# Introduction To Artificial Intelligence

**الدراسة الأولية/المرحلة الثالثة /الفصل الدراسي الثاني**

**أستاذ المادة**
**أ.م.د. رواء داود حسن**

2020-2021
UNIVERSITY OF BAGHDAD
College of Science – Department of Computer Science

# Lecture 1: AI: DEFINITION AND APPLICATIONS

## 1.1 What is Artificial Intelligence?

**[Def. 1]** Artificial intelligence (AI) may be defined as the branch of computer science that is concerned with the automation of intelligent behaviour. This definition is particularly appropriate in that it emphasizes a conviction that AI is a part of computer science and, as such, 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. However, this definition suffers from the fact that intelligence itself is not very well defined or understood. Although most of us are certain that we know intelligent behaviour when we see it, it is doubtful that anyone could come close to defining intelligence in a way that would be specific enough to help in the evaluation of a supposedly intelligent computer program, while still capturing the vitality and complexity of the human mind **[Luger 2009]**.

**[Def. 2]** Artificial intelligence (AI) is the field that studies *the synthesis and analysis of computational agents that act intelligently*. Let us examine each part of this definition.
An **agent** is something that acts in an environment; it does something. Agents include worms, dogs, thermostats, airplanes, robots, humans, companies, and countries. We are interested in what an agent does; that is, how it **acts**. We judge an agent by its actions. An agent acts **intelligently** when

- what it does is appropriate for its circumstances and its goals, considering the short-term and long-term consequences of its actions
- it is flexible to changing environments and changing goals
- it learns from experience
- it makes appropriate choices given its perceptual and computational limitations

A **computational agent** is an agent whose decisions about its actions can be explained in terms of computation. That is, the decision can be broken down into primitive operations that can be implemented in a physical device. This computation can take many forms. In humans this computation is carried out in "wetware"; in computers it is carried out in "hardware." Although

there are some agents that are arguably not computational, such as the wind and rain eroding a landscape, it is an open question whether all intelligent agents are computational **[Poole and Mackworth].**

### 1.2 What can AI do today?

A concise answer is difficult because there are so many activities in so many subfields. Here I sample a few applications;

**Robotic vehicles:** A driverless robotic car named STANLEY sped through the rough terrain of the Mojave dessert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge. STANLEY is a Volkswagen Touareg outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and acceleration.

**Speech recognition:** A traveller calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.

**Autonomous planning and scheduling:** A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft. REMOTE AGENT such as Mars Exploration Rove (Opportunity) nicknamed Oppy, generated plans from high-level goals specified from the ground and monitored the execution of those plans—detecting, diagnosing, and recovering from problems as they occurred.

**Game playing:** IBM's DEEP BLUE became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match. Kasparov said that he felt a "new kind of intelligence" across the board from him. Newsweek magazine described the match as "The brain's last stand."

**Spam fighting:** Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting what, for many users, could comprise 80% or 90% of all messages, if not classified away by algorithms. Because the spammers are

continually updating their tactics, it is difficult for a static programmed approach to keep up, and learning algorithms work best.

**Logistics planning:** During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART, to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques generated in hours a plan that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

**Robotics:** The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.

**Machine Translation:** A computer program automatically translates from Arabic to English, allowing an English speaker to see the headline "Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus." The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totalling two trillion words. None of the computer scientists on the team speak Arabic, but they do understand statistics and machine learning algorithms

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics. The two most fundamental concerns of AI researchers are *knowledge representation* and *search*, to which this course provides an introduction. The first of these addresses the problem of capturing in a language, i.e., one suitable for computer manipulation, the full range of knowledge required for intelligent behaviour. Search is a problem-solving technique that systematically explores a space of problem states, i.e., successive and alternative stages in the problem-solving process.

# Lecture 2: AI AS REPRESENTATION

Knowledge representation languages can be defined as tools for helping humans solve problems. As such, a representation should provide a natural framework for expressing problem-solving knowledge; it should make that knowledge available to the computer and assist the programmer in its organization.

## 2.1 Predicate Calculus

Predicate calculus, as a *representation language for artificial intelligence*, its advantages include a well-defined formal semantics and complete inference rules. Predicate calculus provides the ability to access the components of a sentence. For example, denote the entire sentence "it rained on Tuesday," you can create a predicate weather that describes a relationship between a date and the weather:

weather (tuesday, rain).

Through inference rules we can manipulate predicate calculus expressions, accessing their individual components and inferring new sentences. Predicate calculus also allows expressions to contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for all values of X, where X is a day of the week, the statement weather (X, rain) is true; i.e., it rains every day.

### The Syntax of Predicates

Predicate calculus symbols, like the tokens in a programming language. We represent predicate calculus symbols as strings of letters and digits beginning with a letter. Blanks and nonalphanumeric characters cannot appear within the string, although the underscore, _, may be used to improve readability.

**Definition:** Predicate Calculus Symbols

The alphabet that makes up the symbols of the predicate calculus consists of:

1. The set of letters, both upper- and lowercase, of the English alphabet.

2. The set of digits, 0, 1, …, 9.

3. The underscore, _.

*Symbols* in the predicate calculus begin with a letter and are followed by any sequence of these legal characters.

Legitimate characters in the alphabet of predicate calculus symbols include

a R 6 9 p _ z

Examples of characters not in the alphabet include

# % @ / &

Legitimate predicate calculus symbols include

George fire3 tom_and_jerry bill XXXX friends_of

Examples of strings that are not legal symbols are

3jack no blanks allowed ab%cd ***71 duck!!!

Symbols are used to denote objects, properties, or relations in a world of discourse. As with most programming languages, the use of "words" that suggest the symbol's intended meaning assists us in understanding program code. Thus, even though I(g,k) and likes(george, kate) are formally equivalent (i.e., they have the same structure), the second can be of great help (for human readers) in indicating what relationship the expression represents. It must be stressed that these descriptive names are intended solely to improve the readability of expressions. The only "meaning" that predicate calculus expressions have is given through their formal semantics. Parentheses "( )", commas ",", and periods "." are used solely to construct well-formed expressions and do not denote objects or relations in the world. These are called *improper symbols*. Predicate calculus symbols may represent either variables, constants, functions, or predicates. Constants name specific objects or properties in the world. Constant symbols must begin with a lowercase letter. Thus george, tree, tall, and blue are examples of well-formed constant symbols. The constants true and false are reserved as truth symbols.

Variable symbols are used to designate general classes of objects or properties in the world. Variables are represented by symbols beginning with an uppercase letter. Thus George, BILL, and KAte are legal variables, whereas geORGE and bILL are not. Predicate calculus also allows functions on objects in the world of discourse. Function symbols (like constants) begin with a lowercase letter. Functions denote a mapping of one or more elements in a set (called the domain of the function) into a unique element of a second set (the range of the function). A *function expression* is a function symbol followed by its arguments. The arguments are elements from the domain of the function; the number of arguments is equal to the arity of the function. The arguments are enclosed in parentheses and separated by commas. For example,

f(X,Y)
father(david)
price(bananas)

are all well-formed function expressions. Each function expression denotes the mapping of the arguments onto a single object in the range, called the *value* of the function. For example, if father is a unary function, then father(david) is a function expression whose value (in the author's world of discourse) is george. The act of replacing a function with its value is called *evaluation*.

A predicate calculus *term* is either a constant, variable, or function expression. Thus, a predicate calculus term may be used to denote objects and properties in a problem domain. Examples of terms are:

cat
times(2,3)
X
blue
mother(sarah)
kate

Symbols in predicate calculus may also represent predicates. Predicate symbols, like constant and function names, begins with a lowercase letter. A predicate names a relationship between zero or more objects in the world. Examples of predicates are

like(goerge, kate).

likes(goegre, sarah, tuesday).
likes(X, sarah).
Friends(X,Y).
relationship.
friends(father_of(david), father_of(andrew)).

An atomic sentence is a predicate constant of arity $n$ "argument number", followed by $n$ terms, $t_1, t_2, \cdots, t_n$ , enclosed in parentheses and separated by commas. We may combine *atomic predicates* to form sentences using logical operators. These operators are: ∧ , ∨, ¬ , →, and ≡.

When a variable appears as an argument in a sentence, it refers to unspecified objects in the domain. Predicate calculus includes two symbols, the variable quantifiers ∀ and ∃, that constrain the meaning of a sentence containing a variable. A quantifier is followed by a variable and a sentence, such as

∃Y friends(Y, peter)
∀X likes(X, ice_cream)

The universal quantifier, ∀, indicates that the sentence is true for all values of the variable. In the example, ∀X likes(X, ice_cream) is true for all values in the domain of the definition of X. The existential quantifier, ∃, indicates that the sentence is true for at least one value in the domain. ∃Y friends(Y, peter) is true if there is at least one object, indicated by Y that is a friend of peter. For predicates p and q and variables X and Y:

¬ ∃ X p(X) ≡ ∀ X ¬ p(X)
¬ ∀ X p(X) ≡ ∃ X ¬ p(X)
∃ X p(X) ≡ ∃ Y p(Y)
∀ X q(X) ≡ ∀ Y q(Y)
∀ X (p(X) ∧ q(X)) ≡ ∀ X p(X) ∧ ∀ Y q(Y)
∃ X (p(X) ∨ q(X)) ≡ ∃ X p(X) ∨ ∃ Y q(Y)

We conclude this section with an example of the use of predicate calculus to describe a simple world. The domain of discourse is a set of family relationships in a biblical genealogy:

mother(eve,abel)

mother(eve,cain)

father(adam,abel)

father(adam,cain)

∀ X ∀ Y father(X, Y) ∨ mother(X, Y) → parent(X, Y)

∀ X ∀ Y ∀ Z parent(X, Y) ∧ parent(X, Z) → sibling(Y, Z)

Examples of English sentences represented in predicate calculus are:

If it doesn't rain on Monday, Tom will go to the mountains.

