Problem Solving Techniques

Most important: make sure you understand exactly what the question is asking – if not, you have no hope of answer it!!

Never be afraid to ask for another explanation of a problem until it is clear.

Play around with the problem by constructing examples to get insight into it.

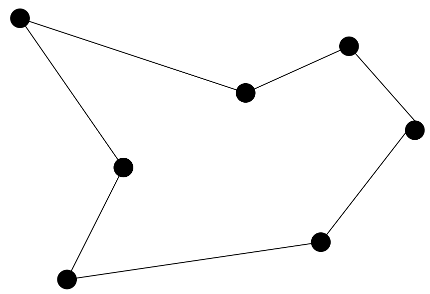
Ask yourself questions. Does the first idea which comes into my head work? If not, why not?

Am I using all information that I am given about the problem?

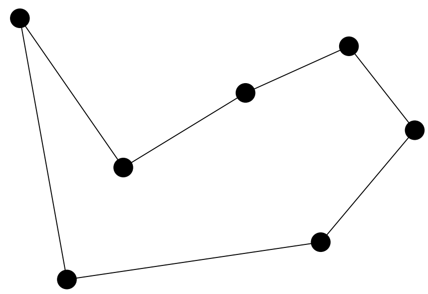
Read Polya's book *How to Solve it*.

16-1: The Euclidian traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane.

Bentley suggested simplifying the problem by restricting attention to bitonic tours, *that is tours which start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right back to the starting point.*



non-bitonic

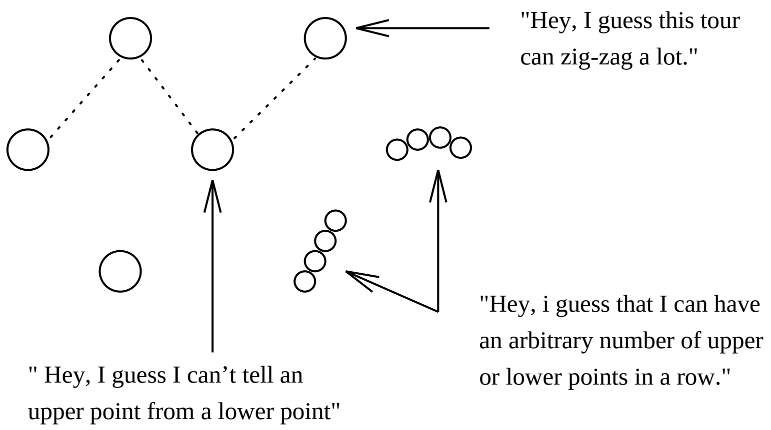


bitonic

Describe an $O(n^2)$ algorithm for finding the optimal bitonic tour. You may assume that no two points have the same x-coordinate. (Hint: scan left to right, maintaining optimal possibilities for the two parts of the tour.)

Make sure you understand what a bitonic tour is, or else it is hopeless.

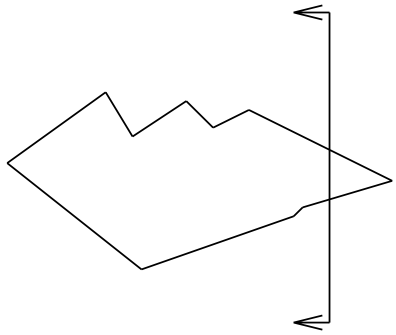
First of all, play with the problem. Why isn't it trivial?



Am I using all the information?

Why will they let us assume that no two x-coordinates are the same? What does the hint mean? What happens if I scan from left to right?

If we scan from left to right, we get an open tour which uses all points to the left of our scan line.



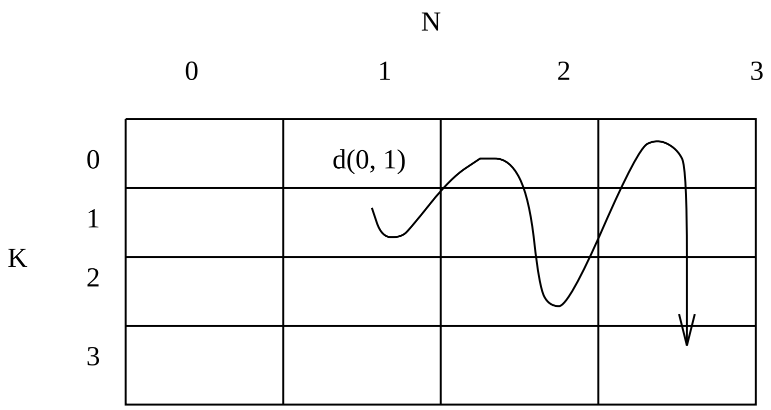
In the optimal tour, the kth point is connected to exactly one point to the left of k. $(k \neq n)$ Once I decide which point that is, say x. I need the optimal partial tour where the two endpoints are x and $k-1$, because if it isn't optimal I could come up with a better one. Hey, I have got a recurrence! And look, the two parameters which describe my optimal tour are the two endpoints.

