Weighted graph algorithms

Serge Kruk

October 22, 2021

Standard algorithms on weighted graphs

- Naturally occurring problems.
- We say edges have ‘weight’ or ‘length’ interchangeably.
- We usually consider undirected graphs.
- All are good examples of the Greedy design technique
- Along the way, we see interesting and useful data structures.

Minimum spanning tree (MST)

Problem

Given a graph, find a spanning tree of minimum total weight.

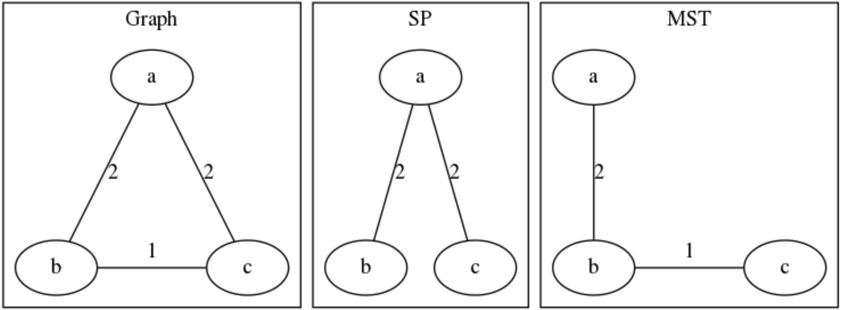
- A tree means no cycles.
- Spanning means all nodes are included and connected.

Discuss

Is that the same as the tree of shortest paths?

Not the same

- For one thing, shortest paths are rooted.
- Contrast the tree of shortest paths from node a and the MST.

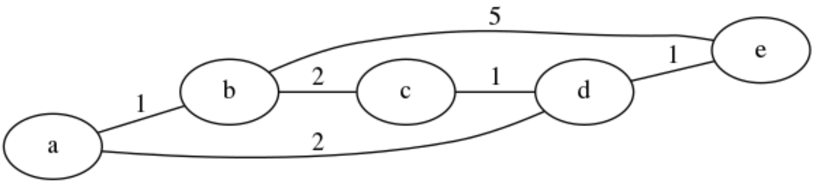


Prim's (also discovered by Dijkstra) algorithm for MST

A greedy algorithm

- Select an arbitrary vertex; it forms the beginning of tree T.
- While T does not span the graph:
    - Pick an edge of lowest cost between T and the rest of the graph.
    - Add it to T.

Example of MST via Prim



$$
\begin{aligned}
T &= (\{a\}, \{\}) \\
&= (\{a, b\}, \{(a, b)\}) \\
&= (\{a, b, d\}, \{(a, b), (a, d)\}) \\
&= (\{a, b, d, c\}, \{(a, b), (a, d), (d, c)\}) \\
&= (\{a, b, d, c, e\}, \{(a, b), (a, d), (d, c), (d, e)\})
\end{aligned}
$$

Total cost : 5

Correctness

Discuss

Correctness

- At each step of Prim, we add a vertex to T_k producing T_{k+1}. Therefore, Prim terminates with T_n.
- Each step adds a vertex with one end in T_k and one in its complement. Therefore, no cycles are created; we have a spanning tree at the last step.
- We need to show T_n is of minimum weight.
  - Say that there is a tree T' of minimum weight different from T_n. In the order where edges were added let $e = (u, v)$ be the first edge of T_n not in T'. Let $V(T_k)$ be the vertex set before addition.
  - Since T' is a spanning tree, it contains a path from u to v. That path has an edge from $V(T_k)$ to its complement. Say f. By the choice Prim made at step k, we know that the weight of f is no less than the weight of e_k. Replace f by e in T'.
  - The new tree is also spanning, has same weight as T', but has one edge more in common with T_n. Repeat until no edges differ.
  - T_n is of minimum weight.

Runtime

We cannot discuss runtime until we specify our data structures.

What do we need?

To execute Prim we need:

- A graph structure.
- A tree structure (could use the graph structure).
- A structure to hold the edges between the tree and the rest of the graph.
    - If this can also help us pick the cheapest edge, all the better!

