Algorithm Design & Analysis Chapter -03 (Backtracking & Branch and Bound) T.E(Computer)

By

I.S Borse

SSVP'S BSD COE ,DHULE

Outline – Chapter 3

1. Backtracking

i) Eight Queens Problem

ii) Graph Coloring

iii) Hamilton Cycles

iv) Knapsack Problem

2. Branch and Bound

i) Traveling salesman's problem

ii) lower bound theory-comparison trees for sorting /searching

iii) lower bound on parallel computation.

A short list of categories

- Algorithm types we will consider include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

Backtracking

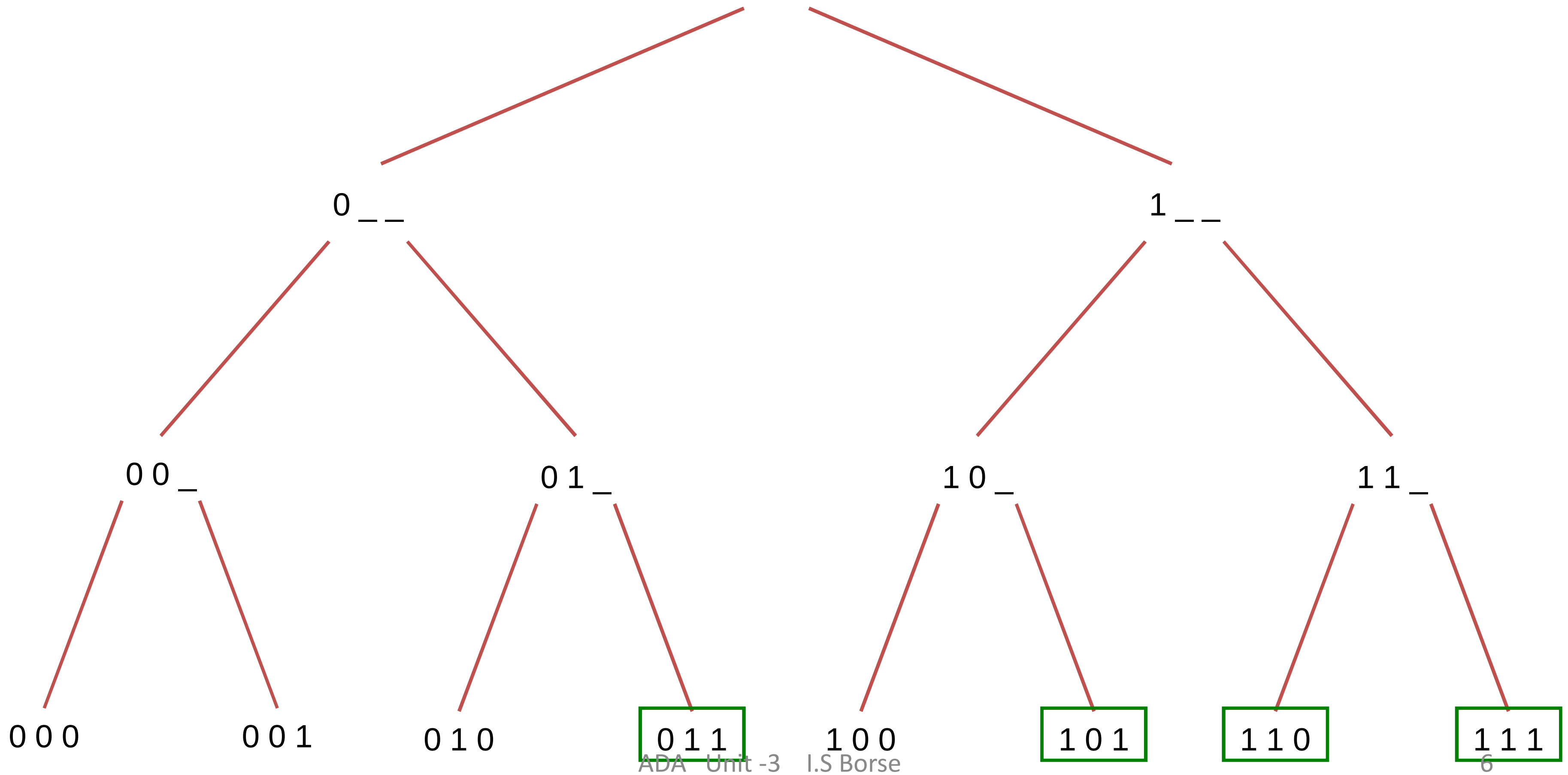
- Suppose you have to make a series of *decisions,* among various *choices,* where
  - You don't have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that "works"

Backtracking

- *Problem:*

  Find out all 3-bit binary numbers for which the sum of the 1's is greater than or equal to 2.
- The only way to solve this problem is to check all the possibilities: (000, 001, 010, ...,111)
- The 8 possibilities are called the search space of the problem. They can be organized into a tree.

Backtracking: Illustration



Backtracking

- For some problems, the only way to solve is to check all possibilities.
- Backtracking is a systematic way to go through all the possible configurations of a search space.
- We assume our solution is a vector (a(1),a(2), a(3), ..a(n)) where each element a(i) is selected from a finite ordered set S.

Backtracking and Recursion

- Backtracking is easily implemented with recursion because:
- The run-time stack takes care of keeping track of the choices that got us to a given point.
- Upon failure we can get to the previous choice simply by returning a failure code from the recursive call.

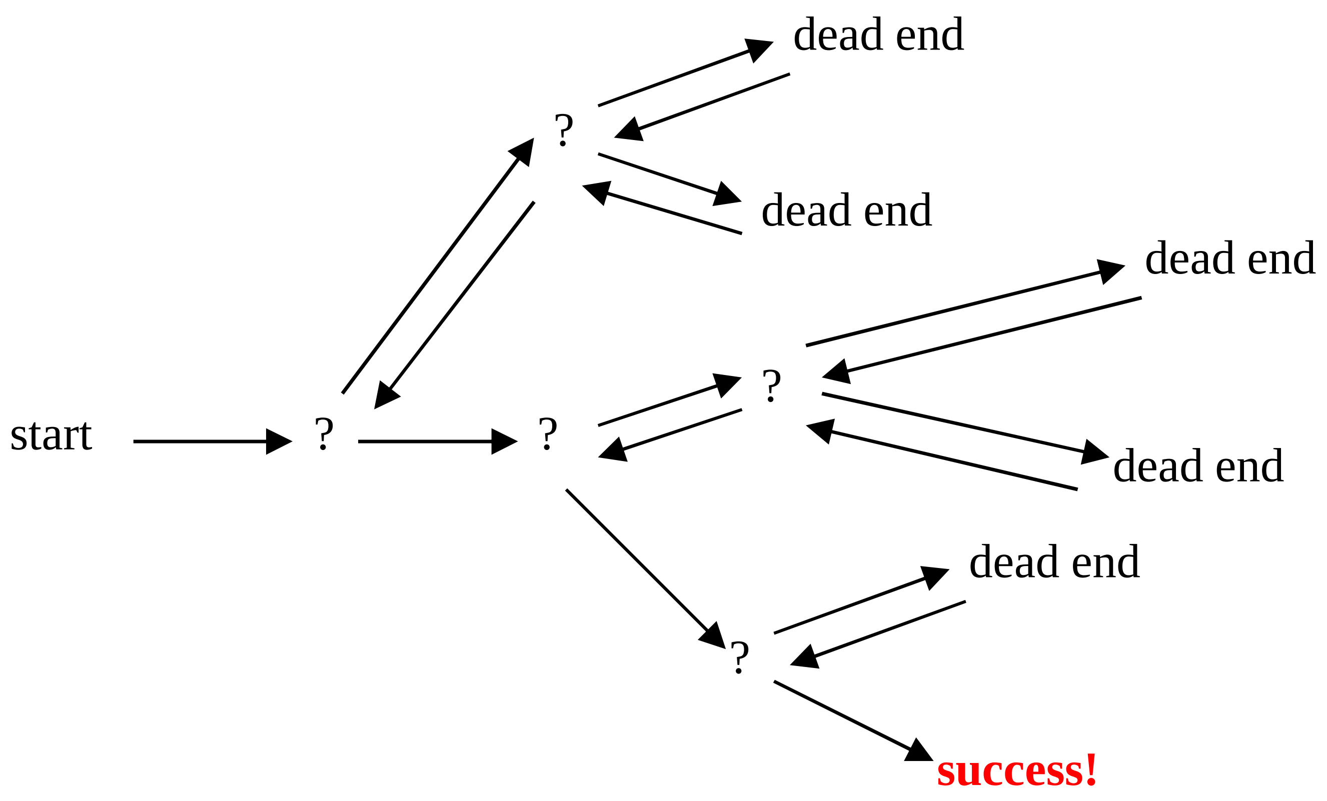
Improving Backtracking: Search Pruning

- It help us to reduce the search space and hence get a solution faster.
- The idea is to a void those paths that may not lead to a solutions as early as possible by finding contradictions so that we can backtrack immediately without the need to build a hopeless solution vector.

Solving a maze

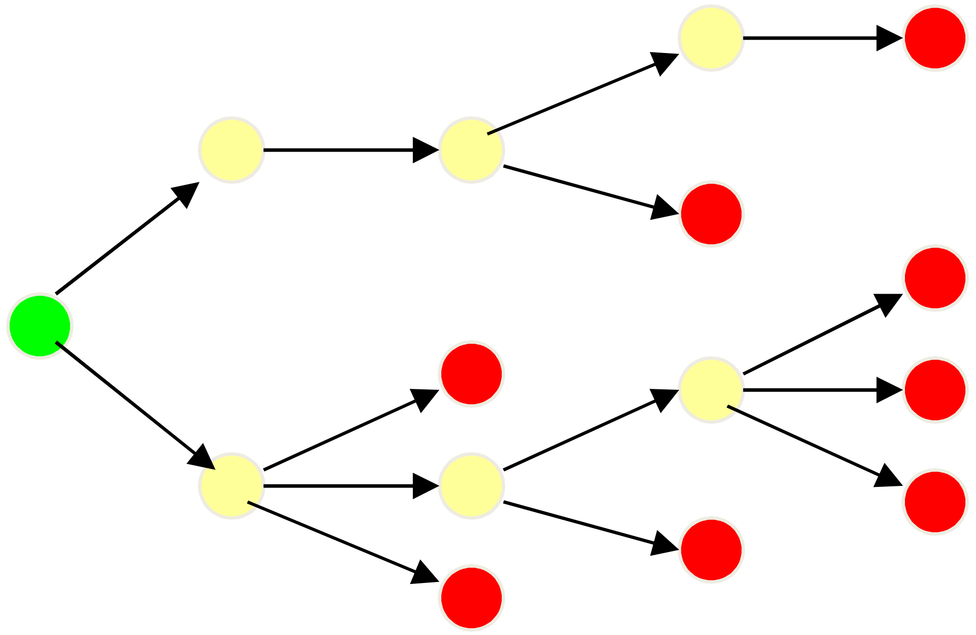
- Given a maze, find a path from start to finish
- At each intersection, you have to decide between three or fewer choices:
  - Go straight
  - Go left
  - Go right
- You don’t have enough information to choose correctly
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many types of maze problem can be solved with backtracking

Backtracking (animation)



Terminology I

A tree is composed of nodes



There are three kinds of nodes:

🟢 The (one) root node

🟡 Internal nodes

🔴 Leaf nodes

Backtracking can be thought of as searching a tree for a particular "goal" leaf node

Terminology II

- Live node: A node which has been generated and all of whose children have not yet generated
- E-node: The live node whose children are currently being generated
- Dead node : which is not expanded further or all of whose children have been generated
- DFS: Depth first node generation with bounding function is called backtracking.

The backtracking algorithm

- Backtracking is really quite simple--we “explore” each node, as follows:
- To “explore” node N:

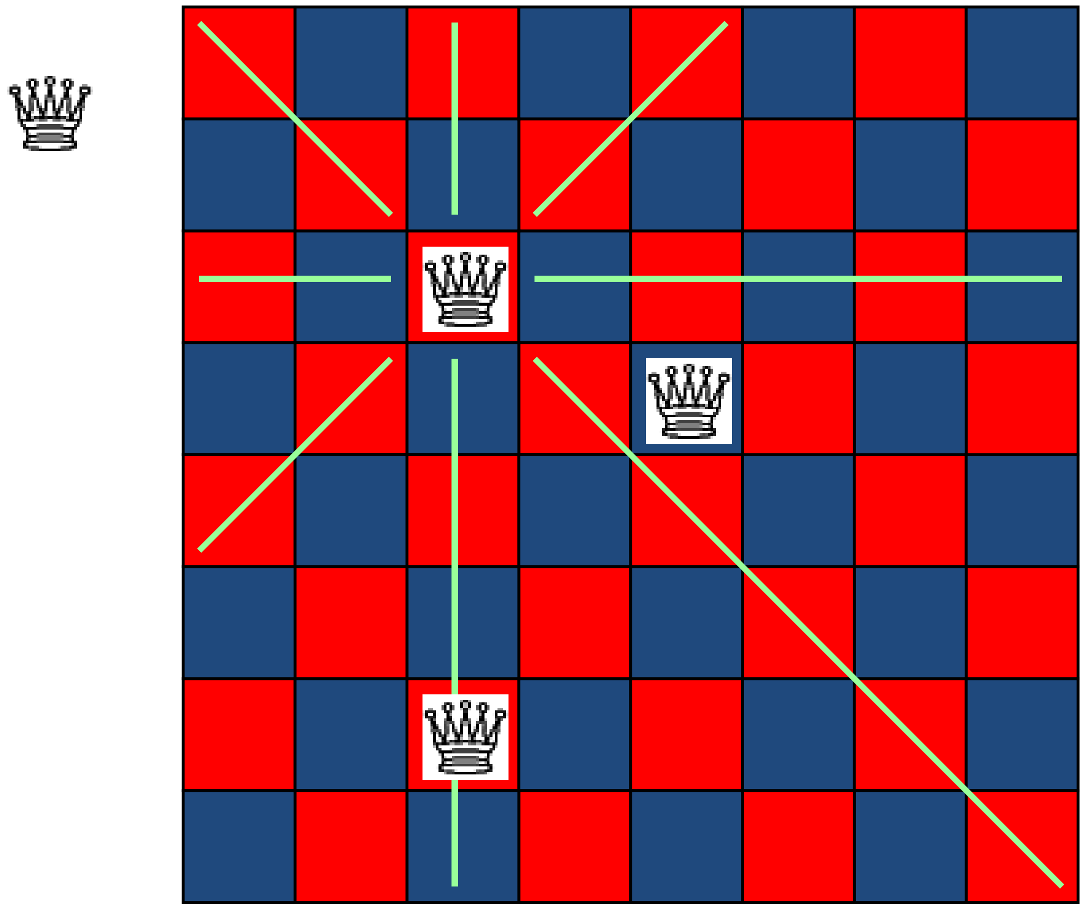
```
1. If N is a goal node, return “success”
2. If N is a leaf node, return “failure”
3. For each child C of N,
    3.1. Explore C
        3.1.1. If C was successful, return “success”
4. Return “failure”
```

n-Queens Problem

- Given: n-queens and an nxn chess board
- Find: A way to place all n queens on the board s.t. no queens are attacking another queen.

- How could you formulate this problem?
- How would you represent a solution?

How Queens Work



First Solution Idea

- Consider every placement of each queen 1 at a time.
  - How many placements are there?

First Solution Idea

- Consider every possible placement
  - How many placements are there?

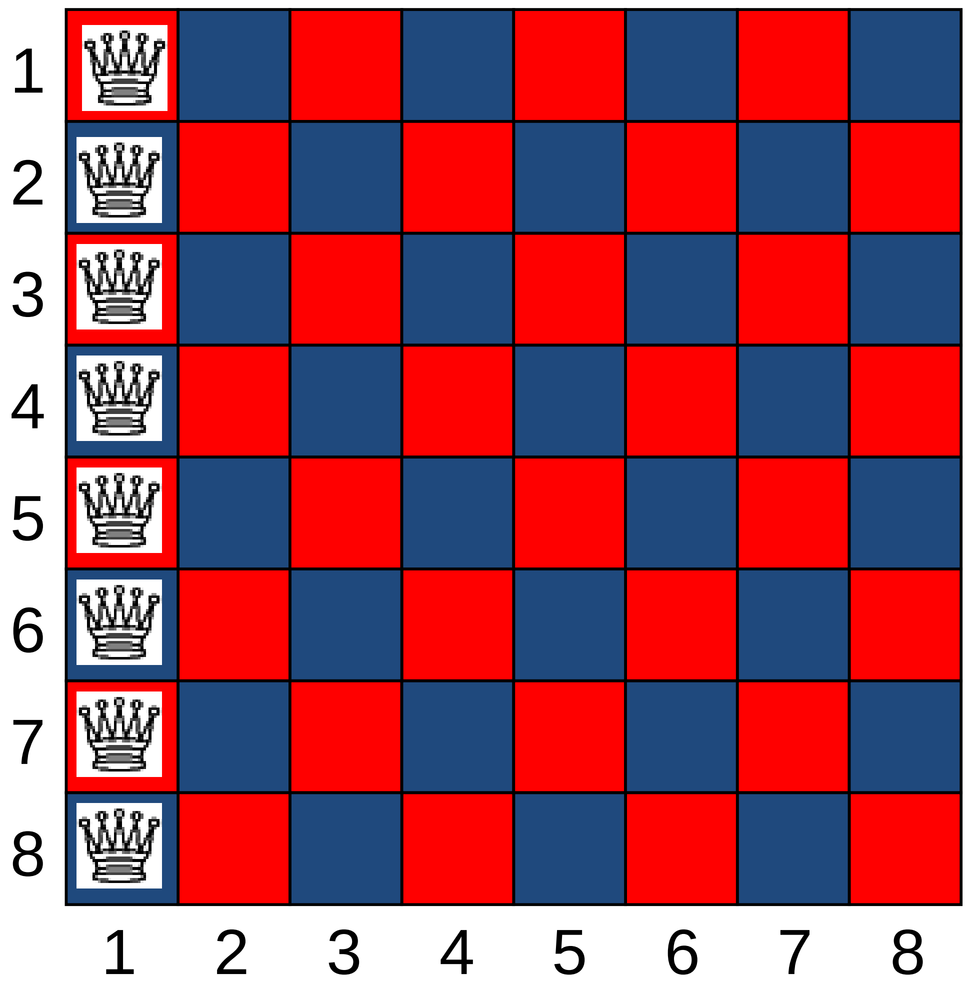
$$\binom{n^2}{n} \Rightarrow \binom{64}{8} = 4{,}426{,}165{,}368$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

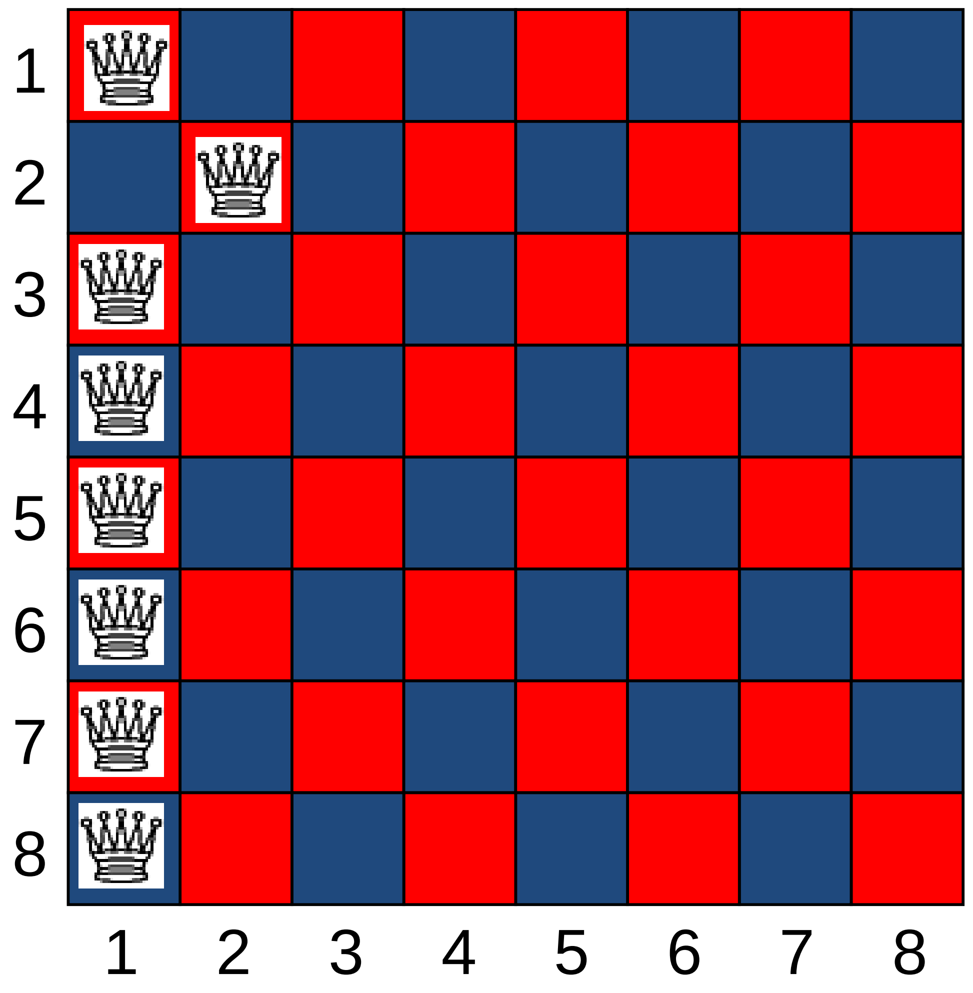
Second Solution Idea

- Don’t place 2 queens in the same row.
  - Now how many positions must be checked?

