Unit 7.1 Backtracking

Algorithms

EE3980

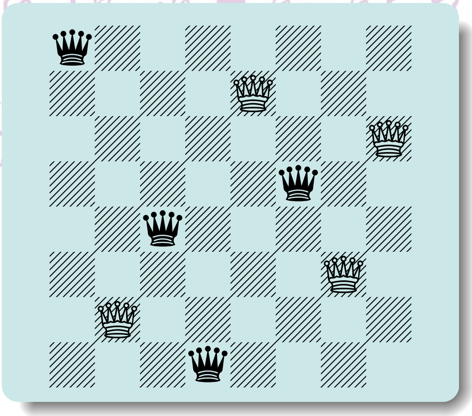
May 2, 2018

Backtracking Algorithms

- The backtracking algorithms are to deal with problems to generate a desired solution expressible as an n-tuple, $(x_1, x_2, \cdots, x_n)$, where x_i are chosen from a finite set S_i, and the solution satisfies or minimizes/maximizes a criterion function $P(x_1, x_2, \cdots, x_n)$.
- Suppose m_i is the size of S_i. Then, there are $m = m_1 \times m_2 \times \cdots \times m_n$ possible candidates for satisfying the function P.
- The brute force approach is to form all m candidates and evaluate criterion function on each of them, selects all (or the optimal) solutions.
- The backtracking method form the n-tuple one components at a time, and then use the modified criterion function $P_i(x_1, x_2, \cdots, x_i)$ to see if the current vector can meet the overall criterion. If it cannot, the current vector is ignored immediately.
  - The number of tries is substantially smaller with backtracking methods.

The 8-Queens Problem

- A queen in a chess game can attack any other piece if
  - It is in the same row
  - It is in the same column
  - It is in the same diagonal (two directions)
- The 8-queens puzzle is to place 8 queens on a chessboard such that they don't attack each other.



The 8-Queens Problem — Algorithms

Algorithm 7.1.1. N-Queen puzzle

```
Algorithm Place(k, i)
// Test if it is legitimate to place Q[k] = i.
{
    for j := 1 to k − 1 do { // check already placed queens.
        if (Q[j] = i or |Q[j] − i| = |j − k|) then return false ;
    }
    return true ;
}
Algorithm NQueens(k, n)
// Place k'th queen onward, if successful print out solution.
{
    for i := 1 to n do { // all possible positions for Q[k]
        if Place(k, i) then { // placing Q[k] = i is legitimate
            Q[k] := i;
            if (k = n) then write (Q[1 : n]); // a solution found.
            else NQueens(k + 1, n); // place Q[k + 1]
        }
    }
}
```

The 8-Queens Problem — Algorithms, II

- In the NQueens algorithm, array $Q[i]$, $1 \le i \le 8$, stores the column position of the queen in row i.
  - Since each row can have only one queen, using array Q reduces the problem to a 1-D problem.
- The algorithm places a queen at all possible columns and tests if the column is legitimate
  - If so continue to the next row
  - If not, try the next column
- At termination, all possible solution will be printed
- NQueens algorithm significantly reduces the number of tryouts by using the Place function
  - The largest search space has 8^8=16,777,216 cases
  - 8 queens not in the same row nor the same column, there are $8!$ =40,320 cases, (0.24%)
  - NQueue further reduces the test cases significant using Place function

The 8-Queens Problem — Algorithms, III

- An iterative version, NQueens_I, is given next
  - Same time complexity as the recursive version
  - The number of try-outs is identical in either case
  - Smaller heap space for function calls for iterative version
- Both NQueens and NQueens_I algorithms can still be improved.
- As written, both algorithms can handle n-queen problems.

