Introduction To Artificial Intelligence

Illya Starikov

Last Modified: April 26, 2018

1 Introduction To AI

- What is AI?
  - Systems that act like humans
  - Systems that think like humans
  - Systems that think rationally
  - Systems that act rationally

2 Intelligent Agents

- Computer agents…
  - Perceive environment
  - Operate autonomously
  - Persist over prolonged periods
- Rational Agents…
  - Are affected by their environment.
  - Use sensors to interpret their environment
  - From said sensors, it acts on them via actuators
  - These are summarized by Figure 1. The main focus of the semester is filling in the `?`.

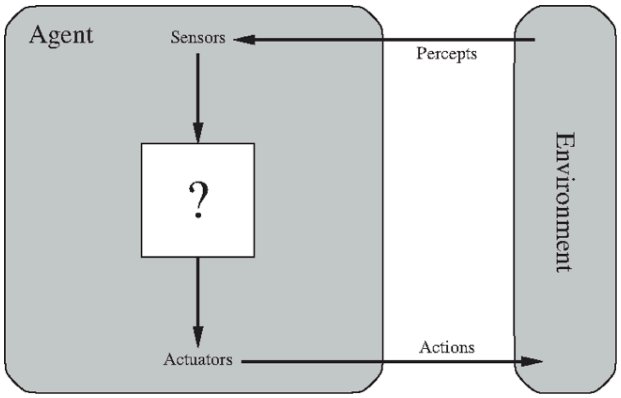

Figure 1: An Agent Cycle

- Rational behavior depends on…
  - Agent's performance measure
  - Agent's prior knowledge
  - Possible percepts and actions
  - Agent's percept sequence
- We define a rational agents as follows:

  > "For each possible percept sequence, a rational agent selects an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and any prior knowledge the agent has."

- PEAS[1] descriptoin and properties…

  Observable/Partially Observable If it is possible to determine the complete state of the environment at each time point from the percepts it is observable; otherwise it is only partially observable.

  Deterministic/Stochastic/Strategic If the next state of the environment is completely determined by the current state and the action executed by the agent, then it is determinstic. If the environment is deterministic except for the actions of other agents, then the environment is strategic. If it is randomly determined, then it is stochastic.

  Episodic/Sequential In an episodic environment, each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself. Subsequent episodes do not depend on the actions in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

  Static/Semi-Dynamic/Dynamic If an environement is unchanged while an agent is deliberating, it is static. The environment is semi-dynamic if the environment itself does not change with the passage of time but the agent's performance score does. If the environment is constantly changing, then it is dynamic.

  Discrete/Continuous If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete (For example, chess); otherwise it is continuous (For example, driving).

  Discrete/Continuous If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete (For example, chess); otherwise it is continuous (For example, driving).

  Single Agent/Multi-Agent The environment may contain other agents which may be of the same or different kind as that of the agent.

  Competitive/Cooperative If agents are in direct competition with each other (i.e., chess), they are competitive. If they have a common goal, they are cooperative.

  Known/Unknown An environment is considered to be known if the agent understands the laws that govern the environment's behavior. For example, in chess, the agent would know that when a piece is "taken" it is removed from the game.

- Some agent types include…

  Simple Reflex Agents Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the condition-action rule: `if condition then` action.

[1] Performance Measure, Environment, Actuators, and Sensors

Model-Based Reflex Agents A model-based agent can handle partially observable environments. Its current state is stored inside the agent maintaining some kind of structure which describes the part of the world which cannot be seen. This knowledge about "how the world works" is called a model of the world, hence the name "model-based agent".

Goal-Based Agents Goal-based agents further expand on the capabilities of the model-based agents, by using "goal" information. Goal information describes situations that are desirable. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. Search and planning are the subfields of artificial intelligence devoted to finding action sequences that achieve the agent's goals.

Utility-Based Agents Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is. This measure can be obtained through the use of a utility function which maps a state to a measure of the utility of the state. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. The term utility can be used to describe how "happy" the agent is.

Learning Agents Learning has the advantage that it allows the agents to initially operate in unknown environments and to become more competent than its initial knowledge alone might allow. The most important distinction is between the "learning element", which is responsible for making improvements, and the "performance element", which is responsible for selecting external actions.

