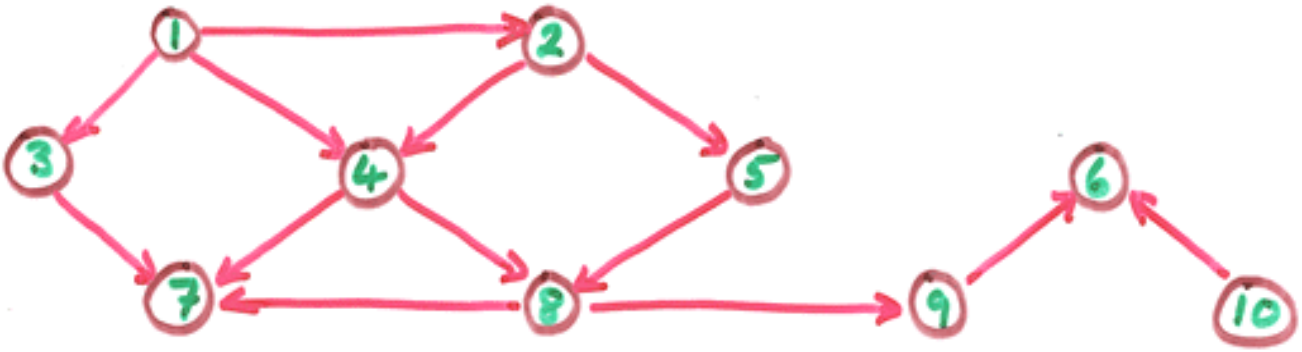
Graphs

Read chapters 8 & 9 of Weiss

The trees we've been looking at are special cases of a more general design — graphs. These are general arrangements of nodes and edges ...



The edges will always be directed, some people call these graphs digraphs. Edges can carry weights or costs, a path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_N$ such that (v_i, v_{i+1}) is an edge (with the correct direction), the length of a path is the number of edges on the path ($N-1$ in our example path), and the cost of a path is the sum of the weights along each edge of the path. A simple path never repeats vertices (except possibly the last may equal the first — called a loop or cycle if the length ≥ 1). As a formal statement, we allow paths of length 0, namely a path from a vertex to itself using no edges. An acyclic graph has no cycles.

A graph is complete if there is an edge between each pair of vertices. If a graph has paths from each vertex to each other vertex, then it is strongly connected; if it's not strongly connected, but the underlying undirected graph is connected, then the graph is weakly connected.

We could represent our graph by a two-dimensional adjacency matrix, where the weight of an edge from vertex k to vertex l is in position (k, l) of the matrix...

This time, assume each edge has weight 1, and we use . to represent the value ∞, for clarity. We use 0 weight to show absent connection.

Notice how wasteful such a sparsely connected graph is in matrix form, both for space and for the initialization cost — essentially $O(n^2)$ where n is the number of vertices. A better arrangement is to build lists of adjacent vertices for each vertex; the nodes in these lists could hold both the vertex 'name' and the edge weight ...

Notice that in this adjacency list, the space requirement is now just O(e), where e is the # edges. Initialization is now O(n), and building the graph is O(e).

Given an acyclic graph, we can write a simple routine to perform a topological sort. This is a non-unique list of vertices such that if there is a path from vertex v to vertex w, then w appears after v in the sorted list. (The reason for disallowing cycles is obvious!!) For example, a topological sort of our example graph could produce ...

1, 2, 4, 5, 8, 3, 7, 9, 10, 6

To do this in a more organized fashion, find any vertex having no incoming edges. Print and remove this vertex (together with its outgoing edges). Repeat until finished. Define the indegree of a vertex v as the # edges (u,v) coming in to v, and now assume that a graph has been read into an adjacency list where each vertex also stores its indegree ...

```
class Vertex
  {
    String name;
    LList adj;          // list of adjacent vertices
    int dist;           // for cost
    Vertex path;        // for previous vertex on shortest path
    boolean known;      // for possible later use
    int indegree; int topNum;

    public Vertex ( String appel)
       { name = appel; adj = new LList (); reset (); }
    public void reset ()
       { dist = Graph.INFINITY; path = null; }
  }
```

Integer.MAX_VALUE

Then our topological sort routine might be ...

```
void topSort() throws CycleFound
  {
    Vertex v, w;

    for (int i = 0; i < NUM_VERTICES; i++)
      {
        v = findZero();
        if (v == null)
          throw new CycleFound();
        v.topNum = i;
        adjustGraph();
      }
  }
```

a method which for each vertex w adjacent to v (i.e., one step out along an edge) performs w.indegree--;

a method to find any vertex of indegree 0 in the current version of the graph which has not yet had a topological number assigned to it.