The 8-Queens Problem — Iterative Algorithms

Algorithm 7.1.2. N-Queen puzzle, iterative solution

```
Algorithm NQueens_I(n)
// Find all solutions for N-Queen problem iteratively.
{
      k := 1; Q[k] := 0;
      while (k > 0) do {
            Q[k] := Q[k] + 1;
            while (Q[k] ≤ n) do {
                  if Place(k, Q[k]) then {
                        if (k = n) then write (Q[1 : n]); // a solution is found
                        else {
                              k := k + 1; Q[k] := 0; // try next row and initialize
                        }
                  }
                  Q[k] := Q[k] + 1;
            }
            k := k − 1; // done with this row, backtrack to previous row
      }
}
```


Sum of Subsets Problem

- Given a set of n distinct positive numbers, $\{w_i, 1 \le i \le n\}$ and m, $m > 0$, the sum of subsets problem is to find all the combinations of those n numbers whose sum is m.
- Example, given the set $\{4, 11, 15, 24\}$ and $m = 15$.
  - Two subsets, $\{4, 11\}$ and $\{15\}$, have the sum equals to 15.
- It is assume that the set is ordered in nondecreasing order, $w_i \le w_{i+1}, 1 \le i < n$, and

$$w_1 \le m, \tag{7.1.1}$$

$$\sum_{i=1}^{n} w_i \ge m. \tag{7.1.2}$$

  otherwise, there is not solution possible.
- Let $\{x_i | 1 \le i \le n\}$ be the solution, $x_i = 0$ or $x_i = 1$, then

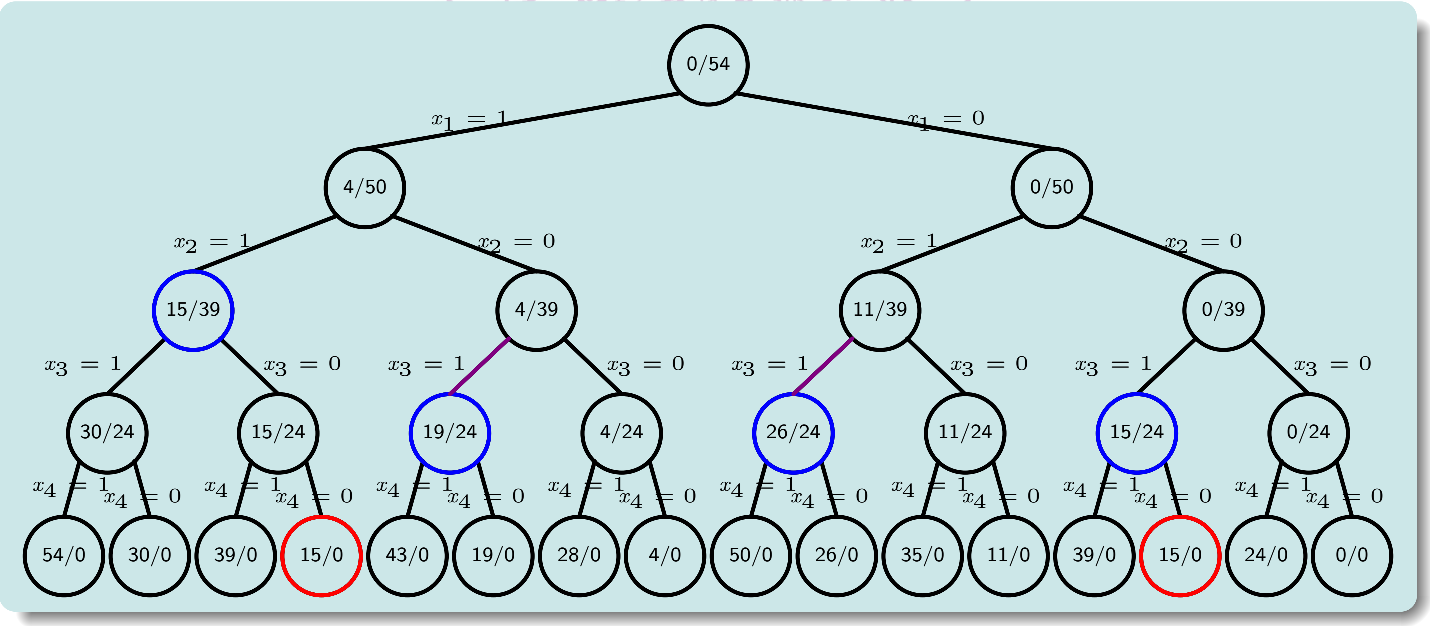
$$\sum_{i=1}^{n} x_i w_i = m. \tag{7.1.3}$$

- To find the solution, all combinations are to be tested.
  - Backtracking approach can be applied.

