COMP 3711H Notes

Student HONG, Lanxuan
SID 21035307

1 Fundamental Sorting Problem

For general sorting problems, the algorithm takes a sequence of numbers as **input** and **output** a reordering of the numbers in certain, say increasing, order. We will first introduce several sorting problems and solutions, and then see how they can be viewed as a whole. Most sorting problems uses the important idea of **divide and conquer**, with the famous **master's theorem** to analyze its time complexity.

1.1 Insertion Sort

One of the most intuitive sorting algorithm is **insertion sort** (see Fig.1), where numbers are inserted from tails to the ordered head. Though its best case running time is $O(n)$, its worst case and average case running time is $O(n^2)$. By thinking intuitively, this algorithm is doing much unnecessary comparisons to local the exact order of the inserted number. This inspire us to find more efficient algorithms with better running time.

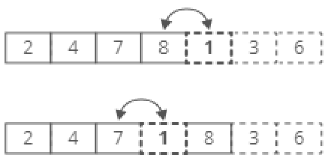


Figure 1: Insertion Sort

1.2 Merge Sort

Merge sort, as inferred in its name, is a sorting algorithm that merges 2 sub-sorted list into one sorted list. The core idea in merge sort is **divide and conquer** (which is a big topic to be covered in the whole section, we will then give a general idea of how this kind of problem can be solved.).

1. Idea: A list of numbers are **divided** into 2, and then sorted (**conquered**) respectively. In the merging (**combine**) part, we need to compare the 2 sorted list from the left-most element to right and decide which one is smaller. Continue this process until one sub-list is empty and then copy the rest of the other list to the main branch.

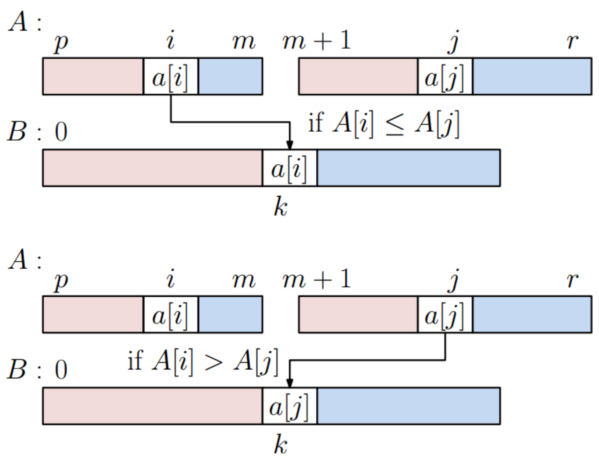


Figure 2: Merge Sort

2. Algorithm

```
MergeSort(A,p,r):
{
    if (p<r):
    {
        m = (p+r)/2
        MergeSort(A,p,m)
        MergeSort(A,m+1,r)
        merge(A,p,m,r)
    }
}
merge(A,p,m,r):
{
    if (A[p]>A[m+1]):
        merge(A,p+1,m,r)
    else:
        merge(A,p,m+2,r)
}
```

3. Runtime Analysis:

1.3 Quick Sort

Though being fast, merge sort has its own problem that it takes more space to merge the arrays. To save space in sorting, **quick sort** which finish sorting at its own place will be introduced and its idea of **randomization** is also important in designing other algorithms.

1. Idea: in the given array, randomly pick one number as the **pivot** (suppose every numbers are different). Part all the elements into 2 groups by comparison with the pivots (see Fig.3). Recursively do the same operations on each parted groups until all the groups are parted, and it automatically did the sorting.

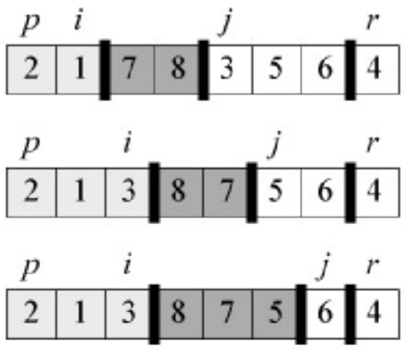


Figure 3: Partition

2. Algorithm:

```
QuickSort(A,p,r):
    q = Partition(&A,p,r);
    QuickSort(A,p,q-1);
    QuickSort(A,q+1,r);
Partition(&A,p,r): // assume the last digit is selected as pivot
    x = A[r];
    i = p - 1
    for j in (p,r):
        if A[j] > x:
            j++
        else:
            i++
            swap A[i] and A[j]
    swap A[r] and A[i+1]
    return i+1
```

3. Expected runtime analysis:

1.4 Heap Sort

Building data structures to help algorithm implementation is a common way in algorithm design. In terms of sorting problems, **heap sort** which use heap to organize elements is also a good sorting solution. Heap sort can be done in an array, with children's position at $2i$ and $2i+1$ while parents' position at $\lfloor i/2 \rfloor$. For a max heap, the parent must be greater than the children, with the largest value on the top. The height of the heap is $\Theta(logn)$.

The heapify process to maintain the heap is shown as follows, as the large value on the bottom will bubble up to the top.

```
Max-heapify(A,i):
    l <- left(i), r<- right(i);
    if A[l] > A[i]:
        largest = l;
    else:
        largest = i;
    if A[r] > A[largest]:
        largest <- r;
    exchange A[i] with A[largest]
    Max-heapify(A,largest)
```

We use the same heapify on every node (to bubble down to a right position) to build the heap. The running time to build a heal has a loose upper bound $\Theta(nlogn)$, with every node going through at most the height of the heap. It's tight upper bound, however, is $\Theta(n)$. We can derive this by considering the following: the n/2 nodes at last level, will not bubble down. Their parents, with a large amount n/4, will at most bubble down one layer.

$$\frac{n}{4} \times 1 + \frac{n}{8} \times 2 + ... + 1 \times logn = \Theta(n)$$

The idea of heapsort, is to pick the top value of the heap and heapify again. For each heapify, it takes at most logn times to bubble up, so for each node, the running time is $\Theta(nlogn)$

1.5 A Summary of Sorting

The sorting mentioned so far are all **comparison sortings**, and we found them having an upper bound $\Theta(nlogn)$. Here we will show that this upper bound is valid for all comparison sortings, and then will provide some other methods of sortings not based on comparison and faster than $\Theta(nlogn)$.

Consider there are boxes with values inside and we want to know the sequence of them by comparison. The question becomes, how many comparisons are needed to place all the boxes in the right place, say, in increasing order. By permutation we know there are at most $n!$ outcomes to place them, and what comparison does is to produce a dependence of a_i and a_j and remove the other possibilities until only one possibility exist.

The solution here is to draw a binary decision tree with each node correspond to the relationship between a_i and a_j, and two branches showing different comparison results. The number of leave should be $n!$ to show all possible outcomes. A binary tree with $n!$ leaves has the least height $logn!$ (the best case is under balanced condition), which can be simplified into the form $\Theta(nlogn)$.

Comparison is **NOT** the only way to do sortings. In fact, there are more sorting methods that can be used. For example, when we try to sort names by **alphabet**. We will compare/differ Name[0] and put them into 26 buckets. Then we do the same thing to Name[1],...until the end of the Name.

1.6 D&C: Master's Theorem

The divide and conquer algorithm, which applies the idea of separate and recursion, is very useful in sorting, as shown in merge sort and other problems.Here we will provide a general case to analyze the time complexity, after which we will show how this solution can be applied.

