an edge to s – this is the first layer of the search. We then include all additional nodes that are joined by an edge to any node in the first layer – this is the second layer. We continue in this way until no new nodes are encountered. Time complexity: $\mathcal{O}(|V|{+}|E|)$.

Depth-First-Search

Another natural method to find the nodes reachable from s is the approach you might take if the graph G were truly a maze of interconnected rooms and you were walking around in it. You'd start from s and try the first edge leading out of it, to a node v. You'd then follow the first edge leading out of v, and continue in this way until you reached a "dead end" – a node for which you had already explored all its neighbors. You'd then backtrack until you got to a node with an unexplored neighbor, and resume from there. We call this algorithm depth-first search (DFS), since it explores G by going as deeply as possible and only retreating when necessary. Time complexity: $\mathcal{O}(|V|{+}|E|)$.

Minimum Spanning Tree

A subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Prim's algorithm

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Time complexity when using binary heap priority queues and adjacency list: $\mathcal{O}(|E|\log|V|)$.

You made this? I made this – Daniel Pettersson