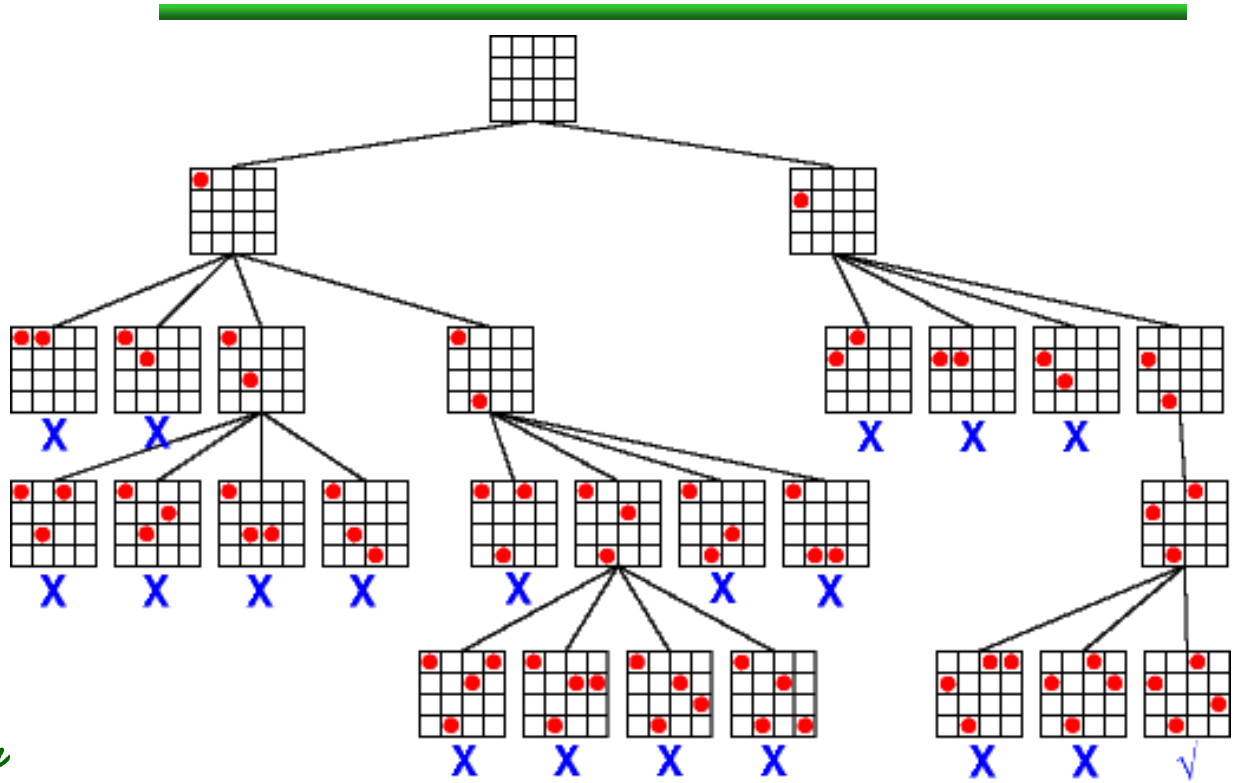
Backtracking Algorithm Design

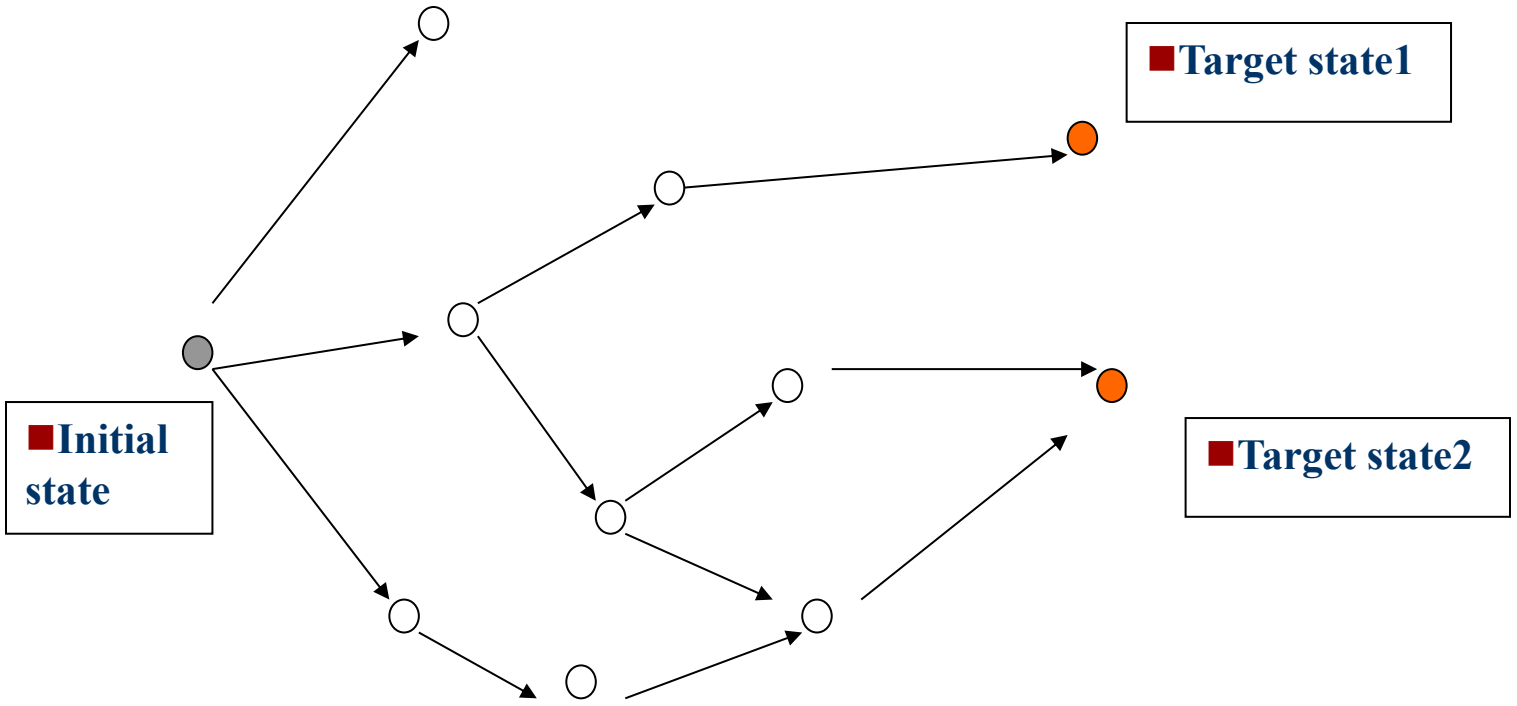


Prof. Bibhudatta

Department of CSE, NIT Rourkela,

How to solve a problem by searching

Search in a space of possible states.



Generate and Test

- **Take a state**
- Generate next states by applying relevant rules

  (not all rules are applicable for a given state!)
- **Test to see if the goal state is reached**.

  If yes - stop

  If no - add the generated states into a **stack/queue** (do not add if the state has already been generated) and repeat the procedure.

- **To get the solution**: Record the paths

Problem vs. Solution Space

- For a systems developer, the real world may be associated with the problem space, the system implementation as the solution space.
- The human's way of characterizing the problem or decision space can be called the **problem space**
- The **solution space** is the range of potential solutions that might be recommended. For a knowledge modeler coming from this context, therefore, problem and solution space may both be part of the "real world."
- In optimization a **candidate solution** is a member of a set of possible solutions to a given problem. A candidate solution does not have to be a likely or reasonable solution to the problem.

- The space of all candidate solutions is called the **feasible region**, **feasible set**, **search space**, or **solution space**.

Solving NP-complete problems

- At present, all known algorithms for NP-complete problems require time that is super polynomial in the input size, and it is unknown whether there are any faster algorithms.
- The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:
- **Approximation**: Instead of searching for an optimal solution, search for an "almost" optimal one.
- **Randomization:** Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability.
- **Restriction**: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- **Parameterization**: Often there are fast algorithms if certain parameters of the input are fixed.
- **Heuristic**: An algorithm that works "reasonably well" on many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

Tackling Difficult Combinatorial Problems

- There are two principal approaches to tackling difficult combinatorial problems (NP-hard problems):
- Use a strategy that guarantees **solving the problem exactly** but doesn't guarantee to find a solution in polynomial time
- Use an **approximation algorithm** that can find an approximate (sub-optimal) solution in polynomial time

Exact Solution Strategies

- **Exhaustive search (brute force)**
  - useful only for small instances
- **Dynamic programming**
  - applicable to some problems (e.g., the knapsack problem)
- **Backtracking**
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- **Branch-and-bound**
  - further refines the backtracking idea for optimization problems

Backtracking, Branch-and-Bound, Heuristic Search

- Sometimes there is a need to find a non-approximate solution to the NP-complete problem, even if finding this solution may take a long time.
- This time may still be practical, if the size of the problem is relatively small and/or we are lucky to avoid problem instances that elicit the worst-case running time.
- Two algorithm design techniques for dealing with NP-completeness that have shown promise in practice are: **BACKTRACKING** and **BRANCH-AND-BOUND**.