¬ weather(rain, monday) → go(tom, mountains)

Emma is a Doberman pinscher and a good dog.

gooddog(emma) ∧ isa(emma, doberman)

All basketball players are tall.

∀X (basketball_player(X) → tall(X))

Some people like anchovies.

∃ X (person(X) ∧ likes(X, anchovies))

Nobody likes taxes.

¬ ∃ X likes(X, taxes)

Every person who buys a policy is smart

∀X person(X) ∧ (∃Y policy(Y) ∧ buy(X,Y)) → smart(X)

There is an agent who sells policies only to people who are not insured

∃X agent(X) ∧ ∀Y,Z policy(Y) ∧ sells(X,Y,Z) → (person(Z) ∧ ¬ insured(Z))

A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth

Below Citizen(y,c,r) means that y is a citizen of c for reason r.

$\forall X\ person(X) \wedge born(X,uk) \wedge (\forall Y\ parent(Y,X) \rightarrow ((\exists R\ citizen(Y, uk, R)) \vee resident(Y,uk))) \rightarrow citizen(X, uk, birth)$

Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they cannot fool all of the people all of the time.

$\forall X\ politicians(X) \rightarrow (\exists Y\ \forall T\ person(Y) \wedge fools(X,Y,T)) \wedge (\exists T\ \forall Y\ person(Y) \rightarrow fools(X,Y,T)) \wedge \neg (\forall T\ \forall Y\ person(Y) \rightarrow fools(X,Y,T))$

**Lecture 3:**

### 2.1.1 Unification

To apply inference rules, an inference system must be able to determine when two expressions are the same or match. Unification is an algorithm for determining the substitutions needed to make two predicate calculus expressions match. Unification and inference rules allow us to make inferences on a set of logical assertions. Unification is complicated by the fact that a variable may be replaced by any term, including other variables and function expressions of arbitrary complexity. These expressions may themselves contain variables. In defining the unification algorithm that computes the substitutions required to match two expressions, a number of issues must be taken into account. First, although a constant may be systematically substituted for a variable, any constant is considered a "ground instance" and may not be replaced. Neither can two different ground instances be substituted for one variable. Second, a variable cannot be unified with a term containing that variable. X cannot be replaced by p(X) as this creates an infinite expression: p(p(p(p(...X)...))). For example,

L1 = P(X, Y, b)
L1= P(Z, W, b)

F= { (X/Z), (Y/W) }

L1 = P(X, Y, b)
L1= P(X, Y, b)

FL1 = FL2

Another example,

L1 = P(a, f(b), b)
L1= P(Z, W, c)

F= { (a/Z), (f(b)/ W) , b≠c}

No unification

Substitutions are also referred to as bindings. A variable is said to be *bound* to the value substituted for it.

### 2.1.2 Skolemization

Existential quantifiers are eliminated by a process called skolemization. When an expression contains an existentially quantified variable, for example, (∃Z) (foo(…,Z,…)), it may be concluded that there is an assignment to Z under which foo is true.

Skolemization identifies such a value. Skolemization does not necessarily show how to produce such a value; it is only a method for giving a name to an assignment that must exist. If k represents that assignment, then we have foo(…,k,…). Thus:

(∃X) (dog(X)) may be replaced by dog(fido)

where the name fido is picked from the domain of definition of X to represent that individual X. fido is called a *skolem constant*. If the predicate has more than one argument and the existentially quantified variable is within the scope of universally quantified variables, the existential variable must be a function of those other variables. This is represented in the skolemization process:

(∀X) (∃Y) (mother(X,Y))

This expression indicates that every person has a mother. Every person is an X and the existing mother will be a function of the particular person X that is picked. Thus skolemization gives:

(∀X) mother(X, m(X))

which indicates that each X has a mother (the m of that X). In another example:

(∀X)(∀Y)(∃Z)(∀W) (foo(X,Y,Z,W))

is skolemized to: (∀X)(∀Y)(∀W) (foo(X,Y,f(X,Y),W)).

### 2.1.3 Conversion to Clause Normal Form

All resolution proof procedures require that all statements in the database describing a situation to be converted to a standard form called *clause form*. This is motivated by the fact that resolution is an operator on pairs of disjuncts to produce new disjuncts. The form the database takes is referred to as a *conjunction of disjuncts*. It is a conjunction because all the clauses that make up the database are assumed to be true at the same time. It is a disjunction in that each of the individual clauses is expressed with disjunction (or ∨) as the connective. The algorithm which best describes the steps that produces the CNF of a logical expression is:

$$(\forall X)([a(X) \wedge b(X)] \rightarrow [c(X,I) \wedge (\exists Y)((\exists Z)[c(Y,Z)] \rightarrow d(X,Y))]) \vee (\forall X)(e(X))$$

1- First we eliminate the → by using the equivalent form a → b ≡ ¬ a ∨ b.

$$(\forall X)(\neg[a(X) \wedge b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))] \vee (\forall X)(e(X))$$

2- Next we reduce the scope of negation.

$$(\forall X)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))]) \vee (\forall X)(e(X))$$

3- Next we standardize by renaming all variables so that variables bound by different quantifiers have unique names.

$$(\forall X)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\exists Y)((\exists Z)[\neg c(Y,Z)] \vee d(X,Y))]) \vee (\forall W)(e(W))$$

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

$$(\forall X)(\exists Y)(\exists Z)(\forall W)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\neg c(Y,Z) \vee d(X,Y))] \vee e(W))$$

5- At this point all existential quantifiers are eliminated by a process called *skolemization*.

$$(\forall X)(\forall W)([\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\neg c(f(X),g(X)) \vee d(X,f(X)))]) \vee e(W))$$

6- Drop all universal quantification.

$$[\neg a(X) \vee \neg b(X)] \vee [c(X,l) \wedge (\neg c(f(X),g(X)) \vee d(X,f(X)))] \vee e(W)$$

7- Next we convert the expression to the conjunct of disjuncts form. This requires using the associative and distributive properties of ∧ and ∨.

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

$$[\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)] \wedge$$
$$[\neg a(X) \vee \neg b(X) \vee \neg c(f(X),g(X)) \vee d(X,f(X)) \vee e(W)]$$

8- Now call each conjunct a separate clause.

(1) $\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)$
(2) $\neg a(X) \vee \neg b(X) \vee \neg c(f(X),g(X)) \vee d(X,f(X)) \vee e(W)$

9- The final step is to *standardize the variables* apart again. This requires giving the variable in each clause generated by step 8 different names.

(1) $\neg a(X) \vee \neg b(X) \vee c(X,l) \vee e(W)$
(2) $\neg a(U) \vee \neg b(U) \vee \neg c(f(U),g(U)) \vee d(U,f(U)) \vee e(V)$

**Lecture 4:**

### 2.1.4 Resolution Theorem Proving

Resolution is a technique for proving theorems in the propositional or predicate calculus that has been a part of AI problem-solving research. The resolution principle describes a way of finding contradictions in a database of clauses with minimum use of substitution. Resolution refutation proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known (have been assumed) to be true. It then uses the resolution rule of inference to show that this leads to a contradiction. Once the theorem prover shows that the negated goal is inconsistent with the given set of axioms, it follows that the original goal must be consistent. This proves the theorem. Resolution refutation proofs involve the following steps:

1. Put the premises or axioms into clause form (2.1.3).
2. Add the negation of what is to be proved, in clause form, to the set of axioms.
3. Resolve these clauses together, producing new clauses that logically follow from them.
4. Produce a contradiction by generating the empty clause.
5. The substitutions used to produce the empty clause are those under which the opposite of the negated goal (what was originally to be proven) is true (more is said on this topic when we present strategies for refutation in the follow up discussion).

Resolution refutation proofs require that the axioms and the negation of the goal be placed in a normal form called clause form. Clause form represents the logical database as a set of disjunctions of literals. A literal is an atomic expression or the negation of an atomic expression.

The most common form of resolution, called *binary resolution*, is applied to two clauses when one contains a literal and the other its negation. If these literals contain variables, the literals must be unified to make them equivalent. A new clause is then produced consisting of the disjuncts of all the predicates in the two clauses minus the literal and its negative instance,

which are said to have been "resolved away." The resulting clause receives the unification substitution under which the predicate and its negation are found as "equivalent".

To make this explanation precise and clear we take a simple example to prove that "Fido will die" from the statements that "Fido is a dog" and "all dogs are animals" and "all animals will die."

Reasoning by resolution first converts these predicates to clause form:

| PREDICATE FORM | CLAUSE FORM |
|---|---|
| 1. ∀(X) (dog) (X) → animal (X)) | ¬ dog (X) ∨ animal (X) |
| 2. dog (fido) | dog (fido) |
| 3. ∀(Y) (animal (Y) → die (Y)) | ¬ animal (Y) ∨ die (Y) |

Negate the conclusion that Fido will die:

| | |
|---|---|
| 4. ¬ die (fido) | ¬ die (fido) |

Resolve clauses having opposite literals, producing new clauses by resolution as in Figure 1. This process is often called *clashing*.