- The definition of a problem solving agent is as follows…

  Problem-solving agents are goal based agents that decide what to do based on an action sequence leading to a goal state.

3 Introduction To Search

- Open-loop problem solving steps are as follows…
  1. Problem-formulation (actions & states)
  2. Goal-formulation (states)
  3. Search (action sequences)
  4. Execute solution
- All well-defined problems have the following…
  - An initial state
  - An actions set (usually denoted by ACTIONS(s))
  - A transition model (usually denoted by RESULT(s, a))
  - A goal test
  - A step cost (usually denoted by C(s, a, s'))
  - Path cost
  - Solution / optimal solution[2]
- Search trees…
  - Root corresponds to initial state
  - Search algorithms iterate through goal testing and expanding a state until goal is found
  - Order of state expansion is critical

```
1:  function TREE-SEARCH(problem)
2:      frontier ← using initial problem state
3:
4:      loop
5:          if EMPTY(frontier) then
6:              return Fail
7:          end if
8:
9:          choose leaf node and remove it from frontier
10:
11:         if chosen node contains goal state then
12:             return corresponding solution
13:         end if
```

[2]The optimal solution is just the solution with the least path cost

```
14:
15:         expand chosen node and add resulting nodes to frontier
16:     end loop
17: end function
```

- The keyword **choose** is critical; there are different ways to choose nodes, providing better search algorithms.
- However, this doesn't deal with repeated students, redundant paths, or loops. For graphs, we have as follows.

```
1: function GRAPH-SEARCH(problem)
2:     frontier ← using initial problem state
3:     explored set ← {}
4:
5:     loop
6:         if EMPTY(frontier) then
7:             return Fail
8:         end if
9:
10:        choose leaf node and remove it from frontier
11:
12:        if chosen node contains goal state then
13:            return corresponding solution
14:        end if
15:        explored set ← explored set ∪ chosen node
16:
17:        if chosen node ∉ frontier or explored set then
18:            expand chosen node and add resulting nodes to frontier
19:        end if
20:    end loop
21: end function
```

- Search node data structure
  - n.STATE
  - n.PARENT-NODE
  - n.ACTION
  - n.PATH-COST
- ***States are not search nodes***
- We define a child nodes as follows.

```
1: function CHILD-NODE(problem, parent, action)
```

2: **return** node with:
3: STATE = problem.RESULT(parent.STATE, action)
4: PARENT = parents
5: ACTION = action
6: PATH-COST = parents.PATH-COST + problem.STEP-COST(parent.STATE, action)
7: **end function**

- The frontier…
  - Is a set of leaf nodes
  - Implemented as a queue with operations…
    - EMPTY
    - POP
    - INSERT
  - Queue types: FIFO, LIFE (stack), and priority queue.
- Explored set…
  - Set of expanded nodes
  - Implemented typically as a hash table for constant time insertion and lookup
- Problem-solving performance…
  - Completeness
  - Optimality
  - Time complexity
  - Space complexity
- Complexity in AI…

  b Branching Factor

  d Depth of Shallowest Goal Node

  m Max Path Length in State Space

  Time Complexity Generated Nodes

  Space Complexity Max Number of Nodes Stored

  Search Cost Time + Space Complexity

  Total cost Search + Path Cost

4 Uninformed Search Algorithms

An uninformed (a.k.a. blind, brute-force) search algorithm generates the search tree without using any domain specific knowledge.

4.1 Breadth First Tree Search (BFTS)

- Frontier is a FIFO Queue
- Complete if b and d are finite
- Optimal if path-cost is non-decreasing function of depth
- Time complexity is $\mathcal{O}\left(b^d\right)$
- Space complexity is $\mathcal{O}\left(b^d\right)$

```
 1: function Breadth-First-Search(problem)
 2:     node ← a node with State = problem.Initial-State,Path-Cost = 0
 3:
 4:     if problem.Goal-Test(node.State) then
 5:         return Solution(node)
 6:     end if
 7:
 8:     frontier ← a FIFO queue with node as the only element
 9:     explored ← an empty set
10:
11:     loop
12:         if Empty?(frontier) then
13:             return failure
14:         end if
15:
16:         node ← Pop(frontier)                ▷ Chooses the shallowest node in frontier
17:         explored ← explored ∪ node.State
18:
19:         for all action ∈ problem.Actions(node.State) do
20:             child ← Child-Node(problem, node, action)
21:
22:             if child.State ∉ explored, frontier then
23:                 if problem.Goal-Test(child.State) then
24:                     return Solution(child)
25:                 end if
26:
27:                 frontier ← Insert(child, frontier)
```

```
            end if
        end for
    end loop
end function
```

4.2 Uniform Cost Search

- $g(n)$ is the lowest path-cost from start node to node n
- Frontier is a priority queue ordered by $g(n)$

```
function UNIFORM-COST-SEARCH(problem)
    node ← a node with STATE = problem.INITIAL-STATE,PATH-COST = 0
    frontier ← a priority ordered by PATH-COST, with node as the only element
    explored ← an empty set

    loop
        if EMPTY?(frontier) then
            return failure
        end if

        node ← POP(frontier)                    ▷ Chooses the shallowest node in frontier
        explored ← explored ∪ node.STATE

        for all action ∈ problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)

            if child.STATE ∉ explored, frontier then
                if problem.GOAL-TEST(child.STATE) then
                    return SOLUTION(child)
                end if

                frontier ← INSERT(child, frontier)
            end if
        end for
    end loop
end function
```

4.3 Depth First Tree Search (DFTS)

- Frontier: LIFO queue (Stack)
- Not Complete

- Not Optimal
- Time Complexity: $\mathcal{O}(b^m)$
- Space complexity: $\mathcal{O}(bm)$
- There exists a backtracking version
  - Space Complexity: $\mathcal{O}(b^m)$
  - Modifies rather than copies state description.
- The implementation is identical to Breath First Search, except the Frontier.

4.4 Depth-Limited Tree Search (DLTS)

- Frontier: LIFO queue (Stack)
- Not Complete When $l < d$
- Not Optimal
- Time Complexity: $\mathcal{O}(bl)$
- Space complexity: $\mathcal{O}(bl)$
- Diameter: Min number of steps to get from any state to any other state.

```
1: function DEPTH-LIMITED-SEARCH(problem, limit)
2:     return RECURSIVE-DLS(NODE(problem.INITIAL-STATE), problem, limit)
3: end function
4:
5: function RECURSIVE-DLS(node, problem, limit)
6:     if problem.GOAL-TEST(node.STATE) then
7:         return SOLUTION(node)
8:     else if limit = 0 then
9:         return cutoff
10:    else
11:        cutoff occurred? ← false
12:
13:        for Action ∈ problem.ACTIONS(node.STATE) do
14:            child ← CHILD-NODE(problem, node, action)
15:            result ← RECURSIVE-DLS(child, problem, limit - 1)
16:
17:            if result = cutoff then
18:                cutoff occurred?true
19:            else if result ≠ failure then
```

```
            return result
        end if
    end for

    if cutoff occurred? then
        return cutoff
    else
        return failure
    end if
  end if
end function
```

4.5 Iterative-Deepening Depth-First Tree Search (ID-DFTS)

- Complete if b is finite
- Optimal if path-cost is non-decreasing function of depth
- Time Complexity: $\mathcal{O}\left(b^d\right)$
- Space Complexity: $\mathcal{O}\left(()\, bd\right)$

```
function Iterative-Deepening-Search(problem)
  for depth = 0 to ∞ do
    result ← Depth-Limited-Search(problem, depth)

    if result ≠ cutoff then
      return result
    end if
  end for
end function
```

4.6 Bidirectional Breath First Search (BiBFTS)

- Complete if b is finite
- Not optimal "out of the box"
- Time Complexity: $\mathcal{O}\left(b^{d/2}\right)$
- Space Complexity: $\mathcal{O}\left(b^{d/2}\right)$

5 Informed Search Algorithms

An informed search algorithm generates the search tree with using any domain specific knowledge.

Note, for these algorithms,

$g(n) =$ lowest path-cost from start node to node n
$h(n) =$ estimated non-negative path-cost of cheapest path from node n to a goal node
$=$ with $h(goal)$

where,

- $h(n)$ is a heuristic function.
- Heuristics incorporate problem-specific knowledge.
- Heuristics need to be relatively efficient to compute.
- The function $h(n)$ is admissible if

$$\forall n \, (h(n) \leq C^*(n))$$

  In other words, a heuristic is admissible if the estimated cost is never more than the actual cost from the current node to the goal node.