- **Backtracking** is a general algorithm design technique, which involves searching through a large set of possibilities in a systematic way.
- **Branch-and-bound** algorithm design technique is an extension of **backtracking** for optimization problems.

Backtracking

- **Backtracking** is a type of algorithm that is a refinement of brute force search In backtracking, multiple solutions can be eliminated without being explicitly examined, by using specific properties of the problem. It can be a strategy for finding solutions to constraint satisfaction problems.
- American Mathematician D. H. Lehmer coined the term "BACKTRACK" in 1950s.

- Backtracking A method of solving problems automatically by a systematic search of the possible solutions; the invalid solutions are eliminated and are not retried.

Explaining Backtracking

- Constraint satisfaction problems are problems with a complete solution, where the order of elements does not matter.
- The problems consist of a set of variables each of which must be assigned a value, subject to the particular constraints of the problem.
- Backtracking attempts to try all the combinations in order to obtain a solution.
- Its strength is that many implementations avoid trying many partial combinations, thus speeding up the running time.
- Backtracking is closely related to combinatorial search.

Backtracking

- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for *any* remaining component need to be considered.
- In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

Solution finding with backtracking

- In backtracking a solution is expressed as n-tuple ($x_1, \ldots x_n$), where x_i are chosen from some finite set $\mathbf{S_i}$
- Suppose m_i is the size of the set $\mathbf{S_i}$,
- Let P($x_1, \ldots x_n$) be the criterion function that satisfies one or many solution expressed as n-tuple($x_1, \ldots x_n$)
- Then $m=m_1\ m_2\ m_3 \ldots m_n$ n-tuples are the possible candidates that satisfying the function **P**
- In **brute force** method, from all ***m*** possible n-tuples are evaluated using P, and save those which is optimum.
- Backtracking algorithm has ability to yield the same answer by with far fewer than ***m*** trials.

Solution finding with backtracking

- The solution finding uses the basic idea, to build up the solution vector one component at a time and to use modified criterion functions P($x_1, \ldots x_i$) (some times called bounding function) to test whether the vector being formed has any chance to become a solution(success)
- If it is realized that the partial vector $(x_1, x_2 \ldots x_i)$ can in no way lead to an optimal solution , then $m_{i+1} \ldots m_n$ possible test vectors can be ignored entirely.
- The problem solved using backtracking requires to satisfy a complex set of constraints (*explicit* and *implicit* constraints)

Explicit constraints

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{aligned} x_i \geq 0 \quad &\text{or} \quad S_i = \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 \quad &\text{or} \quad S_i = \{0,1\} \\ l_i \leq x_i \leq u_i \quad &\text{or} \quad S_i = \{a : l_i \leq a \leq u_i\} \end{aligned}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Explicit constraints

- ***Explicit constraints*** are the rules that restricts each x_i to take on values only from a given set.

Example

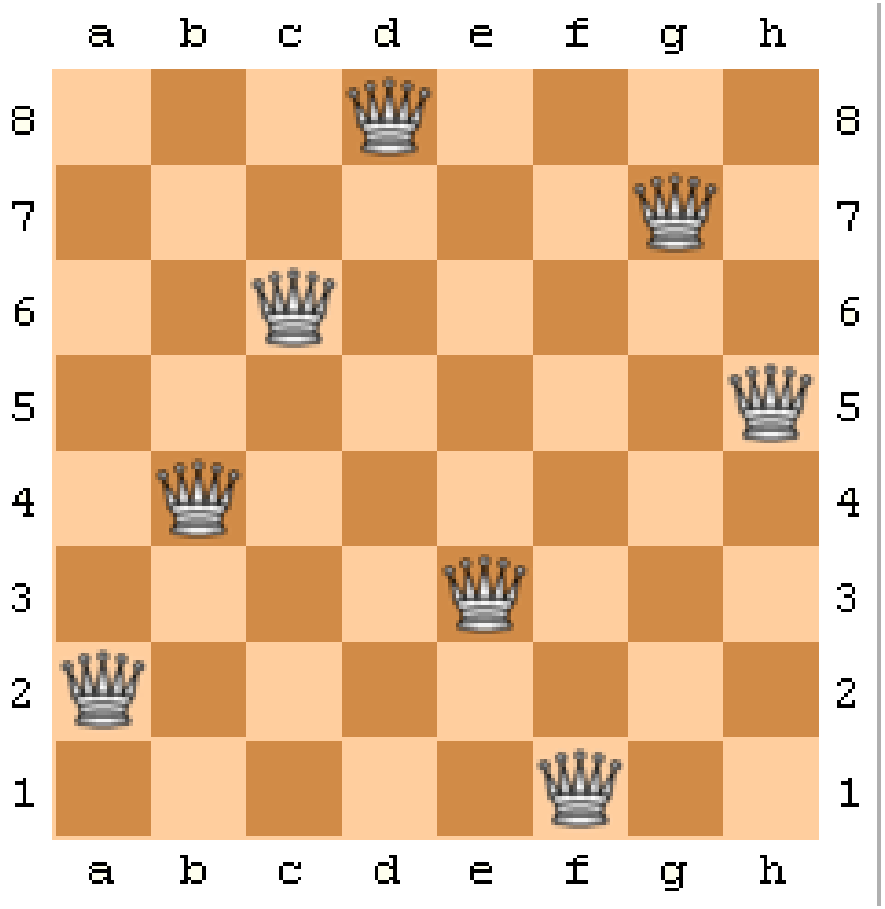
- $x_i \geq 0$ or S_i = { all nonnegative real number}
- $x_i = 0$ or 1 or $S_i = \{ 0, 1\}$
- $l_i \leq x_i \leq u_i$ or $S_i = \{a: l_i \leq a \leq u_i \}$

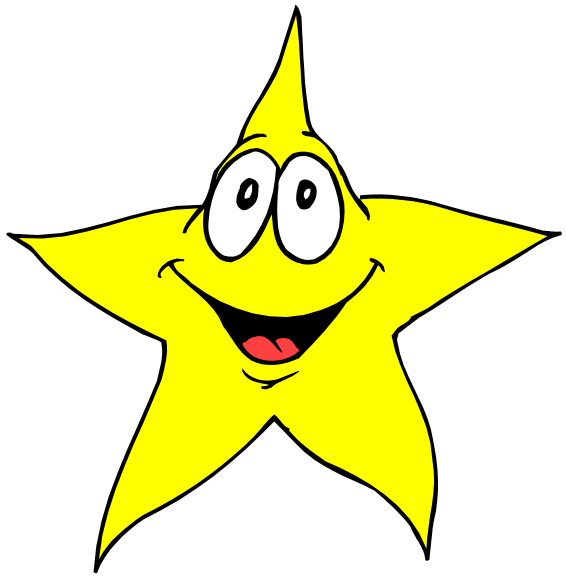
- Explicit constraints depends on the particular instance I of the problem being solved.
- All tuples that satisfy the Explicit constraints define a possible solution space for the instance I.

Implicit constraints

- The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.
- Implicit constraint describes the way in which the x_i must relate to each other.