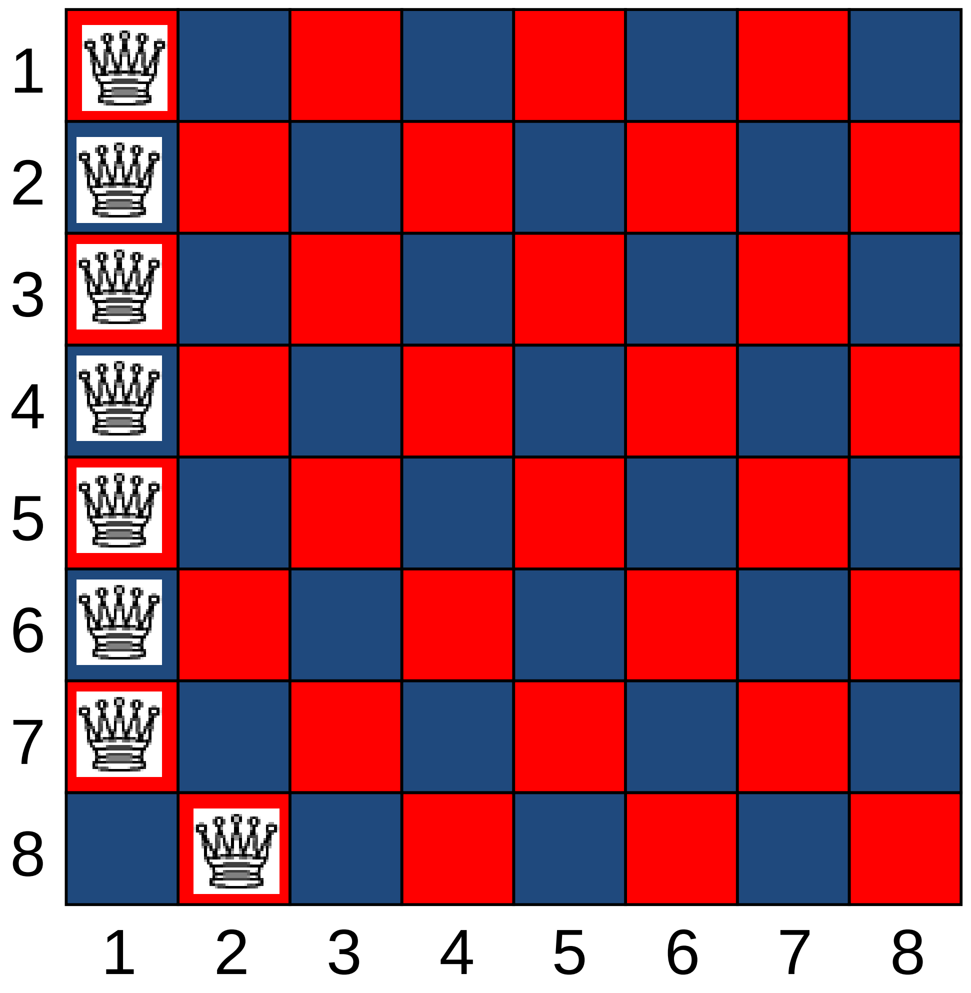
The Eight Queens Problem



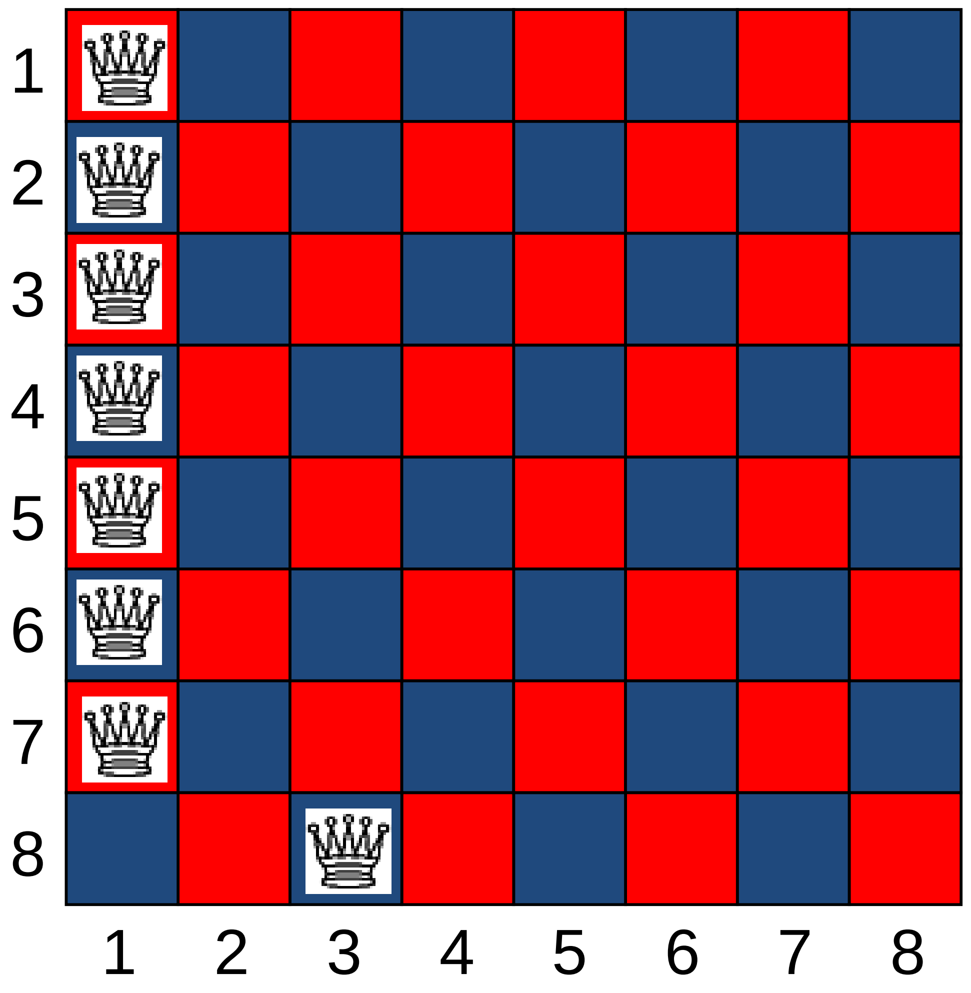
The Eight Queens Problem



The Eight Queens Problem



The Eight Queens Problem



Second Solution Idea

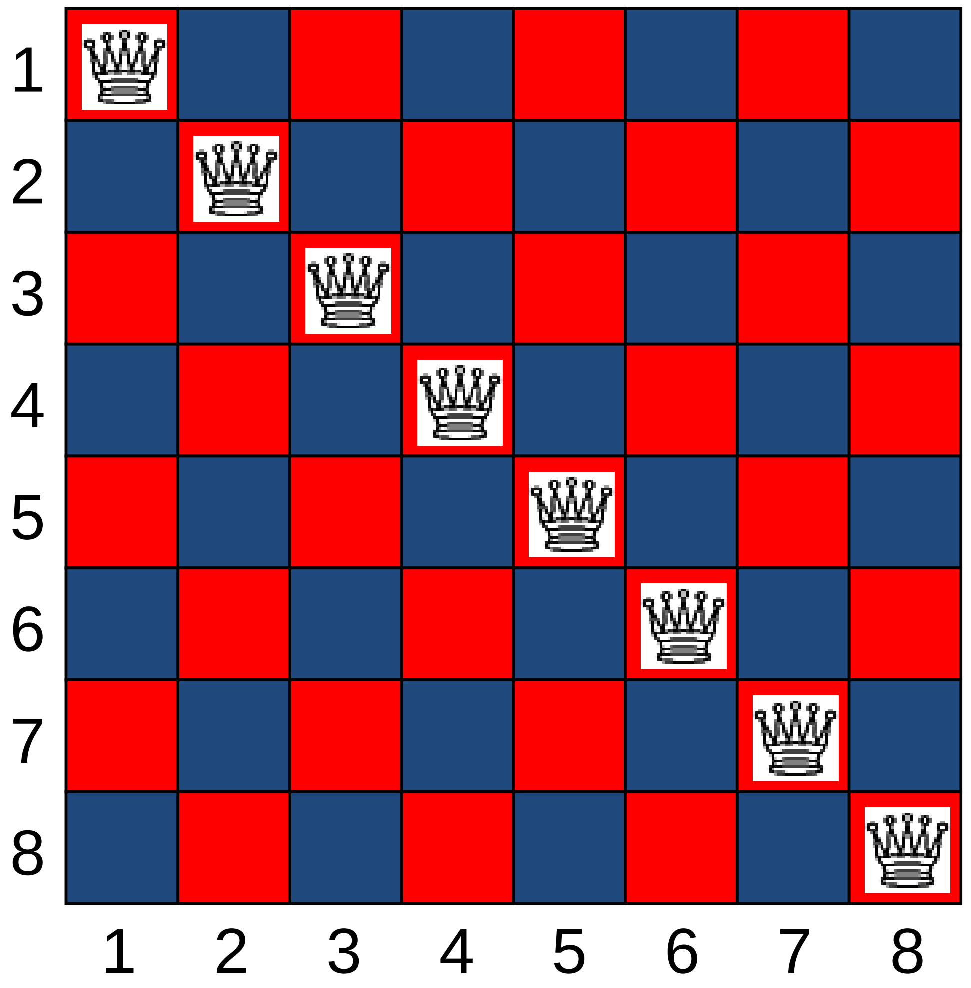
- Don’t place 2 queens in the same row.
  - Now how many positions must be checked?

Represent a positioning as a vector [x_1, …, x_8]
Where each element is an integer 1, …, 8.

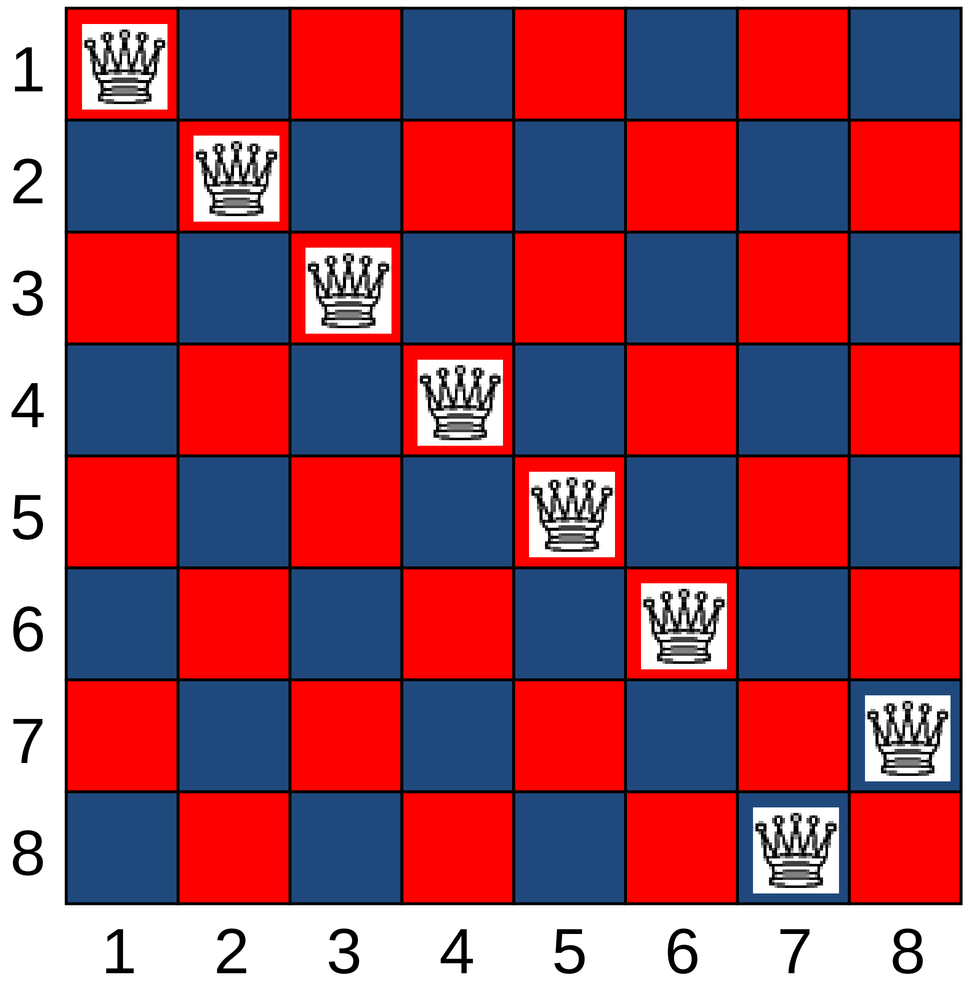
$$n^n = 8^8 = 16,777,216$$

Third Solution Idea

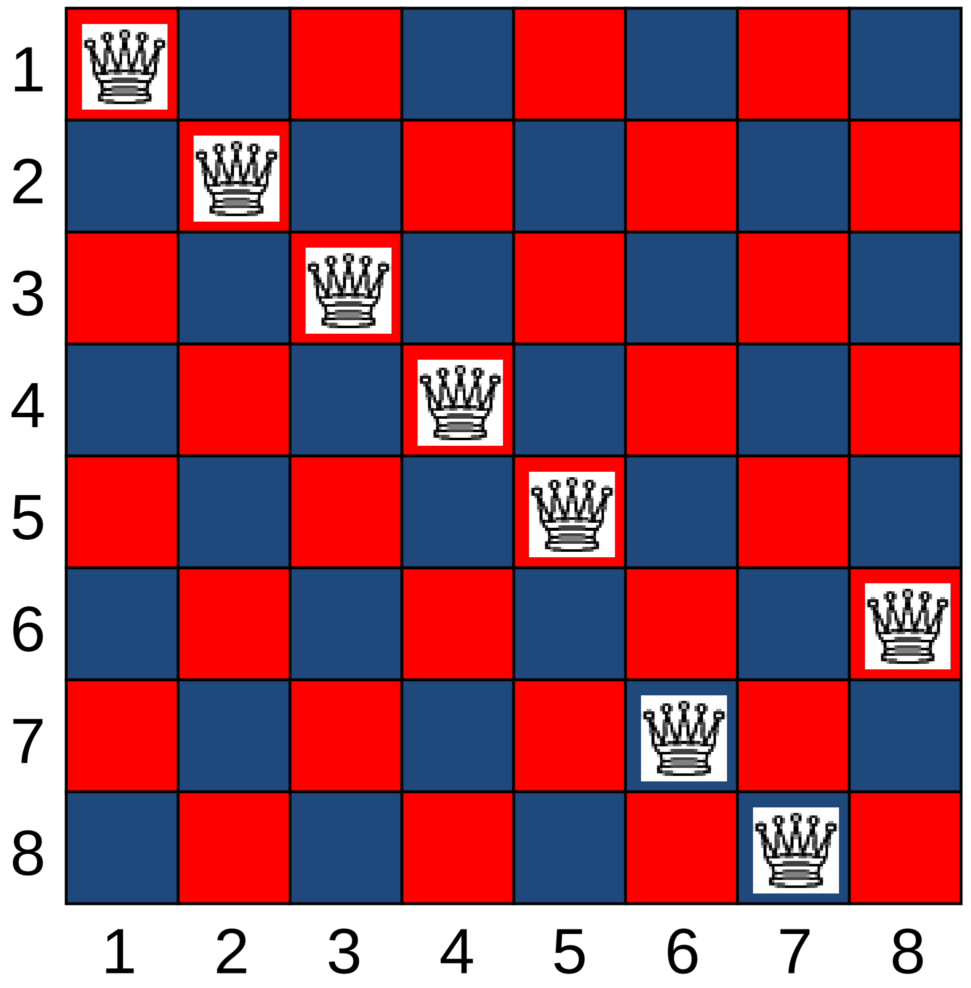
- Don’t place 2 queens in the same row or in the same column.
- Generate all permutations of (1,2…8)
  - Now how many positions must be checked?



