Greedy Algorithms

- in optimization problems the algorithm needs to make a series of choices whose overall effect is to maximise the total benefit or minimise the total cost of some system

greedy

- always makes the choice that looks best at that moment
  ↓
  making a locally optimal choice, hoping it leads to a globally optimal solution

- not expensive to compute each individual size

- but once the choice is made, it cannot be undone

Dijkstra's Algorithm

- does guarantee optimality in finding the shortest path

Shortest Path Problem :
The problem of finding the shortest path from one vertex in a graph G to another vertex.
"Shortest" can have different meanings : least edges, least weight etc

$G = (V, E)$

vertex set (V), edge set (E) → directional or nondirectional = bidirectional

Dijkstra's works for weighted, directed graphs
weights must be non-negative

Overview

$S_{initial} = \{\text{source node}\}$

2 sets of vertices
- S : set of vertices whose shortest path from source node has been determined. they form the tree
- V−S : the remaining vertices ← empty or not is the flag for termination (but what if the vertice can't be reached from the source?)
- Dijkstra is incremental

- d : array, size |V|, stores estimate length of shortest path
- pi : array, size |V|, stores predecessors for each vertex

worst case : fully connected.

Basic Steps

1. initialise d and pi
   - d → ∞, pi → null
2. set S to empty (or source node)
3. while there are still vertices in V−S
   - → move u, the vertice with the shortest path estimate from source, to S from V−S
   - → for all vertices in V−S connected to u, update shortest distance to the source
     - ↳ update only if $d[v] > d[v']$

       $d[v]$: og. in d

       $d[v']$: new value from u

Psuedo Code of Dijkstra's

```
Dijkstra_ShortestPath (Graph G, Node source){

    for each vertex v {
        d[v] = ∞ ;
        pi[v] = null pointer ;
        S[v] = 0 ;                    // S[v] = 1 if v ∈ S
    }

    d[source] = 0 ,
{V-S} →  put all vertices in a priority queue. Q. Sorted by
    d[v] in increasing order.

    while not Empty(Q) {
        u = ExtractCheapest(Q) ;
        S[u] = 1
        for each v adjacent to u
            if ( S[v] ≠ 1 and d[v] > d[u] + w[u,v]) {
                remove v from Q ;
                d[v] = d[u] + w[u,v] ;
                pi[v] = u ;
                insert v into Q accordingly
            }
    }
}
```

guarantee's minimisation

Time complexity

- worst case : $O(|V|^2)$

↓

fully connected graph

Proof of correctness

Property of shortest path.

lemma 1 In a weighted graph G, suppose that a shortest path from $x \to z$ consists of a path P from $x \to y$ followed by a path Q from $y \to z$, then P is the shortest path from $x \to y$ and Q is the shortest path from $y \to z$

↓

guarantees use of pi to find shortest path from all nodes to other nodes

proof by contradiction

WILL

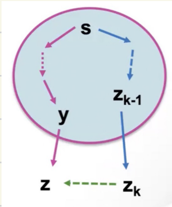
Theorem D1 → proves Dijkstra is optimal

∵ the algo is iterative, proof by induction

Let $G = (V, E, W)$ ← weights. Let S be a subset of V and s be a member of S.
Assume $d[y]$ is the shortest distance in G from s to y, for each y in S. Let z be the next vertex chosen to go into S.
If edge (y, z) is chosen to minimise $d[y] + W(y, z)$ over all edges with one vertex in S and one vertex in $V - S$,
the the path consisting of the shortest path from s to y followed by the edge (y, z) is the shortest path from s to z

Proof of D1

P : $s \to y$ (shortest) + edge (y, z)
$W(P)$ = distance along P
P' = any shortest path different from P, ie. $P' = s, z_1, \ldots, z_k, \ldots z$
z_k → first vertex in P' that's not in set S



prove pink (P) always > blue (P') by contradiction.

