University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

INTELLIGENT SEARCH TECHNIQUES

تقنيات بحث ذكية

م.م رشا محمد محسن

cs.uotechnology.edu.iq

"Artificial Intelligence Concept and Fundamentals"

1. Definition of Artificial Intelligence

Artificial intelligence (AI) is technology that enables computers and machines to simulate human learning, comprehension, problem solving, decision making, creativity and autonomy.AI The branch of computer science that is concerned with the automation of intelligent behavior.AI must be based on sound theoretical and applied principles of that field. These principles include the data structures used in knowledge representation, the algorithms needed to apply that knowledge, and the languages and programming techniques used in their implementation. For any computing system it is very important to achieve an acceptable level of software quality. The basic goal of software quality is the prevention of software faults or, at least, the lowering of software fault rates.

2. Principles fundamentals of A.I.

Artificial Intelligence and Artificial Evolution is an attempt to make a computer, a robot, or other piece of technology 'think' and process data in the same way as we humans do. AI therefore has to study how the human brain 'thinks', learns, and makes decisions when it tries to solve problems or execute a task. The aim of AI is to improve technology by adding functionality related to the human acts of reasoning, learning, and problem-solving.

AI is governed by four concepts or principles, these four concepts are:

- **Explanation**

The primary principle is that AI systems should be able to provide clear explanations for their actions. No "Umm…" or "Well, it's kind of like…" allowed. This involves elaborating on how they process data, make decisions, and arrive at specific outcomes.

- **Meaningful**

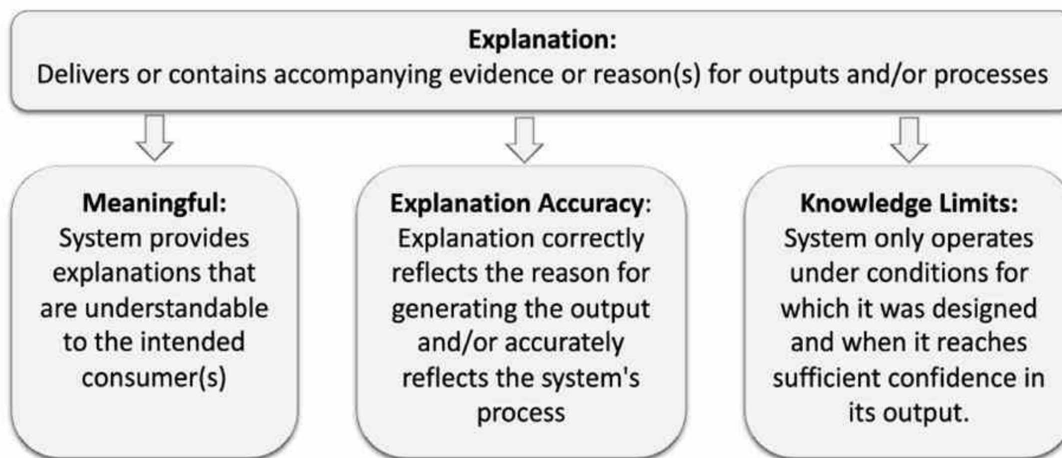
The explanations provided by AI systems need to be understandable and meaningful to humans, especially non-experts. Convoluted, technical jargon won't help a user understand why a certain decision was made. It will just lead to more confusion and lack of trust.

- **Explanation accuracy**

Yes, AI systems must provide explanations. And it's equally important for these explanations to be accurate.

- **Knowledge limits**

We all have limits that we're generally aware of, and AI should be no different. It's crucial for AI systems to be aware of their limitations and uncertainties. A system should operate only "under conditions for which it was designed and when it reaches sufficient confidence in its output," says NIST.



Part 1:"Knowledge Representation"

1. Knowledge Representation Schemes

Knowledge representation (KR) is a part of the Artificial Intelligence which responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real-world problems such as communicating with humans in natural language. It is also a way which describes how we can represent knowledge in Artificial Intelligence. Knowledge representation is not just storing data into database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

In Al, there are four basic categories of representational schemes: logical, procedural, network and structured representation schemes.

1. **Logical representation** uses expressions in **formal logic** to represent its **knowledge base**. Predicate Calculus is the most widely used representation scheme.

2. **Network representation** captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs -represent relations or associations between them.

3.Procedural representation represents knowledge as a set of instructions for solving a problem. These are usually if-then rules we use in rule-based systems.

4. Structured representation extends network representation schemes by allowing each node to have complex data structures named slots with attached values.

1.1 Logical Representation

One way of representing fact is the language of logic. The logical formalism applies because it immediately suggests a powerful way of deriving new knowledge from the old mathematical deduction. In this formalism, we can conclude that a new statement is true by proving that it follows from the already known statements. There are two types of logic representation, these are:

- Propositional logic.
- Predicate logic.

1.1.1 Propositional Logic

Propositional logic, also known as sentential logic or Boolean logic is the branch of logic that deals with propositions which can either be true or false. It forms the basis of many logical systems and is fundamental in the field of computer science, mathematics, and philosophy.

Here are some key concepts of propositional logic:

1. **Proposition:** A statement that is either true or false. For example, "It is raining" is a proposition.

2. **Logical Connectives:** Symbols used to combine propositions. The main logical connectives are
   - AND (∧): True if both propositions are true.
   - OR (∨): True if at least one proposition is true.
   - NOT (~ , ┐): Inverts the truth value of the proposition.
   - IMPLIES (→): True if the first proposition implies the second.
   - IFF (↔): True if both propositions are logically equivalent.

3. **Truth Table:** A table used to determine the truth value of a proposition based on the truth values of its components.

4. **Tautology:** A proposition that is always true, regardless of the truth values of its components.

5. **Contradiction:** A proposition that is always false.

For example:

P: It is sunny today.

Q: The sun shines on the window.

R: The blinds are down.

(P→Q): If it is sunny today, then the sun shines on the window

(Q→R): If the sun shines on the window, the blinds are brought down.

(~R): The blinds are not yet down.

<u>Some Examples:</u>

- **Using AND (∧)**

- Let p: "I will go to the park."
- Let q: "It is sunny."

The compound proposition p∧q ("I will go to the park and it is sunny") is true if both p and q are true. If either p or q is false, the whole compound proposition is false.

- **Using OR (∨)**

- Let p: "I have a cat."
- Let q: "I have a dog."

The compound proposition p ∨ q ("I have a cat or I have a dog") is true if at least one of p or q is true. It is only false if both p and q are false.

- **Using NOT (~)**

- Let p: "I like spinach."

The proposition ~p ("I do not like spinach") inverts the truth value of p. If p is true (I like spinach), then ~p is false, and vice versa.

- **Using IMPLIES (→)**

- Let p: "I study."
- Let q: "I will pass the exam."

The compound proposition p→q ("If I study, then I will pass the exam") is true in all cases except when p is true and q is false.

- **Using IFF (↔)**

- Let p: "I have a driver's license."
- Let q: "I can legally drive."

The compound proposition p ↔ q ("I have a driver's license if and only if I can legally drive") is true if both p and q have the same truth value (either both are true or both are false).

- **Truth Table for p∨~q**

Let's consider p: "It is snowing," and q: "It is cold."

P	Q	~p	p∨~q
T	T	F	T
T	F	F	T
F	T	T	F
F	F	T	T

Note that a definition of truth is not assigned to this proposition; it can be either true or false in terms of binary logic. Propositions can also be combined to create compound propositions as shown below:

– $\sim(\sim p) = p$ double negation

– $p \land q = q \land p$ commutatively

$p \lor q = q \lor p$

– $(p \land q) \land R = p \land (q \land R)$ associativity

$(p \lor q) \lor R = p \lor (q \lor R)$

– $p \lor (q \land R) = (p \lor q) \land (p\ R)$ distributive

$p \land (q \lor R) = (p \land q) \lor (p \land R)$

– $\sim(p \land q) = \sim p \lor \sim q$ DE Morgan's law

$\sim(p \lor q) = \sim p \land \sim q$

– $p \leftrightarrow q = (p \rightarrow q) \land (q \rightarrow p) = (\sim p \lor q) \land (\sim q \lor p)$

Example: Prove that (P ∧ Q) is not equivalent to (P → Q); in other word prove (P∧ Q) ≢ (P → Q)

Sol :

P	Q	(P∧ Q)	(P → Q)	(P∧ Q) ≢ (P → Q)
T	T	T	T	T
T	F	F	F	T
F	T	F	T	F
F	F	F	T	F

Example: Represent the following knowledge using the propositional logic method.

"If it is sunny today, then the sun shines on the screen. If the sun shines on the screen, the blinds are brought down. The blinds are not down."

Sol:

P → Q.

Q → R.

~R

1.1.2 Predicate Logic

In the previous section, propositional logic was explored. One issue with this type of logic is that it is not very expressive. In this section, we will explore predicate calculus otherwise known as Predicate logic or First Order Predicate Logic (FOPL) is one of the oldest and most important knowledge representation schemata used in AI. Using FOPL, we can see both predicates and variables to add greater expressiveness as well as more generalization to our knowledge. In FOPL, knowledge is built up from constants (the object of the knowledge), a set of predicates (relationships between the knowledge) and some number of functions

(indirect references to other knowledge). We can say that predicate logic is a high-level human-oriented language for describing problems and problem-solving methods. **<u>The basic ingredient logic is the following</u>**:

- **Connectives:**

  ~ or ~ Represents negation.

  & or ∧ Represents conjunction or AND.

  | or Represents disjunction or OR.

  → Represents implication (if…then).

  ↔ Represents equivalence or if and only if (iff).

- **Quantifiers:**

  ∃ Existential quantifier (there exists).

  ∀ Universal quantifier (for all values).

- **Constants:**

  Fixed value terms belong to a given domain. Usually denoted by letters near the beginning of the English alphabet and number, e.g., a, b, D, 100, etc.

- **Variables:**

  Terms that can assume different values over a given domain. They are usually denoted by words and letters near the end of the English alphabet, e.g., x, y, z, etc.

- **Auxiliary symbols:**

  ), (, [ ], { } are used for punctuation.

- **Functions:**

  Function symbols defined over a domain (say D) map n element (n > 0) to a single element of the domain. Here n is called the rank or ary or degree of the function. Letters f, g, h and words such as age-of ( ), and cause-of ( ) represent functions. An n-ary function is written as f(t1, t2,…, tn) where t are terms defined over some domain. A 0-ary function is a constant.

– **Predicates:**

Predicates denote relations or functional mapping from the elements of domain D to the values true or false. Letters and words near the middle of the alphabet such as p, q, R, EQUAL, etc. are used to represent predicates. Like functions, predicates can have n (n>= 0) terms as arguments. A 0-ary predicates are referred to as atomic formulas or atoms. When we want to refer to an atomic formula or its negation, we use the worker literal. A predicate which has no variables is called a ground atom.

Examples: consider the following sentences and their predicate form:

- Caesar was a man.

Man (Caesar)

- Caesar was a ruler.

Ruler (Caesar)

- All Romans were either loyal to Caesar or hated him.

∀x (Roman (x) → Loyal to (x, Caesar) V Hate (x, Caesar))

- Marcus was born in 40 A.D.

Born (Marcus, 40)

- If it doesn't rain tomorrow, Tom will go to the mountains.

~rain (weather, tomorrow) →go(tom, mountains).

- All basketball players are tall.

∀ X (basketball _ player(X) →tall (X))

- Some people like fish.

∃X (people(X) ^ likes(X, fish)),

- If wishes were horses, beggars would ride.

equal(wishes, horses) → ride(beggars).

- Nobody likes taxies

~ ∃ X likes(X,taxi).

- All basketball players are tall.

∀x, y Play (x, y) ⋀ Game (y, basketball) → Tall (x)
∀x Play-basketball (x) → Tall (x)

- All dark streets are dangerous.

∀x Street (x) ⋀ Dark (x) → Dangerous (x)
∀x Dark-street (x) → Dangerous (x)

- All Policemen protect all people from crime.

∀x, y Policemen (x) ⋀ People (y) → Protect (x, y, crime)

- A burning stove is hot. If I put my hand on a burning stove, it will be hurt.

∀x Stove (x) ⋀ Burn (x) → Hot (x)
∀x, y Stove (x) ⋀ Burn (x) ⋀ Put (y, hand, x) → Hurt (y, hand)

- If I am in a quiet dark street and I see an old man then I will not worry about my safety.

∀x, y Street (x) ⋀ Quit-dark (x) ⋀ See (x, y) ⋀ Old-man (y) → ~ Worry (x, safety)

- Some students like AI.

∃x Student (x) ⋀ Like (x, AI)

- If one of two integers is positive and the other is negative then the product is negative.

∃x, y, z Int (x) ⋀ Int (y) ⋀ Pos(x) ⋀ Neg (y) ⋀ Pro (x, y, z) → Neg (z)

Example: Convert the following sentences into Predicate logic form:

"All people who are not poor and are smart are happy. Those people who read are not stupid. John can read and is wealthy. Happy people have exciting lives.

Solution:

∀ X (~ poor (X) ∧ smart (X) → happy (X)).

∀ Y (read (Y) → ~ stupid (Y)).

read (john) ∧ wealthy (john).

∀ Z (Happy (Z) → exciting (Z)).

1.1.3 Resolution Theorem Proving

Resolution is a technique for proving theorems in the predicate logic using the***re solution by refutation algorithm***. Their solution refutation proof procedure answers a query or deduces a new result by reducing these to clauses to a contradiction.

The **Resolution by Refutation Algorithm** includes the following steps:-

a) Convert the statements to **predicate calculus**(predicate logic).

b) Convert the statements from **predicate calculus** to **clause form**.

c) Add the negation of what is to be proved to the clause form.

d) Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause.