Master's Theorem: For the recurrence:

$$T(n) = aT(\frac{n}{b}) + n^k$$

where $a \geq 1, b > 1$ are constants and positive integer n. The base case $T(1)$ is considered to be any constant.Then

- Case 1: if $a > b^k$, then $T(n) = \Theta(n^{log_b a})$
- Case 2: if $a = b^k$, then $T(n) = \Theta(n^k logn)$
- Case 3: if $a < b^k$, then $T(n) = \Theta(n^k)$

Draw Recursion Tree to derive it:

1.7 D&C: Long Integer Multiplication

Here we we see another problem solved by divide and conquer. In the **Long integer multiplication problem**, we want to consider the arithmetic performance on a certain (long) length of integers. For addition, subtraction and comparison on integers of length n, it takes $\Theta(n)$ times. Take addition as an example:

$$a = a_n \times 10^n + \ldots + a_1 \times 10 + a_0$$
$$b = b_n \times 10^n + \ldots + b_1 \times 10 + b_0$$
$$a + b = \sum_{i=0}^{n} (carry_{i-1} + a_i + b_i)\%10 \times 10^i$$

For the most intuitive multiplication algorithm, we must multiply each digit at the bottom to each digit on the top, resulting in $\Theta(n^2)$ times. Applying the idea of divide and conquer, we can get another way to multiply ab.(see Fig.4)

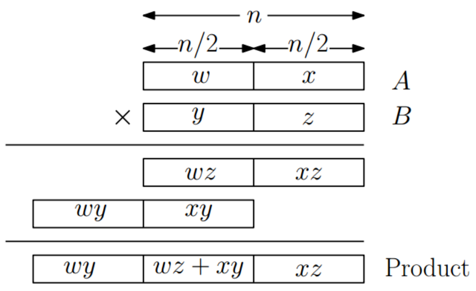


Figure 4: Long Integer Multiplication

Runtime Analysis:

$$T(n) \leq 4T(\frac{n}{2}) + cn = \Theta(n^2)$$

Nothing improved:)

Karastuba's idea: The reason why this divide improvement fail is because it still did 4 times of multiplication within the 4 parts. That means each digit is still multiplying every digit in the other integer. There is a smart way to avoid such multiplication and reduce the time of multiplication into 3 times.

$$AB = \overline{wx} \cdot \overline{yz} = xz + 10^n wy + 10^{\frac{n}{2}}((w + x)(y + z) - wx - yz)$$

Runtime Analysis:

1.8 D&C: Closest pair

The **closest pair** is another application of divide and conquer algorithm, which comes from the field of computational geometry. The problem is, given a set P of n points, and wish to find the closest pair of points $p, q \in P$. It is obvious that this problem can be solved within $\Theta(n^2)$ times by calculating and comparing all the distances, thus finding the smallest. The solution, however, can be controlled in $\Theta(nlogn)$ times, with a clever use of D&C.

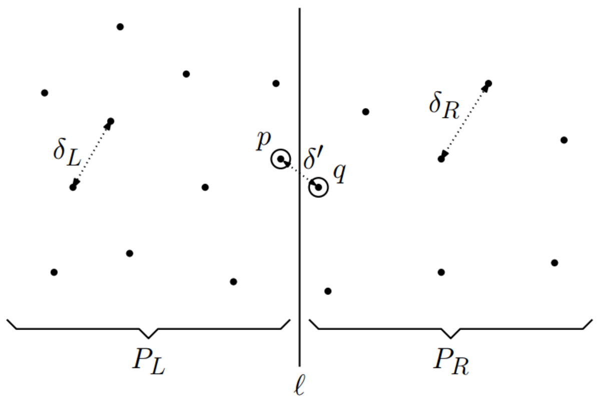


Figure 5: Closest Pair Solution

Like any D&C algorithm, this approach involves three basic elements:

- **Basis:** if $|P| < 3$, just solve the problem in constant time.
- **Divide:** partition the points into 2 sub-arrays P_L and P_R based on their x-coordinate. **Note that** here a pre-step should be done, which is to sort all the points by x-coordinate and y-coordinate once, which will spend $\Theta(nlogn)$ times. Then we will solve the problems in P_L and P_R respectively and recursively. Since the sortings were done, we only need $\Theta(n)$ times to find the medium.
- **Conquer:** compute the closest pair within each subsets P_L and P_R, then we will get the closest pair from each side which is $\delta = min(\delta_L, \delta_R)$.
- **Combine:** This is the most tricky part in this solution. We must note that this δ is not necessarily the final answer, because there may be 2 points that are very close to each other but at the border. That is, for p,q, their distance is closer than δ. Assuming that this can be done in $\Theta(n)$ times, the whole algorithm can be finished, just as merge sort, in $\Theta(nlogn)$ times.

Now, the thing to solve is, how do we finish the combine step within $\Theta(n)$ times. The trick is to use its upper bound δ. We first conclude that both p and q must lie within distance δ in terms of x-axis, from both side of the mid-line. We will get a set S denoting the points within that vertical strip of width 2δ. Now we use the sortings of y-coordinate to get $S_y =< s_1, ..s_m >$ and observe that if S_y contains 2 points that are within distance δ of each other, they must be within a constant number(which is **7**) of positions of each other in the sorted array s_y.

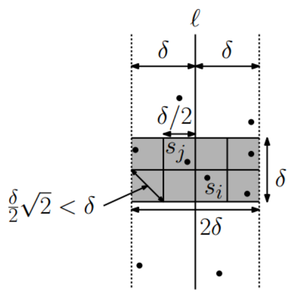


Figure 6: Closest Pair Between Sets

As shown in Fig. 6, this resided rectangle shows that in each small square there will be at most 1 point(otherwise σ is not the smallest distance for each side). That is to say, for each point, we only need to examine a constant number of points and thus can control the running time within $\Theta(n)$.

1.9 Medians and Selection

This section shows a tricky but surprising algorithm which shows the power of recurrence and D&C. Given a set of n numbers, find the median of these numbers with assumption that each numbers are identical, i.e. distinct. This problem can be generalized into the **selection problem**, which is to select the element of rank k(at position k when sorted) from a set of numbers. It is obvious that this problem can be solved in $\Theta(n \log n)$ by sorting and choosing, but since a rigorous sorting is not required, this problem can be solved within $\Theta(n)$.

The **Sieve Technique** applied in this algorithm design is a special case in D & C algorithm where the number of subproblem is just 1. The strategy is, after doing some analysis of data, a large amount of numbers (meaning a fixed constant fraction of n) can be eliminated from further consideration, and thus doing this elimination recursively to pick the right choice.

For this selection problem, we can randomly choose a pivot and then partition the array into numbers less than the pivot and numbers greater than the pivot(Notice that this partition step is similar to quick sort). Instead of sorting each parts, one part will be eliminated if the wanted rank is not included by comparing k with the position of pivot after partition.

If choosing pivot and partition can be done in $\Theta(n)$ times, it is obvious that the recursive elimination also takes $\Theta(n)$ times(recall the idea of "elimination" in heap sort). The partition was already discussed in quick sort. The only problem remain is **choosing the pivot** in $\Theta(n)$ times such that it separate the array into 2 part with same degree of amount.

In quick sort, we used the randomization process to show that a random pivot is good enough, with **expected running time** $\Theta(n)$. Here is another (complicated but intelligent) way.

1. partition A into $m = \lceil \frac{n}{5} \rceil$groups of 5 elements.
2. compute the median of each group, returning m medians.
3. compute the median of these medians recursively. The median of medians, x, is the desired pivot. It's obvious that x should be at least $\frac{n}{4}$ and at most $\frac{3n}{4}$.

Questions: groups of 3? groups of 4? Runtime Analysis?

2 Graph

2.1 Terminology

- **Graph:** A **Graph** $G = (V, E)$ is a structure that represents a discrete set V objects, which are **vertices** or **nodes**, and a set of pairwise relations E between these objects, which are **edges**. The term graph means an undirected graph while the term **digraph** means a directed graph, where the edges are directed from one node to another (ordered pairs).

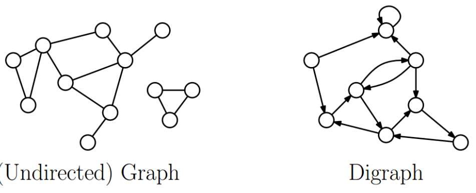
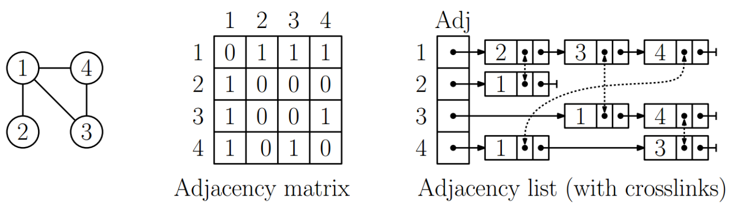


Figure 7: Graph and Digraph

- **Edges:** Given an directed edge $e = (u, v)$, u is the **origin** of the edge and v is the **destination**. For undirected edges eu, v, both u and v are **endpoints**. The edge e is **incident** both u and v, while u and v are **adjacent** because of edge e.
- **Degree:** The degree of a graph is the total amount of edges incident node u, denoted as $deg(u)$. For digraph, the **in/out-degree** is the total edges in/out of node u, denoted as $in/out - deg(u)$. A vertex u is **isolated** if $deg(u) = 0$. The total number of edges **m** should be $O(n^2)$ with the total number of degrees being 2m for graph, m for both in-degree and out-degree in digraph. In real practice, we will consider the size of the graph considering both nodes and vertices and being $O(m + n)$, where in a **sparse** graph $m = O(n)$ and otherwise **dense**.
- **Path and Cycle:** The **path** of a graph or digraph is an ordered sequence containing k vertices that goes through from one vertex to another where k is the **length** of the path. A path is **simple** of all the vertices it pass are distinct, otherwise it's a **cycle** which contain at least one node goes through twice. A graph, or digraph, is said to be **acyclic** if it contains no simple cycles (which are cycles with distinct nodes and edges). An **acyclic and connected** (connected means there is a path between any 2 vertices) graph is called a **(free) tree**. Different from the trees in recurrence and other data structures, this tree has no root, which means it can have a random source. An acyclic undirected graph can form a collection of trees, which becomes a **forest**. For digraphs, an acyclic digraph is called a directed acyclic graph, or **DAG** for short.
- **Representation:**
  1. One way is by **Adjacency Matrix**. For digraph $G = (V, E)$, the adjacency matrix of G is

     $$A[v, w] = \begin{cases} 1, & \text{if } (u, v) \in E \\ 0, & \text{otherwise} \end{cases}$$

     For undirected graph, the representation is similar without order, thus the matrix would be symmetric.
  2. **Adjacency List:** Use an array $A[1, 2, ...n]$ of pointers where each $A[v]$ stores a list containing the vertices that are adjacent to v to represent a graph. The advantage of Adjacency list is that it only requires $O(m + n)$ storage instead of $O(n^2)$ storage for adjacency matrix, which is convenient for Plano graphs.



	1	2	3	4
1	0	1	1	1
2	1	0	0	0
3	1	0	0	1
4	1	0	1	0

Adj			
1	2	3	4
2	1		
3		1	4
4	1		3

Figure 8: Adjacency Representation of Graph

2.2 Graph Traversal: BFS and DFS

There are a number of approaches used for solving problems on graphs. One of the most important approach is based on the notion of systematically visiting all the vertices and edges of a graph, which is called **graph traversals**. These traversals impose a type of tree structure on the graph, which will make other algorithms easier to implement.

2.2.1 Breadth-first Search

The strategy of BFS is shown as follows:

Given a graph $G = (V, E)$, BFS starts at some source vertex s and **discover** all its neighbors. When all the neighbors that were undiscovered are discovered, the vertex is **processed** completely. Initially all vertices except for the source are colored white (**undiscovered**). When a vertex has first been discovered, but not processed, it is colored gray and pop into a FIFO queue as part of the **frontier**. The elements in the frontier will be processed one by one and colored black after processing.

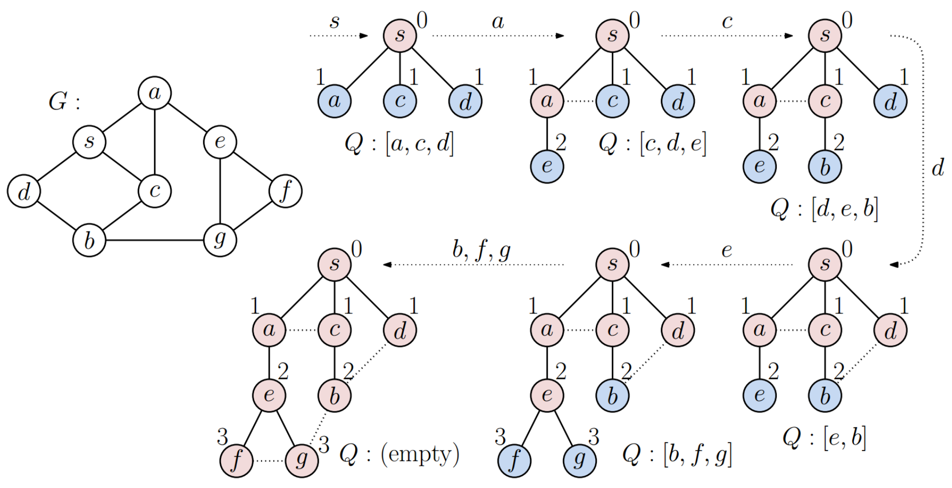


Figure 9: BFS

As shown in Fig.9, some edges will be discovered as part of the shortest path and shown in the tree, which are **tree edges**, while other edges are "missing" in the tree which are **cross edges**. The **predecessor** pointers in BFS forms the reversed rooted tree. For undirected graph, the cross edges always go between 2 nodes that are at most one level apart in the BFS tree, i.e. at the same level, predecessor level, or descendent level, otherwise it will be discovered first. As all the nodes are initialized in $\Theta(V)$ times and only explored once, with all the edges explored twice for undirected graph and once for directed graph, the running time $T(n) = \Theta(V + E)$.

```
BFS(G,s){
    for each u in V{
        mark[u] = undiscovered
        d[u] = infty
        pred[u] = null
    }
    mark[s] = discovered
    d[s] = 0
    Q.Enqueue(s)
    while(Q is nonEmpty){
        u = Q.dequeue
        for each v in Adj(u){
            if (mark[v] = undiscovered)
                mark[v] = discovered
                d[v] = d[u]+1
                pred[v] = u
                Q.Enqueue(v)
        }
        mark[u] = finished}
    }
```

2.2.2 Depth-First Search

The strategy of DFS is shown as follows:

Given a graph $G = (V, E)$, DFS starts at some source vertex and **discover** either one of its neighbors. When every reachable vertices from a vertex u are discovered (in fact, they should all be visited and finished), the vertex u is finished, or **processed**. After finishing a vertex u, follow the predecessor pointer to find the predecessor of u and continue the process.

```
\* DFSVisit for all undiscovered vertices in G *\
DFSVisit(u)
{
    mark[u] = discovered
    d[u] = ++time // the time that u was discovered
    for each (v in Adj(u)):
        if (mark[v] == undiscovered):
            pred[v] = u
            DFSVisit(v)
    mark[u] = finished/processed
    f[u] = ++time // the time that u was finished/processed
}
```

As shown in Fig.10, the whole DFS process is done by recursion and it naturally build a recursive tree. This hierarchical structure naturally impose a nesting structure on the **discovery-finish** time intervals, which is described by the **parenthesis lemma**. The idea of lemma is that the discovery time stamp of descendant vertices is always later than the discovery time stamp of ancestor vertices; while the finish time stamp off descendant vertices is always earlier. Otherwise the 2 vertices are unrelated with disjoint discovery and finishing time.

For undirected graphs, the non-tree edges are **back edges/forward edges**, for all these edges must incident 2 vertices on the same path. A sketch of proof is that if they are not on the same path, there must no overlap of discovery time, which contradicts with the parenthesis lemma for adjacent vertices. For directed graphs there are **back edges**(pointing to ancestor or self), **forward edges**(pointing to descendant) and **cross edges**(vertices not on the same path).

