课程网站 - uiuc cs374
算法 Jeff Erickson

Describing Algorithms

- **What**: A precise specification of the problem that the algorithm solves.
- **How**: A precise description of the algorithm itself.
- **Why**: A proof that the algorithm solves the problem it is supposed to solve.
- **How fast**: An analysis of the running time of the algorithm

what

- restate problems in terms of formal, abstract, mathematical objects that we can reason about formally
- determine if the problem carries any hidden **assumptions** and state those assumptions explicitly
- describe the type and meaning of each input parameter, and exactly how the eventual output depends on the input parameters.

example:

> the lattice and duplation-and-mediation algorithms both solve the same problem: Given two non-negative integers x and y, each represented as an array of digits, compute the product $x \cdot y$, also represented as an array of digits.

How

- the clearest way to present an algorithm is using a combination of pseudocode and structured English

  > Pseudocode uses the structure of formal programming languages and mathematics to break algorithms into primitive steps; the primitive steps themselves can be written using mathematical notation, pure English, or an appropriate mixture of the two

- If your algorithm has a loop, write it as a loop, and explicitly describe what happens in an arbitrary iteration; if your algorithm is recursive, write it recursively, and explicitly describe the case boundaries and what happens in each case.

Why

correctness proofs usually involve induction

How fast

- Big-O analysis or θ analysis
- Sometimes the running time of an algorithm depends on a particular implementation of some underlying data structure of subroutine
- sometimes we are interested in computational resources other than time, such as space

Recursion

- If the given instance of the problem can be solved directly, solve it directly
- Otherwise, reduce it to one or more simpler instances of the same problem.

there must be no infinite sequence of reductions to simpler and simpler instances

Analyzing running time - recursion tree

1. recursion tree：
   - The value of each node is the amount of time spent on the corresponding subproblem excluding recursive calls.
   - The leaves of the recursion tree correspond to the base case(s) of the recurrence.
   - Thus, the overall running time is the sum of the values of all nodes in the tree
2. ignoring floors and ceilings is okay - domain transformation:

$$
\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) \\
&\leq 2T(n/2+1) + O(n)
\end{aligned}
$$

   let $S(n) = T(n+\alpha)$, try to find such α that $S(n) \leq 2S(n/2) + O(n)$ and thus $S(n) = O(nlogn)$

   $$
   \begin{aligned}
   S(n) &= T(n+\alpha) \\
   &\leq 2T(n/2+\alpha/2+1) + O(n) \$\$Thus, \$\alpha = 2\$and Obviously S() Finally, we get that \\
   &= 2S(n/2-\alpha/2+1) + O(n)
   \end{aligned}
   $$

   T(n) = S(n - 2) = O((n - 2)log(n - 2)) = O(nlogn)
   $$

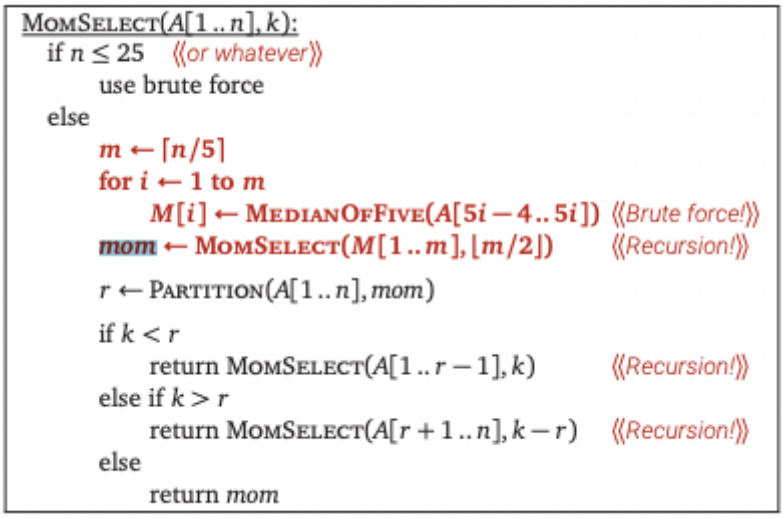
Divide and Conquer

the pattern

1. Divide the given instance of the problem into several independent smaller instances of exactly the same problem. .
2. Delegate each smaller instance to the Recursion Fairy. .
3. Combine the solutions for the smaller instances into the final solution for the given instance.
   If the size of any instance falls below some constant threshold, we abandon recursion and solve the problem directly, by brute force, in constant time.

examples

1. quickselect
   worst case - if the chosen pivot is always either the smallest or the largest,
   $T(n) = O(n^2)$
   good pivot - Momselect



```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟪or whatever⟫
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i−4..5i])   ⟪Brute force!⟫
      mom ← MomSelect(M[1..m], ⌊m/2⌋)        ⟪Recursion!⟫
      r ← Partition(A[1..n], mom)
      if k < r
          return MomSelect(A[1..r−1], k)      ⟪Recursion!⟫
      else if k > r
          return MomSelect(A[r+1..n], k−r)    ⟪Recursion!⟫
      else
          return mom
```

> mom is larger than $\lfloor \lceil n/5 \rceil /2 \rfloor - 1 \approx n/10$ block medians, and each block median is larger than 2 elements. Thus, mom is larger than at least 3n/10 elements. Similarly, mom is smaller than at least 3n/10 elements. Thus, in the worst case, the second recursive call spend $T(7n/10)$
> $T(n) \leq T(n/5) + O(n) + T(7n/10)$, by recursion tree
> $\rightarrow T(n) \leq n \sum_{k=0}^{s} \sum_{i=0}^{k} (1/5)^i (7/10)^{k-i} = n \sum_{k=0}^{s} (1/5 + 7/10)^k = n \sum_{k=0}^{s} (9/10)^k$
> $\rightarrow T(n) = O(n)$

2. fast multiplication
    - Karatsuba's algorithm see: Discrete Structures > Running time analysis of some algorithms $O(n^{log3})$
    - Toom-Cook algorithm $O(n^{1+1/logk})$
    - Fast Fourier transform based algorithm: Schonhage-Strassen algorithm $O(nlognloglogn)$
    - 2019 David Harvey, Joris van der Hoeven $O(nlogn)$
3. exponentiation a^n - divide a or divide n by 2
   computes a^n must perform at least $\Omega(logn)$ multiplications, because each multiplication at most doubles the largest power computed so far

Backtracking

the pattern

A backtracking algorithm tries to construct a solution to a computational problem **incrementally**, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates every alternative and then chooses the best one.

In each recursive call to the backtracking algorithm, we need to make exactly one decision, and our choice must be consistent with all **previous decisions**(key feature of backtracking).

thus, each recursive call requires(they are what we need to figure out when we design the signature for the backtracking function):

1. the data we have not yet processed
2. a suitable summary of the decision we've already made

and we implement the function by recursive brute force:

> Try all possibilities for the next decision that are consistent with past decisions, and let the Recursion Fairy worry about the rest.

most of the backtracking algorithms are only intermediate results on the way to more efficient algorithms

examples

1. n - queens
2. game trees
    - **a state of the game** consists of 1) the locations of all the pieces and 2) identity of the current player. these state can be connected into a game tree, which has an edge from state x to state y iff the current player in state x can make a legal move to transform game state from x to y
    - def: a state is good for the current player if either the current player has already won, or if the current player can move to a bad state for the opposing player
    - def: a state is bad for the current player if either the current player has already lost, or if every available move leads to a good state for the opposing player
    - def: if a state is good for player1, then it's bad for the other player. vice versa.
    - it can be proved that for any state, any player, it's either good or bad for that player. (using induction: base: for the state where the winner has been determined(leaf nodes of the game tree), if it's not good for the current player -> the current player does't win -> the current player lose -> the state is bad for the current payer. for the opposing player, if it's not good, then for the current player, it's not bad -> current player doesn't lose -> current player win -> it's good for the current player -> it's bad for the opposing player. induction: for any internal nodes of the game tree, suppose that for any states reachable from the current state, the prop is true, then if the current state is not good for the current player -> there is no such a move that leads to a bad state for the opposing player -> every available move leads to a not bad state for the opposing player -> every available move leads to a not good state for the current player -> because for every reachable state, the prop is true, it means that every available move leads to a bad state for the current player, i.e. every available move leads to a good state for the opposing player -> the current state is bad for the current player; if the current state is not good for the opposing player -> the current state is not bad for the current player -> there is a move that leads to a not good state for the opposing player -> there is a move that leads to a bad state for the opposing player -> the current state is good for the current player -> the current state is bad for the opposing player. proved)
    - for any two-player game without randomness or hidden information that ends after a finite number of moves, we can apply this game tree model to figure out whether a player will win or lose the game if he and his opponent plays perfectly at any point during the game.

```
// whether the state specified by location and current is good
for the current player
// if isGood,the currentPlayer will win the game even if his
opponent plays perfectly. Otherwise, the currentPlayer will
lose the game(opponent win) if his opponent does not make any
mistakes.
isGood(location, currentPlayer):
        if by location, currentPlayer win
                return true
        if by location, currentPlayer lose
                return false
        for all moves from state(location, currentPlayer) ->
state(location', currentPlayer') // for simplicity, suppose
that currentPlayer' is the opponent
                if !isGood(location', opponent)
                        return true

        return false
```

3. subset sum
4. text segmentationlc139 word break
5. Longest Increasing Subsequence
6. optimal binary search tree

Dynamic Programming

intro-fibonacci

```
RecFibo(n):
        if n = 0
                return 0
        if n = 1
                return 1
        return RecFibo(n - 1) + RecFibo(n - 2)
```

time complexity:
$T(n) = T(n-1) + T(n-2) + 1, T(0) = 1, T(1) = 1$
$\rightarrow T(n) = 2F_{n+1} - 1$
think of the recursion tree as a binary tree of additions($v_{parent} = \sum_{c\ is\ child\ of\ parent} v_c$), the leaf value of which is the solution of the corresponding subproblem. i.e. F_0 in leaves representing subproblem $RecFibo(0)$ and F_1 in leaves representing subproblem $RecFibo(1)$. Then, by induction, the value of the node representing subproblem $RecFibo(i)$ is F_i. Also, by induction, we can know that for a tree of additions, $v_r = \sum_{i\ is\ a\ leaf\ node\ of\ the\ tree\ with\ root\ r} v_i$. The root of the recursion tree has value F_n and values of its leaves are either $F_0 = 0$ or $F_1 = 1$, so we know that the recursion tree has F_n leaves storing F_1. Moreover, by induction, we can know that for a node with value $F_i (i >= 1)$, the tree rooted at this node has F_{i-1} leaves storing F_0(for i = 1, obvious. suppose that it's true for all the descendants of node storing F_r, then, for this node, # leaves storing F_0 = # leaves of left subtree storing F_0 + # leaves of right subtree storing F_0 = $F_{r-2} + F_{r-3} = F_{r-1}$. proved). Thus, the recursion tree has F_{n-1} leaves storing F_0. Now, we know that the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves. Since it's a full binary tree, the number of nodes of the recursion tree is $2 * \#leaves - 1 = 2F_{n+1} - 1$. proved.

memoization

writing down the results of our recursive calls and looking them up again if we need them later.

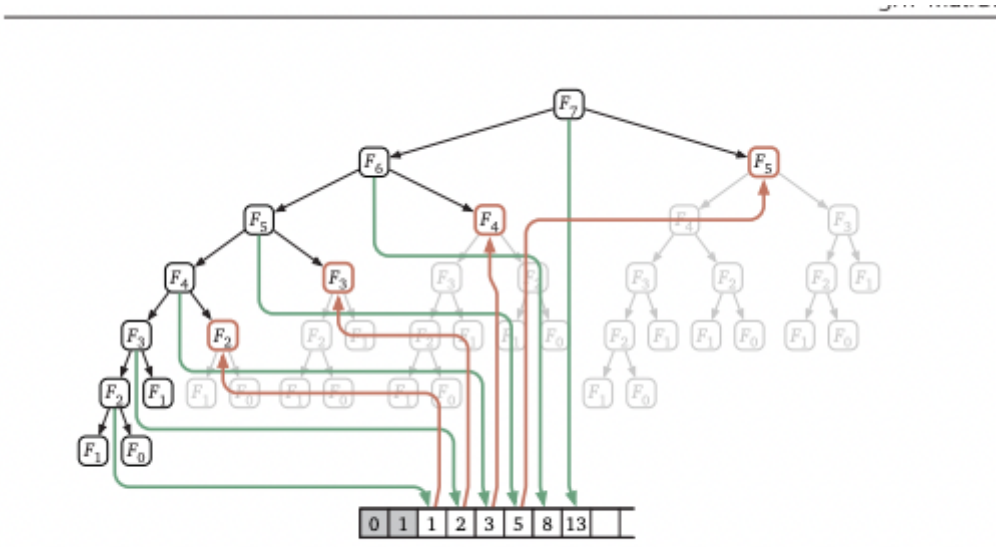


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

we can fill F[] deliberately - n replace the memoized recurrence with a simple for-loop that intentionally fills the array in that order, instead of relying on a more complicated recursive algorithm to do it for us accidentally - dynamic programming

now, for fibonacci problem, uses O(n) additions. Because F_n is approximately 2n/3 bits, we can not think of the time complexity of addition as O(1). Instead, it's O(n). Thus, this dynamic programming version solution actually runs in $O(n^2)$ time.