Sum of Subsets, Example

- Example, $w = \{4, 11, 15, 24\}$, $m = 15$
- Two numbers shown in each node s/r

$$s = \sum_{i=1}^{k} x_k w_k \qquad r = \sum_{i=k+1}^{n} w_k$$



Sum of Subsets, Algorithm

- A recursive algorithm to find all the solutions.

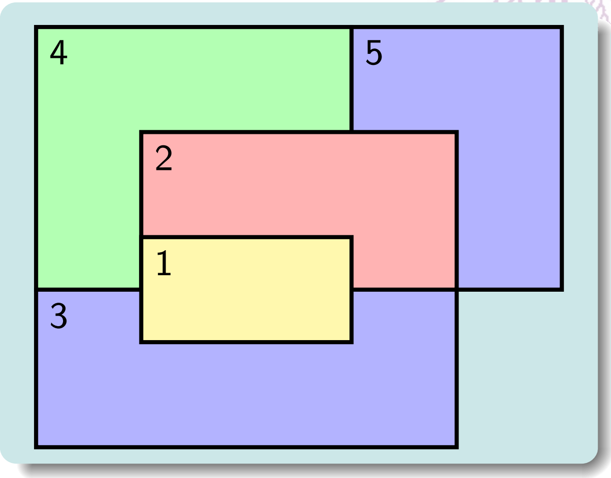
Algorithm 7.1.3. Sum of Subsets

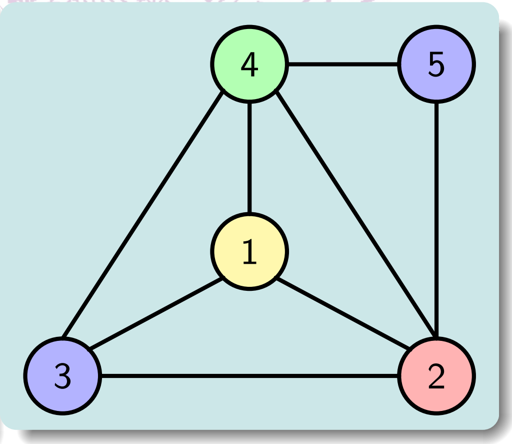
```
1  Algorithm SumOfSub(s, k, r)
2  // To test x[k], s = Σ_{i=1}^{k-1} w[i]x[i] and r = Σ_{i=k}^{n} w[i].
3  {
4        x[k] := 1; // try to include w[i]
5        if (s + w[k] = m) then write (x[1 : k]); // one solution found
6        else if (s + w[k] + w[k + 1] ≤ m) then
7              SumOfSub(s + w[k], k + 1, r − w[k]);
8        if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then { // x[i] = 0 case
9              x[k] := 0;
10             SumOfSub(s, k + 1, r − w[k]);
11       }
12 }
```

- Note that the termination condition of this algorithm, checking k against n, should be included
- With proper checking, lines 6 and 8, number of unsuccessful search is significantly reduced.

Graph Coloring

- Given a map with n regions, the m-colorability decision problem is to find if one can assign m colors to the map such that each region has a color and no two adjacent regions have the same color.
- Note that the map with n regions can be transformed into a graph.
  - Each region is represented by a node,
  - Adjacent regions are connected by an edge between the nodes.
- The adjacency relationship can also be represented by the adjacency matrix.





