Graph algorithms - glossary

accessible

a vertex *y* is *accessible* from *x* if there exists at least one walk starting at *x* and ending at *y*. As walks of length 0 are valid, this means that each vertex is accessible from itself

adjacency

vertex *y* is adjecent to vertex *x* if there is an edge from *x* to *y*

closed walk

a walk that starts and ends in the same vertex.

connected component

a subset of the set of vertices, so that each vertex in the subset is accessible from each vertex of the subset, and which is maximal (there is no proper superset with the same property). The *connected component* term is used only for undirected graphs; for a directed graph, the same concept is called *strongly connected component.*

cost of a walk

the sum of the costs of the edges that form that walk. Note that a zero length walk has a cost of zero.

cycle

a closed walk of length at least 1, with no other repeating vertices except for the fact that the first is the same as the last, and with no repeating edges. In an undirected graph, this means that a cycle has the length at least 3.

incidency

vertex *x* is incident to edge *e* if *x* is an endpoint of *e*

length of a walk

the number of edges along the walk, or, equivalently, the number of vertices minus one.

path

a walk with no repeating vertices

strongly connected component

see *connected component*

walk

a sequence of 1 or more vertices, $(x_0, x_1,...,x_k)$ such that each vertex has an edge to the next one. The *length* of the walk is the number of edges along it, or, equivalently, the number of vertices minus one. Repeating vertices or edges are allowed. A walk of length 0 is allowed; it has a single vertex (so the starting vertex is the same as the destination vertex) and zero edges.

Graph algorithms - Graph representation

Internal representation

Choosing a representation for the data is a matter of trade-offs.

To choose a good representation for the data, we always need to know:

- What does the data look like — how big is the data and how large is each part relative to the others;
- What operations do we need to perform — how often is each operation performed.

With respect to the graph size, we have *dense graphs*, where $m = \Theta(n^2)$, *sparse graphs*, where $m = O(n)$, and some intermediate graphs.

In dense graphs, the degrees of most vertices are of the order of $\Theta(n)$. In sparse graphs, the degrees of most vertices are small ($O(1)$), but we can sometimes have a few vertices of very high degree.

For instance, the graph corresponding to a road network is a sparse graph. If we represent intersections by vertices, each vertex (intersection) usually has 3 or 4 neighbours (out of the hundreds of millions intersections in the world).

Typically, we have the following operations to be performed:

- Given vertices x and y, test if (x,y) is an edge;
- Given a vertex x, parse the set $N^{out}(x)$ of outbound neighbours of x;
- Given a vertex x, parse the set $N^{in}(x)$ of inbound neighbours of x;
- Parse the set of vertices of the graph.

Adjacency matrix

We have a $n \times n$ matrix with 0-1 or true-false values, defined as: $a_{x,y}=1$ if there is an edge from x to y, and 0 otherwise.

.

Memory: $\Theta(n^2)$

Test edge: $O(1)$

Parse neighbours: $\Theta(n)$

Summary: Adjacency matrix is good for dense graphs, but bad for sparse graphs. Imagine a graph with 10^8 vertices and 4×10^8

edges, but which occupies 10^{16} bits (or around 1000TB).

List of edges

It involves keeping a collection containing the edges (as pairs of vertices). It is compact for sparse graphs, but all operations need to parse the full collection.

Memory: *Θ(m)*

Test edge: *O(m)*

Parse neighbours: *O(m)*

List of neighbours

For each vertex, we keep a collection of its neighbours (inbound or outbound or both).

The collection of neighbors may be a vector, a linked list, or a set. The set allows to quickly test if (x,y) is an edge if x has a lot of outbound neighbors; the vector is more compact and works reasonably if the above test is not very often performed or if the number of outbound neighbours is small.

To get from the vertex to the set of neighbours, we can use a vector where the vertex is the index, or a map (dictionary) where the vertex is the key.

The vector is more compact and faster, but requires the vertices to be consecutive integers (which, in turn, means that removing a vertex requires the re-numbering of all the vertices following it).

Memory: *Θ(n+m)*

Test edge: *O(deg(x))*

Parse neighbours: *Θ(deg(x))*

External interface

- it is useful to separate implementation from the algorithms
- for particular problems, it is possible to have an implementation that is far from the above implementations

Read-only operations

- parse the outbound neighbours of a given vertex
- parse the inbound neighbours of a given vertex
- test if (x, y) is an edge
- parse all the vertices

Operations concerning edge costs

- get the cost of the edge (x, y) (assuming it is an edge)
- parse the outbound/inbound edges of a given vertex, returning the cost of the edge along with the neighbour vertex

Type for vertex

- Must provide test for equality
- For efficiency, it should provide either a hash function or a comparator function
- It should be a template argument of the interface (for static typed languages)

Example, Python:

```
class Vertex:
        def __eq__(self, other):
                if not isinstance(other, self.__class__):
                        return False
                ...
        def __ne__(self, other):
                return not self.__eq__(other)
        def __hash__(self):
                ...
```

Example, Java:

```
class Vertex:
        boolean equals(Object other) {
                if (! other instanceof Vertex) return false;
                Vertex otherVertex = (Vertex)other;
                ...
        }
        int hashCode() {
                ...
        }
}
```

Example, C++

```
class Vertex:
        bool operator==(Vertex const& other) {
                ...
        }
        bool operator<(Vertex const& other) {
                ...
        }
}
```

Return type for parse operations

collection, by value

Simple to describe; needs to perform a copy; interface may be sensitive to the type of collection

collection, by reference

Simple to describe; the graph may be inadvertently be changed if the outside code changes the result; what to do if the internal representation is different?; interface may be sensitive to the type of collection

iterable

No copy is needed; very flexible; a bit harder to implement

Example, Python:

```
class Graph:
        # Return by reference; beware of possible change by user
        def parseNout(self, x):
                return self.__out[x]

        # return a copy
        def parseNout(self, x):
                l = []
                for y in self.__out[x]:
                        l.append(y)
                return l

        # return a copy
        def parseNout(self, x):
                return [y for y in self.__out[x]]

        # return an iterable
        def parseNout(self, x):
                for y in self.__out[x]:
                        yield y

for y in g.parseNout(x):
        ...
# Beware:
s = g.parseNout(x)
s.append(...)
```

Example, Java:

```
class Graph:
        // return by reference
        public Iterable parseNout(Vertex x) {
                return _out.get(x);
        }
        // return a copy
        public Iterable<Vertex> parseNout(Vertex x) {
                return new ArrayList<Vertex>(_out.get(x));
        }
        // return a read-only wrapper over the direct reference
        public Iterable parseNout(Vertex x) {
                return Collections.unmodifyableList(_out.get(x));
        }
        private Map > _out;
}
```

Example, C++

```
class Graph:
        // Standard C++ collection
        class iterator {...}
        iterator parseNout_begin(x){..}
        iterator parseNout_end(x){..}

        // ad-hoc - return by value
        list parseNout(Vertex x)
}
```

Graph algorithms - graph traversal

Problem

- Given a starting vertex *s*, find all vertices that are accessible from it;
- Additionally, find a path, or a minimum length path, from the starting vertex to a given destination vertex;

Breadth-first traversal algorithm

The algorith below visits all the vertices that are accessible from the start vertex. They are visited in the order of increasing distances from the starting vertex. A *previous* vector or map is computed, allowing us to compute the minimum length path from the starting vertex to any choosen accessible vertex.

```
Input:
    G : graph
    s : a vertex
Output:
    accessible : the set of vertices that are accessible from s
    prev : a map that maps each accessible vertex to its predecessor on a path from s
to it
Algorithm:
    Queue q
    Dictionary prev
    Dictionary dist
    Set visited
    q.enqueue(s)
    visited.add(s)
    dist[s] = 0
    while not q.isEmpty() do
        x = q.dequeue()
        for y in Nout(x) do
            if y not in visited then
                q.enqueue(y)
                visited.add(y)
                dist[y] = dist[x] + 1
                prev[y] = x
            end if
        end for
    end while
    accessible = visited
```

Accessibility - proof of correctness

The proof comes in 3 parts:

1. All returned vertices are accessible,
2. The algorithm finishes,
3. All accessible vertices are returned.

1. When a vertex is put into the queue, it is also put into the visited set, and no vertex is ever removed from the visited set, so any vertex in the queue is also in the visited set.

Next, we claim that, at each iteration, all vertices in the visited set are accessible from the start. At the beginning, this is true, because only the starting vertex is in the visited set. Next, a vertex is put in the visited set only if it is the outbound neighbour of a vertex in the queue; that vertex is therefore in the visited set and so it is accessible from start, so the added vertex is also accessible from start.

2. A vertex added to the queue only if not already in the visited set; it is also added in the visited set and never removed. So, any vertex gets at most once in the queue. So, the algorithm finishes in at most *n* iterations of the main loop. At each iteration, the inner loop executes outdeg(x) times, which sums up, on all iterations, to the total number of edges. So, the algorithm runs in $O(n+m)$.

3. Suppose, by contradiction, that there is a vertex *y* accessible from *s* (start), but which is not in the visited set at the end. Since *y* is accessible from *s*, there is a path $(s=x_0,x_1,...,x_k=y)$ from *s* to *y*. On that path, there must be a first vertex that is not visited, so, there is an *i* such that x_i is visited and x_{i+1} is not visited.

This means that there was a moment when x_i was visited and added to the queue, and, at a later time, there was an iteration when x_i was processed. At that moment, x_{i+1} was discovered as an unvisited neighbour of x_i and added to the visited set, which contradicts the hypothesis.

3'. (alternative) We claim that, at each iteration, for any vertex *x* accessible from start, either *x* is in the visited set, or there is a walk going from start to *x* that has a vertex in the queue and that vertex is followed only by vertices not in the visited set.

The above condition is true in the beginning. When going from one iteration to the next, there are two changes: the top vertex is extracted from the queue, and its neighbours are inserted into the queue and into the visited set.

If $(s=x_0,...,x_j,x_{j+1},...,x_k=t)$ is a path from the starting vertex, x_j is the top of the queue and $x_{j+1},...,x_k$ are not visited, then x_{j+1} is added to the queue and is followed only by non-visited vertices.

If, though, a vertex x_l, with $l>j$, is added to the visited set, it is also added to the queue and is followed only by non-visited vertices.

Minimum length path - proof of correctness

Note that `dist` contains the length of the path that would be retrieved from the `prev` map.

First, we remark that vertices are processed in groups with increasing values of `dist`. That is, first we process the start vertex that has a `dist` of 0, and the algorithm puts vertices with a `dist` of 1 in the queue. Then they are processed and the vertices put in the queue will have a `dist` of 2, then the vertices with `dist`=2 are processed and vertices with `dist`=3 are put into the queue, and so on

Next, we prove that if `dist[x]=k`, then there exists a walk of length `k` from start to `x`. This can be proven by indunction on the iterations. Initially, we have a zero length walk from start to itself. Next, when we set `dist` for a vertex, it is set based on a previous vertex, as `dist[y]=dist[x]+1`. By induction hypothesis, there is a walk of length `dist[x]` from start to `x` and, by adding the edge `(x,y)` to it, we get a walk of length `dist[x]+1` from start to `y`.

Finally, we will prove that `dist[x]` is indeed the length of the minimum length walk from start to `x`. Suppose $(s=x_0, x_1,...,x_k)$ is a minimum length walk from *s* to some vertex x_k. We claim that `dist[`x_i`]`$=i$, for all vertices in the walk. Let *i* be the first vertex for which the claim is false. It means that, when x_{i-1} was processed, x_i was already processed (otherwise it would have got assigned a `dist` of i). But this means that it got an even smaller value (`dist[`x_i`]`$<i$), which means that there is a strictly better walk to x_i, which contradicts our assumption.

