20/10/2022

BACK-TRACKING:- (Only for non-optimization problems)

Given a problem and some constraints, Find a soln that satisfies these constraints.

↳ **Idea:-** Construct solutions one component at a time and evaluate such partially constructed candidates as follows.

How to Find soln? ⟹ *

Construct Partial soln which satisfies these constraints [one component at a time]

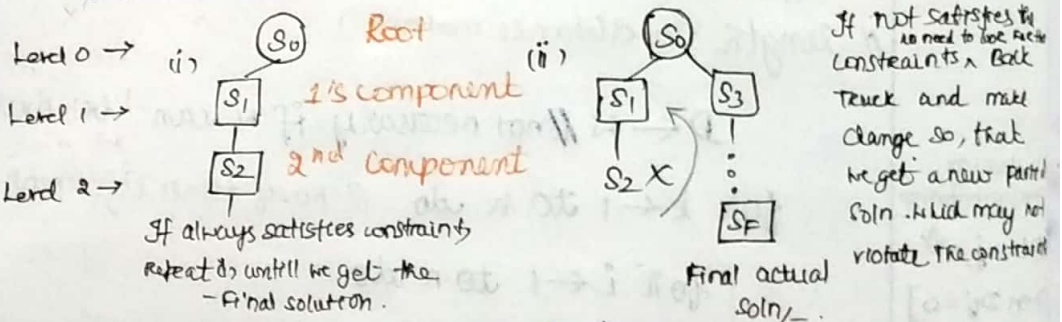
↳ (i) If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the 1st remaining legitimate option for the next component. // Repeat above process if not satisfied go to (ii) below.

Evaluate these partially constructed solutions. ⟹

↳ (iii) If there is no legitimate option, for the next component, no alternatives for any remaining component need to be considered.

(ii) Backtracking - going back to previous stages, make some changes to the partial soln and now check whether we can proceed further. If so, then continue else, repeat the same process.

↳ In this case, the algorithm backtracks (going back to previous stages) to replace the last component of the partially constructed soln. with its next option.



NOTE

[Root (S_0) is the Initial cond.]

↳ **State-space tree:-** (tree of choices)

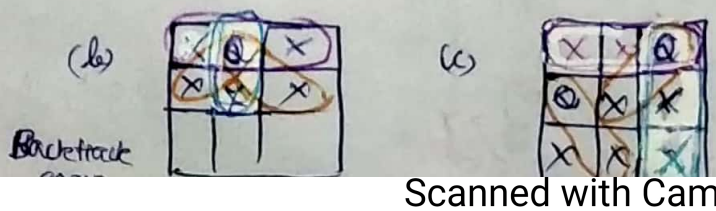
↳ Its root represents an initial state before the search for a soln. begins.

↳ The nodes of the first level in the tree represent the choices made for the first component of a solution

↳ The nodes of the 2nd level represent the choices for the second component and so on;

[$S_1, S_2 \ldots S_F$ are collectively called "state space tree"]

Starting from root constructing component (P. Soln)



[Leaves that still have scope for extending the soln and may lead to find the final soln]

else non-promising leaves

↳ Promising :-

⇒ A node in a state space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution.

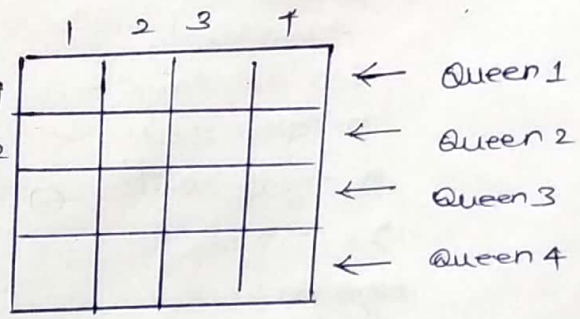
Leaves represent either non-promising dead ends (or) complete solutions found by the algorithm.

⇒ N-QUEEN PROBLEM :-

[Given NxN chessboard and 'N' queens]

→ The problem is to place 'n' queens on an n x n chessboard so that no 2 queens attack each other by being in same row or, same column (or) in same diagonal.

Eg :- [4-Queens - Problem]



∴ For 4 Queen - problem [SOLUTION ↓ - EXISTS]

(Refer next page)

↳ Do we have solution for 1 Queen problem? ✓

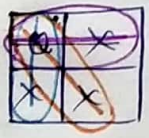
yes [since there is only one box]

[Backtracking not only means going back to the last previous partial soln, it may be any of the previous one]

↳ For 2 Queen problem ? NO

HINT

[Same row
Same column
Diagonal]

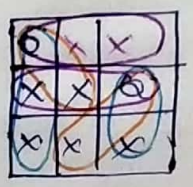


← 1st Queen

can't insert 2nd Queen

↳ For 3 Queen problem ? NO.

(a)



← 1st Queen

← 2nd Queen

[We can't change the position of 2nd Queen so Backtrack to 1st Queen and change its position and try on].

* Refer pdf theory (4)

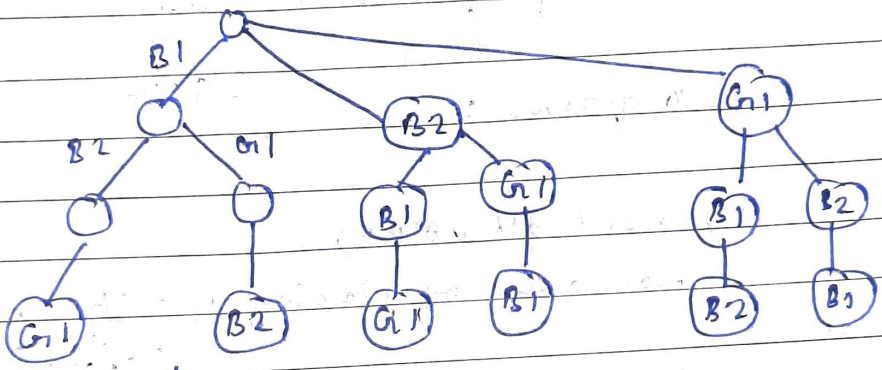
BACKTRACKING

(brute force approach)

- Try all possible solns, pick best.
  This is not used for optimisation problem. Its used when we get multiple solutions and we need all of them.
- Represent soln in form of state space tree

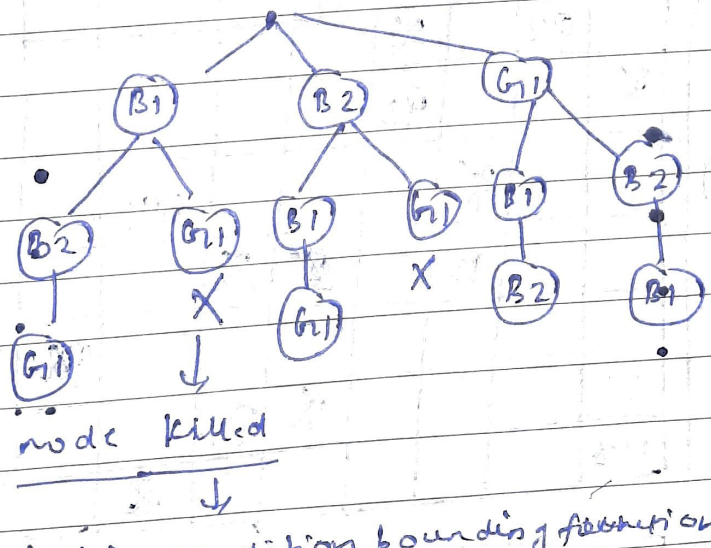
* Example: arrange B, B2 G1,

State sp. Tree:



represents all 6 possibilities.

* Backtracking always has constraints say G1 not in middle.



node killed

↓

imposed/this condition bounding function
killed through

if no bounding function is applied until we reach last level → its a solution

* Branch n bound also uses brute force approach:
  - Backtracking – follow DFS
  - B & B – follow BFS

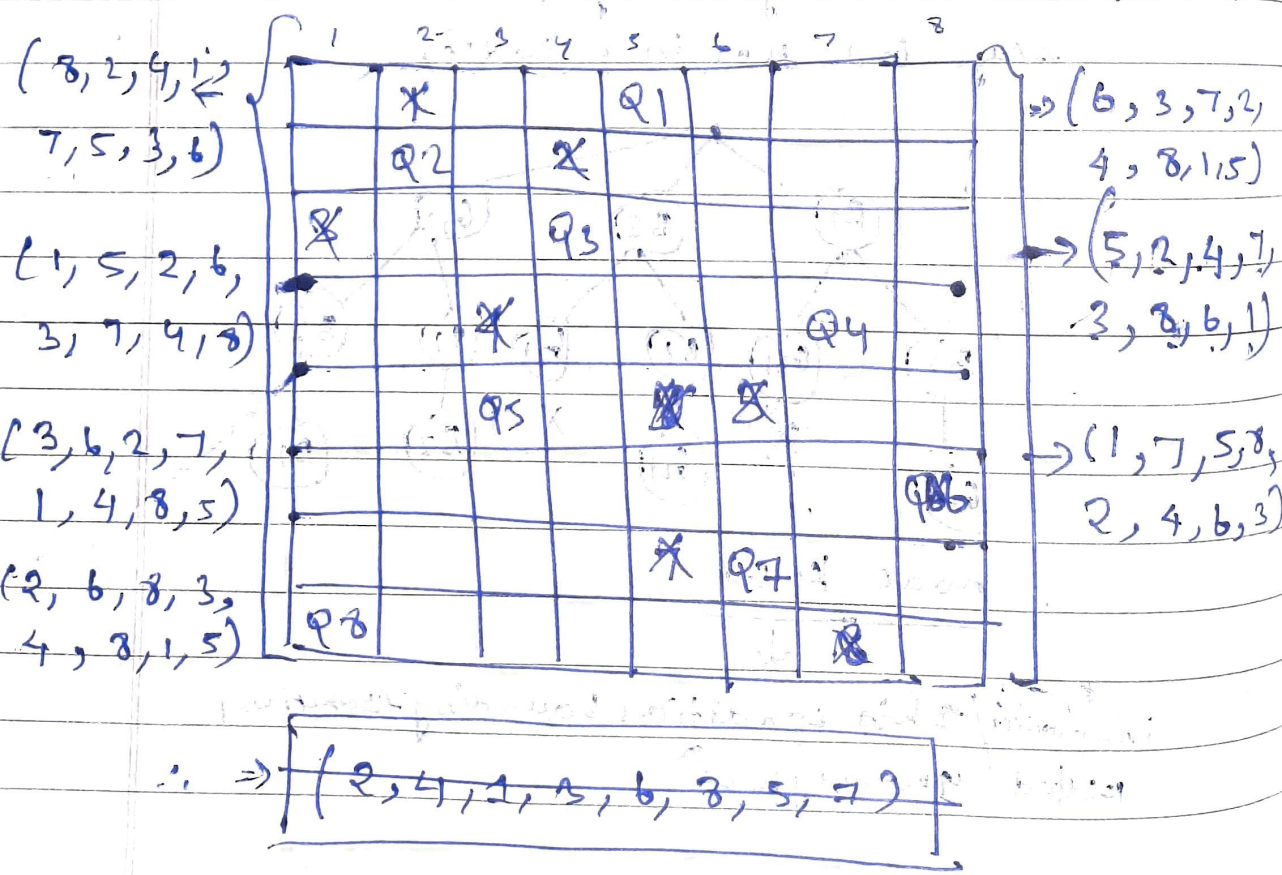
* N-Queen's problem:

For 4 queens : 2413 (col nos of (q_1, q_2, q_3, q_4))
or
3142

8 queens :

We need all solns;
no. ways of placing, Q = 16 C 4
(4 x 4)

To reduce; we enable constraints
Not same row, col, ~~diagonal~~, dicgonal.



∴ ⇒ (2, 4, 1, 3, 6, 8, 5, 7)

Max no. of nodes in SST :

$$1 + \sum_{i=0}^{n-1} \left[\prod_{j=0}^{i} (n-j) \right]$$

For 4-queens;

$1 + 4 + 4 \times 3 + 4 \times 3 \times 2 + 4 \times 3 \times 2 \times 1 = \boxed{65}$

* Other examples

Graph coloring, sum of subsets.

* Hamiltonian cycle

If graph is given, we must start from a starting vertex, visit all other nodes exactly once and return back to the starting vertex. → forms cycle.

Problem - To find all hamiltonian cycles.
(similar to travelling salesman)

Travelsalesman → minimization

Hamiltonian → Not so

* conditions :

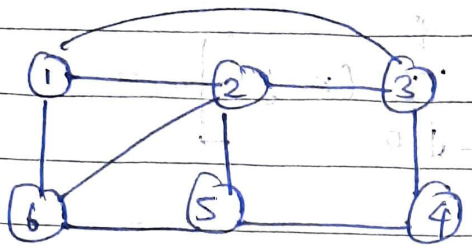
1. all nodes must be connected
(directed / non-directed)
2. This is NP-Hard problem
(exponential time taking)

So, there's no easy way / shortcut to find if Hamiltonian cycle is present in the graph / not.

↓

only manual.

Example:



c1 - 1, 2, 3, 4, 5, 6, 1

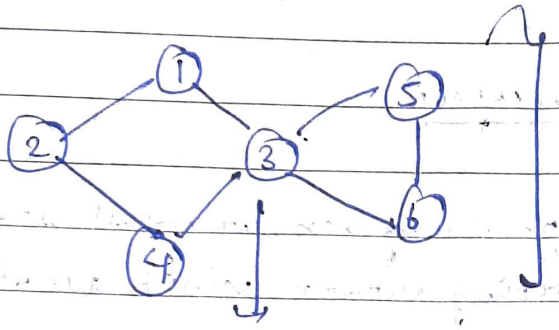
c2 - 1, 6, 2, 5, 4, 3, 1

c3 = 1, 2, 6, 5, 4, 3, 1

~~c4~~ - 1, 3, 4, 5, 2, 6, 1

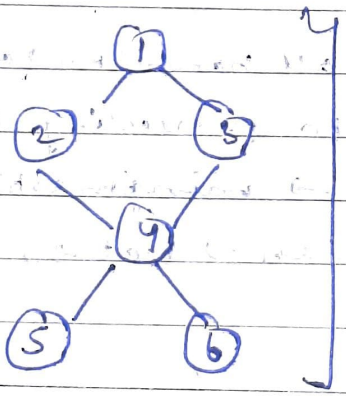
4 wrong (same as c2)

~~c5~~ = 2, 3, 4, 5, 6, 1, 2 (same as c1)



articulation point

Hamiltonian not possible because this (3) is junction/connecting point of 2 graphs, 3 must be repeated.



Hamiltonian doesn't exist because of pendant vertices (5, 6)

To solve problem, use an array, draw state space tree accordingly.

1. Construct a ~~state space tree~~ Adjacency matrix from given graph
2. put values in array s.t. there's no repetition, after assigning check if its before value & the present value share an edge.