The statements that produced from **predicate calculus** method are nested and very complex to understand, so this will lead to more complexity in resolution stage ,therefore the following steps are used to convert the

Predicate calculus to **clause form**:-

1.Eliminate (→) by replacing each instance of the form

(P → Q)by expression **(~P∨Q)**

2.Reduce the scope of negation.

- $\sim(\sim a) \equiv a$
- $\sim(\forall X)\ b(X) \equiv \exists X \sim b(X)$
- $\sim(\exists X)\ b(X) \equiv \forall X \sim b(X)$
- $\sim(a \wedge b) \equiv \sim a \vee \sim b$
- $\sim(a \vee b) \equiv \sim a \wedge \sim b$

3. Standardize variables: rename all variables so that each quantifier has its own unique variable name. ***For example***,

-$\forall X\ a(X) \vee \forall X\ b(X) \equiv \forall X\ a(X) \vee \forall Y\ b(Y)$

4. **Move all quantifiers to the left** without changing their order.

***For example*,**

-$\forall X\ a(X) \vee \forall Y\ b(Y) \equiv \forall X\ \forall Y\ a(X) \vee b(Y)$

5. **Eliminate existential quantification** by using the equivalent function.

For example,

-$\forall X \exists Y\ mother(X,Y) \equiv \forall X\ mother(X,m(X))$

-$\forall X\ \forall Y\ \exists Z\ p(X,Y,Z) \equiv \forall X\ \forall Y\ p(X,Y,f(X,Y))$

6. **Remove universal quantification** symbols.

For example,

-$\forall X\ \forall Y\ p(X,Y, f(X,Y)) \equiv p(X,Y, f(X,Y))$

7.**Use the associative and distributive properties** to get a conjunction of disjunctions called **conjunctive Normal Form(CNF)**.

For example,

-$a \vee (b \vee c) \equiv (a \vee b) \vee c$

-a∧(b∧c)≡(a∧b)∧c

-a∨(b∧c)≡(a∨b)∧(a∨c)

-a∧(b∨c)≡(a∧b)∨(a∧c)

8. **Split each conjunct** into a separate clause. ***For example***,

-(~a(X)∨~ b(X)∨e(W)) ∧(~b(X)∨~d(X,f(X))∨e(W))

□~a(X)∨~ b(X)∨e(W)

□~b(X)∨~d(X,f(X))∨e(W)

9. **Standardize variables** again so that each clause contains variable names that do not occur in any other clause .***For example***,

-(~a(X)∨ ~ *b(X)*∨*e(W)*)∧(~*b(X)*∨~d(X,f(X))∨*e(W)*)

□ ~a(X)∨~ b(X)∨e(W)

□ ~b(Y)∨~d(Z,f(Z))∨e(V)

Example: Use the Resolution Algorithm for proving that John is happy with regard the following story

Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery. Is John happy?

A: Convert all sentences into Predicate logic form:

1. □X (pass(X,h.exam) ∧win(X,lottery) →happy (X)).
2. □X □Y(study(X) □lucky(X) →pass(X,Y)
3. ~ study(john) ∧lucky(john)
4. □X Lucky(X) → win(X, lottery)

happy (john) **Required to prove it**

B. Convert the statements from predicate calculus to clause form.

1. Remove (→)

•∀X ~(pass(X,ai_exam) ∧win(X,lottery))∨ happy(X)

•∀X ~(study(X)∨lucky(X)) ∨ ∀E pass(X,E)

• ~study(john) ∧lucky(john)

•∀X ~(lucky(X))∨ win(X,lottery)

2.Reduce ~

•∀X(~pass(X,ai_exam) ∨ ~ win(X,lottery)) ∨ happy(X)

•∀X ~) study(X)∧~lucky(X)) ∨ ∀E pass(X,E)

• ~study(john)∧lucky(john)

•∀X ~lucky(X)∨win(X,lottery)

3.Standardize Variables

•∀X(~pass(X,ai_exam)∨~win(X,lottery))∨ happy(X)

•∀Y ~) study(Y)∧~lucky(Y))∨ ∀E pass(Y,E)

• ~study(john)∧lucky(john)

•∀Z ~lucky(Z)∨win(Z,lottery)

4. Move all quantifiers to the left

•∀X(~pass(X,ai_exam)∨~win(X,lottery))∨ happy(X)

•∀Y ∀E ~) study(Y)∧~lucky(Y))∨ pass(Y,E)

• ~study(john)∧lucky(john)

•∀Z ~lucky(Z)∨win(Z,lottery)

5.Remove ∃ •Nothing to do here.

6.Remove ∀

- ~) pass(X,ai_exam) ∨ ~win(X,lottery))∨ happy(X)
- ~) study(Y)∧~lucky(Y)) ∨ pass(Y,E)
- ~study(john) ∧lucky(john)
- ~lucky(Z) ∨ win(Z,lottery)

7. CNF

- ~pass(X,ai_exam)∨~win(X,lottery)∨ happy(X)
- ~) study(Y)∧~lucky(Y))∨ pass(Y,E) ≡ **(a∧b(∨c ≡ c∨)a∧b)**

The second statement becomes: pass(Y,E)∨~study(Y)∧pass(Y,E)∨~lucky(Y)

- ~study(john)∧lucky(john)
- 8.Split ∧
- ~pass(X,ai_exam)∨~win(X,lottery)∨ happy(X)
- pass(Y,E)∨~study(Y)
- pass(Y,E)∨~lucky(Y)
- ~study(john)
- lucky(john)
- ~lucky(Z)∨win(Z,lottery)

9.Standardize Variables

- ~pass(X,ai_exam)∨~win(X,lottery)∨ happy(X)
- pass(Y,E)∨~study(Y)
- pass(M,G)∨~lucky(M)
- ~study(john)
- lucky(john)
- ~lucky(Z)∨win(Z,lottery)lucky(Z)∨win(Z,lottery)

C.Add the negation of what is to be proved to the clause form.

- ~ happy(john).

Add edit to the other six clauses and shown below:

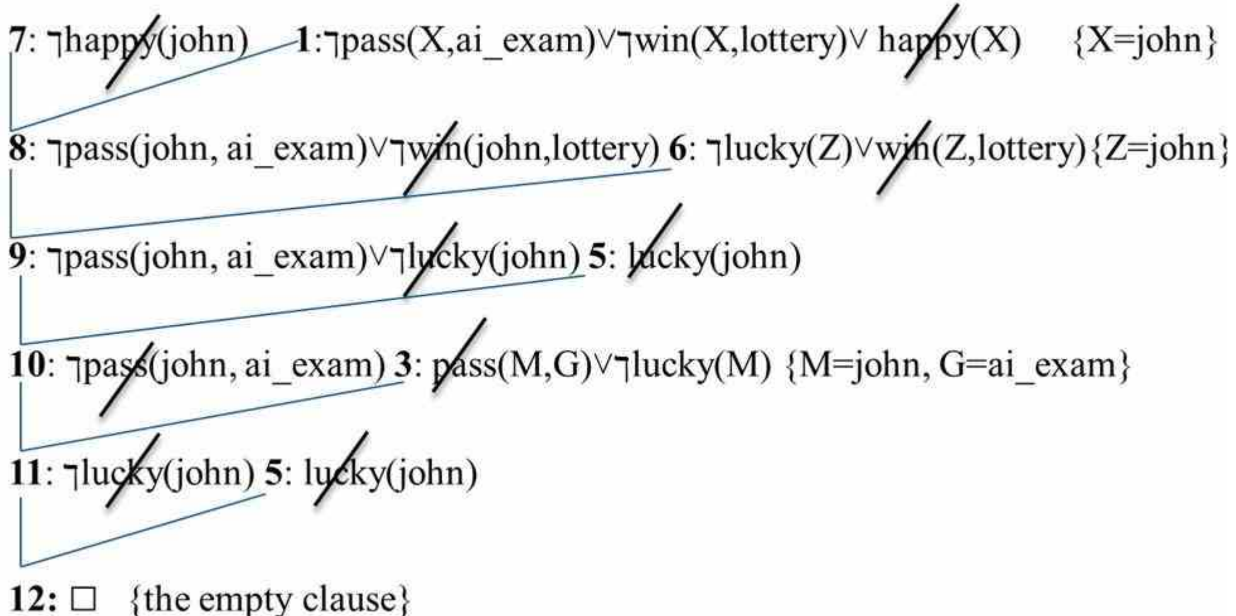
- ~pass(X,ai_exam)v~win(X,lottery)v happy(X)
- pass(Y,E)v~study(Y)
- pass(M,G)v~lucky(M)
- ~study(john)
- lucky(john)
- ~lucky(Z)vwin(Z,lottery)

- ~ happy(john).

D.Resolve the clauses to producing new clauses and producing a contradiction by generating the empty clause

There are two types of resolution ,the first one is *backward resolution* and the second is *forward resolution* .

1.Backward Resolution



2.Forward Resolution

1:¬pass(X,ai_exam)∨¬win(X,lottery)∨happy(X) 6: ¬lucky(Z)∨win(Z, lottery){Z=X}

8: ¬ pass(X,ai_exam)∨happy(X) ∨ ¬lucky(X) 5: lucky(john) {X=john}

9:¬ pass(john, ai_exam)∨happy(john) 3: pass(M,G)∨¬lucky(M) {M=john, G=ai_exam}

10: happy(john) ∨ ¬lucky(john) 5: lucky(john)

11 : happy(john) 7: ¬happy(john)

12: □ {the empty clause}

∴ **John is happy**

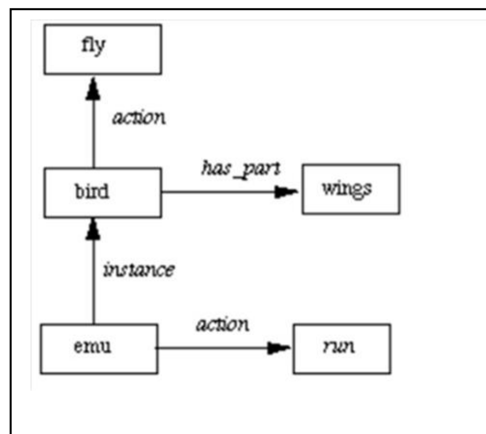
1.2Network Representation

1.2.1 Semantic Representation

A semantic network is an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world and arcs representing relationships between those objects. A semantic network represents knowledge as a graph with nodes corresponding to facts or concepts and arcs to relations or associations between concepts.

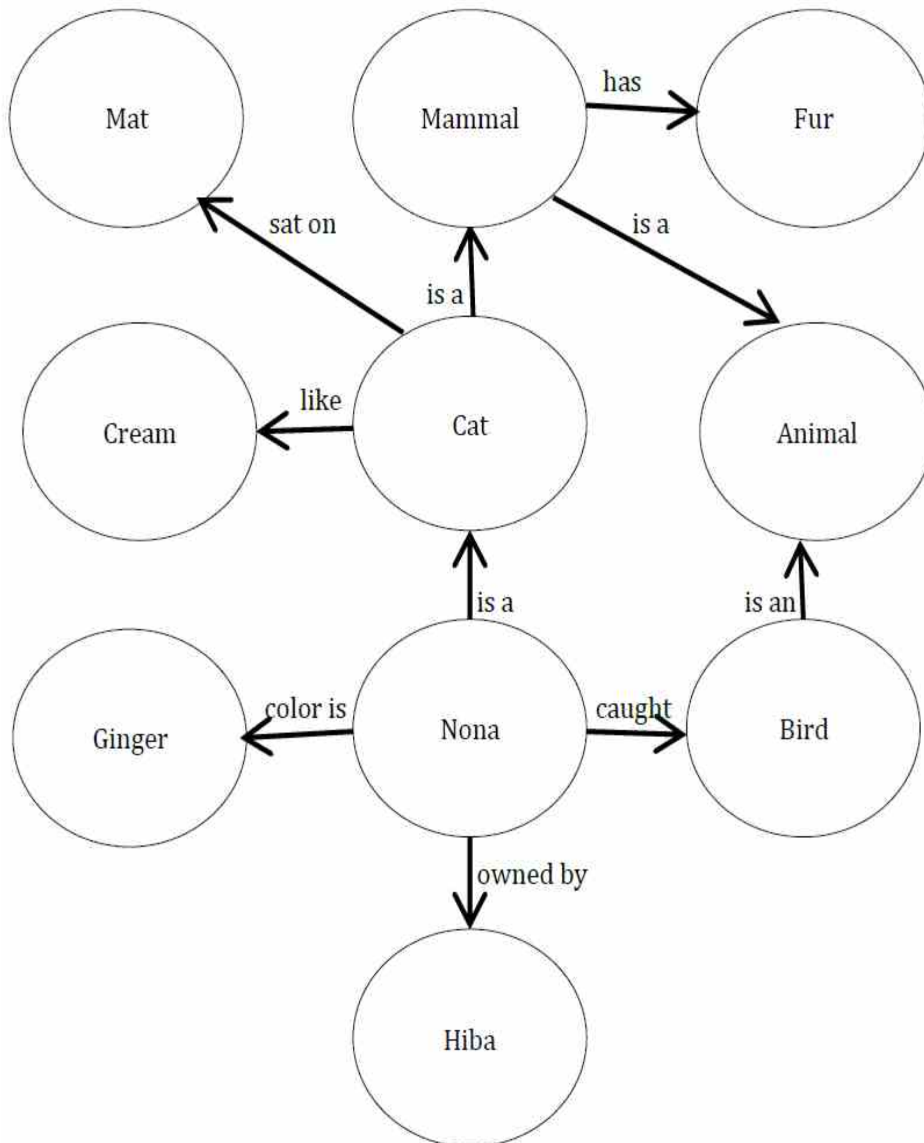
<u>**Example1,** represent the following facts using a semantic network:</u>

- Emus are birds.
- Typically birds fly and have wings.
- Emus run.