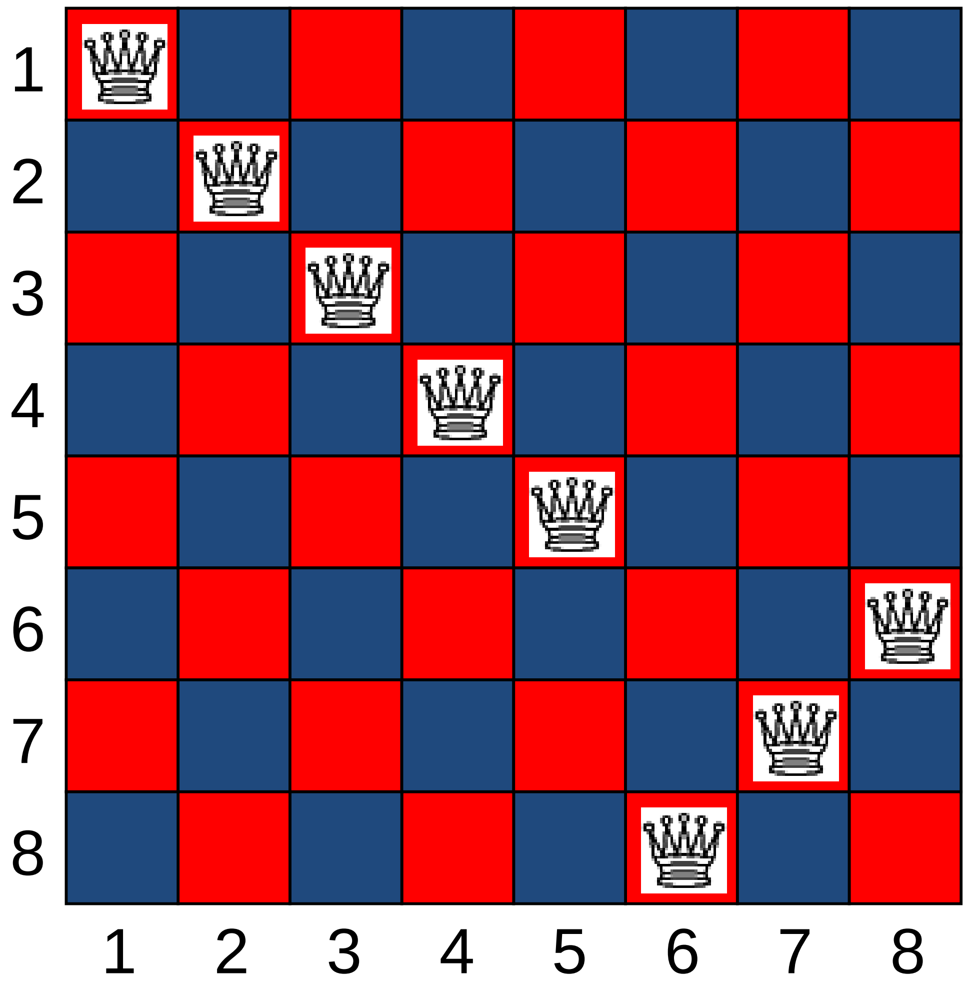
(1,2,3,4,5,6,7,8)



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



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



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

Third Solution Idea

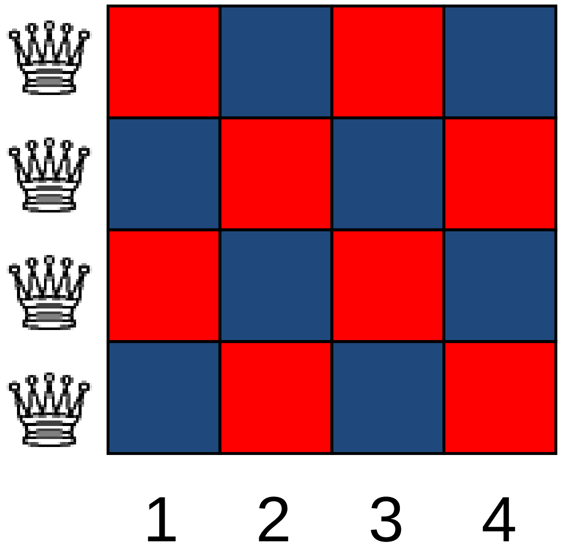
- Don’t place 2 queens in the same row or in the same column.
- Generate all permutations of (1,2…8)
  - Now how many positions must be checked?

$$n! = 8! = 40,320$$

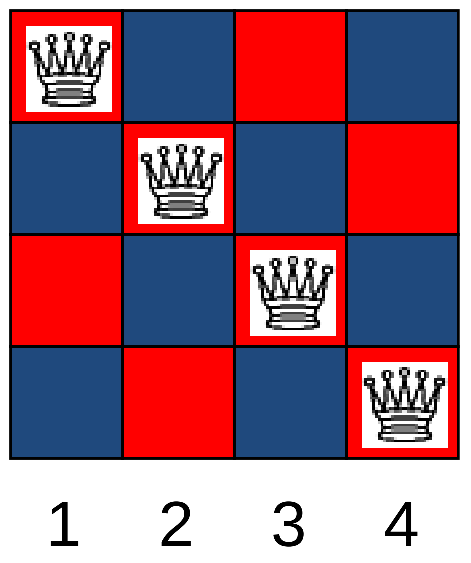
- We went from $C(n^2, n)$ to n^n to $n!$
  - And we’re happy about it!

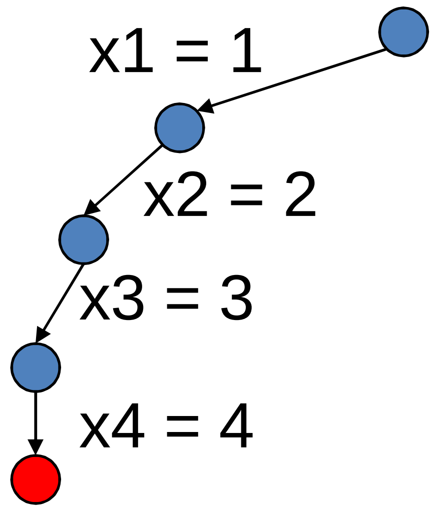
- We applied *explicit constraints* to shrink our search space.

Four Queens Problem



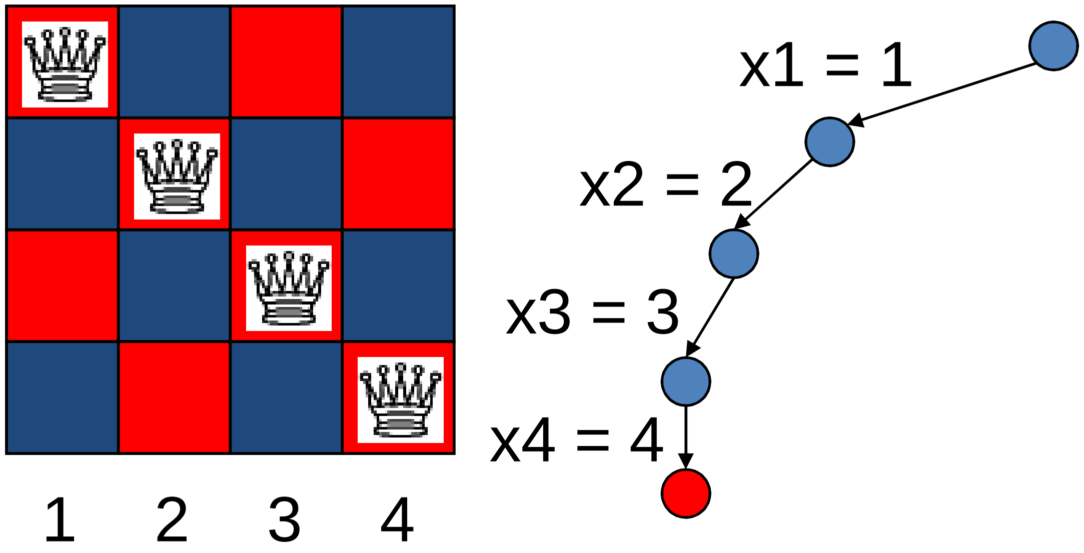
Four Queens Problem





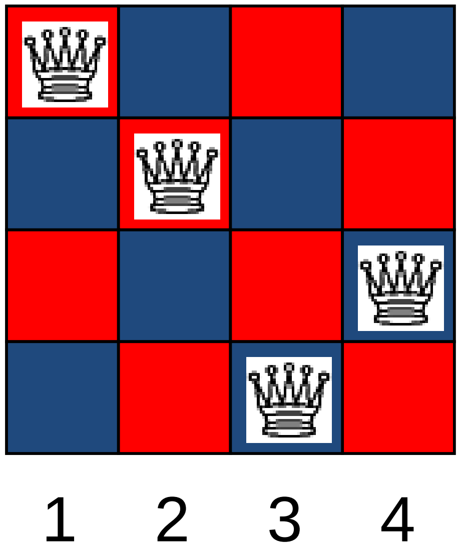
Each branch of the tree represents a decision to place a queen.
The criterion function can only be applied to leaf nodes.

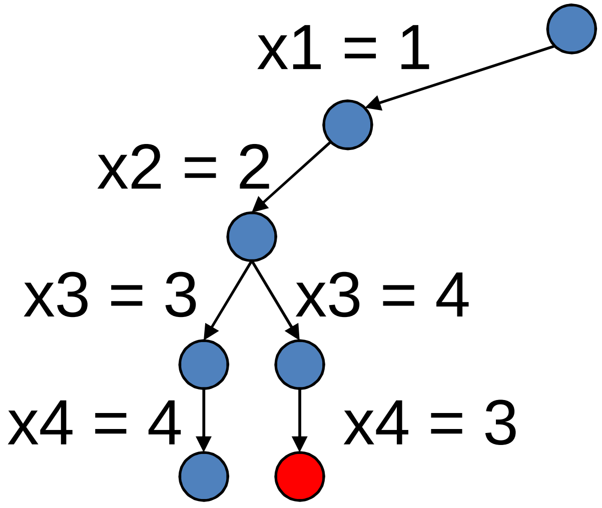
Four Queens Problem



Is this leaf node a solution?

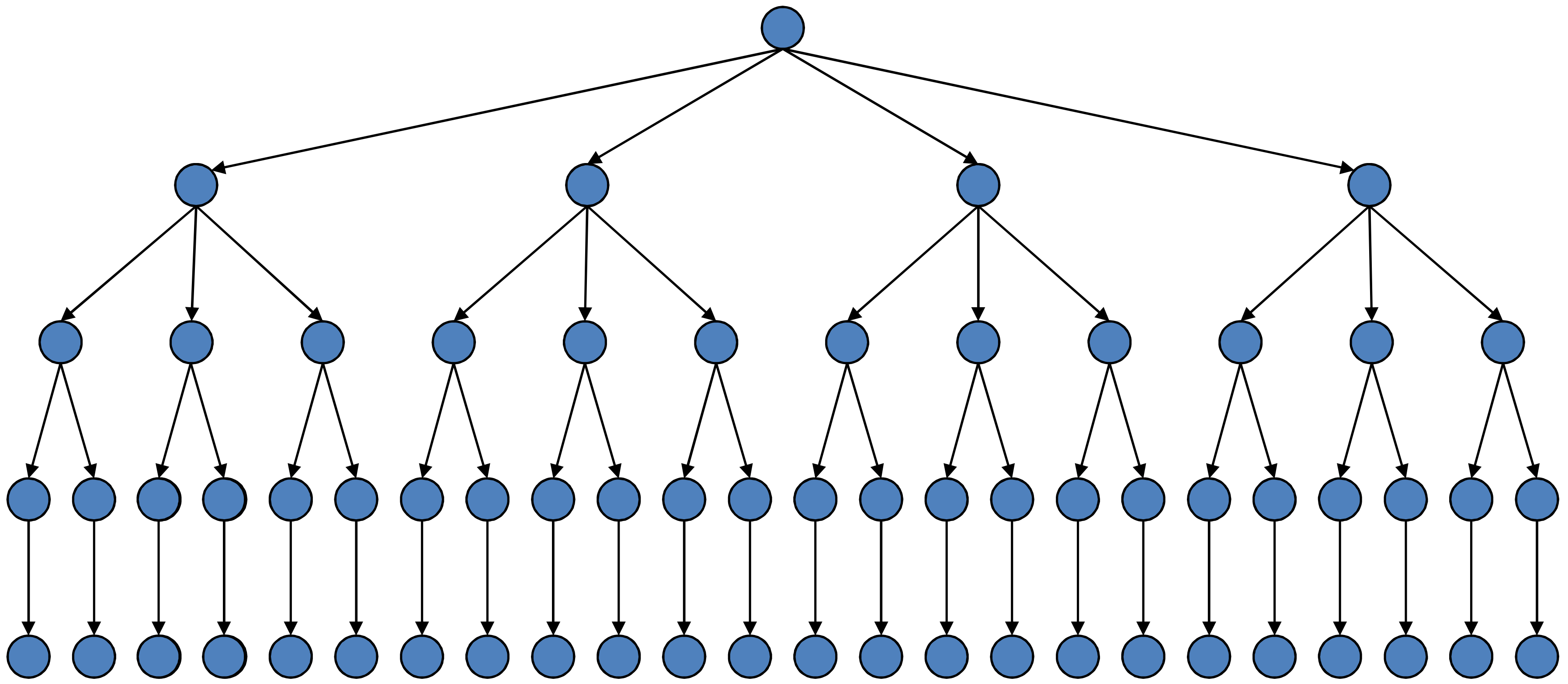
Four Queens Problem





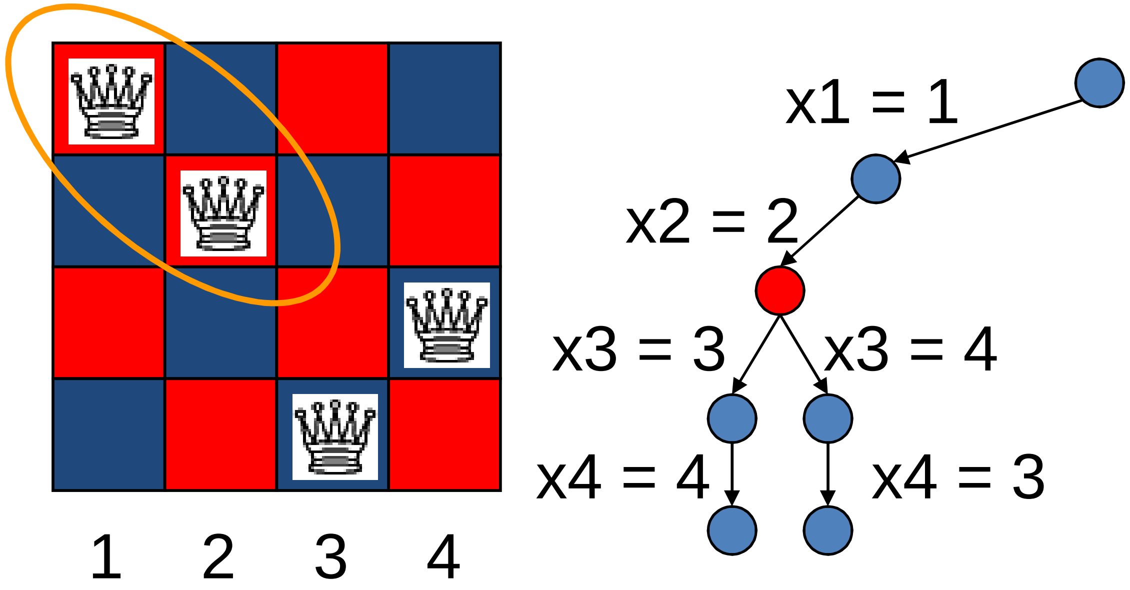
Is this leaf node a solution?

Four Queens Problem



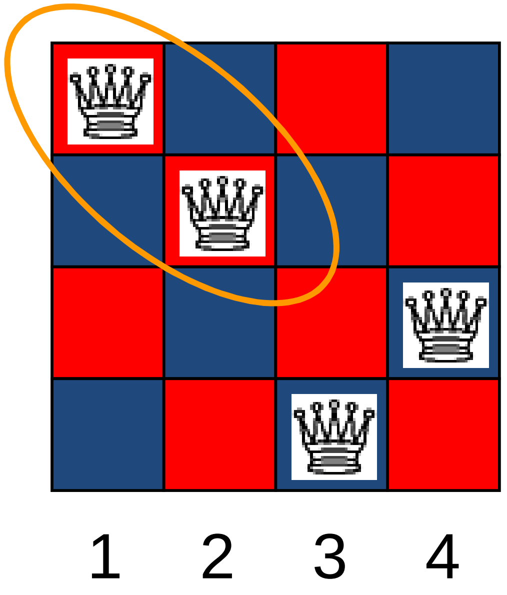
Using the criterion function, this is the search tree.

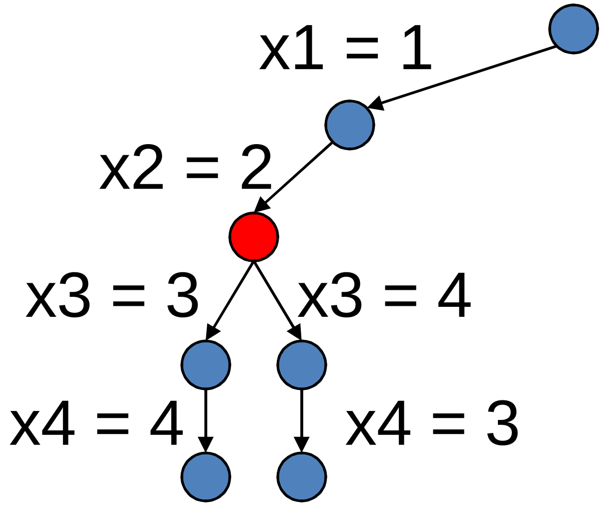
Four Queens Problem



A better question:
Is any child of this node *ever* going to be a solution?

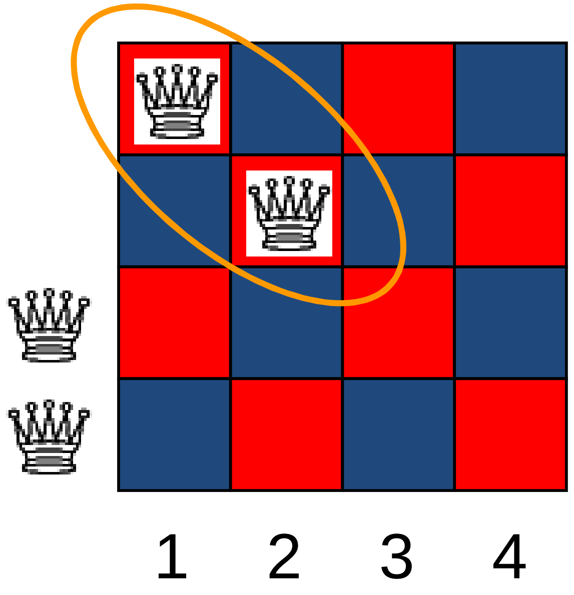
Four Queens Problem

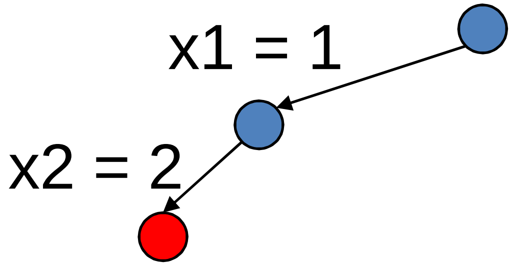




A better question:
Is any child of this node *ever* going to be a solution?
No. 1 is attacking 2. Adding queens won't fix the problem.

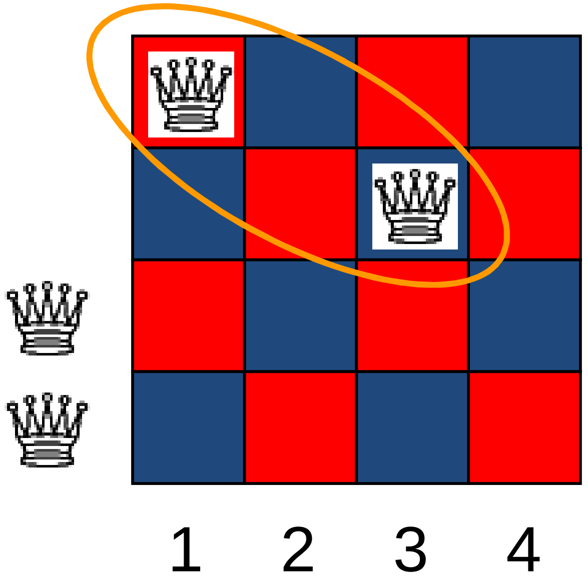
Four Queens Problem

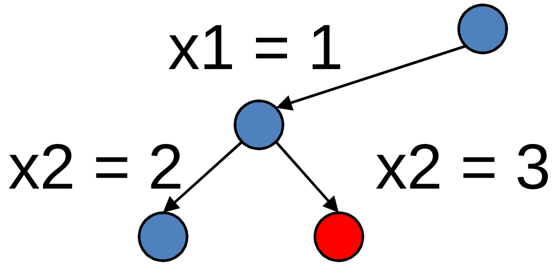




The partial criterion or feasibility function checks that a node may eventually have a solution.
Will this node have a solution? no.

