Backtracking

Backtracking(D.H. Lehmer :1950)

One of the most general Technique to solve problem

Used to:

1. Search a set of Solutions

2. **Optimal solution** satisfying certain constraint

Backtracking

General Idea:

Solution produced by backtracking looks like

$$\langle x[1], x[2], x[3], \ldots, x[n] \rangle$$

Where **x[i]** is chosen from finite set S_i **satisfying certain condition**

Backtracking produces the solution ***component by component***

Backtracking

Explicit Constraint:

Explicit constraints are rules that restricts each x[i] to take on values from a given set

Example:

$$x[i] \geq 0$$

$$x[i] \in \{0, 1\}$$

$$2 \leq x[i] \leq 5$$

Backtracking

Implicit Constraint:

Implicit constraints are rules that determines which of the tuples in solution space **satisfy the criterion function**

Backtracking

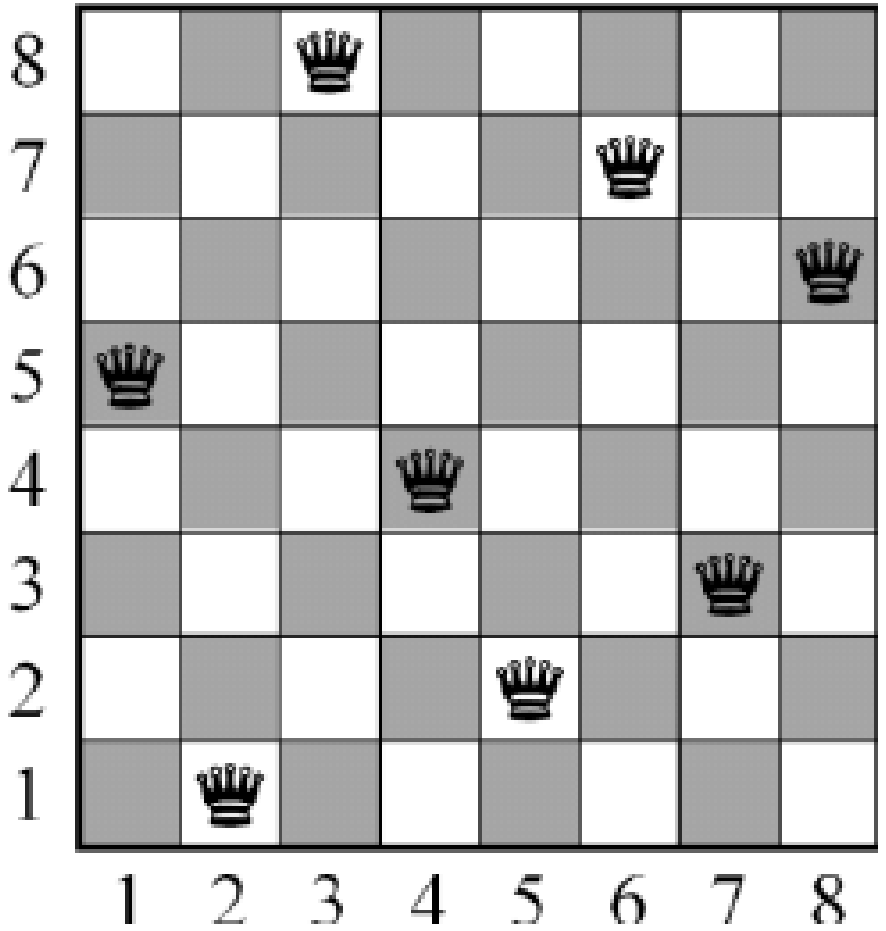
Application:

- 8-Queen Problem
- Hamiltonian Cycle Problem
- Graph Coloring Problem

8-Queens Problem

Place 8 queens on an 8 x 8 chessboard so that no **2 queens** attack each other

QUEEN



8-Queens Problem

2- Queens attack each other if

They are in same row

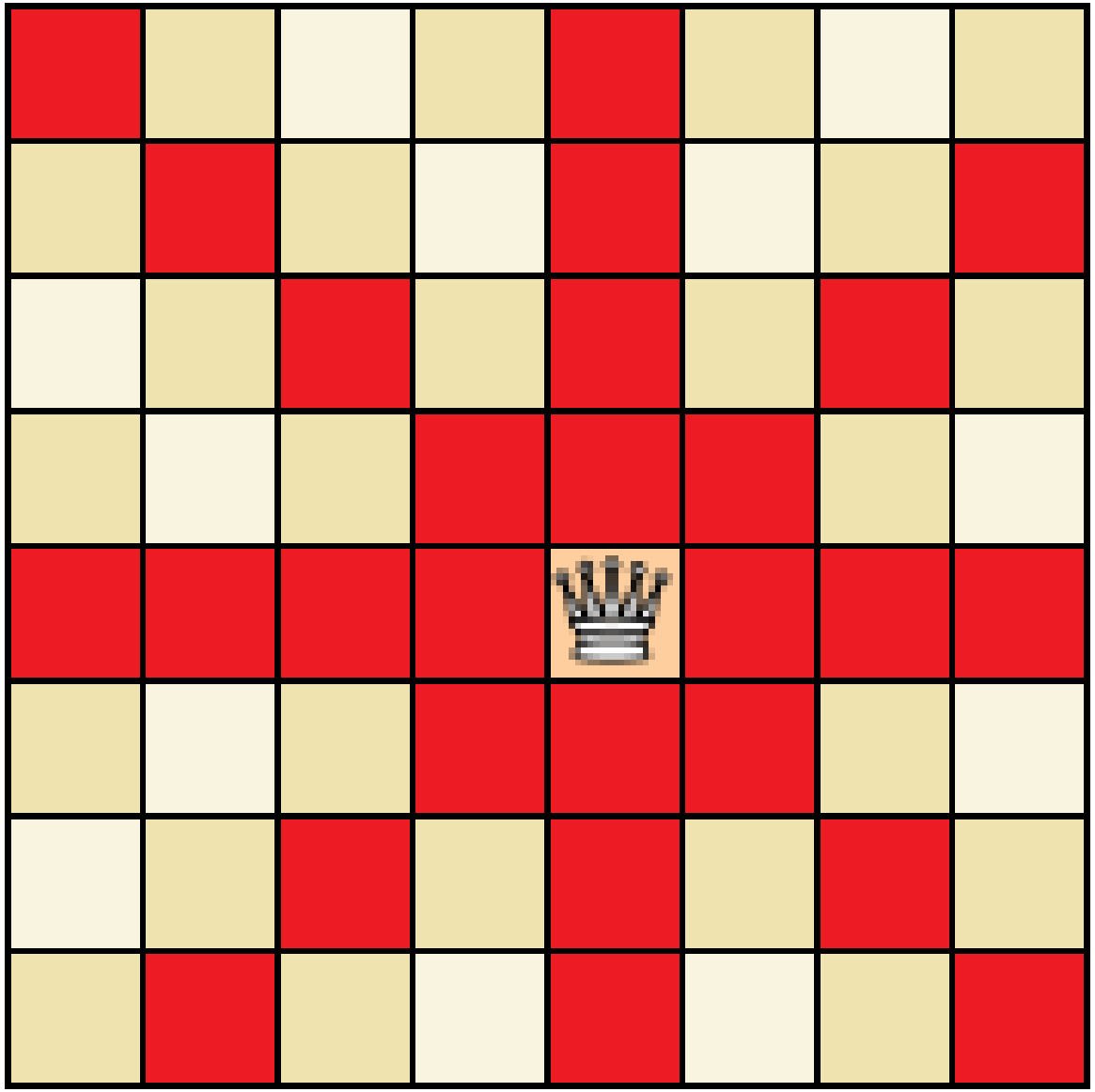
or

They are in same column

or

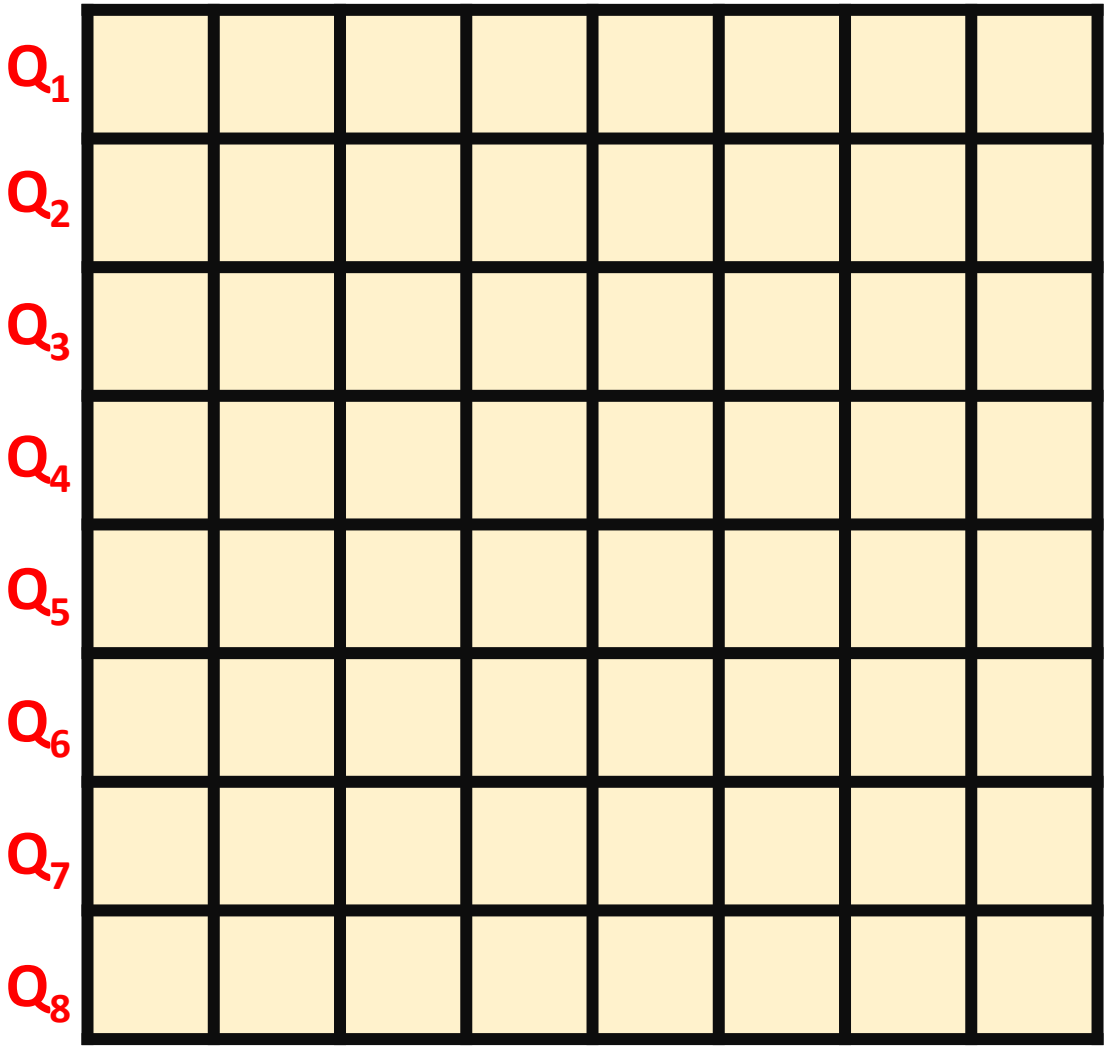
They are in same diagonal

8-Queens Problem



8-Queens Problem

Let us assume that i^{th} queen is placed at i^{th} row



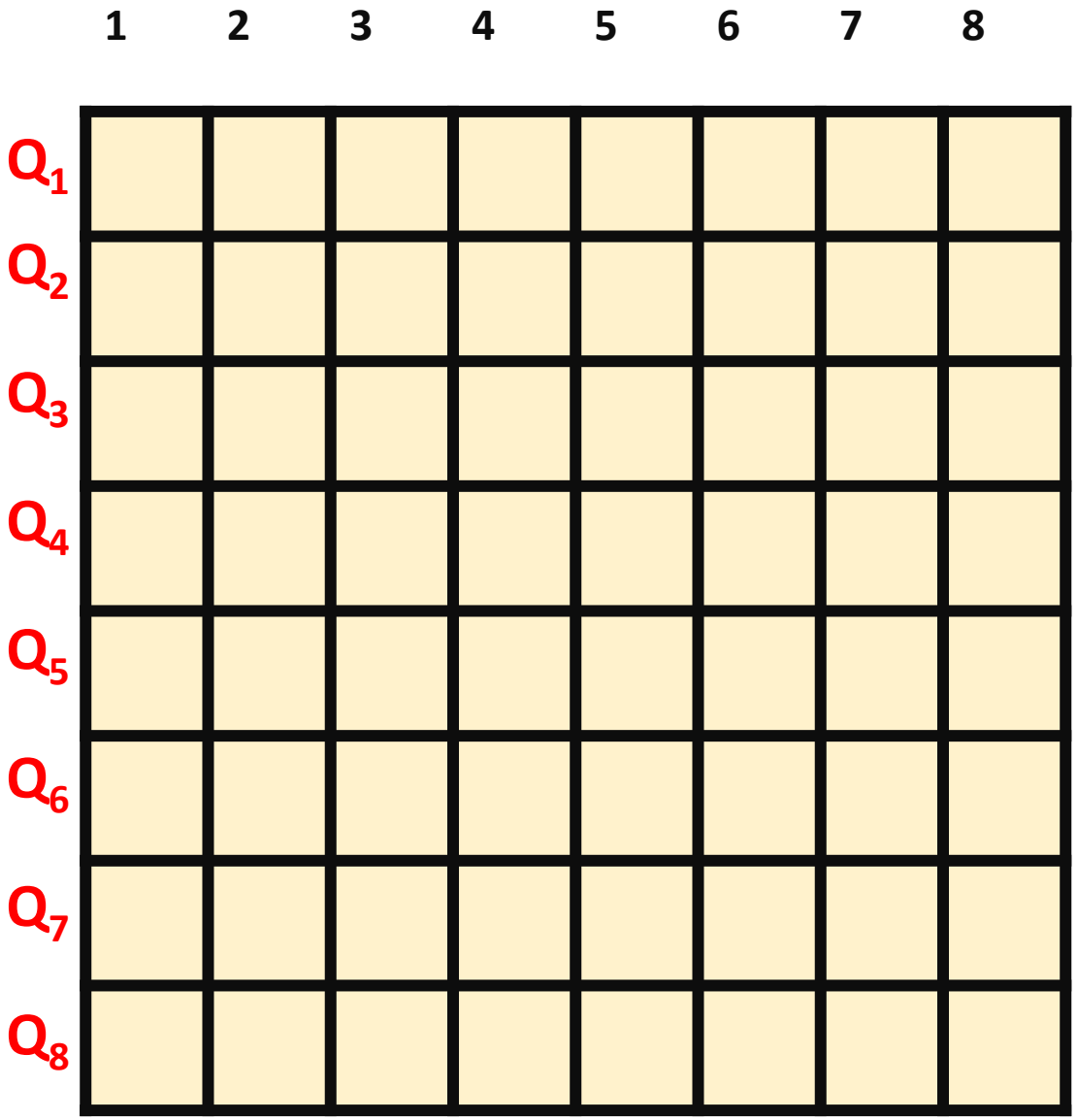
8-Queens Problem

Solution: [x[1], x[2], x[3], x[4], x[5], x[6], x[7], x[8]]

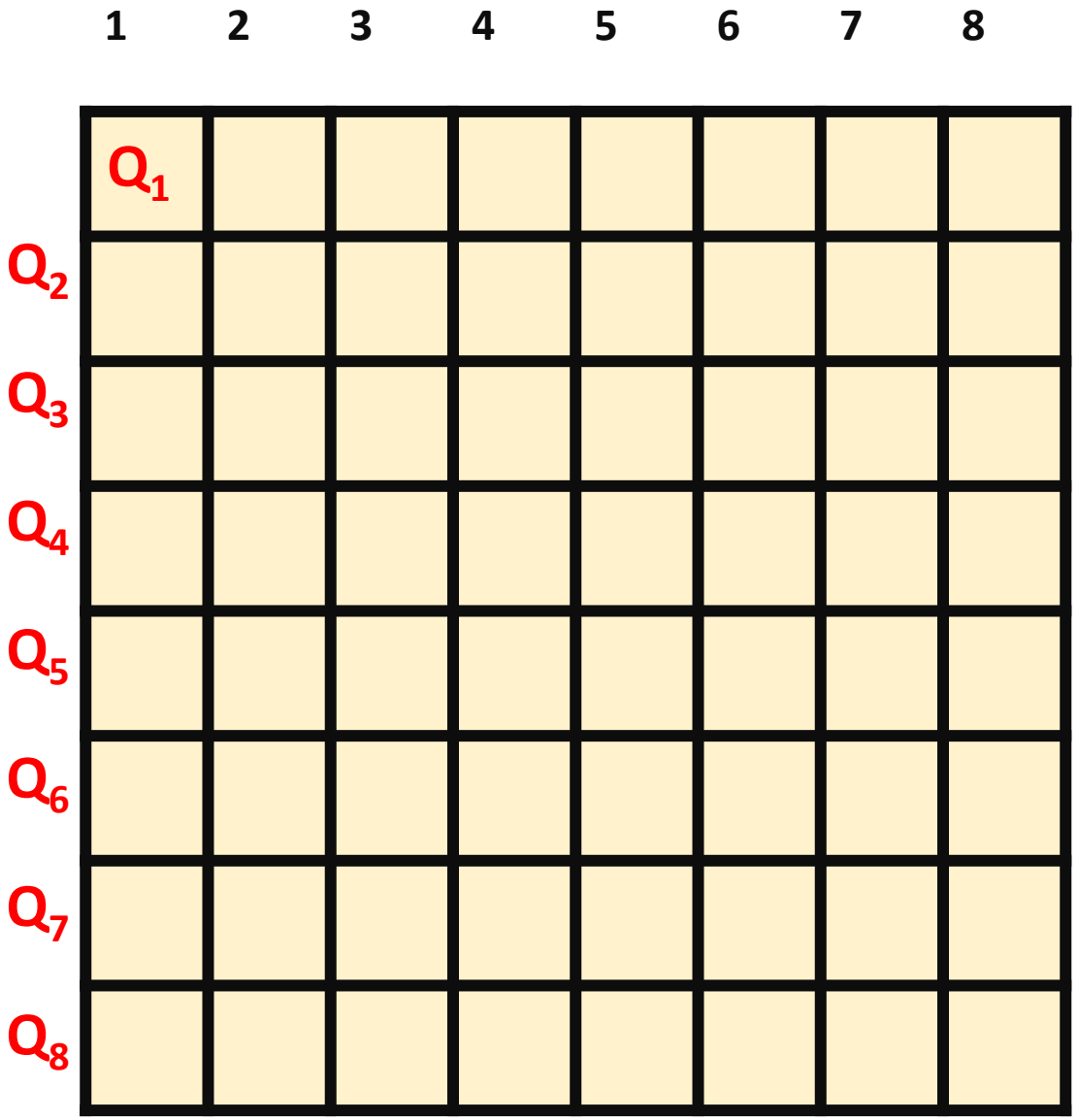
x[i] : the *column number* of i^{th} Queen

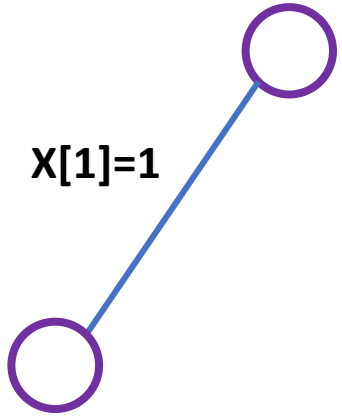
$x[i] \in \{1, 2, 3, 4, 5, 6, 7, 8\}$ $\quad where\ 1 \leq i \leq 8$