- The function $h(n)$ is consistent if

$$\forall n, n' \, (h(n) \leq c(n, a, n') + h(n'))$$

  In other words, a heuristic is consistent if the cost from the current node to a successor node, plus the estimated cost from the successor node to the goal is less than or equal to the estimated cost from the current node to the goal.

For the major search search algorithms:

5.1 A^* Proof

Theorem 1. *A^*TS employing heuristic $h(n)$ is optimal if $h(n)$ is admissible.*

Proof. Suppose suboptimal goal node G appears on the frontier and let the cost of the optimal solution be C^*. From the definition of $h(n)$ we know that $h(G) = 0$ and because G is suboptimal we know $f(G) > C^*$. Together this gives $f(G) = g(G) + h(G) = g(G) > C^*$.

If there is an optimal solution, then there is a frontier node N that is on the optimal solution path. Our proof is for an admissible heuristic, so we know $h(N)$ does not overestimate; thus $f(N) = g(N) + h(N) \leq C^*$.

Together this given $f(N) \leq C^* < f(G) \implies f(N) < f(G)$.

As A^*TS expands lower f-cost nodes before higher f-cost nodes, N will always be expanded before G. Thus, A^*TS is optimal. □

5.2 Consistent $\implies$ Admissible

Theorem 2. *Every consistent heuristic is admissible*

Proof. Take an arbitrary path to an arbitrary goal node G denoted $S_1, S_2, S_3, \ldots, S_G$. As h is consistent, we know $\forall n, n' : h(n) \leq c(n, a, n') + h(n')$.

$S_1\, S_2$	$h(S_1) - h(S_2) \leq c(S_1, S_2)$
$S_2\, S_3$	$h(S_2) - h(S_3) \leq c(S_2, S_3)$
$\vdots$	$\vdots$
$S_{G-1}\, S_G$	$h(S_{G-1}) - h(S_G) \leq c(S_{G-1}, S_G)$
$S_1\, S_G$	$h(S_1) \leq c(S_1, S_G)$
$S_2\, S_G$	$h(S_2) \leq c(S_2, S_G)$
$\vdots$	$\vdots$
$S_k\, S_G$	$h(S_k) \leq c(S_k, S_G)$

□

As this is true for all goal nodes S_G, it's also true for the optimal goal node. Thus,

$$\forall h \in \{1, 2, \ldots, h\}, h(S_k) \leq C^*(S_k)$$

which is the definition of admissibility.

5.3 Best First Search

- Select node to expand based on evaluation function $f(n)$.
- Node with lowest $f(n)$ selected as $f(n)$ correlated with path-cost.
- Represent frontier with priority queue sorted in ascending order of f-values.

5.4 Greedy Best First Search

- Incomplete (so also not optimal).
- Worst-case time complexity and space complexity: $\mathcal{O}(b^m)$.
- Actual complexity depends on accuracy of $h(n)$.

5.5 A^* Search

- $f(n) = g(n) + h(n)$.
- $f(n)$ is estimated cost of optimal solution through node n
- If $h(n)$ satisfies certain conditions, $A*$ Search is complete optimal
- Is Optimally efficient for consistent heuristics.
- Run-time is a function of the heuristic error.
- A^* Graph Search not scalable due to memory requirements.

6 Adversarial Search

An adversarial environment is characterized by:

- Competitive multi-agent
- Turn-taking

The simplest types are:

- Discrete
- Deterministic
- Two-Player
- Zero-Sum Games
- Perfect Information

Now, the search problem is described by:

S_0 Initial State

Player(s) Which player has the move?

Actions(s) Set of legal moves.

Result(s, a) Defines transitional model.

Terminal-Test(s) Is this a game over state?

Utility(s) Associates player-dependent values with terminal states.