8- queen problem





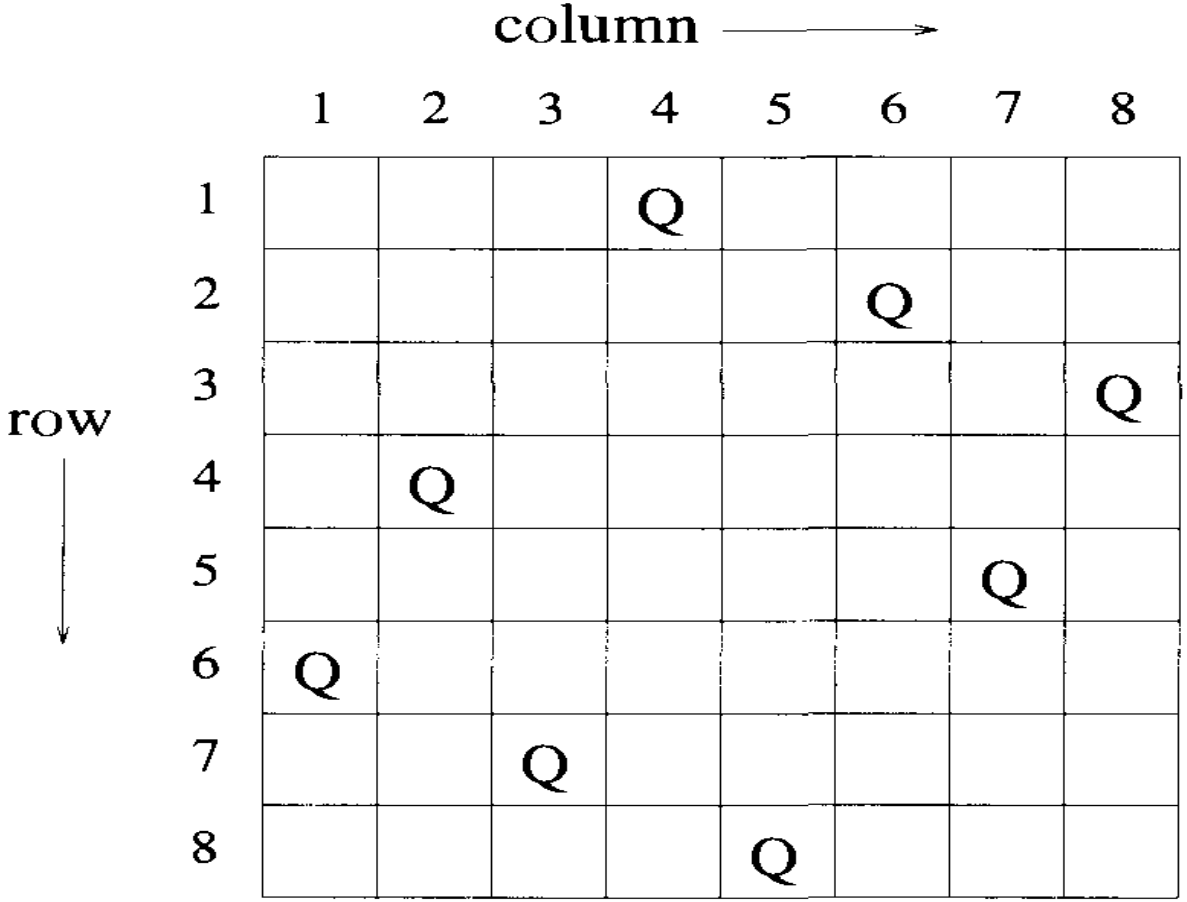
8- queen problem

- Let us number the rows and columns of the chessboard 1 through 8
- The queens can also be numbered 1 through 8.
- Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i.
- All solutions to the 8-queens problem can therefore be represented as 8-tuples(x_i,..,.x_8),where x_i is the column on which queen i is placed.

8- queen problem

- The **explicit constraints** using this formulation are Si = {1,2,3,54,6,,7, 8}, $1 \leq i \leq 8$.
- There fore the solution space consists of 8^8 8-tuples.
- The **implicit constraints** for this problem are that no two x_i's can be the same (i.e.all queens must be on different columns)and no two queens can be on the same diagonal

One solution to 8-queen Problem



(4, 6, 8, 2, 7, 1, 3, 5)

8- queen problem

- The problem of finding all solutions to the 8-queens problem can be quite computationally expensive, as there are 4,426,165,368 (i.e., $_{64}C_8$) possible arrangements of eight queens on an 8×8 board, but only **92 solutions**.

- **N Queens** Completion Is **NP Complete**. The problem of putting eight **queens** on the chess board so as no **queen** attacks another is a solved problem, as is placing **n queens** on an nxn board. However if you place some **queens** on the board and ask for a completion then the problem is **NP complete**

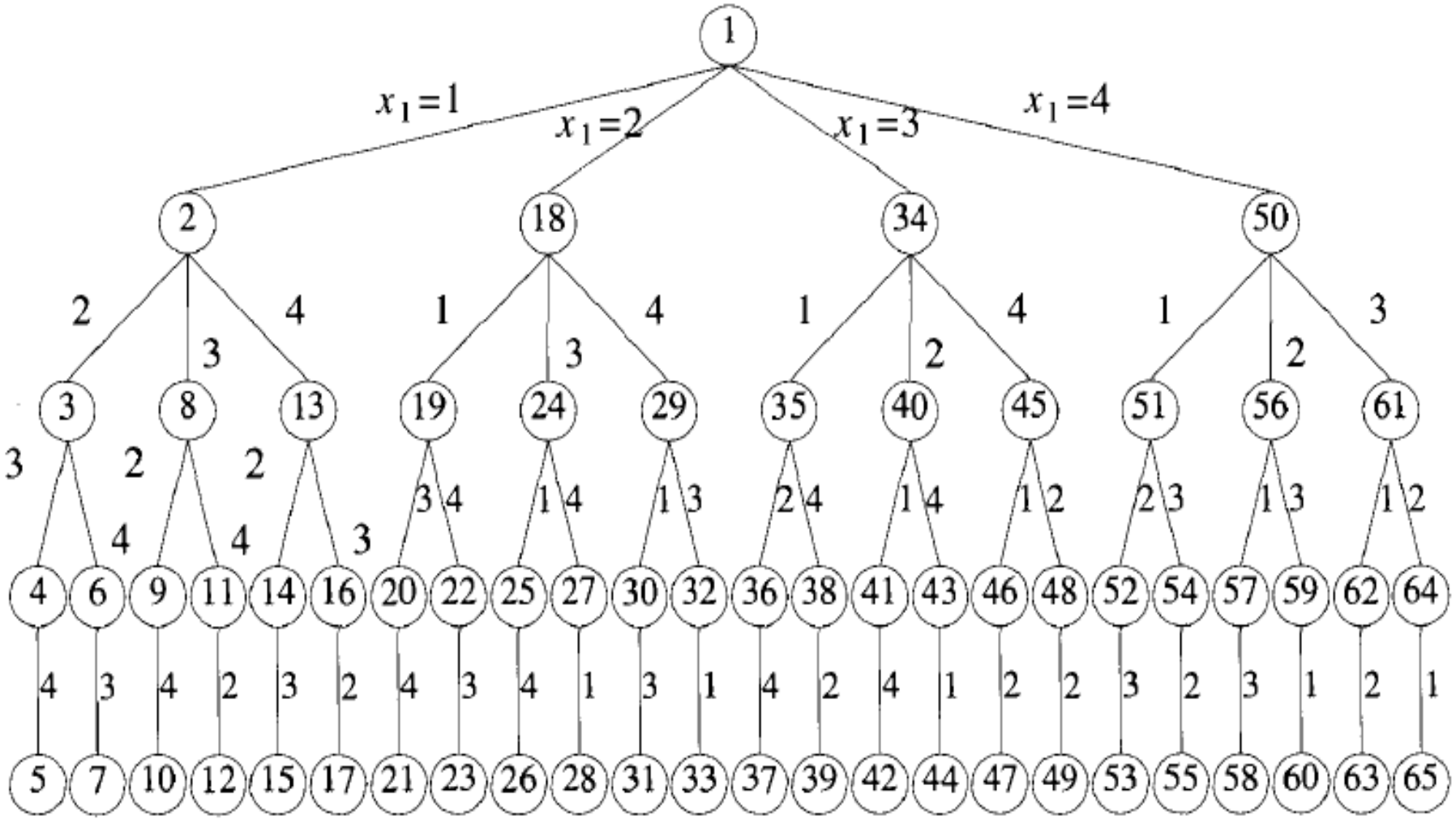
4-Queen Solution Space

- Let x_i be the position of the i^{th} queen, then the solution is (x_1, x_2, x_3, x_4,)

- **BAD:** Number the squares 1-16 so that .Then the number of possible combinations is 4^{16}
- **BETTER:** Use the row constraint so that where the x_i represents the column number of the queen in row i. There are 4^4 = 64 different 4-tuples.
- **BEST:** Add the column constraint so that no two queens are in the same column. That is $x_i \neq x_j$. The solutions are now permutations of (1,2,3,4) and there are 4! = 24 possible 4-tuples

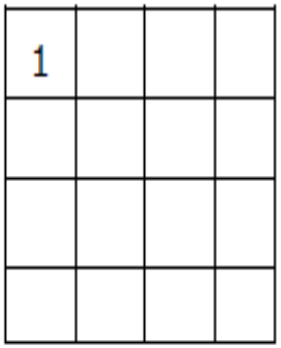
- **Explicit Constraints:** Restrictions on the values that the x_i can take. E.g. 4-Queens {1,2,3,4}
- **Implicit constraints:** Describe the ways that the x_i relate to each other. E.g. 4-Queens - no two Queens can be in the same column ($x_i \neq x_j$.) or on the same diagonal.

Tree organization of the 4-queenssolutionspace :DFS

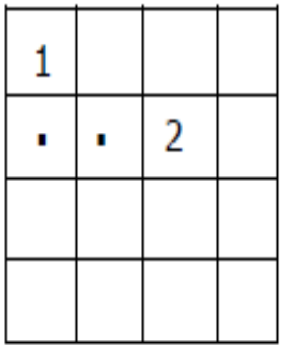


4-Queen Solution Space

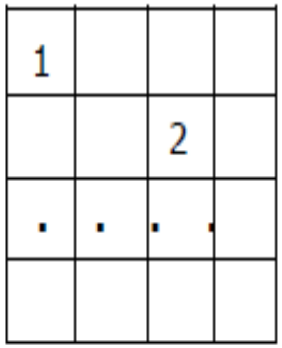
- A tree is called a permutation tree.
- The edges are labeled by possible values of x_i.
- Edges from level 1 to level 2 nodes specify the values for x_i.
- The leftmost subtree contains all solutions with $x_1 = 1$;its leftmost subtree contains all solutions with $x_1 = 1$and $x_2 = 2$,and soon.
- Edges from level i to level i +1 are labeled with the values of x_i.
- The solution space is defined by all paths from the root node to a leaf node. There are 4!= **24** leaf nodes in the tree



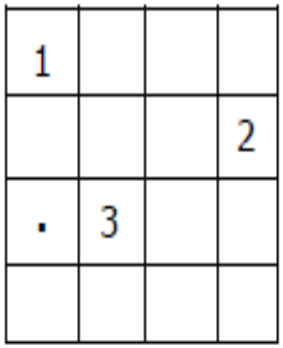
(a)