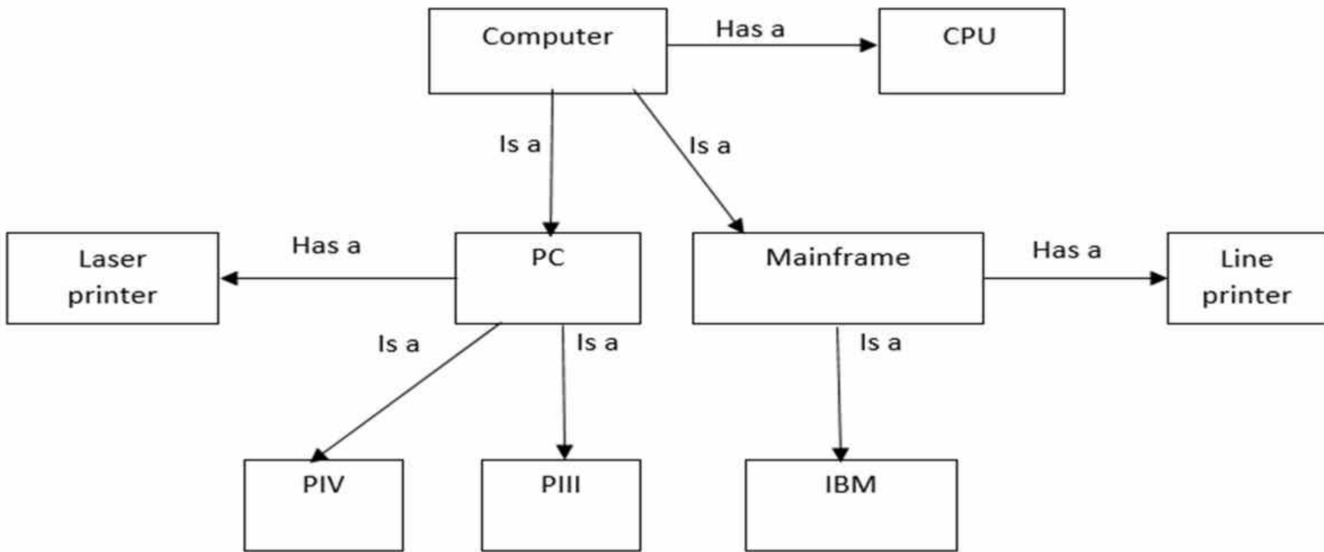


Example2, represent the following facts using a semantic network:

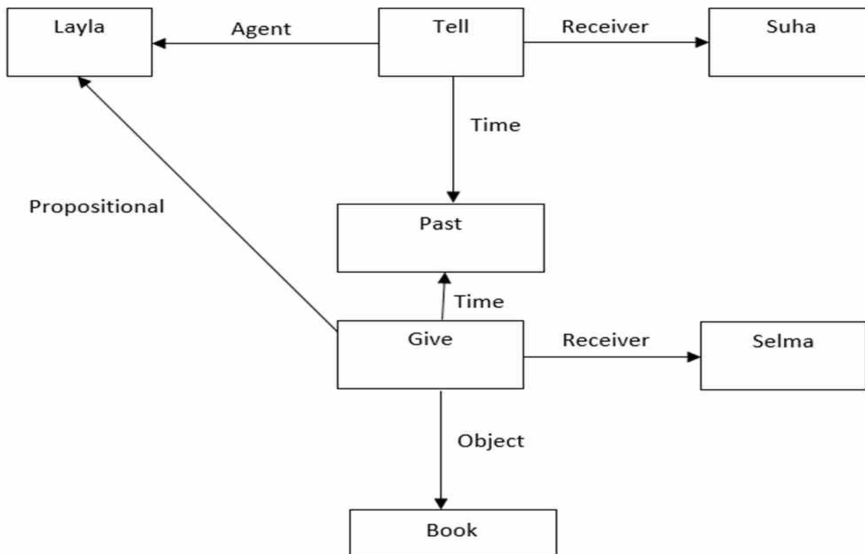
Nona is a cat. Nona caught a bird. Nona is owned by Hiba. Nona is ginger in colour. Cats like cream. The cat sat on the mat. A cat is a mammal. A bird is an animal. All mammals are animals. Mammals have fur.



Example 3 Computer has many parts like a CPU and the computer divided into two types, the first one is the mainframe and the second is the personal computer. Mainframe has a line printer with large sheet but the personal computer has a laser printer. IBM as example to the mainframe but PIII and PIV as examples to the personal computer



Example 4: Layla told suha,she gave Selma a book,

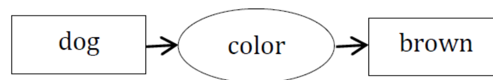


1.2.2Conceptual Representation

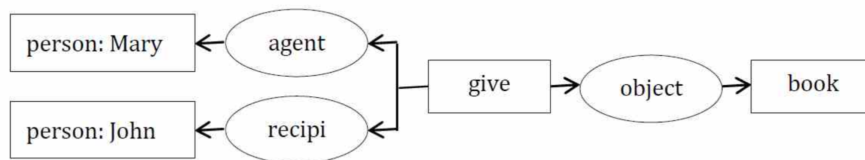
A conceptual graph is a finite, connected, bipartite graph. The nodes of the graph are either concepts or conceptual relations. Conceptual graphs do not label arcs; instead, the conceptual relation nodes represent the relation between concepts. Because conceptual graphs are bipartite; concepts can be arcs to conceptual relations and vice versa.

Examples, represent the following sentences using a conceptual graph:

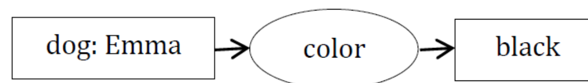
- The colour of the dog is brown.
-



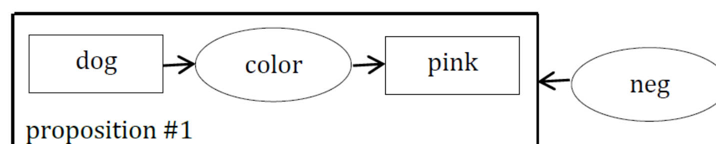
- Mary gave John the book.
-



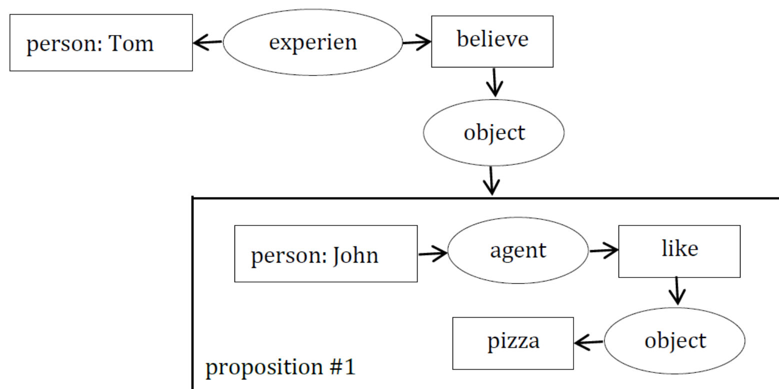
- The dog named Emma is black.



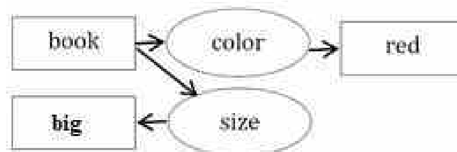
- There are no pink dogs.



– Tom believes that Jane likes pizza.

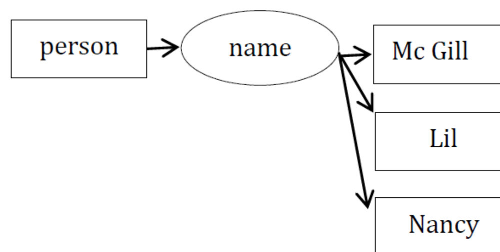


– The red book is big.



– There is a person called McGill, Lil and Nancy.

–



WORKSHEET 1

1. **Convert the following sentences into Predicate logic form:**
   John likes all kind of games. Football is game .chess is game too .anything anyone plays and is not killed by is games. Tom plays tennis and still alive .tom likes anything that john likes. Is John like chess?
2. **Given the following Predicate logic statements , prove ~ s(W):**
   (1) $\forall X\ [\ (\forall Y\ s(Y) \wedge v(X, Y)) \Rightarrow ((\exists Z\ \sim t(X, Z)) \wedge v(X, X))\ ]$

   (2) $\forall X \forall Y\ s(Y) \Rightarrow t(X,Y) \wedge v(X,Y)$

3. Represent the following sentences using Conceptual Graph method:

- John likes small cars.
-Mary gave Tom red book.
- John thinks that Mary gave the book to Tom.
John thinks that Mary gave the book to Tom in the classroom.

4.Represent the following sentences using a semantic network.

Frosty is a snowman. A snowman is made of snow. Snow is frozen water. It is slippery and soft. Snow is cold. Ice is also cold. It is also frozen water, but unlike snow, which is soft, ice is hard. Ice is clear in color."
Let’s say we have two propositions:

5.Let’s say we have two propositions:

p: "It is raining."
q: "The ground is wet."
Determine the truth value of the statement: "**If it is raining, then the ground is wet, or it is not raining."**

6. **Represent the following sentences using a semantic network and** conceptual graph

Teachers and student is people. Teacher work in the university .they has lectures, subjects and good information .john is teacher .he teach artificial intelligence. Students learn in university .tom is student in computer sciences department .computer science is department in university.it consist of six branches SW,IS,AI,DS,NW and MM.

Part 2: "Search Techniques"

1. Search Algorithms

Search is an important aspect of AI. Search can be defined as a problem-solving technique that enumerates a problem space from an initial position in search of a goal position (or solution). The manner in which the problem space is searched is defined by the search algorithm or strategy. As search strategies offer different ways to enumerate the search space. Ideally, the search algorithm selected is one whose characteristics match that of the problem at hand.

State Space Search

State Space Search is a collection of several states with appropriate connections (**links**) between them. Any problem can be represented as such a space search to be solved by applying some rules with technical strategy according to the suitable intelligent search algorithm. To locate a solution, state space search entails methodically going through every potential state for an issue. This approach can be used to solve a variety of AI issues, including path finding, solving puzzles, playing games, and more. The fundamental concept is to visualize the issue as a graph with nodes standing in for states and edges for transitions.

To provide a formal description of a problem must do the following operations:

1. Define space search (**state space**) that contains all possible states.
2. Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the initial states.
3. Specify a set of rules that describe the actions available.
4. Specify one or more states that would be acceptable as solutions to the problem. These states are called **goal states**.
5. Determine a suitable **search strategy** to reach the goal.

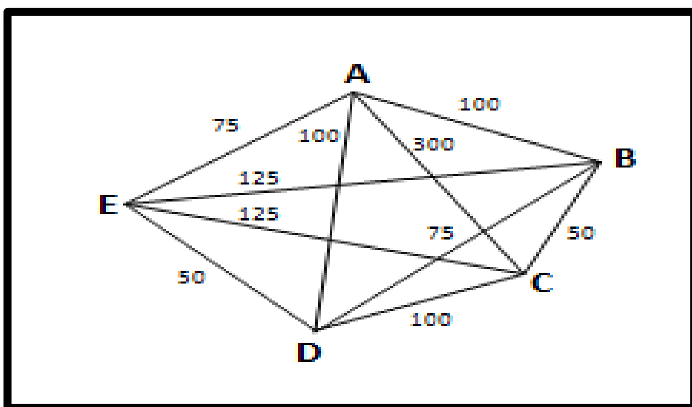
Some common terms in the searching issues

- **Search Tree:** is a tree in which the root node is the start state and has a reachable set of children.
- **Search Node:** is a node in the search tree.
- **Goal:** is a state that an agent is trying to reach.
- **Action:** is something (operators, possible moves, rules) that an agent can choose to do.
- **Branching Factor:** is the number of actions available to the agent

Traveling Salesman Problem (TSP)

The TSP concept depends on finding a path for a specified number of cities (visiting all cities only once and returning to the city that started with) where the distance of the path is optimized by finding the shortest path with minimized cost.

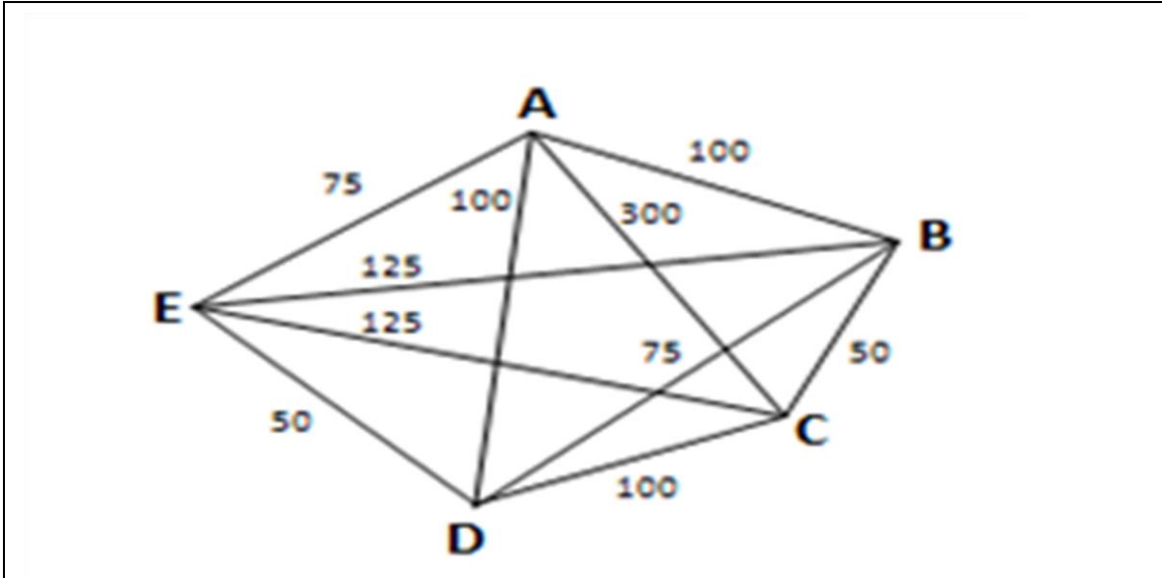
Example: The below figure shows a full connected graph, (A,B,C,etc) are cites and the numbers associated with the links are the distances between the cities. Starting at A, find the shortest path through all the cities, visiting each city exactly once returning to A.

