Class Notes
CIS 502 Analysis of Algorithm
4-Greedy Algorithm

Da Kuang

University of Pennsylvania

Contents

1 Optimization Problem

Definition 1.1 ***Optimization Problem:*** *Minimize or maximize some function subject to some constrains.*

Now we start with a special class of optimization problem:

- Given a set of elements, pick a subset.
- Constrains tell you which subsets are allowed.
- Any allowed subset is a feasible solution.
- Goal is to find a feasible solution with greatest/ least value.

1.1 Minimum Spanning Tree Problem

Minimum spanning tree problem is an example of optimization problem.

Input: Connected, undirected graph $G \in (V, E)$ together with a weight function $w : E \to \mathbb{R}^+$.

Definition 1.2 *The **feasible solution** is a set of edges forming an acyclic connected graph on all vertices.*

Definition 1.3 *The **cost** of a solution is the sum of the weights of the edges in the solution.*

The are problems for which the optimal solution can be pick by choosing one element at a time.

Definition 1.4 *The **greedy algorithm** builds up solution as by taking the next element to be one of the optimal cost value that can be added feasibly.*

Most of the time Greedy algorithm itself is simple but it is difficult to prove correctness.

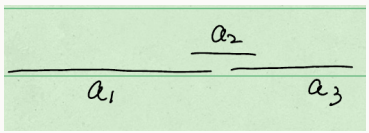
2 Activity Selection Problem

- **Input:** n activities $a_1, \cdots, a_n$, where a_i starts at time x_i and ends at time y_i.
- **Feasible Solution:** Any subset of these activities such that no two activities in the subset overlap.
- **Objective Function:** Maximize the number of activities we schedule.

2.1 Some Attempts

Criterions to be greedy on:

- Pick the activities with shortest duration.
  It dose not work. The counter example is as follows:



- Pick the activities what finish first.
  Sort the activities by finish time and then renumber them so that $f_1 \leq f_2 \cdots \leq f_n$

2.2 Proposed Greedy Algorithm

Given a set of activities,

- Pick the earliest finishing activities that remain.
- Remove all activities that conflict with the chosen activity.
- Repeat.

To prove the correctness, we start by arguing that the first choice algorithm is not wrong.

Claim: **Greedy Choice Property:** Fist choice made by greedy algorithm is not wrong. To be more specific, in activities selection problem, if greedy algorithm

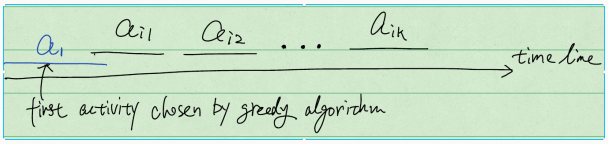
choose an activity at first, then there is an optimal feasible solution that contain a_1.

Proof: Suppose for contradiction that no optimal feasible solution uses a_1. Let O be the subset of activities in some optimal solution. We can order the activities in O by finish time.

Let $a_{i1}, a_{i2}, \cdots, a_{ik}$ be the activities in O so ordered. We can make an **exchange argument** as the following plot. Throw out a_i from O and include a_1 in instead to get a new set of activities O'.

There are some properties about O'.

- $|O'| = |O|$
- O' is feasible.
- O' is also optimal and contains a_1. $\Longrightarrow\!\Longleftarrow$



■

There is a way to construct an optimal solution starting with a_1. This optimal solution should certainly exclude activities that conflict with a_1. Recursively need to solve a smaller problem consisting of activities that do not conflict with a_1. In particular, the smaller problem is finding the optimal subset of activities out of the remaining activities.

Claim: **Optimal Substructure Property** In the set of activities, we need to pick an optimal feasible subset activities.

Proof:

- A: Original set of activities
- A': Set of activities that remain after throwing out a_1 and its conflicting activities.

Any solution to A' that gives value k can be extended to a solution to A of value $(k+1)$ by adding a_1. So need optimal solution to A' ■

In general. Optimal Substructure Property inductively assumes that greedy solves problem with fewer than n activities optimally.

Greedy Choice Property and Optimal Substructure Property imply that greedy solve n-activity problem optimally.

2.2.1 Time

We sort the activities by finish time take $O(n \log n)$. Then the rest of steps can be done in linear or constant time.

