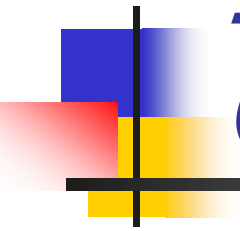
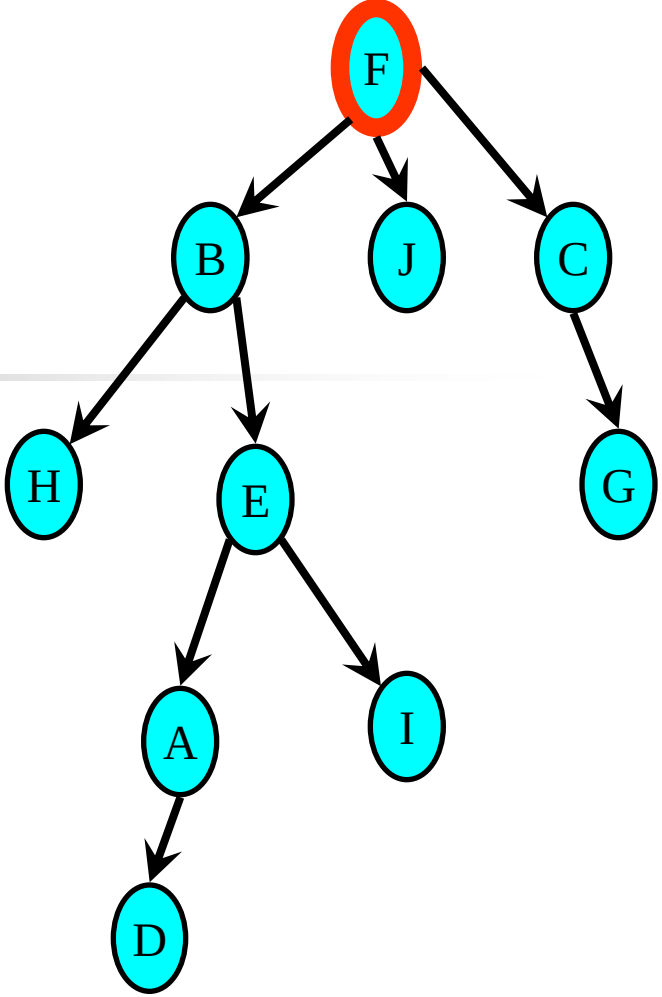


Introduction to Artificial Intelligence (G51IAI)

Dr Rong Qu

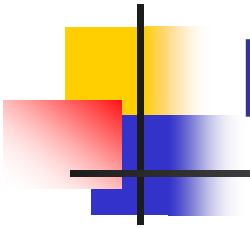
Problem Space and Search Tree

Trees



- Nodes
  - Root node
  - Children/parent of nodes
  - Leaves
- Branches

- Average branching factor
  - average number of branches of the nodes in the tree



Problem Space

- Many problems exhibit no detectable regular structure to be exploited, they appear “chaotic”, and do not yield to efficient algorithms

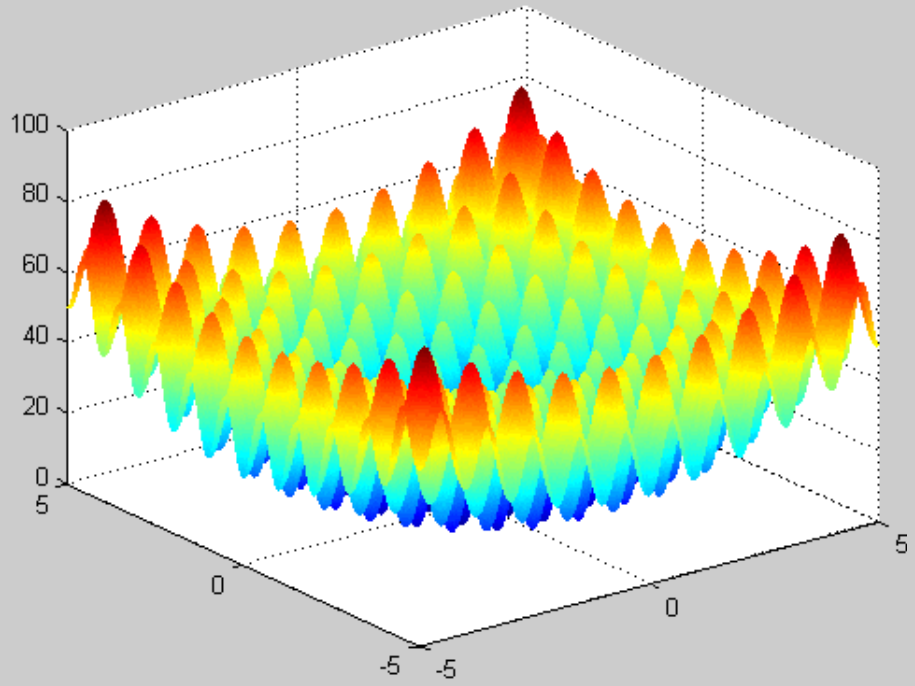
Problem Space

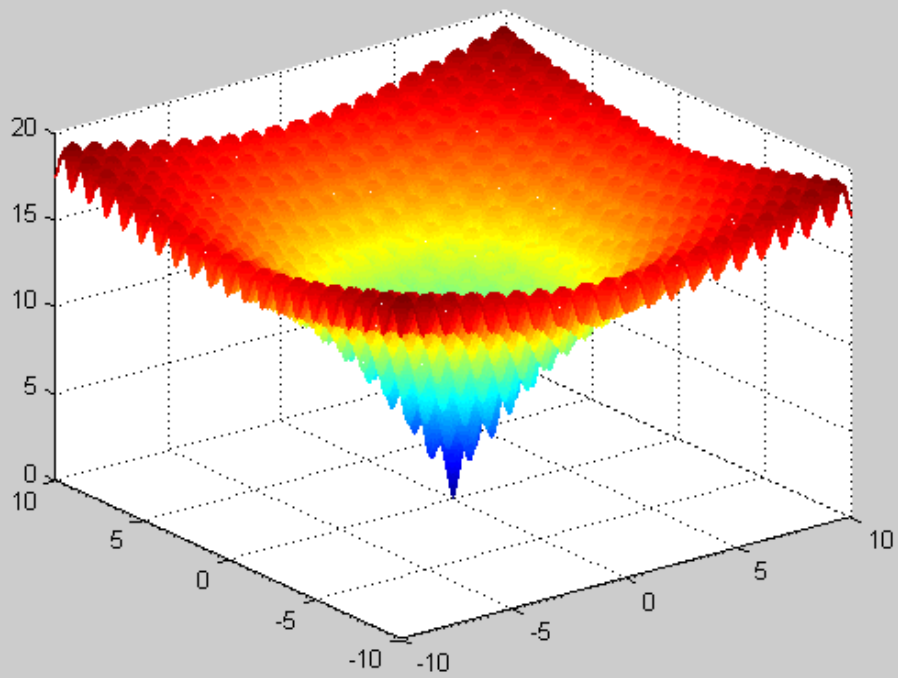
$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

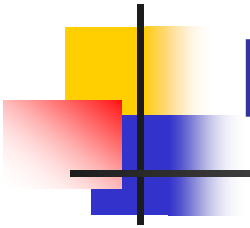
$$x_i \in [-5.12, 5.12]$$

$$F(\vec{x}) = -20 \cdot \exp\left(-0.2\sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos(2\pi \cdot x_i)\right) + 20 + e$$

$$x_i \in [-30, 30]$$





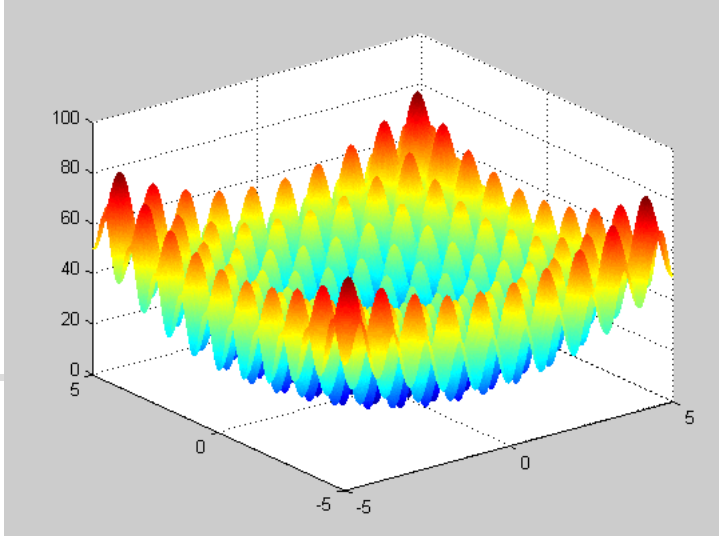


Problem Space

The concept of search plays an important role in science and engineering

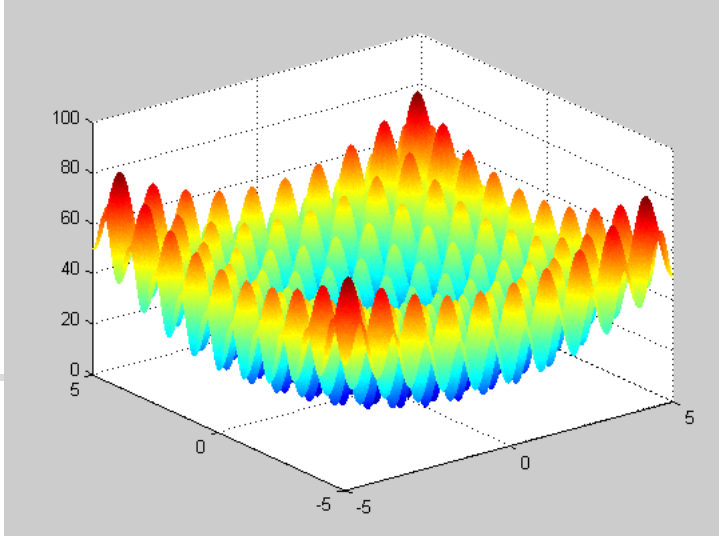
In one way, any problem whatsoever can be seen as a search for "the right answer"

Problem Space

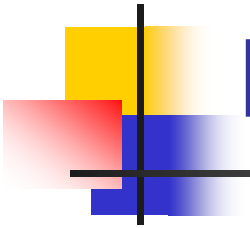


- Search space
  - Set of all possible solutions to a problem
- Search algorithms
  - Take a problem as input
  - Return a solution to the problem

Problem Space



- Search algorithms
  - Uninformed search algorithms (3 hours)
    - Simplest naïve search
  - Informed search algorithms (2 hours)
    - Use of heuristics that apply domain knowledge
    - Dr Hyde



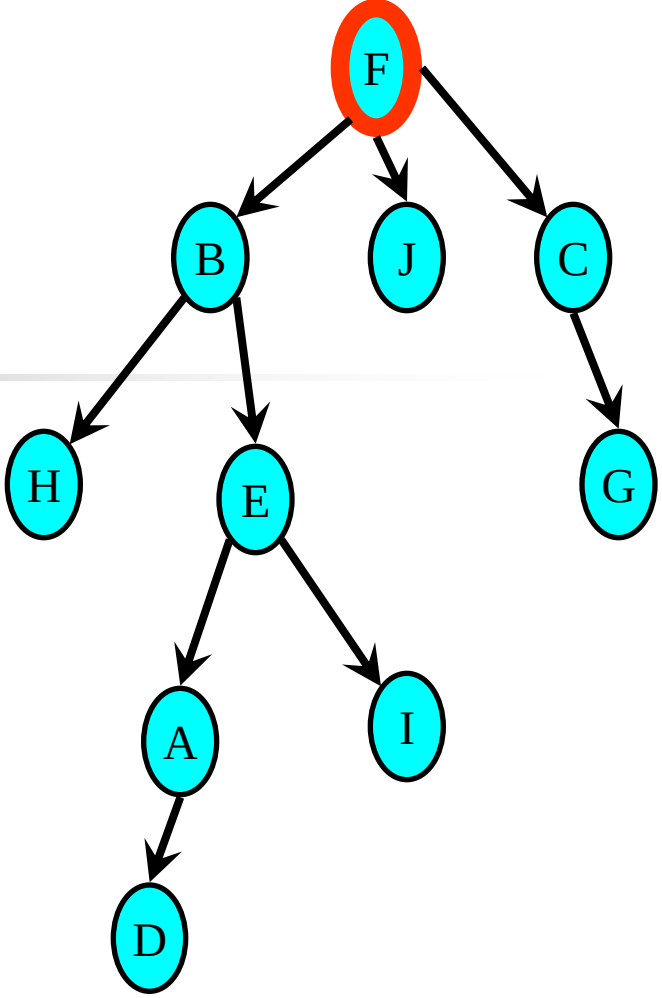
Problem Space

- Often we can't simply write down and solve the equations for a problem
- Exhaustive search of large state spaces appears to be the only viable approach