This is a little wasteful, since findZero() is clearly O(n) and is applied n times, so our algorithm is $O(n^2)$. If the graph is sparse, then only a few vertices will have their indegrees updated in a given iteration. Better would be to store those (unassigned) vertices of indegree 0 separately, and whenever a vertex's indegree becomes 0, that vertex is stored there. We'll use a queue for this storage. This change now makes our algorithm $O(n + e)$.

```
void topSort() throws CycleFound   // O(n+e) version
  {
    Queue q;
    int i = 0;
    Vertex v, w;

    q = new Queue();
```

```
queueZero();
while (!q.isEmpty())
  {
    v = q.dequeue();
    v.topNum = ++i;
    adjGraph();
  }
if (i != NUM_VERTICES)
  throw new CycleFound();
}
```

a method which for each vertex v performs
if (v.indegree == 0)
q.enqueue(v);

a method which for each w adjacent to v performs
if (--w.indegree == 0)
q.enqueue(w);

A second question often raised in the context of graphs is finding the shortest path between two vertices in a graph. This is usually approached in two flavours; where the edges are unweighted, and then where various weights/costs are assigned to the various edges.

Taking our example graph again, we could ask for the lengths of the shortest paths from any given vertex s to every other vertex of the graph. If there is no path to a given vertex in the directed graph, then that 'path' has length 'infinity'. For example, if s is vertex #2, we can find all vertices of shortest path length 0 from 2, then all those of length 1 from these (path length 1), then all those of length 1 from these (path length 2), et similia.

however, it's better to use queues as we did for the topological sort to give an $O(n+e)$ algorithm. (We will assume that dist for each vertex has been reset to the value INFINITY before we start...)

```
void unweighted ( Vertex s )
  {
    Queue q;
    Vertex v, w;

    q = new Queue();
    q.enqueue(s);
    s.dist = 0;

    while (! q.isEmpty())
      {
        v = q.dequeue();
        for (each w adjacent to v)
          if (w.dist == INFINITY)
            {
              w.dist = v.dist + 1;
              w.path = v;
              q.enqueue(w);
            }
      }
  }
```

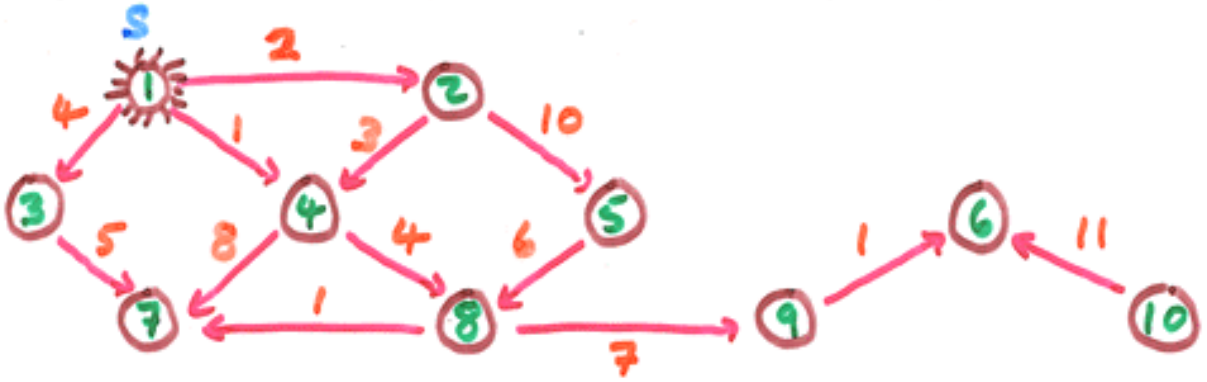
on first pass this sets v = s. Subsequent passes pull previously handled vertices from the queue to deal with their adjacent vertices...

this increments the w.dist for w adjacent to v provided w.dist hasn't previously been given a reasonable value

Now we attack the weighted edge flavour (initially assuming for sanity that all edge weights are non-negative).