$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Graph Coloring, Algorithm

- The following algorithm solves for the m-coloring problem for a graph, G, with n vertices and adjacency matrix A.
  - Global Array x is the solution found, $x[i]$ is the color for vertex i.
- The algorithm should be invoked by
  mColoring$(n, m, 1)$;

Algorithm 7.1.4. m-Color Algorithm

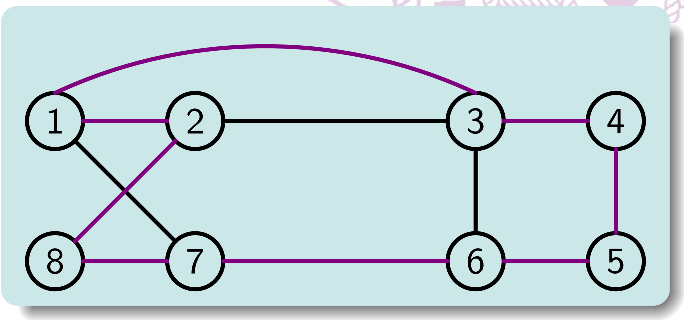
```
Algorithm mColoring(n, m, k)
// Recursively assign all possible, at most m, colors to node k.
{
    for x[k] := 1 to m do {
        i := 1; // check for colored and adjacent nodes with the same color
        while (i < k and ((A[i, k] = 0) or (x[i] ≠ x[k]))) do i := i + 1;
        if (i = k) then { // color acceptable
            if (k == n) then write (x[1 : n]); // a solution is found
            else mColoring(n, m, k + 1);
        }
    }
}
```

Graph Coloring, Complexity

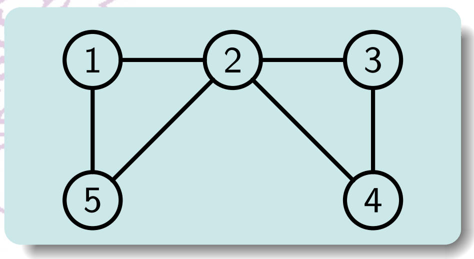
- In algorithm mColoring, Algorithm (7.1.4), the for loop, lines 4-11, is executed m times at each recursive call
  - And mColoring is called recursively for n times
- Again in algorithm mColoring, the while loop, line 6 is executed at most n times
- Thus the total time complexity is $\mathcal{O}(nm^n)$
- Note that given a graph G with degree d, then G can be colored using $d+1$ colors.
- The smallest m that can color a graph G is also called the chromatic number of G.
- Note that $m \leq d+1$ and m can be found by using the Algorithm mColoring using different m.

Hamiltonian Cycles

- Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle is a closed path along n edges of G that visits every vertex once and returning to its starting position.
  - If a Hamiltonian cycle begin at a vertex $v_i \in V$ and the vertices are visited in the order $(v_1, v_2, \cdots, v_{n+1})$, then the edge $(v_i, v_{i+1}) \in E$, $1 \leq i \leq n$, and the v_i are distinct except $v_1 = v_{n+1}$.



G_1 with Hamiltonian cycles.



G_2 No Hamiltonian cycle.

Hamiltonian Cycles — Algorithm

Algorithm 7.1.5. Hamiltonian Cycle

```
1  Algorithm Hamiltonian(n, k)
2  // Recursive algorithm to find the next vertex of a Hamiltonian cycle.
3  {
4        for x[k] := 1 to n do { // all possible vertices
5              if (E[x[k − 1], x[k]] = 1) then { // only connect to x[k − 1]
6                    i := 1;
7                    while ((i < k) and (x[i] ≠ x[k])) i := i + 1; // check if x[k] distinct
8                    if (i = k) then // x[k] has not been used
9                          if (k < n) Hamiltonian(n, k + 1); // move to the next vertex
10                         else {
11                               if (E[k, 1] = 1) then write (x[1 : n]); // print solution
12                         }
13             }
14       }
15 }
```


Hamiltonian Cycles — Algorithm, II

- Backtracking approach to solve the Hamiltonian cycle problem.
- $x[1:n]$ is the solution vector.
- $E[1:n, 1:n]$ is the adjacency matrix
  - $E[i,j] = 1$ if $(i,j) \in E$ is an edge in G
  - Otherwise, $E[i,j] = 0$.
- `Hamiltonian` should be invoked by

  `Hamiltonian`$(n, 2)$;

  with $x[1] = 1$.
- Thus, this algorithm always find the Hamiltonian cycle starting from vertex 1.
- Note that the depth of the recursive call is n
  - The maximum number of `Hamiltonian` recursive call at level k is $n - k$ since each vertex on the path must be distinct
  - Thus, the number of function call is bounded above by $(n-1)!$
- The `while` loop of line 7 is executed at most n times
- The worst case time complexity of `Hamiltonian` algorithm is $\mathcal{O}(n!)$
  - Due to the sparsity of the adjacency matrix, this algorithm has much lower complexity in practice.