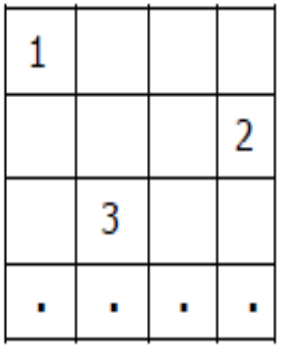
(b)



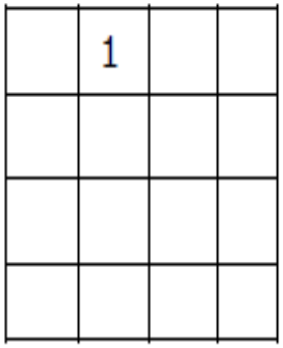
(c)



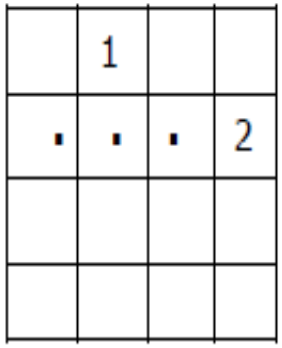
(d)



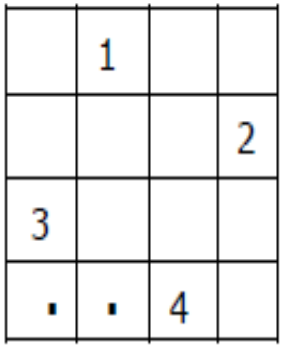
(e)



(f)

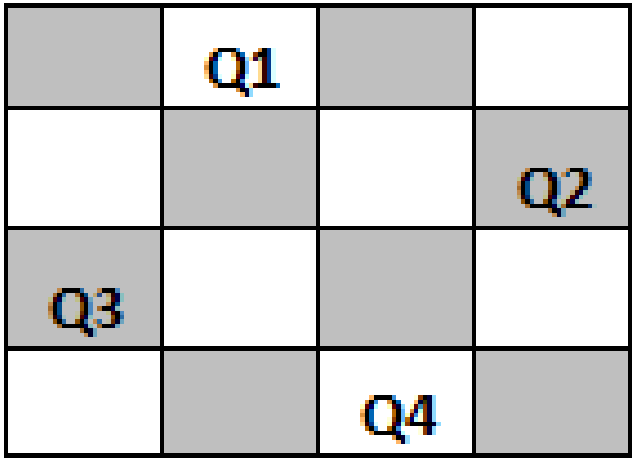


(g)

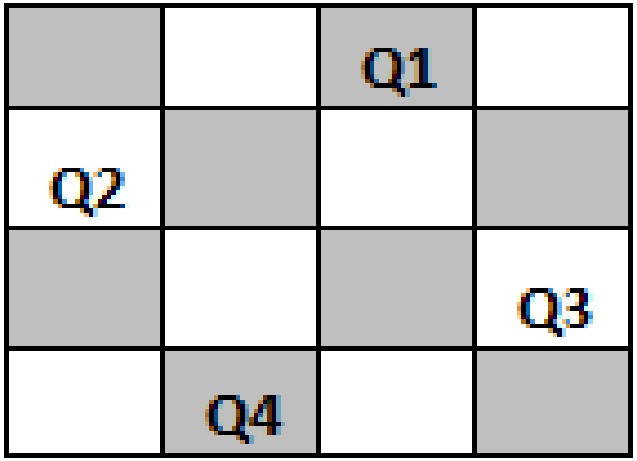


(h)

4-Queen Solution



Solution 1



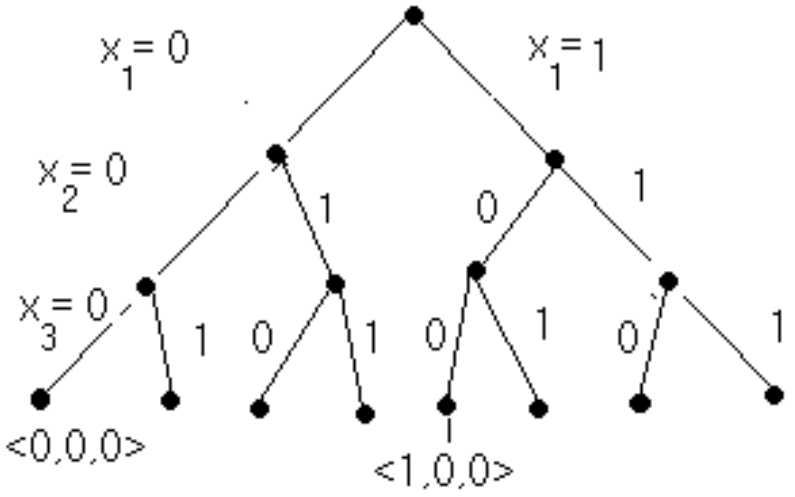
Solution 2

Implementing Backtracking

- This kind of processing is often implemented by constructing a tree of choices being made, called the ***state-space tree***.
- Its **root** represents an **initial state** before the search for a solution begins.
- The **nodes of the first level** in the tree represent the **choices made for the first component** of a solution,
- the nodes of the second level represent the choices for the second component, and so on.
- A node in a state-space tree is said to be ***promising*** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called ***non-promising***.

Implementing Backtracking

- **Leaves** represent either **non-promising dead ends** or **complete solutions** found by the algorithm.
- If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component;
- if there is no such option, it backtracks one more level up the tree, and so on.



Implementing Backtracking

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- "Prune" *nonpromising nodes*
  - DFS stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node's parent to continue the search

Steps involved in Backtracking

- Define a solution space that includes the answer to the problem instance.
- Organize this space in a manner suitable for search.
- Search the space in a depth-first manner using bounding functions to avoid moving into subspaces that cannot possibly lead to the answer.
- The Solution space is organized while the search is conducted.
- At any time during the search only the path from the start node to the current E-node is saved, so the space requirement for this is O(length of longest path from start node)
- The size of the solution space is exponential or factorial of the longest path.

Control Abstraction: Backtracking

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

- This recursive version is initially invoked by Backtrack(1)
- The solution vector(x_1,...x_n,), is treated as a global array x[1:r].
- All the possible elements for the k^{th} position of the tuple that satisfy B_k are generated one by one, and adjoined to the current vector {x_1,.,.x_{k-1}).
- Each time x_k is attached, a check is made to determine whether a solution has been found. Then the algorithm is recursively invoked.
- When the for loop of line 7 is exited, no more values for x_k exist and the current copy of Backtrack ends

Recursive Backtracking algorithm

Iterative Backtracking algorithm

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
11          {
12              if (x[1], ..., x[k] is a path to an answer node)
13                  then write (x[1 : k]);
14              k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

Iterative Backtracking algorithm

Efficiency of Backtracking algorithm

1. The time to generate the next x_k
2. The number of x_k satisfying explicitly constraints
3. The time for bounding function B_k
4. The number of x_k satisfying B_k

Backtracking Algorithm

- Based on depth-first recursive search

Approach

1. Tests whether solution has been found
2. If found solution, return it
3. Else for each choice that can be made
   a) Make that choice
   b) Recur
   c) If recursion returns a solution, return it
4. If no choices remain, return failure

- Some times called "search tree"

Summary :Backtracking

- Backtracking is a systematic way to go through all the possible configurations of a search space.
- In the general case, we assume our solution is a vector v = (a_1, a_2, ..., a_n) where each element ai is selected from a finite ordered set S_i,
- We build from a partial solution of length k v = (a_1, a_2, ..., a_k) and try to extend it by adding another element. After extending it, we will test whether what we have so far is still possible as a partial solution.
- If it is still a candidate solution, great. If not, we delete a_k and try the next element from S_k: Compute S_1, the set of candidate first elements of v.

