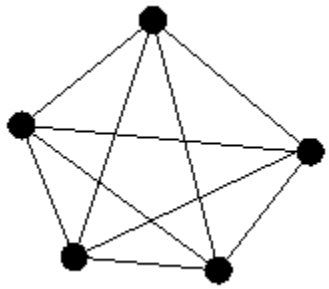
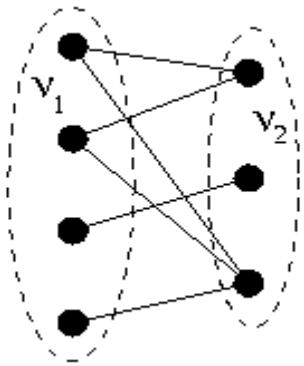
Complete Graph



A graph is said to be *complete* if there is an edge between every pair of vertices.

Bipartite Graph



A graph is said to be *bipartite* if the vertices can be split into two sets V1 and V2 such there are no edges between two vertices of V1 or two vertices of V2.

Uninformed Search

Searching is a process of considering possible sequences of actions, first you have to formulate a goal and then use the goal to formulate a problem.

A **problem** consists of four parts: the **initial state**, a set of **operators**, a **goal test** function, and a **path cost** function. The environment of the problem is represented by a

state space. A **path** through the state space from the initial state to a goal state is a **solution**.

In real life most problems are ill-defined, but with some analysis, many problems can fit into the state space model. A single general search algorithm can be used to solve any problem; specific variants of the algorithm embody different strategies. Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**. Complexity depends on *b*, the **branching factor** in the state space, and *d*, the **depth** of the shallowest solution.

This 6 search type below (there are more, but we only show 6 here) classified as uninformed search, this means that the search have no information about the number of steps or the path cost from the current state to the goal - all they can do is distinguish a goal state from a non-goal state. Uninformed search is also sometimes called blind search.

Breadth First Serach (BFS)

Breadth-first search expands the shallowest node in the search tree first. It is complete, optimal for unit-cost operators, and has time and space complexity of O(b^d). The space complexity makes it impractical in most cases.

Using BFS strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and their successors, and so on. In general, all the nodes at depth *d* in the search tree are expanded before the nodes at depth *d+1*.

Algorithmically:

```
BFS(G,s) {
  initialize vertices;
  Q = {s};
  while (Q not empty) {
    u = Dequeue(Q);
    for each v adjacent to u do {
      if (color[v] == WHITE) {
        color[v] = GRAY;
        d[v] = d[u]+1; // compute d[]
        p[v] = u;  // build BFS tree
        Enqueue(Q,v);
    }
  }
  color[u] = BLACK;
}
```

BFS runs in O(V+E)

Note: BFS can compute d[v] = shortest-path distance from s to v, in terms of minimum number of edges from s to v (un-weighted graph). Its breadth-first tree can be used to represent the shortest-path.

BFS Solution to Popular JAR Problem

```
#include<stdio.h>
#include<conio.h>
#include<values.h>

#define N 105
#define MAX MAXINT

int act[N][N], Q[N*20][3], cost[N][N];
int a, p, b, m, n, fin, na, nb, front, rear;

void init()
{
     front = -1, rear = -1;
     for(int i=0; i<N; i++)
            for(int j=0; j<N; j++)
                   cost[i][j] = MAX;
     cost[0][0] = 0;
}

void nQ(int r, int c, int p)
{
     Q[++rear][0] = r, Q[rear][1] = c, Q[rear][2] = p;
}

void dQ(int *r, int *c, int *p)
{
     *r = Q[++front][0], *c = Q[front][1], *p = front;
}

void op(int i)
{
     int currCapA, currCapB;
     if(i==0)
            na = 0, nb = b;
     else if(i==1)
            nb = 0, na = a;
     else if(i==2)
            na = m, nb = b;
     else if(i==3)
            nb = n, na = a;
     else if(i==4)
     {
            if(!a && !b)
```

```
                return;
            currCapB = n - b;
            if(currCapB <= 0)
                return;
            if(a >= currCapB)
                nb = n, na = a, na -= currCapB;
            else
                nb = b, nb += a, na = 0;
    }
    else
    {
            if(!a && !b)
                return;
            currCapA = m - a;
            if(currCapA <= 0)
                return;
            if(b >= currCapA)
                na = m, nb = b, nb -= currCapA;
            else
                nb = 0, na = a, na += b;
    }
}

void bfs()
{
    nQ(0, 0, -1);
    do{

            dQ(&a, &b, &p);
            if(a==fin)
                break;
            for(int i=0; i<6; i++)
            {
                op(i); /* na, nb will b changed for this func
                          according to values of a, b
                        */
                if(cost[na][nb]>cost[a][b]+1)
                {
                    cost[na][nb]=cost[a][b]+1;
                    act[na][nb] = i;
                    nQ(na, nb, p);
                }
            }
    } while (rear!=front);
}

void dfs(int p)
{
    int i = act[na][nb];
    if(p==-1)
            return;
    na = Q[p][0], nb = Q[p][1];
    dfs(Q[p][2]);
    if(i==0)
```

```
            printf("Empty A\n");
     else if(i==1)
            printf("Empty B\n");
     else if(i==2)
            printf("Fill A\n");
     else if(i==3)
            printf("Fill B\n");
     else if(i==4)
            printf("Pour A to B\n");
     else
            printf("Pout B to A\n");
}

void main()
{
     clrscr();
     while(scanf("%d%d%d", &m, &n, &fin)!=EOF)
     {
            printf("\n");
            init();
            bfs();
            dfs(Q[p][2]);
            printf("\n");
     }
}
```

