1 Backtracking

So far, we covered the following algorithm design techniques:

1. incremental,
2. divide-and-conquer,
3. dynamic programming,
4. greedy.

Each technique allows us to make progress, either implied (incremental approach), or as part of the optimal substructure (divide-and-conquer, dynamic programming, greedy approach). If we cannot use the problem's structure to make progress and we want an exact, not an approximate solution, we can use the backtracking technique, but only as a last resort.

1.1 Navigating a maze

Consider the problem of navigating a maze. In general, one does not know the structure of the maze when beginning. In fact, a maze may be constructed not to have any particular structure. So there is no obvious way to break down the problem into smaller pieces.

One way to solve a maze is to try every potential solution in the search space, i.e. every possible sequence of steps from the maze entrance to its exit. In order to narrow down and systematize the search, we will do the following: if we hit a dead end, then we back up until the last choice that we made and make another choice.

1.2 A definition

The basic strategy of backtracking is to *systematically* explore the space of all potential solutions. It does so by:

- extending a partial solution in some feasible way,
- trying another extension if the extension fails,

- backing up to a smaller partial solution when the options for extending a partial solution are exhausted.

We should use backtracking only as a last resort since it is expensive.

2 n-Queens problem

Input: An integer $n \geq 4$.

Output: A placement of n queens on an $n \times n$ chessboard so that no two can attack each other.

Recall that a queen may move any length along a diagonal or any length along a row or column. So, for example, if $n = 4$, a solution is $(1, 3)$, $(2, 1)$, $(3, 4)$ and $(4, 2)$.

In order to use backtracking for this problem, we need to describe what a potential solution looks like.

2.1 Potential solutions

As seen in the example above, a potential solution is n points in an $n \times n$ array:

$$(x_1, y_1), (x_2, y_2), ..., (x_n, y_n).$$

Note that an exhaustive search will check $(n^2)^n$ possible solutions, one for each set of n coordinates. Let us use backtracking to narrow down the search space. To do this, we need to develop tests that check whether a partial solutions cannot be extended to a complete solution.

2.2 Analyzing the partial solutions and pruning

We can make several observations that eliminate some of the possible partial solutions:

1. There must be exactly one queen per row. This is because two queens in a row would allow them to attack each other, while fewer than one queen per row would not allow n queens to be located on the board. This means that the solutions must have the form:

   $$(1, y_1), (2, y_2), ..., (n, y_n).$$

2. There must be exactly one queen per column, for the same reason as above. This means that $(y_1, y_2, ..., y_n)$ must be a permutation of $(1, 2, ..., n)$.

   The search space has thus been decreases to "only" $n!$ permutations.

3. There is at most one queen on each diagonal, since two queens on a diagonal would be able to attack each other. Suppose we have queens at positions (a, b) and (c, d) in a possible solution. This restriction means that $|a - c| = |b - d|$.

We say that we prune a partial solution if we do not extend it any further because it cannot be extended to a complete solution.

2.3 The strategy

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

Step 1: Choose a column for the queen in row 1.

Step i: Given the positions of the first $i - 1$ queens $(1, y_1), (2, y_2), ..., (i - 1, y_{i-1})$, choose the position for (i, y_i). If you cannot complete Step i for some choice of previous positions $(1, y_1), (2, y_2), ..., (i - 1, y_{i-1})$, then backtrack and re-choose the position for $(i - 1, y_{i-1})$.

We will prune a partial solution if it forces the placement of two queens on the same column or diagonal. Consider the example of the 4-Queens problem:

Step 1: Place a queen in the 1st row. The choices are $(1, 1), (1, 2), (1, 3), (1, 4)$. We first choose $(1, 1)$

Step 2: Place a queen in the 2nd row. Given our choise of $(1,1)$, we cannot choose either $(2,1)$ [same column] or $(2,2)$ [same diagonal]. The remaining choices are $(2,3),(2,4)$.

Step 3: Given our choices of $(1,1),(2,3)$ we cannot choose $(3,1)$ [same column], $(3,2)$ [same diagonal], $(3,3)$ [same column], or $(3,4)$ [same diagonal]. So we eliminate choice $(1,1),(2,3)$.