Graph algorithms - Strongly connected components

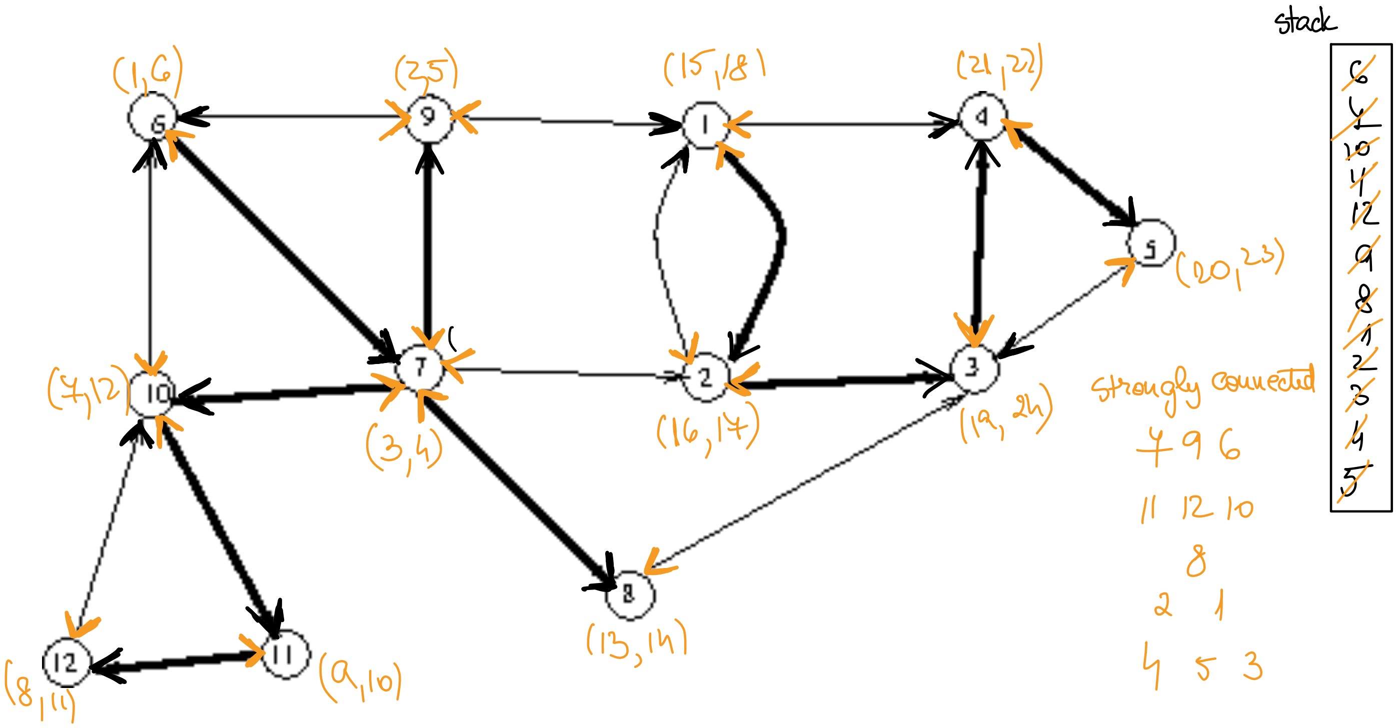
The Kosaraju algorithm

```
Input:
    G : directed graph
Output:
    comp : a map that associates, to each vertex, the ID of its strongly connected
component

Subalgorithm DF1(Graph G, vertex x, Set& visited, Stack& processed)
    for y in Nout(x) do
        if y not in visited then
            visited.add(y)
            DF1(y)
        end if
    end for
    processed.push(x)

Algorithm:
    Stack processed
    Set visited

    for s in X do
        if s not in visited then
            visited.add(s)
            DF1(G, s, visited, processed)

    visited.clear()
    Queue q
    int c = 0
    while not processed.isEmpty() do
        s = processed.pop()
        if s not in visited then
            c = c + 1
            comp[s] = c
            q.enqueue(s)
            visited.add(s)
            while not q.isEmpty() do
                x = q.dequeue()
                for y in Nin(x) do
                    if y not in visited then
                        visited.add(y)
                        q.enqueue(y)
                        comp[y] = c
                    end if
                 end for
             end while
         end if
     end while
```

* any strongly connected component remains strongly conected even after the graph is transposed.

Kosajaru's alogrithm:

1) DFS traversal + add to stack when the vertex is finished
2) Transpose graph
3) Pop one by one, see neighbours and update visited []
4) When we are "done" we get a strongly connected comp.



Correctness

To clarify the terminology, at each time during the depth-first traversal, each vertex can be in one of 3 possible states:

- not visited yet (not yet in the *visited* set);
- on the execution stack (on the path from the current root to the current node);
- fully visited (and thus in the processed stack).

First, it is easy to show that a depth-first (DF) traversal started from a root visits all the vertices that are accessible from that root. It can ve shown by contradiction: assume a vertex that is accessible from the root, but never gets visited. Consider then a path from the root to that vertex; that path must have, at some point, a visited vertex followed by a non-visited vertex. But this means that, when that last visited vertex was visited, its successor must have been visited, too.

Next, for each SCC, we consider the *representative vertex* of the SCC the first vertex, from that SCC, to get (partially) visited. We claim two things:

- The representative of a SCC is the first vertex, from that SCC, to be visited, but the last to be fully processed and added to the stack. Consequently, it will also be the first to be taken out of the stack in the second phase.
- If SCC B is accessible from SCC A, then the representative of A is fully processed *after* the representative of B.

Indeed, let *x* be the representative of a SCC. At some point, *x* is processed, becoming effectively the root of DFS. Until it gets fully processed, all the vertices accessible from *x* are processed, unless at least a path to them goes through an already visited vertex. However, the path cannot go through any of the ancestors of *x*, because none is accessible from *x*, as *x* is the first vertex of its SCC that is processed. As for the fully visited vertices, all the vertices accessible from them are already fully processed. Therefore, all the vertices in the SCC of *x* are processed between the time *x* is first touched until the time *x* is finished.

.

Now consider the second phase. At each iteration of the main loop, a representative of a new SCC is picked up from the processed stack. Let's call *x* that vertex and *A* its SCC. Now, all the vertices from which *x* is accessible are either members of *A* or members of another SCC (let's call it *Z*) from which *A* is accessible. But, in the second case, the representative of *Z* must have been put in the processed stack after *x* and, therefore, it is already processed and its SCC retrieved.

On the reflexive-transitive closure and the reduced graph

The *reflexive-transitive closure* of a graph is the accessibility relation in that graph.

Given a graph, we can define the relation *x~y* if *x* is accessible from *y* and *y* is accessible from *x*. Then, ~ is an equivalence relation, and it defines a partitioning of the set of vertices in the graph. The parts are the strongly connected components.

Next, we can define an accessibility relation between components, by saying that component *B* is accessible from component *A* if there is a vertex in *B* that is accessible from a vertex in *A*. We immediately see that if there is a vertex in *B* that is accessible from a vertex in *A*, then each vertex in *B* is accessible from\ each vertex in *A*.

Finally, we can define a new graph in which the vertices are the SCC of the original graph and where we put an edge between two vertices if there is an edge between a vertex of the first component and a vertex of the second component. This is the *reduced graph* of the strogly connected components.

It is easy to see that, in the reduced graph, there are no cycles. If there was a cycle, the SCCs in that cycle would be a single SCC.

Tarjan's algorithm

Tarjan's algorithm is also based on performing a DFS in the graph, but it computes, for each vertex, the earliest ancestor vertex (closest to the root) that is directly reachable from that vertex or a descendent of that vertex in the DFS tree (this is called *lowlink*).

The SCC representatives are recognized by the fact that their lowlink is equal to themselves.

Actually retrieving the component is done as follows: a stack is maintained where a vertex is added when its processing starts. Then, when a vertex is recognized as SCC representative at the end of its processing, all the vertices up to and including the SCC representatives are poped out of the stack and marked as part of the SCC.

Graph algorithms - Minimum cost walk by dynamic programming

Find minimum cost walk in presence of negative cost edges (but no negative cost cycles).

Note: if there is a negative cost cycle that can be inserted into the walk from start to end, then there is no minimum cost walk - by repeating the cycle, we can obtain walks of cost as small as we want

s = starting vertex, t = ending (target) vertex.

Distances in the graph

Define $d(x,y)$= the cost of the minimum cost walk from x to y, or ∞ if y is inaccessible from x.

Note that there is always a path achieving $d(x,y)$ (if y is accessible from x at all).

Minimum cost walks by length

Define $w_{k,x}$ = the cost of minimum cost walk of length at most k from s to x, or ∞ if no such walk exists.

We have a recurrence relation:

- $w_{0,s}=0$;
- $w_{0,x}=\infty$, for $x\neq s$;
- $w_{k+1,x}=\min(w_{k,x}, \min_{y\in N^{in}(x)}(w_{k,y}+c(y,x)))$;

Based on the recurrence relation above, we can easily compute $w_{k,x}$ for any vertex x and for any natural number k. We compute w row by row (in increasing order of k).

Since the minimum cost is always achieved by a path, $d_{n-1,t}$ gives the minimum cost from s to t.

To retrieve the path, we go back from t, reconstructing how we achieved each value of w.

Graph algorithms - Bellman-Ford algorithm

Problem

Given a graph with no negative costs cycles and two vertices *s* and *t*, find a minimum cost walk from *s* to *t*.

Idea

The algorithm keeps two mappings:

- *dist[x]* = the cost of the minimum cost walk from *s* to *x* known so far
- *prev[x]* = the vertex just before *x* on the walk above.

Initially, *dist[s]*=0 and *dist[x]*=∞ for $x \neq s$; this reflects the fact that we only know a zero-length walk from *s* to itself.

Then, we repeatedly performs a *relaxation* operation defined as follows: if *(x,y)* is an edge such that *dist[y]* > *dist[x]* + *c(x,y)*, then we set:

- dist[y] = dist[x] + c(x,y)
- prev[y] = x

The idea of the relaxation operation is that, if we realize that we have a better walk leading to *y* by using *(x,y)* as its last edge, compared to what we know so far, we update our knowledge.

The algorithm

```
Input:
    G : directed graph with costs
    s, t : two vertices
Output:
    dist : a map that associates, to each accessible vertex, the cost of the minimum
            cost walk from s to it
    prev : a map that maps each accessible vertex to its predecessor on a path from s
to it
Algorithm:
    for x in X do
        dist[x] = ∞
    end for
    dist[s] = 0
    changed = true
    while changed do
        changed = false
        for (x,y) in E do
            if dist[y] > dist[x] + c(x,y) then
                dist[y] = dist[x] + c(x, y)
                prev[y] = x
                changed = true
            end if
        end for
    end while
```

Proof of correctness

The proof is in three parts:

- at each stage, *dist* and *prev* correspond to existing walks (this comes immediately from how the relaxation operation works;
- the algorithm finishes;

- when the algorithm finishes, *dist[x]* = *d(s,x)* for all vertices *x*.

For the last two parts, we notice that, at iteration *k*, we have that *dist[x]* $\leq w_{k,x}$ (see the Bellman's dynamic programming algorithm). This makes the Bellman-Ford finish in at most *n-1* iterations and end with the correct distances.

Graph algorithms - Dijkstra's algorithm

Problem

Given a graph with non-negative costs and two vertices *s* and *t*, find a minimum cost walk from *s* to *t*.

Idea

Dijkstra's algorithm still relies on Bellman's optimality principle; however, it computes distances from the starting vertex in increasing order of the distances. This way, the distance from start to a given vertex doesn't have to be recomputed after the vertex is processed.

This way, Dijkstra's algorithm looks a bit like the breadth-first traversal; however, the queue is replaced by a priority queue where the top vertex is the closest to the starting vertex.

The algorithm

```
Input:
    G : directed graph with costs
    s, t : two vertices
Output:
    dist : a map that associates, to each accessible vertex, the cost of the minimum
            cost walk from s to it
    prev : a map that maps each accessible vertex to its predecessor on a path from s
to it
Algorithm:
    PriorityQueue q
    Dictionary prev
    Dictionary dist
    q.enqueue(s, 0)              // second argument is priority
    dist[s] = 0
    found = false
    while not q.isEmpty() and not found do
        x = q.dequeue()      // dequeues the element with minimum value of priority
        for y in Nout(x) do
            if y not in dist.keys() or dist[x] + cost(x,y) < dist[y] then
                dist[y] = dist[x] + cost(x, y)
                q.enqueue(y, dist[y])
                prev[y] = x
            end if
        end for
        if x == t then
            found = true
        endif
    end while
```

- If all costs are non-negative, the algorithm above doesn't put a vertex into the priority queue once it was extracted and processed (see proof below).
- If there are negative costs, but no negative cost cycles, then a vertex may be processed multiple times. However, if we eliminate the exit on dequeueing the target vertex, the algorithm finishes after a finite number of steps and the result is correct.
- If there is a negative cost cycle accessible from the starting vertex, then the algoritm can end with an incorrect result or it can run forever.

Proof of correctness (for non-negative costs)

Non-negative costs case

We claim that, when a vertex is dequeued from the priority queue, its `dist` is equal to the cost of the minimum cost walk from the start to it.

Suppose the contrary. Let x be the first vertex for which the above statement is false. So, we have that $dist[x]$ is strictly smaller than the cost of the minimum cost walk from s to x.

Let S be the set of vertices that were in the priority queue and have already been dequeued from it when x gets dequeued ($x \notin S$). On the best walh from s to x the vertex just before x cannot be in S, otherwise $dist[x]$ would have been correctly computed when that vertex was dequeued.

So, let (y,z) be the first edge on the minimum cost walk from s to x that exists S.

In the image below, the upper walk is the minimum cost walk, and the lower one is the one found by the algorithm, and implied by the values of *dist* and *prev*.

However, since x is at the top of the priority queue and not z, we have that $cost(s,...,y,z) \geq cost(s,...,x)$ and, since all edges have non-negative costs, $cost(z,...,x) \geq 0$. Therefore, the bottom walk, found by the algorithm, cannot have a larger cost than the minimum cost walk, which prove our claim.