Uniform Cost Search (UCS)

Uniform-cost search expands the least-cost leaf node first. It is complete, and unlike breadth-first search is optimal even when operators have differing costs. Its space and time complexity are the same as for BFS.

BFS finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. UCS modifies BFS by always expanding the lowest-cost node on the fringe.

Depth First Search (DFS)

Depth-first search expands the deepest node in the search tree first. It is neither complete nor optimal, and has time complexity of O(b^m) and space complexity of O(bm), where m is the maximum depth. In search trees of large or infinite depth, the time complexity makes this impractical.

DFS always expands one of the nodes at the deepest level of the tree. Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower levels.

Algorithmically:

```
DFS(G) {
  for each vertex u in V
    color[u] = WHITE;
  time = 0; // global variable
  for each vertex u in V

    if (color [u] == WHITE)
      DFS_Visit(u);
}

DFS_Visit(u) {
  color[u] = GRAY;
  time = time + 1; // global variable
  d[u] = time; // compute discovery time d[]
  for each v adjacent to u
    if (color[v] == WHITE) {
      p[v] = u; // build DFS-tree
      DFS_Visit(u);
    }
  color[u] = BLACK;
  time = time + 1; // global variable
  f[u] = time; // compute finishing time f[]
}
```

DFS runs in O(V+E)

DFS can be used to classify edges of G:

1. Tree edges: edges in the depth-first forest
2. Back edges: edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree
3. Forward edges: non-tree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree
4. Cross edges: all other edges

An undirected graph is acyclic iff a DFS yields no back edges.

DFS algorithm Implementation

Form a one-element queue consisting of the root node.

Until the queue is empty or the goal has been reached, determine if the first element in the queue is the goal node. If the first element is the goal node, do nothing. If the first element is not the goal node, remove the first element from the queue and add the first element's children, if any, to the **front** of the queue.

If the goal node has been found, announce success, otherwise announce failure.

Note: This implementation differs with BFS in insertion of first element's children, DFS from **FRONT** while BFS from **BACK**. The worst case for DFS is the best case for BFS and vice versa. However, avoid using DFS when the search trees are very large or with infinite maximum depths.

N Queens Problem

Place n queens on an n x n chess board so that no queen is attacked by another queen.

The most obvious solution to code is to add queens recursively to the board one by one, trying all possible queen placements. It is easy to exploit the fact that there must be exactly one queen in each column: at each step in the recursion, just choose where in the current column to put the queen.

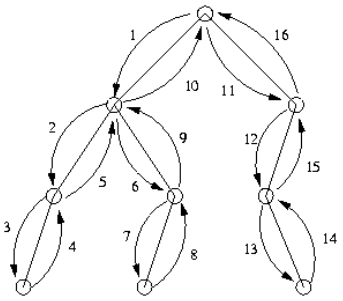
DFS Solution

```
1 search(col)
2    if filled all columns
3       print solution and exit
4    for each row
5       if board(row, col) is not attacked
6          place queen at (row, col)
7          search(col+1)
8          remove queen at (row, col)
```

Calling search(0) begins the search. This runs quickly, since there are relatively few choices at each step: once a few queens are on the board, the number of non-attacked squares goes down dramatically.

