

Game Theory and Knowledge Representation

Unit 3

Brandon Syiem

Representation and Mappings

- Knowledge or facts has to be represented in some form that can be manipulated and understood by computers.
- Up until this point we have talked about heuristic functions and that AI programs exploit knowledge but we have glossed over the details of how Knowledge is represented.
- To solve complex AI problems, we need :
 - Large amounts of knowledge
 - Mechanism for representing and manipulating said knowledge.

Cont ..

- Two different entities
 - Knowledge or facts
 - Representation of facts
- The way we usually structure these entities are in levels:
 - **Knowledge level** - describing facts
 - **Symbol level** - representation of objects from the knowledge level

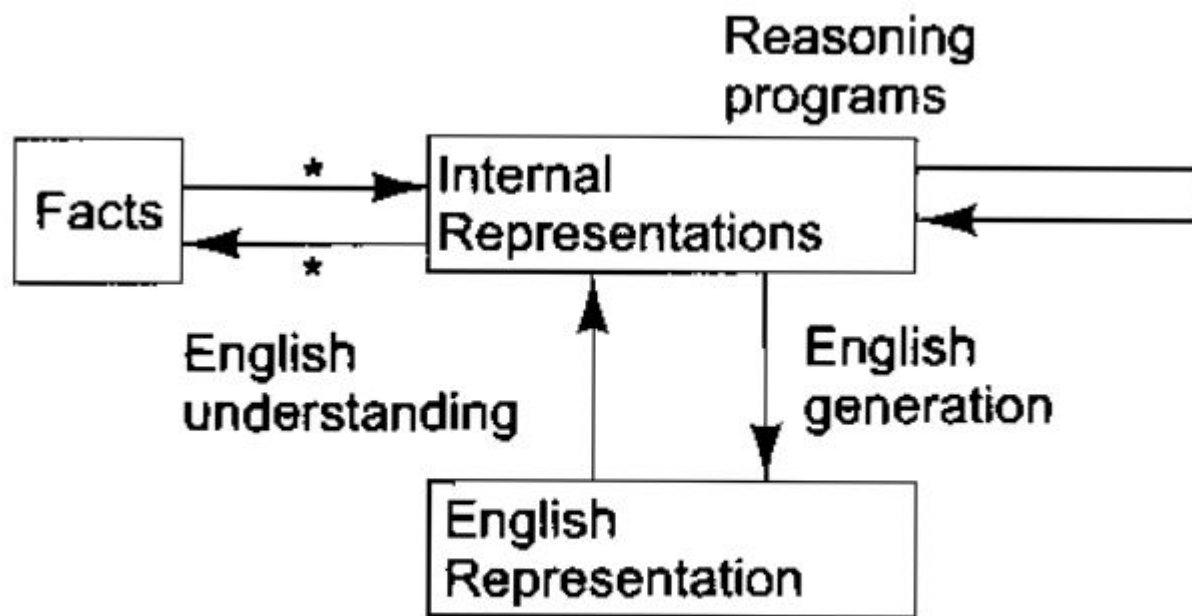


Fig. 4.1 *Mappings between Facts and Representations*

Cont ..

- Mapping from facts to representations is called **forward mapping**
- Mappings from representation to facts is called **backward mapping**
- Both forward and backward mappings together is called ***representation mapping***

<https://www.slideshare.net/ameykerkar/knowledge-representation-and-predicate-logic>

Example

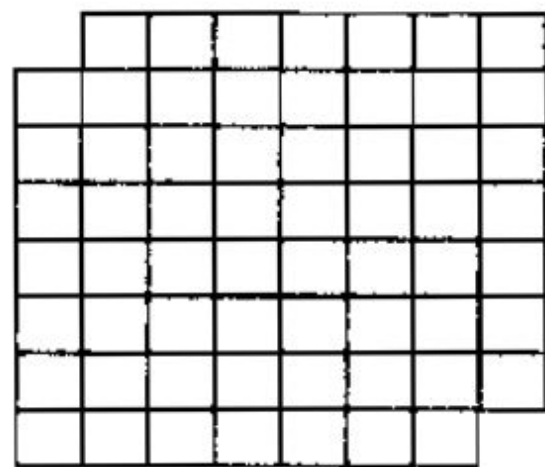
- Spot is a dog
- `dog(Spot)`
- All dogs have tails
- $\forall x: \text{dog}(x) \rightarrow \text{hastail}(x)$
- `hastail(Spot)`
- Spot has a tail

- `parent(dan, jim).`
- `parent(lily, jim).`
- `parent(dan, tommy).`
- `parent(lily, tommy).`
- `sibling(X,Y):-`
 - `parent(Z,X),`
 - `parent(Z,Y),`
 - `X\=Y.`

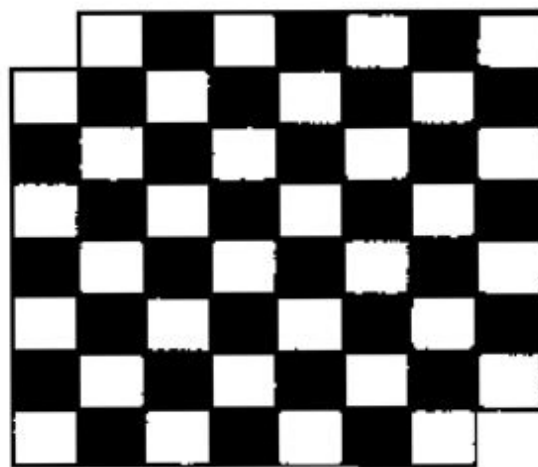
Cont ..

- Is not one-to-one mapping
- Not functions but many-to-many relations
- Therefore we need to decide on what exactly a fact represents
 - All dogs have tails
 - Every dog has a tail
 - Every dog has atleast one tail
- AI programs manipulate the representation of facts given to it, which results in new representations.

The Mutilated Checker board Problem. Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?



(a)



(b)

Number of
black squares = 30

Number of
white squares = 32

(c)

Fig. 4.2 *Three Representations of a Mutilated Checker board*

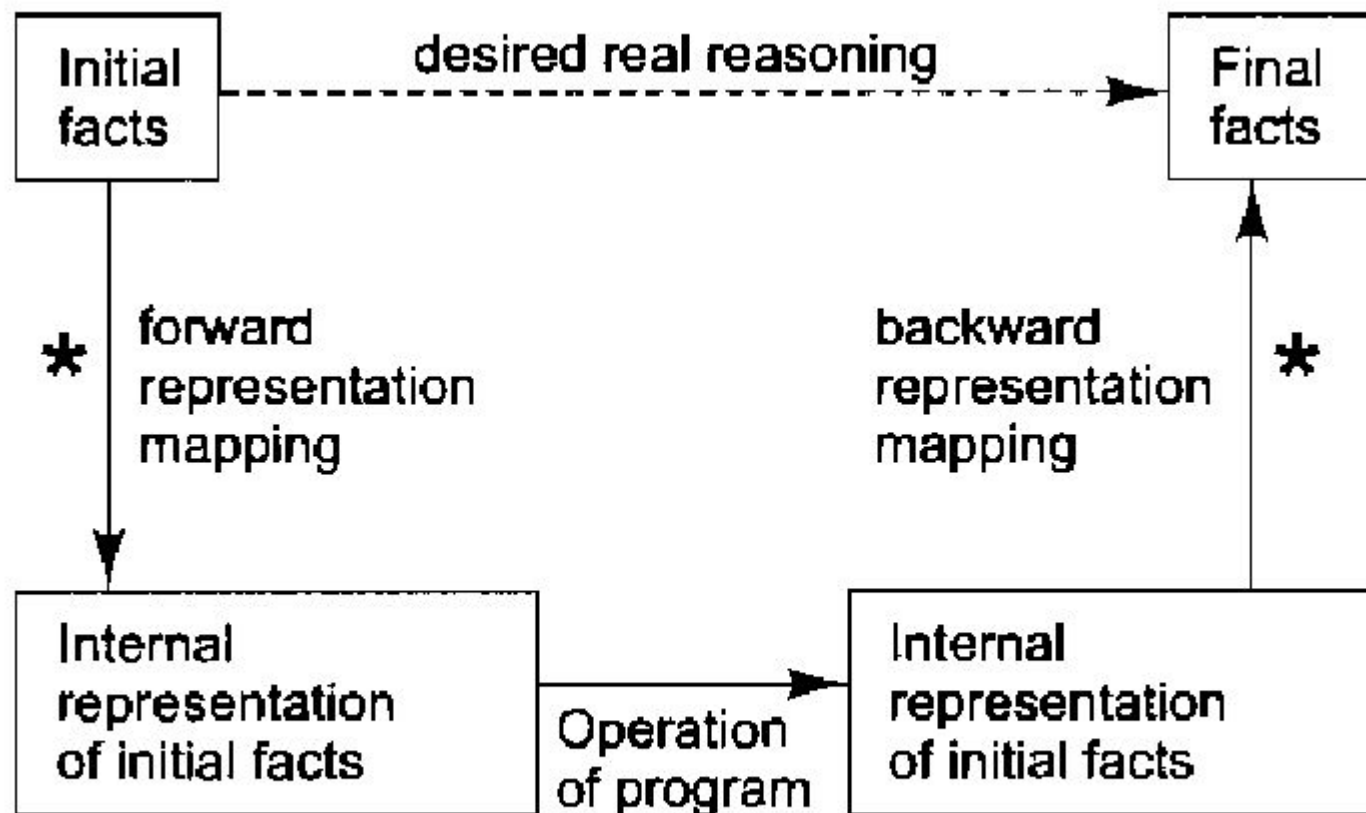


Fig. 4.3 *Representation of Facts*

Cont ..