The case of negative costs

Graph algorithms - A* algorithm

Problem

Given a graph with non-negative costs, two vertices s and t, and, for each vertex x, an estimation $h(x)$ of the distance from x to t find a minimum cost walk from s to t.

Idea

The goal of the A* algorithm is to avoid computing paths that start from s but go in a direction opposite to t. For instance, if we want to go from Cluj to Paris, we won't try a route through Moscow.

To be able to exclude such unpromising routes, we need, in addition to the graph itself, an estimation of the distance from each vertex to the target vertex. This estimation is part of the input data.

Of course, not any estimation function will work. There are two conditions on the estimation function:

- (strong condition): for all edges (x,y), we have $c(x,y) \geq h(x) - h(y)$ (in other words, the estimation does not decrease, along an edge, faster than the cost of that edge); in addition, $h(t)=0$;
- (weak condition): for all vertices x, we have $h(x) \leq d(x,t)$ (in other words, the estimation is always an underestimation).

If the graph represents places in space (cities, intersections, etc), then the estimation function could be the euclidian distance.

Essentially, the A* algorithm is identical with Dijkstra's algorithm, with one difference: the priority of a vertex x in the priority queue is not $dist[x]$ but rather $dist[x]+h(x)$.

The algorithm

```
Input:
    G : directed graph with costs
    s, t : two vertices
    h : X -> R the estimation of the distance to t
Output:
    dist : a map that associates, to each accessible vertex, the cost of the minimum
            cost walk from s to it
    prev : a map that maps each accessible vertex to its predecessor on a path from s
to it
Algorithm:
    PriorityQueue q
    Dictionary prev
    Dictionary dist
    q.enqueue(s, h(s))
    dist[s] = 0
    found = false
    while not q.isEmpty() and not found do
        x = q.dequeue()
        for y in Nout(x) do
            if y not in dist.keys() or dist[x] + cost(x,y) < dist[y] then
                dist[y] = dist[x] + cost(x, y)
                q.enqueue(y, dist[y]+h(y))
                prev[y] = x
            end if
        end for
        if x == t then
            found = true
        endif
    end while
```

Proof of correctness

We claim that:

- If the estimation satisfies the strong condition, then, each time a vertex x is dequeued, *dist[x]* is the cost of the minimum cost walk from s to x. It immediately follows that, at the end, when t is dequeued, we have the minimum cost walk from s to t.
- If the estimation only satisfies the weak condition, we still get the minimum cost walk from s to t, but some vertices may be dequeued and enqueued several times.

Strong condition estimate

One way of proving the correctness is as follows. We set a new cost function on the edges, defined as

$c'(x,y) = c(x,y) - h(x) + h(y)$

A walk from s to t with the new cost function will have a cost

$c'(s=x_0,x_1,\ldots,x_k=t) = c'(x_0,x_1) + c'(x_1,x_2) + \ldots + c'(x_{k-1},x_k) =$

$= c(x_0,x_1) - h(x_0) + h(x_1) + c(x_1,x_2) - h(x_1) + h(x_2) + \ldots + c(x_{k-1},x_k) - h(x_{k-1}) + h(x_k) =$

$= c(x_0,x_1,\ldots,x_k) - h(s) + h(t)$

Consequently, for all the walks from s to t, the difference between the cost c and c' is the same, so, the minimum cost walk is the same for both costs.

Finally, notice that the A* algorithm is, essentially, the Dijkstra algorithm for the cost c', and that c' is non-negative.

Graph algorithms - Minimum cost walk between all pairs of vertices

Find minimum cost walk between all pairs of vertices. Negative cost edges are ok; negative cost cycles are not.

Matrix multiplication

Define $w_{k,x,y}$ = the cost of minimum cost walk of length at most k from x to y, or ∞ if no such walk exists.

We have a recurrence relation. The base case is:

- $w_{1,x,x}=0$;
- $w_{1,x,y}=cost(x,y)$, if (x,y) is an edge of the graph;
- $w_{1,x,y}=\infty$, if $x\neq y$ and (x,y) is not an edge of the graph;

The actual recurrence is:

- $w_{k+l,x,y}=\min(w_{k,x,z}+w_{l,z,y})$.

The idea is to compute $w_{k,x,y}$ for a value of k based upon the already computed values for $k/2$. We need to get to a k greater than n.

The number of operations for computing all $w_{k,x,y}$ for a given k is $O(n^3)$. Doing this up to a k greater than n will take $O(n^3 \log n)$.

To retrieve the path, we can define a second array, $f_{k,x,y}$ = the next vertex after x on the walk of cost $w_{k,x,y}$. When the minimum in the recurrence is reached for some intermediate vertex z, we set $f_{2k,x,y} = f_{k,x,z}$.

Floyd-Warshall algorithm

It is also based on dynamic programming. We need to have a numbering of all vertices of the graph, $X=\{z_0,z_1,...,z_{n-1}\}$.

Then, we define $w_{k,x,y}$ = the cost of minimum cost walk from x to y, using, as intermediate vertices, only those in the set $\{z_0,z_1,...,z_{k-1}\}$.

The recursion starts like for the matrix multiplication algorithm. Then, we have:

- $w_{k+1,x,y}=\min(w_{k,x,y}, w_{k,x,zk}+w_{k,zk,y})$.

Finally, $w_{n,x,y}$ is allowed to use all vertices in the graph.

The algorithm is thus (assuming vertices are the numbers from 0 to n-1):

```
for i=0 to n-1 do
    for j=0 to n-1 do
        if i==j then
```

```
            w[i,j] = 0
        else if (i,j) is edge in G then
            w[i,j] = cost(i,j)
            f[i,j] = j
        else
            w[i,j] = infinity
        endif
    endfor
endfor
for k=0 to n-1 do
    for i=0 to n-1 do
        for j=0 to n-1 do                try to go through k
            if w[i,j] > w[i,k]+w[k,j] then
                w[i,j] = w[i,k]+w[k,j]
                f[i,j] = f[i,k]
            endif
        endfor
    endfor
endfor
```

use best so far

The algorithm complexity is O(n^3).

Graph algorithms - Directed acyclic graphs (DAGs)

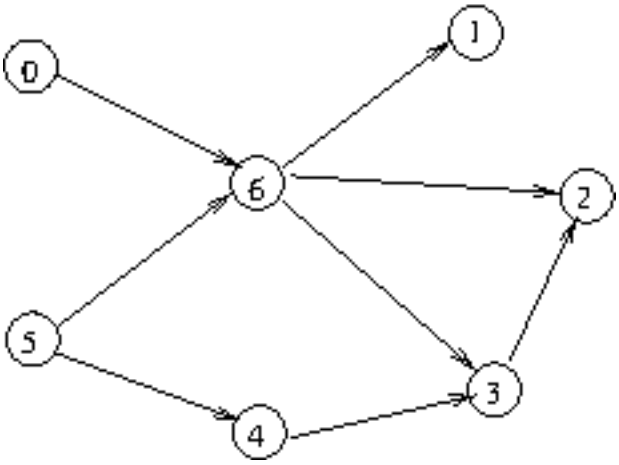
Basics

A *directed acyclic graph* (DAG) is a directed graph having no cycle.

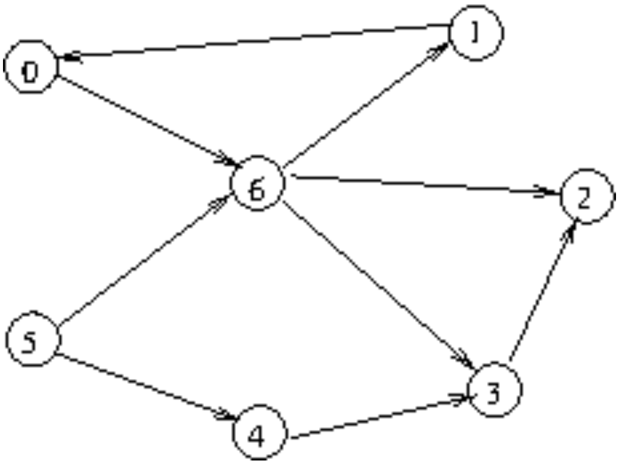
Directed acyclic graphs are often used for representing dependency relations, for example:

- vertices are activities in a project, and an edge *(x,y)* means that activity *y* cannot start before activity *x* is completed (because *y* depends on the end product of *x*);
- vertices are topics in a book, and an edge *(x,y)* means that topic *y* cannot be understood without first understanding topic *x*;
- vertices are computation steps or computation results, and an edge *(x,y)* means that computing *y* takes as inputs the result for *x*.

DAG example



Cycle-containing graph



Topological sorting

Often, when dependency relations are involved, the following two problems need to be solved:

1. Find if there is any circular dependency (in other words, if the dependency graph is a DAG or not);
2. Put the items in an order compatible with the dependency restrictions, that is, put the vertices in a list such that whenever there is an edge *(x,y)*, then *x* comes before *y* in that list.

The latter problem is called *topological sorting*. Note that the solution is not, generally, unique.

Finding if a directed graph has cycles or not is done while attempting to do the topological sorting.

Property: Topological sorting is possible, for a directed graph, if and only if there are no cycles in the graph.

If a graph has a cycle, then it is obvious that topologically sorting it is impossible: Suppose we have a topological sorting, and let *x* be the first vertex from the cycle that appears in the topological sorting. Then, let *y* be the preceeding vertex in that cycle; we have the edge *(y,x)*, but *y* comes after *x* in the topological sorting, which is not allowed.

For proving the other way round, we use the construction algorithms below. We'll prove that neither one fails unless there is a cycle in the input graph.

Predecessor counting algorithm

The ideea is the following: we take a vertex with no predecessors, we put it on the sorted list, and we eliminate it from the graph. Then, we take a vertex with no predecessors from the remaining graph and continue the same way.

* always take the vertex with no predecessors

Finally, we either process all vertices and end up with the topologically sorted list, or we cannot get a vertex with no predecessors, which means we have a cycle. Indeed, if, at some point, we cannot get a vertex with no predecessors, we can prove that the remaining graph at that point has a cycle. Take a vertex, take one of its predecessors (at least one exists), take a predecessor of its, and so on, obtaining an infinite sequence. But the set of vertices is finite, so, we must have repeating vertices, i., e., a cycle.

It remains to get an efficient way to implement finding vertices with no predecessors and removing them from the graph. Here, the idea is to not actually remove vertices, but to keep, for each vertex, a counter of predecessors still in the graph. The algorithm follows:

```
Input:
    G : directed graph
Output:
    sorted : a list of vertices in topological sorting order, or null if G has cycles
Algorithm:
    sorted = emptyList
    Queue q
    Dictionary count
    for x in X do
        count[x] = indeg(x)
        if count[x] == 0 then
            q.enqueue(x)
        endif
    endfor
    while not q.isEmpty() do
        x = q.dequeue()
        sorted.append(x)
        for y in Nout(x) do
            count[y] = count[y] - 1
            if count[y] == 0 then
                q.enqueue(y)
            endif
        endfor
    endwhile
    if sorted.size() < X.size() then
        sorted = null
    endif
```

Depth-first search based algorithm

This is based on the Murphy's law *whatever you're starting to do, you realize something else should have been done first.* Only that, when we discover that, we do that something and, finally, do our activity. This leads to the following simplified algorithm:

```
do(x):
    for y in Nin(x)
        if y not yet done then
            do(y)
        endif
    endfor
    actually do x
```

Performing the above requires:

- a list where to store vertices on *actually do x*;
- a fast way to verify if an activity was performed (this would be a set with the same content as the sorted list, but with quicker access by value);
- a way to detect cyclic dependencies; for this, we will detect whenever performing *do(x)* invokes again *do(x)* before doing *actually do x*.

The result is:

```
Input:
    G : directed graph
Output:
    sorted : a list of vertices in topological sorting order, or null if G has cycles
Subalgotithm TopoSortDFS(Graph G, Vertex x, List sorted, Set fullyProcessed, Set 
inProcess)
    inProcess.add(x)
    for y in Nin(x)
        if y in inProcess then
            return false
        else if y not in fullyProcessed then
            ok = TopoSortDFS(G, y, sorted, fullyProcessed, inProcess)
            if not ok then
                return false
            endif
        endif
    endfor
    inProcess.remove(x)
    sorted.append(x)
    fullyProcessed.add(x)
    return true

Algorithm:
    sorted = emptyList
    fullyProcess = emptySet
    inProcess = emptySet
    for x in X do
```

```
        if x not in fullyProcessed then
            ok = TopoSortDFS(G, x, sorted, fullyProcessed, inProcess)
            if not ok then
                sorted = null
                return
            endif
        endif
    endfor
```

DAGs, strongly connected components, and preorder relations

Property: A directed graph is a DAG if and only if it has no loops and each of its strongly connected components consists in a single vertex.

