Algorithms & Data Structures
Summary

Reinier Sanders

January 12, 2022

Contents

Chapter 1

Growth of Functions

Complexity of algorithms is denoted by the Big-$\mathcal{O}$ notation. This represents an asymptotic bound on how long the algorithm will run, expressed in terms of the input.

1.1 Asymptotic Upper Bound

The asymptotic upper bound of a function is denoted by $\mathcal{O}$. Formally:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}^+ |\ \exists c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

In words $f \in \mathcal{O}(g)$ means: "asymptotically" f doesn't grow faster than g.

Lemma

Reflexivity:

$$f \in \mathcal{O}(f)$$

Lemma

Transitivity:

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \Rightarrow f \in \mathcal{O}(h)$$

1.2 Asymptotic Lower Bound

The asymptotic lower bound of a function is denoted by Ω. Formally:

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R}^+ |\ \exists c \in \mathbb{R}^+\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$$

In words $f \in \Omega(g)$ means: "asymptotically" f doesn't grow slower than g.

Lemma

Transpose Symmetry:

$$f \in \mathcal{O}(g) \Leftrightarrow g \in \Omega(f)$$

1.3 Asymptotic Tight Bound

The asymptotic tight bound of a function is denoted by Θ. Formally:

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

In words $f \in \Theta(g)$ means: "asymptotically" f grows as fast as g.

Chapter 2

Breadth-First Search

2.1 Graphs

A graph is a pair $G = \{V, E\}$, with V the set of vertices/nodes and $E \subseteq V \times V$ the set of edges.

$(u, v) \in E$ is an edge from u to v, denoted as $u \to v$. We say that vertex v is adjacent/neighbours vertex u. This adjacency is symmetric if G is undirected, or if the edge is bidirectional in a directed graph.

$|E| = \mathcal{O}(|V|^2)$.

Two variants: directed (as defined above) and undirected, where $(u, v) \in E \Leftrightarrow (v, u) \in E$.

Graphs can be weighted, where each edge has an associated weight given by a weight function $w : E \to \mathbb{R}$. Besides that, graphs can also be dense ($|E| \simeq |V|^2$), or sparse ($|E| \ll |V|^2$).

A graph can be connected, meaning there is a path between every pair of vertices ($|E| \geq |V| - 1$).

2.2 Representation of (Finite) Graphs

Dense graphs: use adjacency matrix. Sparse graphs: use adjacency lists.

2.2.1 Adjacency Lists

Each vertex is given a list of its adjacent vertices. These lists are then stored in an array *Adj*. So for $u \in V$, $Adj[u]$ consists of all vertices adjacent to u. Weights of the edges between vertices can also be stored in this list.

Storage requirement for both directed and undirected graphs is $\Theta(V + E)$.

Adjacency lists are space efficient when a graph is sparse and can be modified to support many graph variants. Determining if an edge $(u, v) \in E$ is not efficient. We have to search in u's adjacency list which takes $\Theta(\text{degree}(u))$ time, $\Theta(V)$ in the worst case.

2.2.2 Adjacency Matrix

An adjacency matrix is a matrix $A = |V| \times |V|$. A is given by $A[i,j] = a_{ij} = 1$ if $(i,j) \in E$, or 0 otherwise. For undirected graphs $A = A^T$ holds.

Storing requirement is $\Theta(V^2)$, so it is not memory efficient for sparse graphs. To list all vertices adjacent to u takes $\Theta(V)$ time and to determine if $(u,v) \in E : \Theta(1)$. Instead of bits, the weight of the edges can also be stored.

2.3 Breadth-First Search (BFS)

Input: a graph G, either directed or undirected and a source vertex $s \in V$. It outputs for all $v \in V : d[v] = \delta(s,v)$ (length of the shortest path from s to v, this is ∞ if there is no path) and $\pi[v] = u$ such that (u,v) is the last edge on the shortest path from s to v. u is then v's predecessor.

Expands frontier of undiscovered vertices uniformly across breadth of the frontier. Vertex is discovered when first encountered during search. Vertex is finished if all adjacent vertices have been discovered. Figure 2.1 shows the order in which nodes are discovered.

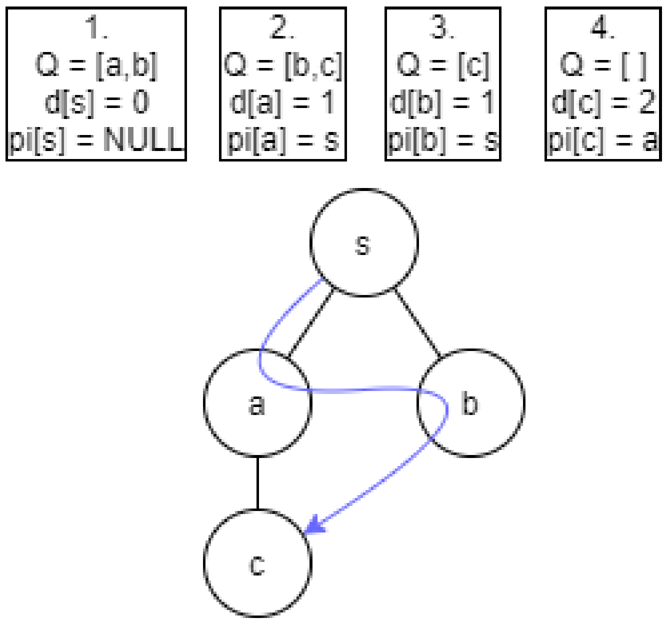


Figure 2.1: Order of node traversal in BFS.

Total running time of BFS is $\mathcal{O}(V + E)$, linear in the size of the adjacency list representation of the graph.

Chapter 3

Depth-First Search

3.1 Depth-First Search (DFS)

Input: a graph G, either directed or undirected. No source vertex is given. It outputs two timestamps on each vertex. Integers between 1 and $2|V|$: $d[v]$ = discovery time (v goes from undiscovered to discovered) and $f[v]$ = finishing time (v goes from discovered to finished). For all $v : d[u] < f[u]$. For all v, u with $v \neq u : d[u] \neq d[v]$ and $d[u] \neq f[v]$. It also outputs $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.

Whereas BFS explores nodes in the order they are added to the frontier, DFS first explores the edges of the most recently discovered vertex v. When all the edges of v have been explored, backtrack to explore other edges leaving v predecessor. As the name suggests, searches deep first. Continues until all vertices reachable from original source are discovered. If undiscovered vertices remain, one of them is chosen as new source and DFS is repeated. Figure 3.1 shows the order in which nodes are discovered.

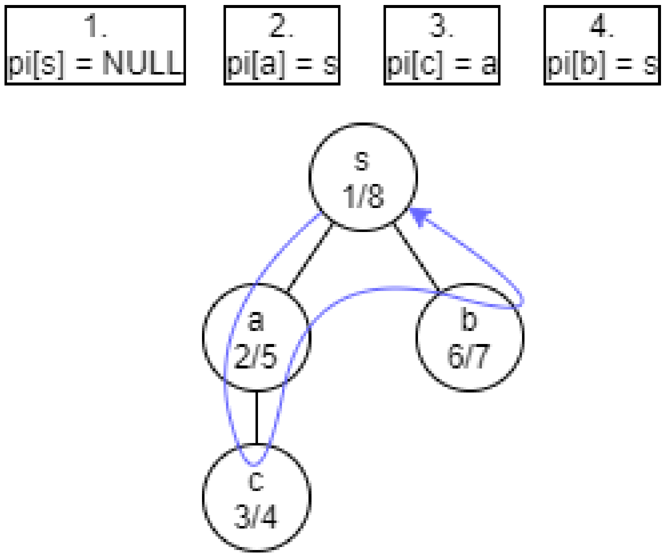


Figure 3.1: Order of node traversal in DFS.