Trivial graph struct

- A dictionary with one entry per node containing a dictionary of its neighbours, each with weight.

```
def newgraph(v=None):            # By default empty. Can contain a s
  return {v:{}} if v is not None else {}
def nodes(G):                    # A list of all nodes
  return list(G)
def nodecount(G):                # The number of nodes
  return len(G)
```

Trivial graph struct

```
def addarc(G,e):
  (u,v,w) = e
  H=G.get(u,None)
  if H is None:
    G[u]={v:w}
  else:
    G[u][v]=w
  H=G.get(v,None)
  if H is None:
    G[v]={}
def addedge(G,e):                  # Adds (u,v) and (v,u)
  (u,v,w) = e
  addarc(G,(u,v,w))
  addarc(G,(v,u,w))
```

Trivial graph struct

```python
def neighbors(G,u):                    # Returns a list of neighbors
  a=[]
  for v,w in G[u].items():
    a.append((v,w))
  return a
```

Trivial graph struct

```
def arcs(G):                    # A list of triples (u,v,w)
  all=[]
  for u in G.keys():
    for v,w in neighbors(G,u):
      all.append((u,v,w))
  return all
def edges(G):                   # A list of triples (u,v,w)
  all=[]
  for u in G.keys():
    for v,w in neighbors(G,u):
      if u<v:
        all.append((u,v,w))
  return all
def nodeingraph_p(v,G):         #  True iff node is in the graph
  H = G.get(v,False)
  return H != False
def boundaryedge_p(u,v,G):      #  True iff u is in G and v is not
  H=nodeingraph_p(u,G)
  return H!=False and not nodeingraph_p(v,G)
```

Example of simple graph

```
G={'a':{'b': 10, 'c':11}, 'b':{'c':15}, 'c':{}} # Creation
addedge(G, ('b','d',10))                       # Addition of a bidirection
print("All nodes of G: ", nodes(G))
print("Neighbours of b in G: ", neighbors(G,'b'))
print("All edges in the form (u,v,w) of G : ", edges(G))
T = newgraph('a')
addedge(T, ('a', 'c', 10))
print(nodeingraph_p('a',G), " should be true")
print(nodeingraph_p('a',T), " should be true")
print(boundaryedge_p('a','b',G), " should be false")
print(boundaryedge_p('a','b',T), " should be true")
print(boundaryedge_p('b','a',T), " should be false")
print(boundaryedge_p('b','c',T), " should be false")
```

```
All nodes of G:  ['a', 'b', 'c', 'd']
Neighbours of b in G:  [('c', 15), ('d', 10)]
All edges in the form (u,v,w) of G :  [('a', 'b', 10), ('a', 'c
True  should be true
True  should be true
```

The boundary between T and not T

- We need to inspect edges between our tree and nodes not visited yet
- We need to extract the cheapest of those
- It seems a min-heap would be perfect!

A min-heap, slightly modified to track weight

```
def weight(edge):
    return edge[2]
def newheap(n):
    return [0]*(n+1)
def insert(a,e):
    a[0] = a[0] + 1
    a[a[0]] = e
    heapfixup(a,a[0])
def heapfixup(a,i):
    while i > 1:
      p = i // 2
      if weight(a[p]) > weight(a[i]):       # We compare the edge we
          a[p],a[i] = a[i],a[p]
          i = p
      else:
          return
```

A min-heap, slightly modified to track weight

```
def extractsmallest(a):
    e,a[1],a[0] = a[1],a[a[0]],a[0]-1
    heapfixdown(a,1)
    return e
def heapfixdown(a,i):
    while 2*i <= a[0]:
        c = 2*i
        if c+1 <= a[0]:
            if weight(a[c+1]) < weight(a[c]):
                c = c+1
        if weight(a[i]) > weight(a[c]):
            a[i],a[c] = a[c],a[i]
            i = c
        else:
            return
```

Prim (Inspired by Daniel Sumindan and Meredith Benson)

```
def prim(G):
  V = nodes(G)
  n = len(V)
  q = newheap(n*n)
  T = newgraph(0)
  for (v,w) in neighbors(G,0):
    insert(q,(0, v, w))
  while nodecount(T) < n:
    (u,v,w) = extractsmallest(q)
    if boundaryedge_p(u,v,T):
      addedge(T,(u,v,w))
      for (t,w) in neighbors(G,v):
        if not nodeingraph_p(t,T):
          insert(q,(v,t,w))
  return T
```