This is an example of depth first search, because the algorithm iterates down to the bottom of the search tree as quickly as possible: once k queens are placed on the board, boards with even more queens are examined before examining other possible boards with only k queens. This is okay but sometimes it is desirable to find the simplest solutions before trying more complex ones.

Depth first search checks each node in a search tree for some property. The search tree might look like this:



The algorithm searches the tree by going down as far as possible and then backtracking when necessary, making a sort of outline of the tree as the nodes are visited. Pictorially, the tree is traversed in the following manner:

Suppose there are d decisions that must be made. (In this case d=n, the number of columns we must fill.) Suppose further that there are C choices for each decision. (In this case c=n also, since any of the rows could potentially be chosen.) Then the entire search will take time proportional to c^d, i.e., an exponential amount of time. This scheme requires little space, though: since it only keeps track of as many decisions as there are to make, it requires only O(d) space.

Knight Cover

Place as few knights as possible on an n x n chess board so that every square is attacked. A knight is not considered to attack the square on which it sits.

Depth First with Iterative Deepening (DF-ID)

An alternative to breadth first search is iterative deepening. Instead of a single breadth first search, run D depth first searches in succession, each search allowed to go one row deeper than the previous one. That is, the first search is allowed only to explore to row 1, the second to row 2, and so on. This ``simulates'' a breadth first search at a cost in time but a savings in space.

```
1 truncated_dfsearch(hnextpos, depth)
2    if board is covered
3       print solution and exit
4    if depth == 0
5       return
6    for i from nextpos to n*n
7       put knight at i
8       search(i+1, depth-1)
9       remove knight at i
10      dfid_search
11    for depth = 0 to max_depth
12       truncated_dfsearch(0, depth)
```

The space complexity of iterative deepening is just the space complexity of depth first search: O(n). The time complexity, on the other hand, is more complex. Each truncated depth first search stopped at depth k takes ck time. Then if d is the maximum number of decisions, depth first iterative deepening takes c^0 + c^1 + c^2 + ... + c^d time.

Which to Use?

Once you've identified a problem as a search problem, it's important to choose the right type of search. Here are some things to think about.

Search	Time	Space	When to use
DFS	O(c^k)	O(k)	Must search tree anyway, know the level the answers are on, or you aren't looking for the shallowest number.
BFS	O(c^d)	O(c^d)	Know answers are very near top of tree, or want shallowest answer.
DFS+ID	O(c^d)	O(d)	Want to do BFS, don't have enough space, and can spare the time.

d is the depth of the answer,k is the depth searched,d <= k .

Remember the ordering properties of each search. If the program needs to produce a list sorted shortest solution first (in terms of distance from the root node), use breadth first search or iterative deepening. For other orders, depth first search is the right strategy.

If there isn't enough time to search the entire tree, use the algorithm that is more likely to find the answer. If the answer is expected to be in one of the rows of nodes closest to the root, use breadth first search or iterative deepening. Conversely, if the answer is expected to be in the leaves, use the simpler depth first search.

Be sure to keep space constraints in mind. If memory is insufficient to maintain the queue for breadth first search but time is available, use iterative deepening.

Depth Limited Search

Depth-limited search places a limit on how deep a depth-first search can go. If the limit happens to be equal to the depth of shallowest goal state, then time and space complexity are minimized.

DLS stops to go any further when the depth of search is longer than what we have defined.

Iterative Depending Search

Iterative deepening search calls depth-limited search with increasing limits until a goal is found. It is complete and optimal, and has time complexity of O(b^d)

IDS is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on. In effect, IDS combines the benefits of DFS and BFS.

Bidirectional Search

Bidirectional search can enormously reduce time complexity, but is not always applicable. Its memory requirements may be impractical.

BDS simultaneously search both forward form the initial state and backward from the goal, and stop when the two searches meet in the middle, however search like this is not always possible.

Superprime Rib

A number is called superprime if it is prime and every number obtained by chopping some number of digits from the right side of the decimal expansion is prime. For example, 233 is a superprime, because 233, 23, and 2 are all prime. Print a list of all the superprime numbers of length n, for n <= 9. The number 1 is not a prime.

For this problem, use depth first search, since all the answers are going to be at the nth level (the bottom level) of the search.

Betsy's Tour

A square township has been partitioned into n 2 square plots. The Farm is located in the upper left plot and the Market is located in the lower left plot. Betsy takes a tour of the township going from Farm to Market by walking through every plot exactly once. Write a program that will count how many unique tours Betsy can take in going from Farm to Market for any value of n <= 6.