Total running time of DFS is $\Theta(V + E)$.

3.2 Depth-First Trees

Predecessor subgraph forms a depth-first forest, composed of several depth-first trees. Edges are called tree edges and there are 4 different kinds:

- Tree edge: found by exploring (u, v).
- Back edge: (u, v), where u is a descendant of v in the depth-first tree.
- Forward edge: (u, v), where v is a descendant of u, but not a tree edge.
- Cross edge: any other edge. Can go between vertices in the same tree or in different trees.

3.3 Directed Acyclic Graphs (DAG)

Directed graph with no cycles. Good for structures that have a partial order: $a > b$ and $b > c \Rightarrow a > c$. It is always possible to make a total order from a partial order. Often used to model dependency graphs.

Lemma

A directed graph G is acyclic if and only if DFS of G yields no back edges.

This is proven by showing that a back edge implies a cycle. Suppose there is a back edge (u, v). Then v is ancestor of u in depth-first forest. Therefore there is a path $v \leadsto u$, so $v \leadsto u\ v$ is a cycle.

3.4 Topological Sort

A DAG can be topologically sorted to show total order that extends partial order. Used for course prerequisites, module compilation, pipeline of computing jobs, etc. It results in a linear ordering of vertices in G such that if $(u, v) \in E$, then u appears before v.

First DFS is run to compute finishing times for all vertices. Then as each vertex is finished, insert it into the front of a linked list. Return the linked list of vertices. Time complexity: $\Theta(V + E)$.

3.5 Strongly Connected Components (SCC)

G is strongly connected if every pair of vertices is reachable from one another. An SCC is a maximal set of vertices $C \subseteq V$ such that, for all $u, v \in C$, both $u \leadsto v$ and $v \leadsto u$ exist.

The component graph $G^{SCC} = (V^{SCC}, E^{SCC})$. V^{SCC} contains one vertex for each SCC in G. E^{SCC} has an edge if there's an edge between the corresponding SCCs in G.

G^{SCC} is a DAG, since there can be no cyclic paths between two distinct strongly connected components. If there is, then they would be the same SCC.

Lemma

Let C and C' be distinct SCC's in G. Let $u, v \in C$ and $u', v' \in C'$. Suppose there is a path between u and u' in G. Then there cannot also be a path between v' and v in G.

G^T is the transpose of directed graph G, meaning it has all its edges reversed in direction. G^T can be made in $\Theta(V + E)$ time if adjacency lists are used. G and G^T naturally have the same SCCs.

To determine SCCs, DFS is run on G to determine finishing times for all vertices. G^T is computed. DFS is run on G^T, but in the main loop vertices are considered in order of decreasing finishing times. The vertices in each tree of the depth-first forest formed in the second DFS are output as separate SCC. The complexity of this algorithm is $\Theta(V + E)$.

Lemma

Let C and C' be distinct SCCs in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

Chapter 4

Heaps & Dijkstra

4.1 Heaps

A heap is a data structure. It can be sorted with Heapsort, but also has priority queues as an application besides sorting.

4.1.1 Binary Heap Data Structure

A heap is stored in an array in such a way that it represents a binary tree. Array elements can be mapped to tree nodes and vice versa:

- Root = $A[0]$
- Left$[i]$ = $A[2i]$
- Right$[i]$ = $A[2i+1]$
- Parent$[i]$ = $A[\lfloor i/2 \rfloor]$

Length$[A]$ = number of elements in array A. Heap-size$[A]$ = number of elements in heap stored in A. Heap-size$[A] \leq$ Length$[A]$.

An example heap as an array can be $A = \{26, 24, 20, 18, 17, 19, 13, 12, 14, 11\}$. The first element is always the root of the tree. The following elements are child nodes. These fill the tree from left to right, going down one level once every parent node has at most 2 child nodes. The final row is simply filled left to right.

Heaps can be either:

- Max-Heaps: for every node excluding the root, value is at most that of its parent. Largest element is stored at the root. In any subtree, no values are larger than the value stored at the root of the subtree.
- Min-Heaps: for every node excluding the root, value is at least that of its parent. Smallest element stored at the root. In any subtree, no values are smaller than the value stored at the root of the subtree.

The height of a node in the tree is the number of edges on the longest downward path from the node to a leaf. The height of the tree is the height of the root. Height of a heap with n nodes = $\lfloor \log(n) \rfloor$. Number of leaves = $\lceil n/2 \rceil$. Number of nodes of height $h \leq \lceil n/2^{h+1} \rceil$. Basic operations on a heap run in $\mathcal{O}(\log(\mathrm{n}))$ time.

4.1.2 Heapsort

Uses max-heaps to sort items in place, in $\mathcal{O}(n \log n)$ time. Convert the array of size n to a max-heap. Swap first and last elements in the array. Now, largest element is in the last position, where it belongs. That leaves $n-1$ elements to be placed correctly. However, this $n-1$ sized array is no longer a max-heap. Thus float the element at the root down one of its subtrees so that the array remains a max-heap. Repeat the swapping of the first and last elements and max-heapifying until array is sorted.

Maintaining the heap property is done by swapping the value of a node with the larger of its two child nodes. This can cause that subtree to no longer be a heap however, so we need to recursively do this until all subtrees satisfy the max-heap property.

We can convert an array to a max-heap using MaxHeapify. By calling it on each non-leaf element from the bottom-up it recursively swaps elements until the max-heap property is satisfied.

4.1.3 Priority Queues

Another popular and important application of heaps are priority queues. There are max and min priority queues. It maintains a dynamic set S of elements. Each set element has an associated key value. The goal is to support insertion and extraction efficiently.

- Max-priority queue. Insert(S, x), inserts element x into set S. Maximum(S), returns element of S with the highest key. Extract-Max(S), removes and returns element of S with the largest key. Increase-Key(S, x, k), increases the value of element x's key by an amount k.
- Min-priority queue. Insert(S, x), inserts element x into set S. Minimum(S), returns element of S with the lowest key. Extract-Min(S), removes and returns element of S with the lowest key. Decrease-Key(S, x, k), decreases the value of element x's key by an amount k.

Extract-Max, Extract-Min and Insert run in $\mathcal{O}(\log n)$.

4.2 Dijkstra

So far we have looked at unweighted graphs. Dijkstra's algorithm is designed to find the shortest path in weighted graphs.

4.2.1 Shortest Path

The weight of an entire path $w(p)$ is simply the weights of the edges within that path added up. A shortest path from u to v is a path of minimum weight from u to v. The shortest path weight from u to v is defined as:

$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$$

Theorem

A subpath of a shortest path is itself a shortest path.