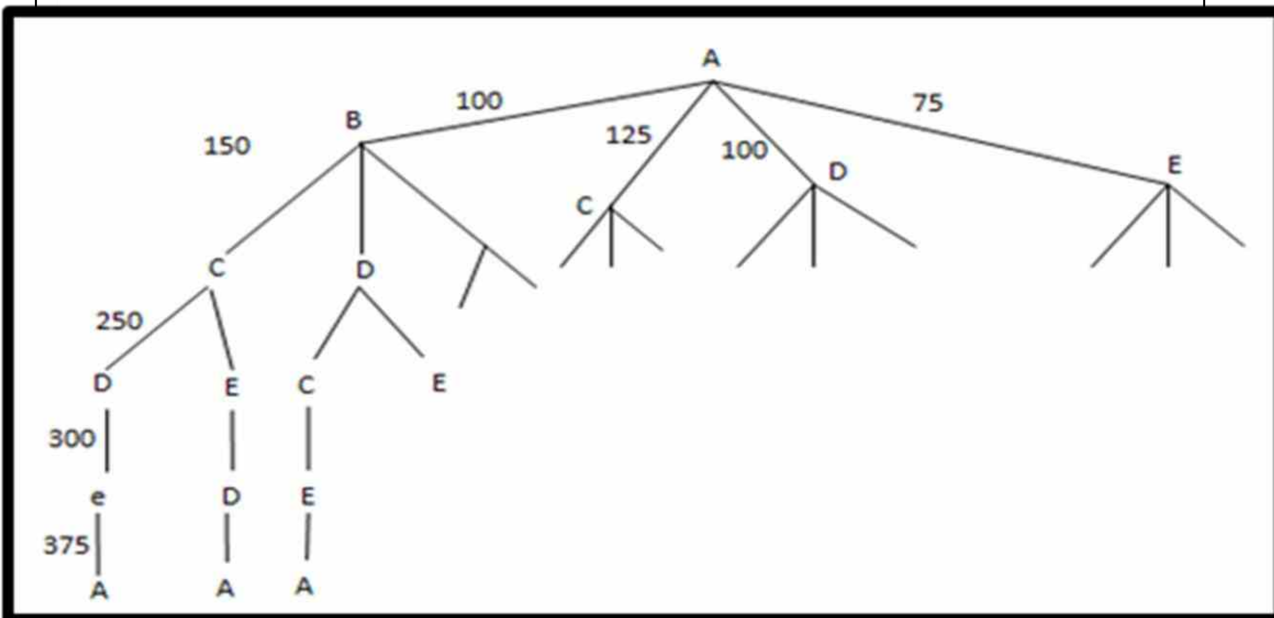


"An instance of traveling Salesman Problem"

The complexity of exhaustive search in the traveling Salesman is **(N-1)**! , where N is the No. of cities in the graph. There are several techniques that reduce the search complexity:

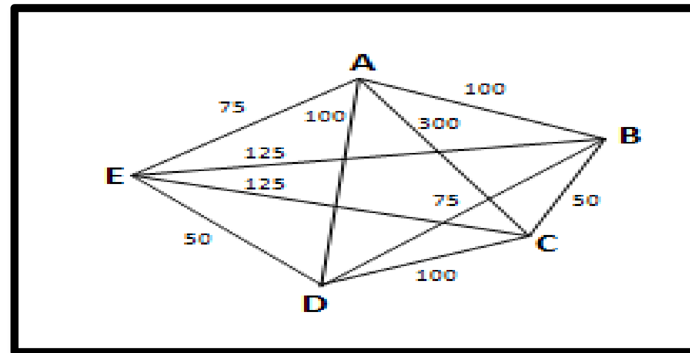
1- Branch and Bound Algorithm: Generate one path at a time, keeping track of the best circuit so far. Use the best circuit so far as a bound of future branches of the search. Figure below illustrate branch and bound algorithm.





A B C D E A= 375 A B C
E D A= 425
A B D C E A= 474 ……………………

2. Nearest Neighbor Heuristic: At each stage of the circuit, go to the nearest unvisited city. This strategy reduces the complexity to N, so it is highly efficient, but it is not guaranteed to find the shortest path, as the following example:



Distance of nearest neighbor path is A E D B C A=550
Is not the shortest path, the comparatively high distance of arc (C, A) defeated the heuristic.

2. Search Techniques Types

There are different types of searches used in artificial intelligence; They are commonly divided into three categories. One of them is uninformed(blind , the other is informed or Heuristic searches and Random searches

1- ***Blind Search*** is a technique to find the goal without any additional information that helps to infer the goal, with this type there is no consideration with process time or memory capacity.

2- ***Heuristic Search*** always has an evaluating function called the heuristic function which guides and controls the behavior of the search algorithm to reach the goal with minimum cost, time and memory space.

3- ***Random Search*** is a special type of search in which it begins with the initial population that is generated randomly and the search algorithm will be the responsible for generating the new population based on some operations according to a special type function called fitness function.

2.1 Blind Search

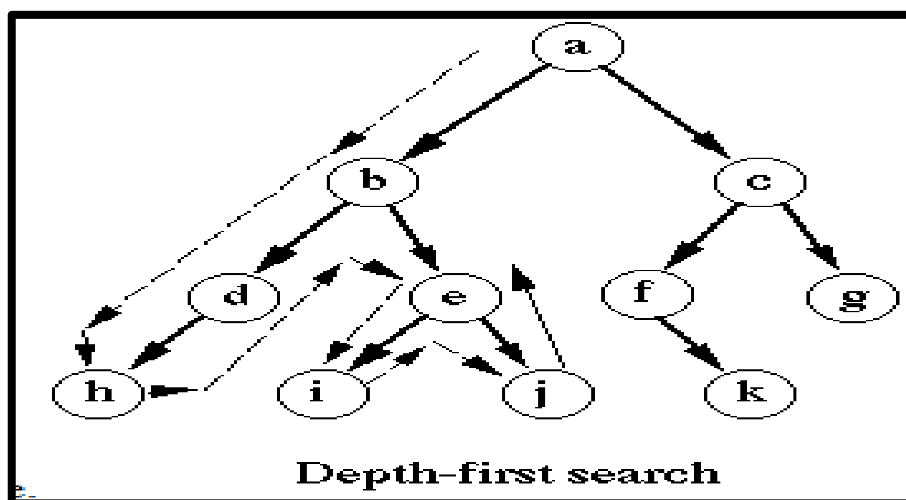
Blind Search, also known as **Uninformed Search**, refers to search algorithms that explore the problem space without any domain-specific knowledge (heuristics) to guide the search. These algorithms only rely on the problem's structure and the state space to find a solution. They do not have any additional information about how close a state is to the goal state.

In blind search, the algorithm expands nodes without any sense of direction or focus towards the goal. As a result, it may end up exploring a large portion of the search space, which can be inefficient for large or complex problems.

There are several types of blind search algorithms, each with different strategies for exploring the state space. Below are the most commonly used ones:

2.1.1 Depth–First–Search Algorithm

In a depth-first search, when a state is examined, all of its children and their descendants are examined before any of its siblings. The depth- first search goes deeper into the search space whenever this is possible only when no further descendants of a state can found owe it.

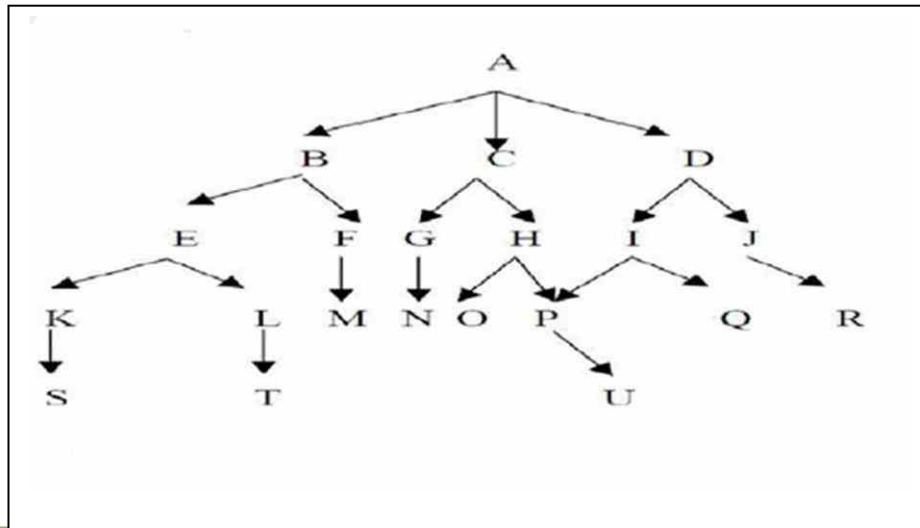


Depth-first search

For Example :

Start State: A

Goal State: U



SOL:

[open]	[closed]
[A]	[]
[B,C,D]	[A]
[E,F,C,D]	[B,A]
[K,L,F,C,D]	[E,B,A]
[S,L, F,C,D]	[K,E,B,A]
[L, F,C,D]	[S,K,E,B,A]
[T, F,C,D]	[L,S,K,E,B,A]
[F,C,D]	[T,L,S,K,E,B,A]
[M,C,D]	[F,T,L,S,K,E,B,A]
[C,D]	[M,F,T,L,S,K,E,B,A]
[G,H,D]	[C,M,F,T,L,S,K,E,B,A]
[N,H,D]	[G,C,M,F,T,L,S,K,E,B,A]
[H,D]	[N,G,C,M,F,T,L,S,K,E,B,A]
[O,P,D]	[H,N,G,C,M,F,T,L,S,K,E,B,A]
[P,D]	[O,H,N,G,C,M,F,T,L,S,K,E,B,A]
[U,D]	[P,O,H,N,G,C,M,F,T,L,S,K,E,B,A]
[D]	[U,P,O,H,N,G,C,M,F,T,L,S,K,E,B,A]

Final Path: A→B→E→K→S→L→T→F→M→C→G→N→H→O→P→U

Depth – First – Search Algorithm

Begin

Open: = [start];

Closed: = [ ];

While open ≠ [ ] do

Remove leftmost state from open, call it x; If x is a goal then return (success)

Else begin

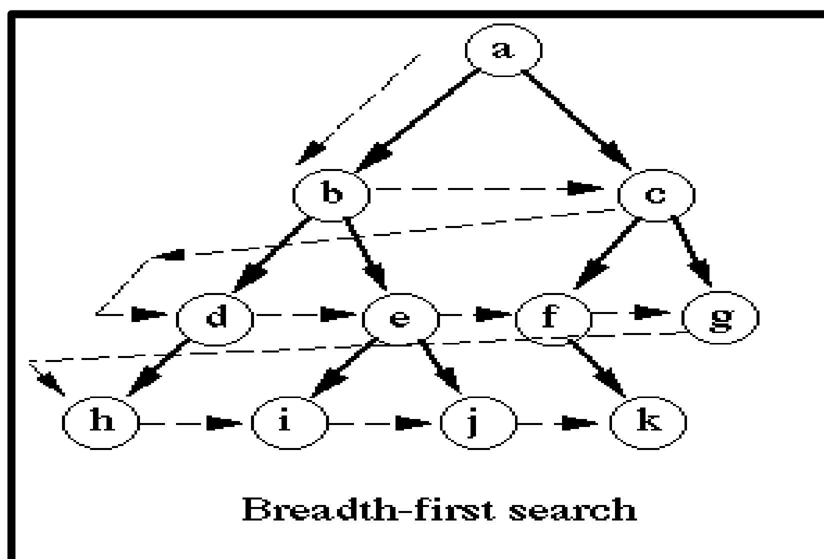
Generate children of x; Put x on closed;

Eliminate children of x on open or closed; put remaining children on left end of open End;

Return (failure) End

2.1.2 Breadth – First – Search Algorithm

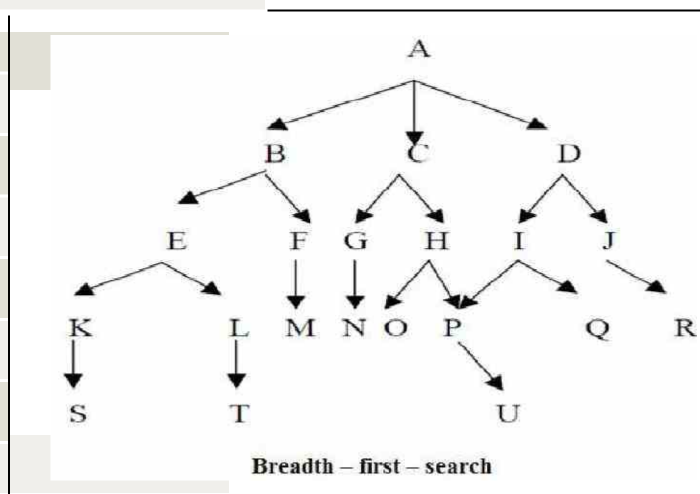
In breadth-first search, when a state (node) is examined, all of its siblings are examined before any of its children. The space is searched level-by-level, proceeding all the way across one level before doing down to the next level.