- ideally , the backward mapping is many-to-one(function) and onto (every fact is mapped to at least one representation).
- Not always the case

Approaches to Knowledge representation

- Good representation of knowledge in a particular domain should exhibit the following four properties:
 - **Representational Adequacy** - ability to represent all kinds of knowledge in that domain
 - **Inferential Adequacy** - ability to manipulate the representational structure to derive new structures that corresponds to new knowledge derived from the old.
 - **Inferential Efficiency** - ability to incorporate into the knowledge structure additional information that can be used to guide the inference mechanism in promising directions
 - **Acquisitional Efficiency** - ability to obtain new information easily.
- **Simple relational knowledge (RDBMS - hard to make inferences)**
- **Inheritable Knowledge (Classes, instances, inheritance),**
- **Inferential knowledge (predicate logic),**
- **Procedural Knowledge (specifies control flow)**

Logic representation

- We will focus on the use of using Logic for our representation.
- This is because it allows for a very powerful method to infer new knowledge - mathematical deduction.
- **Proposition :- Statement that evaluates to either true or false**
- **Predicate :- Statement that is true or false based on the value of its arguments;**
 - **function $P:X \rightarrow \{true,false\}$**

Representing Simple Facts in Logic

- Use of propositional logic to represent Knowledge
- Easy to represent real-world knowledge as *well-formed formulas (wff)*

It is raining.

RAINING

It is sunny.

SUNNY

It is windy.

WINDY

If it is raining, then it is not sunny.

RAINING $\rightarrow \neg$ SUNNY

Cont ..

- How to represent
 - John is a man
 - JOHNMAN
 - Luke is a man
 - LUKEMAN
- The above representation makes it difficult to infer anything, such as the similarities between John and Luke.
- Instead we can represent the above as:
 - man(john)
 - man(luke)
 - But this require the use of predicates applied to some arguments

Cont ..

- Problem
 - How do we represent something like
 - “All men are mortal”
 - MORTALMAN - propositional logic representation
 - Mortal is a man?
 - mortal(man) - predicate logic representation
 - “man” is mortal?
 - Use a relation
 - $\forall x:\text{man}(x) \rightarrow \text{mortal}(x)$
- Cannot represent **objects** or **quantifiers** using propositional logic.

First-order Predicate Logic

- To overcome the problem faced by the use of propositional logic
 - We use first-order predicate logic
- Use of *statements with arguments as well-formed formulas*
- Easier to reason with knowledge using propositional logic rather than predicate logic
 - Problems expressed in propositional logic are *decidable*.
 - There exists a decision procedure that returns the correct answer within a finite set of steps
 - Problems expressed in predicate logic are *semi-decidable*.
 - There exist a decision procedure that returns the correct answer if the answer is true, it may give no answer if the answer is false.

**Difference between Propositional and Predicate Logic

- In propositional logic the "atoms" are propositions, with them you can always build new ones using the logical connectives, for example you could have a proposition p which means "there is a dog in Luca's house" and another proposition q which means "Luca is Sandra's boyfriend", now you can say "there is a dog in Sandra's boyfriend's house" in the language of propositional logic in the following way: $p \wedge q$
- In predicate logic you can "break" those "atoms" and work with the "subatomic particles", and so this form of logic allows us to analyze the internal structure of the propositions. Now you can use quantifiers, terms, relations and functions. You can define, for example: $d(x)$.
 - means "x is a dog" $h(x,y)$ means "x is in y's house" $b(x)$ means "the boyfriend of x" and finally define Sandra as the constant s , now your proposition would be:

$$\exists x(d(x) \wedge h(x,b(s)))$$

Predicate Logic EXAMPLE - pg.100

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Romans were either loyal to Caesar or hated him.
6. Everyone is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.

Predicate representations

1. **man(marcus)**
2. **pompeian(marcus)**
3. **$\forall x: \text{pompeian}(x) \rightarrow \text{roman}(x)$**
4. **ruler(caesar)** { * what if there were more than one person named caesar }
5. **$\forall x: \text{roman}(x) \rightarrow \text{loyalto}(x, \text{caesar}) \vee \text{hate}(x, \text{caesar})$** { *implies that a roman can both hate and be loyal to caesar - OR operation }
 - a. **$\forall x: \text{roman}(x) \rightarrow [\text{loyalto}(x, \text{caesar}) \wedge \neg \text{hate}(x, \text{caesar})] \vee [\neg \text{loyalto}(x, \text{caesar}) \wedge \text{hate}(x, \text{caesar})]$**
{XOR operator}
6. **$\forall x: \rightarrow y: \text{loyalto}(x, y)$**
 - a. **$\exists y: \forall x: \text{loyalto}(x, y)??$**
 - b. **$\forall x: \exists y: \text{loyalto}(x, y)$** [We will use this]
7. **$\forall x: \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassasinate}(x, y) \rightarrow \neg \text{loyalto}(x, y)$**
 - a. May mean that people **always** try to assassinate rulers they are not loyalto
 - b. Or may mean that people **may** attempt to assassinate rulers they are not loyal to
8. **tryassasinate(marcus, caesar)**

Cont ..

- We want to know if
 - Marcus was loyal to Caesar
 - We need to check $\neg \text{loyalto}(\text{marcus}, \text{caesar})$
- Computes backwards.

$$\begin{array}{rcl}
 \neg \text{loyalto}(\text{Marcus}, \text{Caesar}) & & \\
 \uparrow & (7, \text{substitution}) & \\
 \text{person}(\text{Marcus}) \wedge & & \\
 \text{ruler}(\text{Caesar}) \wedge & & \\
 \text{tryassassinate}(\text{Marcus}, \text{Caesar}) & & \\
 \uparrow & (4) & \\
 \text{person}(\text{Marcus}) & & \\
 \text{tryassassinate}(\text{Marcus}, \text{Caesar}) & & \\
 \uparrow & (8) & \\
 \text{person}(\text{Marcus}) & &
 \end{array}$$

Fig. 5.2 *An Attempt to Prove $\neg \text{loyalto}(\text{Marcus}, \text{Caesar})$*

We are stuck as we have no way of proving Marcus was a person.
 We have to add another fact : $\forall x: \text{man}(x) \rightarrow \text{person}(x)$

Issues

- English sentences are ambiguous
- Different ways to represent knowledge
 - marcus was a man = man(marcus)
 - Fails to capture past tense
- Sometime simple facts are omitted, such as the one seen in the last slide (“all men are people”) which may cause issues.
- Which to check?
 - \neg **loyalto(marcus,caesar)**
 - **loyalto(marcus,caesar)**

Computable Functions and Predicates

- Computable predicates
- Suppose the number of facts for a particular predicate is very large
 - Example
 - `gt(5,2)`
 - `gt(3,1)`
 - `gt(2,1)`
 - ...
- Computable predicates allow us to replace *easy to compute* predicates such as the greater than predicate above by a procedure to calculate them rather than trying to infer them or trying to map them out one by one.

Cont ..

- Computable functions
- Example
 - `gt(2+3,4)`
- Computable function, in our case the '+' sign, needs to sum the value of 2 and 3 and then send 5 and 4 as the arguments to `gt`.

Why do we need computable functions and predicates?

1. $\text{born}(x,t)$

2. $\text{died}(x,t)$

3. $\text{dead}(x,t)$

4. $\text{mortal}(x)$

5. $\text{man}(x)$

1. 'x' was born in year 't'

2. 'x' died in year 't'

3. 'x' is dead at year 't'

4. 'x' is mortal

5. 'x' is a man

- Set of predicates

- Let's say we want to represent this knowledge in predicate logic

- No mortal lives more than 150 years

- $\forall x: \forall t1: \forall t2: \text{born}(x,t1) \wedge \text{mortal}(x) \wedge \text{gt}(t2 - t1, 150) \rightarrow \text{dead}(x,t2)$

1. $man(Marcus)$
2. $Pompeian(Marcus)$
3. $born(Marcus, 40)$
4. $\forall x : man(x) \rightarrow mortal(x)$
5. $\forall x : Pompeian(x) \rightarrow died(x, 79)$
6. $erupted(volcano, 79)$
7. $\forall x : \forall t_1 : \forall t_2 : mortal(x) \wedge born(x, t_1) \wedge gt(t_2 - t_1, 150) \rightarrow dead(x, t_2)$
8. $now = 1991$
9. $\forall x : \forall t. [alive(x, t) \rightarrow \neg dead(x, t)] \wedge [\neg dead(x, t) \rightarrow alive(x, t)]$
10. $\forall x : \forall t_1 : \forall t_2 : died(x, t_1) \wedge gt(t_2, t_1) \rightarrow dead(x, t_2)$

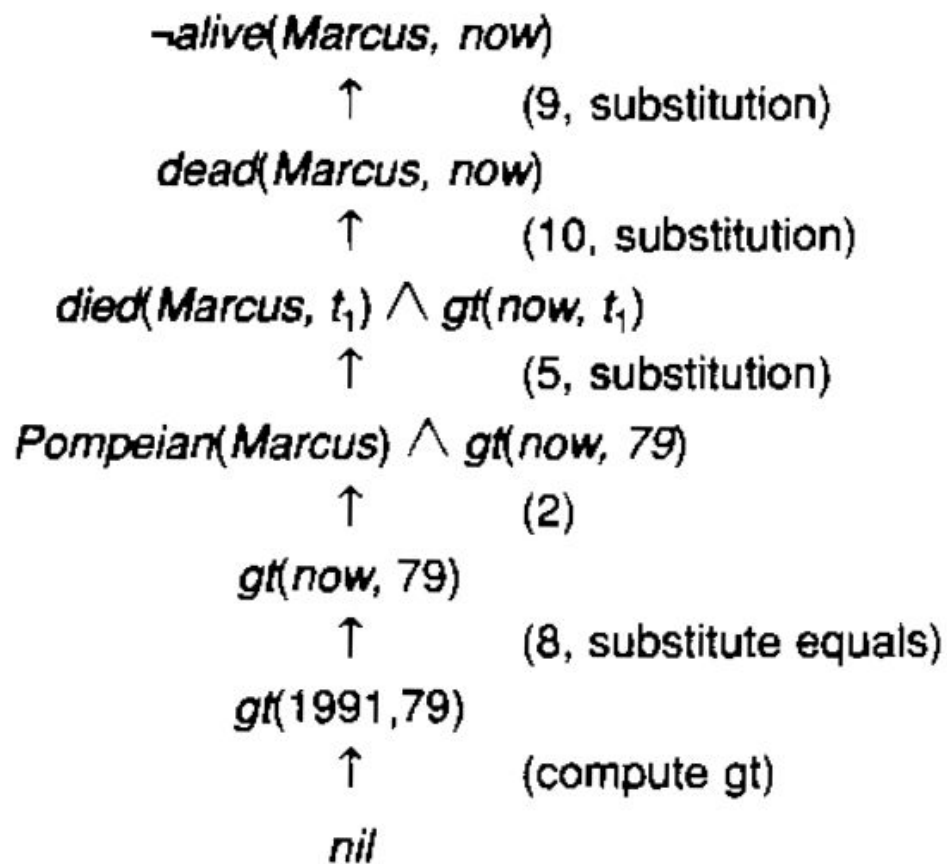


Fig. 5.5 *One Way of Proving That Marcus Is Dead*

Cont ..

- Even simple problems require many steps to prove
- Requires substitutions, matching , *modus ponens*, *modus tollens*, etc.

Resolution

- Proof by *refutation*
 - Contradiction
 - *To prove that a statement is true, resolution first negates the statement and attempts to produce a contradiction.*
 - Say we want to prove A , then resolution will first assume $\neg A$ (A is not true) and carry out a set of steps till we get a contradiction.
 - If $\neg A$ produces a contradiction $\Rightarrow A$ is true.
- Operates on statements that have been converted to **Clause form**

Resolution in Propositional Logic

- Need to convert statements to clause form
 - Use normal forms
 - CNF (Conjunctive Normal Form) - conjunction of disjunctions
 - $(P \vee Q \vee A) \wedge (P \vee A)$
 - DNF(Disjunctive Normal Form) - disjunction of conjunctions
 - $(P \wedge Q \wedge A) \vee (P \wedge A)$
- **Disjunction of literals is called a Clause**
 - $P \vee Q \vee A$
 - $P \vee Q$
 - P

Inference rules:

Commutative	$p \wedge q \iff q \wedge p$	$p \vee q \iff q \vee p$
Associative	$(p \wedge q) \wedge r \iff p \wedge (q \wedge r)$	$(p \vee q) \vee r \iff p \vee (q \vee r)$
Distributive	$p \wedge (q \vee r) \iff (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \iff (p \vee q) \wedge (p \vee r)$
Identity	$p \wedge T \iff p$	$p \vee F \iff p$
Negation	$p \vee \sim p \iff T$	$p \wedge \sim p \iff F$
Double Negative	$\sim(\sim p) \iff p$	
Idempotent	$p \wedge p \iff p$	$p \vee p \iff p$
Universal Bound	$p \vee T \iff T$	$p \wedge F \iff F$
De Morgan's	$\sim(p \wedge q) \iff (\sim p) \vee (\sim q)$	$\sim(p \vee q) \iff (\sim p) \wedge (\sim q)$
Absorption	$p \vee (p \wedge q) \iff p$	$p \wedge (p \vee q) \iff p$
Conditional	$(p \implies q) \iff (\sim p \vee q)$	$\sim(p \implies q) \iff (p \wedge \sim q)$

Modus Ponens	$p \implies q$ p $\therefore q$	Modus Tollens	$p \implies q$ $\sim q$ $\therefore \sim p$
Elimination	$p \vee q$ $\sim q$ $\therefore p$	Transitivity	$p \implies q$ $q \implies r$ $\therefore p \implies r$
Generalization	$p \implies p \vee q$ $q \implies p \vee q$	Specialization	$p \wedge q \implies p$ $p \wedge q \implies q$
Conjunction	p q $\therefore p \wedge q$	Contradiction Rule	$\sim p \implies F$ $\therefore p$

Example :
 assamese(x) -> indian(x)

Modus Ponens
 assamese(Jim) ->
 indian(Jim)

Modus Tollens
 \sim assamese(Jim)
 Does not imply
 \sim indian(Jim)

\sim indian(Jim) ->
 \sim assamese(Jim)

Procedure to convert a statement to CNF

1. Eliminate implications and biconditionals using formulas:

- $(P \Leftrightarrow Q) \Rightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$
- $P \rightarrow Q \Rightarrow \neg P \vee Q$

2. Apply De-Morgan's Law and reduce NOT symbols so as to bring negations before the atoms. Use:

- $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$
- $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q$

3. Use distributive and other laws & equivalent formulas to obtain Normal forms.


Conversion to CNF example

Q. Convert into CNF : $((P \rightarrow Q) \rightarrow R)$

Solution:

$$\begin{aligned}\text{Step 1: } ((P \rightarrow Q) \rightarrow R) &\implies ((\neg P \vee Q) \rightarrow R) \\ &\implies \neg(\neg P \vee Q) \vee R\end{aligned}$$

$$\text{Step 2: } \neg(\neg P \vee Q) \vee R \implies (P \wedge \neg Q) \vee R$$

$$\text{Step 3: } (P \wedge \neg Q) \vee R \implies (P \vee R) \wedge (\neg Q \vee R)$$


CNF

Cont ..

- Up until now we have started with the statement we want to prove and used substitution, matching and 'eliminated' true facts from our equation till we arrive at an empty state that suggests that we have no more statements to prove.
- Resolution works in a different way.
 - As mentioned previously, it starts with the negation of the statement we wish to prove and attempts to arrive at a contradiction

Basics of Resolution

- Iterative process, where in each step
 - Two clauses are compared (**resolved**), called **parent clauses**
 - New clause is inferred from the comparison.
 - Shows how the parent clauses interact with each other
- EXAMPLE
 - $\text{winter} \vee \text{summer}$
 - $\neg \text{winter} \vee \text{cold}$

Example Cont ..

- For the two clauses in our system, only **winter** or \neg **winter** can be true at a time.
 - If **winter** is **true** then **cold** has to be true in our second clause
 - If \neg **winter** is **true** then **summer** has to be true in our first clause
 - So our inference from the interaction of the two clauses is:
 - $\text{summer} \vee \text{cold}$

Basics of Resolution

- Resolution works by taking two parent clauses that contain the same literal, one in positive form and one that is negated.
- The **resolvent** is obtained by combining the literals of the parent clauses minus the ones that cancel out.
- If the **resolvent** is an empty clause, then we have a contradiction.
- Problem: How would this work with predicate logic when each predicate argument may take an infinite amount of values?
 - Based on **Herbrand's Theorem** [Chang and Lee, 1973]

Resolution in Propositional Logic Algorithm

Let **F** be the set of axioms and **P** be the proposition that we wish to prove.

1. Convert the axioms (F) to clause form
2. Negate P and convert to clause form. Add it to the set of Clauses obtained in 1.
3. Loop until contradiction or no progress can be made.
 - a. Select two clauses (parent clauses)
 - b. Resolve the parent clauses which results in the resolvent
 - The resolvent is a disjunction of all the literals present in the parent clauses except for literals that cancel each other out
 - c. If the resolvent is an empty clause
 - Quit and return a contradiction
 - d. Else
 - Add the resolvent to the original set of clauses

Given the set of axioms, we want to prove R

Given Axioms	Converted to Clause Form	
P	P	(1)
$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$	(2)
$(S \vee T) \rightarrow Q$	$\neg S \vee Q$	(3)
	$\neg T \vee Q$	(4)
T	T	(5)

Fig. 5.7 *A Few Facts in Propositional Logic*

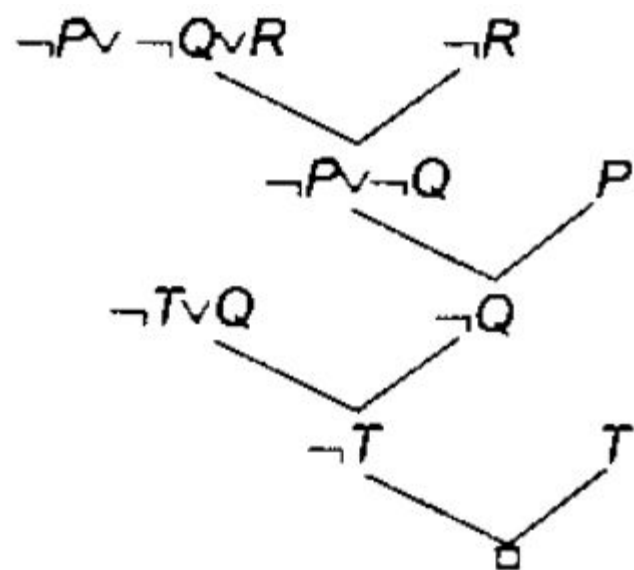


Fig. 5.8 *Resolution in Propositional Logic*

Resolution in Predicate Logic

- First we need to learn how to convert predicate logic representations to Clause form
- This is a little more complicated than in propositional logic as we have to consider how to convert quantifiers as well.