Small test

```
G = newgraph()
for e in [(0,1,1),(0,3,2),(1,2,2),(1,4,5),(2,3,1),(3,4,1)]:
  addedge(G,e)
print(G)
T=prim(G)
print(T)
total=0
for (u,v,w) in edges(T):
  total += w
print('Total MST cost is ',total)
```

```
{0: {1: 1, 3: 2}, 1: {0: 1, 2: 2, 4: 5}, 3: {0: 2, 2: 1, 4: 1},
{0: {1: 1, 3: 2}, 1: {0: 1}, 3: {0: 2, 2: 1, 4: 1}, 2: {3: 1},
Total MST cost is  5
```

Runtime

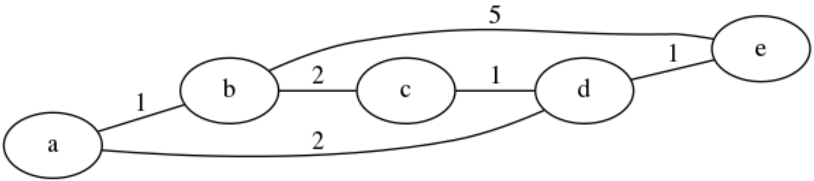
- Adjacency matrix : $O(|V|^2)$
- Binary heap and Linked List: $O((|V| + |E|)\log|V|)$
- Fibonacci heap and Linked List: $O(|E| + |V|\log|V|)$

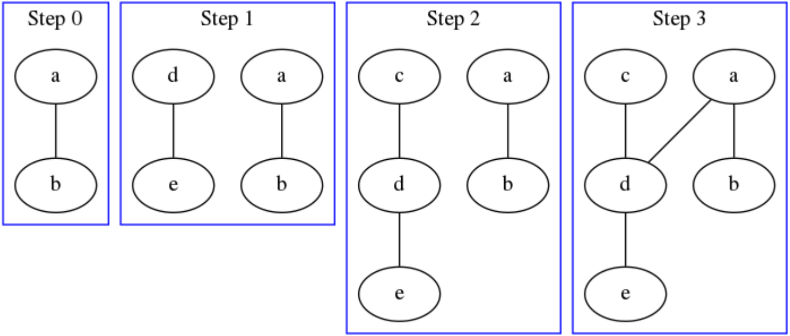
Kruskal's algorithm for MST

A greedy algorithm

- Start with an empty forest T
- While T does not span the graph:
  - Pick an edge (i,j) of minimum weight from G and delete it.
  - If adding (i,j) to T does not form a cycle, add it.

Example





Discuss

- Correctness
- Efficiency

Correctness

Proof similar to Prim's

Efficiency

- Outer loop is executed $|E|$ times.
- Multiplied by
  - Time to extract minimum weight edge
  - Time to test if edge would create a cycle
  - Time to add edge to tree structure

To extract minimum weight edge

Best data structure

A min heap? $O(\log E)$

To add edge to tree structure

- If linked list
- If Adjacency matrix
- If Incidence matrix
- If Rooted tree vector

To test if edge creates a cycle

Discuss

A new data structure

Used to keep sets of nodes of each tree in the forest.

Union-Find (aka disjoint sets)

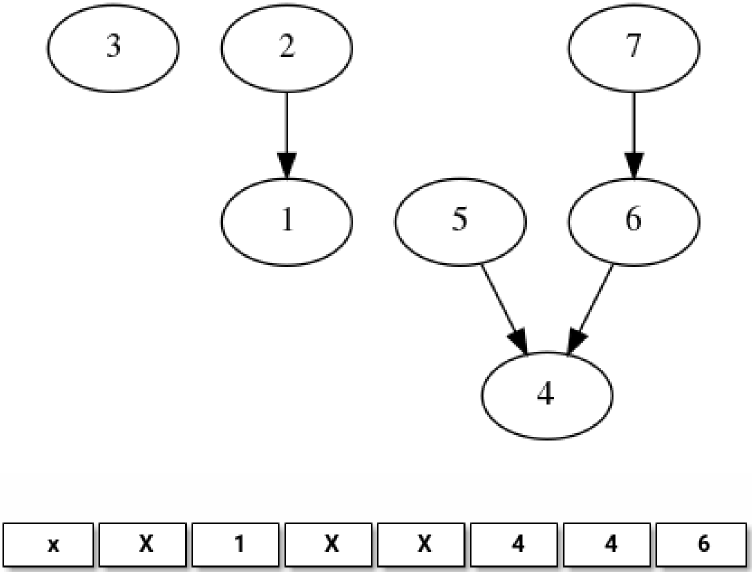
- Initialize(n): Create the data structure for n elements.
- Find(v): Returns a set ID of tree containing v.
- Union(u,v): Attach the tree of u to the tree of v.