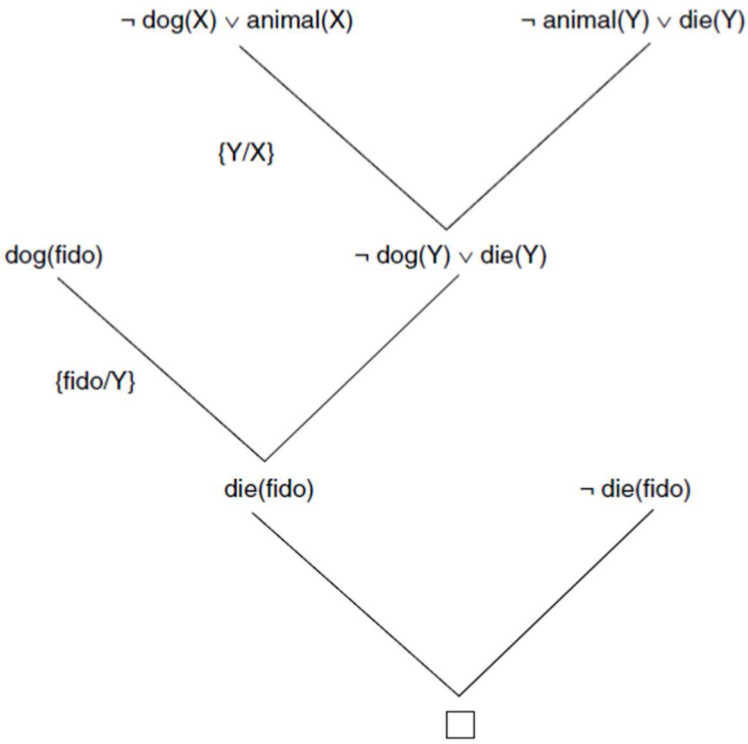



**Figure 1.** Resolution proof for the dead dog problem.

We now present an example of a resolution refutation for the predicate calculus.

Consider the following story of the "happy student":

> 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?

First change the sentences to predicate form:

Anyone passing his history exams and winning the lottery is happy.

$\forall X (pass(X,history) \wedge win(X,lottery) \rightarrow happy(X))$

Anyone who studies or is lucky can pass all his exams.

$\forall X \forall Y (study(X) \vee lucky(X) \rightarrow pass(X,Y))$

John did not study but he is lucky.

$\neg study(john) \wedge lucky(john)$

Anyone who is lucky wins the lottery.

$\forall X (lucky(X) \rightarrow win(X,lottery))$

These four predicate statements are now changed to clause form:

1. $\neg pass(X, history) \vee \neg win(X, lottery) \vee happy(X)$
2. $\neg study(Y) \vee pass(Y, Z)$
3. $\neg lucky(W) \vee pass(W, V)$
4. $\neg study(john)$
5. $lucky(john)$
6. $\neg lucky(U) \vee win(U, lottery)$

Into these clauses is entered, in clause form, the negation of the conclusion:

7. $\neg happy(john)$

The resolution refutation graph of Figure 2 shows a derivation of the contradiction and, consequently, proves that John is happy.

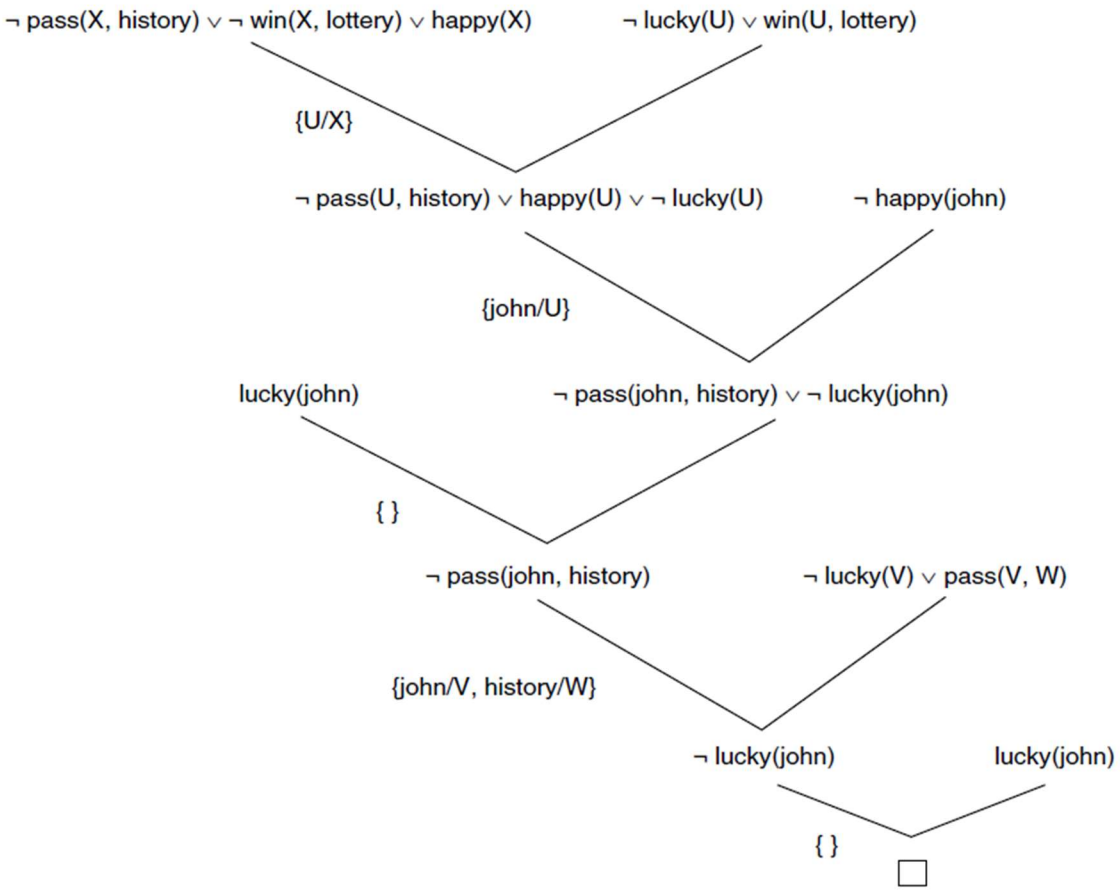



**Figure 2.** One resolution refutation for the happy student problem.

As a final example in this subsection we present the "exciting life" problem; suppose:

> 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. Can anyone be found with an exciting life?

We assume ∀X (smart (X) ≡ ¬ stupid (X)) and ∀Y (wealthy (Y) ≡ ¬ poor (Y)), and get:

∀X (¬ poor (X) ∧ smart (X) → happy (X))

∀Y (read (Y) → smart (Y))

read (john) ∧ ¬ poor (john)

∀Z (happy (Z) → exciting (Z))

The negation of the conclusion is: ¬ ∃ W (exciting (W))

These predicate calculus expressions for the "exciting life" problem are transformed into the following clauses:

poor (X) ∨ ¬ smart (X) ∨ happy (X)

¬ read (Y) ∨ smart (Y)

read (john)

¬ poor (john)

¬ happy (Z) ∨ exciting (Z)

¬ exciting (W)

The resolution refutation for this example is found in Figure 3.

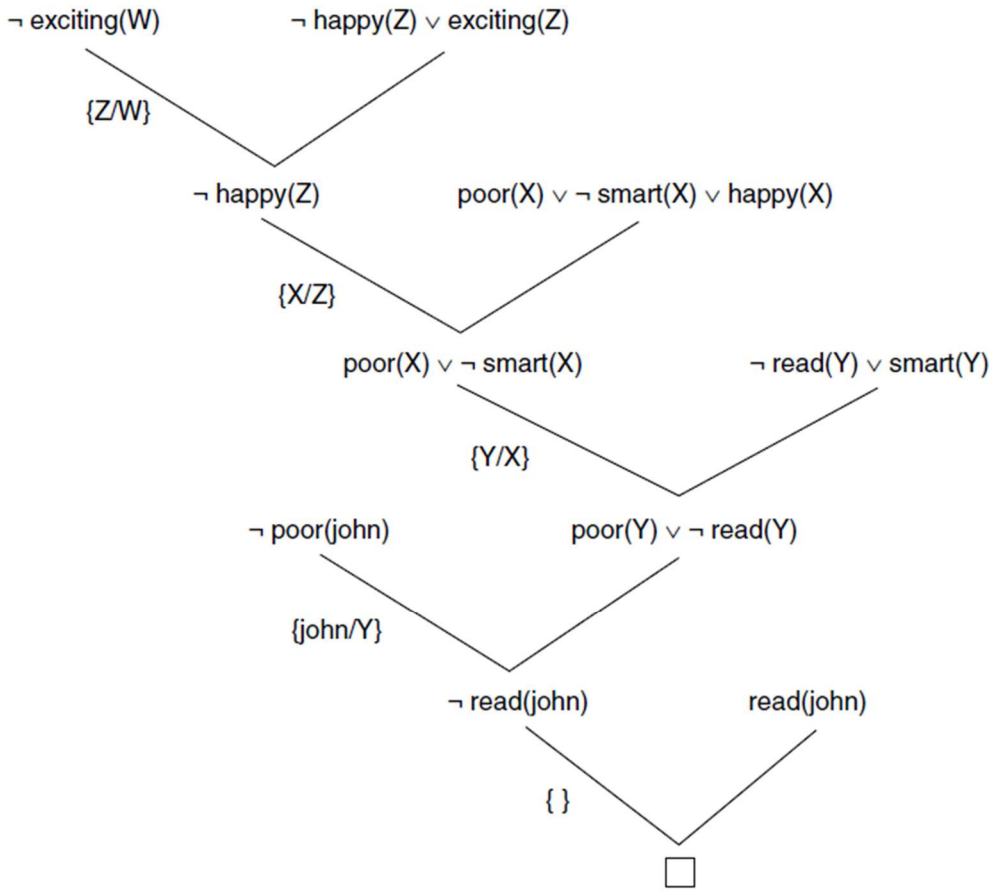



**Figure 3.** Resolution proof for the exciting life problem.

**Lecture 5:**

# 1. STRUCTURES AND STRATEGIES FOR STATE SPACE SEARCH

This section introduces the theory of state space search. To successfully design and implement search algorithms, a programmer must be able to analyse and predict their behaviour.