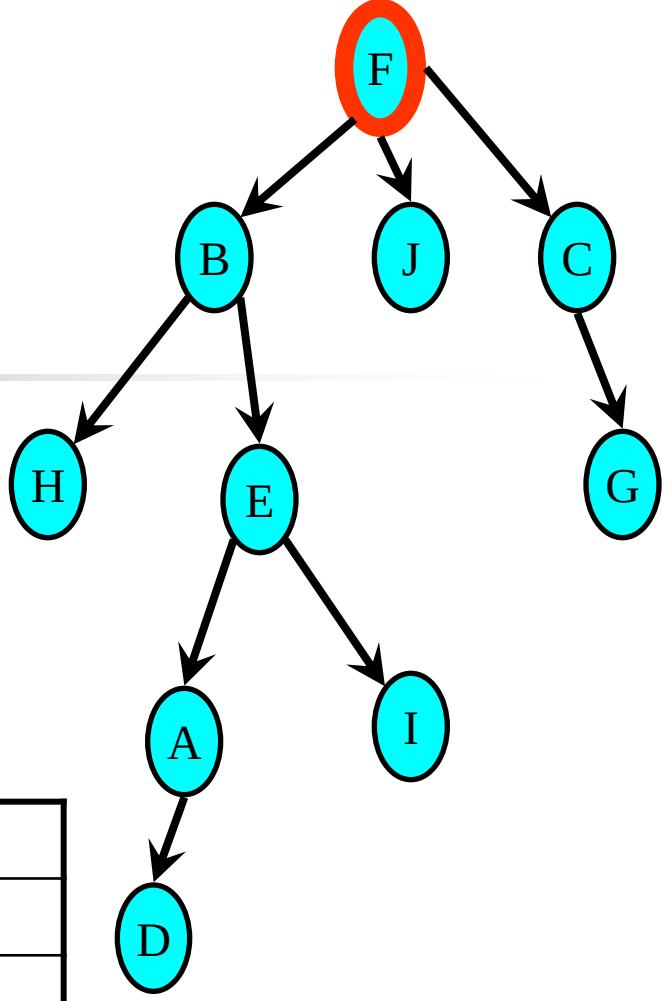
How?

Trees

- **Depth** of a node
  - Number of branches away from the root node
- Depth of a tree
  - Depth of the deepest node in the tree
  - Examples: TSP vs. game



Trees

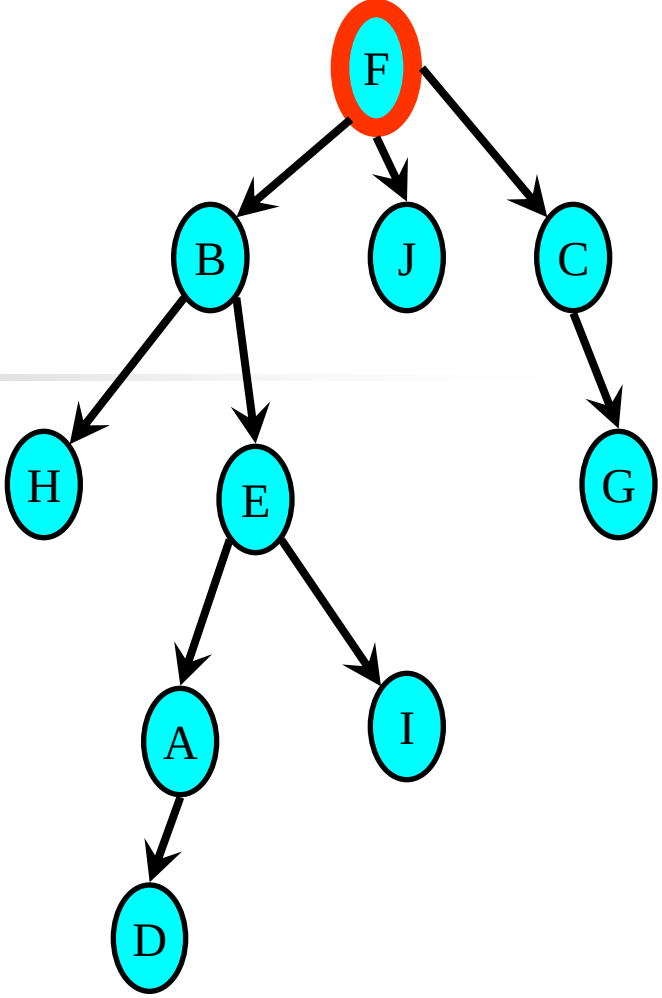


- Tree size
  - **Branching factor** $b = 2$ (binary tree)
  - Depth d

d	nodes at d, 2^d	total nodes
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
5	32	63
6	64	127

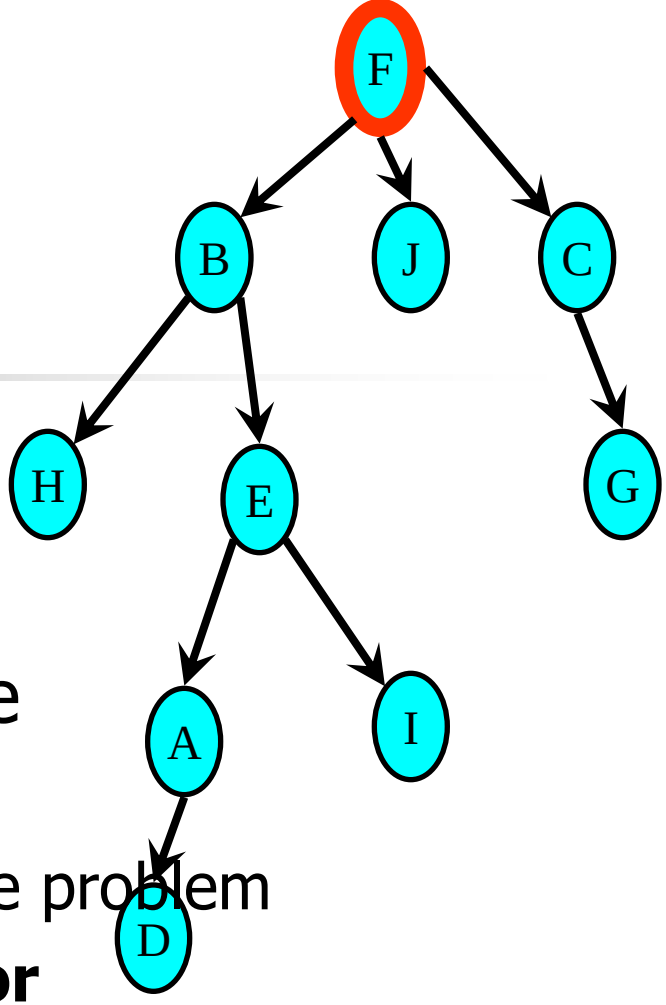
Exponentially - Combinatorial explosion

Trees



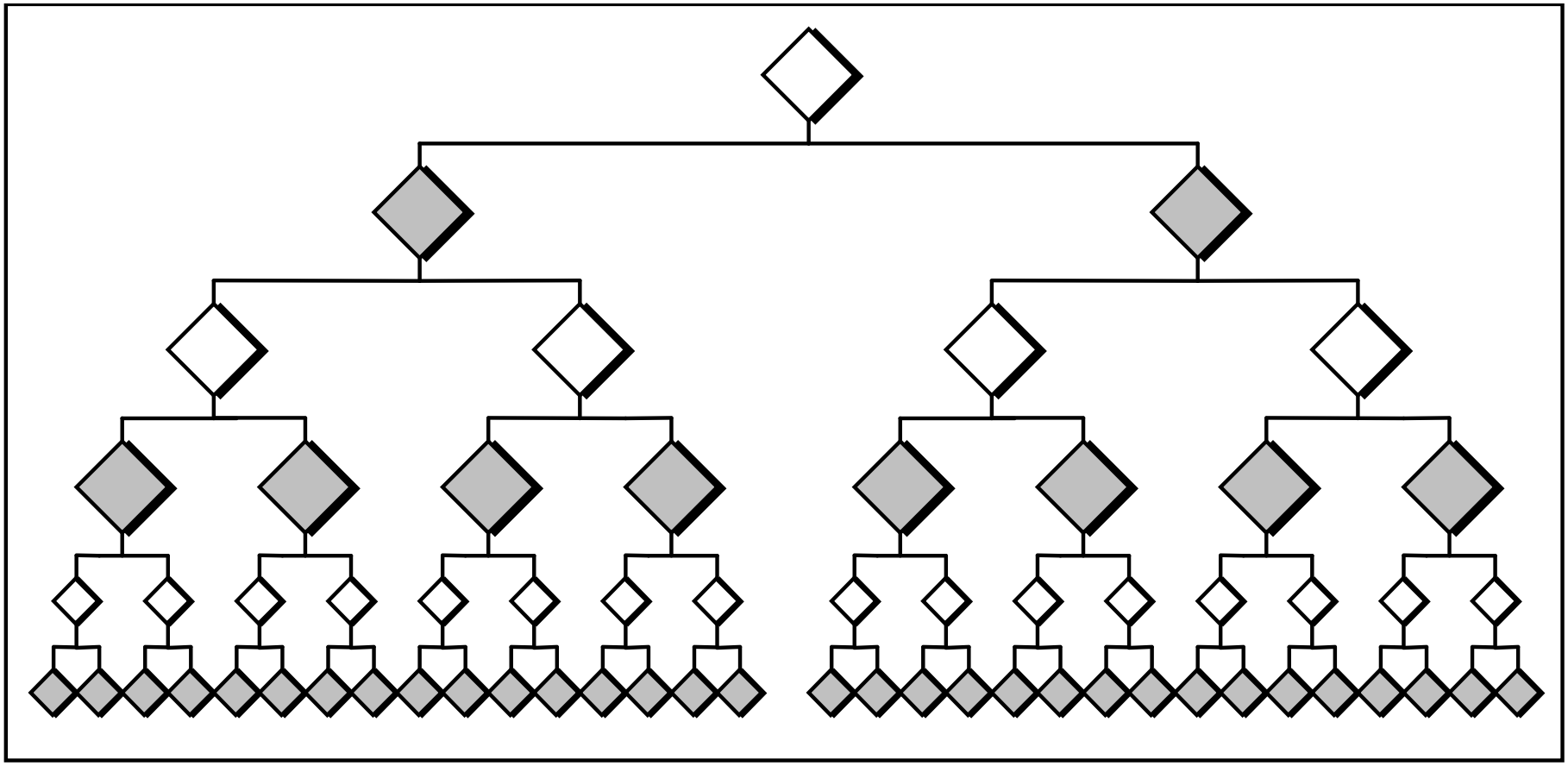
Exponentially -
Combinatorial explosion

Search Tree

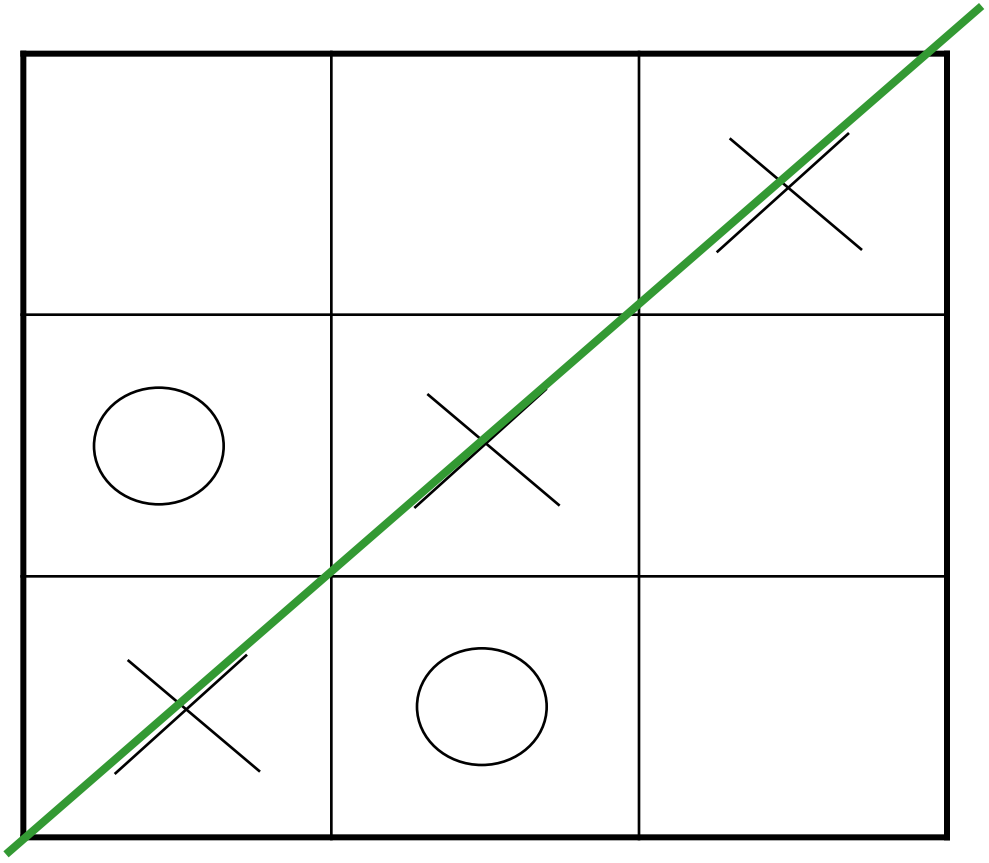


- Heart of search techniques
- Managing the data structure
  - Nodes: states of problem
  - Root node: initial **state** of the problem
  - Branches: moves by **operator**
  - Branching factor: number of **neighborhoods**

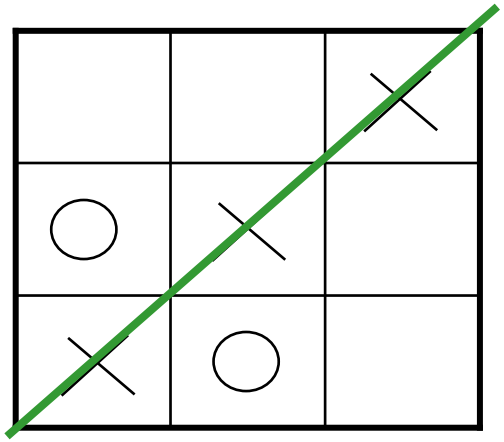
Search Tree – Define a Problem Space

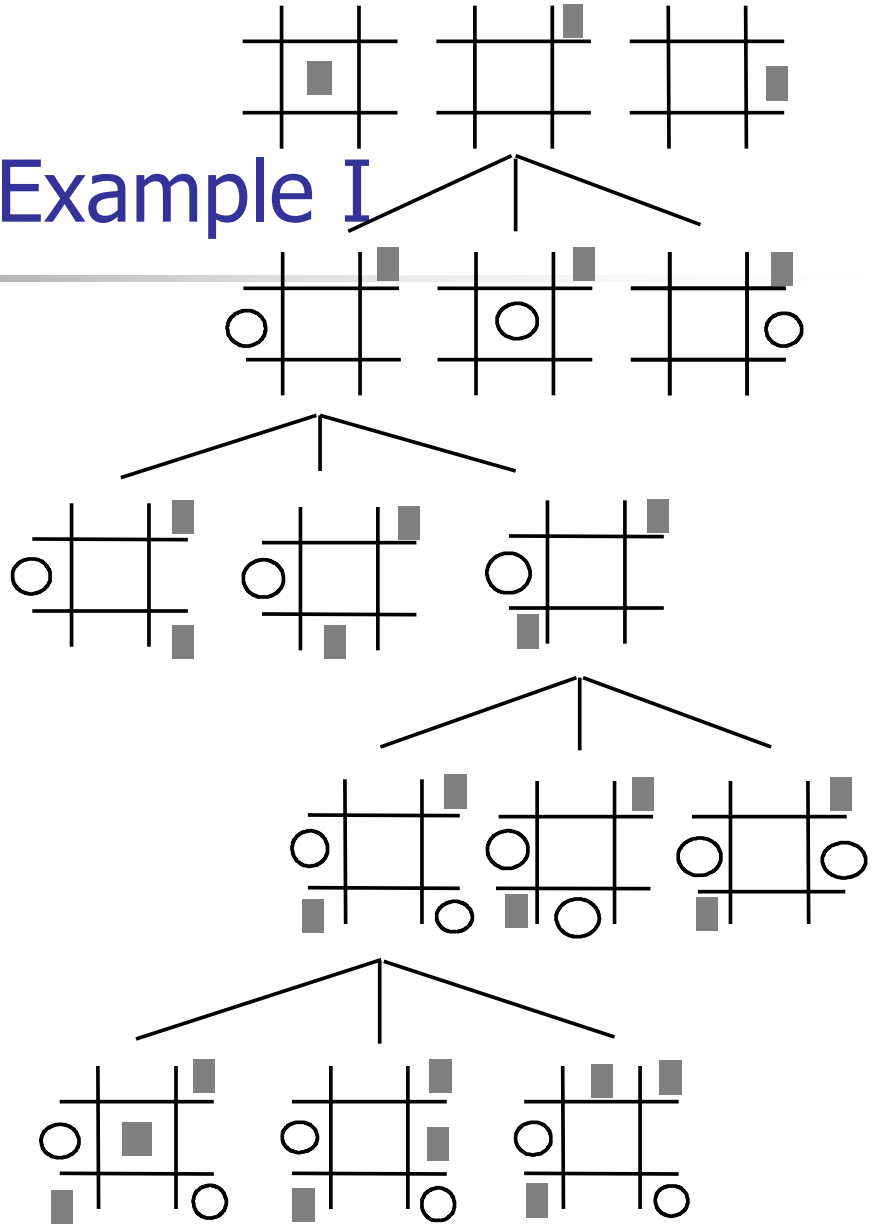


Search Tree – Example I



Search Tree – Example I

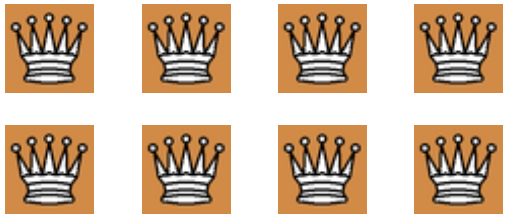


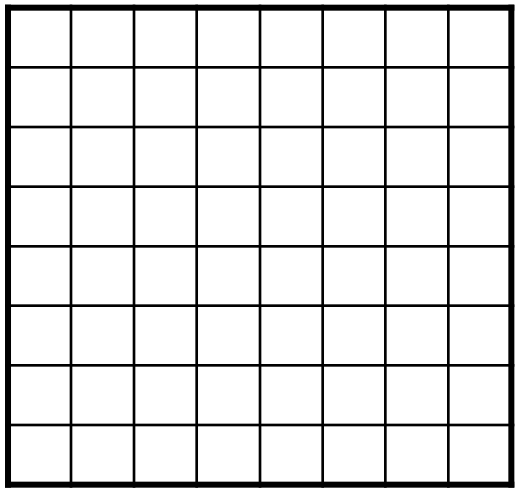


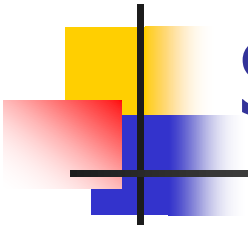
Compared with TSP tree?

Search Tree – Example II

- 1st level: 1 root node (empty board)
- 2nd level: 8 nodes
- 3rd level: 6 nodes for each of the node on the 2nd level (?)
- …

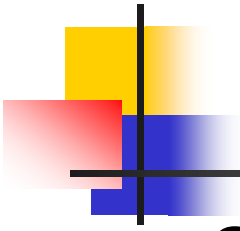






Search Trees

- Issues
  - Search trees grow very quickly
  - The size of the search tree is governed by the **branching factor**
  - Even the simple game with branching factor of 3 has a complete search tree of large number of potential nodes
  - The search tree for chess has a branching factor of about 35



Search Trees

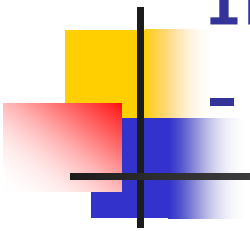
Claude Shannon delivered a paper in 1949 at a New York conference on how a computer could play chess.

Chess has 10^{120} unique games (with an average of 40 moves - the average length of a master game).

Working at 200 million positions per second, Deep Blue would require 10^{100} years to evaluate all possible games.

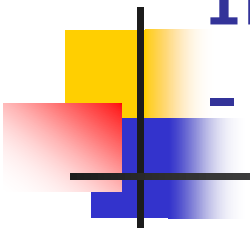
To put this is some sort of perspective, the universe is only about 10^{10} years old and 10^{120} is larger than the number of atoms in the universe.

Implementing a Search - What we need to store

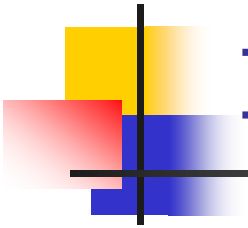


- State
  - This represents the state in the state space to which this node corresponds
- Parent-Node
  - This points to the node that generated this node. In a data structure representing a tree it is usual to call this the parent node

Implementing a Search - What we need to store

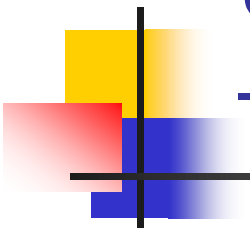


- Operator
  - The operator that was applied to generate this node
- Depth
  - The number of branches from the root
- Path-Cost
  - The path cost from the initial state to this node



Implementing a Search - Datatype

- Datatype node
  - Components:
    - STATE,
    - PARENT-NODE,
    - OPERATOR,
    - DEPTH,
    - PATH-COST

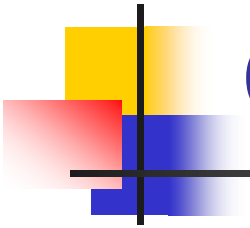


Using a Tree – The Obvious Solution?

- It can be wasteful on space
- It can be difficult to implement, particularly if there are varying number of children (as in tic-tac-toe)
- It is not always obvious which node to expand next. We may have to search the tree looking for the best leaf node (sometimes called the fringe or frontier nodes). This can obviously be computationally expensive

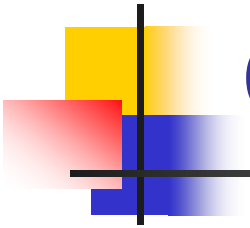
Using a Tree – Maybe not so obvious

- Therefore
  - It would be nice to have a "simpler" data structure to represent our tree
  - And it would be nice if the next node to be expanded was an O(1)* operation
  - *Big O: Notation in complexity theory
    - How the size of input affect the algorithms computational resource (time or memory)
    - Complexity of algorithms



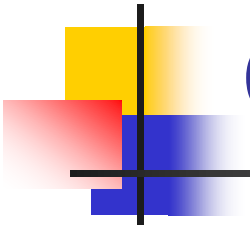
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



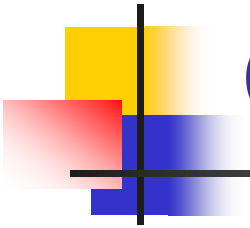
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



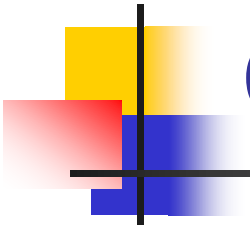
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



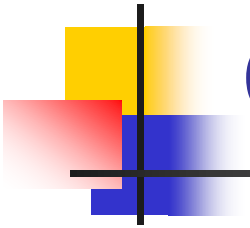
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



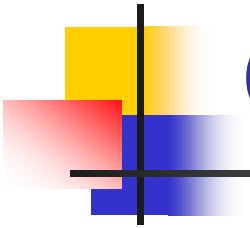
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



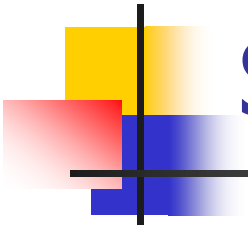
General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



General Search

- Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
  - nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  - Loop do
    - If nodes is empty then return failure
    - node = REMOVE-FRONT(nodes)
    - If GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    - nodes = QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
  - End
- End Function



Summary of Problem Space

- Search space
- Search tree (problem formulation)
- General search algorithm

- Read Chapter 3 AIMA