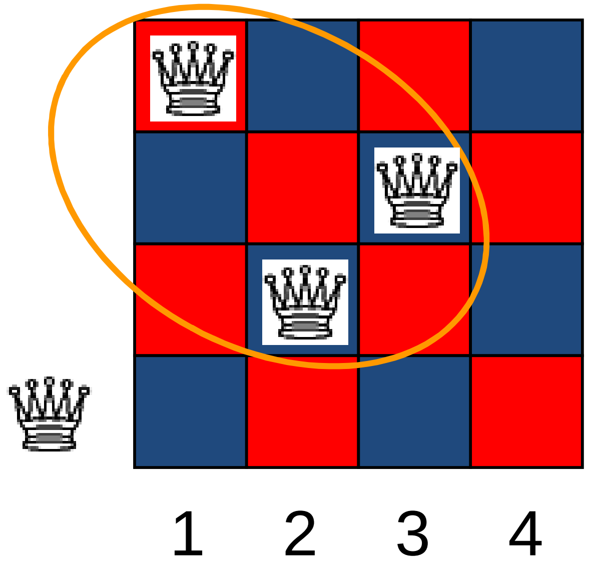
Four Queens Problem

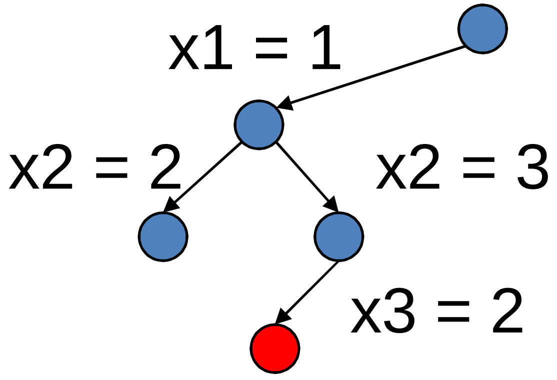




Will this node have a solution? Maybe.

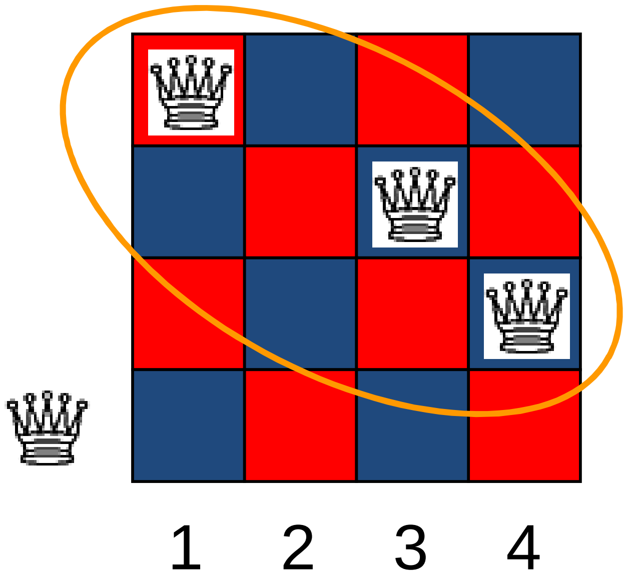
Four Queens Problem

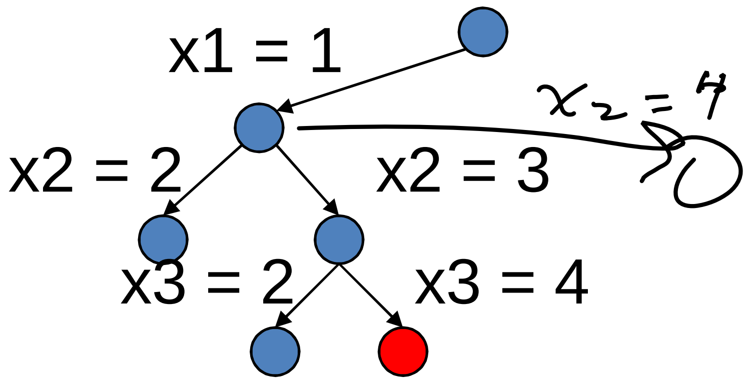




Will this node have a solution? No.
Etc.

Four Queens Problem

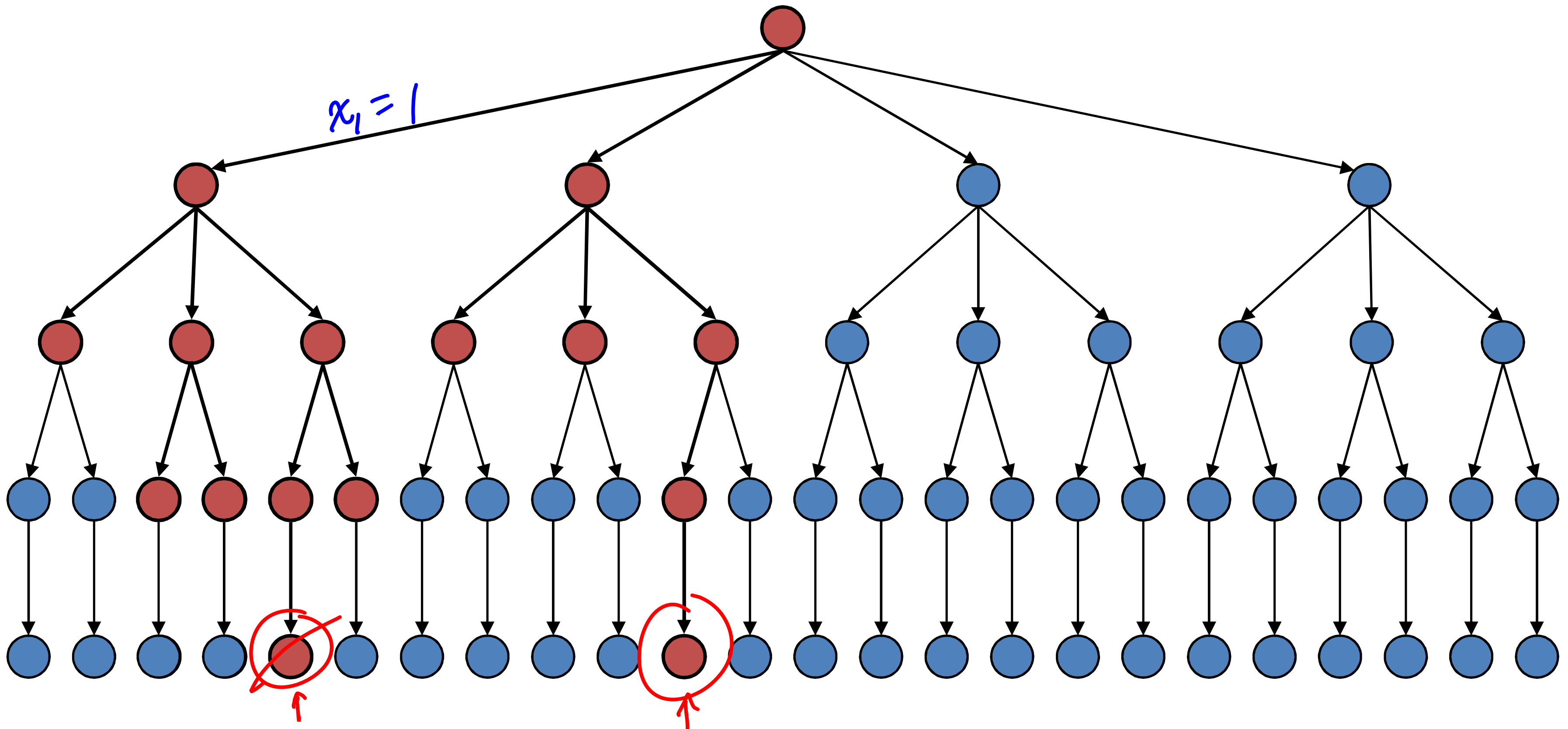




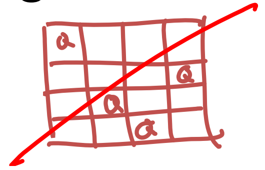
Will this node have a solution? No.

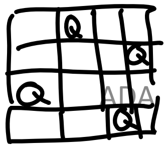
Etc.

Four Queens Problem



Using the feasibility function, this is the search tree.





Eight Queens problem

Two queens are placed at positions (i ,j) and (k ,l).

They are on the same diagonal only if

$i - j = k - l$ or $i + j = k + l$

The first equation implies

$J – l = i – k$

The second implies

$J – i = k – i$

Two queens lies on the same diagonal iff

$| j – l| = |i - k|$

Continue…………..

For instance P1=(8,1) and P2=(1,8)

So i =8, j=1 and k=1, l=8

i + j = k + l

8+1 = 1+8 =9

J – l =k- i

1- 8 = 1 - 8

Hence P1 and P2 are on the same diagonal

Eight queens problem – Place

Return true if a queen can be placed in Kth row and ith column otherwise false x[] is a global array whose first (k-1) value have been set. Abs(x) returns absolute value of r

```
1.  Algorithm Place(K , i)
2.  {
3.  For j= 1 to k-1 do
4.  If ((x[ j ] = i )                        // two in the same  column
5.  Or ( Abs ( x [ j ]- i) = Abs ( j- k)))  // same diagonal
6.   then return false
7.  Return true
8.  }
```

Computing time O(k-1)

Solution to N-Queens Problem

Using backtracking it prints all possible placements of n Queens on a n*n chessboard so that they are not attacking

```
Algorithm NQueens( K, n)
{
For i= 1 to n do
    {  if place(K ,i) then
          {   x[K] := i
If ( k = n ) then                    //obtained feasible sequence of length n
 write ( x[1:n])                   //print the sequence
Else Nqueens(K+1 ,n) //sequence is less than the length so backtrack
          }
}
}
```

Four color theorem.

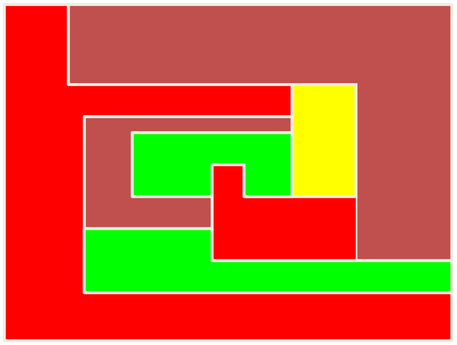
- How many colors do you need for a planar map?
  - Four.
- Haken and Appel using a computer program and 1,200 hours of run time in 1976. Checked 1,476 graphs.
- First proposed in 1852.

Full example: Map coloring

- The Four Color Theorem states that any map on a plane can be colored with no more than four colors, so that no two countries with a common border are the same color
- For most maps, finding a legal coloring is easy
- For some maps, it can be fairly difficult to find a legal coloring

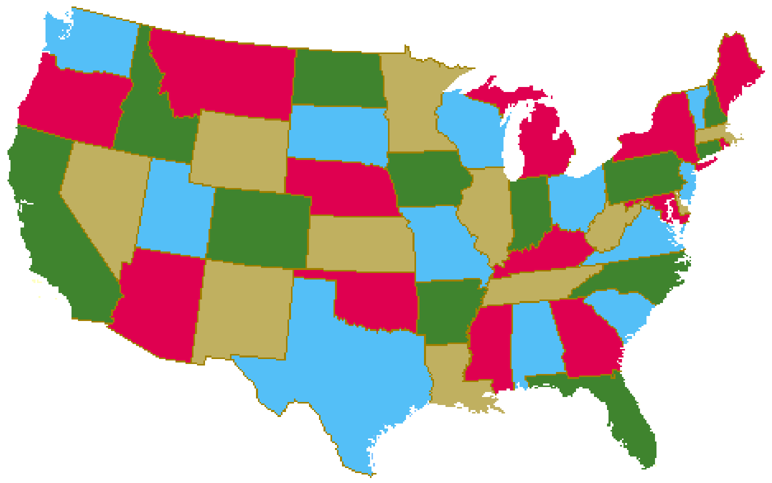
Coloring a map

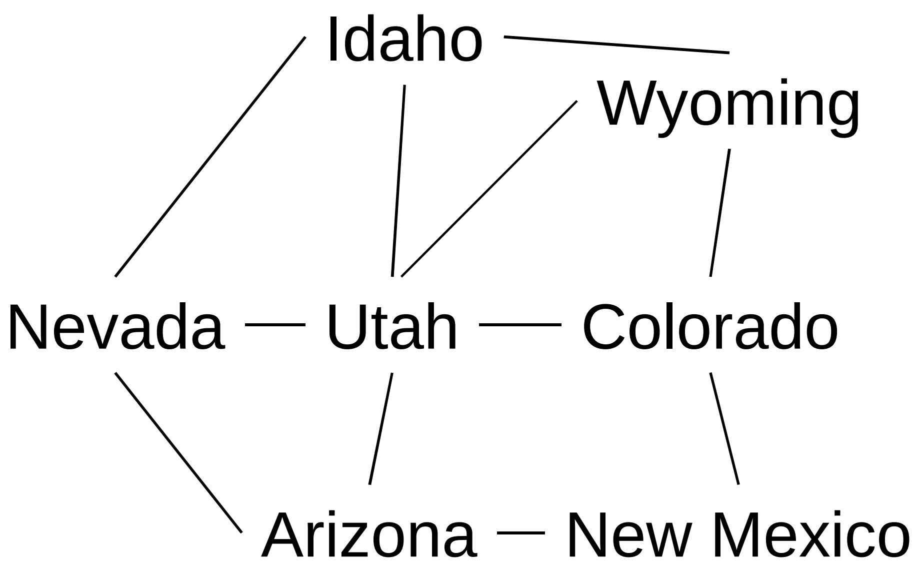
- You wish to color a map with not more than four colors
  - red, yellow, green, blue
- Adjacent countries must be in different colors



- You don't have enough information to choose colors
- Each choice leads to another set of choices
- One or more sequences of choices may (or may not) lead to a solution
- Many coloring problems can be solved with backtracking

Map coloring as Graph Coloring





Graph Coloring Problem

- Assign colors to the vertices of a graph so that no adjacent vertices share the same color
  - Vertices i, j are adjacent if there is an edge from vertex *i* to vertex *j*.
- Find all *m*-colorings of a graph
  - Find all ways to color a graph with at most *m* colors.
  - m is called chromatic number

Graph coloring
The *m*-Coloring problem

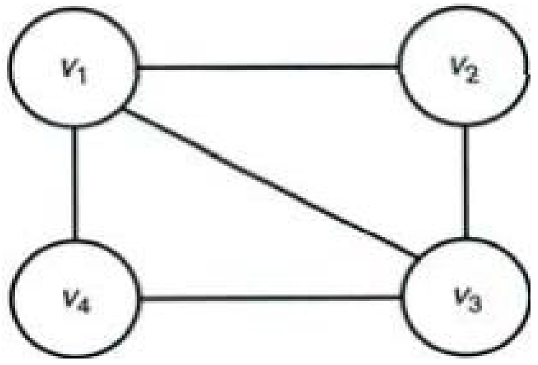
Finding all ways to color an undirected graph using at most *m* different colors, so that no two adjacent vertices are the same color.

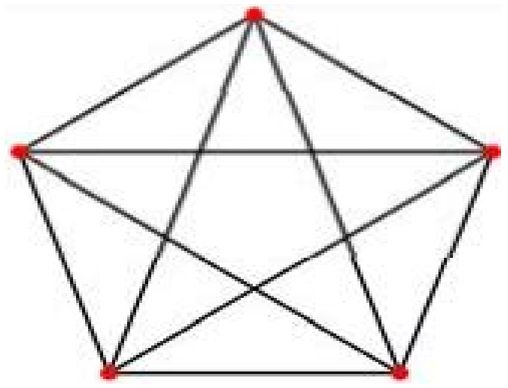
Usually the *m*-Coloring problem consider as a unique problem for each value of *m.*

Graph Coloring

Planar graph

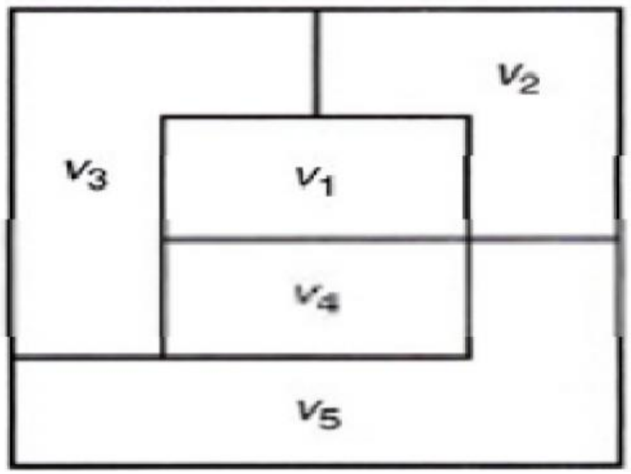
It can be drawn in a plane in such a way that no two edges cross each other.

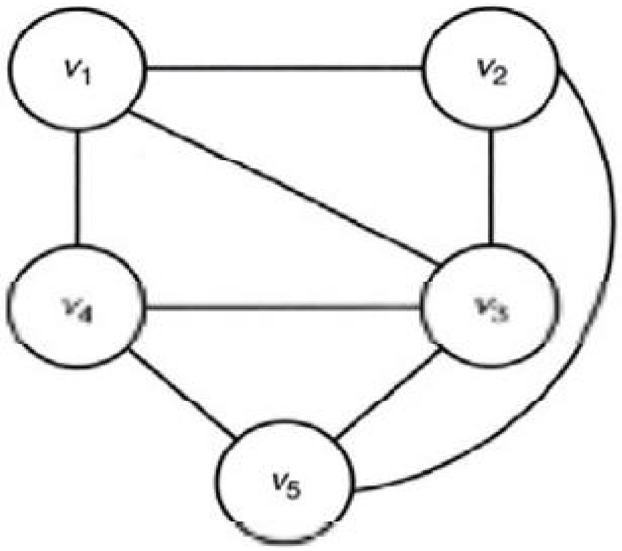




Graph Coloring

corresponded planar graph





Graph Coloring

- The top level call to *m*_coloring
- m_coloring(0)
- `The number of nodes in the state space tree for this algorithm

Nextvalue(k)

X[1]…x[k-1] have been assigned integer value in the range [1,m] such that adjacent vertices have distinct integer. A value for x[k] is determined in the range [0,m]. X[k] is assigned the next highest numbered color while maintaining distinctness from the adj. vertices of vertex k. if no such color exists, then x[k] is 0

```
Algorithm Nextvalue(k)
{   repeat
       {    X [k]:= (x[k] +1) mod (m+1) ;          //next higher color
            If ( x[k] = 0) then return;            // all colors have been used
     for j := 1 to n do
      {                   // check if this color is distinct from adjacent colors
        If ((G[k, j] != 0) and (x[k] = x[ j ] ))
                       //if (k , j)is and edge if adj. vertices have the same color.
        then break;
       }
      If (j = n+1) then return;              //new color found
     } until (false);                        //otherwise try to find another color
}
```

Graph Coloring

This algorithm was formed using recursive backtracking schema. The graph is represented by its boolean adjacency matrix G[1:n,1:n]. All assignment of 1,2..,m to the vertices of the graph such that adjacent vertices are assigned distinct integer are printed. K is the index of the next vertex to color

```
1.  Algorithm mcoloring(k)
2.  {   repeat
3.          {                           // generate all legal assignment for x[k]
4.   nextvalue(k);                      //assign to x[k] a legal value
5.  If (x[k] = 0 ) then return ; //no new color possible
7.  If ( k = n) then          // at most m color have been used to color the n vertices
8.  Write (x[1 : n];
9.  Else mcoloring(k+1)
10.         } until (false)
11. }
```

Graph coloring
mcoloring problem

- Number of internal nodes in the state space tree is

$$^{n}\Sigma_{i=1} m^{i}$$

At each node, 0(m n) time is spent by nextvalue to determine the children corresponding to legal colorings.