Breadth-first search

For Example Start State: A Goal State: U

[open]	[closed]
[A]	[]
[B,C,D]	[A]
[C,D,E,F]	[B,A]
[D,E,F,G,H]	[C,B,A]
[E,F,G,H,I,J]	[D,C,B,A]
[F,G,H,I,J,K,L]	[E,D,C,B,A]
[G,H,I,J,K,L,M]	[F,E,D,C,B,A]
[H,I,J,K,L,M,N]	[G,F,E,D,C,B,A]
[I,J,K,L,M,N,O,P]	[H,G,F,E,D,C,B,A]
[J,K,L,M,N,O,P,Q]	[I,H,G,F,E,D,C,B,A]
[K,L,M,N,O,P,Q,R]	[J,I,H,G,F,E,D,C,B,A]
[L,M,N,O,P,Q,R,S]	[K,J,I,H,G,F,E,D,C,B,A]
[M,N,O,P,Q,R,S,T]	[L,K,J,I,H,G,F,E,D,C,B,A]
[N,O,P,Q,R,S,T]	[M,L,K,J,I,H,G,F,E,D,C,B,A]
[O,P,Q,R,S,T]	[N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[P,Q, R,S,T]	[O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[Q, R,S,T,U]	[P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[R,S,T,U]	[Q, P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[S,T,U]	[R,Q, P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[T,U]	[S, R,Q, P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[U] **Goal**	[T, S, R,Q, P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]
[]	[U,T, S, R,Q, P, O, N,M,L,K,J,I,H,G,F,E,D,C,B,A]



Breadth – first – search

Final-Path:

A→B→C→D→E→F→G→H→I→J→K→L→M→N→O→P→Q→R→S→T→U

<u>Breadth – First – Search Algorithm</u>

Begin

Open: = [start]; Closed: = [ ]; While open ≠[ ] do Begin

Remove left most state from open, call it x; If x is a goal then return (success)

Else

Begin

Generate children of x; Put x on closed;

Eliminate children of x on open or closed; Put remaining children on right end of open End

End

Return (failure) End.

2.2 heuristic Search

The heuristic search strategies (also called Informed Search). A heuristic is a method that might not always find the best solution but is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency. Heuristic search is useful in solving problems which:-

- Could not be solved any other way.
- Solution takes an infinite time or very long time to compute.

<u>There are several methods for heuristic search, from these are:-</u>

1- Hill Climbing.

2- Best-First Search.

3- A algorithm.

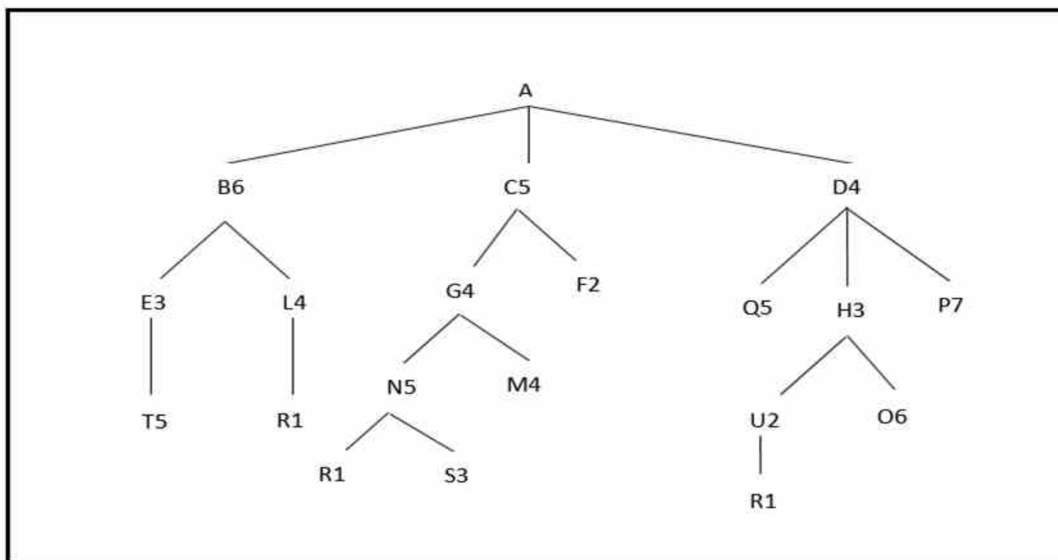
4- A* algorithm.

2.2.1 Hill Climbing

The idea here is that you don't keep the big list of states around. You just keep track of the one state you are considering, and the path that got you there from the initial state. At every state, you choose the state leads you closer to the goal according to the **heuristic estimate,** and continue from there.

The name "**Hill Climbing**" comes from the idea that you are trying to find the top of a hill, and you go in the direction that is up from wherever you are. This technique often works, but since it only uses local information.

Example 1: Search for **R** with local minima from **A**



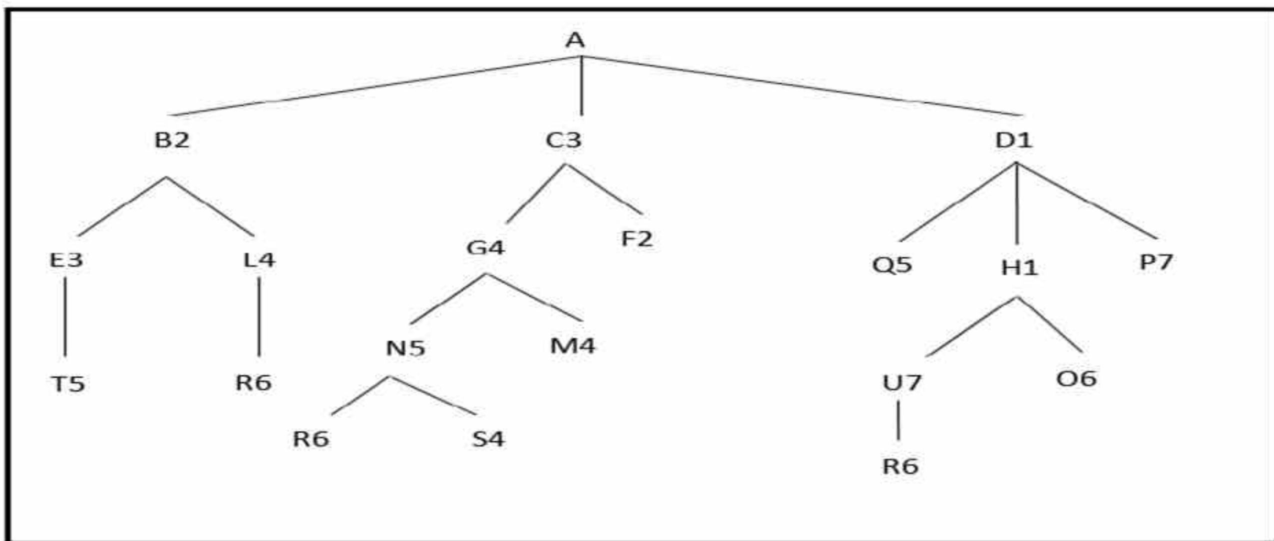
Sol:

Open	**close**	**P**
[A]	[]	A
[D4,C5,B6]	[A]	D4
[H3,Q5,P7]	[D4,A]	H3

[U2,O6]	[H3,D4,A]	U2
[R1]	[U2,H3,D4,A]	R1
[]	[R1,U2,H3,D4,A]	

Final Path: A→D4→H3→U2→R1

Example 2: Search for **R** with local maxima from **A**.



Sol:

Open	**close**	**P**
[A]	[]	A
[C3,B2,D1]	[A]	C3
[G4,F2]	[C3,A]	G4
[N5, M4]	[G4,C3,A]	N5
[R6,S4]	[N5,G4,C3,A]	R6
[S4]	[R6,N5,G4,C3,A]	

Final Path: A→C3→G4→N5→R6

Hill Climbing Search algorithm

```
Begin
Open: = [Initial state];            %initialize
Closed: = [ ];
CS= initial state;
Path= [initial state];
Stop= FALSE;
While open <> [ ] do                %states remain
Begin
   If CS=goal then return path
          Generate all children of CS and put them into open;
  If open= [ ] then
         Stop= TRUE
  Else Begin X= CS;
 For each state Y in open do Begin
        Compute the heuristic value of y (h(Y));
 If Y is better than X then
       X=Y
 End;
 If X is better than CS then
    CS=X
 Else
   Stop= TRUE;
```

End;

End;

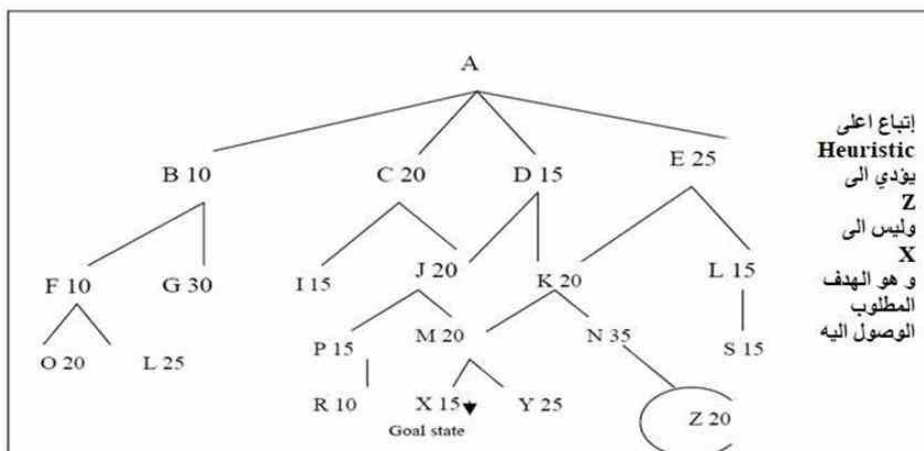
Return (FAIL); **%open is empty**

End.

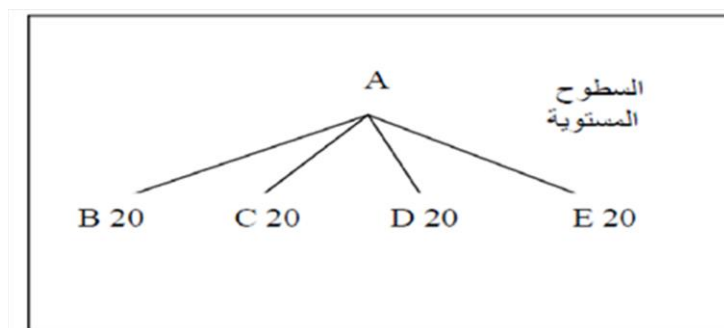
Hill Climbing Problems

Hill climbing may fail due to one or more of the following reasons:-

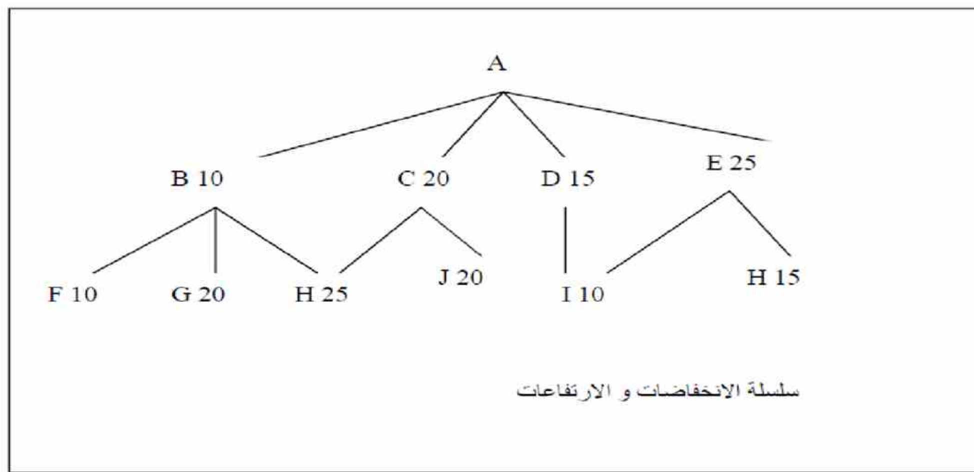
1. **A local maxima** : Is a state that is better than all of its neighbors but is not better than some other states.



2. **Plateau:** Is a flat area of the search space in which a number of states have the same best value, in plateau it's not possible to determine the best direction in which to move.



3. **A ridge:** Is an eas, but that cannot be trav



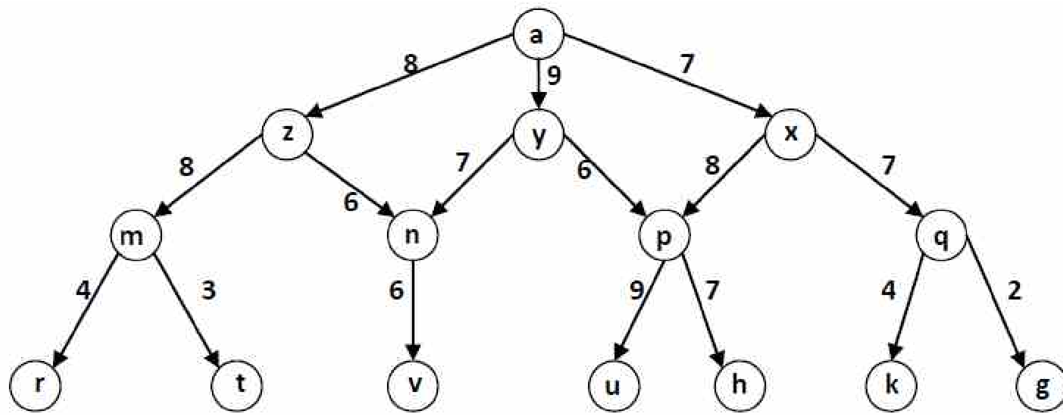
2.2.2 Best-First Search

Which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successors. Then the next step begins.

<u>For each state $f(n) = h(n)$ where $h(n)$ is the heuristic function that computes the heuristic value for each state n.</u>

For example:



Find the path from **a** to **k** using Best first Search algorithm.

