CSIT113
Problem Solving

Week 10

Please enrol in your Spring 2020 subjects as soon as possible.

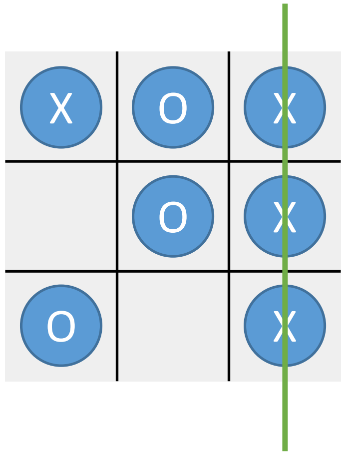
ADVANTAGES:

- To ensure that you get into the computer lab or workshop of your choice
- To map out your timetable for Spring session
- To get details of your subject

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

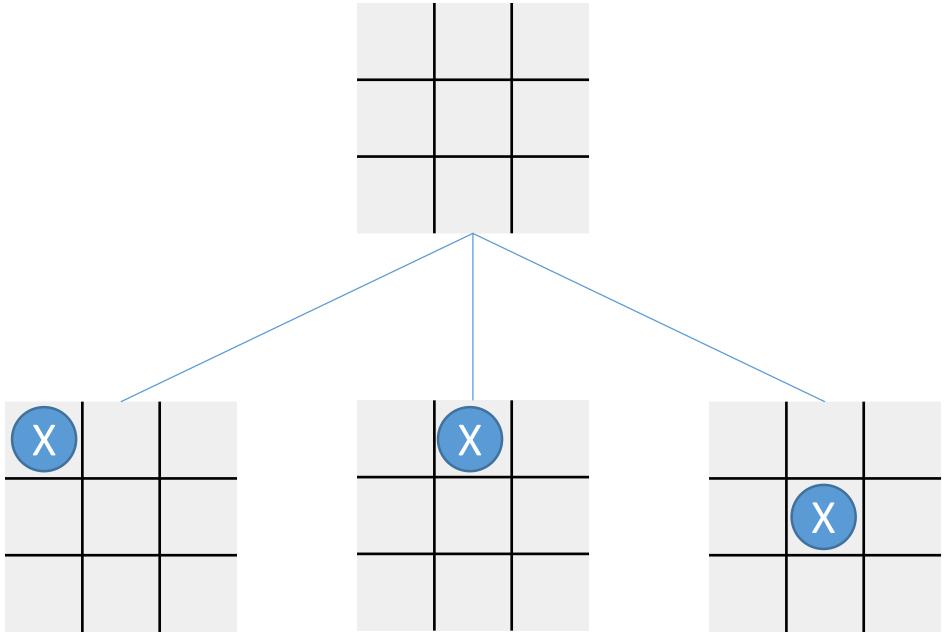
Games and Graphs

- Consider the game of Tic-Tac-Toe (Noughts and Crosses)
  - Two players X and O alternate play on a 3×3 grid.
  - Each player puts their symbol in one of the empty squares.
  - The winner is the first player to establish a line of three of their symbol.
    - Horizontal
    - Vertical
    - Diagonal
  - A draw is possible
  - Consider the following sample game:
  - …and X wins!



Games and Graphs

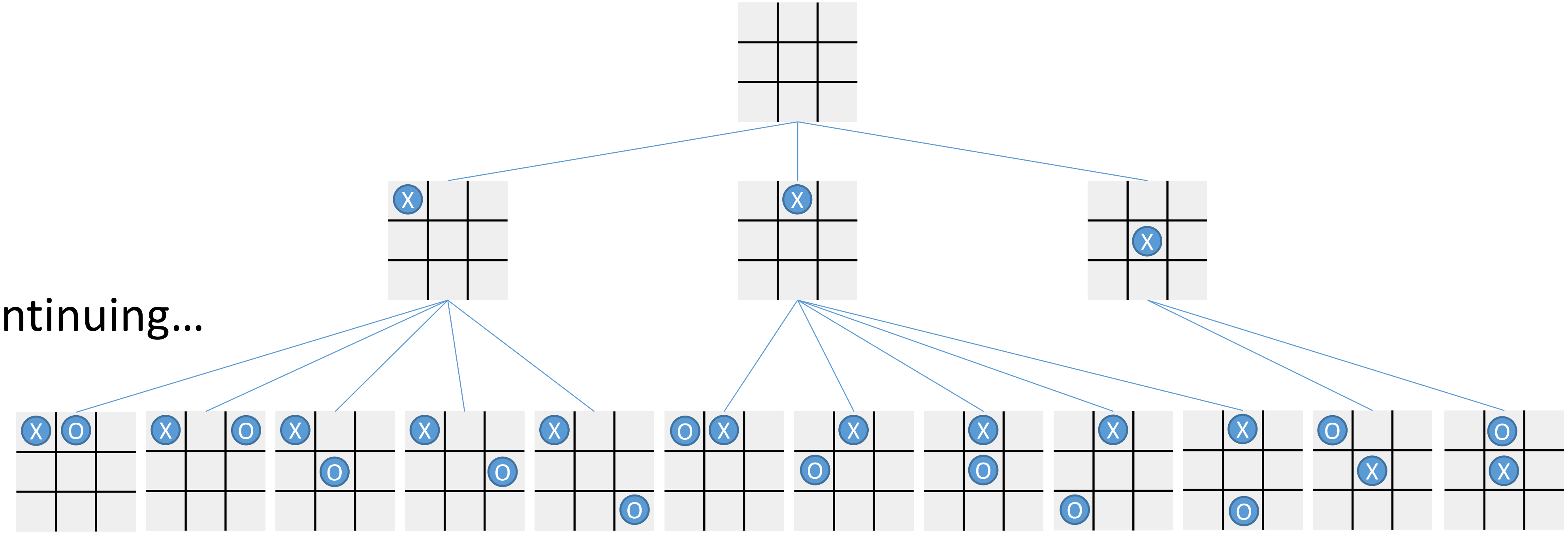
- We can construct a tree showing all possible positions in the game.
- The root is an empty board.
- The next level shows all possible first moves



- Note: there are only three possible first moves
  - The rest are reflections or rotations of these.

