Backtracking and Branch and Bound

Introduction

- Solutions to many combinatorial optimization problems include exhaustive search
  - Optimal solution desired at cost of speed
  - Exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property
- Backtracking can be used
  - To reduce the cost of search
  - To list all possible solutions for a combinatorial problem

Backtracking: Overview

- Systematic/intelligent way to iterate through all the possible configurations of a search space
  - Configurations may represent
    - all possible arrangements of objects (permutations)
    - all possible ways of building a collection of them (subsets)
  - Configurations must be generated only once, and potential configurations must not be missed
- Model combinatorial search solution as a vector a = (a_1, a_2, ..., a_k)
  - Vector might represent an arrangement where a_i contains the ith element of the permutation
  - Or represent a given subset *S*, where a_i is true if and only if the *i*th element of the universe is in *S*.

Backtracking: Overview

- Strategy
  - At each step during backtracking
    - try to extend a given partial solution $a = (a_1, a_2, \ldots, a_k)$ by adding another element at the end
    - Test if the extending lead to a solution or not
    - If solution not found explore if proceeding further will lead to a solution or if we have to go back to a previous partial solution
- Constructs a tree of partial solutions
  - Each node represents a partial solution
  - Edge indicates an advancement of a solution

Search Space Tree

- A rooted tree where each level represents a choice in the solution space that depends on
  - the level above and
  - any possible solution is represented by some path starting out at the root and ending at a leaf
- Root represents state where no partial solution has been made
- A leaf represents the state where all choices making up a solution have been made

Backtracking Overview

- constructs a tree of partial solutions, where each vertex represents a partial solution
  - This tree also called a “state-space tree”
  - A node in a state-space tree is ***promising*** if it corresponds to a partial solution that may still lead to a complete solution;
    - its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child
  - Otherwise, it is called ***nonpromising***
    - Leave represent nonpromising solutions or dead-ends
    - algorithm backtracks to the node's parent to consider the next possible option for its last component

Backtracking Overview

- Corresponds to doing a DFS of the state-space tree
- Backtrack-DFS(A, k)

```
if A = (a1, a2, ..., ak) is a solution, report it.
else
    k = k + 1
    compute Sk
    while Sk = ∅ do
        ak = an element in Sk
        Sk = Sk − ak // Sk is a finite set where ak belongs to
        Backtrack-DFS(A, k)
```

Backtracking - Procedure

```
backtrack(int a[], int k, data input) {
    if (is_a_solution(a, k, input) process_solution(a, k, input)
    else {
        k=k+1;
        construct_candidates(a,k,input,c,ncandidates);
        for (i=0; i<ncandidates; i++) {
            a[k] = c[i];
            make_move(a,k,input);
            backtrack(a,k,input);
            unmake_move(a,k,input);
            if (finished) return; /* terminate early *
        }
    }
}
```

Backtracking: Procedure Details

- is a solution(a,k,input):
  - tests whether the first k elements of vector a from a complete solution for the given problem
- construct candidates(a,k,input,c,ncandidates):
  - fills an array c with the complete set of possible candidates for kth position of a, given contents of first $k - 1$ positions
- process solution(a,k,input):
- make move(a,k,input) and unmake move(a,k,input)
  - Modify data structure in response to latest move

Problem 1: Constructing Subsets

- How many subsets are there of an n-element set, say the integers {1, . . . , n}?
  - there are 2^n subsets of n elements
- Solution
  - set up an array/vector of n cells that represents a subset
  - The value of ai is true or false and signifies whether the i^{th} item is in the given subset.
  - The termination happens when k=n

Solution

- void subsets(int k, boolean[] a) {
  - if (k == N) {

      Do something ... maybe print it.

      Return; }

    // A[k] is not in the subset.

    a[k] = false;

    subsets(k + 1, a);

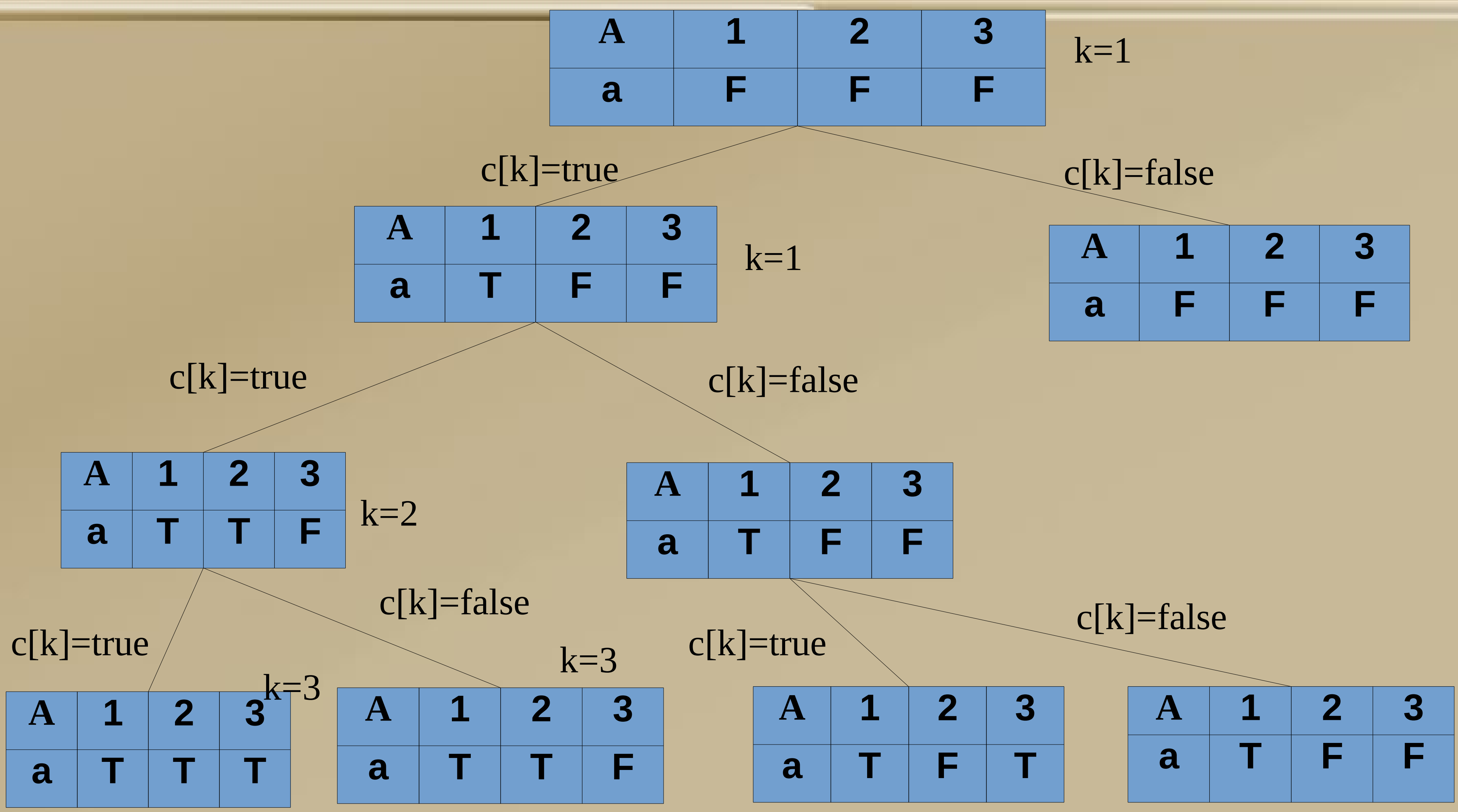
    // A[k] is in the subset.

    a[k] = true;

    subsets(k + 1, a);

  }

Example


A 1 2 3
a F F F
k=1
c[k]=true
c[k]=false
A 1 2 3
a T F F
k=1
A 1 2 3
a F F F
c[k]=true
c[k]=false
A 1 2 3
a T T F
k=2
A 1 2 3
a T F F
c[k]=true
c[k]=false
k=3
A 1 2 3
a T T T
k=3
A 1 2 3
a T T F
c[k]=true
c[k]=false
A 1 2 3
a T F T
A 1 2 3
a T F F

N-Queens Problem

- n-Queens Problem Place n queens on an n×n chessboard so that no two queens attack each other
  - Two queens cannot be in the same column, row, or diagonal
- Solution trivial for n=1
  - No solution for n=2 or 3
- 4 Queens Problem
  - Each queen to be placed in its own column

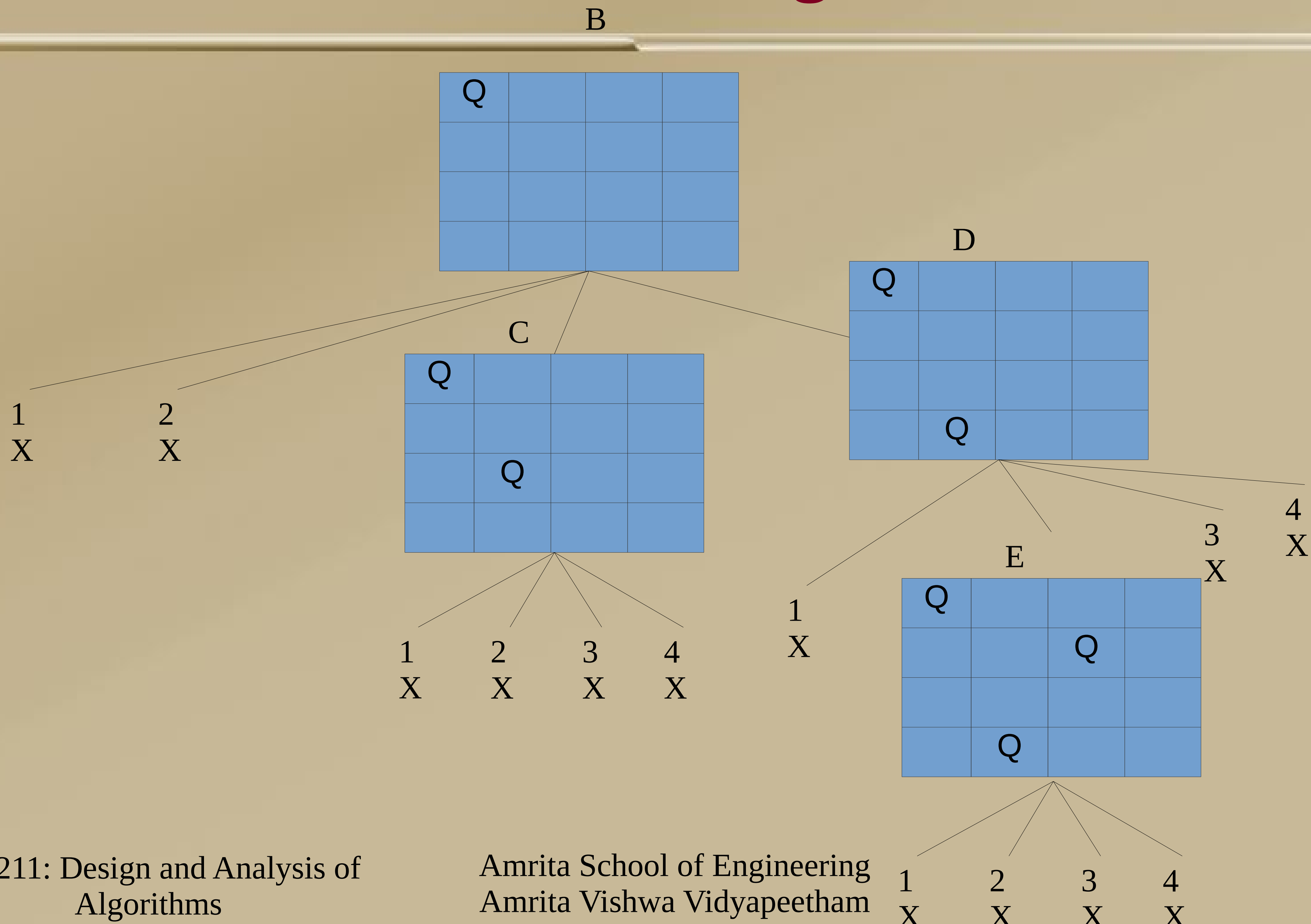
A

	Q-1	Q-2	Q-3	Q-4
1				
2				
3				
4				

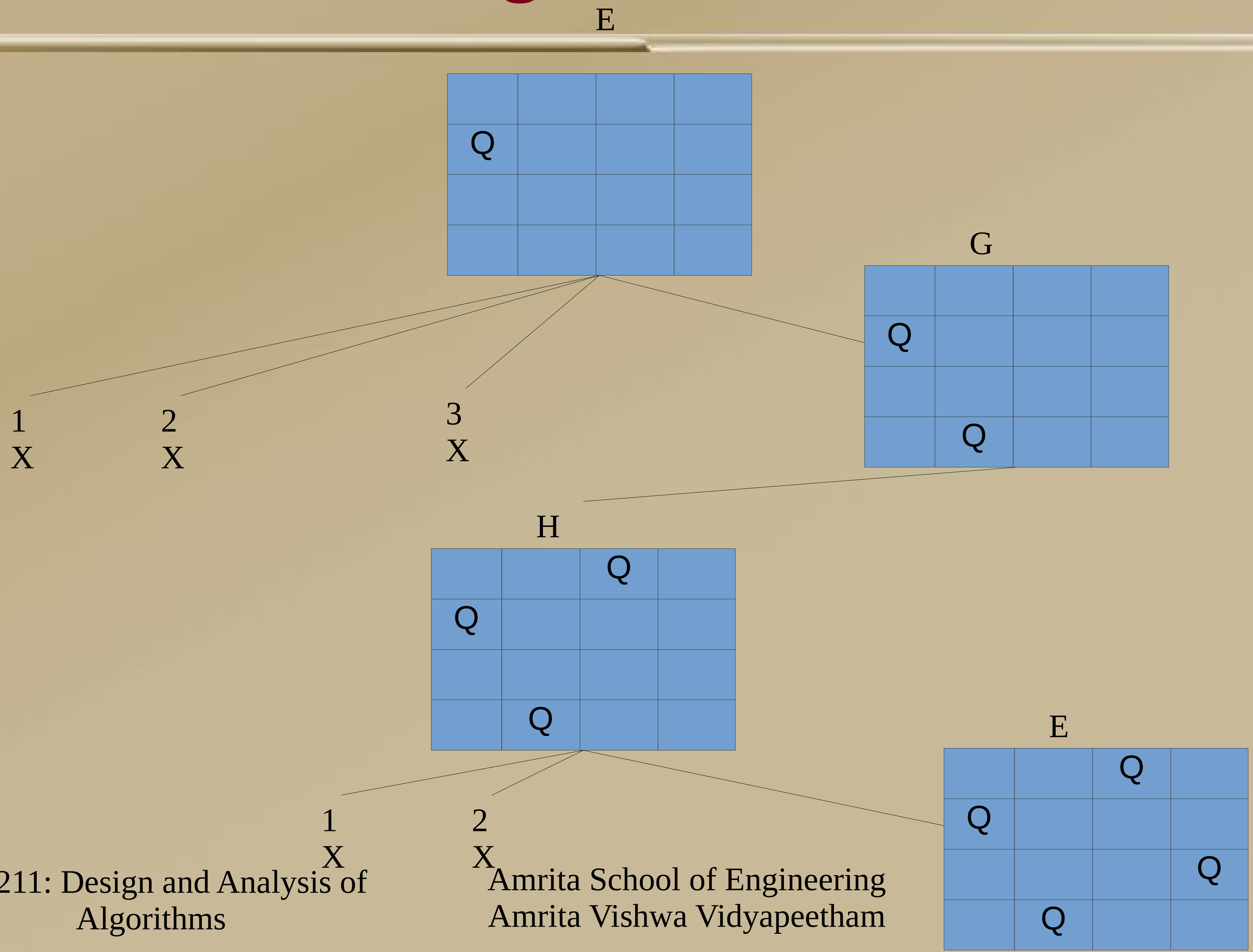
4-Queens- Backtracking Solution

- Start with queen1 and place in first possible position – [1,1], place queen2, in rows 1 and 2 of the second column
  - Not acceptable
  - Acceptable solution is row 3 and column 2
- State space tree
  - Each node is a configuration for the column and possible row
  - X denotes an unacceptable configuration

4-Queens: Backtracking: Dead-end



Backtracking: Possible Solution



Pseudocode

- tryConfig(i):

```
for j = 1 to n:
    if safe then:
        select jth candidate;
        set queen
        if i < n then:
            tryConfig(i+1);
        else
            record solution
        remove queen
```

Src: http://www.brian-borowski.com/software/nqueens/

Performance

- Exhaustive Search
  - Number of placements = 16! / 4!(16 − 4)!
  - = (16 · 15 · 14 · 13)/(4 · 3 · 2) = 1820.
- Backtracking
  - Consider the number of combinations of n objects taken at k at a time, consider only queens placed at different columns – solution candidates = 4^4 = 256
  - Queens must be at different rows
    - Solution candidates = 4! = 24
    - For 8-queens problem solution candidates = 40,320

Branch and Bound

- In an optimization problem
  - *Feasible solution* is a point in the problem's search space that satisfies all the problem's constraints
  - *Optimal solution* is a feasible solution with best value to objective function
- Backtracking stops when solution is infeasible
  - This idea can be strengthened

Branch and Bound

- Two aspects required in this approach
  - a way to provide, for every node of a state-space tree, a bound on the best value of the objective function[1] on any solution that can be obtained
  - The value of best solution seen so far
- Principle Idea
  - If node's bound value is not better than the best seen so far node is non promising, hence pruned.
  - no solution obtained from the node can yield a better solution than the one already available.

Search Space Pruning

- A search along a path is terminated if
  - The value of the node's bound is not better than the value of the best solution seen so far
  - Constraints of solution already violated, hence node represents no feasible solution
  - The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)

0-1 Knapsack Problem

- Construct a search space tree
  - if there are N possible items to choose from, then the kth level represents state where it has been decided which of the first k items have or have not been included in the knapsack.
    - The path shows the choices made for the first k items ie the selection from first k items
  - branch going to the left indicates the inclusion of the next item while a branch to the right indicates its exclusion

O-1 Knapsack

- At each node record
  - total weight w of the selection
  - the total value v of this selection
  - Upper bound b
    - $b = v + (W – w) (v_{i+1} /w_{i+1})$
    - v – total value of items already selected
    - W-w – remaining capacity of knapsack
    - v_{i+1} /w_{i+1}- best per unit payoff among the remaining items