Conversion to Clause Form

1. Eliminate \rightarrow .

$$P \rightarrow Q \equiv \neg P \vee Q$$

2. Reduce the scope of each \neg to a single term.

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

$$\neg \forall x: P \equiv \exists x: \neg P$$

$$\neg \exists x: P \equiv \forall x: \neg P$$

$$\neg \neg P \equiv P$$

3. Standardize **variables** so that each quantifier binds a unique variable.

$$(\forall x: P(x)) \vee (\exists x: Q(x)) \equiv$$

$$(\forall x: P(x)) \vee (\exists y: Q(y))$$

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

$$(\forall \mathbf{x}: P(\mathbf{x})) \vee (\exists \mathbf{y}: Q(\mathbf{y})) \equiv \\ \forall \mathbf{x}: \exists \mathbf{y}: (P(\mathbf{x}) \vee (Q(\mathbf{y})))$$

5. Eliminate \exists (Skolemization).

$$\exists \mathbf{x}: P(\mathbf{x}) \equiv P(\mathbf{c}) \quad \text{Skolem constant}$$

$$\forall \mathbf{x}: \exists \mathbf{y} P(\mathbf{x}, \mathbf{y}) \equiv \forall \mathbf{x}: P(\mathbf{x}, \mathbf{f}(\mathbf{x})) \quad \text{Skolem function}$$

6. Drop \forall .

$$\forall \mathbf{x}: P(\mathbf{x}) \equiv P(\mathbf{x})$$

- Convert the formula into a **conjunction of disjuncts**.

$$(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$$

- Create a **separate clause** corresponding to each conjunct.
- Standardize apart the **variables** in the set of obtained clauses.

Step 9: rename the variables so that no two variables have the same name.
 I.e., for this to apply, we assume that

$$(\forall x : P(x) \wedge Q(x)) = \forall x : P(x) \wedge \forall x : Q(x)$$

Unification Algorithm

- In propositional logic, it's easy to see if two literals cancel out i.e.,
 - L and $\neg L$ cancel each other out.
- In predicate logic its is much more difficult
 - $\text{man}(\text{jim})$ and $\neg \text{man}(\text{jim})$ cancel out
 - $\text{man}(\text{jim})$ and $\neg \text{man}(\text{spot})$ do not cancel out
- To match in predicate logic
 - The predicate must match
 - Each substitution of the predicates arguments must match
- The algorithm used for this matching is called the ***unification algorithm***

Cont ..

- We want to unify the following:
 - $P(x,x)$
 - $P(y,z)$
- The predicate matches, now we need to make consistent substitutions for the arguments (goes from left to right).
 - We can substitute y for x (written as y/x) in our first argument
 - $P(y, x)$
 - $P(y, z)$
 - Now we can try and substitute z for x (z/x)
 - But this means we have to substitute x for y and z which is inconsistent

Cont ..

- Instead we need to carry out the substitution throughout the arguments.
- Say we substitute y for x (y/x), if we carry out the substitution throughout the arguments, we would get,
 - $P(y, y)$
 - $P(y, z)$
- Now, we can substitute z for y (z/y) and we would have a consistent substitution i.e., the two literals ($P(x,x)$ and $P(y,z)$) are ***unifiable***.
- The substitutions are written as:
 - $(z/y)(y/x)$
 - Apply substitution from right to left

Note: The full algorithm is in the book by E.Rich and K.Knight in page - 115 to 116

Resolution in Predicate Logic Algorithm

Let **F** be the set of axioms and **P** be the proposition that we wish to prove.

1. Convert the axioms (F) to clause form
2. Negate P and convert to clause form. Add it to the set of Clauses obtained in 1.
3. Loop until contradiction or no progress can be made.
 - a. Select two clauses (parent clauses)
 - b. Resolve the parent clauses which results in the resolvent
 - The resolvent is a disjunction of all the literals present in the parent clauses except for unifiable literals T1 and T2 such that T1 is in one parent and $\neg T2$ is in the other parent.
 - c. If the resolvent is an empty clause
 - Quit and return a contradiction
 - d. Else
 - Add the resolvent to the original set of clauses

Knowledge Representation using Rules

- Rule based knowledge systems play a very important role in AI
- We have discussed Rules and how they are used in our search techniques and production system
- But we have yet to describe how rules are represented.
- rules and searching were described as separate entities
- Rules represent both knowledge about the relationships in the world and also knowledge about how to solve problems.

Procedural vs Declarative Knowledge

- We use logic representations as a basis for this discussion
- We have viewed logical assertions as a **declarative** representation of knowledge
 - The knowledge is specified but how that knowledge is used is not specified.
 - Must be used alongside a program that knows what to do with the knowledge, such as a resolution theorem prover.
 - In this case we view logical assertions (axioms) as data to a program
- View logical assertions as a *program* itself or **Procedural** knowledge.
 - **Atomic assertions** (facts) can be viewed as starting points
 - While **implications** can be viewed as a reasoning path (similar to a control structure).

Cont ..

- **Declarative** - specify logic but not control flow.
 - **Definition** - declarative representation is one in which knowledge is specified but the use to which that knowledge is put is not given.
- **Procedural** - specifies control flow with logic.
 - **Definition** - procedural representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.

Axioms

man(Marcus)

man(Caesar)

person(Cleopatra)

$\forall x : \text{man}(x) \rightarrow \text{person}(x)$

ANSWERS

y = Marcus

y = Caesar

y = Cleopatra

What we want to find.

$\exists y : \text{person}(y)$

- If our program is viewed **declaratively**, any of the above answers could be selected.
- If the program is viewed **procedurally**, and the assertions are examined in the order they appear, then our program will return with the answer:
 - *y = Cleopatra*

What is the answer our program will return if we assume the same examination method i.e., in the order they appear in the program???

man(Marcus)

man(Caesar)

$\forall x : \textit{man}(x) \rightarrow \textit{person}(x)$

person(Cleopatra)

What is the answer our program will return if we assume the same examination method i.e., in the order they appear in the program???

man(Marcus)

man(Caesar)

$\forall x : \textit{man}(x) \rightarrow \textit{person}(x)$

person(Cleopatra)

- $y = \textit{Marcus}$
- The implication rule will be reached first and the program will then try to find $\textit{man}(x)$.

Logic Programming

- Views logical assertions as a program.
- Focus is on PROLOG.
- A PROLOG program is described as a series of logical assertions.
 - Each assertion is in the form of a ***Horn Clause***
 - **Horn Clause** is a clause with at most one positive literal.
 - The advantage of using Horn clause is that the logic is **decidable**.

$\forall x : \text{pet}(x) \wedge \text{small}(x) \rightarrow \text{apartmentpet}(x)$

$\forall x : \text{cat}(x) \vee \text{dog}(x) \rightarrow \text{pet}(x)$

$\forall x : \text{poodle}(x) \rightarrow \text{dog}(x) \wedge \text{small}(x)$

$\text{poodle}(\text{fluffy})$

A Representation In Logic

$\text{apartmentpet}(X) \text{ :- } \text{pet}(X), \text{small}(X).$

$\text{pet}(X) \text{ :- } \text{cat}(X).$

$\text{pet}(X) \text{ :- } \text{dog}(X).$

$\text{dog}(X) \text{ :- } \text{poodle}(X).$

$\text{small}(X) \text{ :- } \text{poodle}(X).$

$\text{poodle}(\text{fluffy}).$

A Representation in PROLOG

Facts about PROLOG

- PROLOG programs, although declarative, use a control structure that is procedural
 - I.e., the assertions are tested in the order the assertions are written in the program.
- The input to the program is a goal to be proved.
- Backward reasoning is applied.
- Program is read
 - Top to bottom
 - Left to right
 - Depth-First Search is used for searching with backtracking.
- Logical Assertions contain **facts** (no variables) and **rules** (implications - they contain variables)

1. In logic, variables are explicitly quantified. In PROLOG, quantification is provided implicitly by the way the variables are interpreted (see below). The distinction between variables and constants is made in PROLOG by having all variables begin with upper case letters and all constants begin with lower case letters or numbers.
2. In logic, there are explicit symbols for *and* (\wedge) and *or* (\vee). In PROLOG, there is an explicit symbol for *and* ($,$), but there is none for *or*. Instead, disjunction must be represented as a list of alternative statements, any one of which may provide the basis for a conclusion.
3. In logic, implications of the form "*p* implies *q*" are written as $p \rightarrow q$. In PROLOG, the same implication is written "backward," as $q :- p$. This form is natural in PROLOG because the interpreter always works backwards from a goal, and this form causes every rule to begin with the component that must therefore be matched first. This first component is called the *head* of the rule.

Control Strategy of Prolog

- Start with goal.
- If a fact can prove the rule directly, then done.
- Else look for rules whose head (consequent) matches the goal.
- Matching is done via unification.
- Backward reasoning is used until a path is found from the goal to a fact.
- Uses DFS with backtracking
- At each choicepoint (guessing a variables value), consider values in the order they appear in the program.
- If goal is made up of conjunctive parts, each part is proven in order.

Negations in PROLOG

- Negation cannot be represented explicitly in PROLOG

$$\forall x : dog(x) \rightarrow \neg cat(x)$$

- Instead PROLOG uses a strategy called ***negation as failure***
- Which means that if a statement cannot be proven, it will be considered FALSE.
 - cat(fluffy). - returns FALSE
 - cat(mittens). - also returns FALSE, even though the program knows nothing about “mittens”.