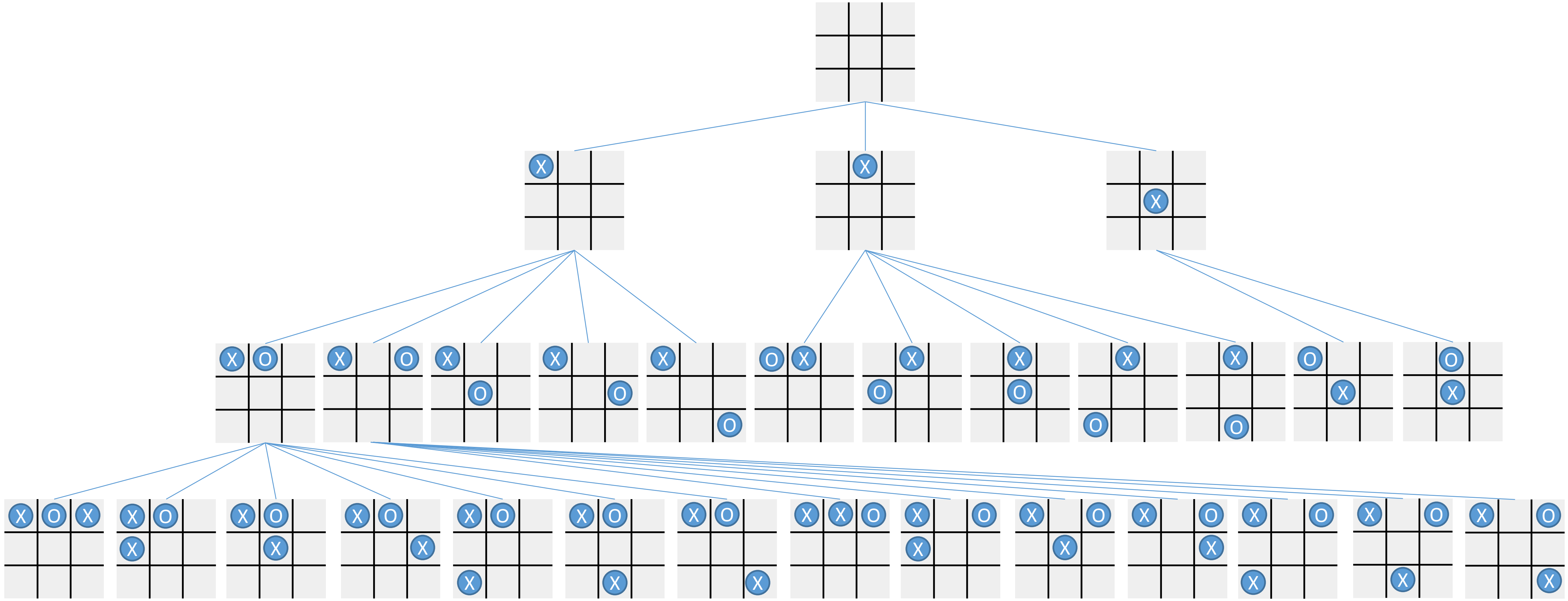
Games and Graphs

- Continuing…



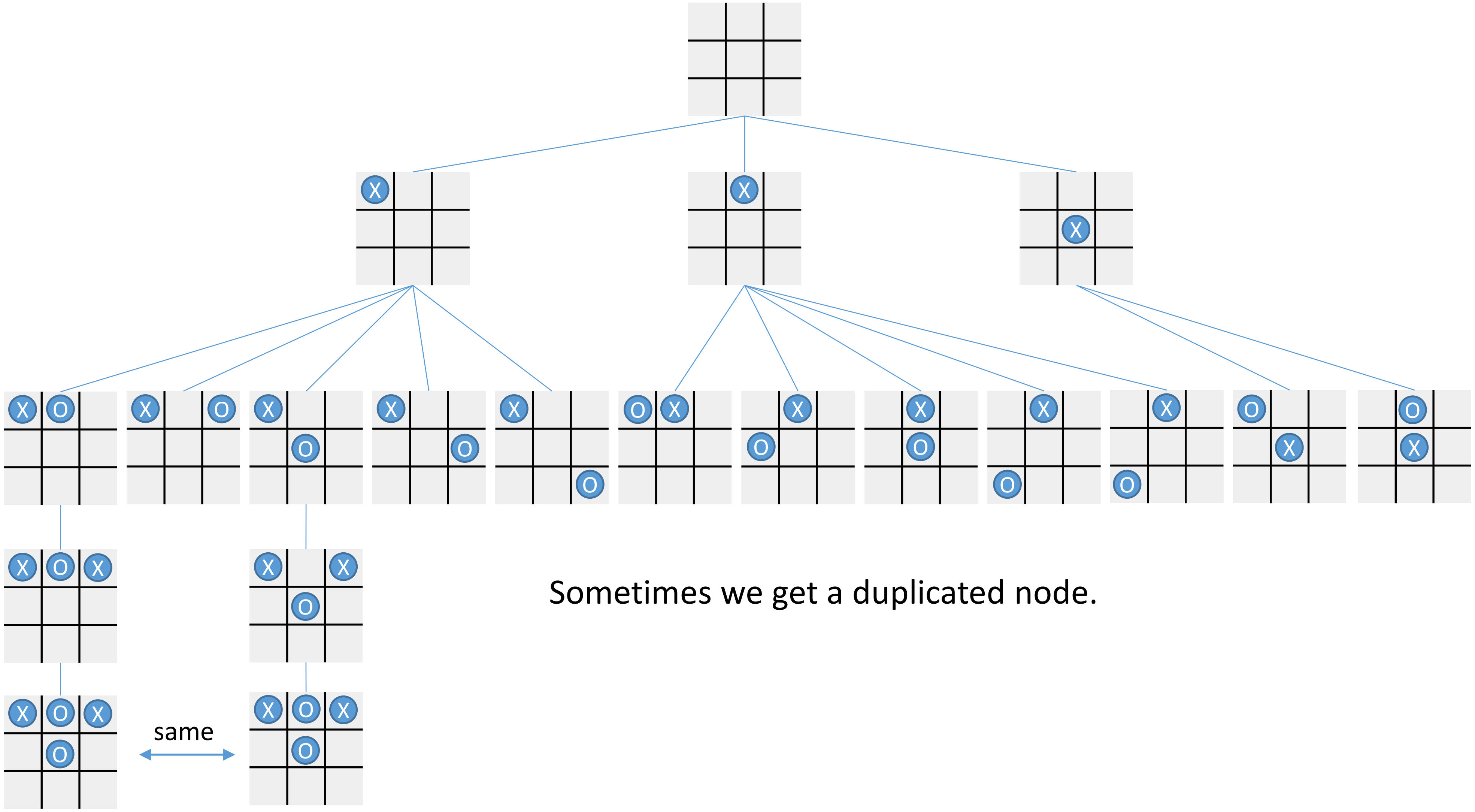
- And so on…

Games and Graphs



As you can see this is going to be a pretty big tree!

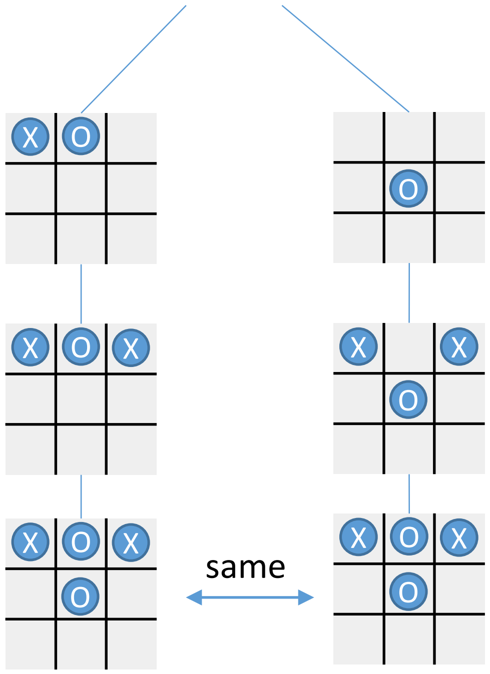
Games and Graphs



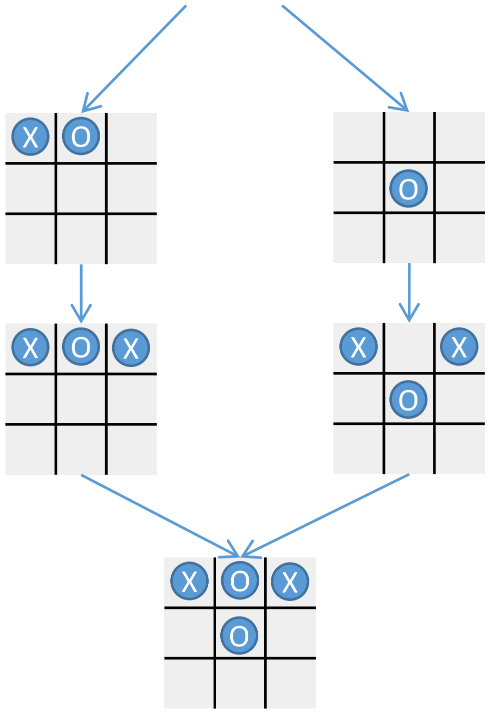
Sometimes we get a duplicated node.

Games and Graphs

- We can eliminate duplicated nodes
- This turns our game tree into a game graph
- This game graph is directed and acyclic

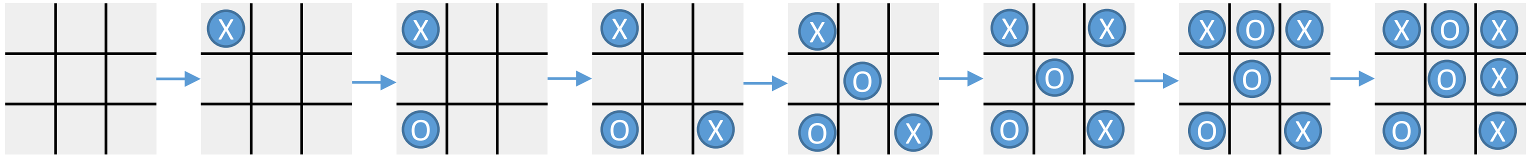


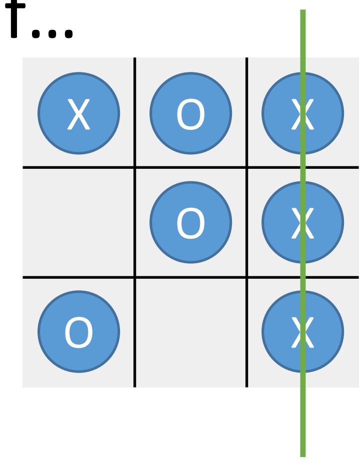
Becomes…



Games and Graphs

- The terminal nodes (leaves) of the game graph correspond to winning/losing or drawn positions.
- Thus, our sample game is one path from the root to a leaf…





- …is one possible complete game.

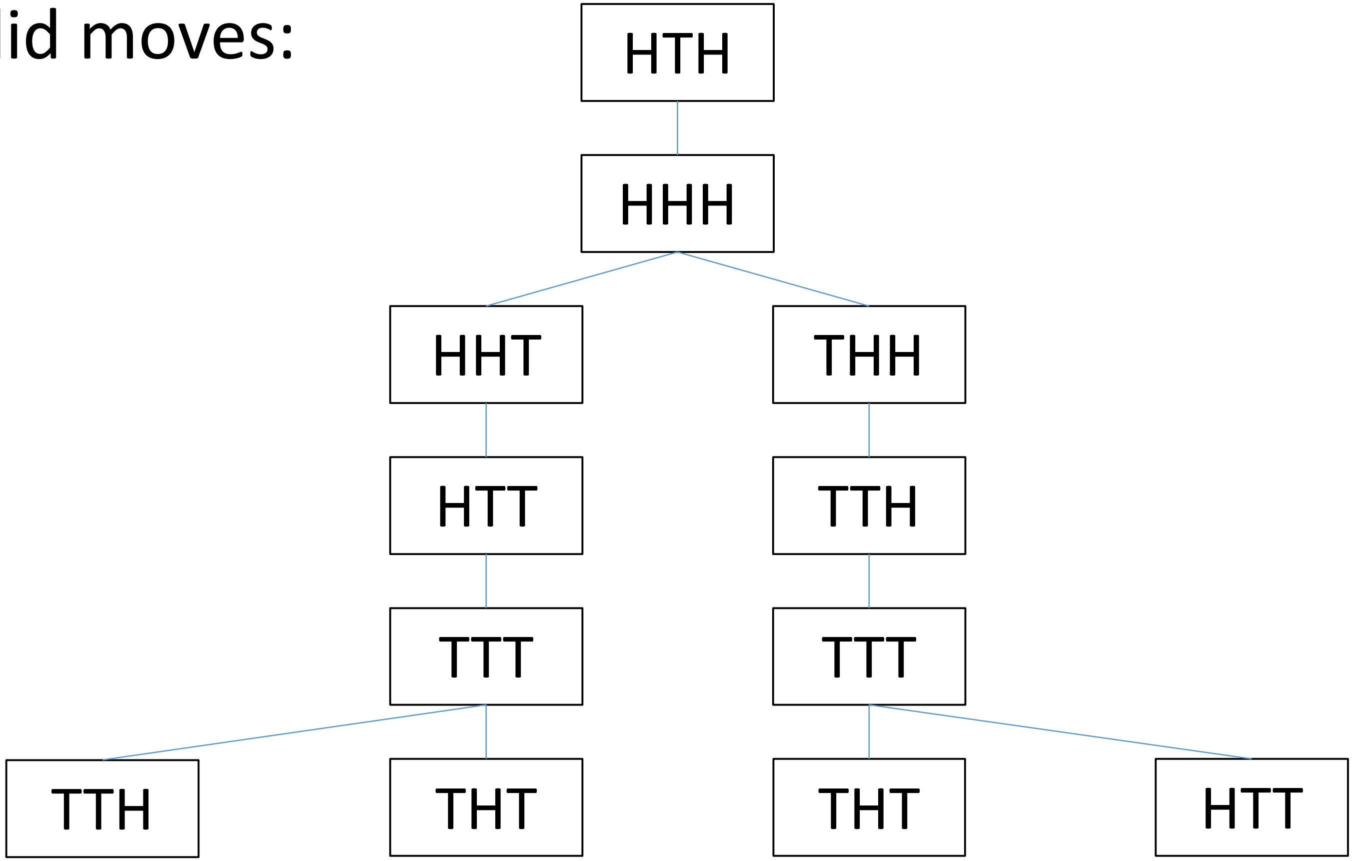
A Solitaire Game.

- Let us now look at a game with a much simpler game graph.
- We start with three coins…

- We can flip the centre coin at any time.
- We can flip either end coin only if the other two coins show the same face.
- Starting Position is HTH
- Ending Position is THT

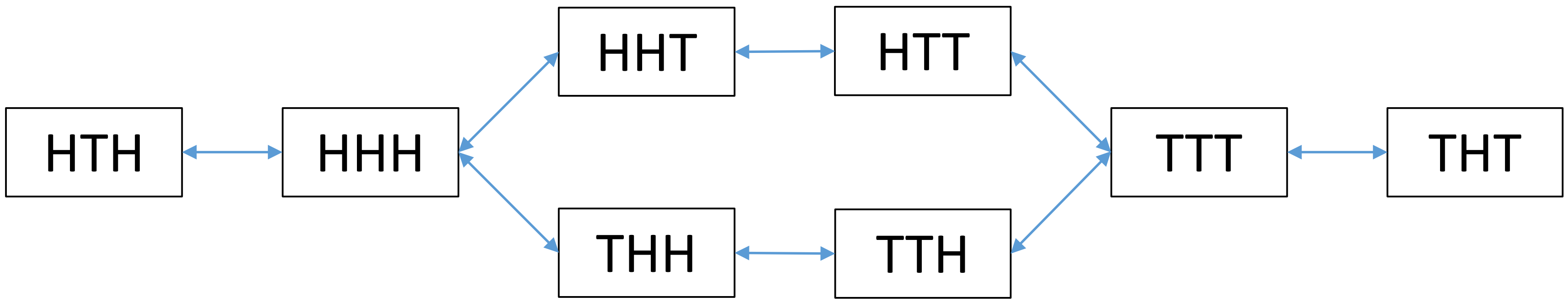
A Solitaire Game.

- We can construct the game tree by starting at the initial position and trying all valid moves:



A Solitaire Game.

- We can simplify this by combining duplicated nodes to get the equivalent game graph.



- Any path from the start node to the end node is a solution to the game.

State-Trees and State-Graphs

- We can create a tree or graph for any problem – not just for games.
- Each node represents a partial attempt at a solution.
- The leaves represent final states for the problem – these may be:
  - Solutions
  - or Dead ends.
- The difficulty is to find a path from the root to a solution – without building the whole tree.
- Any solution strategy can be viewed as a way of choosing the next node in the tree.

Brute Force Graphs

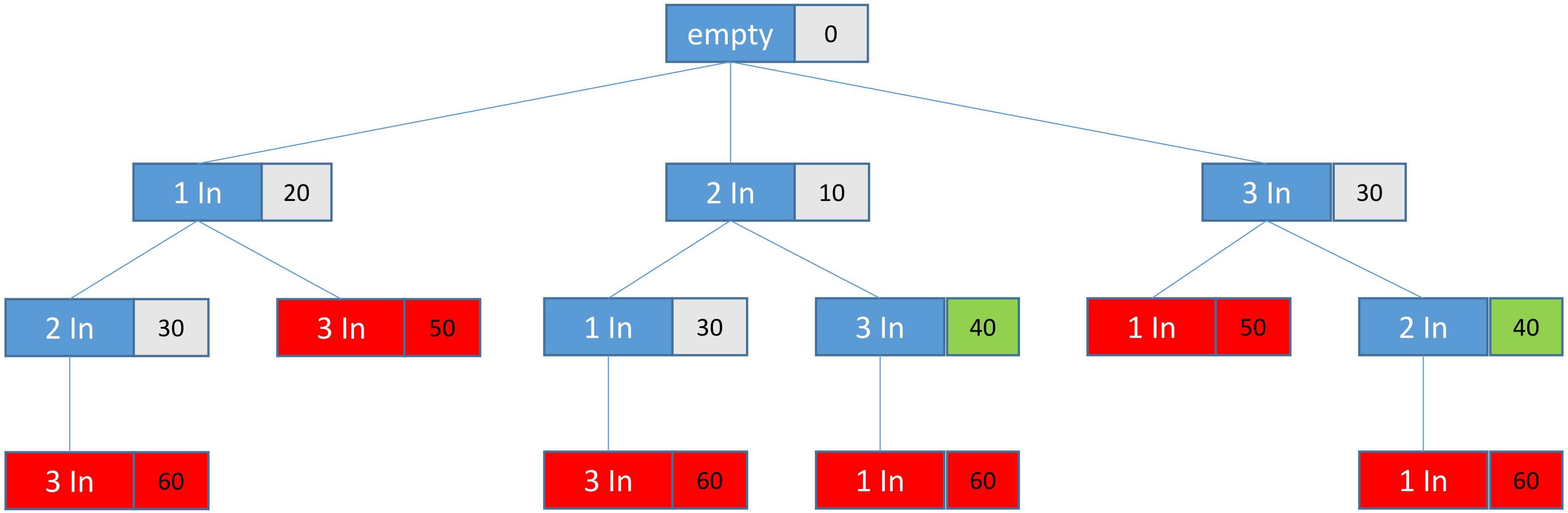
- Consider the **discrete** backpack problem:
  - We have a set of objects, each with a weight and a value.
  - We need to assemble the greatest possible value into our backpack.
  - We must put all of an object in the backpack.
- We could construct the state tree in two different ways:
  1. At each level, decide which object to add to the pack next.
  2. At each level, decide whether to add the next object to the pack or not.

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3

Backpack capacity = 7

- Tree 1:



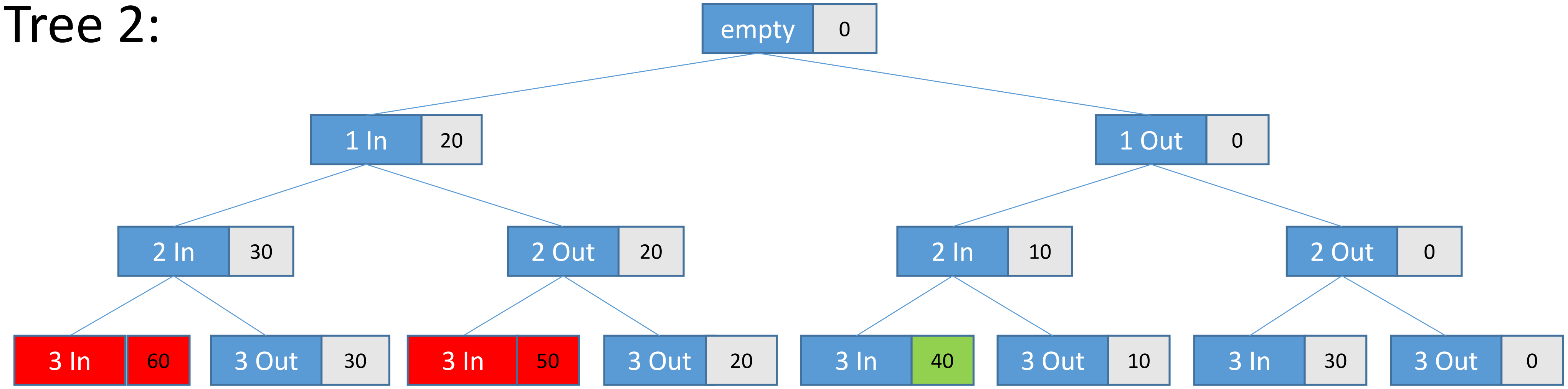
- Red nodes are illegal. Green nodes are optimal.

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3

Backpack capacity = 7

- Tree 2:



- Again Red is illegal, green is best.

What’s wrong with this?

- The big problem we have here is that we have no easy way to find the best terminal node.
- We need some way of evaluating the nodes as we go.
- The alternative is to construct the whole tree and then find the solution somewhere within it.
- If we have 10 objects we have a tree with over 1000 leaves!
- Each extra object doubles the number.
- Do not even think about trying to fit the best selection from 100 objects!

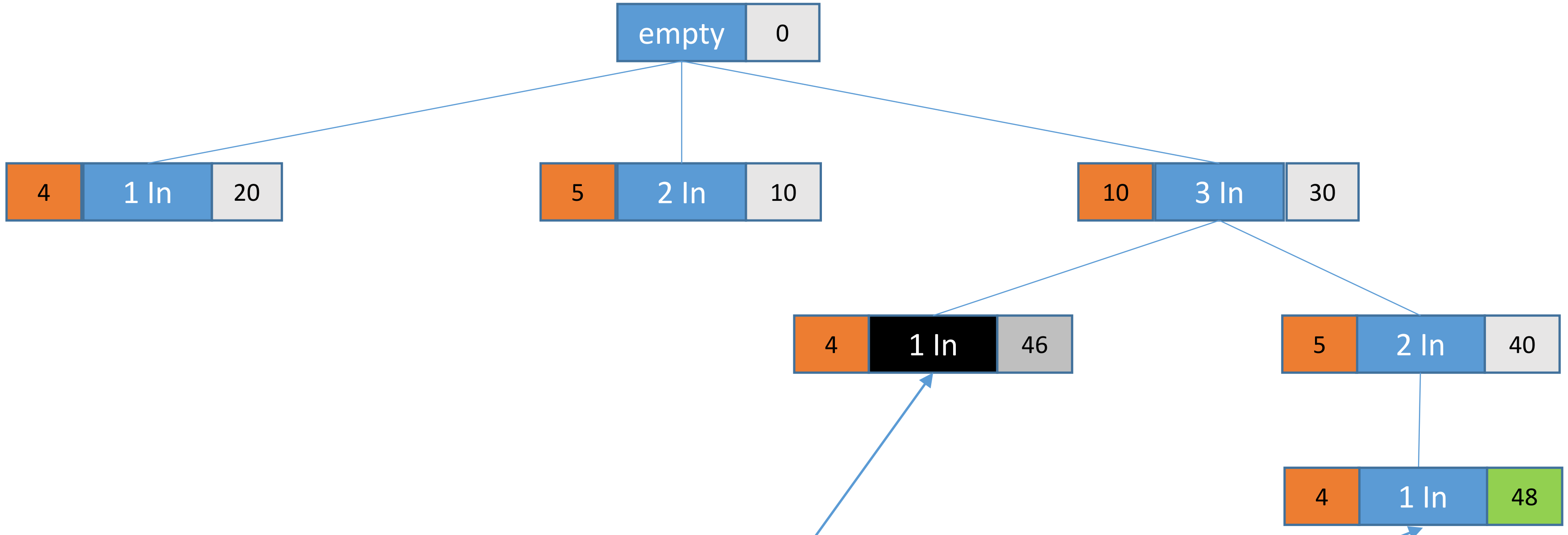
How do we fix this.

- We need some way to decide which node is the most promising at each level.
- We call this measure of “promisingness” a *metric*.
- We could then choose this node to further expand.
- This is the basis of the greedy strategy.
- Let’s look at solving the **continuous** backpack problem in terms of a state-tree.
- We will start with the same set of objects and use value/weight as our metric.

- Consider the following problem:

Object	1	2	3
Value	20	10	30
Weight	5	2	3
V/W	4	5	10

Backpack capacity = 7



Note the partial amounts of object 1: Here, and here

When Greedy Fails

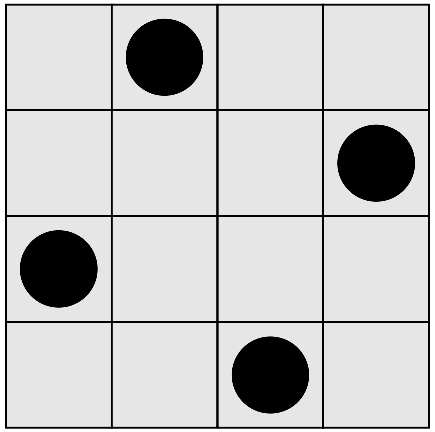
- As we have seen in previous weeks the greedy strategy does not always lead to a solution.
- Sometimes the promising node turns out to be wrong.
- What can we do in this case?
  - Brute force?
  - Something else?
- Brute force always works but often we can find a better strategy.
- We will examine one such strategy, *backtracking*.

The N Queens Problem

- This problem involves placing n chess queens on a square $n \times n$ board so that no two queens share:
  - the same row;
  - the same column;
  - the same diagonal.

The N Queens Problem

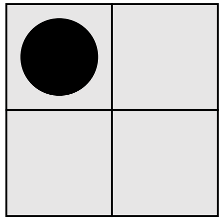
- Here is a sample 4 x 4 solution:



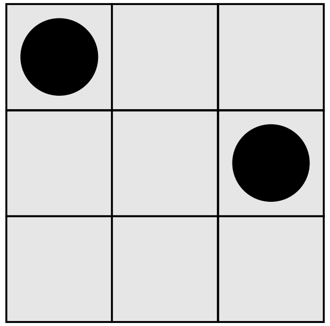
We Have a Problem

- One difficulty with this problem is that there is no solution for some values of *n*.

  - *n* = 2



  - *n* = 3



Brute Force 1

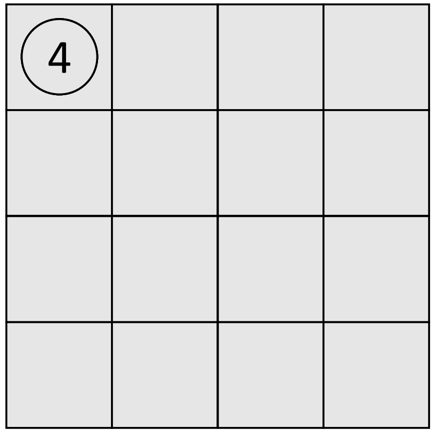
- The simplest statement of the problem (the one that uses least information) is as follows:
  - Place n queens on a square board with n^2 squares so that no two queens threaten each other.
- If we number the squares from 1 to n^2 we can try all possible values for each queen.

Brute Force 1

- Thus all queens start on square 1 and we try moving each queen to all possible squares independently.
- For a 4 x 4 board this approach looks like this…

Brute Force 1

- Start



Brute Force 1

- Fail

3	1	1	1
1	1	1	1
1			

Brute Force 1

- … much later…

1			
		1	
1	1	1	1
1	1	1	

Brute Force 1

- … much, much later…

	1		
			1
1		1	1
1	1	1	

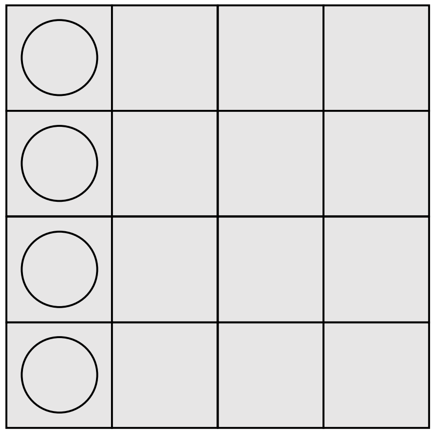
- …we get a solution!

What Next?

- This approach is clearly stupid!
- We are using none of the information inherent in the problem.
- For example, each queen must be in a different row.
- Let’s use this to construct a less brutish brute force solution

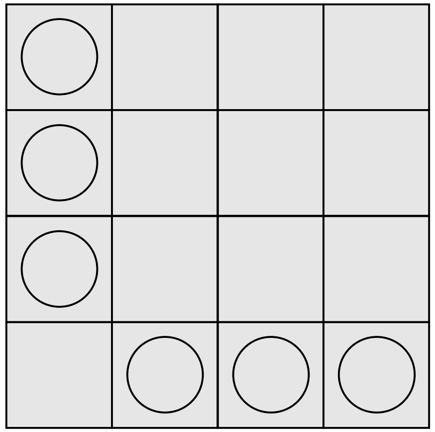
Brute Force 2

- Start



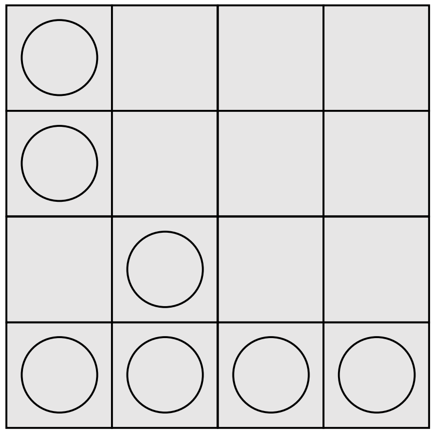
Brute Force 2

- Fail



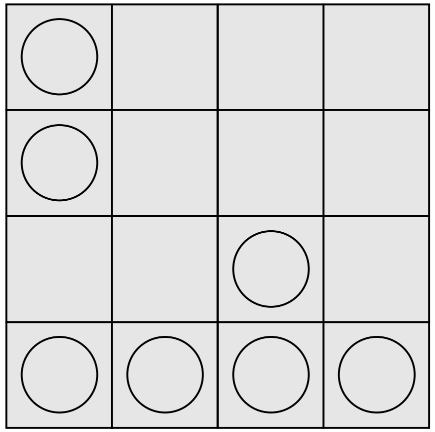
Brute Force 2

- Fail



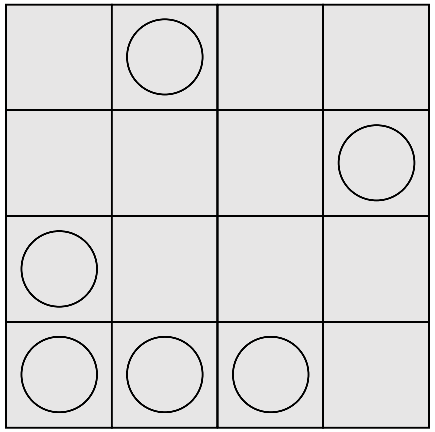
Brute Force 2

- Fail



Brute Force 2

- … some time later…



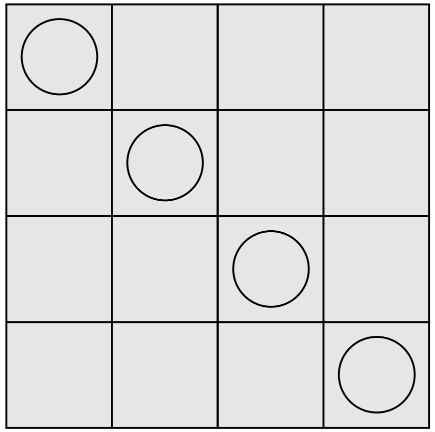
- …success!

What Next?

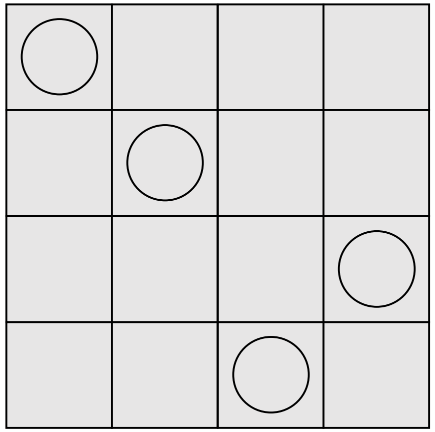
- This is better but we still are doing a lot of needless work.
- Let us add the fact that each queen is in a different column.
- We can use this to build brute force 3.

Brute Force 3

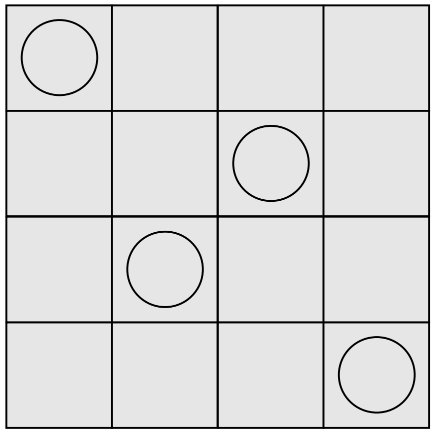
- Start



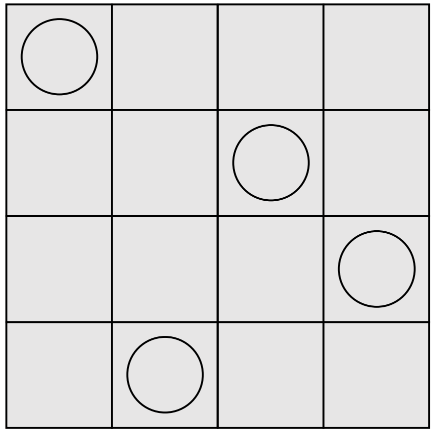
Brute Force 3



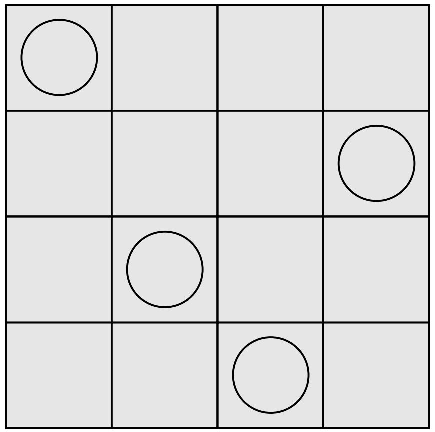
Brute Force 3



Brute Force 3

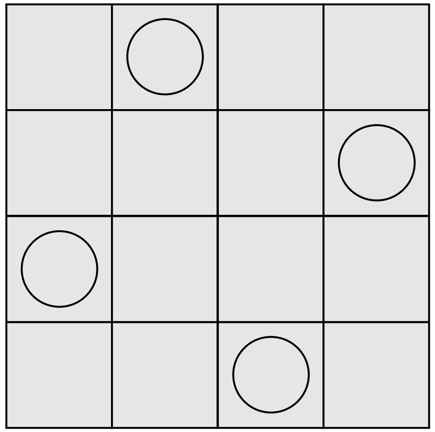


Brute Force 3



Brute Force 3

- … some time later…



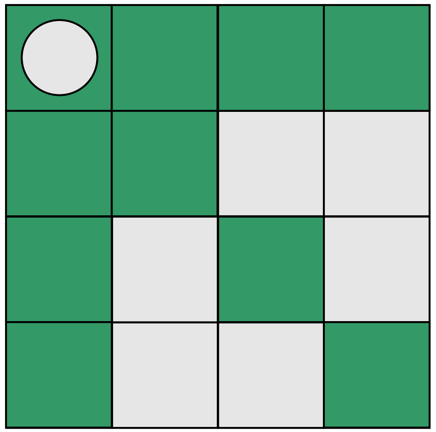
- …success.

What Next?

- This is even better but we still are doing a lot of needless work.
- Perhaps our basic strategy, placing all the queens at once, is at fault.
- What if we try placing them one at a time?

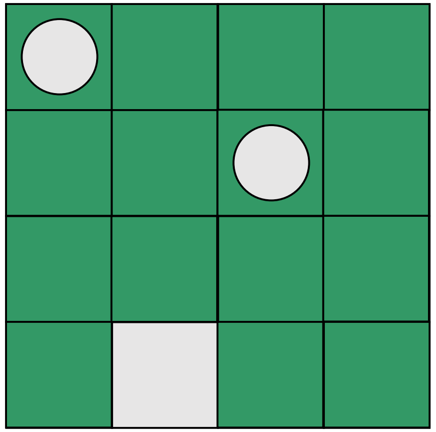
One-By-One

- Place a queen in row 1



One-By-One

- Place a queen in row 2



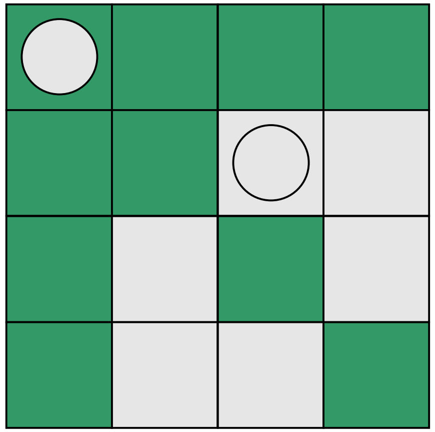
- We cannot place the next queen!
- Now what do we do?

Backtrack

- When we placed each queen we took the first available square.
- But there were other squares available.
- What we have to do is go back to the last place we had another move available to us and try it instead.
- This procedure is known as backtracking.

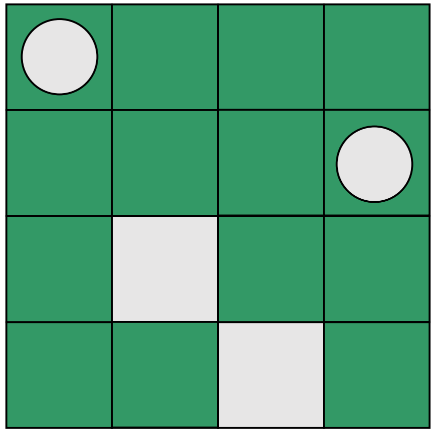
One-By-One

- This didn't work



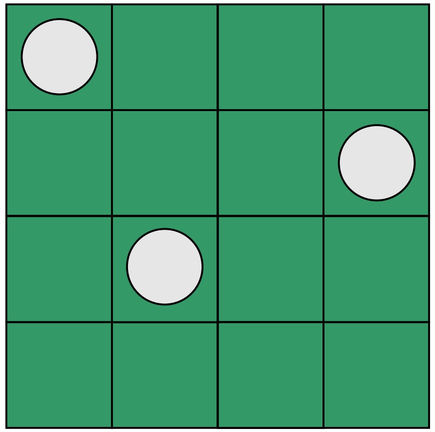
One-By-One

- So try this



One-By-One

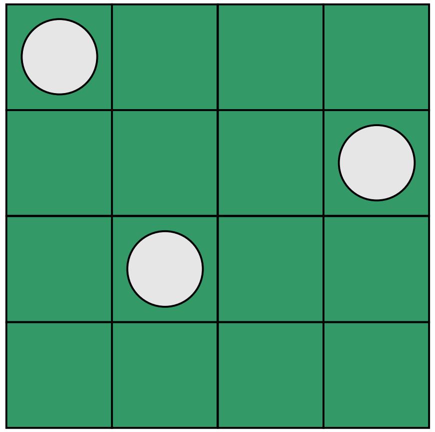
- Place a queen in row 3



- Again, we cannot continue.

One-By-One

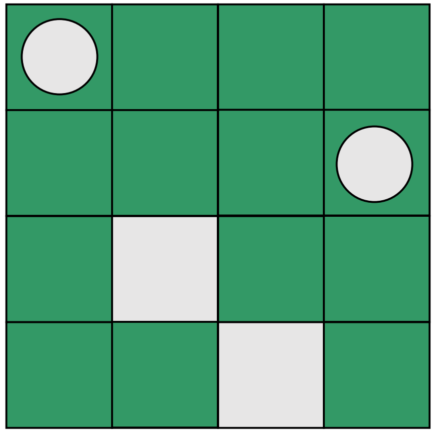
- We cannot place a queen in row 4!



- Backtrack

One-By-One

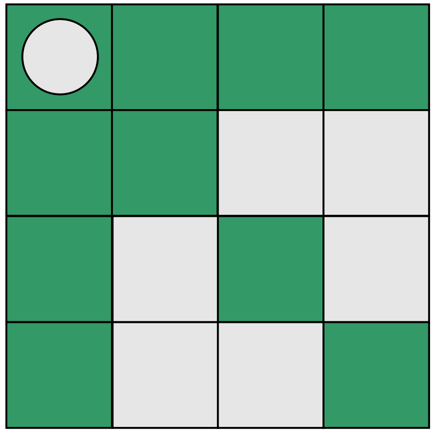
- And we have run out of possibilities on row 2



- Backtrack again!

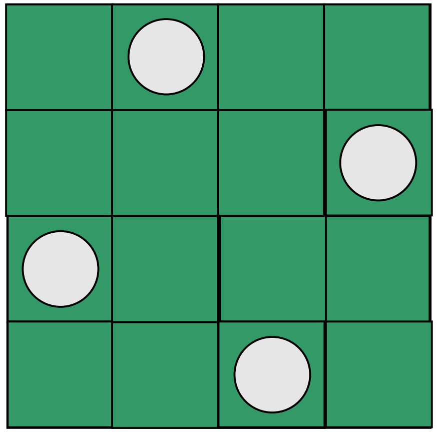
One-By-One

- This didn't work.



One-By-One

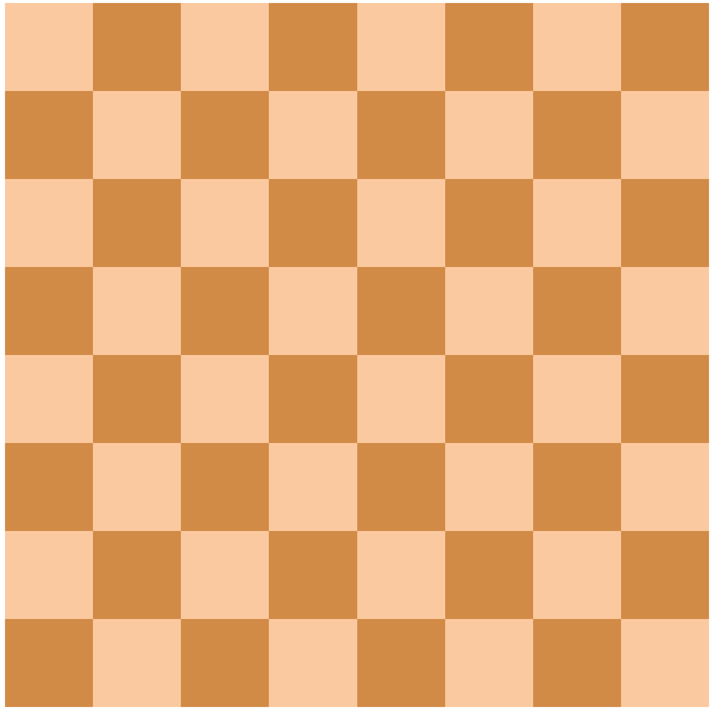
- So try this.



- Success!

8x8

- Here is an animation of the backtracking strategy being used to solve the 8 Queens problem.



Efficiency

- Let us compare the different strategies:

Strategy	Number of Moves
Brute Force 1	6031
Brute Force 2	115
Brute Force 3	11
Backtracking	8 (partial)

- As *n* is increased, the value of backtracking becomes even greater.

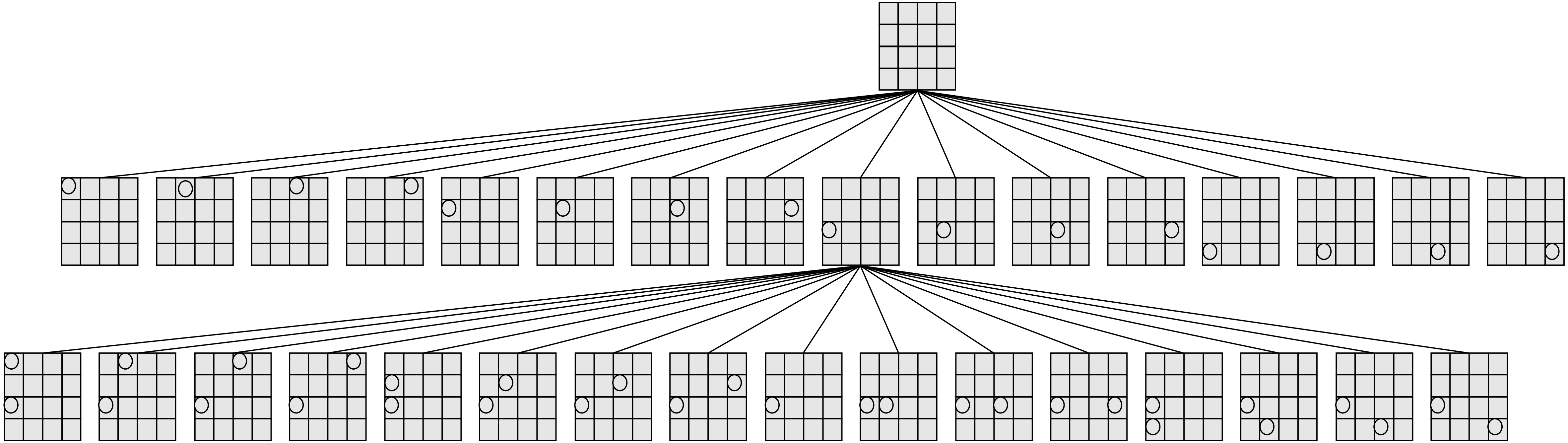
Game Trees

- We can understand what is happening by analysing the underlying data structure that describes the problem.
- The ***game-tree***.
- This is a tree constructed by taking each move in turn and linking it to each possible continuation.

Game Trees

- In the first brute force algorithm, each node has 16 children.
- As the tree has 4 levels, there are 65,536 leaves in the tree, each corresponding to one possible arrangement of queens.
- The algorithm is to look at each leaf in turn until we find the solution.
- With n queens this gives n^{2n} leaves.

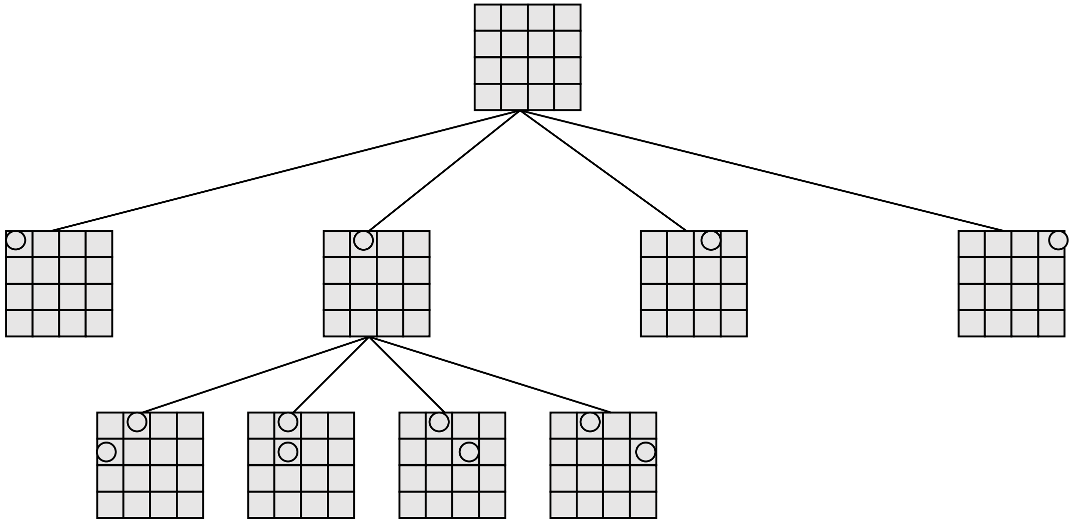
Game Trees



Game Trees

- In the second brute force algorithm, each node has 4 children.
- As the tree has 4 levels, there are 256 leaves in the tree, each corresponding to one possible arrangement of queens.
- This reduction in the size of the tree is what makes this much faster to solve.
- With n queens this gives n^n leaves.

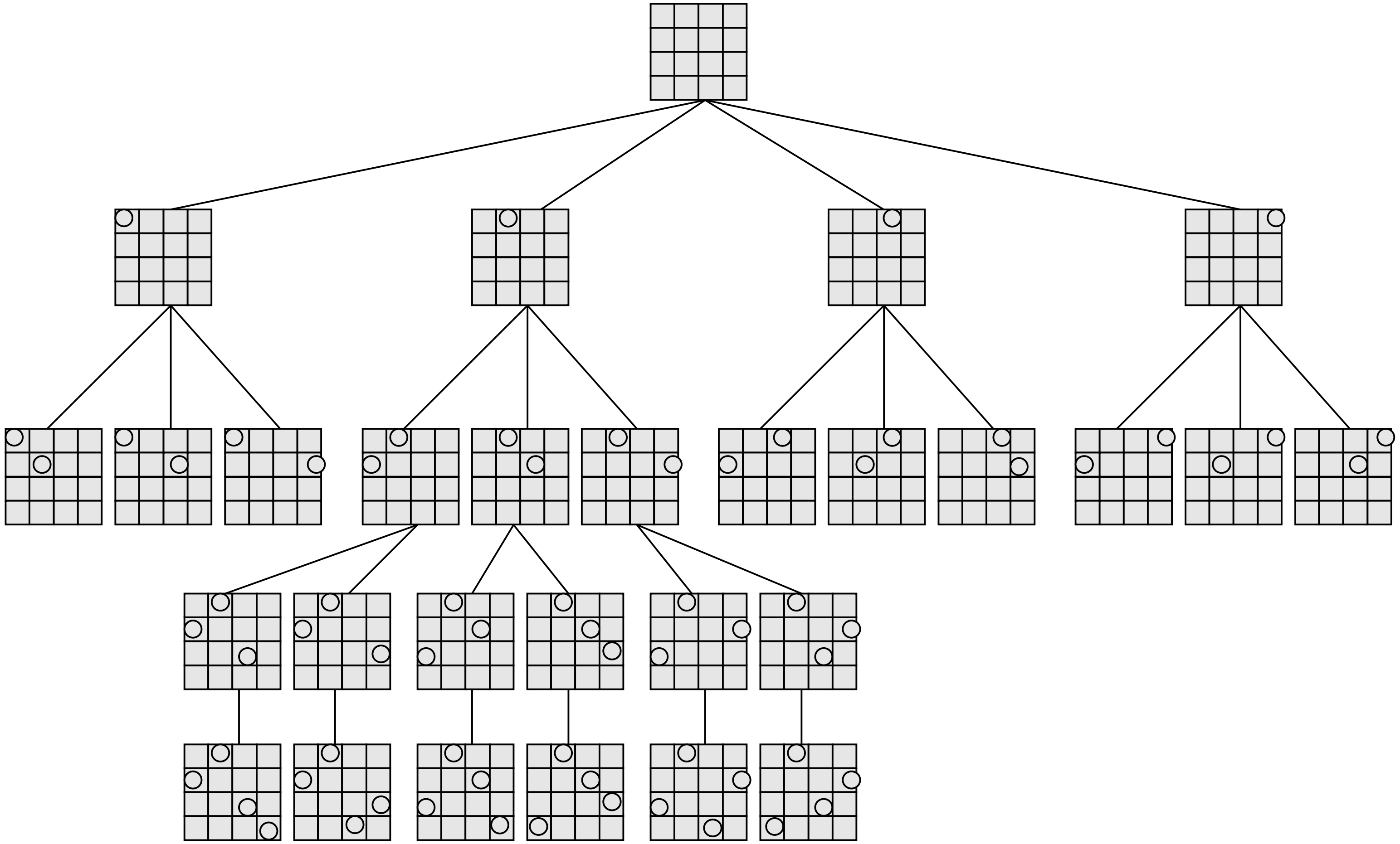
Game Trees



Game Trees

- In the third brute force algorithm, the root node has 4 children.
- Each of these has 3 children.
- Each of these has 2.
- And each of these has 1.
- There are a total of 24 leaves.
- We examine each of these in turn.
- With *n* queens this gives *n*! leaves.

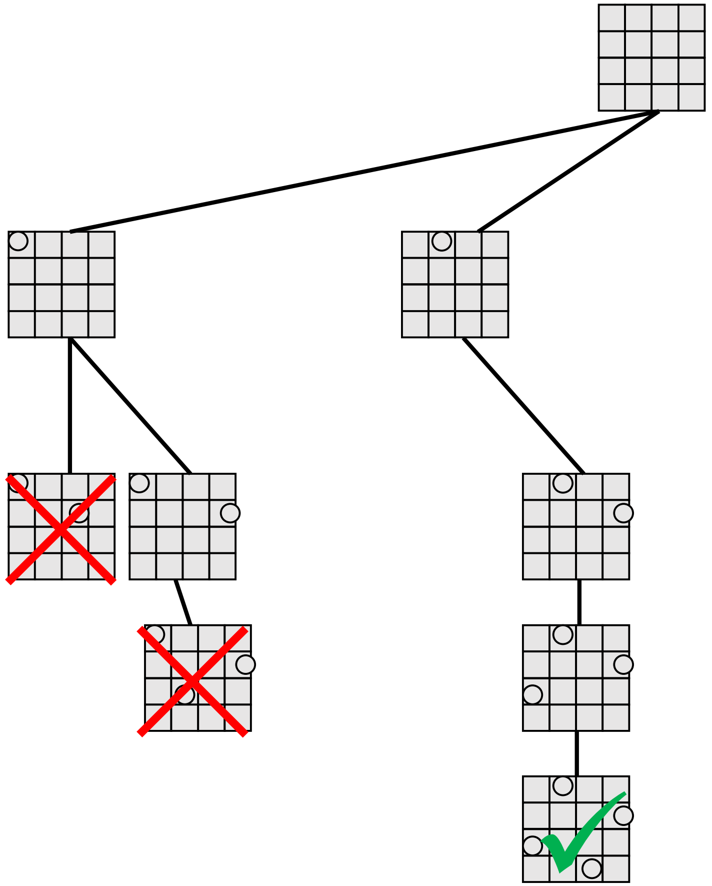
Game Trees



Game Trees

- The backtracking algorithm takes a different approach.
- Instead of looking only at the leaves, look at the internal nodes as we construct them.
- Stop building the sub-tree as soon as you get an illegal position.
- This dramatically reduces the work we have to do.

Game Trees



Game Trees

- As *n* gets bigger, the advantage becomes greater.

n	n^{2n}	n^n	$n!$	Backtrack
4	65,536	256	24	8
5	9,765,625	3,125	120	5
6	2,176,782,336	46,656	720	34
7	678,223,072,849	823,543	5,040	10
8	281,474,976,710,656	16,777,216	40,320	?

Game Trees

- This approach of constructing a tree of solutions and pruning it as it is built can be applied to a wide range of problems.
- Even so, we eventually reach a point where this approach is still too time consuming.
    - Consider solving the 1,000 queens problem.
- At this point we need to introduce ways of pruning the game tree even more effectively.

Metrics

- In the *n*-queens problem we explored the state tree in order.
  - We considered possible positions, left to right, in each row.
- What if we could attach a value to each node that provided some estimate of how likely its branch was to contain the solution?
- This would provide a better order in which to select branches to explore.
  - Select the branch with highest value.
- We call such a measure a *metric*.
- Adding a metric can dramatically improve backtracking.

Metrics and Greedy Strategies

- Every greedy algorithm is based on a metric.
- For greedy to work we need the metric to be perfect.
  - It must guarantee that the solution is down the chosen path.
  - The likelihood is either zero or one.
- In other cases we have a metric that is not perfect.
  - It does not guarantee that the solution is down the chosen path.
  - The likelihood is between zero and one.
- In such cases we cannot use a greedy strategy with guaranteed success.
- We can, however, use backtracking.

Heuristics

- When a metric is not perfect we call it a *heuristic*.
- Heuristics can improve the performance of a number of solution strategies.
- A good heuristic can dramatically improve the performance of these strategies.
- Good:
    - Close match to reality (the metric leads quickly to the solution)
    - Easy/cheap to evaluate
- Sometimes backtracking, even with a good heuristic, is not enough.
- We need even better strategies.