Theorem

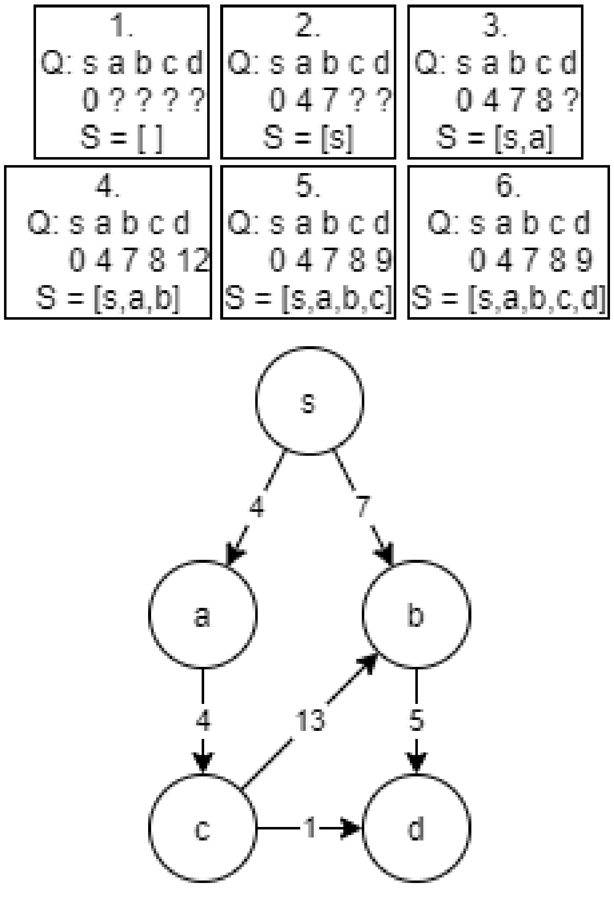
For all $u, v, x \in V$, we have $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$.

If a graph contains a negative-weight cycle, then a shortest path may not exist as you can go into that loop and decrement the path weight indefinitely.

4.2.2 Dijkstra's Algorithm

Dijkstra's algorithm solves the problem: from a given source vertex, find the shortest path weights for all other vertices. If all edge weights are non-negative, all shortest path weights must exist.

A greedy approach is to maintain a set of S vertices, whose shortest path distances from s are known. At each step add the vertex $v \in V - S$ to S, whose distance estimate from s is minimal. Update the distance estimates of vertices adjacent to v. This is depicted in Figure 4.1.



1\.
Q: s a b c d
0 ? ? ? ?
S = [ ]

2\.
Q: s a b c d
0 4 7 ? ?
S = [s]

3\.
Q: s a b c d
0 4 7 8 ?
S = [s,a]

4\.
Q: s a b c d
0 4 7 8 12
S = [s,a,b]

5\.
Q: s a b c d
0 4 7 8 9
S = [s,a,b,c]

6\.
Q: s a b c d
0 4 7 8 9
S = [s,a,b,c,d]

Figure 4.1: Shortest path finding in Dijkstra's Algorithm.

Dijkstra's algorithm has complexity $\Theta(V) \cdot T_{\text{Extract-Min}} + \Theta(E) \cdot T_{\text{Decrease-Key}}$. For an array this means a total of $\mathcal{O}(V^2)$. For a binary heap this becomes $\mathcal{O}(E \cdot \log(V))$. A worst case Fibonacci heap has time complexity $\mathcal{O}(E + V \cdot \log(V))$.

4.2.3 Using BFS as Dijkstra

Suppose all edges in a graph have a weight of 1. In that case we can use a FIFO queue instead of a priority queue. The FIFO queue in BFS now mimics the priority queue in Dijkstra. The invariant here being that v comes after u in Q, which implies $d[v] = d[u]$ or $d[v] = d[u] + 1$. The time complexity now becomes that of BFS: $\mathcal{O}(V + E)$.

Chapter 5

Bellman-Ford

Extension of Dijkstra, used to find all shortest paths from a source vertex s in negative-weight graphs, or it determines that a negative-weight cycle exists. It runs in $\mathcal{O}(V \cdot E)$.

Initially the distances to all vertices other than the source vertex are set to ∞. The algorithm loops 1 to $|V| - 1$ times. In each iteration, each edge $(u, v) \in E$ is checked and if $d[v] > d[u] + w(u, v)$, then $d[v] = d[u] + w(u, v)$. After all vertices are looped over in this fashion, all edges are checked again and if $d[v] > d[u] + w(u, v)$ for some edge, a negative-weight cycle exists. If there is no negative-weight cycle, $d[v] = \delta(s, v)$ for all vertices $v \in V$ when Bellman-Ford terminates. We can see the process detailed in Figure 5.1:

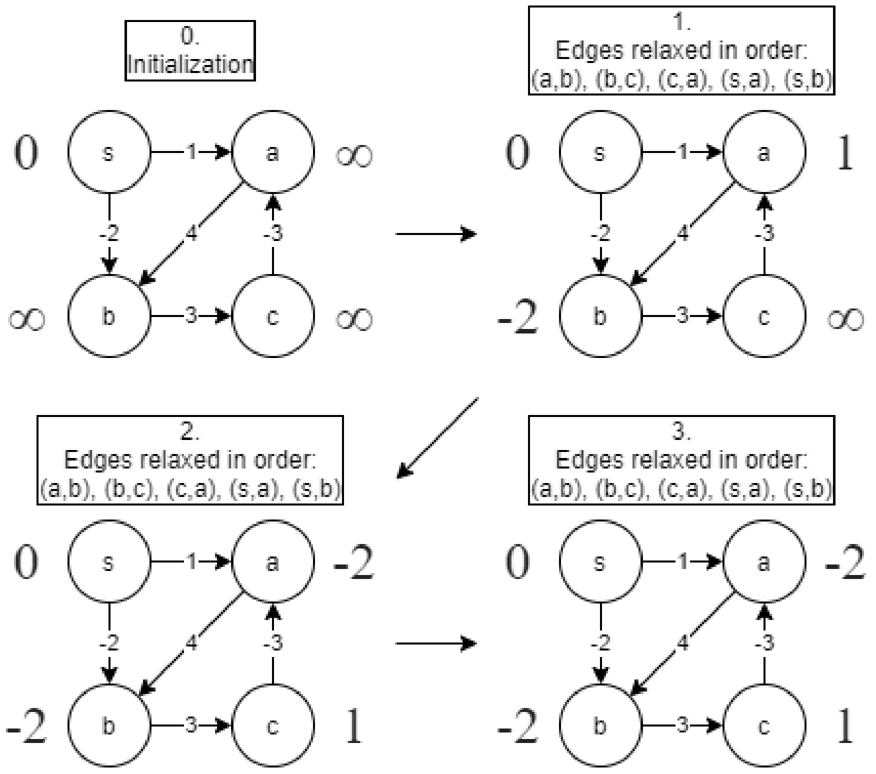


Figure 5.1: Bellman-Ford algorithm.

Chapter 6

Network Flow

A flow network is a simply a weighted directed graph, with a dedicated source (s) and sink (t) node. The source node only has outgoing edges, the sink node only incoming. The edge weights are usually referred to as capacity and are non-negative. Flow networks are used to model a network that transports things between nodes through its edges, where the amount of things you can transport cannot exceed the capacity of an edge.

6.1 Minimum Cut Problem

Flow networks have *st*-cuts, which is a partition (A, B) of the vertices with $s \in A$ and $t \in B$. The capacity of a cut is the sum of capacities of the edges from A to B. The min-cut problem is the problem of finding the cut with minimum capacity.

6.2 Maximum Flow Problem

A flow network has an *st*-flow. This is a function f that satisfies:

- For each $e \in E : 0 \leq f(e) \leq c(e)$ (capacity)
- For each $v \in V - \{s, t\} : \sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (flow conservation)

The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$. The max-flow problem is finding a flow of maximum value. Basically, finding the maximum amount of things you can transport through a given flow network.

A greedy algorithm keeps track of the current flow, which is initially set to 0 in each edge. It then uses BFS or DFS to find an $s \rightsquigarrow t$ path P, where each edge has $f(e) < c(e)$. The flow is then augmented along this path P. This is repeated until you get stuck. This approaches the maximum flow, but does not compute it correctly yet.