Proof: A DAG obviously cannot have loops. In addition, if there are two distinct vertices *x* and *y* in the same strongly connected component (SCC), then there is a path from *x* to *y* and a path from *y* to *x* and those paths together form a cycle; therefore, in a DAG, any SCC can have at most 1 vertex. For the other way round, let's prove that a graph with no loops and with only 1-vertex SCCs is DAG. Suppose the contrary, that we have a cycle. If the cycle has length 1, it is a loop. If the cycle is longer, it has at least 2 vertices, which lie in the same SCC. Thus, we have a contradiction.

Note the similarity between the topological sorting DFS-based algorthm and the algorithm for determining the SCCs. This is not a coincidence and, moreover, the SCC algorithm finds the SCC in a topological order, in the *condensed graph* defined below.

Given a graph *G* that may have cycles, we can construct the *condensed graph G'* as follows: each SCC of *G* appears as a vertex of *G'*, and we put an edge *(A, B)* in *G'* if and only if there is at least an edge in *G* between a vertex of component *A* and a vertex of component *B*.

It is easy to see that *G'* is a DAG. Moerover, the SCC algorithm determines the SCCs in a topological order with respect to *G'*.

Scheduling problem

Input: you are given a list of activities to be done for a project, and each activity has a list of prerequisite activities and a duration

Output: a scheduling of the activities (the starting and the ending time for each activity). If activity B depends on activity A, then B must start when or after A ends; however, two activities that do not depend on each other can be executed in parallel.

The goal is to execute the project as quickly as possible - from the time the first activity or activities start, to the time the last activity or activities end.

There may be several valid scheduling, all yielding the same total project duration. Two of them are more interesting:

- *Earliest scheduling,* where every activity is scheduled as early as possible;
- *Latest scheduling,* where every activity is scheduled as late as possible - but while keeping the project finish as early as possible;

Example

Act.	Prerequisites	Duration	Earliest	Latest
0	-	1	0-1	1-2
5	-	2	0-2	0-2
6	0,5	5	2-7	2-7
4	5	1	2-3	6-7
1	6	2	7-9	8-10
3	4,6	2	7-9	7-9
2	3,6	1	9-10	9-10

Graph algorithms - Trees

Definition and properties

Definition: A *tree* is an undirected graph that is connected and has no cycles

- We understand by *cycle* a closed walk with no repeating vertices (except that the first and the last vertex are the same) and no repeating edges. This means that, if there is an edge between vertices 1 and 2, the walk (1, 2, 1) is not a cycle because it uses the same edge twice (once in each direction).
- The smalles tree that fits the definition consists in a single isolated vertex.
- There is a big difference between *non-rooted trees* considered here and the *rooted trees* used especially for data structures. Any tree becomes a rooted tree by distinguishing any of its vertices as root and directing all edges away from the root. Viceversa, any rooted tree becomes a non-rooted tree if we forget the distinguished vertex and the parent-child direction of edges.
- For data structures, we most often distinguish an order among the children of a vertex. Thus, there are rooted tree with no order among children (we simply have a root and all edges directed away from it), and rooted tree with order on children (where, in addition, we distinguish an order among the children). For binary trees, in addition, we sometimes distinguish between a node with only a left child and a node with only a right child (this is the case for binary search trees, for example). All these kind of trees are distinct. Bottom line: trees studied here are non-rooted and there is no order among the neighbours of any given node.

Equivalent properties for an undirected graph:

1. The graph is a tree;
2. The graph is connected and has at most *n-1* edges (where *n* is the number of vertices);
3. The graph has no cycles and has at least *n-1* edges;
4. The graph is connected and minimal (if we remove any edge, it becomes non-connected);
5. The graph has no cycles and is maximal (if we add any edge, it closes a cycle);
6. For any pair of vertices, there is a unique path connecting them.

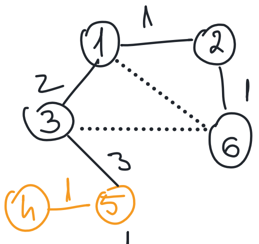
The minimum spanning tree problem

Given a graph with non-negative costs, find a tree with the same vertices and a subset of the edges of the original graph (a *spanning tree*) of minimum total cost.

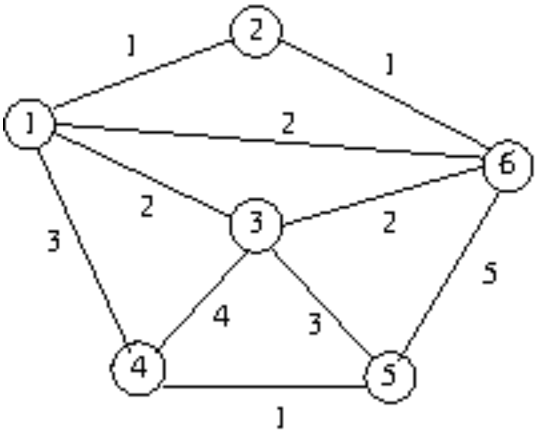
Example: input graph:

Sorted

~~(1,2)~~	~~1~~
~~(2,6)~~	~~1~~
~~(4,5)~~	1
(1,6)	2
~~(1,3)~~	~~2~~
(3,6)	2
~~(3,5)~~	3
(1,4)	3
(3,4)	4
(6,5)	5



done



There are two well-known algorithms for solving this problem: Kruskal's algorithm and Prim's algorithm.Kruskal's algorithm

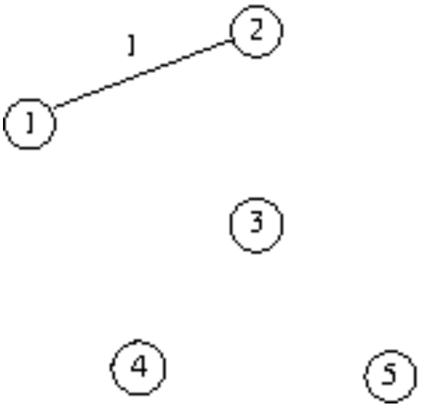
Idea

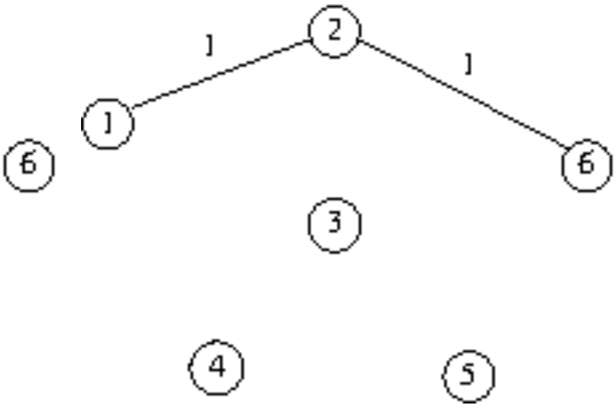
The idea is to start with a graph with all the vertices and no edges, and then to add edges that do not close cycles. This way, as the algorithm progresses, the graph will consist in small trees (it will be what is called a *forest* - a graph with no cycles, meaning that its connected components are trees), and those trees are joined together to form fewer and larger trees, until we have a single tree spanning all the vertices. In doing all the above, we use the edges in increasing order of their cost.

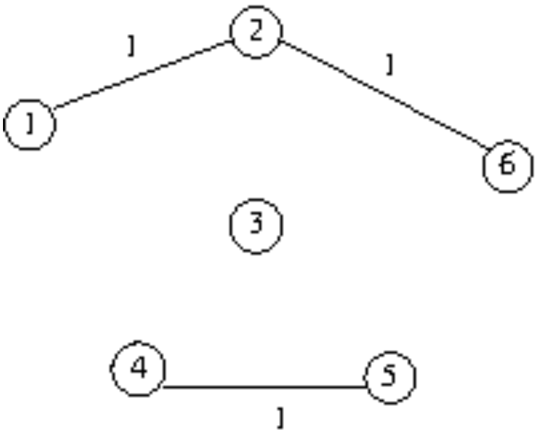
The basic algorithm

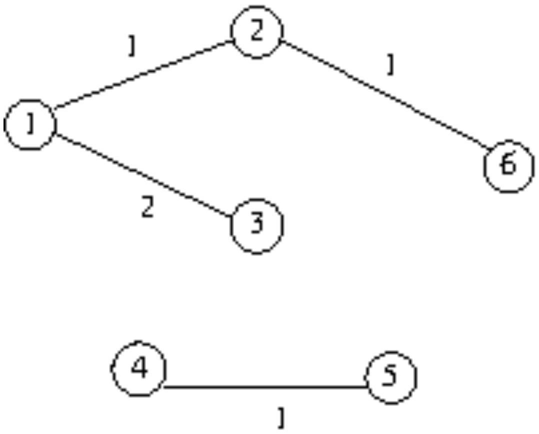
```
Input:
    G : undirected graph with costs
Output:
    edges : a collection of edges, forming a minimum cost spanning tree
Algorithm:
    $e_0,...,e_{m-1}$ = list of edges sorted increasing by cost
    edges = ∅
    for i from 0 to m-1 do
        if edges ∪ {$e_i$} has no cycles then
            edges.add($e_i$)
        end if
    end for
```

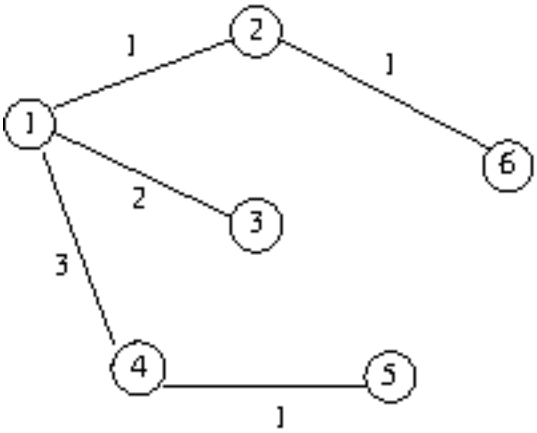
Edge	Cost
1-2*	1
2-6*	1
4-5*	1
1-6	2
1-3*	2
3-6	2
1-4*	3
3-5	3
3-4	4
5-6	5











Issue with the basic algorithm

The difficult part here is how to test the existence of cycles. There is a much easier way: to keep track of the connected components of `edges`, and to notice that a cycle is formed when adding a new edge if and only if the endpoints of the edge are in the same component.

Keeping track of the connected components is an interesting problem in itself.

Ideas:

- Each component is kept as a rooted tree (independent of the tree of the original graph);
- Each component has a *representative* vertex that is the root of the tree; each vertex in the component has a pointer to its parent;
- Therefore, to test if two vertices are in the same component, we go from each of them up to the representative of its component, and we verify if we reach the same vertex;
- When joining two components, we place the representative of one component as a child of the representative of the other. This way, all vertices in the first component are moved to the second.
- There are 2 optimizations to prevent the height of the rooted tree to increase:
  - When joining two components of different heights, the representative of the shortest is set as child of the representative of the tallest (never vice-versa);
  - When retrieving the representative, we *compress the path*, that is, we place all the vertices along the path to the representative as direct children of the representative.

Proof of correctness

The proof is a clasical proof for a greedy algorithm: we compare the Kruskal's solution with the optimal solution for the problem, find the first difference, and modify the optimal solution, without loosing the optimality, so that to match better the Kruskal's solution. By repeating the above step, we turn the optimal solution into Kruskal's solution without loosing the optimality, thus proving that Kruskal's solution is optimal.

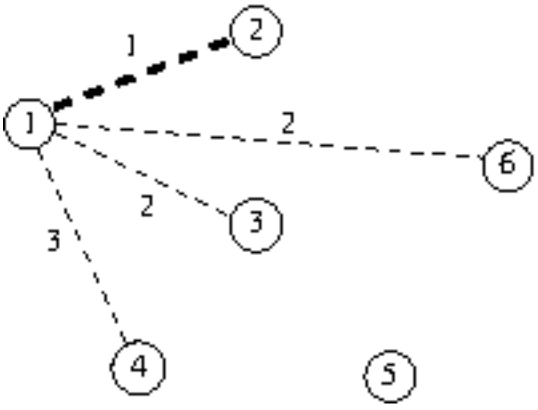
Prim's algorithm

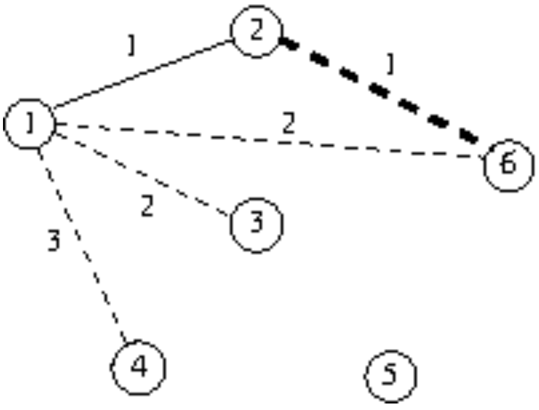
Idea

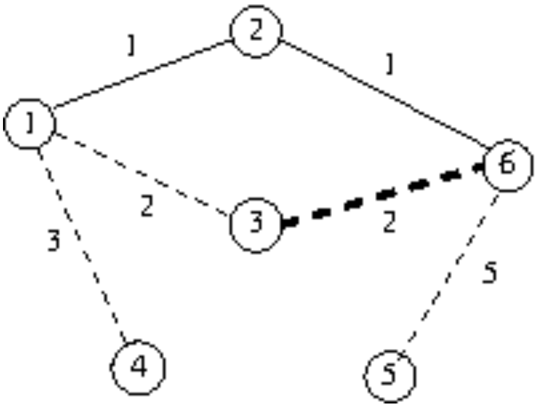
Prim's algorithm is similar to Kruskal's algorithm; however, instead of starting with lots of trees and joining them together, Prim's algorithm starts with a single tree consisting in a single vertex, and then grows it until it covers all the vertices. At each step, an edge is added, connecting an exterior vertex to the tree. Among all the edges connecting a vertex outside the tree with one in the tree, it is choosen the edge of smallest cost.