Summary :Backtracking

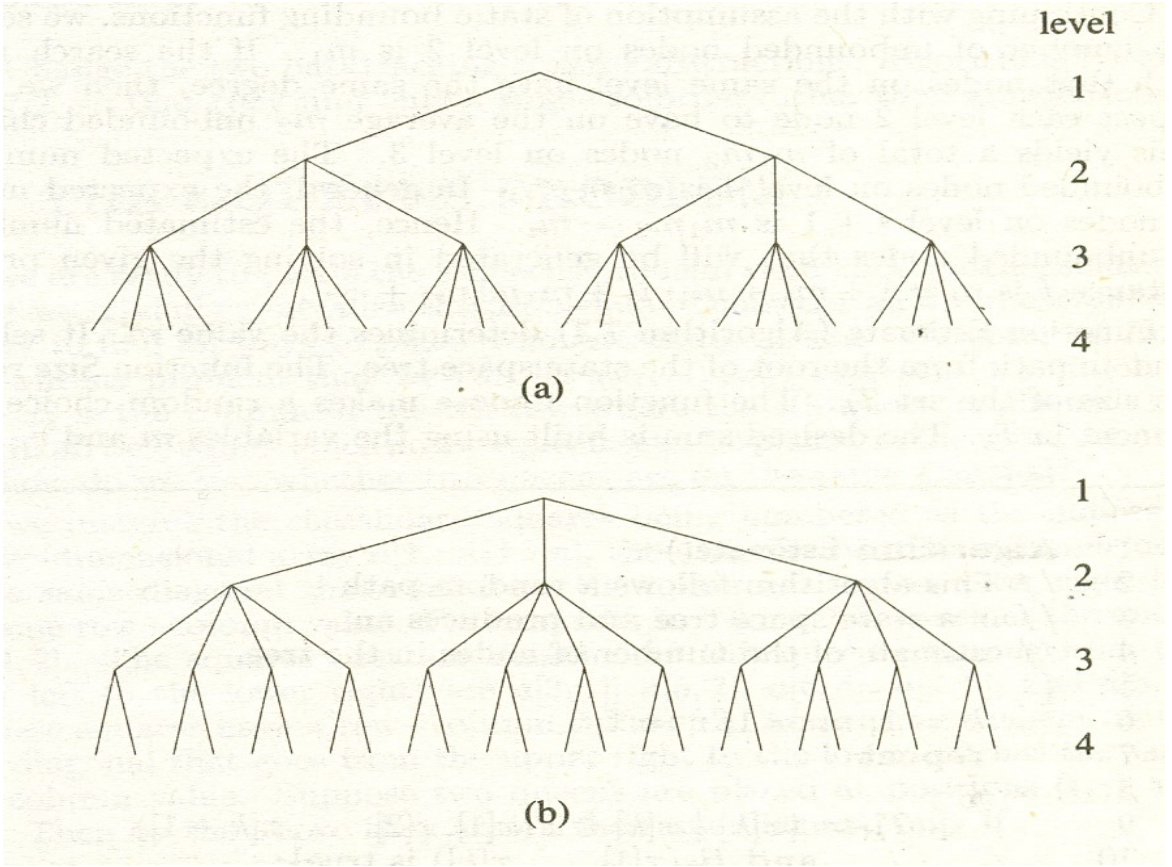
- Backtracking can easily be used to iterate through all subsets or permutations of a set.
- Backtracking ensures correctness by enumerating all possibilities.
- For backtracking to be efficient, we must prune the search space.

Summary :Backtracking

- A backtracking algorithm systematically considers all possible outcomes for each decision.
- In this sense, backtracking algorithms are like the brute-force algorithms and distinguished by the way in which the space of possible solutions is explored.
- Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better.

Rearrangement of search tree

- Searching can be efficient with the rearrangement of solution space.



Estimate

```
1   Algorithm Estimate()
2   // This algorithm follows a random path
3   // in a state space tree and produces an
4   // estimate of the number of nodes in the tree.
5   {
6       k := 1; m := 1; r := 1;
7       repeat
8       {
9           T_k := {x[k] | x[k] ∈ T(x[1], x[2], ..., x[k − 1])
10                  and B_k(x[1], ..., x[k]) is true};
11          if (Size(T_k) = 0) then return m;
12          r := r * Size(T_k); m := m + r;
13          x[k] := Choose(T_k); k := k + 1;
14      } until (false);
15  }
```

Excercise

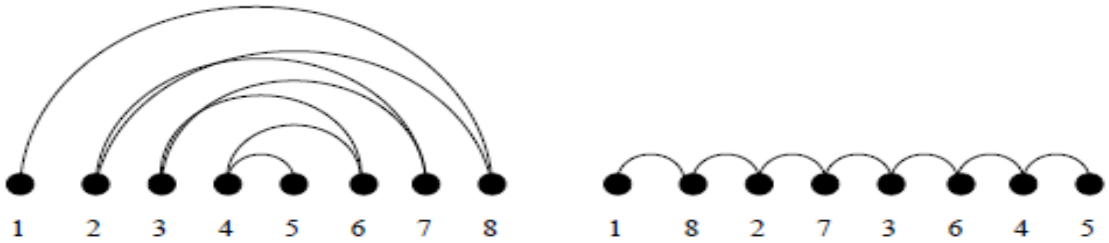
1. What are the main steps of a backtracking algorithm?
2. What is dead-end in a backtracking or branch-and-bound algorithm?
3. What are the worst-case running times of the backtracking and branch-and-bound algorithms discussed in the text?
4. How branch-and-bound is different from backtracking?

Thank you

SEARCHING SOLUTION SPACE

Solving Bandwidth with backtracking

- The *bandwidth problem takes as input a graph G, with n* vertices and m edges (ie. pairs of vertices). The goal is to find a permutation of the vertices on the line which minimizes the maximum length of any edge.



- The bandwidth problem has a variety of applications, including circuit layout, linear algebra, and optimizing memory usage in hypertext documents

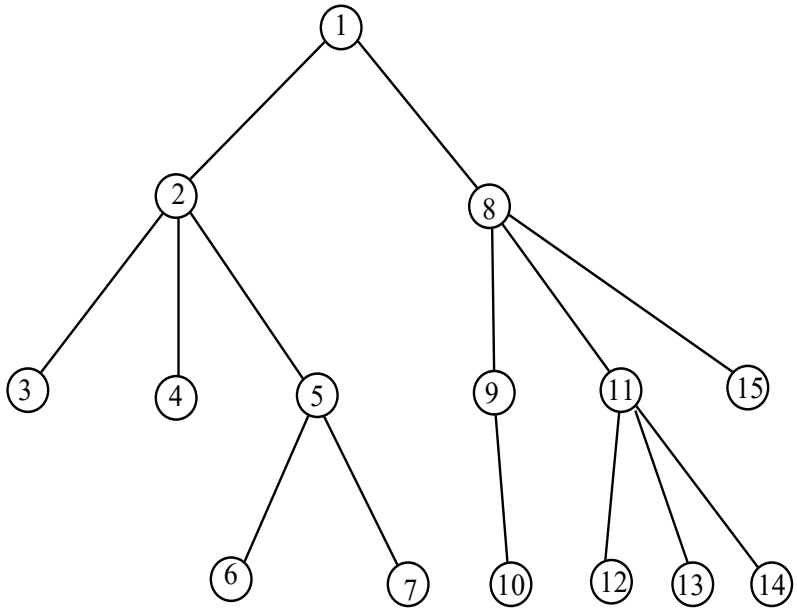
Solving Bandwidth with backtracking

- The problem is NP-complete, meaning that it is *exceedingly* unlikely that you will be able to find an algorithm with polynomial worst-case running time. It remains NP-complete even for restricted classes of trees.
- Since the goal of the problem is to find a permutation, a backtracking program which iterates through all the n!
- possible permutations and computes the length of the longest edge for each gives an easy **O(n! ·m)** algorithm. But the goal of this assignment is to find as practically good an algorithm as possible

The backtrack traversal

- Backtracking is really an algorithm for tree traversal(Solution Space).
- The goal of tree traversal is to start at the root and systematically visit every node, without missing any and without unnecessary duplication.
- The backtrack traversal strategy works as follows:

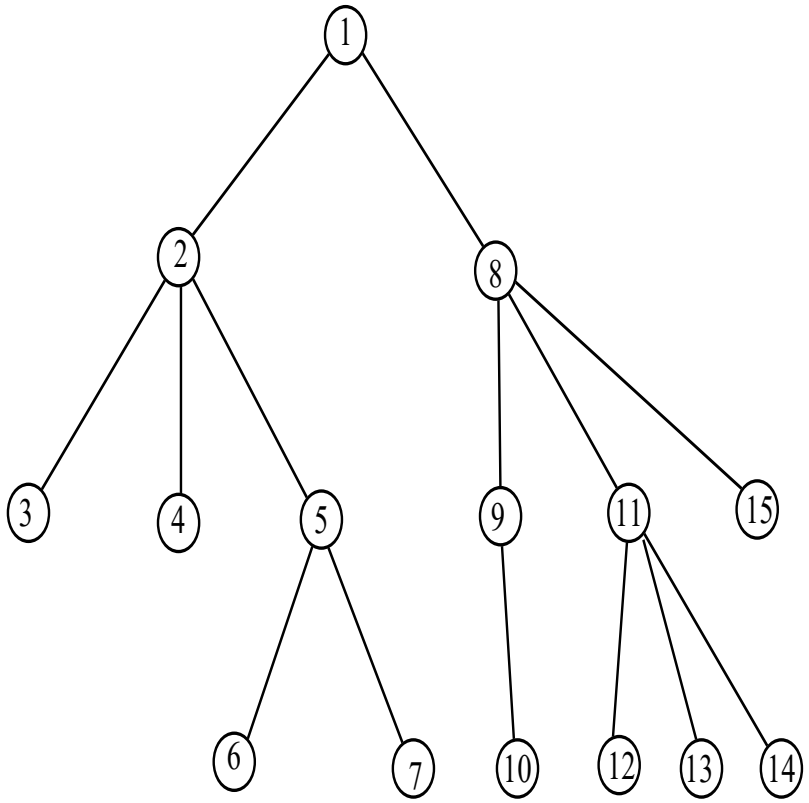
  1,2,3,2,4,2,5,6,5,7,5,2,1,
  8,9,10,9,8,11,12,11,13,11,14,
  11, 8,15,8,1