The process we're going to describe is known as Dijkstra's algorithm; it's a particular example of a class of techniques called greedy algorithms which essentially proceed at each stage with what appears to be the best 'local' solution.

At each step we choose a vertex v having the smallest dist amongst the unknown vertices, and then set as known the shortest path from s to v. The various values of dist are then updated. Taking our standard example, and adding weights to the edges, and starting at vertex 1...



A table of vertex values changes as follows...

vertex:	1	2	3	4	5	6	7	8	9	10	
known:	F	F	F	F	F	F	F	F	F	F	initially
dist:	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	initially
path:	0	0	0	0	0	0	0	0	0	0	initially
known:	T	F	F	F	F	F	F	F	F	F	vertex 1 now known, so adjust adjacents
dist:	0	2	4	1	∞	∞	∞	∞	∞	∞	vertex 1 now known, so adjust adjacents
path:	0	1	1	1	0	0	0	0	0	0	vertex 1 now known, so adjust adjacents
known:	T	F	F	T	F	F	F	F	F	F	vertex 4 now known, so adjust adjacents summary dists
dist:	0	2	4	1	∞	∞	9	5	∞	∞	vertex 4 now known, so adjust adjacents summary dists
path:	0	1	1	1	0	0	4	4	0	0	vertex 4 now known, so adjust adjacents summary dists
known:	T	T	F	T	F	F	F	F	F	F	vertex 2 now known
dist:	0	2	4	1	12	∞	9	5	∞	∞	vertex 2 now known
path:	0	1	1	1	2	0	4	4	0	0	vertex 2 now known
known:	T	T	T	T	F	F	F	F	F	F	vertex 3 now known
dist:	0	2	4	1	12	∞	9	5	∞	∞	vertex 3 now known
path:	0	1	1	1	2	0	4	4	0	0	vertex 3 now known

known:	T	T	T	T	F	F	F	T	F	F	vertex 8
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	F	F	T	T	F	F	vertex 7
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	T	F	T	T	F	F	vertex 5
dist:	0	2	4	1	12	∞	6	5	12	∞	now known
path:	0	1	1	1	2	0	8	4	8	0	
known:	T	T	T	T	T	F	T	T	T	F	vertex 9
dist:	0	2	4	1	12	13	6	5	12	∞	now known
path:	0	1	1	1	2	9	8	4	8	0	
known:	T	T	T	T	T	T	T	T	T	*F	vertex 6
dist:	0	2	4	1	12	13	6	5	12	∞	now known
path:	0	1	1	1	2	9	8	4	8	0	

* this will then change to T at end

The code might then be ...

```
public Vertex [] createTable()
  {
     Vertex [] t = readGraph();
     for (int i=0; i< t.length; i++)
       {
          t[i].known = false;
          t[i].dist  = Graph.INFINITY;
          t[i].path  = null;
       }
     NUM_VERTICES = t.length;
  }
void printPath( Vertex v)
  { if (v.path != null)
     { printPath(v.path);
       System.out.print(" to ");
     }
    System.out.print(v.name);
  }
```

some method to grab the graph with its adjacency lists, etc..

we started this by 0 in the tables above + the example.

recursive path printing

```
void dijkstra ( Vertex s )
  {
     Vertex v, w;
     s.dist = 0;

     for ( ; ; )
       {
          v = unknown vertex with smallest dist
          if (v == null)  break;
          v.known = true;

          for each w adjacent to v
             if ( ! w.known )
                if ( v.dist + edge wt. from v to w < w.dist )
                   {
                      w.dist = v.dist + edge wt.;
                      w.path = v;
                   }
       }
  }
```

This is an $O(e + n^2)$ algorithm

There are many improvements we could make, especially in the case of a very sparse graph, but we save these embellishments for later courses.