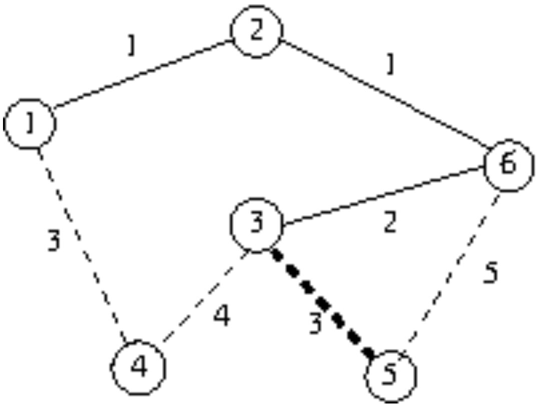
The algorithm

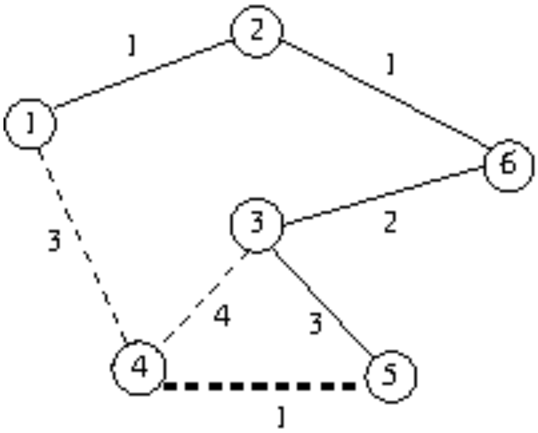
```
Input:
    G : directed graph with costs
Output:
    edges : a collection of edges, forming a minimum cost spanning tree
Algorithm:
    PriorityQueue q
    Dictionary prev
    Dictionary dist
    edges = ∅
    choose s in X arbitrarily
    vertices = {s}
    for x in N(s) do
        dist[x] = cost(x, s)
        prev[x] = s
        q.enqueue(x, d[x])              // second argument is priority
    while not q.isEmpty() do
        x = q.dequeue()      // dequeues the element with minimum value of priority
        if x ∉ vertices then
            edges.add({x, prev[x]})
            vertices.add(x)
            for y in N(x) do
                if y not in dist.keys() or cost(x,y) < dist[y] then
                    dist[y] = cost(x, y)
                    q.enqueue(y, dist[y])
                    prev[y] = x
                end if
            end for
        end if
    end while
```






1
2
1
1
6
2
3
3
2
5
4
5


1
2
1
1
6
3
3
2
5
4
3
4
5


1
2
1
1
6
3
3
2
4
3
4
5
1

Graph algorithms - NP-hard problems

Minimum cost walk or path with negative cost cycles

Minimum cost *walk* problem

Looking for the minimum cost walk between two given vertices, assuming that at least one walk exists:

if the graph contains only positive cost cycles (or no cycles at all)
    the minimum cost walk between two vertices exists and is a path
if the graph contains only positive and zero cost cycles
    the minimum cost walk between two vertices exists; additionally, there is always a path with the same cost
there is at least one negative cost cycle
    the minimum cost walk between two vertices may not exist, because the set of costs of the walks may be unbounded towards -∞

Minimum cost *path* problem - dynamic programming approach

The concatenation of two paths is not necessary a path - the two paths that get concatenated may have some vertices in common.

To use the dynamic programming approach, one needs to parametrize the table with the set of used vertices, in addition to the last vertex:

w[k,x,S] = the cost of the minimum cost path from the starting vertex *s* to vertex *x*, of length *k*, and using only the vertices in the set *S*.

$w[k,x,S] = \min_{y \in Nin(x) \cap S}(w[k-1,y,S \setminus \{x\}])$

The problem is that the number of *w* values to compute grows exponentially fast with the size of the graph.

P, NP, and NP-complete problems

Crash course on Turing machine

What is the turing machine

A Turing machine has:

- a *finite state machine* - at each moment, the machine is in one state out of the finite set of states;
- a infinite *tape* with infinitely many cells, each cell containing, at each moment, one symbol out of a finite alphabet (set of possible symbols).
- a *read-write head,* positioned at each moment on one of the cells on the tape.

At each step, depending on the *current state* and on the *symbol in front of the read-write head*, the machine will:

- go to some state;
- write some symbol to the tape (where the head is located);

- move the head one position to the left, one position to the right, or keep it in place.

Start ing the Turing machine

At the beginning

- the Turing machine is in one distinguished state - the initial state
- the input data is encoded on the tape
- all the tape except for a finite part is filled with a distinguished symbol of the alphabet - the *blank* symbol

Stopping the Turing machine

There is one or more *final states* for the Turing machine. The machine stops when it gets into either of the final states

The stopping state can be used for the output of the execution - for example, there can be two final states: a *yes* state and a *no* state.

Also, the content of the tape at the end may be interpreted as the output of the execution.

Complexity classes

P class of problems

The execution time is the number of steps until reaching the final state.

The execution time is compared to the size of the input data, that is, the number of symbols used for encoding the input data on the tape.

Problems are classified according to the complexity of the best algorithm for solving them (the best Turing machine that solves them).

The class *P* consists in all problems for which there is a Turing machine and a polynomial such that the machine solves the problem and the number of steps is bounded by the polynomial applied to the size of the input data.

Non-deterministic Turing machine

For a *non-deterministic Turing machine*, there are several possible next actions (next state, symbol written on the tape, and the movement of the head) for a single current state (state of the machine, plus the symbol on the tape).

Thus, there are multiple executions possible. (The number of executions grows exponentially fast with the number of steps.)

For yes/no problems, the link between the executions and the answer is the following:

- the machine has 2 final states: *ok* and *fail*;
- if at least one execution finished in *ok*, the answer to the problem is considered *yes*;
- if all executions finish in *fail*, the answer to the problem is considered *no*.

The class *NP* contains all yes/no problems for which there is a non-deterministic Turing machine that solves the problem in polynomial time.

Obviously, $P \subseteq NP$.

Note that *yes* and *no* are not symmetrical to each other. The class Co-NP contains the problems whose inverse (that is, interchanging *yes* with *no*) are in NP.

NP problems

Generally, a NP problem is a problem for which a *yes* answer means there is a "solution" that is a vector of polynomial size (polynomial in the size of the input) and whose correctness can also be checked in polynomial time.

Examples:

- given a graph and two vertices, is there a path from the first to the second?
- given a graph, is there a cycle in it?
- given a graph, is there a Hamiltonian cycle in it?
- given a graph with costs, two vertices, and an integer *k*, is there a path of cost at most *k* between the vertices?
- given a graph with costs, and an integer *k*, is there a Hamiltonian cycle of cost at most *k*?

Polynomial reducibility

Problem A *reduces* to problem B if there is a way of solving A as follows:

- the input for A is transformed, through a polynomial-time algorithm, into a valid input for B;
- a solution for B is applied;
- the output from B is transformed, through a polynomial-time algorithm, into an output for A;
- the result from the above sequence is the correct answer of the original problem A.

If A reduces to B it means that A is not (much) more complex than B. In particular, if A reduces to B and B is polynomial (belongs to P), then A is polynomial, tool.

This also means that, if A reduces to B and A is known to be hard, then B is hard, too. If B is not polynomial and A reduces to B, then A is not polynomial, either.

Note: DO NOT apply the above in reverse. If A reduces to B and B is known to be hard, this does not say anything about A. It only says that there is an expensive way to solve A (by reducing it to the hard problem B); but nothing prevents the existence of a better solution for A.

NP hard and NP complete problems

An NP-hard problem is a problem such that all NP problems reduce to it.

An NP-complete problem is a problem that is both NP and NP-hard.

The first problem to be proven NP-complete is the *boolean satisfiability* (SAT) problem (Cook-Levin theorem, 1971): given a boolean expression in normal conjunctive form, is there an assignment for the variables such that the expression has the value *true*?

$E = (x_{1,1} \vee x_{1,2} \vee ... \vee x_{1,k1}) \wedge (x_{2,1} \vee x_{2,2} \vee ... \vee x_{2,k2}) \wedge ... \wedge (x_{n,1} \vee x_{n,2} \vee ... \vee x_{n,kn})$,

where each variable is either one of the input variables or its negate.

It is not known whether P = NP or not. This is a million-dollar open problem!

Known NP-complete problems

3SAT

3SAT is a special case of SAT where the disjunctions are limited to 3 terms.

SAT is reducible to 3SAT by replacing each longer disjunction
$x_1 \vee x_2 \vee ... \vee x_k$
by a conjunction of size 3 disjunctions containing newly introduced variables:
$(x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge (\neg y_2 \vee x_4 \vee y_3) \wedge ... \wedge (\neg y_{k-3} \vee x_{k-1} \vee x_k)$

Hamiltonean cycle and friends

The existence of a Hamiltonean cycle is proven to be NT-complete

Note: this is both in a directed graph and in an undirected graph. There is an interesting way of reducing the directed Hamiltonean cycle problem to the undirected Hamiltonean cycle problem. TBA

Traveling Salesman Problem (TSP) can be phrased as a yes/no problem by putting an upper limit on the cost: given a (directed) graph with costs, and an integer *k*, is there a Hamiltonean cycle of cost at most equal to *k*?

The Hamiltonean cycle problem reduces to TSP, even to TSP in a complete graph. Simply put a cost of 1 on edges that exist in the original graph and 2 on those that do not exist in the original graph, and find a solution of cost *n* to TSP.

Now we can show that the minimum cost path problem, in the general case where negative cost cycles may exist, is NP-hard. Indeed, the Hamiltonean path problem can easily reduce to it.

Note: for TSP on an undirected graph satisfying the triangle inequality, there is an approximate solution no worse than twice the optimal cost. Build a Minimum Spanning Tree and parse it in pre-order for the solution.

Clique, independent set, vertex cover

A *clique* in an undirected graph is a subset of vertices of a graph such that the induced subgraph is complete (for every pair of vertices in the clique, there is an edge between them).

The *k*-clique problem is: given a graph and an integer *k*, is there a clique of size *k*.

An *independent set* in an undirected graph is a subset of vertices of a graph such,for every pair of vertices in the set, there is an no edge between them.

Edge cover: given an undirected graph, find a (minimum) set of vertices such that every edge has at least one endpoint in the set.

There is a simple reducibility relation between these 3 problems!

Other problems

A *vertex cover* in an undirected graph is a subset *A* of vertices such that any vertex of the graph is either a member of *A* or a direct neighbor of a vertex in *A*.

k-coloring: given an undirected graph and an integer *k*, assign to each vertex a number in {1,2,...,k} (a "color") such that any two adjacent vertices have distinct numbers associated to them (distinct colors).

3-way matching: given 3 sets X, Y and Z, disjoint and all having the same number of elements, and a set T ⊆ X × Y × Z of triplets, find a subset U ⊆ T such that each element of X, Y, and Z appears in exactly one triple in U.

Graph algorithms - Flows

Transport graph and maximum flow

In a transport graph, we have a source vertex, where a producer of some commodity is located, a destination vertex, and all the other vertices act as intermediates. Each edge represents the possibility to transport a certain amount of that commodity; the amount is the *capacity* of that edge.

The goal is to plan how to transport a maximum amount of that commodity from the source to the destination.

More formally, we have:

- a directed graph G=(X,E);
- a source vertex *s* and a destination vertex *t*;
- each edge *(x,y)* has a positive capacity *cap(x,y)*;

A *flow* can be established through the graph. A flow is an assignment of a flow value to each edge, such that:

- the flow through each edge is between zero and the capacity of that edge: $0 \leq \text{flow}(x,y) \leq \text{cap}(x,y)$
- for each vertex, except for the source or the destination, the inbound flow is equal to the outbound flow: for any $x \in X$, $\sum_{y\in N^{in}(x)}\text{flow}(y,x) = \sum_{y\in N^{out}(x)}\text{flow}(x,y)$

For the source vertex, there is a positive net outbound flow. That value is called the *total value of the flow*. It is $v_{flow} = \sum_{y\in N^{out}(s)}\text{flow}(s,y) - \sum_{y\in N^{in}(s)}\text{flow}(y,s)$.