Since the number of solutions is required, the entire tree must be searched, even if one solution is found quickly. So it doesn't matter from a time perspective whether DFS or BFS is used. Since DFS takes less space, it is the search of choice for this problem.

Udder Travel

The Udder Travel cow transport company is based at farm A and owns one cow truck which it uses to pick up and deliver cows between seven farms A, B, C, D, E, F, and G. The (commutative) distances between farms are given by an array. Every morning, Udder Travel has to decide, given a set of cow moving orders, the order in which to pick up and deliver cows to minimize the total distance traveled. Here are the rules:

1. The truck always starts from the headquarters at farm A and must return there when the day's deliveries are done.
2. The truck can only carry one cow at time.
3. The orders are given as pairs of letters denoting where a cow is to be picked up followed by where the cow is to be delivered.

Your job is to write a program that, given any set of orders, determines the shortest route that takes care of all the deliveries, while starting and ending at farm A.

Since all possibilities must be tried in order to ensure the best one is found, the entire tree must be searched, which takes the same amount of time whether using DFS or BFS. Since DFS uses much less space and is conceptually easier to implement, use that.

Desert Crossing

A group of desert nomads is working together to try to get one of their group across the desert. Each nomad can carry a certain number of quarts of water, and each nomad drinks a certain amount of water per day, but the nomads can carry differing amounts of water, and require different amounts of water. Given the carrying capacity and drinking requirements of each nomad, find the minimum number of nomads required to get at least one nomad across the desert.

All the nomads must survive, so every nomad that starts out must either turn back at some point, carrying enough water to get back to the start or must reach the other side of the desert. However, if a nomad has surplus water when it is time to turn back, the water can be given to their friends, if their friends can carry it.

This problem actually is two recursive problems: one recursing on the set of nomads to use, the other on when the nomads turn back. Depth-first search with iterative deepening works well here to determine the nomads required, trying first if any one can make it across by themselves, then seeing if two work together to get across, etc.

Addition Chains

An addition chain is a sequence of integers such that the first number is 1, and every subsequent number is the sum of some two (not necessarily unique) numbers that appear in the list before it. For example, 1 2 3 5 is such a chain, as 2 is 1+1, 3 is 2+1, and 5 is 2+3. Find the minimum length chain that ends with a given number.

Depth-first search with iterative deepening works well here, as DFS has a tendency to first try 1 2 3 4 5 ... n, which is really bad and the queue grows too large very quickly for BFS.

Informed Search

Unlike Uninformed Search, Informed Search knows some information that can be used to improve the path selection. Examples of Informed Search: Best First Search, Heuristic Search such as A*.

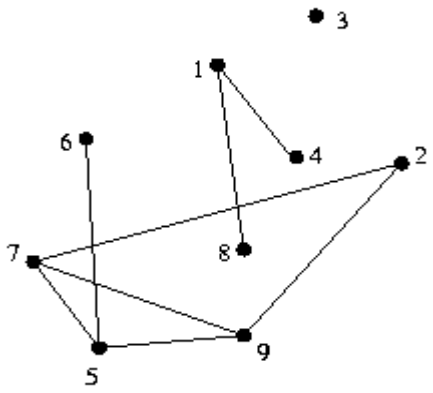
Best First Search

we define a function f(n) = g(n) where g(n) is the estimated value from node 'n' to goal. This search is "informed" because we do a calculation to estimate g(n)

A* Search

f(n) = h(n) + g(n), similar to Best First Search, it uses g(n), but also uses h(n), the total cost incurred so far. The best search to consider if you know how to compute g(n).

Components



Given: a undirected graph (see picture on the right)

The *component* of a graph is a maximal-sized (though not necessarily maximum) subgraph which is connected.

Calculate the component of the graph.

This graph has three components: {1,4,8}, {2,5,6,7,9}, & {3}.

Flood Fill Algorithm

Flood fill can be performed three basic ways: depth-first, breadth-first, and breadth-first scanning. The basic idea is to find some node which has not been assigned to a component and to calculate the component which contains. The question is how to calculate the component.

In the depth-first formulation, the algorithm looks at each step through all of the neighbors of the current node, and, for those that have not been assigned to a component yet, assigns them to this component and recurses on them.

In the breadth-first formulation, instead of recursing on the newly assigned nodes, they are added to a queue.

In the breadth-first scanning formulation, every node has two values: component and visited. When calculating the component, the algorithm goes through all of the nodes that have been assigned to that component but not visited yet, and assigns their neighbors to the current component.

