Algo Cheat Sheet

by lavas via cheatography.com/27670/cs/8074/

BFS

Takes a root node and sets the distance to 0 and puts it in a queue(FIFO), while every other node is set to infinity. Iteratively explores the neighbors of the dequeued node and adds them to queue and updates their distance.
O(V+E)

DFS

Initialize each node as unvisited and no time of arrival or departure. From an established root node recursively visit a vertex of the node, noting the time of arrival and finish. Set the node as visited when setting the finishing time.
O(V+E)

Decomposing a directed graph into its strongly connected components
Determining articulation points, bridges & biconnected components of an undirected graph

DFS Parenthesis Theorem

In any DFS of a graph G = (V, E), for any two vertices u and v, exactly one of the followings holds:
1. The interval [d[u], f[u]] and [d[v], f[v]] are entirely disjoint
2. The interval [d[u], f[u]] is contained entirely within the interval [d[v], f[v]], and u is a descendant of v in the depth-first tree
3. The interval [d[v], f[v]] is contained entirely within the interval [d[u], f[u]], and v is a descendant of u in the depth-first tree

Classification of Edges

DFS can be used to classify edges of G:
1. Tree edges: edges in the depth-first forest G . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. Back edges: edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges.

Classification of Edges (cont)

3. Forward edges: nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. Cross edges: all other edges.
Modify DFS so that each edge (u, v) can be classified by the color of the vertex v that is reachable when the edge is first explored:
1. WHITE indicates a tree edge
2. GRAY indicates a back edge
3. BLACK indicates a forward or cross edges

Topological Sort

1. Call DFS(G) to compute finishing time f[v] for each vertex
2. As each vertex is finished, insert it onto the front of linked list
3. Return the linked list of vertices
O(V+E)

Greedy

Greedy algorithms are typically used to solve optimization problems & normally consist of
Set of candidates
Set of candidates that have already been used
Function that checks whether a particular set of candidates provides a solution to the problem
Function that checks if a set of candidates is feasible
Selection function indicating at any time which is the most promising candidate not yet used
Objective function giving the value of a solution; this is the function we are trying to optimize

Kruskal's MST Algorithm

```
A <- 0 // initially A is empty
for each vertex v   V[G] // line
2-3 takes O(V) time
     do Create-Set(v) // create
set for each vertex
sort the edges of E by
nondecreasing weight w
for edge E, in order by
nondecreasing weight
     do if Find-Set(u) != Find-
Set(v) // u&v on different trees
           then A <- A U {(u,v)}
                         Union(u,v
)
 return A
Create-Set(u): create a set
containing u.
Find-Set(u): Find the set that
contains u.
Union(u, v): Merge the sets
containing u and v.
O(E*log(E))
```

Dynamic Programming

Similar to divide-and-conquer, it breaks problems down into smaller problems that are solved recursively.
In contrast, DP is applicable when the sub-problems are not independent, i.e. when sub-problems share sub-sub-problems. It solves every sub-sub-problem just once and save the results in a table to avoid duplicated computation.

Applicability to Optimization Problems

Optimal sub-structure (principle of optimality): for the global problem to be solved optimally, each sub-problem should be solved optimally. This is often violated due to sub-problem overlaps. Often by being “less optimal” on one problem, we may make a big savings on another sub-problem.

Applicability to Optimization Problems (cont)

Small number of sub-problems: Many NP-hard problems can be formulated as DP problems, but these formulations are not efficient, because the number of sub-problems is exponentially large. Ideally, the number of sub-problems should be at most a polynomial number

LP Overview

Decision variables - mathematical symbols representing levels of activity of a firm.
Objective function - a linear mathematical relationship describing an objective of the firm, in terms of decision variables - this function is to be maximized or minimized.
Constraints – requirements or restrictions placed on the firm by the operating environment, stated in linear relationships of the decision variables.
Parameters - numerical coefficients and constants used in the objective function and constraints.

Selection

Divide the n elements of input array into n/5 groups of 5 elements each and at most one group made up of the remaining (n mod 5) elements.
Find the median of each group by insertion sort & take its middle element (smaller of 2 if even number input).
Use Select recursively to find the median x of the n/5 medians found in step 2.
Partition the input array around the median-of-medians x using a modified Partition. Let k be the number of elements on the low side and n-k on the high side.
Use Select recursively to find the ith smallest element on the low side if i   k , or the (i-k)th smallest element on the high side if i > k

P

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.
Given a graph connected G, can its vertices be coloured using two colours so that no edge is monochromatic?
Algorithm: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

NP

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.
This means that if someone gives us an instance of the problem and a certificate to the answer being yes, we can check that it is correct in polynomial time.

NP Complete

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.
Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

NP Complete (cont)

It can be shown that every NP problem can be reduced to 3-SAT. The proof of this is technical and requires use of the technical definition of NP (based on non-deterministic Turing machines). This is known as Cook's theorem.
What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time.

NP-Hard

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.
The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.
But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.
The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one.
As another example, any NP-complete problem is NP-hard.