Step 2: Given our choice of $(1,1)$, the remaining choice in row 2 is $(2,4)$.

Step 3: Given the choice $(1,1),(2,4)$, we cannot choose $(3,1)$ [same column], $(3,3)$ [same diagonal], $(3,4)$ [same column]. Our only choice is $(3,2)$.

Step 4: Given our partial solution $(1,1),(2,4),(3,2)$, we cannot choose any of $(4,1)$ [same column], $(4,2)$ [same diagonal], $(4,3)$ [same diagonal] or $(4,4)$ [same column]. Since we have no more choices in step 3 and step 2, we backtrack to step 1 and eliminate partial solution $(1,1)$.

Step 1: We choose $(1,2)$.

Step 2: given our choice of $(1,2)$, we cannot place a queen in $(2,1)$ [same diagonal], $(2,2)$ [same column] or $(2,3)$ [same diagonal]. That leaves only $(2,4)$ as a choice.

Step 3: Given choice $(1,2),(2,4)$, we cannot place a queen in $(3,2)$ [same column], $(3,3)$ [same diagonal] or $(3,4)$ [same column]. This leaves only choice $(3,1)$.

Step 4: Given choice $(1,2),(2,4),(3,1)$, we cannot place a queen in $(4,1)$ [same column], $(4,2)$ [same column], or $(4,4)$ [same column]. This leaves only choice $(4,3)$.

A solution is thus: $(1,2),(2,4),(3,1),(4,3)$. Draw the tree representing our backtracking steps as an exercise!

2.4 Algorithms

We will consider a recursive version of the n-Queens algorithm. Let $Q[1...n]$ store the positions of the n queens, i.e. $Q[i]$ will be the column of the queen in row i.

```
// extends a partial solution of size k-1 to one of size k,
// unless the partial solution is a complete solution, in
// which case it is output; the initial call is NQueens(Q,1,N)
NQueens(Q,k,n)
if k=n+1 then
  output Q[1..n]
  quit // if only one solution is needed
for i = 1 to n
  Q[k] = i
  if Feasible(Q, k)
    then NQueens(Q, k+1, n)
```

In both algorithms, we use a procedure `Feasible(Q, k)` that checks whether $Q[1..k]$ is feasible, given that $Q[1..k-1]$ is. This procedure checks that no queen in rows $1, ..., k-1$ is in the same column or diagonal as the queen in position $(k, Q[k])$.

```
Feasible(Q, k)
for i = 1 to k-1
  if Q[k] = Q[i] or |Q[i] - Q[k]| = |i-k| then
    return false
return true
```

3 General recursive backtracking

```
Backtrack(partial solution)
if partial solution is a solution then
  output solution
  return (or quit)
for all possible extensions of partial solution
  if extension is feasible then
    Backtrack(extension)
```

4 Graph Coloring

Input: A (undirected) graph $G = (V, E)$ and an integer k.

Output: An assignment of one of k colors to each node, such that adjacent nodes get different colors.

For example, let $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (2, 3), (2, 4), (3, 4)\}$ and suppose that $k = 3$. A valid coloring c of G is: $c(1) = R, c(2) = G, c(3) = B, c(4) = R$.

Example 1 *A classical theorem in graph theory, the Four Color Theorem, proved in 1976 using a computer, states that any planar graph can be properly colored with four colors. So, for example, K_5 and $K_{3,3}$ are not planar.*

Example 2 *The register allocation problem is a graph coloring problem in disguise.*

4.1 Potential solutions

Suppose that $|V| = n$. Then $(c_1, c_2, ..., c_n)$ is a possible coloring of G where c_i is the color of node i in G. Note that there are k^n possible colorings. A coloring is *feasible* or *valid* if no two adjacent nodes are given the same color, that is, if $(i, j) \in E$ then $c_i \neq c_j$.

Consider a graph $G = (V, E)$ where $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}$ and let $k = 3$. There are six valid colorings of G given in the following table:

node	p	q	r	s	t	u
1	R	R	G	G	B	B
2	G	B	B	R	R	G
3	B	G	R	B	G	R
4	R	R	G	G	B	B

Note that all these colorings are sort of equivalent. They all share the following structure:

- The same color is used for both node 1 and node 4. For colorings p and q, it is R, for colorings r and s, it is G, and for colorings t and u, it is B.

- Nodes 2 and 3 must have distinct colors different from each other and from the color used for nodes 1 and 4.

Definition 1 *Two colorings are equivalent if one can be transformed into another by permuting the k colors.*

4.2 Using backtracking to find valid colorings

We will use the following strategy to find all valid colorings of a graph $G = (V, E)$:

1. Order nodes arbitrarily.
2. Assign the first node a color.
3. Given a partial assignment of colors $(c_1, c_2, ..., c_{i-1})$ to the first $i - 1$ nodes, try to find a color for the i-th node in the graph.
4. If there is no possible color for the i-th node given the previous choices, backtrack to a previous solution.

Consider the graph G given earlier, where $E = \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\}$ and $k = 3$.

Step 1: Choose a color for node 1. It can be one of: R, B or G. Say we choose R.

Step 2: Given partial coloring (R), we choose a color for node 2. It can be one of: G or B. Say we choose G.

Step 3: Given partial coloring (R, G), we choose a color for node 3. It cannot be either R or G, so it must be B since k = 3.

Step 4: Given partial coloring (R, G, B), we choose a color for node 4. It cannot be B or G, so it must be R. This gives the coloring (R, G, B, R) which is coloring p.

We have no more choices of colors in step 4, and in step 3. We have one choice in step 2.

Step 2: Given partial coloring (R), we choose a different color for node 2. We choose B for node 2.

Step 3: Given partial coloring (R, B), we choose a color for node 3. It cannot be R nor B, so it must be G.

Step 4: Given partial coloring (R, B, G), we choose a color for node 4. It cannot be B or G, so it must be R. This gives the coloring (R, B, G, R) which is coloring q.

We have no more choices of colors in steps 4, 3 and 2. We go back to step 1.

Step 1: We choose a different color for node 1, say B. This will produce a branch in the tree equivalent to the first branch where R and B are switched. Thus we will get the colorings t and u.

If we choose G for node 1 then, we will again get a branch equivalent to the first one with R and G swapped. This will produce the colorings r and s.

4.3 Algorithm

We now give a recursive version of the graph coloring algorithm. Let $C[1...j-1]$ be a partial coloring for the first $j-1$ nodes.

```
Color(C,j,k,n)
if j = n+1 then
  output C
  return or quit
for i = 1 to k
  C[j] = i
  if valid(C,j,n) then
    Color(C,j+1,k,n)
```

where

```
Valid(C,j,n)
for all neighbors v of j with v < j
  if C[v] = C[j] then
    return false
return true
```

4.4 Pruning

If we are simply looking for a single solution, we can cut off the equivalent branches of the tree to save time. For example, the three main branches of the backtracking tree obtained in section 4.2 all gave equivalent solutions. We need only consider the first branch if we want a single solution.

The following algorithm prunes the tree to remove equivalent branches. It uses the following strategy:

- Keep track of the largest color used so far.
- Try the previously used colors first.
- Never try more than one new color.

```
ColorP(C,j,last,k,n)
if j = n+1 then
  output C
  return or quit
for i = 1 to last //try old colors first
  C[j] = i
  if valid(C,j,n) then
    ColorP(C,j+1,last,k,n)
if last < k then
  C[j] = last + 1
  ColorP(C,j+1,last+1,k,n)
```

5 Subset sum

We now return to a problem introduced in homework 3.

Input: A set $T = \{t_1, t_2, ..., t_n\}$ of positive integers; an integer M.

Output: A subset S of T such that $\sum_{x \in S} x = M$.

You solved this problem using dynamic programming. Let us apply backtracking to it.

5.1 Potential solutions

Let $(x_1, x_2, ..., x_n)$ be a representation of a potential solution such that $x_i = 1$ if $t_i \in S$ and $x_i = 0$ if $t_i \notin S$.