## 3.1 Graph Theory

Graph theory is our best tool for reasoning about the structure of objects and relations; indeed, this is precisely the need that led to its creation in the early eighteenth century. The Swiss mathematician Leonhard Euler invented graph theory to solve the "bridges of Konigsberg problem." The city of Konigsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges, as indicated in Figure 4.

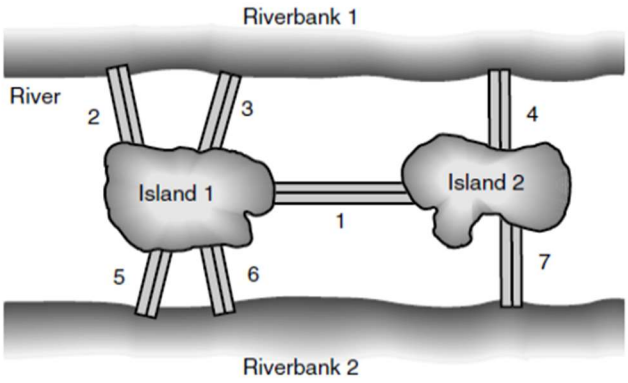



**Figure 4.** The city of Konigsberg.

By representing a problem as a *state space graph*, we can use *graph theory* to analyse the structure and complexity of both the problem and the search procedures that we employ to solve it.

A graph consists of a set of *nodes* and a set of *arcs* or *links* connecting pairs of nodes. In the state space model of problem solving, the nodes of a graph are taken to represent discrete *states* in a problem-solving process, such as the results of logical inferences or the different configurations of a game board. The arcs of the graph represent transitions between states. These transitions correspond to logical inferences or legal moves of a game.

Backing to the bridges of Konigsberg problem, it asks if there is a walk around the city that crosses each bridge exactly once. Although the residents had failed to find such a walk and doubted that it was possible, no one had proved its impossibility. Devising a form of graph theory, Euler created an alternative representation for the map, presented in Figure 5. The riverbanks (rb1 and rb2) and islands (i1 and i2) are described by the nodes of a graph; the bridges are represented by labelled arcs between nodes (b1, b2, b7). The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.

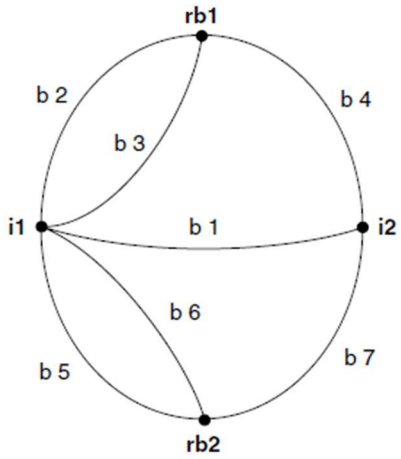



**Figure 5.** Graph of the Konigsberg bridge system.

Alternatively, we may represent the Konigsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a bridge. Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed. A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

| | |
|---|---|
| connect(i1, i2, b1) | connect(i2, i1, b1) |
| connect(rb1, i1, b2) | connect(i1, rb1, b2) |
| connect(rb1, i1, b3) | connect(i1, rb1, b3) |
| connect(rb1, i2, b4) | connect(i2, rb1, b4) |
| connect(rb2, i1, b5) | connect(i1, rb2, b5) |

connect(rb2, i1, b6) connect(i1, rb2, b6)

connect(rb2, i2, b7) connect(i2, rb2, b7)

**Prove whether such a walk is possible or impossible!**

D E F I N I T I O N

GRAPH

A graph consists of:

A set of *nodes* $N_1$, $N_2$, $N_3$, ..., $N_n$, ..., which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc ($N_3$, $N_4$) connects node $N_3$ to node $N_4$. This indicates a direct connection from node $N_3$ to $N_4$ but not from $N_4$ to $N_3$, unless ($N_4$, $N_3$) is also an arc, and then the arc joining $N_3$ and $N_4$ is *undirected*.

If a *directed* arc connects $N_j$ and $N_k$, then $N_j$ is called the *parent* of $N_k$ and $N_k$, the *child* of $N_j$. If the graph also contains an arc ($N_j$, $N_l$), then $N_k$ and $N_l$ are called *siblings*.

A *rooted* graph has a unique node $N_S$ from which all paths in the graph originate.
That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

An ordered sequence of nodes [$N_1$, $N_2$, $N_3$, ..., $N_n$], where each pair $N_i$, $N_{i+1}$ in the sequence represents an arc, i.e., ($N_i$, $N_{i+1}$), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

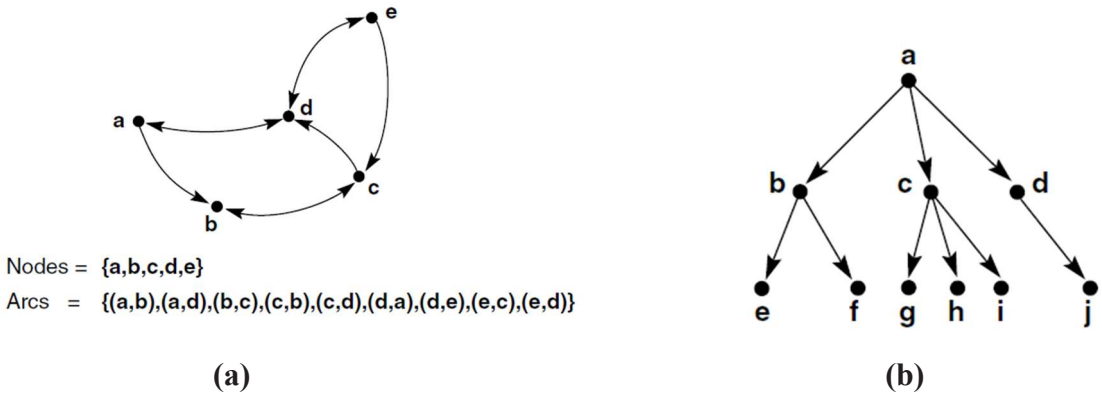
A path that contains any node more than once (some $N_j$ in the definition of path above is repeated) is said to contain a *cycle* or *loop.*

A *tree* is a graph in which there is a unique path between every pair of nodes.
(The paths in a tree, therefore, contain no cycles.)

The edges in a rooted tree are directed away from the root. Each node in a rooted tree has a unique parent.

Two nodes are said to be *connected* if a path exists that includes them both.

Figure 6 (a) is a is a labelled, directed graph. Figure 6 (b) depicts a tree.




**(a)** **(b)**

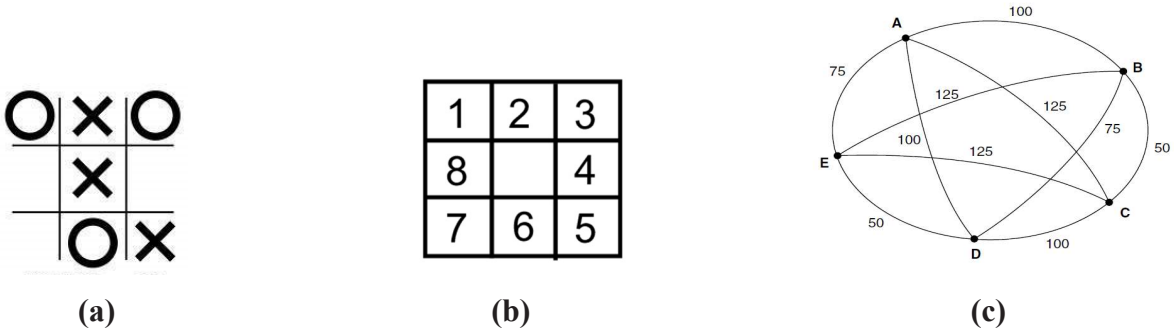
**Figure 6**. (a) A labeled directed graph (b) A rooted tree, exemplifying family relationships.