6.3 Ford-Fulkerson

To improve this algorithm, we need to make a residual graph. This graph is composed of residual edges. A normal edge $e = (u, v) \in E$ in a flow network has flow $f(e)$ and capacity $c(e)$. For example, an edge $e = (u, v)$ has $f(e) = 6$ and $c(e) = 17$. To make the residual edge we "undo" the flow sent through this edge. We make a new edge $e^R = (v, u)$ and calculate the residual capacity: $c_f(e) = c(e) - f(e)$ if $e \in E$, or $f(e)$ if $e^R \in E$. In our example the residual capacities become $c_f((u, v)) = c((u, v)) - f((u, v)) = 17 - 6 = 11$, since (u, v) was a part of the original graph and is thus in E. The new edge (v, u) gains a residual capacity equal to the original flow in (u, v), namely 6.

If we repeat this process for each edge, we end up with a residual graph $G_f = (V, E_f)$. It consists of residual edges with positive residual capacity. $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$. The key property is that f' is a flow in G_f if and only if $f + f'$ is a flow in G, where the flow on a reverse edge negates the flow on a forward edge.

Now that we have a residual graph, we can find an augmenting path. It is a simple $s \rightsquigarrow t$ path P in the residual graph G_f. The bottleneck capacity of that path P is the minimum residual capacity of any edge in P. The key property is if f is a flow and P an augmenting path in G_f, then f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$. Augmenting paths can be best chosen based on:

- Max bottleneck capacity in path
- Sufficiently large bottleneck capacity
- Fewest number of edges (can use BFS to find shortest paths)

The Ford-Fulkerson algorithm uses residual graphs and augmenting paths to find the maximum flow. Initially the flow on all edges is set to 0. An augmenting path P is found in the residual graph G_f using BFS or DFS. The flow along P is augmented. This is repeated until you get stuck.

Complexity is $\mathcal{O}(mnC)$, where $m = \#$edges, $n = \#$nodes and $C =$ max capacity. If all edges have integer weights, then $\mathcal{O}(E \cdot f)$, with $E =$ amount of edges and $f =$ maximum flow.

Improved by Edmonds-Karp, which runs in $\mathcal{O}(V \cdot E^2)$, independent of the flow.

6.4 Relationship Between Flows and Cuts

The augmenting path theorem states that a flow f is a max-flow if and only if there are no augmenting paths. The max-flow min-cut theorem states that the value of the max-flow is equal to the capacity of the min-cut.

6.5 Bipartite Matching

A matching M is a subset of edges $M \subseteq E$ in an undirected graph in which each node appears in at most one edge in M. Max matching is finding the max cardinality matching in a graph.

A graph is bipartite if the nodes can be partitioned into two subsets L and R such that every edge connects a node in L to one in R. Bipartite matching is finding the max cardinality matching in a bipartite graph.

A bipartite graph can be turned into a flow network by adding source and sink nodes. The source node is connected to all nodes in L, the sink nodes to all nodes in R. Direct all edges from

L to R and assign infinite capacity. Do the same for s and t, but add assign unit (1) capacity to those edges. A max-cardinality matching in bipartite graph G is equal the max-flow in the flow network version of G. Ford-Fulkerson can solve bipartite matching in $\mathcal{O}(m \cdot n)$ time, but can be reduced to $\mathcal{O}(m\sqrt{n})$ (Hopcroft-Karp 1973).

Matching can also be done for non-bipartite graphs, but their structure is more complicated. The best known algorithm can solve it in $\mathcal{O}(m\sqrt{n})$ (Micali-Vazirani 1980, Vazirani 1994).

6.6 Disjoint Paths

Two paths are edge-disjoint if they have no edge in common. The disjoint path problem is finding the max number of edge-disjoint paths in a digraph. If we assign unit capacity to every edge and thus turn the graph into a flow network, then the max flow equals the max number of edge-disjoint paths. This is useful for communication networks by finding out how many edges need to be removed in order to render a communication destination unreachable.

Menger's theorem states that the max number of edge-disjoint paths is equal to the min number of edges whose removal disconnects the destination from source node.

The max number of edge-disjoint paths can also be found for undirected graphs. Each edge is replaced with a bidirectional edge and is assigned a unit capacity. The max flow of this network is then equal to the max amount of edge-disjoint paths in the undirected graph.

6.7 Flow Network Circulation

A flow network can represent a supply and demand chain. Add a new source and sink node. Make an edge from the source node to nodes that have a negative value (so $d[v] < 0$), with capacity $-d[v]$. For nodes with $d[v] > 0$, add an edge to the sink node with capacity $d[v]$. The original graph has circulation if and only if the modified graph has a max flow of value $D = \sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$. This method can be adjusted for upper and lower bounds (i.e. a certain amount of flow needs to go through an edge). Say an edge (u, v) has lower and upper bounds $[2, 9]$. Then the lower bound is added to u and subtracted from v and the upper bound. The new value of the upper bound is the new capacity (in this case $9 - 2 = 7$).

Chapter 7

Divide and Conquer

Divide and conquer algorithms are a collection of algorithms that break up (divide) the problem into smaller subproblems that are easier to solve (conquer). The result of these subproblems is then combined into a solution of the original problem. Usually this strategy is implemented using recursion.

7.1 Merge Sort

Divide array/vector into two vectors of roughly similar size. Recursively sort each subvector (trivial for small sizes). Combine ordered vectors into a new ordered vector via an auxiliary merge function. It has input two ordered sequences `A[p...q]` and `A[q+1...r]`. Outputs ordered vector `A[p...r]`. This function evaluates the first elements of both vectors and places the smallest in a new vector, removing it from the original. It keeps doing this until both vectors are empty. By constantly choosing the next smallest element, the resulting vector is sorted.

7.2 Solving Recurrence Relations

Recursive algorithms can often be described with a recurrence relation. In the case of merge sort, the base case is an array of size 1, so $T(n) = \Theta(1)$.

Dividing the vector in two is constant ($T(n) = \Theta(1)$), conquering the two subproblems: $2T(n/2)$. Merging is linear: $\Theta(n)$. Thus merge sort has complexity $\Theta(1) + 2T(n/2) + \Theta(n)$ if $n > 1$. By solving this recurrence relation, we get $\mathcal{O}(n \log(n))$.

We can rewrite this as $T(n) = c$ if $n = 1$, or $2T(n/2) + cn$ if $n > 1$. From this we can see that each level of the recursion adds cn to the cost. Each level, the problem is split into 2 subproblems of half the size of the previous level. This creates a recurrence tree of height $\log(n)$. Therefore the cost of the whole tree is $cn \cdot \log(n)$ and thus the complexity is $\mathcal{O}(n \log(n))$ (verify using induction, or substitution). This is while assuming the input n is a power of 2. If that is not the case the recurrence becomes $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$ if $n > 1$.

7.3 Karatsuba

Proved that multiplying two n-digit numbers can be done in $\mathcal{O}(n^{\lg(3)}) \approx \mathcal{O}(n^{1.58})$, instead of the previously theorized $\Omega(n^2)$, by using divide and conquer.

Chapter 8

Greedy Algorithms

Another form of algorithm finding strategy. It basically means: Follow the most obvious approach, this is often the correct one. So far we have seen Dijkstra as an example of a greedy algorithm.

8.1 Interval Scheduling

Say we have a job j which starts at some time s_j and is done at some time f_j. Two different jobs j_a and j_b are compatible if the time between their starting and finishing times doesn't overlap. The goal is to find the largest set of jobs that are compatible with each other.