> faster fibo:
>
> $$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$
>
> If we use repeated squaring, computing the nth power of something requires only O(log n) multiplications. Here, because "something" is a 2 *2 matrix, that means O(log n) 2* 2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute Fn in only O(logn) integer arithmetic operations. Because the fastest multiplication algorithm known runs in O(nlogn) time, the running time of this fast fibo is $O(nlogn)$

pattern

dynamic programming is recursion without repetition. It stores the solutions of intermediate subproblems. **It's usually an improvement of backtracking!**

steps to develop a dynamic programming algorithm for a problem:

1. formulate the problem recursively
2. build solutions to your recurrence **from the bottom up**
    - choose a memoization data structure to store solutions to subproblems (array, tree,...)
    - identify dependencies between subproblems. if a depends on b, draw an arrow from b to a.
    - find a good evaluation order - topology sorting

whenever you think of greedy algorithm, your subconscious is telling you to use dynamic programming.

examples

1. lc139 word break
2. subset sum
3. Longest Increasing Subsequence
4. optimal binary search tree
5. edit distance

6. dynamic programming on trees - usually use recursion to implement tree traversal[lc124.binary tree maximum path sum]

Greedy algorithm

pattern

typically, we prove the correctness by:

exchange argument

The idea of a greedy exchange proof is to incrementally modify an optimal solution into the solution produced by your greedy algorithm in a way that doesn't worsen the solution's quality. So the quality of your solution is at least as great as the optimal solution and thus is also optimal.

1. assume that there is an optimal solution that is different from the greedy solution
2. find the first difference between 2 solutions
3. argue that we can exchange the optimal choice for the greedy choice without making the solution worse
4. thus, there are some optimal solution modified from the original one that contains the entire greedy solution.once we prove that it equals to the greedy solution, the correctness is proved

charging argument

It is typically used to show that an algorithm produces optimal results by proving the existence of a particular injective function from the optimal solution to the algorithm output or vice versa.

examples

1. storing files on tape - sort increasingly by L/F. proof: if there is 2 adjacent decreasing elements, we can improve cost by swapping them.
2. scheduling classes: [lc435.non-overlapping intervals]
3. huffman codes - find an binary code by which the encoded string is the shortest

- an optimal code tree must be a full binary tree
- there is an optimal code tree in which the least frequent x and y are siblings and have the largest depth of any leaf.

> Let T be an optimal code tree, and suppose this tree has depth d. Because T is a full binary tree, it has at least two leaves at depth d that are siblings. Suppose those two leaves are not x and y, but some other characters a and b. Swapping x and a gives us a new optimal tree. And then swapping y and b gives us a optimal tree where x and y are maximum-depth siblings, as required.

- huffman code is optimal
  > let T be an optimal code tree for [1...n] (without loss of generality, assume that 1 and 2 have the smallest frequency) s.t. the least-frequent 1 and 2 are siblings(so, depth(1,T) == depth(2,T). remove 1 and 2 from the tree, and use their parent to encode n + 1, (depth(n + 1, T') = depth(1,T) - 1) we get a code tree T' for [3..(n+1)] (f(n + 1) == f[1] + f[2]).
  >

$$
\begin{aligned}
cost(T') &= \sum_{i=3}^{n} f(i)depth(i,T') + f(n+1)depth(n+1,T') \\
&= \sum_{i=1}^{n} f(i)depth(i,T) - f(1)depth(1,T) - f(2)depth(2,T) + f(n+1)depth(n+1,T \\
&= cost(T) - (f(1)+f(2))depth(1,T) + (f(1)+f(2))(depth(1,T)-1) \\
&= cost(T) - (f(1)+f(2)) \\
&\geq cost^*(3..n+1)
\end{aligned}
$$

  Thus, cost*(1..n) = cost(T)* >= *cost*(3..(n + 1)) + f(1) + f(2). Suppose we've got an optimal code tree for [3..(n + 1)] T, we replace the leaf node that stores n + 1 with an internal node with two leaf children (one stores 1 and the other stores 2). Then, we get a new tree T'' that encodes [1..n].

$$
\begin{aligned}
cost(T'') &= \sum_{i=1}^{n} f(i)depth(i,T'') \\
&= \sum_{i=3}^{n} f(i)depth(i,T) + f(1)(depth(n+1,T)+1) + f(2)(depth(n+1,T)+1) + f(\\
&= cost(T) + (f(1)+f(2)-f(n+1))depth(n+1,T) + f(1) + f(2) \\
&= cost(T) + f(1) + f(2) \\
&= cost^*(3..n+1) + f(1) + f(2)
\end{aligned}
$$

> we've known that cost*(1..n)* >= *cost*(3..n+1) + f(1) + f(2). since T'' has a cost reaching the lower bound, T'' is an optimal code tree for [1..n] and cost*(1..n)* = *cost*(3..n + 1) + f(1) + f(2).
> Therefore, only if we can build an optimal code tree for [3..n + 1], we can build an optimal code tree for [1..n]. Huffman code tree is exactly a tree built in this way.

```
BuildHuffman([1..n]):
        // pq is an priorityqueue sorting elements ascendingly by f[i]
        // it tracks currently what elements we want to build an optimal
code tree for
        enqueue 1..n into pq
        for (i in (n + 1) to (2n - 1)) {
                first <- pq.poll()
                second <- pq.poll()
                f[i] = f[first] + f[second]
                pq.offer(i)
                L[i] = first
                R[i] = second
                P[first] = i
                P[second] = i
        }
        // 2n - 1 is the root of the huffman code tree
```

4. stable matching
    - a match is unstable if there is a doctor α and hospital B that would be both happier with each other than with with their current match(unstable pair).
    - Gale-Shapley algorithm
      doctor offers its position to the best doctor who has not rejected it; doctor tentatively accepts the offer and reject the previous one(if any) if the current offer is better, otherwise, he rejects the current offer.
      time complexity: O(n^2) && omega(n^2)
      correctness:
      because each hospital makes an offer to a doctor at most once, the algorithm must terminate. And suppose that there is a doctor unmatched(and also, a hospital unmatched) when the algorithm terminates(all the hospital unmatched have exhaust their pref list), it indicates that he has not received any offer yet, which implies that none of the hospitals has exhausted their pref list, contradicting to that there is an unmatched hospital that exhausts its list. It follows that when the algorithm terminates, every doctor is matched.

      We now need to prove that the matching is stable. Let α be a doctor matched to A. For any hospital B other than A, obviously, (α, B) is not an unstable pair if α prefers A over B. Let B be a hospital α prefers compared to A. Because every doctor accepts the best offer he receives, α receives no offer he likes more than

A. Thus, he didn't receive offer from B, which means B has been matched to a doctor listed prior to α in its pref list. That is, B prefers its current pair β over α. Then, (α, B) is not an unstable pair in this case, either. We conclude there is no unstable pair; the matching is stable.
other fun facts:

Corollary 4.8. *The Gale-Shapley algorithm matches best(A) with A, for every hospital A.*

In other words, the Gale-Shapley algorithm computes the *best possible* stable matching from the hospitals' point of view. It turns out that this matching is also the *worst* possible from the doctors' point of view! Let ***worst*(α)** denote the lowest-ranked feasible hospital on doctor α's preference list.

Corollary 4.9. *The Gale-Shapley algorithm matches α with worst(α), for every doctor α.*

Graph

Basic Definitions

G = (V, E)

- simple graph: graph without parallel edges and loops(edge from a vertex to itself)
- neighbor: for any edge uv in an undirected graph, u a neighbor of v and vice versa. u and v are adjacent. for any directed edge u $\rightarrow$ v, u is ***predecessor*** of v, v is a ***successor***. ***in-degree*** is the number of predecessors, ***out-degree*** is the number of successors
- walk: a sequence of edges. ***path***: a walk that visits each vertex at most once ***cycle***: a closed walk that enters and leaves each vertex at most once. ***forests***: graph without any cycles. ***tree***: connected acyclic graph - one component of a forest a graph is a tree iff V = E + 1. ***spanning tree***: subgraph of G that is a tree and contains all the vertices of G
- for directed graph: is ***strongly connected*** if every vertex is reachable from every other vertex

Representations

each vertex is represented by an integer identifier and thus can be sorted

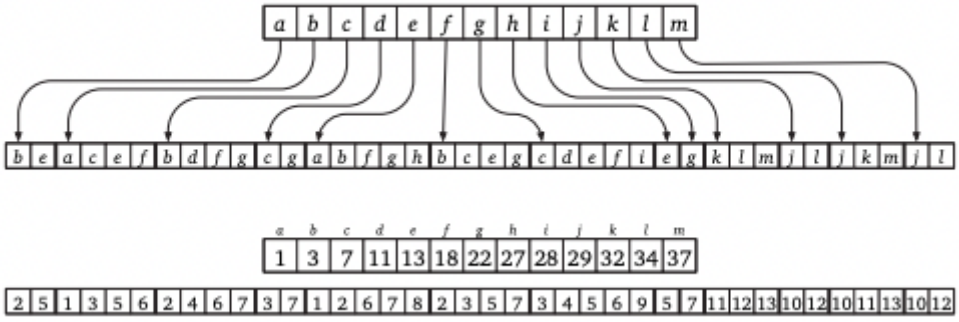
drawing

- planar and embedding: A graph is planar if it has a drawing where no two edges cross; such a drawing is also called an embedding

adjacency list

For undirected graphs, each edge uv is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge uv is stored only once, in the neighbor list of the tail u.
the overall space complexity is O(V + E)
2 common implementations:

1. array of singly-linked list（standard adjacency list)/hash table/balanced binary search tree
2. adjacency array:
   1) uses a single array $edges$ to store all edge records, where the records of edges incident to each vertex are stored in a contiguous interval(records for each vertex are sorted)
   2) use another array to store the index of the first edge incident to each vertex in $edges$



   with such implementation, we can check whether u and v are adjacent in O(log(deg(u)))

adjacency matrix

if the graph is undirected, then A[u, v] := 1 if and only if uv $\in$ E. it's always ***symmetric***
if the graph is directed, then A[u, v] := 1 if and only if u $\rightarrow$ v $\in$ E

comparison

Comparison

Table 5.1 summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.[9]

	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u \to v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Table 5.1. Times for basic operations on standard graph data structures.

Traversal

reachability question asks whether there is a path from s to v [leetcode习题]
we can use the graph traversal algorithm - ***whatever-first search*** to solve it

algorithm - whatever first search

```
WhateverFirstSearch(s):
        put ('', s) into bag
        while bag is not empty:
                take (p,v) from the bag
                if v is unmarked
                        mark v
                        parent[v] = p
                        for vu in v's edges:
                                put (v, u) into bag // we can only put edges (v,w) s.t. w is unmarked. However, even though, we still need to check whether w is marked when we take (v,w) out of bag. Because w may have been marked when (v,w) is taken out of the bag though it was not marked when the edge is put into the bag
```

correctness

1. by induction on the shortest-path distance from s to v, we can prove that ***all reachable vertices from s will be marked***: suppose that all the reachable vertices with shortest-path distance < n are marked. Then, for an arbitrary vertex v reachable

from s with shortest-path distance n, there is a shortest path $s->v1...v_{n-1}->v$. There is at least a path from s to v_{n-1} with length n - 1. Thus, v_{n-1} is a reachable vertex and its shortest-path distance is less than or equal to n - 1. By inductive hypothesis, it has been marked. when the algorithm marks it, (v_{n-1}, v) will be put into the bag, so it must later take the pair out of the bag and finally mark v if it has not been marked. proved.