6.1 Minimax

- Time complexity: $\mathcal{O}(b^m)$.
- Space complexity: $\mathcal{O}(b\,m)$.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Max's}\textsc{Utility}(s) & \leftrightarrow \textsc{Terminal-Test}(s) \\ \max_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \leftrightarrow \textsc{Player}(s) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,\ a)) & \leftrightarrow \textsc{Player}(s) = \textsc{Min} \end{cases}$$

Just as there is Depth-Limited Depth First Search, there's Depth-Limited Minimax:

- State Evaluation Heuristic estimates Minimax value of a node.
- Note that the Minimax value of a node is always calculated for the Max player, even when the Min player is at move in that node.

$$\textsc{Minimax}(\textsc{s},\ \textsc{d}) = \begin{cases} \textsc{Eval}(\textsc{s}) & \leftrightarrow \textsc{Cuttoff-Test}(\textsc{s}) \\ \max_{a \in \textsc{Actions}(\textsc{s})} \text{H-}\textsc{Minimax}(\textsc{Result}(\textsc{s},\ \textsc{a}),\ \textsc{d} + 1) & \leftrightarrow \textsc{Player}(\textsc{s}) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(\textsc{s})} \text{H-}\textsc{Minimax}(\textsc{Result}(\textsc{s},\ \textsc{a}),\ \textsc{d} + 1) & \leftrightarrow \textsc{Player}(\textsc{s}) = \textsc{Min} \end{cases}$$

6.2 State Evaluation

A good state evaluation heuristic should:

1. Order the terminal states in the same way as the utility function.
2. Be relatively quick to compute.
3. Strongly correlate non-terminal states with chance of winning.

Also, to take the weighted linear state evaluation heuristic:

$$\textsc{Eval}(\textsc{s}) = \sum_{i=1}^{n} w_i\, f_i(s) \tag{1}$$

6.3 Alpha-Beta Pruning

Some terminology:

α Worst value that Max will accept at this point of the search tree.

β Worst value that Min will accept at this point of the search tree.

Fail-Low Encountered value $\leq \alpha$.

Fail-High Encountered value $\geq \beta$.

Prune If fail-low for Min-player or fail-high for Max-player.

With Alpha-Beta Pruning, we get the following complexities:

- Worst-Case: $\mathcal{O}\left(b^d\right)$.
- Best-Case: $\mathcal{O}\left(b^{d/2}\right)$
- Average-Case: $\mathcal{O}\left(b^{3d/4}\right)$.

6.4 Move Ordering

Some heuristics for move ordering:

Knowledge Based Try captures first in chess.

Principal Variant (PV) A sequence of moves that programs consider best and therefore expect to be played.

Killer Move The last move at a given depth that caused $\alpha\beta$-pruning or had best minimax value.

History Table Track how often a particular move at any depth caused $\alpha\beta$-pruning or had best minimax value.

6.4.1 History Table

There are two options for history tables:

1. Generate set of legal moves and use History Table value as f value.
2. Keep moves with History Table values in a sort array and for a given state traverse the array to find the legal move with the highest History Table value.

6.5 Search Depth Heuristics

Some search depth heuristics can include:

- Time-Based/State-Based
- Horizontal Effect[3]
- Singular Extensions/Quiescence Search

When there is a time to move constraint, some plausible algorithms include:

- Constant
- Percentage of remaining time
- State dependent
- Hybrid

[3]The phenomenon of deciding on a non-optimal principal variant because an ultimately unavoidable damaging move seems to be avoided by blocking it till passed the search depth.

6.6 Quiescence Search

Quiescence states are states that could have giant swings in the heuristic values. For these states,

- When search depth reached, compute quiescence state evaluation heuristic.
- If state quiescent, then proceed as usual; otherwise increase search depth if quiescence search depth not yet reached.

6.7 Transition Tables

A transition table is a hash table of previously calculated state evaluation heuristic values. The speedup is particularly large for iterative deepening search algorithms.

7 State-Space Search

State space search is a process in which successive configurations or states of an instance are considered, with the intention of finding a goal state with a desired property. Some concepts related to state-space search are:

- Complete-state formulation
- Objective function
- Global optima
- Local optima
- Ridges, plateaus, and shoulders
- Random search and local search

Some examples are included below.