Algorithm: The backtrack traversal

Algorithm: The backtrack traversal

1. begin at the root
2. while (true)
3. {
4. while(true)
5. {
6. break when there are no more unexplored paths from the current node.
7. descend along the left-most untraversed edge.
8. }
9. // An attempt to descend along a new path has failed, so backtrack:
10. break if at the root (all nodes have been visited.)
11. ascend one level (backtrack to parent of current node)
12. }



Worst-case analysis of the asymptotic complexity of backtracking

- Suppose that solution sequences are of length n, and the number of possible values in each position of a sequence is k.
- Then the total number of possible sequences is kn, and the worst case is that when every sequence of length less than or equal to n must be generated.
- Then the number of nodes in the backtracking tree is

$$1 + k + k^2 + k^3 + \ldots + k^n = (k^{(n+1)} - 1)/(k-1) \cong k^n$$

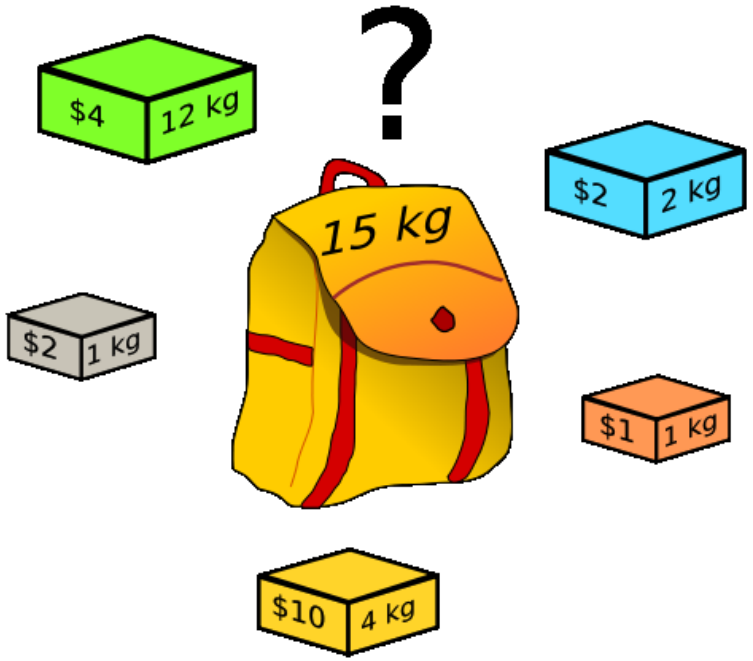
- Note that the total number of nodes in the backtracking tree is not much different from the number of leaves.
- In a binary tree of height h, for example, there are **2h** leaves, and 2h-1 internal nodes; thus, there are slightly more leaves than non-leaves. If the branching factor is greater than 2 and all leaves occur a distance h from the root, then an even greater fraction of the nodes are leaves.

Analysis of the backtrack algorithm

- In the worst case, the filter test must be applied to each one of the k^n leaf nodes. Even if the complexity of the filter test is Q(1), backtracking is exponential. No wonder these programs run so slowly!
- In general, exponential algorithms are computationally intractable for all but the smallest problems. Fortunately, as will be shown in the next section, there are ways to speed up the basic backtrack process and make it practical

Backtracking algorithms

- Backtracking algorithms are based on a **DEPTH-FIRST** recursive search
- A backtracking algorithm:
  - Tests to see if a solution has been found, and if so, returns it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recur
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

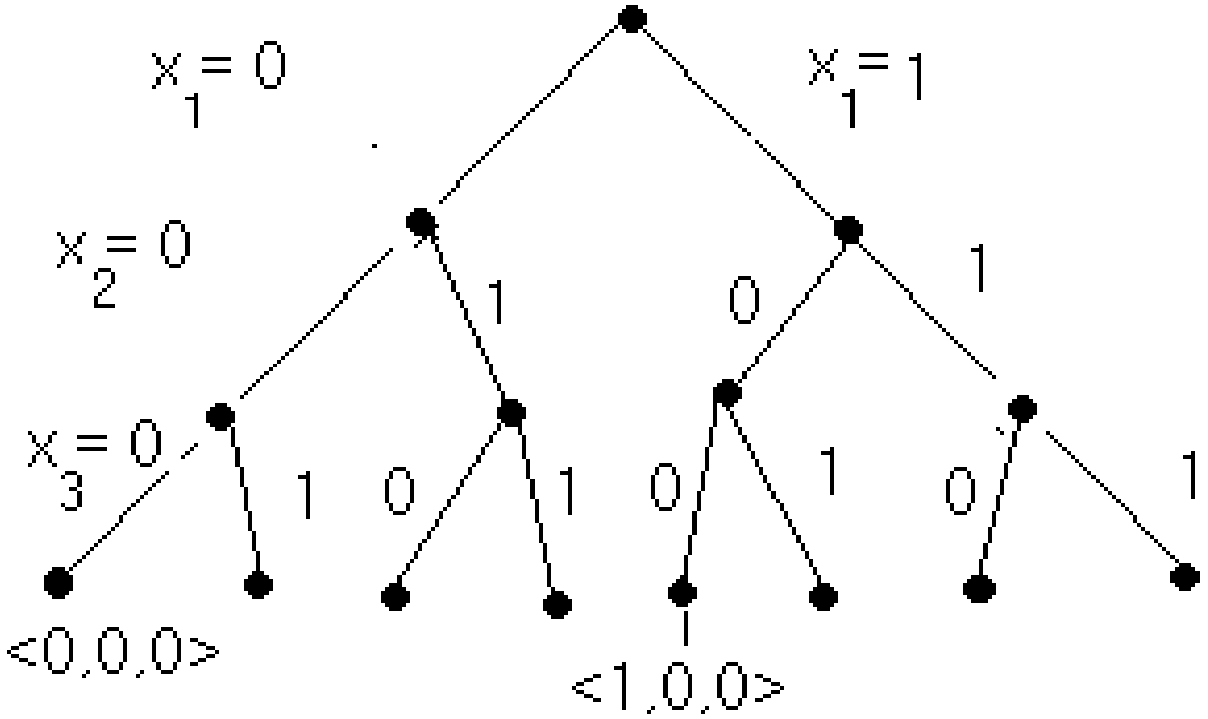


7.6 knapack problem pp.38?

0/1 Knapasack problem

0/1 Knapsack problem

- 0/1 Knapsack problem is an NP-hard
- The solution space for this problem is consists of the 2^n districts ways to assign zero or one values to the x_i's.



0-1 Knapsack Problem

- The maximization problem for 0/1 knapsack to be represented as follows:

Maximize $\sum_{i=1}^{n} p_i x_i$

Subjected to $\sum_{i=1}^{n} w_i x_i \leq m$

$$x_i = 0 \text{ or } 1, \qquad 1 \leq i \leq n$$

- The solution space can be arranged as two different way, (i)fixed tuple size formulation and (ii) variable tuple size formulation.
- Bounding functions are needed to kill some live nodes without expanding them.
- A good bounding function for 0-1 knapsack problem can be obtained by using upper-bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendents.
- If upper bond is not higher than the value of the best solution determined so far, then the live node can be killed.

- In fixed tuple size formulation an upper-bound for node Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$, and using the greedy algorithm.
- Upper bound of any node Z at level k+1 of the state space tree can be computed using algorithm Bound(cp,cw,k) .
- The computation assumes $p[i]/w[i] \geq p[i+1]/w[i+1]$, for $1 \leq i < n$.
- The back tracking algorithm to solve 0-1 knapsack problem can be invoked with parameter as follows:
- Bknap(1,0,0)
- When fp ≠ -1, x[i], $1 \leq i \leq n$ is such that $\sum_{i=1}^{n} p_i x_i$ =fp

Bounding function for backtracking

```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
9           if (c < m) then b := b + p[i];
10          else return b + (1 - (c - m)/w[i]) * p[i];
11      }
12      return b;
13  }
```