Dijkstra's algorithm fails if we were to allow negative edge weights, since there could be 'cheaper' paths appearing later going back to previously marked 'known' vertices.

An $O(en)$ algorithm (!!) to deal with this could be ...

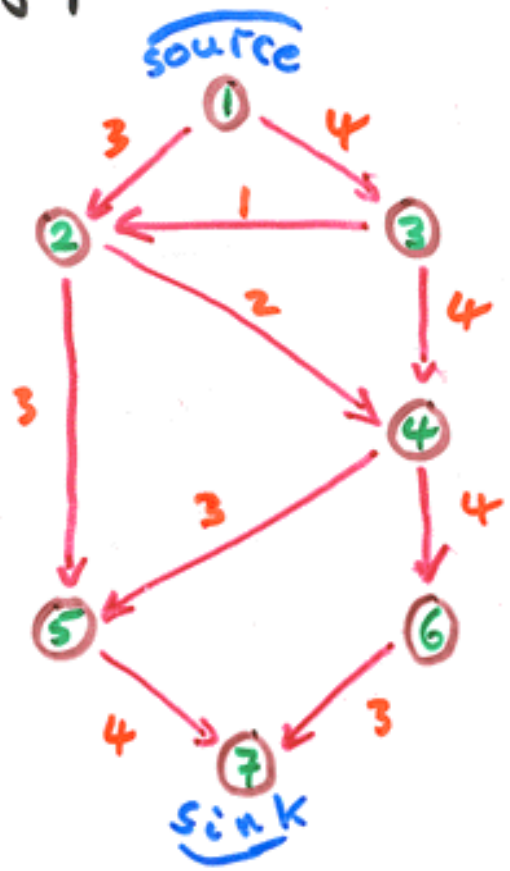
```
void negwts ( Vertex s )
  {
     Queue q;
     Vertex v, w;
     q = new Queue();
     q.enqueue(s);

     while (! q.isEmpty())
       {
          v = q.dequeue();
          for each w adjacent to v
            if (v.dist + edge cost from v to w < w.dist)
              {
                 w.dist = v.dist + edge cost;
                 w.path = v;
                 if (w is not already in q)
                   q.enqueue(w);
              }
       }
  }
```

This code doesn't quite work correctly - see comments following...

The above code is an amalgam of our unweighted and Dijkstra solutions. The idea is to start by putting s on a queue, then for each dequeued v we find all adjacent w with current dist larger than v.dist + edge cost, update w.dist and w.path, and ensure that w is on the queue. We could use the now unused known to indicate a vertex's presence on the queue. This is then repeated until the queue is empty. Notice that each vertex should only be able to dequeue at most n times, so to avoid infinite looping if there are negative costs we should ensure that no vertex gets dequeued more than (n+1) times!!

If we know that the graph is acyclic, then we can improve Dijkstra to an $O(e+n)$ algorithm by doing the selecting and updating within essentially a topological sort algorithm (using this order to declare vertices 'known'). This kind of graph appears frequently in practical applications.

Looking again at general weighted edge graphs, but with an almost reversed emphasis, leads to their use in analysing network flow problems. Here the edge-values refer to permitted flow rather than assessed cost. We'll use a fresh graph to illustrate this problem ... Whether you think of this as traffic flow, plumbing, or internet connectivity is up to you!



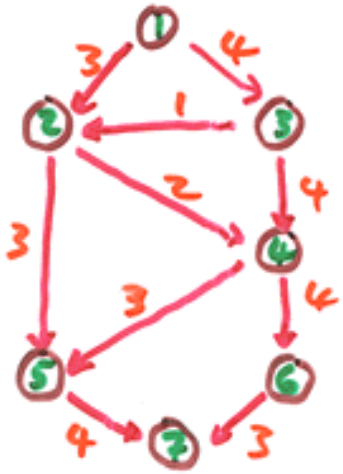
We'll start by considering an intuitively reasonable, but flawed approach, and then adjust it to work. Our assumptions are that at each vertex (apart from the source & sink), "what comes in must go out". Our approach will be to build two related graphs — one to display the flow choices, the other to display the residual flow available ...

flow graph

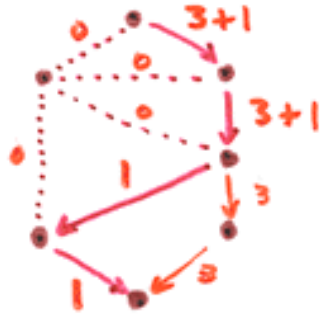
Initial set-up

We start by choosing any path from 1 to 7, labelling this in the flow graph with the maximum flow this whole path can accomodate.