FORWARD VS BACKWARD REASONING

- Forward Reasoning
 - Start from initial state(s) to goal state
- Backward Reasoning
 - Start from goal state to initial state

Forward Reasoning

- Start with initial state(s) as the root of the tree.
- Begin generating the next level of the tree by matching the **left side** of rules to the root node and using the **right side** to generate the new states.
- Continue this process for each new generated node until a goal state is reached.
- Can be directed by incoming data.
- Read **Forward-Chaining System**

Backward Reasoning

- Start with goal state as the root of the tree.
- Begin generating the next level of the tree by matching the **right side** of rules to the root node and using the **left side** to generate the new states.
- Continue this process for each new generated node until an initial state is reached.
- Directed by the goal and hence called *goal directed reasoning*.
- Read **Backward-Chaining System**

Forward vs Backward Reasoning

- The choice of forward or backward reasoning may impact the number of paths being explored.
- Read About “**Combining Forward AND Backward Reasoning**”
- Are there more possible start states or goal states? We would like to move from the smaller set of states to the larger (and thus easier to find) set of states.
- In which direction is the branching factor (i.e., the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
- Will the program be asked to justify its reasoning process to a user? If so, it is important to proceed in the direction that corresponds more closely with the way the user will think.
- What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning makes sense. If it is a query to which a response is desired, backward reasoning is more natural.

Matching

- How to select an applicable rule to the current state?
- Simple way is to run through each rule and see if the current state satisfies the preconditions
 - a. Can be very slow for large problems that require many rules.
 - b. Sometimes rules cannot be matched to current state very easily.
- One way to solve “a” is by using **indexing**.
 - a. An example is if we use a state-to-state definition of rules in chess (shown in next figure)
 - b. We can represent each board configuration (or state) as a large number and then use a simple **hash function** to map from the large number to an index pointing to the rule(s).
 - c. Every rule applicable to a specific state will be mapped to the same index.

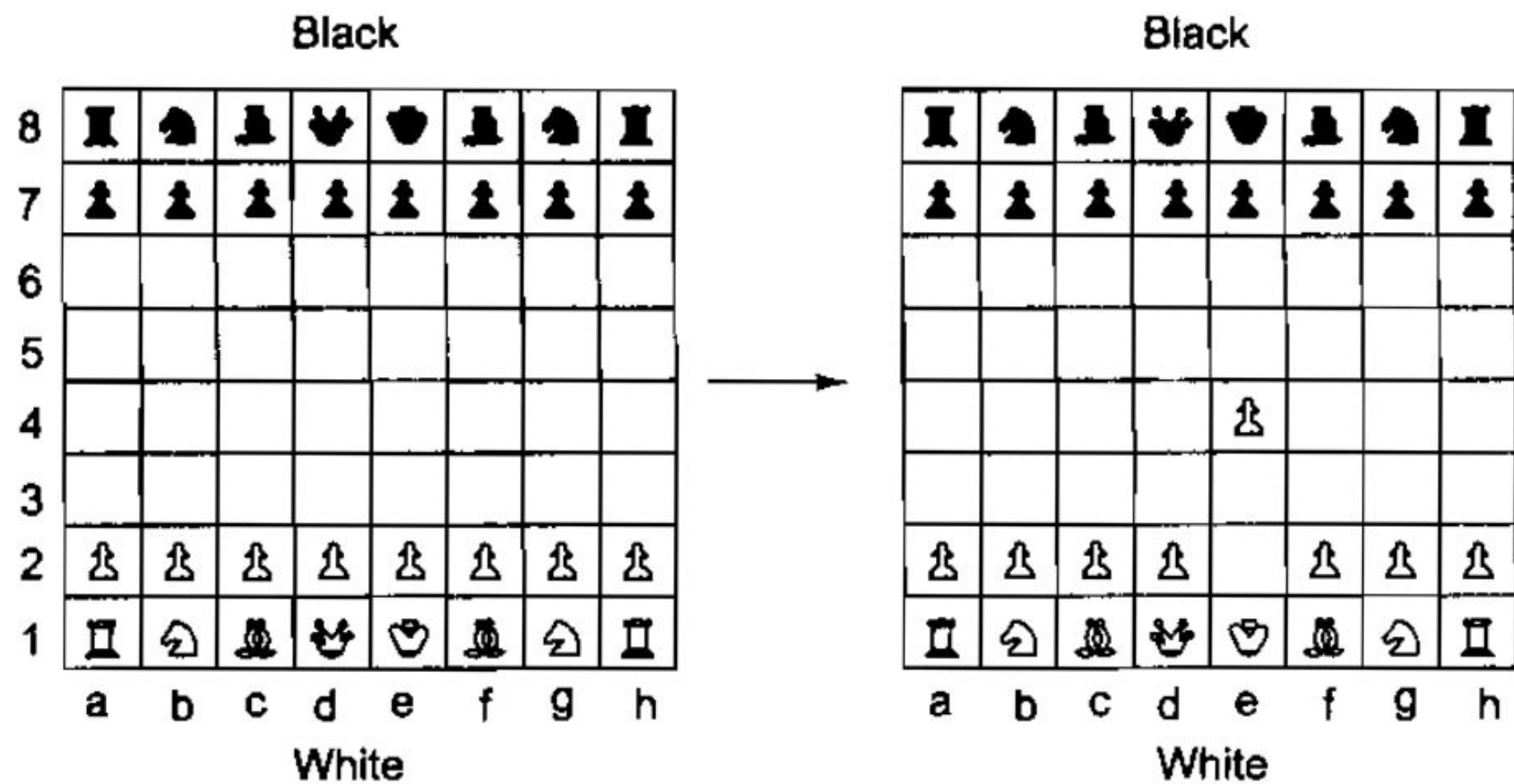


Fig. 6.4 *One Legal Chess Move*

White pawn at		
Square(file e, rank 2)		
AND		
Square(file e, rank 3)	→	move pawn from
is empty		Square(file e, rank 2)
AND		to Square(file e, rank 4)
Square(file e, rank 4)		
is empty		

Fig. 6.5 *Another Way to Describe Chess Moves*

Instead of the above easier way to write rules..

Matching

- PROLOG indexes its rules by the predicates the rule contains.
- Matching with Variables
 - For simple condition to single element, we can use unification.
 - Inefficient.
 - works for only one rule, but we want the match to return all rules whose preconditions the current state satisfies.
- Complex and Approximate matching - read pg. 140 (E.Rich & K.Knight)
- Conflict resolution - read pg. 141 (E.Rich & K.Knight)

Control Knowledge

- Core of AI is Search
- But search is intractable, must use knowledge to restrict search
- This knowledge about which paths will most likely lead us to a goal state is called **search control knowledge**.

Cont ..

1. Knowledge about which states are more preferable to others.
2. Knowledge about which rule to apply in a given situation.
3. Knowledge about the order in which to pursue subgoals.
4. Knowledge about useful sequences of rules to apply.

Assignment 2 - question 3.

1. Consider the following knowledge base:

$\forall x : \forall y : cat(x) \wedge fish(y) \rightarrow likes - to - eat(x,y)$

$\forall x : calico(x) \rightarrow cat(x)$

$\forall x : tuna(x) \rightarrow fish(x)$

tuna(Charlie)

tuna(Herb)

calico(Puss)

- (a) Convert these wff's into Horn clauses.
- (b) Convert the Horn clauses into a PROLOG program.
- (c) Write a PROLOG query corresponding to the question, "What does Puss like to eat?" and show how it will be answered by your program.
- (d) Write another PROLOG program that corresponds to the same set of wff's but returns a different answer to the same query.

Game Playing

Overview

- Game playing has had a considerable amount of focus in AI
- Arthur Samuel was the first to create a program in the early 1960's that could play checkers (and also learn from its mistakes)
- Why is game playing a good domain for AI?
 - Easy to determine win or loss (success or failure)
 - Does not require a large volume of knowledge
 - This second point, as we know now, is not true. Just playing chess requires a vast amount of knowledge.

Cont ..

- All AI solutions require Searching
- We can look at search procedures as a “Generate-and-Test” procedure
 - Where there is varying amount of work done by the generator before testing commences.
- Extremes of Generate-and-Test algorithm:
 - Generate whole solutions and then test.
 - Generate nodes after a single move and then test the generated node.
- To improve on the above procedures, we need to improve both the **generator** and the **tester**.
 - Generate only good moves [**plausible move generator**]
 - Test will recognize best moves and those will be explored first. (Example: Heuristic Search)

Cont ..