Example 3 *Let* $T = \{4, 7, 6, 3, 1\}$ *and* $M = 10$.

$(1, 0, 1, 0, 0)$ *gives* $4 + 6 = 10$*, so it is a valid solution.*

$(0, 0, 1, 1, 1)$ *gives* $6 + 3 + 1 = 10$*, so it is a valid solution.*

$(1, 0, 0, 1, 1)$ *gives* $4 + 3 + 1 = 8$*, so it is* **not** *a valid solution.*

We will use the following backtracking strategy:

1. Start with $S = \{\}$.
2. Given a partial list (of length $i - 1$) of elements used, check if element i can be used.
3. Backtrack or prune as needed.

Let us develop a test for pruning a partial solution $X = (x_1, x_2, ..., x_k)$. We can clearly prune X if

$$\sum_{i=1}^{k} x_i t_i > M$$

Furthermore, X can be pruned if

$$\sum_{i=1}^{k} x_i t_i + \sum_{i=k+1}^{n} t_i < M.$$

Let us implement this strategy on input $T = \{4, 7, 6, 3, 1\}$ and $M = 10$. In drawing the tree, we will label each internal node with the sum so far.

Step 1: Choose 4 or not?

Suppose we don't and get partial solution (0).

Step 2: Choose 7 or not?

Suppose we don't and get partial solution $(0, 0)$.

Step 3: Choose 6 or not?

Suppose we don't and get partial solution $(0, 0, 0)$. Since $3 + 1 = 4 < 10$, this partial solution won't lead to a solution so we prune.

Suppose we choose 6, and get partial solution $(0, 0, 1)$.

Step 4: Choose 3 or not?

Suppose we don't and get partial solution $(0, 0, 1, 0)$. Since $6 + 1 = 7 < 10$, this partial solution won't lead to a solution, so we prune.

Suppose we choose 3, and get partial solution $(0, 0, 1, 1)$.

Step 5: Choose 1 or not?

Suppose we don't and get partial solution $(0, 0, 1, 1, 0)$. Since $6 + 3 = 9 < 10$, this partial solution won't lead to a solution, so we prune.

Suppose we choose 1, and get solution $(0, 0, 1, 1, 1)$.

We have no more choices in steps 4, 5 and 6.

Step 2: Suppose we choose 7 and get partial solution $(0, 1)$.

Step 3: Choose 6 or not?

Suppose we don't and get partial solution $(0, 1, 0)$.

Step 4: Choose 3 or not?

Suppose we don't and get partial solution $(0, 1, 0, 0)$. Since $7 + 1 = 8 < 10$, this partial solution won't lead to a solution, so we prune.

Suppose we choose 3, and get solution $(0, 0, 1, 0, 1, 0)$.

We have no more choices to make in step 4.

Step 3: Suppose we choose 6 and get partial solution $(0, 1, 1)$. Since $7+6 = 13 > 10$, this is not a solution.

We have no more choices to make in steps 3 and 2.

Step 1: Suppose we choose 4, and get partial solution (1).

Step 2: Choose 7 or not?

Suppose we don't and get partial solution $(1, 0)$.

Step 3: Choose 6 or not?

Suppose we don't and get partial solution $(1, 0, 0)$. Since $4 + 3 + 1 = 8 < 10$, this partial solution won't lead to a solution, so we prune.

Suppose we choose 6, and get solution $(1, 0, 1, 0, 0, 0)$.

We have no more choices to make in step 3.

Step 2: Suppose we choose 7 and get partial solution $(1, 1)$. Since $4+7 = 11 > 10$, this is not a solution.

5.2 The algorithm

```
SubsetSum(S,T,k,n)
if k = n+1 then
  output S
  return
S[k] = 0
if InRange(S,T,k,n) then
  SubsetSum(S, k+1, n)
S[k] = 1
if InRange(S,T,k,n) then
  SubsetSum(S, k+1, n)
```

where

```
InRange(S,T,k,n)
S1 = Sum_{i=1}^{k} T[i]S[i]
if S1 > M then
  return false
S2 = Sum_{i=k+1}^{n} T[i]
if S1 + S2 < M then
  return false
return true
```