Sol:

Open	Closed
[a]	[]
[z8, y9, x7]	[a]
[x7, z8, y9]	[a]
[p8, q7, z8, y9]	[a, x7]
[q7, p8, z8, y9]	[a, x7]
[k4, g2, p8, z8, y9]	[a, x7, q7]
[g2, k4, p8, z8, y9]	[a, x7, q7]
[k4, p8, z8, y9]	[a, x7, q7, g2]

The goal (k) is found

Best-First Search algoritm

Begin

Open: = [Initial state]; **%initialize**

Closed: = [ ];

While open <> [ ] do **%states remain**

Begin

Remove leftmost state from open, call it **X**;

If X = goal then return the path from initial to **X**

Else

Begin

Generate children of **X**;

For each child of **X** do

Case

The child is not on open or closed;

Begin

Assign the child a heuristic value;

Add the child to open

End;

The child is already on open;

If the child was reached by a shorter path Then give the state on open the shorter path The child is already on closed;

If the child was reached by a shorter path then

Begin

Remove the state from closed;

Add the child to open

End;

End; **%case**

Put **X** on closed;

Re-order states on open by heuristic merit (best leftmost)

End;

Return **FAIL** **%open is empty**

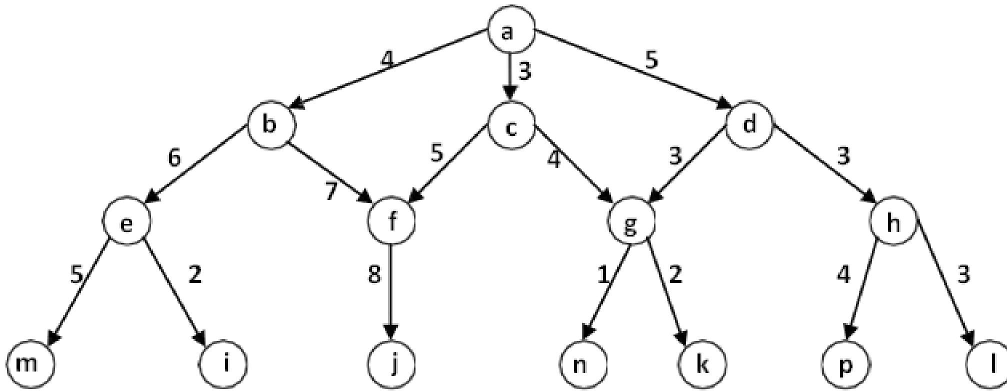
End.

2.2.3 A-search algorithm.

A algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the initial state and the current state and is denoted by g(n). A notice will be mentioned here that the same steps that are used in the best first search are used in an A algorithm but in addition to the g(n) as follow;

f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between initial state to current state n.

Example:



Find the path from **a** to **k** using A-search algorithm

Sol:

Open	Closed
[a]	[]
[b4, c3, d5]	[a]
[b4+1, c3+1, d5+1]	[a]
[c4, b5, d6]	[a]
[f5, g4, b5, d6]	[a, c4]
[f5+2, g4+2, b5, d6]	[a, c4]
[f7, g6, b5, d6]	[a, c4]
[b5, g6, d6,, f7]	[a, c4]
[e6, f7, g6, d6, f7]	[a, c4, b5]
[e6+2, f7+2, g6, d6, f7]	[a, c4, b5]
[g6, d6, f7, e8, f9]	[a, c4, b5]
[n1, k2, d6, f7, e8, f9]	[a, c4, b5, g6]
[n1+3, k2+3, d6, f7, e8, f9]	[a, c4, b5, g6]
[n4, k5, d6, f7, e8, f9]	[a, c4, b5, g6]
[k5, d6, f7, e8, f9]	[a, c4, b5, g6, n4]
Stop the goal (k) is found	

2.2.4 A*-search algorithm.

A* algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the current state and the goal state and is denoted by g(n).

f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between current state n to goal state.

A* Search Algorithm

Begin

Open: = [Initial state]; **%initialize**

Closed: = [ ];

While open <> [ ] do **%states remain**

Begin

Remove leftmost state from open, call it **X**;

If X = goal then return the path from initial to **X**

Else Begin

Generate children of **X**; For each child of **X** do Begin

Add the distance between current state to goal state to the heuristic value for each child **%make the g(n)**

Case

The child is not on open or closed;

Begin

Assign the child a heuristic value; Add the child to open

End;

The child is already on open;

If the child was reached by a shorter path Then give the state on open the shorter path The child is already on closed;

If the child was reached by a shorter path then Begin

Remove the state from closed; Add the child to open

End;

End; **%case**

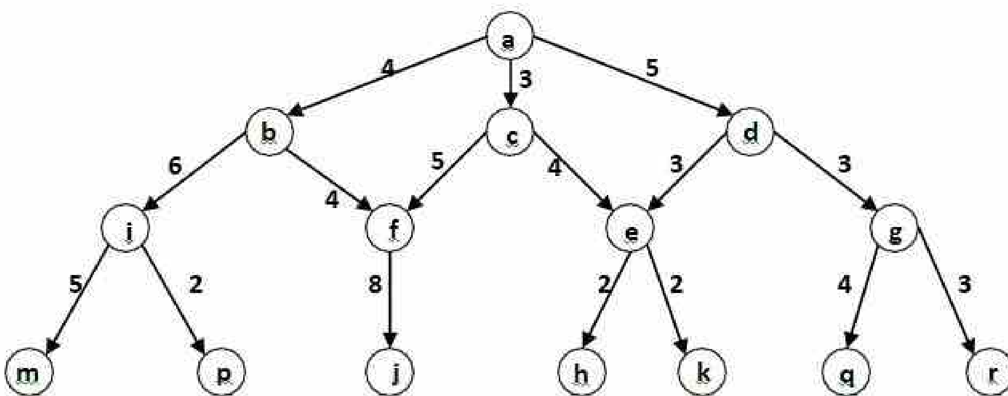
Put **X** on closed;

Re-order states on open by heuristic merit (best leftmost) End;

Return **FAIL** **%open is empty**

End.

Example:



Find the path from **a** to **k** using A*-search algorithm

Sol:

Open	**Closed**
[a]	[ ]
[b4, c3, d5]	[a]
[b4+4, c3+2, d5+2]	[a]
[c5, d7, b8]	[a]
[f5, e4, d7, b8]	[a, c5]
[f5+3, e4+1, d7, b8]	[a, c5]
[e5, d7, f8, b8]	[a, c5]
[h2, k2, d7, f8, b8]	[a, c5, e5]
[h2+2, k2+0, d7, f8, b8]	[a, c5, e5]
[k2, h4, d7, f8, b8]	[a, c5, e5]

Stop, the goal (k) is found

A Comparison between Heuristic Search and Blind Search

	Blind Search	**Heuristic Search**
1	In term of complexity: it is less complex.	In term of complexity: it is more complex.
2	In term of memory capacity: usually need more memory capacity.	In term of memory capacity: usually need less memory capacity.
3	In term of run time consuming: usually consumes more run time.	In term of run time consuming: usually consumes less run time.
4	Usually does not find the optimal solution path.	Usually finds the optimal solution path or nearly the optimal solution path.
5	It does not have a guider in search behavior.	It has a guider in search behavior (Heuristic Function).
6	It is not efficient in game playing.	It is efficient in game playing such as Minmax or Alpha-Beta procedures.

Using Heuristic in Games

The sliding-tile puzzle consists of three black tiles, three white tiles, and an empty space in the configuration shown in Figure (1). The puzzle has two legal moves with associated costs:

- A tile may move into an adjacent empty location. This has a cost of 1.
- A tile can hop over one or two other tiles into the empty' position, this has a cost equal to the number of tiles jumped over.

The goal is to have all the white tiles to the left of all the black tiles. The position of the blank is not important.

a. Analyze the state space with respect to complexity and looping.

b. Propose a heuristic for solving this problem and analyze it.

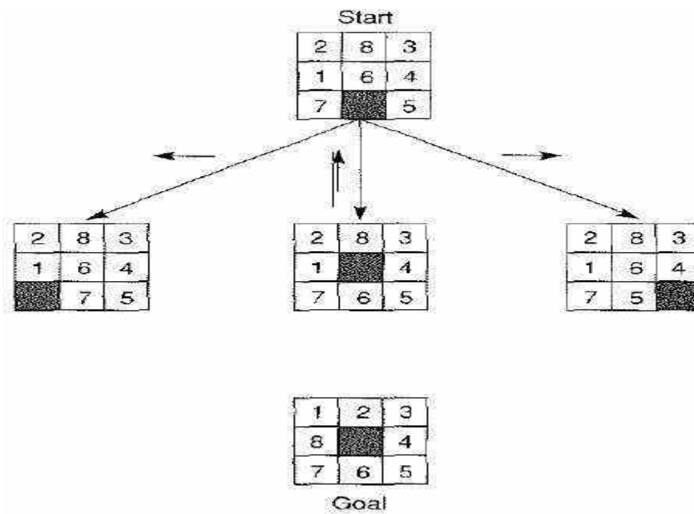
the sliding block puzzle

The 8-puzzle Problem

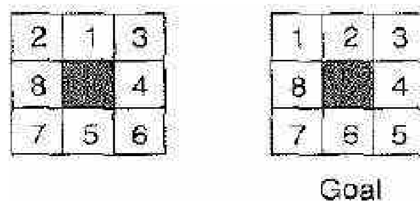
We now evaluate the performance of several different heuristics for solving the 8-puzzle. Shows a start and goal state for the 8- puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when it is compared with the goal. This is intuitively appealing, because it would seem that, all else being equal; the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next. However, this heuristic does not use all of the information available in a board configuration; because it does not take into account the distance the tiles must be moved.

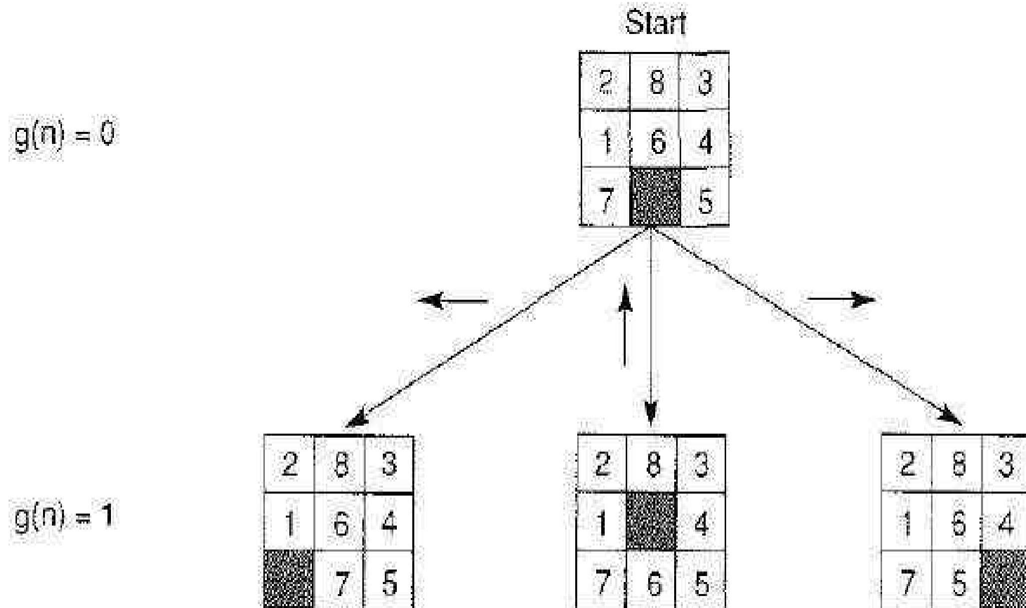
A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state. Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in apposite locations, it takes (many) more than two moves to put them back in place, as the tiles must "go around" each other



The start state, first moves, and goal state for an example 8-puzzle.



An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.



Values of f(n) for each state, 6 4 6

where:
f(n) = g(n) + h(n),
g(n) = actual distance from n to the start state, and
h(n) = number of tiles out of place.