We can consider jobs in some natural order:

- Consider jobs in ascending order of starting time s_j.
- Consider jobs in ascending order of finishing time f_j.
- Consider jobs in ascending order of interval $f_j - s_j$.
- Consider jobs in ascending order of number of conflicts with other jobs c_j.

For each of these cases except the second we can come up with counterexamples. Thus we choose an earliest-finish-time-first approach. We sort the jobs based on their finish times f_j. We then go through each of these in ascending order and add j to the output set A if it is compatible with other jobs already in A. Can be done in $\mathcal{O}(n \log n)$ time.

8.2 Interval Partitioning

Basically same as scheduling, but now we may have d overlapping jobs or activities. In this case we take an earliest-start-time-first approach. Initially $d = 0$. We sort the jobs based on starting time. Now we go through them in ascending order and if a job j is compatible with any 'room' k, we schedule j in k. If not, we allocate a new 'room' $d + 1$, schedule j in that room and assign $d \leftarrow d + 1$. Can be done in $\mathcal{O}(n \log n)$ time.

8.3 Minimum Spanning Trees

A spanning tree of graph G is a maximally acyclic (addition of any edge creates a cycle) and minimally connected (removal of any edge disconnects it) subgraph of G. A minimum spanning tree is a spanning tree of a weighted graph, where the sum of edge costs is minimal.

We can find the minimum spanning tree with Prim's algorithm. It starts at a random node S and repeats $n-1$ times: add the min weight edge with one endpoint in S to the tree and add the new node to S.

Another way is with Kruskal's algorithm. Consider edges in ascending order of weight and add to the tree unless that would create a cycle.

A third way is through the reverse-delete algorithm, which considers edges in descending order of weight and removes an edge unless that would disconnect the graph.

8.4 Stable Matching

A matching is a set S of ordered pairs from sets $M \times W$, where $M = \{m_1, ..., m_n\}$ and $W = \{w_1, ..., w_n\}$, such that each member of M and each member of W appears in at most one pair. A perfect matching is when each of the members appear in exactly one pair.

If we let each member from both sets make a ranking of the other set on which members the prefer the most, then if we have a perfect matching and pairs $(A, B), (C, D)$ are matched but A prefers D over B and B prefers C over A, we call that an instability. A stable matching is thus a perfect matching that contains no instabilities.

We can determine whether a stable matching exists for every set of preference lists, or construct a stable matching from a set of preference lists if it exists using the Gale-Shapley algorithm, which repeatedly breaks up and reforms pairs until they become stable.

Chapter 9

Dynamic Programming

A way to vastly speed up algorithms that would otherwise compute some steps multiple times.

9.1 Fibonacci Numbers

The Fibonacci value of a given number is determined by:

$$\begin{aligned} &F_0 = 0 \\ &F_1 = 1 \\ &F_n = F_{n-1} + F_{n-2} \text{ (for } n > 1) \end{aligned}$$

In code this is:

```
int F(int n) {
    if(n <= 1) {
        return n;
    } else {
        return F(n-1) + F(n-2);
    }
}
```

We can see that for instance for F_5, we need to compute F_4 and F_3. However, in order to compute F_4 we need to compute F_3 again! This time waste on recomputing results can grow exponentially. If instead, we stored the intermediate results in an array, we only have to look them up:

```
int fib[ ];

int F'(int n) {
    if(n <= 1) {
        return n;
    } else if(fib[n] != 0) {
        return fib[n];
    }

    int f = F(n-1) + F(n-2);
    fib[n] = f;
    return f;
}
```

This reduces Fibonacci to $\mathcal{O}(n)$, as the complexity is now only determined by the operations on the array.

9.2 Top-Down vs. Bottom-Up

Problems can be solved top-down, which is subdividing a big problem into smaller ones by way of recursion, or bottom-up where small subproblems are solved and results are combined. This approach removes recursion overhead. For instance bottom-up Fibonacci:

```
int fib[ ];

int F''(int n) {
    fib[0] = 0;
    fib[1] = 1;

    for(int i = 2; i < n; i++) {
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib[n];
}
```

So where divide and conquer splits a big problem into smaller subproblems, dynamic programming splits the problem into overlapping subproblems. Dynamic programming has four steps:

1. Characterize structure for optimal solution.
2. Recursively define value of optimal solution.
3. Compute the value of optimal solution.
4. Construct optimal solution from computed information.

9.3 All-Pairs Shortest Path

The input is a weighted graph $G = (V, E)$ without negative cost cycles. The output is the shortest paths between all pairs of vertices.

A naive algorithm enumerates all pairs of vertices (s, t) and uses Bellman-Ford to compute the shortest path from s to t. This results in a complexity of $\mathcal{O}(n^2)$ pairs $\times$ $\mathcal{O}(n^2)$ to get the shortest path $= \mathcal{O}(n^4)$.

To improve this, we can use the property of a shortest path $p = v_0, v_1, ..., v_n$, that any subpath of p, $p_{ij} = v_i, ..., v_j$, is itself a shortest path.

If we make an adjacency matrix where an edge (u, v) is denoted by its weight (infinite if there is no connection), then we can form the following recurrence relation:

$$d_{ij}^{(k)} = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

With $d_{ij}^{(k)}$ being the distance in the matrix $D^{(k)}$ at row i and column j. Matrix $D^{(0)}$ is then the adjacency matrix we start with and $1 \leq k \leq n$.

$d_{ij}^{(k-1)}$ is the shortest path from i to j that does not go through k. $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ is the path that goes through k.

The Fold-Warshall algorithm is used to find the all-pairs shortest path, given the "weighted adjacency matrix" $D^{(0)}$ of a graph:

```
Floyd-Warshall(n x n matrix D(0)) {
    for(k = 1 to n) {
        D(k) = new n x n matrix;
        for(i = 0 to n) {
            for(j = 0 to n) {
                D(k)[i][j] = min(D(k-1)[i][j], D(k-1)[i][k] + D(k-1)[k][j]);
            }
        }
    }
    return D(n);
}
```

It has a time and space complexity of $\mathcal{O}(n^3)$. After the algorithm is finished we have a full matrix. To extract the actual shortest paths we look at the matrices of predecessors $\Pi^{(k)}$. Where $\pi_{ij}^{(k)}$ is the predecessor of j in a shortest path from i to j that only goes through vertices $\{1, 2, ..., k\}$.

For $k = 0$:

$$\pi_{ij}^{(0)} = \text{ nil if } i = j \text{ or } w_{ij} = \infty$$

$$\pi_{ij}^{(0)} = i \text{ if } i \neq j \text{ and } w_{ij} < \infty$$

For $k > 0$:

$$\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

Going through k does not improve the cost, so we keep the old predecessor.

$$\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

We found a better path that goes through k, and take the predecessor in this path.

9.4 Matrix-Chain Product

The product of two matrices A (size $n \cdot m$) and B (size $m \cdot k$) is given by:

$$C_{ij} = \sum_{l=1}^{m} A_{il} B_{lj}$$

With a complexity of $\mathcal{O}(nmk)$. Matrix product is associative, but different parenthesizations yield different costs. The matrix-chain product has a list of matrices as input and outputs a parenthesization that minimizes the number of multiplicative operations.