- If the **tester** is fast, it will not be very accurate and vice-versa
- To get a more accurate read on the desirability of a state/node, we need a more accurate tester, which means it will be slower.
- Thus we can reduce the amount of nodes being tested by coming up with a “smarter” **generator** called the ***plausible-move generator*** (using some heuristic function again).
 - Plausible move generator generates promising move
 - Tester will have to test less nodes
 - More time can be spent accessing a node

Game playing problems

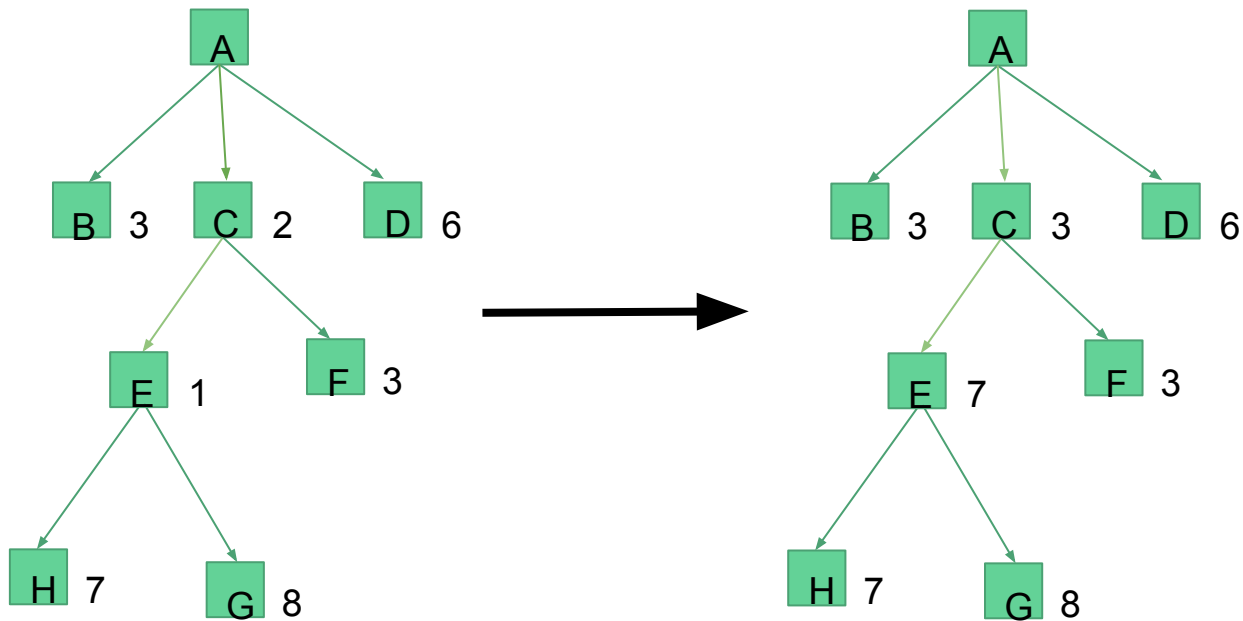
- It is not feasible to search all paths in the tree to find a goal state.
 - Usually a max number of ten to twenty moves (levels in the tree) can be accessed in the time given.
 - Each level of the tree in the case of a game playing AI solution is called a **ply** which signifies a turn.
- A game state or configuration is evaluated using a function similar to the heuristic function (h') called the ***static evaluation function***.
 - It is difficult to write a very accurate static evaluation function and hence should be applied as many layers (plys) down a tree as time permits.

Cont ..

- Two important knowledge-based components a game playing program are:
 - A good plausible-move generator.
 - A good static evaluation function.
- We also need a good way to search ahead as many moves as possible

Single Player Games

- We can use A^* algorithm to go as deep into the tree as possible.
- Then propagate the h' from the terminal nodes back up the tree



The left figures illustrate how we could use A^* to play a simple single player game.

Propagation of “best” * value is done when we reach terminal nodes.

*Minimizing

Two player Games

- This is not as simple as the single player game.
- Need to account for the moves of the opponent as well.
- The algorithm used to tackle this problem is called the **Minimax Algorithm**
 - Uses a maximizing function - higher value = better chances of winning
 - Maximize on the players turn
 - Minimizes on the opponent turn
 - Assumes that opponent plays optimally.

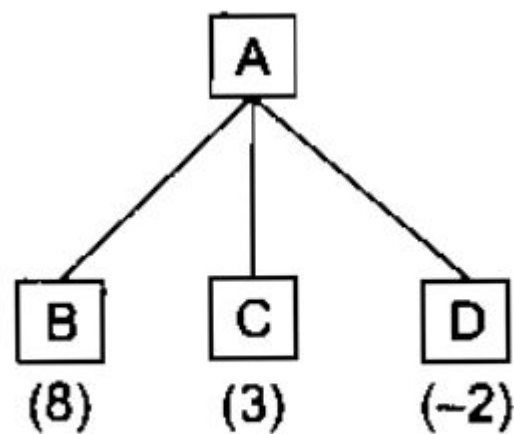


Fig. 12.1 *One-Ply Search*

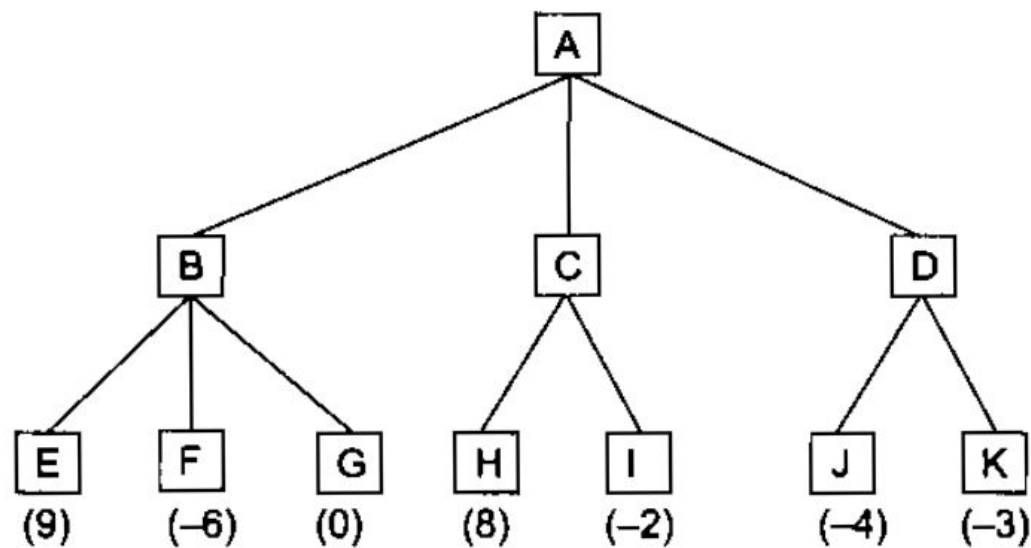


Fig. 12.2 *Two-Ply Search*

Minimax

- **MOVEGEN(Position, Player)** - the plausible-move generator, which returns a list of nodes representing the moves that can be made by *Player* in *Position*. We will call the two players **Maximizing_Player** and **Minimizing_Player**.
- **STATIC(Position, Player)** - the static evaluation function which returns a number representing the goodness of *Position* from the perspective of *Player*.

Cont ..

- One challenge of defining Minimax as a recursive function is that it needs to return two values, the back propagated **static evaluation function value** of the chosen path and the “best” **Path** (**path is not necessary, best node from current Position is enough).
- We will define the Minimax function as:
 - **MINIMAX(Position, Current_Depth, Player)**
- We stop the Minimax function if
 - It reaches a certain depth.
 - There are no more successors from the node (Position).
 - Use a function **DEEP_ENOUGH(Position, Depth)**, which return true if Depth has exceeded a certain level.

Algorithm: MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(*Position*, *Depth*), then return the structure

VALUE = STATIC(*Position*, *Player*);

PATH = nil

This indicates that there is no path from this node and that its value is that determined by the static evaluation function.

2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(*Position* *Player*) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.

For each element SUCC of SUCCESSORS, do the following:

- (a) Set RESULT-SUCC to

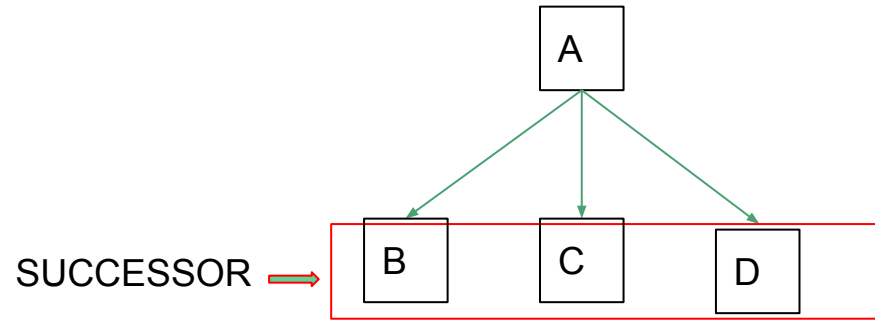
$\text{MINIMAX}(\text{SUCC}, \text{Depth} + 1, \text{OPPOSITE}(\text{Player}))$

This recursive call to MINIMAX will actually carry out the exploration of SUCC.

- (b) Set NEW-VALUE to $-\text{VALUE}(\text{RESULT-SUCC})$. This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
- (c) If $\text{NEW-VALUE} > \text{BEST-SCORE}$, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
- (i) Set BEST-SCORE to NEW-VALUE.
 - (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of $\text{PATH}(\text{RESULT-SUCC})$.