The depth-first formulation is the easiest to code and debug, but can require a stack as big as the original graph. For explicit graphs, this is not so bad, but for implicit graphs, such as the problem presented has, the numbers of nodes can be very large.

The breadth-formulation does a little better, as the queue is much more efficient than the run-time stack is, but can still run into the same problem. Both the depth-first and breadth-first formulations run in N + M time, where N is the number of vertices and M is the number of edges.

The breadth-first scanning formulation, however, requires very little extra space. In fact, being a little tricky, it requires no extra space. However, it is slower, requiring up to N*N + M time, where N is the number of vertices in the graph.

Breadth-First Scanning

```
# component(i) denotes the component that node i is in


1 function flood_fill(new_component)
2 do
3    num_visited = 0
4    for all nodes i
5       if component(i) = -2
6          num_visited = num_visited + 1
7          component(i) = new_component
8          for all neighbors j of node i
9             if component(i) = nil
10               component(i) = -2
11 until num_visited = 0
12 function find_components
13    num_components = 0
14    for all nodes i
15       component(node i) = nil
16    for all nodes i
17       if component(node i) is nil
18         num_components = num_components + 1
19         component(i) = -2
20         flood_fill(component(num_components))
```

Running time of this algorithm is O(N 2), where N is the numbers of nodes. Every edge is traversed twice (once for each end-point), and each node is only marked once.

Company Ownership

Given: A weighted directed graph, with weights between 0 and 100.

Some vertex A ``owns'' another vertex B if:

1. A = B
2. There is an arc from A to B with weight more than 50.
3. There exists some set of vertices C 1 through C k such that A owns C 1 through C k, and each vertex has an arc of weight x 1 through x k to vertex B, and x 1 + x 2 + ... + x k > 50.

Find all (a,b) pairs such that a owns b.

This can be solved via an adaptation of the calculating the vertices reachable from a vertex in a directed graph. To calculate which vertices vertex A owns, keep track of the ``ownership percentage'' for each node. Initialize them all to zero. Now, at each recursive step, mark the node as owned by vertex A and add the weight of all outgoing arcs to the ``ownership percentages.'' For all percentages that go above 50, recurse into those vertices.

Street Race

Given: a directed graph, and a start point and an end point.

Find all points p that any path from the start point to the end must travel through p.

The easiest algorithm is to remove each point in turn, and check to see if the end point is reachable from the start point. This runs in O(N (M + N)) time. Since the original problem stated that M <= 100, and N <= 50, this will run in time easily.

Cow Tours

The diameter of a connected graph is defined as the maximum distance between any two nodes of the graph, where the distance between two nodes is defined as the length of the shortest path.

Given a set of points in the plane, and the connections between those points, find the two points which are currently not in the same component, such that the diameter of the resulting component is minimized.

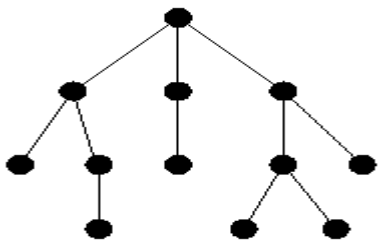
Find the components of the original graph, using the method described above. Then, for each pair of points not in the same component, try placing a connection between them. Find the pair that minimizes the diameter.

Connected Fields

Farmer John contracted out the building of a new barn. Unfortunately, the builder mixed up the plans of Farmer John's barn with another set of plans. Farmer John's plans called for a barn that only had one room, but the building he got might have many rooms. Given a grid of the layout of the barn, tell Farmer John how many rooms it has.

Analysis: The graph here is on the non-wall grid locations, with edge between adjacent non-wall locations, although the graph should be stored as the grid, and not transformed into some other form, as the grid is so compact and easy to work with.

Tree



Tree is one of the most efficient data structure used in a computer program. There are many types of tree. Binary tree is a tree that always have two branches, Red-Black-Trees, 2-3-4 Trees, AVL Trees, etc. A well balanced tree can be used to design a good searching algorithm.A Tree is an undirected graph that contains no cycles and is connected.Many trees are what is called *rooted*, where there is a notion of the "top"

