Algorithms Final Cheat Sheet

Randomized Algorithms

The Union Bound

Given events $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$, we have

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr\left[\mathcal{E}_i\right]$$

We usually want the union on the left-hand side to represent a "bad event" that we're trying to avoid, and want a bound on its probability in terms of constituent "bad events" on the right-hand side.

Verifying AB=C

Choose a random vector $\overline{r} = (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$. Then compute $B\overline{r}$ and then $A(B\overline{r})$; finally, compute $C\overline{r}$.
If $A(B\overline{r}) \neq C\overline{r}$, then $AB \neq C$.
If $AB \neq C$ and if $\overline{r}$ is chosen uniformly at random from $\{0,1\}^n$, then $\Pr(AB\overline{r} = C\overline{r}) \leq \frac{1}{2}$.
Repeated trials increase the runtime to $\Theta(kn^2)$, and probability of error at most 2^{-k}.
If it returns false, then it is always right, but if it returns true, then it does so with some probability of error.

Law of Total Probability

Let $E_1 \ldots E_n$ be mutually disjoint events in the sample space Ω and let $\bigcup_{i=1}^{n} E_i = \Omega$. Then

$$\Pr(B) = \sum_{i=1}^{n} \Pr(B \cap E_i) = \sum_{i=1}^{n} \Pr(B|E_i)\Pr(E_i)$$

Expected Value

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j]$$

Suppose we have a coin which has a probability $p > 0$ of landing heads, and a probability $1 - p$ of landing tails. If we flip the coin until we get a heads, what's the expected number of flips we will perform?
For $j > 0$, we have $\Pr[X = j] = p(1-p)^{j-1}$; in order for the process to take exactly j steps, the first $j - 1$ flips must come up tails, and the jth must come up heads.
Thus, $E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \frac{1}{p}$.

Linearity of Expectation

Given 2 random vars X and Y in the same probability space, $E[X + Y] = E[X] + E[Y]$
Expected Value of sum of two dice throws=
2*1/36 + 3*1/6 + ... + 7*1/36+
3*1/36 + 4*1/6 + ... + 8*1/36 +
.....
.....
7*1/36 + 8*1/6 + ... + 12*1/36
= 7
Expected Value of sum of 2 dice throws = 2*(Expected value of one dice throw) = 2*(1/6 + 2/6 + ... 6/6) = 2*7/2 = 7
Expected value of sum for n dice throws is = n * 7/2 = 3.5 * n

Memoryless guessing

You attempt to guess cards uniformly at random as they are revealed in a shuffled deck. You can expect to get 1 correct, independent of n (use Linearity of Expectation).

Memory guessing

Now, if you remember which cards have been revealed (and randomly choose un-revealed cards), you can expect to correctly predict $H(n) = \Theta(\log n)$ cards (this is the harmonic series).

Coupon collection

Suppose a cereal brand offers n types of coupons in boxes, at random. How many do you expect to buy before getting a coupon of each type?
Let j be the number of coupons already collected. Then $E[X_j] = n/(n-j)$, since $(n-j)/n$ is the odds of getting a new coupon. Thus, $E[X] = nH(n) = \Theta(n \log n)$.

Conditional Expectation

Define the *conditional expectation* of X, given $\mathcal{E}$, to be the expected value of X computed only over the part of the sample space corresponding to $\mathcal{E}$. Then
$E[X|\mathcal{E}] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j|\mathcal{E}]$.

Quicksort

when pivot is selected uniformly at random the expected number of comparisons in quicksort is $O(n \log n)$.

Hashing

Let n be the number of items stored in the map, and m be the size of the map.

Uniform

A hashing chosen uniformly at random means that for all x and all i, $\Pr[h(x) = i] = \frac{1}{m}$.

Universal

For any two items in the universe, the probability of collision is as small as possible: $\Pr[h(x) = h(y)] = \frac{1}{m}$ for all $x \neq y$.

Near-universal

The probability of collision is *close* to ideal:
$\Pr[h(x) = h(y)] \leq \frac{2}{m}$ for all $x \neq y$.

k-uniform

The probability that each key maps to the corresponding hash value is $\Pr[\bigwedge_{j=1}^{k} h(x_j) = i_j)] = \frac{1}{m^k}$ for all distinct $x_1, \ldots, x_k$ and all $i_1, \ldots, i_k$.

Load Factor

For a universal hash function, the probability of collision is $E[C_{x,y}] = \Pr[C_{x,y} = 1] = 1$ if $x = y$ and $1/m$ otherwise. Thus, the expected number of collisions is $\frac{n}{m}$.
This load factor is $\alpha = n/m$.