7.1 Hill Climbing

7.1.1 Steepest-Ascent Hill-Climbing

Steepest-Ascent Hill-Climbing is simply a loop that continually moves in the direction of increasing value; that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

```
1: function HILL-CLIMBING(problem)
2:     current ← MAKE-NODE(problem.INITIAL-STATE)
3:
4:     loop
5:         neighbor ← highest-valued successor of current
6:         if neighbor.VALUE ≤ current.VALUE then
7:             return current.STATE
8:         end if
9:
10:        current ← neighbor
11:    end loop
12: end function
```

7.1.2 Stochastic Hill-Climbing

The Stochastic version is almost identical to Steepest-Ascent Hill-Climbing, except it chooses at random from among uphill moves. The probability of selection can vary with the steepness of the uphill move. It does show on average slower convergence, but also less chance of premature convergence

7.1.3 First-Choice Hill-Climbing

The First-Choice version is almost identical to Steepest-Ascent Hill-Climbing, except it chooses the first randomly generated uphill move. Although it is greedy, incomplete, and suboptimal, it's also very practical when the number of successors is large. It has an even slower convergence rate, but also a low chance of premature convergence as long as the move generation order is randomized

7.1.4 Random-Restart Hill-Climbing

The Random-Restart version is almost identical to all hill climbing, except it restarts until a goal is found. It's trivially complete. The expected number of restarts is:

$$\text{restarts} = \frac{1}{p}$$

where p is the probability of a successful hill climb given a random initial state.

7.2 Simulated-Annealing

Simulated-Annealing is hill-climbing except instead of picking the best move, it picks a random move. If the selected move improves the solution, then it is always accepted. Otherwise, the algorithm makes the move anyway with some probability less than 1. The probability decreases exponentially with the "badness" of the move, which is the amount ΔE by which the solution is worsened (i.e., energy is increased.)

```
1: function SIMULATED-ANNEALING(problem, schedule)
2:     current ← MAKE-NODE(problem.INITIAL-STATE)
3:
4:     for t = 1 to ∞ do
5:         T ← schedule(t)
6:         if T = 0 then
7:             return current.STATE
8:         end if
9:
10:        next ← random successor of current
11:        ΔE ← next.VALUE − current.VALUE
12:        if ΔE > 0 then
13:            current ← next
14:        else
15:            current ← next only with probability e^(ΔE/T)
16:        end if
17:    end for
18: end function
```

7.3 Population Based Local Search

7.3.1 Deterministic Local Beam Search

The local beam search is *similar* to Steepest-Ascent Hill-Climbing. The algorithm keeps track of k states rather than a single state. Although this may seem like Random-Restart Hill-Climbing with k random restarts.

In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the parallel search threads. In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing.

7.3.2 Stochastic Beam Search

Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

7.3.3 Evolutionary Algorithms

A genetic algorithm (GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.

```
1: function SIMULATED-ANNEALING(problem, FITNESS-FUNCTION)
2:     while some individual is fit enough, or enough tie has elapsed do
3:         NEW_POPULATION ← empty set
4:
5:         for i = 1 to SIZE(population) do
6:             x ← RANDOM-SELECTION(population, FITNESS-FUNCTION)
7:             y ← RANDOM-SELECTION(population, FITNESS-FUNCTION)
8:             child ← REPRODUCE(x, y)
9:
10:            if small random probability then
11:                child ← MUTATE(child)
12:            end if
13:            add child to new_population
14:
15:
16:            return the best individual in population, according to FITNESS-FUNCTION
17:
18:            function REPRODUCE(x, y)
```

```
19:             n ← LENGTH(x)
20:             c ← random number from 1 to n
21:
22:             return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
23:         end function
```

7.3.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a stochastic population-based optimization technique which assigns velocities to population members encoding trial solutions. It acts according to the rule:

$$\overrightarrow{v}^{+} = c_1 \cdot rand() \cdot (\overrightarrow{p_{\text{best}}} - \overrightarrow{p_{\text{present}}}) + c_2 \cdot rand() \cdot (\overrightarrow{g_{\text{best}}} - \overrightarrow{p_{\text{present}}})$$

and updating $\overrightarrow{p_{\text{present}}}$ as:

$$\overrightarrow{p_{\text{present}}} += \overrightarrow{v}$$

7.3.5 Ant Colony Optimization

In Ant Colony Optimization (ACO), a set of software agents called artificial ants search for good solutions to a given optimization problem. To apply ACO, the optimization problem is transformed into the problem of finding the best path on a weighted graph. The artificial ants (hereafter ants) incrementally build solutions by moving on the graph. The solution construction process is stochastic and is biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.