3 Linear Algebra

Definition 3.1 *V is a **vector space over** $\mathbb{R}$ if*

- *For $v_1,\ v_2 \in V$, $v_1 + v_2 \in V$.*
- *For any $\alpha \in \mathbb{R}$, $v \in V$, $\alpha v \in V$.*

Definition 3.2 *Given a finite set of vectors, $v_1, v_2, \cdots, v_k$, then **span** S is as follows*

$$S(v_1, v_2, \cdots, v_k) = \{v : v = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_k v_k,\ \alpha_i \in \mathbb{R}\}$$

$S(v_1, v_2, \cdots, v_k)$ is a vector space.

Definition 3.3 *$\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_k v_k$ is a **linear combination** of vectors.*

Definition 3.4 *The set of vector $v_1, \cdots, v_k$ is called **linear dependent** if there exist some coefficient $\alpha_1, \cdots, \alpha_k$ not all 0, so that $\sum \alpha_i v_i = 0$*

Definition 3.5 *A set of vector is said to be **linearly independent** if it is not linearly dependent.*

Definition 3.6 *If $v_1, \cdots, v_k$ are linearly independent and their span in V, then $v_1, \cdots, v_k$ form a **basis** of V.*

Definition 3.7 *If $v_1, \cdots, v_k$ is a basis for v and $u_1, \cdots, u_m$ is another basis. Then $m = k$.*

Proof. Suppose for contradiction that $m > k$. Since v_i's from a basis,

$$\begin{aligned} u_1 &= \alpha_{11}v_1 + \cdots + \alpha_{1k}v_k \\ u_2 &= \alpha_{21}v_1 + \cdots + \alpha_{2k}v_k \\ &\vdots \\ u_m &= \alpha_{m1}v_1 + \cdots + \alpha_{mk}v_k \end{aligned}$$

Will prove $(u_1, \cdots, u_m)$ is linearly dependent. Need to show $\sum x_i u_i = 0$ for some $(x_1, \cdots, x_m)$ not all zero.

$$x_1u_1 + x_2u_2 + \cdots + x_mu_m = 0$$

substitue u_i with v_i's,

$$\begin{aligned} &(x_1\alpha_{11} + x_2\alpha_{21} + \cdots + x_m\alpha_{m1})v_1 \\ &+ \cdots \\ &+ (x_1\alpha_{1k} + x_2\alpha_{2k} + \cdots + x_m\alpha_{mk})v_k = 0 \end{aligned}$$

All the coefficients above should be 0. There are k coeficientes, m equations and we assume $m > k$. Therefore, there are infinite possible combinations of x_i's. Therefore, there must be a solution which is not all zeros. Because if there is not such solution, then there should be only one solution which is all zeros. ■

4 Maximum Total Weight Problem

Suppose you are Google and you assign a vector to each possible document. When a user searches for a term, you want to collect some vectors that are indicative of that search term. You also want to be diverse, e.g., if a user

searches for "jaguar" you want to return some documents with cars and others with the animal. We could model this by returning vectors that are linearly independent.

- **Inputs:** vectors $(v_1, v_2, \cdots, v_n)$ with weights $(w_1, w_2, \cdots, w_n)$.
- **Goal:** Find a basis for the space spanned by $(v_1, v_2, \cdots, v_n)$ of maximum total weights.

Note. A single vector is linear independent to any other vector if it is a zero vector.

4.1 Greedy Algorithm

- Sort vectors and renaming them by descending order of weights.
- S is an empty set of vectors initially.
- For each vectors v_i in this order, if $S \cup \{v_i\}$ is linearly independent and v_i is an non-zero vector, then $S = S \cup \{v_i\}$.

4.2 Correctness

Theorem 4.1 ***(Exchange Property)*** *Suppose A and B are finite subset of linearly independent vectors in $\mathbb{R}^d$, and suppose $|A| < |B|$. $\exists b \in B$, such that $A \cup \{b\}$ in linearly independent.*

Proof. Indeed, there must be a vector in B that lies outside the space spanned by A because of the dimensions of space $< B >$ and $< A >$. ■

Theorem 4.2 ***(Hereditary property)***. *If $R \subset S$ and S is a linearly independent set of vectors, then R is also linearly independent.*

Greedy algorithm for finding a maximum of basis works because of vector has the exchange and hereditary properties.