Runtime

Using a linked list, we have a bound for searching the hash table: $\Omega(1 + \alpha)$. Using a balanced binary tree, we have a bound of $O(1 + \log \alpha)$.
If we instead create a recursive hash table, we end up with a bound of $O(\log_m n)$.

Open Addressing

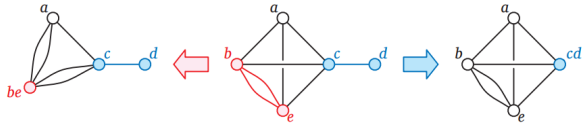
If we simply attempt to fill empty spaces with values, rather than use chaining to address collisions, we reduce our expected time to $O(1)$, unless the hash table is almost completely full.

Max Flows and Min Cuts

Rand MinCut

uniformly at random select edge and collapse it. Repeat until only two nodes left the remaining edges are the ones to be cut.

Suppose G has only one minimum cut, if it actually has more than one just pick your favorite, and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertexes, G must have at least $\frac{kn}{2}$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $\frac{k}{\frac{kn}{2}} = \frac{2}{n}$. In other words, the probability that we don't screw up on the very first step is at least $1 - \frac{2}{n} = \frac{n-2}{n}$. Probability that the overall alg returns the mincut is $P(2) = 1$,
$P(n) \geq \frac{n-2}{n} \cdot P(n-1) = \prod_{i=3}^{n} \frac{i-2}{i} = \frac{2}{n(n-1)}$



Blind Guess

```
function GuessMinCut(G)
    for i ← n, 2 do
        pick a random edge e in G
        G ← G/e
    end for
    return the only cut in G
end function
```

$P(n) = \frac{2}{n(n-1)}$

Repeated Guessing

```
function KargerMinCut(G)
    mink ← ∞
    for i ← 1, N do
        X ← GuessMinCut(G)
        if |X| < mink then
            mink ← |X|
            minX ← X
        end if
    end for
    return minX
end function
```

Set $N = c\binom{n}{2} \ln n$ for some constant c. $P(n) \geq 1 - \frac{1}{n^c}$. KargerMinCut computes the min cut of any n-node graph with high probability in $O(n^4 \log n)$ time.

Not-So-Blindly Guessing

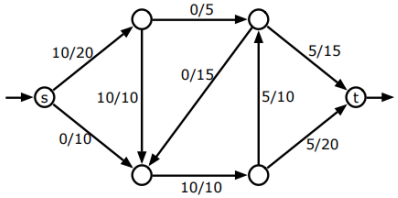
```
function Contract(G, m)
    for i ← n, m do
        pick a random edge e in G
        G ← G/e
    end for
end function
function BetterGuess(G)
    if G has more than 8 vertices then
        G1 ← Contract(G, n/√2 + 1)
        G2 ← Contract(G, n/√2 + 1)
        X1 ← BetterGuess(G1)
        X2 ← BetterGuess(G2)
        return min(X1, X2)
    else
        use brute force
    end if
end function
```

$P(n) \geq 1/\log n$. The running time is $O(n^2 \log n)$.

Flows

A *flow* is a function f that satisfies the *conservation constraint* at every vertex v: the total flow *into* v is equal to the total flow *out* of v.

A flow f is *feasible* if $f(e) \leq c(e)$ for each edge e. A flow *saturates* edge e if $f(e) = f(c)$, and *avoids* edge e if $f(e) = 0$.

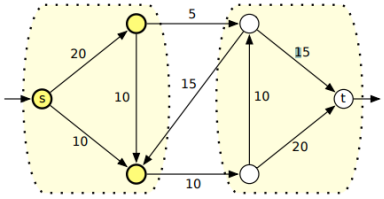


An (s, t)-flow with value 10. Each edge is labeled with its flow/capacity.

Cuts

A *cut* is a partition of the vertices into disjoint subsets S and T - meaning $S \cup T = V$ and $S \cap T = \emptyset$ - where $s \in S$ and $t \in T$.

If we have a capacity function c, the *capacity* of a cut is the sum of the capacities of the edges that start in S and end in T. The definition is asymmetric; edges that start in T and end in S are unimportant. The *min-cut problem* is to compute a cut whose capacity is as large as possible.

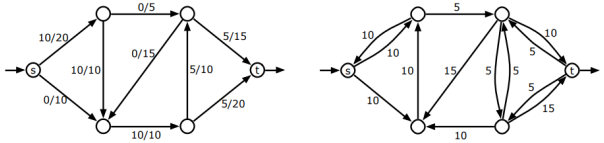


An (s, t)-cut with capacity 15. Each edge is labeled with its capacity.