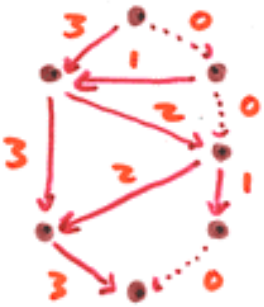
residual graph

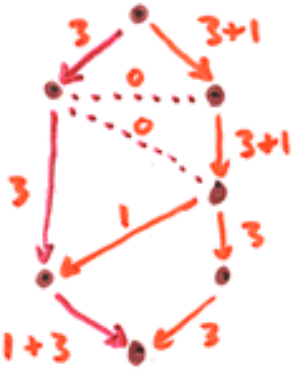


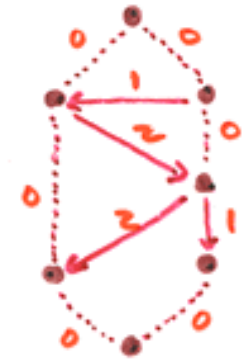
Having now reached this situation, we repeat the process (looking at the residual graph).



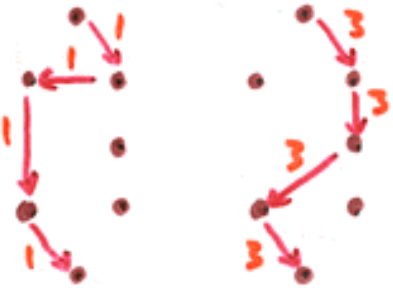
Again we chose our path somewhat at random, and a final flow arrangement can be found easily from here. (Note that for our graph, the solution is not unique.) The process clearly terminates since the residual graph now provides no way of getting from the top to the bottom.







The flow with this algorithm can be seen if our first two choices of paths were these following...

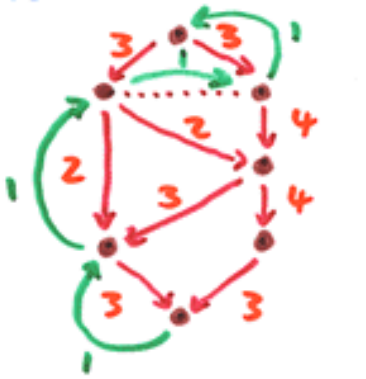


This leaves room 2 to enter right!

There's an easy fix for this — to allow our algorithm to "change its mind", each insertion in the flow graph will be accompanied by an equal and opposite insertion in the residual graph. Hence our previously 'bad' choice would appear as follows ...

flow graph

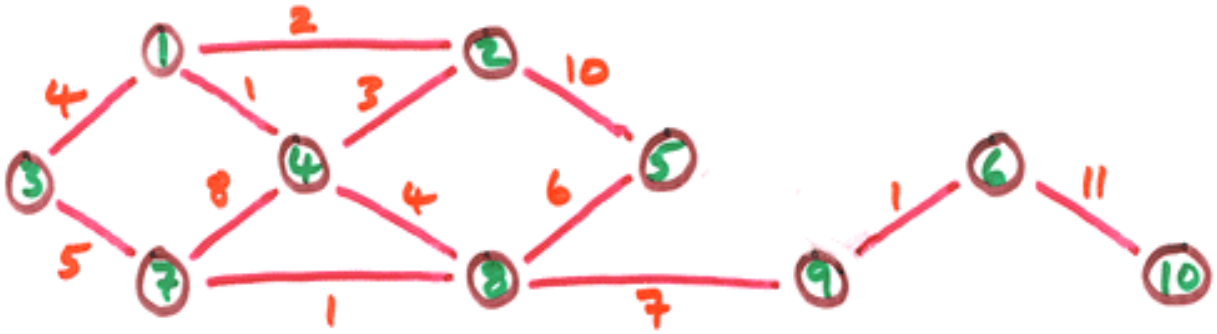
residual graph



We continue in this fashion, always augmenting the total flow. This is not necessarily a particularly efficient algorithm — given integer flows and a final max flow of F, this is an $O(e.F)$ algorithm at worst!!

