# SIN NOTES

## BLOCK 1: KNOWLEDGE REPRESENTATION AND SEARCH

# CHAPTER 1.-RULE-BASED SYSTEMS (RBS). REPRESENTATION IN RBS: FACTS AND RULES. PATTERN MATCHING

### 1.-KNOWLEDGE REPRESENTATION

A **knowledge representation** is the specific knowledge about the problem (domain-dependent knowledge) and the general knowledge about how to solve the problem.

There are two types of **knowledge**:

- **Declarative description**:
  - Set of clauses **describing properties**: **WHAT** something is.
  - Knowledge representation is independent of how it will be later used (**separation of knowledge and control**).
  - Types of declarative knowledge: relational, inheritable, deducible.
- **Procedural description**:
  - Set of procedures to **describe how to use knowledge**: **HOW** to do something.
  - Procedural knowledge includes information about how to use the knowledge (**knowledge and control are mixed**).

### 2.-RULE-BASED SYSTEMS (RBS)

**Rule-Based Systems** are composed of three elements:

- **Rule base** (includes rules) which represent how the world changes (problem actions).
- **Working memory** (includes facts) which represent the elements of the problem.
- **Inference engine**: control mechanism of an RBS.

It has a special type of if-then rule (p1^p2^ … ^pn => a1^a2^ … ^an):

- **Antecedent** (preconditions or premise): a conjunction of conditions.
- **Consequent** (effects, conclusion or action): a conjunction of actions. An action can be:
  - **Add** a fact to the working memory.
  - **Remove** a fact from the working memory. Negation by failure, that is that, what's not in the working memory, it is false.
  - **Query** the user
  - **External actions**.

The **working memory**, also called short-term memory, database or **Fact Base** (**declarative knowledge**), contains **facts** about the world which can be observed directly or inferred from a rule. Facts are temporary knowledge which may be modified by the rules.

The **rule base**, also called **Knowledge Base** (**procedural knowledge**), contains **rules**, each rule is a step in a problem solving process (inference step). A rule is a collection of conditions (preconditions) and the operations to be taken if the conditions are met (effects). Rules are persistent knowledge about the domain. Typically, only modified from the outside of the system. Rules are used to represent the **problem actions**.

The **inference engine** processes the information in the Working Memory and Rule Base. It is the domain-independent reasoning mechanism for RBS. It selects a rule from the Rule Base to apply. An **inference engine** is defined by:

- The **type of reasoning chaining**.
- The **process of pattern-matching**.
- The **control mechanism for selecting and executing rules**.

There are **two types of inference engine**:

- **Forward chaining** or **data-driven reasoning** (rules match the antecedent and infer the consequent).
- **Backward chaining** or **goal-driven reasoning** (rules match the consequent and prove the antecedent).

## 3.-CLIPS: A TOOL FOR BUILDING RBS

**CLIPS** (C Language Implementation Production System) is an expert system shell, a tool for building RBS. It provides:

- **Facts** for building the Working Memory (Fact Base).
- **Rules** for building the Rule Base.
- **Support for procedural knowledge and object-oriented representations**.
- A **forward-chaining inference** engine based on Rete match algorithm.
- Different strategies for solving conflicts (selecting a rule instance).

## 4.-WORKING MEMORY IN CLIPS

**Facts** are the elementary information items. The **Working Memory (WM)** is a list of facts. A fact is a list of atomic values that are referenced positionally (**ordered fact**).

An **ordered fact** is a symbol followed by a sequence of zero or more fields separated by spaces and delimited by an opening parenthesis on the left and a closing parenthesis on the right. The first field of an ordered fact specifies a “relation” that applies to the remaining fields in the ordered fact.

**<fact> ::= (<symbol> <constant>*)**

**<constant> ::= <symbol> | <string> | <integer> | <float> | <instance-name>**

Construct to define an initial group of facts:

**(deffacts <deffacts-name> [<comment>] <fact>*)**

## 5.-RULE BASE IN CLIPS

The **Rule Base** contains a list of rules. **Rules consist of two parts**:

- The **antecedent**, premise or condition (Left-Hand Side –LHS- of the rule): it represents the conditions that must hold in the current state for the rule to be applicable.
- The **consequent**, conclusion or action (Right-Hand Side –RHS- of the rule): it represents the actions to execute over the current state provided that the conditions of the LHS of the rule are satisfied (applicable rule), and the rule is selected for execution.

```
(defrule <rule name> ["comment"]
   <conditional-element>*          ; left-hand side (LHS)  of the rule
                                   ; conditions to be satisfied
=>
   <action>*)       ; right-hand side (RHS)  of the rule
                     ; actions to be performed when the rule fires (the rule is
                    ;  selected for execution)
)
```

Conditional elements in the LHS of a rule can be of two types:

- **Pattern**: restriction which is used to determine which facts satisfy the condition specified by the pattern. Patterns amounts to the use of **first-order predicate log**ic. Patterns are matched against facts.
- **Test**: a query, a boolean expression that evaluates to TRUE or FALSE.

All patterns and tests in the LHS **must fulfil** for the rule to be applicable:

1. A pattern fulfils if at least one fact in the WM matches the pattern
2. A test fulfils if it is evaluated to TRUE

The RHS of a rule represents **procedural knowledge** (executable commands):

- **Assert**: adds facts in the WM (**assert <fact>+**)
- **Retract**: deletes facts from the WM (**retract <fact-index>+**)

## 6.-PATTERN-MATCHING

**Pattern-matching** (syntactic unification). It is the process of matching patterns of the LHS of rules against facts in the Working Memory. CLIPS inference engine automatically matches patterns against the current state of the Working Memory and determines which rules are applicable (rules whose LHS is satisfied).

