Non-deterministic Algorithms

As we have seen, a state space is a useful abstraction in analyzing problems. The value of the abstraction is that it provides a particular kind of modularization: One can consider separately the space to be searched and the algorithm used to search it.

However, as algorithms become complex, the language of states and operators quickly becomes too inexpressive and awkward. What is needed is a representation that combines the abstraction of a state space with the expressivity of a procedural programming language. This is achieved in the notion of a *non-deterministic algorithm.*

We will be using non-deterministic algorithms only at the level of pseudo-code. There are a couple of programming languages, such as Prolog, that support non-determinismm but these don't have ordinary programming language operators. (We will discuss Prolog later in the semester.) Non-determinstic operators cannot be easily added to standard programming languages; we will discuss the implementation of non-deterministic algorithms below.

The language of non-deterministic algorithms consists of six reserved words: **choose, pick, fail, succeed, either/or** . These are defined as follows:

- **choose** X satisfying P(X). Consider alternatively all possible values of X that satisfy P(X), and proceed in the code. One can imagine the code as forking at this point, with a separate thread for each possible value of X. If any of the threads succeed, then the choice succeeds. If a choose operator in thread T generates subthreads T1 ... Tk, then T succeeds just if at least one of T1 ... Tk succeeds.

  If thread T reaches the statement "**choose** X satisfying P(X)" and there is no X that satisfied P(X), then T fails.

- **pick** X satisfying P(X). Find any value V that satisfies P(V) and assign X := V. This does not create a branching threads.
- **fail** The current thread fails.
- **succeed** The current thread succeeds and terminates.
- **either** S1 **or** S2 **or** S3 ... **or** Sk. Analogous to choose. Create k threads T1 ... Tk where thread Ti executes statement Si and continues.

Thus, a thread fails and terminates if either (a) it encounters a **choose** statement with no choices; (b) it encounters a **fail** statement. A thread succeeds and terminates if either (a) it terminates in the usual way (by reaching the end of the code) or (b) it encounters a **succeed** statement. A thread that goes into an infinite loop is considered to fail, though of course this is not generally detectable.

Relation to state space

A non-deterministic algorithm applied to a program can be considered as defining a state space as follows: Each time you encounter a a **choose** or an **either/or** statement in executing the code, that defines a node in the state space. The state at this point is the state of the algorithm (value of the variables etc.) An operator on state S is a "chunk of code" executed between this call to **choose** and the next call to **choose.**

Another way to conceptualize non-deterministic algorithms

Another way to conceptualize non-deterministic algorithms is to imagine that the algorithm can "magically" make a choice that leads to success. A non-deterministic algorithm is correct if (a) there is a sequence of choices that leads to success; and (b) any sequence of choices that leads to success correctly solves the problem.

Implementation

Non-deterministic algorithms are not generally implemented using actual threads; the number of threads would have to be equal to the size of the state space, and most operating systems do not allow this. Rather, the non-deterministic algorithm is considered as defining the state space described above, and any program that exhaustively searches that space is a valid implementation of the non-deterministic algorithm. The search technique is up to the implementor; he can use depth-first search, breadth-first, iterative deepening ...

Some examples

General comment on pseudo-code: I use a combination of the notations I like in Pascal, C, and Ada, together with English. I hope it's clear enough. In particular, the labelling of procedure parameters as "in", "out", and "in/out" is taken from Ada. Following Pascal, assignment is notated := and equality is notated =. I declare variables only when I feel like it.

N Queens problem

Place N queens on an NxN board so that no two queens can take one another.

```
function attacked(in I,J,B) : returns boolean;
{ attack = false;
  for K := 1 to I-1 do
     if ((B[K] = J) or  ((B[K]-J)=(K-I)) or ((B[K]-J)=(I-K))
       then attack = true;
  return(attack);
}

N-QUEENS1(in N : integer; out B : array[1..N] of integer)
  /* B[I] = the row of the queen in the Ith column. -1 initially */

{  B := -1;
   for I := 1 to N do {
     choose J in 1..N such that not attacked(I,J,B)
     B[I]=J;
    }
}
```

N-QUEENS1 above fills in the board left to right. Using "pick" we can generalize that to fill in columns in arbitrary order, to be chosen by the implementor.

```
function attacked(in I,J,B) : returns boolean;
{ attack = false;
  for K := 1 to N do
     if ((B[K] != -1) and
         ((B[K] = J) or  ((B[K]-J)=(K-I)) or ((B[K]-J)=(I-K)))
       then attack = true;
  return(attack);
}

N-QUEENS2(in N : integer; out B : array[1..N] of integer)
  /* B[I] = the row of the queen in the Ith column. -1 initially */

{  B := -1;
   for K := 1 to N do {
     pick I in 1..N such that B[I]=-1;
     choose J such that not attacked(I,J,B)
     B[I]=J;
    }
}
```

EXACT SET COVER

Given: A set O and a collection C of subsets of O.
Find: A subcollection D of C such that every element of O is in exactly one set in D. (In other words, the union of the sets in D is O and the intersection of any two sets in D is empty.)

```
Example:
O = { a,b,c,d,e,f,g,h,i,j,k,l }
C = { C1, C2, C3, C4, C5, C6, C7, C8 } where
   C1 = {a, b, c}
   C2 = {a, d}
   C3 = {b, c, d, e}
   C4 = {b, c, h, k}
   C5 = {c, f, g}
   C6 = {e, g, i}
   C7 = {f, j, l}
   C8 = {f, k}
```

Solution: D={C2, C4, C6, C7}

(Note: EXACT SET COVER is a problem where you cannot a priori predict the depth of the shallowest goal state, so iterative deepening may well to work much better than DFS.)

Non-deterministic algorithm:

```
XCOVER1(in O,C)
{ D := empty;
  DU := empty;   /* union of elements in D */
  while (there are elements in O not in DU) {
     choose set X in C such that X does not overlap DU;
     add X to D;
     DU := DU union X;
   }
  return(D)
}
```

Corresponding search space: State: A subcollection of C.
Operator on state S: Add to S an element of C that does not overlap the union of the sets in S.
Start state: The empty collection.
Goal state: An exact set cover.

A problem with the above algorithm is that it generates each collection D in every possible order. For example, one branch will first choose X=C2, then X=C4, then X=C6, then X=C7. A different branch will first choose X=C7, then X=C2, then X=C6, then X=C4, finding the same solution in a different order. Indeed, every permutation will be discovered on a separate branch of the algorithm. That is, the state space is *non-systematic*; different paths through the state space lead to the same state.

We can fix the above problem by requiring that every subcollection of D be constructed in a specified order:

```
XCOVER2(in O,C)
{ pick L to be an ordering on the sets in C;
  D := empty;
  DU := empty;   /* union of elements in D */
  while (there are elements in O not in DU) {
     choose set X in L such that X does not overlap DU;
     add X to D;
     DU := DU union X;
     L := the tail of L following X;
   }
  return(D)
}
```

Thus, if L is picked as the order C1, C2, C3, C4, C5, C6, C7, C8, this will only find the solution in the order C2, C4, C6, C7. The operator "pick" is appropriate for L because the solution can be found whatever order is picked for L (though some orderings may be more efficient than others) and therefore, when one ordering fails, there's no point in going back and trying a different ordering.

An alternative solution to the same problem:

```
XCOVER3(in O,C)
{ D := empty;
```

```
   DU := empty;   /* union of elements in D */
   while (there are elements in O-DU)
      pick an element W in O-DU;
      choose a set X such that W is in X and X does not overlap DU;
      add X to D;
      DU := DU union X;
    }
   return(D)
}
```

Thus, for example, if the elements W are picked in alphabetical order, the algorithm will first choose a set to cover "a"; then, on the branch where it has chosen C2, it will choose a set to cover "b"; then, on the branch where it has chosen C4, it will choose a set to cover "e", then on the branch where it has chosen C6, it will choose a set to cover "f".

Here, the selection of element W can even depend on the earler choices of sets in D. Whatever selection criterion is used for W, any set D can only be constructed in one way. For instance, suppose that we pick the first element of O-DU that comes after the last element of DU, going cylically around. Then the algorithm will first choose a set to cover "a"; then, on the branch where it has chosen C2, it will choose a set to cover "e"; then, on the branch where it has chosen C6, it will choose a set to cover "j"; then on the branch where it has chosen C7, it will choose a set to cover "b".

BETWEENNESS problem

In a list of items, we say that Y is between X and Z if either [X precedes Y and Y precedes Z] or [Z precedes Y and Y precedes X]. Thus, for example, in the list [C,A,D,E,B,F], D is between C and B; E is between F and D; and so on.

The following problem is known as the BETWEENNESS problem: The input is a set of constraints of the form ``Y is between X and Z". The problem is to find a list that obeys all these constraints. For instance, given the constraints

A is between C and B.
D is between B and C.
D is between A and E.
E is between D and F.
B is between F and A.

one solution is the list [C,A,D,E,B,F]. The BETWEENNESS problem is NP-complete, so presumably there is no polynomial-time algorithm.

Non-deterministic algorithms

```
BETWEEN1 (Z : set of items; B : set of betweenness constraints)
return list of items;

O := empty list;
for each item X in Z  do {
  choose a position in O to insert X;
  if any constraint in B is violated then fail;
 }
return O.
```

State space: State: Permutation of Z[1..K].
Operator on state S: Add Z[K+1] to S at some position in S such that the constraints are satisfied.
Start state: empty set.
Goal state: Permutation of Z observing the constraints.

```
BETWEEN2 (Z : set of items; B : set of betweenness constraints)
return list of items;
{  O := empty list;
   while Z is non-empty do {
      choose an item X in Z;
      add X to the end of O;
      if any constraint in B is violated then fail;
      delete X from Z;
    }
return O;
```

State space:\\ State: Ordered list of items (first K items in the output).
Operator on state S: Add some item in Z to the end of S, consistent with the constraints
Start state: empty list
Goal state: Permutation of Z satisfying the constraints.

```
BETWEEN3 (S : set of items; B : set of betweenness constraints)
return list of items;
{ G := the graph which has the items of S as vertices and has no edges;
  for each constraint C of the form ``Y is between X and Z'' in B do
    either add edges X -> Y and Y -> Z to G;
    or add edges Z -> Y and Y -> X to G;
  if G has a cycle then fail;
return a topological sort of G.
```

State space:
State: A graph over the items in S consistent with the first K constraints in C.
Operator on S: Take the K+1st constraints "Y is between X and Z" and either add arcs X -> Y and Y -> Z or add arcs Z -> Y and Y -> X to S.
Start state: The graph with no edges.
Goal state: A graph that enforces all the constraints in C.

Semantics and implementation of a non-deterministic algorithm

Let's start with the case where there are no "pick" statements. A *successful* branch in a non-deterministic algorithm is some execution of the code with some particular sequence of choices at the "choose" and "either/or" statements that ends in success (either in the execution of "succeed" or in reaching the end of the program.) A non-deterministic algorithm *succeeds* if some branch succeeds. A non-deterministic algorithm A *returns* value V if V is returned by some successful branch of A. (Thus, A can return many different values.) Algorithm A *correctly solves* problem P if the following holds

- If A returns V, then V is a solution to P.
- If A returns no value, then P has no solution.

Note that it is not required that A return every possible solution to P.

If there are "pick" statements, the first definitions become a little more involved. Let A be a non-deterministic algorithm. Let B be a branch of an execution of A up to some execution E of a statement S of the form "pick X such that Z(X)". A *picking strategy* Q is a function that maps any such branch B to a value of X that satisfies the condition in S. any execution of a statement "pick X such that Z(X)" into some particular X such that Z(X). A successful branch of non-deterministic algorithm A under pick strategy Q is one execution of the code with some choice of values at the "choose" and "either/or" statements, where the values picked at the execution of pick statements is governed by Q, that ends in success. Algorithm A succeeds under pick strategy Q if some branch of A under Q succeeds. Algorithm A returns value V under Q if some successful branch of A under Q returns V. Algorithm A correctly solves problem P if, for every pick strategy Q, the following conditions hold:

- If A returns V under Q, then V is a solution to P.
- If A returns no value under Q, then P has no solution.

Therefore, an implementation I of non-deterministic algorithm A is correct if it satisfies the following properties:

- If I returns value V, then A returns V.
- If I returns no value, then A returns no value.

Again it is not required that I find *every* value returned by A, just that if A finds some value, then I finds some value.

It follows immediately that if I is a correct implementation of algorithm A and A correctly solves problem P, then I correctly solves problem P.

Converting a non-deterministic algorithm to code executing DFS

Let S be an instance in the code of the statement "choose X such that P(X)". Each execution E1 of S is translated an execution of the loop

```
for (X such that P(X)) do g(X)
```

where g(X) executes all the code that occurs between the execution E1 and the eventual success or failure. In particular, if S is executed again within the branch following E1 then these further executions must occur within the function g(X) (which is presumably recursive.)

For instance, the code

```
while (A) do
  { choose X such that P(X);
    B(X);
    if (C) then succeed else if (D) then fail;
  }
```

can be implemented as a DFS as follows:

```
f(X)
{ if (A) {
   found := false;
   for (X1 such that P(X1)) {
      B(X1);
      if (C) then found := true;
        else if (D) then noop;
        else found := f(X1);
      if found then exitloop;
    }
   return(found)
  }
}
```