It can be easily shown that the total flow value is equal to the net inbound flow into the destination vertex: $v_{flow} = \sum_{y\in N^{in}(t)}\text{flow}(y,t) - \sum_{y\in N^{out}(t)}\text{flow}(t,y)$.

The classical problem to be solved is to set a maximum flow in the transport graph, that is, to maximize the total flow value among all possible flows.

Cuts, capacities, and the flow across a cut

To analize the flow, we need the concept of a *cut*. A cut is, essentially, a partitioning of the vertices into two sets: one containing the source and the other containing the destination. Then, we analyse the capacities and the flow along the edges between vertices in one subset and the other subset.

The *net flow* across the cut is the total "left to right" flow (the total flow along the edges leading from the set containing the source to the set containing the destination), minus the "right to left" flow. Formally, assuming that the cut is (A, X\A), with $s\in A$ and $t\in X\setminus A$:
$\text{flow}(A,X\setminus A) = \sum_{(x,y)\in E,\, x\in A,\, y\in X\setminus A}\text{flow}(x,y) - \sum_{(x,y)\in E,\, x\in X\setminus A,\, y\in A}\text{flow}(x,y)$

The capacity of the cut is, however, only the "left to right" capacity: $\text{cap}(A,X\setminus A) = \sum_{(x,y)\in E,\, x\in A,\, y\in X\setminus A}\text{cap}(x,y)$

It is clear that, for any cut, the flow across the cut is less than or equal to the capacity of the cut. The maximum flow is obtained when all "right to left" edges are saturated and all "left to right" edges have zero flow.

On the other hand, the flow across any cut is the same, and is equal to the total value of the flow. (Actually, the total value of the flow is the flow across a cut that has only the source vertex on the "left", and all other vertices on the "right".)

The naïve algorithm

A flow of zero everywhere is clearly a valid flow. Starting from it, we can increase the flow while keeping it valid by the following approach:

- find a path from the source to the destination, consisting only of non-saturated edges;
- compute the *capacity of the path* as being the smalles of the residual capacities of its edges (the residual capacity of an edge is the difference between the capacity and the current flow through that edge);
- increase the flow, on all the edges of the path, with a value equal to that capacity.

It is clear that, by following the steps above, the flow remains valid. However, we may end up with a flow that cannot be increased by this approach, yet a flow of larger total value still exists.

Ford-Fulkerson algorithm and Ford-Fulkerson theorem

A correct algorithm can be devised starting from the (incorrect) naïve algorithm. This algorithm (Ford-Fulkerson) is the following:

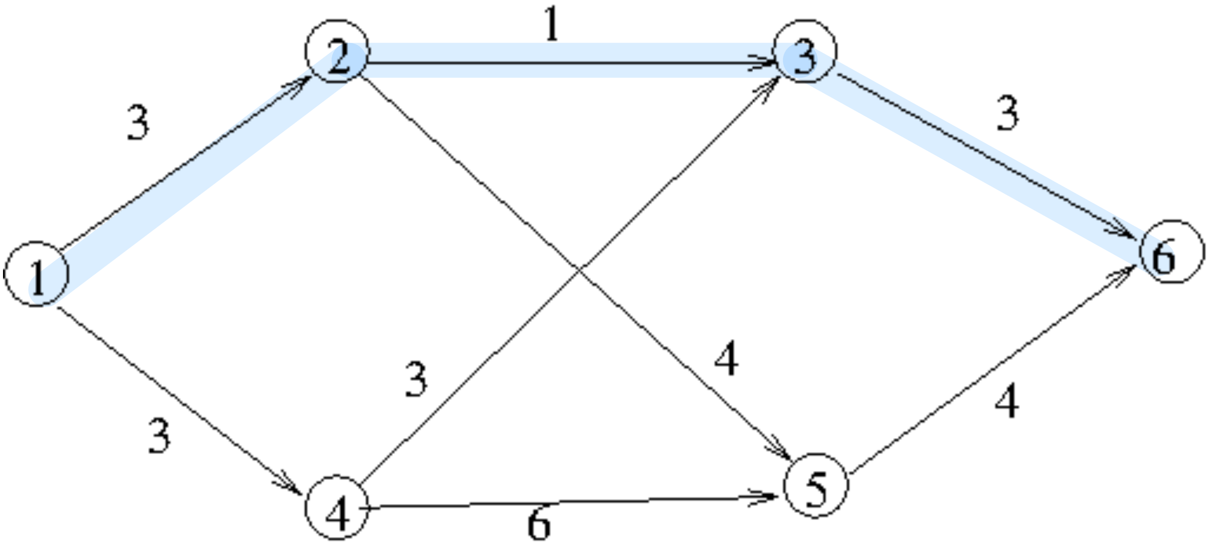
1. again, start with a zero flow
2. for the current flow, construct a *residual graph* containing the same vertices as the original graph, but:
    - for each non-saturated edge of the original graph, put an edge with the same direction and with the remaining capacity (the capacity is the difference between the original capacity and the current flow through that edge)
    - for each edge with non-zero flow, put an edge in the reverse direction and with a capacity equal to the value of that flow (this edge signifies that the flow can be increased in the reverse direction by decreasing the forward flow). Note that this is the only difference compared to the naïve algorithm.
3. find a path from source to destination in the residual graph;
4. compute the capacity of the above path;
5. update the flow: for forward edges, increase the flow by a value equal to the capacity of the path; for backward edges, decrease the flow on the corresponding forward edges by the same value.
6. repeat the steps 2-5 until no path can be found in the residual graph from source to destination

To show that the algorithm produces the optimal flow, consider the cut having on the left-hand side all the vertices that are accessible from the source in the residual graph (since there is no path to destination, the destination is on the right-hand side of the cut). No edge can exist in the residual graph across the cut from left to right. Because of the way the residual graph was constructed, it follows that the left-to-right edges in the original graph are saturated and the right-to-left edges have zero flow. So, that cut is saturated. So, no larger flow can exist.

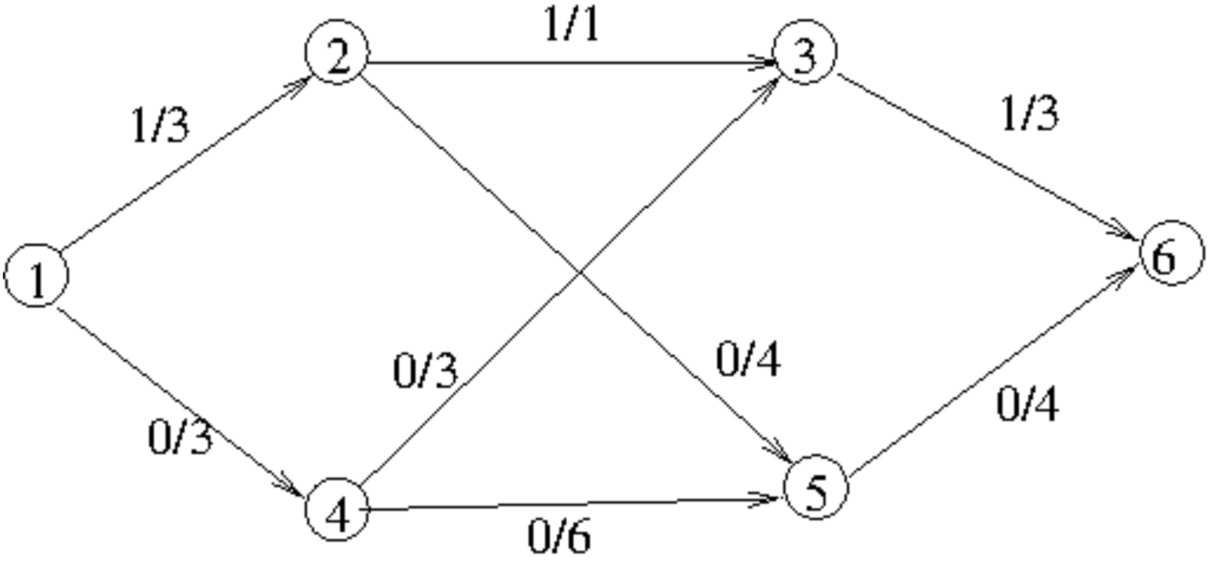
It follows that the value of the maximum flow is equal to the capacity of the minimum cut. This statement is called the Ford-Fulkerson theorem.

Example

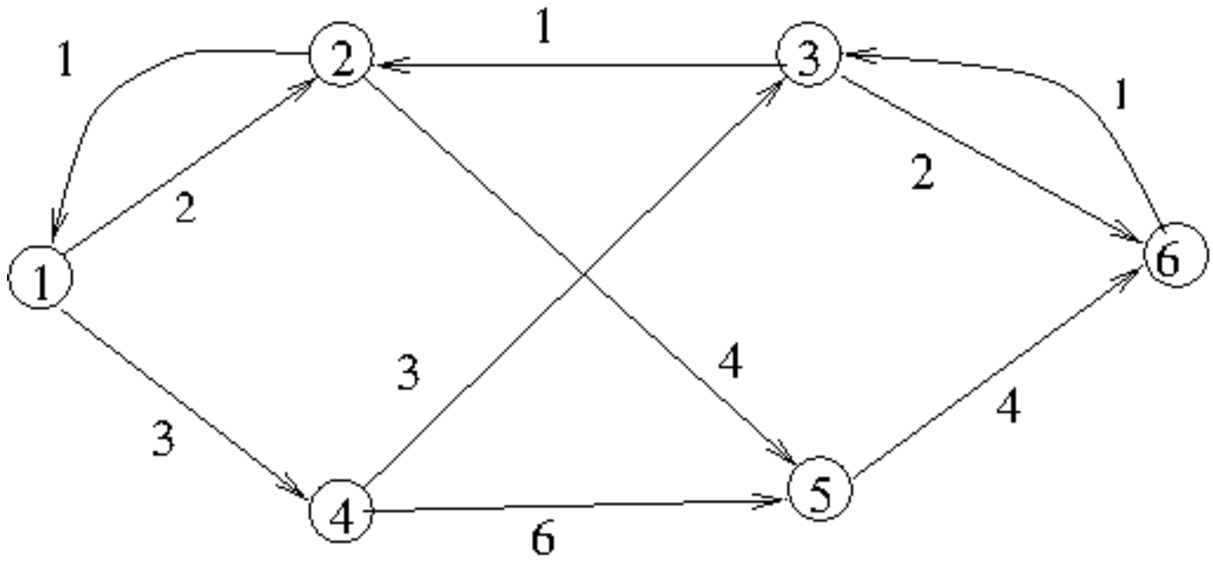
Input graph:



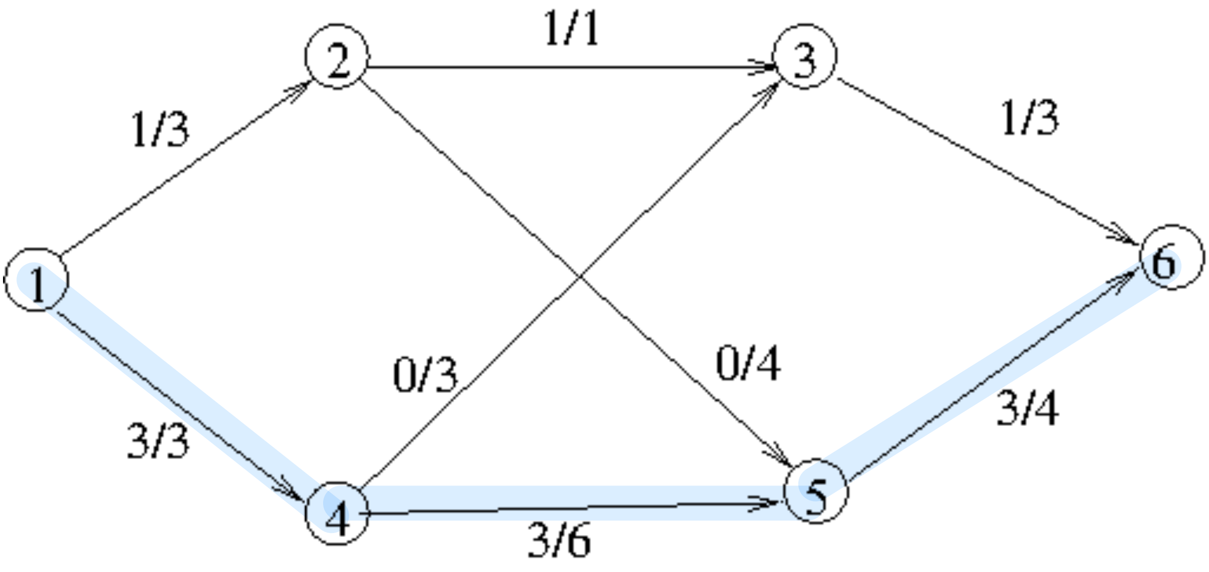
After augmenting path 1,2,3,6 (capacity = 1):