Another thing we can ask about graphs is to find a minimum spanning tree. Since the solution is easier for undirected graphs, we'll restrict our attention to this situation. What we mean by such an animal is of course that it should be a tree (no circuits), that every vertex of the original graph be a vertex of this tree, and that 'minimal' means that the total edge costs in the tree are as low as possible. There is absolutely no guarantee that such an animal be unique. A few moments thought will convince you that if the graph has n vertices then the minimum spanning tree will have (n-1) edges.

Using our standard example (and removing the directedness) gives ...



with minimum spanning tree ...

having total 'cost' 37.

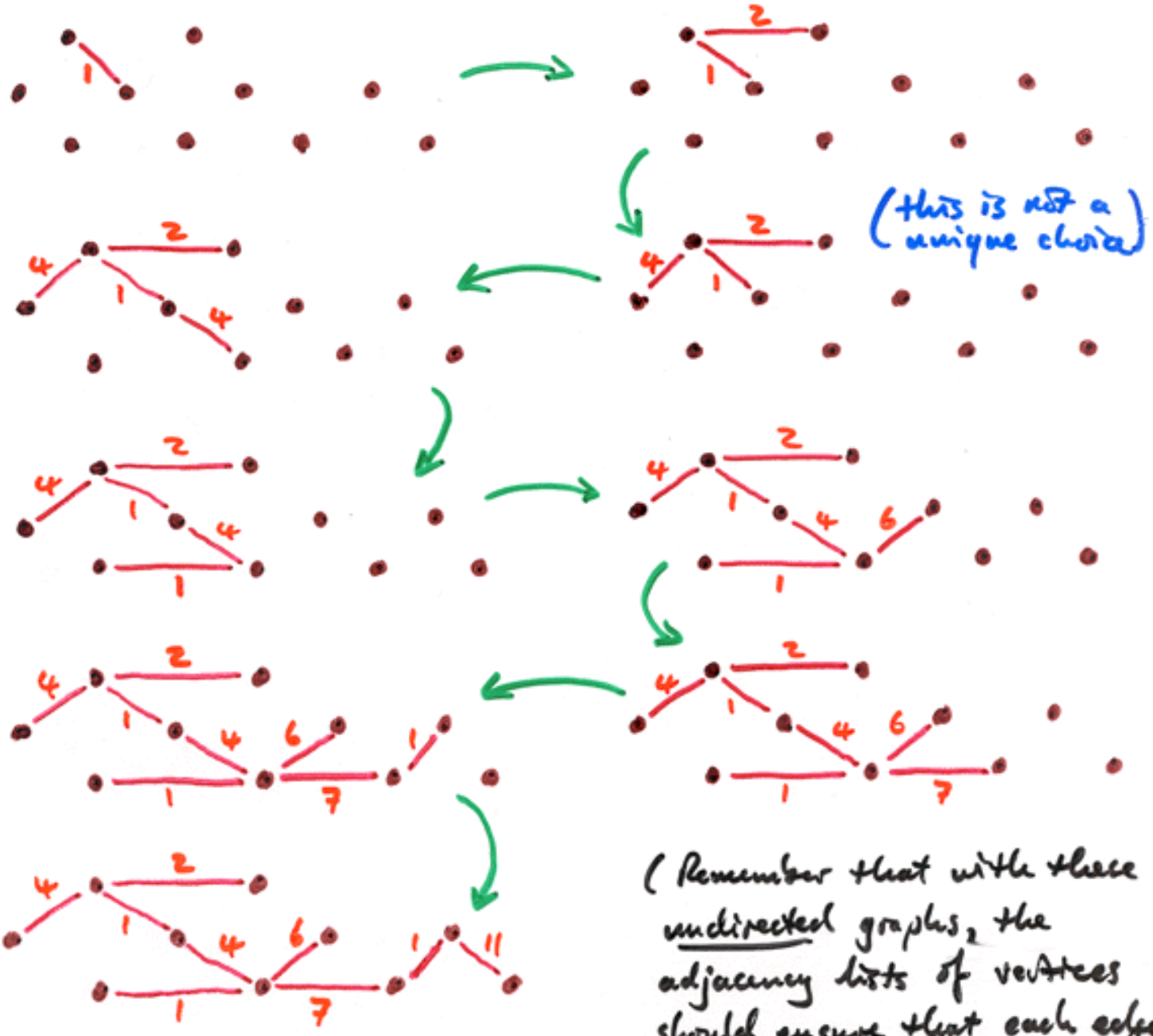
There are two natural approaches to this problem. The first (Prim's algorithm) builds the tree by picking any vertex as root, and then adds the vertex v by finding an edge u—v of minimum cost where u is in the tree and v is not yet in it. This is reminiscent of Dijkstra, so we would maintain a table of 'known' vertices, weights of 'cheapest' edge connecting a given vertex to a known vertex, and the vertex most recently used to change the dist. The starting table would be (if starting at vertex 4) ...

