Backtracking

- enumerates a set of partial candidates by generating a tree structure of potential candidates and traversing the tree structure (potential search tree)
- traverses in depth-first inorder fashion from root (recursively)
- at each node, c, if see does not lead to a valid solution, the subtree rooted at c is pruned
- otherwise, check if c is a valid solution or recursively check the subtree rooted at c
- the two tests (valid solution or tree to prune) are user-defined procedures
- the actual search tree that is traversed is only one part of the potential search tree

Outline

- provide the data, P, that should be checked
  - root(P): return the partial candidate at the root of the search tree
  - reject(P,c): return true only if the partial candidate c is not worth completing
  - accept(P,c): return true if c is a solution of P, and false otherwise
  - first(P,c): generate the first extension of candidate c
  - next(P,s): generate the next alternative extension of a candidate, after the extension s
  - output(P,c): use the solution c of P, as appropriate to the application

```
procedure bt(c)
  if reject(P,c) then return
  if accept(P,c) then output(P,c)
  s ← first(P,c)
  while s ≠ Λ do
    bt(s)
    s ← next(P,s)
```

Related problems

- eight queens
- crosswords
- verbal arithmetic
- sudoku
- peg solitaire
- combinatorial
  - parsing
  - knapsack
- general constraint satisfaction
  - P - instance data

    - F - predicate testing satisfaction

```
function first(P,c)
  k ← length(c)
  if k = n
    then return Λ
  else return (c[1], c[2], ..., c[k], 1)

function next(P,s)
  k ← length(s)
  if s[k] = m
    then return Λ
  else return (s[1], s[2], ..., s[k−1], 1 + s[k])
```

- additionally:
    - solution spaces
    - traveling salesperson
    - convex hull/graham scan
    - permutation generation
- permutations

```
permute(i)
   if i == N:
      output A[N]
   else:
      for j = i to N:
         swap(A[i], A[j])
         permute(i + 1)
         swap(A[i], A[j])
```

Knuth - Introduction to Backtracking

- Queen's problem
- Walker's backtrack
- Permutations and Langford pairs
- word rectangles
- commafree codes
- dynamic ordering of choices
- sequential allocation redux

- estimated cost of backtrack

Template for recursive backtracking

```
void find_solutions(n, other params) {
    if (found a solution) {
        ++solutions_found;
        display_solution();
        if (solutionsFound >= solutionTarget) {
            exit(0);
        }
        return;
    }

    for (val = first to last) {
        if (is_valid(val, n)) {
            apply_value(val, n);
            find_solutions(n + 1, other params);
            remove_value(val, n);
        }
    }
}
```

in general

- useful for constraint satisfaction problems that involve assigning values to variables according to a set of constraints.
- n-queens:
    - variables = queen's position in each row
    - constraints = no two queens in same row, column, diagonal
- map coloring
    - variables = each state's color
    - constraints = no two bordering states with the same color
- many others: factory scheduling, room scheduling, etc.
- backtracking reduces the # of possible value assignments that we consider, because it never considers invalid assignments.…
- using recursion allows us to easily handle an arbitrary number of variables.
- stores the state of each variable in a separate stack frame

recursion vs. iteration

- recursive methods can often be easily converted to a non-recursive method that uses iteration
- this is especially true for methods in which:
    - there is only one recursive call
    - it comes at the end (tail) of the method these are known as tail-recursive methods

- when comfortable with recursion, some algorithms are easier to implement using recursion
- some data structures lend themselves to recursive algorithms
- recursion is a bit more costly because of the overhead involved in invoking a method
- rule of thumb:
  - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
  - otherwise, use iteration