- Suppose greedy returns vectors $G = \{v_{i1}, v_{i2}, \cdots, v_{ik}\}$.

- Suppose, for contradiction, there is an optimal solution that return $O = \{v_{j1}, v_{j2}, \cdots, v_{jk}\}$, which has a better total of weights.

Claim: G is at least as good as O.

Proof:

Let v_{il} be the first vector chosen by greedy algorithm that is not in O. Let $A = \{v_{i1}, \cdots, v_{il}\}$, $B = O$.

Hence,

- If $l = k$, G is at least as good as O.
- If $l < k$, $|A| < |B| = k$. By exchange property, applied repeatedly, we can keep adding vectors from B to A while preserving linear independent to A until $|A| = |B|$. In the end, after borrowing some elements from B, the only difference between A and B is that A has v_{il} instead of some vectors from B. Moreover, that vector has weight $\leq w_{il}$. So $w(A) \geq w(O)$.
  Note that even if $w(A) = w(O)$, A agrees still more with greedy algorithm than O. Suppose in the "worest case", no matter how to choose l, the weight of expanded A always equal to the weight of B. We are still able to prove that greedy algorithm is as good as O by induction. So the claim still holds.

■

5 Matroid

Definition 5.1 *A matroid is a pair $(S, \Im)$ where S is a finite set and $\Im$ is some set of subsets of S, designated "independent sets", such that*

- *$\Im$ satisfies the hereditary property. More formally, if $A \subset B$ and $B \in \Im$, then $A \in \Im$.*
- *$\Im$ satisfies the exchange property. More formally, if $A, B \in \Im$ and $|A| < |B|$, then there exists some $x \in B - A$ such that $A \cup \{x\} \in \Im$.*

All **maximal independent sets** in a matroid have the same size.

Note that it can be proved by contradiction with exchange property. If there are two maximal independent sets with different size, then the smaller one is able to use exchange property to take an element from the larger maximal independent set. But the smaller set should not be able to increase its size any more, So we get a contradiction.

A greedy algorithm is optimal for finding a maximum weight maximum independent set in a matroid.

Why there are maximum twice? Because we allow weights to be negative. Therefore, we have the second the maximum. If the weights are all positive, the the second maximum becomes redundant.

Remark: Matroid helps the greedy algorithm not to get into a dead-end or trace back and revise the decision we have made. It is because the the exchange property make sure that one can always go ahead and complete it with a good solution.

5.1 Graphic Matroid

There are many kinds of matroid but graphic matroid is one of the most famous matroid.

Definition 5.2 *Let $M = (S, \Im)$ be a matroid. Given a weighted, connected, undirected graph $G = (V, E)$ where $S = E$,*

- *If $A \subset E$ then $A \in \Im$ if and only if A is acyclic. That is, a set of edges A is **independent** if and only if the subgraph $G_A = (V, A)$ forms a forest.*
- *If $B \in \Im$ and $A \subset B$, then $A \in \Im$. So M is **hereditary**.*

Now let's proof the **exchange** property.

Lemma 5.1 *If $A, B \in \Im$ and $|A| < |B|$, then there exist $e \in B - A$ such that $A \cup \{e\} \in \Im$.*

Proof. B is a collection of edges with no circle in it. The graph has n vertices. Suppose there is no edge at the beginning. Then every time after adding an

edge to connect to vertices, the number of components reduce by 1 because to the new connection. So the graph on B has $n - |B|$ components, and the graph on A has $n - |A|$ components.

Therefore, $n - |A| > n - |B|$. B must contain at least one edge (u, v), where u and v lie in different connected components of the graph formed by A. This edge (u, v) can be added to A to get a larger independent set. ■

The graphic matroid is closely related to the minimum-spanning-tree problem which will be covered in the next section.

6 Find Maximum Weight Spanning Tree

6.1 Maximum Weight Minimum Spanning Tree

Input: a connected, weighted, undirected graph $G = (V, E)$ with weight function $w : E \to R$.

Goal: Find a maximum weight spanning tree in G.

Because this is the problem of finding a maximum weight of maximum independent set in a graphic matroid, the greedy algorithm is optimal.

6.2 Kruskal's Algorithm

Main Idea: Sort the elements in the ground set (edges) in decreasing order of weight and then repeatedly add edge as long as adding new edge keeping the graph acyclic.