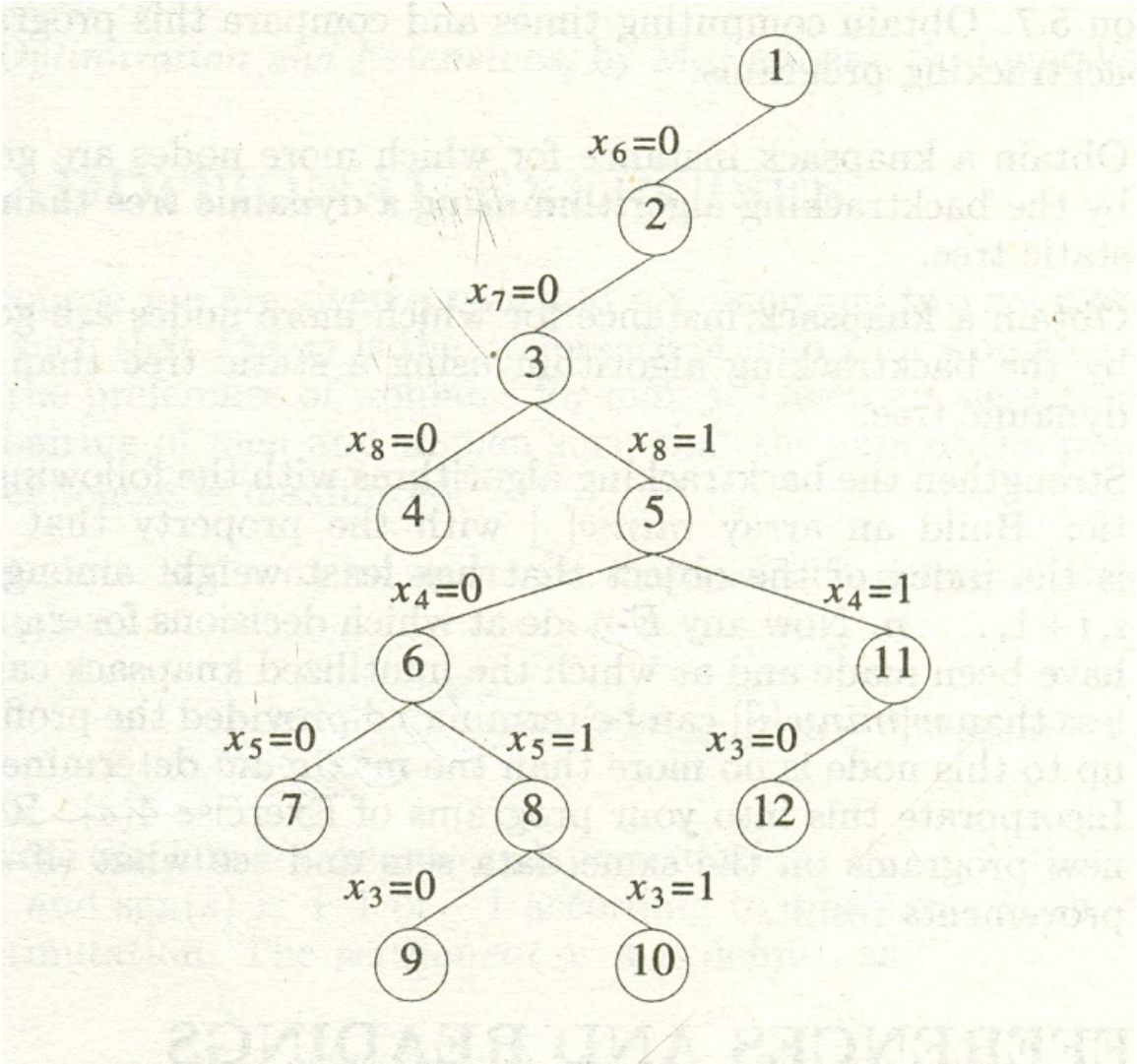
Backtracking algorithm 0-1 knapsack problem

```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```

Example: 0-1 knapsack problem

- P={11,21,31,33,43,53,55,65}
- W={1,11,21,23,33,43,45,55}
- M = 110 and n=8

Dynamic state space tree

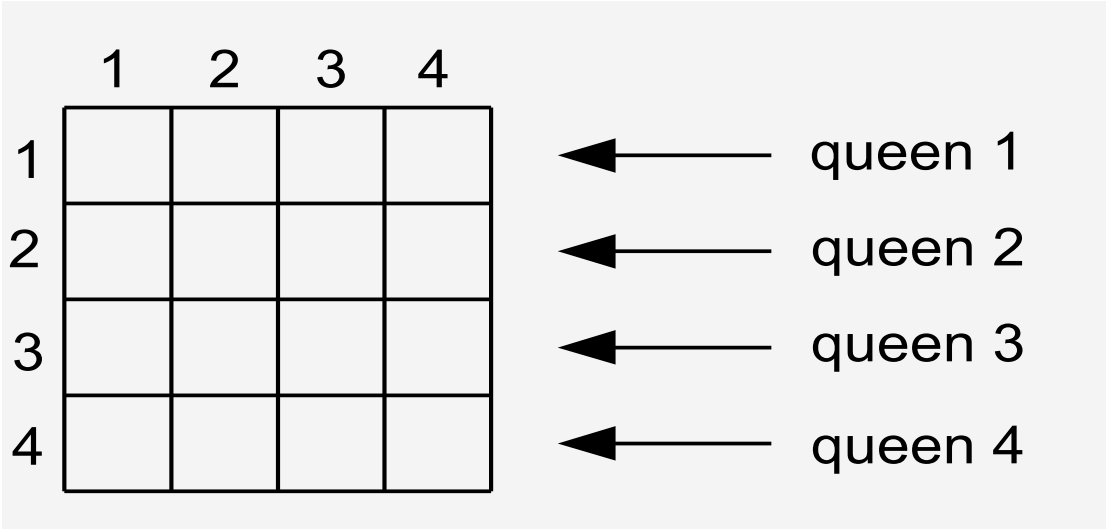


7.2 The 8-queens problem pp.373-376

N-QUEENS PROBLEM

Example: *n*-Queens Problem

The problem is to place *n* queens on an *n*-by-*n* chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

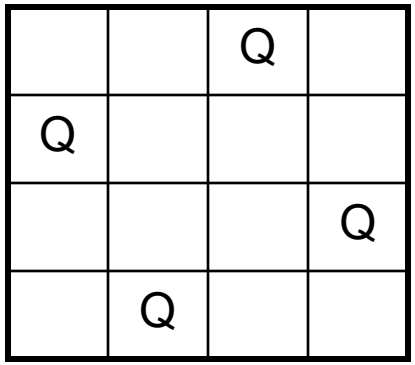


Solving the *n*-Queens Problem Using Backtracking

- The *n*–queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal.
- Using backtracking, this algorithm outputs all solutions to this problem.
- We place queens successively in the columns beginning in the left column and working from top to bottom.
- When it is impossible to place a queen in a column, we return to the previous column and move its queen down.

The n-Queens Problem

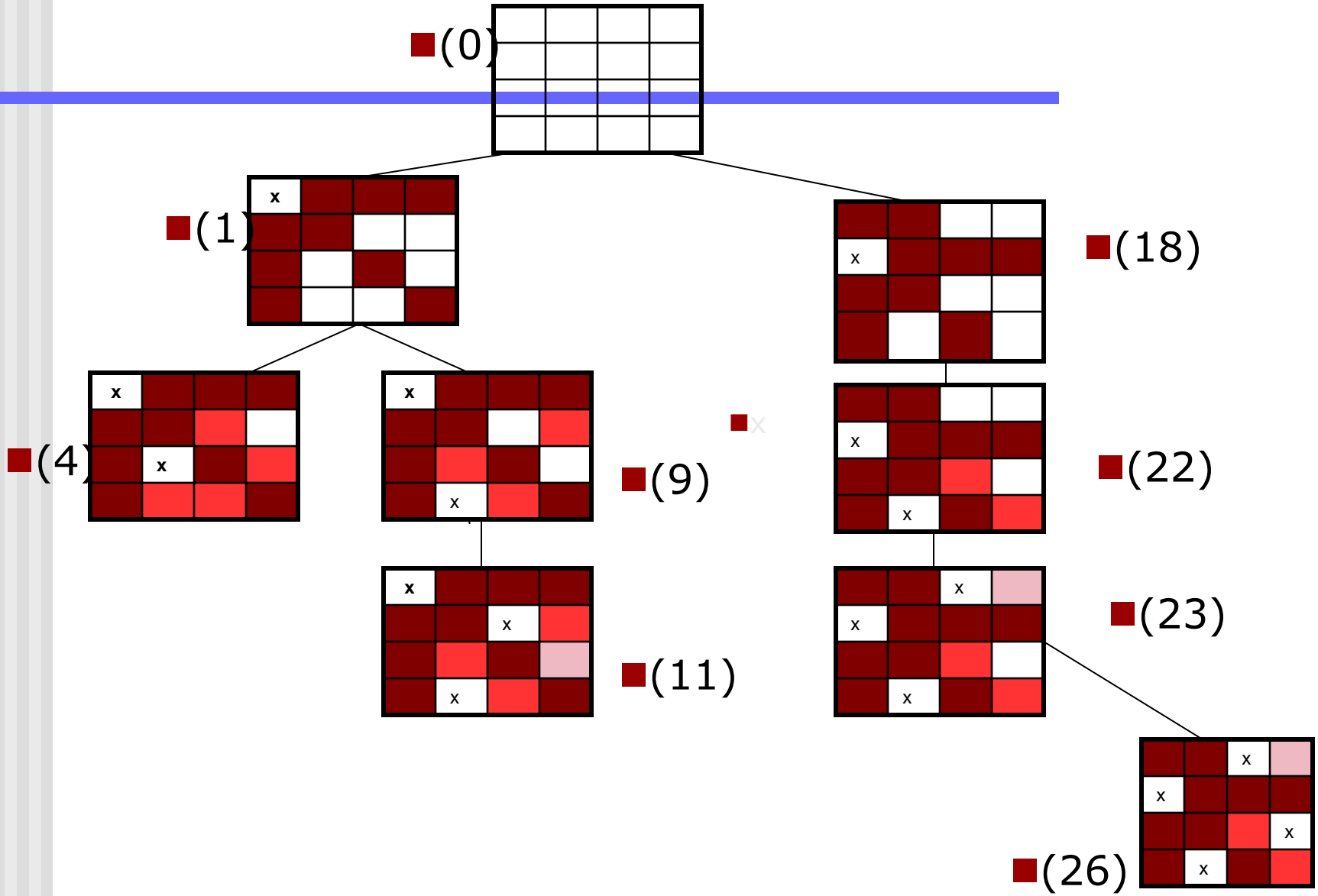
- The problem is to place n queens (chess pieces) on an n by n board so that no two queens are in the same row, column or diagonal

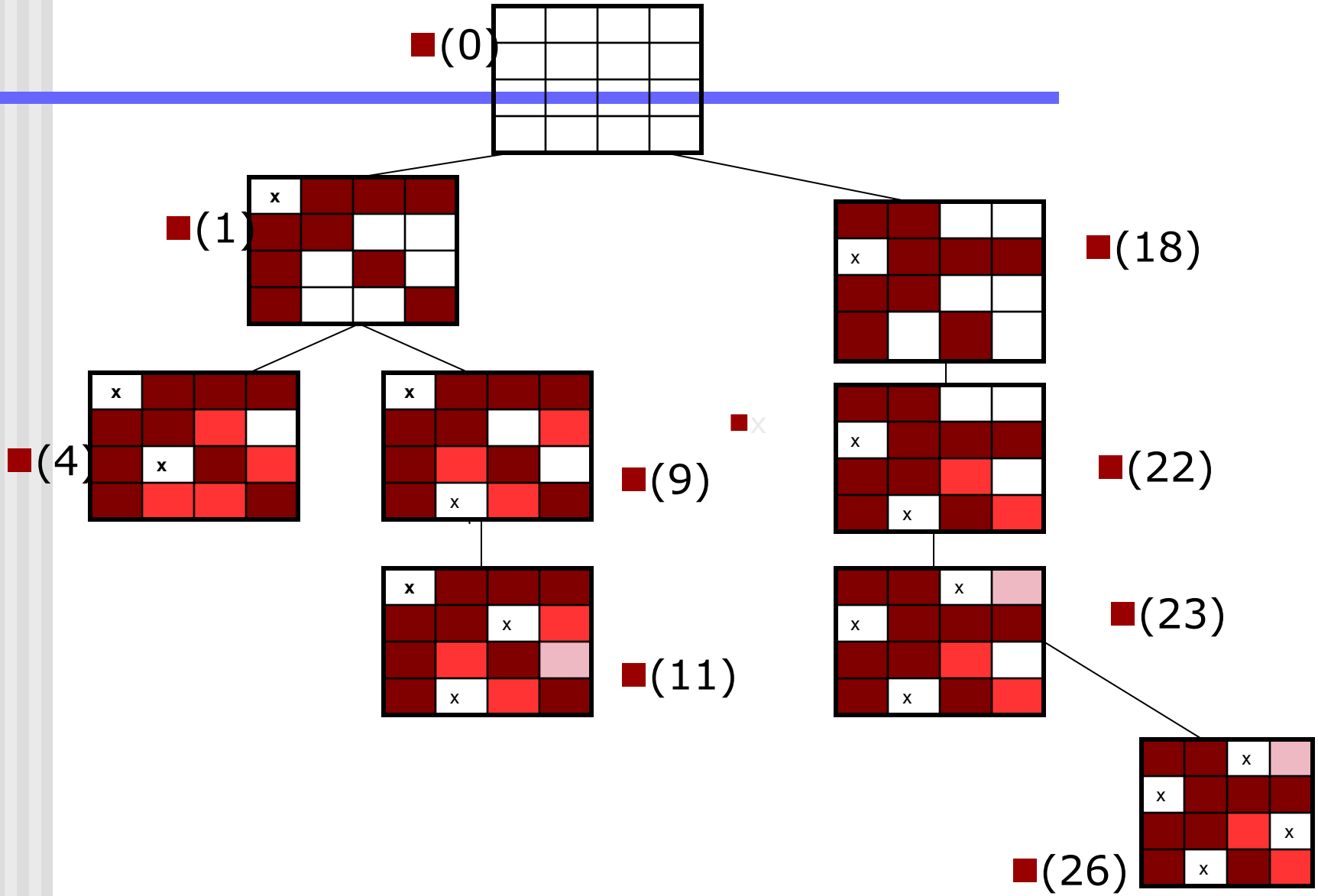


- The 4 by 4 board above shows a solution for n = 4
- But first we will introduce an algorithm strategy called ***backtracking***, which can be used to construct ***all*** solutions for a given n.

The n-Queens Problem

- We can solve this problem by generating a dynamic tree in a depth-first manner
- We will try to place a queen in each column from the first to the last
- For each column, we must select a row
- We start by placing a queen in row 1 of the first column
- Then we check the rows in the second column for positions that do not conflict with our choice for the first column
- After placing a queen in the second column, we continue to the third, etc.
- If at any point we find we cannot place a queen in the current column, we back up to the previous column and try a different row for that column
- We then continue trying to place more queens
- This process can be visualized by a tree of configurations
- The nodes are partial solutions of the problem
- The root is an empty board
- The children of the root will be boards with placements in the first column
- We will generate the nodes of the tree dynamically in a depth-first way


(0)
(1)
(4)
(9)
(11)
(18)
(22)
(23)
(26)


(0)
(1)
(4)
(9)
(11)
(18)
(22)
(23)
(26)

Backtracking Solution to n-Queens Problem

- We could continue with the previous example to obtain all possible solutions for n = 4
- Our goal now is to convert this approach into an explicit algorithm
- We track the position of the queens by using an array row
- row[k] is the row in column k containing a queen
- The key function is the recursive function **rn**_queens
- When **rn**_queens(k,n) is called, queens have been successfully placed in columns 1 to k-1
- The function then attempts to place a queen in column k
- If it is successful, then
  - if k == n, it prints the solution
  - if k < n, it makes the call rn_queens(k+1,n)
  - it then returns to the call rn_queens(k-1,n)

Backtracking Solution to n-Queens Problem

- To test for a valid position for a queen in column k, we use a function ***position_ok(k,n)*** which returns true if and only if the queen tentatively placed in column k does not conflict with the queens in positions 1 to k-1
- The queens in columns i and k conflict if
  - they are in the same row: row[i] == row[k]
  - or in the same diagonal: | row[k] – row[i] | == k – i
- We are thus led to our first version of the functions

Backtracking Solution to n-Queens Problem