Union-Find

To check if u and v are in different trees:

```
Find(v) == Find(u)
```

Implementation via one array of 'parent' nodes.


3
2
7
1
5
6
4
x
X
1
X
X
4
4
6

Union-Find first implementation

```
def initialize(n):
    for i in range(n):
        p[i] = None
def find(u):
    if p[u] == None:
        return u
    else:
        return find(p[u])
def union(u,v):
```

Union-Find first implementation

```
def initialize(n):
    return [None]*n

def find(p,u):
    if p[u] == None:
        return u
    else:
        return find(p,p[u])
def union(p,u,v):
    pu, pv = find(p,u), find(p,v)
    p[pu] = pv
```

Small tests

```
p = initialize(10)
find(p,2)
union(p,2,6)
find(p,2)
find(p,6)
union(p,3,5)
union(p,2,3)
[find(p,2),    find(p,3),    find(p,5),    find(p,6)]
```

5 5 5 5

Same code using lexical closure

```
(let ((p))
  (defun initialize (n)
    (setq p (make-array n :initial-element nil)))
  (defun find-u (u)
    (if (null (aref p u)) u (find-u (aref p u))))
  (defun union-u-v (u v)
    (setf (aref p (find-u u)) (find-u v))))
```

Union-Find runtime

- The runtime depends on the height of the chain of parents.
- It could be completely degenerate, therefore every find could be $\Theta(n)$ where n is the total number of elements in the sets.

Fix?

Union-Find fixing degenerate case

Keep an array of heights and do union to minimize increase.

Union-Find better implementation

```
def initialize(n):
    return  [None]*n,[0]*n
def find(p,u):
    return u if p[u] == None else find(p,p[u])
def union(p,r,u,v):
    pu, pv = find(p,u), find(p,v)
    if r[pu] < r[pv]:
        p[pu] = pv
    elif r[pv] < r[pu]:
        p[pv] = pu
    else:
        p[pv],r[pu] = pu,r[pu]+1
```

Small tests

```
p,r = initialize(10)
find(p,2)
union(p,r,2,6)
find(p,2)
find(p,6)
union(p,r,3,5)
union(p,r,2,3)
[find(p,2),    find(p,3),    find(p,5),    find(p,6)]
```

2 2 2 2

Runtime?

Efficiency

Runtime is $\Theta(\log n)$ for a graph of n nodes.

The worst case occurs whenever we always merge two trees of the same height. Let us assume we have $n = 2^k$. To merge identical trees we start by merging pairs of singletons. We started with 2^k trees and end up with 2^{k-1} trees on two nodes. At the next step we merge every pair of trees on two nodes to obtain 2^{k-2} trees on 4 nodes.
We can do this how many times until we obtain a single tree? k times, or $\log n$.

Efficiency

- Can we do even better?
- Yes, using path compression. Hard, tricky and very sophisticated proofs.
- Result: $O(\alpha^{-1}(n))$ the inverse Ackerman function.
- And then you have achieved the ultimate speed.
    - If interested, read Bob Tarjan's paper.