Total time = $^{n-1}\Sigma_{i=0} m^{i+1} n = {}^{n}\Sigma_{i=1} m^{i} n$

$$1 + m + m^2 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1}$$

$$= 0(n\ m^{n})$$

Graph Coloring

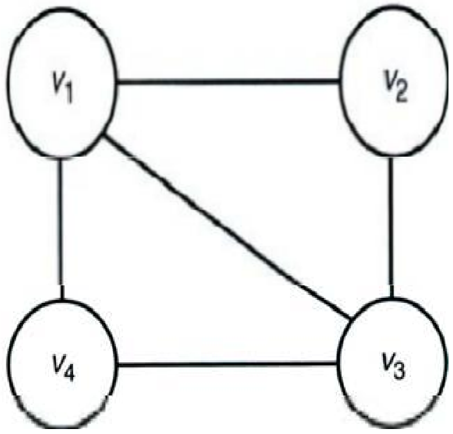
Example

2-coloring problem

No solution!

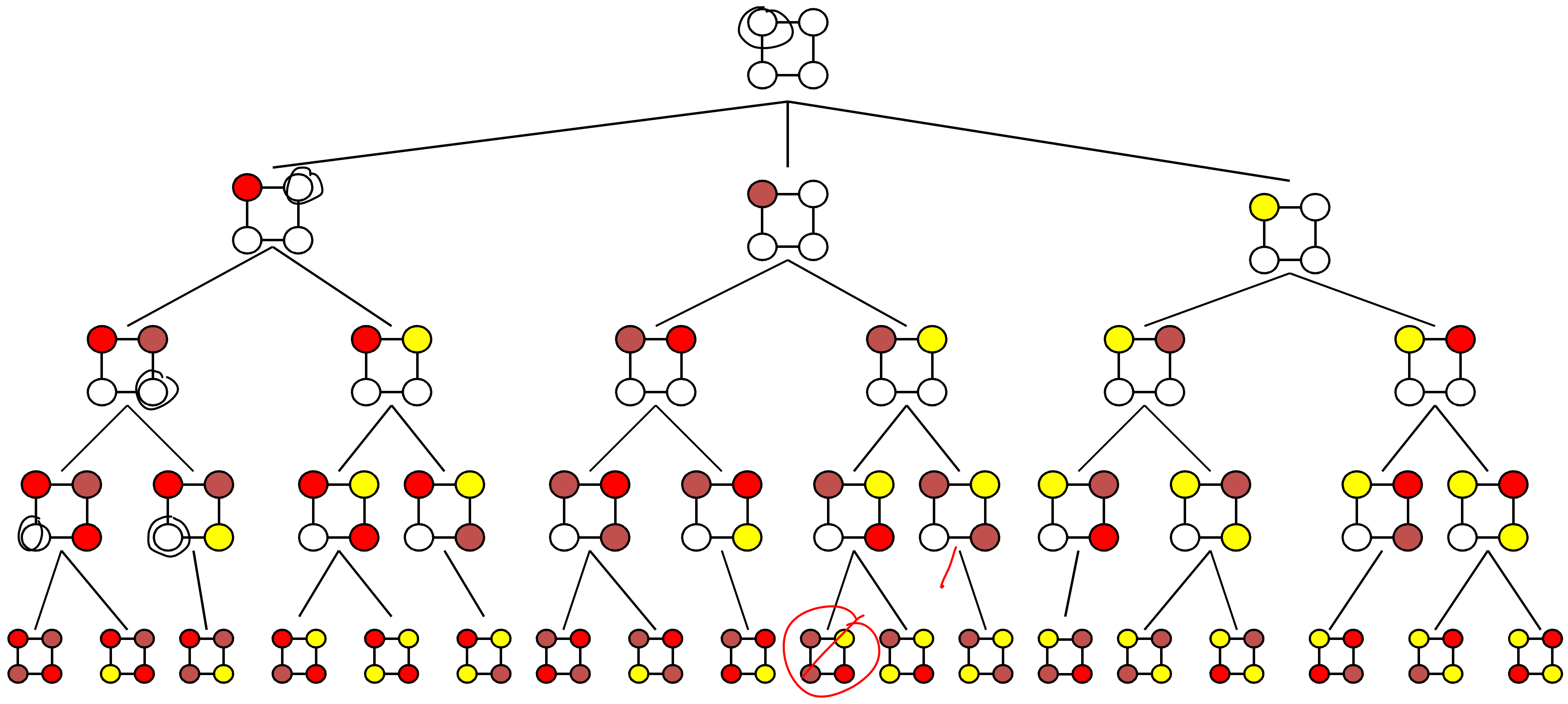
3-coloring problem

Vertex	*Color*
*v*1	color1
*v*2	color2
*v*3	color3
*v*4	color2



red
blue
yellow

Small Example



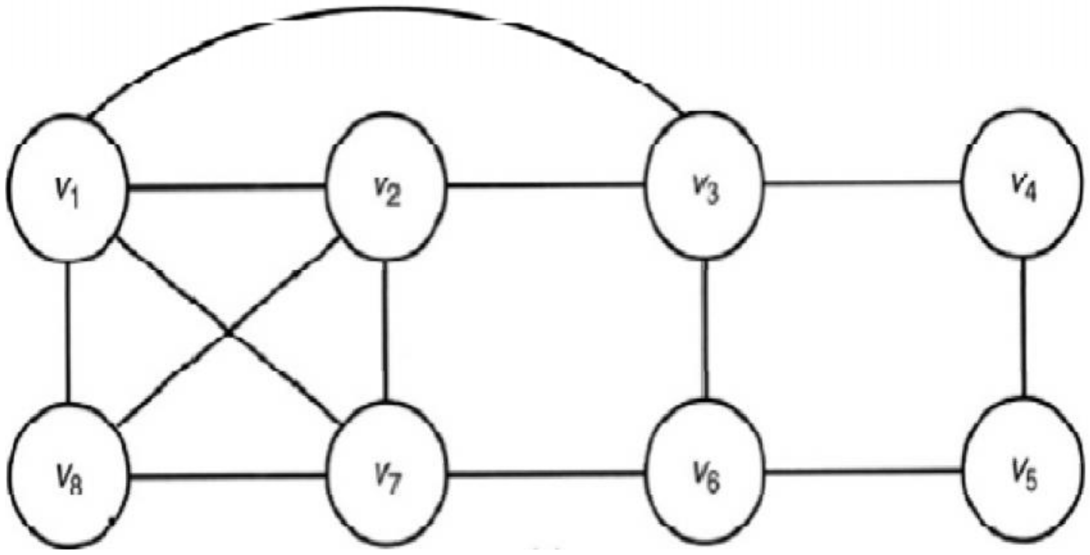
Enumerate all possible states; identify solutions at the leaves.
Naively, non-solutions will be generated. More on that later.
Note, most non-solutions not shown in this cartoon.

Hamiltonian cycle (HC) Definitions

- Hamiltonian cycle (HC): is a cycle which passes once and exactly once through every vertex of G and returns to starting position
- Hamiltonian path: is a path which passes once and exactly once through every vertex of G (G can be digraph).
- A graph is Hamiltonian iff a Hamiltonian cycle (HC) exists.

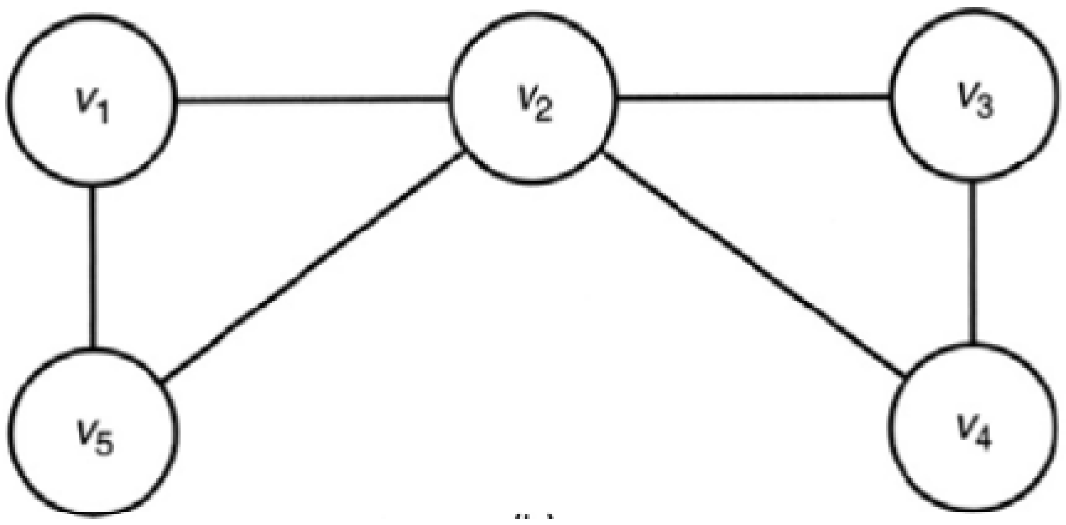
The Hamiltonian Circuits Problem

- Hamiltonian Circuit
- [v1, v2, v8, v7, v6, v5, v4, v3, v2]



The Hamiltonian Circuits Problem

- No Hamiltonian Circuit!



Backtrack Algorithm

- Search all the potential solutions
- Employ pruning of some kind to restrict the amount of researching
- Advantage:

  Find all solution, can decide HC exists or not
- Disadvantage

  Worst case, needs exponential time. Normally, take a long time

Application

- Hamiltonian cycles in fault random geometric network
- In a network, if Hamiltonian cycles exist, the fault tolerance is better.

Heuristic Algorithm

```
Initialize path P
While {
  Find new unvisited node.
  If found { Extend path P and pruning on the graph. If this
    choice does not permit HC, remove the extended node.
  } else
     Transform Path. Try all possible endpoints of this path
   Form cycle. Try to find HC
}
```

X[1: k-1] is a path o k-1 distinct vertices if x[k]=0, then no vertex has as yet been assigned to x[k]. After execution x[k] is assigned to the next highest number vertex which does not already appear in x[1, k-1] & is connected by an edge to x[k-1].otherwise x[k] =0, if k=n then in addition x[k] is connected to x[1].

```
    Algorithm Nextvalue(k)
1.    {
2.  repeat    {
3.            x[k] := (x [k] +1 mod (n +1)                    //next vertex
4.            If ( x[k] = 0) then return
5.             If G[x[k-1],  x[k] ≠ 0 ) then
6.               {                                          //Is there an edge?
7.                  for j =1 to k-1 do if (s[j] = x[k]) then break;
                                                            //check distinctness
8.                  If ( j= k) then                       //if true then vertex is distinct
9.                  If (( k< n ) or (( k =n) and G [x[n] , x[1] ≠ 0))
10.                 then return
11.              }
12.          } until ( false) ;
13.  }
```

This algorithm uses the recursive formulation of backtracking to find all the hamiltonion cycles of a graph. The graph is stored as an adjacency matrix G[1:n, 1:n]. All cycles begins at node 1.

```
Algorithm Hamiltonian(k)
{
    repeat
      {                                            //generate  values for x[k]
        Nextvalue(k);                          //assign a legal next value to x[k]
        if ( x[k] = 0 ) then return
        if (k = n) then write ( x[1:n]);
            else Hamiltonian(k + 1);
      } until(false);
}
```

Knapsack backtracking

We are given n objects and a knapsack or bag. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Given

n = number of weights

w = weights

P =profits m= knapsack capacity

Using greedy approach Choosing a subset of the weights such that

$$\text{Max} \quad \Sigma_{1\leq i \leq n} \, p_i x_i \qquad \text{subject to} \quad \Sigma_{1\leq i \leq n} \, w_i x_i \leq m$$

$$0 \leq X_i \leq 1 \qquad 1 \leq i \leq n$$

The backtracking method

- **A given** problem **has a set of constraints and possibly an objective function**
- **The** solution **must be feasible and it may optimize an objective function**
- **We can represent the** solution space **for the problem using a** state space tree
  - **The *root* of the tree represents 0 choice,**
  - **Nodes at depth 1 represent first choice**
  - **Nodes at depth 2 represent the second choice, etc.**
  - **In this tree a *path* from a root to a leaf represents a candidate solution**

Backtracking

- Definition: **We call a node *nonpromising* if it cannot lead to a feasible (or optimal) solution, otherwise it is *promising***

- Main idea: **Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is nonpromising backtracking to the node's parent**

Template for backtracking in the case of optimization problems

```
Procedure checknode (node v )
{
    node u ;
   if ( value(v) is better than best )
         best = value(v);
    if (promising (v) )
         for (each child u of v)
              checknode (u );
}
```

- *best* is the best value so far and is initialized to a value that is equal to or worse than any possible solution.
- *value*(*v*) is the value of the solution at the node.

Backtracking solution to the 0/1 knapsack problem

m is the size of knapsack; n is the number of wt. and profits. W[] and p[] are the wt. & profits. p[i]/w[i] >= p[i+1]/w[i+1]. fw is the final wt. of knapsack, fp is the final max. profit . x[k] = 0 if w [k] is not in the knapsack, else x[k]=1

```
Algorithm Bknap( k, cp, cw)
 {       If ( cw + w[k] ≤ m)     then                // generate left child then
              {      y[k]  := 1
                     If ( k< n) then Bknap(k +1, cp + p[k], cw + w[k]);
                     If (( cp + p[k] > fp ) and (k=n) then
{            fp := cp + p[k];  fw := cw + w[k];
             for j:= 1 to k do x[ j ] := y[ j ];
}  }
    If bound (cp, cw ,k) ≥  fp  ) then             //generate right child
{
         y[k] := 0; if ( k < n) then Bknap( k + 1, cp, cw);
         If (( cp > fp ) and ( k=n )) then
       {  fp:= cp; fw := cw;
         For j :=1 to k do x[ j ] := y[ j ]
}    }   }
```

Knapsack problem

It determines an upper bound on the best solution obtainable by expanding any node Z at level K+1 of the state space tree. The object weights and profits are w[i] and p[i]. It is assumed that p[i]/w[i] >= p[i+1]/w[i+1]

```
Algorithm bound( cp, cw, k)
//cp is the current profit, cw is the current wt total, k is the index of last
removed item and m is the knapsack size
{
        b := cp; c := cw;
        for i:= k+1 to n do
        {
            c:= c + w[ i ];
            if ( c < m) then b:= b + p[ i ] ;
            else return b+( 1- (c-m)/ w[ i ]*p[ i ];
        }
        return b;
}
```

Example

- Suppose n = 4, W = 16, and we have the following:

i	p_i	w_i	p_i / w_i
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

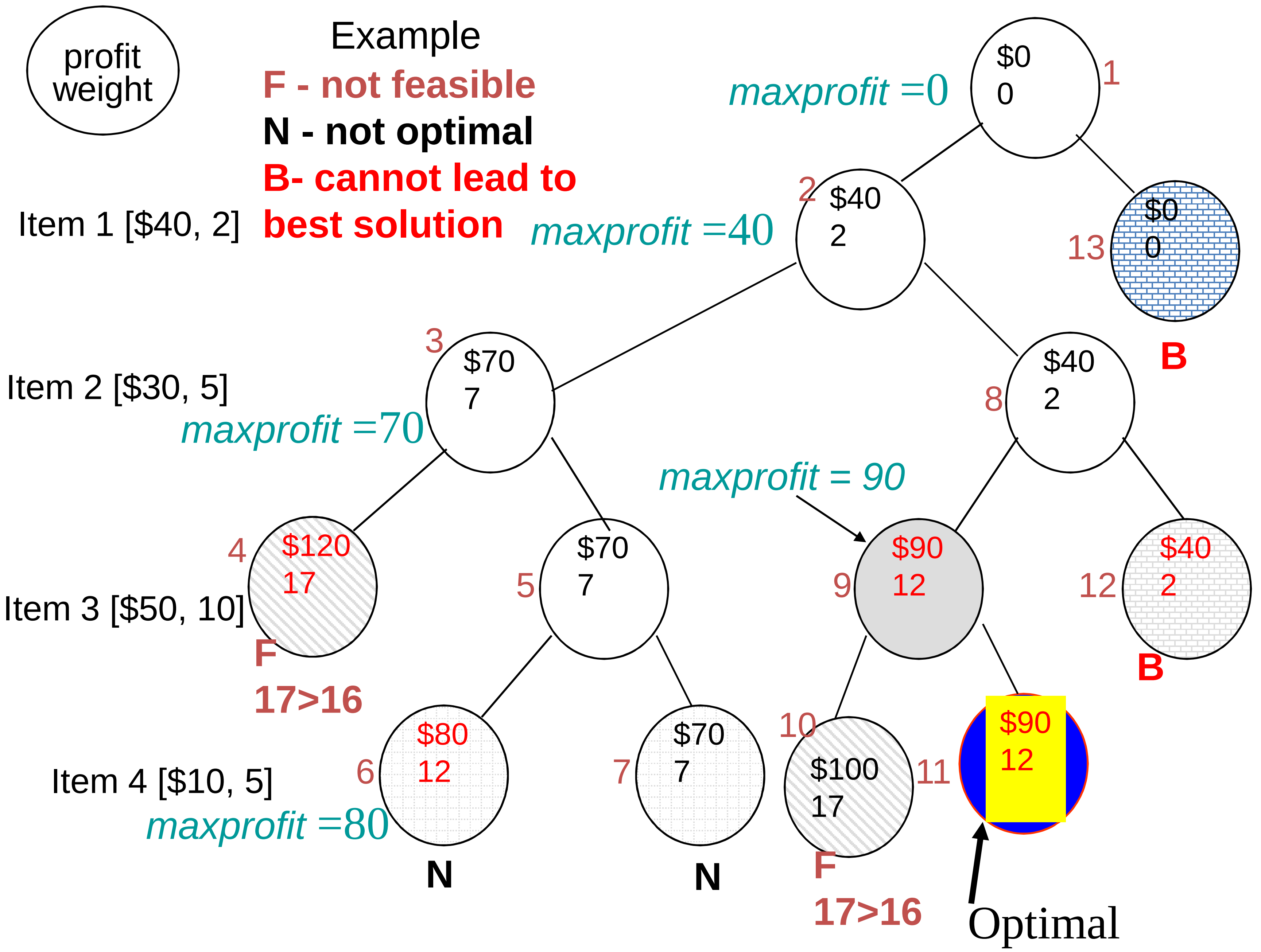
- Note the the items are in the *correct order* needed by *KWF*

Example

F - not feasible

N - not optimal

B- cannot lead to best solution



Worst-case time complexity

Check number of nodes:

$$1+2+2^2+2^3+\ldots+2^n=2^{n+1}-1$$

Time complexity:

$$\theta(2^n)$$

When will it happen?