A naive algorithm that lists all possible parenthesizations and picks the best one has complexity $\mathcal{O}(4^n)$. We have to define a structure of parenthesizations (which matrices are grouped together). An optimal substructure is then one where both groups are parenthesized optimally.

```
Matrix-Chain-Order(list of int p(0),p(1),...,p(n)) {
    for(i = 1 to n)
        m[i,i] = 0;

    for(l = 2 to n)
        for(i = 1 to n - l + 1) {
            j = i + l - 1;
            m[i,j] = infinity;

            for(k = i to j - 1) {
                q = m[i,k] + m[k + 1,j] + p(i-1)p(k)p(j);
                if(q < m[i,j]) {
                    m[i,j] = q;
                    s[i,j] = k;
                }
            }
        }
    return m and s;
}
```

The time complexity is $\mathcal{O}(n^3)$. Space is $\mathcal{O}(n^2)$.

9.5 Weighted Interval Scheduling

Algorithm to find maximum value subset of compatible jobs, where each job i has a starting time s_i, finishing time f_i and a value v_i. We can keep track of the largest index $i < j$ such that i is compatible with j via $p(j)$. Recurrence:

$$V[j] = \begin{cases} 0 & j = 0 \\ \max(v_j + V[p(j)], V[j-1]) & j > 0 \end{cases}$$

Where $V[j]$ is the value of solution for jobs $1, 2, ..., j$. The choice in the recurrence is either including task j and optimal jobs compatible with $j : v_j + V[p(j)]$), or not including $j : V[j-1]$. Complexity is $\mathcal{O}(n \lg n)$ for time and $\mathcal{O}(n)$ for space.

9.6 Text Justification

Many text editors try to put as many words as possible on the first line, which can lead to ugly gaps in the text. LaTeX minimizes ugly spaces using dynamic programming. Algorithm takes words and their lengths, as well as the length of a line L and outputs the "best" arrangement of words onto lines.

We can express bad arrangements with "badness". Suppose words $w_i, ..., w_j$ form one line.

$$badness(i,j) = \begin{cases} \infty & length(i,j) > L \\ (L - length(i,j))^3 & \text{otherwise} \end{cases}$$

Where $length(i,j) = p_i + ... + p_j +$ spaces between words. The goal is now to split the words such to minimize badness.

$$T[j] = \begin{cases} 0 & j = 0 \\ \min_{1 \le i \le j}(T[i-1] + badness(i,j)) & j > 0 \end{cases}$$

Time complexity $\mathcal{O}(n^2)$, space $\mathcal{O}(n)$.

9.7 Longest Common Subsequence

Measure how related two strings are (for instance DNA in two organisms). Two strings $X = x_1, x_2, ..., x_n$ and $Y = y_1, y_2, ..., y_m$. Output is maxmimum length of common subsequence in X and Y.

$$c[i,j] = \begin{cases} 0 & i = 0 \lor j = 0 \\ c[i-1, j-1] + 1 & i,j > 0 \land x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & i,j > 0 \land x_i \neq y_j \end{cases}$$

Where $c[i,j]$ is the length of an LCS of X_i and Y_j. Time and space complexity of $\mathcal{O}(mn)$.

Chapter 10

Binary Search Trees

A binary tree consists of a root, left subtree (maybe empty) and a right subtree (maybe empty). Each node has ≤ 2 children. This results in a small storage, simple operations and small expected depth (worst case n for a tree with n elements, best case $\log n$ for a balanced tree).

If all values of nodes in the left subtree are smaller or equal to that of the root and all values of nodes in the right subtree are larger or equal to that of the root, the tree is a binary search tree.

A binary search tree is complete when every link is filled, except possibly the bottom layer which is filled left to right.

A binary search tree can be traversed "in-order", by visiting the left subtree, then the node and then the right subtree. This gives an ordered listing of the keys of all the nodes in the tree.

Nodes can be found recursively by checking the left subtree if the to be found value is less than the current node and checking the right subtree if it is larger.

New nodes can be inserted by proceeding down the tree and checking the left and right subtrees' values. If the correct insertion spot is found (i.e. the right subtree at that point is bigger and left is smaller), pointers are made to its parent and children.

Minimum value of a BST can be found by finding the left-most node. Conversely, the maximum value is the right-most node. The successor of a node is the smallest value in its right subtree (i.e. the next largest value in the tree). If the node doesn't have a right subtree but does have a successor, then its successor is the lowest ancestor of that node whose left child is also an ancestor of that node.

When nodes are deleted, they are not physically removed from the tree. They are just marked as deleted. Leaf nodes are just removed. A node with a single child has its parent point to its child. If a node has two child nodes, replace it with its successor.

By keeping a BST balanced (i.e. each left subtree contains roughly the same amount of nodes as each right subtree), complexity of operations is reduced from $\mathcal{O}(n)$ to $\mathcal{O}(\lg n)$.

AVL trees are balanced BSTs where every node requires the heights of left and right children to differ at most ± 1. Thus height(n) = max(height(n.left), height(n.right))+1.

Chapter 11

Red-Black Trees

A balanced binary search tree, so height $\mathcal{O}(\lg n)$ is guaranteed.

It is a binary search tree with an added bit per node, to denote its color which is either red or black. All other attributes of BST are inherited.

All empty trees are colored black. We use a single sentinel `nil` for all leaves of red-black tree T, with $color[nil]$ = black. The root's parent is also $nil[T]$.

RB trees have 5 properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.
5. For every node, all paths from the node to descendant leaves contain same number of black nodes.

Height of a node is the number of edges in a longest path to a leaf. Black-height of a node x, $bh(x)$ is the number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x itself. Black-height of a red-black tree is the black-height of its root. $bh(x) \leq h(x) \leq 2bh(x)$.

Lemma

Let x be a node in a RB tree. The length $h(x)$ of a longest path from x to a descendant leaf node is at most twice the length $s(x)$ of a shortest descending path from x to a leaf:

$$h(x) \leq 2bh(x) \leq 2s(x)$$

Insertion and deletion in a RB tree are not straightforward. There are two new operations: Left-Rotate(T,x) and Right-Rotate(T,y). Left rotation on x makes x the left child of y, and the left subtree of y into the right subtree of x as shown in Figure 11.1.

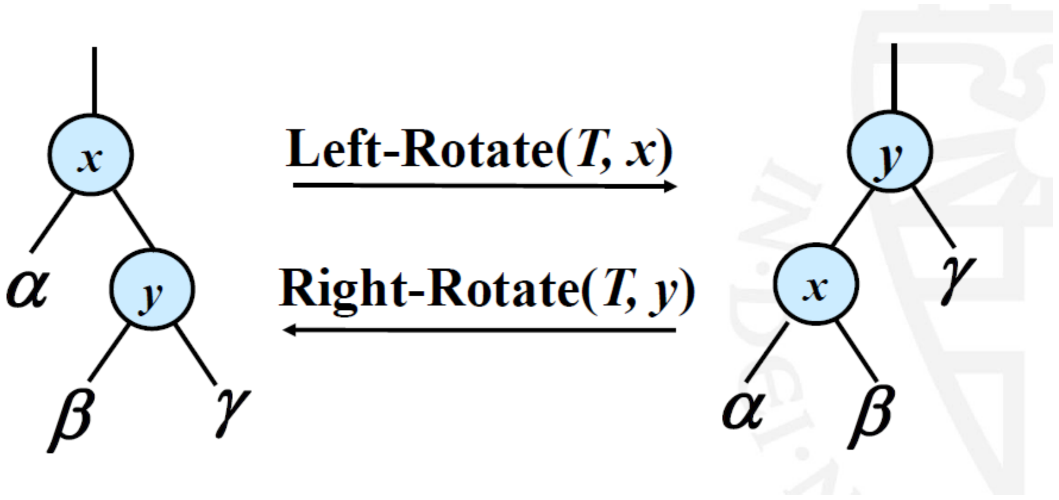


Figure 11.1: Left and Right rotate in a RB tree.

11.1 Insertion

To insert new nodes, we use the insert from BST to insert a node x into T. Color that node red and then fix the tree by recoloring nodes and performing rotation to preserve RB tree property.

There are some cases to solve when inserting a new node z. We do this iteratively and check for each case every loop. After each iteration we move z up two spaces.

1. z is root: $color[z]$ = black.
2. z's uncle y is red:
   - $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are not other violations of property 4.
   - Make $p[z]$ and y black. Now z and $p[z]$ are both not red. Property 5 might be violated.
   - Make $p[p[z]]$ red, which restores property 5.
   - Next iteration has $p[p[z]]$ as new z.
3. z's uncle y is black and z is a right child:
   - Left rotate around $p[z]$, such that $p[z]$ and z switch roles. Now z is a left child and both z and $p[z]$ are red.
   - Takes us to the last case.
4. z's uncle y is black, z is a left child:
   - Make $p[z]$ black and $p[p[z]]$ red.
   - Right rotate on $p[p[z]]$. This ensures property 4 is maintained.
   - No longer have 2 reds in a row.
   - $p[z]$ is now black, so no more iterations.

Because of all these fixups, instertion takes $\mathcal{O}(\lg n)$ time.

11.2 Deletion

Deletion works similarly. We delete using BST deletion and then iteratively fix the tree to maintain RB properties. If the to be deleted node y has original color black, violations can occur.

We can correct these by giving one of y's child nodes, x, an imaginary 'extra black'. So if $color[x] = black$ it now becomes *doubly black* and if $color[x] = red$ it now becomes *red & black*. The idea is to move this 'extra black' up the tree with rotations and recolourings until either, x points to a red & black node in which case we just color it black, or x points to the root, in which case we discard the 'extra black'.

Let's call x's sibling w, then we have the following cases:

1. w is red:
    - w must have black children.
    - Make w black and $p[x]$ red.
    - Left rotate on $p[x]$.
    - New sibling of x was a child of w before rotation, so it must be black.
    - Leads into one of the next cases.
2. w is black, both w's children are black:
    - Take 1 black off x (singly black) and off w (red).
    - Move that black to $p[x]$.
    - Next iteration uses $p[x]$ as x.
    - If entered this case from case 1, then $p[x]$ was red. So new x is red & black, color of the new x is therefore red and we terminate the loop.
    - New x gets colored black.
3. w is black, w's left child is red, w's right child is black:
    - Make w red and w's left child black.
    - Right rotate on w.
    - New sibling w of x is black with a red right child, so we go to the last case.
4. w is black, w's right child is red:
    - Make w be $p[x]$'s color c.
    - Make $p[x]$ black and w's right child black.
    - Left rotate on $p[x]$.
    - The 'extra black' on x is now removed and no other properties are violated, so we are done.

Like insertion, deletion can be done in $\mathcal{O}(\log n)$ time.

Chapter 12

Hashing

Say we have a set of keys that we want to store in a table. We can define a symbol table S, which can holds n records. We want to perform operations on S:

- `Insert(S,x)`
- `Delete(S,x)`
- `Search(S,k)`

How can we organize the data structure S, such that these operations have low runtime complexities?

We can use a hash function h to map the universe U of all keys $S = \{k_1, k_2, ..., k_n\}$ into $\{0, 1, ..., m-1\}$, where $m = |T|$, the size of the table. However, if for some $k_i, k_j \in S \rightarrow h(k_1) = h(k_2)$ a collision occurs. We can resolve these collisions in two ways.

12.1 Choosing a Hash Function

12.1.1 Division Method

Assume all keys are integers, then $h(k) = k \mod m$. Don't pick m with a small divisor d. A large number of keys that are congruent modulo d can adversely affect uniformity. In the extreme case that $m = 2^r$, then the hash doesn't even depend on all the bits of k. Don't choose m too close to a power of 2 or 10, or otherwise often used numbers. This method has its flaws, but it is simple.

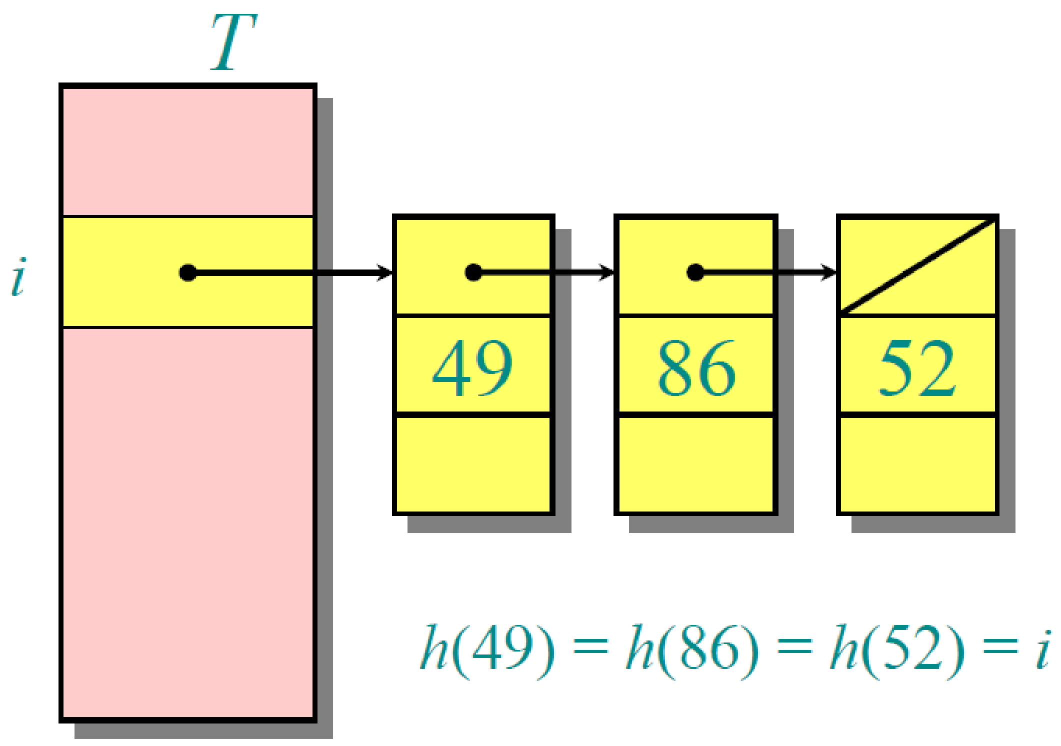
12.1.2 Multiplication Method

$$h(k) = (A \cdot k \mod 2^w) \text{ rsh } (w - r)$$

12.2 Resolving Collisions

12.2.1 Chaining

Resolving collisions by chaining simply means that we link records in the same slot into a list. So for instance if $h(k_1) = i$ we insert it into slot i and add k_1 to the chaining list of slot i. If for some other k_2, $h(k_2) = i$, we insert it into slot i and add k_2 to i's chaining list:

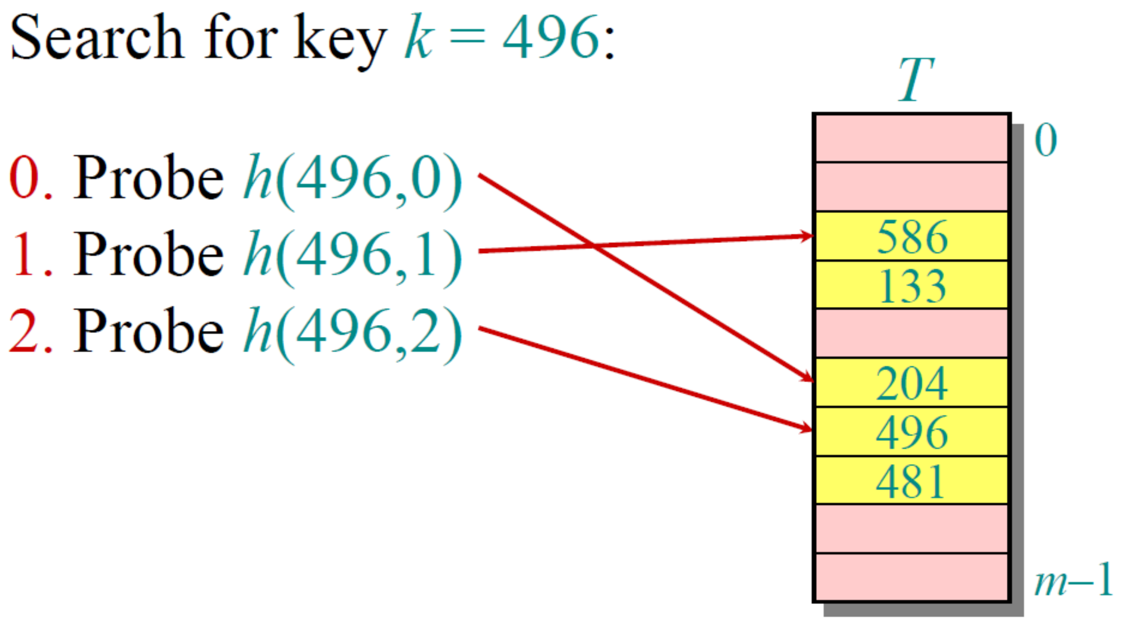


Worst case every key hashes to the same slot, in which case access time $= \Theta(n)$ if $|S| = n$.

If we make the case of simple uniform hashing, i.e. each key $k \in S$ is equally likely to be hashed to any slot of table T, then we can define a load factor for T. Let n be the number of keys in the table m the number of slots. The load factor α is then given by $\alpha = n/m =$ average number of keys per slot. The expected time for a(n) (un)successful search for a record with a given key is then $\Theta(1\alpha)$. Expected search time $= \Theta(1)$ if $\alpha = \mathcal{O}(1)$, or equivalently, if $n = \mathcal{O}(m)$.

12.2.2 Open Addressing

If we find a collision, we simply probe for another open slot in T:



We can use different probing strategies to find an open slot. If we make the assumption of uniform hashing, then each key is equally likely to have any one of the $m!$ permutations as its probing sequence. If the load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

This leads to the implications:

- If α is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, the expected number of probes is $1/(1-0.5) = 2$.
- If the table is 90% full, the expected number of probes is $1/(1-0.9) = 10$.

12.3 Probing Strategies

12.3.1 Linear Probing

Given hash function $h'(k)$, linear probing uses:

$$h(k,i) = (h'(k) + i) \mod m$$

Each time a collision is found, $i \leftarrow i + 1$. Suffers from primary clustering, where long runs of occupied slots build up over time, increasing average search time.

12.3.2 Double Hashing

Given two hash functions $h_1(k)$ and $h_2(k)$:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$$

Generally produces excellent results, but $h_2(k)$ must be co-prime with m. One way to do so is to make m a power of 2 and design $h_2(k)$ to only produce odd numbers.

Chapter 13

Complexities

If not explicitly stated otherwise, these complexities refer to the run-time complexity.

13.1 Data Structures

13.1.1 Adjacency Lists

- Space: $\mathcal{O}(V + E)$
- Determine if $(u, v) \in E$: $\mathcal{O}(degree(u))$
  $\mathcal{O}(V)$ in the worst case.

13.1.2 Adjacency Matrix

- Space: $\mathcal{O}(V^2)$
- Determine if $(u, v) \in E$: $\mathcal{O}(1)$
- List all neighbors of $u \in V$: $\mathcal{O}(V)$

13.1.3 Heaps

- Space: $\mathcal{O}(n)$
- $\mathcal{O}(\log n)$ for basic operations.

13.1.4 Binary Search Trees

- Space: $\mathcal{O}(n)$
- $\mathcal{O}(n)$ for basic operations (worst case).
  $\mathcal{O}(\log n)$ for basic operations (average & best case).

13.1.5 AVL Trees

- Space: $\mathcal{O}(n)$
- $\mathcal{O}(\log n)$ for basic operations.

13.1.6 Red-Black Trees

- Space: $\mathcal{O}(n)$
- $\mathcal{O}(\log n)$ for basic operations.

13.1.7 Hash Tables

- Space: $\mathcal{O}(n)$
- $\Theta(1 + \alpha)$ for search, if collisions are resolved by chaining.
- $\mathcal{O}(1/(1 - \alpha))$ for search, if $\alpha = n/m < 1$ and collisions are resolved by open addressing.

13.2 Sorting Algorithms

13.2.1 Heapsort

$\mathcal{O}(n \log n)$

13.2.2 Merge Sort

$\mathcal{O}(n \log n)$

13.3 Basic Graph Algorithms

13.3.1 BFS (Breadth-First Search)

$\mathcal{O}(V + E)$

13.3.2 DFS (Depth-First Search)

$\mathcal{O}(V + E)$

13.4 Shortest Path Algorithms

13.4.1 Dijkstra

$\mathcal{O}(E \log V)$ if priority queue (binary heap) data structure is used.

13.4.2 Bellman-Ford

$\mathcal{O}(V \cdot E)$

13.4.3 Floyd-Warshall

$\mathcal{O}(n^3)$

13.5 Network Flow Algorithms

13.5.1 Ford-Fulkerson

$\mathcal{O}(m \cdot n \cdot C)$
$\mathcal{O}(E \cdot f)$ if graph has integer capacities.
Solves bipartite matching in $\mathcal{O}(m \cdot n)$

13.5.2 Edmonds-Karp

$\mathcal{O}(V \cdot E^2)$

13.5.3 Hopcroft-Karp

Solves bipartite matching in $\mathcal{O}(m\sqrt{n})$

13.6 Divide & Conquer Algorithms

13.6.1 Karatsuba Multiplication

$\mathcal{O}(n^{\log 3}) \approx \mathcal{O}(n^{1.585})$

13.7 Greedy Algorithms

13.7.1 Interval Scheduling

$\mathcal{O}(n \log n)$

13.7.2 Minimum Spanning Trees

- Prim: $\mathcal{O}(m \log n)$
- Kruskal: $\mathcal{O}(m \log m)$
- Borůvka: $\mathcal{O}(m \log n)$
  $\mathcal{O}(n)$ if graph is planar.

13.8 Dynamic Programming Algorithms

13.8.1 Matrix-Chain Product

- Space: $\mathcal{O}(n^2)$
- Time: $\mathcal{O}(n^3)$

13.8.2 Weighted Interval Scheduling

- Space: $\mathcal{O}(n)$
- Time: $\mathcal{O}(n \log n)$

13.8.3 Text Justification

- Space: $\mathcal{O}(n)$
- Time: $\mathcal{O}(n^2)$

13.8.4 Longest Common Subsequence

- Space: $\mathcal{O}(mn)$
- Time: $\mathcal{O}(mn)$