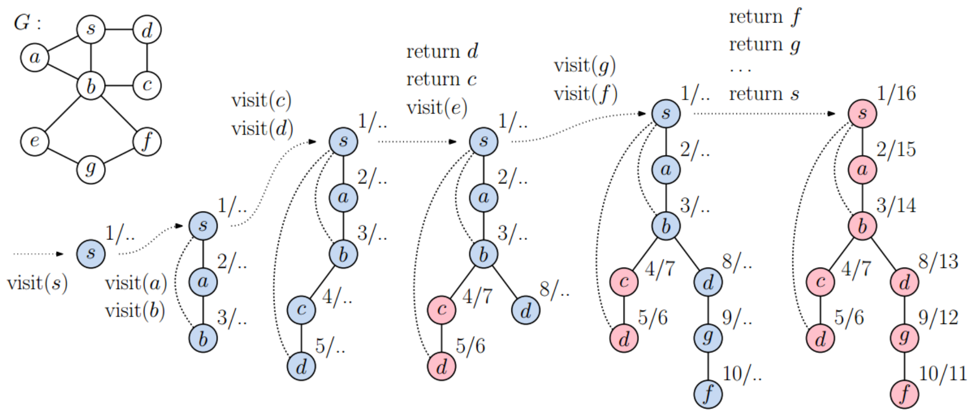


Figure 10: DFS

For the same reason as in BFS, the running time of DFS is $T(n) = \Theta(V + E)$.

One important application of DFS is to check the existence of **cycles**. By parenthesis lemma, a digraph is acyclic if its DFS tree has no back edge(check back edge by discovered time and finished time). For undirected acyclic graph, it is sufficient to show that its DFS tree has no forward edge or back edge.

Another application is to find **bridges** i.e., any edge whose removal results in a disconnected graph. It is trivial that a connected graph without bridge must have cycle. So some of the process is similar to acyclic detection.

Claim that: An edge (u, v) is a bridge iff it is a tree edge and there is no back edge within v's sub-tree that lead to a vertex whose discovery time is strictly smaller than v's discovery time.[1] To restrict their back edge, define $low(u)$ representing the lowest back edge level its sub-tree can achieve. The cut vertex have same level of distance and $low(u)$, meaning that it cannot be skipped by any edge.

[1]If it is not a tree edge, then it can be replaced by the tree edges and thus not bridge. Trace all the back edges within the sub-tree of v and make sure that once it go down, they will never turn back to levels higher than u.

2.3 Shortest Path: Dijkstra and Bellman-Ford

Consider the problem of computing shortest paths (**single source shortest path problem**) in a directed graph. The problem is presented as: given a digraph $G = (V, E)$ and a source vertex $X \in V$, and we want to compute the shortest path from s to every other vertex in G. The algorithm should be applicable to undirected graph as well, by simply assuming that each edge consists of 2 directed edges going in opposite direction. In BFS, we can find the shortest path of all vertices from the source vertex in $O(m + n)$. In that computation, however, we assume that each path has no weights, i,e., the weights are equal to unity. Generally, we assume that the graph is **weighted**, meaning that each edge $(u, v) \in E$ has a weight $w(u, v)$. The distance of the path is defined by the smallest sum of weights, denoted by $\delta(s, v)$.

2.3.1 Dijkstra

One of the earliest and most famous algorithm in computer science is the **Dijkstra Algorithm**. It was invented to solve weighted shortest path problem for **non-negative** weight. We will first present the strategy and then prove its correctness.

- **Strategy:** The basic structure of Dijkstra's algorithm is to maintain an estimate of the shortest path (the shortest path known by the algorithm so far) for each vertex, call this $d[v]$, by a **priority queue**. Initially all $d[v] = \infty$ and $d[s] = 0$. This path will be updated, or **relaxed**, when a path to v is shorter than $d[v]$. The strategy for Dijkstra's algorithm is to claim that for a subset of nodes $S \subseteq V$, $d[v \in S] = \delta(s, v)$. And by relaxing the closest unprocessed vertex(discovered but not in S) v by comparing $d[v]$ to $d[u \in S] + \omega(u, v)$. This infers that v is processed and can be added to S.

  Intuitively, this idea is correct because for any unprocessed vertex v with shortest path within the unprocessed vertices(put in the priority queue), it is impossible to find a shorter path from the unprocessed vertices. The only possibility for it the optimize distance is relaxing it by comparing with known shortest path.

- **Steps:**
  1. **Build and update** the priority queue for new discovered vertices.
  2. **Extract min** from the priority queue based on its key value, i.e. distance.
  3. **Decrease key** by computing the new shortest path based on the new information and reorganize the priority queue.

- **Code**

```
dijkstra(G,w,s){
    for each (u in V){
        d[u] = +infinity
        mark[u] = undiscovered
        pred[u] = null
    } d[s] = 0
    Q // priority queue sorted by d[u]
    while(Q is nonEmpty){
        u = ExtractMin(Q)
        for each (v in Adj[u]){ // relaxing step
            if (d[u]+w(u,v)<d[v])
                d[v] = d[u]+w(u,v)
                decrease v's key in Q to d[v]
                pred[v] = u
        }
    mark[v] = finished
    }
}
```

- **Runtime Analysis:**

$$\begin{aligned} T(n) &= \sum_{u \in V} T(ExtractMin) + deg\{u\}T(DecreaseKey) \\ &= \sum u \in V(\log n + deg\{u\} \log n) = \Theta((m + n) \log n) \end{aligned}$$

To show this algorithm correctly compute the shortest path of each nodes from the source vertex, it is sufficient to show that

1. Any sub-path within the shortest path should be the shortest path.
2. Suppose $d[u]$(the shortest path from S to u) is not the true shortest path, i.e. $d[u] > \delta(s, u)$. The shortest path then must go through some(at lease one) node out of S. Suppose the first edge across S is (x, y) where x is processed and y is not processed(obviously it can at most go out once for the shortest path, otherwise it is not shortest).
$$d[y] = d[x] + \omega(x, y) = \delta(x) + \omega(x, y)$$
since (s, y) is a sub-path of (s, u), $\delta(s, u) \geq d[y]$. While $(\delta(s, u) \geq)d[y] \geq d[u](> \delta(s, u))$ because u is extracted first, we lead to a contradiction. Note that this $\delta(s, u) > d[y]$ is true only when weights are positive.
3. Conclusion: When a new vertex u is added to the set of known-shortest-path vertices S, $d[u] = \delta(s, u)$ i.e. when u is taken out from the priority queue, cannot find a shorter distance to u than the shortest path from S to u.

Some intuitive idea of Dijkstra Algorithm:

2.4 Bellman-Ford Algorithm

This algorithm advance the Dijkstra algorithm to allow **negative edges**(negative cycles always not allowed).

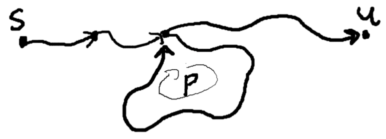


Figure 11: Negative Loops Not Allowed

One intuitive observation is that there must be at most $|V| - 1$ edges on the shortest path so that no vertex will be visit twice, i.e. no loops containing in the shortest path.[2] Then we will apply relaxation along each edge E to propagate the shortest path information until the distance converges.

```
bellman-ford(G,w,s){
    for each (u in V){
        d[u] = infty
        pred[u] = null
    }
    d[s] = 0
    repeat{ // repeat until converges, within (|V|-1) times to propagate longest information
        converge = true
        for each (u,v) in E{
            if (d[u]+w(u,v)<d[v]){
                d[v] = d[u] + w(u,v) \\ update
                pred[v] = u
                converged = false
            }
        }until(converge)
        // print the path by inverted pointers
    }
}
```

Running time: $\Theta(mn)$ for it should visit all edges each propagation, and need at most $|V| - 1$ propagation.

Proof of correctness:

[2]If positive loop, remove it directly; if negative loop, not allowed in assumption.

2.5 Greedy Algorithm: Minimum Spanning Tree

This is a problem on **undirected graph** with weighted edges. It came from the problem of communications networks and circuit design where we wish to **minimize** the length of the whole connecting network. An obvious observation is that the spanning tree is always a **connected acyclic** sub-graph of the graph, since any cycle can be reduced without destroying connectivity. The computational problem, called **Minimum Spanning Tree (MST)**, can be described as follows:

Given a weighted connected undirected graph $G = (V, E)$, a spanning tree is an acyclic subset of edges $T \subseteq E$ that connects all the vertices. For all spanning trees with total cost $\omega(T)$,

$$\text{Minimize: } \omega(T) = \sum_{e \in T} \omega(e)$$

2.5.1 Big Picture and Generic-MST

- Here are some facts need to know about tree and spanning tree:
  1. A tree with n vertices has exactly n-1 edges.
  2. There is a unique path between any 2 vertices of a tree, adding any edge between any 2 vertices of a tree creates a unique cycle. Removing any edge from this cycle restores a tree(It can be either the added edge or other edges), though not always the same tree.
  3. Deleting any edge from a tree breaks a tree into 2 sub-trees.
- **Terminologies**(see Fig.12)
  - For $G = (V, E)$, a **cut** is a partition of the vertex set V into 2 nonempty subsets S and $V - S$. The edge **crossing the cut** $(S, V - S)$ if one end point of the edge is in S while the other endpoint is in $V - S$. A cut **respects** A, which is a subset of edges, if no edges in A crosses the cut.
  - $A \subseteq E$ is **viable** if A is a subset of edges in some MST. An edge $e \in E - A$ is **safe** for A if $A \cup E$ is viable.

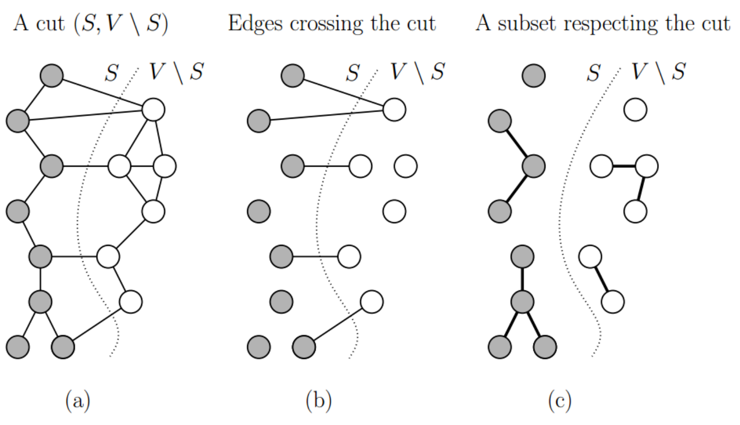


Figure 12: MST Terminology

- **Strategy:** A is the set of connected edges. Find a cut respect A, pick a lightest edge e cross the cut and add this edge into set A. Finish the process until A forms a spanning tree.
- **Correctness: MST Lemma** Let $G = (V, E)$ be a connected weighted graph, let $A \subseteq E$ be a viable subset. Let $(S, V - S)$ be any cut that respects A, and let $e = (u, v)$ be a lightest edge crossing this cut, then e is safe for A. The correctness of the strategy can be proved by induction of this lemma.

  Proof of MST lemma: let T be a MST of G that includes A.
  - If T contains e then $A \cup e$ is also a subset of T which is a MST. The lemma is valid.
  - If T does not contain e. Consider the path in T from u to v, since u and v must be on the opposite side of the cut $(S, V - S)$, there must be an edge (x, y) on this path that crosses this cut. Since e is the shortest path cross the edge, (x, y) can be updated into (u, v).

    $$T' = T + \{u, v\} - \{x, y\}$$

    which is no longer than T.

2.5.2 Kruskal's Algorithm

Kruskal's algorithm is a variant of general minimal spanning tree. It gives a way to quickly find the shortest edge crossing the cut by attempting to add edges to A in increasing order of weight, i.e., lightest weights first, while taking care **NOT** to add an edge if it create a cycle.

Since all the edges are selected in increasing order of weights, it will be the lightest edge among all for cuts crossing it while respect A. The only exception is when the edge creates a circle, then the cut cannot "respect" A, which was excluded by the algorithm. The only tricky part in this algorithm is to detect efficiently whether the addition of an edge will create a cycle in A. This could be done by DFS, but there is a faster test for this problem using **disjoint set union-find** data structure, supporting create, find and union. Implementation for Kruskal's Algorithm is shown as follows.

```
A = EmptySet
for each vertex v in V{
    create_set(v) // a set is a component in MST
}
sort the edges of E in order of increasing weight // O(mlogm)
for each edge (u,v) in E with order{ // O(logn) times for each iteration, m iterations in total
    if (find_set(u) == find_set(v)){ // A and {(u,v)} contain a cycle
        skip
    }
    if (find_set(u) == find_set(v)){  //A and {(u,v)} does not contain a cycle
        add(u,v) to A
        union(u,v) // u and v are in the same set
    }
    return A // updata A
}
```

The data structure that supports such operation is **disjoint set union-find** data structure. It is constructed by trees as follows: Every vertices points to the vertex that "leads" itself, where initially each vertex point to itself.

- find: follow the pointer to find the head node(leader).
- union: point the head of one sub-tree(smaller) to the other head whose sub-tree is larger.

2.5.3 Prim's Algorithm

Prim's algorithm is another greedy algorithm for computing MST. It provides another way to select the next safe edge to add at each step other than sorting method in Kruskal's algorithm. Intuitively, Kruskal's algorithm works by merging 2 trees together until all the vertices are in the same tree, while Prim's algorithm builds the tree up by adding leaves on at a time to the current tree.

Very similar to Dijkstra algorithm, with only difference being, instead of computing the total length of the path, we only care about the shortest distance between A and $S - A$.

Comparison between Prim's algorithm and Dijkstra algorithm.

2.6 Greedy Algorithm: Huffman Coding

In many optimization algorithm a series of selections need to be made. A simple design technique for optimization problem is based on a **greedy approach**, that builds up a solution by selecting the best alternative in each step, until the entire solution is constructed. This strategy does not always lead to optimal solution, though still very useful for its efficiency. **Huffman Coding** is one of the most well-known examples of greedy algorithm.

Huffman codes provides a **variable-length** code instead of fixed-length code method to use fewer bits for more frequent characters for a known probabilities of characters. A valid Huffman code should be **prefix code** with the following property for encoding and decoding: no code words should be prefix of other code words, i.e., as soon as we see a codeword appearing as a prefix in the encoded text, we may decode it.

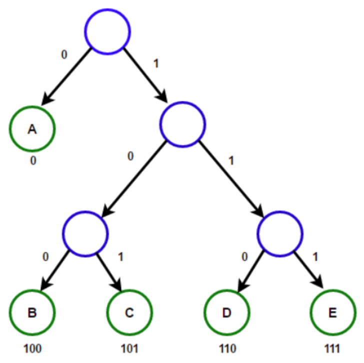


Figure 13: Prefix Tree

The problem is stated as follow: given alphabet C and $p(x)$ for probability of occurrence of each character in C, compute a prefix tree that minimize $B(T)$, i.e.minimize expected codeword length(represented by depth of T).

$$B(T) = \sum_{x \in C} p(x)d(x)$$

The intuition behind the algorithm is to build up the prefix tree from the leaf level. Take 2 characters x and y, and merge them into a single dummy character, z, where $p(z) = p(x)p(y)$ and replace the x, y in the characters. The we repeat the same process for the new alphabet optimally and recursively, with one element deduction. And since x and y appears at the bottom of the prefix tree, it is natural to assign the character with lowest probability to them. The implementation as follows:

```
huffman(C,prob){
    for each x in C{
        add x to Q sorted by prob[x]
    }
    for i in (1,|C|-1){
        z = new internal tree node
        left[z] = x = Q.extract_min()
        right[z] = y = Q.extract_min()
        prob[z] = prob[x]+prob[y]
        insert z into Q
    }
    return the last element(should have prob 1) in Q as root.
}
```

Proof of correctness

The general approach is the same as what was done in the proof of MST: show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost, i.e.at least same, by identifying an appropriate place where the two solutions (first) differ. This approach was applied in MST and Dijkstra algorithm, and will be shown again in interval scheduling problem.

Here are some of our observations and statements.

1. Every optimal prefix tree must be full, i.e.every internal node should have 2 children.
   proof: if an internal node have only 1 children, then it can be replaced by its children to decrease its cost while still being a prefix tree.

2. let x, y be the 2 characters with smallest frequency, and there is an optimal tree T in which x, y are siblings at the maximum depth.
   proof: let T be any optimal prefix tree with a, b being siblings at the bottom.
   $$T(B') = T(B) - p(a)d(a) - p(b)d(b) + p(x)d(a) + p(y)d(b) \leq T(B)$$
   which implies that $B(T')$ is at least not worse than $T(B)$, i.e. $T(B')$ is optimal.

3. let T_{n-1} be the tree that results by replacing the sub-tree x, y with a single dummy leaf z, where $p(z) = p(x) + p(y)$.
   $$B(T_{n-1}) = B(T_n) - p(z)(d(z) + 1) + p(z)d(z) = B(T_n) - p(z)$$
   Now, in order to proof the correctness of $B(T_n)$, it is sufficient to show that if $B(T_{n-1})$ is optimal, then $B(T_n)$ will be optimal if it add the smallest $p(z)$.

2.7 More Greedy Algorithm: Scheduling

2.7.1 Interval Scheduling

The problem is stated as follows: Given a set R of n activities requests with start-finish times $[s_i, f_i]$ for $1 \leq i \leq n$. Determine a subset of R of maximum cardinality consisting of requests that are mutually non-conflicting.

There are many greedy ways(which fails) for such question.

- Earliest activity first(which might last long)
- Shortest activity first(which might stuck in the middle)
- Activity with least conflict first(which might lead to a tie)

Greedy algorithm does not always succeed. For this question, the correct greedy way is such, that compare to the optimal way, will be at least not worse than the optimal way. This is how the algorithm is designed, and how the correctness is shown. The algorithm is to have **earliest finish first**, and keep tracking the next earliest finish activity not overlapping with scheduled activities.

Here is the proof of correctness. Consider the optimal solution O and the solution given by greedy strategy G. If $O = G$ then G is the optimal solution. If $O \neq G$ then consider the first activity that O and G differs. There is an activity j such that o_1 to o_{j-1} is the same as g_1. to g_{j-1}, while $o_j \neq g_j$. By definition, since the greedy strategy always catch the earliest finish activity, o_j will at least not finish earlier than g_j. Then we can replace o_j by g_j without changing the optimal solution. Thus we can make $O = G$ by such substitution and thus G is the optimal solution.

2.7.2 Interval Partitioning

This problem is a variant of interval scheduling. Suppose there are infinite number of sources, i.e. rooms, and we are to find the solution to finish all activities while consuming the least sources.

In fact, this problem can be viewed as an instance of coloring problem. That is, assigning a color to each "room" and use a new color if a new room is required. In other words, we must assign different colors to 2 conflicting activities.

2.7.3 Scheduling to Minimize Lateness

Another variant of interval scheduling problem is minimizing lateness problem. Given n tasks with execution time t_i, deadline d_i and finish time f_i, want to give an order of tasks to minimize the maximum lateness:

$$L(S) = \max_{1 \leq i \leq n} \max(0, f_i - d_i)$$

This problem is equivalent to optimize $\max(f_i - d_i) \to 0$.

The greedy solution to this problem is to let **Earliest Deadline First(EDF)**. That is to finish the task with earliest deadline first and get it away first. Use the same morph strategy above to prove its correctness. Suppose there is an optimal solution O and our solution G. Suppose O and G differs at a and b and we need to show the lateness will not increase if we swap a,b.

- Case 1(greedy solution): $L_1(S) = max(T + t_a - d_a, T + t_a + t_b - d_b)$
- Case 2(Optimal Solution): $L_2(S) = max(T + t_a + t_b - d_a, T + t_b - d_b)$

Note that both term in case 1 is smaller than $T + t_a + t_b - d_a$, then case 1 is always minor than case 2.

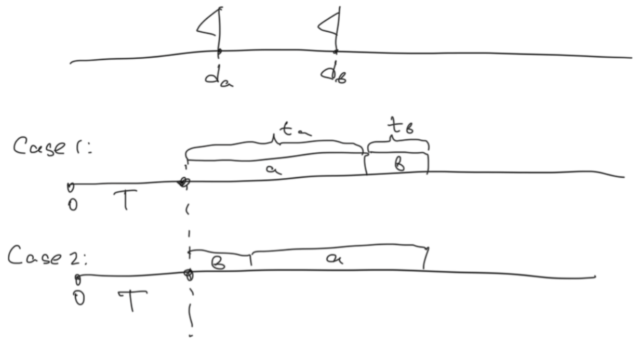


Figure 14: Correctness of Minimize Lateness

3 Dynamic Programming

Dynamic programming, DP for short, is a powerful technique for solving optimization problems that have certain well-defined clean structural properties. Unlike recursive solution in D& C, in which the subproblems are disjoint, in DP, the subproblems overlap each other, which makes it possible to reduce running time.

DP solutions reply on 2 important structural qualities:

- **optimal structure**: for the global problem to solve optimally, each subproblem should be solved optimally. (Not always true, since some problems can be solved optimally when subproblems are solved sub-optimally).
- **Overlapping subproblems**: number of distinct subproblems can be reasonably small, ideally polynomial in the input size. Based on this property, a DP problem can be solved recursively in **memoized version**, or in bottom up way as loops.

3.1 Weighted Interval Scheduling

Here is another variant of interval scheduling problem, where each activities is assigned to a weight v_i and the objective is to maximize the total non-overlapping activities with highest weight. Since no greedy solution is known for this problem, a DP solution will be demonstrated.

3.1.1 Recursive Formulation

DP solutions are based on a decomposition of a problem into smaller subproblems. First sort the requests in non-decreasing order of finish time as in greedy algorithm. Then consider the last activity $[s_n, f_n]$. This activity can either not in the optimal solution, or in the optimal solution.

- $[s_n, f_n]$ not in optimal solution: safely ignore it and solve $n-1$ size problem optimally
- $[s_n.f_n]$ in optimal solution: schedule this activity, receiving the profit of v_n, and eliminate all the activities overlapping this one. Solve the problem optimally for $p(n)$ size problem, where $p(n)$ is the largest index where $f_p < s_n$.

Here is the definition and recursive formula:

- $opt(j)$ denotes the optimize solution, maximum total weight in this problem
- $p(j)$ for the largest index where $f_p < s_n$, representing the remaining activities after elimination.

$$opt(j) = max \begin{cases} opt(j-1) \text{ where j is not in opt} \\ v_j + opt(p(j)) \text{ where j is in opt} \end{cases}$$

3.1.2 Momoized Solution and Bottom-up Construction

We can either run the solution recursively, using memoized method,

```
memoized-WIS(j){
    if (j==0) return 0
    else if (M[j] has been computed) return M[j]
    else{
        leaveWeight = memoized-WIS(j-1)
        takeWeight = v[j] + memoized (p[j])
        if (leaveWeight > takeWeight){
            M[j] = leaveWeight
            pred[j] = j-1
        }
        else{
            M[j] = takeWeight
            pred[j] = p[j]
        }
        return M[j]
    }
}
```

or calculate it bottom up and store the result in space $O(n^2)$.

```
bottom-up-WIS(){
    M[0] = 0
    for (i=1 to n){
        leaveWeight = M[j-1]
        takeWeight = M[p[j]] + v[j]
        if (leaveWeight > takeWeight){
            M[j] = leaveWeight
            pred[j] = j-1
        }
        else{
            M[j] = takeWeight
            pred[j] = p[j]
        }
    }
}
```

3.2 Longest Common Subsequence

Problem Statement: Given 2 sequences $X =< x_1, x_2, ..., x_m >$ and $Z =< z_1, z_2, ..., z_k >$. Z is a subsequence of X if there is a strictly increasing sequence of k indices $< i_1, i_2, ..., i_k >$ such that $Z =< x_{i_1}, x_{i_2}, ..., x_{i_k} >$. Given sequences $X =< x_1, x_2, ..., x_m >$ and $Y =< y_1, y_2, ..., y_n >$, determine their longest common subsequence.