```
n_queens(n) {
  rn_queens(1,n)
}

rn_queens(k,n) {
   for row[k] = 1 to n
      if (position_ok(k,n) )
         if ( k == n ) {
            for i = 1 to n)
                print(row[k] + "  ")
            println
         }
         else
            rn_queens(k+1,n)
}

position_ok(k,n) {
  for i = 1 to k-1
     if (row[k] == row[i]  ||  abs(row[k]-row[i]) == k-i)
               return false
  return true
}
```

Recursive algorithm for n-queen

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Placement algorithm

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k – 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k – 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] – i) = Abs(j – k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Running Time

- Improvements can be made so that position_ok(k,n) runs in $O(1)$ time rather than $\boldsymbol{O}(k)$ time. See the text.
- We will will obtain an upper bound for the running time of our algorithm by bounding the number of times rn_queens(k,n) is called for each k<n
  - **k = 1**: **1** time, by n_queens
  - **1 < k < n**: at most **n(n-1)⋯(n-k+2)**, since there are n(n-1)⋯(n-k+2) ways to place queens in the first k-1 columns in distinct rows
- Ignoring recursive calls, rn_queens(k,n) executes in $\Theta(n)$ time for k < n.
- This gives a worst-case bound for rn_queens(k,n) of
  n [n(n-1)⋯(n-k+2)] for 1 < k < n
- For k = n, there is at most one placement possible for the queen. Also, the loop in rn_queens executes in $\Theta(n)$ time.
- There are n(n-1)⋯2 ways for the queens to have been placed in the first n-1 columns, so the worst case time for rn_queens(n,n) is
  n [n(n-1)⋯2]

Running Time

- Combining the previous bounds, we get the bound

$$\begin{aligned} & \mathbf{n\,[\,1 + n + n(n\text{-}1) + \cdots + n(n\text{-}1)\cdots 2]} \\ = \; & \mathbf{n{\cdot}n!\,[\,1/n! + 1/(n\text{-}1)! + \cdots + 1/1!\,]} \end{aligned}$$

- A result from calculus: $e = \sum_{i=0}^{\infty} \frac{1}{i!} = 1 + \sum_{i=1}^{\infty} \frac{1}{i!}$
- This means that $\mathbf{n{\cdot}n!\,[\,1/n! + 1/(n\text{-}1)! + \cdots + 1/1!\,] \leq n{\cdot}n!\,(e\text{-}1)}$
- We thus have that our algorithm runs in $\boldsymbol{O(n{\cdot}n!)}$ **time**

Exercise

1. Describe and analyze algorithms for the following generalizations of SUBSETSUM:

   (a) Given an array $X[1..n]$ of positive integers and an integer T, compute the *number* of subsets of X whose elements sum to T.

   (b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the *weight* of the corresponding element $X[i]$, compute the *maximum weight* subset of X whose elements sum to T. sIf no subset of X sums to T, your algorithm should return $-\infty$.

2. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function $lcs(A, B)$, which gives the length of the *longest* common subsequence of A and B.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function $scs(A, B)$, which gives the length of the *shortest* common supersequence of A and B.

Exercise

(c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.

(d) Give a simple recursive definition for the function $sos(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.

(e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.