Theorem 1 (Maxflow Mincut Theorem) *In any flow network, the value of the maximum flow is* equal *to the capacity of the minimum cut.*

Residual Capacity

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$



A flow f in a weighted graph G and the corresponding residual graph G_f.

Augmenting Paths

Suppose there is a path $s = v_0 \to v_1 \to \cdots \to v_r$ in the residual graph G_f. This is an *augmenting path*. Let $F = \min_i c_f(v_i \to v_{i+1})$ denote the maximum amount of flow that we can push through the augmenting path in G_f. We can augment the flow into a new flow function f':

$$f'(u \to v) = \begin{cases} f(u \to v) + F & \text{if } u \to v \in s \\ f(u \to v) - F & \text{if } v \to u \in s \\ f(u \to v) & \text{otherwise} \end{cases}$$

Ford-Fulkerson

Starting with the zero flow, repeatedly augment the flow along *any* path from s to t in the residual graph, until there is no such path.

Further Work

The fastest known maximum flow algorithm, announced by James Orlin in 2012, runs in $O(VE)$ time.

Flow/Cut Applications

Edge-Disjoint Paths

A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths; several edge-disjoint paths may pass through the same vertex, however.
Assign each edge capacity 1. The number of edge-disjoint paths is exactly equal to the value of the flow. Using Orlin's algorithm is overkill; the the maximum flow has value at most $V - 1$, so Ford-Fulkerson's original augmenting path algorithm also runs in $O(|f^*|\, E) = O(VE)$ time.

Vertex Capacities and Vertex-Disjoint Paths

If we require the total flow into (and out of) any vertex v other than s and t is at most some value $c(v)$, we transform the input into a new graph. We replace each vertex v with two vertices v_{in} and v_{out}, connected by an edge $v_{in} \to v_{out}$ with capacity $c(v)$, and then replace every directed edge $u \to v$ with the edge $u_{out} \to v_{in}$ (keeping the same capacity).
Computing the maximum number of *vertex-disjoint* paths from s to t in any directed graph simply involves giving every vertex capacity 1, and computing a maximum flow.

SAT and CNF-SAT

Formula Satisfiability/SAT

Given a boolean formula like $(a \vee b \vee c \vee \overline{d}) \Leftrightarrow ((b \wedge \overline{c}) \vee \overline{(\overline{a} \Rightarrow d)} \vee (c \neq a \wedge b))$, is it possible to assign boolean values to $a, b, c, \ldots$ so that the formula evaluates to True?
We can transform any boolean circuit to a formula in linear time (using depth-first search), and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT. Thus, SAT is NP-hard.
To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula True. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is NP-complete.

CNF

A boolean form is in *conjunctive normal form* if it is a conjunction (**AND**) of several clauses, each of which is the disjunction (**OR**) of several *literals*, each of which is either a variable or its negation. For example, $(a \vee b \vee c \vee d) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b})$.

3CNF

A 3CNF formula is a CNF formula with exactly 3 literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to True?

1. Make sure every **AND** and **OR** gate has only two inputs. If any gate has $k > 2$ inputs, replace it with a binary tree of $k - 1$ two-input gates.
2. Write down the circuit as a formula, with one clause per gate.
3. Change every gate clause into a CNF formula
4. Make sure every clause has exactly three literals

Every binary gate in the original circuit will be transformed into at most 5 clauses.

NP-Hardness

P *is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.*

NP *is the set of decision problems with the following property: if the answer is Yes, then there is a* proof *of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a Yes answer quickly if we have the solution in front of us.*

co-NP *is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.*

NP-hard: *A problem* Π *is* NP-hard *if a polynomial-time algorithm for* Π *would imply a polynomial-time algorithm for every problem in NP.*

NP-complete *is a problem that is both NP-hard and an element of NP.*

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.
Every decision problem in P is also in NP and also in co-NP.

