Implementation Details

There are three typical patterns for recursive backtracking algorithms, depending on the goal:

- find a solution
- find all solutions
- find an optimal solution

Implementation Patterns

```
// find a single solution
solution solve ( partial solution, subproblem ) {
  if the partial solution is complete,
    return it as the solution
  else
    for each legal next choice
      generate partial solution and subproblem
       with that choice made
      result = solve(new partial solution,
                     new subproblem)
      if result is a solution,
        return it
    return no solution
}
```

an alternative is that the partial solution is updated so that it holds the complete solution, and true/false is returned indicating whether a solution has been found

Implementation Patterns

```
// find all solutions
void solve ( partial solution, subproblem,
             solution list ) {
  if the partial solution is complete,
    add it to the solution list
  else
    for each legal next choice
      generate partial solution and subproblem
       with that choice made
      solve(new partial solution,
            new subproblem,solution list)
}
```


Implementation Patterns

```
// find optimal solution
solution solve ( partial solution, subproblem,
                 best solution so far ) {
  if the partial solution is complete,
    return the better of it and the best so far
  else
    for each legal next choice
      generate partial solution and subproblem
       with that choice made
      result = solve(new partial solution,
                     new subproblem,best so far)
      if result is a solution and better than the
       best so far,
        update the best so far
    return the best so far
}
```

Running Time

How long does this take?

- DFS is O(n+m)
  - *n* = number of vertices, *m* = number of edges

How big is the state space graph?

- branching factor *b* – number of next choices
- longest path *h* – largest number of decisions needed to reach a base case

→ worst case $n = O(b^h)$, $m = O(b^{h+1})$

- if there are multiple paths to the same vertex, *n* can be much smaller – but without storing discovered vertices, repeat visits are handled the same as new visits (and storing discovered vertices takes exponential space)

This...is not good.

Key Points – Making Backtracking Practical

- recursive backtracking is generally not practical without additional effort
  - DFS is O(n+m) where $n = O(b^h)$
    - b = branching factor – number of next options for each choice
    - h = length of longest path – (maximum) number of choices made to get to a complete solution

Key Points – Making Backtracking Practical

- while reducing how much is explored is the dominating factor, it is also important to be efficient in what is done for each subproblem
  - determining whether or not to prune must be efficient
  - modify/restore rather than copying for generating subproblems and partial solutions
  - exploit clever representations

Generating New Partial Solutions and Subproblems

- making a choice typically means an incremental change to the current partial solution and subproblem
- generating the new by copying the old may be expensive
  - copying a collection takes time proportional to the size of the collection

Instead, it may be more efficient to modify the current partial solution and subproblem and then undo.

```
for each legal next choice
  add choice to partial solution and remove
    from subproblem
  result = solve(modified partial solution,
               modified subproblem)
  remove choice from partial solution and add
    to subproblem
```