2. for any marked vertex v, the path of parent edges v -> parent(v) -> parent(parent(v)) ->... eventually leads back to s(but ***not*** necessarily the shortest path!!!). by induction on the order in which vertices are marked, we can prove it. and this claim implies that all the marked vertices is reachable. Combined with the previous claim, we can conclude that the algorithm marks a vertex iff it's reachable.

time complexity

depends on graph representation(let T be the time required to insert or delete a item from the bag)

- adjacency list - O(V + ET)
- adjacency matrix - O(V^2 + ET)

implementation/variants

1. when the bag is a stack, it's ***Algorithms by Jeff > DFS traversal***.
2. when the bag is a queue, it's ***bfs*** - the breadth first spanning tree formed by the parent edges contains shortest paths from s to every other vertex in its component
3. when the bag is a priority queue - best-first search. since there are at most E edges in the bag, T is logE, and the time complexity is O(V + ElogE).
    - Prim's algorithm, for edge (u, v) ,use weight(u,v) as its priority
    - Dijkstra's algorithm, use dist[u] + weight(u,v) as its priority(there is another implementation of Dijkstra's algorithm which puts vertices instead of edges into bags as we do here - see sedgewick's algorithms,4th edition)
    - maximum flows (max minimum)- define width(path) to be the minimum weight of any edges in the path, to find the path that have the largest width(widest path) from s to every other reachable vertex. use min(width[u](the width of the widest path from s to u),weight(u,v)) as its priority and poll the largest out of the bag first.

application

label components
```
CountAndLabel(G):
count <- 0
unmark all vertices
for all vertices v
if v is unmarked
count <- count + 1
Label(v, count) // whatever first search
```

```
Label(v, count):
put (",v) into bag
while bag is not empty
        take (p, v) out of bag
        if v unmarked
                mark v
                comp[v] <- count
                for (v, w) in v's edges
                        put (v,w) in bag
```

```
a fun fact about unmarking vertices:
you can do it in ***O(1)*** !!!
> On the other hand, if we store a time-stamp at every vertex indicating
the last time it was “marked”, then we can “unmark every vertex” in O(1)
time by recording the start time of our traversal, and considering a
vertex “marked” if its time stamp is later than the recorded start time.
```

DFS Traversal

（! based on directed graph)

any graph, whatever characteristics this graph has (acyclic or not, and so on), can be DFS!! in fact, we use DFS to detect whether the graph has some features of interest to us

in practice, we usually implement dfs recursively instead of using whatever first search

```
DFS(v):
mark v
previsit(v)
for each edge vw
        if w is unmarked
                parent[w] = v
                dfs(w)
postvisit(v)

DFSAll(G):
for each vertex v
        unmark v
for each vertex v
        if v is unmarked
                DFS(v)
```

The parent pointers assigned by DFS(v) define a tree rooted at v whose vertices are precisely reach(v) - after DFSAll, we get a depth-first forest. for a Graph, how the depth-first forest is depends on how we DFSAll the graph.

3 Status of Vertices

```
dfs(v, clock):
mark v
clock <- clock + 1
v.pre = clock // v.pre is assigned immediately after pushing v onto the
recursion stack
for each edge vw
        if w is unmarked
                clock <- dfs(w, clock)
clock <- clock + 1
v.post <- clock // v.post is assigned before popping v off the recursion
stack
```

for any v and w, [v.pre, v.post] and [w.pre, w.post] is either nested or disjoint. [v.pre, v.post] contains [w.pre, w.post] iff v is an ancestor of w. v.pre defines a preordering of the vertices while v.post defines a postordering one

we say a vertex v is:

1. new - clock < v.pre
2. active - v.pre <= clock < v.post
3. finish - v.post <= clock

v is active iff it's in recursion stack. That is, all active vertices are in the recursion stack, which indicates a directed path comprising these vertices

Detecting Cycles

for any u -> v, if u.post < v.post, then when dfs(u) is called, v is active. otherwise, if

- v is finish, v.post < u.post
- v is new dfs(u) will call dfs(v) directly or indirectly. Both imply that v.post < u.post thus, v is in the recursion stack. and there is directed path from v to u. Therefore, there is a directed cycle through u -> v.

on the other hand, if there is a cycle, then, there must be an edge u -> v s.t. u.post < v.post

so, we can conclude that ***there is a cycle iff there is an edge u -> v s.t. u.post < v.post***

This inspires our first cycle detecting algorithm:
dfs(O(V + E)) to compute v.post and then check (O(E)) whether there is an edge u -> v s.t u.post < v.post. the graph contains a directed cycle iff there is such an edge. (total time complexity: O(V + E))

```
isAcyclic(G):
unmark all v
clock <- 0
for each v
        if v unmarked
                clock <- dfs(v, clock) // dfsAll(G), there is no extra
requirement for v(for example, no edge comes to it), we can start dfs in
any unmarked v and get correct pre/postordering traversal of G
for each edge u -> v
        if u.post < v.post return false
return true
```

Actually, we can check v's status during dfs. once we find it active, we detect a cycle through current edge we're checking and return false(it's not a DAG)(O(V + E))

```
isAcyclic(G):
set all v's status to be new
for each v
        if v's status is new
                if (!isAcyclicVertex(v)) return false
return true

isAcyclicVertex(u):
u.status = active
for each edge u -> v
        if v.status is new
                if (!isAcyclicVertex(v)) return false
        if v.status is active
                return false
        if v.status is finish
                do nothing
u.status = finish
return true
```

we can use 2 ($\lceil log3 \rceil$) boolean maps (1. key: vertex value: is marked 2. key: vertex, value: isActive) to maintain vertices' status

isMarked	isActive	status
true	true	active
true	false	finish
false	false	new
false	true	can not happen

Topological Sort

a topological ordering of a directed graph G is total order < on V s.t. u < v for any edge u -> v. In other words, a topological ordering arranges all the vertices along a line so that all edges point from left to right.

there is a topological ordering iff there G is a DAG

⇒

suppose there is a cycle: u ->...->v -> u. we have to put u to the left of v to get a topological ordering. however, v -> u will be an edge point from right to left, which destroys the ordering

⇐

we've known that if there exists an edge u -> v s.t. u.post < v.post, there is a cycle. In other words, if G is DAG, for any edge u -> v, u.post >= v.post. we define R: u R v if u.post >= v.post. This R is the total order we want.

by the second part of the proof, we get an algorithm for finding topological sorting for a DAG: dfs and then sort vertices by their post decreasingly -> ***topological ordering is reversed postordering***

it also indicates that if we want to process a DAG in reverse topological order, it suffices to process each vertex just before it pops off the stack(at the end of its recursive dfs)

Strong Connectivity

definitions

a directed graph is strongly connected iff for any u、v, u and v are strongly connected(u can reach v while v can reach u)

strongly connectivity is an equivalence relation(reflexive, symmetric, transitive) over the set of vertices. the equivalence classes of this relation are ***strongly connected components(strong components.)***

strong component graph scc(G) is another graph obtained from G by contracting each strong component to a single vertex and collapsing parallel edges. it's a DAG.

compute strong component

for single vertex v

```
// time complexity: O(V + E)
whatever-first-search(v) and collect all vertices reachable from v
reverse directions of all edges in G
```

```
whatever-first-search(v) and collect all vertices reachable from v
intersect these 2 sets
```

compute all strong components - kosaraju and sharir's algorithm

scc(G) is a DAG, there is a sink component C(no edge in scc is from it). and for an arbitrary vertex v in C, any vertex that is in reach(v) is in C. also, for any vertices in C, since they are in the same component as v, they are in reach(v). Thus, we conclude that reach(v) = C. So, we can pick an arbitrary vertex from the sink component C and find all vertices reachable from it. The set of these vertices are exactly C.
it gives us an intuitive outline for algorithms to compute strong components

```
StrongComponents(G):
count <- 0
while G is non-empty
        C <- empty
        count <- count + 1
        v <- any vertex in a sink component of G
        whatever first seach(v) to find reach(v), put them to C, label
them with count
        remove C and its incoming edges from G
```

how to find sink component?

lemma1: for a fixed DFS, these 4 statements are equivalent:

1. u is the ancestor of v in depth-first forest
2. u.pre <= v.pre <= v.post <= u.post
3. after DFS(v) is called, u is active
4. before DFS(u) is called, there is a path from u to v where all vertices are new
   proof:
   1 -> 2: u is the ancestor of v implies that there is a path in depth-first forest that's from u to v. by induction on the length of the path in the forest, for any path u -> ... -> w(parent(v))->v with length n, u.pre <= w.pre<=w.post <= u.post and w.pre <= v.pre <= v.post <= w.post, then u.pre <= v.pre <= v.post <= u.post. proved
   1 -> 4: we've proved for any path from u to v in depth-first forest, u.pre <= v.pre. in particular, for each tree edge u -> v, u.pre <= v.pre. thus, if u is the ancestor of v, there exists a path from u to v in depth-first forest u ->w1 -> w2...->v and u.pre <=

w1.pre <= w2.pre <= ...<=v.pre. that is, for this path, before DFS(u) is called (clock < u.pre), every vertex in this path is new. proved.
2 -> 3: if u is new, then v.pre < u.pre, contradict. if u is finish, then u.post < v.post, contradict. proved
3 -> 1: u is active, the recursion stack implies a path consisting of tree edges from u to v, i.e. u is ancestor of v
4 -> 1: it's logically equivalent to prove its contrapositive. suppose u is not the ancestor of v. for any path from u to v, let w be the first vertex that is not the descendant of u, and x is the predecessor of w. then x is the descendant of u and u.pre < x.pre < x.post < u.post. because w is not the descendant of u, when DFS(x) is called, w is finish or active
if w is finish, w.pre < w.post < x.pre < u.post, then w.pre < u.pre. otherwise, u.pre < w.pre < w.post < u.post, since we've proved that 1) is logically equivalent to 2), this implies u is ancestor of w, contradict.
if w is active, w.pre < x.pre < x.post < w.post, then w.pre < u.pre. otherwise, u.pre < w.pre < x.pre < x.post < w.post < u.post, u is ancestor of w, contradict.
Thus, w.pre < u.pre, which means that before DFS(u) is called, for any path from u -> v, there always exists one vertex w that is not new. proved.

lemma2: let u, v be any vertices from C, then for any path from u to v, for any vertex w in the path, w is in C.
proof: for any vertex x in C, x can reach u, and thus can reach w. on the other hand, v can reach x, thus w can reach x. so w is in C.

lemma3: for a DFS, each strong component C contains exactly one vertex v s.t. v either does not have a parent or parent(v) is not in C. and this v is the vertex in C that has the smallest v.pre. (we call it the root of C)
proof:
let v be the vertex that has the smallest v.pre (i.e. DFS(v) is called the earliest). if it has a parent, then by definition, parent(v) is not in C. Also, by definition, we can know that before DFS(v) is called, each vertex in C is new. Moreover, for any vertex w in C, because w and v are in the same C, there exists a path consisting of vertices that are in C. we can conclude that before DFS(v) is called, there is a path from v to w where all vertices are new. by lemma1, v is ancestor of w in depth-first forest(there is a path consisting of tree edges from v to w). by lemma 2, parent(w) is in C. proved.

lemma4: the last vertex in postordering of G lies in the source component of G
proof:
fix a DFSAll, let v be the last vertex in postordering of G. then DFS(v) is the last one called

in DFSAll and thus v has no parent. by lemma3, for any w(w != v) in the same component(let it be C) as v, parent(w) is in C(thus, any vertex in C either has no parent or its parent is in C); for any w in C, v.pre <= w.pre. for any vertex w in C and any edge to w: u -> w. when DFS(u) is called:

1. w is new, then parent(w) = u, so u is in C
2. w is finish, then v.pre <= w.pre < u.pre. if v.post < u.pre, it contradicts to that v is the last vertex in postordering. so v.pre < u.pre < u.post < v.post. u is descendant of v. v can reach u. so any vertex in C can reach u. u can reach w and thus can reach any vertex in C. so u is in C
3. w is active.then there is a cycle through u -> w.so for any vertex in C, u can reach w and then can reach it. w can reach u and this vertex can reach w as well. so u is in C.
   Therefore, C is the source component of G.

so we can find a sink component of G by finding a source component of rev(G), i.e. dfs rev(G) and pick the last vertex in its postordering

pseudocode:

```
KosarajuSharir(G): // v.root was used to store the root of the strong
compoent C where v in (see the definition of root in lemma3)
// get rev(G)'s postordering
initialize an empty stack st
for each v:
        if v is unmarked:
                DfsRevG(v, st)

// compute strong component
while st is not empty:
        v <- st.pop
        if v.root = none
                DfsAndLabel(v, v)


DfsRevG(v, st):// we must use dfs to get postorder
mark v
for each w -> v
        if w unmarked
```

```
                DfsRevG(w, st)
st.push(v)

DfsAndLabel(v, r):// actually, we can use whatever-first search
v.root = r
for each v -> u
        if u.root = none
                DfsAndLabel(u, r)
```

Minimum Spanning Trees

for an edge weighted undirected graph, to find a spanning tree(for connected graph) that minimizes the function $\sum_{e \in T} w(e)$
what is a spanning tree:

1. subgraph of G
2. spanning: includes all of the vertices
3. tree: connected and acyclic

uniqueness

if all edge weights in a connected graph G are distinct, then G has a unique minimum spanning tree

proof: it's logically equivalent to prove its contrapositive. That is, if G has 2 different minimum spanning tree, then some pairs of edges have the same weight.
let T and T' be 2 different minimum spanning trees of G. there must be some edges in T but not in T' while some edges in T' but not in T. let e be the edge with the smallest weight in T/T‘ and e' be the edge with the smallest weight in T'/T. Without loss of generality, suppose that w(e) <= w(e'). Add e to T', then we get a cycle through e in T'(because 2 endpoints of e are connected in T'). And in this cycle, there must be an edge e'' that's not in T(otherwise, if all the edges of the cycle are in T, T will contain a cycle. Contradict to the fact that T is a tree). So, e'' is an edge in T'/T. w(e'') >= w(e') >=w(e). On the other hand, for the new spanning tree T'' = T' - e'' + e. w(T'') = w(T') - w(e'') + w(e) >= w(T')(because T' is a minimum spanning tree). So w(e'') <= w(e). Thus, w(e'') = w(e). proved.

