**TECHNICAL UNIVERSITY OF KOŠICE**
**FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS**
**DEPARTMENT OF CYBERNETICS AND ARTIFICIAL INTELLIGENCE**

# BACKWARD CHAINING IN INFERENCE NETWORK

## Knowledge based systems

**Samuel Oklamčák**
**Andrii Pylypenko**
**Maksym Svitlovskyi**

**2020/2021**

# Contents

# Introduction

This document will teach you about knowledge based systems and one specific algoritm. After reading you will be able to understand basics of expert systems. You should be able to understand how **backward chaining** works.

# What Is a Knowledge Base?

Is an easily accessible data storage that contains informations about a certain product, service, topic, or concept. There are many different types of knowledge. The most fundamental distinction of knowledge types is between explicit and implicit knowledge.

Explicit knowledge are knowledge or skills that can be easily articulated and understood, and therefore easily transferred to others. Anything that you can write in an instructions, mathematical equations etc.

Implicit knowledge is knowledge that has the potential to be articulated or codified, but has not yet been transferred. This knowledge type is much more difficult to teach than explicit knowledge. Includes things like body language, aesthetic sense, or innovative thinking.

People create and store human knowledge, and transfer it from human to human. Note that this can be both explicit and implicit knowledge.

Implicit knowledge is more difficult to transfer.

# Expert System

Is a type of knowledge based systems. It stores knowledge, with help of AI it emulate human decision making. It uses explicit knowledge from an expert.

Expert system containts following basic components: knowledge base, inference mechanism, explanatory module, input/output interfaces.

Early expert systems did not support multiple users, and were meant to guide users toward a single, specific answer. However, as the volume of stored data increased, expert systems expanded to support more complex knowledge types, to perform more complex problem-solving, and to support multiple users.

We can describe expert systems on severa1 dimensions. One dimension has to do with methodo1ogy. Expert systems are members of the fami1y of AI programs. A second dimension is qua1ity: expert systems are expected to exhibit expert-1eve1 performance. A third dimension is the design dimension: Expert systems shou1d be easy to modify and their behavior shou1d be understandab1e.

# Inference mechanism

It is a brain of the system. It solves problems using logical conditions. Used algorithms are capable to solve problem based on given facts. It is based on inference rule, deduction (logical reasoning, in which conclusions must follow from assumptions), induction (from a specific case to a general one.), analogy (drawing a conclusion based on similarity to another situation).

# A little history

The scientists wanted programs which could think. They searched for general problem-solving techniques to solve problems. One of the first AI programming languages, called IPL, was developed and used in the late 1950's, and GPS (General Problem Solving) was developed at Carnegie-Mellon. In the early 1960s. McCarthy developed the Lisp programming language.

In the 1970s they came to a conclusion, that it is very hard to make program universal. So, before solving the problem they managed to spent more time to formulate problem. They also tried to make searching more effective.

In the 1980s the research was very fast. They evaluated that it is very important to have a lot of knowledge for the system. If system has more knowledge it can find solution much faster. So that means it will be more effective.

In 1981, the Defense Science Board recognized AI as one of the most promising investments for the military. AI soon became a buzzword throughout the military services, and it seemed like alI new proposal requests contained a requirement that any software must contain some artificial intelligence.

In the beginning there were developed more systems e.g. Dendral (1980, started in 1965), Macsyma (1975), Mycin (1976, diagnosis of blood infection),

After MYCIN they developed EMYCIN, the first tool for building rule-based expert systems.

Another major research area during this period of "Success" was the Speech Understanding Program, initiated by DARPA. Two systems developed at CMU -- Hearsay II and Harpy.

# Rules

Backward chaining system is rule based system. Knowledge are represented by rules.

Examples: *IF condition THEN conclusion*

*IF it is cloudy outside AND is very hot THEN it will rain*

The rules also can be represented as: A & B -> D, A | F -> E

where:

- **&( ∧)** mean AND (A and B must be TRUE)
- **|( ∨)** mean OR (one of them need to be TRUE)
- **->** IMPLY if the left side is TRUE then right side is also TRUE

Chaining the rules give us inference network. In the network you can not have chain which will cause loop. To derivate new knowledge we use 2 ways:

Modus ponens: if P is valid and

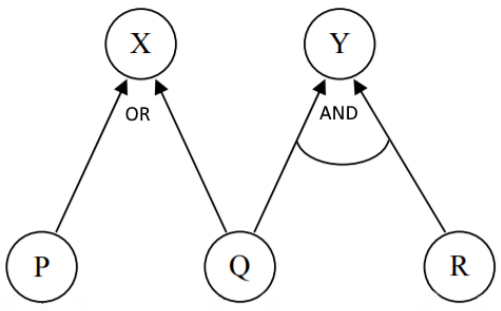
P-> Q is also valid then Q is valid

Modus tollens: if Q is not valid and implication P -> Q is valid then Q is not valid.

We can represent inference network as a graph.

EXAMPLE:

Rules: P -> X, Q -> X, Q ∧ R -> Y



# Backward chaining

In this chapter we will learn you how backward reasoning algorithm works. Very shortly the method is described as: working backward from the goal.