$\text{OPPOSITE}(\text{Player})$ - if Maximizing_Player, return Minimizing_Player and vice-versa

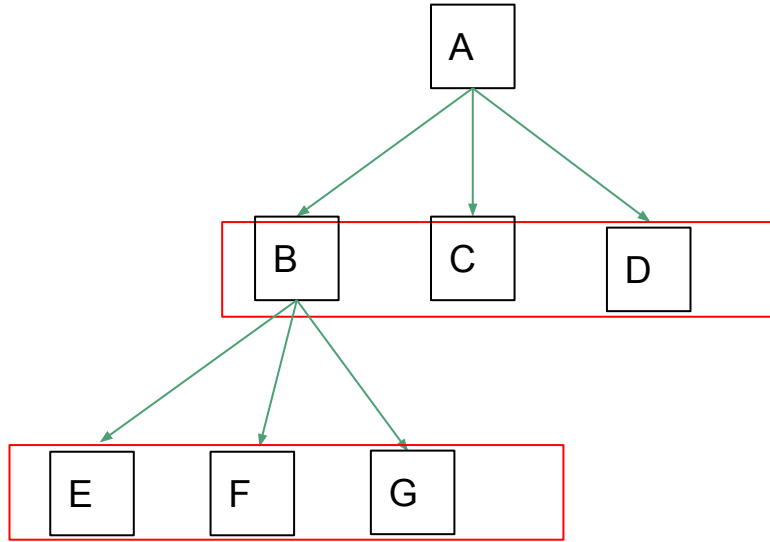
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
 VALUE = BEST-SCORE
 PATH = BEST-PATH



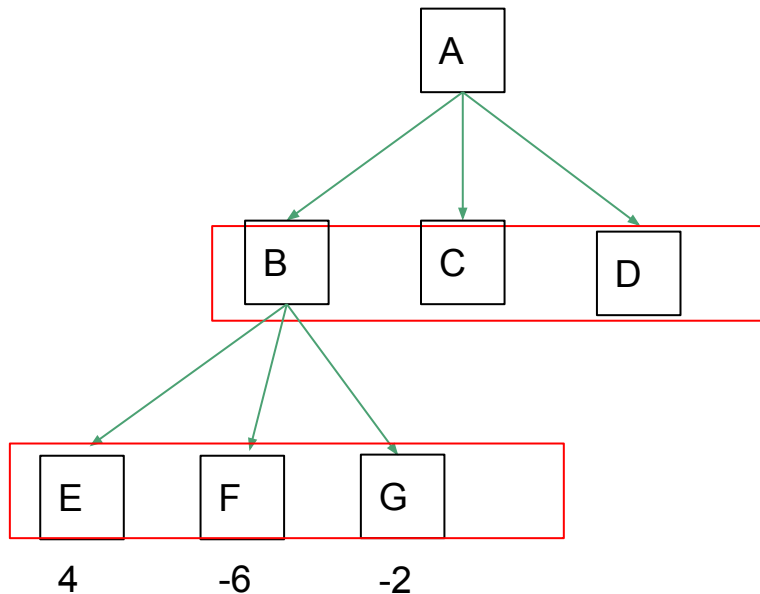
BEST_SCORE = -10

BEST_SCORE = -10

SUCCESSOR 



SUCCESSOR →

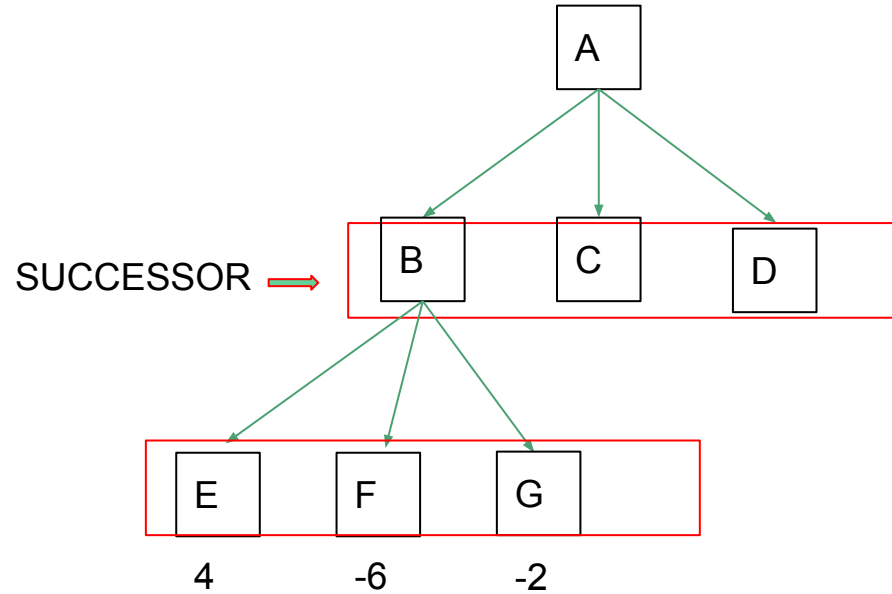


If DEEP_ENOUGH returns at depth = 2, then we calculate the static evaluation function of E, F and G

Here, we have to negate the values of the static evaluation function, as we are dealing with the **Minimizing_Player**

The values at E, F and G represent values with respect to maximizing_player

When returning to B, the NEW_VALUE from E, F and G respectively are -4, 6, 2



Propagate it back up one level.

B's value = - (value returned by minimax)

B = 6

When propagating back to A, the value will be negated again to represent the true value for maximizing player which is

B = -6

Next step will be to expand C from Successor list.

Alpha-Beta Cutoffs

- Improvement on Minimax algorithm
- Uses a branch-and-bound method to eliminate solutions that are worse than known solutions.
- Include two *bounds*, one for each player.
 - A **lower** bound for the **Maximizing player**, called the **alpha**
 - An **upper** bound for the **Minimizing player**, called the **beta**