In fact, even though the graph has some pair of edges sharing the same weight, we can apply a consistent method to break the tie when processing the graph and think of it as a

graph in which edge weights are distinct.

algorithm

generic idea

initially, F consists of V one-vertex trees;
add edges to F until we get a connected F while maintaining the invariant:

> F is a subgraph of the minimum spanning tree of G

if we can finally get the connected F, it's the minimum spanning tree(F is connected, so $V = V_F <= E_F + 1$; F is subgraph of MST, so $E_F <= E_{MST} = V - 1$. Thus, $E_F = V - 1$. F is also a subgraph of minimum spanning tree. Suppose that there are some edges in MST/F, then $E_{MST} > E_F = V - 1$, which is impossible. Thus, F is actually MST)

the question is which edge-adding strategy we should use to

1. maintain the invariant (Q1)
2. and to make sure that the algorithm will halt with a connected F(Q2)

3 status of edge in G/F

1. **useless**: its endpoints both in the same component of F
2. **safe**: the minimum-weight edge with exactly one endpoint in some component of F
3. **undecided**: others

lemma: for any subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S.

proof:
it's logically equivalent to prove its contrapositive. That is, for any subset S, if a spanning tree of G does not contains minimum-weight edge, it's not the minimum spanning tree. Let T be such a tree and e(u -> v) be the minimum-weight edge. Without loss of generality, suppose that u in S while v is not. Then there is path from u to v in T and there must be an edge e' (e' $\neq$ e) in the path that points from a vertex in S to a vertex not in S. since e is the minimum-weight edge among all edges that have exactly one endpoint in S, w(e') > w(e). Add e to T and remove e', we can get another spanning tree T'. w(T') = w(T) - w(e') + w(e) < w(T). So T is not the minimum spanning tree. proved.

conclusion:

1. think of a component of F as S, then the minimum spanning tree must contains its safe edge. So we can conclude that **minimum spanning tree contains every safe edge**.
2. what's more , **useless edge can't be in the minimum spanning tree**. Because F is a subgraph of the minimum spanning tree, if useless edge(u -> v) is in minimum spanning tree as well, e and the path from v to v in F will form a cycle in the minimum spanning tree. contradict!

so,

1. we can add safe edge in each iteration without violating the invariant.(obviously, F is still a subgraph of MST since safe edge is in MST) (Q1 solved)
2. What's more, if F is not connected, suppose that u is from component 1 and v is from component 2. Because G is connected, there is a path between u and v in G. In this path, there must be an edge with exactly one endpoint in component 1(and this edge is obviously not in F). That is, we can continuously find a safe edge and add it to F only if F is not connected. Because there are only finite edges in G, this process will terminate with a connected F(Q2 solved)

> we've got our **edge-adding strategy**: add safe edge!!

boruvka/sollin's algorithm

```
Boruvka(V, E)
F <- (V, empty edge set)
(count, comp) <- CountAndLabel(F)// count is the number of components F
has and comp[v] is the component id that v lies in
while (count > 1) {
        AddAllSafeEdges(E, F, count, comp)
        (count, comp) <- CountAndLabel(F)
}
return F

_____________________________________________________
CountAndLabel(F) // regular dfs
count <- 0
initialize comp with comp[i] = undefined
for each vertex v in F:
        if comp[v] undefined:
```

```
                dfs(v, count, comp)
                count ++
return (count, comp)

-----------------------------------------------------------------
AddAllSafeEdges(E, F, count, comp)
initialize safe with safe[i] = null // the size of safe is count(the
number of components in F). safe[i] is the safe edge of the i_th
component
for each uv(u < v) in E:
        if (comp[u] != comp[v]): // this edge is a candidate of
comp[u]'s and comp[v]'s safe edge
                if w(uv) < w(safe[comp[u]])):
                        safe[comp[u]] = uv
                if w(uv) < w(safe[comp[v]]):
                        safe[comp[v]] = uv
for each component i:
        // add safe[i]. attention: some safe[i] can share the same edge.
make sure that you don't add them repeatedly
        uv <- safe[i]
        if comp[u] == i:
                if comp[v] > i: //this edge can't have been added to F
yet
                        F.add(u,v)
                else: // this edge may be the comp[v]'s safe edge as
well and have been added
                        if safe[comp[v]] != uv: // has not been added
yet
                                F.add(u,v)
        else:
                // similar to the other branch
```

time complexity:
CountAndLabel: $O(V_F + E_F)$,$E_F \leq V_F - 1$ -> $O(V_F) = O(V)$
AddAllSafeEdges:

1. init safe: O(count)
2. iterate over G‘s edges to find the safe edge for each component: O(E)

3. add safe edges to F: O(count)

thus, each iteration of the while loop in boruvka takes O(V) + O(count) + O(E)
because count <= V <= E + 1 (G is connected , so V <= E + 1), we can conclude that it takes O(E)

Also, once we add a safe edge to F, the number of components decreases by 1. For a F with $count$ components, there are at least $count/2$ distinct safe edges (otherwise, there will be a safe edge shared by at least 3 components, which is impossible). So, after adding all distinct safe edges, the number of components is reduced by at least count/2, i.e. count is reduced by at least a factor of two by each iteration.

Therefore, there are at most log(V) iterations. (count is initially V)

we conclude that the time complexity of the algorithm is O(ElogV)

prim's algorithm

naive version - keep edges in queue

only one nontrivial component T; all the other components are isolated vertices.
repeatedly add T's safe edge to T.

```
PrimMST(G):
// init isInT
for each v in G:
         isInT[v] <- false
 pq <- a priorityqueue the top of which is the egde with the smallest
weight
 s <- an arbitrary vertex
 put (s, s)(weight = 0) to pq
 while pq is not empty: // can early stop when we've added v - 1 edges
          take (p, v) from pq
          if !isInT[v]: // (p,v) is T's safe edge, add it to T
                  isInT[v] = true
                  edgeTo[v] = p
                  for u in G.adj(v):
                          if !isInT[u] // currently, only one endpoint
in T, is candidate of T's safe edge
```

```
                                    put (v, u) to pq
```

Actually, it's whatever-first search in the case when the bag is a priority queue. Thus, the time complexity is O(V + ET) = O(E + ET) = O(ET) = O(ElogE). If we carefully only add one of parallel edges to pq, E will be at most V^2.

Thus, O(ElogE) = O(ElogV).

eager implementation- keep vertices in queue

adding safe edge to T can be considered as adding the vertex closest to T.
every time we add a vertex v to T, we introduce a new 'bridge' to T for the remaining vertices, specifically, for the vertices adjacent to v. Thus, these vertices' distance to T should be updated. and then, we keep on adding the closest vertex to T, until all the vertices are added to T.

for this version of prim algorithm, we can take advantage of **Fibonacci heap** to realize a time complexity O(E + VlogV) (for a binary heapalgorithm which supports decreasekey in logn, it's ElogV)

> Fibonacci heap
>
> 1. insert: amortized 1
> 2. decreasekey: amortized 1
> 3. extractmin: amortized logn

```
ImprovedPrim(G)
s <- an arbitrary vertex
dist[s] = -infinite
edgeTo[s] = s
insert (s, dist[s]) to FibHeap // key: vertex v value: v's distance to T
for other vertex v:
        dist[v] = infinite
        insert (v, dist[v]) to FibHeap // can insert later
while FibHeap is not empty:
        take u from heap
        dist[u] = -infinite
        for v in G.adj(u):
```

```
                if weight(uv) < dist[v]:
                        edgeTo[v] = u
                        dist[v] = weight(uv) // if insert later: if
(dist[v] == infinite) dist[v] = weight(uv); insert v else dist[v] =
weight(uv); decreasekey v
                        decreasekey v
```

initial insert takes O(V)/O(1)
each taking vertex from the heap takes logn and we'll take V times: O(VlogV)
for each taken vertex, we examine corresponding edges and decreasekey/ insert key if needed. it takes O(E) in total

thus, time complexity is O(V + E + VlogV) = O(E + VlogV)

kruskal's algorithm

basic ideas

scan all edges by increasing weight; if an edge is safe, add it to F

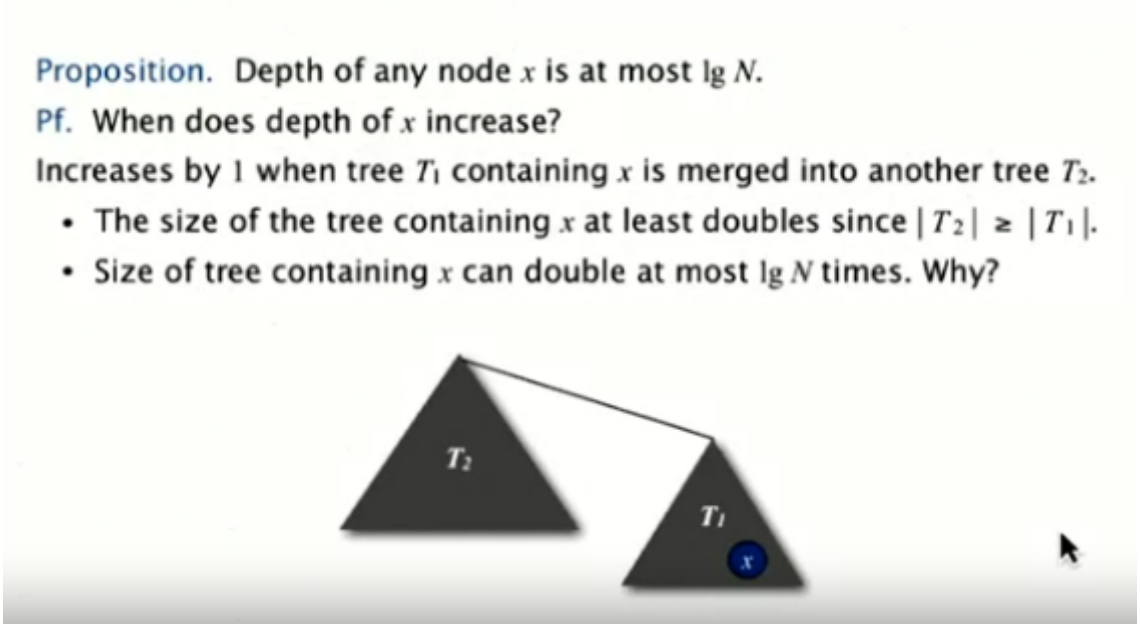
```
Kruskal(G)
sort all edges by weight
uf <- UF(G.V)
F <- (V, empty edge set)
for each edge in edges: // can early stop when we've added v - 1 edges
        if uf.union(edge.u, edge.v): // union u ,v successfully. i.e.
u,v is previously in different components
                add edge to F
return F
```

sort takes O(ElogE) = O(ElogV)
iterate over edges and add safe edge takes O(E + VlogV)/ O(ElogV), depending on how UF is implemented.

thus, the total time complexity is O(ElogV)

Union Found: v1

sedgewick version



thus, the time complexity of union is O(logN)

Union Found: v2

graph version
prop: amortized time complexity is O(1).

proof: for any vertex v, when its component label changes, the size of the component containing v at least doubles since we merge the smaller component into the larger one. initial size * $2^k \leq$ final size $\leq$ N. Thus, k $\leq logN$ i.e. for any vertex v, its component label changes at most logN times.

for union(i, j), if i, j is in the same component, it takes $\theta(1)$; if i and j are in different components, it takes θ (number of the vertices whose component labels change). Since for any v, its component label changes at most logN times, the total time complexity of condition 2 is at most NlogN. Therefore, amortized time complexity is O(1).

Shortest Paths

for an edge weighted directed graph, find the shortest path from a source vertex s to a target vertex t.

Negative Edges

a shortest path from s to t exists if and only if there is at least one path from s to t, but there is no path from s to t that touches a negative cycle.

> negative edges are allowed, but negative cycle is not !!!

All of the algorithms for directed graph also work for undirected graphs with essentially trivial modifications, if and only if **negative edges are prohibited**. Correctly handling negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, **because this would transform any negative edge into a short negative cycle.**

Generic Idea

dist(u)::= length of a tentative path $s \rightsquigarrow u$
u -> v is **tense** if dist(u) + weight(u -> v) < dist(v). If an edge is tense, the tentative path from s to v $s \rightsquigarrow v$ should be updated to a clearly shorter path: $s \rightsquigarrow u \rightarrow v$. It's called **relaxing the edge** $u \rightarrow v$**.**

```
Relax(u, v):
dist[v] = dist[u] + weight(u -> v)
edgeTo[v] = u
```

the generic idea of shortest path algorithms is:

```
FordSSSP(s):
        InitSSSP(S)
        while there is at least one tense edge:
                relax any tense edge

InitSSSP(s):
        dist[s] = 0
        edgeTo[s] = s
        for other veretx v:
                dist[v] = infinity
                edgeTo[v] = null
```

correctness:

1. at first, we need to prove that for any vertex u reachable from s, dist(u) is not infinite. (1)
   proof:
   for any vertex u reachable from s, there is a path T: s = v0 -> v1 ->...->vn = u and dist(v0) = 0. suppose dist(u) is infinite, then there must be an edge vi -> vj s.t. dist(vi)

is not infinite and dist(vj) = infinite -> dist(vi) + 1 < dist(vj) -> vi -> vj is tense, contradict to that there is no tense edges. proved.