BE

$w(p) = d[y] + w[y, z]$

$w(p') = d[z_{k-1}] + w(z_{k-1}, z_k)$ + distance from z_k to z

always $d[z_{k-1}] + w(z_{k-1}, z_k) \geq d[y] + w(y, z)$

TESTED

theorem D2 → Given a directed weighted graph G with non-negative weights and a source vertex s, Dijkstra's algorithm computes the shortest distance from s to each vertex of G that is reachable from s

proof by induction:
1. first is shortest
2. assume $v_1 \dots v_k$ all shortest
3. for v_k

if all are shortest, then min d to v_k will also be shortest.

not unique

Minimum Spanning Tree

- Subgraph of Graph $G = (V, E)$ = $G' = (V', E')$:
  $V' \subseteq V$
  $E' \subseteq E$
  $E' \subseteq V' \times V'$
  all the edges that join $v \in V'$

- Spanning tree : a connected, acyclic subgraph containing all vertices of a graph
  (any connections, just no loops) $V' = V$, $E' \subset E$

- Minimum Spanning Tree : a minimum-weight spanning tree is a weighted graph

Prim's Algorithm

- works on undirected graph
- builds upon a single partial minimum spanning tree, at each step adding an edge connecting the vertex nearest to but not already in the current partial minimum spanning tree

Steps

T: chosen
P: $V - T \cap$ adjacent to $v \in T$

1. vertex is chosen, put in T
2. initialise P
3. every iteration: $u \in P$ will be connected to T, deleted from P, v adjacent to u, not in P will be added to P
4. when all vertexes are connected to T, P will be empty we are done.
5. vertex to be added will be chosen by the greedy method, among all vertices in P connected to T, we choose one with minimum cost.

- will still need d, s, pi, pq
- minimize edges across.

Psuedocode

$E \lg V$

```
PrimMST (G, s, n)
{
  initialise all vertex as unseen
  Reclassify s as tree vertex   ← pick a random
  Reclassify all adj v of s as fringe

  while (fringe isn't empty)
  {
    Select edge v with minimum weight between
    tree and fringe
    Reclassify V as tree, add edge tv to tree
    Reclassify unseen adj v of v to fringe,
    update seen if necessary.  ← updateFringe
  }
}
```

check →

```
updateFringe (pq, G, v) {
  ∀ w adj to v {
    if (s[w] != 1)  ← not in tree
      weight = weight of edge of w
      if (d[w] == ∞) {
        d[w] = weight
        pi[w] = v              ← weight
        insert (pq, w, d[w])
      } else if (d[w] > weight) {
        d[w] = weight
        pi[w] = v
        decreaseKey(pq, w, weight);
} } }
```

MST Property

- this is a sufficient + necessary condition to say that a tree is an mst

let T be a spanning tree of G, $G = (V, E, w)$ a connected weighted graph. Suppose for every edge $(u, v) \in E$ and not in T, if (u, v) is added to T, it creates a cycle such that (u, v) is a maximum weight edge on that cycle. Then T has the minimum spanning tree property.

MST → most efficient way to visit all nodes
Shortest → — " ———————— a particular node from another.

try with a tree diagram

Dijk v/s Prim → visit all nodes in one tour.

↳ goal is to find shortest path from source to one node at a time

$E \log E$

Kruskal's Algorithm

- computes mst
- only considers edges of increasing order
- add next edge to tree T unless it creates cycle
- uses greedy strategy

- sort edges in increasing order
- select smallest one and add
- if cycle is there, remove, skip
- stop when all nodes

- Proof of correctness
by contradiction

suppose tree produced kruskal is not MST
there is some edge u-v, not in T which creates a cycle, in which some other edge x-y has weight $W(x-y) > W(u-v)$

As $w(x-y) > w(u-v)$, edge x-y must be processed after u-v in Kruskals

At the time u-v is processed, it should be in T because it doesn't form a cycle

this contradicts that u-v is not in T

Kruskal is dynamic

- checking for cycle. → Union Find.

Dynamic Equivalence Relations

here, connection is a equivalence relation

- a binary f^n is an equivalence relation if it is
  - → reflexive ← helps satisfy MST
  - → transitive
  - → symmetric ← undirected graph

- any pair of nodes on MST is equivalent

- dynamic equivalence ⇒ equivalence relation will change with a number of operations

- given N objects in S : def 3 operations

1. initialise the object
2. connect 2 objects : add them into relation R
3. check for path connecting them

Union-Find

- find (p) : which component does P belong to
- connected (p, q) : are they connected
- union (p, q) : connect them

↑

these 3 methods implement dynamic equivalence relation

3 algos

Quick Find | QuickUnion | WQUPC

Quick Find

↳ fastest

- Interger array id [] of length N

id[p] is the id of the component p belongs to.
similar to root.

find (p) ⟶ whats the id
connected (p,q) ⟶ same id?
union (p, q) ⟶ change all those whose id = q to p