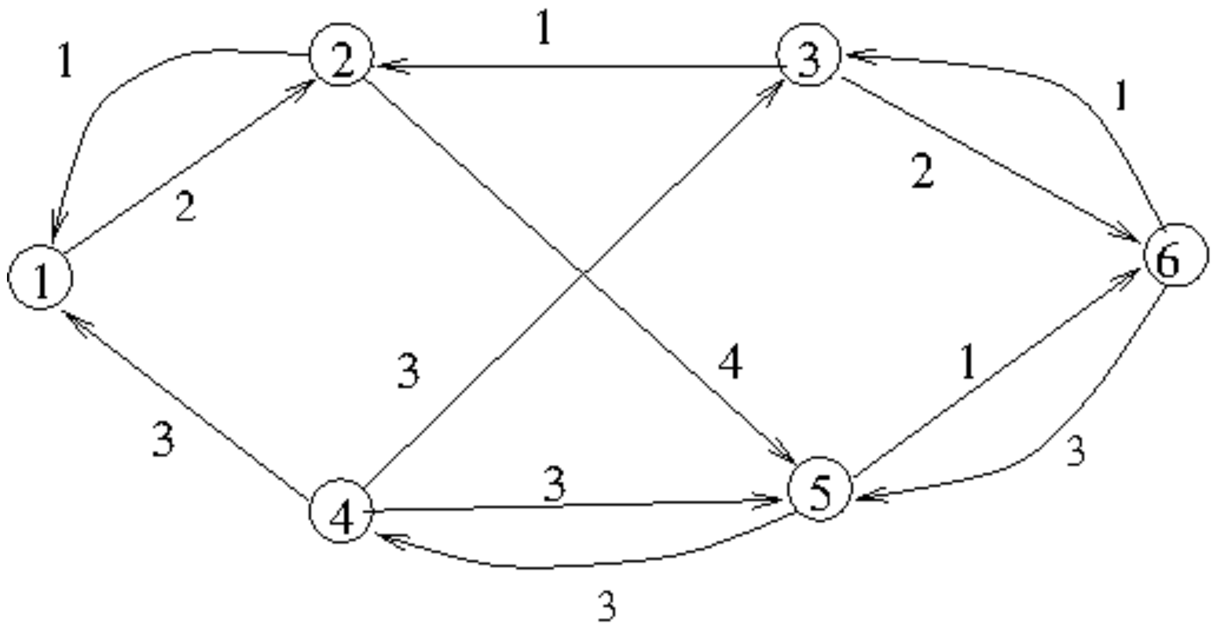
with residual graph



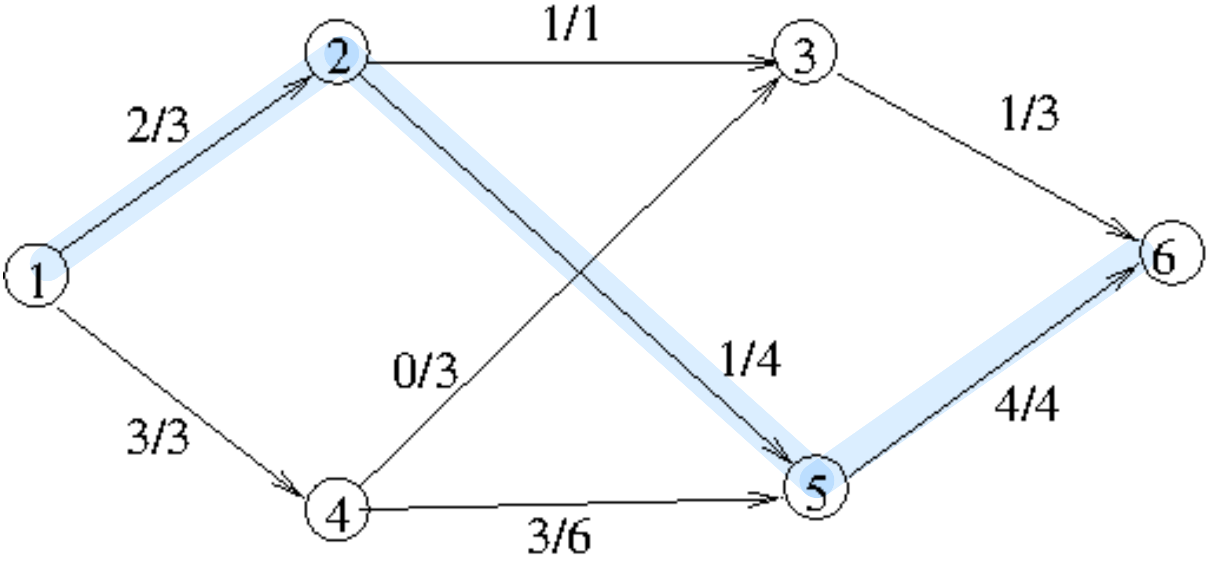
After augmenting path 1,4,5,6 (capacity = 3):



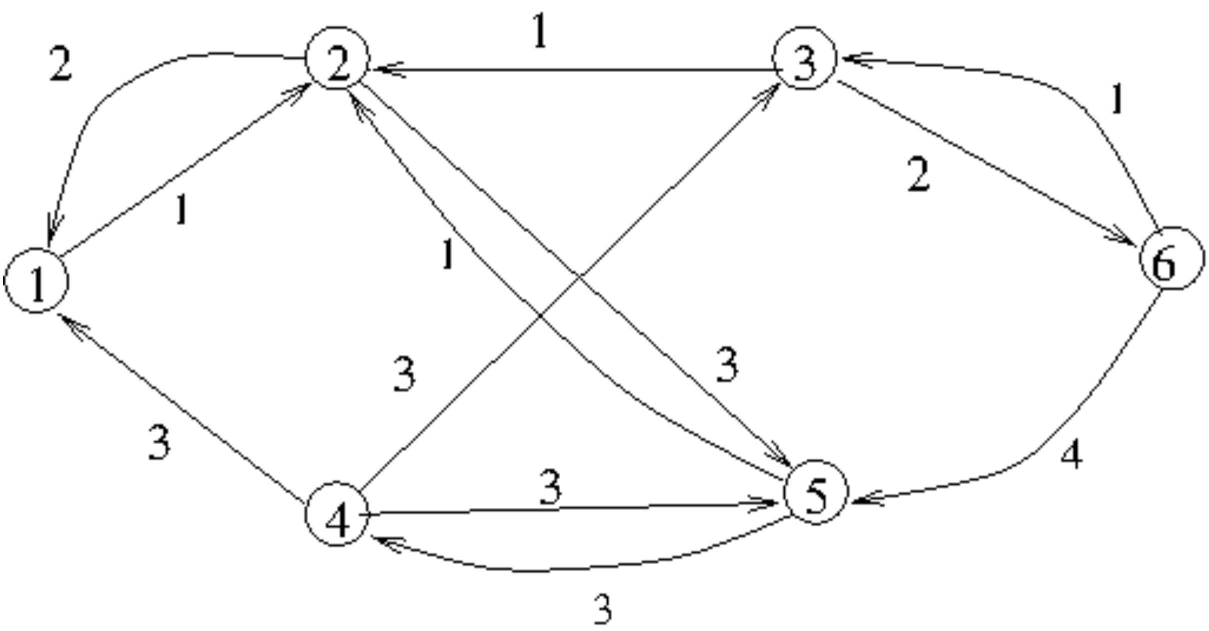
with residual graph



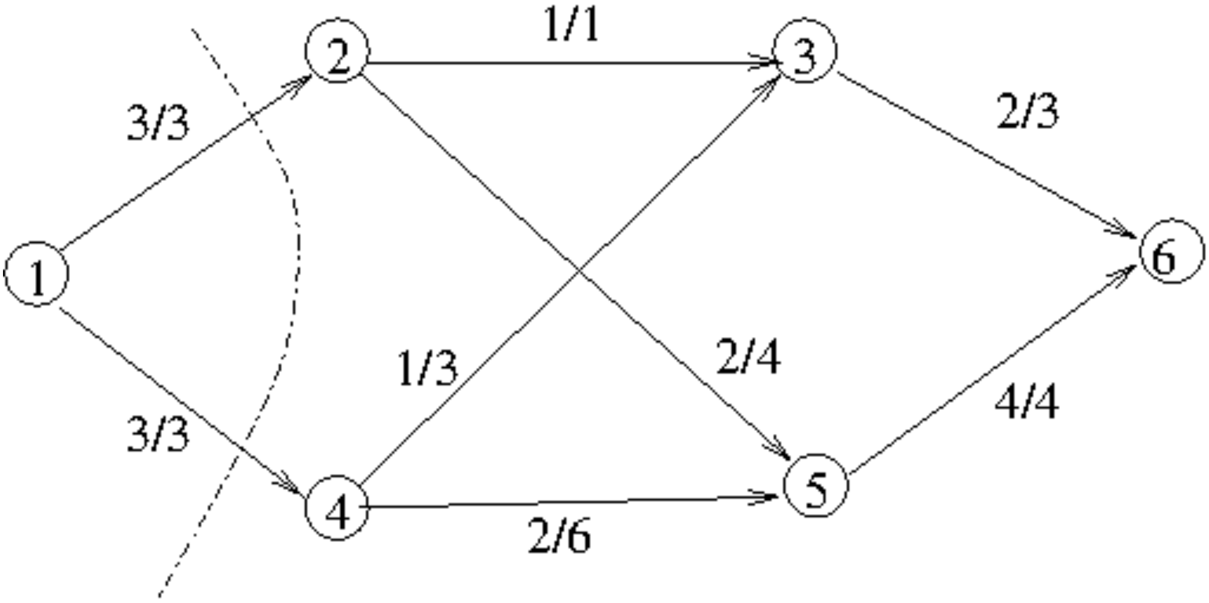
After augmenting path 1,2,5,6 (capacity = 1):



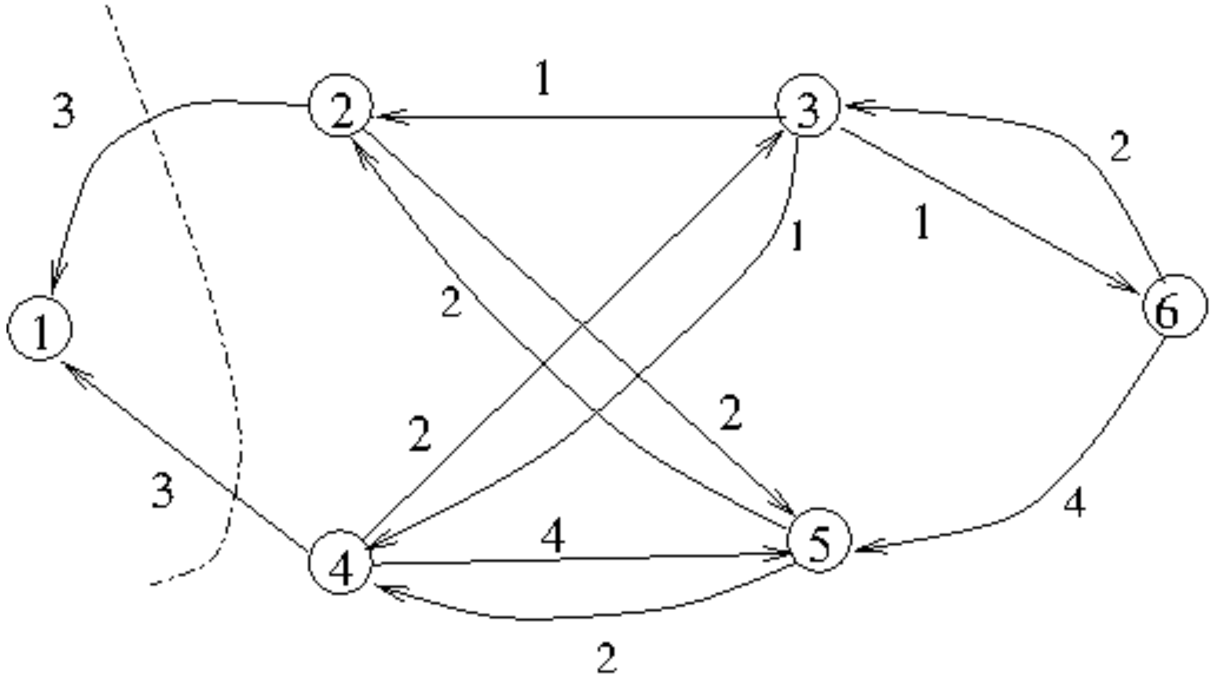
with residual graph



The naïve algorithm stops here. Ford-Fulkerson, however, finds the augmenting path 1,2,5,4,3,6 (capacity = 1):



with residual graph



No augmenting path can be found any more. The cut {1} to {2,3,4,5,6} is saturated (capacity=6, flow=6).