2. Then, we prove that if for any edge u -> v, dist(v) $\leq$ dist(u) + weight(u -> v)(the condition when the algorithm halts), then for any dist(t), it's the length of the shortest path from s to t.
   proof:
   let s = v0 -> v1 -> ....-> vn = t be the shortest path T from s to t:
   dist(v1) <= dist(v0) + weight(v0 -> v1)
   dist(v2)<=dist(v1) + weight(v1 -> v2)
   ...
   dist(vn) <= dist(vn-1) + weight(vn-1 -> vn)
   thus, dist(v1) +...+dist(vn) <= dist(v0) + ... + dist(vn-1) + length(T). given (1):
   dist(vn) <= dist(v0) + length(T) i.e. dist(t) <= length(T)
   because T is the shortest path, dist(t) >= length(T). Thus, dist(t) = length(T). proved.
3. Also, if there is no negative cycle, at any moment during execution of the algorithm, dist(v) is either infinite or the length of some simple path from s to v. Thus, if G has no negative cycles, the algorithm eventually halts, because there are only a finite number of simple paths in G.

> there are different instantiations of this generic relaxation algorithm. However, the efficiency and correctness of each instantiation depends on the structure of the input graph.

for the correctness of each algorithm we introduce below, except for giving a self-contained proof, we prove it by showing the algorithm is an instantiation of the generic algorithm as well.

Unweighted Graph: BFS

```
ShortestPath(s, G):
InitSSSP(s)
queue.offer(s)
when queue is not empty:
        u <- queue.poll()
        for u->v in G.adj(u):
                if dist(u) + 1 < dist(v):
                        dist(v) = dist(u) + 1 // 1
```

```
edgeTo[v] = u // 2
queue.offer(v) // 3
```

prop: at any moment during the execution of algorithm, for vertices v0,....vn in queue, dist(v0) <= dist(v1) <= ... <=dist(vn) <= dist(v0) + 1

proof:
we'll prove it by induction.
at the very beginning, there is only one vertex s in queue and the prop is obviously true.
suppose that currently, the prop is true, then after polling v0 from the queue and push a new vertex u s.t. dist(u) = dist(v0) + 1:
if n = 0, there is only one vertex u in the queue and the prop is true
if n > 0 and dist(vn) = dist(v0), for vertices in the queue, i.e. v1,....,vn,u, dist(v0) = dist(v1) <=...<=dist(vn) = dist(v0) <= dist(u) = dist(v0) + 1 = dist(v1) + 1. the prop is true
if n > = and dist(vn) = dist(v0) + 1, for vertices in the queue, i.e. v1, ..., vn, u, dist(v0) <= dist(v1) <= ... <= dist(vn) = dist(v0) + 1 = dist(u)
because dist(v0) <= dist(v1), dist(u) = dist(v0) + 1 <= dist(v1) + 1. Thus, we get dist(v1) <= ... <= dist(vn) <= dist(u) <= dist(v1) + 1. The prop is true.
proved.

correctness-bfs

this prop implies that vertices are polled out of the queue with the dist of which non-decreasing. It follows that for any vertex v, line 1 - 3 are executed **at most once**. (suppose that we are now at the moment when we check whether dist(u) + 1 < dist(v), if for v, line 1 - 3 has been executed, which implies that v has been pushed into the queue. if it has been polled out of the queue, then dist(v) <= dist(u). if it's currently in the queue, then dist(v) <= dist(u) + 1. whatever, dist(v) <= dist(u) + 1. Thus, line 1 - 3 won't be executed again.) Therefore, each vertex is pushed into and polled out of the queue at most once.

Because each vertex is pushed into the queue at most once, the algorithm will halt.

Also, for any reachable v, let s= v0 -> v1 ->....->vl = v be an arbitrary path from s to v, we'll prove that for any vertex vi in this path, dist(vi) <= i(In particular, dist(v) = dist(vl) <= l = length of the path.)
when i = 0, dist(v0) = dist(s) = 0 <= 0.
suppose that dist(vj-1) <= j-1 (j >= 1), we need to prove that dist(vj) <= j. since dist(vj-1) is not infinite, it must have been pushed into the queue. when we polled it out of the queue and check whether dist(vj-1) + 1 < dist(vj), dist(vj) will be assigned dist(vj-1) + 1 or keep its

value which is <= dist(vj-1) + 1. Whatever, dist(vj) <= dist(vj-1) + 1. Since dist(vj-1) is assigned a non-infinite value and pushed into the queue at most once, dist(vj - 1) won't change anymore. On the other hand, dist(vj) won't increase(in fact, it won't change either). Thus, when the algorithm halts, dist(vj) <= dist(vj-1) + 1 <= j - 1 + 1 = j. (this proof can be modified to a version which proves that bfs is essentially an instantiation of the generic algorithm, which relax all the tense edges)
proved.

Thus, for any reachable v and an arbitrary path from s to v, dist(v) <= the length of the path. that is, the length of the shortest path from s to v >= dist(v). and obviously, there is a path T: s -> ... -> edgeto(v) -> v with length dist(v). So, dist(v) is the length of the shortest path and T is such a path.

performance

we can conclude that the time complexity of the algorithm is O(V + E) because each vertex is pushed into the queue at most once.

Moreover, in the correctness part, we proved that when the algorithm halts, dist(u) is the length of the shortest path from s to u. Thus, for each reachable u, dist(u) is not infinite, which implies that u has been pushed into the queue at least once.

It follows that each reachable vertex v is pushed into the queue exactly once(dist(v) and edgeTo is assigned a well-defined value exactly once). Thus, to be more specific, the time complexity is $\theta(V + E)$

Directed Acyclic Graph: DFS-topoordering

it's a dynamic programming program (thus, it's correct)

1. recursive relation:

$$dist(v) = \begin{cases} min_{u_i \to v \in E} dist(u_i) + weight(u_i \to v), v \neq s \\ 0, v = s \end{cases}$$

2. data structure: an array dist of length V
3. evaluation order: topological order of G i.e. the post ordering of reversed G

```
ShortestPath(G, s):
revG <- G.reverse()
init dist
```

```
for each vertex v:
        if dist[v] undefined:
                DFS(v, revG, dist, s)


DFS(v, revG, dist, s):
        if v = s, dist[v] = 0 else dist[v] = infinity
        for each v -> u in revG:
                if dist[u] undefined:
                        DFS(u, revG, dist, s)
                dist[v] = min(dist[v], dist[u] + weight(v -> u))
```

or

```
ShortestPath(G, s):
for all vertices v in topological order:
        if v = s:
                dist[v] = 0
        else:
                dist[v] = infinity
        for each u -> v:
                if dist[u] + weight(u -> v) < dist[v]:
                        dist[v] = dist[u] + weight(u -> v)
```

time complexity: O(V + E)

relation with generic algorithm

we can modify the second version a bit and see clearly it's essentially an instantiation of our generic algorithm

```
ShortestPath(G, s):
InitSSSP(s)
for all vertices v in topological order:
        for each v -> u:
                if v -> u is tense:
                        Relax(v, u)
```

for an arbitrary edge v -> u in the graph , it will be check whether it's tense and be relaxed in need once. After the check and the necessary relaxation, the edge is not tense: dist(v) + weight(v -> u) >= dist(u). since dist(v) won't change anymore(because it may only be changed when we examine its topo-ordering predecessors, which won't be examined again) and dist(u) won't increase. the edge is still not tense when the algorithm halts. Therefore, by this algorithm, we relax all tense edges until there is no tense edge. it's an instantiation of our generic algorithm and it's correct.

Dijkstra's Algorithm

```
ShortestPath(G, s):
InitSSSP(s)
pq.offer(s)
while pq is not empty:
        u <- pq.poll()
        for each u -> v:
                if u -> v is tense:
                        Relax(u, v)
                        if v in pq:
                                pq.decreaseKey(v, dist[v])
                        else:
                                pq.insert(v, dist[v])
```

(replace pq with a queue, we'll get bfs!!)
works for **negative weights.**

correctness

prop: at all times during the execution of the algorithm, if there is an outgoing edge of u: u -> v is tense, u is either in the pq or is the vertex most recently polled out from the pq.
proof:
suppose that u is neither in the pq nor the vertex most recently polled out:

1. u has never been pushed into pq -> dist(u) = infinite -> none of its outing edge is tense. contradict.
2. u has been pushed into pq and polled out. At the last time when u was polled out of pq, all of its outgoing edges would have been relaxed, i.e. for an arbitrary u -> v, dist(u) + weight(u -> v) >= dist(v). Because dist(v) won't increase, u -> v may become

tense only if dist(u) decrease after this last polling out. since if dist(u) decreases, u is either already in the pq or will be inserted into the pq, given the fact that u currently is not in pq, dist(u) don't decrease after u's last polling out. Therefore, all outgoing edges of u keep relaxed until now. contradict.
proved.

Thus, Dijkstra's algorithm is actually an instantiation of the generic algorithm. It's correct, provided that there are no negative cycles (negative edge is okay!)

if there are no **negative edges** in G, we can derive a **self-contained proof** of correctness for Dijkstra's algorithm.

let ui be the ith vertex polled out of the pq, di be the value of dist(ui) just after ui is polled out. u0 = s and d0 = 0.
prop: $d_i <= d_{i+1}$
proof:
just after ui is polled out, for any vertex v in pq, dist(v) >= di.
after examining all the outgoing edges of ui and relaxing them if necessary, dist of some vertices v previously in pq are updated to dist(ui) + weight(ui -> v) and some new vertices are inserted into pq the dist value of which is dist(ui) + weight(ui -> v). Because there are no negative edges, dist(ui) + weight(ui -> v) >= dist(ui). Thus, it's still true that for any vertex v in pq, dist(v) >= di. u_{i+1} is polled out of this pq and obviously, $d_{i+1} \geq d_i$.

Thus, for any j > i, dj >= di.

prop: each vertex v is polled out at most once.
proof: when ui = v is polled out of pq, suppose that it has been polled out previously, i.e. there is some j s.t. j < i and uj = ui = v. That is, just after uj = v is polled out, dist(v) = dj. when v is reinserted into pq after uj = v is polled, dist(v) must have been updated to a value < dj. so di < dj. contradict to the fact that for any i > j, di >= dj. proved.

the rest of the correctness proof is similar to that of bfs algorithm.Algorithms by Jeff > correctness-bfs

performance

In the correctness part, we proved that when the algorithm halts, dist(u) is the length of the shortest path from s to u. Thus, for each reachable u, dist(u) is not infinite, which implies that u has been inserted into pq at least once. And for those u that's not reachable, dist(u) is infinite, indicating that they are never inserted into pq.
when we analyze the performance, only take those reachable vertices into account.

no negative edges

combined with the fact that each vertex is polled out at most once, each reachable vertex is inserted/polled out exactly once. it follows that each edge is examined exactly once. Therefore, the time complexity is O(VT_{poll} + E + E$max(T_{insert}, T_{decreaseKey})$). If we use a standard binary heap algorithm, it will be O(VlogV + E + ElogV) = O((V +E)logV) . If we use a fibonacci pq, it will be O(VlogV + E).

negative edges

For graphs without negative cycles, but no other restrictions on edge weights, the worst-case running time of D is actually **exponential**.

Bellman-Ford Algorithm

```
ShoretestPath(G,s):
        InitSSSP(s)
        i <- 0
        while (true) {
                i ++
                relaxed <- false
                for every edge u -> v:
                        if u -> v is tense:
                                Relax(u, v)
                                relaxed <- true
                // after i_th iteration
                if not relaxed:
                        break
        }
```

this algorithm is obviously an instantiation of the generic algorithm and is correct.

we'll give a self-contained correctness proof below.

correctness

let $dist_{\leq i}(u)$ be the length of the shortest path from s to u consisting of at most i edges.
prop: after i_{th} iteration, for any vertex v, dist(v) <= $dist_{\leq i}(v)$
proof: we prove it by induction on i.
base case: i = 0, for v = s, $0 = dist(s) \leq dist_{\leq 0}(s) = 0$. for $v \neq s$, dist(v) = ∞,

$dist_{\leq 0}(v) = \infty$, $dist(v) \leq dist_{\leq 0}(v)$.
induction: suppose that after j-1 th iteration (j >= 1), for any vertex, dist(v) <= $dist_{\leq j-1}(v)$. after jth iteration, for v = s, dist(s) = 0, $dist_{\leq j}(s) = 0$, dist(s) $\leq dist_{\leq j}(s)$. for v $\neq s$, let T s = v0 -> v1 -> ... -> v_{t-1} -> vt = v (t <= j) be the shortest path from s to v consisting of at most j edges. if there is no such a path, $dist_{\leq j}(v) = \infty$, dist(v) $\leq dist_{\leq j}(v)$. Otherwise, s = v0 -> ...->v_{t-1} is the shortest path from s to v_{t-1} consisting of at most j - 1 edges, the length of which is $dist_{\leq j-1}(v_{t-1})$. we know that after j-1 th iteration, $dist(v_{t-1}) \leq dist_{\leq j-1}(v_{t-1})$. in j th iteration, after relaxing $v_{t-1} -> v_t$ if necessary, dist(vt) <= dist(vt-1) + weight(vt-1 -> vt) <= $dist_{\leq j-1}(v_{t-1})$ + weight(vt-1 -> vt) = $dist_{\leq j}(v_t)$ because dist(vt) won't increase, after jth iteration, dist(v) <= $dist_{\leq j}(v)$. proved.

Thus, after V-1 th iteration, for any vertex v, $dist(v) \leq dist_{\leq V-1}(v)$.

Also, we can prove that if there is no negative cycle, $dist_{\leq V-1}(v)$ is the length of the shortest path: There must be a simple path T the shortest path from s to v (let T' be a shortest path with cycle, then the cycle must have length >= 0 and we can remove the cycle to get a simple path with the shortest length). T has at most V - 1 edges. Then length(T) >= $dist_{\leq V-1}(v)$. And obviously, $dist_{\leq V-1}(v)$ >= length(T). Thus, $dist_{\leq V-1} = length(T)$.

Then, we can conclude that if there is no negative cycles, after V - 1 th iteration, for any vertex, $dist(v) \leq lenOfShortestPath(v)$ on the other hand, dist(v) is the length of a simple path from s to v -> $dist(v) \geq lenOfShortestPath(v)$. It follows that after V - 1 iteration, dist(v) is the length of the shortest path and s -> ... -> edgeTo(v) -> v is the shortest path, if there is no negative cycle.

since dist(v) is the length of the shortest path, there should be no tense edges anymore.

Therefore, for a graph without negative cycles, at most at Vth iteration, relaxed will be false and the algorithm will halt with dist storing the shortest path length.

Said differently, if at Vth iteration, we still relax some edges, it indicates that there is a negative cycle. we can modify the original version to ensure that it will report the negative cycle instead of looping infinitely.

```
ShoretestPath(G,s):
        InitSSSP(s)
        i <- 0
        while (true) {
                i ++
```

```
                relaxed <- false
                for every edge u -> v:
                        if u -> v is tense:
                                Relax(u, v)
                                relaxed <- true
                // after i_th iteration
                if not relaxed:
                        break
                else:
                        if i == V:
                                report negative cycle!
                                break
        }
```

performance

time complexity: O (VE)

Moore's version

```
  ShortestPath(G, s):
        InitSSSP(s)
        q.offer(s)
        // q is in the end of 0_th phase
        phase = 0
        while q is not empty and phase < V:
                repeat q.size:
                        u <- q.poll()
                        for each u -> v:
                                if u -> v is tense:
                                        Relax(u, v)
                                        if v is not in q:
                                                q.offer(v)
                phase ++ // in the end of phase
        if q is not empty:
                report negative cycle
```

q maintains vertices whose outgoing edges might be tense. when the queue is empty, there is no vertex that has at least one tense outgoing, i.e. all the edges are not tense. Thus, it's an instantiation of the generic algorithm and is correct.

we'll provide a self-contained proof for correctness below.
prop: in the end of i*th phase, $dist(v) \le dist*{\le i}(v)*
$proof : basecase : similartostandardbellmaninduction : supposethatitheendofj - 1thphase(j >$
. in the end of jth phase, for v = s, dist(s) = 0, $dist_{\le j}(s) = 0$, dist(s) $\le dist_{\le j}(s)$. for v $\neq s$, let T s = v0 -> v1 -> ... -> v_{t-1} -> vt = v (t <= j) be the shortest path from s to v consisting of at most j edges. if there is no such a path, $dist_{\le j}(v) = \infty$, dist(v) $\le dist_{\le j}(v)$. Otherwise, s = v0 -> ...->v_{t-1} is the shortest path from s to v_{t-1} consisting of at most j - 1 edges, the length of which is $dist_{\le j-1}(v_{t-1})$. we know that in the end of j-1 phase, $dist(v_{t-1}) \le dist_{\le j-1}(v_{t-1})$. denote the value of dist(v_{t-1}) in the end of j - 1 phase with $dist_{j-1}(v_{t-1})$, $dist_{j-1}(v_{t-1}) \le dist_{\le j-1}(v_{t-1})$. if vt-1 in queue in the end of j-1 phase, after necessary relaxation of vt-1 -> vt, $dist_{j-1}(v_{t-1}) + weight(v_{t-1} -> v_t) \ge dist_{poll\ v_{t-1}}(v_{t-1}) + weight(v_{t-1} -> v_t) \ge dist_j(v_t)$. if vt-1 is not in queue, then vt-1 -> vt is not tense in the end of j - 1 phase, i.e. $dist_{j-1}(v_{t-1}) + weight(v_{t-1} -> v_t) \ge dist_{j-1}(v_t) \ge dist_j(v_t)$. Thus, $dist_j(v) = dist_j(v_t) \le dist_{j-1}(v_{t-1}) + weight(v_{t-1} -> v_t) \le dist_{\le j-1}(v_{t-1}) + weight(v_{t-1} -> v_t)$ proved.

similar to the proof of standard bellman-ford, this indicates that in the end of V - 1 phase, dist(v) is the length of the shortest path and there is no tense edges, given that there is no negative cycle. thus, at most in the end of V phase, there will be no vertex in the queue and the algorithm halts, if there is no negative cycle.

performance: O((V + E)V)

dp version

$$dist_{\le i}(v) = \begin{cases} 0, v = s \\ \infty, i = 0, v \neq s \\ min_{u \to v \in E}\{dist_{\le i-1}(u) + weight(u \to v)\}, otherwise \end{cases}$$

correctness:
i = 0 case is trivial. for otherwise case:
let T_j be an arbitrary path from s to v consisting of at most i edges: s = v_{j_0} -> ... ->v_{j_t} = v, t <= i.
$dist_{\le i}(v) = min_j\ len(Tj) = min_j\ len_j(s \rightsquigarrow v_{j_{t-1}}) + weight(v_{j_{t-1}} -> v) = min_{u->v \in E}\{min_{j_{t-1}=u} le$
let $T_j(u)$ (u -> v $\in$ E)be an arbitrary path from s to u consisting of at most i - 1 edges, $T_j(u) -> v$ is a path from s to v consisting of at most i edges.

$dist_{\le i}(v) \le min_{u \to v \in E}\{min_j T_j(u) + weight(u \to v)\} = min_{u \to v \in E}\{dist_{\le i-1}(u) + weight(u \to v)\}$
thus, $dist_{\le i}(v) = min_{u \to v \in E}\{dist_{\le i-1}(u) + weight(u \to v)\}$

use a 2D array dist(i, v) to store solutions of subproblems. evaluate it from top to bottom. dist(V - 1, v) is the length of the shortest path from s to v.

```
ShortestPath(G, s):
for i in 0 until V:
        dist[i, s] = 0 // others are infinity
for i in 1 until V:
        for each vertex v:
                for each u -> v:
                        if dist[i - 1, u] + weight(u -> v) < dist[i, v]
                                dist[i, v] = dist[i - 1, u] + weight(u -
> v)
```

modify it a bit:

```
ShortestPath(G, s):
for i in 0 until V:
        dist[i, s] = 0 // others are infinity
for i in 1 until V:
        for each edge u -> v:
                if dist[i - 1, u] + weight(u -> v) < dist[i, v]
                        dist[i, v] = dist[i - 1, u] + weight(u -> v)
```

time complexity: $\theta(VE)$

Summary

no negative cycles!!

	negative edges(has cycle)	**negative edges(no cycle)**	**no negative edges(equal weight)**	**no negative edges**
bfs	not aplicable	not applicable	$\theta(V+E)$	not applicable
dfs	not applicable	$\theta(V+E)$	no cycle: $\theta(V+E)$	no cycle: $\theta(V+E)$

	negative edges(has cycle)	**negative edges(no cycle)**	**no negative edges(equal weight)**	**no negative edges**
Dijkstra(1)	exponential	exponential	**O(VlogV + E + V) = O(VlogV + E)(vertex is inserted once and will never be decreasedkey. each insert takes a constant time. each poll takes O(logV) time)**/ fib pq: O(VlogV + E)	O((V + E)logV)/ fib pq: O(VlogV + E)
bellman-ford(2)	$O(VE)$	$O(VE)$	$O(VE)$	$O(VE)$
moore bellman(3)	O((V + E)V)	O((V + E)V)	O((V + E)V)	O((V + E)V)
dp	$\theta(VE)$	$\theta(VE)$	$\theta(VE)$	$\theta(VE)$

(1): won't halt if there is negative cycles
(2),(3): can detect negative cycles

Applications

parallel job scheduling

given a set of jobs with durations and precedence constraints, schedule the jobs so as to achieve the minimum completion time, while respecting the constraints(given that you can do 2 jobs simultaneously)
solution: construct a graph

1. add source vertex and sink vertex
2. 2 vertices for each job(start and end)
3. 3 edges for each job: duration, source to start, end to sink
4. one edge for each precedence constraint(0 weight)
   find the longest path from source to sink(prove the longest path distance is the earliest timestamp by induction on topoordering)

arbitrage detection

edges are -ln(exchange rate from currency v to w);
multiplication of exchange turns to addition

we'll try to find a negative cycle

All-Pairs Shortest Paths

no negative cycle in the graph

Johnson's Algorithm

Basic idea

```
APSP(V, E, w):
        for every vertex s
                dist[s,.] = SSSP(V, E, w, s)
```

the time complexity depends on the time complexity of SSSP. Therefore, we need to find the most efficient SSSP algorithm. By Algorithms by Jeff > Summary, modifying the original graph to make it get rid of negative edges without changing the shortest path between an arbitrary pair of s and t, then running Dijkstra on the modified graph to find the shortest path is a good choice.

Get Rid of Negative Edges

how to get rid of negative edges?
$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$
$\pi(u)$ is a price associated with vertex v.

then for any path $s \rightsquigarrow t$, $length' = \pi(s) + length - \pi(t)$. Because all the paths from s to t change length by exactly the same amount, the shortest path from s to t with the new weight is the same as the shortest path from s to t with the original weight.

how to define $\pi(v)$ for every vertex v?

Algorithm

1. use bellman-ford to calculate dist(s, v)
2. $\pi(v) = dist(s, v)$. Because there is no tense edge, i.e.
   $dist(s, u) + weight(u \to v) \geq dist(s, v)$,
   $weight'(u \to v) = dist(s, u) + weight(u \to v) - dist(s, v) \geq 0$

to ensure that $dist(s, u)$ is well-defined, that is, it's not infinity, we need find a s can reach every vertex in the graph. by adding a dummy vertex s with zero-weight edges to every

vertices in the graph, we can always find such a vertex.

```
JohnsonAPSP(G)
        // add dummy vertex
        add vertex s
        for each vertex v in V:
                add s -> v (weight = 0)
        // calculate pi, there are E + V edges now
        pi = bellman(G, s)
        update weight with pi
        // run Dijkstra to find shortest path
        for each vertex u except s:
                dist[u, .] = Dijkstra(u)
                for each vertex v except s:
                        dist[u, v] <- dist[u, v] + pi[v] - pi[u]
```

time complexity is V + V(E + V) + E + V + V(V + E)logV + VV = V(V + E)logV

Dynamic Programming

$dist(u, v, l)$ ::= the shortest path from u to v consisting of at most l edges

$$dist(u,v,l) = \begin{cases} 0, u = v \\ \infty, u \neq v, l = 0 \\ min_{s \to v \in E}(dist(u, s, l-1) + weight(s \to v)), otherwise \end{cases}$$

$dist(u, v, V-1)$ is what we want.

```
DpAPSP(G):
for each vertex u:
        for each vertex v:
                if (u != v) dist[u, v, 0] = infty else dist[u, v, 0] = 0
for l from 1 to V - 1:
        for each vertex u:
                for each vertex v:
                        dist[u, v, l] = dist[u, v, l- 1]
                        for each s -> v:
                                if (dist[u, s, l - 1] + weight(s -> v) <
dist[u, v, l]):
```

```
                                        dist[u, v, l] = dist[u, s, l -
1] + weight(s -> v)
```

it can be modified to a more bellman-like version as we do to the dp version of SSSP

```
DpAPSP(G):
for l from 0 to V - 1:
        for each vertex u:
                for each vertex v:
                        if (u != v) dist[u,v,l] = infty else dist[u, v,
l] = 0
for each vertex u:
        for l from 1 to V - 1:
                for each edges s -> v:
                        if (dist[u, s, l - 1] + weight(s -> v) < dist[u,
v, l]):
                                dist[u, v, l] = dist[u, s, l - 1] +
weight(s -> v)
```

Thus, this algorithm is essentially run dpSSSP for each vertex. time complexity is $O(VEV) = O(V^2E)$

Divide and Conquer

for any path from s to t consisting of at most $2l$ edges($l \geq 1$),pick the midpoint as v, it's in P = $\bigcup_{v \in V}$ {path from s to v consisting of at most l edges + path from v to t consisting of at most l edges}. and for each path in P, it's a path from s to t consisting of at most 2l edges. Therefore, $dist(s,t,2l)$ = $min_{v \in V}(dist(s,v,l) + dist(v,t,l))$
we can develop a recursive relation as below:

$$dist(s,t,2^i) = \begin{cases} 0, s = t \\ weight(s \to t), s \neq t, i = 0 \\ min_{v \in V}(dist(s,v,2^{i-1}) + dist(v,t,2^{i-1})), s \neq t, i > 0 \end{cases}$$

what we want is $dist(s,t,2^{i*})$ s.t. 2^{i*} >= V - 1. pick $\ulcorner logV \urcorner$ as $i*$ and store $dist(s,t,2^i)$ in dist[s,t,i], we get a more efficient dynamic programming algorithm:

```
DivideAndConquer(G):
for each edge u -> v:
        dist[u,v,0] = weight(u -> v)
for each vertex u:
        dist[u,u,0] = 0
//others are infty
for i in 1 to ceil(logV):
        for each vertex u:
                for each vertex v:
                        dist[u, v, i] = dist[u, v, i - 1]
                        for each vertex s:
                                if (dist[u, s, i - 1] + dist[s, v, i -
1] < dist[u, v, i]):
                                        dist[u,v,i] = dist[u, s, i - 1]
+ dist[s, v, i - 1]
```

the time complexity is $O(V^3 logV)$

Floyd-Warshall algorithm

$dist(u, v, r)$ ::= the shortest path from u to v passing through vertices indexed at most r vertices in graph are indexed from 1 to V. dist(u,v,V) is what we want.

for any r > 0, the path from u to v passing through vertices indexed at most r either contains vertex r or not. for those paths that don't contain r, the shortest distance is dist(u, v, r - 1). for those paths that contain r, they can be divided into 2 subpath: a path from v to r passing through vertices indexed at most r - 1 and a path from r to v passing through vertices indexed at most r - 1. The shortest distance is dist(u, r, r - 1) + dist(r, v, r - 1).
Finally, we get that
$dist(u, v, r) = min(dist(u, v, r - 1), dist(u, r, r - 1) + dist(r, v, r - 1))(r > 0)$

the complete recursive relation is

$$dist(u, v, r) = \begin{cases} 0, u = v \\ weight(u \to v), u \neq v, r = 0 \\ min(dist(u, v, r - 1), dist(u, r, r - 1) + dist(r, v, r - 1)), u \neq v, r > 0 \end{cases}$$

the algorithm is

```
KleeneAPSP(G):
for each vertex u:
        for each vertex v:
                if (u = v) dist[u,v,0] = 0 else dist[u,v,0] = weight(u -
> v)
for r in 1 to V: // can rephrase it to for each vertex r
        for each vertex u:
                for each vertex v:
                        dist[u,v,r] = dist[u,v,r-1]
                        if u != v && dist[u, r, r - 1] + dist[r, v, r -
1] < dist[u,v,r]:
                                dist[u,v,r] = dist[r,v,r-1] +
dist[u,r,r-1]
```

the time complexity is $O(V^3)$

Maximum Flows and Minimum Cuts

an edge-weighted digraph(flow network), source vertex s, and target vertex t. each edge has a positive capacity.
def:

1. flow of the flow network: net flow of s, denoted with $|f|$
2. capacity of cut(A,B)(A is the set including s): sum of capacity of edges pointing from a vertex in A to a vertex in B, denoted with $||A, B||$
   lemma: for any flow f, for any cut(A,B)(A is the set including s), the net flow of the cut = $|f|$
   proof: by induction on the size of A and flow equilibrium of each vertex

prop: for any flow f and any cut(A,B), $|f|$ <= $||A, B||$
proof: $|f|$ = net flow of cut(A,B) = sum of outgoing flows - sum of incoming flows <= $||A, B||$

then, if there is a cut^* and a flow f^* s.t. $|f^*| = ||cut^*||$. f^* is the max flow and the cut^* is the min cut
proof: for any flow f, $|f|$ <= $||cut^*||$ = f^* ; for any cut, $||cut||$>= f^* = $||cut^*||$

fortunately, we can find such a cut and . This is the famous **Maxflow-Mincut Theorem**

> In every flow network with source s and target t, the value of the maximum(s,t)-flow is equal to the capacity of the minimum(s,t)-cut

proof:
augmenting path:
residual capacity:

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

augmenting path P is a undirected path connected s and t: s = v0 - v1 -... - vn = t s.t. $c_f(v_i \to v_{i+1}) > 0, v_i \neq v_j$

we prove the theorem by proving these three conditions are equivalent

1. there exists a cut whose capacity equals the value of flow f
2. f is a max flow and the cut is the min cut
3. there is no augmenting path with respect to f
   (**augmenting path theorem**: a flow f is a maxflow iff no augmenting paths)
   $1 \to 2$
   proved
   $2 \to 3$
   by proving its contrapositive, if there is an augmenting path, we can increase flow by sending flow along the augmenting path.
   $F = min_{0 \leq i < n}\ c_f(v_i \to v_{i+1}) > 0$

   $$f'(u \to v) = \begin{cases} f(u \to v) + F, u - v\ in\ P \\ f(u \to v) - F, v - u\ in\ P \\ f(u \to v), otherwise \end{cases}$$

4. $0 \leq f'(u \to v) \leq c(u \to v)$:
   1. u - v in P, let $u = v_i, v = v_{i+1}$,

      $$\begin{aligned} 0 \leq f'(u \to v) &= f(u \to v) + F \\ &= c(u \to v) - c_f(u \to v) + F \\ &= c(u \to v) - c_f(v_i \to v_{i+1}) + min_{0 \leq i < n}\ c_f(v_i \to v_{i+1}) \\ &\leq c(u \to v) - c_f(v_i \to v_{i+1}) + c_f(v_i \to v_{i+1}) \\ &= c(u \to v) \end{aligned}$$

   2. v - u in P, let $v = v_i, u = v_{i+1}$

$$
\begin{aligned}
c(u \to v) \geq f(u \to v) > f'(u \to v) &= f(u \to v) - F \\
&= c_f(v \to u) - F \\
&\geq c_f(v_i \to v_{i+1}) - c_f(v_i \to v_{i+1}) \\
&= 0
\end{aligned}
$$

3. otherwise, $c(u \to v) \geq f(u \to v) = f'(u \to v) \geq 0$

5. for any vertex v except s, t, equilibrium is not violated:
   $\sum_{u \to v \in E} f(u \to v) = \sum_{v \to w \in E} f(v \to w)$
   1. for those vertices not in P,
      $\sum_{u \to v} f'(u \to v) = \sum_{u \to v} f(u \to v) = \sum_{v \to w \in E} f(v \to w) = \sum_{v \to w \in E} f'(v \to w)$
   2. otherwise, let v = v_i(0 < i < n), if $v_{i-1} \to v_i \in E$ and $v_i \to v_{i+1} \in E$,

$$
\begin{aligned}
\sum_{u \to v \in E} f'(u \to v) &= \sum_{u \to v \in E \text{ and } u \neq v_{i-1}} f(u \to v) + f'(v_{i-1} \to v_i) \\
&= \sum_{u \to v \in E \text{ and } u \neq v_{i-1}} f(u \to v) + f(v_{i-1} \to v_i) + F \\
&= \sum_{u \to v} f(u \to v) + F
\end{aligned}
$$

$$
\begin{aligned}
\sum_{v \to w \in E} f'(v \to w) &= \sum_{v \to w \in E \text{ and } w \neq v_{i+1}} f(v \to w) + f'(v_i \to v_{i+1}) \\
&= \sum_{u \to v \in E \text{ and } u \neq v_{i-1}} f(u \to v) + f(v_i \to v_{i+1}) + F \\
&= \sum_{v \to w} f(v \to w) + F
\end{aligned}
$$

   then $\sum_{u \to v \in E} f'(u \to v) = \sum_{v \to w \in E} f'(v \to w)$
   similarly, for other combinations of the existence of edge $v_i \to v_{i+1}$, $v_{i+1} \to v_i$, $v_{i-1} \to v_i$, $v_i \to v_{i-1}$, we can prove the equilibrium is maintained

6. $|f'| > |f|$
   1. $v_0(s) \to v_1 \in E$:

$$
\begin{aligned}
|f'| &= \sum_{s \to w \in E} f'(s \to w) - \sum_{u \to s \in E} f'(u \to s) \\
&= \sum_{s \to w \in E \text{ and } w \neq v_1} f(s \to w) + f(s \to v1) + F - \sum_{u \to s \in E} f(u \to s) \\
&= |f| + F \\
&> |f|
\end{aligned}
$$

   2. $v_1 \to v_0(s) \in E$:

$$
\begin{aligned}
|f'| &= \sum_{s \to w \in E} f'(s \to w) - \sum_{u \to s \in E} f'(u \to s) \\
&= \sum_{s \to w \in E} f(s \to w) - \sum_{u \to s \in E \; and u \neq v_1} f(u \to s) - f(v_1 \to s) + F \\
&= |f| + F \\
&> |f|
\end{aligned}
$$

$3 \to 1$
let (A, B) be a cut where A is the set connected to s by an undirected path with no full forward or empty backward edge(vertex to which there is an augmenting path from s . because there is no augmenting path, t can't be in A, (A,B) is a valid cut). then for such a cut, any outgoing edge u -> v must be full, otherwise, there is a undirected path from s to u with no full forward or empty backward edge : P, P + u -> v is an undirected path from s to v with no full forward or empty backward edge, v is in A. contradict. similarly, any incoming edge must be empty.
then |f| = net flow of (A,B) = outgoing flow - incoming flow =||A,B||

if edge capacity is integers between 1 and U, then we are confident that ford-fulkerson algorigthm will terminate because flow <= sum of edge capacities and is always a integer. once we find an augmenting path, flow increases at least by 1. since the value of flow is upper bounded, we can't always find an augmenting path and the algorithm will terminate.

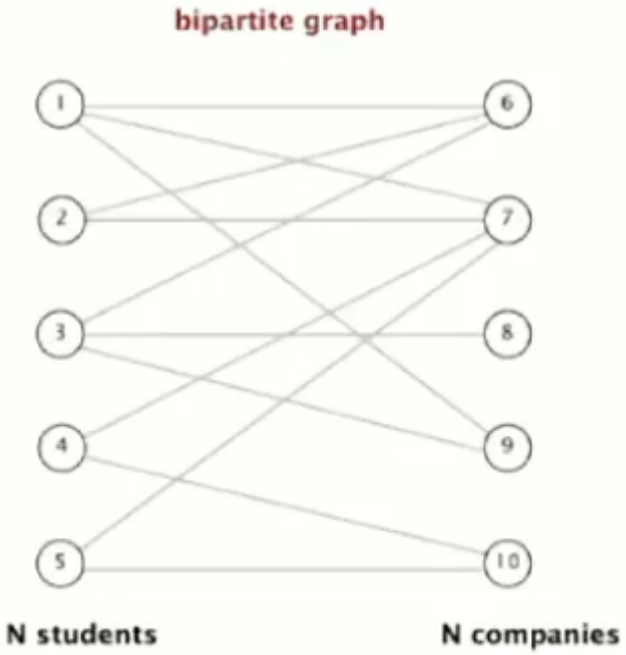
residual network:
if $c_f(u \to v) > 0$, then there is an edge $u \to v$ in residual network
augmenting path in original network is equivalent to directed path in residual network

Applications

Bipartite matching

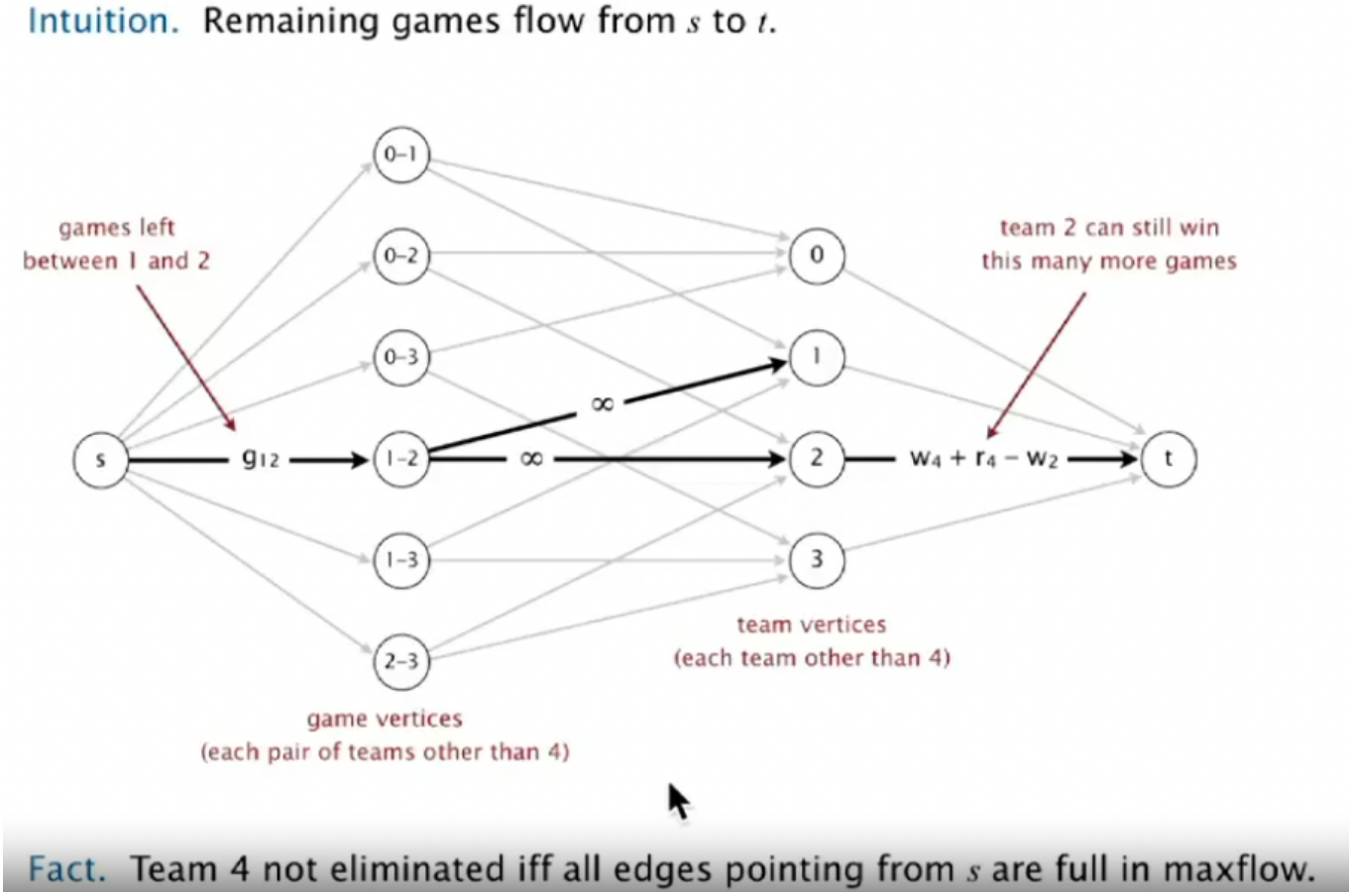
Bipartite matching problem

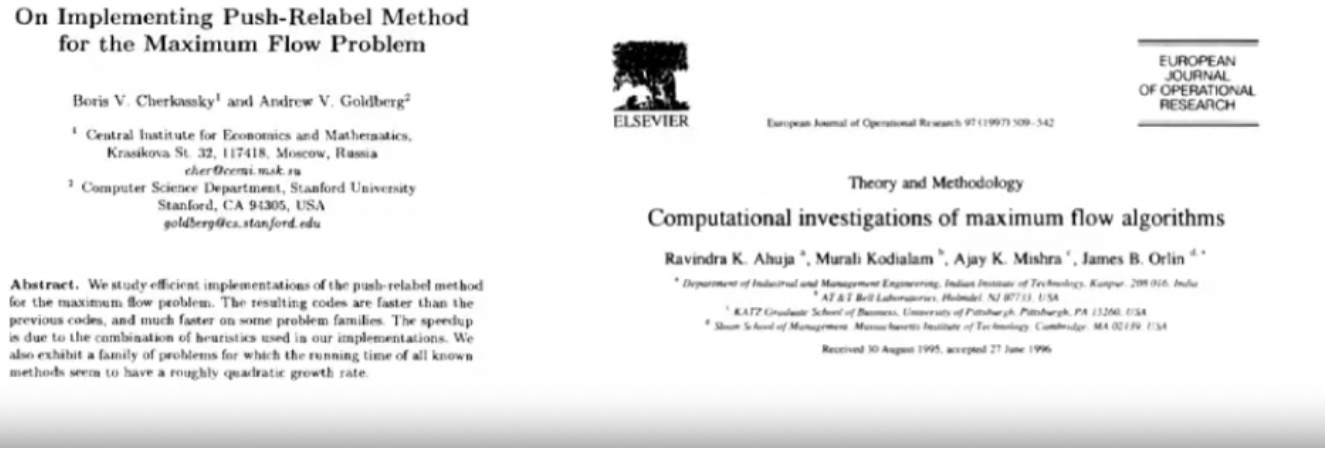
Given a bipartite graph, find a perfect matching.



given min cut (A,B) students in A can only be matched to companies in A.

baseball elimination





On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky[1] and Andrew V. Goldberg[2]

[1] Central Institute for Economics and Mathematics, Krasikova St. 32, 117418, Moscow, Russia
cher@cemi.msk.su
[2] Computer Science Department, Stanford University Stanford, CA 94305, USA
goldberg@cs.stanford.edu

Abstract. We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.

ELSEVIER

European Journal of Operational Research 97 (1997) 509–542

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

Theory and Methodology

Computational investigations of maximum flow algorithms

Ravindra K. Ahuja [a], Murali Kodialam [b], Ajay K. Mishra [c], James B. Orlin [d,*]

[a] Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur 208 016, India
[b] AT & T Bell Laboratories, Holmdel, NJ 07733, USA
[c] KATZ Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA
[d] Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Received 30 August 1995, accepted 27 June 1996

teams in mincut (C) union teams that have won more times than the the possible maximum times the target team can win(maxWin = $w_{target} + r_{target}$)(M) is R - i.e. R = C $\cup$ M

certificate of elimination:

let C = {t1, t1, ..., tk}, because any edges pointing from mincut must be full, edge t_i -> t is full($1 \leq i \leq k$). That is, after some games between teams neither of which is target team finished, team t_i would win capacity(t_i -> t) = maxWin - w_i times out of all these games. And the total win times of these teams = $\sum_{i=1}^{k} maxWin - w_i + w_i$ = k maxWin.

Moreover, there must be an edge pointing from s to a game node i - j s.t. i and j both in C is not full.

proof: Suppose all such edges are full, then there must be an edge pointing from s to a game node k - t s.t. either k or t is not in C not full (otherwise, all the edges pointing from s are full, for any team node, there can't be an augmenting path from s to the team node, which implies that mincut contains no team nodes, contradict). and we know that edge k - t -> k is not full and k-t -> t is not full, then k and t are both in C (path s -> k - t -> k, s -> k-t -> t are paths that contains neither full forward edge nor empty backward edge), contradict to the fact that at least one of k and t is not in C. proved.

such an edge indicates that there are still capacity - flow > 0 games between team i and team j (i and j are both in C) remaining. after these games are finished, the total win times of teams in C increase by capacity - flow to kmaxWin + capacity - flow > kmaxWin.

for each team in S \ C, after games between teams (not target team) finished, it must have won at least w_i > maxWin. denote the number of such teams with n, the total win time of these teams > nmaxWin

then the total win time of S > (n + k) maxWin. at least one team in S win >= total win time of S/(n + k) > maxWin times, which indicates that target team can be eliminated.

String

Radix: denoted with R, is the number of possible values of the key

Sort

Key-indexed count sorting

```
KeyCountSort(a):
count <- an array of length R + 1
for each element e in a:
        count[e.key + 1] ++ // count[i]::= the frequency of key i - 1
for key in 0 to R - 1:
        count[key + 1] <- count[key] + count[key + 1]// count[i]:: the
cumulative frequency from key 0 to key i - 1. Moreover, it can be
interpreted as the index to insert an element with key i.
for each element e in a:
```

```
        aux[count[e.key] ++] = e
copy aux to a
```

sort algorithms

LSD Radix Sorting

least significant digit first sorting

1. consider characters from right to left
2. use key count sorting to sort characters

not just for string, but also for any fixed width elements. for example, it can also be used to sort 32-bit integers

Correctness

prop: after sorting the i_th character, substrings from i are sorted stably. in particular, if i = 0, the strings are sorted
proof: by induction on i
base case: i = n - 1, key count sorting stably sorts substrings from n - 1
induction: suppose that substrings from i + 1 are sorted, then after sorting i_th character using key count sorting, if str1[i] $\neq$ str2[i], key count sorting puts them in proper relative order. if str1[i] = str2[i]: if substring1[i + 1] < substring2[i + 1], since substrings from i + 1 are sorted, str1 is put before str2 when we starts sorting the i_th character and after the stable sorting of i_th character, str1 is still put before str2. similarly, we can prove that when substring1[i + 1] is equal to or larger than substring2[i + 1], str1 and str2 are ordered primarily by the i_th character, secondarily by substring[i + 1], and finally by their original position. That is, substring from i is sorted stably.

Performance

$\theta(WN)$

MSD radix sorting

most significant digit first

1. partition array into R parts by first character
2. recursively sort strings in each part

potential for disastrous performance

1. much too slow for small subarrays: each function call needs its own count array
2. huge number of small subarrays because of recursion

solution: cut off to insertion sort

performance: O(WN) random: $Nlog_RN$

3-way radix quicksort

1. 3 - way partition
2. for less part and more part, recursively sort the entire string. for the equal part, recursively sort substring from i + 1

manber-myers

suffix array

sort string suffix array. and we can find the occurrence of a specific substring by doing binary search on the sorted suffix array

longest repeat substring

1. suffix array: sort and suffix array, and calculate the longest common prefix of neighboring elements in the sorted suffix array to find out the longest repeated substring
2. brute force: i, j is the start position of the repeated substrings. brute force all possible i,j pairs to find those who have the longest common prefix

how to sort the suffix array efficiently?
manber-meyer

1. phase 0: sort by the first character
2. phase i + 1: given that the strings have been sorted by the first 2^i characters, sort the strings by the first 2^{i+1} characters
   at most logN phase, and in phase i, we can compare string in constant time(for substring(j,n), query the position of substring(j + 2 ^ i) after phase i - 1. compare the position) , so take at most nlogn for sorting in each phase.

Substring Search

Knuth-Morris-Pratt

1. construct Deterministic finite state automaton(dfa)
2. once reach the final state, matched
   dfa[i][j] ::= the next state that i would transfer j into (state: the number of characters in pattern that have been matched; the length of the suffix that matches the prefix of the pattern)
   simulation:

```
Simulate(str):
        state <- 0
        for each character c in str:
                state <- dfa[c][state]
```

after simulate a string on DFA, you get a final state, which is the length of this string's longest suffix that matches a prefix of pattern

how to build DFA?

1. obviously, dfa[pattern.charAt(j)][j] = j + 1
2. for those c != pattern.charAt(j), we've known that the longest suffix(state) after taking transition c can't be j + 1, that is, it must be <= j. So, to simulate pattern[0..j - 1] + c, it's equivalent to simulate pattern[1..j - 1] + c on DFA. denote the state we reach after simulating pattern[1..j-1] on DFA with x_j(obviously, x_j <= j - 1), dfa[c][j] = dfa[c][x_j](x_j <= j - 1). Now, the problem is how to get x_j. x_j = dfa[pattern.charAt(j - 1)][x_{j-1}] base case is $x_1 = 0$.

```
BuildDFA(pattern):
        x <- 0
        dfa[pattern.charAt(0)][0] = 1
        for j in 1 until n:
                for c in R:
                        dfa[c][j] = dfa[c][x]
                dfa[pattern.charAt(j)][j] = j + 1
                x = dfa[pattern.charAt(j)][x]
```

Boyer-Moore

1. scan character in pattern from right(r) to left
2. skip if mismatch (r - M < i <= r)
    1. once you find a character in str (str.charAt(i)) is not in pattern, you can move to i + M to evaluate the string (since substring from j s.t. j <= i can't match the pattern);
    2. mismatch character in pattern: align it with its most rightest occurrence in pattern (let's say j), move to Math.max(r + 1, i + M - j - 1) to evaluate the string

heuristic takes about N/M character compares. however, in the worst case, it can be MN.

Rabin-Karp

basic idea - **modular hashing**

1. compute a hash of pattern characters 0 to M - 1
2. for each i in str, compute a hash (x_i)of characters from i to i + M - 1
3. if pattern hash = substring hash, check for a match(las vegas version; monte carlo version: return match if hash matches)

how to efficiently compute x_{i+1} given that we know x_i?

x_{i+1} = $(x_i - t_i * R^{m-1}) * R + t_{i+m}$ (precompute R^{m-1})