For a given n, W=n

P_i = 1, w_i=1 (for 1 <=i<=n-1)

P_n=n w_n=n

Branch and Bound

Introduction

- In backtracking, we used depth-first search with pruning to traverse the (virtual) state space. We can achieve better performance for many problems using a breadth-first search with pruning. This approach is known as branch-and-bound
- Branch and Bound is a general search method.
- Starting by considering the root problem (the original problem with the complete feasible region), the lower-bounding and upper-bounding procedures are applied to the root problem.
- If the bounds match, then an optimal solution has been found and the procedure terminates.

Branch and Bound

Introduction

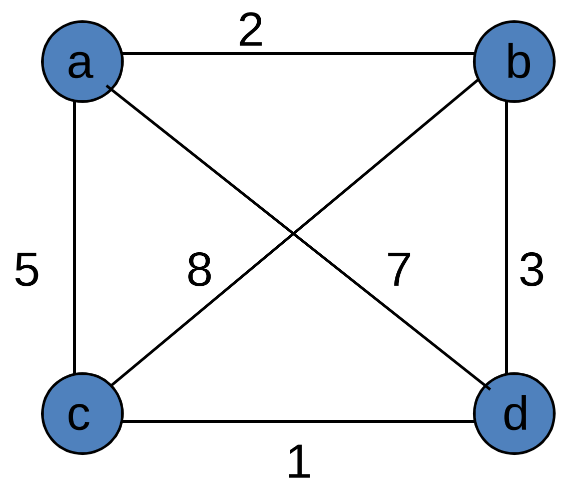
- Otherwise, the feasible region is divided into two or more regions, these subproblems partition the feasible region.
- The algorithm is applied recursively to the subproblems. If an optimal solution is found to a sub problem, it is a feasible solution to the full problem, but not necessarily globally optimal.

Branch and Bound
Introduction

- If the lower bound for a node exceeds the best known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node. Therefore, the node can be removed from consideration.
- The search proceeds until all nodes have been solved or pruned, or until some specified threshold is met between the best solution found and the lower bounds on all unsolved subproblems.

Branch and Bound

Traveling salesman problem

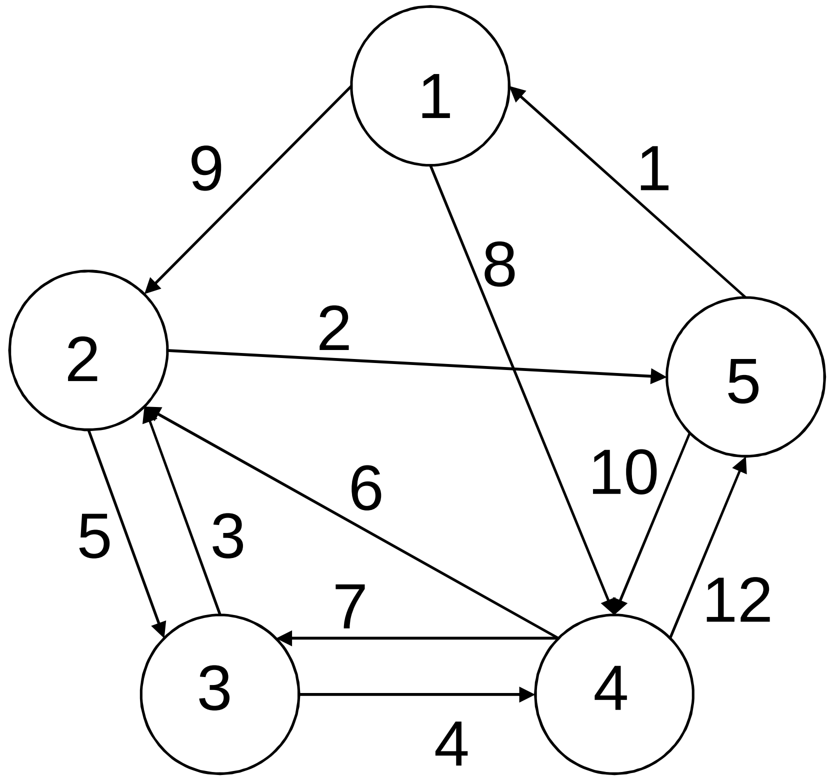


{ a, b, c, d } represents 4 cities

The weights represent distances between cities

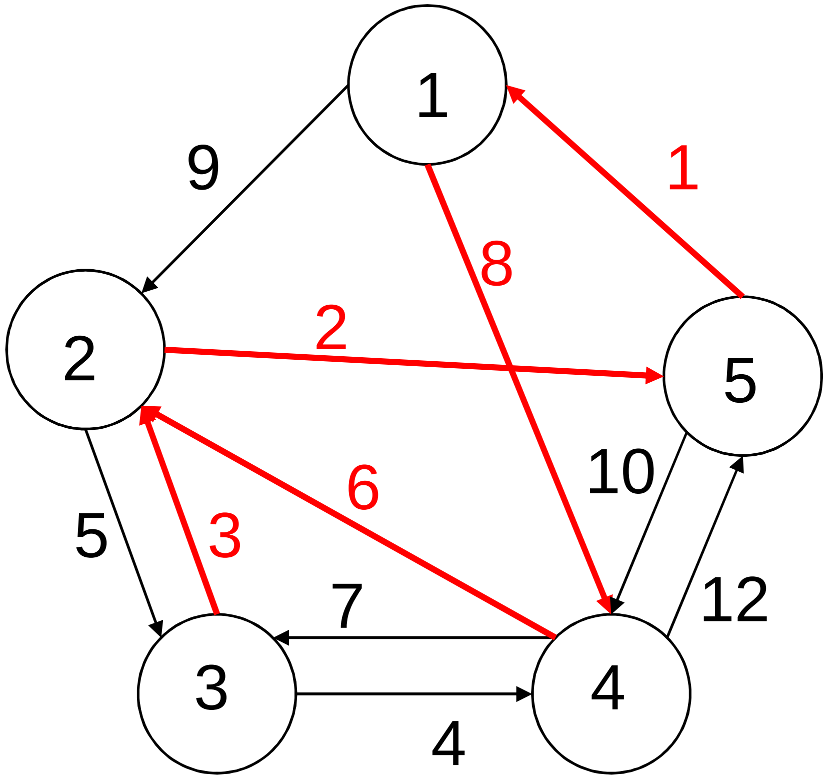
Problem: find the shortest path from a city (say a), visit all other cities exactly once, and return to the city where it started (city a).

Bound on TSP Tour



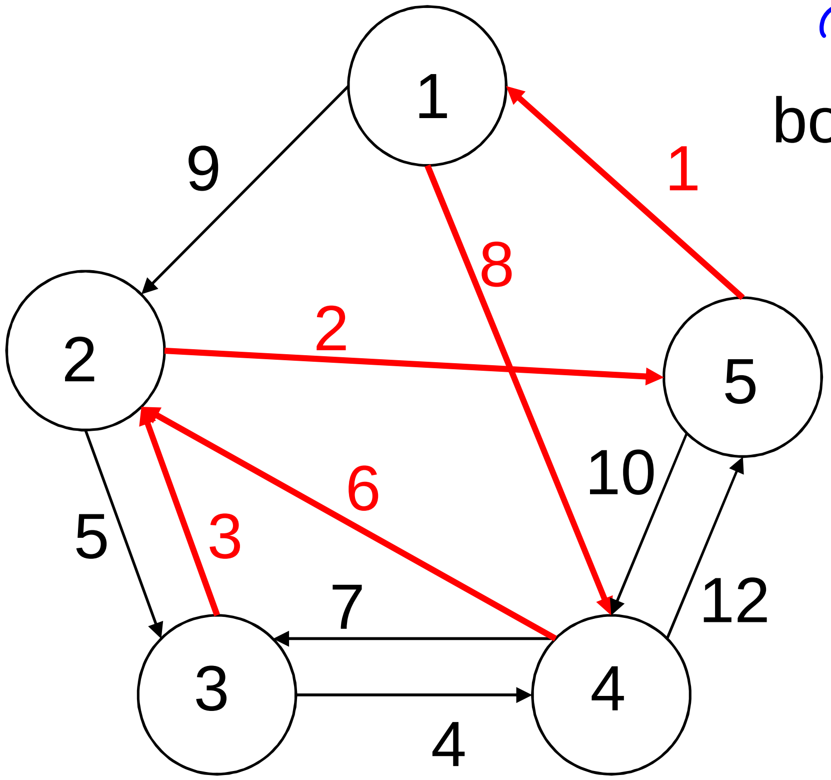
Every tour must leave every vertex and
and arrive at every vertex.

Bound on TSP Tour



What’s the cheapest way to leave each vertex?

Bound on TSP Tour

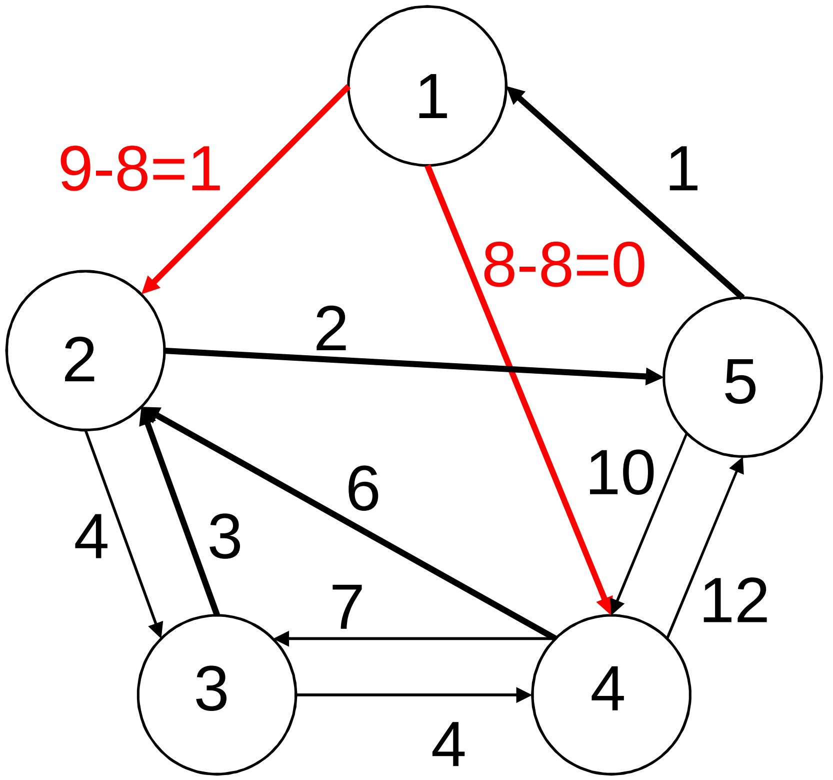


rough draft

bound=8+6+3+2+1
=20

Save the sum of those costs in the bound (as a rough draft).
Can we find a tighter lower bound?

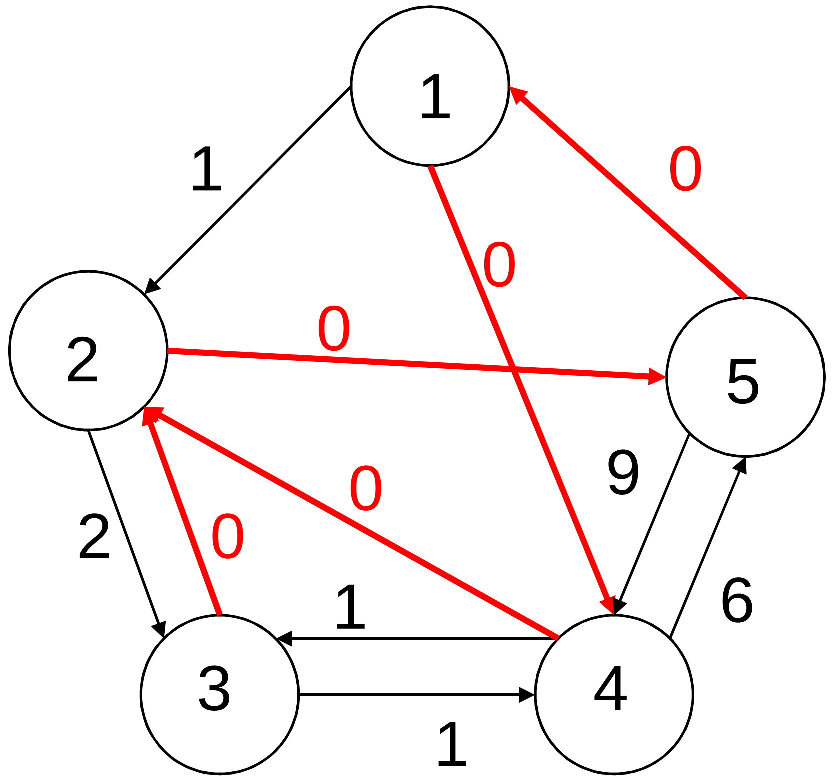
Bound on TSP Tour



bound=20

For a given vertex, subtract the least cost departure from each edge leaving that vertex.

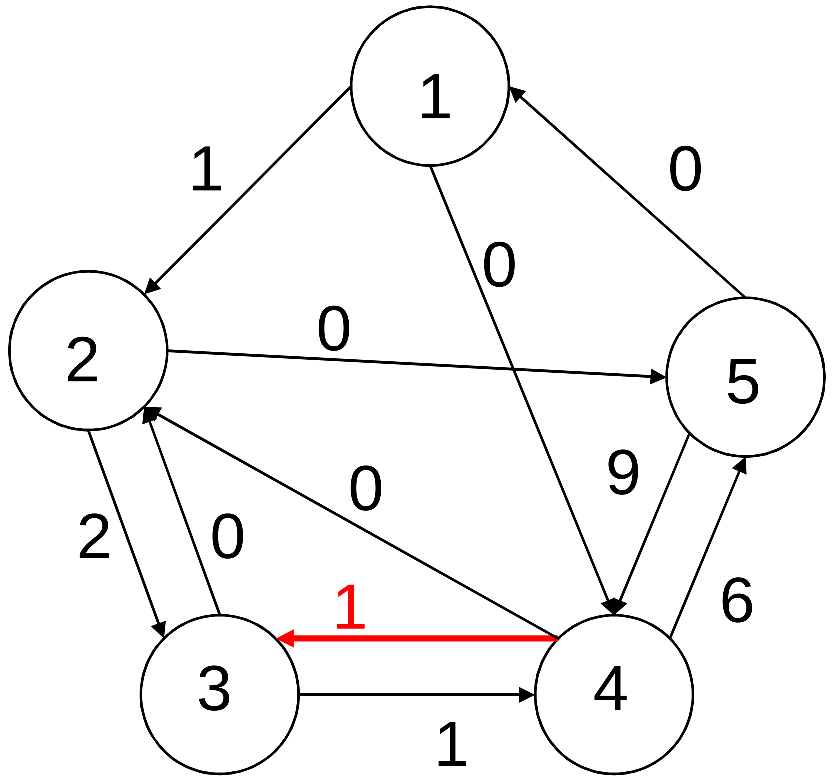
Bound on TSP Tour



bound=20

Repeat for the other vertices.

Bound on TSP Tour

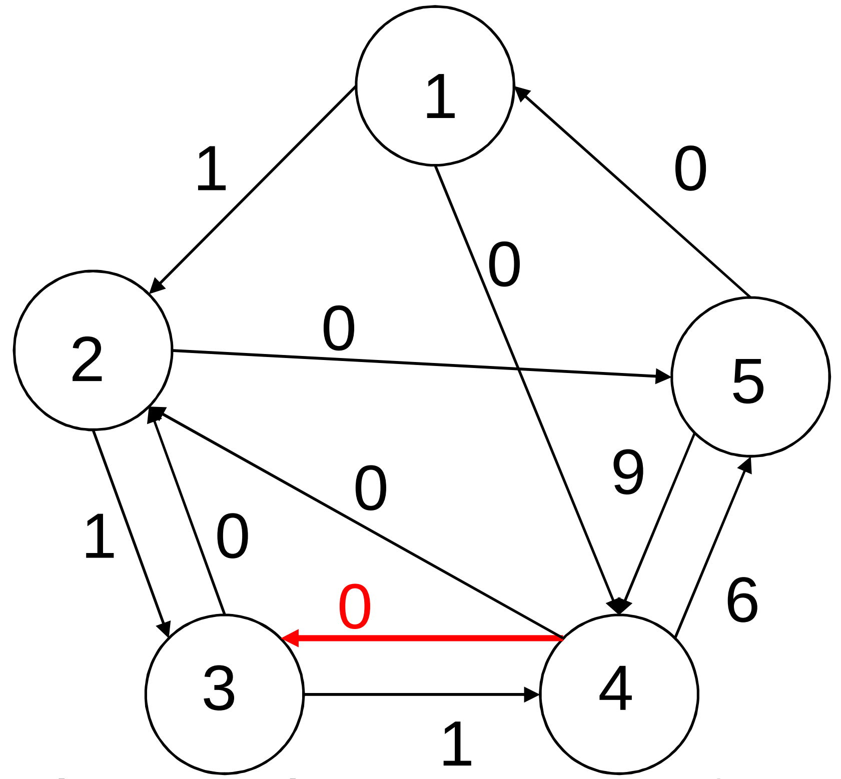


bound=20

Does that set of edges now having 0 residual cost arrive at every vertex?

In this case, the edges never arrive at vertex 3.

Bound on TSP Tour



bound=21

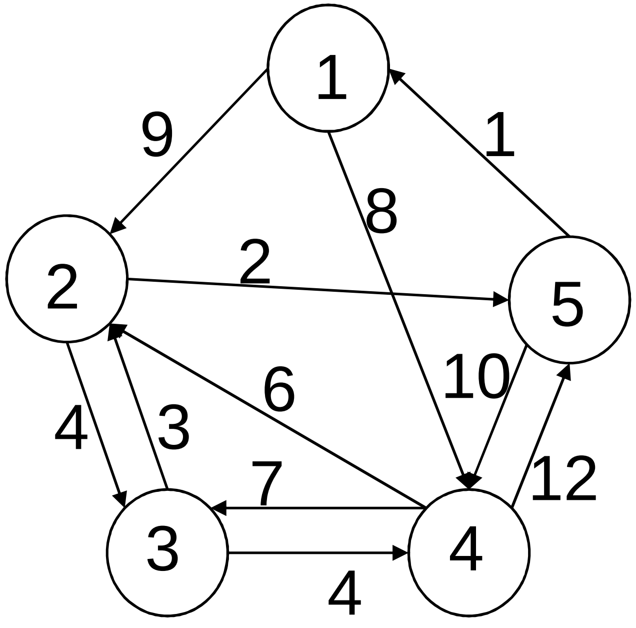
We have to take an edge to vertex 3 from somewhere. Assume we take the cheapest. Subtract its cost from other edges entering vertex 3 and add the cost to the bound.

We have just tightened the bound.

The Bound

- It will cost at least this much to visit all the vertices in the graph.
  - there’s no cheaper way to get in and out of each vertex.
  - the edges are now labeled with the *extra* cost of choosing another edge.

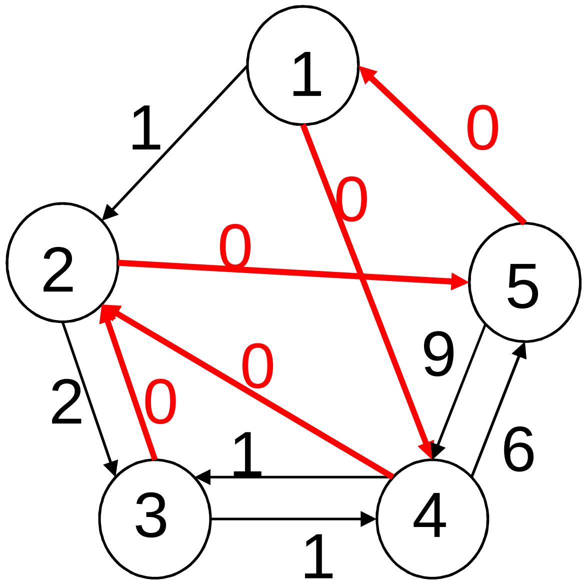
Bound on TSP Tour



$$\begin{pmatrix} 999 & 9 & 999 & 8 & 999 \\ 999 & 999 & 4 & 999 & 2 \\ 999 & 3 & 999 & 4 & 999 \\ 999 & 6 & 7 & 999 & 12 \\ 1 & 999 & 999 & 10 & 999 \end{pmatrix}$$

Algorithms do this using a cost matrix.

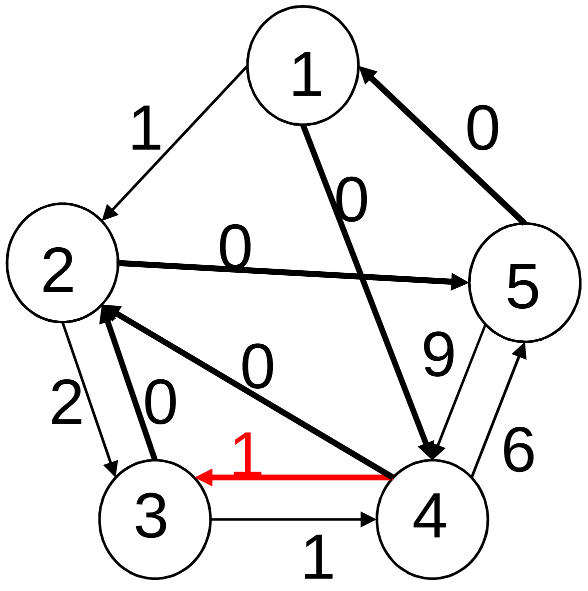
Bound on TSP Tour



$$\left(\begin{array}{ccccc} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 2 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & \textcircled{1} & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{array}\right) \begin{array}{c} 8 \\ 2 \\ 3 \\ 6 \\ 1 \\ \hline 20 \end{array}$$

Reduce all rows.

Bound on TSP Tour



$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound: 20 + 1 = 21

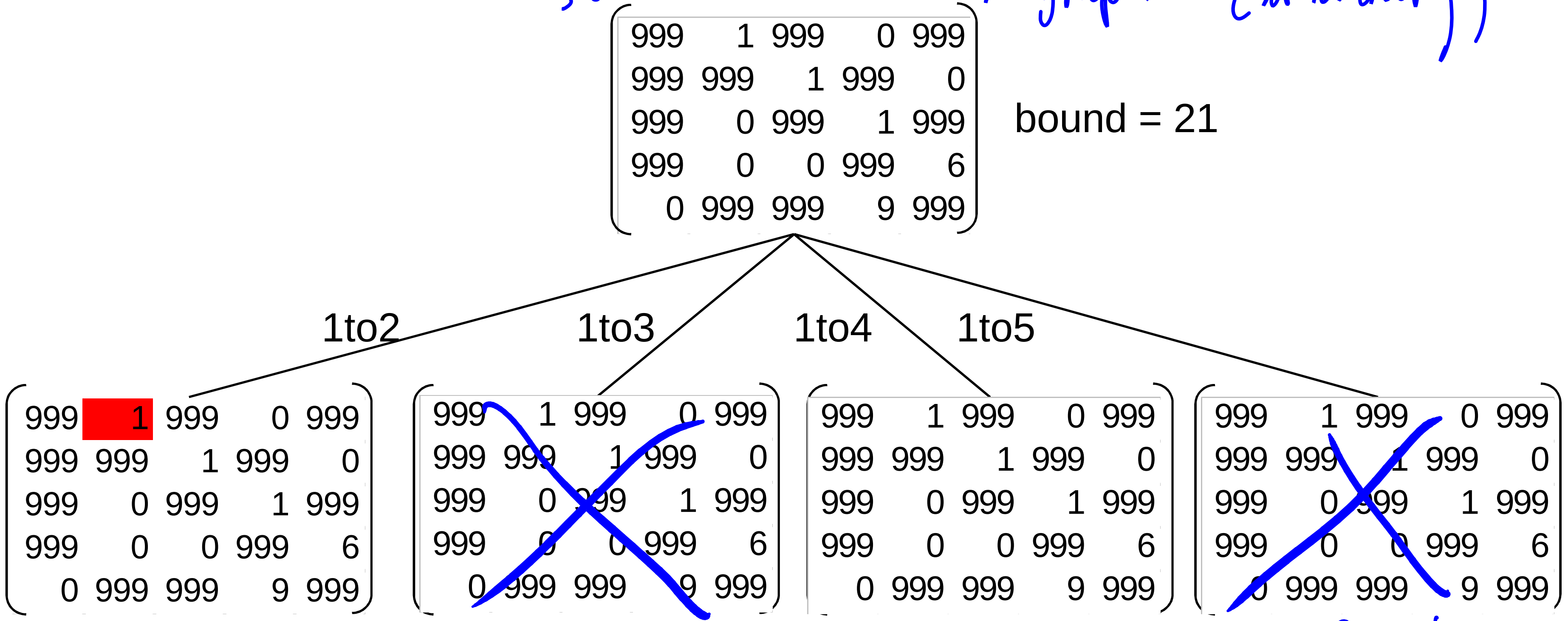
Then reduce column #3. Now we have a tight bound.

Using this bound for TSP in B&B

start at ~~node~~ vertex 1 in graph (arbitrary)

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21



1to2:

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21+1

1to3:

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

infeasible

1to4:

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21

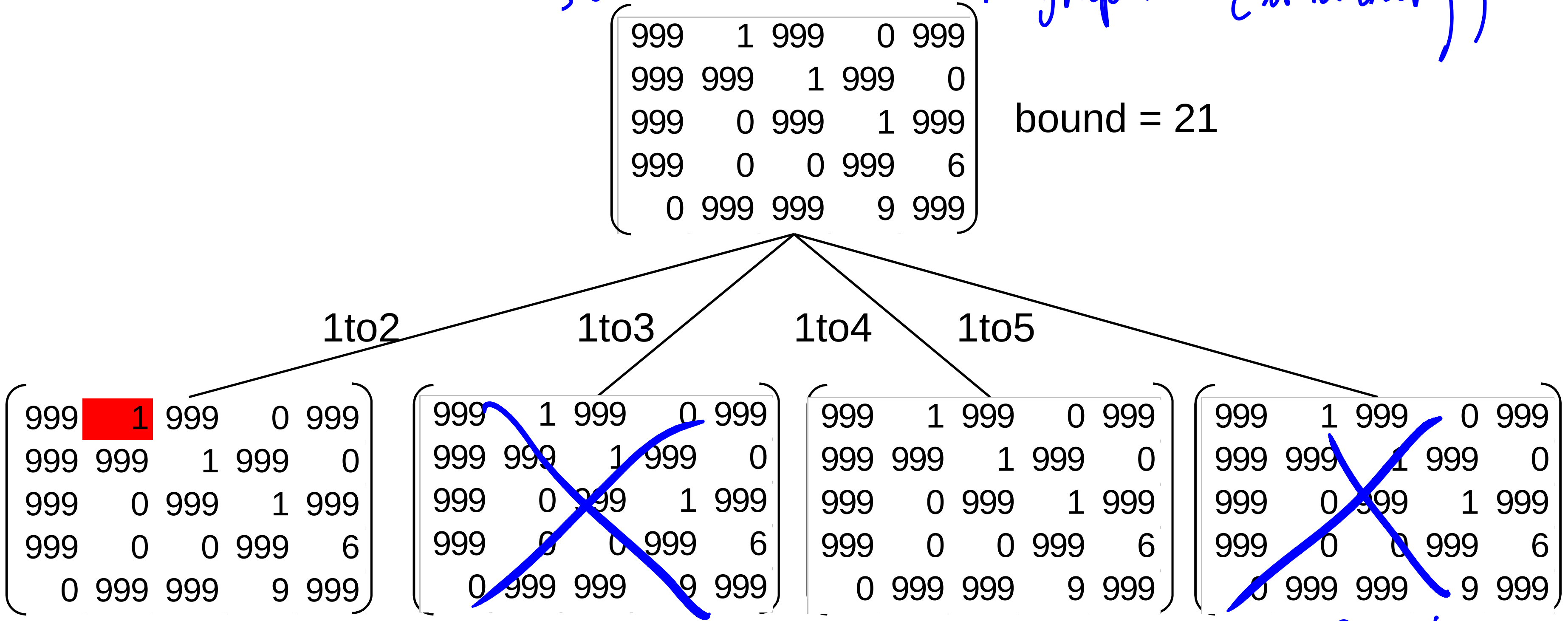
1to5:

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

infeasible

Using this bound for TSP in B&B

start at ~~node~~ vertex 1 in graph (arbitrary)



$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21

1to2

$$\begin{pmatrix} 999 & \boxed{1} & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21+1

1to3

$$\cancel{\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}}$$

infeasible

1to4

$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21

1to5

$$\cancel{\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}}$$

infeasible

Focus: going from 1 to 2

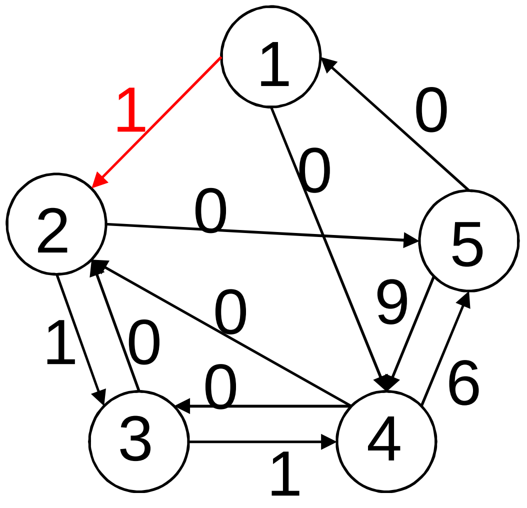
$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

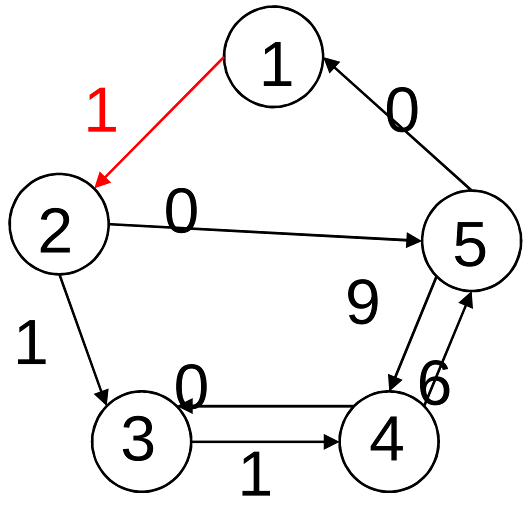
bound = 21

1to2

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 999 & 999 & 1 & 999 \\ 999 & 999 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21+1





Add extra cost from 1 to 2, exclude edges from 1 or into 2.

Focus: going from 1 to 2

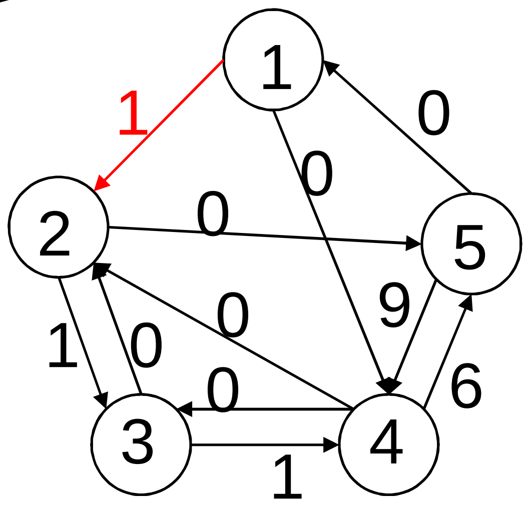
$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

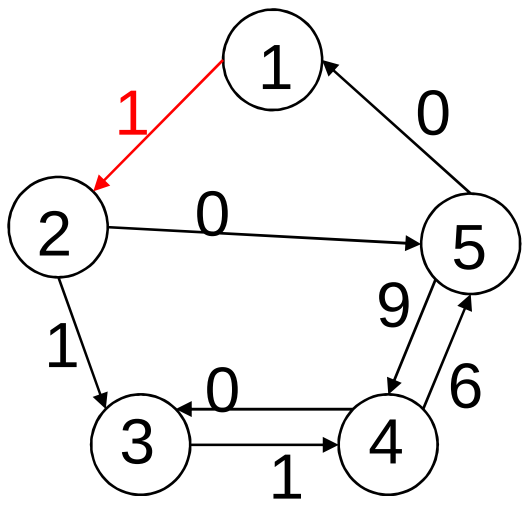
bound = 21

1to2

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 999 & 999 & 1 & 999 \\ 999 & 999 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21+1+1





No edges into vertex 4 w/ 0 reduced cost.

Focus: going from 1 to 2

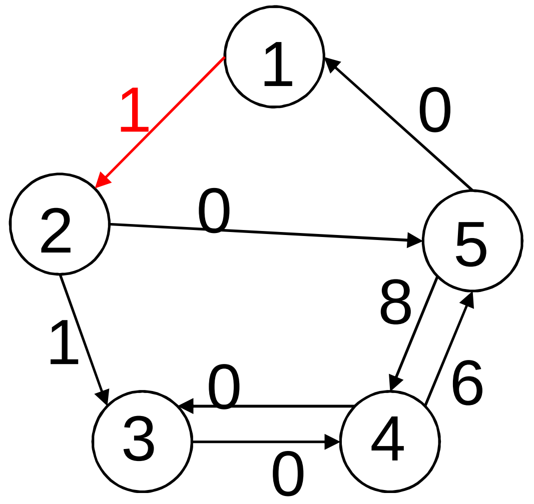
$$\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}$$

bound = 21

1to2

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 999 & 999 & 0 & 999 \\ 999 & 999 & 0 & 999 & 6 \\ 0 & 999 & 999 & 8 & 999 \end{pmatrix}$$

bound = 21+1+1 = 23



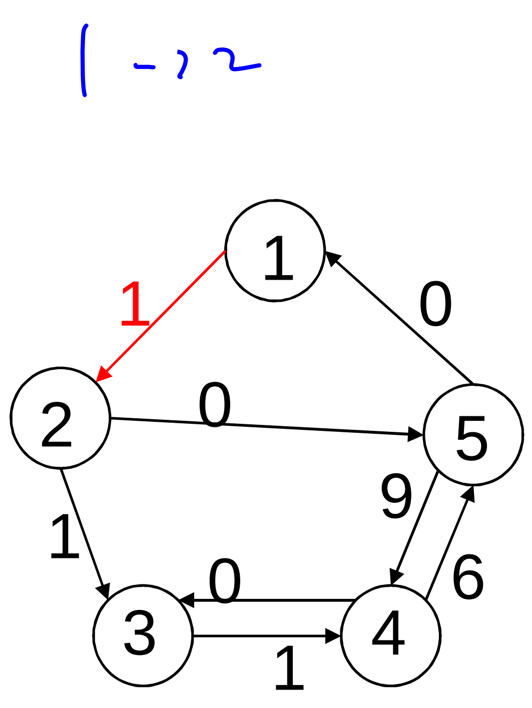
Add cost of reducing edge into vertex 4.

Bounds for other choices.

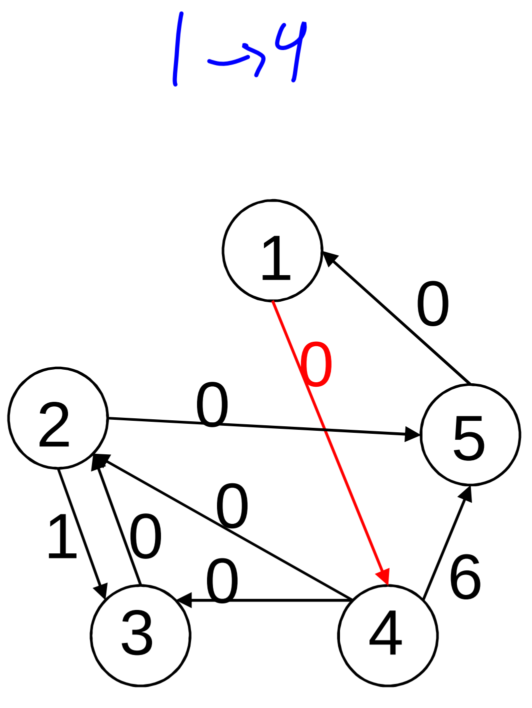
1to2(23),1to4(21)

$$\begin{tikzcd}
 & \begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix} \text{ bound = 21} \arrow[dl, dash, "1to2"'] \arrow[d, dash, "1to3"'] \arrow[dr, dash, "1to4"] \arrow[drr, dash, "1to5"] & & \\
\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 999 & 999 & 0 & 999 \\ 999 & 999 & 0 & 999 & 6 \\ 0 & 999 & 999 & 8 & 999 \end{pmatrix} & \cancel{\begin{pmatrix} 999 & 999 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}} & \begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 999 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix} & \cancel{\begin{pmatrix} 999 & 1 & 999 & 0 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 1 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 9 & 999 \end{pmatrix}} \\
\text{bound = 23} & \text{bound = 999} & \text{bound = 21} & \text{bound = 999}
\end{tikzcd}$$

Leaves us with Two Possibilities on Priority Queue



bound = 23



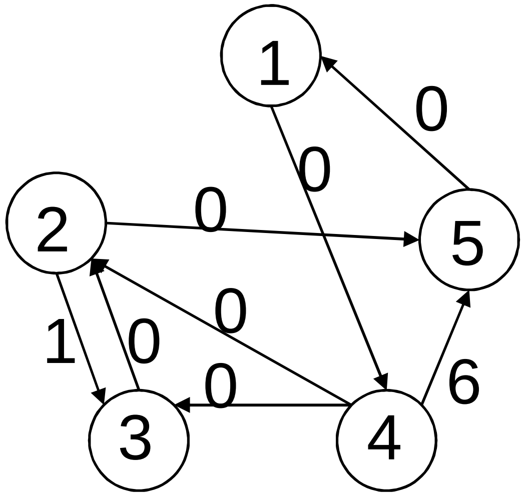
bound = 21

Leaving Vertex 4

4to2(22), 4to3(21)
4to5(28),1to2(23),

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 1 & 999 & 0 \\ 999 & 0 & 999 & 999 & 999 \\ 999 & 0 & 0 & 999 & 6 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 21



4to2 4to3 4to5

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 0 & 999 & 0 \\ 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 22

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 0 \\ 999 & 0 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 21

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 0 & 999 & 999 \\ 999 & 0 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 28

Leaving Vertex 3

4to2(22), 3to2(21)
1to2(23),

$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 0 \\ 999 & 0 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 21

3to2

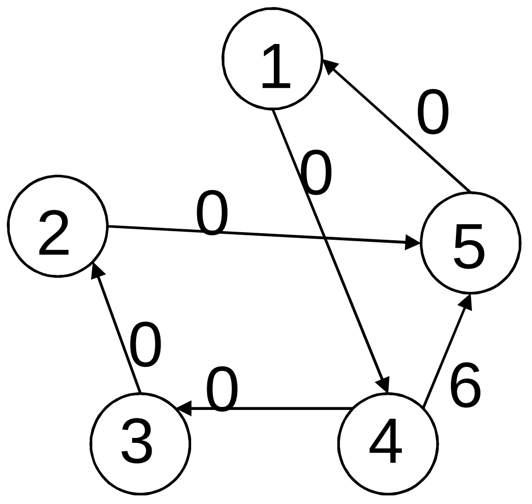
$$\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 0 \\ 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}$$

bound = 21

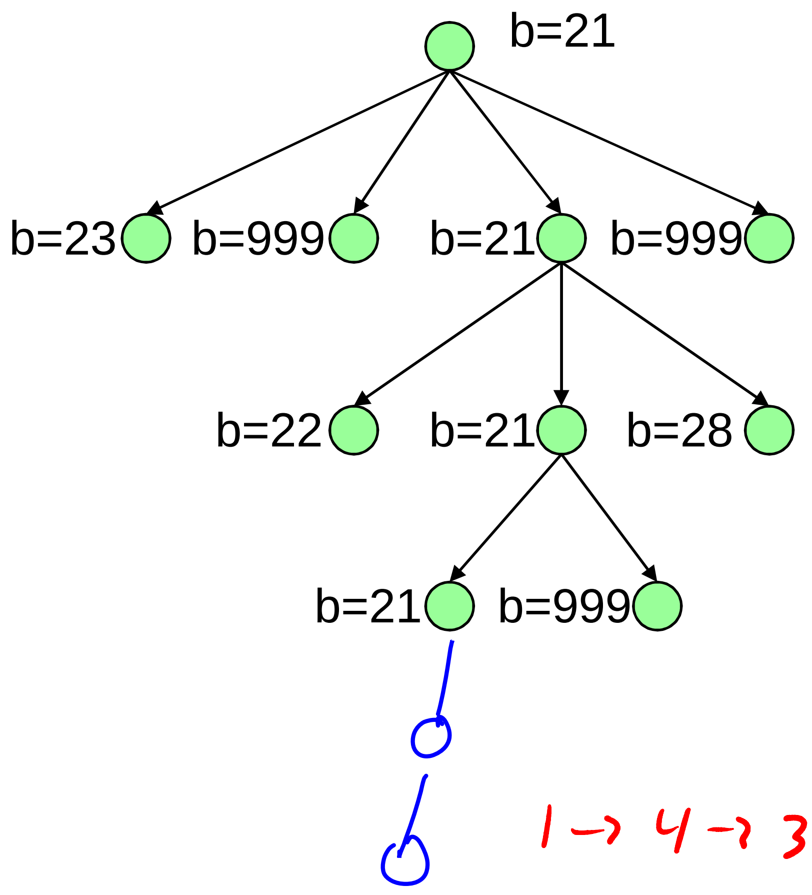
3to5

$$\cancel{\begin{pmatrix} 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 999 & 999 & 999 & 999 & 999 \\ 0 & 999 & 999 & 999 & 999 \end{pmatrix}}$$

bound = 999



Search Tree for This Problem



$1 \to 4 \to 3 \to 2 \to 5 \to 1$

Conclusion

- Although a number of algorithms have been proposed for the integer linear programming problem, the *branch-and-bound* technique has proven to be reasonably efficient on practical problems, and it has the added advantage that it solves continuous linear programs as sub problems.

- The technique is also used in a lot of software in global optimization.

Lower and Upper Bound Theory

- **How fast can we sort?**
- Most of the sorting algorithms are **comparison sorts**: only use comparisons to determine the relative order of elements.
- **Examples**: insertion sort, merge sort, quicksort, heapsort.
- The best worst-case running time that we've seen for comparison sorting is O(n lg n) .

- **Is O(n lg n) the best we can do?**
- **Lower-Bound Theory** can help us answer this question

Lower and Upper Bound Theory

- *Lower Bound*, L(n), is a property of the specific problem, i.e. sorting problem, MST, matrix multiplication, not of any particular algorithm solving that problem.

- *Lower bound theory* says that no algorithm can do the job in fewer than L(n) time units for arbitrary inputs, i.e., that every comparison-based sorting algorithm must take at least L(n) time in the worst case.

- L(n) is the minimum over all possible algorithms, of the maximum complexity.

Lower and Upper Bound Theory

- *Upper bound theory* says that for any arbitrary inputs, we can always sort in time at most U(n). How long it would take to solve a problem using one of the known Algorithms with worst-case input gives us a *upper bound*.
- Improving an *upper bound* means finding an algorithm with better worst-case performance.
- U(n) is the minimum over all known algorithms, of the maximum complexity.
- Both upper and lower bounds are minima over the maximum complexity of inputs of size n.
- The ultimate goal is to make these two functions coincide. When this is done, the optimal algorithm will have L(n) = U(n).

Lower and Upper Bound Theory

There are few techniques for finding lower bounds

1) Trivial Lower Bounds: For many problems it is possible to easily observe that a lower bound identical to n exists, where n is the number of inputs (or possibly outputs) to the problem.

- The method consists of simply counting the number of inputs that must be examined and the number of outputs that must be produced, and note that any algorithm must, at least, read its inputs and write its outputs.

Lower and Upper Bound Theory

Example-2: Finding maximum of unordered array requires examining each input so it is (n).

A simple counting arguments shows that any comparison-based algorithm for finding the maximum value of an element in a list of size n must perform at least n-1 comparisons for any input

Lower and Upper Bound Theory

2) **Information Theory**: The information theory method establishing lower bounds by computing the limitations on information gained by a basic operation and then showing how much information is required before a given problem is solved.

- This is used to show that any possible algorithm for solving a problem must do some minimal amount of work.
- The most useful principle of this kind is that the outcome of a comparison between two items contains one bit of information.

Lower and Upper Bound Theory

Example-1: For the problem of *searching an ordered list* with n Elements for the position of a particular item,

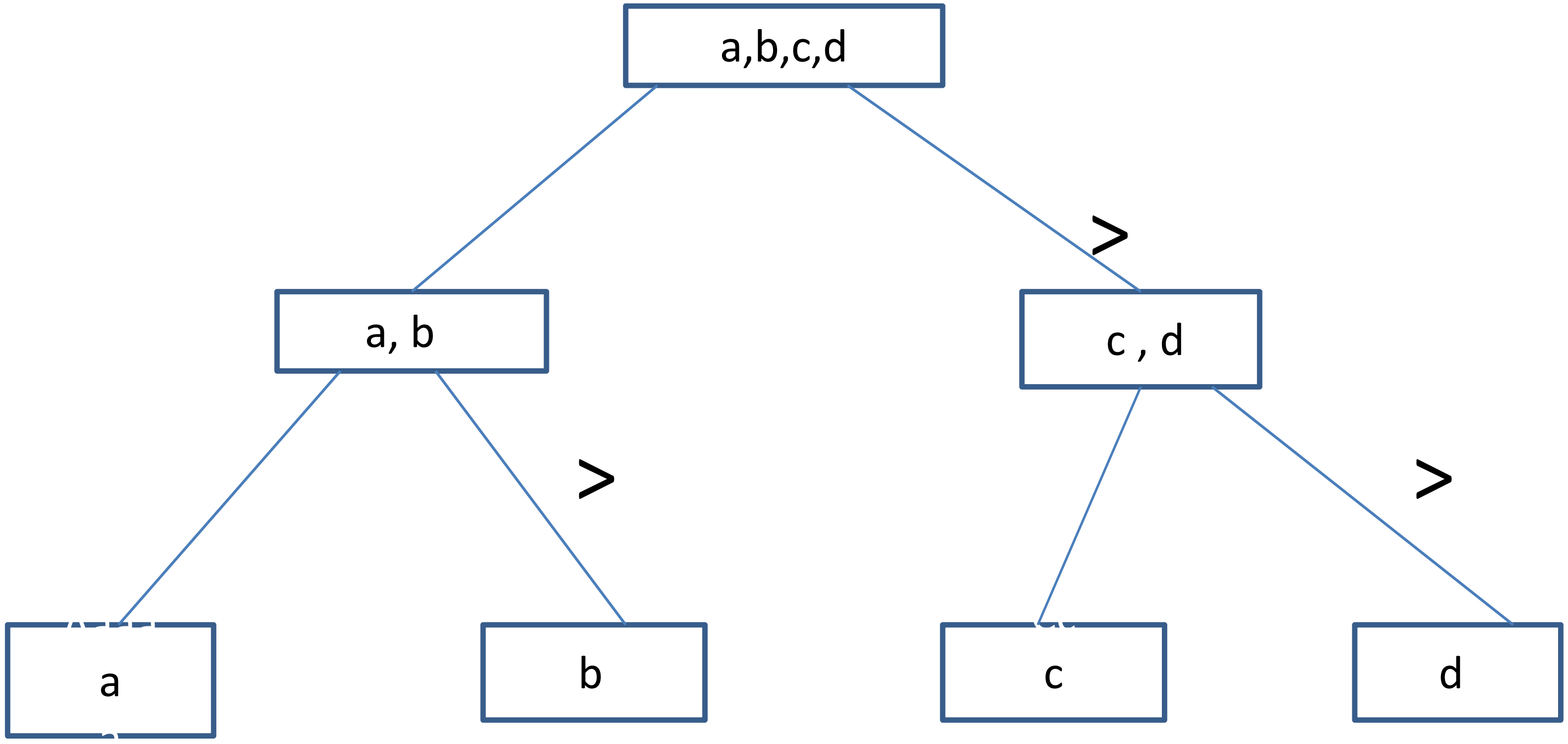
Proof: There are n possible outcomes, input strings

– In this case lgn comparisons are necessary,

– So, unique identification of an index in the list requires lgn bits.

– Therefore, lgn bits are necessary to specify one of the m possibilities.

Lower and Upper Bound Theory



- 2 bits of information is necessary.

Lower and Upper Bound Theory

Example-2: By using information theory, the lower bound for the problem of *comparison-based sorting problem* is (nlgn)

Proof:– If we only know that the input is orderable, then there are n! possible outcomes

– each of the n! permutations of n things.

– Within the comparison-swap model, we can only use comparisons to derive information

– Based on the information theory, lgn! bits of information is necessary by the number of comparisons needed to be done in the worst-case to sort n things.

Lower and Upper Bound Theory

- The result of a given comparison between two list of elements yields a single bit of information (0=False, 1 = True).
- Each of the n! permutations of {1, 2, …, n} has to be distinguished by the correct algorithm.
- Thus, a comparison-based algorithm must perform enough comparisons to produce n! cumulative pieces of information.
- Since each comparison only yields one bit of information, the question is what the minimum number of bits of information needed to allow n! different outcomes is, which is lgn! bits.

Lower and Upper Bound Theory

- How fast lgn! grow? We can bind n! from above by overestimating every term of the product, and bind it below by underestimating the first n/2 terms.
- n/2 x n/2 x …x n/2 x…x 2 x 1

  n! = n x (n-1) x …x 2 x1

  n x n x …x n

  $(n/2)^{n/2}$ n! n^n

  ½(nlgn-n)   lgn! nlgn

  This follows that lgn!   (nlgn)

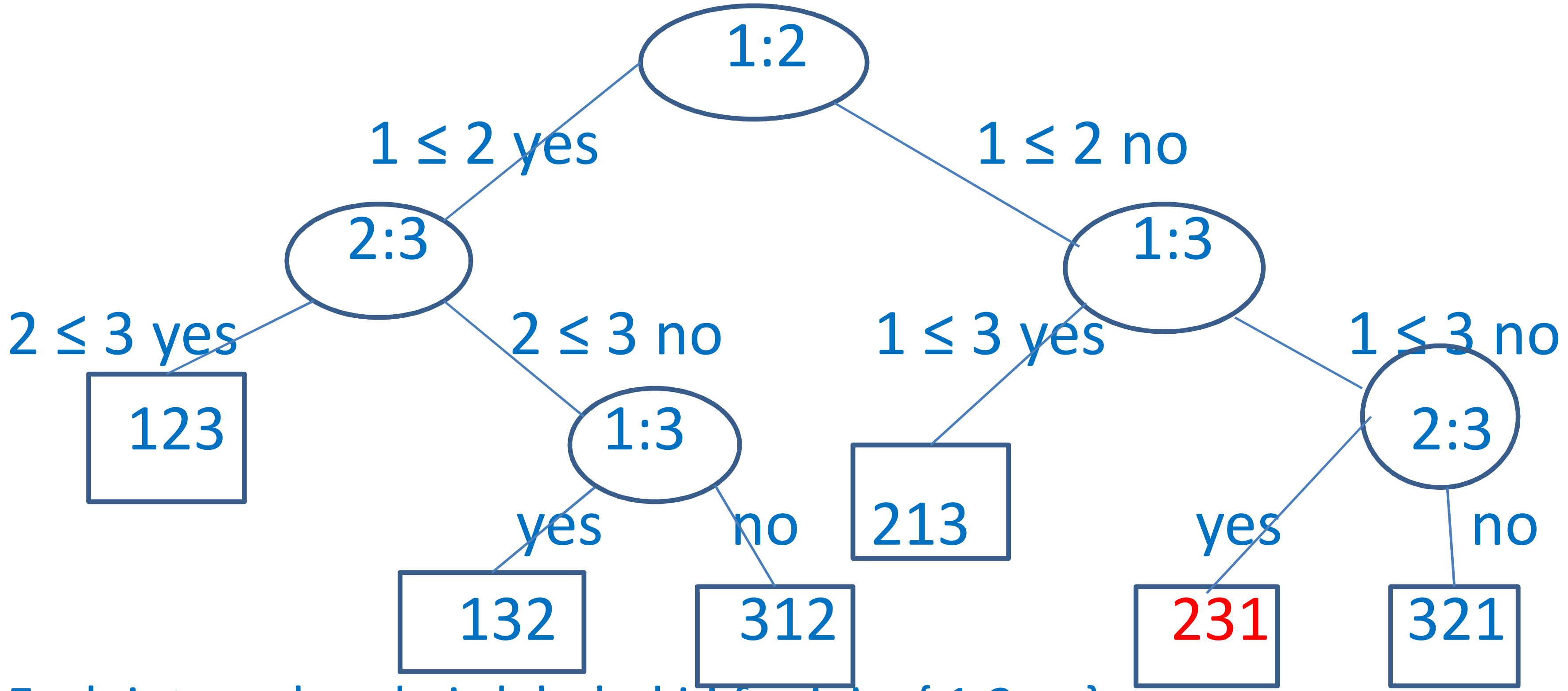
Decision-tree model

- This method can model the execution of any comparison based problem.

  One tree for each input size *n*.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = the height of tree.

Decision-tree Example

- Sort $< a_1, a_2, \ldots a_n>$ e.g =< 9, 4, 6>



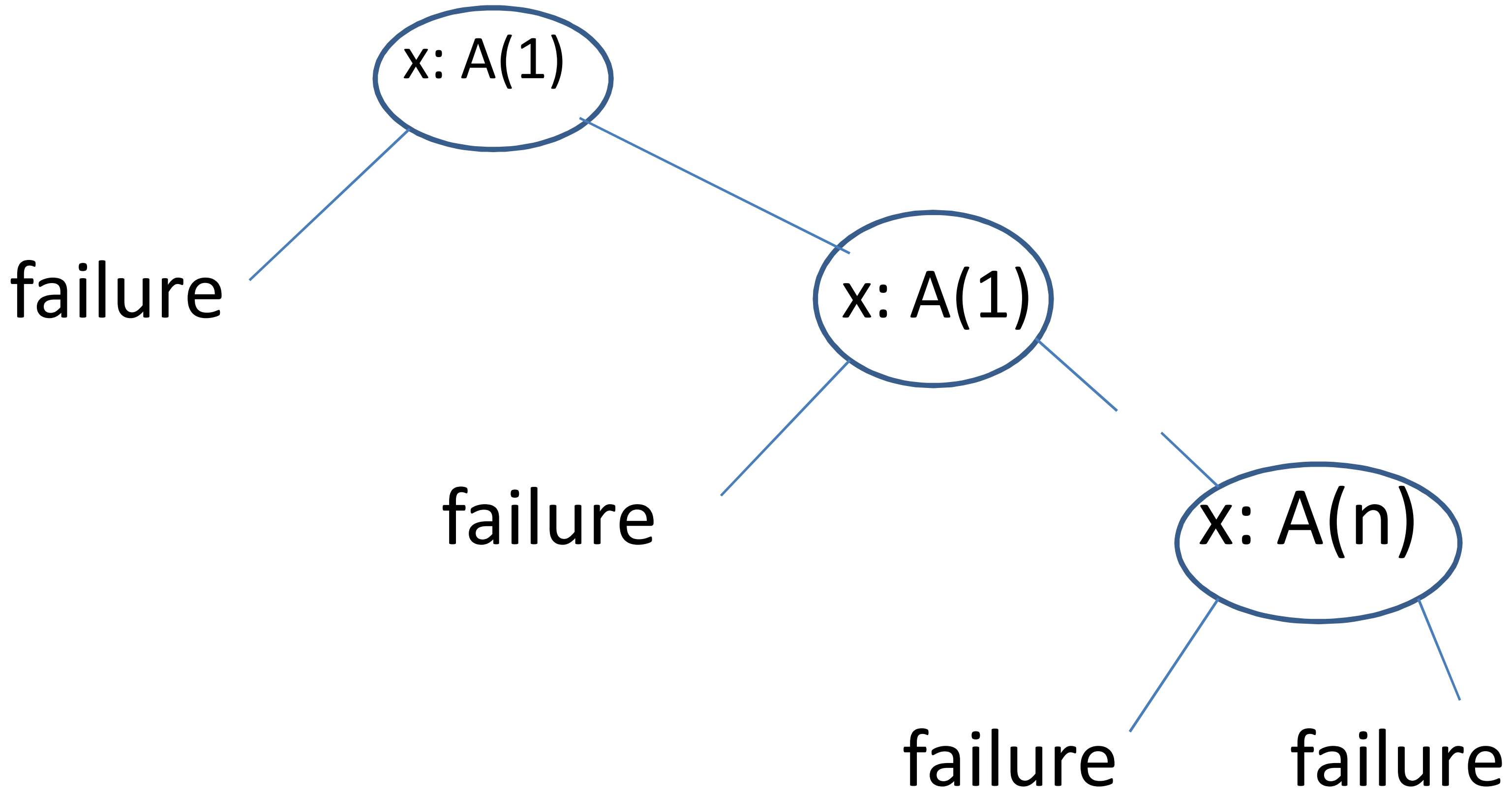
Each internal node is labeled i:j for i, j є { 1,2…n}

The left subtree shows subsequent comparisons if $a_i \leq a_j$

The right subtree shows subsequent comparisons if $a_i \geq a_j$ shows

Decision-tree modeling for searching

X =A[i] algo terminates , X< A[i] left branch or X> A[i] right branch



A comparison tree for a linear search algorithm

Decision-tree model

Example: By using the Decision Tree method, the lower bound for comparison-based searching on ordered input is (lgn)

- **Proof:** Let us consider all possible comparison trees which model algorithms to solve the searching problem.
  - FIND(n) is bounded below by the distance of the longest path from the root to a leaf in such a tree.

Decision-tree model

- There must be n internal nodes in all of these trees corresponding to the n possible successful occurrences of x in A.

  If all internal nodes of binary tree are at levels less than or equal to k (every height k-rooted binary tree has at most $2^{k+1} - 1$ nodes), then there are at most $2^k - 1$ internal nodes.

  Thus, n $2^k - 1$ and FIND(n) = k log (n+1) .

  Because every leaf in a valid decision tree must be reachable, the worst-case number of comparisons done by such a tree is the number of nodes in the longest path from the root to a leaf in the binary tree consisting of the comparison nodes.

Thankyou