1	2	3
8		4
7	6	5

Goal

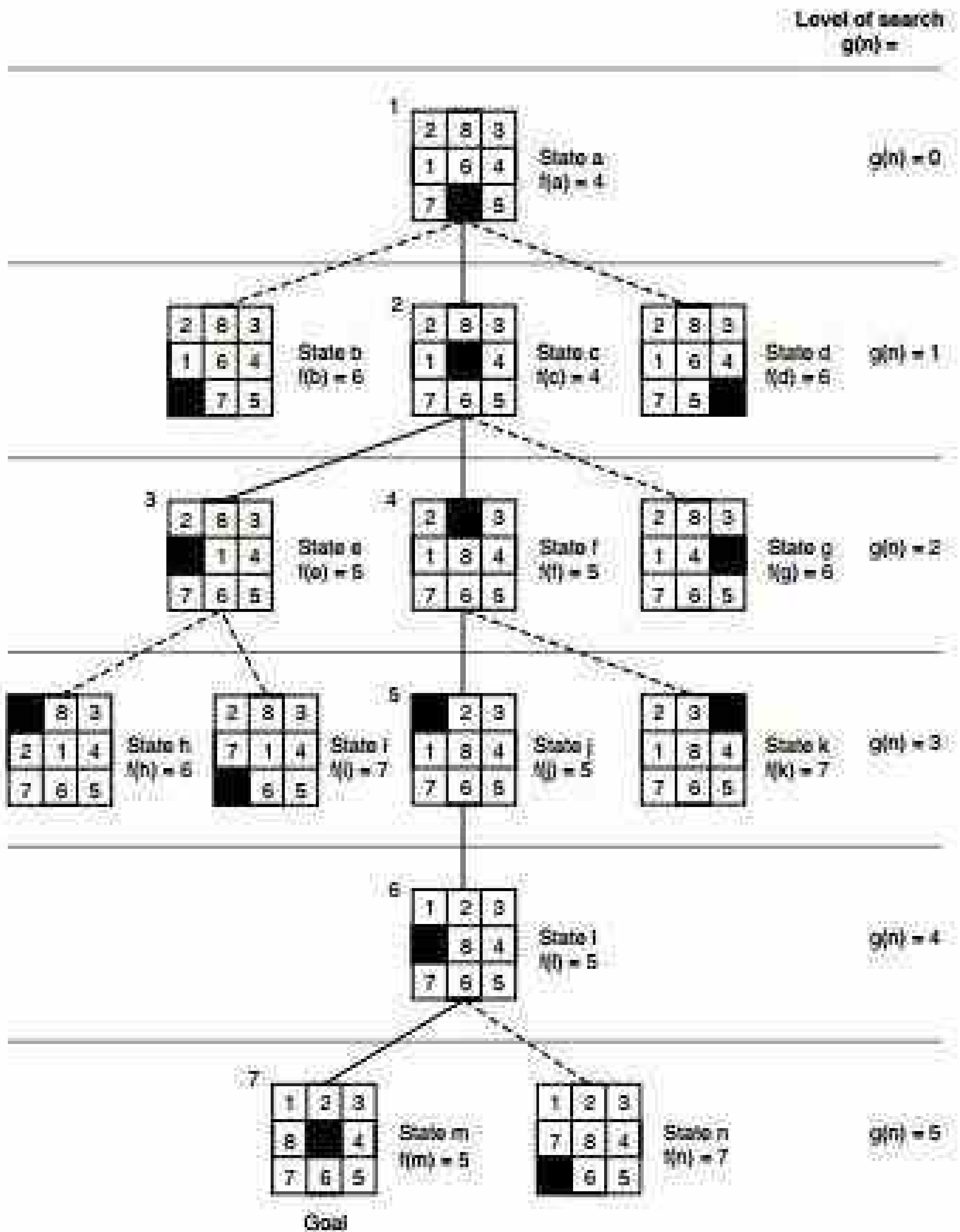
the 8-puzzle problem solving with heuristic values

For the 8-puzzle Grid

There is one center location. There are four corners location. There are four sides location.

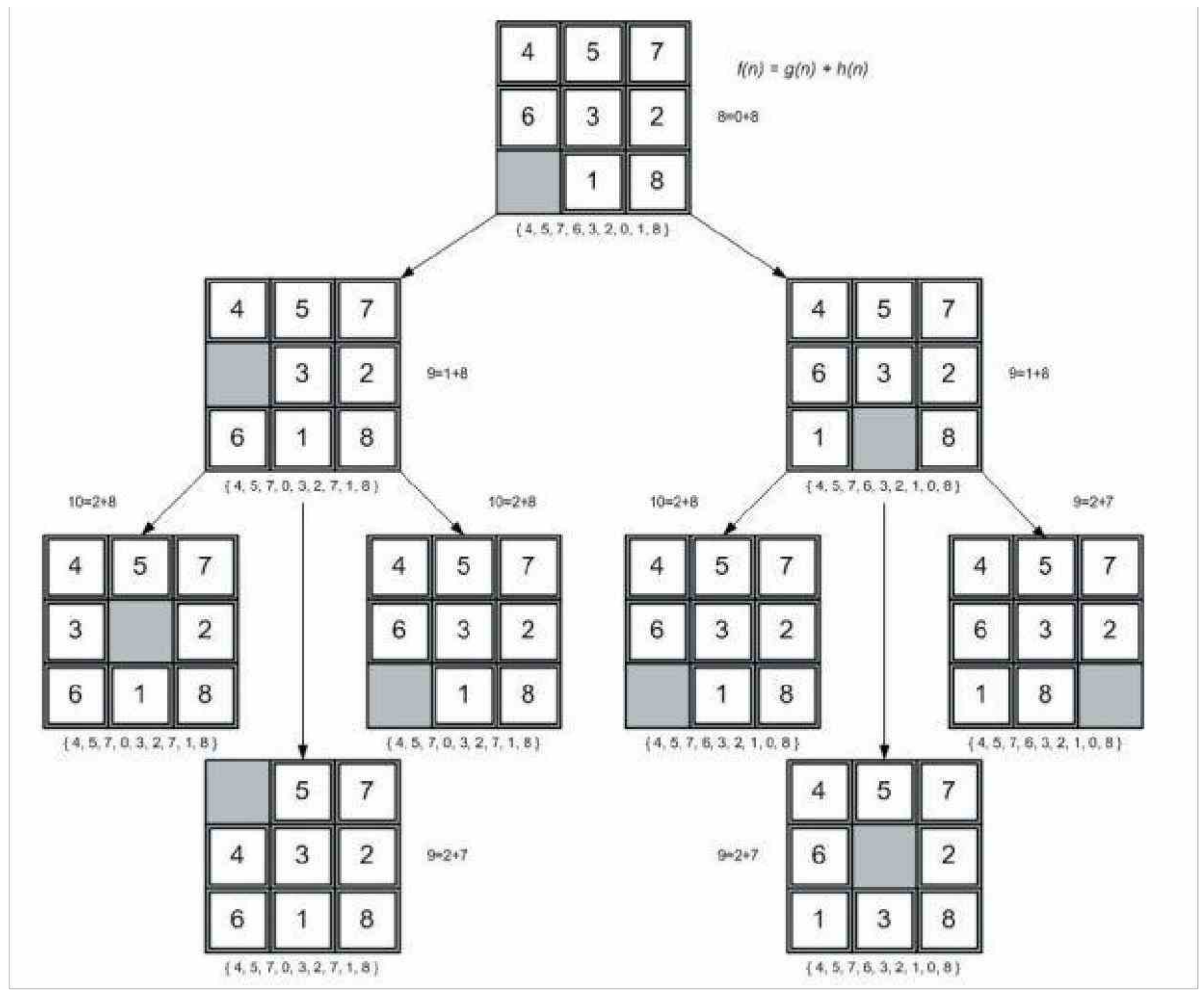
Possible Moves

- *When the space position is in the center of the grid, possible moves = 4.*
- *When the space position is in the corner of the grid, possible moves = 2*
- *When the space position is in the side of the grid, possible moves = 3.*



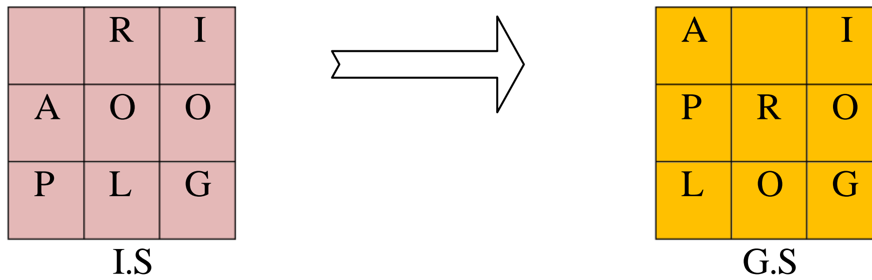
the 8-puzzle problem solved by A-search algorithm

Another Examples of 8-Puzzle Problem

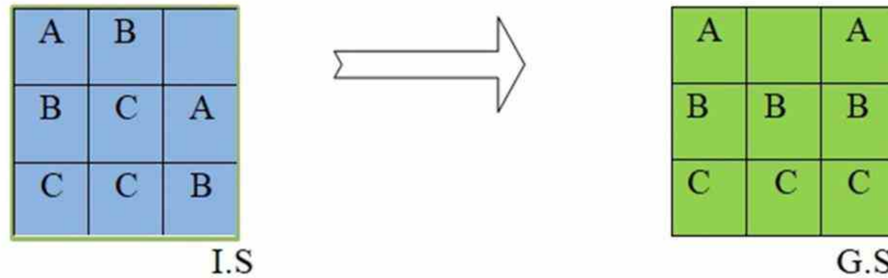


WORKSHEET 2

1. Consider the following **8-puzzle** problem then draw the problem state space to find the goal using A-search algorithm (or Best first or Hill climbing) then list the *solution path.*

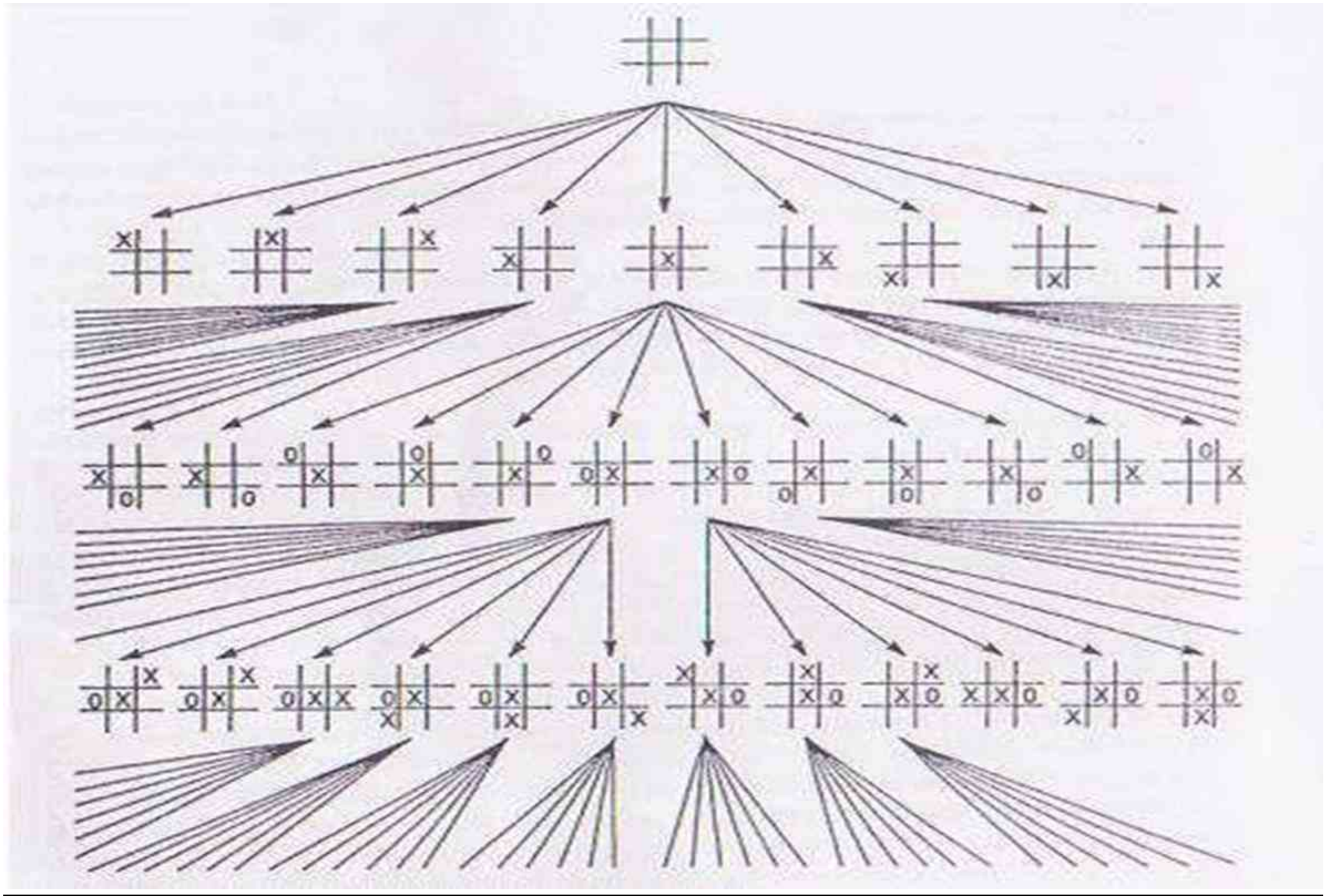


2. Consider the following 8-tiles problem then draw the problem state space to give the following requirements:



Find the goal using Best first search (or A-algorithm or Hill climbing) algorithm