vertex #	1	2	3	4	5	6	7	8	9	10
known:	F	F	F	F	F	F	F	F	F	F
dist:	∞	∞	∞	0	∞	∞	∞	∞	∞	∞
path:	0	0	0	0	0	0	0	0	0	0

The update rule for this is simply that for each vertex v chosen, we reassign...

$$dist_w = \min(dist_w, cost_{v,w})$$

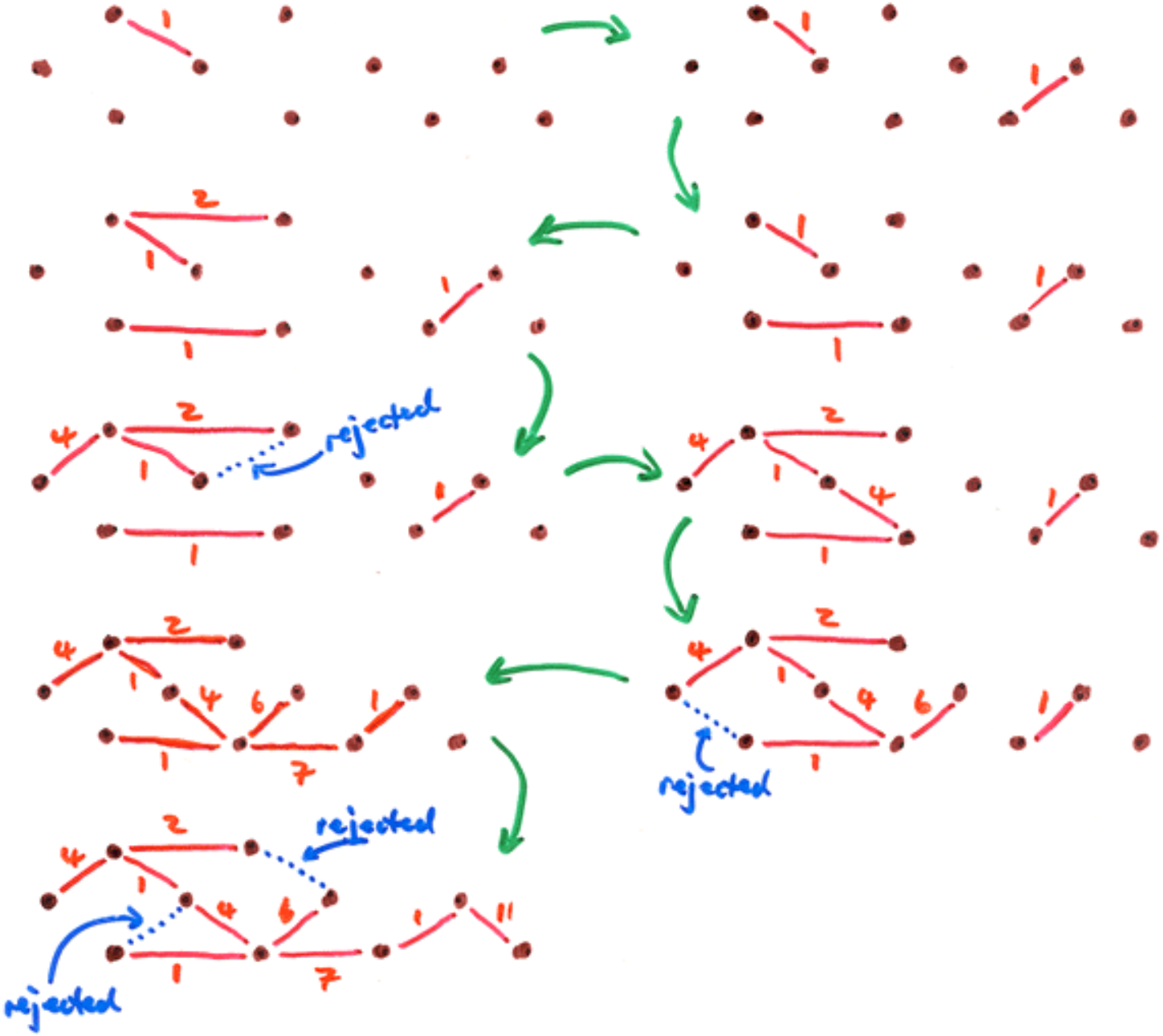
for each vertex w adjacent to v. Graphically, starting with vertex 4, this gives the following sequence of pictures...