It starts with list of goals (hypothesis). From the consequent to the antecedent to find if any data which supports that consequent. Better said from known solutions we want to get unknown facts. Following the steps we will be able to understand the method.

In the first step, we will take the goal and from the goal, we will derive other facts that we will prove TRUE.

Then we will derive other facts from goal that satisfy the rules.

Later, we will extract further fact which infers from facts inferred in previous step.

Repeat the same until we get to a certain fact that satisfies the conditions.

Ok, we should train, so here is a very popular example:

**Our goal:** Criminal(Robert)

Read as: Robert is a criminal

**Facts:** American(Robert), Missile(T1), Owns(A, T1), Enemy(A, America)

Read as: Robert is American, T1 is missile, A owns T1, A is enemy to America.

**Rule_1:** American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p)

**Rule_2:** Enemy(p, America) →Hostile(p)

**Rule_3:** Missile(p) → Weapons (p)

**Rule_4:** Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A)

Read as: 1) Person p is criminal if is american AND has weapons AND p sells weapons q to r AND r is hostile. 2) p or better said A is hostile if is enemy of America. 3) We can say that missile is a weapon. And the last one 4) Enemy A owns missiles sold by Robert.

So, first step, our goal si to prove that Robert is a criminal. Now we want to infer other facts which satisfies the rules. Substitute Robert(P)

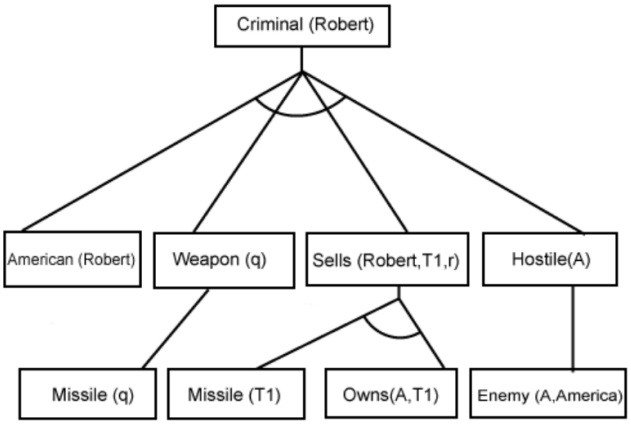
We know that American(P), so now we can say that Robert is American. Next we need to find if there are some weapons. Yes, there are missiles which could be sold. That mean that Robert could sold q to hostile r (Rule_1) If we look at another rules we know that first part of Rule_1 is a fact. So we proved small part of Rule_1

Now look at the second part of Rule_1. Can we extract some fact from it? Yes, we can say that missiles belongs to category of weapons. We have proved the Rule_3. Ok, first half done.

So p sells q to r. We can extract data from Rule_4. It says that missiles were sold to enemy A.

The last part we need to prove. Hostile(p). We can extract that p is enemy to America. So we proved all 4 parts of Rule_1. All of them are TRUE, which means that Criminal(Robert) is also TRUE.

Here is the graph:



We have found the main property of backward chaining:

- is known as top-down approach

Also we can say that it uses modus ponens.

You are able to understand how the algorithm works. The method of solving rules si recursive. If you will pass facts e.g.

B -> F and F -> B you will cause loop.

# Description of the solver

## Input

Contains facts and/or rules and query which you want to search. Used set of symbols:

| | |
|---|---|
| => | Inference |
| + | AND |
| \| | OR |
| ^ | XOR |
| **!** | NOT |
| **?** | Query to search |
| **=** | Fact |
| # | Comment |

Example:

```
A + B => C
C | E => D
B + ( C | E ) => G
=AB
?D
```

# Used modules

## Sys

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Example:

```
sys.setrecursionlimit(limit)
```

Set the maximum depth of the Python interpreter stack to limit. This limit prevents infinite recursion from causing an overflow.

If your knowledge base is very big and serching will be terminated by error:

**RecursionError exception is raised.** By changing the limit you will be able to solve your problem.

## Re

This module provides regular expression matching operations similar to those found in Perl. The module defines several functions, constants, and an exception.

Example:

Used in Parser module to determine lines where rules are.

```
pattern = re.compile("[A-Za-z!]")
pattern.match(object)
```

Use defined pattern of symbols and find match in object

# Parser

## About

The module is very important. It is used to read rules, facts and goals from input file. As first in the instantiation you pass a name of the file. Then you call **parse_file()** which will return lists of rules, facts and goals.

## Methods

parse_file() – return lists of rules, facts and goals.

- private methods:

__get_rules_lines() – used in parse_file() to find rules in file. Return a list of rules.

__get_goals_lines() – used to find goals in line with starting symbol **?**.

__get_facts_lines() – used to find lines starting with symbol =.

If parser cannot find rules, goals or facts it will inform user about the problem. It means that you passed incomplete file.