**Lecture 6:**

### 3.2 The State Space Representation of Problems

In the *state space representation* of a problem, the nodes of a graph correspond to partial problem solution *states* and the arcs correspond to steps in a problem-solving process. One or more *initial states*, corresponding to the given information in a problem instance, form the root of the graph. The graph also defines one or more *goal* conditions, which are solutions to a problem instance. *State space search* characterizes problem solving as the process of finding a *solution path* from the start state to a goal.

A *goal* may describe a state, such as a winning board in tic-tac-toe (Figure 7 (a)) or a goal configuration in the 8-puzzle (Figure 7 (b)). Alternatively, a goal can describe some property of the solution path itself. In the traveling salesperson problem (Figures 7 (c)), search terminates when the "shortest" path is found through all nodes of the graph.




**(a)** **(b)** **(c)**

**Figure 7.** Three different search problems. (a) tic-tac-toe (b) 8-puzzle game (c) travelling salesperson problem.

*Arcs* of the state space correspond to steps in a solution process and paths through the space represent solutions in various stages of completion. Paths are searched, beginning at the start state and continuing through the graph, until either the goal description is satisfied, or they are abandoned.

Now, the state space representation of problems can be formally defined:

D E F I N I T I O N

STATE SPACE SEARCH

A *state space* is represented by a four-tuple [N,A,S,GD], where:

N is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.

A is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process.

S, a nonempty subset of N, contains the start state(s) of the problem.

GD, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:

1. A measurable property of the states encountered in the search.
2. A measurable property of the path developed in the search, for example, the sum of the transition costs for the arcs of the path.

A *solution path* is a path through this graph from a node in S to a node in GD.

**Example 3.1: TIC-TAC-TOE**

The state space representation of tic-tac-toe appears in Figure 8. The start state is an empty board, and the termination or goal description is a board state having three Xs in a row, column, or diagonal (assuming that the goal is a win for X). The path from the start state to a goal state gives the series of moves in a winning game. The states in the space are all the different configurations of Xs and Os that the game can have. Of course, although there are $3^9$ ways to arrange {blank, X, O} in nine spaces, most of them would never occur in an actual game. Arcs are generated by legal moves of the game, alternating between placing an X and an O in an unused location. The state space is a graph rather than a tree, as some states on the third and deeper levels can be reached by different paths. However, there are no cycles in the state space, because the directed arcs of the graph do not allow a move to be undone.

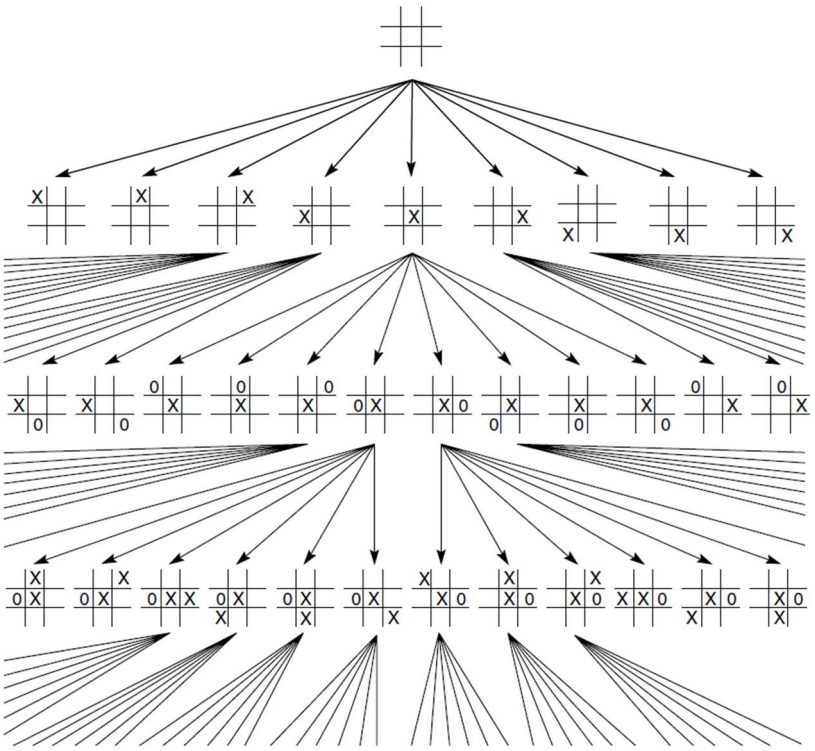


**Figure 8:** Portion of the state space for tic-tac-toe.

In tic-tac-toe, there are nine first moves with eight possible responses to each of them, followed by seven possible responses to each of these, and so on. It follows that 9 ×8 × 7 × ... or 9! (362,880) different paths can be generated.

**Example 3.2: THE 8-PUZZLE**

The 8-puzzle is a 3 ×3 grid in which eight tiles can be moved around in nine spaces. One space is left blank so that tiles can be moved around to form different patterns. The goal is to find a series of moves of tiles into the blank space that places the board in a goal configuration. In order to apply a move, we must make sure that it does not move the blank off the board. Therefore, all four moves are not applicable at all times; for example, when the blank is in one of the corners only two moves are possible. The legal moves are:

move the blank up ↑

move the blank right →

move the blank down ↓

move the blank left ←

The state space representation of 8-Puzzel appears in Figure 9.

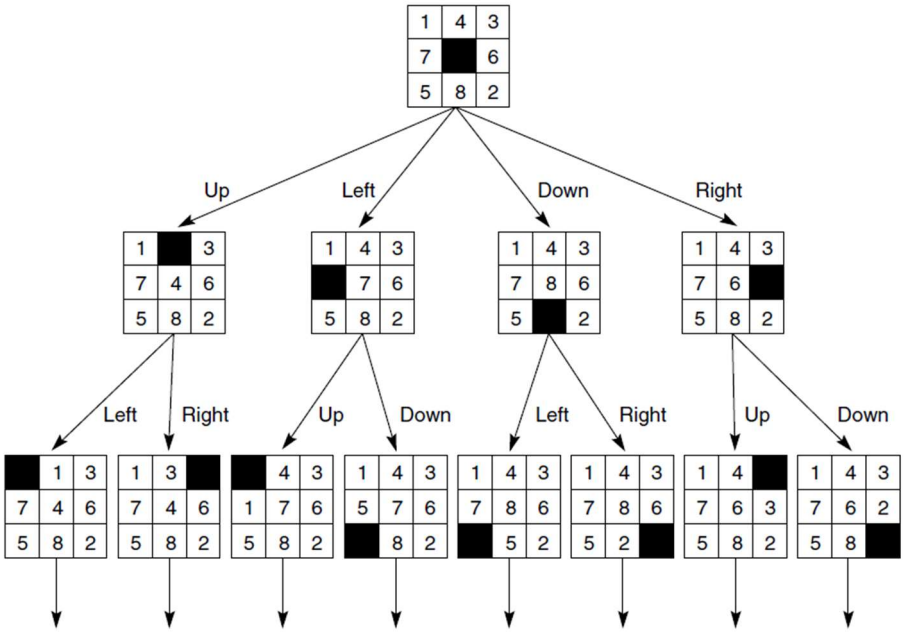



**Figure 9:** State space of the 8-puzzle generated by move blank operations.

As with tic-tac-toe, the state space for the 8-puzzle is a graph (with most states having multiple parents), but unlike tic-tac-toe, cycles are possible. The GD or goal description of the state space is a particular state or board configuration. When this state is found on a path, the search terminates. The path from start to goal is the desired series of moves.

**Example 3.3: THE TRAVELING SALESPERSON**

Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. Figure 7 (c) gives an instance of this problem. The nodes of the graph represent cities, and each arc is labelled with a weight indicating the cost of traveling that arc. The path [A,D,C,B,E,A], with associated cost of 450 miles, is an example of a possible circuit. The goal description requires a complete circuit with minimum cost. Figure 10 shows one way in which possible solution paths may be generated and compared. Beginning with node A, possible next states are added until all cities are included, and the path returns home.

The goal is the lowest-cost path. As Figure 10 suggests, the complexity of exhaustive search in the traveling salesperson problem is (N - 1)!, where N is the number of cities in the graph.

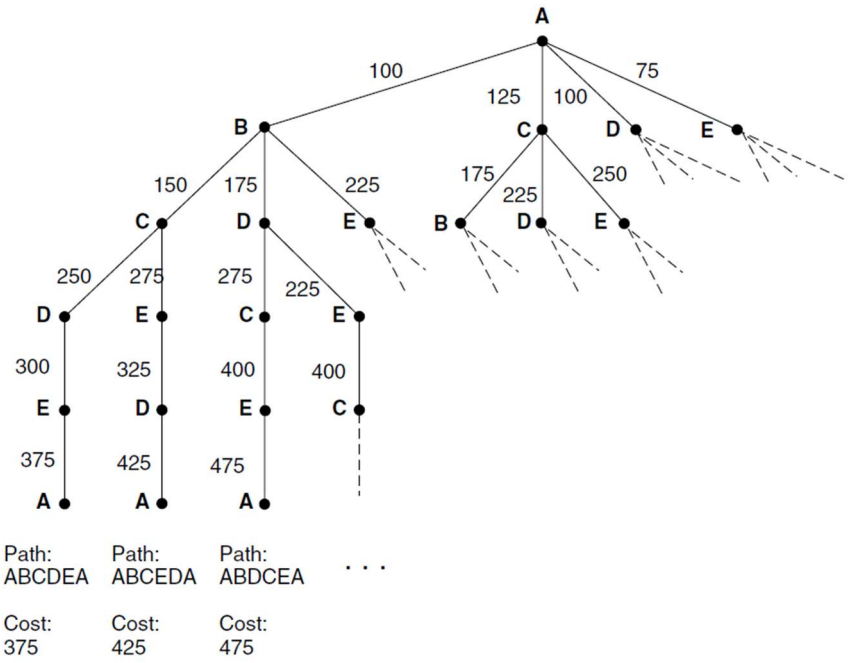



**Figure 10:** Search of the traveling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

**Lecture 7:**

### 3.3 Strategies for State Space Search

In solving a problem, a problem solver must find a path from a start state to a goal through the state space graph. The sequence of arcs in this path corresponds to the ordered steps of the solution. The problem solver would move unerringly through the space to the desired goal, constructing the path as it went, it must consider different paths through the space until it finds a goal.