Consider: Suppose Kruskal's algorithm has found the following edges, what makes the algorithm to decide to pick an edge?

Think in a shallow way. The new edge shall not create a cycle in the graph. To be more specific, if the vertices from the new edge belongs to the same component then it will introduce a cycle since vertices with in the component are

already connected. Conversely, if the vertices come from different components then it cannot be a cycle.

So Kruskal's algorithm maintain the connected components it has discovered so far then includes edge (u, v) lie in different component before the inclusion.

Remark: So far we have focused on finding maximum spanning tree but it turns out that the greedy algorithm works for finding minimum spanning tree as well. The only difference is to sort the weights by descending order for minimum spanning tree. For minimum spanning tree, we can use the same greedy algorithms.

6.2.1 Algorithm

Kruskal's algorithm finds a minimum weight spanning tree in a connected, weighted, undirected graph.

- Sort and renumber the edges in increasing order of weight as $e_1, \ldots, e_n$.
- Maintain a graph G with the same vertices as the original graph to track the edges chosen already.
- When considering $e_i = (u, v)$, add e_i to the graph if and only if u and v lie in different components. To do this, we need two routines.
  - Find operation: return the component for a vertex u.
  - Union operation: merge two components together.

We define a UNION_FIND data structure for find and union operation. Initially no edge is chosen by the algorithm and every vertex is a component by itself. Therefore, we will initialize the data structure to n singleton components.

Now we are going to explore some of the possible data structure to be used for UNION_FIND.

Attempt 0. Maintain the information in array. Make index of array is the vertex name and value is the name of the component.

For this structure, find operation is $O(1)$. Union(u, v) = $O(n)$, since in

worst case you may have to rename all the vertices. In the course of Kruskal's algorithm, $n - 1$ union operations to perform. So this is a rather expensive design choice.

Attemp 1. Suppose u's set contains n_1 elements and v's set contains n_2 elements.

When Union-ing two sets, change the name for the elements in the **smaller** sets. But in the worst case, the union still takes n steps. Moreover, even in a good case, there is only one element in a set but we still need to scan the whole list to know the size of set. Therefore, we introduce a new array to maintain a linked list for each set containing the elements in that set.

Remark: The array is used to from vertex to set and set is used to find the vertices of each set.

Even with the help of new linked list, an individual union operation could still take $\Theta(n)$ steps. However, we can use **amortized analysis** to make the our analysis more tight by looking at the whole sequence instead of single operation. In fact, any **sequence** of k union operations performed by Kruskal's algorithm take only $O(k \log n)$ steps.

Amortized Analysis. In computer science, amortized analysis is a method for analyzing a given algorithm's complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time per operation, rather than per algorithm, can be too pessimistic. The target of amortized analysis is rather special since it is neither average nor worst case.

For example, suppose given two sets A and B, the goal is to calculate the union of the sets. It is known that the cost of operation is $|B| = n$.

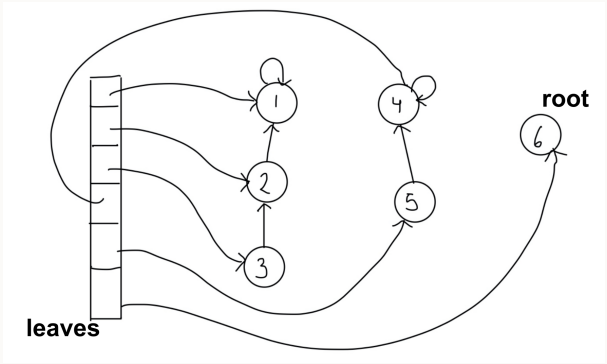
Accounting method is used for analysis:

- Set up an account for each element in the data structure.

- Charge 1 to each element in the smaller set for a union operation.
- Every time an element is charged. It joins a set at least twice as large.
  →No element is charged more than $\log n$ times.

Therefore, the total cost of a sequence of $(n-1)$ union operations to union A and B is at most $n \log n$. (With some most carefully analysis, k union cost at most $k \log k$. But $n \log n$ is good enough for the course.)

Attemp 2. Maintain connected components in a tree so that we have a forest as follows. The name of the component is the element at the root. The name of each vertices are saved in the array of leaves. For each leaf in the tree, keep a pointer to its parent.



FIND. Traverse up the tree by the pointer until you find the end of linked list, which is a root and you will have found the component of the vertex. Hence, FIND cost a proportion of the height of the tree.

UNION. UNION make one root point to the other.

How to find the time complexity of each operation? We can do the same optimization trick like in Amortized Analysis. We make the root of the smaller tree point to the root of the larger tree. Based on that, we claim that all tree heights at all times are $O(\log n)$.

Proof: Think about any element in any of the trees, when the cost for it passes through the root increase by 1? After the tree unions with other tree, in which

case the size of the sets increases at least by twice. Therefore, no element can be more than $O(\log n)$ away from the root. ■

Therefore, we have the confidence to claim that FIND operations are $O(\log n)$ in the worst-case. The UNION operations of arbitrary pairs of elements are also $O(\log n)$.

Recall that $|E| = m$ and $|V| = n$. Kruskal's algorithm does $O(m)$ find operations and $O(n)$ union operations. So overall, $O(m \log n)$ steps for all union and find operations. Note that sorting edges at the beginning already takes $O(n \log n)$ steps. (? Here it should be $O(m \log m)$, but $O(m \log m) = O(n \log n)$?) So far UNION_FIND is not the sole bottleneck since it has the same time complexity with sorting.

But still, there is one more optimization for UNION operation even though it will not help to improve the overall performance of Kruskal's algorithm.

Path Compression. First, let's reflect on one of the features of the tree we just build.

Unlike with the common tree structures, here we start from any internal node pointing by the pointer and then go upward from the child to its parent.

In the common tree (binary search tree, heap tree), we have to limit the number of children because it will cause the traversal order problem and increase the complexity of the problem.

By controversial, the path in our bottom has deterministic direction. Moreover, like always, the height of the tree is the number of operations for solving problem. Therefore, we would keep the tree as much bushy and short as possible to reduce the number of operations.

Now motivated by making the tree bushy and short, we can do path compression as follows:

When doing FIND(u), for each node encountered on the path from leaf u to

the root, make it a child of the root.

Note that path compression could not improve the worst case time of either FIND or UNION. But it gives a great amortized complexity. Any sequence of m starting from the initial UNION-FIND data structure on elements takes at most $O(m \log^* n)$ time.

Definition 6.1 $\log^* n$ *is a really really slow function which almost like a constant. It is the number of times you need to take* log *to get n.*

For example. $\log^* 16 = 3$. $\log^*(2^{16} = 65536) = 4$. $\log^*(2^{65576} = 65536) = 4$

We did not cover the analysis of path compression during class.

6.3 Prim's Algorithm

Prim's algorithm is another greedy algorithm for finding a maximum weight spanning tree. In the canonical greedy algorithm, we keep adding edge one at a time but how to select the right edge in the graph? Here we introduce a concept **cut.**

Definition 6.2 *A* ***cut*** *in a graph* $G = (V, E)$ *is defined by non-trivial partition* $V_1 \cup V_2$ *of* V. *The cut* (V_1, V_2) *is a set of edges that have exactly one end point in* V_1.

Claim: For $e \in \text{cut}(S, V - S)$, if e is the heaviest edge in the set. Then it will not cause a cycle with other edges have been connected by greedy algorithm. Therefore e will be included in the maximum weighted independent set.

Proof: By the canonical algorithm of greedy algorithm, we consider edges in decreasing order of weights. By the time we consider the edge e, we have not consider any other edge cross the two sets of the cut yet since e has the largest weight. Therefore, e will not cause any cycle with the edges that the greedy algorithm has chosen. Because you need two edges cross the two components to get a cycle. ■

Prim's algorithm is one specialization of this idea that if you can find the heaviest edge in each cut then you can build a minimum spanning tree. Prim's algorithm consider a particular set of cuts.

Main Idea: Split the graph into two sets S and $V - S$. Some vertex s is arbitrarily chosen as the initialization of S. First consider the cuts consisting s. Find the heaviest edge (s, v). Bring v into set S. Repeat the steps for the next heaviest cutting edge.