**Patterns** can contain:

- **Constants**.
- **Single-valued variables** (or single-field variables): variable name beginning with a question mark ‘?’. Single-valued variables are bound to a single value.

- **Multi-valued variables** (or multi-field variables): variable name beginning with the sign '$?x'. Multi-valued variables are bound to 0 or more values.
- **Wildcards**. Their name is simply the symbol '?' or '$?'. These are special variables which can be bound to any constant or set of constants but without holding their bindings (they avoid keeping the variable value if this is not necessary).

When two or more multi-valued variables appear in a pattern, the pattern and the fact can match more than once.

## 7.-RULE MATCHING

Choices of matching between a pattern and the facts of the Working Memory.

| | | | |
|---|---|---|---|
| 1 | pattern | fact | Only one match |
| 2 | pattern | fact 1<br>fact 2 | Same pattern matches once with different facts |
| 3 | pattern 1<br>pattern 2 | fact | Different patterns match the same fact |
| 4 | pattern | fact | Several matches pattern-fact due to the use of multi-valued variables |

**Rule-matching**: the LHS of a rule matches when there is a at least a fact in the Working Memory that matches each pattern in the LHS and the tests are evaluated to TRUE. When a rule matches, it is said the rule is applicable and that **an instance of a rule** has been found. There can be more than one match (more than one instance) for the same rule depending on the number of facts that match each pattern.

# CHAPTER 2.-INFERENCE AND CONTROL IN RBS. RETE.

## 1.-INFERENCE ENGINE

Unlike programming logic, which follows a Declarative semantics, RBS follow a Procedural semantics. This difference is due to the procedural nature of the RHS of the rules. **CLIPS uses a forward chaining inference engine** based on the Rete match algorithm (Rete-based inference engine). The control mechanism is referred to as the **recognize-act cycle** of a **Rete-based inference engine**:

1. Rule matching: generates a Conflict Set or Agenda with all the applicable rules or rule instances found.
2. If the Agenda is empty the process stops. Otherwise, a rule instance is selected from the Conflict Set according to a predetermined criterion (Conflict Resolution Strategy).
3. The RHS of the selected instance is executed, updating the Working Memory in accordance or executing the external actions.

## 2.-CONFLICT RESOLUTION STRATEGIES

The **Conflict Set** or **Agenda** is a collection of **rule instances** or **activations** which are those rules which match pattern entities. Zero o more rule instances can be on the Agenda. The top activation on the agenda is always selected for execution.

The conflict resolution strategy is the criterion to select the activation from the Agenda.

**Refraction**: a rule can only be used once with the same set of facts (same fact indexes) in the WM and same variable bindings. Whenever WM is modified, all rules can again be used provided that, at least, a new fact is generated that causes a new activation of the rule. This strategy prevents a single rule and list of facts from being used repeatedly, resulting in an infinite loop of reasoning.

CLIPS applies refraction and does not allow a rule to be fired twice on the same data (same fact indexes and same variable bindings) thus avoiding firing rules over and over again on exactly the same facts.

This behaviour is also supported by the default option of CLIPS of **not allowing fact duplication**. CLIPS does not accept a duplicate fact.

If the agenda contains more than one rule instance, the conflict resolution strategy decides which one to select. CLIPS always selects the top activation of the Agenda to be executed.

The conflict resolution strategy determines how rule instances are ordered in the Agenda.

**Explicit priority**: numeric value is the salience of rules.

In CLIPS, possible salience values range from -10,000 to 10,000. The higher the value, the higher the priority of the rule. If no salience is specified in the rules, all rules have salience 0.

**Depth strategy**: newly activated rules are managed through a priority queue. Ties are broken by using a LIFO strategy.

**Breadth strategy**: newly activated rules are managed through a priority queue. Ties are broken by using a FIFO strategy.

**Random**: each activation is assigned a random number which is used to determine its placement among activations of equal salience. CLIPS also allows to select Random as conflict resolution strategy.

**Specificity**: use the most specific rule, the one with most detail or constraints. The specificity of a rule is determined by the number of comparisons that must be performed on the LHS of the rule. Each comparison to a constant or previously bound variable adds one to the specificity. Each function call made on the LHS of a rule as part of a test conditional element adds one to the specificity.

CLIPS allows to select Simplicity and Complexity as conflict resolution strategies:

- **Simplicity**: ties are broken by giving more priority to rules with equal or lower specificity.
- **Complexity**: ties are broken by giving more priority to rules with equal or higher specificity

## 3.-RETE MATCH ALGORITHMS

**Rete** represents the rules internally as data in the so-called Rete network. The compiler creates the Rete network from the rules specified in the RBS. The Rete network consists of a pattern network and a join network. The Rete network amounts to a finite state machine that consumes modifications of facts.

Rete exploits two properties (common to all RBS):

- **Structural similarity**:
    - Rule conditions contain many similar patterns although not identical.
    - Rete takes advantage of structural similarity in rules (the fact that many rules often contain similar patterns or groups of patterns) by pooling common components so they need not be computed more than once.
    - A compiler groups common patterns before executing a RBS. This compiler generates the Rete network.
- **Temporal redundancy**:
    - Facts in a RBS change slowly over time. Few changes occur in the WM at each cycle even though the WM contains many facts.
    - Rete takes advantage of temporal redundancy by remembering what has already been matched from cycle to cycle and computing only the changes necessary for the newly added or newly removed facts (rules remain static and facts change).
    - It saves the state of the matching process during one cycle and re-computes the changes in this state only for the changes that occur in the WM.
    - This way, the computational cost of the RBS depends on the frequency of changes in the WM, which is usually low, and not on the size of its contents.

# CHAPTER 3.-STATE-BASED REPRESENTATION THROUGH RBS

## 1.-STATE-BASED REPRESENTATION

Problem-solving in AI is usually modelled as a **search** process in a **state space**. We consider a single problem-solving agent. Problems are solved by finding a sequence of states that leads to a solution.

**Problem definition**:

1. Represent the set of possible states in the problem.
2. Specify the initial state.
3. Specify the goal state or some goal test.
4. Transition rules or move operators to move from one problem state to another.

An **action** is a transition that when applied in a state **s** returns a new state as a result of doing the action in **s**.

A **state space** is a set of all states reachable from the initial state by any sequence of actions.

The **goal** is to find the sequence of operators or actions (solution) which being applied to the initial state leads to the goal state. This is done by searching in the state space.

An **environment** is an observable (the agent knows the current state), discrete (finite number of actions), known (the agent knows which states are reached by each action), deterministic (each action has exactly one outcome).

**Problem formulation**:

- **Initial state**: initial state of the problem (s0).
- **Actions**: actions available to the agent, actions that can be executed in the problem. For example: {a1,a2,a3,a4}. Actions(s) returns the set of actions applicable in s, the actions that can be executed in s. Example: Actions(s0)={a1,a2,a3}, Actions(s1)={a1,a4}.
- **Transition model**: Result(s,a) returns the state that results from doing action a in state s. Example: Result(s0,a1) =s1, Result(s1,a1)=s2.

The initial state, actions and transition model implicitly define the **state space**. The state space forms a directed network or **graph**. Nodes represent the problem **states**, edges represent the problem **actions**. A **path** in the state space is a sequence of states connected by a sequence of actions.

The **goal test** determines whether a given state is a goal state or not:

- Define a particular state as the goal state or.
- Define an explicit set of goal states or.
- Define a test that checks whether a given state is a goal state.

**Path cost** function that assigns a numeric cost to each path:

- The step cost of taking action a in state s to reach s' is denoted by c(s,a,s'); usually, the cost of an action is the same independently of the state in which the action is applied.
- The cost of a path is described as the sum of the costs of the individual actions along the path.

## 2.-DESIGN OF STATE-BASED PROBLEMS THROUGH RBS

The **objective** is to generate the search tree of the problem until finding a node that contains the final situation, so rules **MUST NOT** delete facts that prevent the search process from generating the tree, thus, **DO NOT USE** retract. The nodes of the search tree (problem states) must be consistently represented in the WM.

If we **eliminate the retract command** of the rules, the WM stores all of the facts asserted by the rules but we are not able to identify the facts which represent one state or another, so the **solution** is to represent a problem state with a single fact.

## 3.-PROBLEM-SOLVING BY USING RBS

**State representation (patterns and facts)**:

- Use a knowledge representation language that allows to represent patterns, facts and rules.
- Make an RBS design capable to represent the problem states.
- Represent only the information relevant to attain the goal.
- A good design should allow for an easy and clear identification of the key components of the problem states as well as facilitating the design of the LHS and RHS of the rules.

There are **two types of state representation**:

1. **Dynamic information**:
   - Data that may change as a result of the actions execution.
   - Maintain dynamic information in a single fact such that the fact represents a particular problem situation (problem state).
2. **Static information**:
   - Data that do not change along the problem-solving evolution.
   - Static information can be represented with as many facts as desired.

# CHAPTER 4.-SOLVING PROBLEMS BY SEARCHING. UNIFORMED SEARCH

## 1.-SEARCHING FOR SOLUTIONS

A **solution** is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the **branches are actions** and the **nodes correspond to states** in the state space of the problem.

**General Search Process**:

1. Current node <- initial state
2. **Check** whether current node is a **goal state**; if so, finish.
3. **Apply** each legal action to the current state, thereby **generating** a new set of states; we add the branches from the parent node to the new child node.
4. **Choose next node** from the frontier (nodes which are not yet expanded).
5. Start all over again from step 2.

The **frontier** is also called **leaf nodes**, **OPEN LIST** or **fringe**.

A **search tree** is the set of nodes generated during search. **Search algorithms** all share the basic structure seen before; they vary primarily according to how they choose which state to expand next the so-called **search strategy**.

**Tree-search algorithm**:

```
function TREE-SEARCH (problem) return a solution or failure
    Initialize the OPEN LIST using the initial state of the problem
    do                                                                  search-strategy
        if the OPEN LIST is empty then return failure
        expand a node: choose a leaf node for expansion and remove it from the OPEN LIST
        if node contains goal state then return the corresponding solution
        generate children and add the resulting nodes to the OPEN LIST
    enddo
```

The **search space** is not equal to the **state space**, due to repeated states. **Repeated states** are due to:

- **Reversible actions**: loopy paths.
- **Redundant paths**: more than one way to get from one state to another.

Since **we are concerned about reaching the goal**, there is no reason to keep more than one path to any given state because any goal that is reachable by extending one path is also reachable by extending the other.

How to **avoid repeated states**:

- Remember where one has been.
- Augment the TREE SEARCH algorithm with a data structure called **explored set (CLOSED LIST)** which remembers every expanded node.

- Newly generated nodes that match a node in the CLOSED LIST (already expanded) or in the OPEN LIST (generated but not yet expanded) can be discarded instead of being added to the OPEN LIST.
- New algorithm: GRAPH SEARCH algorithm, which separates the state space graph into the explored/expanded region and the unexplored/unexpanded region.

In general, avoiding repeated states and redundant paths is a matter of efficiency.

The OPEN list is implemented as a **priority queue**:

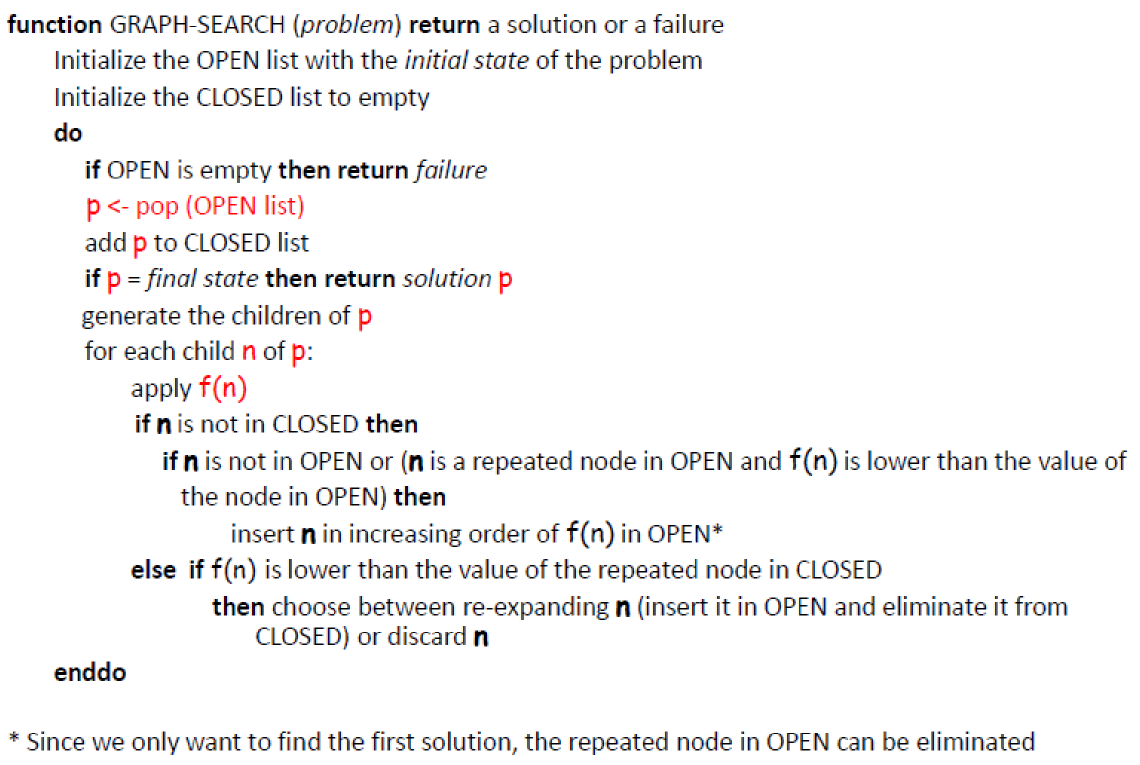
Elements are always ordered in the queue according to some ordering function (in increasing order, function minimization).

Nodes with lower values go to the front of the queue; nodes with higher values go to the back of the queue.

We always pop the element of the queue with the highest priority (lowest value of the ordering function).

For each search strategy, we define an **evaluation function ( f(n) )** that returns a numerical value for each node **n** such that nodes are inserted in the queue in the same order as they would be expanded by the search strategy (increasing order of f(n)).

**Graph-search algorithm**:



```
function GRAPH-SEARCH (problem) return a solution or a failure
    Initialize the OPEN list with the initial state of the problem
    Initialize the CLOSED list to empty
    do
        if OPEN is empty then return failure
        p <- pop (OPEN list)
        add p to CLOSED list
        if p = final state then return solution p
        generate the children of p
        for each child n of p:
            apply f(n)
            if n is not in CLOSED then
                if n is not in OPEN or (n is a repeated node in OPEN and f(n) is lower than the value of
                  the node in OPEN) then
                        insert n in increasing order of f(n) in OPEN*
            else  if f(n) is lower than the value of the repeated node in CLOSED
                    then choose between re-expanding n (insert it in OPEN and eliminate it from
                        CLOSED) or discard n
    enddo
```

* Since we only want to find the first solution, the repeated node in OPEN can be eliminated

**Search in a tree**:

- Maintains the OPEN LIST but not the CLOSED LIST (less storage requirements).
- Can only avoid repeated states in the OPEN LIST (if we are only concerned about reaching the goal).

- Re-expands nodes already explored (waste of time and memory repeating the same search)

**Search in a graph**:

- Maintains the OPEN LIST and CLOSED LIST (bigger memory requirements).
- Control of all repeated states avoids redundant paths (exponential reduction in search).

We will evaluate search strategies performance in four ways:

- **Completeness**: is the strategy guaranteed to find a solution when there is one?
- **Time complexity**: how long does it take to find a solution?
- **Space complexity**: how much memory does it need to perform the search?
- **Optimality**: does the strategy find the highest quality solution when there are several solutions? Solution with the lowest path cost?

We might be interested in finding trade off solutions between:

- **Cost of the solution path**: sum of the costs of the individual actions along the path **g(goal_state))**.
- **Search cost**: cost to find the solution path.

**TOTAL COST** is a combination of the **search cost** and the **path cost of the solution found**.

**Types of search strategies**:

- **Uninformed** or blind search (expansion order of nodes).
- **Informed** or heuristic search (“intelligent” expansion of nodes).

## 2.-BREADTH-FIRST SEARCH

**Breadth first**: expand the shallowest unexpanded node.

**Priority queue (OPEN list)**: which ordering function can we use to insert the shallowest nodes at the front of the queue? Which evaluation function can we use to make shallower nodes take on lower values?

**f(n)=depth(n)=level(n)**

**Characteristics of breadth-first search**:

- **Complete**.
- **Optimal**:
  - Breadth first always returns the shortest solution path (shallowest goal node).
  - Shortest solution is optimal if all actions have the same cost and the path cost is a non-decreasing function of the depth of the node (non-negative action costs).
- **Time complexity (execution time)**: for a branching factor b and depth d:
  - Expanded nodes: $O(b^d)$.
  - Generated nodes: $O(b^{d+1})$.
- **Space complexity (memory requirements)**: for a branching factor b and depth d:

    - For breadth first GRAPH SEARCH in particular, every node generated remains in memory.
    - There will be O($b^d$) in the CLOSED LIST and O($b^{d+1}$) in the OPEN LIST (dominated by the size of the OPEN LIST).
- **Take home lessons**:
    - The memory requirements are a bigger problem for breadth first than is the execution time.
    - Time requirements are still a major factor.

## 3.-UNIFORM-COST SEARCH

**Uniform cost** expands the unexpanded node with the **lowest cost** (lowest value of g(n)). Evaluation function (priority queue, OPEN list): **f(n)=g(n)**.

**Characteristics of uniform-cost search**:

- **Complete**: it is complete if action costs ≥ ε (small positive constant, non-zero costs).
- **Optimal**: if action costs are non-negative g(successor(n)) > g(n). Uniform cost search expands nodes in order of their optimal path cost.
- **Time and space complexity (execution time and memory requirements)**:
    - Let C* be the cost of the optimal solution
    - Assume that every action costs at least ε
    - The algorithm's worst case time and space complexity is O($b^{C^*/\varepsilon}$).

## 4.-DEPTH-FIRST SEARCH

Depth first: expand the **deepest** unexpanded node. Evaluation function (priority queue, OPEN list): **f(n)= level(n)**.

**Backtracking**:

1. Dead end node , non goal node with no applicable actions.
2. Set a depth limit.
3. Repeated state (optional, if control of repeated states).

A list called **PATH** is used to apply Backtracking: it only stores the expanded nodes of the current path and the nodes are released when the function bactracking is invoked.

**BACKTRACKING(n)**:

1. Remove n from PATH.
2. If parent(n ) has more children in OPEN => select the next node in the OPEN list.
3. If parent(n) has no more children in OPEN => BACKTRACKING (parent(n)).

**Characteristics of depth-first tree search**:

- **Time complexity (execution time)**:
    - For a bounded tree with maximum depth level **m**, **O($b^m$)**.
    - If **m=d** then depth first explores as many nodes as breadth first. But **m** itself can be much larger than **d**.

- **Space complexity (memory requirements)**:
    - The TREE SEARCH version has very modest memory requirements. It only stores a single path from the root node to a leaf node (PATH list), along with the remaining unexpanded sibling nodes for each node on the path (OPEN list).
    - For a branching factor **b** and maximum depth level **m**, **O( b * m ))**. **Linear space**.
    - Very efficient handling of the OPEN and PATH lists (very few elements). But there is hardly control of repeated states because lists contains very few elements. In practice, this means that depth first generates the same node multiple times. Even though, the TREE SEARCH version of depth first may actually be faster than breadth first
    - For a GRAPH SEARCH version, there is no advantage of depth first over breadth first.
- **Completeness**:
    - If the tree is unbounded (no maximum depth level) and no control of repeated states, it can get stuck in a loop forever (**no complete**).
    - If the tree is unbounded and control of repeated states, it avoids infinite loops but does not avoid redundant paths. It will eventually find a solution (**complete**).
    - If the tree is bounded (maximum depth level) it might lose the solution if the solution is not within the search space defined by limit m (**no complete**).
- **No optimal**.

## 5.-SEARCH IN CLIPS

**Summary**:

- No control of repeated states in CLIPS (facts representing the same state are different due to the fields like level ). Avoiding fact duplication does not avoid repeated.
- Applicable rules = Generated nodes = OPEN LIST.
- Executed rules = Expanded nodes = CLOSED LIST.
- Breadth first: set the CLIPS Agenda to Breadth.
- Depth first: set the CLIPS Agenda to Depth.
- CLIPS does not implement de TREE SEARCH versions but the GRAPH SEARCH versions of Breadth first and Depth first.

## 6.-ITERATIVE DEEPING DEPTH-FIRST SEARCH

It **iteratively** performs a **depth limited search** from depth limit =0 to depth limit ∞:

- Solves the problem of picking a good depth limit in depth first search.
- Combines the benefits of depth first and breadth first search.
- Complete and optimal (under same conditions of breadth first search).
- Time complexity **O($b^d$)**; space complexity **O(b * d)**.
- Preferred search method when there is a large search space and the depth of the solution is not known.

In breadth first when a solution node is selected for expansion, the tree already contains nodes generated at the next level in depth (d+1). However, this does not occur in ID.

**ID** may seem wasteful because states are generated multiple times. It turns out this is not so costly. The reason is that in a search tree with a similar branching factor at each level, most of the nodes are in the bottom level so it does not matter much that the upper levels are generated multiple times.

**ID is actually faster than breadth first** despite the repeated generation of states due to the high memory requirements of breadth first and the difficulty to access the stores nodes.

**ID** is the **preferred** uninformed search method when **there is a large search space and the depth of the solution is not known**.

## 7.-SUMMARY

| **Criterion** | **Breadth** | **Uniform cost** | **Depth** | **Iterative Deepening** |
|---|---|---|---|---|
| Temporal | $O(b^{d+1})$ | $O(b^{C^*/\varepsilon})$ | $O(b^m)$ | $O(b^d)$ |
| Espacial | $O(b^{d+1})$ | $O(b^{C^*/\varepsilon})$ | $O(b.m)$ | $O(b.d)$ |
| Optima? | Yes * | Yes | No | Yes * |
| Completa? | Yes | Yes ** | No | Yes |

* Optimal if step costs are all identical

** Complete if step costs $\geq \varepsilon$ for positive $\varepsilon$

# CHAPTER 5.-HEURISTIC SEARCH

## 1.-HEURISTIC SEARCH

**Informed (heuristic) search** is the one that uses problem specific knowledge to guide search.

Heuristic search can find solutions more efficiently than an uninformed search. Heuristic search is recommended for complex problems of combinatorial explosion.

It is used in order **to efficiently solve a problem we must sometimes give up using a systematic search strategy** which guarantees optimality and use instead a search strategy that returns a good solution though not the optimal one.

Heuristic search performs an intelligent search guidance and allows for pruning large parts of the search tree.

Heuristic is **appropriate** since:

1. Usually, we don't need to get the optimal solution in complex problem, but a good solution is just enough.
2. Heuristics may not return a good solution for the worst case problem but this case seldom happens.
3. Understanding how heuristics work (or don't work) helps people understand and go deeply into the problem.

We will consider a general approach called **best-first search**:

- An instance of the TREE SEARCH or GRAPH SEARCH algorithm in which a node is selected for expansion based on an **evaluation function f(n)**.
- **f(n) is a cost estimate** so the node with the lowest cost is expanded first from the priority queue (in uninformed search, f(n) is a different function for each strategy so that nodes are inserted in the OPEN LIST according to the expansion order of the strategy).
- In fact, all search strategies can be implemented by using f(n); the choice of **f** determines the search strategy.
- **Two special cases of best first search**: **greedy best first search** and **A***.

## 2.-GREEDY SEARCH

Most best-first algorithms include as a component of **f** a heuristic function h(n).

A **heuristic function** is a rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood.

**h(n)** = **estimated cost** of the cheapest path from the state at node n to the goal state. If n is the goal state then h(n)=0.

**Greedy best first search** expands the node that appears to be closest to goal; it expands the closest node to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: **f(n)=h(n)**.

Characteristics of **Greedy search**:

- **Complete**:
  - NO, it can get stuck in loops.
  - Resembles depth first search (it prefers to follow a single path to the goal).
  - The GRAPH SEARCH version is complete (check on repeated states in the CLOSED LIST).
- **Optimal**: no, at each step it tries to get as close to the goal state as it can; this is why it is called greedy.
- Ti**me complexity (execution time)**:
  - The worst case time complexity is **$O(b^m)$** where **m** is the maximum depth of the search space.
  - Like worst case in depth first search.
  - Good heuristic can give dramatic improvement.
  - The amount of the reduction depends on the particular problem and on the quality of the heuristic.
- **Space complexity (memory requirements)**: **$O(b^m)$** where **m** is the maximum depth of the search space.

## 3.-A$^*$ SEARCH

**A* search** is the most widely known form of best first search. It evaluates nodes by combining g(n), the cost to reach the node n , and h(n), the cost to get from the node to the goal state. Thus, **f(n)=g(n)+h(n)**.

The idea is to avoid paths that are already expensive. **f(n)** is the **estimated total cost** of the cheapest solution through **n**.

Algorithms that use an evaluation function of the form **f(n)=g(n)+h(n)** are called **A algorithms**.

A* search uses an **admissible heuristic function**. A heuristic is admissible if it **never overestimates** the cost to reach the goal, it is like being optimistic.

Formally:

- A heuristic h(n) is **admissible** if for every node n , **h(n) <= h*(n)**, where h*(n) is the **true cost** to reach the goal state from n..
- Because A* search uses an admissible heuristic, it returns the optimal solution.
- h(n) >= 0 so h(G)=0 for any goal G

**Comparison** to other search strategies:

- Breadth first search (optimal if all operators have the same cost). Equivalent to f(n)=level(n)+0, where h(n)=0 < h*(n).
- Uniform cost (optimal). Equivalent to f(n)=g(n)+0 where h(n)=0 < h*(n).
- Depth first (non-optimal). Not comparable to A*.

**Heuristic knowledge**:

- h(n)=0, lack of knowledge
- h(n)=h*(n), maximal knowledge

- If h2(n) >= h1(n) for all n (both admissible) then h2 **dominates** h1 (h2 **is more informed** than
- h1); h2 will never expand more nodes than h1.

h(n) = 0: no computational cost of h(n). (Slow search. Admissible).

h(n)=h*(n): large computational cost of h(n). (Fast search. Admissible).

h(n)> h*(n): very high computational cost of h(n). (Very fast search. No admissible)

In general, **h*** is not known but it is possible to state whether **h** is a lower bound of **h*** or not.

For difficult problems, it is recommended to highly reduce the search space. In these cases, it is worth using h(n) > h*(n) to eventually find a solution at a reasonable cost even though this is not the optimal solution.

For each problem, we must find a balance between the **search cost** and the **cost of the solution path found**.

The **optimality** of a **TREE SEARCH A*** algorithm is guaranteed if h (n) is **an admissible heuristic**.

**Admissibility**:

- h(n) is admissible if it never overestimates the cost to reach the goal (For all n, h(n) <= h*(n)).
- Let **G** be a goal/objective node, and **n** a node on an optimal path to **G**. **f(n)** never overestimates the true cost of a solution through **n**.

$$\mathbf{f(n) = g(n)+h(n) \leq g(n)+h^*(n) = g(G) = f(G) \Rightarrow f(n) \leq g(G)}$$

The optimality of a GRAPH SEARCH A* algorithm where repeated states in CLOSED are re-expanded is guaranteed if h(n) is an **admissible heuristic**.

In a GRAPH SEARCH A* algorithm with no re-expansion of repeated states in CLOSED, the optimal solution is guaranteed if we can ensure the first found path to any node is the best path to the node. This occurs if h(n) is a **consistent heuristic**.

**Consistency**, also called monotonicity is a slightly stronger condition than admissibility: h(n) is consistent if, for every node n and every successor n' of n generated by any action a:

$$\mathbf{h(n) \leq h(n')+c(n,a,n')}$$

Therefore, a GRAPH SEARCH A* algorithm with no re expansion of repeated states that uses a consistent heuristic h(n) also returns the **optimal solution**:

- Because it is guaranteed that when a node **n** is expanded, we have found the optimal solution from the initial state to **n** so **it is not necessary to re expand n in case of a repeated instance**, the cost of the first instance of **n** will never be improved by any repeated node (they will be at least as expensive).
- A derivation of the consistency condition is that the sequence of expanded nodes by the GRAPH-SEARCH version of A* is in non-decreasing order of f(n).

Characteristics of **A* search**:

- **Complete**: YES, if there exists at least one solution, A* finishes (unless there are infinitely many nodes with $f < f(G)$).
- **Optimal**:
    - YES, under consistency condition (for GRAPH SEARCH version).
    - There always exists at least a node n on the OPEN LIST that belongs to the optimal solution path from the start node to a goal node.
    - If C* is the cost of the optimal solution:
        - A* expands all nodes with $f(n) < C^*$
        - A* might expand some nodes with $f(n)=C^*$
        - A* expands no nodes with $f(n) > C^*$
- **Time complexity (execution time)**:
    - $O(b^{C^*/min\_action\_cost})$ exponential with path length.
    - Number of nodes expanded is still exponential in the length of the solution.
- **Space complexity (memory requirements)**:
    - Exponential: it keeps all generated nodes in memory (as do all GRAPH SEARCH algorithms).
    - A* usually runs out of space long before it runs out of time. Hence, space is the major problem, not time.

## 4.-DESIGN OF HEURISTIC FUNCTIONS

A problem with less restrictions on the operators is called a **relaxed problem**. Admissible heuristics can be derived from the cost of an **exact solution to a relaxed version of the problem**. This is usually a good heuristic for the original problem.

**Manhattan distance** is the sum of the distances of the tiles from their goal positions.

## 5.-EVALUATION OF HEURISTIC FUNCTIONS: COMPUTATIONAL COST

**Computational cost (temporal cost)**:

- **Search cost**: number of nodes generated or applied operators, and.
- **Cost of calculating h(n)**: cost for selecting the applicable operator.

Effective branching factor (b*): if the total number of nodes generated by A* for a particular problem is **N**, and the solution depth is **d**, then **b*** is the branching factor that a uniform tree of depth **d** would have to have in order to contain N+1 nodes. Thus,

$$N+1 = 1+b^*+(b^*)^2 + \dots + (b^*)^d$$

b* defines how sharply a search process is focussed toward the goal. It is reasonably independent of path length (d). It can vary across problem instances, but usually is fairly constant for sufficiently hard problems.

A well-designed heuristic would have a value of b* close to 1, allowing fairly large problems to be solved. A value of b* near unity corresponds to a search that is highly focussed toward the goal, with very little branching in other directions. Experimental measurements of b* on a small set of problems can provide a good guidance to the heuristic's overall usefulness.

# CHAPTER 6.-ADVERSARIAL SEARCH

## 1.-GAMES: PROBLEM DEFINITION

Most studied games in Artificial Intelligence are:

- **Deterministic** (but strategic): luck does not play a role.
- **Turn-taking**: play alternatively.
- **Two-player**: person-person, person-computer.
- **Zero sum**: the gain (or loss) of a participant is exactly balanced by the loss (or gains) of the opponent. This opposition makes the situation adversarial.
- **Perfect information**: games are known and limited; e ach player knows perfectly the game evolution and what moves he and his opponent can make (observable).

Players are usually restricted to a small number of actions. Actions outcomes are defined by precise rules. Games require the ability to make some decision even when calculating the optimal decision is infeasible.

We will consider games with two players: **MAX** and **MIN**. MAX moves first and they take turns moving until the game is over.

The **initial state** includes the board position and identifies the player to move.