3.2.1 Recursive Formulation

Intuitively, we need to observe, at which point can the problem separated and solved optimally. Consider the last element in each sequence, this element can either be in the longest common subsequence solution, or not.

- $x_m = y_n$ then the last element is in the lcs.
- $x_m \neq y_n$ then at lease one of them is not in lcs.
- $lcs(i, j)$ is the longest common subsequence of X_i and Y_j.

Recursive Formula:

$$lcs[i,j] = \begin{cases} c[i-1, j-1] + 1 \text{ if the last character match} \\ max(c[i-1, j], c[j-1, i]) \text{ if the last character does not match} \end{cases}$$

3.3 Chain Matrix Multiplication

Problem Statement: Given a sequence of matrices $A_1, A_2, ..., A_n$ and dimensions $p_0, p_1, ..., p_n$ where A_i is of dimension $P_{i-1} \times p_i$. Determine the order of multiplication(by adding parenthesis) that minimized the number of operations.

3.3.1 Recursive Formulation

- $m[i,j]$ will be the number of multiplication to compute $A_i....A_j$. $i \leq k < j$ is the "pivot" chosen. We need to compare all cases, and decide the best "pivot".

$$m[i,j] = min \begin{cases} m(i, i+1) + m(i+2, j) + p_{i-1}p_{i+1}p_j \\ ... \\ m[i,k] + m[k+1, j] + p_{i-1}p_k p_j \end{cases}$$

3.4 0-1 Knapsack Problem

Imagine a thief want to take several items with different weights ω_i and values v_i. The thief carries a knapsack capable of holding total weight W and wish to carry away the most valuable subset of items. Formally, for n items v_i with weights ω_i, the goal is to compute max$\sum v_i$ such that $\sum \omega_i \geq W$.

3.4.1 Recursive Formulation

- dp variable: $V[i, j]$.
- Leave object: if i is not taken, the dp variable would be $V[i-1, j]$
- Take object: if i is taken, the dp variable would be $v_i + V[i-1, j-\omega_i]$, and this is only possible when $\omega_i < j$.

3.4.2 Algorithm

3.5 All pairs shortest path Problem

- Input: a weighted and connected digraph $G(V, E)$.
- Output: length of shortest path between all pairs of vertices.
- Intuitive solution: apply Bellman-ford algorithm on every pair of vertices, which takes $O(n \cdot mn) = O(n^4)$ times.

4 Network Flows

The network flow problem is defined in terms of the metaphor of pushing fluids. Given a **flow network**, which is a directed graph with nonnegative edge weights. Each edge of the network has a given capacity that limits the amount of stuff it is able to carry. The idea is to find out how much flow we can push from a designated source node s to a designated sink node t. We assume that no edges entering s and no edges leaving t. All the flows should satisfy 2 conditions:

- **capacity constraint:** $\forall e \in E, f(e) \leq c(e)$
- **flow conservation:** $\forall u \in V - \{s, t\}, f^{in}(v) = f^{out}(v) = |f|$, i.e, $\sum_{u \in V} f(u, v) = \sum_{u \in V} f(v, u)$

Network flow problem is a very general problem. The bipirate problem and matching problem is a restricted case of network flow problem. The network flow problem itself, is a specified version of **linear programming** problem.

4.1 The Ford-Fulkerson Algorithm

This algorithm is to solve the max-flow problem. The reason why Greedy algorithm does not work is given by the following example. We can observe that the key insight to overcoming the problem with the greedy algorithm is that, in addition to increasing flows on edges, it is possible to decrease flows on edges that already carry flow, as long as the flow never becomes negative. That is, we need a **residual network** or **residual flow** mapping on the original network to show how the flows decreases.

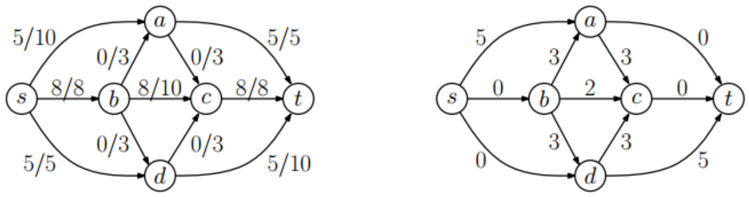


Figure 15: Greedy Algorithm For Network flow

Residual Flow: Given a flow network G and a flow f. Define its residual network G_f to be a flow network with the same source and sink, whose edges are defined as follows:

- forward edge if its original edge is not saturated: for each edge (u, v), for that $f(u, v) < c(u, v)$, create an edge in $G_f : c_f(u, v) = c(u, v) - f(u, v)$.
- backward edge if there is a flow through original edge: for each edge (u, v), for that $f(u, v) > 0$, create and edge (v, u) in G_f and assign $c_f(v, u) = f(u, v)$.

If we can push flow through the residual network, then we can push this additional flow to the original network and improve the original network. This path in G_f from s to t is the **augmenting path**.

```
ford-fulkerson-flow(G=(V,E,s,t)){
    f = 0
    while (true){
        Gf = the residual-network of G for f
        if (Gf has no s-t augmenting path)
            break
        P = any augmenting-path of Gf
        c = minimum capacity edge of P
        augment f by adding c to the flow on every edge of P
    }
    return f }
```

There are 3 issues to consider before declaring this a reasonable algorithm:

- How efficiently can we perform augmentation?
- How many augmentation might be required?
- If no more augmentations can be preformed, have we found the max flow?

The first question is connected to the computation method of augmentation, while the second question is asking for running time, and the third question points out the **termination issue**.

For augmentation method, it requires the following steps:

1. Compute the residual network G_f, which takes $O(V+E)$ times.

2. Search from s to t on G_f, which takes $O(V+E)$ times.(BFS or DFS, determines if any path P is found)

3. Compute the minimum edge on P and increase f for every edge of P

4.2 Max-Flow Min-cut Theorem

For the second and third question, we can solve the terminate problem first and explain the correctness. The **Max-flow Min-cut** theorem states that the following 3 conditions are equivalent:

- f is a max-flow in G
- The residual network G_f contains no augmenting paths
- $|f| = c(X, Y)$ for some cut (X, Y) of G.

The following is the explanation of terms and proofs.

An $s-t$ cut is a partition of the vertex set V into 2 disjoint vertex group X and Y where $s \in X$ and $t \in Y$ and $X + Y = S$. It is trivial that for any $s-t$ cut (X, Y) and any flow f, the net-flow from X to Y is exactly the value of the flow f.

$$f(X, Y) = |f|$$

Why? Consider the definition of "net-flow": $f(X, Y) = \sum f(x, y) - \sum f(y, x)$ and the definition of flow value: $|f| = f^{out}(s) - f^{in}(s)$. X is a expand of s, and due to flow conservation: $|f| = \sum(f^{out}(x) - f^{in}(x)) = \sum f(x, y) - \sum f(y, x) = f(X, Y)$.

Following this, we further define the capacity of an s-t cut (X, Y) indicating the upper bound of the value of the flow. It is also trivial that $|f| \leq c(X, Y)$ for any s-t cut.

Now consider the max-flow min-cut theorem.

- $1 \to 2$ by contradiction.
- $3 \to 1$ by the above lemma.
- $2 \to 3$. If there is no augmenting path from s to t, that means we can always find a cut (X, Y) in **residual network** G_f where Y is not reachable from X, which means that for the original network G, all paths from X to Y are saturated while all paths from Y to X are empty. This satisfy the definition of $|f| = f(X, Y) = c(X, Y)$. And this cut c is the **min-cut**.

This solves the third problem, which the terminate problem, that is, if there is no augmenting path in G_f, the solution is indeed optimal and max-flow.

4.3 Efficiency of Network Flow Algorithms

Then we can consider its running time, which is, how long does it take to ensure that there is no more augmenting path. The most intuitive way is to make 1 progress each time, that is, for integer flow unit, push 1 unit of flow each time, then the running time is $O((m+n)|f|)$.