Theorem 2 (Cook-Levin Theorem) *Circuit satisfiability is NP-complete.*

A **many-one** *reduction from one language* $L' \subseteq \Sigma^*$ *is a function* $f : \Sigma^* \to \Sigma^*$ *such that* $x \in L'$ *iff* $f(x) \in L$. *A* language L *is NP-hard iff, for any language* $L' \in NP$, *there is a* many-one *reduction from* L' *to* L *that can be computed in polynomial time.*

NP-Hard Problems

- SAT
- 3SAT
- Maximum Independent Set: find the size of the largest subset of the vertices of a graph with no edges between them
- Clique: Compute the number of nodes in its largest complete subgraph
- Vertex Cover: Smallest set of vertices that touch every edge in the graph
- Graph Coloring: Find the smallest possible number of colors in a legal coloring such that every edge has two different colors at its endpoints
- Hamiltonian Cycle: find a cycle that visits each vertex in a graph exactly once
- Subset Sum: Given a set X of positive integers and an integer t, determine whether X has a subset whose elements sum to t
- Planar Circuit SAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output True
- Not All Equal 3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one True literal *and* at least one False literal?
- Partition: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \emptyset$, and $\sum_{a \in A} a = \sum_{b \in B} b$?
- 3Partition: Given a set S of $3n$ integers, can it be partitioned into n disjoint three-element subsets, such that every subset has exactly the same sum?

- Set Cover: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$
- Hamiltonian Path: Given a graph G, is there a path in G that visits every vertex exactly once?
- Longest Path: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once.

Subset Sum

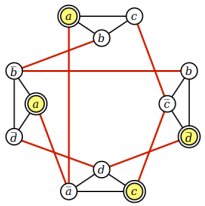
Given a graph G and an integer k, first number edges from 0 to $m-1$; set X contains the integer $b_i = 4^i$ for each edge i, and the integer $a_v = 4^m + \sum_{i \in \delta(v)} 4^i$ where $\delta(v)$ is the set of edges that have v as an endpoint. Finally, we set the target sum: $t = k * 4^m + \sum_{i=0}^{m-1} 2 * 4^i$

Longest Increasing Subsequence [from DAG]

Turn every number in the sequence into a vertex in a graph. Construct a special vertex d. For each vertex v_1, construct an edge to another vertex v_2 if (1) v_2 comes after v_1 in the sequence, and (2) $v_2 > v_1$. Also construct an edge to d.
Every path on this graph is a valid increasing subsequence. The problem of finding the LIS is now the problem of finding the longest path on this graph. Apply DAG.

Maximum Independent Set [from 3SAT]

Construct a graph G which has one vertex for each instance of each literal in the 3SAT formula. Two vertices are connected by an edge if (1) they correspond to literals in the same clause, or (2) they correspond to a variable and its inverse. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into:

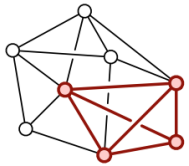


A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

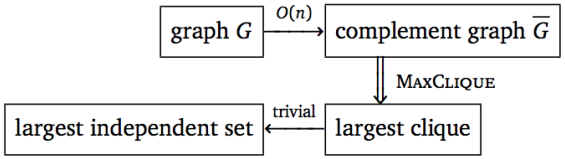
Suppose the original formula had k clauses. Then the formula is satisfiable iff the graph has an independent set of size k.

Clique [from Independent Set]

Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges - (u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, we can compute the largest independent set in a graph by computing the largest clique in the complement of the graph.

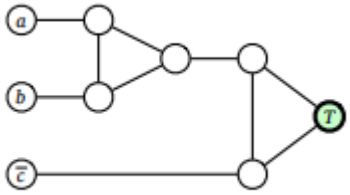


A graph with maximum clique size 4.

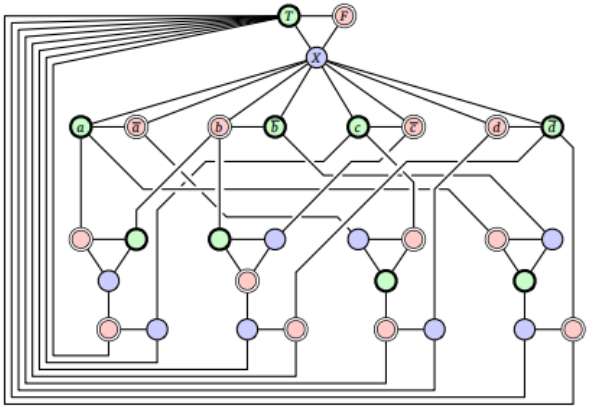


3Color [from 3SAT]

Truth gadget: T, F, and X for true/false/other, variable gadget for variable a connecting a and $\bar{a}$ which must be opposite bools. Clause gadget joining three literal nodes to node T in the truth gadget using give new unlabeled nodes and ten edges.



A clause gadget for $(a \vee b \vee \bar{c})$.



A 3-colorable graph derived from the satisfiable 3CNF formula
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

bipartite graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.
A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.