Kruskal (Inspired by Nguyen Do)

```
def kruskal(G):
  alledges = edges(G)
  n = nodecount(G)
  p,r = initialize(n)
  q = newheap(len(alledges))
  T = newgraph()
  edgecount = 0
  for edge in alledges:
    insert(q,edge)
  while edgecount < n-1:
    (u,v,w) = extractsmallest(q)
    if find(p,u) != find(p,v):
      union(p,r,u,v)
      addedge(T,(u,v,w))
      edgecount += 1
  return T
```

Small test

```
G = newgraph(0)
for e in [(0,1,1),(0,3,2),(1,2,2),(1,4,5),(2,3,1),(3,4,1)]:
  addedge(G,e)
print(G)
T=kruskal(G)
print(T)
total=0
for (u,v,w) in edges(T):
  total += w
print('Total MST cost is ',total)
```

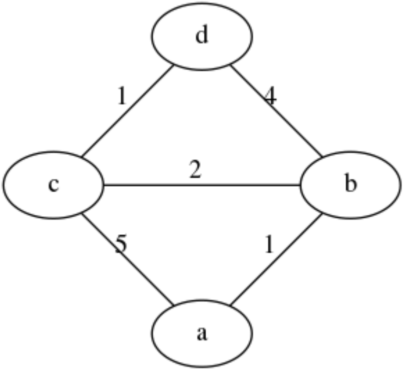
```
{0: {1: 1, 3: 2}, 1: {0: 1, 2: 2, 4: 5}, 3: {0: 2, 2: 1, 4: 1},
{0: {1: 1}, 1: {0: 1, 2: 2}, 3: {4: 1, 2: 1}, 4: {3: 1}, 2: {3:
Total MST cost is  5
```

Shortest paths tree on weighted graphs (Dijkstra)

Yet another greedy algorithm

- Given a root node r which forms a tree T
- Let D be the vector of distances of every node to r.
- While T does not span G
  - Pick a node v of minimum distance in D
  - Add node v to T
  - Update the distance vector D of neighbours of v if you can lower the distance.

Dijkstra example



Dijkstra example

- Running Dijkstra from node a.

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad\qquad p = [-, a, a, -]$$

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad\qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 5] \qquad\qquad p = [-, a, b, b]$$

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad\qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 5] \qquad\qquad p = [-, a, b, b]$$

- Minimum from d, here node c so we add arc (b, c) to our tree.

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad\qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 5] \qquad\qquad p = [-, a, b, b]$$

- Minimum from d, here node c so we add arc (b, c) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 4] \qquad\qquad p = [-, a, b, c]$$

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 5] \qquad p = [-, a, b, b]$$

- Minimum from d, here node c so we add arc (b, c) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 4] \qquad p = [-, a, b, c]$$

- Minimum from d, here node d so we add arc (c, d) to our tree.

Dijkstra example

- Running Dijkstra from node a.
- Setting up the oridinal distance vectors and of parents.

$$d = [0, 1, 5, \infty] \qquad\qquad p = [-, a, a, -]$$

- Minimum from d, here node b so we add the arc (a, b) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 5] \qquad\qquad p = [-, a, b, b]$$

- Minimum from d, here node c so we add arc (b, c) to our tree.
  - Update the distance vector.

$$d = [0, 1, 3, 4] \qquad\qquad p = [-, a, b, c]$$

- Minimum from d, here node d so we add arc (c, d) to our tree.
- We are done. The tree is $\{(a, b), (b, c), (c, d)\}$

Pseudo-code

```
def dijkstra(G,r):
    T,d,p = {r},[X]*|V|,[0]*|V|
    d[r] = 0
    for v in neighbors(G,r):
        d[v] = weight(G,(r,v))
    while size(T) < size(G):
        v = Cheapest(T,d)
        T.append(v)
        for u in neighbors(G,v):
            if d[u] > d[v]+weight(G,(v,u)):
                d[u] = d[v]+weight(G,(v,u))
                p[u] = v
```

Runtime?

Efficiency

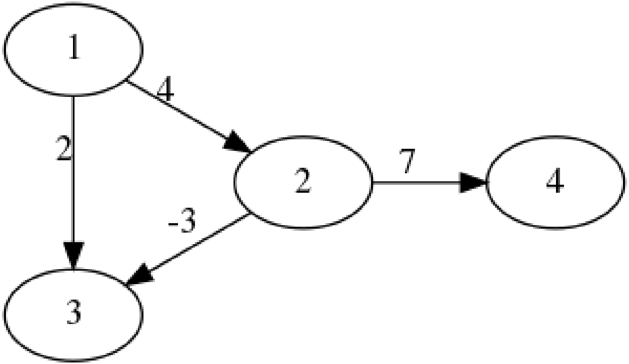
- At most $O(|V|^2)$
- Using a Minheap for distances: $O(|E| + |V|) \log |V|$
- Using a Fibonacci heap : $O(|E| + |V| \log |V|)$ (Non trivial)
- Special case: If G is a DAG, do a TopSort: $O(|E| + |V|)$

Correctness

Principle of optimality is key

If $v_1, \dots, v_j, \dots, v_k$ is a shortest path from v_1 to v_k passing through v_j, then the subpath $v_1, \dots, v_j$ is a shortest path from v_1 to v_j.