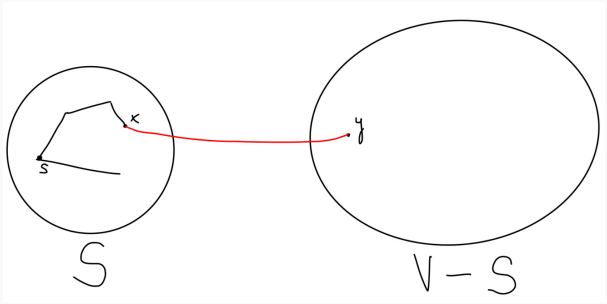
6.4 Algorithm

Initially,

$$d(v) = \begin{cases} w(s,v), \text{ if } (s,v) \in E \\ \infty, \text{ otherwise} \end{cases}$$

For each $v \in V - S$, maintain $d(v)$, weight $\Im$ and the lightest edge (x, y) from $V - S$ to any source vertex x in S.

At each iteration, when we move the vertex y that has the cheapest edge to a vertex in S, we need to update every neighbor v of y so that $d(v) = \min(d(v), w(v, y))$.



6.5 Time Complexity

If we maintain the $d(v)$s in an array A, the complexity is $O(n^2)$ because we need to first find the minimum in A, and once we've found the minimum we need to update all the neighbors of the minimum.

If we maintain the $d(v)$s in a min-heap, the complexity is $O(mlog(n))$. Usually $m \leq n$.

7 Shortest Path

Directed, weighted graph $G = (V, E)$with weight function $w : E \rightarrow \mathbb{R}$.

Note that the algorithm can work on undirected graph as well since undirected graph can be seen as a special directed graph. So a undirected graph problem can be reduced to a directed graph problem but not necessarily vice versa. Moreover, the world is full of directed graph (download/upload speed, railway direction). The directed graph is a more general problem.

There are two common sub-problem about the shortest path:

1. Single source shortest path: Find the shortest path from a given vertex s to all other vertex.
2. All-pair shortest path: $\forall u, v$, find length of the shortest path from $u \rightarrow v$.

In most of the time, other problem can be reduced to these two problems. For example, single destination shortest path problem can be seen as a single source shortest path problem on the reversed graph.

Notice that there is actually no particularly better algorithm to solve single-source-single-destination problem. To find the shortest path from s to t, the best we can do is to find the shortest path from s to all the rest of the vertices. (TODO: This was a worst case example?)

Definition 7.1 ***The length of a path*** *is the sum of weights of all the edges on the path.*

7.1 Single Source Shorest Path

Note: there is no matroid in the graph to solve the problem by greedy algorithm.

7.1.1 Dijkstra's Algorithm

- Input: Directed Graph $G = (V, E)$ and a source vertex s.
- Assumption: all edge weights are non-negative (for example, driving time of the edges, latency of the internet, cost of putting a pipe).
- Goal: Find the shortest path from s to all the vertices.

A path from s that stay eventually within S will be called a **source-side path**.

Main idea Maintain a quantity called $d(v)$ for each vertex v so that,

- $d(v)$ is an upper bound on the length of the shortest path from s to v.
- There is a path from s to v of length $d(v)$.

Based on the assumption, we know that the shortest path from s on the graph is to s itself. We maintain a source set S initially as $\{s\}$ and we know the shortest path form s to each $u \in S$. Grow S until $S = V$. Then we know the shortest path from s to all the vertices. At each state, find a vertex $v \in (V - S)$ for which there is a vertex u in S such that $(d(s, u) + w(u, v))$ is the minimum over all $x \in (V - S)$. Add v to S. Update as appropriate.

Algorithm

- For each vertex $v \in (V - S)$, $d(v)$ is the length of the shortest path from s to v which consist of a source side path to a some vertex $u \in (V - S)$ followed by edge (u, v).
- In each iteration, bring the vertex v with minimum $d(v)$ to S. To make sure d is invariant, we have to update d based on the new S (with a new vertex v). For each neighbor vertex x of v, update $d(x) = \min(d(x), d(v) + w(v, x))$. Note that if v is used to update x, then we remember v as the ancestor of x.

7.1.2 Correctness

Statement: For each vertex $v \in S$, $d(v)$ is equal to the length of the shortest path from s to v.

This statement proves the correctness of the algorithm, since when $S = V$, we have the shortest path from s to all the vertices.

Proof. Initialization: $d(s) = 0$, $d(v) = \infty$ for $\forall v \neq s$.

Prove the statement by induction.