8-Queen Problem

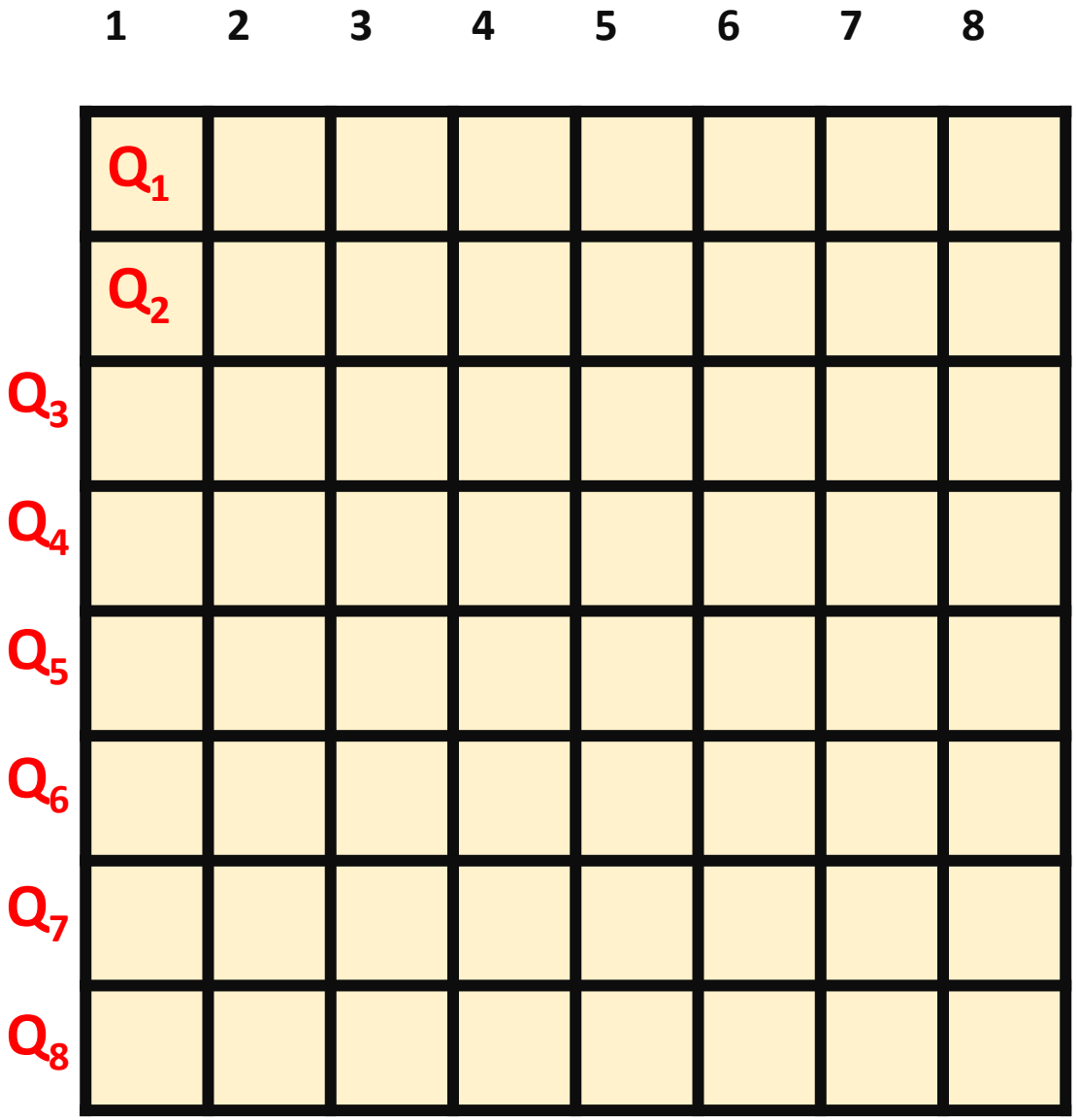


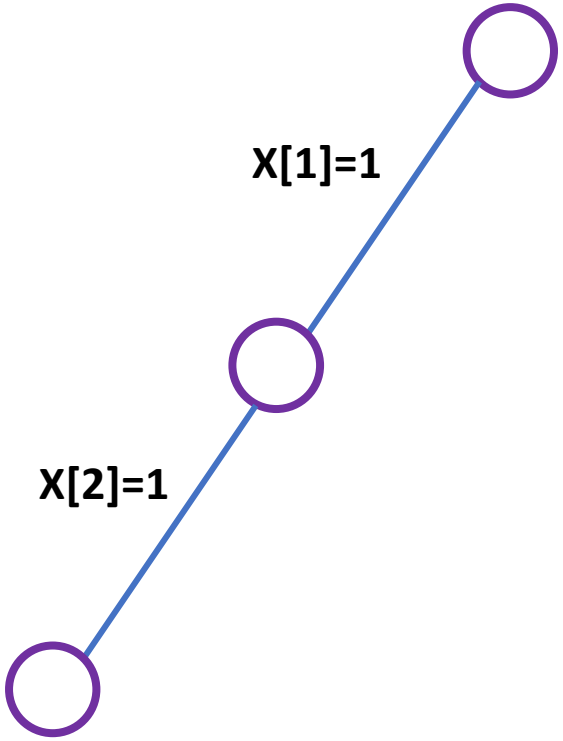
8-Queen Problem



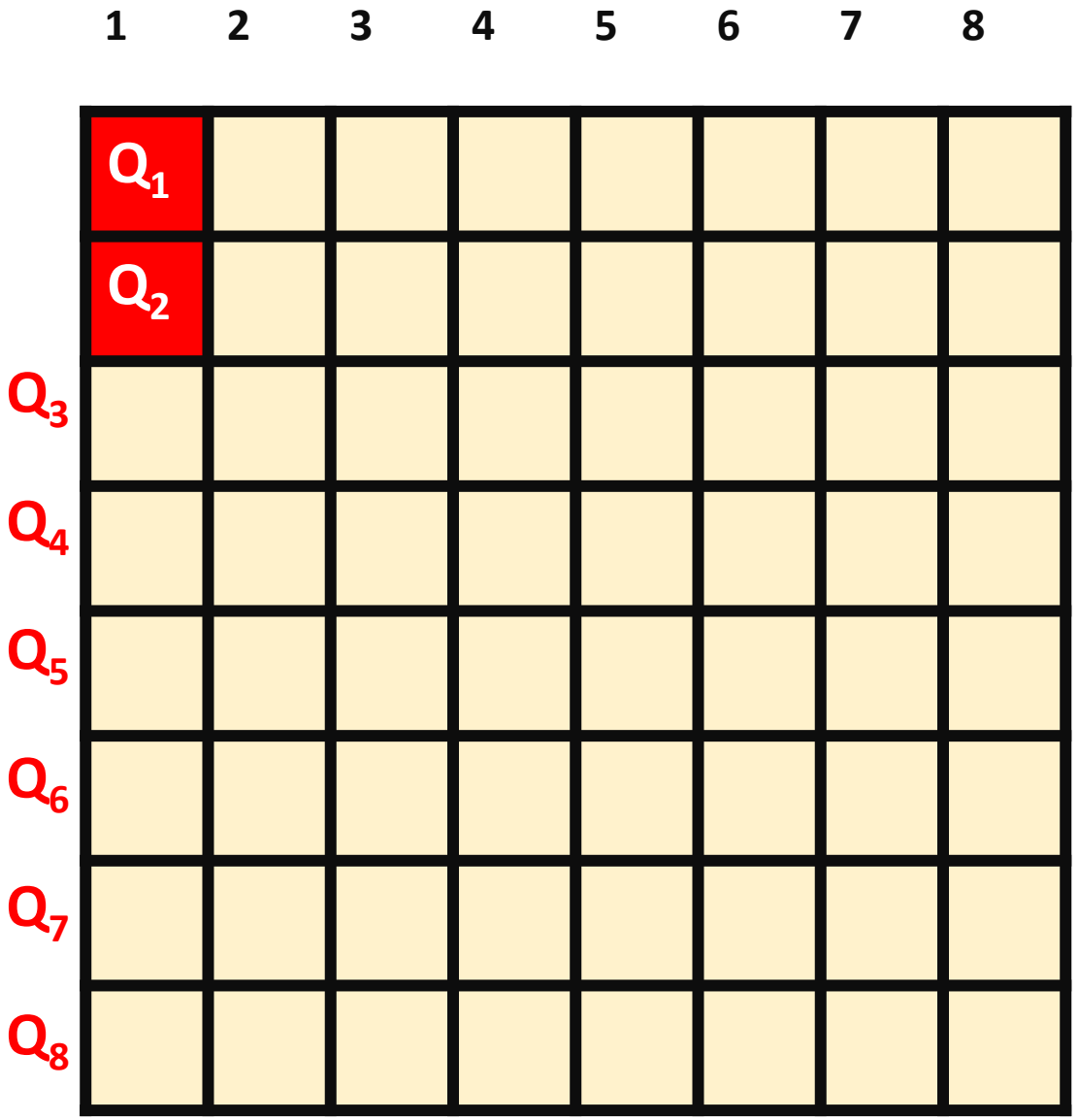


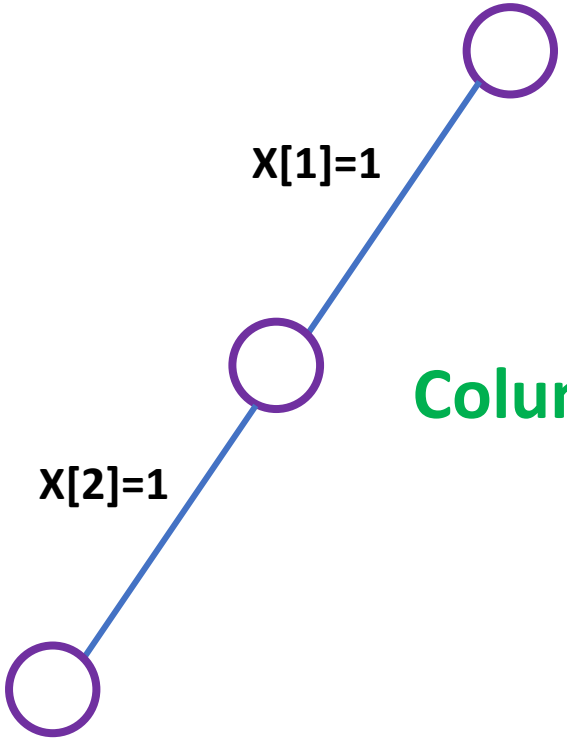
8-Queen Problem





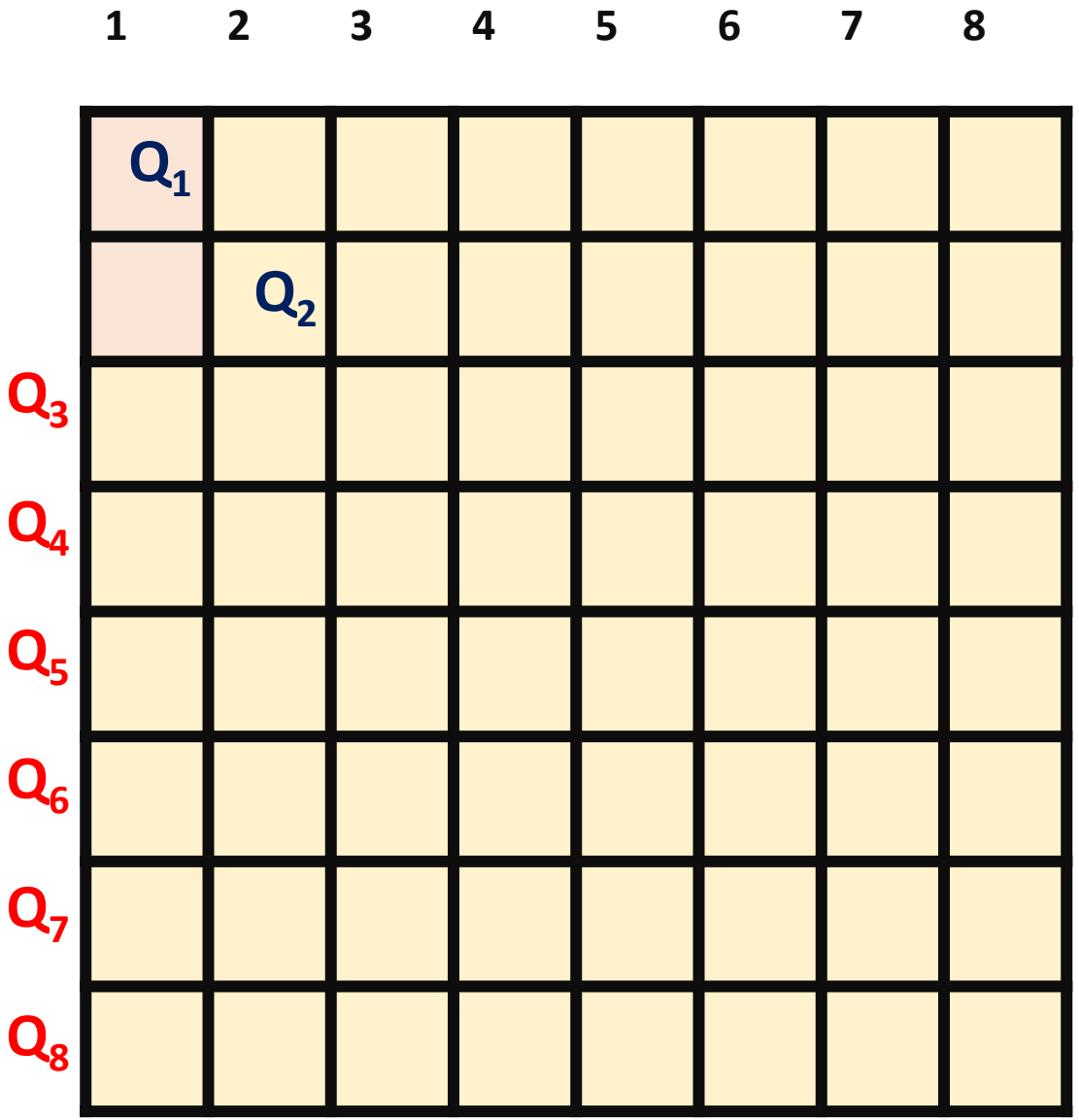
8-Queen Problem

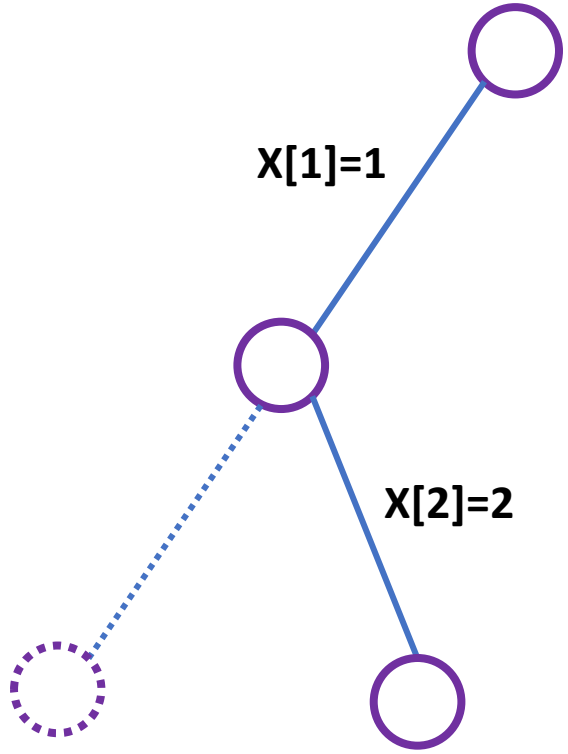




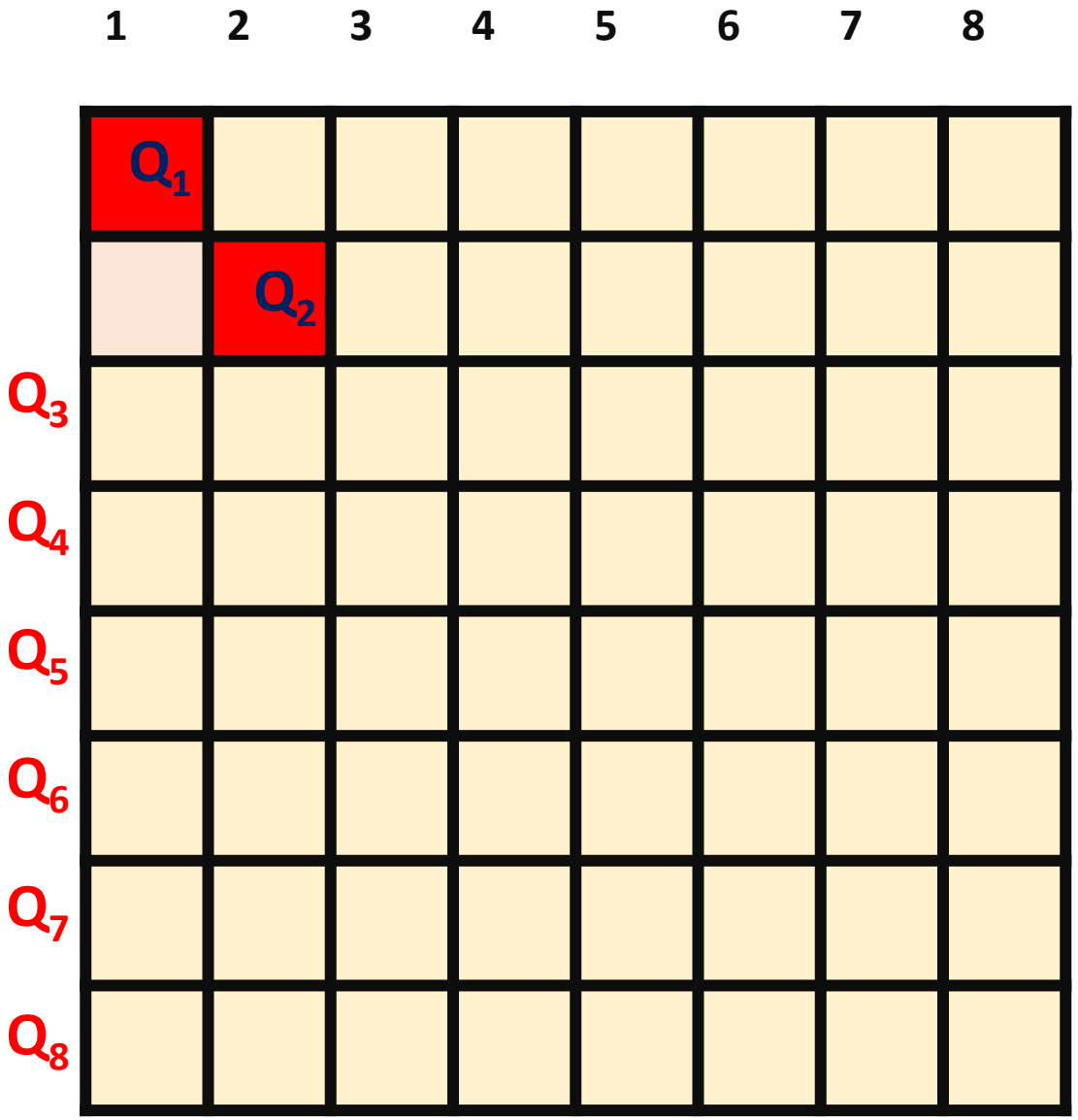
Column Attack

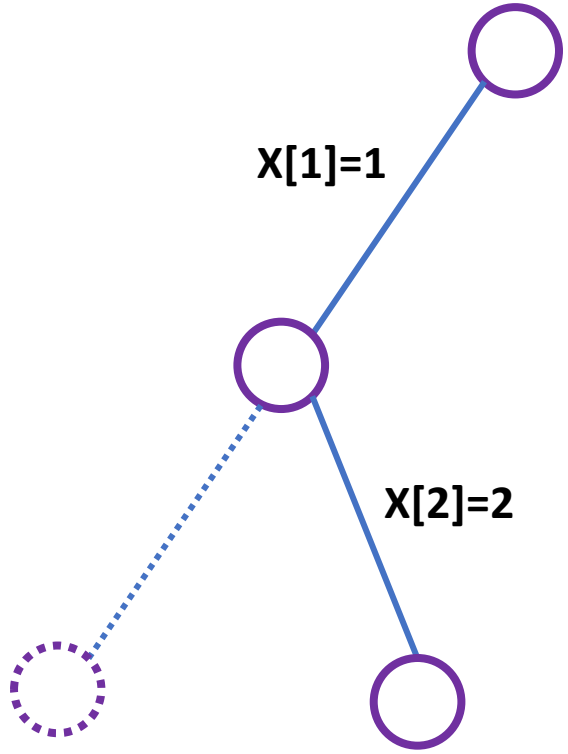
8-Queen Problem





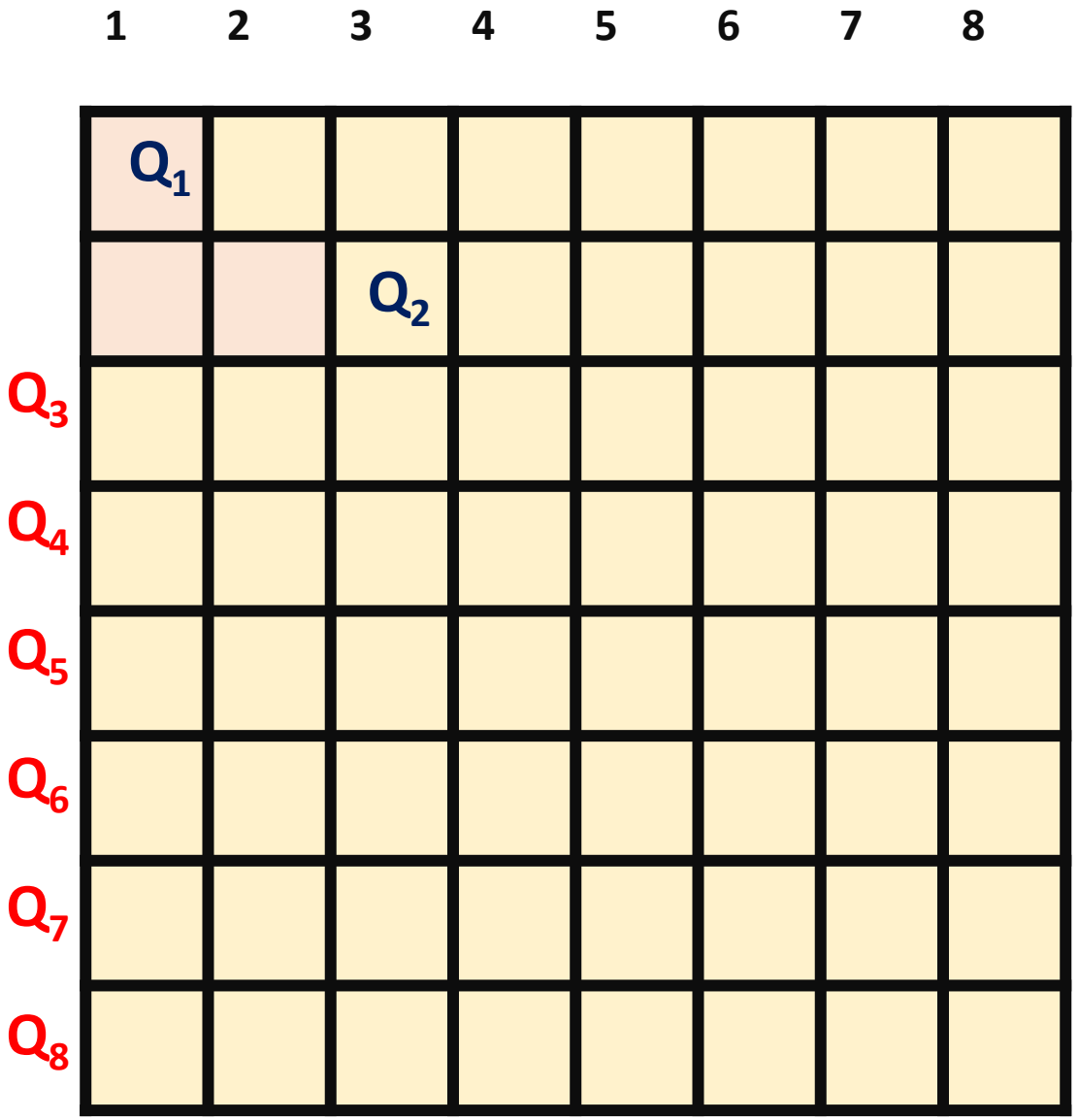
8-Queen Problem

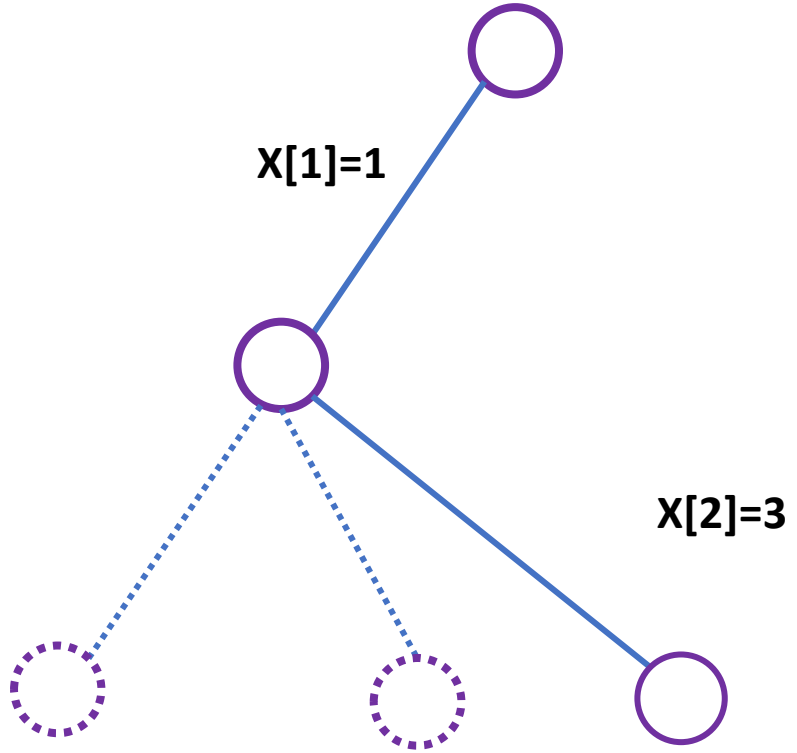




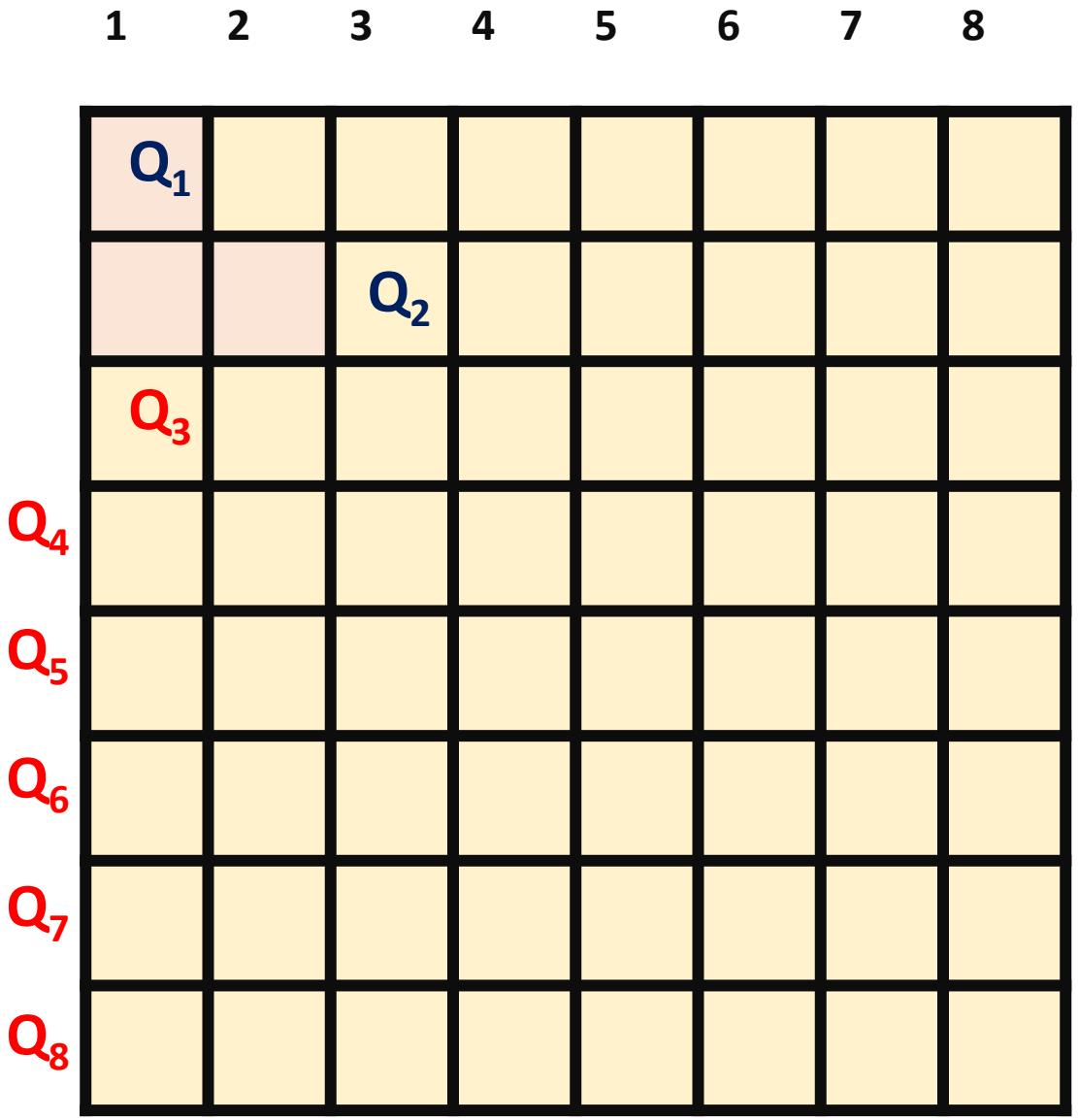
Diagonal Attack

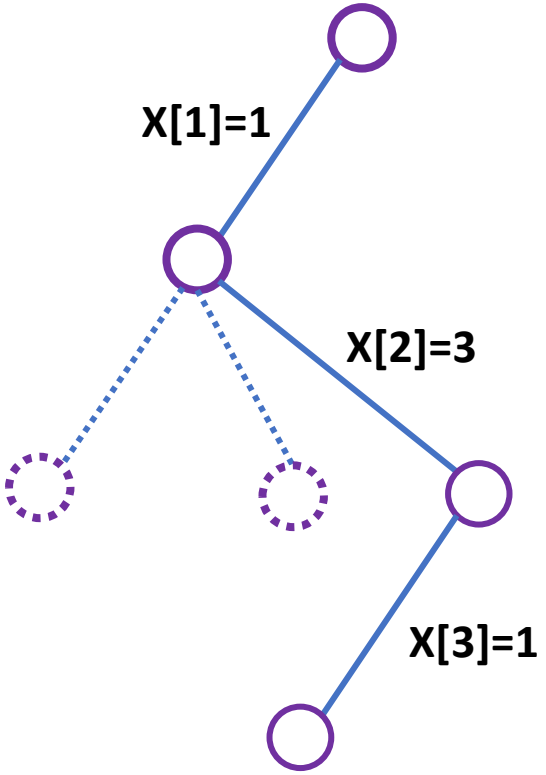
8-Queen Problem



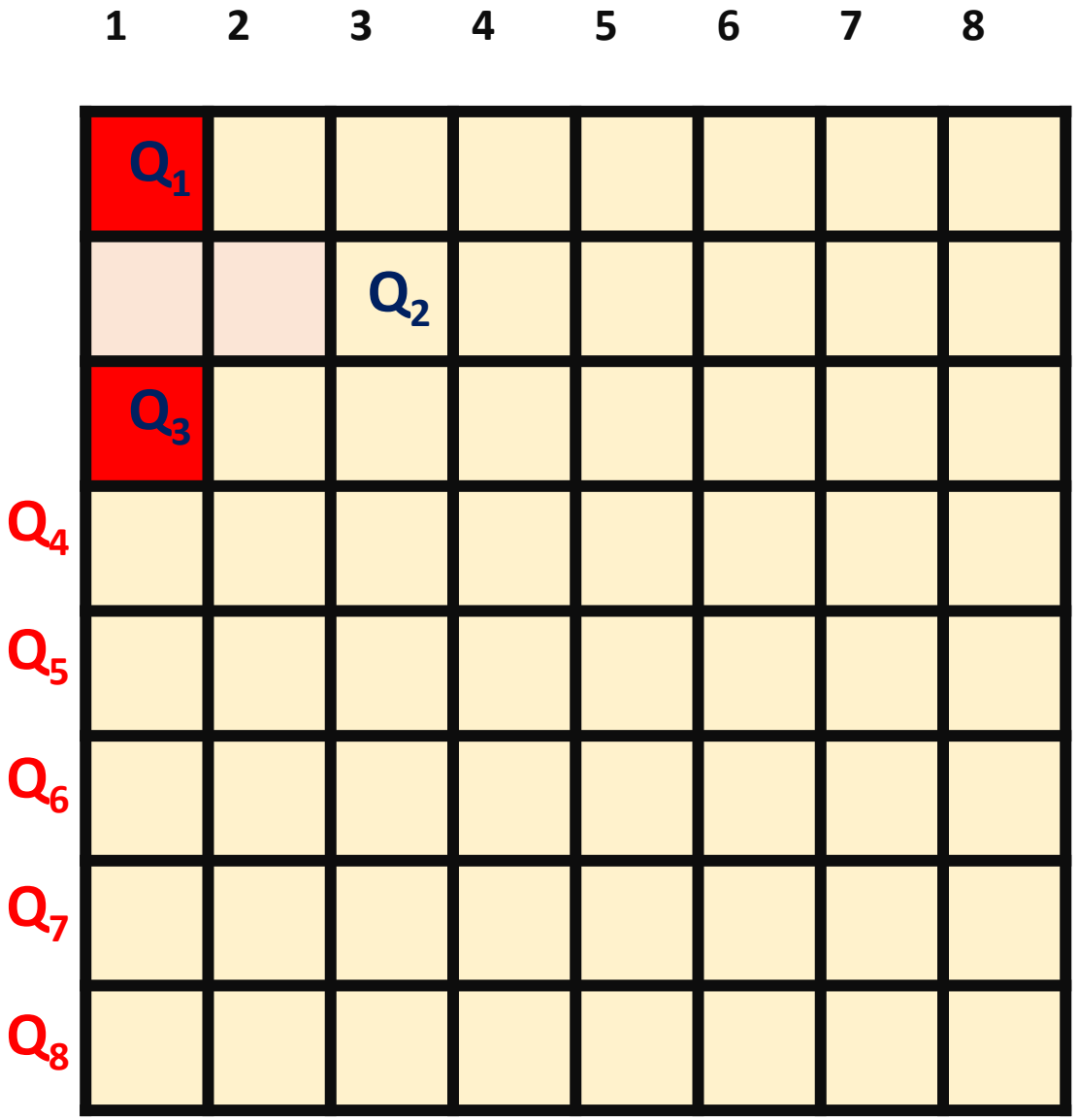


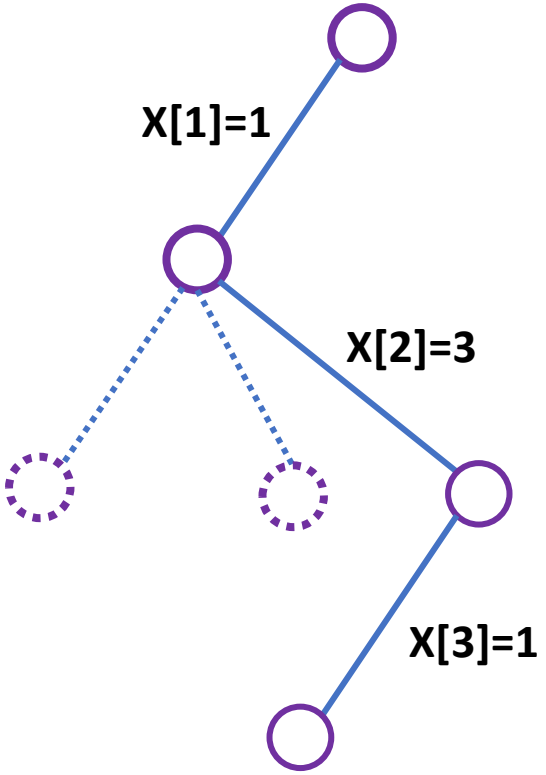
8-Queen Problem





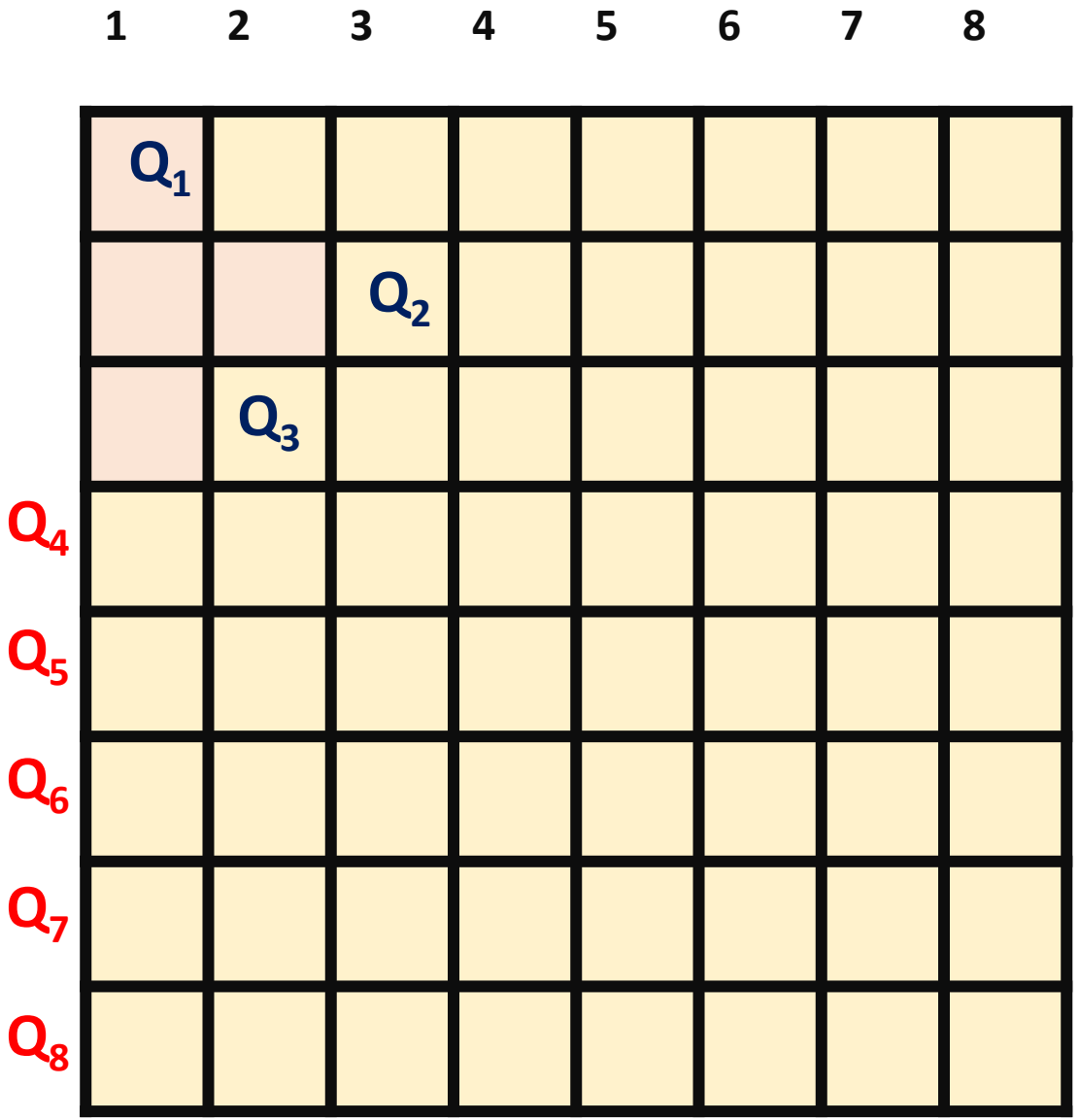
8-Queen Problem

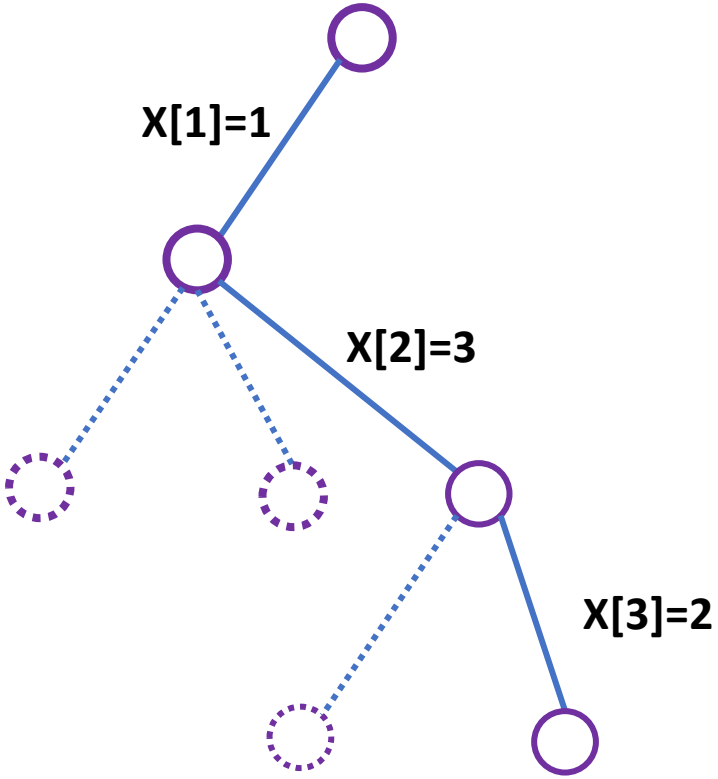




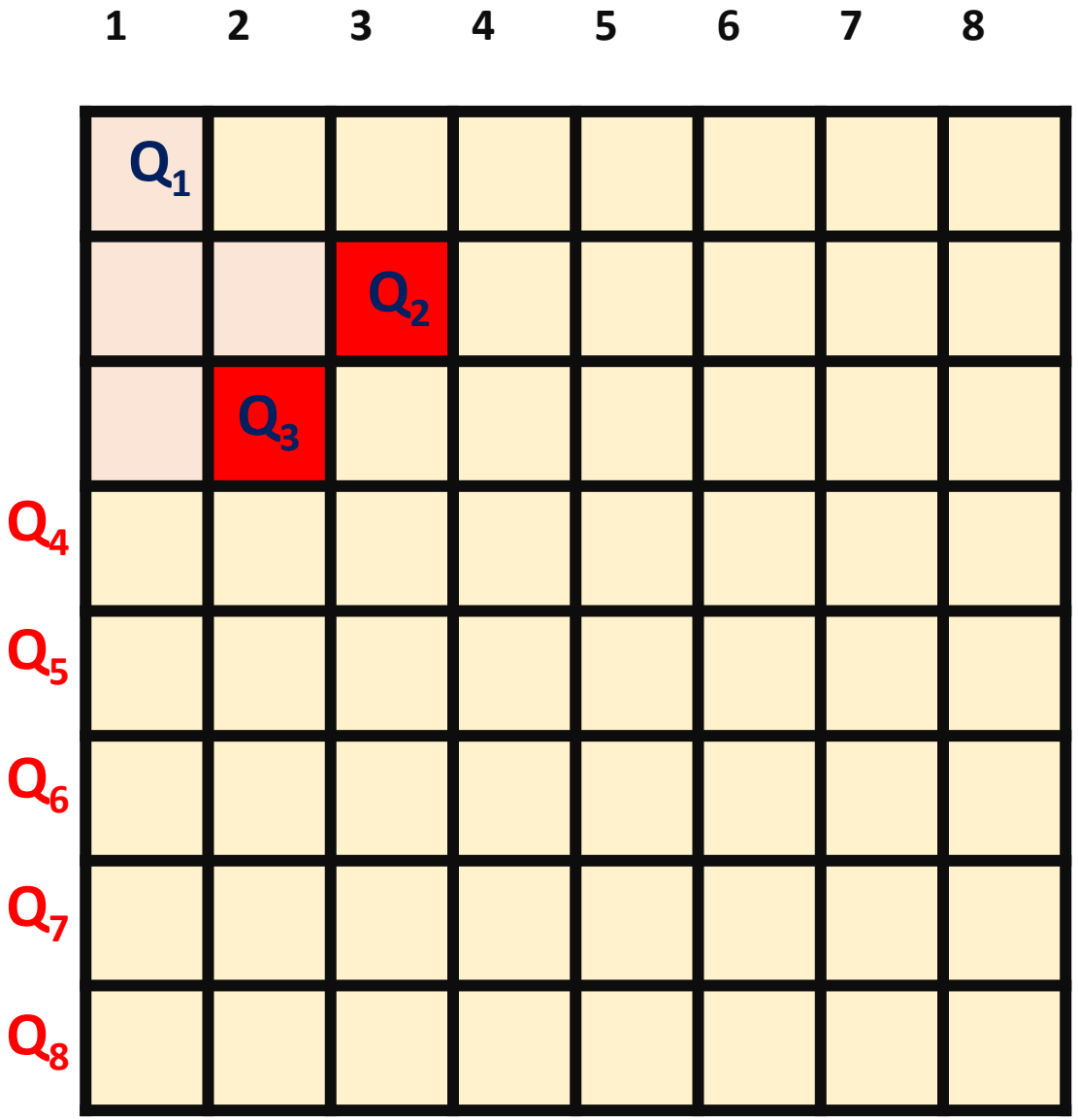
Column Attack

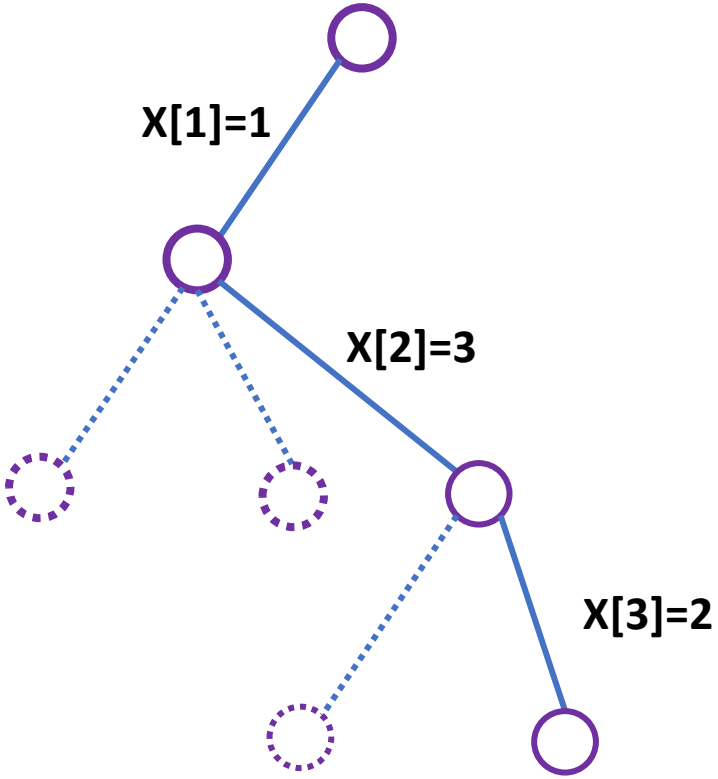
8-Queen Problem





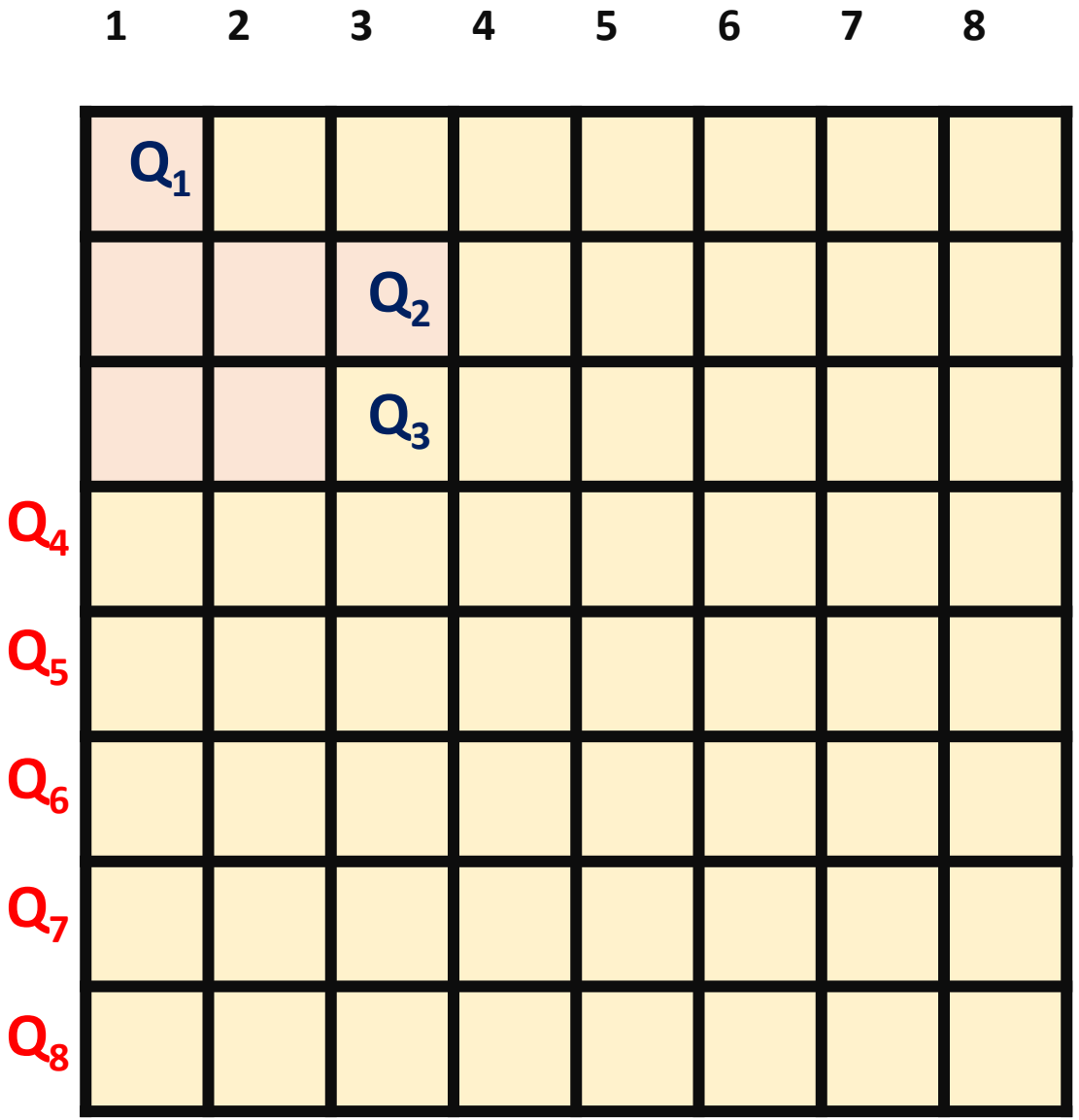
8-Queen Problem

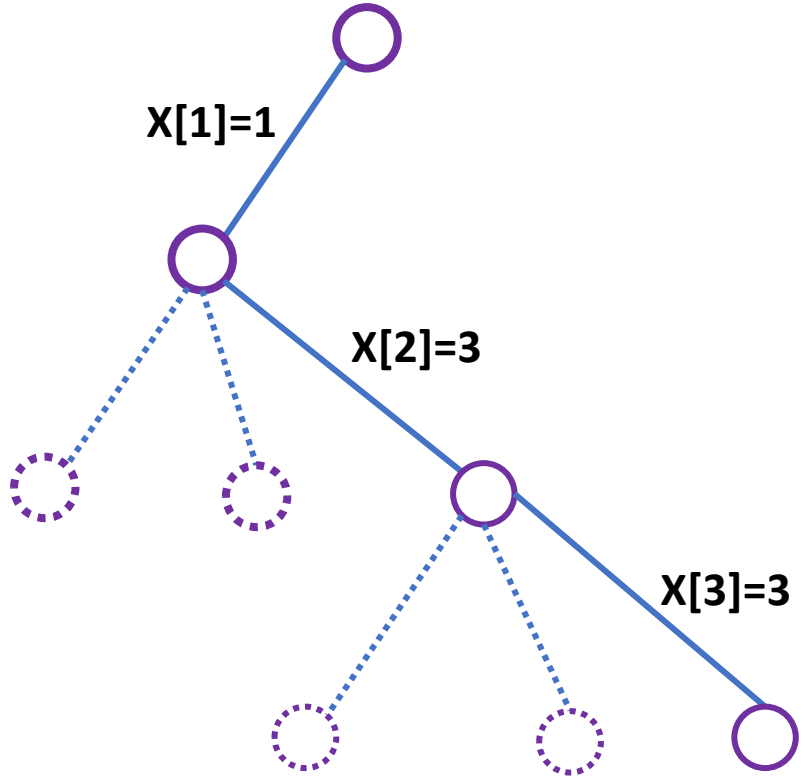




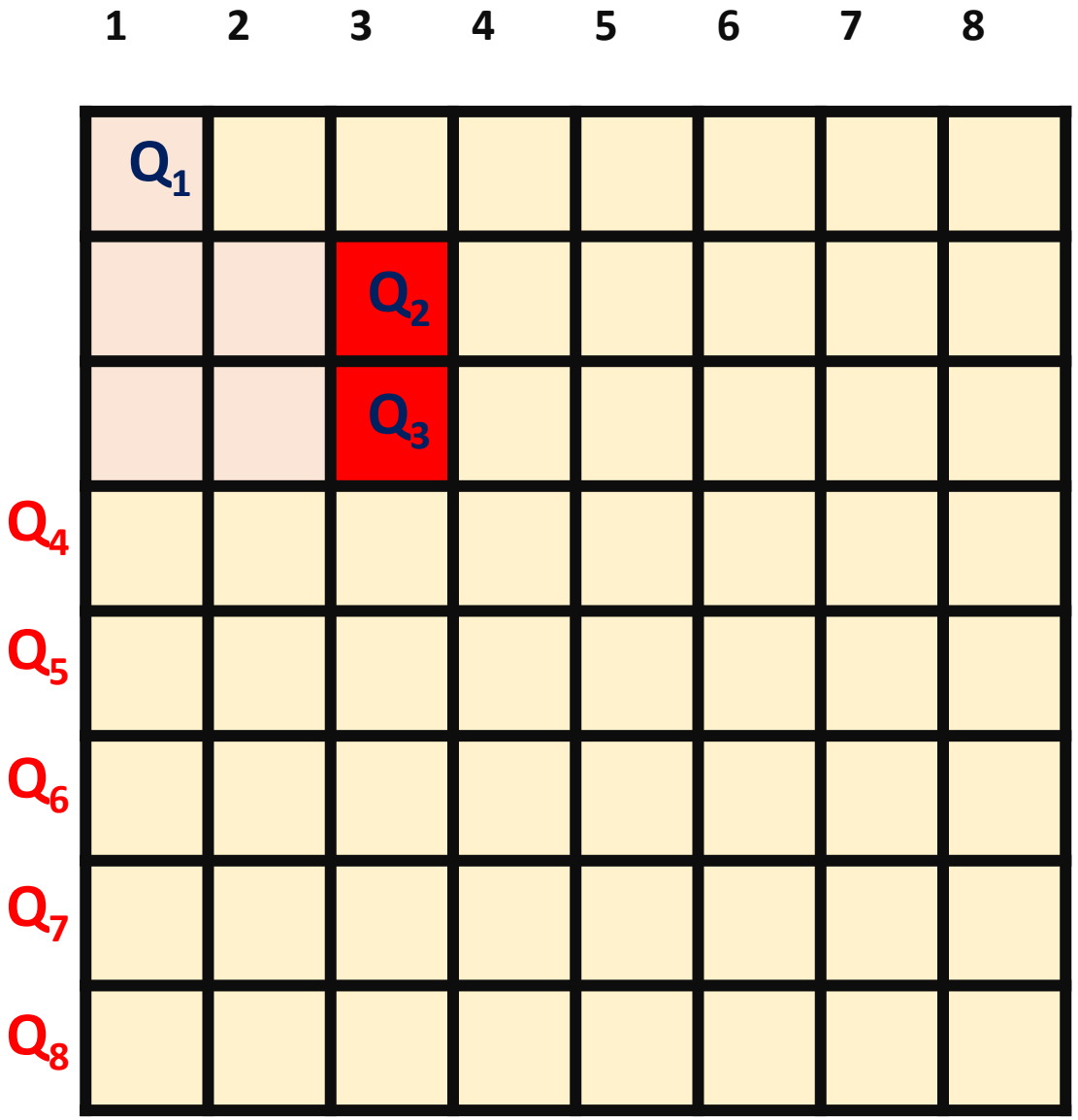
Diagonal Attack

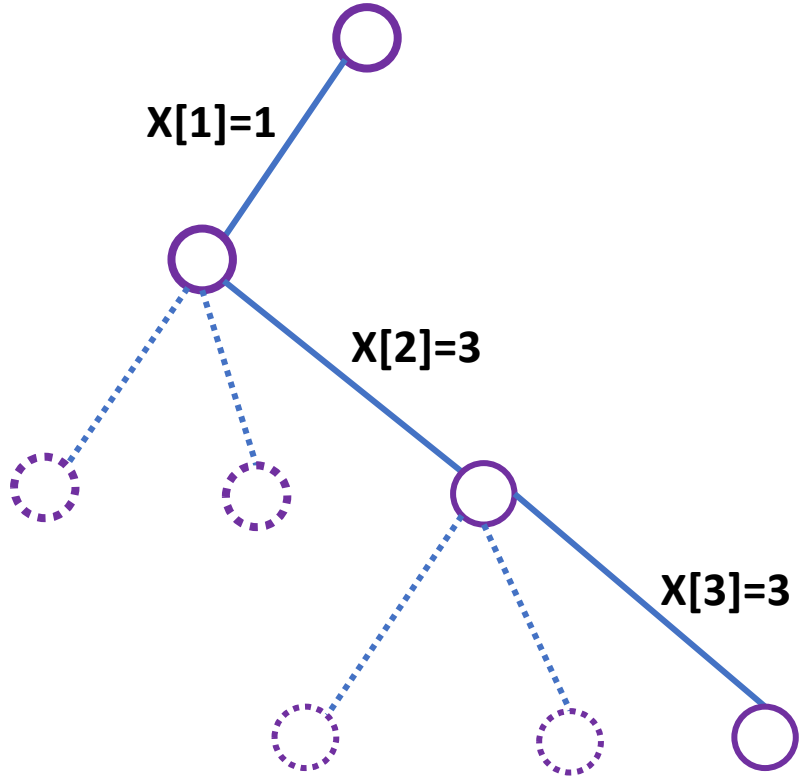
8-Queen Problem





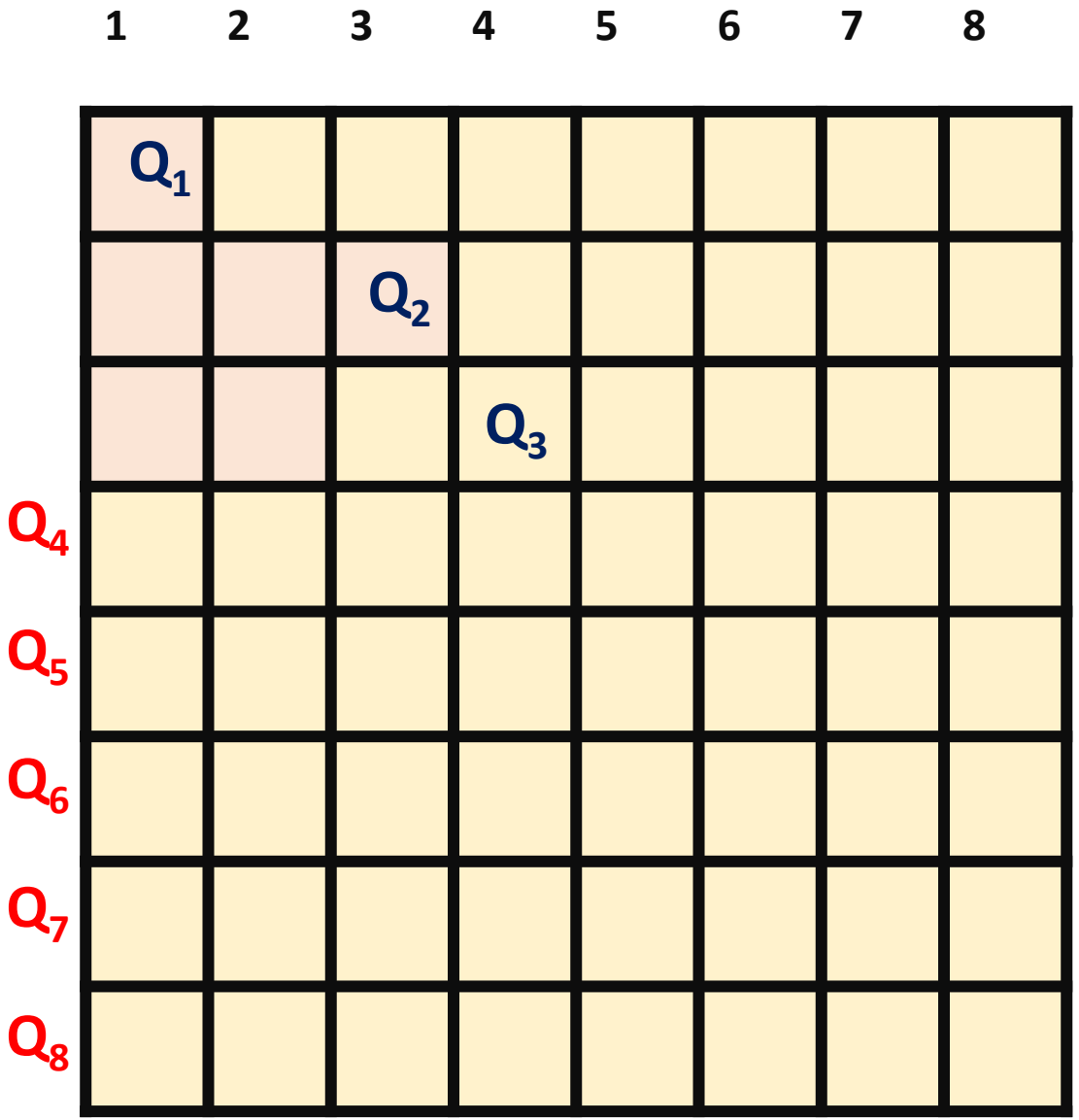
8-Queen Problem

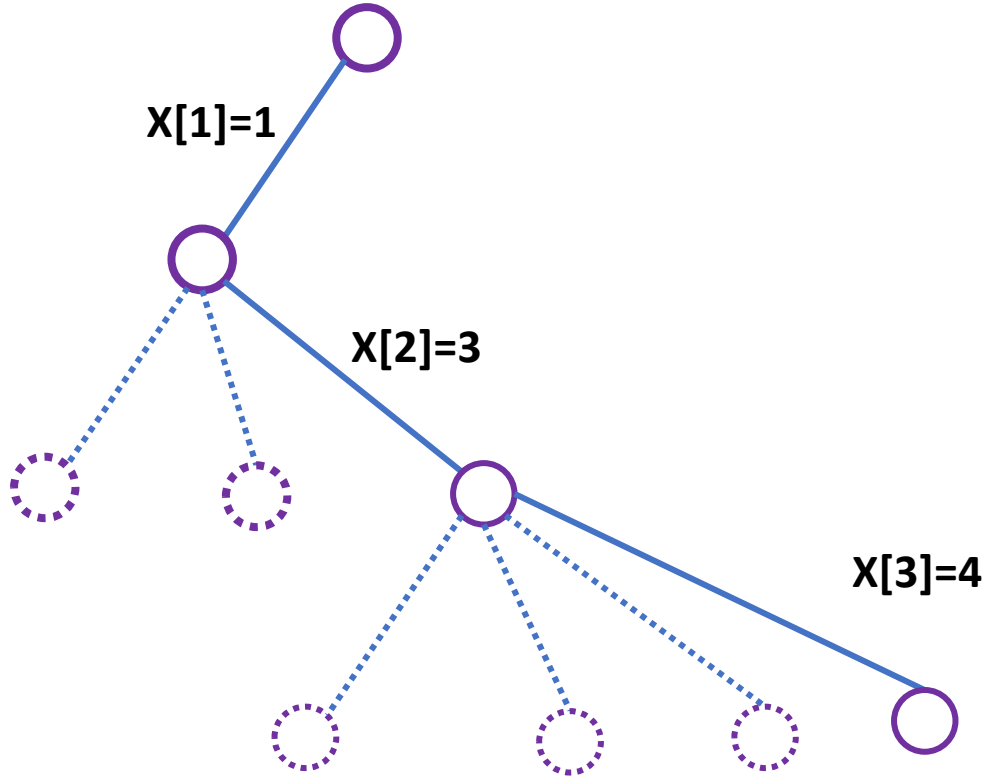




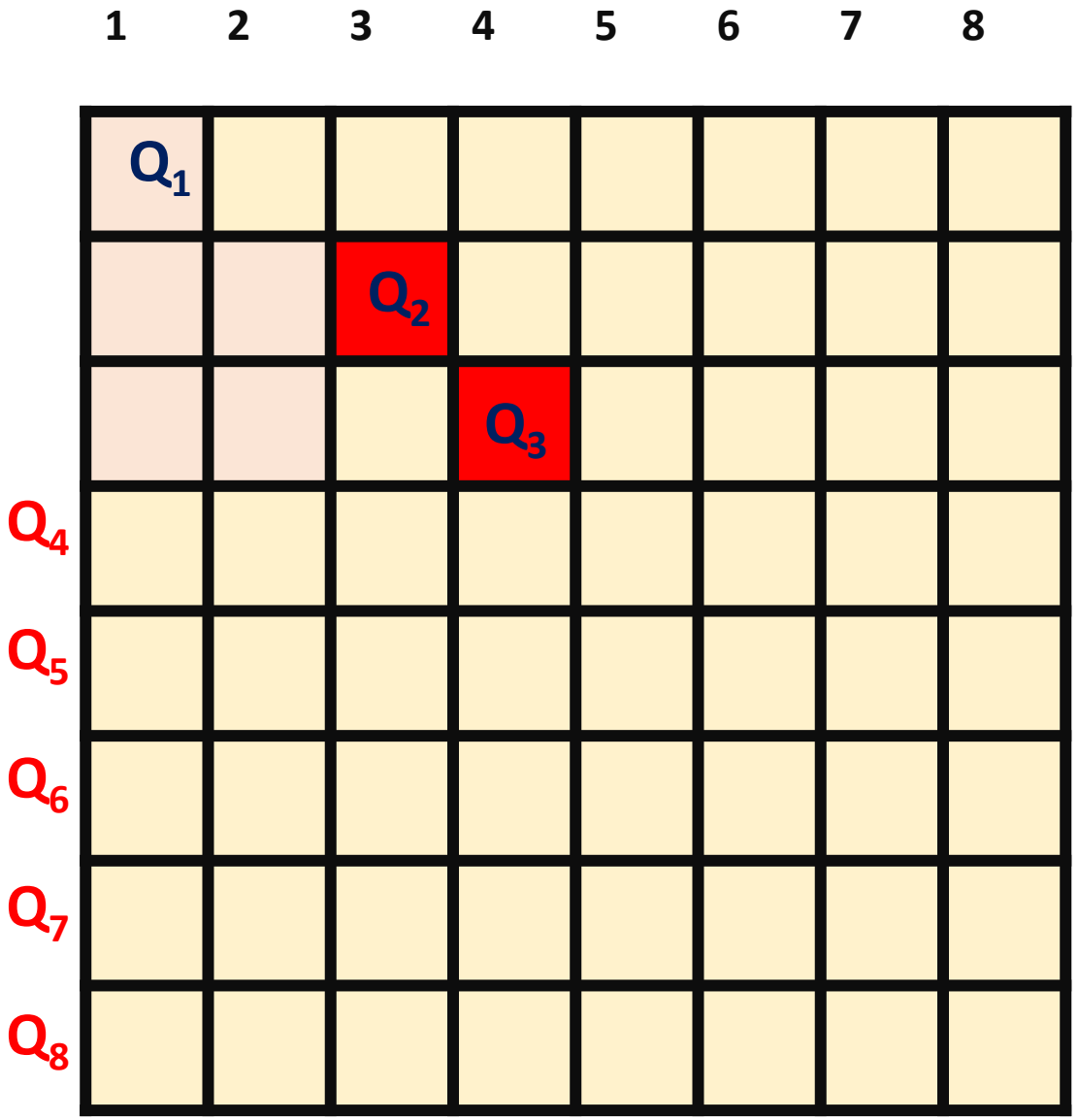
Column Attack

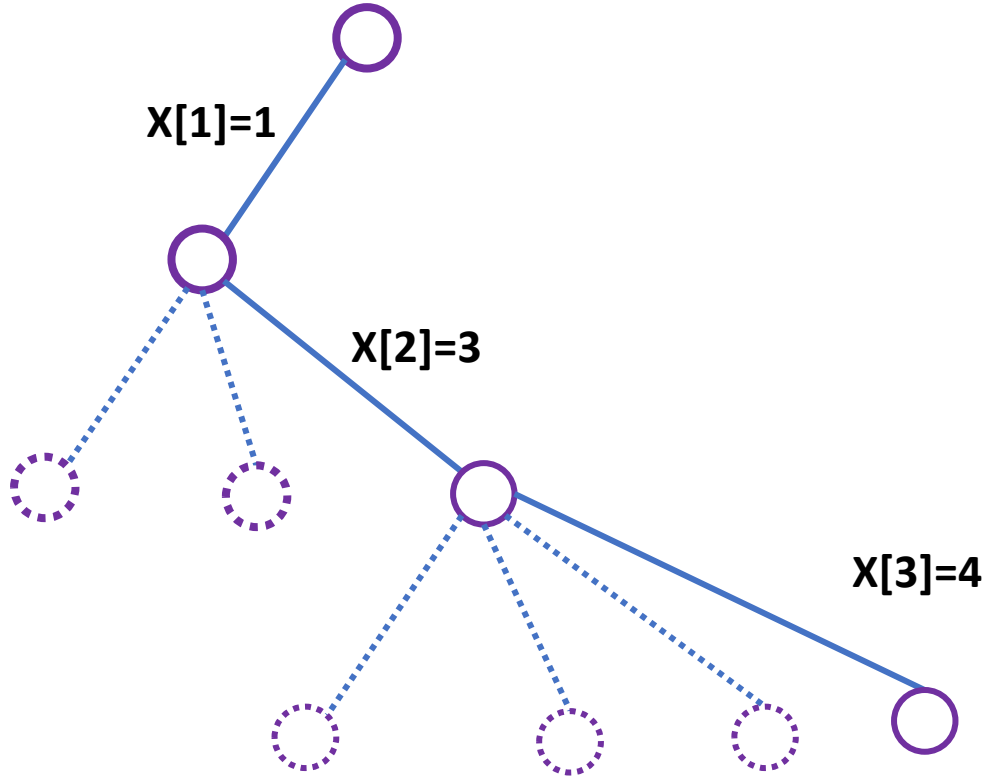
8-Queen Problem





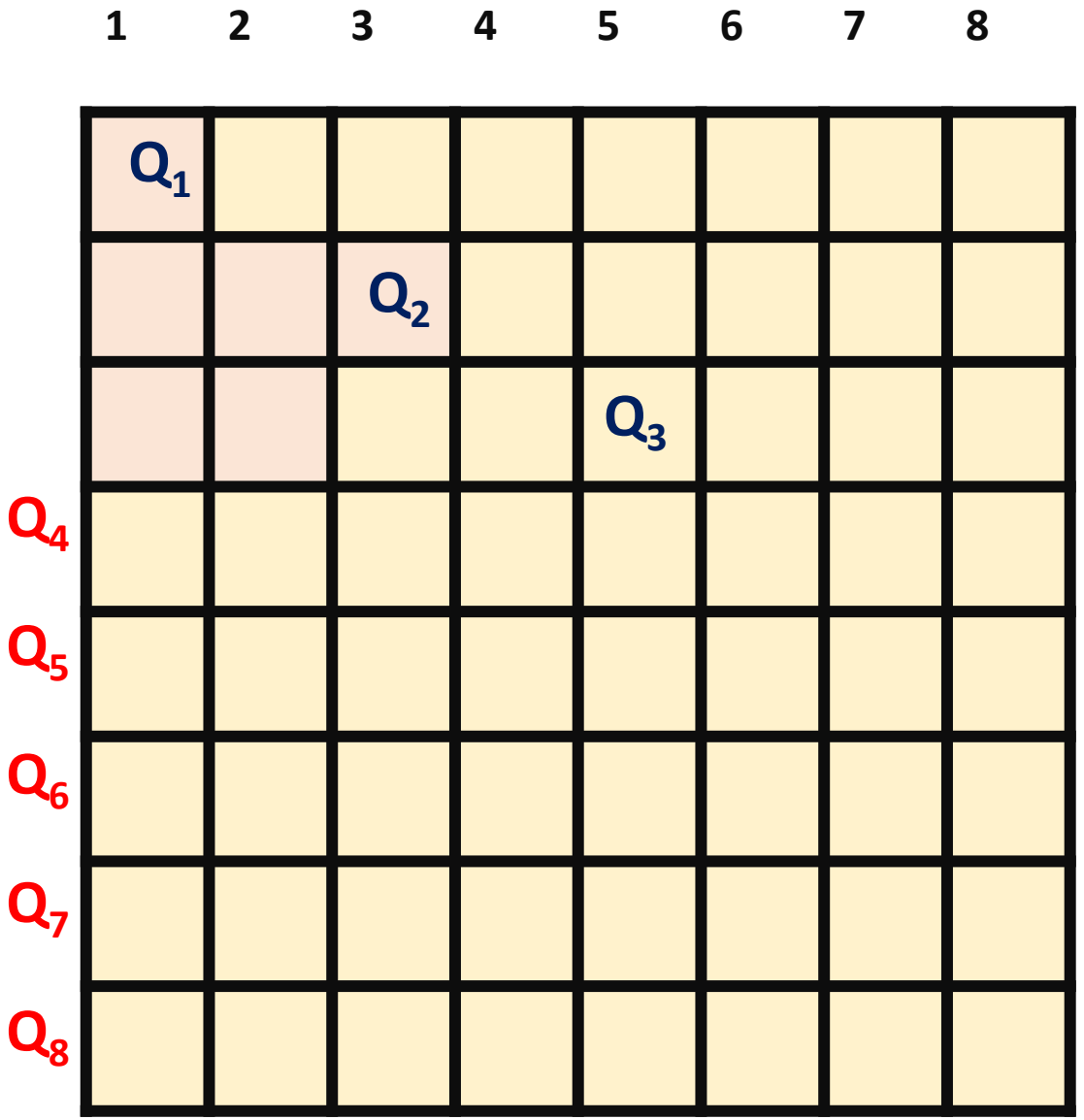
8-Queen Problem

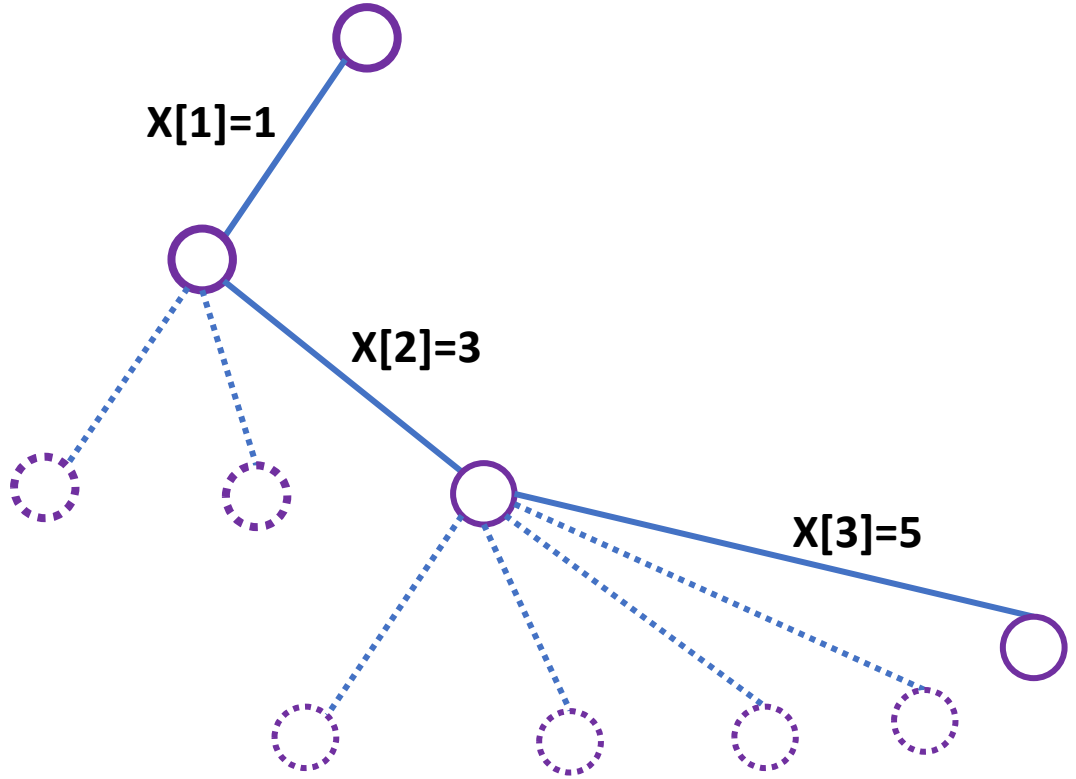




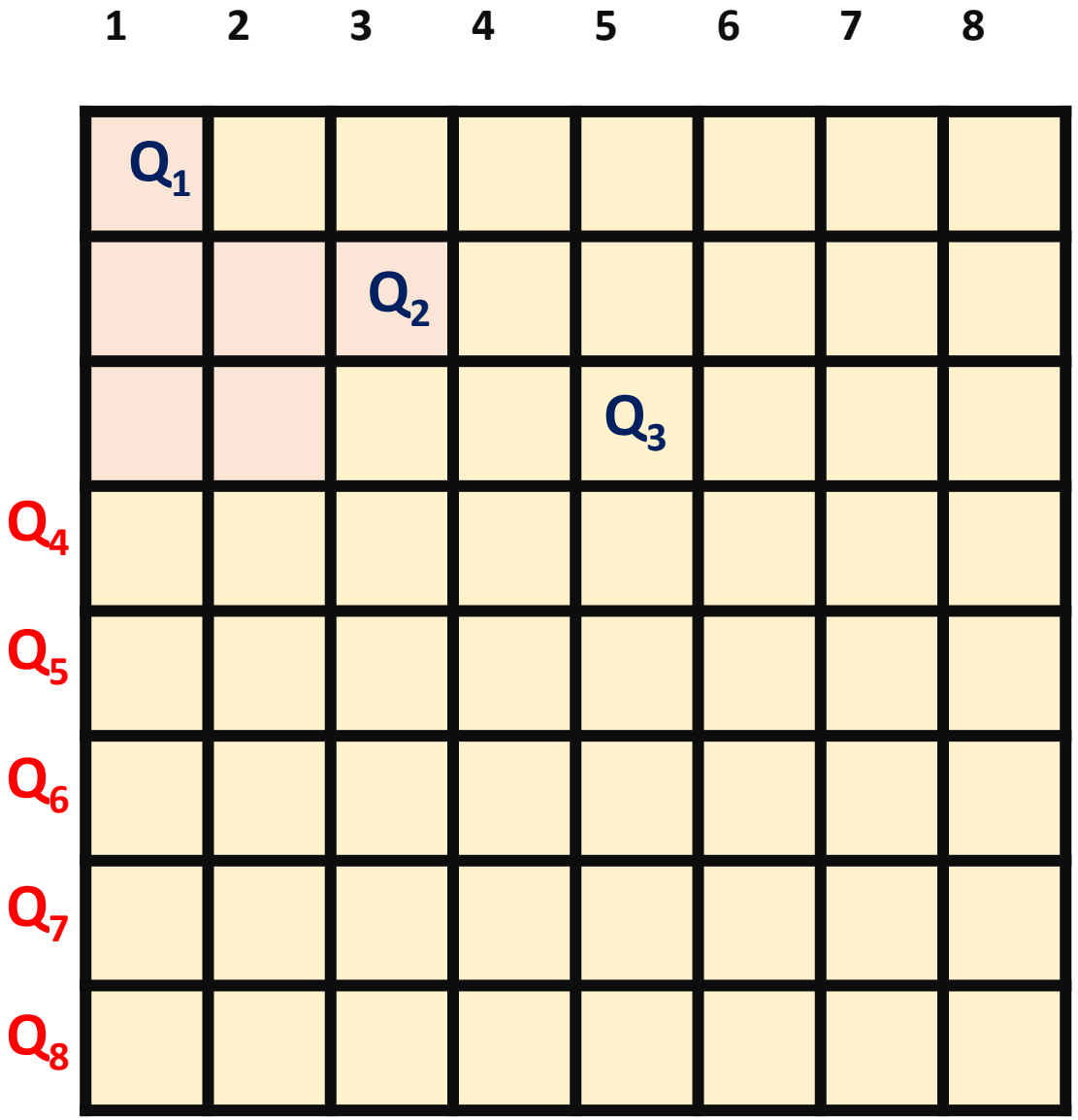
Diagonal Attack

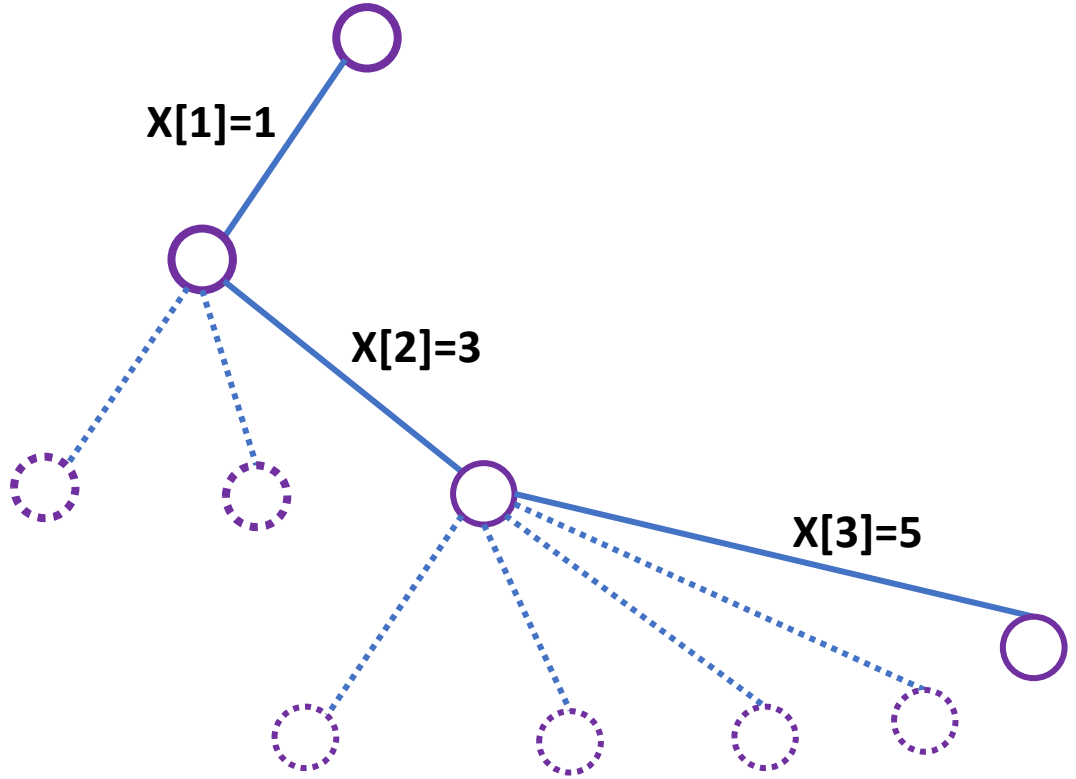
8-Queen Problem





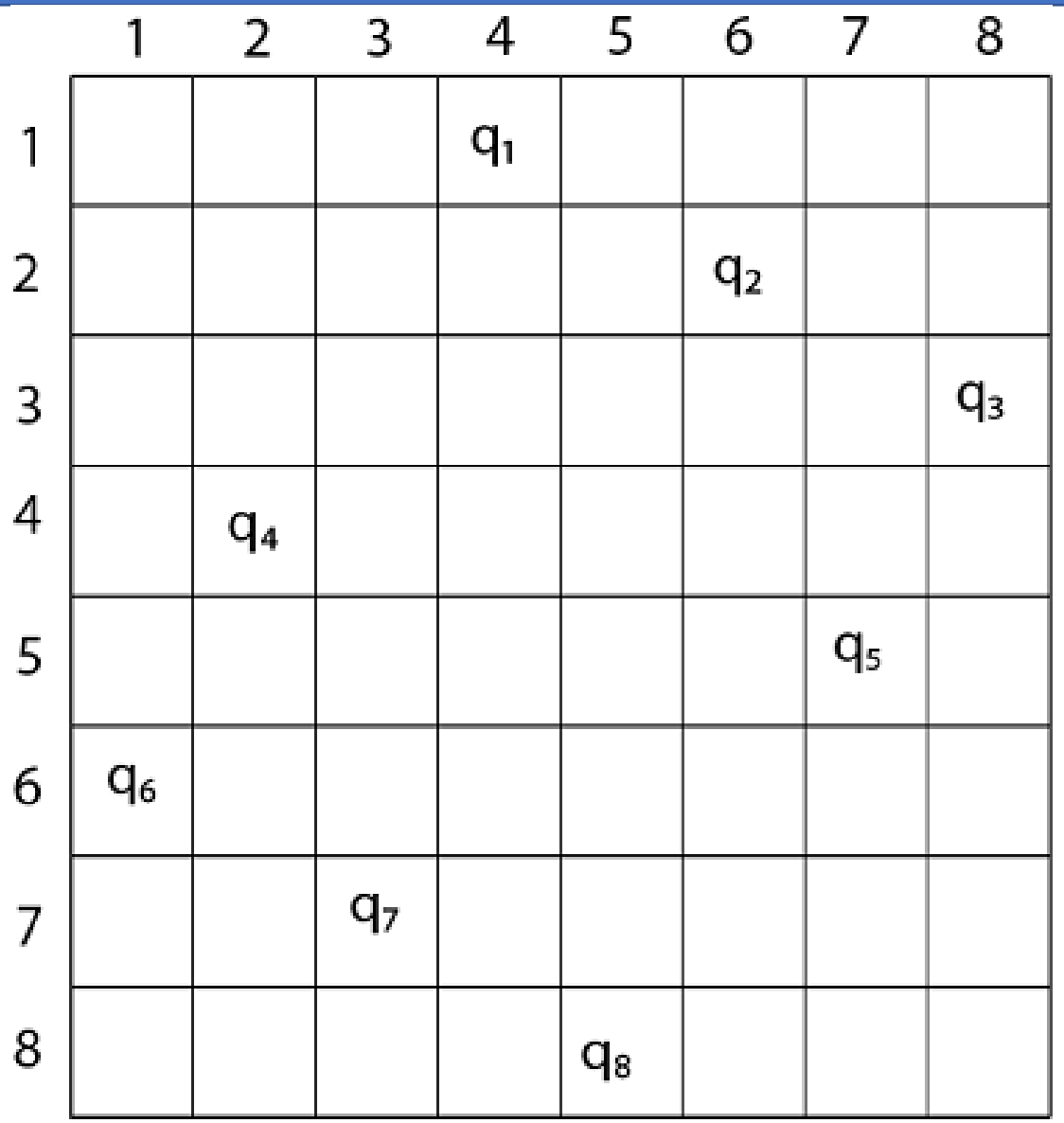
8-Queen Problem



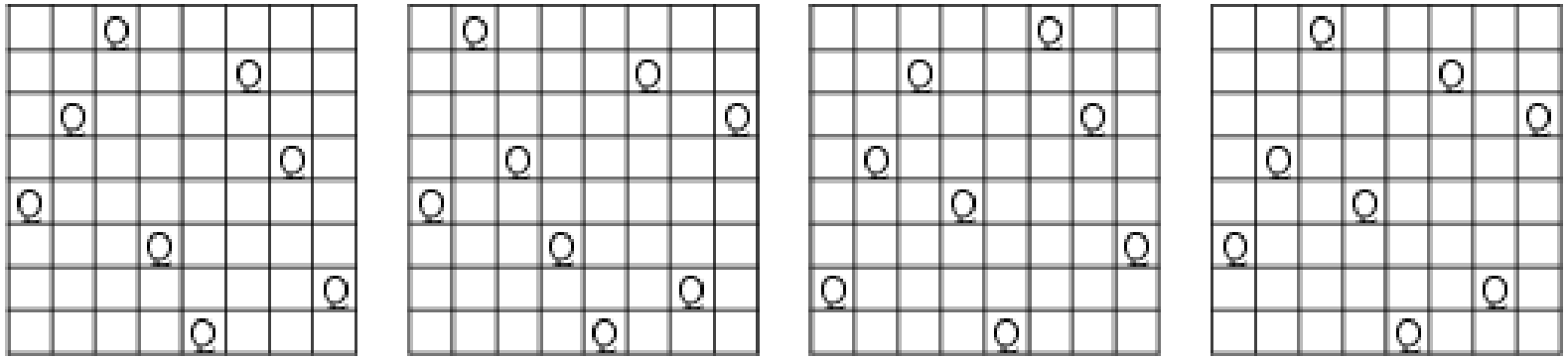


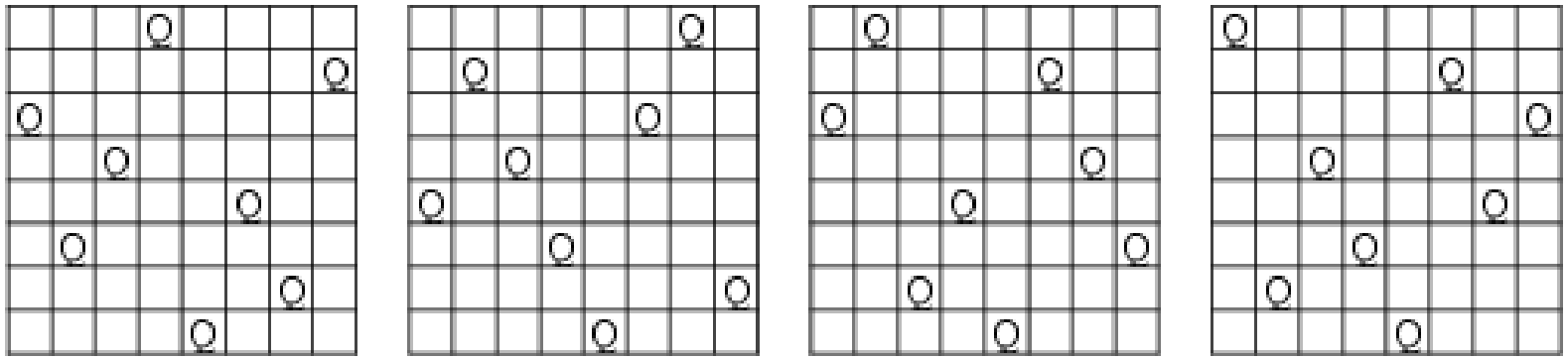
Continue in similar fashion

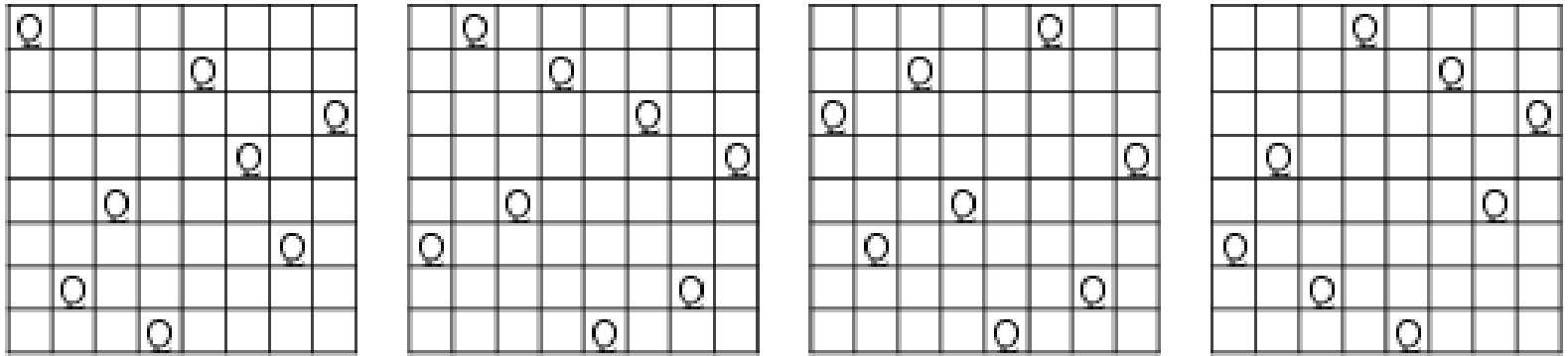
8-Queen Problem



8-Queen Problem



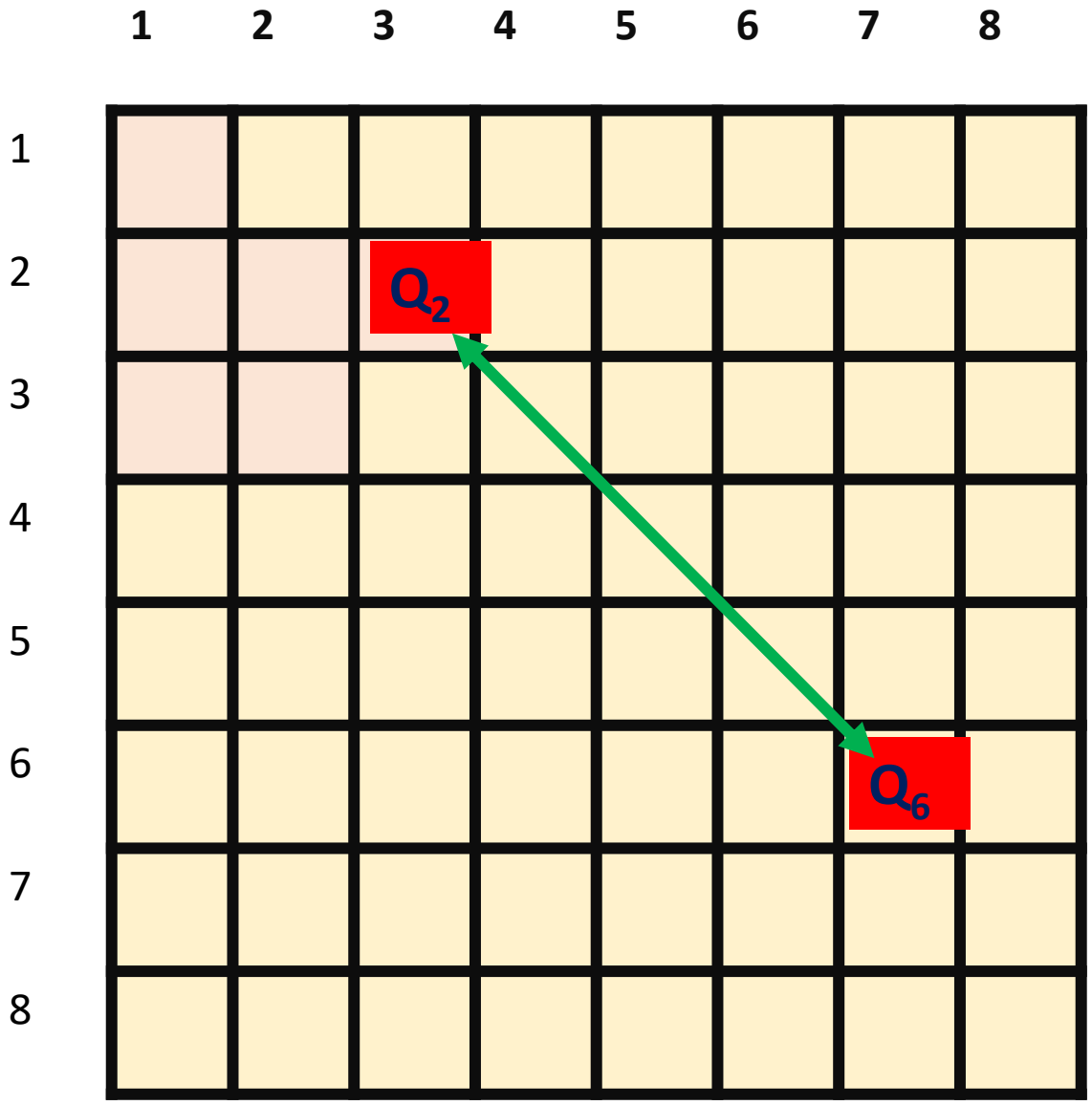




12 Unique Solutions

Total number of Solutions = 92

How to check Diagonal Attack ?



$Q_2 = (2, 3) = (i, x[i])$

$Q_6 = (6, 7) = (j, x[j])$

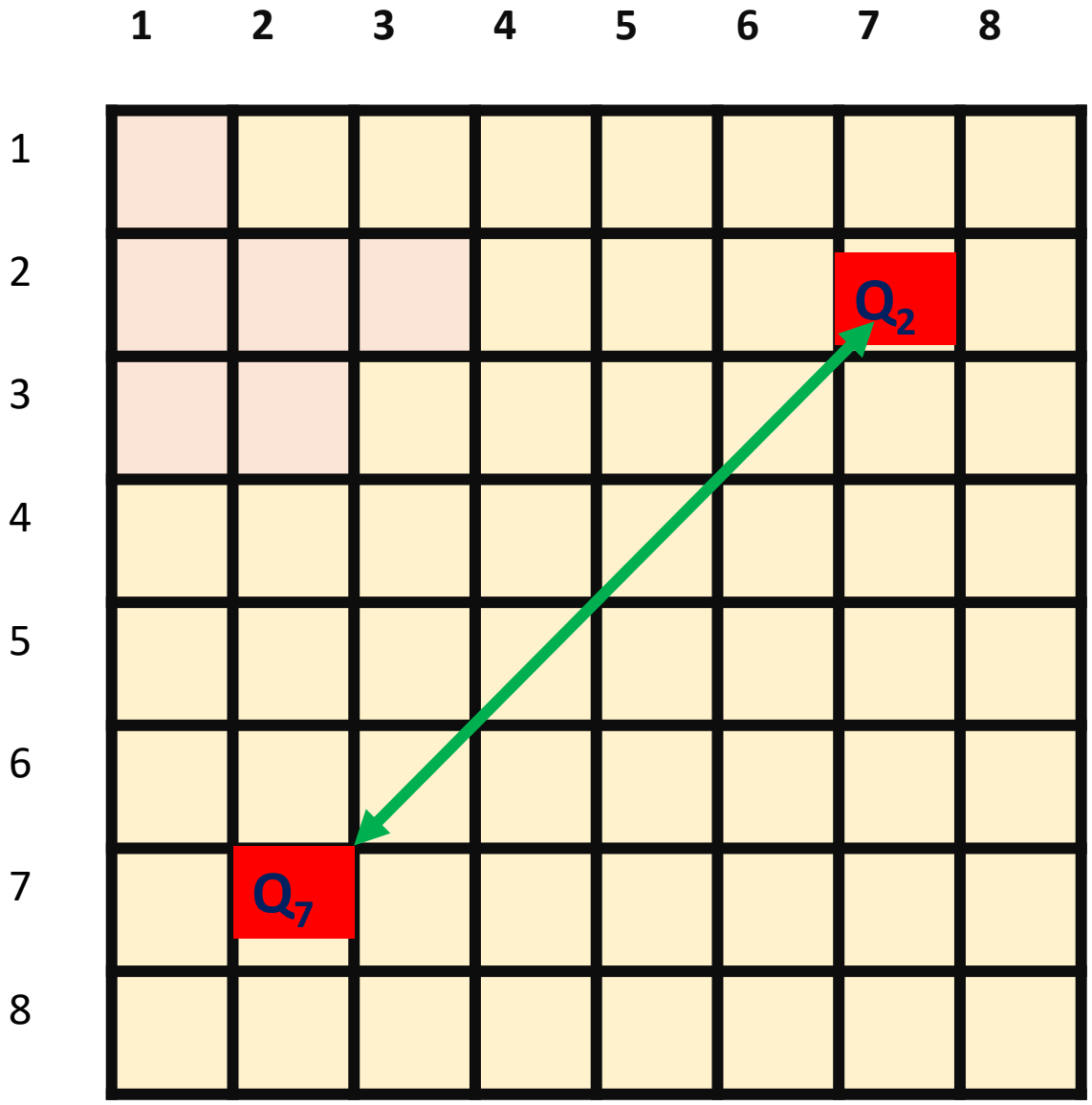
$|i - j| = 4 = |x[i] - x[j]|$

Abs Row Difference

=

Abs Column Difference

How to check Diagonal Attack ?



$Q_2 = (2, 7) = (i, x[i])$

$Q_7 = (7, 2) = (j, x[j])$

$|i - j| = 5 = |X[i] - x[j]|$

Abs Row Difference

=

Abs Column Difference

How to check Diagonal Attack ?

$$|i - j| = |x[i] - x[j]|$$

Abs Row Difference = Abs Column Difference

Diagonal Attack

Algorithm

```
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  }
```

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  }
```

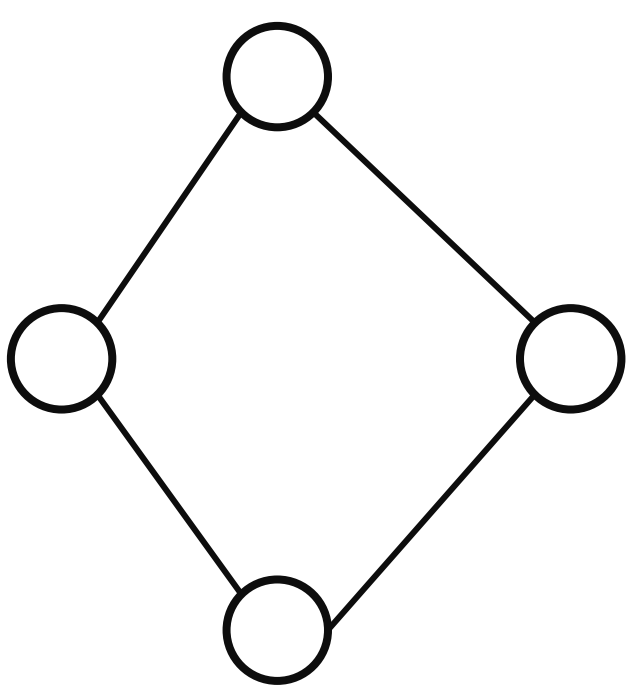
8-Queen Problem

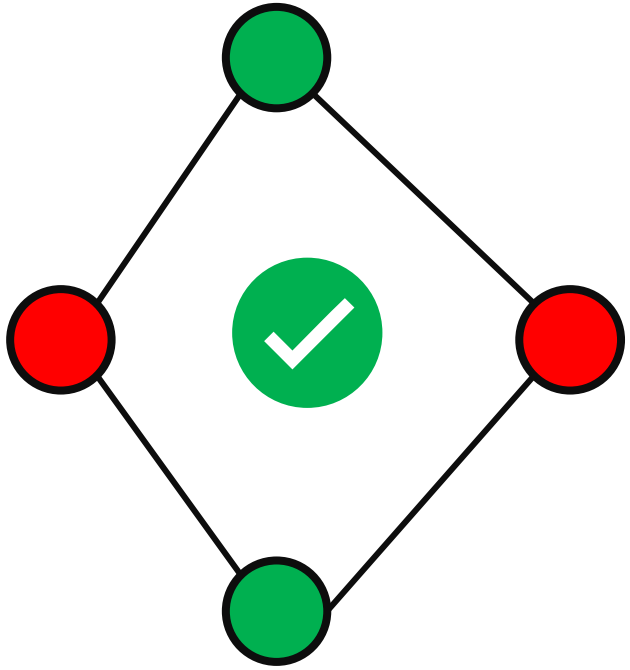
Time Complexity = Exponential

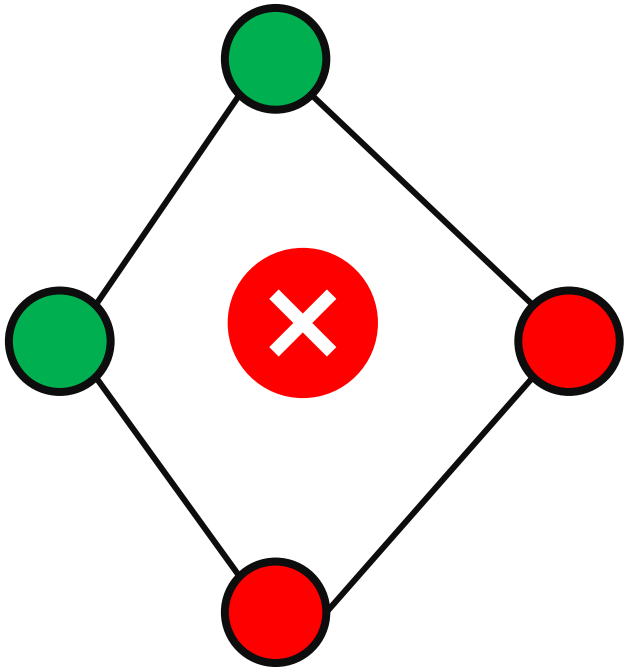
Graph Coloring Problem

Basics:

Proper Coloring: Coloring the given graph so that **no** 2 adjacent vertices get the same color



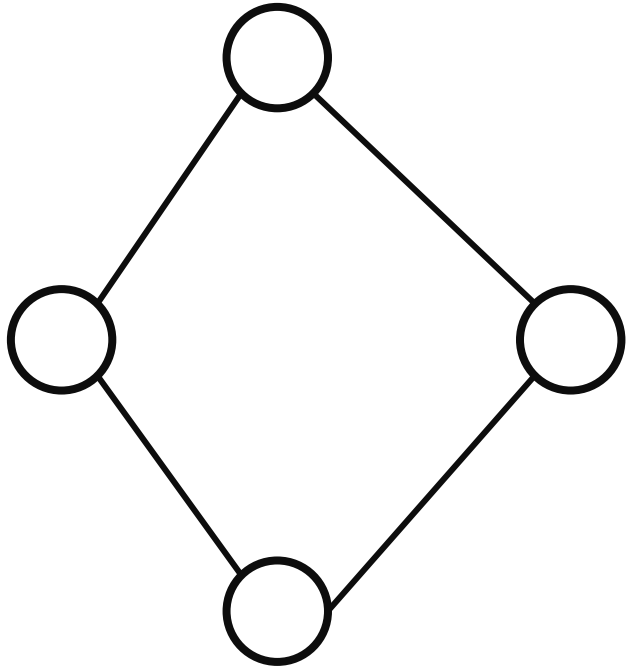


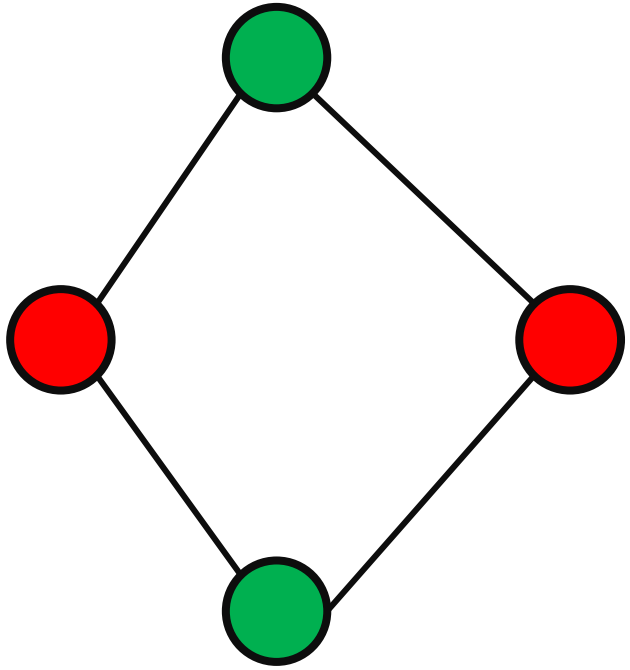


Graph Coloring Problem

Basics:

Chromatic Number: Minimum number of colors required to properly color the given graph



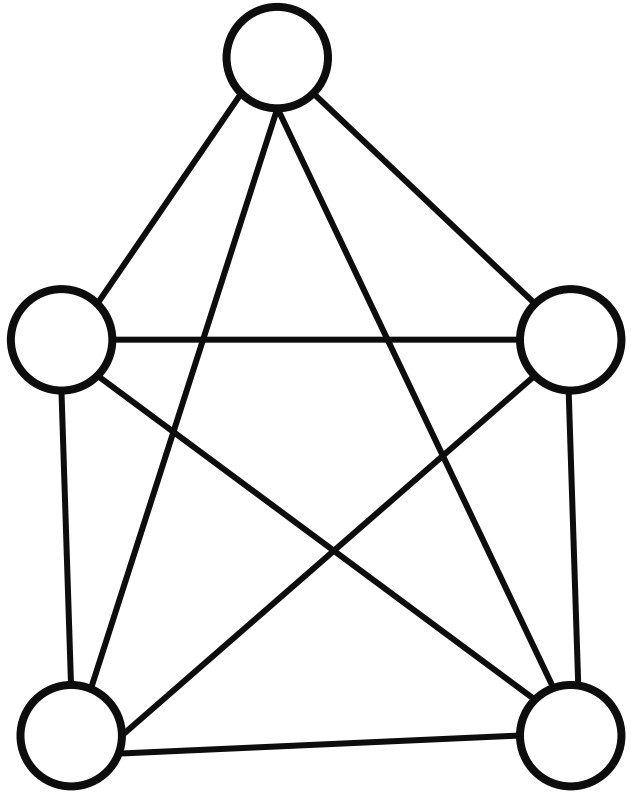


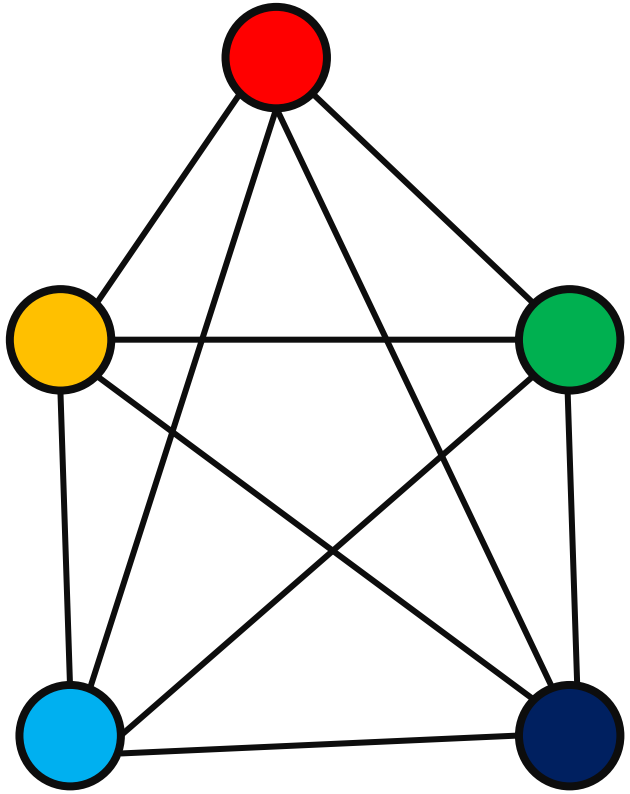
Chromatic number = **2**

Graph Coloring Problem

Basics:

Chromatic Number: Minimum number of colors required to properly color the given graph





Chromatic number = **5**

Graph Coloring Problem

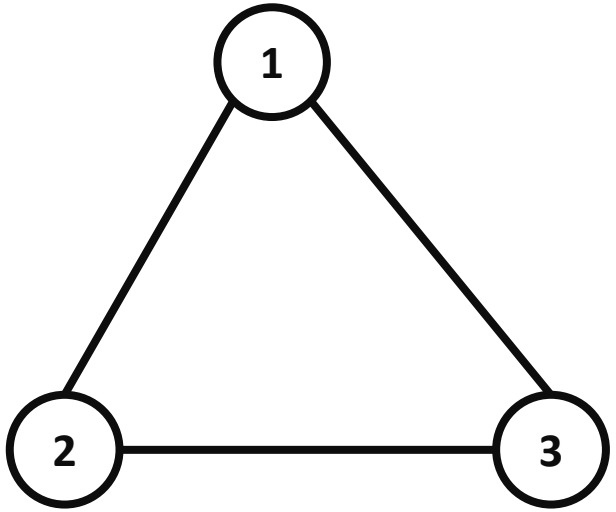
Given:

An undirected graph and a number m

Goal:

To color the given graph with *at most m* colors such that no two adjacent vertices of the graph are colored with the same color

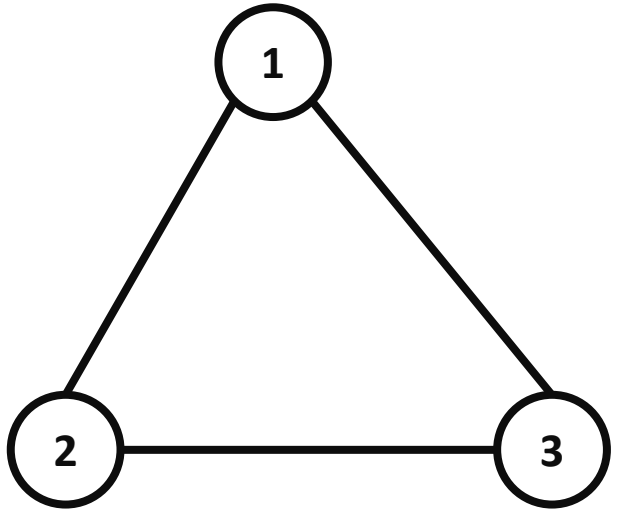
Graph Coloring Problem



m = 3 colors

< Red, Green, Blue >

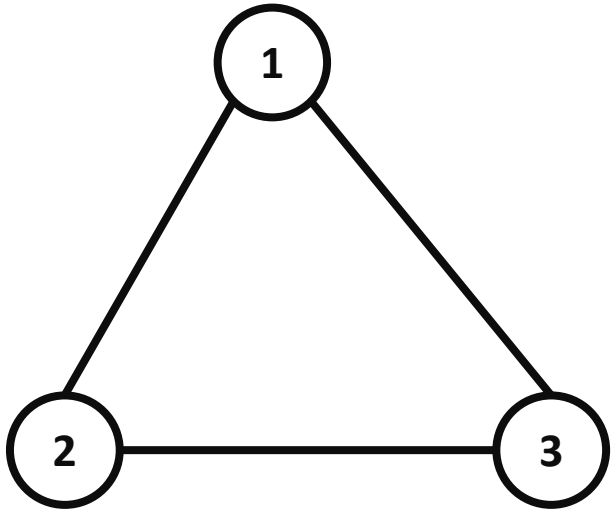
Graph Coloring Problem



m = 3 colors

< Red = 1, Green = 2, Blue = 3>

Graph Coloring Problem



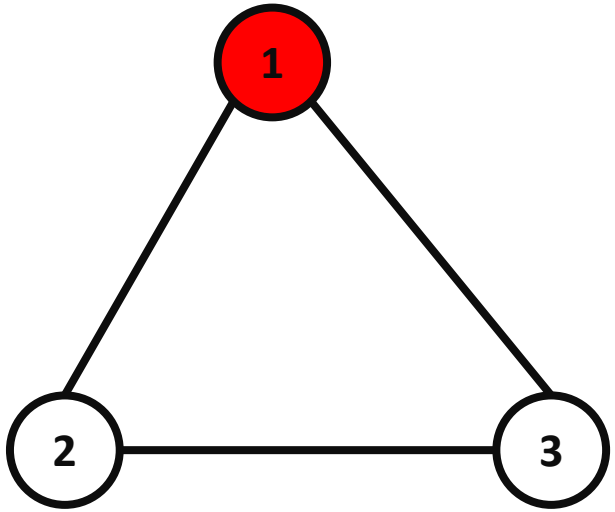
< Red = 1, Green = 2, Blue = 3>

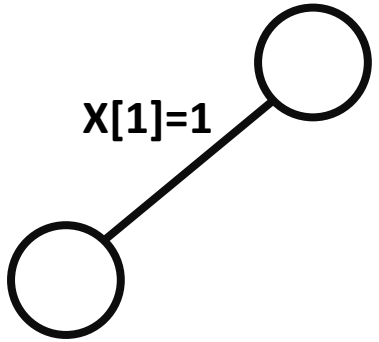
The Solution is : < x[1], x[2], x[3] >

x[i] = the color of vertex i $\forall\, i \in \{1, 2, 3\}$

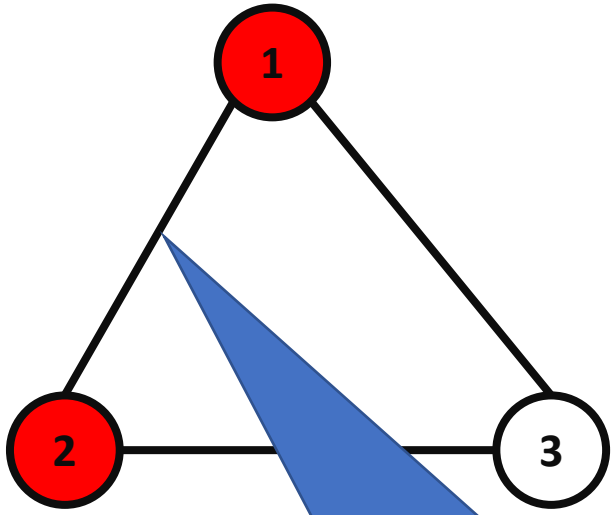
$x[i] \in \{Red = 1, \quad Green = 2, Blue = 3\} \forall\, i \in \{1, 2, 3\}$

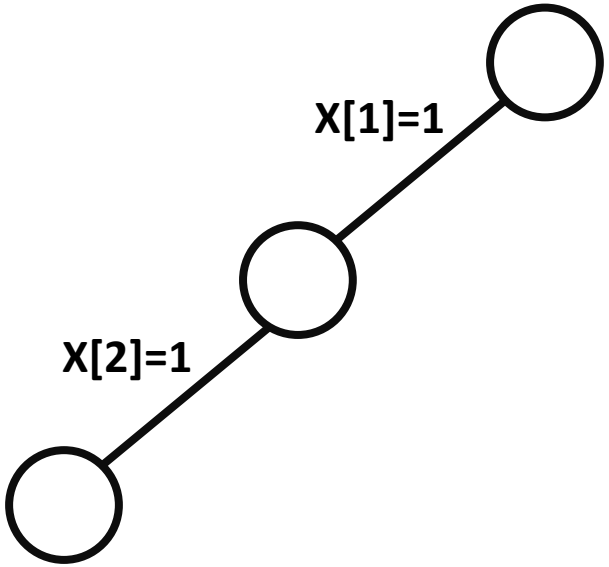
Graph Coloring Problem





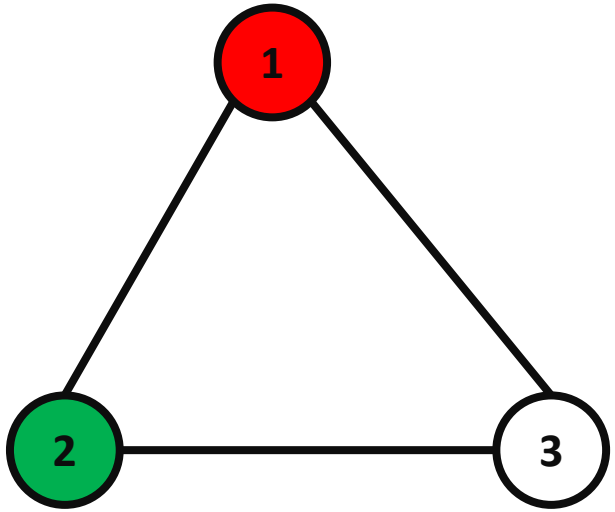
Graph Coloring Problem

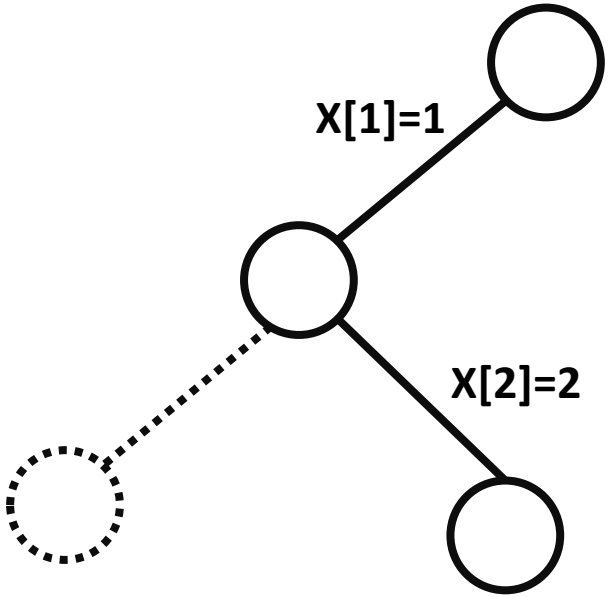




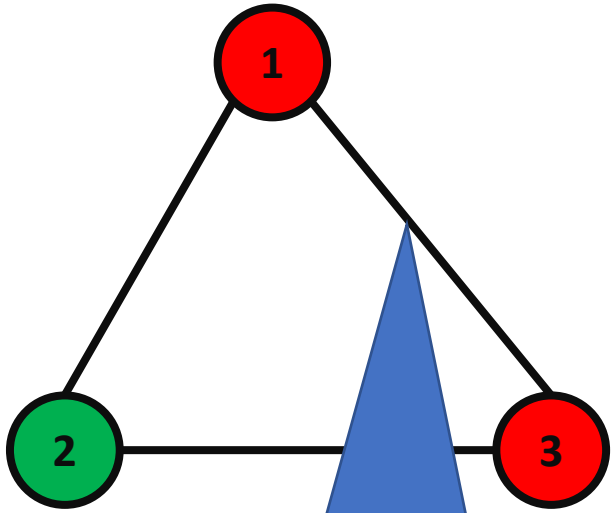
Vertex 1 and 2 are adjacent so they can't have same color

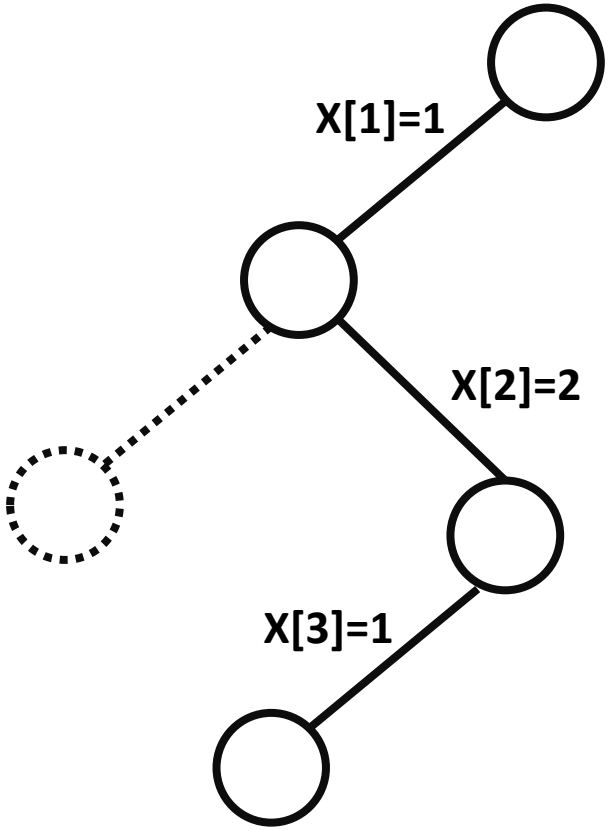
Graph Coloring Problem





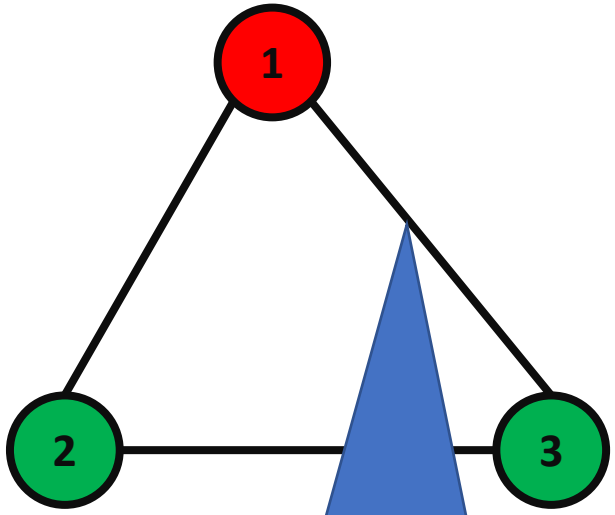
Graph Coloring Problem

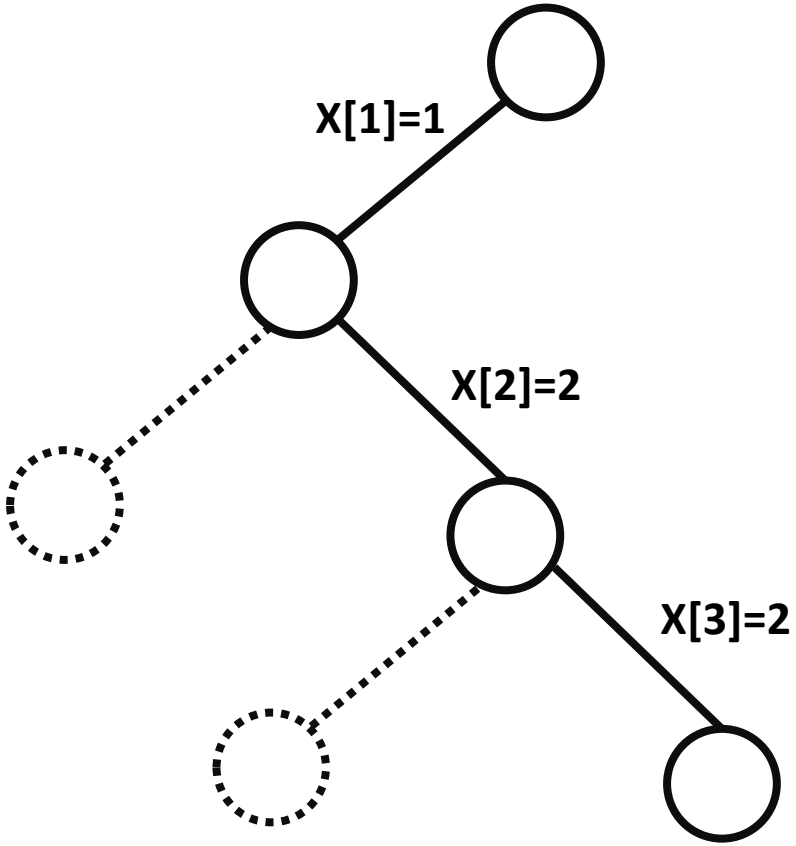




Vertex 1 and 3 are adjacent so they can't have same color

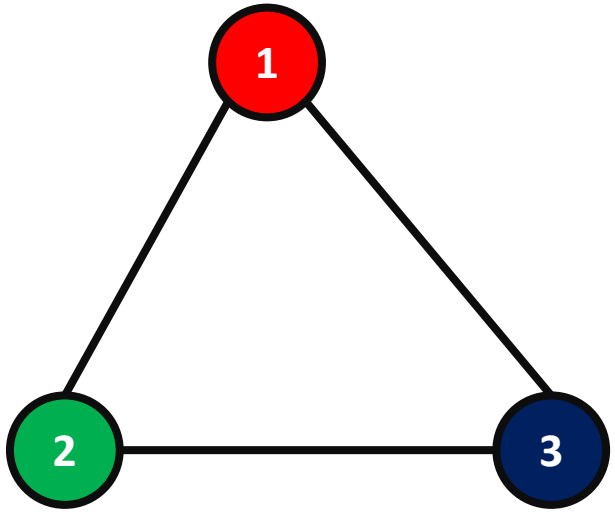
Graph Coloring Problem

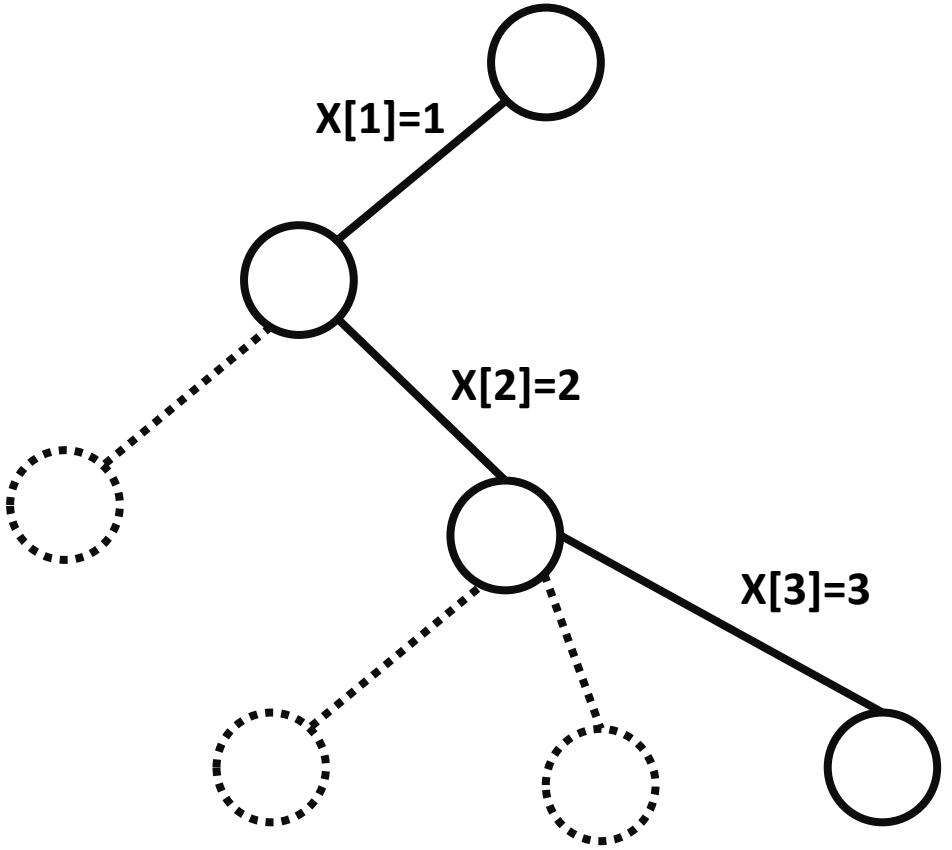




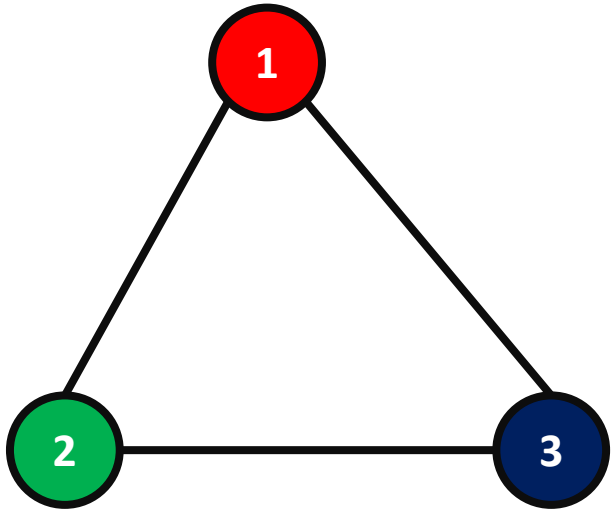
Vertex 2 and 3 are adjacent so they can't have same color

Graph Coloring Problem





Graph Coloring Problem



X[1]	X[2]	X[3]
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Graph Coloring Problem

Time Complexity = Exponential

Graph Coloring Problem

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, . . . , m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Graph Coloring Problem

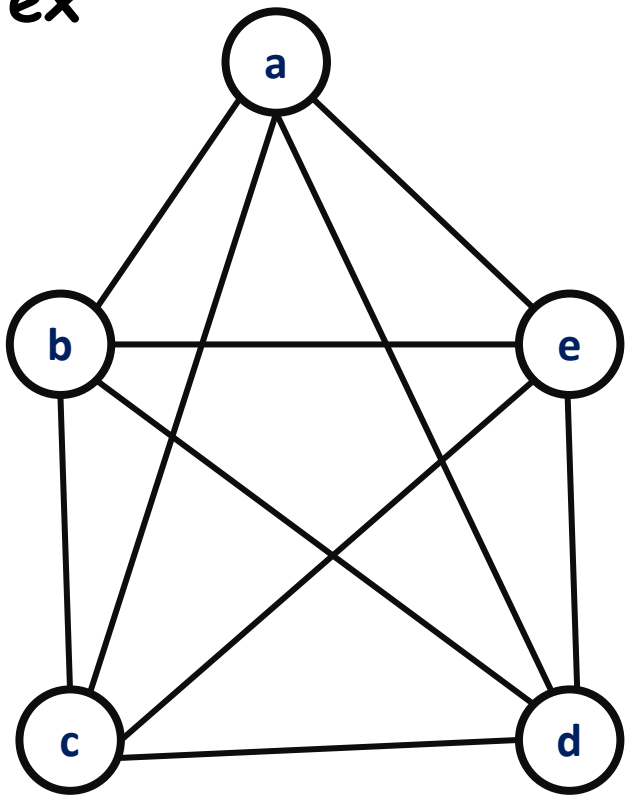
```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k − 1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

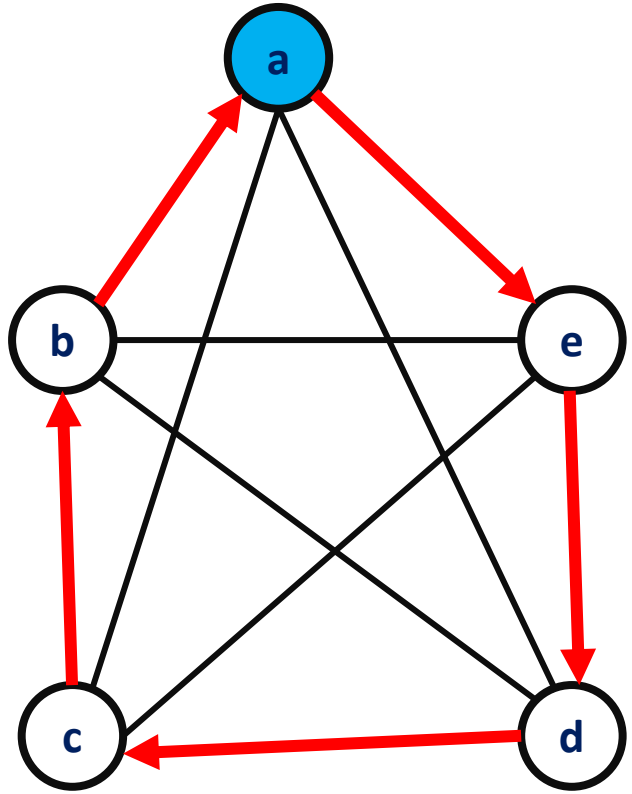
Hamiltonian Cycle Problem

Basic:

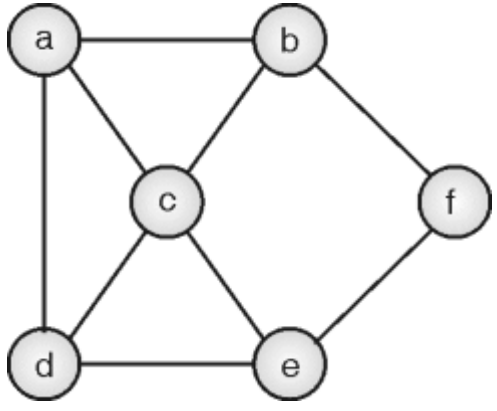
Hamiltonian Cycle:

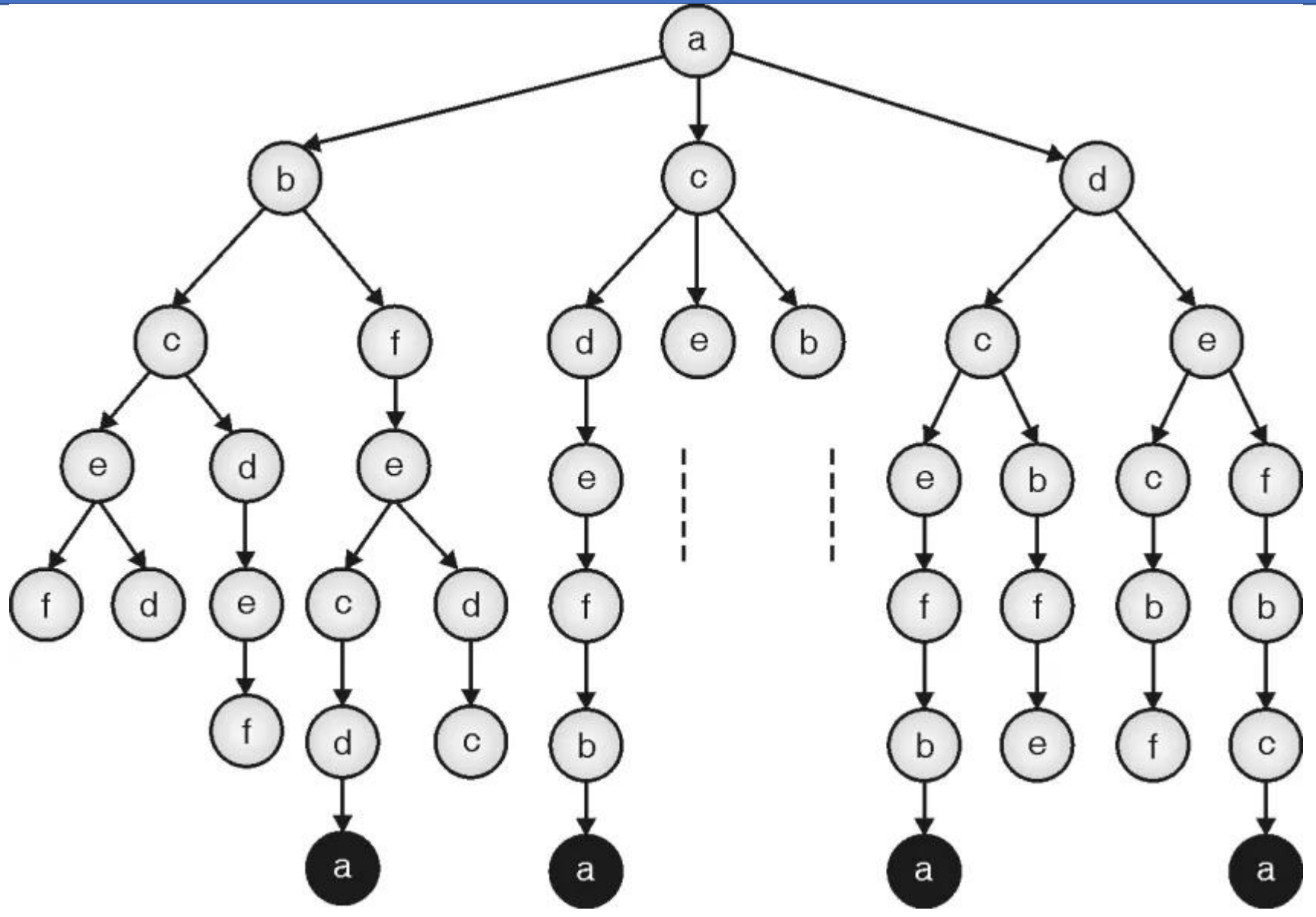
A cycle in a graph which cover all the vertices exactly once except the source vertex





Hamiltonian Cycle Problem





Hamiltonian Cycle Problem

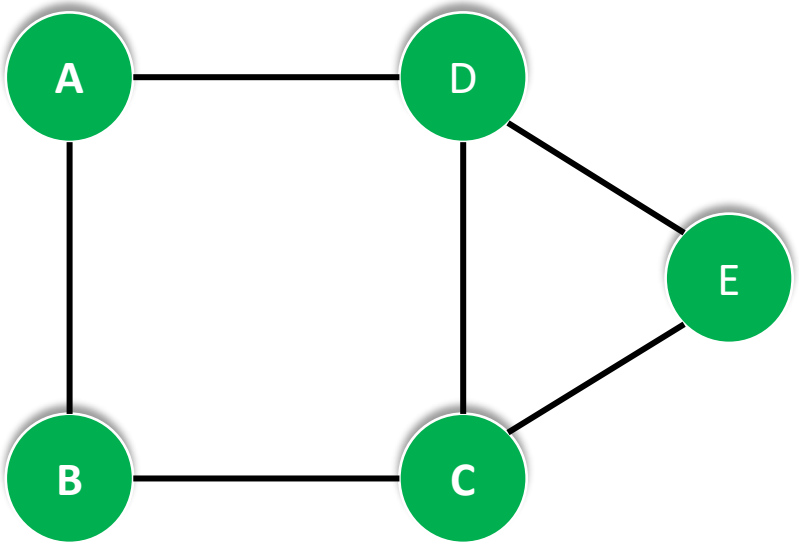
Given:

A graph is given

Goal:

Find Hamiltonian cycle in given graph

Hamiltonian Cycle Problem

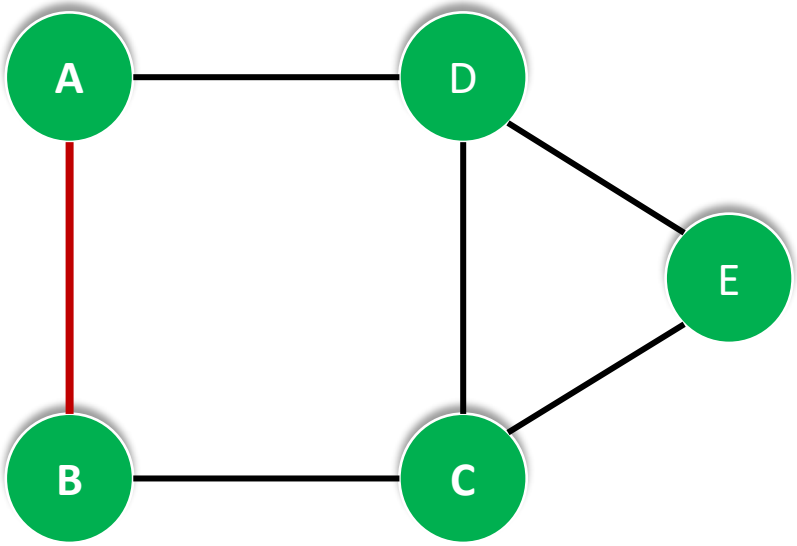


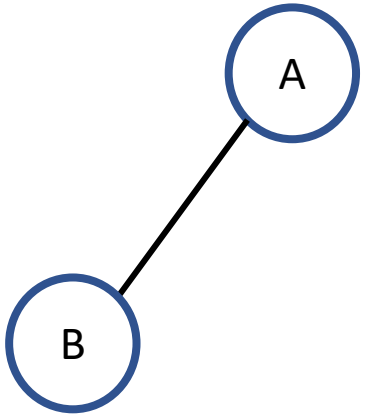
Solution= < x[1], x[2], x[3], x[4], x[5], x[6]] >

$$x[i] \in \{A, B, C, D, E\} \; \forall \; i \in \{1, 2, 3, 4, 5, 6\}$$

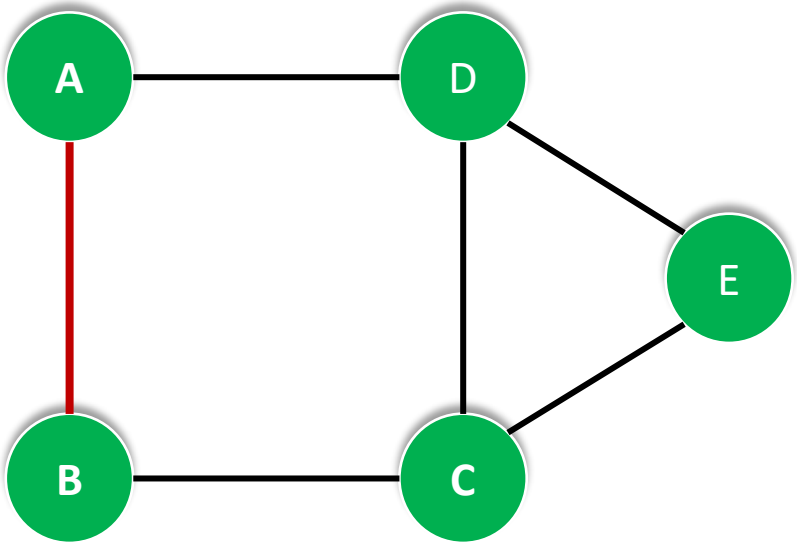
Note : Lexicographical Order to visit the vertices

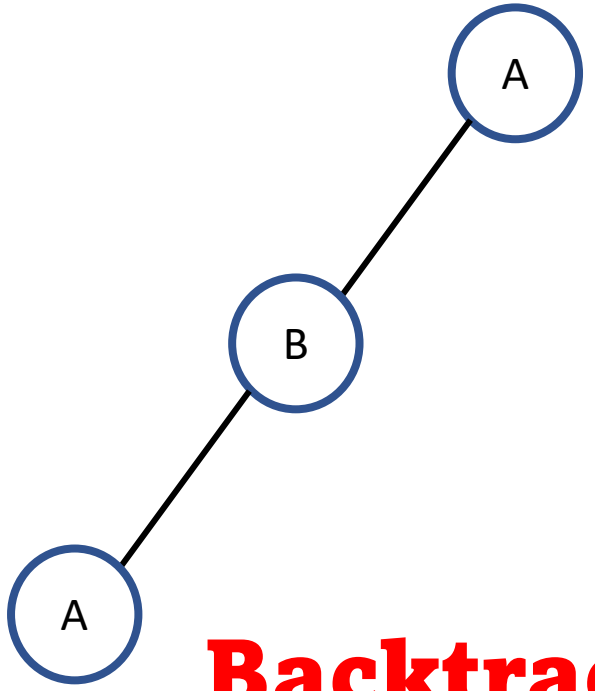
Hamiltonian Cycle Problem





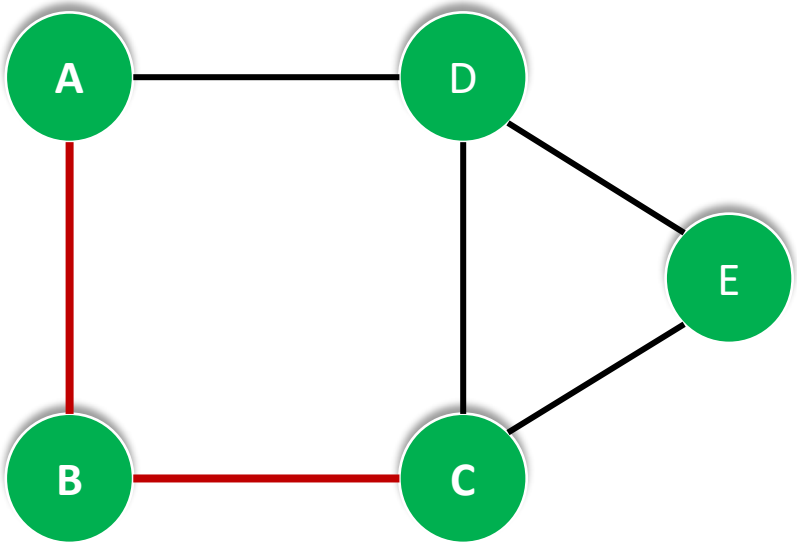
Hamiltonian Cycle Problem

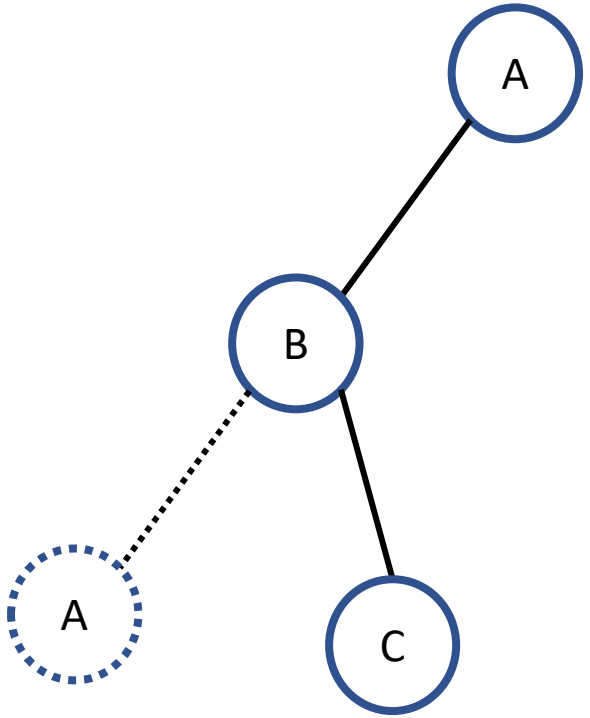




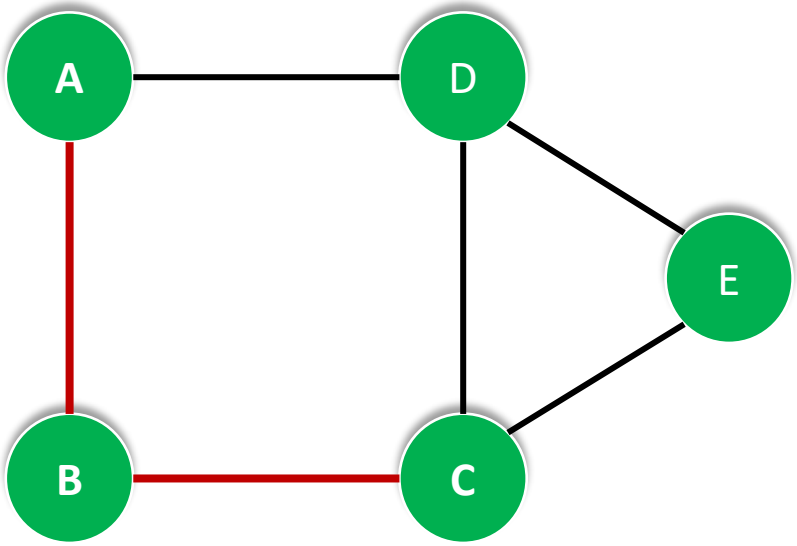
Backtrack as all other vertices are not covered in tour

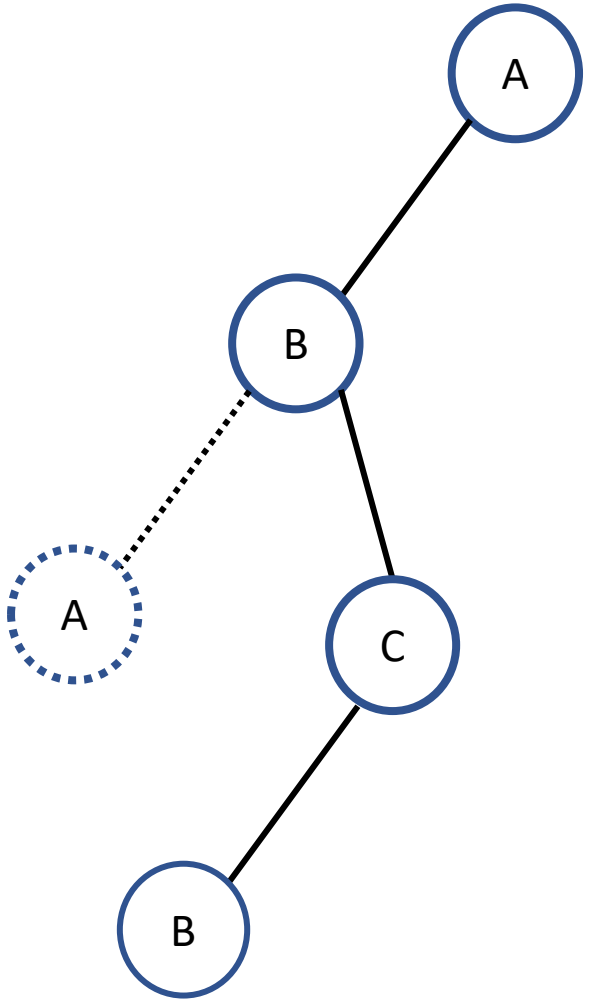
Hamiltonian Cycle Problem



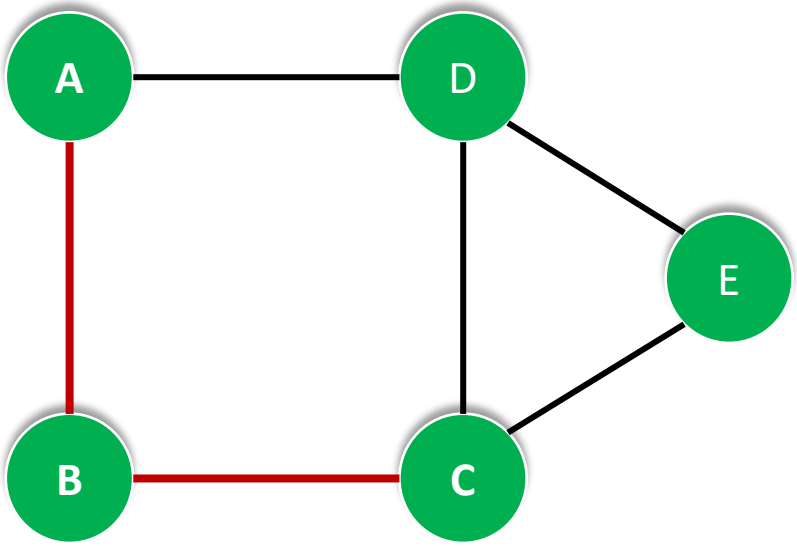


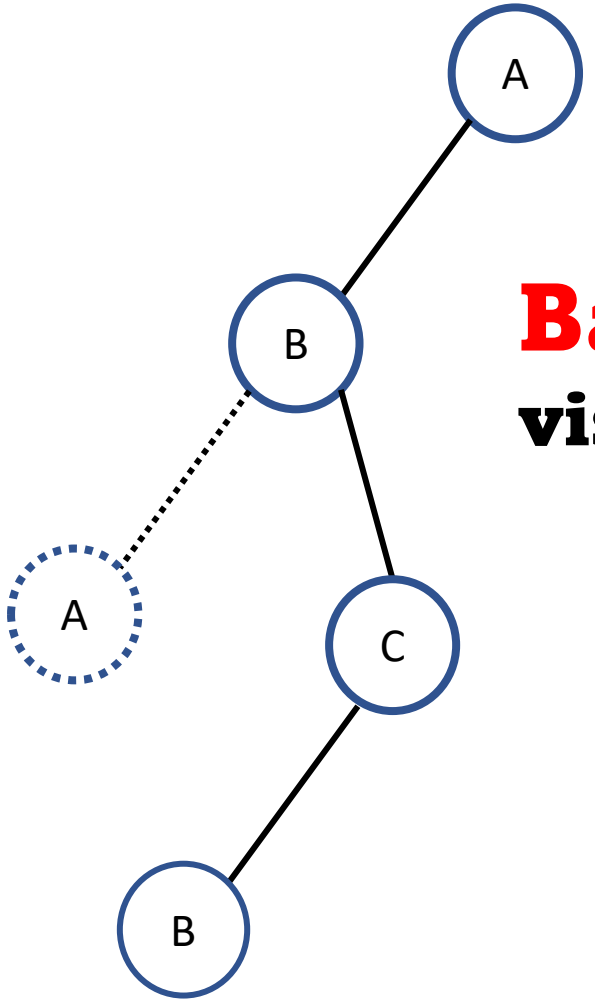
Hamiltonian Cycle Problem





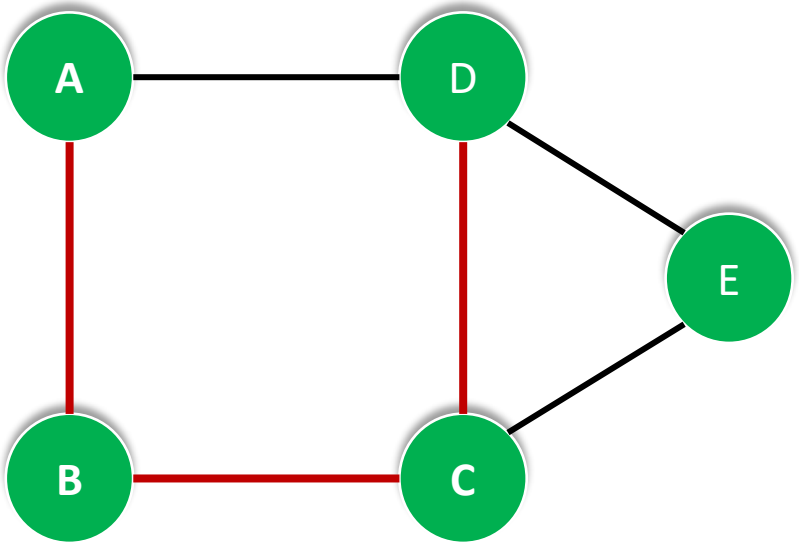
Hamiltonian Cycle Problem

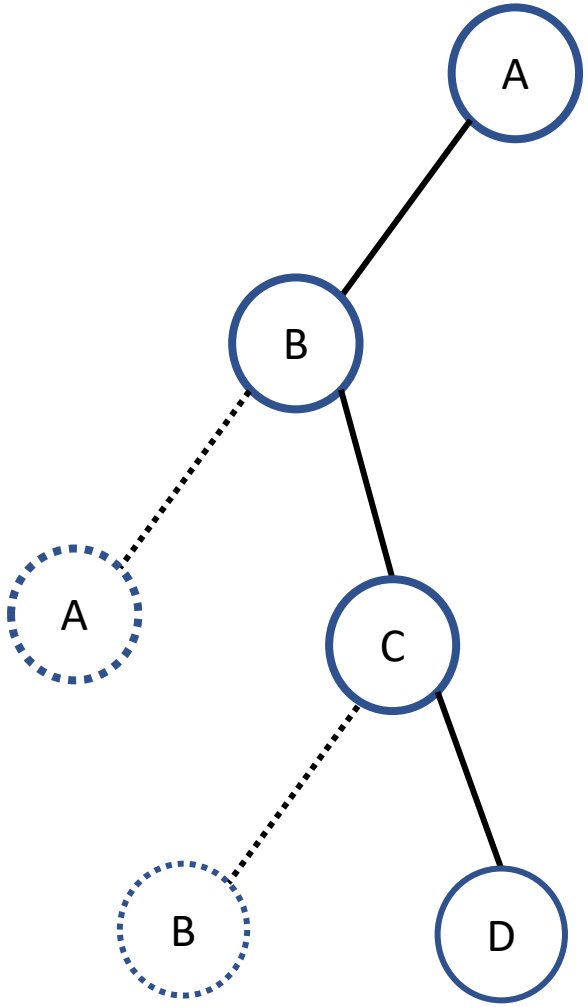




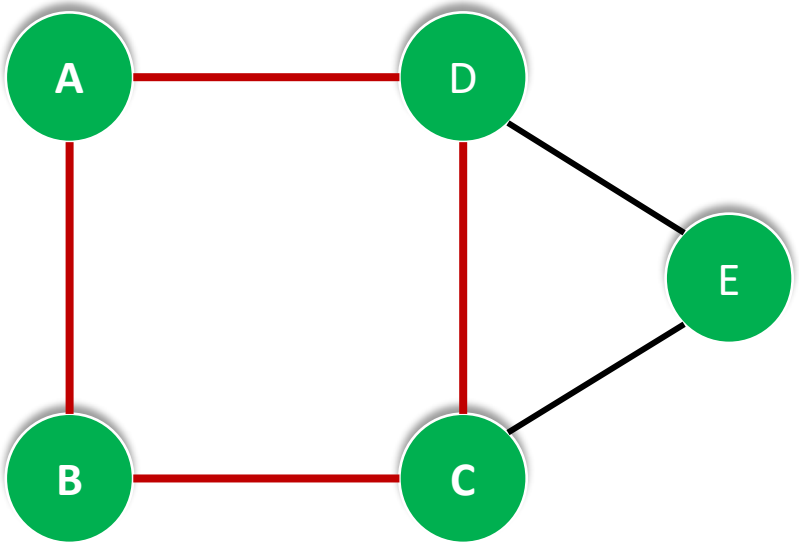
Backtrack **as B is visited twice**

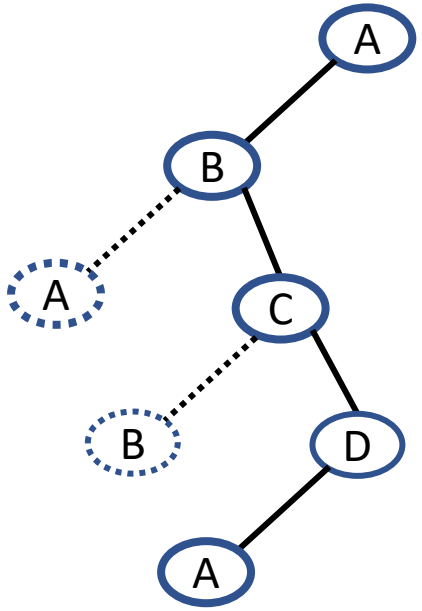
Hamiltonian Cycle Problem



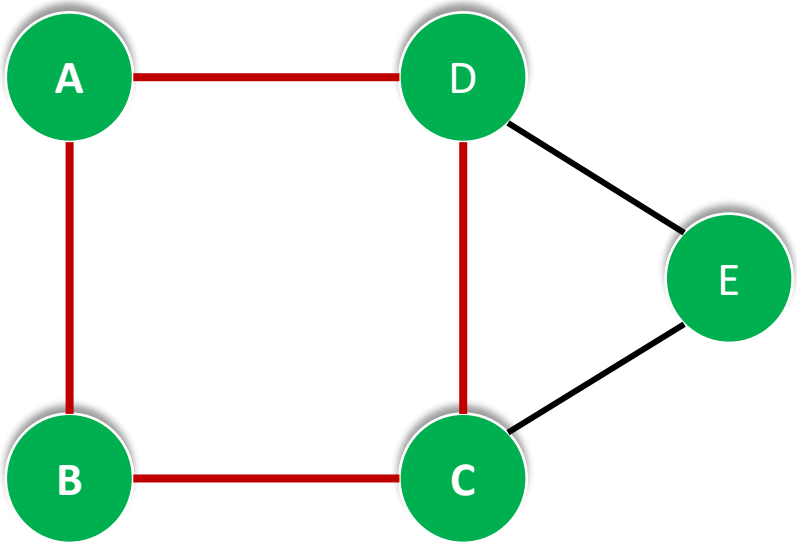


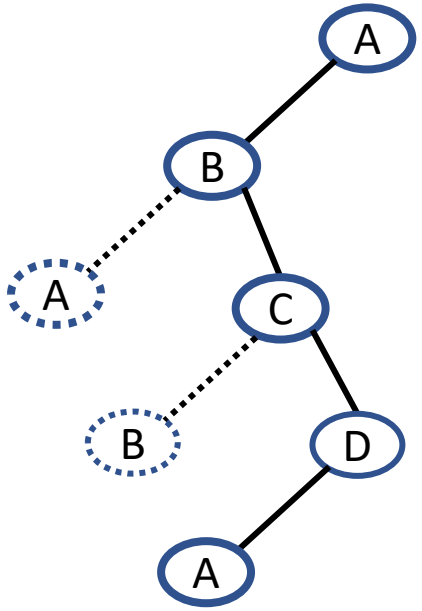
Hamiltonian Cycle Problem





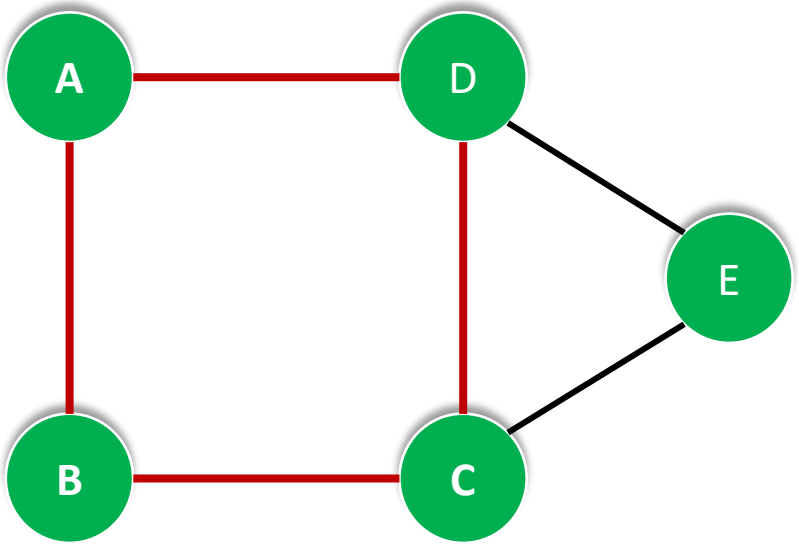
Hamiltonian Cycle Problem

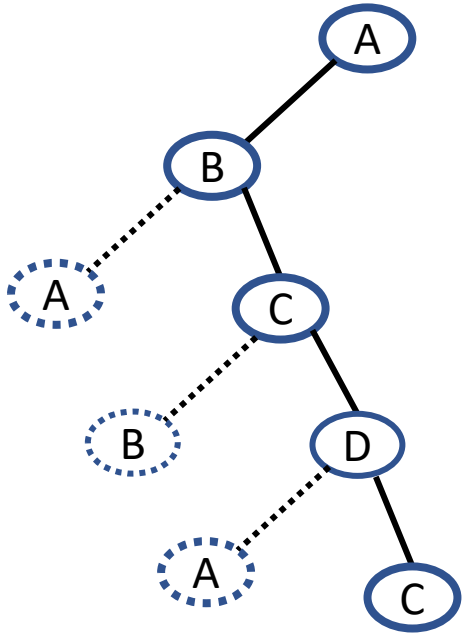




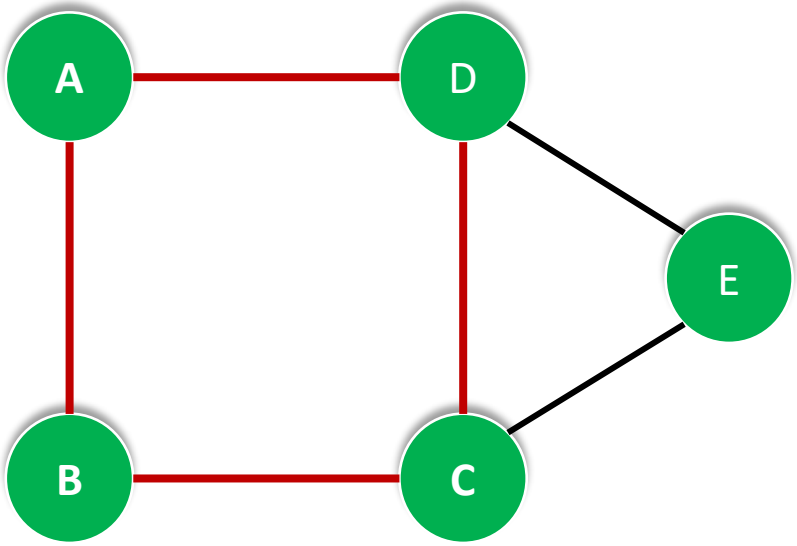
Backtrack as all other vertices are not covered in tour

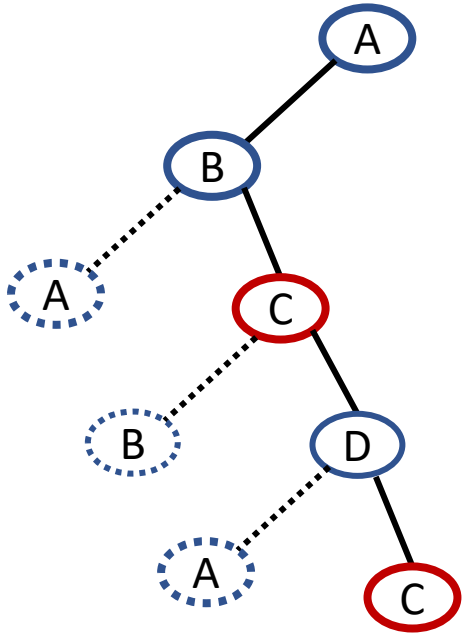
Hamiltonian Cycle Problem





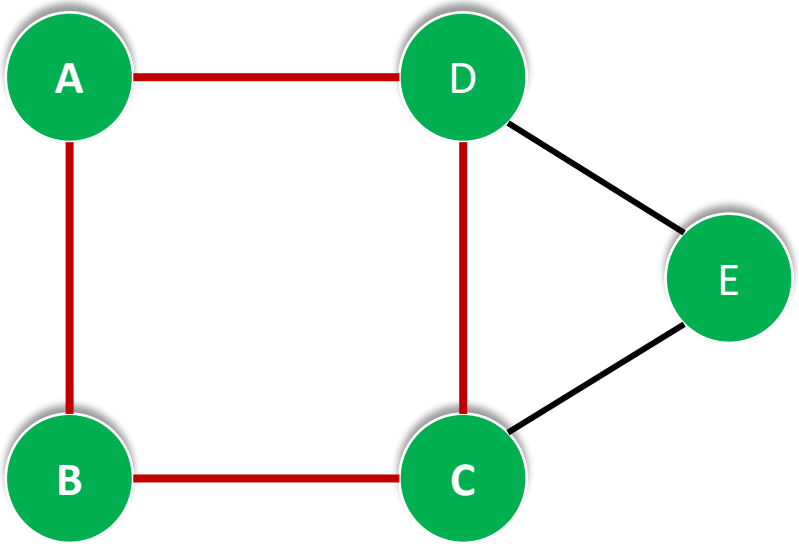
Hamiltonian Cycle Problem

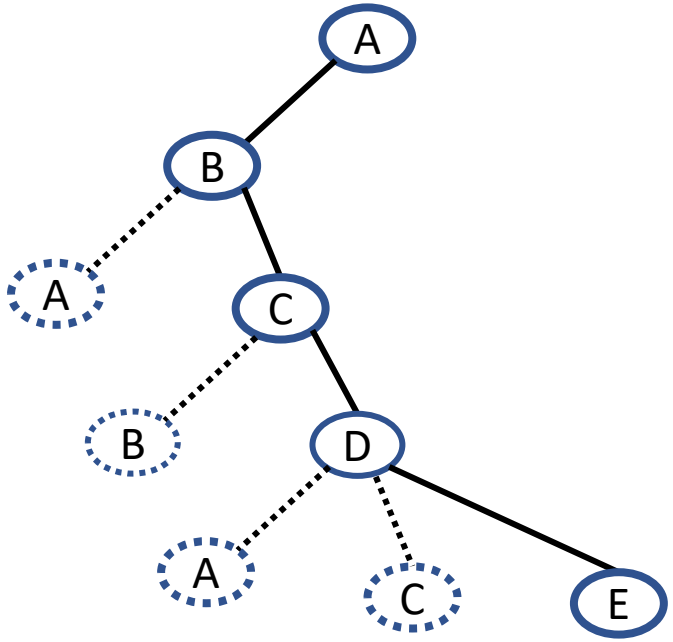




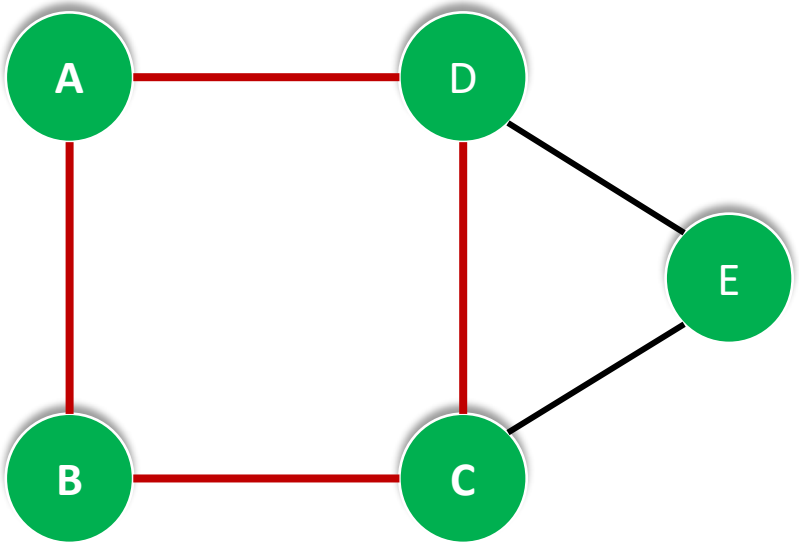
Backtrack **as C is visited twice**

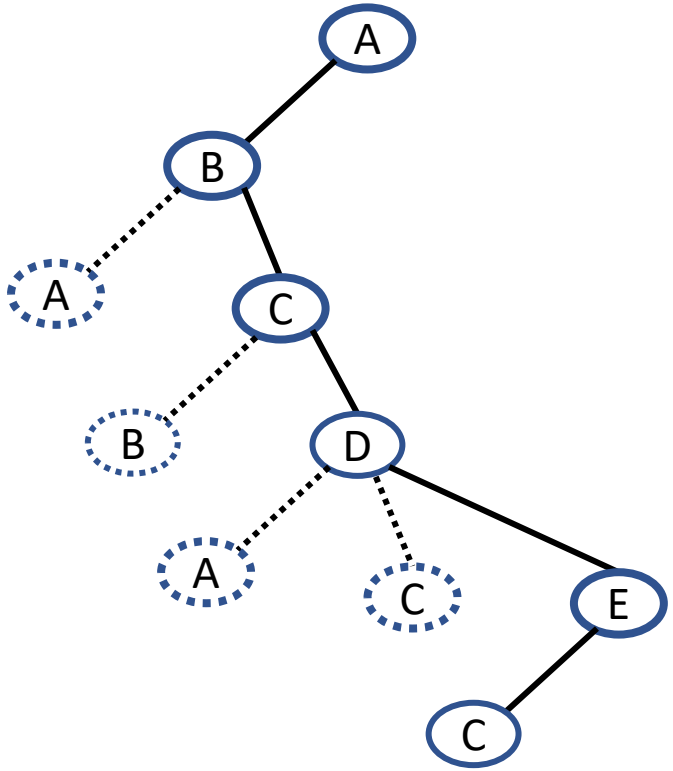
Hamiltonian Cycle Problem



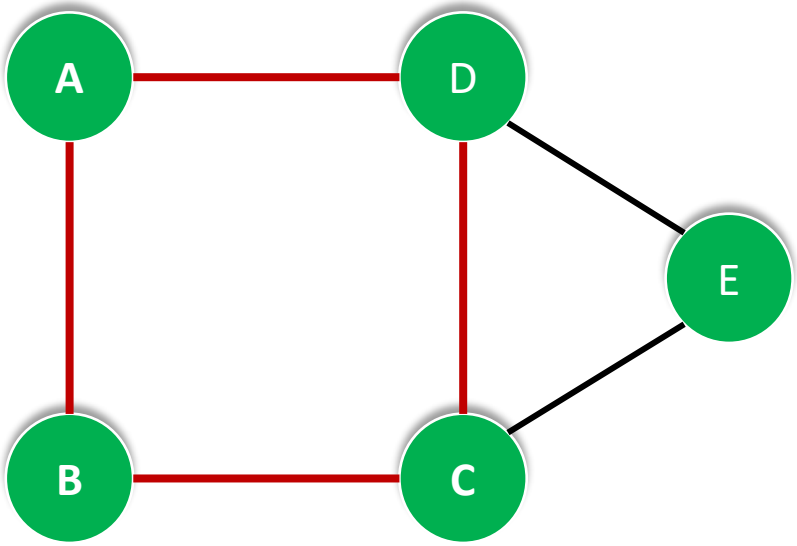


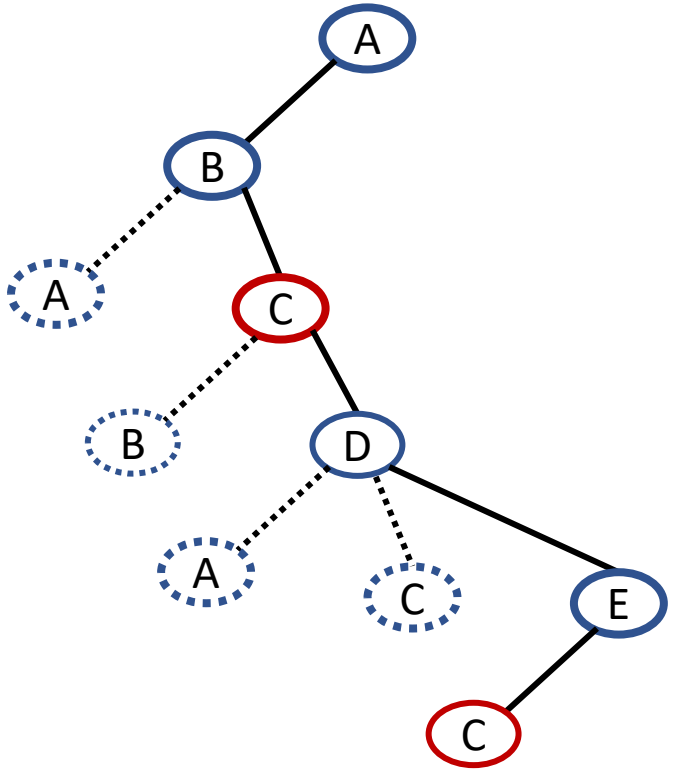
Hamiltonian Cycle Problem





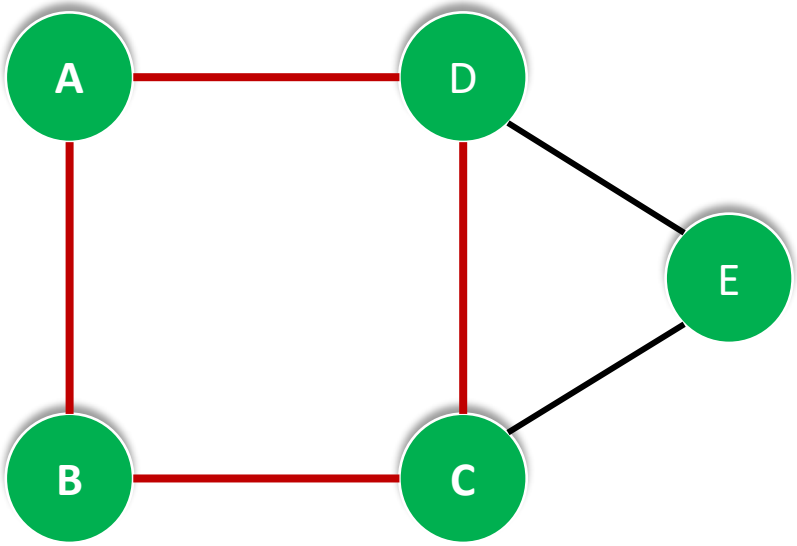
Hamiltonian Cycle Problem

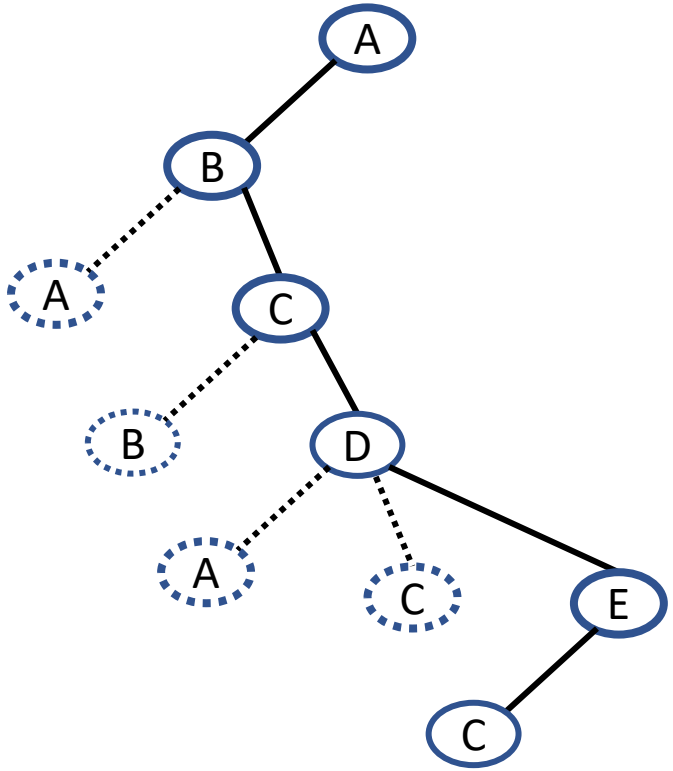




Backtrack **as C is visited twice**

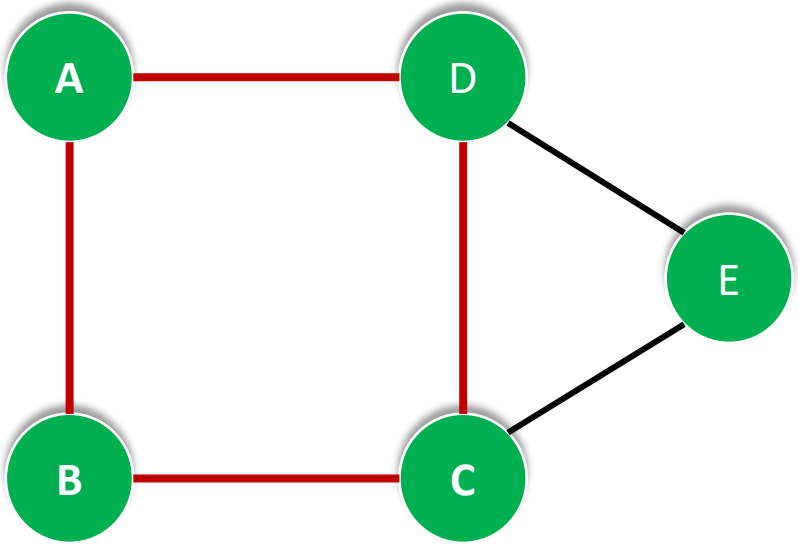
Hamiltonian Cycle Problem

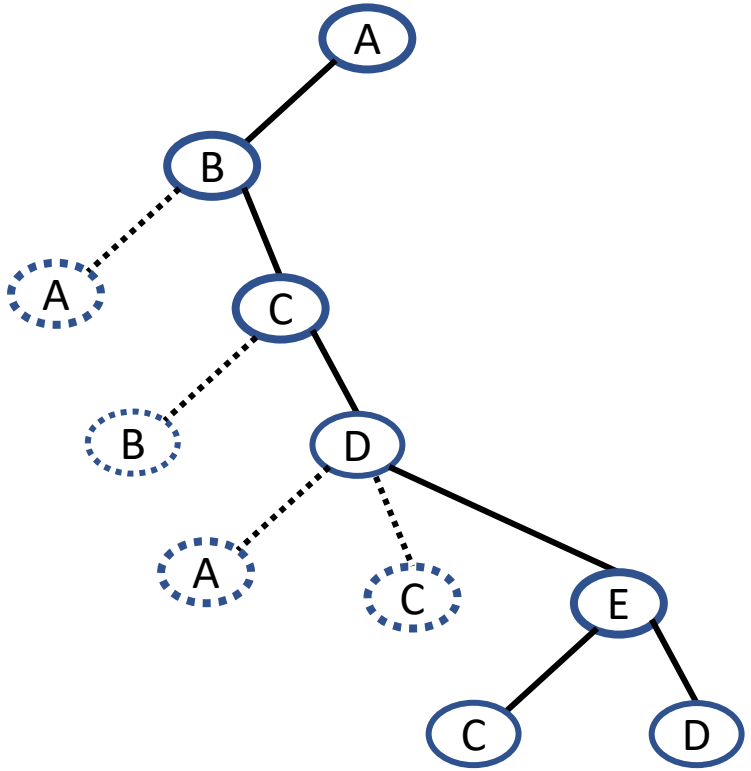




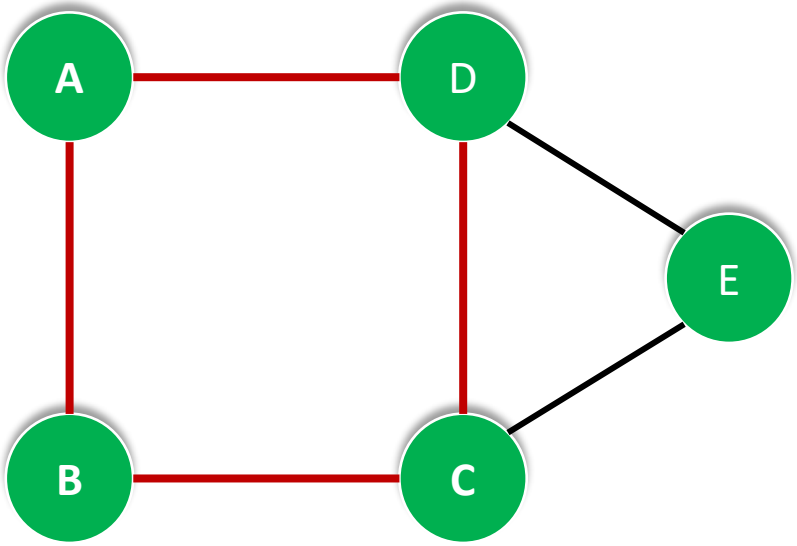
Backtrack **as C is visited twice**

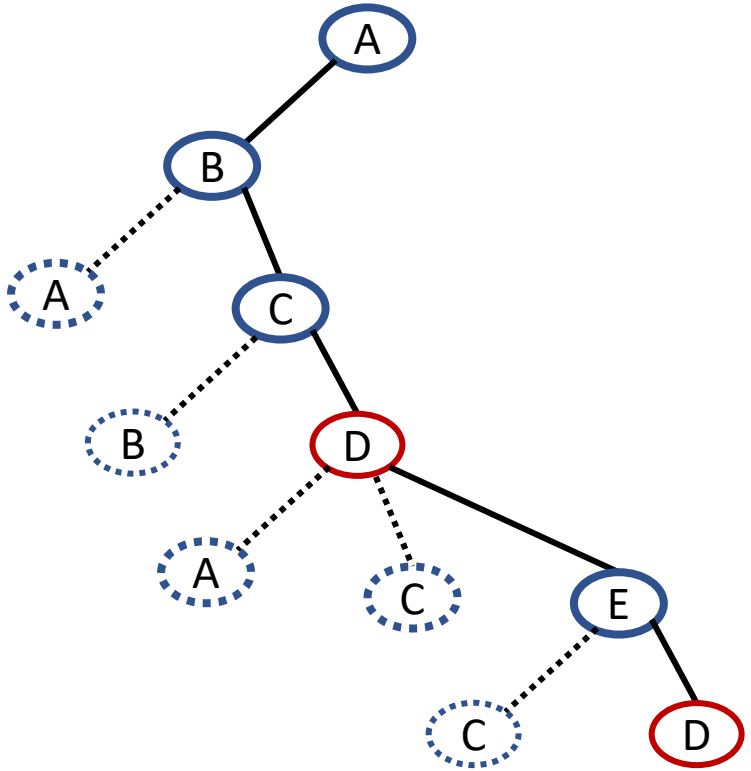
Hamiltonian Cycle Problem





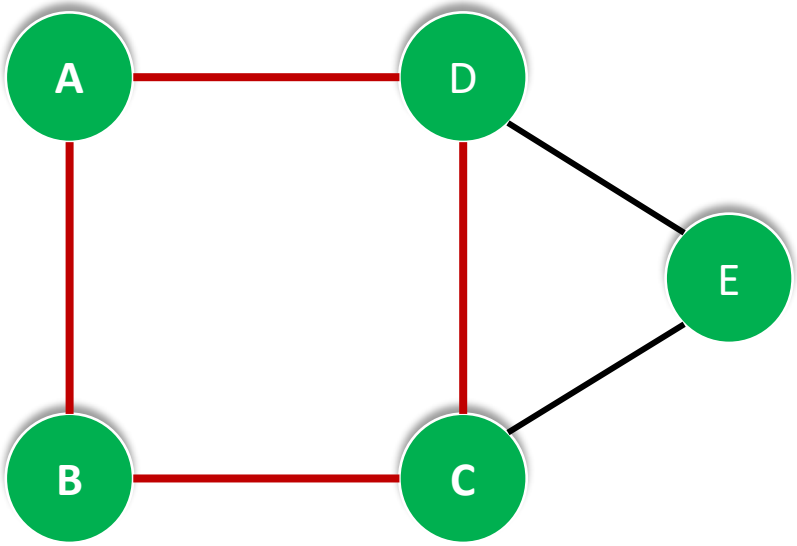
Hamiltonian Cycle Problem

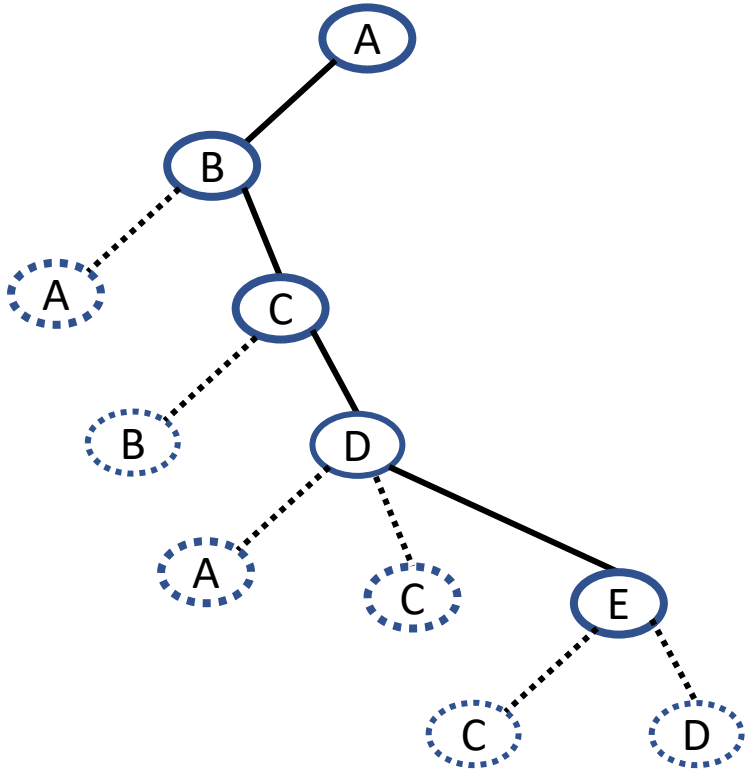




Backtrack **as D is visited twice**

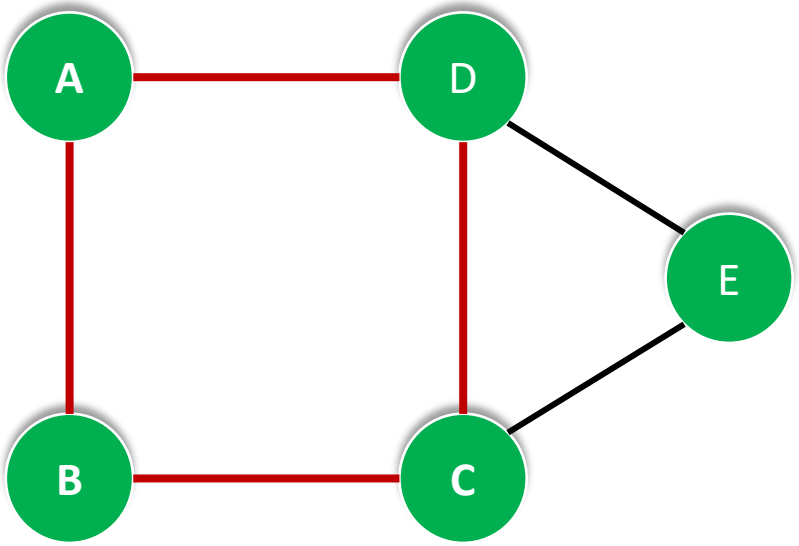
Hamiltonian Cycle Problem

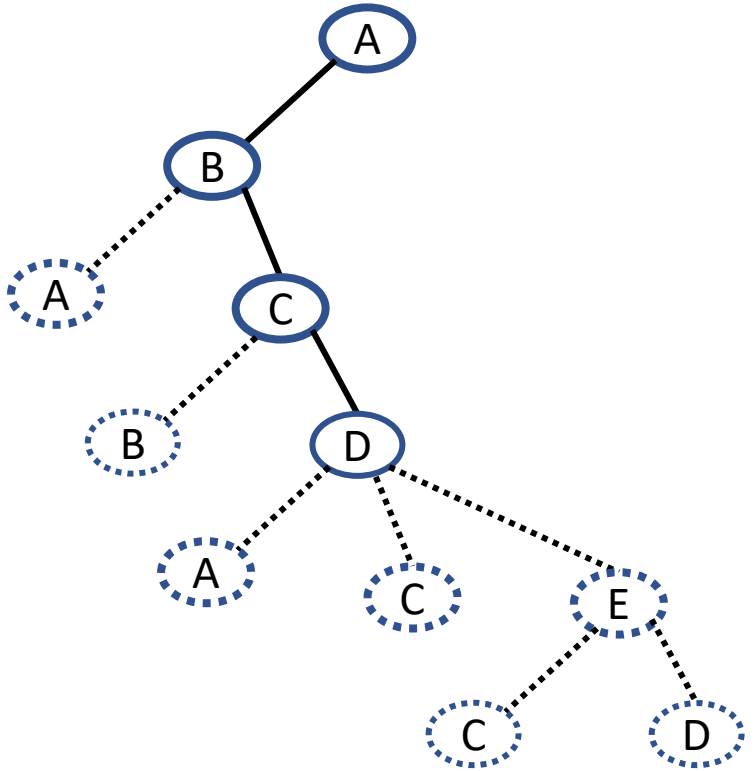




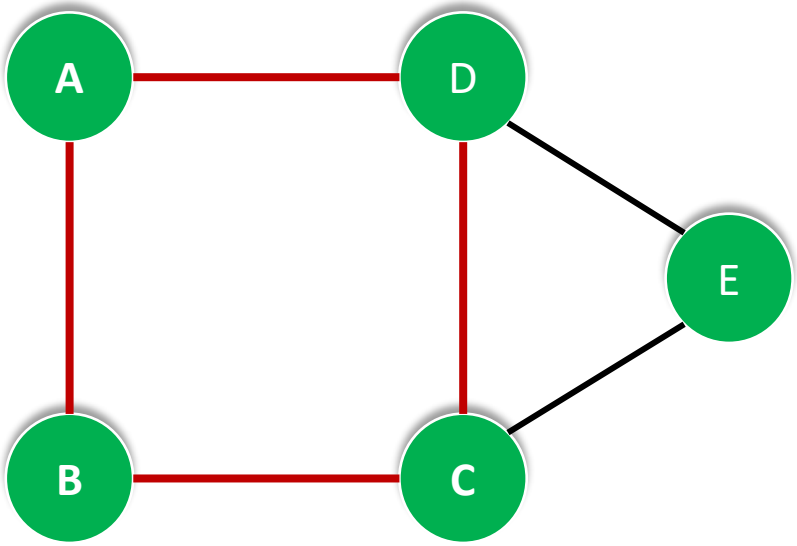
Backtrack **Tour is not completed**

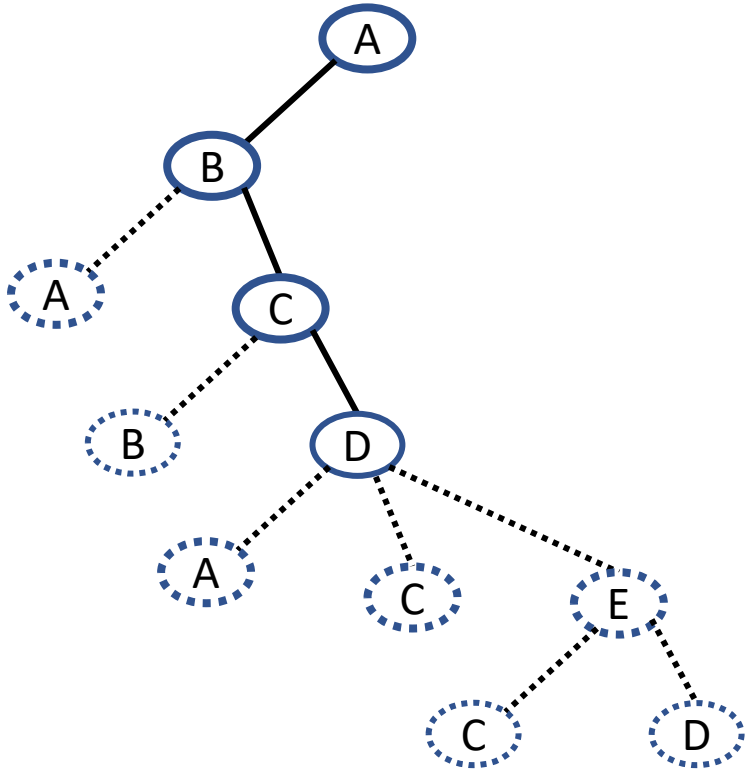
Hamiltonian Cycle Problem





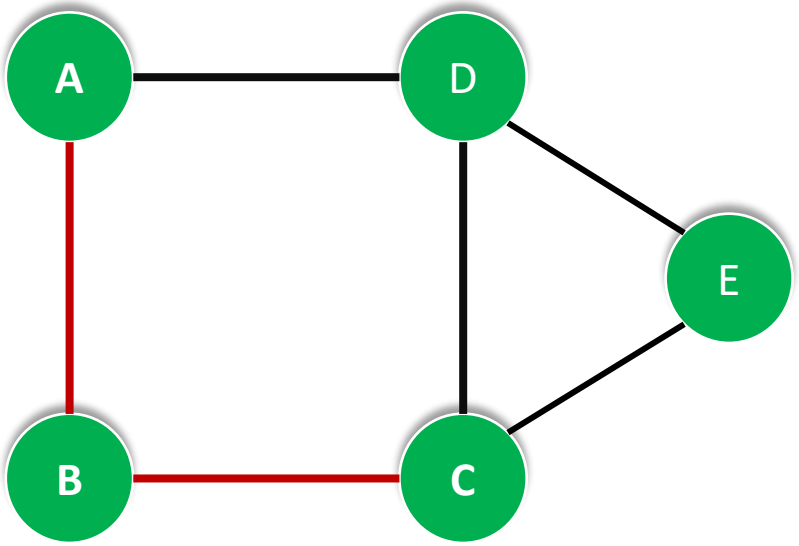
Hamiltonian Cycle Problem

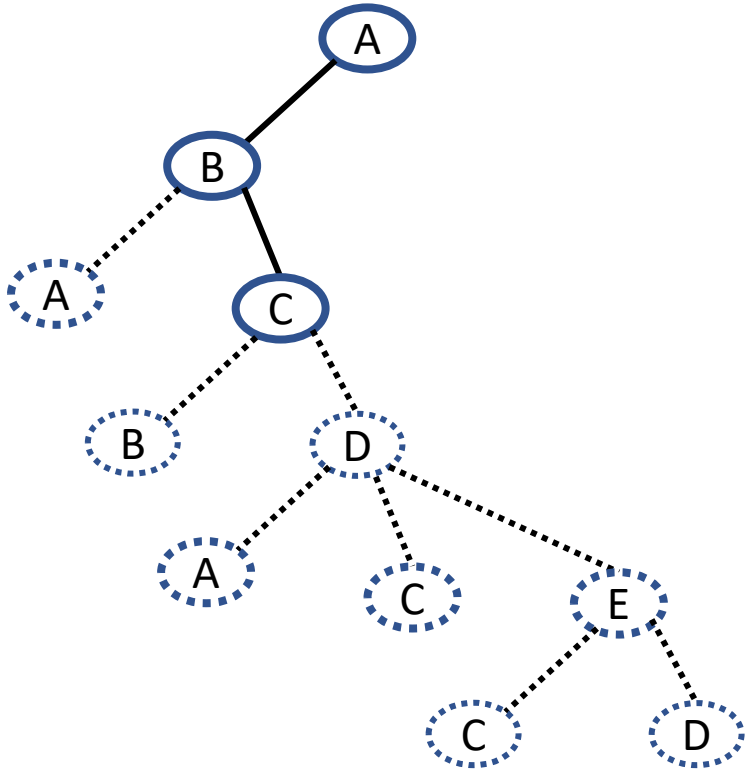




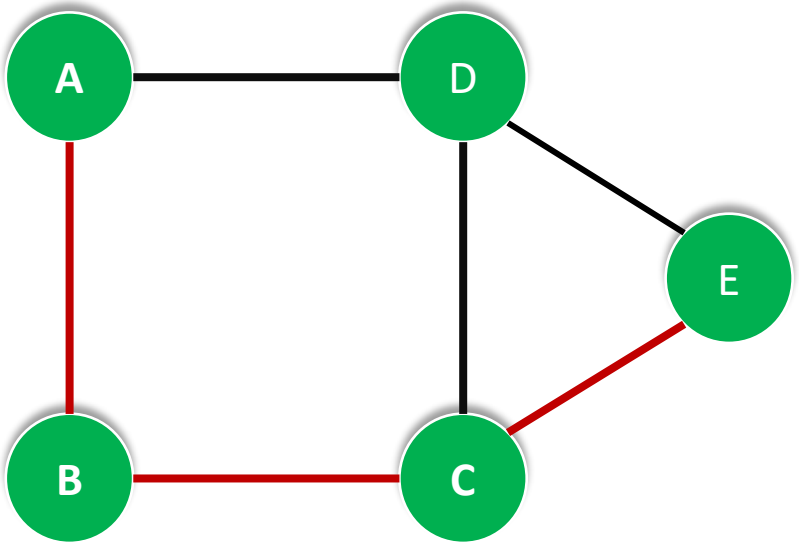
Backtrack **Tour is not completed**

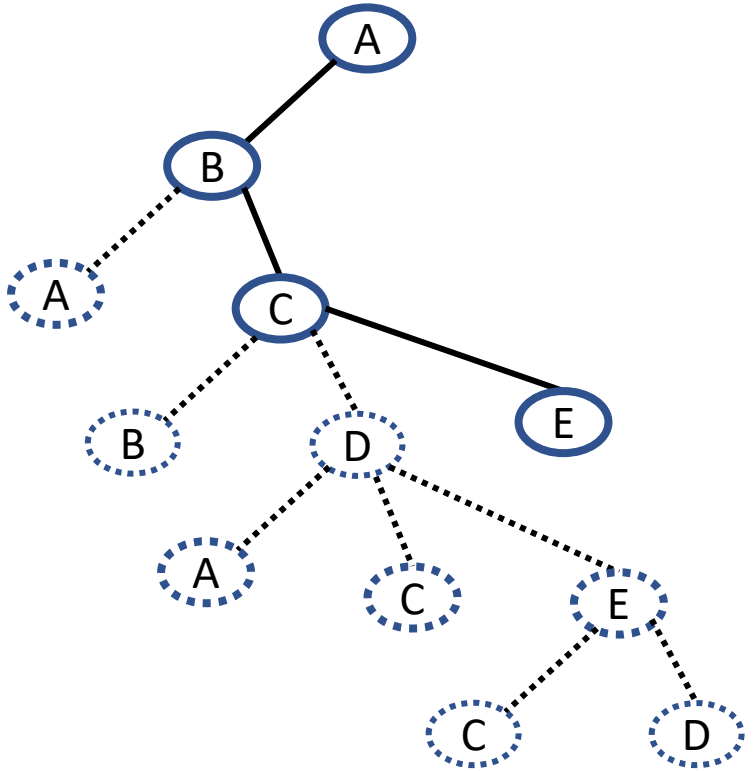
Hamiltonian Cycle Problem



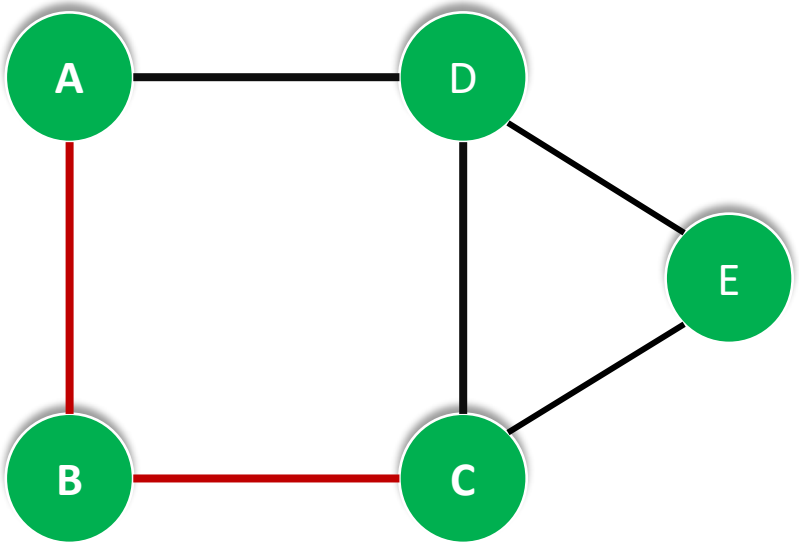


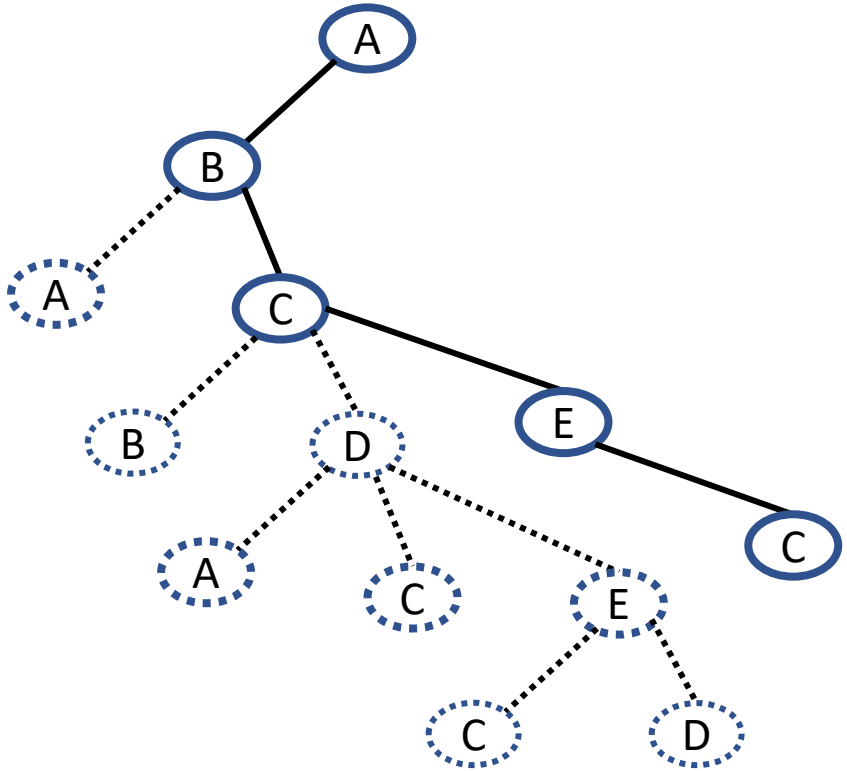
Hamiltonian Cycle Problem



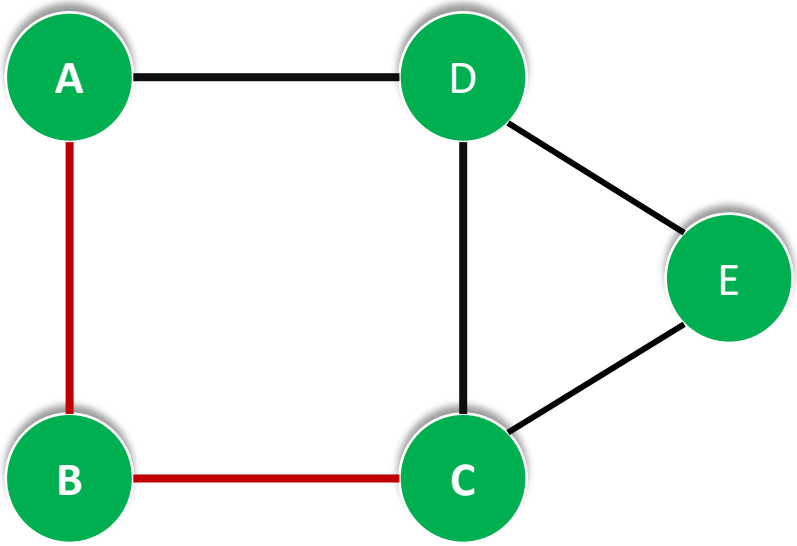


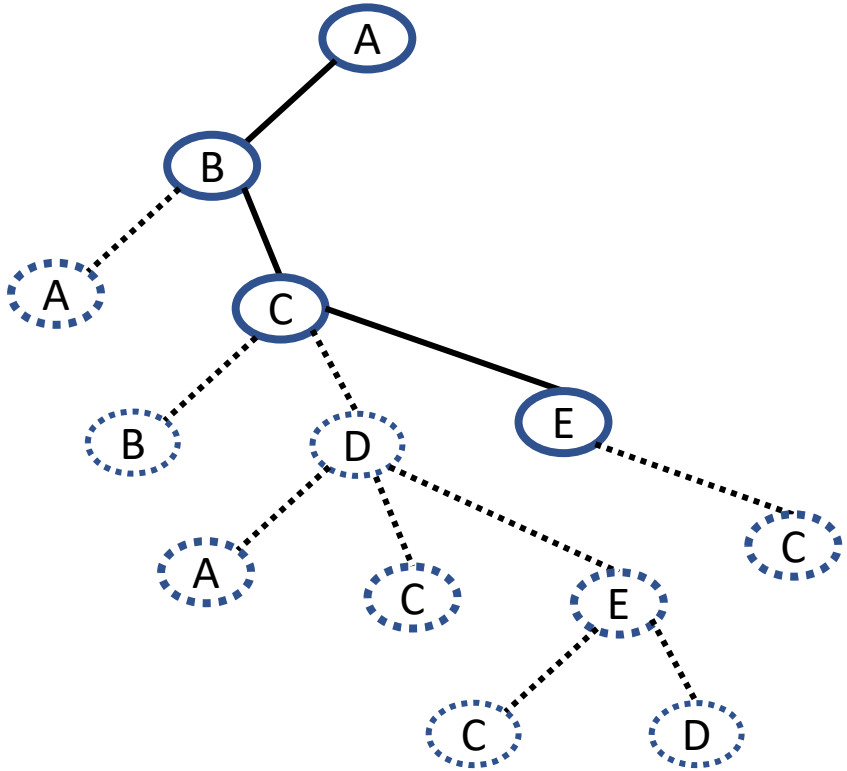
Hamiltonian Cycle Problem





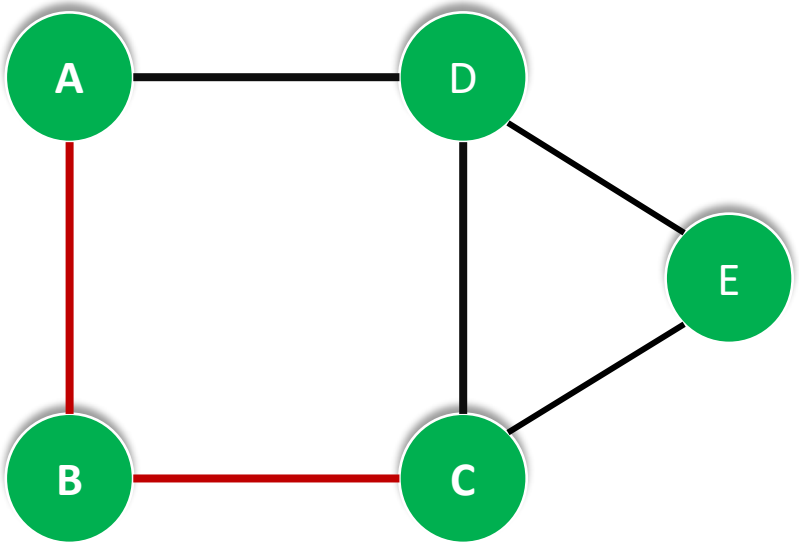
Hamiltonian Cycle Problem

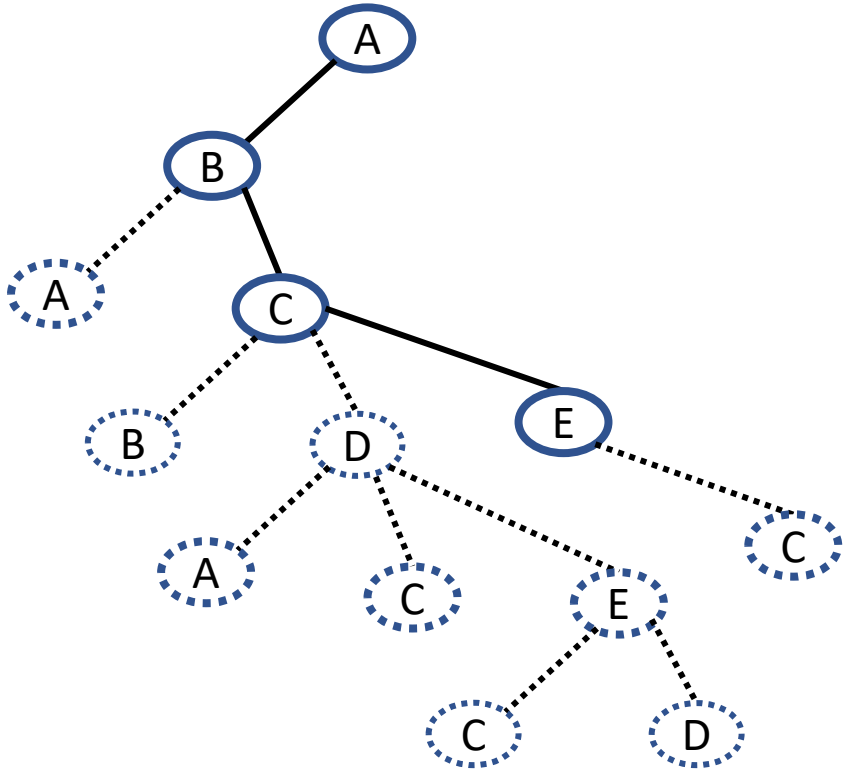




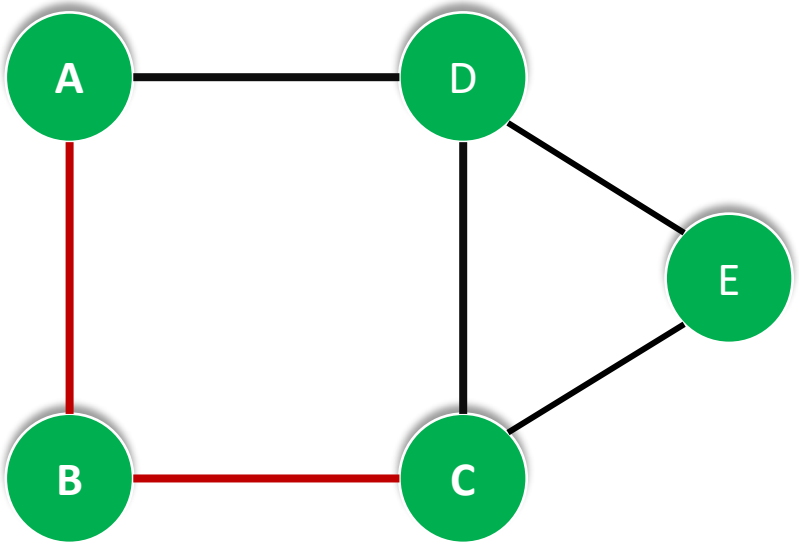
Backtrack **as C is visited twice**

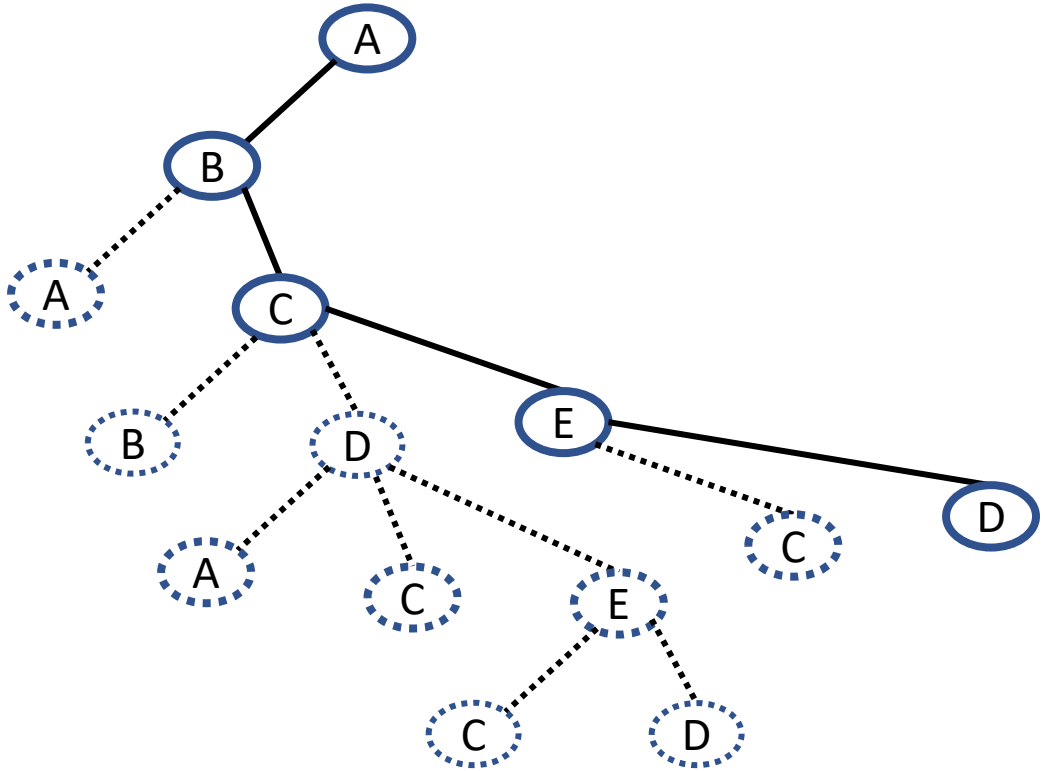
Hamiltonian Cycle Problem



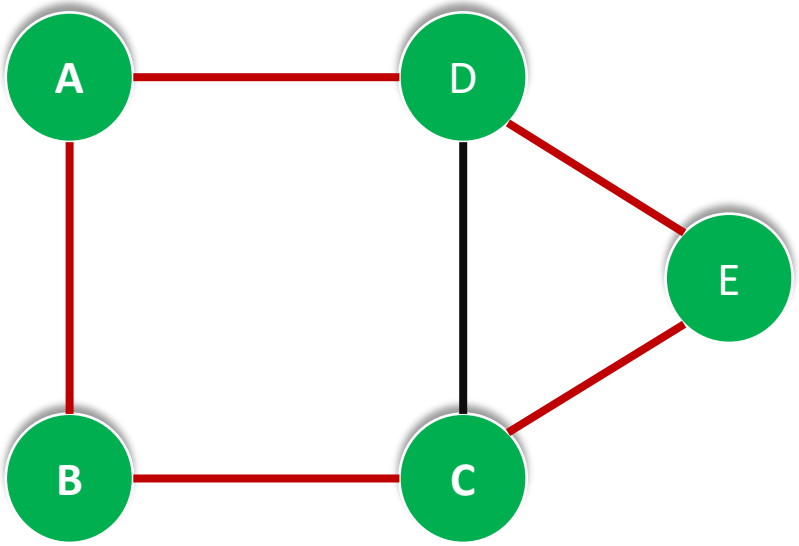


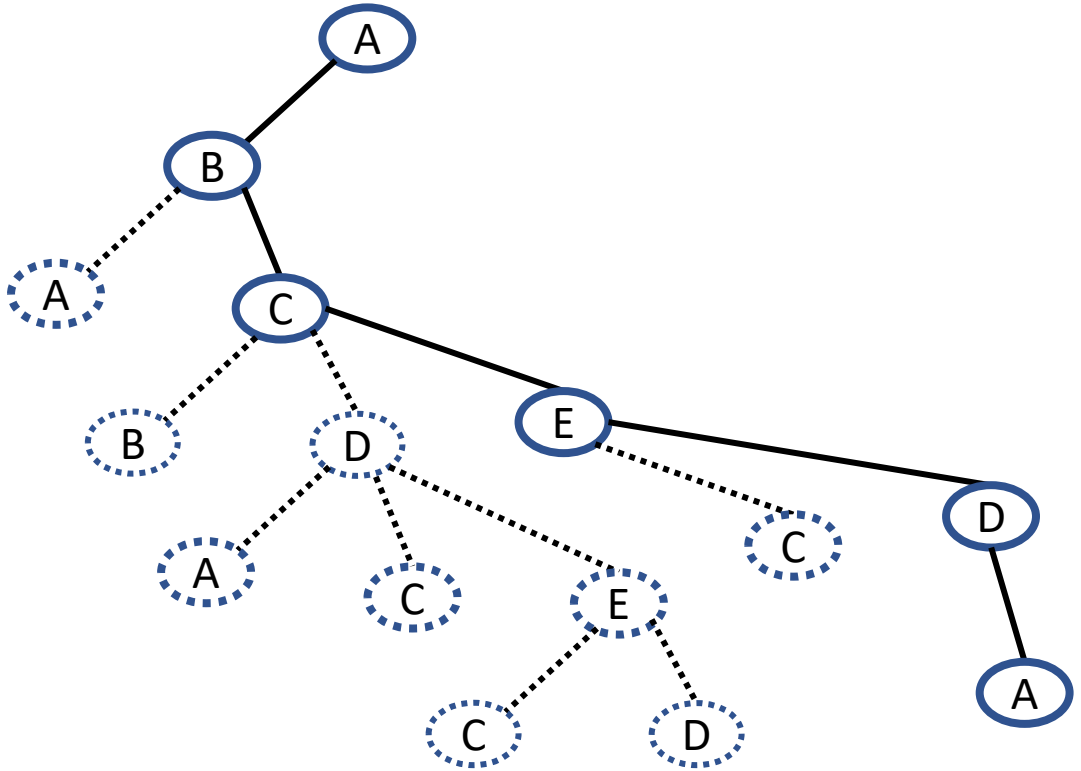
Hamiltonian Cycle Problem



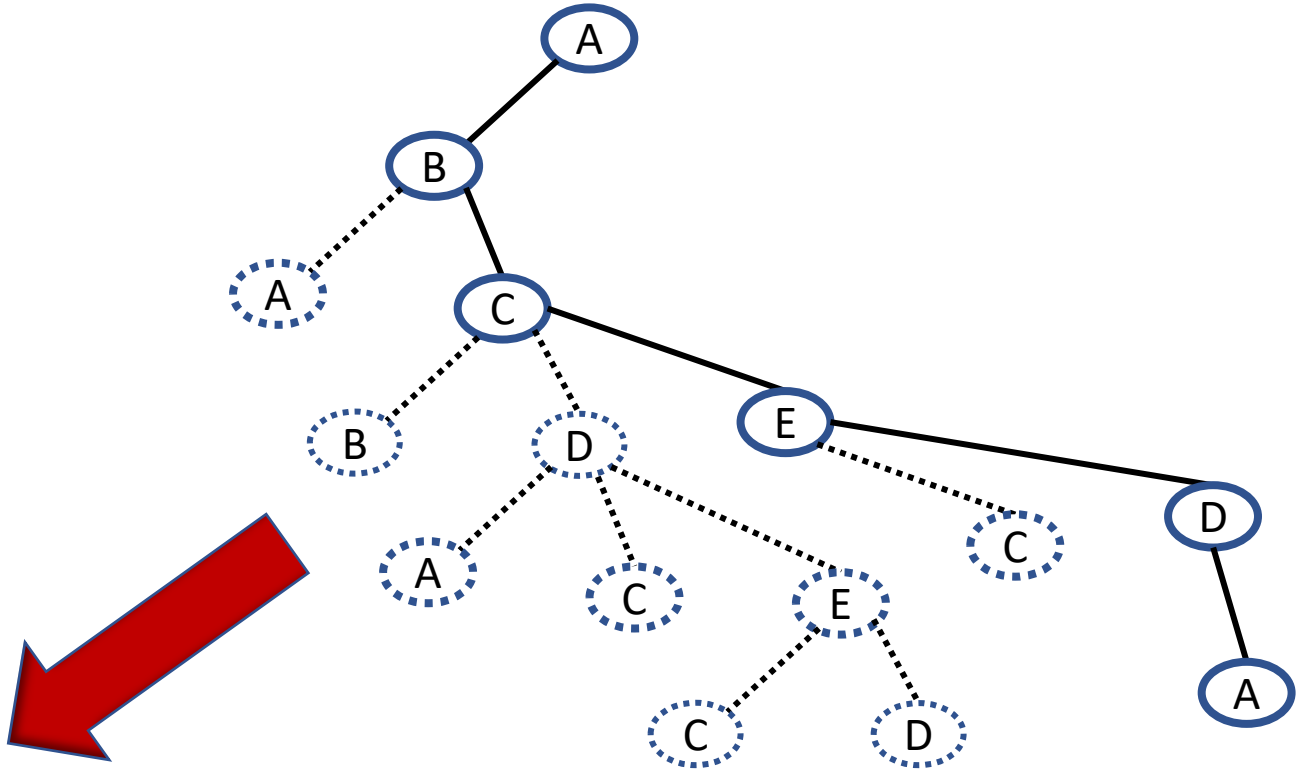


Hamiltonian Cycle Problem





Hamiltonian Cycle Problem



State Space Search Tree

Hamiltonian Cycle Problem

Time Complexity = Exponential

Hamiltonian Cycle Problem

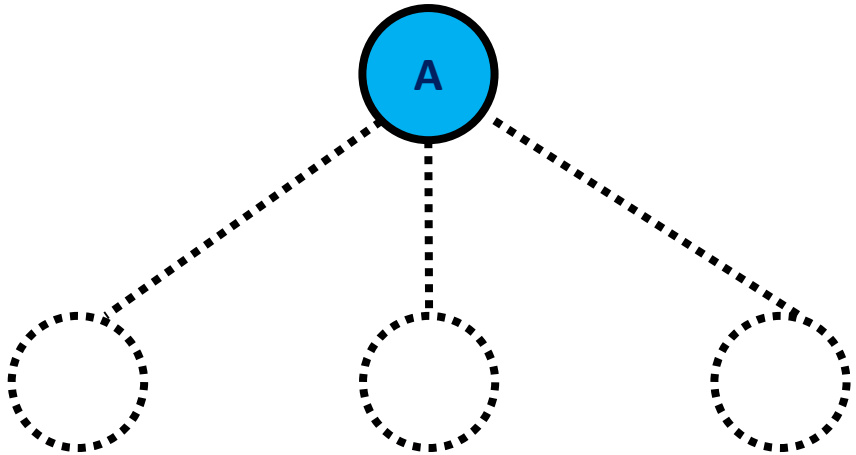
```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Hamiltonian Cycle Problem

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                                  // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then  return;
20          }
21      } until (false);
22  }
```

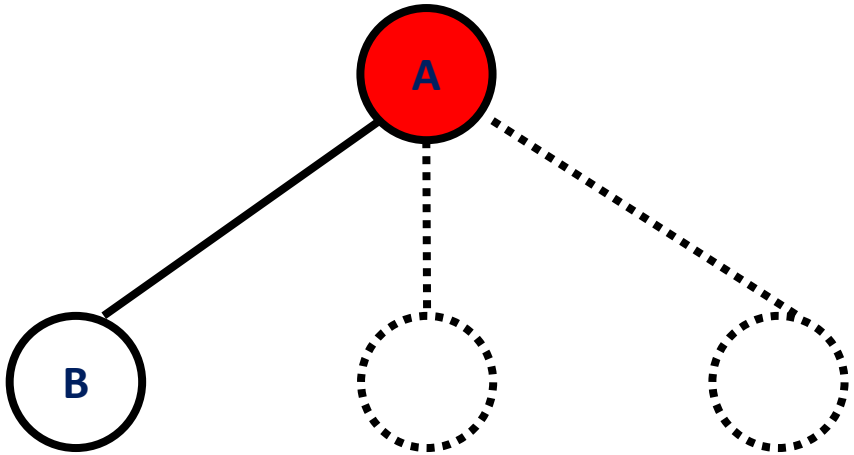
Backtracking

Live Node: A node which has been generated but all of whose children have not yet been generated



Backtracking

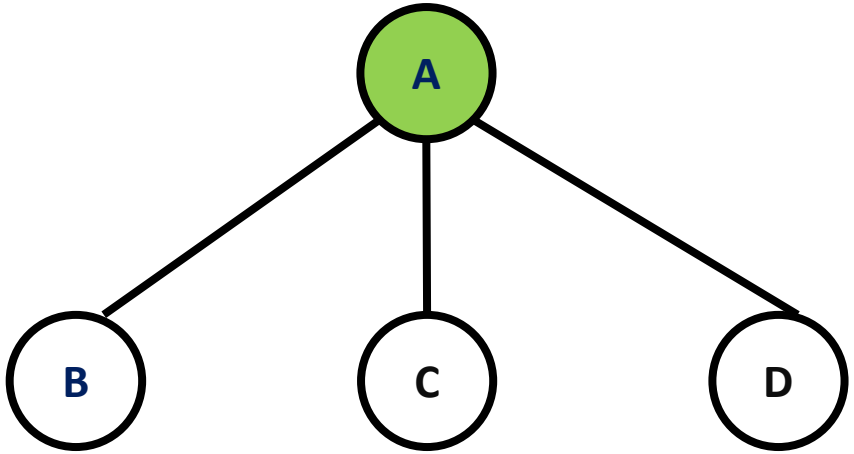
E-Node: A live whose children are currently generated



Backtracking

Dead Node: A node which is not to be expanded further or all of whose children are generated

.



Branch and Bound

It was introduced in 1960 by two researchers, Alisa Land and Alison Diog, from the London School of Economics

Branch and Bound

- Branch and bound (BB, B&B, or BnB) is a method for solving optimization problems
- It breaks them down into smaller sub-problems and
- uses a bounding function to eliminate sub-problems that cannot contain the optimal solution.

Branch and Bound

Types:

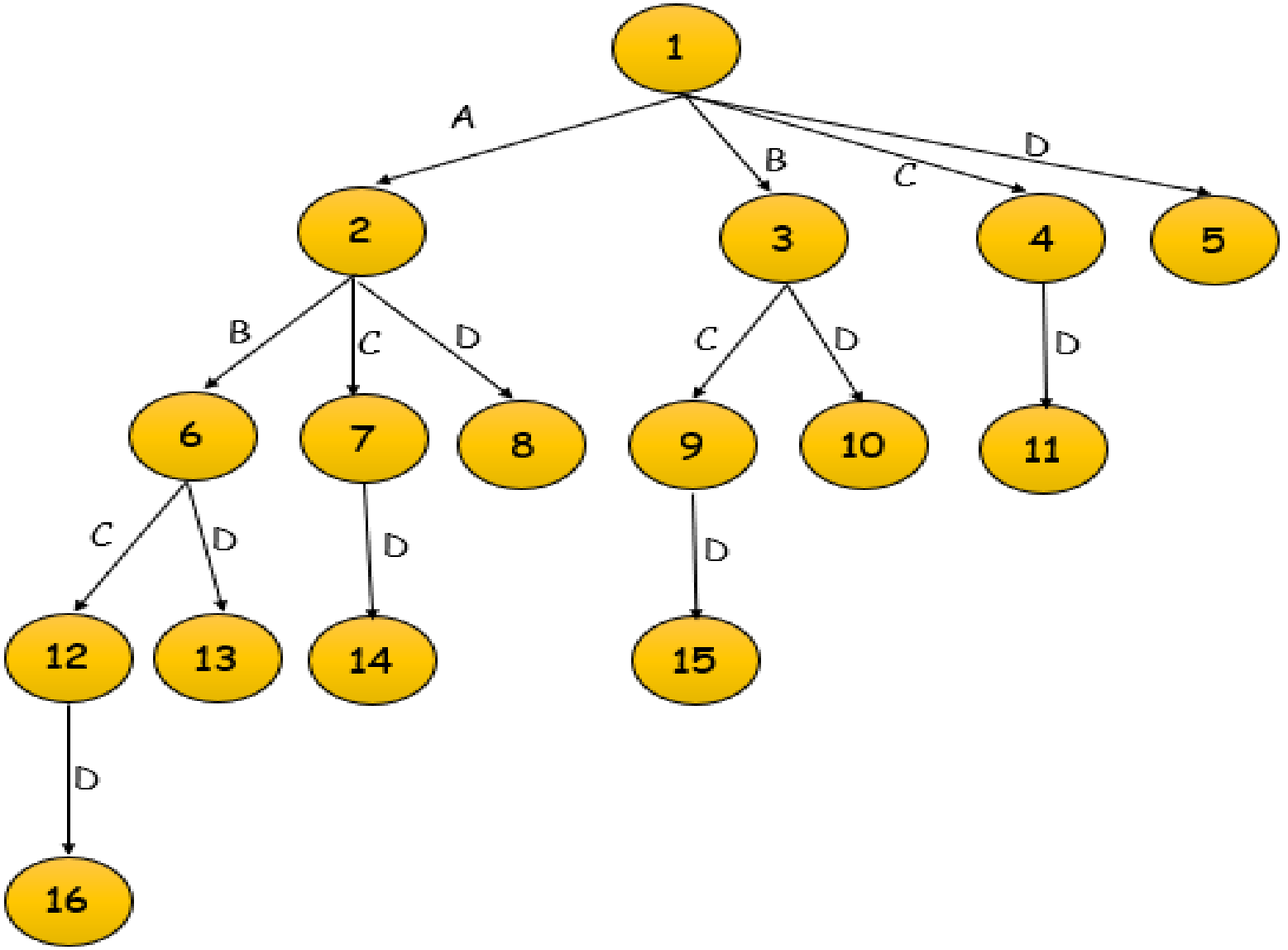
1. FIFO Branch and Bound
2. LIFO Branch and Bound
3. Least Cost-Branch and Bound

FIFO Branch and Bound

It uses Queue data structure

The elements at a particular level are all searched, and then the elements of the next level are searched, starting from the **first child** of the **first node** in the **previous level**

FIFO Branch and Bound

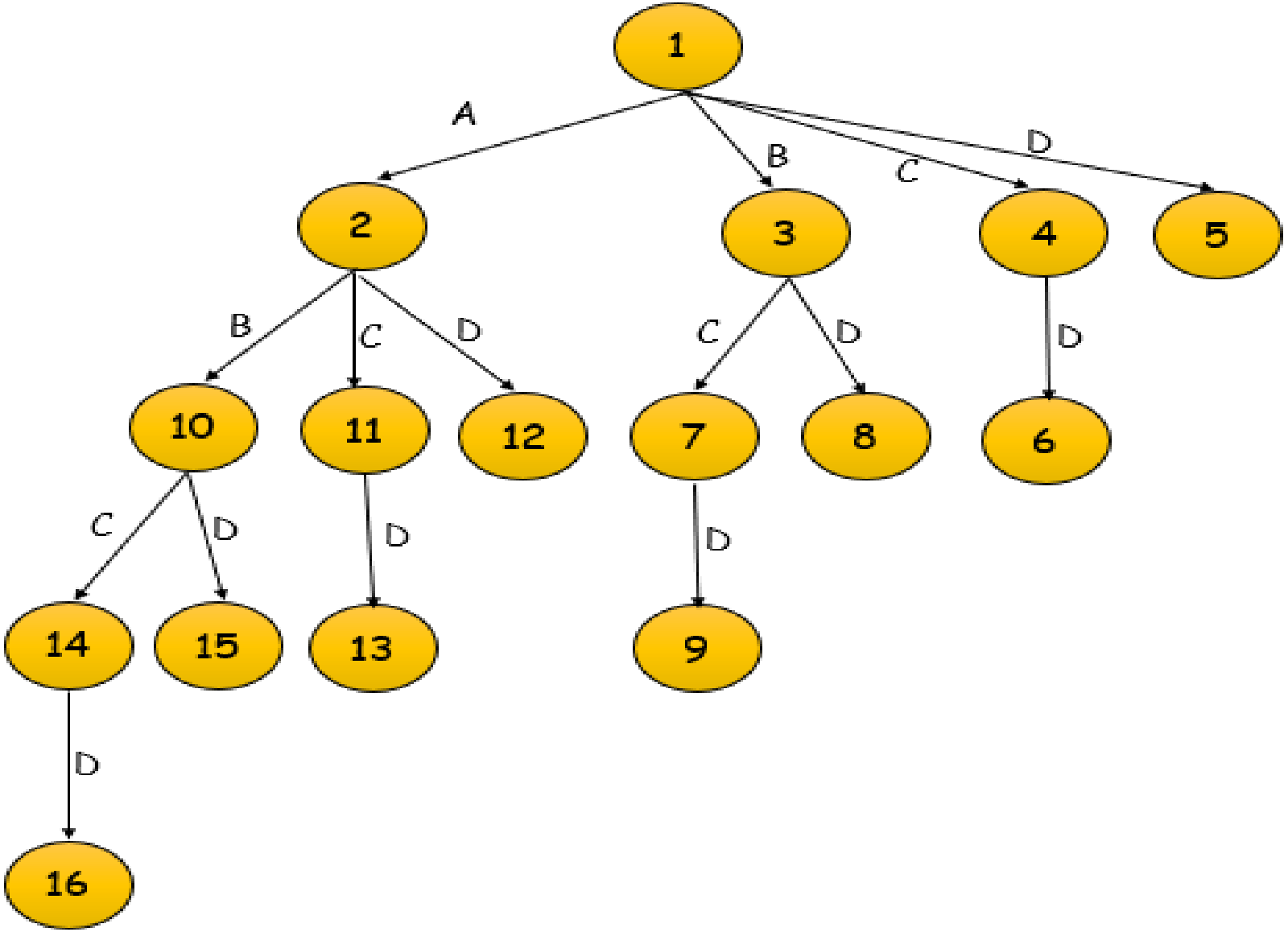


LIFO Branch and Bound

It uses Stack data structure

When all nodes of a level are added in a stack, we pop the topmost element from the stack and then explore it.

LIFO Branch and Bound

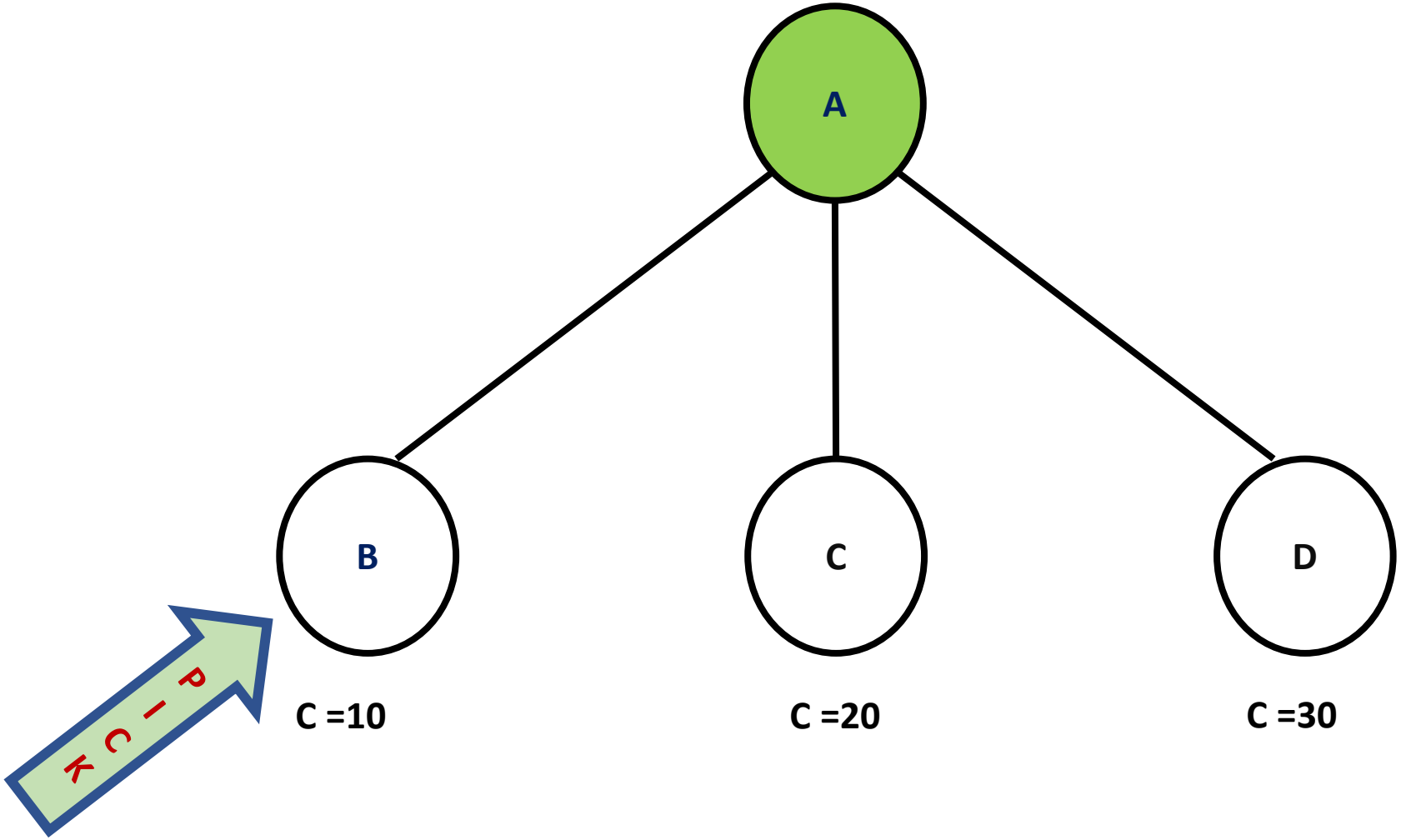


Least Cost Branch and Bound

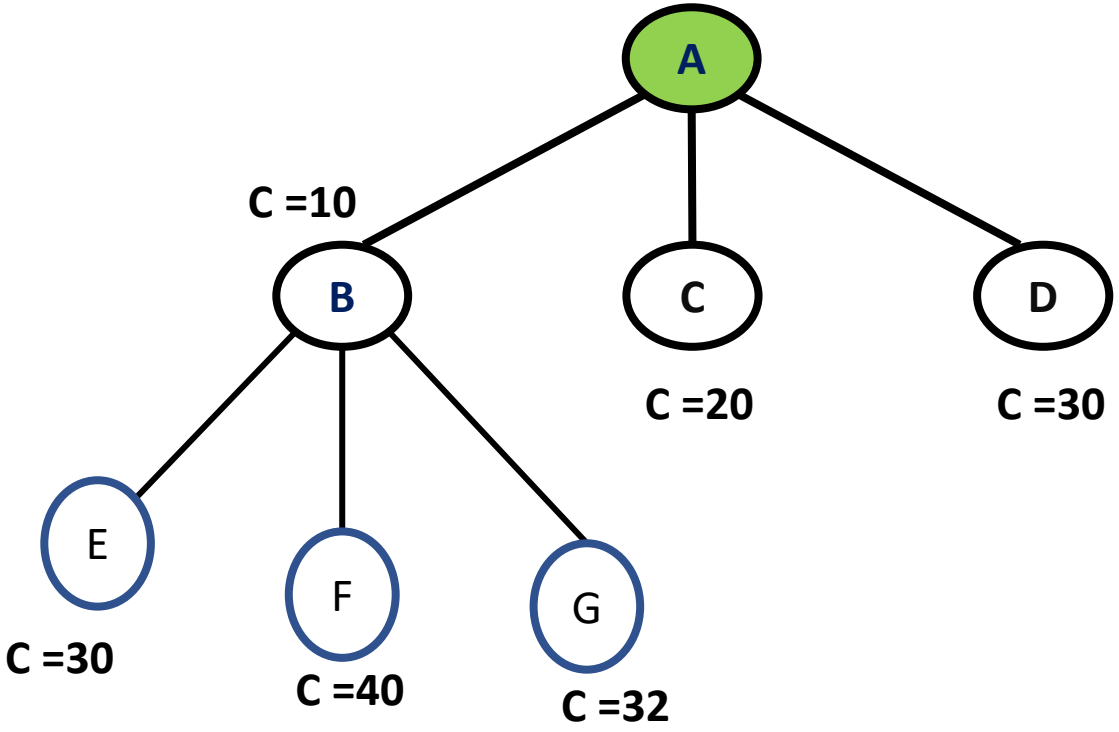
This method of exploration uses the **cost function** in order to explore the state space tree.

In this method, after the children of a particular node have been explored, the next node to be explored would be that node out of the unexplored nodes which has the **least cost**.

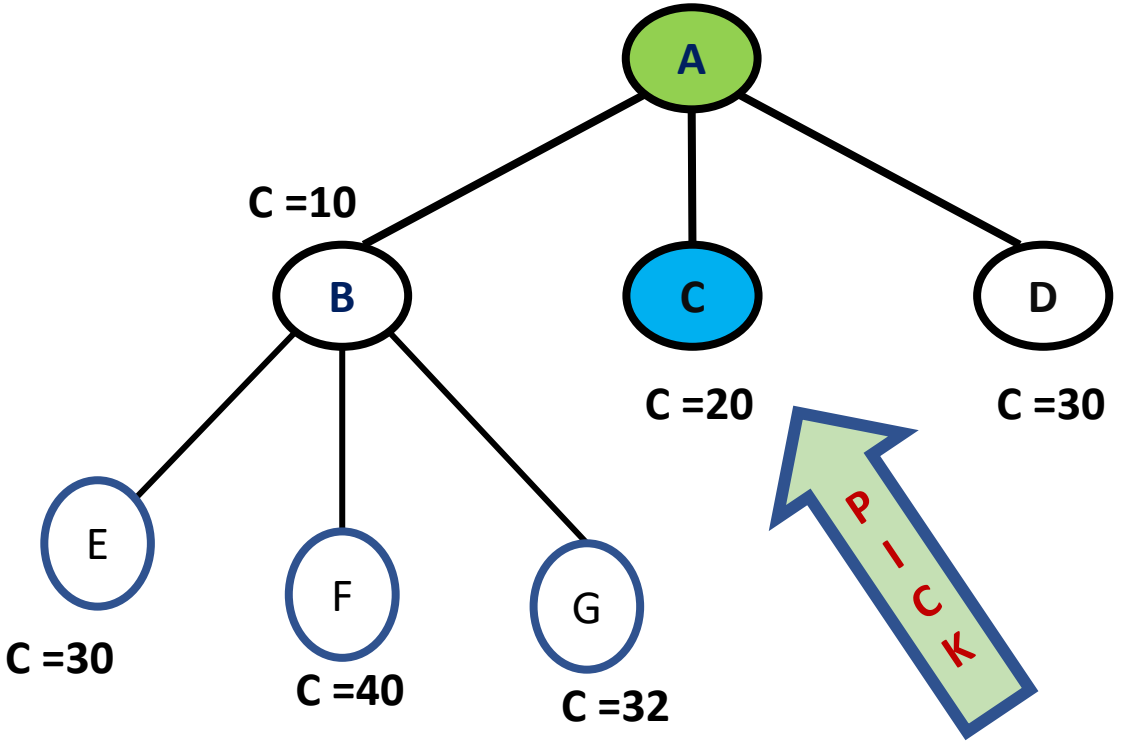
Least Cost Branch and Bound



Least Cost Branch and Bound



Least Cost Branch and Bound



Branch and Bound

Application of Branch and Bound

- Job Assignment Problem
- 0/1 Knapsack problem
- **Travelling Salesman Problem**

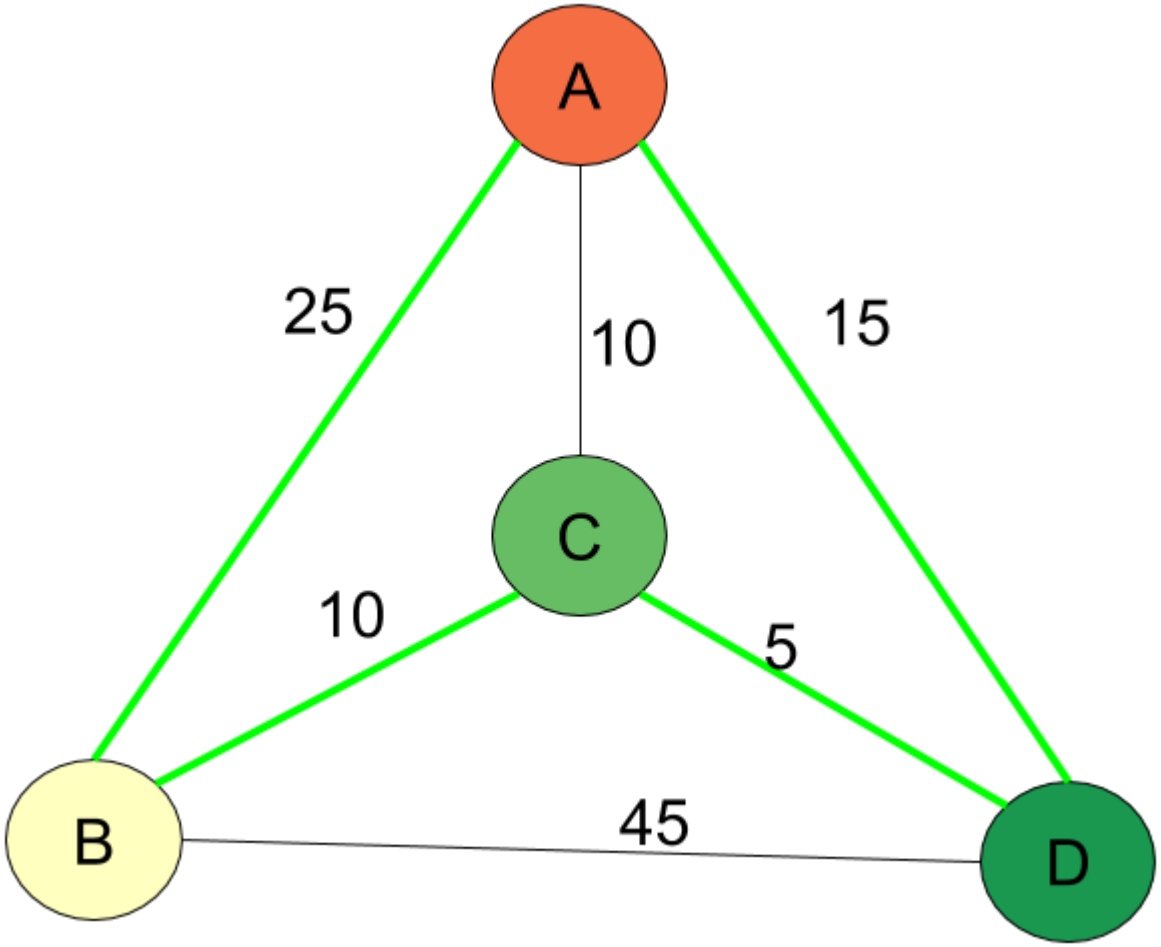
Travelling Salesman Problem

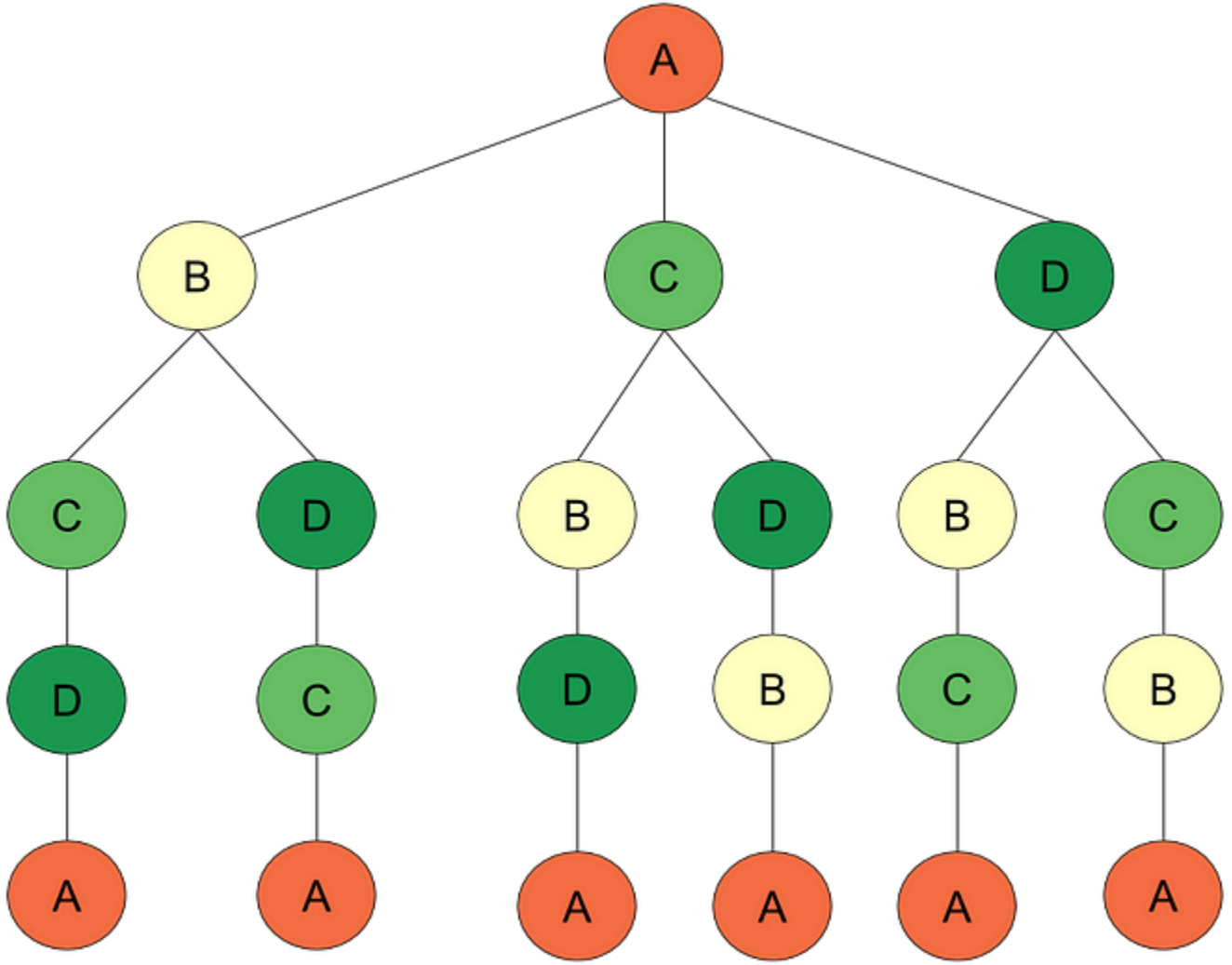
"Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point."

There are two important things to be cleared about in this problem statement,

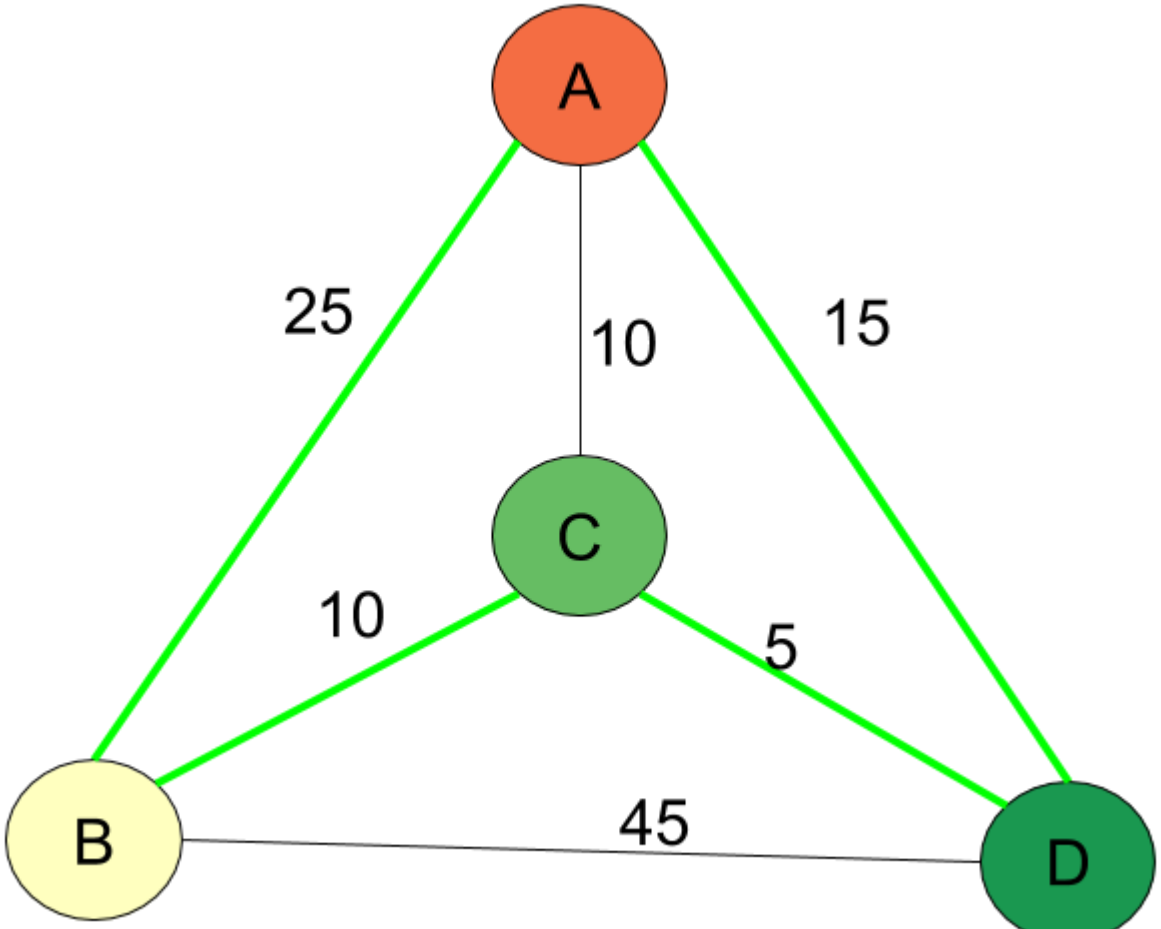
- **Visit every city exactly once**
- **Cover the shortest path**

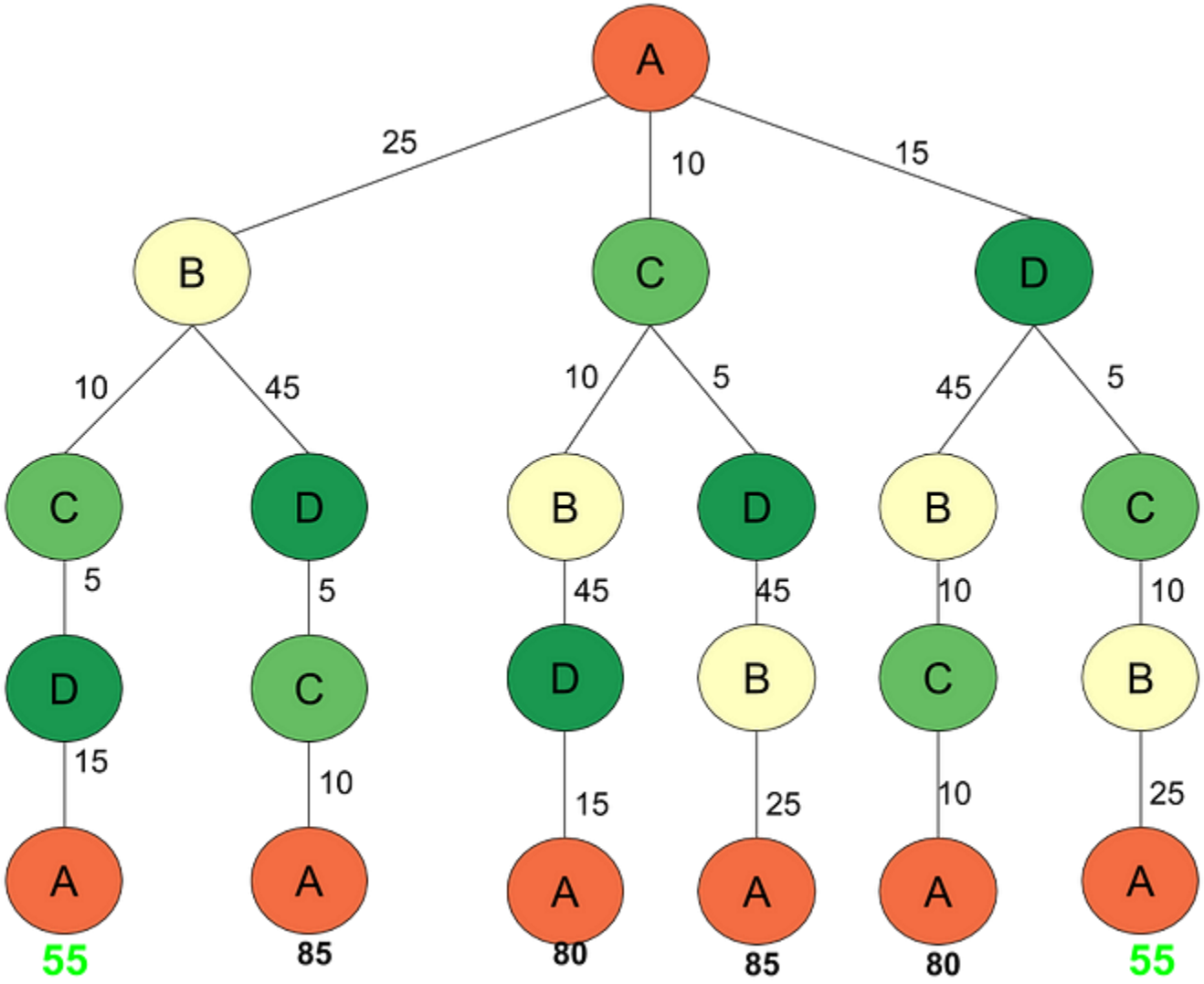
Travelling Salesman Problem



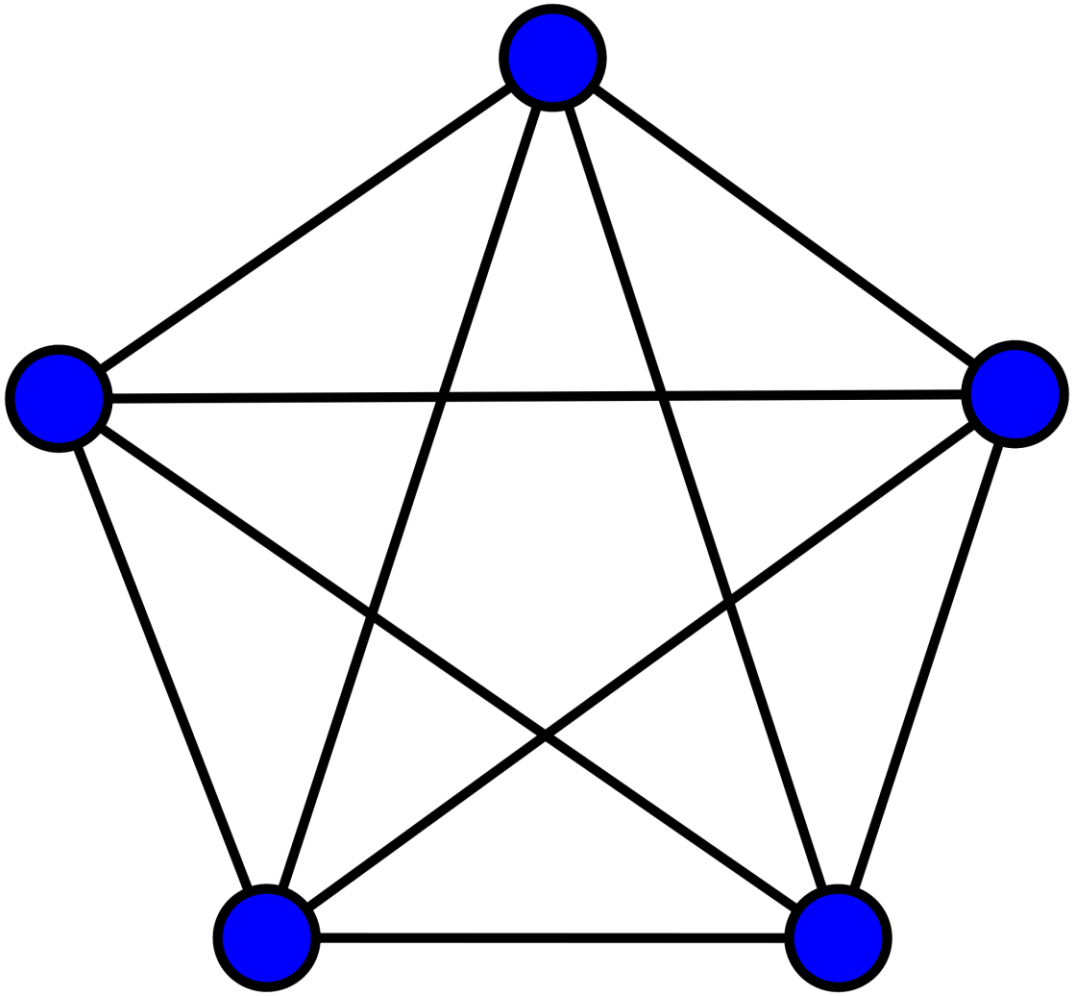


Travelling Salesman Problem





TSP using Branch and Bound



C	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

TSP using Branch and Bound

C	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

Lower bound on solution can be computed by

- ❑ Row reduction and
- ❑ Column Reduction

TSP using Branch and Bound

Row reduction

If the row already contains an entry '0', then-

There is no need to reduce that row

If the row **does not** contain an entry '0', then Reduce that row.

1. Select the least value from that row.
2. Subtract that element from each element of that row.

This will create an entry '0' in that row, thus reducing that row.

TSP using Branch and Bound

Row reduction

C	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞

C	1	2	3	4	5
1	∞	10	20	0	1
2	13	∞	14	2	0
3	1	3	∞	0	2
4	16	3	15	∞	0
5	12	0	3	12	∞

Row Reduction = 10+2+2+3+4=**21**

Row Reduced

TSP using Branch and Bound

Column reduction

If the column already contains an entry '0', then-

There is no need to reduce that column

If the column **does not** contain an entry '0', then Reduce that column.

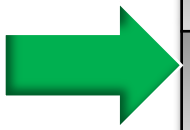
1. Select the least value element from that column.
2. Subtract that element from each element of that column.

This will create an entry '0' in that column, thus reducing that column.

TSP using Branch and Bound

Row reduction

C	1	2	3	4	5
1	∞	10	20	0	1
2	13	∞	14	2	0
3	1	3	∞	0	2
4	16	3	15	∞	0
5	12	0	3	12	∞



C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

Column Reduction = 1 + 3 = **4**

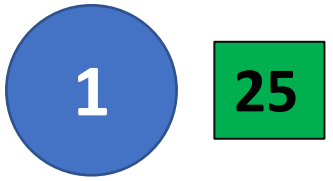
Column Reduced

TSP using Branch and Bound

Reduced Matrix

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

Assume vertex 1 as our source Vertex

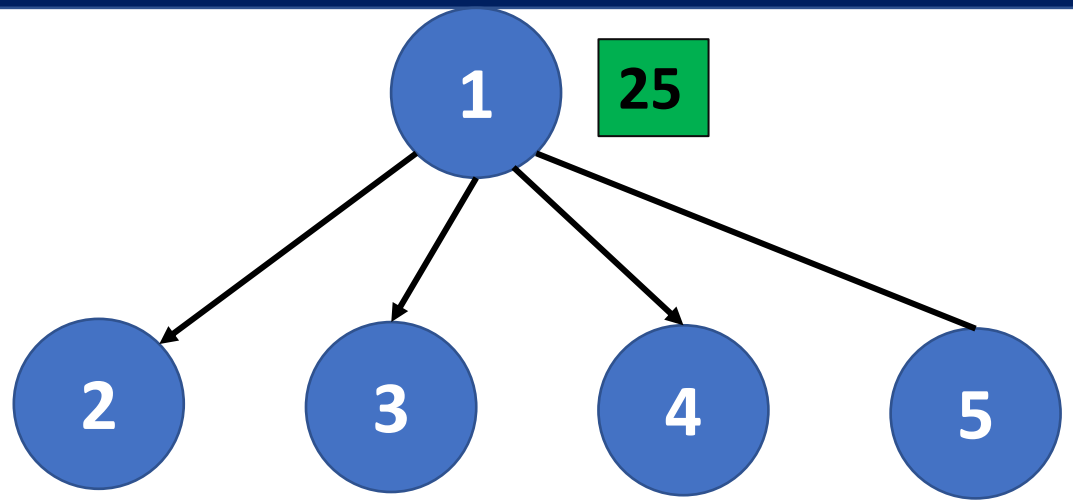


Reduction cost = 21 + 4 = **25**

TSP using Branch and Bound

Reduced Matrix

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞



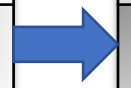
TSP using Branch and Bound

Select edge 1-2:

Set M [1] [ ] = M [ ] [2] = ∞

Set M [2] [1] = ∞

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞



C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

TSP using Branch and Bound

Select edge 1-2:

Set M_1 [1] [ ] = M_1 [ ] [2] = ∞

Set M_1 [2] [1] = ∞

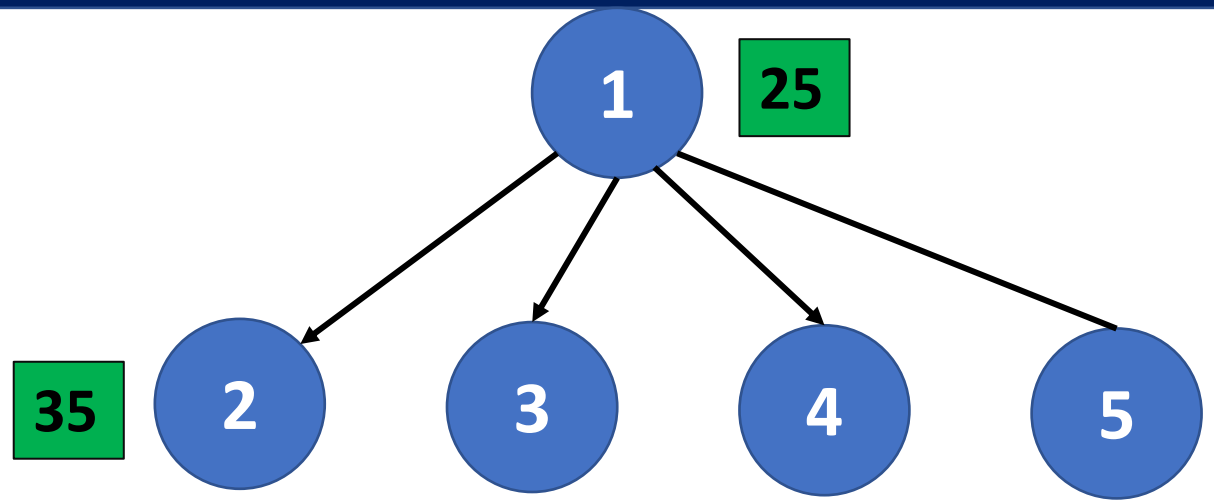
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

Already Reduced

Bound = 25 + 0 +C[1][2]

=25 +10 =35

TSP using Branch and Bound



TSP using Branch and Bound

Select edge 1-3:

Set M [1] [ ] = M [ ] [3] = ∞

Set M [3] [1] = ∞

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	∞	2	0
3	∞	3	∞	0	2
4	15	3	∞	∞	0
5	11	0	∞	12	∞

TSP using Branch and Bound

Reduction

C	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	∞	2	0	0
3	∞	3	∞	0	2	0
4	15	3	∞	∞	0	0
5	11	0	∞	12	∞	0
	11	0		0	0	

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	2	0
3	∞	3	∞	0	2
4	4	3	∞	∞	0
5	0	0	∞	12	∞

Reduction Cost = 11

TSP using Branch and Bound

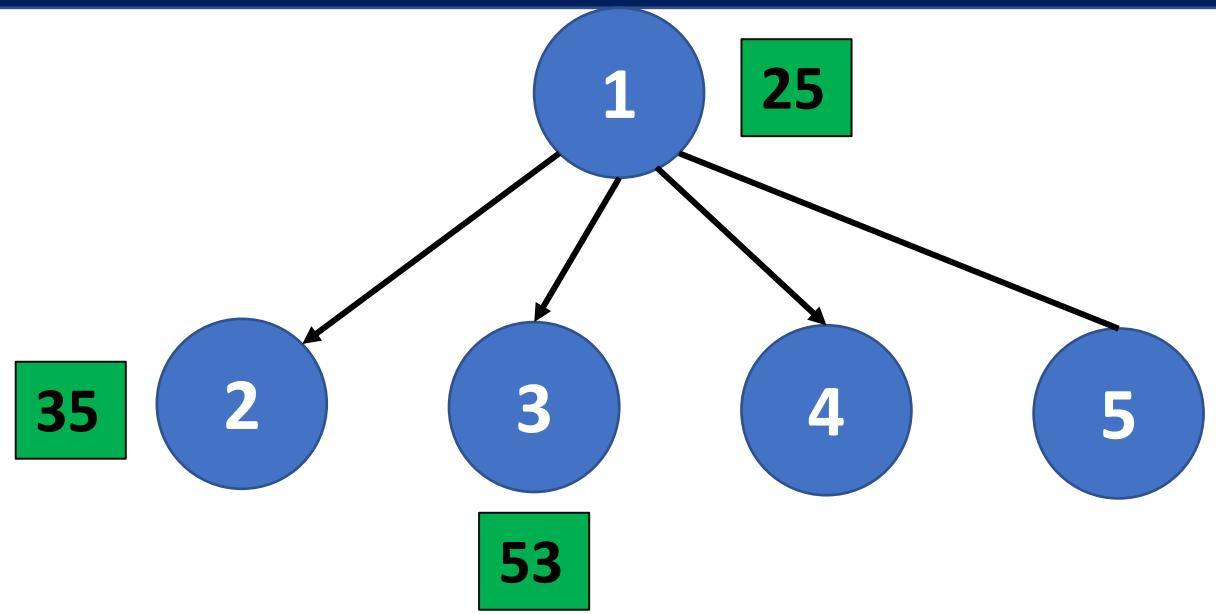
Reduced Matrix

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	2	0
3	∞	3	∞	0	2
4	4	3	∞	∞	0
5	0	0	∞	12	∞

Bound = 25 + 17 + 11

= 53

TSP using Branch and Bound



TSP using Branch and Bound

Select edge 1-4: Set M [1][ ] = M[ ][4] = ∞ Set M [4][1] = ∞

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

Already Reduced

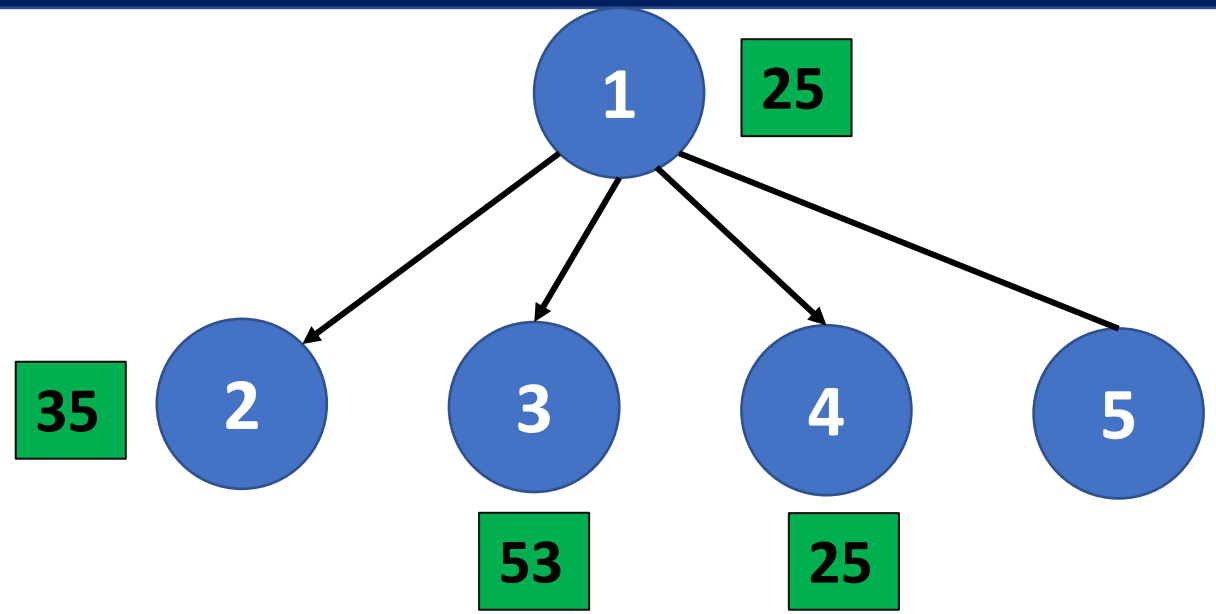
TSP using Branch and Bound

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

Bound = 25 + 0 + 0

= 25

TSP using Branch and Bound



TSP using Branch and Bound

Select edge 1 - 5: Set M [1][ ] = M[ ][5] = ∞ Set M [5][1] = ∞

C	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	2	∞
3	0	3	∞	0	∞
4	15	3	12	∞	∞
5	∞	0	0	12	∞

TSP using Branch and Bound

Reduction

C	1	2	3	4	5	
1	∞	∞	∞	∞	∞	
2	12	∞	11	2	∞	2
3	0	3	∞	0	∞	0
4	15	3	12	∞	∞	3
5	∞	0	0	12	∞	0
	0	0	0	0		

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

Reduction Cost = 2 + 3 = 5

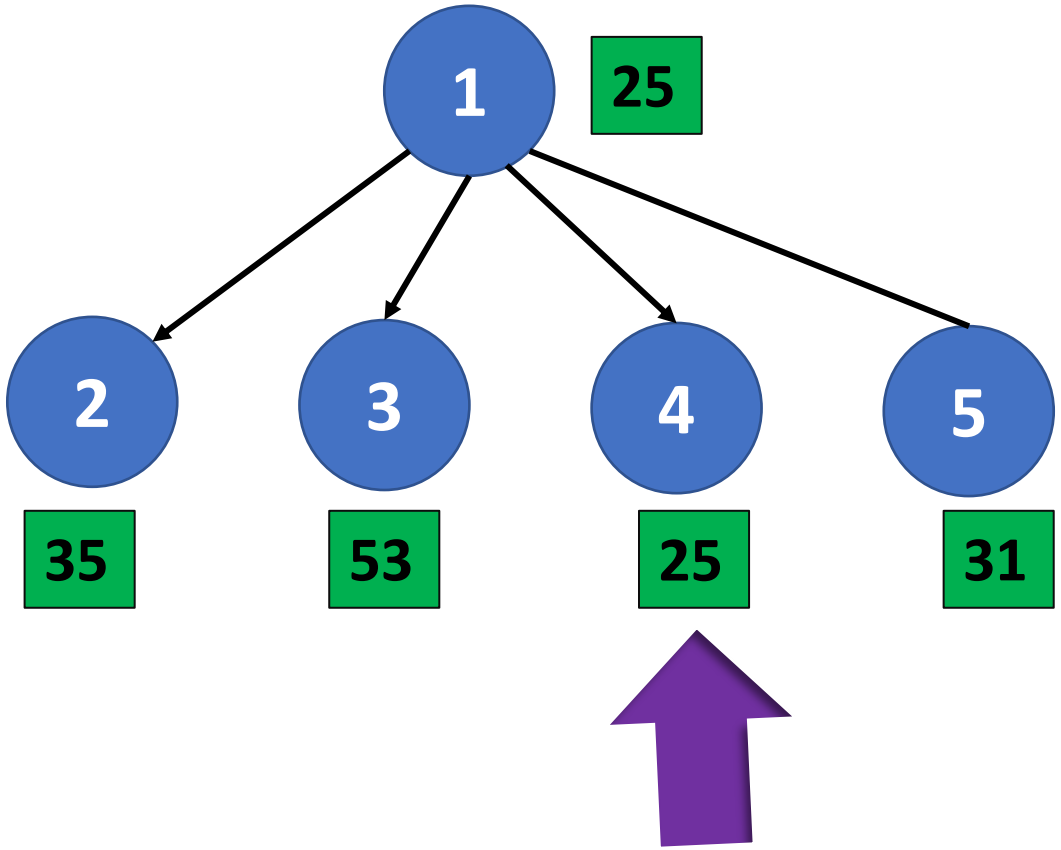
TSP using Branch and Bound

Reduction

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

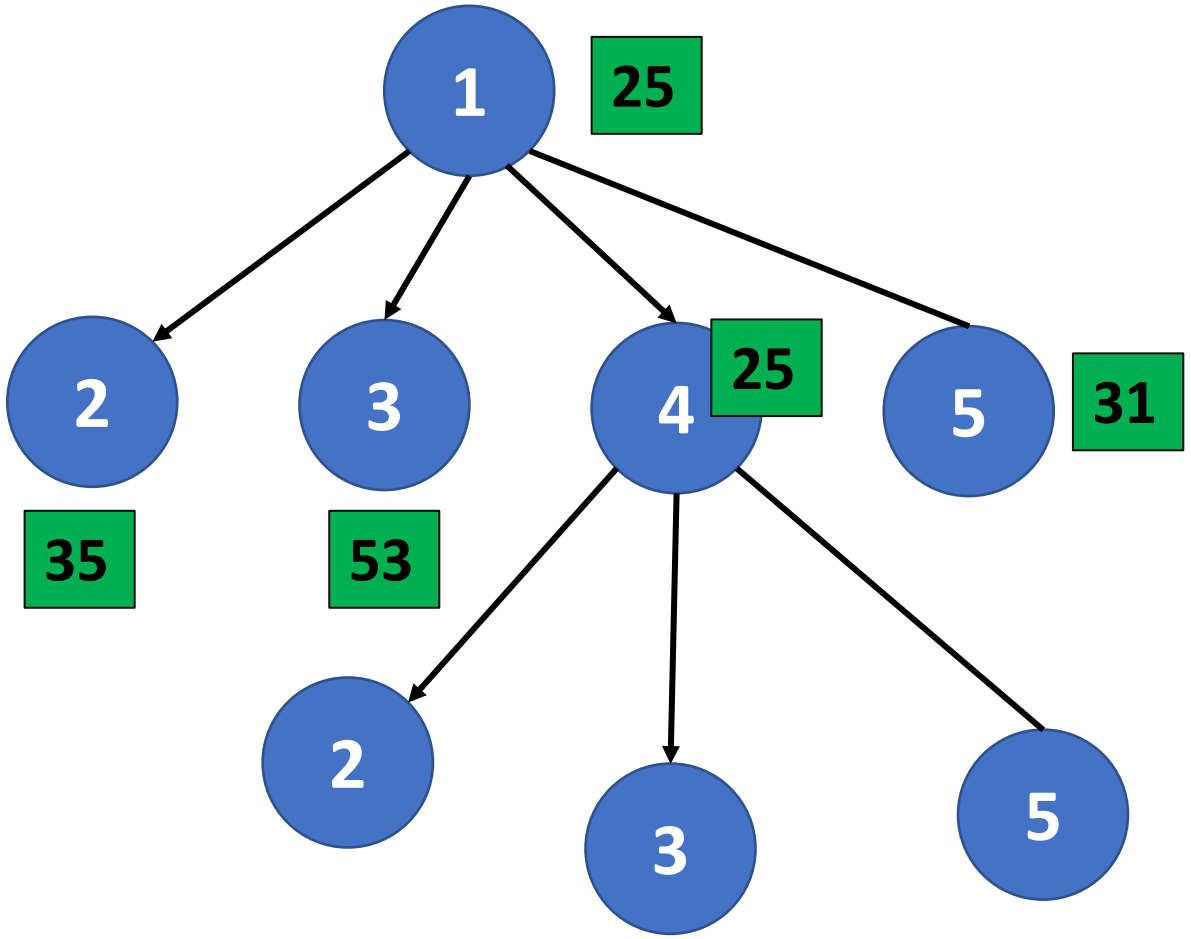
Bound = 25 + 1 + 5 = 31

TSP using Branch and Bound



C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

TSP using Branch and Bound



C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

TSP using Branch and Bound

Select path 1-4-2 : (Add edge 4-2)

Set M [1] [] = M [4] [] = M [] [2] = ∞

Set M [2] [1] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

TSP using Branch and Bound

Select path 1-4-2 : (Add edge 4-2)

Set M [1] [] = M [4] [] = M [] [2] = ∞

Set M [2] [1] = ∞

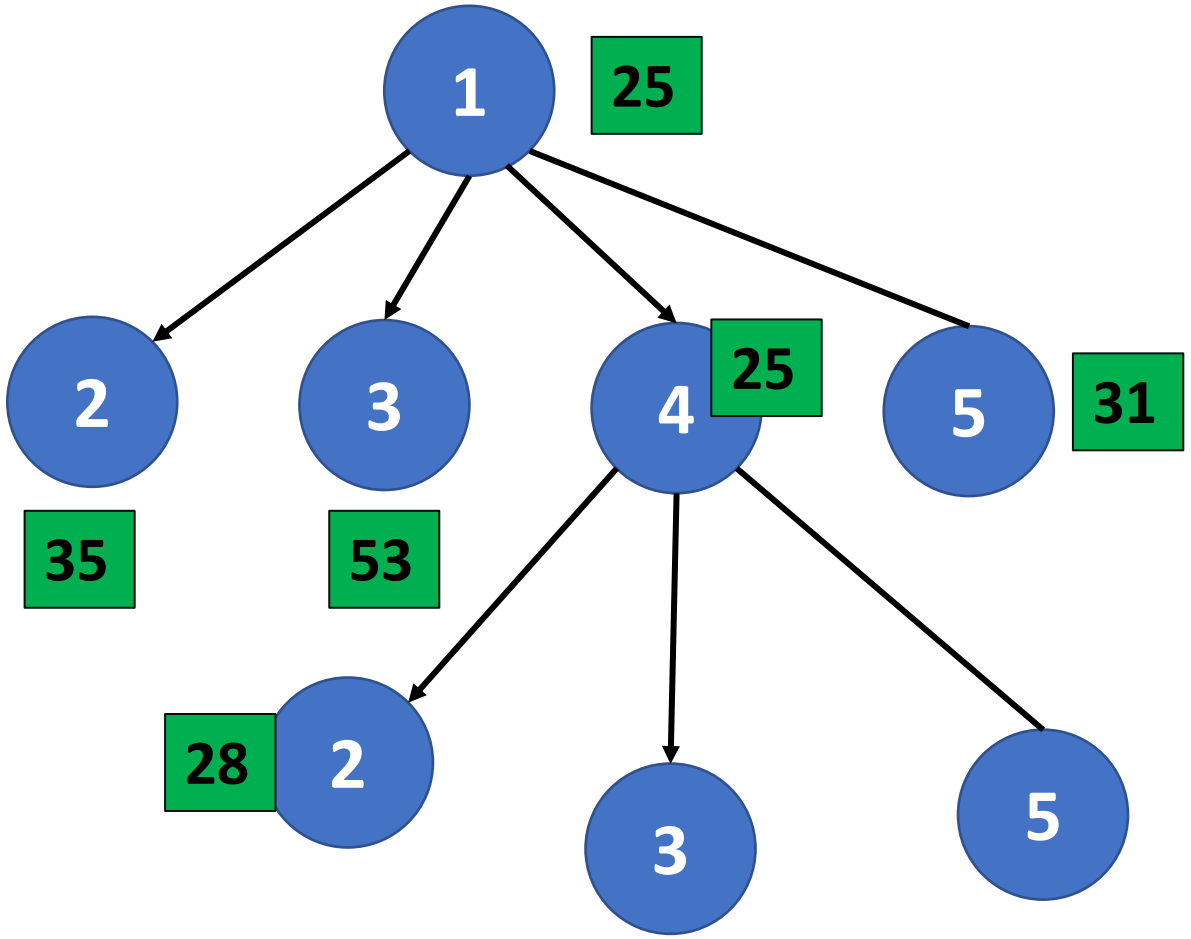
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

Already Reduced

Bound = 25 + 3 + 0

= 28

TSP using Branch and Bound



TSP using Branch and Bound

Select edge 4-3 (Path 1-4-3):

Set M [1] [ ] = M[4] [ ] = M [ ] [3] = ∞

Set M [3][1] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	∞	∞	0
3	∞	3	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	0	∞	∞	∞

TSP using Branch and Bound

Reduce :

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	∞	∞	0
3	∞	3	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	0	∞	∞	∞

11

2

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	∞	0
3	∞	1	∞	∞	0
4	∞	∞	∞	∞	∞
5	0	0	∞	∞	∞

Reduction Cost = 11 +2 = 13

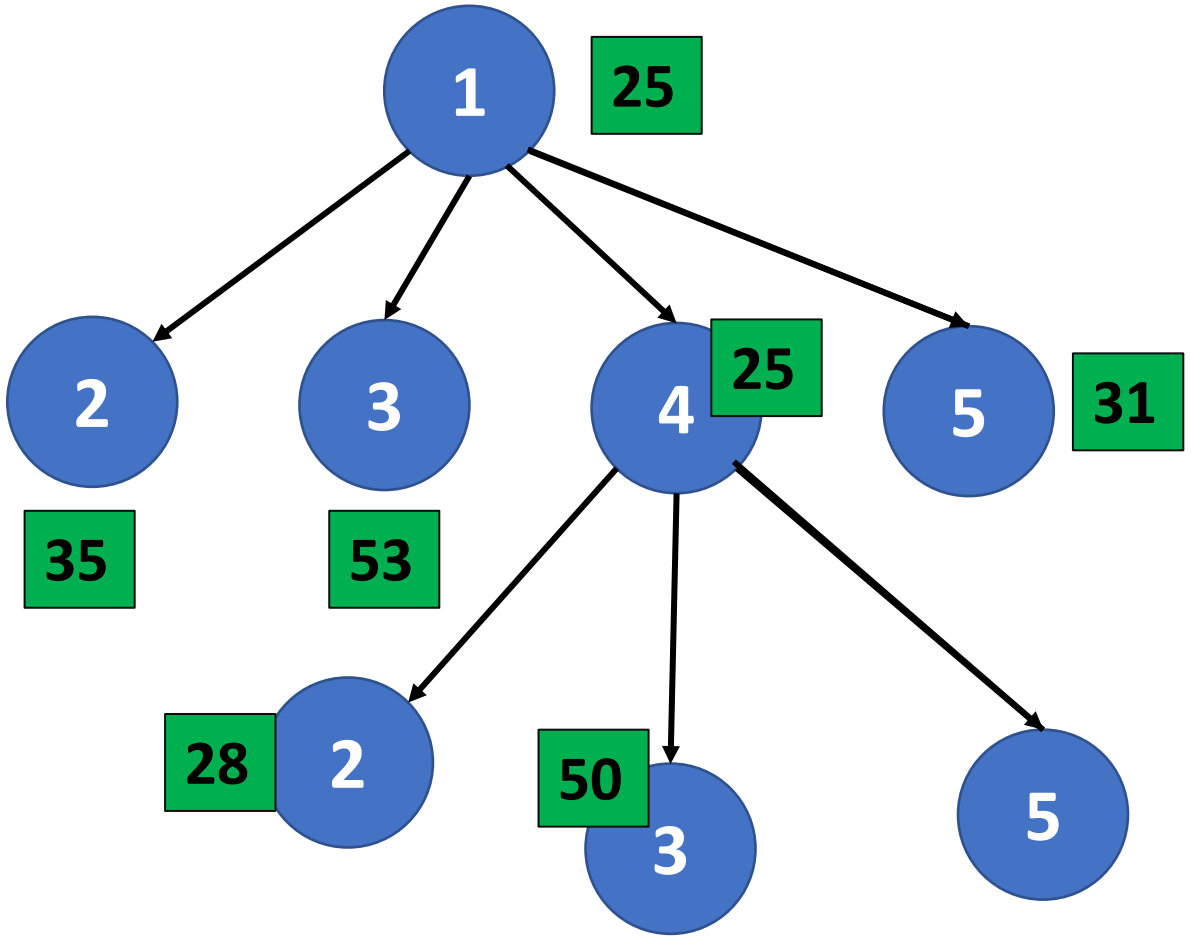
TSP using Branch and Bound

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	∞	0
3	∞	1	∞	∞	0
4	∞	∞	∞	∞	∞
5	0	0	∞	∞	∞

Bound = 25 + 12 + 13

= 50

TSP using Branch and Bound



TSP using Branch and Bound

Select edge 4-5 (Path 1-4-5):

Set M [1] [ ] = M [4] [ ] = M [ ] [5] = ∞

Set M [5][1] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

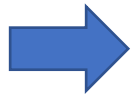
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	∞
3	0	3	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	0	0	∞	∞

TSP using Branch and Bound

Reduce

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	∞
3	0	3	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	0	0	∞	∞

11



C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	0	∞	∞
3	0	3	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	0	0	∞	∞

Reduction Cost = 11

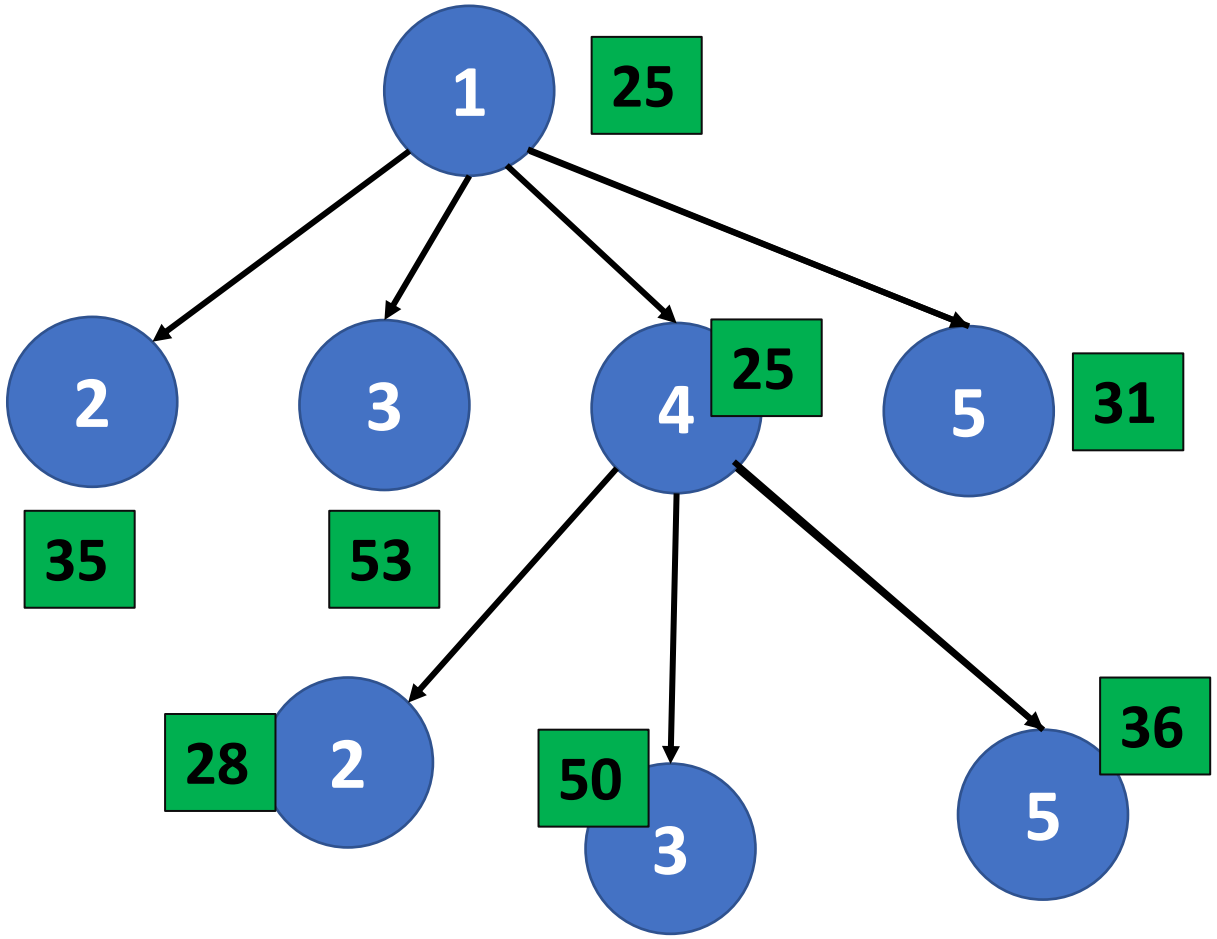
TSP using Branch and Bound

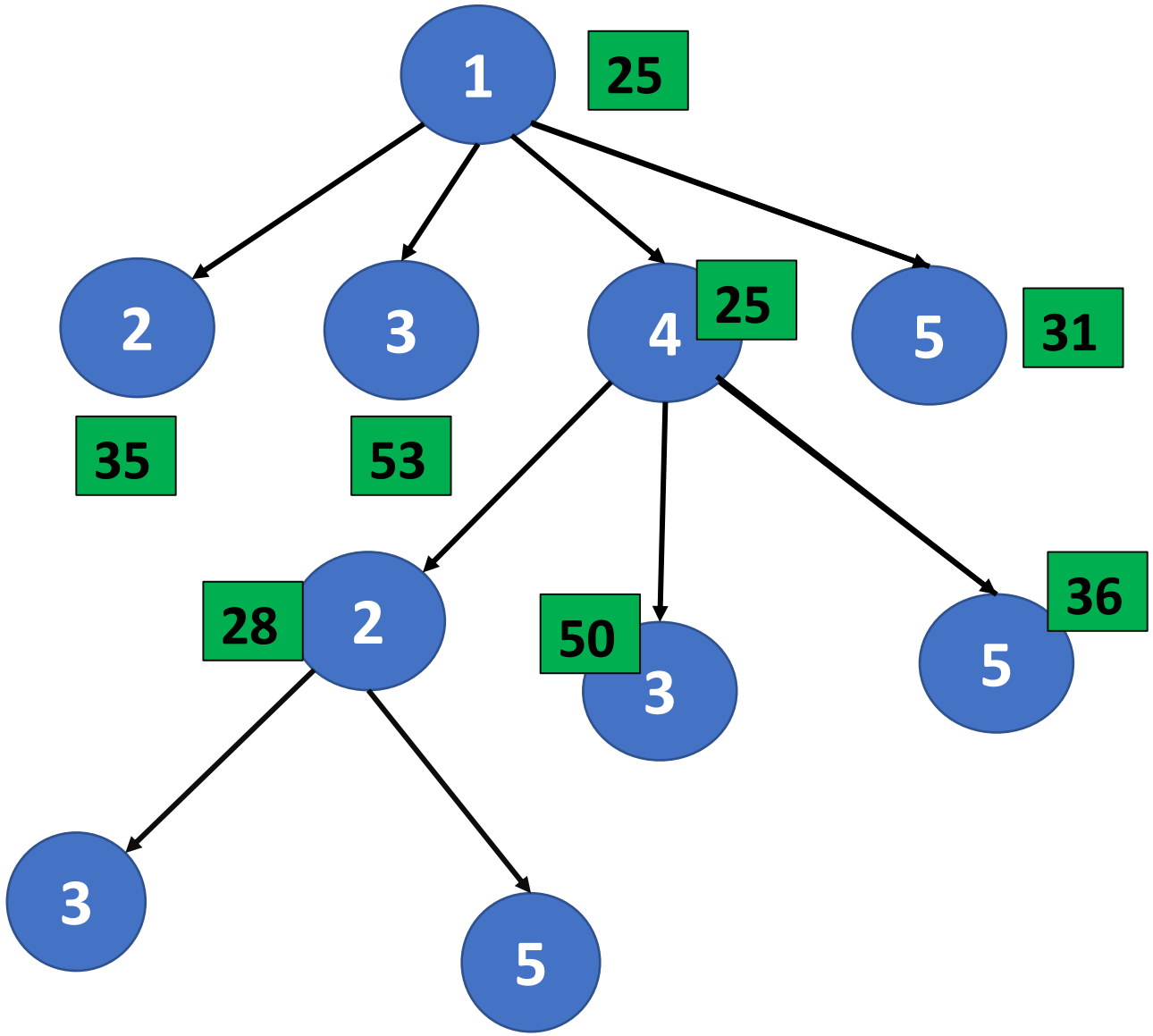
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	0	∞	∞
3	0	3	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	0	0	∞	∞

Bound = 25 + 0 + 11

= 36

TSP using Branch and Bound





C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

TSP using Branch and Bound

Add edge 2-3 (Path 1-4-2-3):

Set M [1][ ] = M [4][ ] = M [2][ ] = M [][3] = ∞

Set M [3][1] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	∞	∞	∞

TSP using Branch and Bound

Reduce

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	∞	∞	∞

11

2

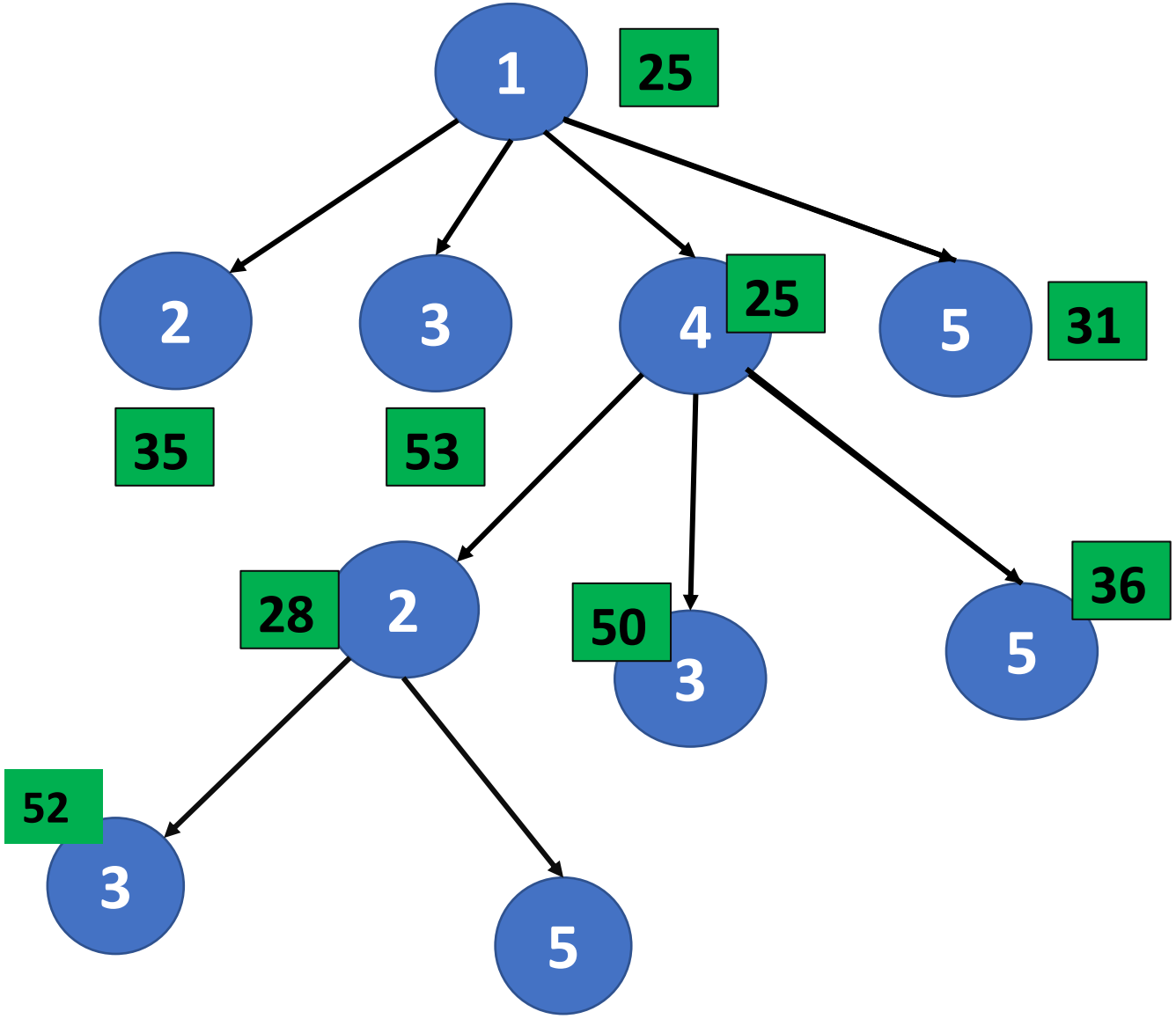
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	0
4	∞	∞	∞	∞	∞
5	0	∞	∞	∞	∞

Reduce Cost = 11 + 2 = 13

TSP using Branch and Bound

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	0
4	∞	∞	∞	∞	∞
5	0	∞	∞	∞	∞

Bound = 28 + 13 + 11
= 52


1
25
2
35
3
53
4
25
5
31
28
2
52
3
5
50
3
5
36

TSP using Branch and Bound

Add edge 2-5 (Path 1-4-2-5):

Set M [1][ ] = M [4][ ] = M[2][ ] = M [ ][5] = ∞

Set M [5][1] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	∞	0
3	0	∞	∞	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

TSP using Branch and Bound

Reduce

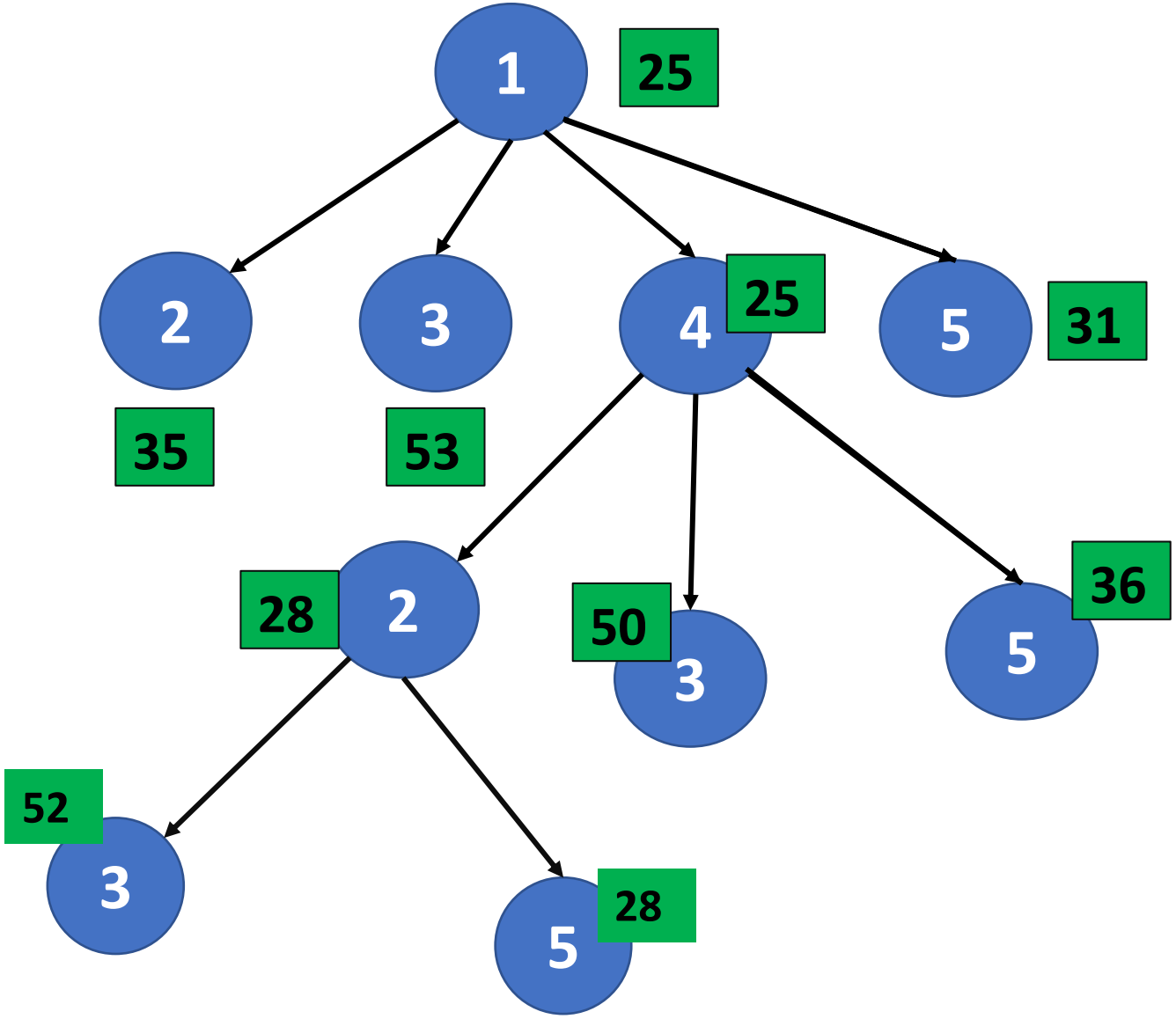
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

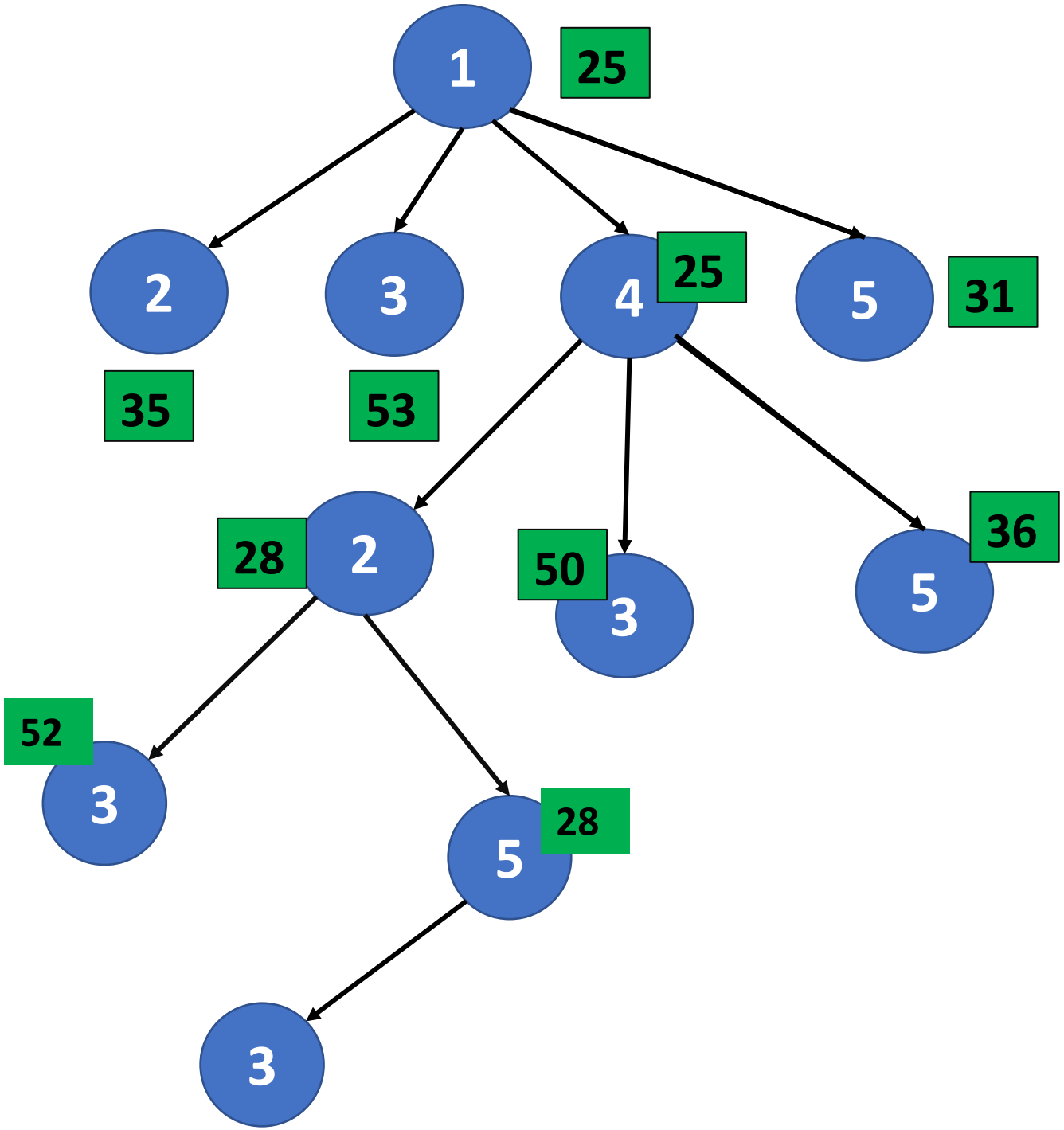
Already Reduced

TSP using Branch and Bound

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

Bound = 28 + 0 + 0 = 28


1
25
2
35
3
53
4
25
5
31
28
2
50
3
5
36
52
3
5
28



C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

TSP using Branch and Bound

Add edge 5-3 (Path 1-4-2-5-3): Set M[5][] = M[][3] = ∞

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	0	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	0	∞	∞

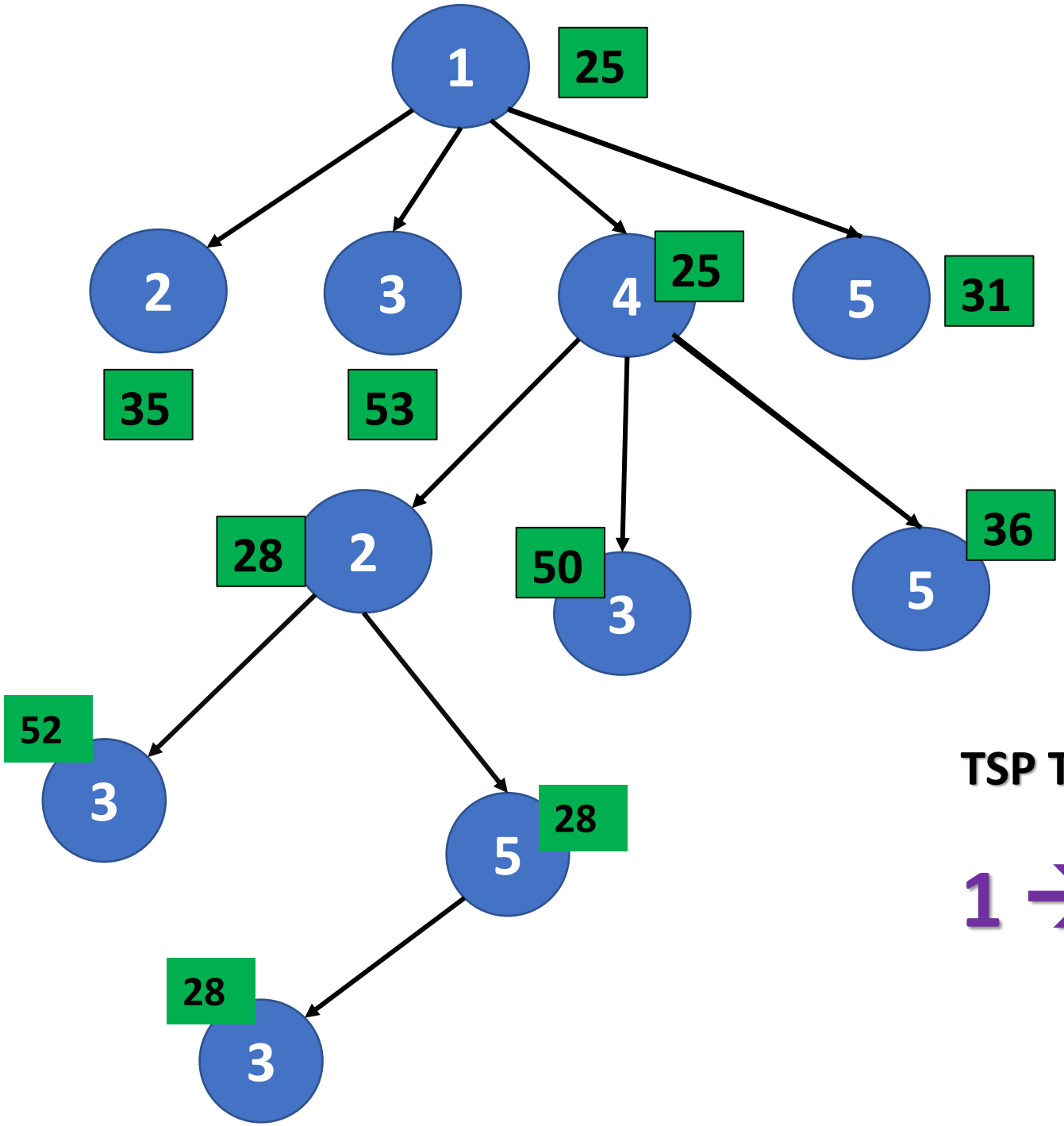
C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞

Reduction Cost = 0

TSP using Branch and Bound

C	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞

Bound = 28 + 0 + 0 = 28



TSP TOUR is

1 → 4 → 2 → 5 → 3 → 1