We begin with backtracking based algorithms called *depth-first search* and *breadth-first search* because they are of the first search algorithms computer scientists' study, and they have a natural implementation in a stack oriented recursive environment. These search algorithms determine the order in which states are examined in the tree or the graph in which there are two possibilities for the order.

Consider the graph represented in Figure 11. States are labelled (A, B, C, . . .) so that they can be referred to in the search process. In **depth-first search**, when a state is examined, all its children and their descendants are examined before any of its siblings. Depth-first search goes deeper into the search space whenever this is possible. Only when no further descendants of a state can be found are its siblings considered.

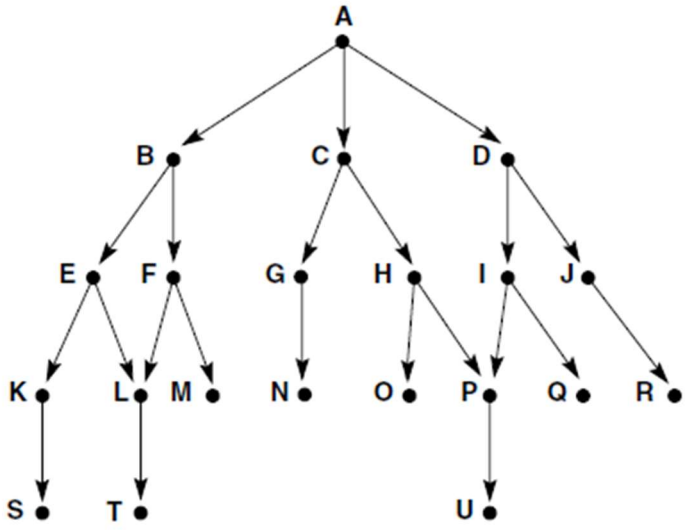



**Figure 11:** Graph for breadth- and depth-first search examples.

```
function depth_first_search;

begin
   open := [Start];                                          % initialize
   closed := [ ];
   while open ≠ [ ] do                                       % states remain
      begin
         remove leftmost state from open, call it X;
         if X is a goal then return SUCCESS                  % goal found
            else begin
               generate children of X;
               put X on closed;
               discard children of X if already on open or closed;   % loop check
               put remaining children on left end of open            % stack
            end
      end;
   return FAIL                                               % no states left
end.
```

A trace of *depth_first_search* on the graph of Figure 11 appears below.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]
8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

and so on until either U is discovered or open = [ ].

**Breadth-first search**, in contrast, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the algorithm move on to the next deeper level.

```
function breadth_first_search;

begin
  open := [Start];                                          % initialize
  closed := [ ];
  while open ≠ [ ] do                                       % states remain
    begin
      remove leftmost state from open, call it X;
        if X is a goal then return SUCCESS                  % goal found
          else begin
            generate children of X;
            put X on closed;
            discard children of X if already on open or closed;   % loop check
            put remaining children on right end of open           % queue
          end
    end
  return FAIL                                               % no states left
end.
```

A trace of *breadth_first_search* on the graph of Figure 11 appears below.

1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].

In both algorithms, each successive iteration of the "while" loop is indicated by a single line (2, 3, 4, . . .). The initial states of open and closed are given on line 1. Assume U is the goal state.

**Breadth-First** Because it always examines all the nodes at level n before proceeding to level n + 1, breadth-first search always finds the shortest path to a goal node. In a problem where it is known that a simple solution exists, this solution will be found.

Unfortunately, if there is a bad branching factor, i.e., states have a high average number of children, the combinatorial explosion may prevent the algorithm from finding a solution using available memory. This is due to the fact that all unexpanded nodes for each level of the search

must be kept on open. For deep searches, or state spaces with a high branching factor, this can become quite cumbersome.

**Depth-First** Depth-first search gets quickly into a deep search space. If it is known that the solution path will be long, depth-first search will not waste time searching a large number of "shallow" states in the graph. On the other hand, depth-first search can get "lost" deep in a graph, missing shorter paths to a goal or even becoming stuck in an infinitely long path that does not lead to a goal.

Depth-first search is much more efficient for search spaces with many branches because it does not have to keep all the nodes at a given level on the open list.

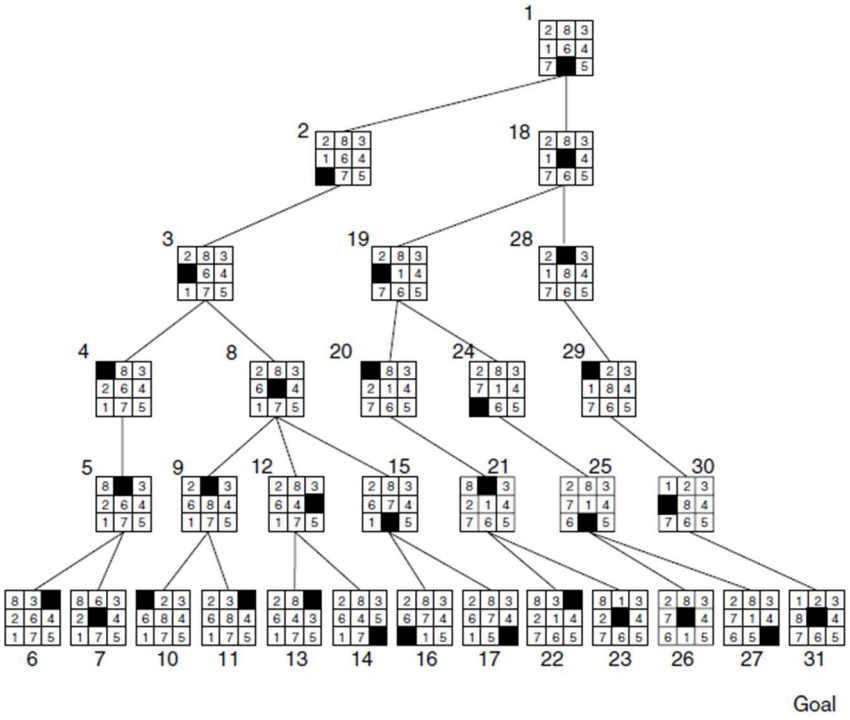



**Figure 12:** Depth-first search of the 8-puzzle with a depth bound of 5.

Figure 12 gives a depth-first search of the 8-puzzle. As noted previously, the space is generated by the four "move blank" rules (up, down, left, and right). The numbers next to the states indicate the order in which they were considered, i.e., removed from open. States left on open when the goal is found are not shown. A depth bound of 5 was imposed on this search to keep it from getting lost deep in the space.

## Lecture 8:

### Heuristic Search

Heuristics and the design of algorithms to implement heuristic search have long been a core concern of artificial intelligence. Game playing and theorem proving are two of the oldest applications in artificial intelligence; both of these require heuristics to prune spaces of possible solutions. It is not feasible to examine every inference that can be made in a mathematics domain or every possible move that can be made on a chessboard. Heuristic search is often the only practical answer. It is useful to think of heuristic search from two perspectives: the heuristic measure and an algorithm that uses heuristics to search the state space. We will consider two types of heuristic algorithms: Heuristic search algorithms (with no cost function) and heuristic search algorithm (with cost function). One form of heuristic information about which nodes seem the most promising is a *heuristic function* $h(n)$, which takes a node $n$ and returns a non-negative real number that is an estimate of the path cost from node $n$ to a goal node. The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbour of a node will lead to a goal. There is no general theory for finding heuristics, because every problem is different.

### 1. The Best-First Search Algorithm

Like the depth-first and breadth-first search algorithms, best-first search uses lists to maintain states: open to keep track of the current fringe of the search and closed to record states already visited. An added step in the algorithm orders the states on open according to some heuristic estimate of their "closeness" to a goal. Thus, each iteration of the loop considers the most "promising" state on the open list.

A trace of the execution of best_first_search on the graph appears in Figure 13. Suppose P is the goal state in this graph. Because P is the goal, states along the path to P will tend to have lower heuristic values. The heuristic is fallible: the state O has a lower value than the goal itself and is examined first. Unlike hill climbing, which does not maintain a priority queue for the selection of "next" states, the algorithm recovers from this error and finds the correct goal.