0/1 Knapsack Problem Revisited

- Given n objects, each with profit p_i and weight w_i, $1 \le i \le n$, to be placed into a sack that can hold maximum of m weight. However, there is an additional constraint that each object must be placed as a whole into the sack, or not at all. That is, find x_i, $1 \le i \le n$, such that

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i, \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \le m, \\ \text{and} \quad & x_i = 0 \text{ or } 1, \qquad 1 \le i \le n. \end{aligned} \tag{7.1.4}$$

- Note that $x_i = 0$ or 1 and the solution space can be expanded as a tree.
- The solution can be found by traversing the tree.
- In the following, we assume the objects are ordered as

$$\frac{p_1}{w_1} \ge \frac{p_2}{w_2} \ge \cdots \ge \frac{p_n}{w_n}. \tag{7.1.5}$$

And, fp is the final profit, fw is the final weight. Both are global variables.

0/1 Knapsack Problem — Algorithm

Algorithm 7.1.6. 0/1 Knapsack

```
Algorithm BKnap(k, cp, cw)
// Find solution of 0/1 knapsack problem. cp/cw/cx: current profit/weight/sol.
{
      if (cw + w[k] ≤ m) then {
            cx[k] := 1;
            if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
            else if ((cp + p[k] > fp) and (k = n)) then { // record solution
                  fp := cp + p[k]; fw := cw + w[k];
                  for i := 1 to n do x[i] := cx[i];
            }
      }
      if ( Bound(cp, cw, k) ≥ fp) {
            cx[k] := 0;
            if (k < n) then BKnap(k + 1, cp, cw);
            else if ((cp > fp) and (k = n)) then {
                  fp := cp; fw := cw;
                  for i := 1 to n do x[i] := cx[i];
            }
      }
}
```

0/1 Knapsack Problem — Bound Algorithm

- Due to Eq. (7.1.5), Bound function can estimate the maximum profit quickly.

Algorithm 7.1.7. Bounding function

```
Algorithm Bound(cp, cw, k)
// Estimate maximum profit for k + 1 to n objects.
{
    mp := cp; mw := cw;
    for i := k + 1 to n do {
        mw := mw + w[i];
        if (mw < m) then mp := mp + p[i];
        else return mp + (1 − (mw − m)/w[i]) ∗ p[i];
    }
    return mp;
}
```

- Note that Bound function returns a floating number instead of an integer.

0/1 Knapsack Problem — Example

- Given 3 objects, $(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$, and $m = 6$. Find the optimal 0/1 knapsack solution, (x_1, x_2, x_3), $x_i = 0$ or $x_i = 1$, $1 \le i \le 3$, that maximizes the profit.
- The calling sequence of BKnap algorithm

```
BKnap(k = 1, cp = 0, cw = 0)
    test y[1] = 1, cw + w[1] ≤ m
    BKnap(k = 2, cp = 1, cw = 2)
        test y[2] = 1, cw + w[2] ≤ m
        BKnap(k = 3, cp = 3, cw = 5)
            test y[3] = 1, cw + w[3] > m, terminates
            test y[3] = 0, Bound= 3, feasible solution: fp = 3, x = (1, 1, 0)
        test y[2] = 0, Bound= 6
        BKnap(k = 3, cp = 1, cw = 2)
            test y[3] = 1, cw + w[3] ≤ m, feasible solution: fp = 6, x = (1, 0, 1)
            test y[3] = 0, Bound= 1, terminates
    test y[1] = 0, Bound= 5.75, terminates
```

- Function Bound helps to reduce the number of evaluations

Summary

- Backtracking algorithm
- 8-queens problem
- Sum of subsets problem
- Graph coloring problem
- Hamiltonian cycles
- 0/1 knapsack problem