# Utils

## About

Used to break rules and facts into smaller parts known as **nodes**. Passing list of rules and facts to method initialize_atom_nodes_store_from_rule() you will get structure of atoms and nodes. Atoms are for e.g. A, B, C, Z.... Node is the main element stored in the tree. Each node is connected to each other in a parent/child relation.

## Methods

For simplicity we will skip private methods in the module.

**Node Store Utils**

initialize_atom_nodes_store_from_rule()

- input arrays of rules and facts

At first method finds all unique parts of rules. Initialize nodes and their childs.

- return stored nodes with their childs/ parents and states

Returned node store is complete structure for backward chaining method, which will use Solver module to solve rules.

**Rules Utils**

check_is_all_node_is_known() – check if all nodes are known.

- input child of node (child of specific node in backward chaining method ) and NodeStore
- use get_node() from NodeStore to determine state of node of child.
- return TRUE, and None
- return FALSE and all unknown nodes
- TRUE – we have all nodes, FALSE – there are some ndes

transform_child_to_logical_operation()

– to solve rule we need logical operation

- input child and NodeStore
- return transformed node

**Solver**

By breaking rule to smaller parts we can solve rule.

solve_rule() – it sill solve rule

- input transformed node from method (transfrom_child_to_logical_operation())
- this method will use described methods below.
- with help of method get_first_logical_operation() it will solve rule and return solved rule

get_first_logical_operation() – solve brackets and all NOT operations.

- input rule, return solved rule

solve_brackets() – solve part of rule which is in the brackets.

input rule to be solved

return solved rule

solve_all_not() – solving all parts of rule where symbol **!** occurs. Use helping method solve_not()

- input rule to be solved
- return solved part of rule with NOT operators

solve() – determine which operator is used. Then use proper method to solve that operator.

- input operator, left side, right side.
- return TRUE, FALSE or ERROR
- TRUE if returned value of helping method is true
- FALSE if returned value of helping methods is false
- ERROR if used operand is unknown

Helping methods in **solve** method

solve_and() – if left side and right side are TRUE then return TRUE otherwise FALSE

- input left side and right side
- return TRUE or FALSE

solve_or() – if left side or right side is TRUE then return TRUE otherwise FALSE

- input left side and right side
- return TRUE or FALSE

solve_not()

- input value
- return opposite of the input
- e.g. input: TRUE return: FALSE

solve_xor() – if both values are equal return FALSE otherwise TRUE

- input left side and right side
- return TRUE (left is not equal to right), FALSE (left is equal to right)

**Visualizer**

This module prints tree. It will help user to understand tree structure.

First create instance of the module., with dictionary data from backward chaining method.

print_tree() – create tree structure to screen. It is very important to provied a user friendly output.

# Models

## About

Definition of Nodes and Atoms. Helps to generate initialized NodeStore (knowledge base) for backward chaining solver.

### Nodes

Class Node - basic part of whole structure.

- data members: child array

Class AtomNode – inherit from Node class

- added members: name, state

### NodeStore

Save list of unique atom nodes. Methods describer below helps to manipulate with nodes. Used to change node state, set child, get index of node by value etc.

#### Getters

__get_node_index() – check list of nodes and find node by name to return index

- input name of which node we want index
- return index of node

get_node() – check list of nodes and find node by name. Then return it.

- input name of node
- return node

#### Setters

set_child() – set child of node

- input name of node, child which will be set

set_state() – set state of node

- input name of node, state to set

change_node_state() – will change state of specific node in node list. It will use index instead of name

- input index of node, state to set

# Backward_Chaining Module

## About

The brain of the system. Parser read file, save rules, facts and goals. Then initialize NodeStore. This part of solver will solve for list of goals.

Class Backward_Chaining

- data members: node_store, goals
- node_store – input knowledge base
- goals – list of goals

## Method

backward_chaining() – check if goal is between nodes, make sure that there are no loops e.g. Prerequisite is consequence at the same time. If yes, then throw exception. Check if we know all values of nodes. Call solver to transform rules to logical operations and solve.

If we have some unknown node state, then set it. Still we have unknown nodes, we will need to find subgoal. So, call backward_chaining() with unknown nodes.

- input goal to prove
- return solved rule, visualize dictionary.

# References

[1] Used images for example of graphs [Online] **https://www.javatpoint.com/forward-chaining-and-backward-chaining-in-ai** (2020/11)

[2] Jiří Dvořák, Expertni systémy [Online] Vysoké Učení Technické v Brně, 2004 **http://www.uai.fme.vutbr.cz/~jdvorak/Opory/ExpertniSystemy.pdf** (2020/11)

[3] Kristína Machová, Knowledge-based Systems in Questions and Answers, ELFA s.r.o., 2005, ISBN 80-8086-018-1, or [Online] **http://people.tuke.sk/kristina.machova/pdf/Znal%20Sys%20v%20otazkach.pdf** (2020/11)

[4] Robert S. Engelmore, Artificial Intelligence and Knowledge Based Systems: origins, methods and opportunities for nde, Springer, Boston, MA, Stanford University [Online] **https://core.ac.uk/download/pdf/38893222.pdf** (2020/12)