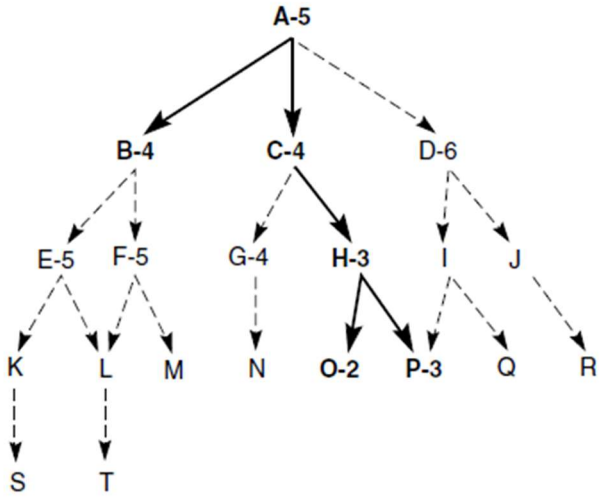



**Figure 13:** Heuristic search of a hypothetical state space.

```
function best_first_search;

begin
  open := [Start];                                          % initialize
  closed := [ ];
  while open ≠ [ ] do                                       % states remain
    begin
      remove the leftmost state from open, call it X;
      if X = goal then return the path from Start 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.
```

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]
3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!

## 2. A algorithm (The Best-First Search Algorithm with cost)

Another way to measure the cost from the start state to the goal state is the *evaluation function $f(n)$ (cost function)*. Cost function can be measured as,

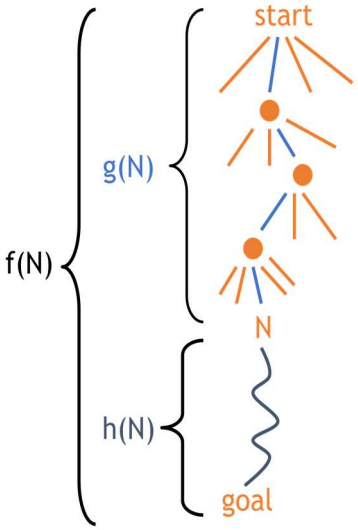
$$f(n) = g(n) + h(n)$$

where

$h(n)$ is the heuristic function that estimates the distance between node $n$ and a goal node.

$g(n)$ is the (known) distance from the start state to a goal node $n$.

$f(n)$ gives you the (partially estimated) distance from the start node to a goal node $n$.




A trace of the execution of best_first_search with cost on the graph appears in Figure 14. Suppose M is the goal state in this graph.

**Lecture 9:**

### 3.3.2 Heuristic in Games

In this subsection, we evaluate the performance of several different heuristics for solving games with multiple playing strategies.

- **8-puzzle board**

Figure 15 shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

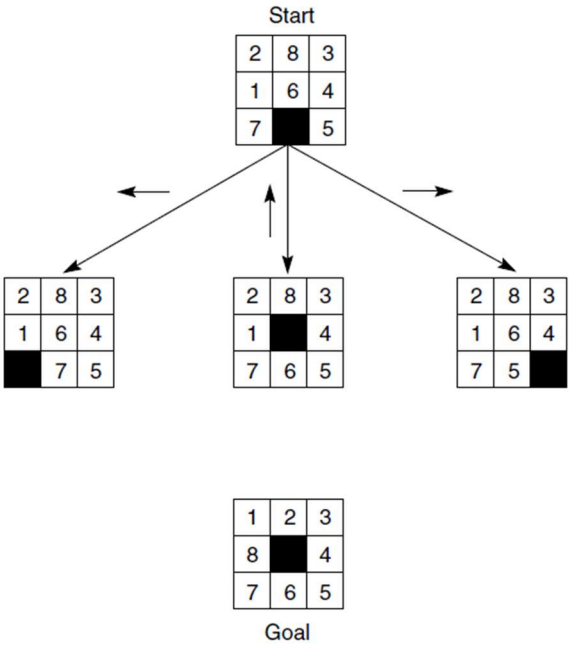



**Figure 15:** The start state, first moves, and goal state for an example 8-puzzle.

The simplest heuristic counts the tiles out of place in each state when compared with the goal, 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 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 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 opposite locations, it

takes (several) more than two moves to put them back in place, as the tiles must "go around" each other. A heuristic that takes this into account multiplies a small number (2, for example) times each direct tile reversal (where two adjacent tiles must be exchanged to be in the order of the goal). Figure 16 shows the result of applying each of these three heuristics to the three child states of Figure 15.

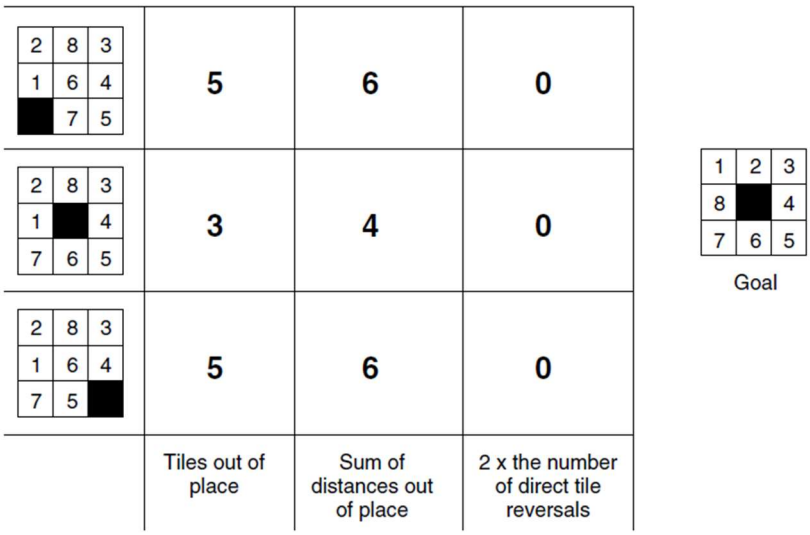



**Figure 16:** Three heuristics applied to states in the 8-puzzle.

If two states have the same or nearly the same heuristic evaluations, it is generally preferable to examine the state that is nearest to the root state of the graph. This state will have a greater probability of being on the shortest path to the goal. The distance from the starting state to its descendants can be measured by maintaining a depth count for each state. This count is 0 for the beginning state and is incremented by 1 for each level of the search. This makes our evaluation function, *f,* the sum of two components:

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

where $g(n)$ measures the actual length of the path from any state $n$ to the start state and $h(n)$ is a heuristic estimate of the distance from state $n$ to a goal.

The full best-first search of the 8-puzzle graph, using $f$ as defined above, appears in Figure 17. Each state is labelled with a letter and its heuristic weight, $f(n) = g(n) + h(n)$. The number at the top of each state indicates the order in which it was taken off the open list. Some states

(h, g, b, d, n, k, and i) are not so numbered, because they were still on open when the algorithm terminates.

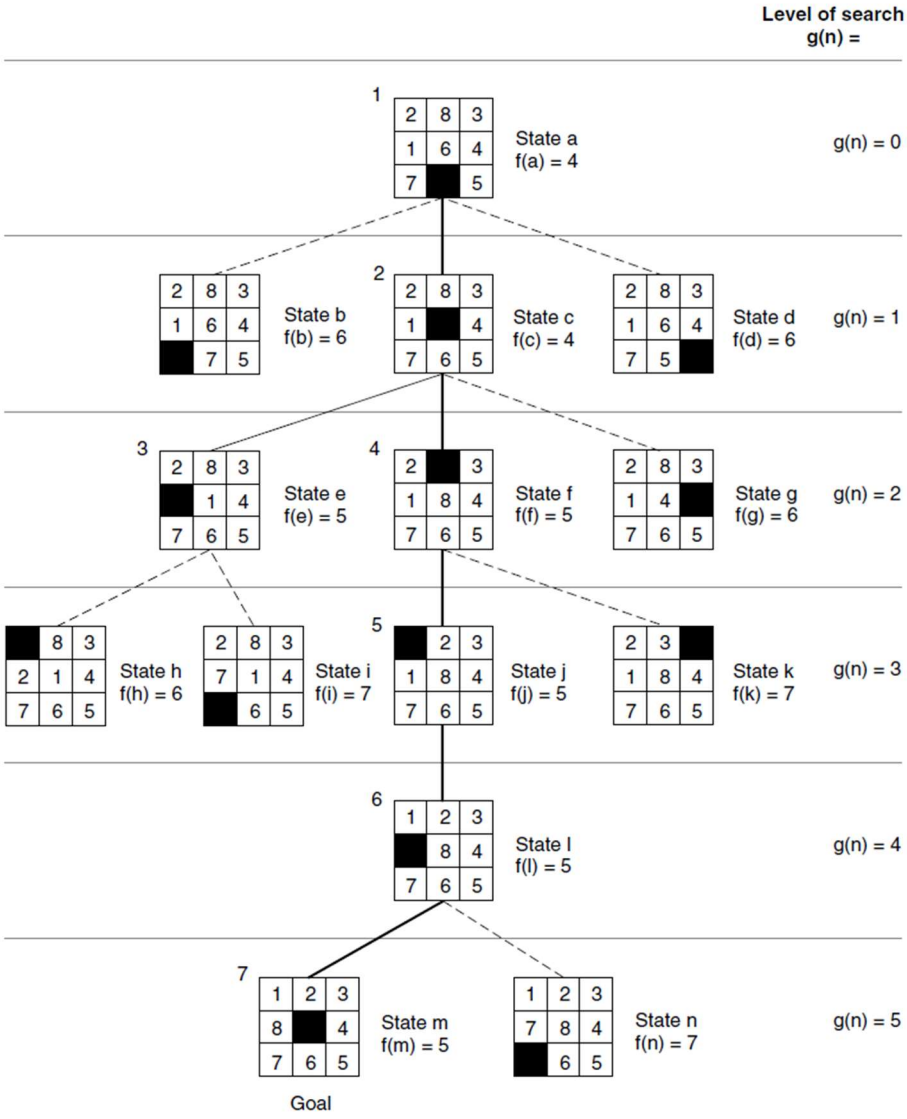



**Figure 17:** State space generated in heuristic search of the 8-puzzle graph.

- **Tic-tac-toe board**

In tic-tac toe, a simple heuristic, however, can almost eliminate search entirely: we may move to the state in which X has the most winning opportunities (The first three states in the tic tac toe game are so measured in Figure 18). In case of states with equal numbers of potential wins, take the first such state found. The algorithm then selects and moves to the state with the highest number of opportunities. In this case X takes the centre of the grid.

