Algorithms Final Cheat Sheet

Processes (w no communication) accessing a single DB
Probability of any success = $p*(1-p)^n - 1$ ...
$p = \Theta(1/n)$ ... $t = t$; prob of fail $\le 1/e$ ...
$t = en * cln(n)$; prob of fail $\le n^{-c}$ ... How long until succeeding at least once: n^{-1}

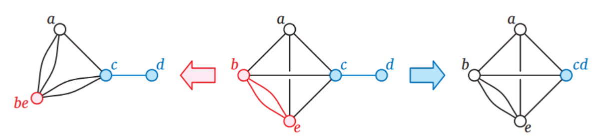
The Union bound: Prob [$\bigcup_{i=1}^{\infty} F_i$] <= $\sum_{n=1}^{\infty} Prob[F_i]$

Verifying AB=C matrix mult
$B\bar{r} -> A(B\bar{r})$ and $C\bar{r}$. if $A(B\bar{r})! = C\bar{r}$ then $AB! = C$
Principle of Deferred Decisions: If $AB! = C$ and $\bar{r}$ is chosen uniformly at random with r_n at 0-1 then Prob($AB\bar{r} = C\bar{r}$) $<= 1/2$
Law of Total Probability: Let $E_1 ... E_n$ be mutually disjoint events in the sample space Ω and let $\bigcup_{i=1}^{n} E_i$ then $\sum_{i=1}^{n} Prob[B|E_i]Prob[E_i]$
Repeated trials increase the runtime to $\Theta(kn^2)$ If it returns false, then it this is right, but if it returns true, then it returns so with some probability of mistake.
Average value: $\sum_{j=1}^{\infty} j * Prob[X = j] = (n+1)/2$ while Prob $[X = j] = 1/n$
To take exactly j steps: Prob $[X = j] = (1-p)^{j-1}p$ $1/p$ for the first success
Linearity of Expectation: Given 2 random vars X and Y in the same probability space, $E[X+Y] = E[X] + E[Y]$
Memoryless guessing expected correct: 1, independent of n
Memory guessing; $H(n) = \Theta(log(n))$ [harmonic series]
Coupon collection: $E[X_j] = n/(n-j)$; n= number of; j = collected; $(n-j)/n$ of getting a new one
$E[X] = nH(n) = \Theta(nlog(n))$
Conditional Probability: $E[X|\alpha] = \sum_{j=0}^{\infty} j*$ Prob $[X = j|\alpha]$
For the MAX 3-SAT there is a randomized algo with polynomial expected run time that is guaranteed to produce a truth assignment satisfying at least a $7/8$ fraction of all clauses. We would need $8k$ trials to get the satisfying assignment.

```
function SELECT(S, k)
    a_i ← randomvarinS
    for each element a_j of S do
        S⁻.append(a_j) if a_j < a_i
        S⁺.append(a_j) if a_j > a_i
    end for
    if S⁻ = k − 1 then
        return a_i
    else if S⁻ ≥ k then
        Select(S⁻, k)
    else
        Select(S⁺, k − 1 − |S⁻|)
    end if
end function
```

Expected num comparisons quicksort:
$2nln(n) + O(n), \Omega(nlogn)$
Uniform: Prob $[h(x) = i] = 1/m$ for all x and all i
Universal: Prob $[h(x) = h(y)] = 1/m$ for all $x! = y$
Near-universal: Prob $[h(x) = h(y)] \le 2/m$ for all $x! = y$
k-uniform: Prob $[\wedge_{j=1}^{k} h(x_j) = i_j)] = 1/m^k$ for all distinct $x_1 ... x_k$ and all $i_1 ... i_k$

Max Flows and Min Cuts



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. $P(n) \ge 1 - \frac{1}{n^c}$. KARGERMINCUT computes the min cut of any n-node graph with high probability in $O(n^4 \log n)$ time.

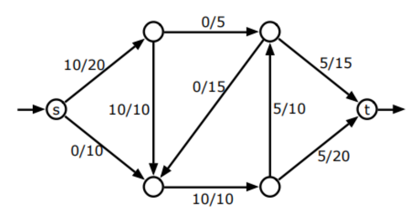
Not-So-Blindly Guessing

```
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
```

$P(n) \ge 1/\log n$. 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 $f(e) \le c(e)$ for each edge e. A flow *saturates* edge e if $f(e) = f(c)$, and *avoids* edge e if $f(e) = 0$.

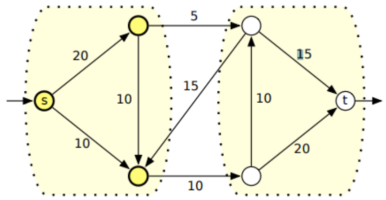


An (s,t)-flow with value 10. Each edge is labeled with its flow/capacity.

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.

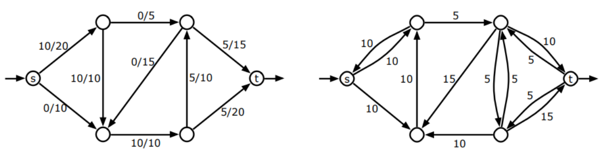


An (s,t)-cut with capacity 15. Each edge is labeled with its capacity.

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.
SATandCNFSAT

NP-Hardness

Definition 2 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.*

Definition 3 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.*

Definition 4 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.*

Every decision problem in P is also in NP and also in co-NP.

Definition 5 *A problem* Π *is* NP-hard *if a polynomial-time algorithm for* Π *would imply a polynomial-time algorithm for every problem in NP.*

Definition 6 *A problem* Π *is* NP-complete *if it is both NP-hard and an element of NP.*

Theorem 7 (Cook-Levin Theorem) *Circuit satisfiability is NP-complete.*

To prove that problem A is NP-hard, reduce a known NP-hard problem to A.

Definition 8 *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?

- Exact 3-Dimensional Matching: Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection of disjoint triples that exactly cover S?
- 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 $\bigcup_i S_i$
- Hitting Set: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$
- 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.
- Steiner Tree: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex?