CS F364
Design & Analysis of Algorithms

Algorithms – Design Techniques

Exact Solutions for Hard Problems

- Search with Backtracking

BackTracking: Template

BackTracking - Approach

- Solution space can also be viewed as “certificate” space:
  - i.e. Searching for a solution can be viewed as “constructing” a “valid” certificate (and testing it)
- Recall non-deterministic (ND) machines
  - ND machines can be simulated deterministically by exploring all possibilities exhaustively !
- (Deterministic) Template for “backtracking”
  1. Systematically construct a solution and test it
     a. If a test fails backtrack to find an alternate solution
  2. Repeat step 1 until valid solution is found.

BACKTRACKING - ALGORITHMIC TEMPLATE - OUTLINE

Algorithm_Template Backtrack(x):

// x is a problem instance

```
F = { (x,{}) }          // F is a set of configurations
while (F not empty) do {
   // inspect configurations in F one by one



}
return "no solution"
```

Backtracking - Algorithmic Template - Configurations

Algorithm_Template Backtrack(x): // x is a problem instance

```
    F = { (x,{}) }            // F is a set of configurations
    while (F not empty) do {
      // inspect configurations in F one by one
      select the most promising configuration (x,y) from F;
      expand (x,y) by making additional choices to get a set of
           new configurations C = { (x1,y1), (x2,y2), …,(xk,yk) };
       for each (xj,yj)  in C {
          // validate(xj,yj)


       }
    }
    return "no solution"
```

BACKTRACKING - ALGORITHMIC TEMPLATE

Algorithm_Template Backtrack(x): // x is a problem instance

```
F = { (x,{}) }            // F is a set of configurations
while (F not empty) do {
   // inspect configurations in F one by one
   select the most promising configuration (x,y) from F;
   expand (x,y) by making additional choices to get a set of
            new configurations C = { (x1,y1), (x2,y2), …,(xk,yk) };
    for each (xj,yj)  in C {
       // validate(xj,yj)
        if solution found return the solution derived from (xj,yj);
        else if  dead end then discard;  // backtrack
        else F = F U { (xj,yj) }
    }
}
return "no solution"
```

Backtracking - Examples: CNF-SAT, HAM-PATH

Backtracking – example – CNF-SAT

- Input : Boolean formula S in CNF
  - A configuration:
    - (S’, y) where S’ is a Boolean formula in CNF and
    - y is a set of assignments to variables not in S’
      - such that making these assignments in S results in S’
  - “Promising configuration”
    - Most constrained of all formulas in F:
      - S’ containing the smallest clause

BACKTRACKING – EXAMPLE – CNF-SAT [2]

- Input : Boolean formula S in CNF
  - Sub problems:
    - Locate the smallest clause C in S'
    - Pick a variable x_j that appears in C
    - Create subproblems by assigning $x_j=1$, $x_j=0$ and simplifying S' accordingly
  - Validation:
    - Assignment creates a contradiction: "dead end"
    - Assignment reduces S' to an empty clause: "found solution"

BACKTRACKING ALGORITHM FOR CNF-SAT

- Algorithm BACK_SAT(S): // S is a Boolean formula in CNF

```
F = { (S,{}) }
while (F not empty) do {
   let (S1,A1) be the configuration in F containing the smallest
    clause;
   let C be the smallest clause in S1 and let x be any var. in C;
   for each b in {0, 1}      {
      let S2 be the formula obtained by simplifying S1 with x=b;
      if  (S2 is empty) then return  A1 U {x=b} ;
      else if  (S2 is a contradiction) then “ignore”;  //backtrack!
      else   F = F U { (S2, A1 U {x=b}) };
     }
}
return “no solution”
```

Backtracking - Example

- Exercise: Hamiltonian Path
  - Input : ?
  - What is a configuration?
  - What is a “Promising configuration”?
  - What are sub problems?
  - How do you validate?

Backtracking
- Application: Prolog

Evaluation of Prolog Queries (given a program):

- Prolog resolves queries against a given program :
  - A program is a set of rules
  - Resolution is achieved by
    - matching the query with rules to generate sub-queries;
    - a query is resolved if all sub-queries (generated recursively) are resolved.
  - Rules are searched for matching
    - When search fails for a sub-query or matching fails the resolver backtracks and searches another path.
      - i.e. ***backtracking is built into Prolog search engine***

Evaluation of Prolog Queries – Resolution steps

1. Match query term with the head of a rule
2. If matching succeeds, add each of the other sub-clauses as a query; continue;
3. If matching fails or sub-query fails, **backtrack**;
4. If no more rules to backtrack fail

PROLOG PROGRAM AND QUERY

- Evaluation of Prolog Queries (given a Prolog program):
  - A program is a set of rules in Horn Clause form
    - e.g.
      - grandparent(X,Y):-parent(X,Z), parent(Z,Y).
      - parent(X,Y):-father(X,Y).
      - parent(X,Y):-mother(X,Y).
      - mother(ada,bebe).
      - mother(bebe,bart).
      - father(bart,catniss).
  - A sample query:
    - grandparent(bebe,catniss)?

Evaluation of Prolog Queries - Example

- match “grandparent(bebe,catniss)” with “grandparent(X,Y)”
  - Step 1
- add “parent(bebe, Z)” and “parent(Z, catniss)” to list of queries
  - Step 2
- for resolving “parent(bebe, Z)” add “father(bebe,Z) as query.
  - Step 2

Evaluation of Prolog Queries – Example [contd..]

- "father(bebe,Z)" fails to match with father(bart, catniss) ;
  - so backtrack and look for another "father" rule. (Step 3)
- "father(bebe,Z)" fails;
  - so backtrack; (Step 4)
- this is subquery for "parent(bebe,Z)"
  - backtrack (step 3)
- add "mother(bebe,Z)" to list of queries
  - (step 2)
- ...

- Exercise: Completely resolve this example query.