Graph algorithms - Maximum flow of minimum cost

Maximum flow of minimum cost problem

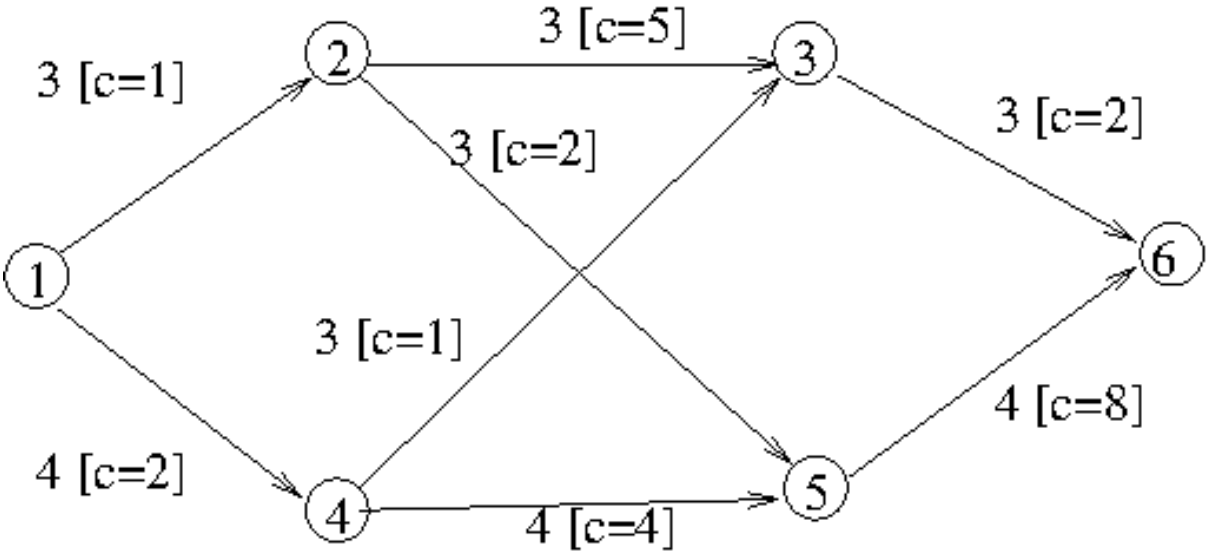
As for the maximum flow, we are given a transport network (a directed graph with capacities associated to edges, plus a source and a destination vertex). Additionaly, each edge has a cost.

In addition to the maximum flow problem, a flow also has a cost. The cost is the sum, for all edges, of the flow along the edge multiplied by the cost of the edge.

In other words, the cost of an edge is the cost of transporting each unit of flow along that edge.

The goal is to find a maximum flow and, among all possibilities to achieve it, to get one that also minimizes the cost.

Example - input graph



Solution

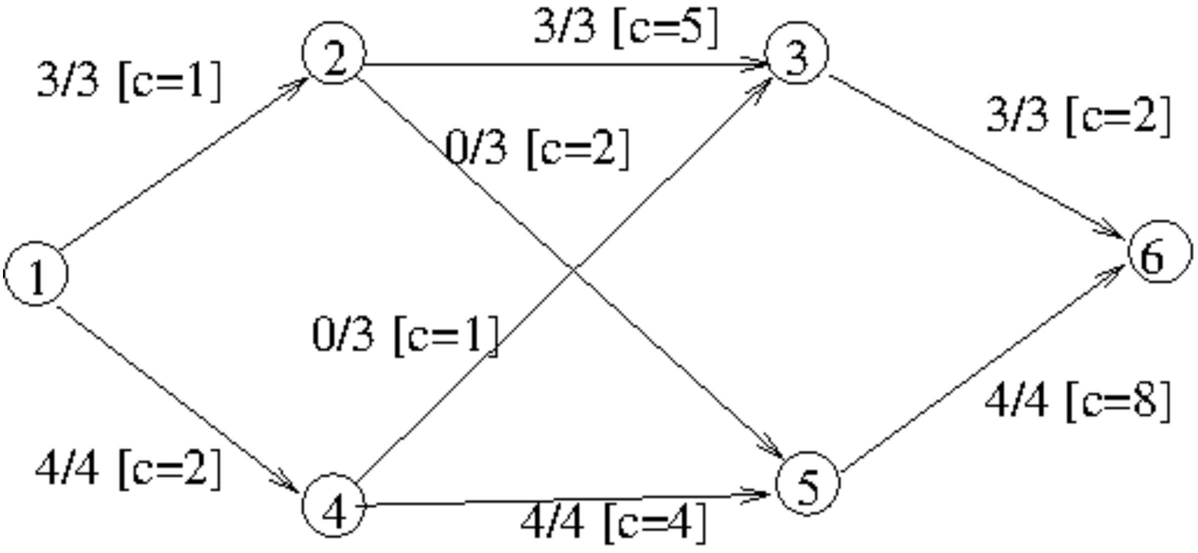
First step is to obtain a maximum flow regardless of the cost. Then we minimize the cost while keeping the value of the flow constant.

We repeat the following steps:

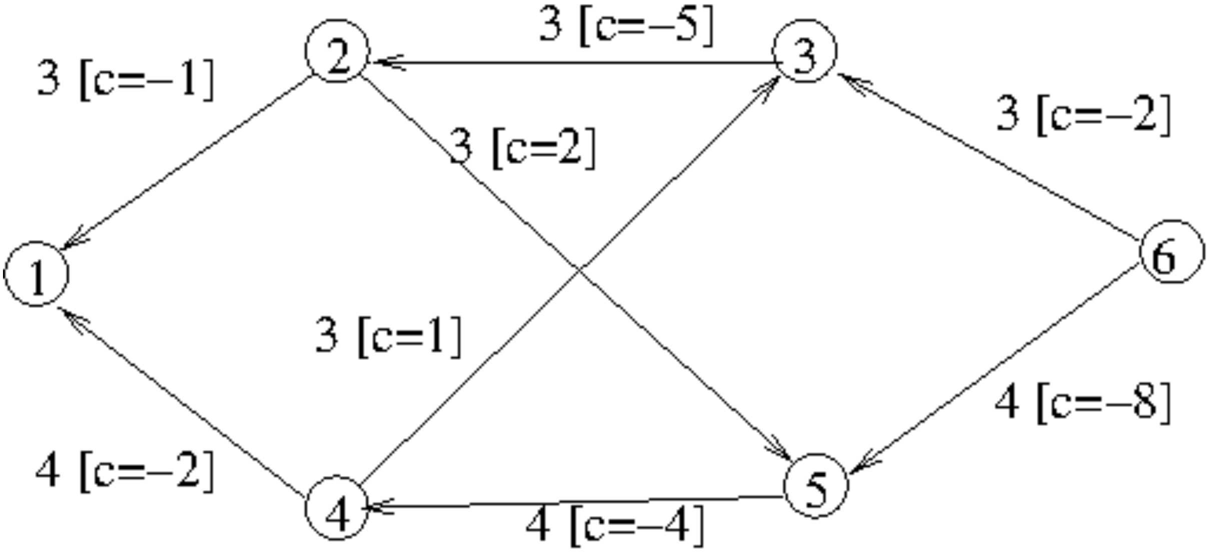
1. construct the residual graph
2. assign costs to edges: the cost of the original edge for the forward residual edges and minus the original cost for the backwards edges
3. find a negative cost cycle in the residual graph
4. increase the flow along the cycle (like for the augmenting paths in Ford-Fulkerson)
5. stop when no negative cost cycle exists any more

Example

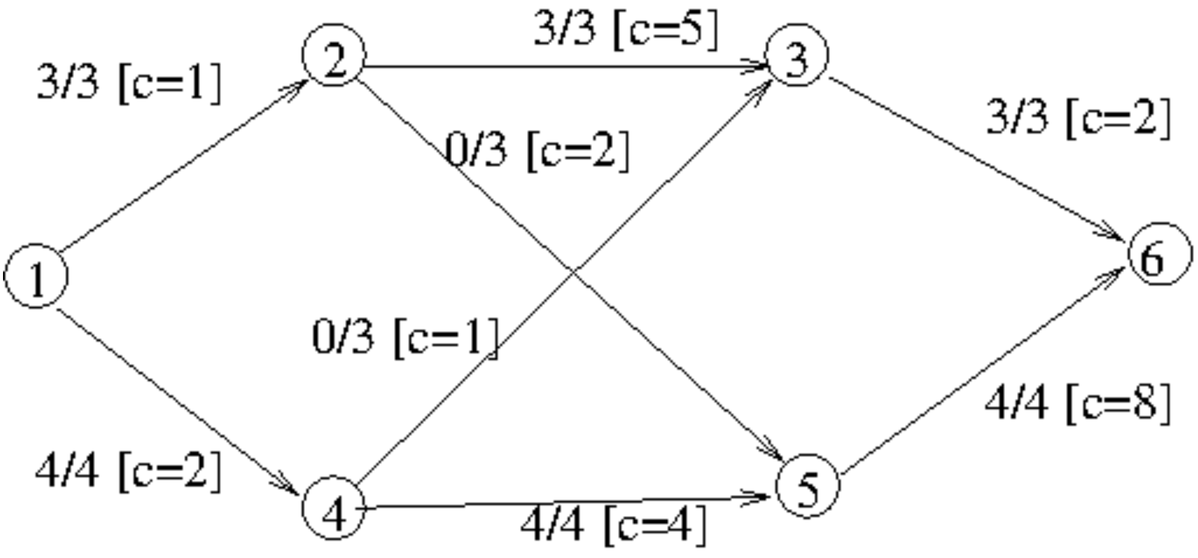
Maximum flow (flow=7, cost=1*3+2*4+5*3+0*2+0*1+4*4+2*3+8*4=80):



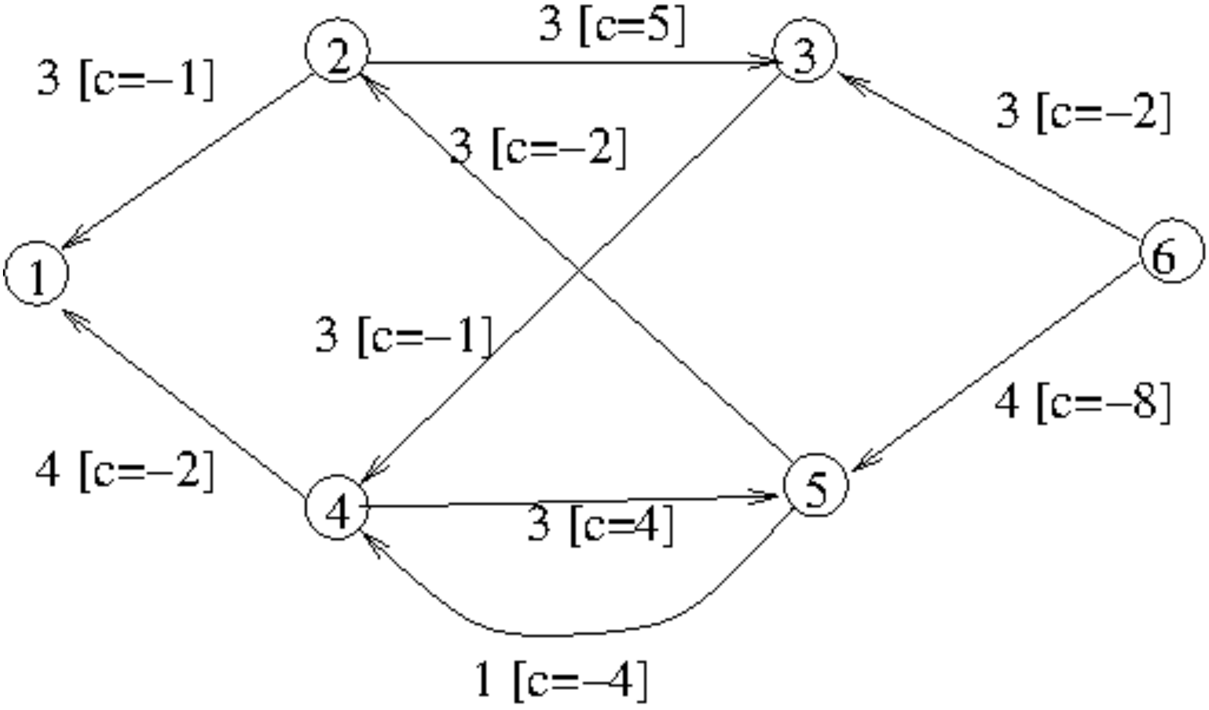
with residual graph



After using negative cost cycle 2, 5, 4, 3, 2 (capacity = 3, cost=-6):



with residual graph



No negative cost cycle can be found. Final flow=7 of cost=80-18=62.