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.

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:
    e0,...,em-1 = list of edges sorted increasing by cost
    edges = ∅
    for i from 0 to m-1 do
        if edges U {ei} has no cycles then
            edges.add(ei)
        end if
    end for
```

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.

TBD

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, dist[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
```

Graph algorithms - Directed acyclic graphs (DAGs)

Basics

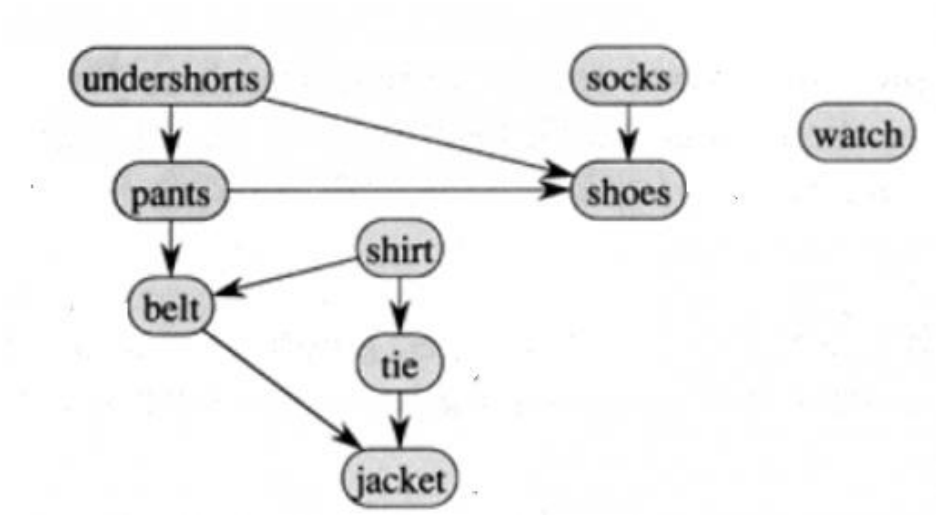
A *directed acyclic graph* (DAG) is a directed graph having no cycle.

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*.

Example: Professor Bumstead gets dressed

The binary dependency relations between different garments are represented in the following directed graph. Each edge (u, v) means that the garment u must be put on before the garment v.



Find a linear order (topological order) of all the garments such that the dependency relations are satisfied.

!!! In the topological order each vertex is after all its predecessors.

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.

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'*.

Topological ordering and acyclic graphs

Define a directed acyclic graph (often known as a DAG for short) to be a directed graph, containing no cycle (a cycle is a set of edges forming a loop, and all pointing the same way around the loop).

Theorem: a graph has a topological ordering if and only if it is a directed acyclic graph.

One direction of the proof is simple: suppose G is not a DAG, so it has a cycle. In any ordering of G, one vertex of the cycle has to come first, but then one of the two cycle edges at that vertex would point the wrong way for the ordering to be topological. In the other direction, we have to prove that every graph without a topological ordering contains a cycle. We'll prove this by finding an algorithm for constructing topological orderings; if the algorithm ever gets stuck we'll be able to use that information to find a cycle.

Algorithm 1: (topological ordering)

```
list L = empty
while (G is not empty)
    find a vertex v with no incoming edges
    delete v from G
    add v to L
```

If this algorithm terminates, L is a topological ordering, since we only add a vertex v when all its incoming edges have been deleted, at which point we know its predecessors are already all in the list.

What if it doesn't terminate? The only thing that could go wrong is that we could be unable to find a vertex with no incoming edges. In this case all vertices have some incoming edge. We want to prove that in this case, G has a cycle. Start with any vertex s, follow its incoming edge backwards to another vertex t, follow its incoming edge backwards again, and so on, building a chain of vertices ...w->v->u->t->s.

We can keep stepping backwards like this forever, but there's only a finite number of vertices in the graph. Therefore, we'll eventually run into a vertex we've seen before: u->w->v->u->t->s. In this case, u->w->v->u is a directed cycle. This procedure always finds a directed cycle whenever algorithm 6 gets stuck, completing the proof of the theorem that a graph has a topological ordering if and only if it is a DAG. Incidentally this also proves that algorithm 6 finds a topological ordering whenever one exists, and that we can use algorithm 6 to test whether a graph is a DAG. Putting algorithm 6 together with the "stepping backwards" procedure provides a fast method of finding cycles in graphs that are not DAGs.

Finally, let's analyze the topological ordering algorithm. The key step (finding a vertex without incoming edges) seems to require scanning the whole graph, but we can speed it up with some really simple data structures: a count I[v] of the number of edges incoming to v, and a list K of vertices without incoming edges.

Algorithm 2: (topological ordering, detailed implementation)

```
list K = empty
list L = empty
for each vertex v in G
    let I[v] = number of incoming edges to v
    if (I[v] = 0) add v to K
while (G is not empty)
        remove a vertex v from K
        for each outgoing edge (v,w)
                decrement I[w]
                 if (I[w] = 0) add w to K
                 add v to L
```