(Remember that with these undirected graphs, the adjacency lists of vertices should ensure that each edge is in two lists!)

The running time of Prim is $O(n^2)$ — this is optimal for dense graphs. If the graph is sparse, then binary heaps should be used here, giving Prim a running time of $O(e \log n)$.

The other natural approach (Kruskal's algorithm) gathers edges in order of 'cheapness', rejecting any edges which don't add any new vertices to the collection. Again, a pictorial portrayal of this gives ...



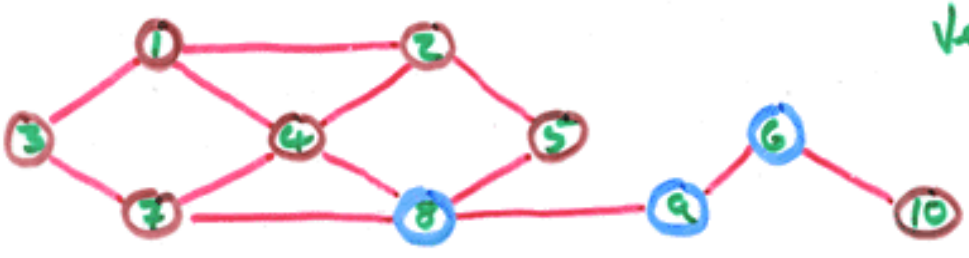
For our final graph problem, we'll look at depth-first searches. This involves starting at some vertex v and then recursively traversing all vertices adjacent to v. If the graph is a tree, this takes $O(e)$ time.

For general graphs, we need to avoid getting stuck in cycles! This is easily done by using known to mean visited. If we initialize v.known = false for all vertices, then a generic depth first search might be ...

```
void dfs ( Vertex v )
  {
    v.known = true;
    for each w adjacent to v
      if ( ! w.known )  dfs(w);
  }
```

This will work for connected undirected graphs; but if the graph is directed, but not strongly connected (i.e. does not have a path from every vertex to every other vertex), then this process will stop prematurely. This is easily fixed by restarting dfs, if necessary, at the 'next' unknown vertex. This gives an $O(n+e)$ traversal of all the vertices.

To illustrate how we might use this approach, we'll investigate how 'connected' a graph is. An undirected graph is biconnected if it has no vertices whose removal would disconnect the graph into two or more components — such vertices are called articulation points. Obvious applications could be the stability of computer networks, electricity power nets, or transportation studies. In our standard example...



Vertices 6, 8 and 9 are articulation points, so this graph isn't biconnected