Note that not only are the other two alternatives eliminated, but so are all their descendants. Two-thirds of the full space is pruned away with the first move, Figure 19.

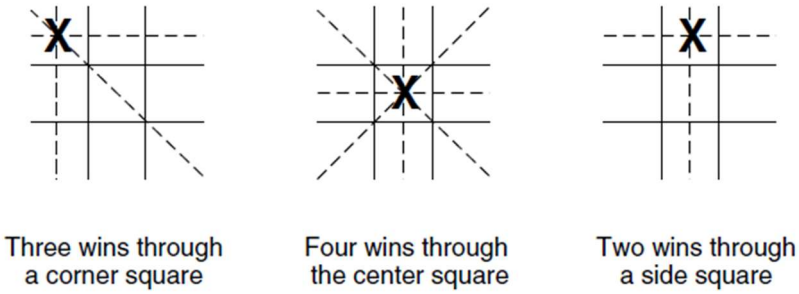



**Figure 18:** The most wins heuristic applied to the first children in tic-tac-toe.

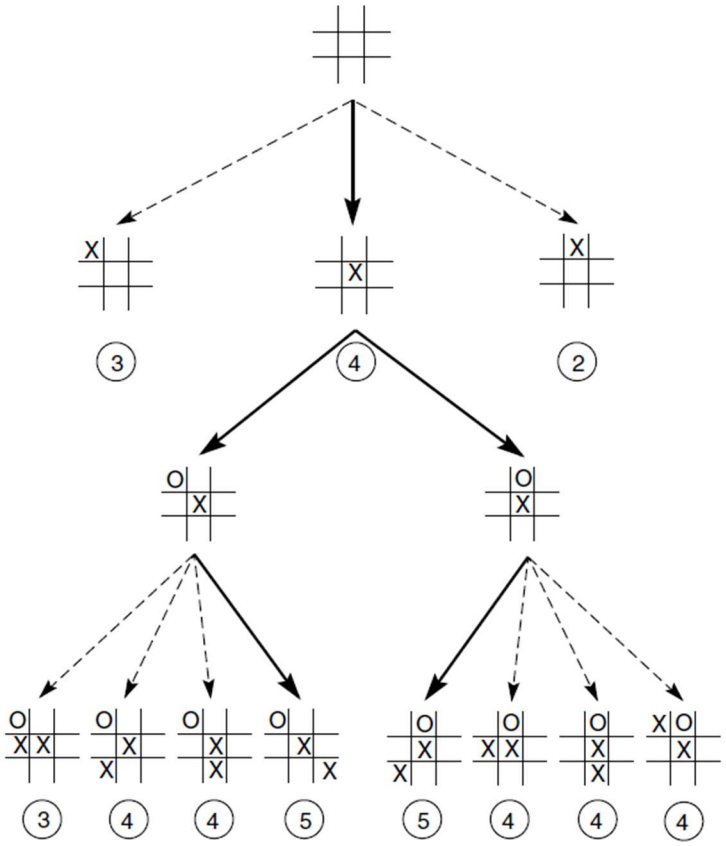



**Figure 19:** Heuristically reduced state space for tic-tac-toe.

## Lecture 10:

- **The Minimax Procedure on Exhaustively Searchable Graphs**

Two-person games are more complicated than simple puzzles because of the existence of a "hostile" and essentially unpredictable opponent. Thus, they provide some interesting opportunities for developing heuristics, as well as greater difficulties in developing search algorithms.

First, we consider games whose state space is small enough to be exhaustively searched; this is to easily search the problem space systematically of possible moves and countermoves by the opponent. We first consider a variant of the game ***nim***, whose state space may be exhaustively searched. To play this game, several tokens are placed on a table between the two opponents; at each move, the player must divide a pile of tokens into two nonempty piles of different sizes. Thus, 6 tokens may be divided into piles of 5 and 1 or 4 and 2, but not 3 and 3. The first player who can no longer make a move loses the game. For a reasonable number of tokens, the state space can be exhaustively searched. Figure 21 illustrates the space for a game with 7 tokens.

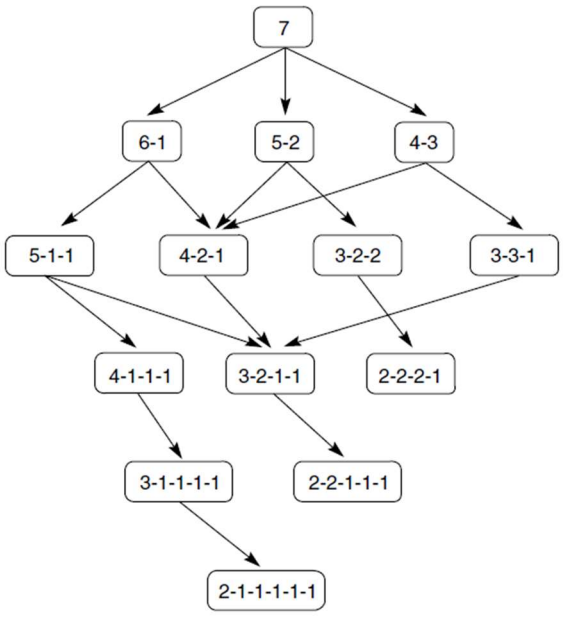



**Figure 21:** State space for a variant of *nim*. Each state partitions the seven matches into one or more piles

Assumes that your opponent uses the same knowledge of the state space as you use and applies that knowledge in a consistent effort to win the game. **Minimax** procedure searches the game space under this assumption. The opponents in a game are referred to as MIN and MAX: MAX represents the player trying to win, or to MAXimize her advantage. MIN is the opponent who attempts to MINimize MAX's score. We assume that MIN uses the same information and always attempts to move to a state that is worst for MAX.

In implementing minimax, we label each level in the search space according to whose move it is at that point in the game, MIN or MAX. In the example of Figure 22, MIN can move first. Each leaf node is given a value of 1 or 0, depending on whether it is a win for MAX or for MIN. Minimax propagates these values up the graph through successive parent nodes according to the rule:

If the parent state is a MAX node, give it the maximum value among its children.

If the parent is a MIN node, give it the minimum value of its children.

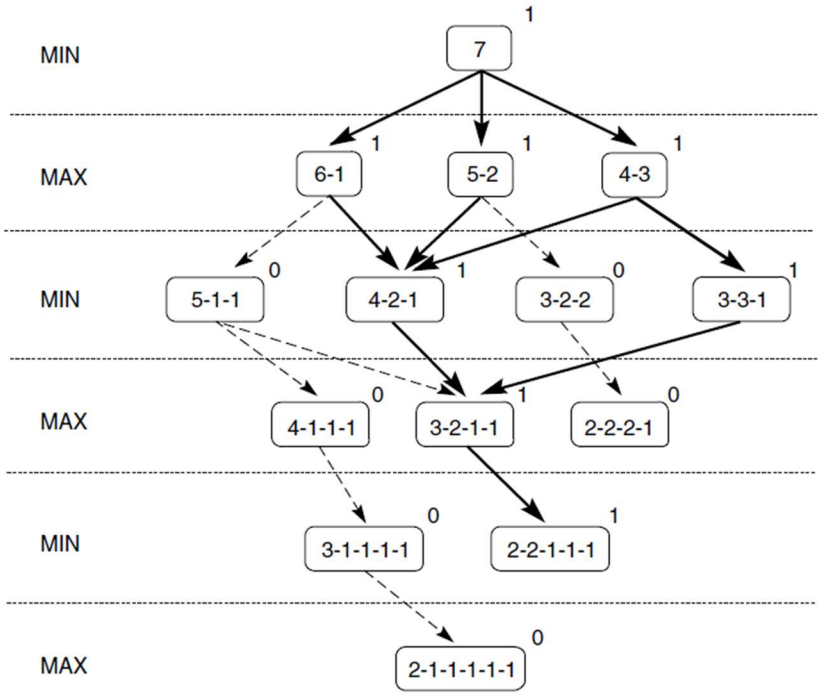



**Figure 22:** Exhaustive minimax for the game of *nim*. Bold lines indicate forced win for MAX. Each node is marked with its derived value (0 or 1) under minimax.

The values of the leaf nodes are propagated up the graph using minimax. Because all of MIN's possible first moves lead to nodes with a derived value of 1, the second player, MAX, always can force the game to a win, regardless of MIN's first move. MIN could win only if MAX played foolishly. In Figure 22, MIN may choose any of the first move alternatives, with the resulting win paths for MAX in bold arrows.