Tic–Tac–Toe Game

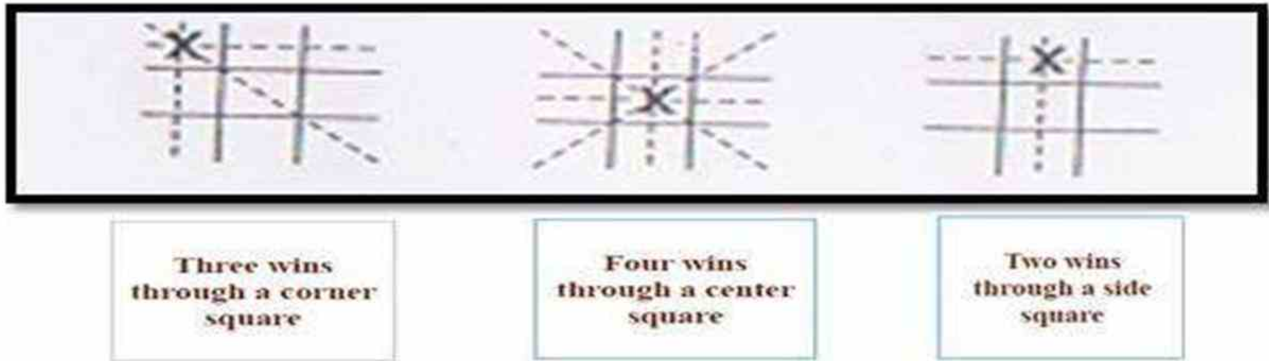


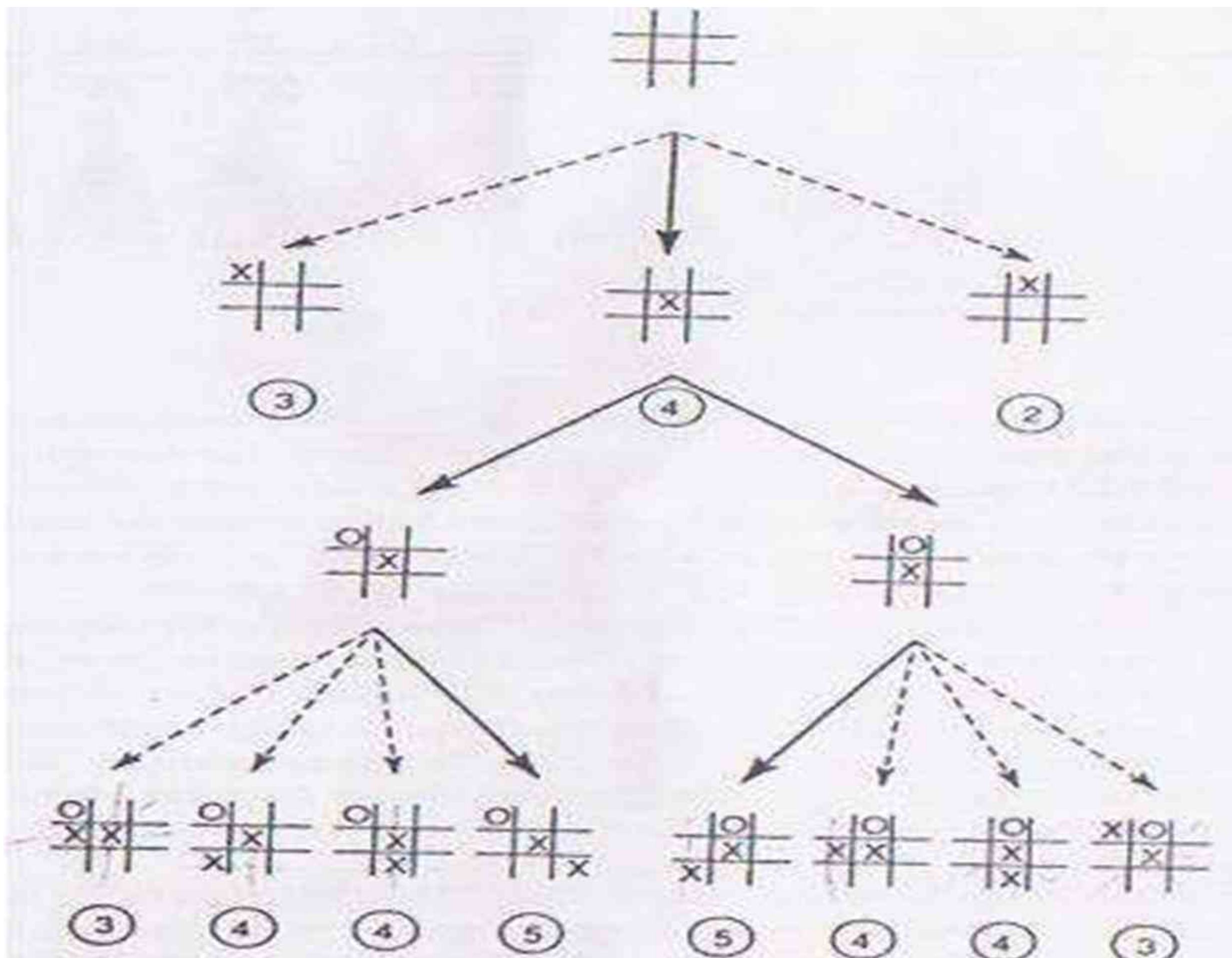
The complexity of the search space is 9!

9! =9*8*7*……….*1

Therefore it is necessary to implement two conditions:

1. Problem reduction
2. Guarantee the solution


Three wins through a corner square
Four wins through a center square
Two wins through a side square


3
4
2
3
4
4
5
5
4
4
3

Part 3: "Control Strategy"

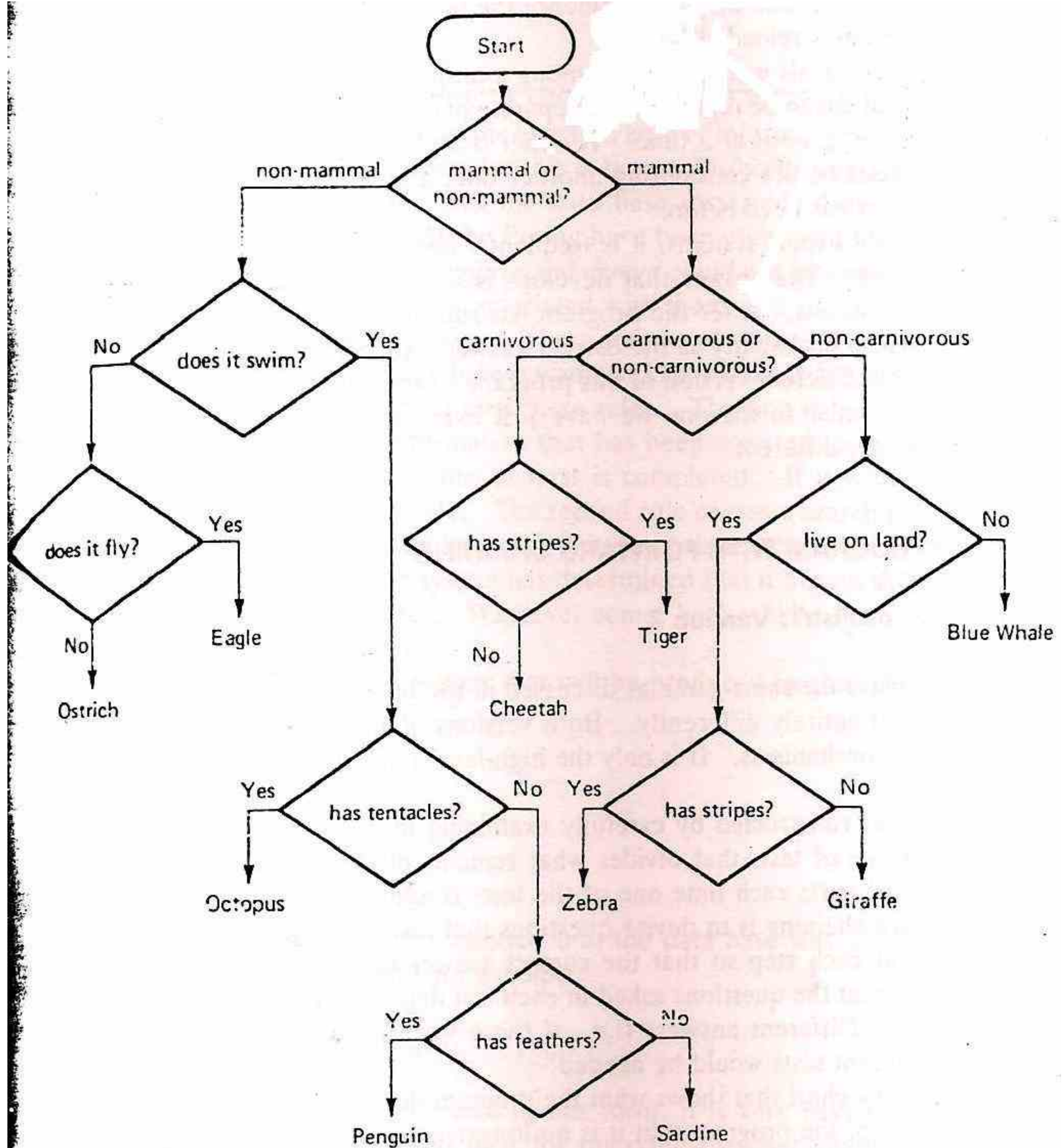
3. control strategy:

A control strategy is a plan or method for managing and directing actions or processes to achieve specific goals. In the context of computer science and artificial intelligence, control strategies are used to guide the decision-making process in systems, such as expert systems or inference engines. They help in determining the order in which rules or operations are applied to reach a conclusion or solve a problem.

- Backward Chaining: Starts with a goal and works backward to determine if the available data support that goal. It's like having the final picture of a puzzle and finding the pieces that fit to recreate it.
- Forward Chaining: Starts with known data and applies rules to derive new information until the goal is reached. It's like starting with the pieces of a puzzle and putting them together until the picture is complete.

Here's a table comparison of the important aspects of Forward Chaining and Backward Chaining:

Aspect	Forward Chaining	Backward Chaining
Starting Point	Known facts or data	Goal or hypothesis
Process	Applies inference rules to derive new data	Works backward from goal to validate with data
Data Flow	Data-driven (facts → conclusion)	Goal-driven (goal → facts)
Efficiency	Can be less efficient due to broad search	More efficient for specific goals
Use Cases	Expert systems, real-time systems	Logic programming, problem-solving
Example	Identifying an animal by its features	Proving an animal is a tiger by its traits



BBF is a classification program. The forward chaining version makes a series of binary decisions. Each is designed to throw away one half the remaining possibilities (or as close to that as possible until only one is left.

Classification Program with Backward Chaining (Bird, Beast, Fish)

```
# Database
db_confirm = set()
db_denied = set()


def confirm(X, Y):
    if (X, Y) in db_confirm:
        return True
    if (X, Y) in db_denied:
        return False
    return check(X, Y)


def denied(X, Y):
    return (X, Y) in db_denied


def check(X, Y):
    reply = input(f"{X} it {Y}?\n")
    if reply.lower() == "yes":
        db_confirm.add((X, Y))
        return True
    else:
        db_denied.add((X, Y))
        return False
```

```
def it_is(category):
    if category == "bird":
        return confirm("has", "feathers") and confirm("does", "lay_eggs")
    elif category == "fish":
        return confirm("does", "swim") and confirm("has", "fins")
    elif category == "mammal":
        return confirm("has", "hair") or confirm("does", "give_milk")
    elif category == "ungulate":
        return it_is("mammal") and confirm("has", "hooves") and confirm("does",
"chew_cud")
    elif category == "carnivorous":
        return confirm("has", "pointed_teeth") or confirm("does", "eat_meat")
    return False


def identify():
    if it_is("ungulate"):
        if confirm("has", "long_neck") and confirm("has", "long_legs") and
confirm("has", "dark_spots"):
            return "giraffe"
        elif confirm("has", "black_strips"):
            return "zebra"
    elif it_is("mammal") and it_is("carnivorous"):
        if confirm("has", "tawny_color"):
            if confirm("has", "black_spots"):
```

```
            return "cheetah"
        elif confirm("has", "black_strips"):
            return "tiger"
    elif it_is("bird"):
        if confirm("does", "fly") and confirm("is", "carnivorous"):
            if confirm("has", "use_as_national_symbol"):
                return "eagle"
        elif not confirm("does", "fly"):
            if confirm("has", "long_neck") and confirm("has", "long_legs"):
                return "ostrich"
            elif confirm("does", "swim") and confirm("has",
"black_and_white_color"):
                return "penguin"
    elif it_is("mammal") and not it_is("carnivorous") and confirm("does", "swim")
and confirm("has", "huge_size"):
        return "blue whale"
    elif not it_is("mammal") and it_is("carnivorous") and confirm("does", "swim")
and confirm("has", "tentacles"):
        return "octopus"
    elif it_is("fish") and confirm("has", "small_size") and confirm("has",
"use_in_sandwiches"):
        return "sardine"
    return "unknown"


def guess_animal():
```

```
    animal = identify()
    print(f"Your animal is a(n) {animal}")
```

 # Example usage

guess_animal()

Classification Program with Forward Chaining (Bird, Beast, Fish)

```
db_confirm = set()
db_denied = set()

def confirm(X, Y):
    if (X, Y) in db_confirm:
        return True
    if (X, Y) in db_denied:
        return False
    return check(X, Y)

def denied(X, Y):
    return (X, Y) in db_denied

def check(X, Y):
    reply = input(f"{X} it {Y}?\n")
    if reply.lower() == "yes":
```

```
        db_confirm.add((X, Y))
        return True
    else:
        db_denied.add((X, Y))
        return False


def it_is(category):
    if category == "bird":
        return confirm("has", "feathers") and confirm("does", "lay_eggs")
    elif category == "fish":
        return confirm("does", "swim") and confirm("has", "fins")
    elif category == "mammal":
        return confirm("has", "hair") or confirm("does", "give_milk")
    elif category == "ungulate":
        return it_is("mammal") and confirm("has", "hooves") and confirm("does",
"chew_cud")
    elif category == "carnivorous":
        return confirm("has", "pointed_teeth") or confirm("does", "eat_meat")
    return False


def find_animal():
    if test1("m"):
        if test2("m", "c"):
            if test3("m", "c", "s"):
```

```
            return "tiger"
        if test3("m", "c", "n"):
            return "cheetah"
    if test2("m", "n"):
        if test3("m", "n", "l"):
            return "unknown"
        if test3("m", "n", "n"):
            return "blue whale"
if test1("n"):
    if test2("n", "w"):
        if test3("n", "w", "t"):
            return "octopus"
        if test3("n", "w", "n"):
            return "unknown"
    if test2("n", "n"):
        if test3("n", "n", "f"):
            return "eagle"
        if test3("n", "n", "n"):
            return "ostrich"
return "unknown"


def test1(X):
    if X == "m":
        return it_is("mammal")
```

```
    return True

def test2(X, Y):
    if X == "m" and Y == "c":
        return it_is("carnivorous")
    if X == "n" and Y == "w":
        return confirm("does", "swim")
    return True

def test3(X, Y, Z):
    if X == "m" and Y == "c" and Z == "s":
        return confirm("has", "stripes")
    if X == "n" and Y == "w" and Z == "t":
        return confirm("has", "tentacles")
    if X == "n" and Y == "n" and Z == "f":
        return confirm("does", "fly")
    return True

def guess_animal():
    animal = find_animal()
    print(f"Your animal is a(n) {animal}")

guess_animal()
```