7.4 Online Search

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment (Chapter 17 relaxes these assumptions), but we stipulate that the agent knows only the following:

- ACTIONS(s)
- c(s, a, s')[4]
- GOAL-TEST(s)

Online Search problems typically interleave computation and action. They environment is typically dynamic, non-deterministic, unknown domains. For the purposes of this section, we assume all exploration problems are safely explorable.

We define the following terms for Online Search Agents:

[4]This cannot be used until RESULT(s, a).

CR Competitive Ration

TAPC Total Actual Path Cost

C* A Optimal Path Cost

From this, we get the Competitive Ratio as follows

$$CR = \frac{TAPC}{C^*}$$

From this, we can see we have the best case of $CR = 1$ and the worst case of $CR = \infty$.

To actually implement online agents, we introduce two new algorithms: Online-DFS-Agent and LRTA* Agent.

7.5 Online-DFS-Agent

```
1: function Online-DFS-Agent(s')
2:     if Goal-Test(s') then
3:         return stop
4:     end if
5:
6:     if s' is a new state (not in untried) then
7:         untried[s'] ← Actions(s')
8:     end if
9:
10:    if s is not null then
11:        result[s, a] ← s'
12:        add s to the front of unbacktracked[s']
13:    end if
14:    if untried[s'] is empty then
15:        if unbacktracked[s'] is empty then
16:            return stop
17:        else
18:            a ← an action b such that result[s', b] = Pop(unbacktracked[s'])
19:        end if
20:    else
21:        a ← Pop(untried[s'])
22:    end if
23:
24:    s ← s'
25:    return a
26: end function =0
```

7.6 LRTA*-Agent

```
1: function LRTA*(s')
2:     if Goal-Test(s') then
3:         return stop
4:     end if
5:
6:     if s' is a new state (not in H) then
7:         H[s'] ← h(s')
8:     end if
9:
10:    if s is not null then
11:        result[s, a] ← s'
12:        H[s] ← min (LRTA*-Cost(s, b, result[s, b], H) ∀b ∈ Actions(s))
13:    end if
14:
15:    a ← an action b in Actions(s') that minimizes LRTA*-Cost(s', b, result[s', b], H)
16:    s ← s'
17:    return a
18: end function
19:
20: function LRTA*-Cost(s, a, s', H)
21:    if s' is undefined then
22:        return h(s)
23:    end if
24:
25:    return c(s, a, s') + H[s']
26: end function
```

8 AI Ethics

For background, some key historical events in AI were as follows:

4th Century BC Aristotle propositional logic.

1600s Descartes mind-body connection.

1805 First programmable machine.

Mid 1800s Charles Babbage's "difference engine" "analytical engine".

Sometime Lady Lovelace's Objection

1847 George Boole propositional logic

1879 Gottlob Frege predicate logic

1931 Kurt Godel: Incompleteness Theorem

> In any language expressive enough to describe natural number properties, there are undecidable (incomputable) true statements

1943 McCulloch & Pitts: Neural Computation

1956 Term "AI" coined

1976 Newell & Simon's "Physical Symbol System Hypothesis" A physical symbol system has the necessary and sufficient means for general intelligent action

1974–80, 1987–93 AI Winters

1980+ Commercialization of AI

1986+ Rebirth of Artificial Neural Networks

1990s Unification of Evolutionary Computation

200+ Rise of Deep Learning

8.1 Weak Vs. Strong AI Hypothesis

The Week AI vs. Strong AI Hypothesis is thus:

Weak AI Hypothesis It's possible to create machines that can act as if they're intelligent.

Strong AI Hypothesis It's possible to create machines that actually think.

This closely relates to the following problems:

- Rene Descartes (1596–1650)
- Rationalism, Dualism, Materialism, Brain in a Vat
- Star Trek & Souls
- Chinese Room[5]

8.2 Ethics

Some ethical concerns with regards to AIs are as follows:

- Autonomous AIs and the Trolley DilemmaUnemployment Inequality
- Human dependency and obsoleteness
- Bias transfer
- Security
- Human-robot relationships
- Rights of sentient beings.

[5]The Chinese room argument holds that a program cannot give a computer a "mind", "understanding" or "consciousness", regardless of how intelligently or human-like the program may make the computer behave.