- **Base case**: After the first iteration, $S = \{s\}$ and $d(s) = 0$, which is correct.
- **Inductive Hypothesis**: Assume the statement is true for $|S| = k$. Suppose v is the $(k+1)$-th vertex brought to S. v has $\min(d(v))$.
- **Inductive Step**: Let it call the path with length $d(v)$ as P. For contradiction, suppose there is a shortest path P' to v. P' must include at least one edge from S to $(V - S)$. Let (x, y) be the first such edge. Then the (s, y) portion of P' is already at least as long as P by the way algorithm works. We get a contradiction by assuming the existence of P'.
  This proves that at the point the algorithm brings v into S, $d(v)$ is the length of the shortest path to v. So we complete the induction.

■

Why negative weight does not work here?

Adding a negative edge into source set can change the length of shortest source-side path for some vertices. But we do not update $d(u)$ for $u \in S$ since we assume introducing more nodes on the path only increase the length of path.

7.2 Implementation

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

In initialization, $d(s) = 0$ and the d for the rest of nodes are $+\infty$. In relax function, if $d(v) > d(u) + w((u, v))$, set $d(v) = d(u) + w((u, v))$ and update the partent of v if necessary.

7.3 Time

Line 5 will execute $|V|$ times and line 8 will execute $|E|$ times. Therefore

$$T = |V|T_{\text{extract-min}} + |E|T_{\text{decrease-key}}$$

8 Huffman Coding

Given a text, which is a string over a large alphabet (say as English Text), we need to encode the text in binary for storage and communication.

Goal: Minimize the length of the binary code of the text.

Related problem: Code has to be a mapping from symbols in the original alphabet, Σ, to binary string.

8.1 Prefix-ambiguity

If one coded word is a prefix of another, then that could lead to ambiguity. Therefore, code must be "prefix free", which are called **prefix code**.

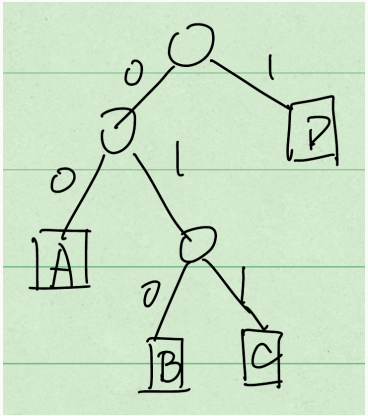
Ideally, we could like to assign shorter code to the words with higher frequency.

8.2 Algorithm

Input: $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_n\}$. f_i is the frequency of σ_i. Assume that $\sum f_i = 1$, $f_i \geq 0 \forall i$.

Goal: Find a prefix code that makes σ_i to $c(\sigma_i)$ to minimize $\sum f_i(\sigma_i)$, which is the average code length.

Binary code can be thought as a binary tree. As example:



Note that the symbols at the leaves are prefix code. The tree corresponding an optimal tree is a full binary tree, i.e. every internal node has two children. It is because if you have a internal node which is not full, then you are able to move everything below it one level up and reduce the code length.

Suppose we know the shape of the optimal tree which is a full binary tree.

Claim: Any full binary tree has at least a pair of sibling leaves at the deepest leaves.

Proof: Choose one leaf x at the deepest level. Its parent must have two children (say x and y). y cannot be internal node since x is the deepest leaf. Therefore, y must be the sibling leaf of x. ■

Claim: **Greedy Choice**: Pick the two symbols with least frequency and put them in the sibling leaves at the deepest level.

Proof: Prove the claim by exchange property. Suppose x and y are the two symbols of the least frequency. Suppose the sibling leaves at the deepest level have symbols a, b and $\{a, b\} \neq \{x, y\}$.

By exchanging the positions of a with x and b with y, we can bring x, y to the deepest level. One can prove that this is no worst than the original tree.

■

Make a new symbol xy in place of x and y so that $f(xy) = f(x) + f(y)$.

Suppose the length from root to x is $(d+1)$. So the contribution of code length that x and y make is

$$\begin{aligned} & f(x)(d+1)+f(y)(d+1) \\ =&(f(x)+f(y))(d+1) \\ =&f(xy)(d+1) \end{aligned}$$

So the same tree with xy made a symbol has cost that of $f(xy)$ lower.

8.3 Algorithm

Combine two least frequent symbol x and y with one symbol xy with $f(xy) = f(x) + f(y)$.

Recursively solve the problem on the $(n-1)$ symbols.