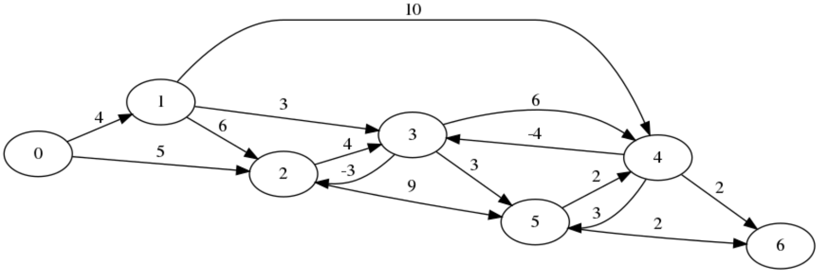
Note that Dijkstra can fail given negative weights.



- Dijkstra will pick arc $(1,3)$ at a cost of 2.
- But there is shorter path $(1,2),(2,3)$ at a cost of 1.

What to do when we have negative weights?

Assuming the question still makes sense. . .

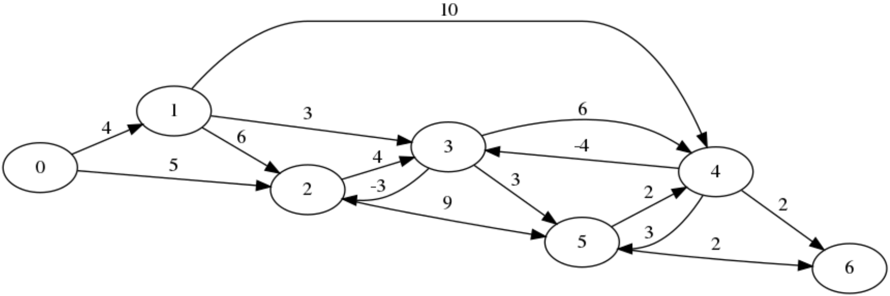


Bellman-Ford

```
def bellman_ford(G,r):
  n = len(nodes(G))
  d,p = [float('inf')]*n,[None]*n
  d[r] = 0
  for _ in range(n):
    for (u,v,w) in arcs(G):
      if d[v] > d[u] + w:
        d[v] = d[u] + w
        p[v] = u
  return p,d
```

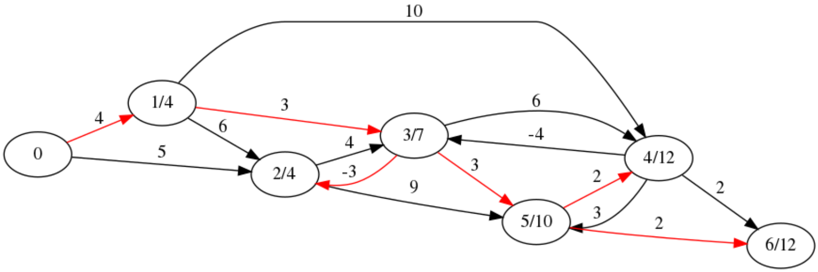
- Contrast with Dijkstra.
  - Same update of vector `d`.
  - Number of iterations is larger.

Example



```
G=newgraph()
for e in [(0,1,4),(0,2,5),(1,4,10),(1,3,3),(1,2,6),(2,3,4),(2,5,9),(
          (4,6,2),(4,5,3),(4,3,-4),(5,4,2),(5,6,2)]:
    addarc(G,e)
print(G)
print(bellman_ford(G,0))
```

```
{0: {1: 4, 2: 5}, 1: {4: 10, 3: 3, 2: 6}, 2: {3: 4, 5: 9}, 4: {
([None, 0, 3, 1, 5, 3, 5], [0, 4, 4, 7, 12, 10, 12])
```



Runtime

- $O(|V|^3)$ (Adjacency matrix)
- $O(|V||E|)$ (Linked list)

Correctness

- LI: at step k, the array d contains the length of a shortest path on at most k arcs.
    - Note that this holds at the start.
    - At each step we consider paths of one more hop for each node.