Let $c[k,n]$ be the optimal cost partial tour where the two endpoints are $k < n$.

$c[k,n] \leq c[k,n-1] + d[n,n-1]$ (when $k < n-1$)

$c[n-1,n] \leq c[k,n-1] + d[k,n]$

$c[0,1] = d[0,1]$



Filling the entities in from N=1 to N', k=1 to N, means we always have what we need waiting for us.

$c[n-1,n]$ takes $O(n)$ to update, $c[k,n]$ $k < n-1$ takes $O(1)$ to update. Total time is $O(n^2)$.

But this doesn't quite give the tour, but just an open tour. We simply must figure where the last edge to n must go.

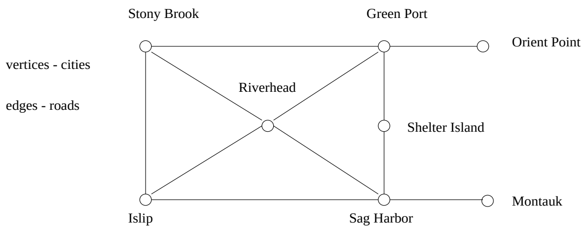
$$Tourcost = \min_{k=1}^{n} C[k,n] + d_{kn}$$

Graphs

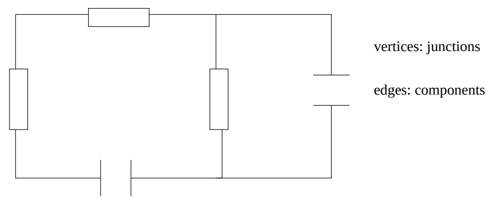
A graph G consists of a set of *vertices* V together with a set E of vertex pairs or *edges*.

Graphs are important because any binary relation is a graph, so graphs can be used to represent essentially *any* relationship.

Example: A network of roads, with cities as vertices and roads between cities as edges.



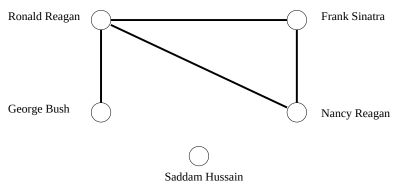
Example: An electronic circuit, with junctions as vertices as components as edges.



To understand many problems, we must think of them in terms of graphs!

The Friendship Graph

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.



This graph is well-defined on any set of people: SUNY SB, New York, or the world.

What questions might we ask about the friendship graph?

- **If I am your friend, does that mean you are my friend?**

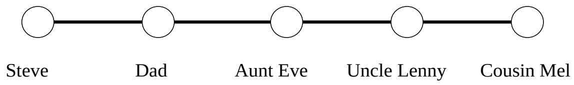
  A graph is *undirected* if (x, y) implies (y, x). Otherwise the graph is directed. The “heard-of” graph is directed since countless famous people have never heard of me! The “had-sex-with” graph is presumably undirected, since it requires a partner.

- **Am I my own friend?**

  An edge of the form (x, x) is said to be a *loop*. If x is y’s friend several times over, that could be modeled using *multiedges*, multiple edges between the same pair of vertices. A graph is said to be *simple* if it contains no loops and multiple edges.

- **Am I linked by some chain of friends to the President?**

  A *path* is a sequence of edges connecting two vertices. Since *Mel Brooks* is my father's-sister's-husband's cousin, there is a path between me and him!



- **How close is my link to the President?**

  If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny. Thus we are often interested in the *shortest path* between two nodes.

- **Is there a path of friends between any two people?**

  A graph is *connected* if there is a path between any two vertices. A directed graph is *strongly connected* if there is a directed path between any two vertices.

- **Who has the most friends?**

  The *degree* of a vertex is the number of edges adjacent to it.

- **What is the largest clique?**

  A social clique is a group of mutual friends who all hang around together. A graph theoretic *clique* is a complete subgraph, where each vertex pair has an edge between them. Cliques are the densest possible subgraphs. Within the friendship graph, we would expect that large cliques correspond to workplaces, neighborhoods, religious organizations, schools, and the like.

- **How long will it take for my gossip to get back to me?**

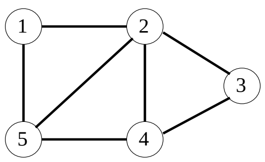
  A *cycle* is a path where the last vertex is adjacent to the first. A cycle in which no vertex repeats (such as 1-2-3-1 verus 1-2-3-2-1) is said to be *simple*. The shortest cycle in the graph defines its *girth*, while a simple cycle which passes through each vertex is said to be a *Hamiltonian cycle*.

Data Structures for Graphs

There are two main data structures used to represent graphs.