4.4 Extensions: Bipartite matching problem

A bipartite graph G has a matching of size x if and only if G'(adding a source and a sink) has a flow of value x.

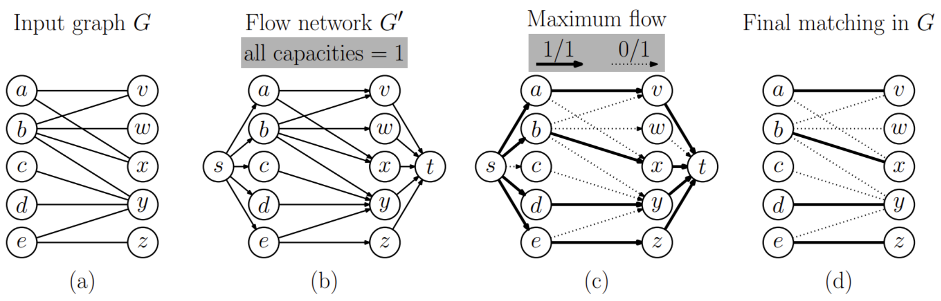


Figure 16: Reducing bipartite graph to max-flow problem

4.5 Max-flow Min-cut Extension problems

5 NP-Completeness

5.1 General Definition

A problem is said to be easy, or can be solved **efficiently** if there is an algorithm to solve the problem in **polynomial time** in worst-case. By **exponential time**, for example $O(2^{\log n^2})$ is considered "hard problems". Here are some examples of "hard problems" and their statement.

- The clique(fully connected) problem: given a graph G and:

  1. **Search problem:** find the largest clique sub-graph.
  2. **Optimization problem:** find the size of the largest clique sub-graph.
  3. **Decision problem:** decide whether a graph has clique sub-graph of size k.

  Intuitively, these 3 problems have different difficulties since the first problem indicates the second and third problem, while the second problem indicate the third problem. But they are in fact of **equal difficulty**, i.e, if one problem can be solved in polynomial time, then others can be solved in polynomial time, vise versa.

- Hamilton Cycle(a simple cycle pass through all vertices) problem:

The **class P** is the set of all decision problems that can be solved in polynomial times. The **class NP** is the set of all decision problem that yes instances can be convinced, or checked in polynomial time(of input size) by (a) **certificate**. For no instances, it is might be confirmed or not yet possible to be checked in polynomial time and there is no way to provide certificate. The Hamilton Cycle problem is an **NP problem**, we can convince its yes instances simply presenting the vertices order of the cycle, which is its certificate, and it takes polynomial time to check whether this cycle is Hamilton cycle.

It is trivial by the definition that

$$P \subseteq NP$$

And the big open question is that:

$$\text{Is } P = NP$$

Most experts **suspect** that:

$$P \neq NP$$

The idea of **NP-completeness** is to grab all hard problems in NP. The set **NP-complete** problems are problems in the complexity class NP for which it is known that if any one is solvable in polynomial time, then they all are, and conversely, if any one is not solvable in polynomial time, then none are. This is made mathematically formal using the notion of **polynomial time reductions**.

5.2 Reduction

A decision problem Π_1 is said to be polynomial-time reducible to a decision problem Π_2 if there is a polynomial-time function computable function(algorithm) f that maps instances of Π_1 to instances of Π_2 such that I is a yes-instance of Π_1 if and only if $f(I)$ is a yes-instance of Π_2.

This **reduction** (known as KARP reduction) indicates that, if there is a polynomial solution for Π_2 then there is a polynomial solution for Π_1. Furthermore, if there is no polynomial solution for Π_1 then there is no polynomial solution for Π_2. It point out that Π_2 is a **harder** problem to Π_1. All such Π_2 for **ALL** NP problems are called **NP-hard problems**. They might be in or not in NP set. The following are formal proof.

1. Lemma 1: If $\Pi_1 \leq_p \Pi_2$ and $\Pi_2 \in P$, then $\Pi_1 \in P$.

   Proof: let f denote the polynomial time reduction from Π_1 to Π_2. Let A_2 be the algorithm for Π_2 that run in polynomial time and consider the following algorithm A_1 for Π_1. Let I be the instance(input) of Π_1.

   ```
   step 1: run f for I to get I' \\ running time: p(|I|)
   step 2: run A2 on I' \\ running time: p'(|I'|) <= p'(p(|I|))
       if output is yes, then output yes for I
       if output is no, then output no for I
   ```

2. Lemma 2: If $\Pi_1 \leq_p \Pi_2$ and $\Pi_1 \notin P$, then $\Pi_2 \notin P$. This lemma is important to define group of hard problems.

3. Lemma 3: transition(if 1 problem is NP hard then all problems no easier than that is NP hard)

The importance of reduction lies as follows: if an NP hard problem can be solved in polynomial times, then all NP problems can be solved in polynomial time. On the other hand, if some NP problems cannot be solved in polynomial time then NP hard problem cannot be solved in polynomial time. The special set lies in between is the NP-complete problem, which is defined as: a decision problem Π is **NP-complete** if

1. $\Pi \in NP$
2. $\Pi' \leq_p \Pi$ for all $\Pi' \in NP$.

5.3 SAT Problem and reduction example

NP-complete problems are a list of equivalent hard core problems in NP problems. The first known NP is the satisfiability (SAT) problem. The SAT problem is stated as follows:

- instance(input): boolean formula f
- decision form: is the formula f consisting of variables and logic operations satisfiable, i.e. is there a way to assign truth values to the variables such that the final result is true, or, can the formula be made true.

5.4 Common NP Problems

1. SAT Problem: Is the formula satisfiable, i.e., is there a way to assign truth values to the variables such that the final result is true. In other words, can the formula be made true. The formula f consist of variables and logic operations. **2-SAT Problem is in P Class, while 3-SAT problem is in NP class**.
2. IS(Independent Set) Problem: Given an undirected graph $G(V, E)$, and a positive integer k, does G have an independent set of size k?
3. CC(Clique Cover) Problem:
4. Clique Problem: Given an undirected graph $G(V, E)$, and an integer k, does G have a subset V' of k vertices such that for each distinct $u, v \in V'$, $\{u, v\} \in E$.
5. VC(vertex cover) problem: a set of vertices covering every edge.
6. DS(Dominating Set): a set dominating all vertices to be in the set, or adjacent.

5.5 Polynomial Reduction Examples

- **Reducing the 3-coloring problem to the CC problem**
- **Reducing the VC problem to DS problem**
- **Reducing the SAT problem to general graph problems, e.g. IS problem**
  1. Show that IS is NP problem. Certificate: a set of k nodes and make sure this sub-graph has no edge. Can be checked in polynomial times.
  2. Show that $3SAT \leq_p IS$: for each clause, create 3 new vertices labeled by the literals. Add edges between all pairs of vertices in the same clause to ensure that no instances result in no instances. Add edges between conflict edges to ensure that yes instances correspond to yes instances. Set k to be the number of clauses and output the graph and k.
  3. Proof of correctness: claim that input formula f is SAT if and only if G has an independent set of size k. Suppose f is SAT then each of the k clauses in f has at least 1 true literals. Let V' be the set of vertices corresponding to 1 such literals in each clauses then k vertices are chosen, and this subset of vertices forms an independent set.

     Suppose G has independent set of size k, observe that exactly 1 vertex correspond to each clause.
- The 3-coloring problem: Given a graph G, can each of its vertices be labeled by one of the 3 different colors, such that no 2 adjacent vertices have the same colors. **Assume that this problem is hard**.
- The Clique problem: Given a Graph G and integer k, can we partition the vertex set into k subsets of vertices $V_1 ... V_k$ such that each V_i is a clique of G.

- **Polynomial reduction algorithm**: to rewrite the 3-Col problem into CCov problem. Consider the graph G for which we want to determine its 3-colorability and feed the CCov problem with $(\bar{G}, k)$.
- Correctness: 2 sides proof.
- Polynomial time.