**Actions (successor function)** are the legal moves that the player can take; each action gives rise to a resulting state (new board position). Successor states represent the states that a player can reach in one move (one turn).

The **terminal state** (leaf nodes, goal state) determines when the game is over.

The **utility function**, also called objective function or payoff function, gives a numerical value for the terminal states. The outcome is a win (+∞), a loss (-∞) or a draw (0).

The initial state and the legal moves for each player define the game tree for the game. MAX is the initial player. Players move alternatively. The numbers on each leaf node indicates the **utility value of the terminal state from the point of view of MAX**. High values are good for MAX and bad for MIN. The **objective** is to use the search tree (particularly the utility of terminal nodes) to **determine the best move for MAX**.

**Problems**:

- In general, game playing can be solved by applying search techniques on AND/OR graphs.
- However, even for simple games, a complete search turns out to be intractable.
- Even though different techniques can be applied in order to reduce search, these are proved not to be enough.

**Solutions**:

- Increase the heuristic information of the method at the cost of perhaps losing admissibility. The goal is to approach $b^* \approx 1$.
- Prune the search tree at a certain point and use methods to select the best initial move.

## 2.-MINIMAX ALGORITHM

When searching for the optimal solution for MAX (best initial move for MAX), MIN has something to say about it. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an **infallible opponent**. It is infeasible, even for simple games, to draw the entire game tree.

**Minimax procedure**:

1. Generate the game tree **up to a certain depth level in breadth-first fashion**, all the way down to the terminal states of the maximum depth level.
2. Apply the utility function to each terminal state to get its value.
3. Use the utility values of the terminal states to determine the utility of the nodes one level higher up in the search tree (**depth-first**). Continue backing up the utility values from the leaf nodes toward the root, one layer at a time (**depth-first**). MIN nodes take the minimum utility value of its successors; MAX nodes take the maximum utility value of its successors.
4. Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the state with the highest utility value. That is, it chooses the action corresponding to the best possible move, the move that leads to the outcome with the best utility (maximum utility) assuming that the opponent plays to minimize utility.

**Minimax-value(n)**:

- Utility(n), if n is a **terminal** node.
- The maximum of the successors, if n is a **MAX** node.
- The minimum of the successors, if n is a **MIN** node.

## 3.-ALPHA-BETA PRUNING

The problem with Minimax search is that the number of game states it has to examine is exponential in the number of moves. It is possible to compute the correct Minimax decision without looking at every node in the search tree. That is, we can eliminate large parts of the tree from consideration.

**Alpha-beta** pruning returns the same move as Minimax would, but prunes away branches that cannot possibly influence the final decision. The MINIMAX procedure separates completely the processes of search-tree generation and position evaluation, so it has a  inefficient strategy. The **α-β pruning** is based on a procedure that generates a terminal node and, simultaneously, evaluates it and backs up its value (**depth first generation and evaluation/exploration**).

**α-β pruning** algorithm:

1. **Initially**:
   a. MAX nodes: **α** values (initially **α** = -∞).
   b. MIN nodes: **β** values (initially **β** = +∞).
2. **Generate a terminal node** in depth first manner. Compute its utility value.
3. **Back up** the value to the parent node and keep the best value for the parent.

4. **Cut-off question**. Whenever a node (MAX or MIN) is given a backed-up value, this value is compared with all its contrary predecessor values in order to apply an **α** cut off or **β** cut-off. If a cut off is produced, go to 3 step.
5. **Provisional/definite value**:
    a. If it is a provisional value, go to 2 step.
    b. If it is definite value, go to 3 step.

**Cut-off**:

- **α**: $\alpha >= \beta$.
- **β**: $\beta <= \alpha$.

Characteristics of **α-β pruning**:

- **MAX nodes are assigned α values**:
    - The α values of MAX nodes are lower bounds and they can never decrease so they can only be assigned values >= α.
    - The α value of a MAX node is set equal to the current largest final backed up value of its successors.
- **MIN nodes are assigned β values**:
    - The β values of MIN nodes are upper bounds and they can never increase so they can only be assigned values <= β.
    - The β value of a MIN node is set equal to the current smallest final backed up value of its successors.

The final values of nodes in an **α-β** procedure are the same as in MINIMAX, except for those nodes whose values come from an α cut off (values could be lower) or β cut off (values could be greater). If a cut-off is produced in a node selectable by the initial MAX node, the node will not be chosen in the game as its value is improved by another selectable node which has not been cut off.

The search efficiency of the **α-β** procedure depends on the number of cut offs that can be made during a search.

The number of cut-offs depends on the degree to which the early α and β values approximate the final backed up values, that is, the moment to which the α and β values allow to make a cut off.

The final value of the start node is set equal to the static value of a terminal node (best node at the expansion depth level): if this node appears soon, the number of cut offs in the depth first search will be optimal thus generating the minimal search-tree.

The number of terminal nodes of depth **d** that would be generated by optimal **α-β search** is about the same as the number of terminal nodes that would have been generated by MINIMAX at depth **d/2**. Therefore, in the same time α-β allows to explore twice as much as the MINIMAX procedure.

## 4.-ADDITIONAL REFINEMENTS

There are different variants of **α-β** to find the best terminal node as soon as possible and maximize the number of cut offs:

1. Ordered expansion at each level.
2. Preliminary ordering.

In game playing it is necessary:

- To use more intelligent procedures to prune the search tree rather than loads of computation.
- To use more intelligent procedures to get a dynamic algorithm behaviour.