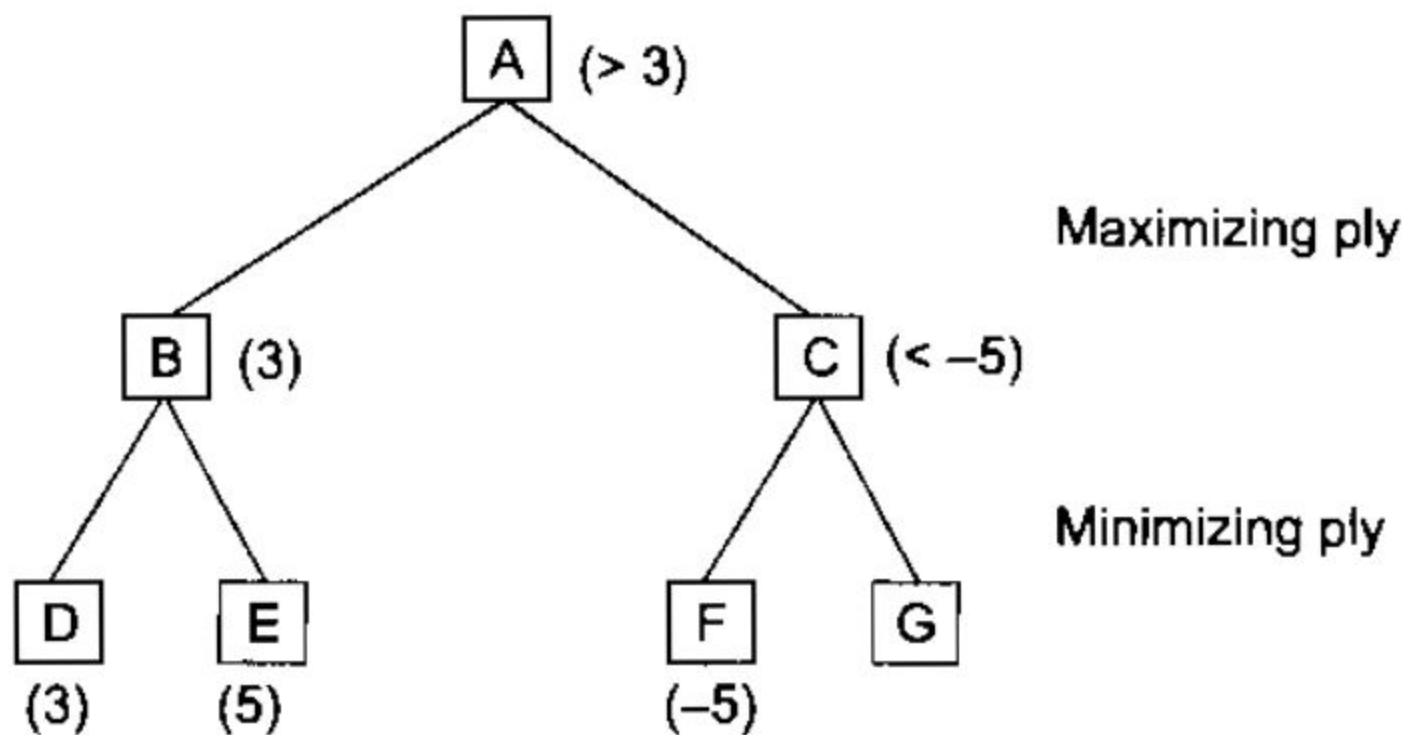


Fig. 12.4 *An Alpha Cutoff*

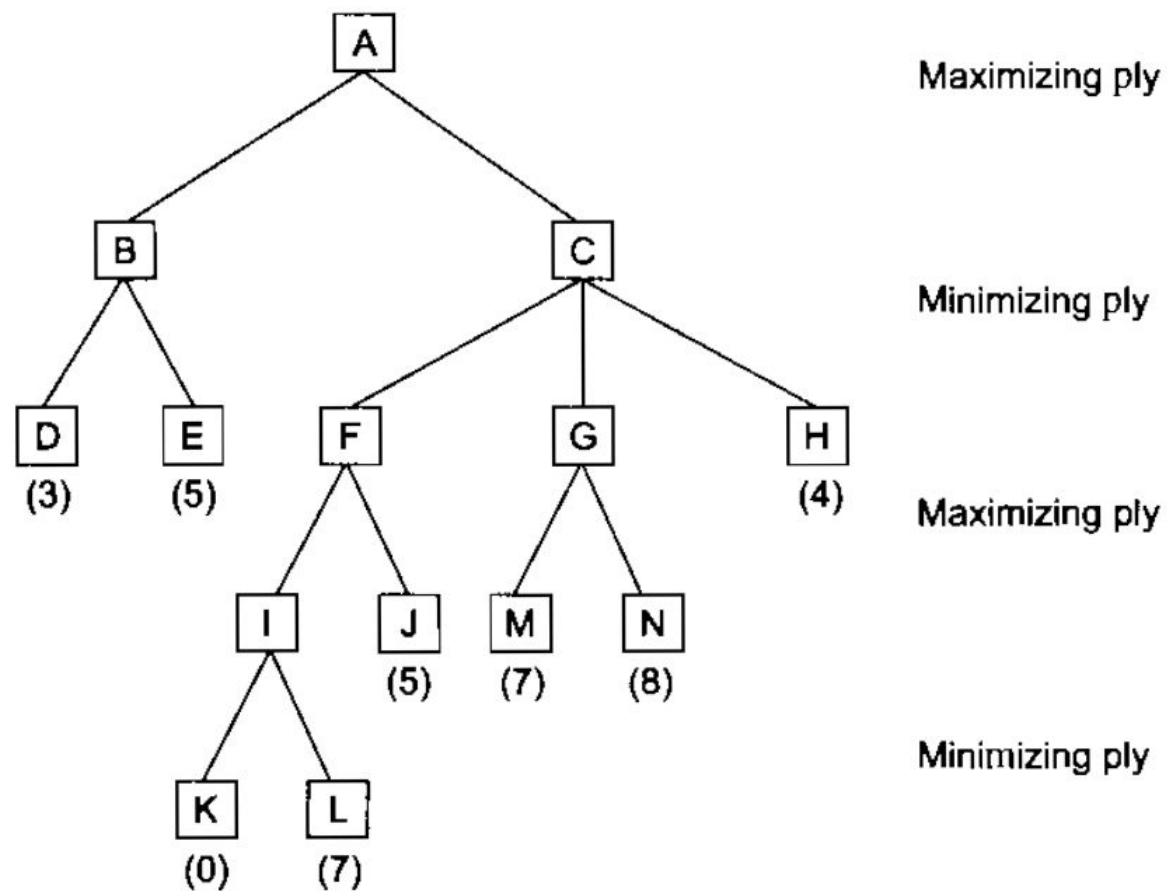
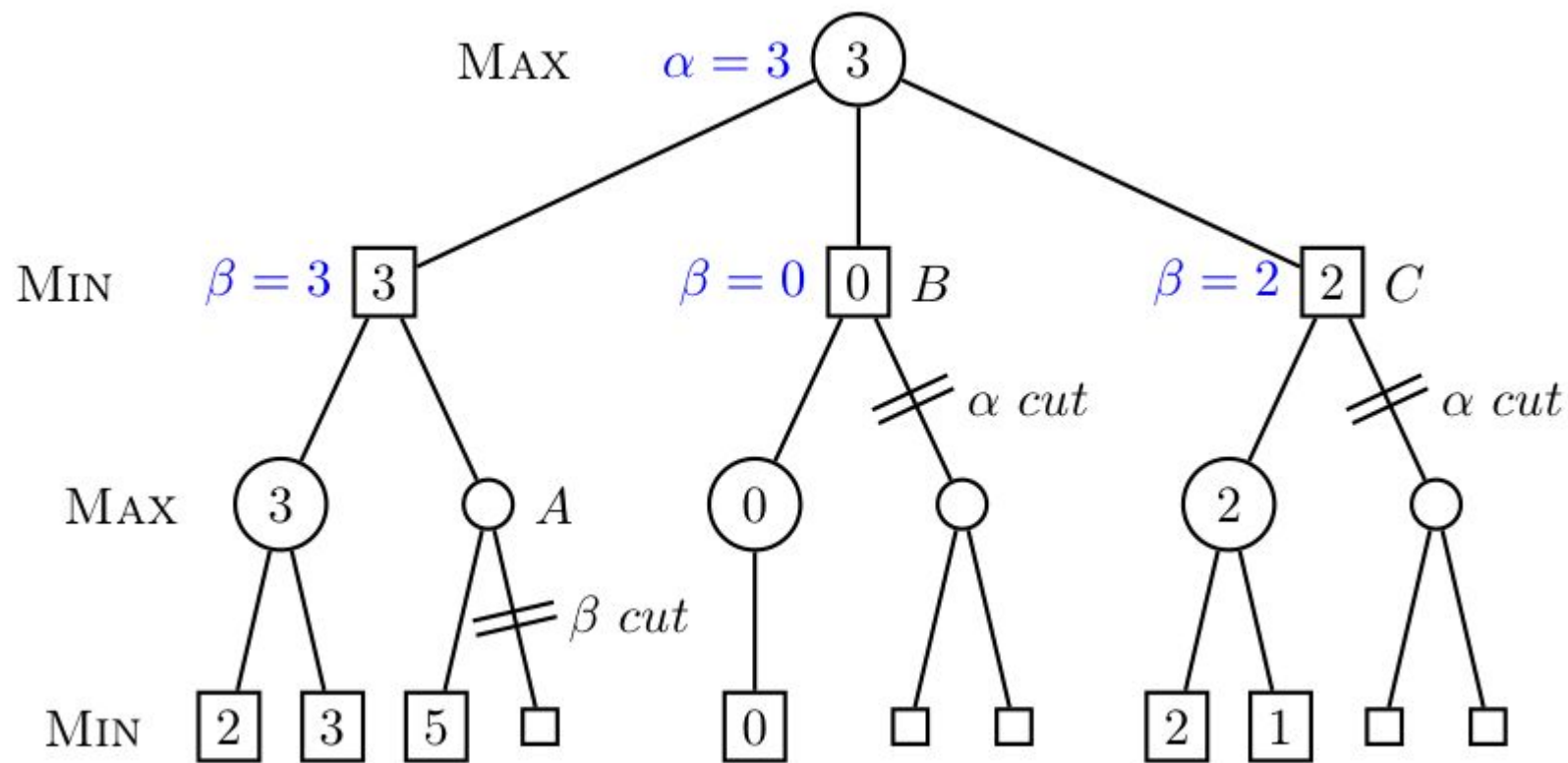


Fig. 12.5 *Alpha and Beta Cutoffs*



Static eval
function

Max = 10
Min = 0

Alpha-Beta Cutoffs

- Improvement on **Minimax algorithm**
- Takes advantage of the DFS nature of the minimax algorithm and uses a **branch-and-bound** method
- Paths are *cut off* if it is clear that its *values* will be **less** than a current threshold for **maximizing levels**
- Paths are *cut off* if it is clear that its *values* will be **greater** than a current threshold for **minimizing levels**

Cont ..

- Ruling out a move by a Maximizing player, means cutting off search at a minimizing level and vice-versa.
- Important:
 - At maximizing levels, only **beta** is used to determine cutoffs.
 - At minimizing levels, only **alpha** is used to determine cutoffs.
- **At Maximizing levels, alpha has to be known since the time a recursive call is made to MINIMAX-A-B, where a Minimizing level is created that requires that alpha for cutoff calculation.**
- Same for minimizing levels and beta.

Cont ..

- At maximizing levels, beta is used to calculate cutoffs but alpha is updated
- At minimizing levels, alpha is used to calculate cutoffs but beta is updated
- Use two thresholds,
 - `USE_THRESHOLD` : the value used for cutoff calculation, either alpha or beta, depending on the level.
 - `PASS_THRESHOLD` : The value that needs to be passed on. If on maximizing levels, this is the alpha value and on minimizing levels this is the beta value.
 - Note: the `PASS_THRESHOLD` is similar to `BEST_SCORE` from the original MINIMAX algorithm.
- `MINIMAX-A-B(Position, Current_Deoth, Player, Use_thresh, Pass_Thresh)`

MINIMAX-A-B(CURRENT,
0,
PLAYER-ONE,
maximum value STATIC can compute,
minimum value STATIC can compute)

These initial values for *Use-Thresh* and *Pass-Thresh* represent the worst values that each side could achieve.

Algorithm: MINIMAX-A-B(*Position, Depth, Player, Use-Thresh, Pass-Thresh*)

1. If DEEP-ENOUGH(*Position, Depth*), then return the structure
 VALUE = STATIC(*Position, Player*);
 PATH = nil
2. Otherwise, generate one more ply of the tree by calling the function MOVE- GEN(*Position, Player*) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, there are no moves to be made; return the same structure that would have been returned if DEEP-ENOUGH had returned TRUE.

4. If SUCCESSORS is not empty, then go through it, examining each element and keeping track of the best one. This is done as follows.

For each element SUCC of SUCCESSORS:

- (a) Set RESULT-SUCC to
MINIMAX-A-B(SUCC, $Depth + 1$, OPPOSITE(Player),
 $-Pass-Thresh$, $-Use-Thresh$).
- (b) Set NEW-VALUE to $-VALUE(RESULT-SUCC)$.
- (c) If $NEW-VALUE > Pass-Thresh$, then we have found a successor that is better than any that have been examined so far. Record this by doing the following.
 - (i) Set $Pass-Thresh$ to NEW-VALUE.
 - (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

- (d) If *Pass-Thresh* (reflecting the current best value) is not better than *Use-Thresh*, then we should stop examining this branch. But both thresholds and values have been inverted. So if *Pass-Thresh* \geq *Use-Thresh*, then return immediately with the value

VALUE = *Pass-Thresh*

PATH = BEST-PATH

5. Return the structure

VALUE = *Pass-Thresh*

PATH = BEST-PATH