Adjacency Matrices

An *adjacency matrix* is an $n \times n$ matrix, where $M[i, j] = 0$ iff there is no edge from vertex i to vertex j



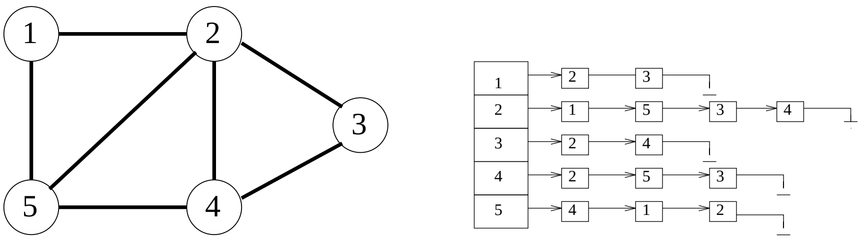
$$\begin{array}{|ccccc|} \hline 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

It takes $\Theta(1)$ time to test if (i, j) is in a graph represented by an adjacency matrix.

Can we save space if (1) the graph is undirected? (2) if the graph is sparse?

Adjacency Lists

An *adjacency list* consists of a $N \times 1$ array of pointers, where the ith element points to a linked list of the edges incident on vertex *i*.



To test if edge (i, j) is in the graph, we search the ith list for j, which takes $O(d_i)$, where d_i is the degree of the ith vertex.

Note that d_i can be much less than n when the graph is sparse. If necessary, the two *copies* of each edge can be linked by a pointer to facilitate deletions.

Tradeoffs Between Adjacency Lists and Adjacency Matrices

Comparison	Winner
Faster to test if (x, y) exists?	matrices
Faster to find vertex degree?	lists
Less memory on small graphs?	lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	matrices (small win)
Edge insertion or deletion?	matrices $O(1)$
Faster to traverse the graph?	lists $m + n$ vs. n^2
Better for most problems?	lists

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. Applications include:

- Printing out the contents of each edge and vertex.
- Counting the number of edges.
- Identifying connected components of a graph.

For *efficiency*, we must make sure we visit each edge at most twice.

For *correctness*, we must do the traversal in a systematic way so that we don't miss anything.

Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Marking Vertices

The idea in graph traversal is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

For each vertex, we can maintain two flags:

- *discovered* - have we ever encountered this vertex before?
- *completely-explored* - have we finished exploring this vertex yet?

We must also maintain a structure containing all the vertices we have discovered but not yet completely explored.

Initially, only a single start vertex is considered to be discovered.

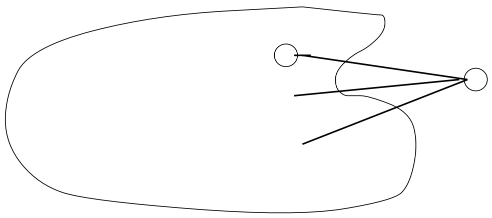
To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do.

Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Suppose not, ie. there exists a vertex which was unvisited whose neighbor *was* visited. This neighbor will eventually be explored so we *would* visit it:



Traversal Orders

The order we explore the vertices depends upon what kind of data structure is used:

- *Queue* – by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.

- *Stack* - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only if we are surrounded by previously discovered vertices. Thus our explorations quickly wander away from our starting point, defining a so-called *depth-first search*.

The three possible colors of each node reflect if it is unvisited (white), visited but unexplored (grey) or completely explored (black).

Breadth-First Search

```
BFS(G,s)
for each vertex u ∈ V[G] − {s} do
        color[u] = white
        d[u] = ∞, ie. the distance from s
        p[u] = NIL, ie. the parent in the BFS tree
color[u] = grey
d[s] = 0
p[s] = NIL
Q = {s}
while Q ≠ ∅ do
        u = head[Q]
        for each v ∈ Adj[u] do
                if color[v] = white then
                        color[v] = gray
                        d[v] = d[u] + 1
                        p[v] = u
                        enqueue[Q,v]
                dequeue[Q]
                color[u] = black
```

Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack.

Discovery and final times are sometimes a convenience to maintain.

DFS(G)
for each vertex $u \in V[G]$ do
    $color[u] = white$
    $parent[u] = nil$
$time = 0$
for each vertex $u \in V[G]$ do
    if $color[u] = white$ then DFS-VISIT[u]

Initialize each vertex in the main routine, then do a search from each connected component. BFS must also start from a vertex in each component to completely visit the graph.

DFS-VISIT[u]
$color[u] = grey$ (*u had been white/undiscovered*)
$discover[u] = time$
$time = time + 1$
for each $v \in Adj[u]$ do
    if $color[v] = white$ then
        $parent[v] = u$
        DFS-VISIT(v)
$color[u] = black$ (*now finished with u*)
$finish[u] = time$
$time = time + 1$