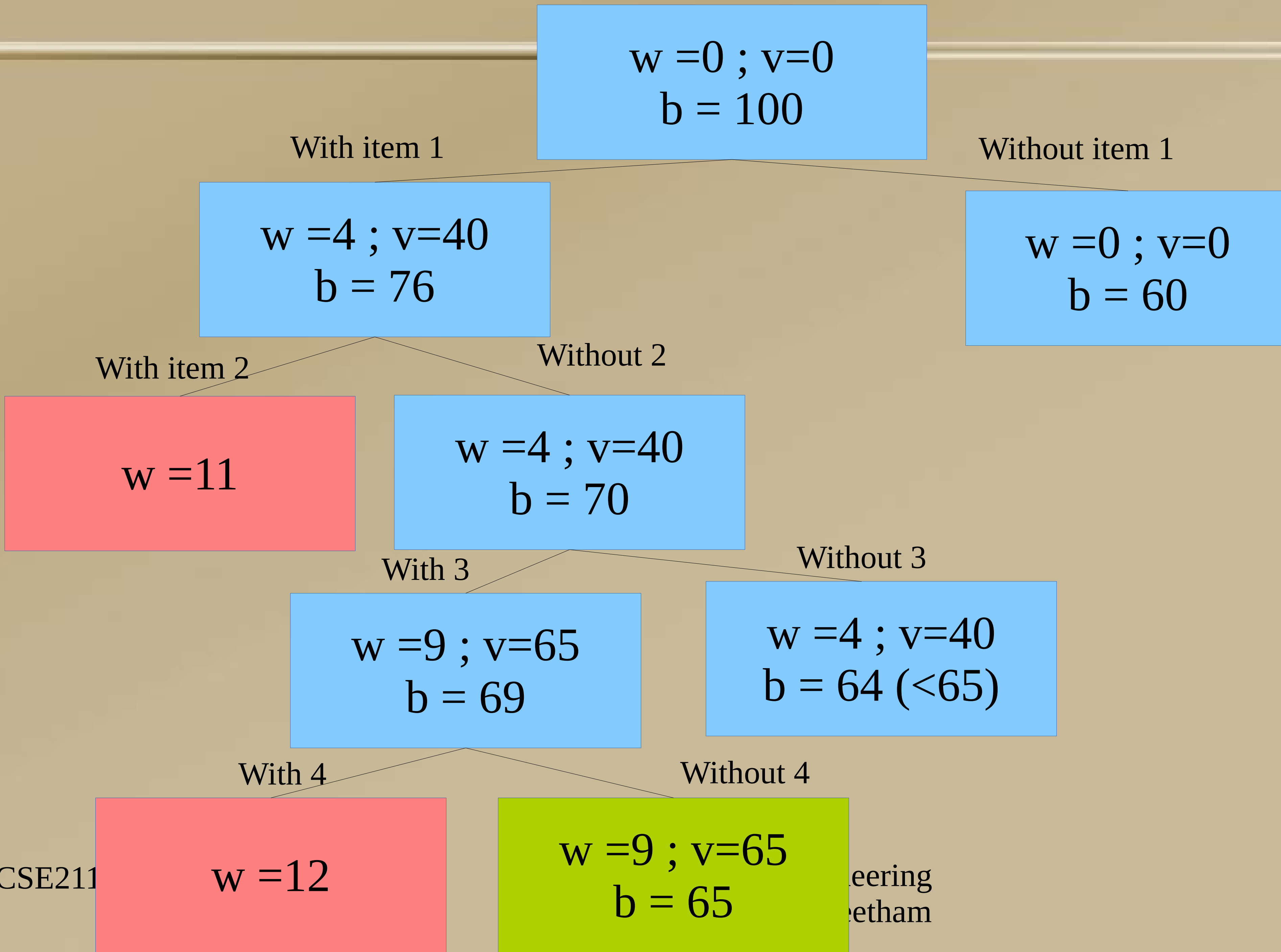
Example

Item	Weight	Value
1	4	40
2	7	42
3	5	25
4	3	12

- Capacity of Knapsack – 10

w =0 ; v=0
b = 100

Solution


w =0 ; v=0
b = 100
With item 1
Without item 1
w =4 ; v=40
b = 76
w =0 ; v=0
b = 60
With item 2
Without 2
w =11
w =4 ; v=40
b = 70
With 3
Without 3
w =9 ; v=65
b = 69
w =4 ; v=40
b = 64 (<65)
With 4
Without 4
w =12
w =9 ; v=65
b = 65

References

- Steven Skiena, “The Algorithm Design Manual”, Springer, 2008
- Anany Levitin, “Design and Analysis of Algorithms”, 2nd Edition, 2006, Addison Wesley
- http://www.seas.gwu.edu/~ayoussef/cs212/branchandbound.html
- http://ocw.mit.edu/courses/sloan-school-of-management/15-053-optimization-methods-in-management-science-spring-2013/tutorials/MIT15_053S13_tut10.pdf