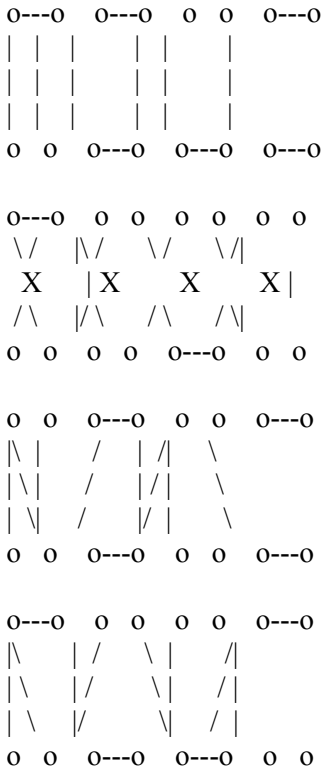
node, which is called the root. Thus, each node has one *parent*, which is the adjacent node which is closer to the root, and may have any number of *children*, which are the rest of the nodes adjacent to it. The tree above was drawn as a rooted tree.

Minimum Spanning Trees

Spanning trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices

has sixteen spanning trees:



Minimum spanning trees

Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. The problem: how to find the minimum length spanning tree?

Why minimum spanning trees?

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

How to find minimum spanning tree?

The stupid method is to list all spanning trees, and find minimum of list. We already know how to find minima... But there are far too many trees for this to be efficient. It's also not really an algorithm, because you'd still need to know how to list all the trees.

A better idea is to find some key property of the MST that lets us be sure that some edge is part of it, and use this property to build up the MST one edge at a time.

For simplicity, we assume that there is a unique minimum spanning tree. You can get ideas like this to work without this assumption but it becomes harder to state your theorems or write your algorithms precisely.

Lemma: Let X be any subset of the vertices of G, and let edge e be the smallest edge connecting X to G-X. Then e is part of the minimum spanning tree.

Proof: Suppose you have a tree T not containing e; then we want to show that T is not the MST. Let e=(u,v), with u in X and v not in X. Then because T is a spanning tree it contains a unique path from u to v, which together with e forms a cycle in G. This path has to include another edge f connecting X to G-X. T+e-f is another spanning tree (it has the same number of edges, and remains connected since you can replace any path containing f by one going the other way around the cycle). It has smaller weight than t since e has smaller weight than f. So T was not minimum, which is what we wanted to prove.

Kruskal's algorithm

We'll start with Kruskal's algorithm, which is easiest to understand and probably the best one for solving problems by hand.

```
Kruskal's algorithm:
sort the edges of G in increasing order by length
keep a subgraph S of G, initially empty
for each edge e in sorted order
    if the endpoints of e are disconnected in S
    add e to S
return S
```

Note that, whenever you add an edge (u,v), it's always the smallest connecting the part of S reachable from u with the rest of G, so by the lemma it must be part of the MST.

This algorithm is known as a *greedy algorithm*, because it chooses at each step the cheapest edge to add to S. You should be very careful when trying to use greedy algorithms to solve other problems, since it usually doesn't work. E.g. if you want to find a shortest path from a to b, it might be a bad idea to keep taking the shortest edges. The greedy idea only works in Kruskal's algorithm because of the key property we proved.

Analysis: The line testing whether two endpoints are disconnected looks like it should be slow (linear time per iteration, or O(mn) total). The slowest part turns out to be the sorting step, which takes O(m log n) time.

Prim's algorithm

Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time.

```
    Prim's algorithm:
    let T be a single vertex x
    while (T has fewer than n vertices) {
        find the smallest edge connecting T to G-T
        add it to T
    }
```

Example of Prim's algorithm in C language:

```
/* usedp=>how many points already used
   p->array of structures, consisting x,y,& used/not used
   this problem is to get the MST of graph with n vertices
   which weight of an edge is the distance between 2 points */

usedp=p[0].used=1; /* select arbitrary point as starting point */
while (usedp<n) {
  small=-1.0;

  for (i=0;i<n;i++) if (p[i].used)
    for (j=0;j<n;j++) if (!p[j].used) {
      length=sqrt(pow(p[i].x-p[j].x,2) + pow(p[i].y-p[j].y,2));

      if (small==-1.0 || length<small) {
        small=length;
        smallp=j;
      }
    }
  minLength+=small;

  p[smallp].used=1;
  usedp++ ;
}
```

Finding Shortest Paths using BFS

This only applicable to graph with unweighted edges, simply do BFS from start node to end node, and stop the search when it encounters the first occurrence of end node.

The **relaxation** process updates the costs of all the vertices, **v**, connected to a vertex, **u**, if we could improve the best estimate of the shortest path to **v** by including **(u,v)** in the path to **v**. The relaxation procedure proceeds as follows: