Problem Solving, Search & Control Strategies: AI Course Lecture 7 – 14 , notes, slides
www.myreaders.info/ , RC Chakraborty, e-mail rcchak@gmail.com , June 01, 2010
www.myreaders.info/html/artificial_intelligence.html

Problem Solving, Search and Control Strategies

Artificial Intelligence

General Problem Solving in AI , topics : Definitions, problem space, problem solving, state space, state change, problem solution, problem description. Search and Control Strategies - search related terms, algorithm's performance and complexity, "Big - o" notations, tree structure, stacks and queues; Search - search algorithms, hierarchical representation, search space, formal statement, search notations, estimate cost function and heuristic function; Control strategies - strategies for search, forward and backward chaining. Exhaustive search - depth-first search algorithm, breadth-first search algorithm, compare depth-first and breadth-first search. Heuristic search techniques - characteristics of heuristic search, heuristic search compared with other search, types of heuristic search algorithms. Constraint satisfaction problems (CSPs) and models - generate and test, backtracking algorithm.

Problem Solving, Search and Control Strategies

Artificial Intelligence

Topics

(Lectures 07, 08, 09, 10, 11, 12, 13, 14, 8 hours)

	Slides
1. General Problem Solving	03-17
Problem solving definitions : problem space, problem solving, state space, state change, structure of state space, problem solution, problem description; Examples of problem definition.	
2. Search and Control Strategies	18-43
Search related terms : algorithm's performance and complexity, computational complexity, "Big - o" notations, tree structure, stacks and queues; Search: search algorithms, hierarchical representation, search space, formal statement, search notations, estimate cost and heuristic function; Control strategies : strategies for search, forward and backward chaining.	
3. Exhaustive Searches	44-52
Depth-first search Algorithm; Breadth-first search Algorithm; Compare depth-first and breadth-first search;	
4. Heuristic Search Techniques	53-59
Characteristics of heuristic search; Heuristic search compared with other search; Example of heuristic search; Types of heuristic search algorithms.	
5. Constraint Satisfaction Problems (CSPs) and Models	60-74
Examples of CSPs; Constraint Satisfaction Models : Generate and Test, Backtracking algorithm, Constrain Satisfaction Problems (CSPs) : definition, properties and algorithms.	
6. References	75

Problem Solving,
Search and Control Strategies

What are problem solving, search and control strategies ?

- **Problem solving is fundamental to many AI-based applications.**

  There are two types of problems.

  – The Problems like, computation of the sine of an angle or the square root of a value. These can be solved through the use of deterministic procedure and the success is guaranteed.

  – In the real world, very few problems lend themselves to straightforward solutions.

  Most real world problems can be solved only by searching for a solution.

  AI is concerned with these type of problems solving.

- **Problem solving is a process** of generating solutions from observed data.
  – a problem is characterized by a set of goals,
  – a set of objects, and
  – a set of operations.

  These could be ill-defined and may evolve during problem solving.

- **Problem space** is an abstract space.
  – A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
  – The problem space may contain one or more *solutions*.

  Solution is a combination of operations and objects that achieve the goals.

- **Search** refers to the search for a solution in a problem space.
  – Search proceeds with different types of *search control strategies*.
  – The *depth-first search and breadth-first search* are the two common search strategies.

1. General Problem solving

Problem solving has been the key areas of concern for Artificial Intelligence.

- **Problem solving is a process of generating solutions** from observed or given data. It is however not always possible to use direct methods (i.e. go directly from data to solution). Instead, problem solving often need to use indirect or model-based methods.

- **General Problem Solver (GPS) was a computer program** created in 1957 by *Simon* and *Newell* to build a universal problem solver machine. GPS was based on Simon and Newell's theoretical work on logic machines. GPS in principle can solve any formalized symbolic problem, like : theorems proof and geometric problems and chess playing.

- GPS solved many simple problems such as the Towers of Hanoi, that could be sufficiently formalized, but **GPS could not solve any real-world problems**.

To build a system to solve a particular problem, we need to

- Define the problem precisely – find input situations as well as final situations for acceptable solution to the problem.

- Analyze the problem – find few important features that may have impact on the appropriateness of various possible techniques for solving the problem.

- Isolate and represent task knowledge necessary to solve the problem

- Choose the best problem solving technique(s) and apply to the particular problem.

1.1 Problem Definitions :

A *problem* is defined by its *elements* and their *relations*.

To provide a formal description of a problem, we need to do following:

a. Define a *state space* that contains all the possible configurations of the relevant objects, including some impossible ones.
b. Specify one or more states, that describe possible situations, from which the problem-solving process may start. These states are called *initial states*.
c. Specify one or more states that would be acceptable solution to the problem. These states are called *goal states*.
d. Specify a set of *rules* that describe the *actions* (*operators*) available.

The problem can then be solved by using the *rules*, in combination with an appropriate *control strategy*, to move through the *problem space* until a *path* from an *initial state* to a *goal state* is found.

This process is known as ***search***.

- Search is fundamental to the problem-solving process.
- Search is a general mechanism that can be used when more direct method is not known.
- Search provides the framework into which more direct methods for solving subparts of a problem can be embedded.

A very large number of AI problems are formulated as search problems.

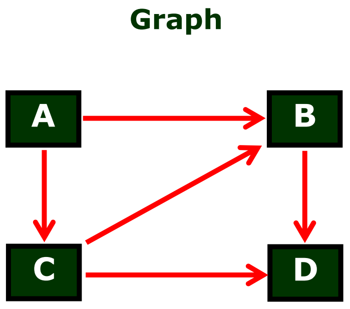
- **Problem Space**

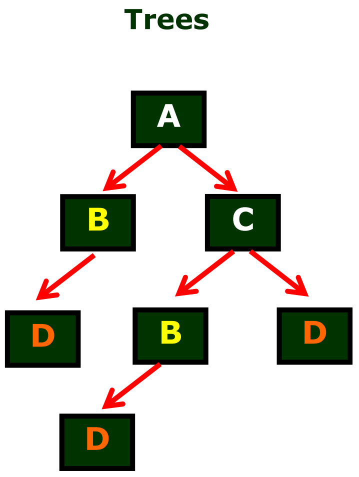
A *problem space* is represented by directed *graph*, where *nodes* represent *search state* and *paths* represent the *operators* applied to change the *state*.

To simplify a search algorithms, it is often convenient to logically and programmatically represent a problem space as a ***tree***. A tree usually decreases the complexity of a search at a ***cost***. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph; e.g., node **B** and node **D** shown in example below.

A **tree** is a graph in which any two vertices are connected by exactly one path. Alternatively, any connected graph with no cycles is a tree.

Examples





• Problem Solving

The term Problem Solving relates analysis in AI. Problem solving may be characterized as a *systematic search* through a range of possible actions to reach some predefined goal or solution. Problem-solving methods are categorized as *special purpose* and *general purpose.*

– **Special-purpose method** is tailor-made for a particular problem, often exploits very specific features of the situation in which the problem is embedded.

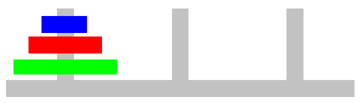
– **General-purpose method** is applicable to a wide variety of problems. One general-purpose technique used in AI is "means-end analysis". It is a step-by-step, or incremental, reduction of the difference between current state and final goal.

Examples : Tower of Hanoi puzzle

– For a Robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT—until the goal is reached.

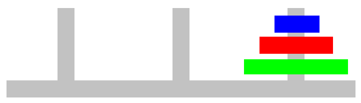
– Puzzles and Games have explicit rules : e.g., the Tower of Hanoi puzzle.

(a)



Start

(b)



Final

Tower of Hanoi puzzle.

- This puzzle may involves a set of rings of different sizes that can be placed on three different pegs.
- The puzzle starts with the rings arranged as shown in Fig. (a)
- The goal of this puzzle is to move them all as to Fig. (b)
- Condition : Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg.

In this Tower of Hanoi puzzle : Situations encountered while solving the problem are described as *states*. The set of all possible configurations of rings on the pegs is called *problem space*.

- **States**

  A *state* is a representation of elements at a given moment. A problem is defined by its *elements* and their *relations*.

  At each instant of a problem, the elements have specific descriptors and relations; the *descriptors* tell - how to select elements ?

  Among all possible states, there are two special states called :

  – *Initial state* is the start point

  – *Final state* is the goal state

- **State Change:** Successor Function

A *Successor Function* is needed for state change.

The successor function moves one state to another state.

Successor Function :

- Is a description of possible actions; a set of operators.
- Is a transformation function on a state representation, which converts that state into another state.
- Defines a relation of accessibility among states.
- Represents the conditions of applicability of a state and corresponding transformation function

- **State Space**

A *State space* is the set of all states reachable from the *initial state*.

Definitions of terms :

- A *state space* forms a *graph* (or map) in which the *nodes* are states and the *arcs* between nodes are actions.
- In *state space*, a *path* is a sequence of states connected by a sequence of actions.
- The *solution* of a problem is part of the map formed by the *state space*.

- **Structure of a State Space**

The *Structures* of *state space* are *trees* and *graphs*.

– Tree is a hierarchical structure in a graphical form; and
– Graph is a non-hierarchical structure.

◇ **Tree** has only one path to a given node;
i.e., a *tree* has one and only one path from any point to any other point.

◇ **Graph** consists of a set of nodes (vertices) and a set of edges (arcs). Arcs establish relationships (connections) between the nodes;
i.e., a graph has several paths to a given node.

◇ **operators** are directed *arcs* between nodes.

Search process explores the *state space*. In the worst case, the search explores all possible *paths* between the *initial state* and the *goal state*.

- **Problem Solution**

  In the *state space*, a *solution is a path* from the *initial state* to a *goal state* or sometime just a *goal state*.

  - A Solution cost function assigns a numeric cost to each path; It also gives the cost of applying the operators to the states.
  - A Solution quality is measured by the path cost function; and An optimal solution has the lowest path cost among all solutions.
  - The solution may be any or optimal or all.
  - The importance of cost depends on the problem and the type of solution asked.

- **Problem Description**

A problem consists of the description of :

- *current state* of the world,
- *actions* that can transform one state of the world into another,
- *desired state* of the world.

◇ **State space** is defined explicitly or implicitly

A state space should describe everything that is needed to solve a problem and nothing that is not needed to the solve the problem.

◇ **Initial state** is start state

◇ **Goal state** is the conditions it has to fulfill

- A description of a desired state of the world;
- The description may be complete or partial.

◇ **Operators** are to change state

- Operators do actions that can transform one state into another.
- Operators consist of : Preconditions and Instructions;
  - Preconditions provide partial description of the state of the world that must be true in order to perform the action,
  - Instructions tell on how to create next state.
- Operators should be as general as possible, to reduce their number.

◇ **Elements of the domain** has relevance to the problem

- Knowledge of the starting point.

◇ **Problem solving** is finding solution

- Finding an ordered sequence of operators that transform the current (start) state into a goal state;

◇ **Restrictions** are solution quality any, optimal, or all

- Finding the shortest sequence, or
- Finding the least expensive sequence defining cost , or
- Finding any sequence as quickly as possible.

Recall : Algebraic Function

Algebraic Function

A function may take the form of a set of *ordered pair*, a *graph*, or a *equation*. Regardless of the form it takes, a function must obey the condition that, no two of its ordered pair have the same first member with different second members.

Relation : A set of ordered pair of the form **(x, y)** is called a relation.

Function : A relation in which no two ordered pair have the same *x-value* but may have same or *different y-value* is called a function. Functions are usually named by lower-case letters such as **f, g**, and **h.**

For example, **f = {(-3, 9), (0, 0), (3, 9) }** , **g = {(4, -2), (4, 2)}**

Here **f** is a function , **g** is not a function.

Domain and Range : The *domain* of a function is the set of all the first members of its ordered pairs, and the *range* of a function is the set of all second members of its ordered pairs.

if function **f = {(a, A), (b, B), (c, C)},**

then its domain is **{a, b, c }** and its range is **{A, B, C}.**

Function and mapping : A function may be viewed as a mapping or a pairing of one set with *elements* of a second set such that each element of the first set (called *domain*) is paired with exactly one *element* of the second set (called *codomain*) . Example :

If a function **f** maps **{a, b, c }** into **{A, B, C, D}**

such that **a** $\rightarrow$ **A** (read " **a** is mapped into **A**"), **b** $\rightarrow$ **B** , **c** $\rightarrow$ **C**

Then the *domain* is **{a, b, c}** and the *codomain* is **{A, B, C, D}**.

Each element of the co-domain that corresponds to an element of the domain is called the *image* of that element. Since **a** is paired with **A** in codomain, **A** is called the *image* of **a**.

The set of image points, **{A, B, C}**, is called the range. Thus, the range is a subset of the *codomain*.

Onto mappings : Set **A** is mapped onto set **B** if each element of set **B** is image of an element of a set **A**. Thus, every function maps its domain onto its range.

[Continued in next slide]

[Continued from previous slide]

Describing a function by an equation : The *rule* by which each x-value gets paired with the corresponding y-value may be specified by an equation. For example, the function described by the equation $y = x + 1$ requires that for any choice of x in the domain, the corresponding range value is $x + 1$. Thus, $2 \to 3$, $3 \to 4$, and $4 \to 5$.

Restricting domains of functions : Unless otherwise indicated, the domain of a function is assumed to be the largest possible set of real numbers. Thus, the domain of $y = x / (x^2 - 4)$ is the set of all real numbers except ± 2 since for these values of x the denominator is 0.

The domain of $y = (x - 1)^{1/2}$ is the set of real numbers greater than or equal to 1 since for any value of x less than 1, the root radical has a negative radicand so the radical does not represent a real number.

Example : Find which of the relation describe function ?

(a) $y = x^{1/2}$, (b) $y = x^3$, (c) $y < x$, (d) $x = y^2$

The equations (a) and (b) produce exactly one value of y for each value of x. Hence, equations (a) and (b) describe functions.

The equation (c), $y < x$ does not represent a function since it contains ordered pair such as $(1, 2)$ and $(1, 3)$ where same value of x is paired with different values of y.

The equation (d), $x = y^2$ is not a function since ordered pair such as $(4, 2)$ and $(4, -2)$ satisfy the equation but have the same value of x paired with different values of y.

Function notation : For any function f, the value of y that corresponds to a given value of x is denoted by $f(x)$.

If $y = 5x - 1$, then $f(2)$, read as **"f of 2"** , represents the value of y.

when $x = 2$, then $f(2) = 5 . 2 - 1 = 9$;

when $x = 3$, then $f(3) = 5 . 3 - 1 = 14$;

In an equation that describes function f, then $f(x)$ may be used in place of y, for example , $f(x) = 5x - 1$.

If $y = f(x)$, then y is said to be a function of x.

Since the value of y depends on the value of x, then y is called the dependent variable and x is called the independent variable.

1.2 Examples of Problem Definitions

- **Example 1 :**

A game of 8–Puzzle

◇ State space : configuration of **8 - tiles** on the board

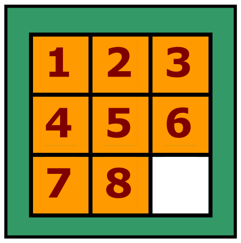
◇ Initial state : any configuration

◇ Goal state : tiles in a specific order

◇ Action : "blank moves"

  - Condition: the move is within the board
  - Transformation: blank moves Left, Right, Up, Dn

◇ Solution : optimal sequence of operators

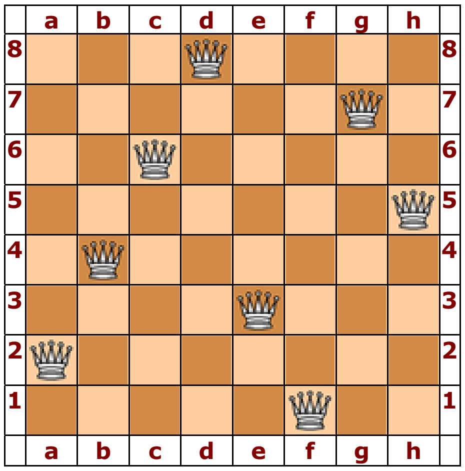


Solution

- **Example 2 :**

A game of n - queens puzzle; n = 8

- State space : configurations **n = 8** queens on the board with only one queen per row and column
- Initial state : configuration without queens on the board
- Goal state : configuration with **n = 8** queens such that no queen attacks any other
- Operators or actions : place a queen on the board.
  - Condition: the new queen is not attacked by any other already placed
  - Transformation: place a new queen in a particular square of the board
- Solution : one solution (cost is not considered)



One Solution

2. Search and Control Strategies

Word "Search" refers to the search for a solution in a problem space.

– Search proceeds with different types of **"Search Control strategies"**.

– A strategy is defined by picking the order in which the nodes expand.

The Search strategies are evaluated in the following dimensions:

Completeness, *Time complexity*, *Space complexity*, *Optimality*.

(the search related terms are first explained, then the search algorithms and control strategies are illustrated).

2.1 Search related terms

- **Algorithm's Performance and Complexity**

  Ideally we want a common measure so that we can compare approaches in order to select the most appropriate algorithm for a given situation.

  ◇ **Performance** of an algorithm depends on internal and external factors.

Internal factors	External factors
‡ **Time** required, to run	‡ **Size** of input to the algorithm
‡ **Space** (memory) required to run	‡ **Speed** of the computer
	‡ **Quality** of the compiler

  ◇ **Complexity** is a measure of the performance of an algorithm. It measures the internal factors, usually in time than space.

- **Computational Complexity**

A measure of resources in terms of Time and Space.

◇ If **A** is an algorithm that solves a decision problem **f** then run time of **A** is the number of steps taken on the input of length **n.**

◇ **Time Complexity** **T(n)** of a decision problem **f** is the run time of the 'best' algorithm **A** for **f** .

◇ **Space Complexity** **S(n)** of a decision problem **f** is the amount of memory used by the `best' algorithm **A** for **f** .

- **"Big - O" notation**

The "Big-O" is theoretical measure of the execution of an algorithm, usually indicates the *time* or the *memory* needed, given the problem size **n**, which is usually the number of items.

◇ **Big-O notation**

The **Big-O** notation is used to give an approximation to the run-time-efficiency of an algorithm ; the letter "**O**" is for order of magnitude of operations or space at run-time.

◇ **The Big-O of an Algorithm A**

– If an algorithm **A** requires time proportional to **f(n)**, then the algorithm **A** is said to be of order **f(n)**, and it is denoted as **O(f(n))**.

– If algorithm **A** requires time proportional to n^2, then order of the algorithm is said to be $O(n^2)$.

– If algorithm **A** requires time proportional to **n**, then order of the algorithm is said to be **O(n)**.

The function **f(n)** is called the algorithm's **growth-rate function**.

If an algorithm has performance complexity **O(n)**, this means that the run-time **t** should be directly proportional to **n**, ie $t \propto n$ or $t = k\,n$ where **k** is constant of proportionality.

Similarly, for algorithms having performance complexity $O(\log_2(n))$, **O(log N)**, **O(N log N)** , $O(2^N)$ and so on.

◊ Example 1 :

1-D array, determine the Big-O of an algorithm ;

Calculate the sum of the **n** elements in an integer array **a[0 . . . . n-1]**.

Line no	Instructions	No of execution steps
line 1	sum = 0	1
line 2	for (i = 0; i < n; i++)	n + 1
line 3	sum += a[i]	n
line 4	print sum	1
	Total	**2n + 3**

For the polynomial **(2*n + 3)** the Big-O is dominated by the 1st term as **n** while the number of elements in the array becomes very large. Also in determining the **Big-O**

- Ignoring constants such as **2** and **3**, the algorithm is of the order **n**.
- So the **Big-O** of the algorithm is **O(n)**.
- In other words the run-time of this algorithm increases roughly as the size of the input data **n** , say an array of size **n**.

◇ **Example 2 :**

2-D array, determine the Big-O of an algorithm;

In a square 2-D array **a[0 . . . . . n-1] [0 . . . . . n-1]** , find the largest element .

Line no	Instructions	No of executions
line 1	max = a[0][0]	1
line 2	for (row = 0; row < n; row++)	n + 1
line 3	for (col = 0; col < n; col++)	n * (n+1)
line 4	if (a[row][col] > max) max = a[row][col].	n * (n)
line 5	print max	1
	Total	$2n^2 + 2n + 3$

‡ Ignoring the constants such as **2, 2** and **3**, the algorithm is of order n^2.

‡ So **Big-O** of the algorithm is **$O(n^2)$**.

‡ In other words, run-time of this algorithm increases roughly as the square of the size of the input data which is n^2 , say an array of size **n x n**.

◇ **Example 3** :

Polynomial in n with degree k

- Let the number of steps needed to carry out an algorithm is expressed as

  $$f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n^1 + a_0$$

  Then **f(n)** is a polynomial in **n** with degree **k** and **f(n)** $\in$ **O(n^k)**.

- To obtain the order of a polynomial function, use the term which is of highest degree and disregard the constants and the terms which are of lower degrees.

  The Big-O of the algorithm is **O(n^k)**.

  In other words, run-time of this algorithm increases exponentially.

◇ Growth Rates Variation

The Growth rates vary with number of inputs and time.

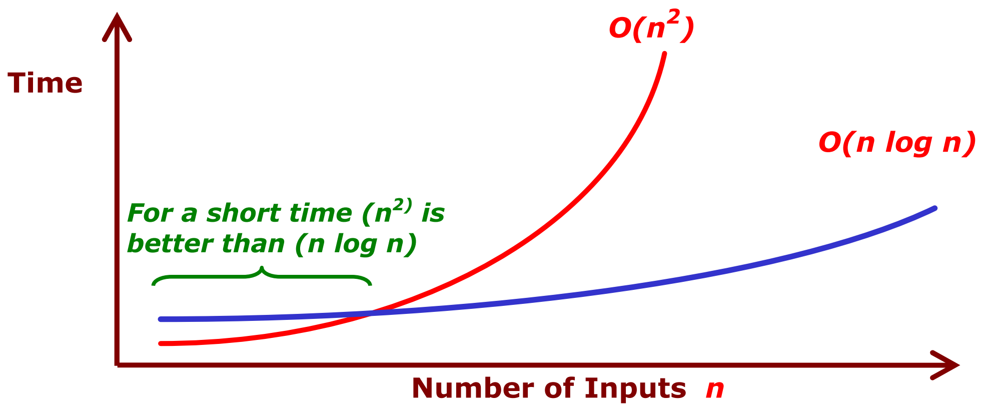


Fig. Growth rates variation with number of inputs and time

If **n** is sufficiently large

then **$\log n < n < n\log n < n^2 < n^3 < 2^n$**.

So is the **Big-O** are

$O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n)$.

Here logn is interpreted as base-2 logarithm. The base of logarithm does not matter because in determining the Big-O of an algorithm constants are ignored as mentioned.

Note : $\log_2 n = \log_{10} n / \log_{10} 2$, where $\log_{10} 2 = 0.3010299957$.

◊ Example of Growth Rates Variation

Problem : If an algorithm requires **1** second run time for a problem of size **8** , then find run time for that algorithm for the problem of size **16** ?

Solutions: If the order of the algorithm is **O(f(n))** then the calculated execution time **T(n)** of the algorithm as problem size increases are as below.

Order of the algorithm O(f(n))		**Run time T(n) required as the problem size increases**
$O(1)$ **constant time**	⇒	**$T(n) = 1$ second ;** run time is independent of the size of the problem.
$O(\log_2 n)$ **logarithmic time**	⇒	**$T(n) = (1*\log_2 16) / \log_2 8 = 4/3$ seconds ;** run time increases slowly with the size of the problem.
$O(n)$ **linear time**	⇒	**$T(n) = (1*16) / 8 = 2$ seconds** run time increases directly with the size of the problem.
$O(n*\log_2 n)$ **log-linear time**	⇒	**$T(n) = (1*16*\log_2 16) / 8*\log_2 8 = 8/3$ seconds** run time increases more rapidly than the linear algorithm with the size of the problem.
$O(n^2)$ **quadratic time**	⇒	**$T(n) = (1*16^2) / 8^2 = 4$ seconds** the run time increases rapidly with the size of the problem.
$O(n^3)$ **cubic time**	⇒	**$T(n) = (1*16^3) / 8^3 = 8$ seconds** the run time increases more rapidly than quadratic algorithm with the size of the problem.
$O(2^n)$ **exponential time**	⇒	**$T(n) = (1*2^{16}) / 2^8 = 2^8$ seconds $= 256$ seconds** the run time increases too rapidly to be practical.

• Tree Structure

Tree is a way of organizing objects, related in a hierarchical fashion.

Tree is a type of data structure where

– each *element* is attached to one or more elements directly beneath it.

– the connections between elements are called *branches*.

– tree is often called *inverted trees* because its *root* is at the top.

– the elements that have no elements below them are called *leaves*.

– a *binary tree* is a special type, each element has two branches below it.

◇ Example

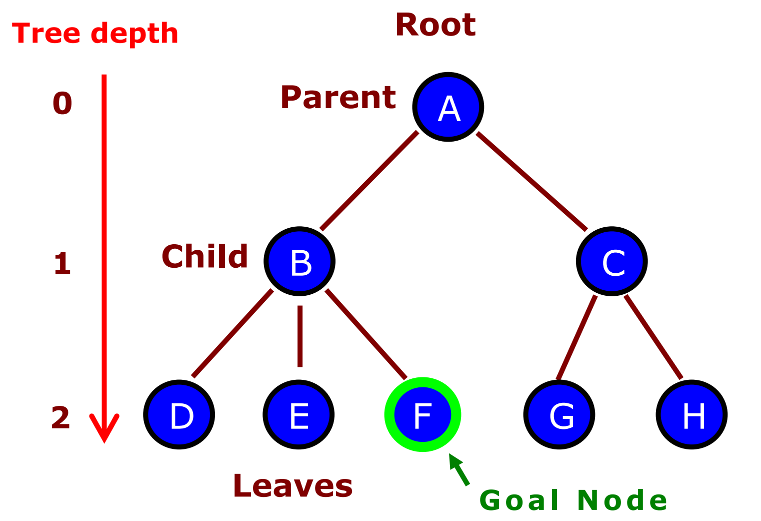


Fig. A simple example unordered tree

◇ Properties

‡ Tree is a special case of a *graph*.

‡ The topmost node in a tree is called the *root node*; at root node all operations on the tree begin.

‡ A node has at most one *parent*. The topmost node called root node has no parents.

‡ Each node has either zero or more *child nodes* below it .

‡ The Nodes at the bottom most level of the tree are called *leaf nodes*. The *leaf nodes* do not have children.

‡ A node that has a child is called the child's *parent node*.

‡ The *depth of a node* ***n*** is the length of the path from the root to the node; The root node is at depth zero.

• Stacks and Queues

The Stacks and Queues are data structures .

It maintains the order *last-in first-out* and *first-in first-out* respectively.

Both *stacks* and *queues* are often implemented as ***linked lists***, but that is not the only possible implementation.

◇ Stack

An ordered list works as Last-In First-Out (LIFO);

Here the items are in a sequence and piled one on top of the other.

‡ The *insertions* and *deletions* are made at one end only, called *Top*.

‡ If Stack **S = a[1], a[2], ... a[n]** then **a[1]** is bottom most element

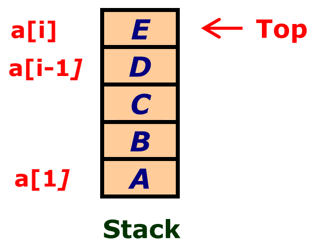
‡ Any intermediate element **a[i]** is on top of element **a[i-1]** where **1 < i <= n.**

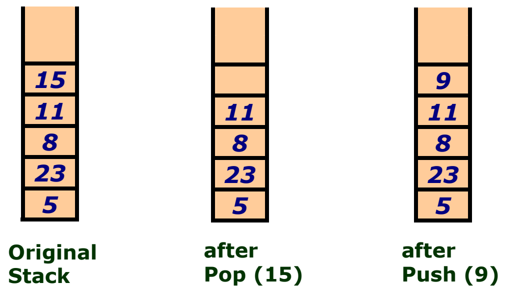
‡ In a Stack all operations take place on Top.

The *Pop* operation removes item from top of the stack.

The *Push* operation adds an item on top of the stack.

Items enter Stack at Top and leave from Top





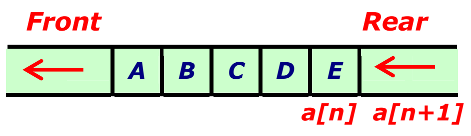
◇ Queue

An ordered list works as First-In First-Out (FIFO)

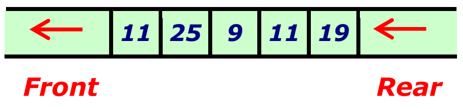
Here the items are in a sequence. There are restrictions about how items can be added to and removed from the list.

- A queue has two ends.
- All *insertions (enqueue)* take place at one end, called *Rear* or *Back*.
- All *deletions (dequeue)* take place at other end, called *Front*.
- If Queue has **a[n]** as rear element then **a[i+1]** is behind **a[i]** where **1 < i <= n.**
- All operation take place at one end of queue or the other end.
  a dequeue operation removes the item at Front of the queue.
  a enqueue operation adds an item to the Rear of the queue.

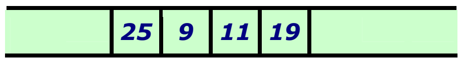
Items enter Queue at Rear and leave from Front



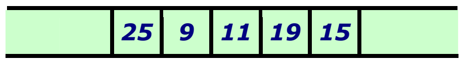
Queue



Original Queue



After Dequeue (11)



After Enqueue (15)

2.2 Search

Search is the systematic examination of states to find path from the start/root state to the goal state.

- search usually results from a lack of knowledge.
- search explores knowledge alternatives to arrive at the best answer.
- search algorithm output is a solution, ie, a path from the initial state to a state that satisfies the goal test.

For general-purpose problem solving : **"Search" is an approach**.

- search deals with finding nodes having certain properties in a graph that represents search space.
- search methods explore the *search space* "intelligently", evaluating possibilities without investigating every single possibility.

Example : Search tree

The search trees are multilevel indexes used to guide the search for data items, given some search criteria.

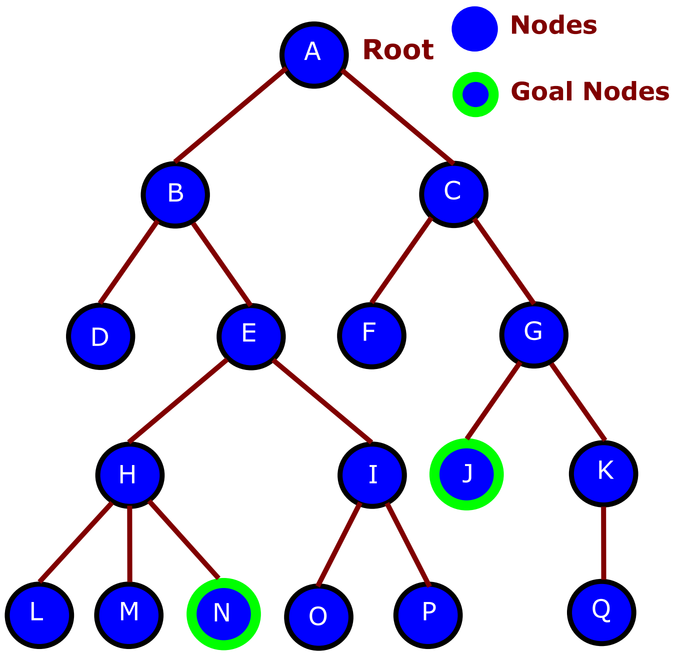


Fig. Tree Search.

The search starts at *root* and explores *nodes* looking for a *goal node*, that satisfies certain conditions depending on the problem.

For some problems, any goal node, **N** or **J ,** is acceptable;

For other problems, it may be a minimum depth goal node, say **J** which is nearest to root.

• Search Algorithms :

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using **heuristic functions .**

The algorithms that use heuristic functions are called **heuristic algorithms**.

– Heuristic algorithms are *not really intelligent*; they appear to be intelligent because they achieve better performance.

– Heuristic algorithms are *more efficient* because they take advantage of feedback from the data to direct the search path.

– Uninformed search algorithms or Brute-force algorithms search, through the search space, all possible candidates for the solution checking whether each candidate satisfies the problem's statement.

– Informed search algorithms use heuristic functions, that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

A good heuristic can make an informed search dramatically out-perform any uninformed search. For example, the Traveling Salesman Problem (TSP) where the goal is to find a *good solution* instead of finding the *best solution*.

In TPS like problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution. Such techniques help in finding a solution within reasonable time and space.

Some prominent intelligent search algorithms are stated below.

1. **Generate and Test Search**
2. **Best-first Search**
3. **Greedy Search**
4. **A* Search**
5. **Constraint Search**
6. **Means-ends analysis**

There are more algorithms, either an improvement or combinations of these.

- **Hierarchical Representation of Search Algorithms**

A representation of most search algorithms is illustrated below. It begins with two types of search - Uninformed and Informed.

Uninformed Search : Also called *blind*, *exhaustive* or *brute-force* search, uses no information about the problem to guide the search and therefore may not be very efficient.

Informed Search : Also called *heuristic* or *intelligent* search, uses information about the problem to guide the search, usually guesses the distance to a goal state and therefore efficient, but the search may not be always possible.

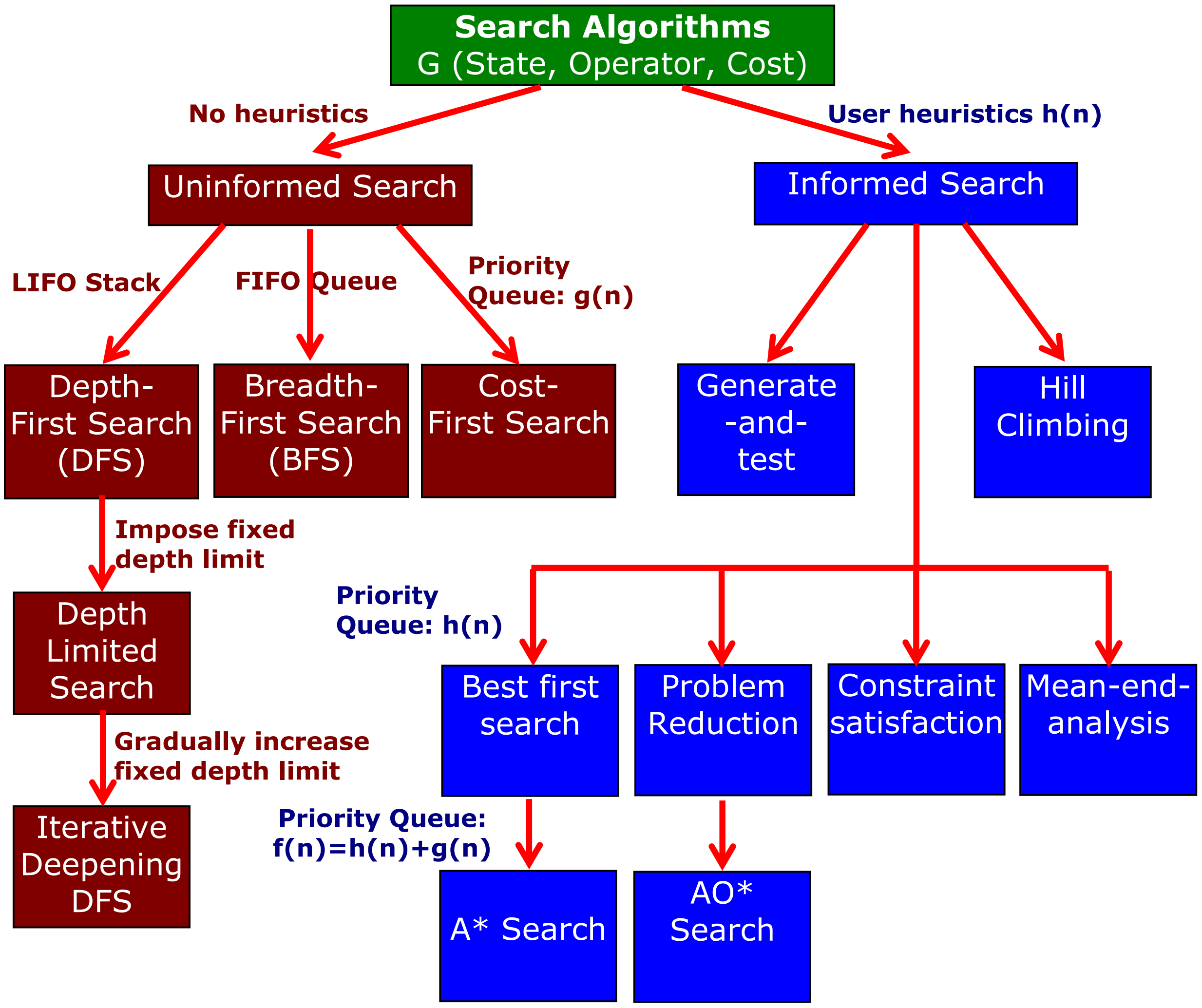


Fig. Different Search Algorithms

- **Search Space**

A set of all states , which can be reached, constitute a search space.
This is obtained by applying some combination of operators defining their connectivity.

Example :
Find route from **Start** to **Goal** state.
Consider the vertices as city and the edges as distances.

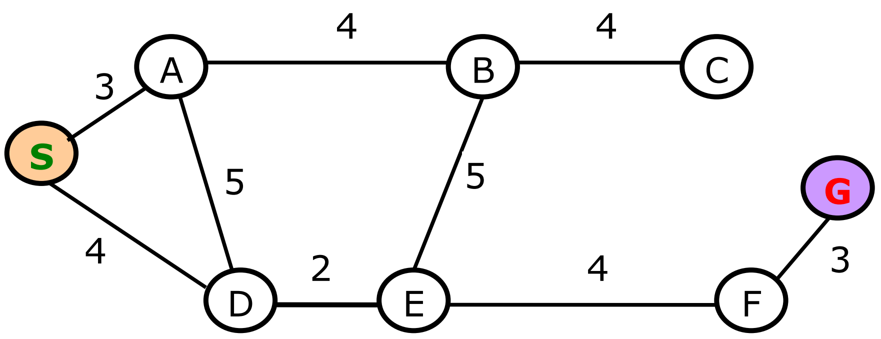


Fig. Search Space

- Initial State **S**
- Goal State **G**
- Nodes represent cities
- Arcs represent distances

- **Formal Statement :**

Problem solving is a set of statements describing the desired states expressed in a suitable language; e.g., *first-order logic*.

The solution of many problems can be described by finding a sequence of actions that lead to a desired goal (e.g., problems in chess and cross).

- the aim is to find the sequence of actions that lead from the initial (start) state to a final (goal) state.
- each action changes the state.

A well-defined problem can be described by an example stated below.

Example

- Initial State: **(S)**
- Operator or successor function : for any state **x** , returns **s(x)**, the set of states reachable from **x** with one action.
- State space : all states reachable from initial by any sequence of actions.
- Path : sequence through state space.
- Path cost : function that assigns a cost to a path;
  cost of a path is the sum of costs of individual actions along the path.
- Goal state : **(G)**
- Goal test : test to determine if at goal state.

- **Search notations**

Search is the systematic examination of states to find path from the start or root state to the goal state. The notations used for defining search are

– **f(n)** is evaluation function that estimates least cost solution through node **n.**

– **h(n)** is heuristic function that estimates least cost path from node **n** to goal node.

– **g(n)** is cost function that estimates least cost path from start node to node **n.**

The relation among three parameters are expressed

$$f(n) = g(n) + h(n)$$

$$\underbrace{\underbrace{\textbf{Start} \xrightarrow{\textbf{actual}} n}_{g(n)} \underbrace{\xrightarrow{\textbf{estimate}} \textbf{Goal}}_{h(n)}}_{f(n)}$$

◇ If **h(n)** $\le$ actual cost of shortest path from node **n** to goal then **h(n)** is an under-estimate.

◇ The estimated values of **f , g , h** are expressed as $\hat{F}$ $\hat{g}$ $\hat{h}$

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

Note: For easy to write, symbol $\wedge$ is replaced by symbol ***** and written

$\hat{g}$ is replaced with **g*** and
$\hat{h}$ is replaced with **h***

◇ The estimates of **g*** and **h*** are given in the next slide

- **Estimate Cost Function g***

  An estimated least cost path from *start node* to *node* ***n,*** is written as **g*(n).**

  - **g*** is calculated as the actual cost so far of the explored path.
  - **g*** is known by summing all path costs from *start* to *current* state.
  - If search space is a *tree*, then **g* = g**, because there is only one path from start node to current node.
  - In general, the search space is a *graph*.
  - If search space is a *graph*, then **g* ≥ g,**
    - **g*** can never be less than the cost of the optimal path; rather it can only over estimate the cost.
    - **g*** can be equal to **g** in a graph if chosen properly.

- **Estimate Heuristic Function h***

  An estimated least cost path from *node* **n** to *goal node* ,
  is written as **h*(n)**

  - **h*** is a heuristic information, it represents a guess; example :
    "how hard it is to reach from current node to goal state ? ".

  - **h*** may be estimated using an evaluation function **f(n)** that measures "*goodness*" of a node.

  - **h*** may have different values;
    the values lie between **$0 \leq h^*(n) \leq h(n)$**;
    they mean a different search algorithm.

  - If **$h^* = h$** , it is a "*perfect heuristic*";
    means no unnecessary nodes are ever expanded.

2.3 Control Strategies

Search for a solution in a problem space, requires *"Control Strategies"* to control the search processes.

The search control strategies are of different types, and are realized by some specific type of **"Control Structures"**.

- **Strategies for search**

  Some widely used control strategies for search are stated below.

  - **Forward search :** Here, the control strategies for exploring search proceeds forward from initial state to wards a solution; the methods are called *data-directed*.
  - **Backward search :** Here, the control strategies for exploring search proceeds backward from a goal or final state towards either a solvable sub problem or the initial state; the methods are called *goal directed*.
  - **Both forward and backward search :** Here, the control strategies for exploring search is a *mixture of both* forward and backward strategies .
  - **Systematic search :** Where search space is small, a systematic (but blind) method can be used to explore the whole search space. One such search method is *depth-first search* and the other is *breath-first search*.

- **Heuristic search :** Many search depend on the knowledge of the problem domain. They have some measure of relative merits to guide the search. The search so guided are called *heuristic search* and the methods used are called *heuristics* .

  Note : A heuristic search might not always find the best solution but it is guaranteed to find a good solution in reasonable time.

  Heuristic Search Algorithms :
  - First, generate a possible solution which can either be a point in the problem space or a path from the initial state.
  - Then, test to see if this possible solution is a real solution by comparing the state reached with the set of goal states.
  - Lastly, if it is a real solution, return, else repeat from the first again.

- **More on Search Strategies :** Related terms

The Condition-action rules and Chaining : Forward and Backward

◇ **Condition-action rules**

- one way of encoding Knowledge is condition-action rules
- the rules are written as **if < condition> then < conclusion >**

Rule: Red_Light	**IF**	**the light is red**	**THEN**	**Stop**
Rule: Green_Light	**IF**	**the light is green**	**THEN**	**Go**
		antecedent		**consequent**

◇ **Chaining**

- Chaining refers to sharing conditions between rules, so that the same condition is evaluated once for all rules.
- When one or more conditions are shared between rules, they are considered "chained."
- Chaining are of two types : Forward and Backward chaining.
- Forward chaining is called *data-driven* and
- Backward chaining is called *query-driven*.

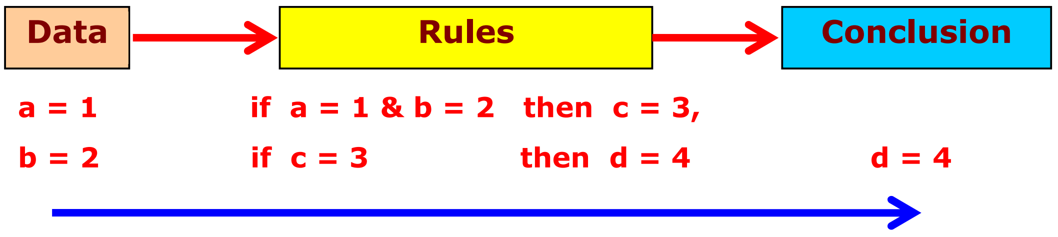
◇ **Activation of Rules**

- The Forward chaining and Backward chaining are the two different strategies for activation of *rules* in the system.
- Forward and Backward chaining are some techniques for drawing *inferences* from Rule base.

Forward Chaining Algorithm

Forward chaining is a techniques for drawing inferences from Rule base. Forward-chaining *inference* is often called *data driven*.

- The algorithm proceeds from a given situation to a desired goal, adding new assertions (facts) found.
- A forward-chaining, system compares data in the *working memory* against the conditions in the IF parts of the rules and determines which rule to fire.
- **Data Driven**



- **Example : Forward Channing**
  - Given : A Rule base contains following Rule set;

Rule 1:	If **A** and **C**	Then	**F**
Rule 2:	If **A** and **E**	Then	**G**
Rule 3:	If **B**	Then	**E**
Rule 4:	If **G**	Then	**D**

  - Problem : Prove that

    If **A** and **B** true Then **D** is true

■ **Solution** : ***(continued - the Forward Chaining problem)***

(i) ǂ Start with input given **A, B** is true and then

ǂ start at **Rule 1** and go forward / down till a rule "fires'' is found.

First iteration :

(ii) ǂ **Rule 3** fires : conclusion **E** is true

ǂ new knowledge found

(iii) ǂ No other rule fires;

ǂ end of first iteration.

(iv) ǂ Goal not found;

ǂ new knowledge found at (ii);

ǂ go for second iteration

Second iteration :

(v) ǂ **Rule 2** fires : conclusion **G** is true

ǂ new knowledge found

(vi) ǂ **Rule 4** fires : conclusion **D** is true

ǂ Goal found;

ǂ Proved

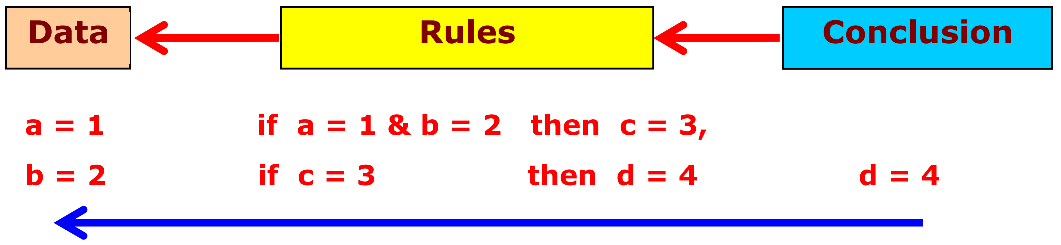
• Backward Chaining Algorithm

Backward chaining is a techniques for drawing inferences from Rule base. Backward-chaining *inference* is often called *goal driven*.

◇ The algorithm proceeds from desired *goal*, adding new assertions found.

◇ A backward-chaining, system looks for the action in the THEN clause of the rules that matches the specified goal.

◇ **Goal Driven**



◇ **Example : Backward Channing**

◇ Given : Rule base contains following Rule set

Rule 1:	If **A** and **C**	Then	**F**
Rule 2:	If **A** and **E**	Then	**G**
Rule 3:	If **B**	Then	**E**
Rule 4:	If **G**	Then	**D**

◇ Problem : Prove

If **A** and **B** true Then **D** is true

◇ **Solution :** ***(continued - the Backward Chaining problem)***

(i) ‡ Start with goal ie **D** is true

‡ go backward/up till a rule "fires'' is found.

First iteration :

(ii) ‡ **Rule 4** fires :

‡ new sub goal to prove **G** is true

‡ go backward

(iii) ‡ **Rule 2** "fires''; conclusion: **A** is true

‡ new sub goal to prove **E** is true

‡ go backward;

(iv) ‡ no other rule fires; end of first iteration.

‡ new sub goal found at (iii);

‡ go for second iteration

Second iteration :

(v) ‡ **Rule 3** fires :

‡ conclusion **B** is true (2nd input found)

‡ both inputs **A** and **B** ascertained

‡ Proved

3. Exhaustive Search

Besides Forward and Backward chaining explained, there are many other search strategies used in computational intelligence. Among the most commonly used approaches are :
Breadth-first search (BFS) and *depth-first search (DFS)*.

A search is said to be exhaustive if the search is guaranteed to generate all reachable states (outcomes) before it terminates with failure.
A graphical representation of all possible reachable states and the paths by which they may be reached is called *decision tree.*

Breadth-first search (BFS) : A Search strategy, in which the highest layer of a decision tree is searched completely before proceeding to the next layer is called Breadth-first search (BFS).

- In this strategy, no viable solution is omitted and therefore guarantee that optimal solution is found.
- This strategy is often not feasible when the search space is large.

Depth-first search (DFS) : A search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path is called Depth-first search (DFS).

- This strategy does not guarantee that the optimal solution has been found.
- In this strategy, search reaches a satisfactory solution more rapidly than breadth first, an advantage when the search space is large.

The Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques.

- **Breadth-First Search Strategy (BFS)**

This is an exhaustive search technique.

- ◇ The search generates all nodes at a particular level before proceeding to the next level of the tree.
- ◇ The search systematically proceeds testing each node that is reachable from a parent node before it expands to any child of those nodes.
- ◇ The control regime guarantees that the space of possible moves is systematically examined; this search requires considerable memory resources.
- ◇ The space that is searched is quite large and the solution may lie a thousand steps away from the start node. It does, however, guarantee that if we find a solution it will be the shortest possible.
- ◇ Search terminates when a solution is found and the test returns true.

- **Depth-First Search Strategy (DFS)**

This is an exhaustive search technique to an assigned depth.

◇ Here, the search systematically proceeds to some depth **d**, before another path is considered.

◇ If the maximum depth of search tree is three, then if this limit is reached and if the solution has not been found, then the search backtracks to the previous level and explores any remaining alternatives at this level, and so on.

◇ It is this systematic backtracking procedure that guarantees that it will systematically and exhaustively examine all of the possibilities.

◇ If the tree is very deep and the maximum depth searched is less then the maximum depth of the tree, then this procedure is "exhaustive modulo of depth" that has been set.

● **Depth-First Iterative-Deepening (DFID)**

DFID is another kind of exhaustive search procedure which is a blend of depth first and breadth first search.

Algorithm : Steps

- First perform a Depth-first search (DFS) to depth one.
- Then, discarding the nodes generated in the first search, starts all over and do a DFS to level two.
- Then three . . . . until the goal state is reached.

3.1 Depth-First Search (DFS)

Here explained the Depth-first search tree, the backtracking to the previous level, and the Depth-first search algorithm

- DFS explores a path all the way to a leaf before backtracking and exploring another path.
- **Example:** Depth-first search tree

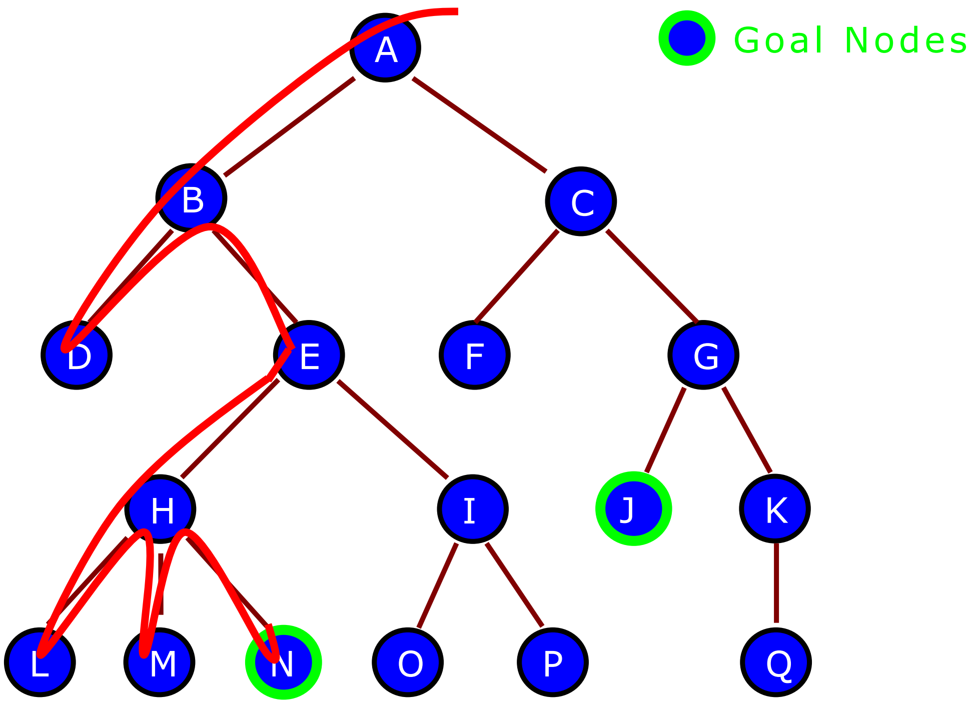


Fig. Depth-first search (DFS)

- Node are explored in the order :

  A B D E H L M N I O P C F G J K Q

- After searching node **A**, then **B**, then **D**, the search *backtracks* and tries another path from node **B .**
- The goal node **N** will be found before the goal node **J.**

◇ **Algorithm -** Depth-first search

```
■ Put the root node on a stack;
  while (stack is not empty)
    { remove a node from the stack;
      if (node is a goal node) return success;
      put all children of node onto the stack; }
  return failure;
```

Note :

- At every step, the stack contains some nodes from each level.
- The stack size required depends on the branching factor **b**.
- Searching level **n**, the stack contains approximately **b** ∗ **n** nodes.
- When this method succeeds, it does not give the path.
- To hold the search path the algorithm required is *"Recursive depth-first search"* and stack size large.

3.2 Breadth-First Search (BFS)

Here explained, the Breadth-first search tree, and the Breadth-first search algorithm.

◇ BFS explores nodes nearest to the root before exploring nodes that are father or further away.

◇ **Example:** Breadth-first search tree

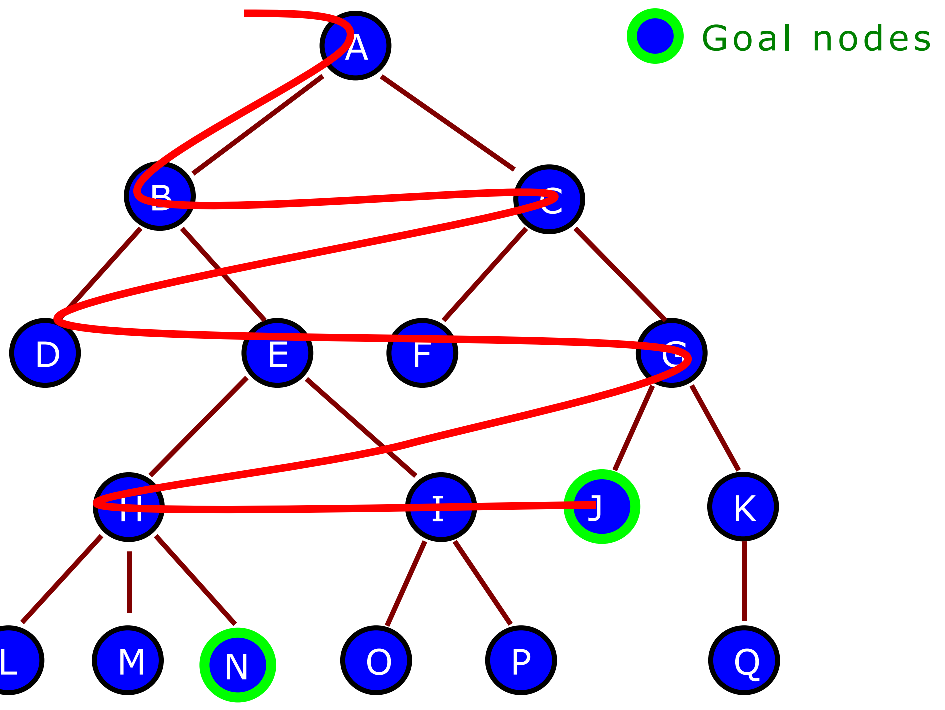


Fig. Breadth-first search (BFS)

▪ Node are explored in the order :

A B C D E F G H I J K L M N O P Q

▪ After searching **A**, then **B**, then **C**, the search proceeds with **D**, **E**, **F**, **G**, ...

▪ The goal node **J** will be found before the goal node **N**.

◇ **Algorithm -** Breadth-first search

```
Put the root node on a queue;
while (queue is not empty)
  { remove a node from the queue;
    if (node is a goal node) return success;
    put all children of node onto the queue; }
return failure;
```

Note :

- Just before the start to explore level **n**, the queue holds all the nodes at level **n-1**.
- In a typical tree, the number of nodes at each level increases exponentially with the depth.
- Memory requirements may be infeasible.
- When this method succeeds, it does not give the path.
- There is no *"recursive"* breadth-first search equivalent to recursive depth-first search.

3.3 Compare Depth-First and Breadth-First Search

Here the Depth-first and Breadth-first search are compared at algorithm level, at feature level and how to over come the limitations.

◇ Compare Algorithms

Depth-first search	Breadth-first search
Put the root node on a stack;	Put the root node on a queue;
while (stack is not empty)	while (queue is not empty)
{	{
remove a node from the stack;	remove a node from the queue;
if (node is a goal node) return success;	if (node is a goal node) return success;
put all children of node onto the stack;	put all children of node onto the queue;
}	}
return failure;	return failure;

◇ Compare Features

Depth-first search	Breadth-first search
‡ When succeeds, the goal node found is not necessarily minimum depth	‡ When succeeds, it finds a minimum-depth (nearest to root) goal node
‡ Large tree, may take excessive long time to find even a nearby goal node	‡ Large tree, may require excessive memory

◇ How to over come limitations of DFS and BFS ?

- ‡ Requires, how to combine the advantages and avoid disadvantages?
- ‡ The answer is *"Depth-limited search"* .

  This means, perform Depth-first searches with a depth limit.

4. Heuristic Search Techniques

For complex problems, the traditional algorithms, presented above, are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using **heuristic functions**

– Blind search is not always possible, because they require too much time or Space (memory).

– Heuristics are **rules of thumb**; they do not guarantee for a solution to a problem.

– Heuristic Search is a weak techniques but can be effective if applied correctly; they require domain specific information

4.1 Characteristics of Heuristic Search

- Heuristics, are knowledge about domain, which help search and reasoning in its domain.
- Heuristic search incorporates domain knowledge to improve efficiency over blind search.
- Heuristic is a function that, when applied to a state, returns value as estimated merit of state, with respect to goal.
  - Heuristics might (for reasons) under estimate or over estimate the merit of a state with respect to goal.
  - Heuristics that under estimates are desirable and called admissible.
- Heuristic evaluation function estimates likelihood of given state leading to goal state.
- Heuristic search function estimates cost from current state to goal, presuming function is efficient.

4.2 Heuristic Search compared with other search

The Heuristic search is compared with Brute force or Blind search techniques

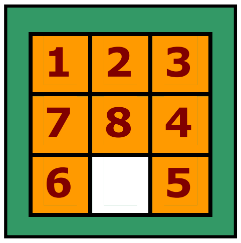
Compare Algorithms

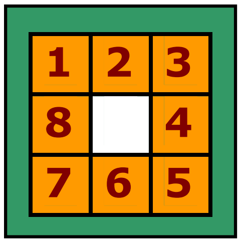
Brute force / Blind search	Heuristic search
◇ Only have knowledge about already explored nodes	◇ Estimates "distance" to goal state
◇ No knowledge about how far a node is from goal state	◇ Guides search process toward goal state
	◇ Prefer states (nodes) that lead close to and not away from goal state

- **Example : 8 - Puzzle**

◇ State space: Configuration of 8- tiles on the board

◇ state **Initial**: any configuration **Goal**: tiles in a specific order





◇ Solution: optimal sequence of operators

◇ Action: “blank moves”

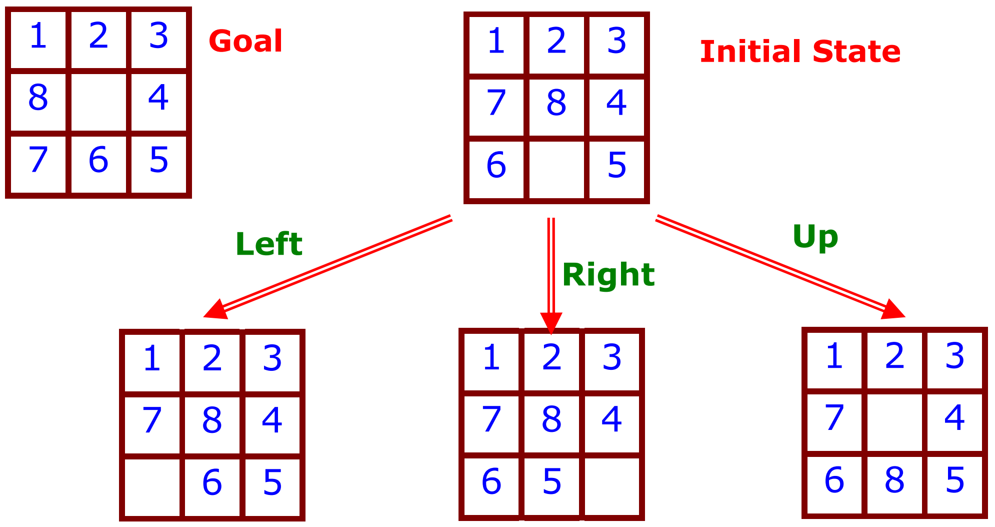
- Condition: the move is within the board
- Directions: Left, Right, Up, Dn

◇ Problem

- which 8-puzzle move is best?
- what heuristic(s) can decide?
- which move is “best” (worth considering first) ?

◊ **Actions**

Figure below shows: three possible moves - left , up, right



Count correct positions	h = 6	h = 4	h = 5

Find Which move is best ?

◊ Apply the Heuristic : Three different approaches

- Count correct position of each tile, compare to goal state
- Count incorrect position of each tile, compare to goal state
- Count how far away each tile is from it is correct position.

Approaches	Left	Right	Up
1. Count correct position	6	4	5
2. Count incorrect position	2	4	3
3. Count how far away	2	4	4

Each of these three approaches are explained below.

◊ **Heuristic :**

Three different approaches

- 1st approach :

  Count correct position of each tile, compare to goal state.
  - Higher the number the better it is.
  - Easy to compute (fast and takes little memory).
  - Probably the simplest possible heuristic.

- 2nd approach

  Count incorrect position of each tile, compare to goal state
  - Lower the number the better it is.
  - The “best” move is where lowest number returned by heuristic.

- 3rd approach

  Count how far away each tile is from it’s correct position
  - Count how far away (how many tile movements) each tile is from it’s correct position.
  - Sum up these count over all the tiles.
  - The “best” move is where lowest number returned by heuristic.

4.3 Heuristic Search Algorithms : types

- Generate-And-Test
- Hill climbing
  - Simple
  - Steepest-Ascent Hill climbing
  - Simulated Anealing
- Best First Search
  - OR Graph
  - A* (A-Star) Algorithm
  - Agendas
- Problem Reduction
  - AND-OR_Graph
  - AO* (AO-Star) Algorithm
- Constraint Satisfaction
- Mean-end Analysis

5. Constraint Satisfaction Problems (CSPs) and Models

Constraints arise in most areas of human endeavor.

Constraints are a natural medium for people to express problems in many fields.

Many real problems in AI can be modeled as Constraint Satisfaction Problems (CSPs) and are solved through search.

Examples of constraints :

- The sum of three angles of a triangle is 180 degrees,
- The sum of the currents flowing into a node must equal zero.

◇ Constraint is a logical relation among variables.

- the constraints relate objects without precisely specifying their positions; moving any one, the relation is still maintained.
- example : "circle is inside the square".

◇ Constraint satisfaction

The Constraint satisfaction is a process of finding a solution to a set of constraints.

- the constraints articulate allowed values for variables, and
- finding solution is evaluation of these variables that satisfies all constraints.

◇ Constraint Satisfaction problems (CSPs)

The CSPs are all around us while managing work, home life, budgeting expenses and so on;

- where we do not get success in finding solution, there we run into problems.
- we need to find solution to such problems satisfying all constraints.
- the Constraint Satisfaction problems_(CSPs) are solved through search.

5.1 Examples of CSPs

Some poplar puzzles like, the Latin Square, the Eight Queens, and Sudoku are stated below.

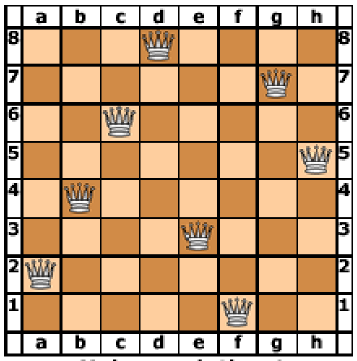
◇ **Latin Square Problem :** How can one fill an **n × n** table with **n** different symbols such that each symbol occurs exactly once in each row and each column ?

Solutions : The Latin squares for **n = 1, 2, 3** and **4** are :

$$\begin{bmatrix} 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

◇ **Eight Queens Puzzle Problem :** How can one put 8 queens on a (8 x 8) chess board such that no queen can attack any other queen ?

Solutions: The puzzle has 92 distinct solutions. If rotations and reflections of the board are counted as one, the puzzle has 12 unique solutions.



Unique solution 1

◇ **Sudoku Problem :** How can one fill a partially completed (9 × 9) grid such that each row, each column, and each of the nine (3 × 3) boxes contains the numbers from 1 to 9.

Problem

	2	6				8	1	
3			7		8			6
4				5				7
	5		1		7		9	
		3	9		5	1		
	4		3		2		5	
1				3				2
5			2		4			9
	3	8				4	6	

Solution

7	***2***	***6***	4	9	3	***8***	***1***	5
3	1	5	***7***	2	***8***	9	4	***6***
4	8	9	6	***5***	1	2	3	***7***
8	***5***	2	***1***	4	***7***	6	***9***	3
6	7	***3***	***9***	8	***5***	***1***	2	4
9	***4***	1	***3***	6	***2***	7	***5***	8
1	9	4	8	***3***	6	5	7	***2***
5	6	7	***2***	1	***4***	3	8	***9***
2	***3***	***8***	5	7	9	***4***	***6***	1

5.2 Constraint Satisfaction Models

Humans solve the puzzle problems stated above while

– trying with different configurations, and
– using various insights about the problem to explore only a small number of configurations.

It is not clear what these insights are ?

For n = 8 queens, on a standard chess board (8 × 8), such that no queen can attack any other queen, the puzzle has 92 distinct solutions.
Humans would find it hard to solve N-Queens puzzle while **N** becomes more.

Example : The possible number of configurations are :

– For 4-Queens there are 256 different configurations.
– For 8-Queens there are 16,777,216 configurations.
– For 16-Queens there are 18,446,744,073,709,551,616 configurations.
– In general, for N - Queens there are we have N^N configurations.
– For N = 16, this would take about 12,000 years on a fast machine.

How do we solve such problems ?

Three computer based approaches or models are stated below. They are

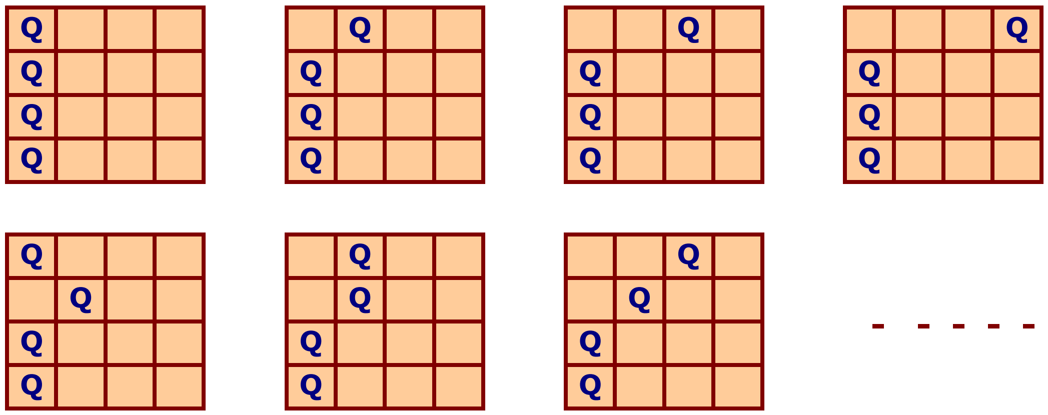
‡ Generate and Test (GT) ,
‡ Backtracking (BT) and
‡ Constrain Satisfaction Problems (CSPs)

◇ Generate and Test (GT) : n = 4 Queens puzzle

One possible solution is to systematically try every placement of queens until we find a solution.

The process is known as "Generate and Test".

Examples of Generate and Test conditions for solutions :



◇ **Backtracking (BT) :** n = 4 Queens puzzle

The Backtracking method is based on systematic examination of the possible solutions.

- The algorithms try each possibility until they find the right one.
- It differs from simple brute force, which generates all solutions, even those arising from infeasible partial solutions.

Backtracking is similar to a depth-first search but uses less space, keeping just one current solution state and updating it.

- ‡ during search, if an alternative does not work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative.
- ‡ when the alternatives are exhausted, the search returns to the previous choice point and tries the next alternative there.
- ‡ if there are no more choice points, the search fails.

This is usually achieved in a *recursive function* where each instance takes one more variable and alternatively assigns all the available values to it, keeping the one that is consistent with subsequent recursive calls.

Note : feasible, infeasible, pruning the solution tree.

- The partial solutions are evaluated for feasibility.
- A partial solution is said to be *feasible* if it can be developed by further choices without violating any of the problem's constraints.
- A partial solution is said to be *infeasible* if there are no legitimate options for any remaining choices.
- The abandonment of infeasible partial solutions is called *pruning the solution tree.*

Backtracking is most efficient technique for problems, like **n**-Queens problem. An example below illustrates to solve **N = 4** Queens problem.

Example : Backtracking to solve **N = 4** Queens problem.

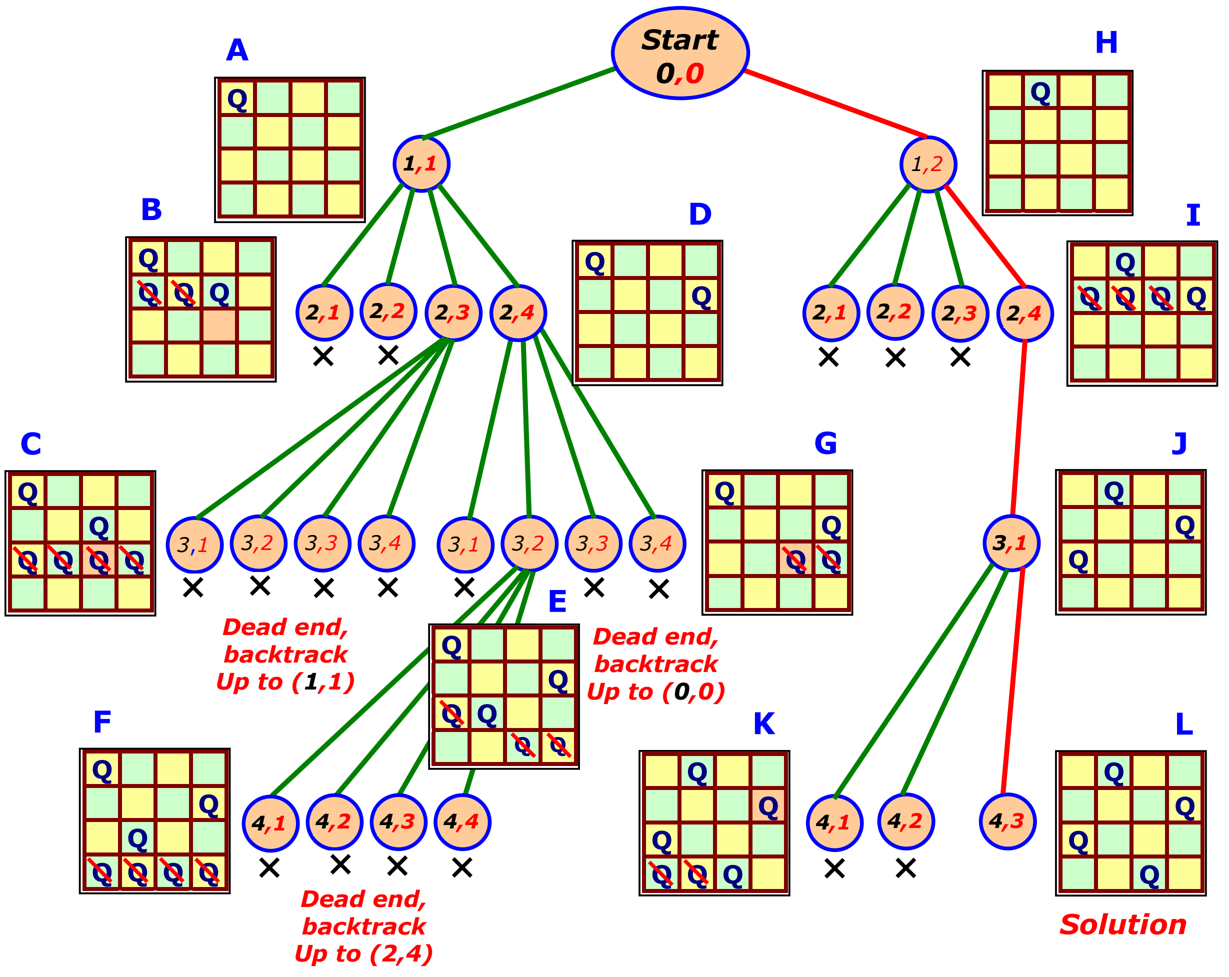

Start 0,0
A
B
C
D
E
F
G
H
I
J
K
L
1,1
1,2
2,1
2,2
2,3
2,4
3,1
3,2
3,3
3,4
4,1
4,2
4,3
4,4
Dead end, backtrack Up to (1,1)
Dead end, backtrack Up to (0,0)
Dead end, backtrack Up to (2,4)
Solution

Fig. Backtracking to solve N = 4 Queens problem.

The Fig above illustrates the *state space tree* for the instance of the **N=4** Queens problem. The ordered pair **(i , j)** in each node indicates a possible **(row, column)** placement of a queen.

Algorithm : Backtracking to solve **N** Queens problem.

The problem proceeds either by rows or by columns.

- for no particularly good reason, select columns to proceed.
- for each column, select a row to place the queen.
- a partial solution is feasible if no two queens can attack each other;

Note : No feasible solution can contain an infeasible partial solution.

1. Move "left to right" processing one column at a time.
2. For column **J**, select a row position for the queen. Check for feasibility.
   a. If there are one or more attacks possible from queens in columns **1** through **(J – 1)**, discard the solution.
   b. For each feasible placement in column **J**, make the placement and try placement in column **(J + 1)**.
   c. If there are no more feasible placements in column **J**, return to column **(J – 1)** and try another placement.
3. Continue until all **N** columns are assigned or until no feasible solution is found.

◇ Constrain Satisfaction Problems (CSPs)

The Backtracking just illustrated, is one main method for solving problems like N-Queens but what is required is a *generalized solution*.

Design algorithms for solving the general class of problems.
For an algorithm not just for solving N-Queens problem, we need to express N-Queens as an instance of a general class of problems;

The N-queens problem can be represented as a *constraint satisfaction problem (CSPs).*

In CSPs, we find states or objects that satisfy a number of constraints or criteria. The CSPs are solved through search.

Before we deal with CSPs, we need to define CSPs and the domain related properties of the constraints. Here we illustrate :

- the definition of CSPs,
- the properties of CSPs and
- the algorithms for CSPs.

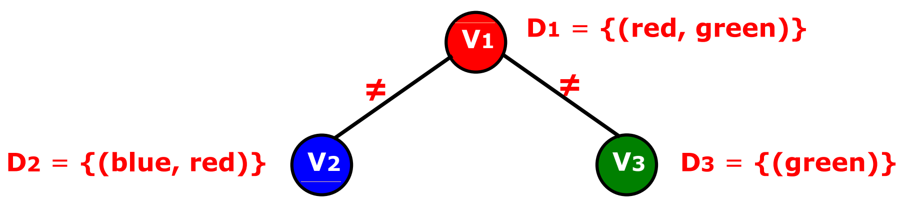
■ Definition of a CSPs

The formal definition of a CSP involves *variables*, their *domains*, and *constraints*. A constraint network is defined by

- a set of variables, $V_1, V_2, \ldots, V_n$,
- a domain of values, $D_1, D_2, \ldots, D_n$ for each variable;
- all variables V_i have a value in their respective domain D_i.
- a set of constraints, $C_1, C_2, \ldots, C_m$;
- a constraint C_i restricts the possible values in the domain of some subset of the variables.
- Problem : Is there a solution of the network, i.e., an assignment of values to the variables such that all constraints are satisfied ?
- Solution to a CSP : Is an assignment to every variable by some value in its domain such that every constraint is satisfied. Also each assignment of a value to a variable must be consistent, i.e. it must not violate any of the constraints.

Example :

Variables	■ V1 , V 2, V 3, . . with Domains D1 , D2 , D3 , . . .
Constraints	■ Set of allowed value pairs {(red, blue), (green, blue), (green, red)}
	■ V1 "not equal to" V2,
Solution	■ Assign values to variables that satisfy all constraints
	■ V1 = red , V2 = blue , V3 = green ,



■ Properties of CSPs

Constraints are used to guide reasoning of everyday common sense. The constraints enjoy several interesting properties.

– Constraints may specify *partial information*; constraint need not uniquely specify the values of its variables;

– Constraints are *non-directional*, typically a constraint on (say) two variables **V_1, V_2** can be used to infer a constraint on **V_1** given a constraint on **V_2** and vice versa;

– Constraints are *declarative;* they specify what relationship must hold without specifying a computational procedure to enforce that relationship;

– Constraints are *additive*; the order of imposition of constraints does not matter, all that matters at the end is that the conjunction of constraints is in effect;

– Constraints are *rarely independent*; typically constraints in the constraint store share variables.

■ **Algorithms for CSPs :** Consider **n = 8** Queens puzzle

As stated above, a CSP consists of 3 components :

– A set of *variables*,

– A set of *values* for each of the variables and

– A set of *constraints* imposed between the variables.

Find a value for each variables that satisfies all the constraints.

‡ **Constraints**

– A constraint is a relation between a local collection of variables.

– The constraint restricts the values that these variables can simultaneously have.

– Examples : one constraint **All-diff(X_1, X_2, X_3)**.

This constraint says that **X_1, X_2, X_3** must take on different values.

If **{1, 2, 3}** is the set of values for each of these variables then: **$X_1 = 1$, $X_2 = 2$, $X_3 = 3$** is ok and **$X_1 = 1$, $X_1 = 1$, $X_1 = 3$** not ok.

‡ **Finding a Solution**

– Finding a global assignment to all of the variables that satisfies all of the constraints is *hard:NP-Complete (nondeterministic polynomial-time hard)*

– The solution techniques work by cleverly searching through the space of possible assignments of values to variables.

– If each variable has **d** values and there are **n** variables, then we have **d^n** possible assignments.

‡ Representations : N = 8 Queens as a CSP

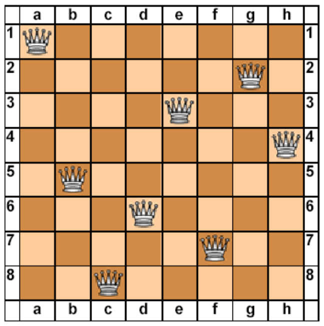
This problem can be represented as a CSP in different ways.

Representation 1

– we need to know where to place each of the **N** queens.

– we could have **N** variables each of which has as a value **1. . . N**^{2}.

– The values represent where we will place the **i** th variable.



Q_1	=	1	
Q_2	=	15	
Q_3	=	21	
Q_4	=	32	
Q_5	=	34	
Q_6	=	44	
Q_7	=	54	
Q_8	=	59	

This representation has

64^8 = 281,474,976,710,656

different possible assignments in the search space.

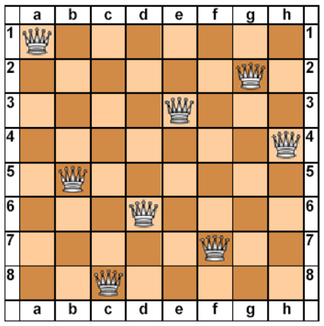
Representation 2

– Here we know, we can not place two queens in the same column.

– Assign one queen to each column, and then find out the rows where each of these queens is to placed.

– We can have **N** variables: $Q_1 , ... , Q_N$.

– The set of values for each of these variables are **{1 , 2 , . . . , N}.**



Q_1	=	1	
Q_7	=	2	
Q_5	=	3	
Q_8	=	4	
Q_2	=	5	
Q_4	=	6	
Q_6	=	7	
Q_3	=	8	

This representation has

8^8 = 16,777,216

different possible assignments in the search space.

It is still too large to examine all of them, but definitely a big improvement .

‡ Constraints :

Translate each of individual conditions into a separate constraint.

Condition 1 : Queens Cannot attack each other

If Q_i cannot attack Q_j for $(i \neq j)$

Then Q_i is a queen to be placed in column i, and

Q_j is a queen to be placed in column j.

The value of Q_i and Q_j are the rows the queens are to be placed.

Condition 2 : Queens can attack each other

- Vertically, if they are in any same column; it is impossible as Q_i and Q_j are placed in different columns.
- Horizontally, if they are in any same row; we need the constraint $Q_i \neq Q_j$.
- If along any diagonal; they cannot be in the same number of columns apart as they are rows apart: we need the constraint $| i - j | \neq | Q_i - Q_j |$; (symbol |.| is absolute value)

‡ Representing the Constraints

- Between every pair of variables (Q_i, Q_j) for $(i \neq j)$, the constraint is C_{ij}.
- For each C_{ij}, an assignment of values to the variables $Q_i = A$ and $Q_j = B$, satisfies this constraint if and only if $A \neq B$ and $|A - B| \neq |i - j|$.

‡ Solutions

A solution to N - Queens problem is any assignment of values to the variables $Q_i, \ldots, Q_N$ that satisfies all of the constraints.

- Constraints can be over any collection of variables.
- The **N - Queens** problems, need only binary constraints, i.e. constraints over pairs of variables.
- By simply enumerating and testing all possible assignments, we can recognize a subset of the variables that make a solution impossible. Thus we can largely improve upon different possible assignments in the search space.
- Finally, by expressing the problem as a CSP we have a systematic way of achieving extra efficiency.

5.3 Remarks

Few remarks on Generic Backtracking, Forward Checking and Variable Ordering are stated below.

◇ Generic Backtracking

- Generic Backtracking is the simplest and oldest algorithm for solving CSP problems.
- The idea is to search in a tree of variable assignments. As we move down the tree we assign a value to a new variable.
- Once we have assigned all of the variables that participate in a constraint, we check that constraint.
- At any point if a constraint is violated we backtrack up the tree.

Note :

- The idea of searching in a tree of variable assignments is very powerful. However generic backtracking (BT) is not a very good algorithm.
- Although BT is much faster than any simple enumeration, but then all algorithms for solving CSPs take time that can grow exponentially with the size of the problem.

◇ Forward Checking

- Forward Checking is based on the idea of looking ahead in the tree to see if we have already assigned a value to one of the unassigned variable impossible.
- It is based on the idea of pruning the domains of the unassigned variables.

◇ Variable Ordering

- Choosing a variable is critical to performance.
- The efficiency of search algorithms depends considerably on the order in which variables are considered for instantiations.
- This ordering effects the efficiency of the algorithm.
- There exist various heuristics for dynamic or static ordering of values and variables.

6. References : Textbooks

1. *"Artificial Intelligence", by Elaine Rich and Kevin Knight, (2006), McGraw Hill companies Inc., Chapter 2-3, page 29-98.*

2. *"Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig, (2002), Prentice Hall, Chapter 3-6, page 59-189.*

3. *"Computational Intelligence: A Logical Approach", by David Poole, Alan Mackworth, and Randy Goebel, (1998), Oxford University Press, Chapter 4, page 113-163.*

4. *"Artificial Intelligence: Structures and Strategies for Complex Problem Solving", by George F. Luger, (2002), Addison-Wesley, Chapter 2-6, page 35-193.*

5. *"AI: A New Synthesis", by Nils J. Nilsson, (1998), Morgan Kaufmann Inc., Chapter 7-9, Page 117-160.*

6. *"Artificial Intelligence: Theory and Practice", by Thomas Dean, (1994), Addison-Wesley, Chapter 3-4, Page 71-131.*

7. *Related documents from open source, mainly internet. An exhaustive list is being prepared for inclusion at a later date.*