T comp :	Init	Union	Find	Conneted
	O(N)	O(N)	O(1)	O(1)

too much

QuickUnion

$\hookrightarrow$ union

- integer array id[] of length N
- root of p is id[id[id[.... id[i]

find(p) : root? → id[id[...]
connected (p, q) : same root?
find(p) == find(q)

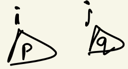
union : id of p's root = id root of q

time complexity

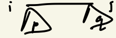
init	union	find	connected
$O(N)$	$O(N)$	$O(N)$	$O(N)$

worst case : linked list ∴ find worst case : $O(N)$

union : we have to use find. ∴ $O(N)$



$sz[i] = 4$ $\quad$ $sz(i<j) \Rightarrow$



$sz\,j = 10$

$sz[j] = 6$

Weighted Quick Union

- modify quickunion to avoid tall trees
- keep track of size of each tree (number of objects)
- balance by linking root of smaller tree to root of larger tree.

- need id of root of p
- array ds sz[i] to count of of obj in tree rooted at i

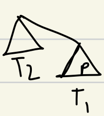
- find() $\longrightarrow$ same as quickUnion : id[id[id[i]]]
- connected() $\longrightarrow$ same as quickUnion
- union(p,q) $\longrightarrow$

make smaller tree lower
update array sz[]

```
i = find(p)    j = find(q)
if (i == j) return
if ( sz[i] < sz[j] )
    id[i] = j ; sz[j] += sz[i]
else
    id[j] = i ; sz[i] += sz[j]
```

<u>time complexity</u>.

- running time depends on depth of node



- max depth ever : $\log_2 N$

depth inc by 1 at every merge of tree with 'p' to another tree

size of the whole tree is least double of size (T_1) ∵

no of nodes

size (T_2) > Size (T_1)

size of tree can double at most $\log_2 N$ times

$\hookrightarrow$ proof next page

every merge $\Rightarrow$ at least doubling

$n \leftarrow \dots \leftarrow 8x \leftarrow 4x \leftarrow 2x \leftarrow x$

max nodes in 1 tree is n

and no of merges $= \log_2 N$

and since depth increases by 1 for every merge
max depth $= \log_2 N$

time comp	init	find	connected	union
	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(\log_2 N)$

IMPROVE !!

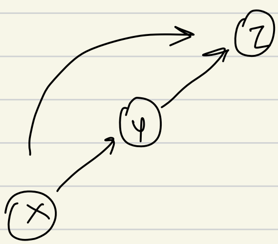
Weighted QuickUnion w/ Path Compression

- when performing find, compress the path
- id of the whole component is the same at the end
- right after computing the root of p, set the id of each node on the path to the root

↑

Ⓐ 2 pass implementation
→ add second loop to find() to set the id[] of each examined node to the root

Ⓑ 1 pass variant
→ make every other node point to its grandparent



id	x	y	z
	y	z	z

x! = y
 id[x] = id[id[x]] = z
 x = z (i = id[x])

time complexity	init	find	connected	union
	O(N)	$\log^* N$	$\log^* N$	$\log^* N$

M times, N nodes

ALL TOGETHER NOW

	Quickfind	QuickUnion	Weighted Quick Union	w/Path comp
WCTC worst case time complexity	MN	MN	N + M log N	N + M log* N

M = no of union-find operations

for 10^{11} unions w/ 10^{11} obj → Quickfind : 3000 yrs
WQUPC : 6 seconds

E log E

Back to Kruskal's

1. arrange edges in inc weight
2. start with the beginning
3. for each edge → find ✓
   connected — if false
   union

use id to keep track

```
public class KruskalMST
{
   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST(EdgeWeightedGraph G)
   {
      MinPQ<Edge> pq = new MinPQ<Edge>(G.edges()); O(|E|)

      UF uf = new UF(G.V());                        O(|V|)
      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();                      O(|E| log|E|)
         int v = e.either(), w = e.other(v);
         if (!uf.connected(v, w))                   O(|E| log*|V|)
         {
            uf.union(v, w);                         O(|V| log*|V|)
            mst.enqueue(e);                         O(|V|)
         }
      }
   }
                                 Overall: O(|E| log|E|)
   public Iterable<Edge> edges()
   {  return mst;  }
}
```

no edges — min edges

build priority queue (or sort)

greedily add edges to MST

edge v–w does not create cycle

merge sets

add edge to MST