Homework/Test questions

- How can we detect negative cycles?
- How could we stop early?

What if we want all (s,t) pairs of shortest paths?

Why?

To find the diameter of the graph. Let us go back to assuming positive weights.

What if we want all (s,t) pairs of shortest paths?

- Could run Dijkstra or Bellman-Ford $|V|$ times.
- Could run Floyd-Warshall.
- Could run multi-commodity flow (later, maybe).

Addition to our trivial graph library

```
def adjacency(G):
  n = len(nodes(G))
  a = [[0 if u==v else float('inf') for u in range(n)]
      for v in range(n)]
  for (u,v,w) in arcs(G):
    a[u][v] = w
  return a
```

Example

```
G=newgraph()
for e in [(0,1,4),(0,2,5),(1,4,10),(1,3,3),(1,2,6),(2,3,4),(2,5,9),(
          (4,6,2),(4,5,3),(4,3,-4),(5,4,2),(5,6,2)]:
    addarc(G,e)

adjacency(G)
```

0	4	5	inf	inf	inf	inf
inf	0	6	3	10	inf	inf
inf	inf	0	4	inf	9	inf
inf	inf	-3	0	6	3	inf
inf	inf	inf	-4	0	3	2
inf	inf	inf	inf	2	0	2
inf	inf	inf	inf	inf	inf	0

All pairs via Floyd-Warshall

```
def floyd_warshall(G):
    allnodes = nodes(G)
    n = len(nodes(G))
    a = adjacency(G)
    for k in allnodes:
        for u in allnodes:
            for v in allnodes:
                a[u][v] = min(a[u][v], a[u][k] + a[k][v])
    return a
```

Example

```
G=newgraph()
for e in [(0,1,4),(0,2,5),(1,4,10),(1,3,3),(1,2,6),(2,3,4),(2,5,9),(
         (4,6,2),(4,5,3),(4,3,-4),(5,4,2),(5,6,2)]:
    addarc(G,e)
a=floyd_warshall(G)
print(a)
```

0	4	4	7	12	10	12
inf	0	0	3	8	6	8
inf	inf	0	4	9	7	9
inf	inf	-3	0	5	3	5
inf	inf	-7	-4	0	-1	1
inf	inf	-5	-2	2	0	2
inf	inf	inf	inf	inf	inf	0

All pairs via Floyd-Warshall

- Returns only distances here (can be modified easily)
- Runtime? $\Theta(|V|^3)$

Correctness

Theorem

A matrix A of size $|V| \times |V|$ represents the shortest paths between any pair of vertices if and only if

$$A_{i,i} = 0$$
$$A_{i,j} \leq A_{i,k} + A_{k,j} \quad \forall i, j, k$$

Where to from here?

Other algorithms:

- Yen's modification of Dijkstra
- All pairs via matrix multiplication
- Dreyfus method
- Out-of-Kilter
- A*

Better data structures:

- Fibonacci heaps
- Path compression

More general approaches:

- Linear programming
- Network flows

Applications:

- Google map
- Chip design

Morals

- All algorithms seen hereare greedy: from a local view, they make the correct global decision.
- Some (D, F-W, B-F) can also be viewed as 'Dynamic programming' algorithms. (To be defined later)

These 'Design Techniques' are not entirely disjoint.

Speaking of 'Design Techniques'

- Brute force
- Avoid computation by saving states
- Pre-compute to simplify later processing
- Scanning and accumulating
- Divide-and-conquer
- Effective data structures
- Greedy approaches (for optimization problems)

Homework/Test questions

- Can you name (describe) an algorithm illustrating each of the above techniques?
- Can we find the maximum weight spanning tree by reversing the weights and running Kruskal?
- Can we find the tree of longuest paths by reversing the weights and running Bellman-Ford?
- Is the minimum spanning tree unique if the edge weights are distinct?
- If you multiply all edge weights by some factor, is the tree of shortest path the same?
- If a graph has negative cycles, does it imply that no pair of nodes have a finite cost shortest path?