Chapter 9
Sorting and Searching

We saw lots of these in CS3220.

A list can be sorted if there is an ordering relation between the items in the list. Assume that there is such an ordering `get(X, Y)` on the lists of numbers, meaning `X` is greater than `Y`, defined as

```
gt(X,Y):-X>Y.
```

If the items in a list are atoms, then we can use the following one instead.

```
gt(X,Y):-X@>Y.
```

Let `sort(List, Sorted)` be such that `Sorted` is the sorted form of `List`, then we can write a few procedures for sorting a list in Prolog.

Bubble sort

The idea for bubble sort is the follows: 1) if we can find two adjacent elements, `X` and `Y`, in `List`, such that `gt(X,Y)` holds, we swap `X` and `Y` to obtain `List1`. Then, continue to sort `List1` into `Sorted`. 2) Otherwise, `List` is already sorted. Thus,

```
bubblesort(List,Sorted):-
 swap(List,List1),!,
 bubblesort(List1,Sorted).
bubblesort(Sorted,Sorted).

swap([X,Y|Rest],[Y,X|Rest]):- gt(X,Y).
swap([Z|Rest],[Z|Rest1]):- swap(Rest,Rest1).
```

Insertion sort

The idea for insertion sort is the following: 1) sort the tail of `List` first. 2) then insert the head, `X`, of `List` into the sorted tail such that the augmented list is sorted. Thus,

```
insertsort([],[]).
insertsort([X|Tail],Sorted):-
  insertsort(Tail,Sorted1),
  insert(X,Sorted1,Sorted).

insert(X, [], [X]).
insert(X,[Y|Sorted],[Y|Sorted1]):-
  gt(X,Y), !, insert(X,Sorted,Sorted1).
insert(X,Sorted,[X|Sorted]).

gt(X, Y):-X>Y.
```

Both bubble sort and insert sort are simple, but inefficient. We recall that their time complexities are $\Theta(n^2)$, where n refers to the length of `List`.

Quick sort

A much better approach to sort a list is quick sort, as we saw in the algorithm course.

To sort a non-empty list, `L`, we do the following:

1) Delete a *pivot,* `X` from `L`, and use it to split the rest of `L` into two parts: `Small`, containing everything less than the pivot; and `Big`, everything bigger than the pivot.

2) Sort `Small` to get `SortedSmall`.

3) Sort `Big` to get `SortedBig`.

4) Concatenate `SortedSmall`, `[X]`, and `SortedBig` into `Sorted`.

An implementation

```
quicksort([], []).

quicksort([X|Tail], Sorted):-
 split(X, Tail, Small, Big),
 quicksort(Small, SortedSmall),
 quicksort(Big, SortedBig),
 conc(SortedSmall, [X|SortedBig], Sorted).

split(X, [], [], []).
split(X, [Y|Tail], [Y|Small], Big):-
 gt(X, Y), !,
split(X, Tail, Small, Big).

split(X, [Y|Tail], Small, [Y|Big]):-
 split(X, Tail, Small, Big).
```

Homework: Exercises 9.1 and 9.4.

Graphs

Recall graphs are used in many different places. A graph is defined as a set of nodes, and a set of edges, where each edge is just a pair of nodes. When edges are directed, we call the graph a *directed graph;* otherwise, simply a *graph.* The edges can be attached with a numbers, representing cost, distance, among other things.

One way to represent a graph in Prolog is to use one clause for each edge. Thus, in a graph, if node `a` is connected to node `b`, we can simply say

```
connected(a,b).
```

On the other hand, if in a directed graph, an edge goes from node `a` to node `b` and is attached with a 3, we can say

```
arc(a,b,3).
```

More graph representations

Another way is to use two lists for each graph, one is for the set of nodes, the other for the set of edges. For example,

```
G1=graph([a,b,c],[e(a,b),e(a,c)]).
G2=digraph([a,b,c],[a(a,b,3),a(a,c,-1)]).
```

Yet another way is to use the idea of an adjacent list. For example,

```
G1=[a->[b],b->[a,c,d],d->[b]].
```

For a undirected graph, we can also use `Edges` to collect all the edges. Thus,

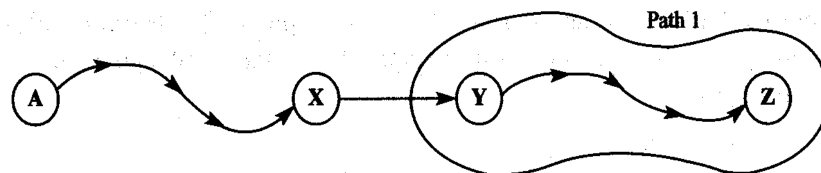
```
Edges([e(a, b), e(b, a), e(b, c), e(b, d), e(d, b)]).
```

We discussed most of these approaches in the data structure course, didn't we?

Find a path

One of the basic operations is to look for a path between two nodes in a graph. The relation `path(A,Z,G,P)` means to look for a path, `P`, between nodes `A` and `Z`, in a graph `G`.

The idea is very simple: 1) if `A=Z`, we are done. 2) Otherwise, find a path, `Path1`, from some node `Y` to `Z`, and then extend this path back to `A`, avoiding nodes in `Path1`.



In the above, `Path1` is an acyclic path in `G`, and `Path` is an acyclic path that goes from `A` to the beginning of `path1`, and continues along `path1` to `Z`.

The construction goes backwards: if `Y` is not the same as `A`, and if there is such a path, then there must be another node `X`, such that `X` is adjacent to `Y` in `G`, then we continue to look for the part from `A` to `X`. Thus the code.

```
path(A,Z,Graph,Path):-
 path1(A,[Z],Graph,Path).

path1(A,[A|Path1],_,[A|Path1]).

path1(A,[Y|Path1],Graph,Path):-
 adjacent(X,Y,Graph),
 not(member(X,Path1)), %acyclic path
  path1(A,[X,Y|Path1],Graph,Path).

adjacent(X,Y,graph(Nodes,Edges)):-
 member(e(X,Y),Edges).
adjacent(X,Y,graph(Nodes,Edges)):-
 member(e(Y,X),Edges).
```

Find a shortest path

We can add in another argument for the `path`, `path1` relations, i.e., its cost. Then, we can use it to look for a shortest path in the graph, using the following goal.

```
?-path(n1,n2,G,MinP,MinC),
   not(path(n1,n2,G,_,Cost)),Cost<MinC.

path(A,Z,Graph,Path,Cost):-
 path1(A,[Z],0,Graph,Path,Cost).

path1(A,[A|Path1],Cost1,Graph,[A|Path1],Cost1).
path1(A,[Y|Path1],Cost1,Graph,Path,Cost):-
 adjacent(X,Y,CostXY,Graph),
 not(member(X,Path1)),
 Cost2 is Cost1+CostXY,
 path1(A,[X,Y|Path1],Cost2,Graph,Path,Cost)
```