Revision notes - CS3243

Ma Hongqiang

February 20, 2018

Contents

1 Introduction

1.1 Definition of Artificial Intelligence(AI)

Artificial Intelligence has the following defining behaviour:

1. Think like a human
2. Think rationally
3. Act like a human
4. Act rationally

Turing test can be used to test (3) whether AI is acting humanly.
For (2), there are problems like

- Can **all intelligent** behaviour be captured by logical rules?
- Due to **computational issues**, a logical solution in principle does not translate to practice.

To (4) act rationally, we define rational behaviour as doing the "right thing", whose definition is subjective:

- Best for whom?
- What to optimise?
- What information is available?
- Unintended effects?

1.2 Rational Agents

Definition 1.1 (Agent)**.**
An agent is an entity that **perceives and acts**.
Formally, an agent is a function f from percept histories to actions, i.e.,

$$f : P^* \to A$$

Ideally, we seek the best-performing agent for a certain task subject to computation limits.

2 Intelligent Agents

From the previous section, we know that agents are anything that can be viewed as **perceiving** its **environment** through **sensors**; **acting** upon that environment through **actuators**. In other words, sensors and actuators are two interfaces through which agent program interacts with the environment.
So an agent comprises of both **architecture** and **program**.

Definition 2.1 (Rational Agent)**.**
A rational agent is an agent that maximise agent success based on what it can perceive and the actions it can perform.
The agent success is judged by a performance measure, defined as an objective criterion for meansuring success of an agent's behaviour.

Rational agents behave in a way such that for each possible percept sequence, it selects an action that is expected to maximise its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.
Remark: Rationality $\neq$ omniscience. Agents can perform **exploration** that help them gather useful information, before **exploitation** of known information.

Definition 2.2 (Autonomous Agent)**.**
An agent is **autonomous** if its behaviour is determined by its own experience, with ability to learn and adapt.

2.1 Task Environment

To specify the task environment for the design of intelligent agent, we focuses on **PEAS**:

- Performance measure
- Environment
- Actuators
- Sensors

There are several properties of task environment listed down in the following table. Detailed explanation in the next page.

Properties	**Complement**
Fully observable	Partially observable
Deterministic	Stochastic
Episodic	Sequential
Static	Dynamic
Discrete	Continuous
Single agent	Multi-agent

Fully observable means sensors provide access to the complete state of the environment at each point in time.
Deterministic means the next state of the environment is completely determinedby the current state and the action executed by the agent.
Episodic means (1) the agents experience is divided into atomic episodes (each episode consists of the agent perceiving and then performing a single action); and (2) the choice of action in each episode does not depend on actions in past episodes.
Static means the environment is unchanged while an agent is deliberating.
Discrete means there exists a finite number of distinct states, percepts, and actions.
Single agent means there is only one agent operating by iteself in an environment.

2.2 Implementation of Agent Function

Since an agent is completely specified by the **agent function** which maps percept sequences to actions, we need to ensure one, or a small equivalence class of it is **rational**.

Definition 2.3 (Table-Lookup Agent)**.**
Table-lookup agent is defined by the following agent function:

---

function TABLE-DRIVEN-AGENT(`percept`) **return** `action`
  static: `percepts`, a sequence initially empty
    `table`, a table of actions, indexed by percept sequences, fully specified
  append `percept` to the end of `percepts`
  `action`← LOOKUP(`percepts`,`table`)
end function

It has the following drawbacks:

- Huge table to store
- Take a long time to build the table
- No autonomy: impossible to learn all correct table entries from experience
- No guidance on filling the correct table entries

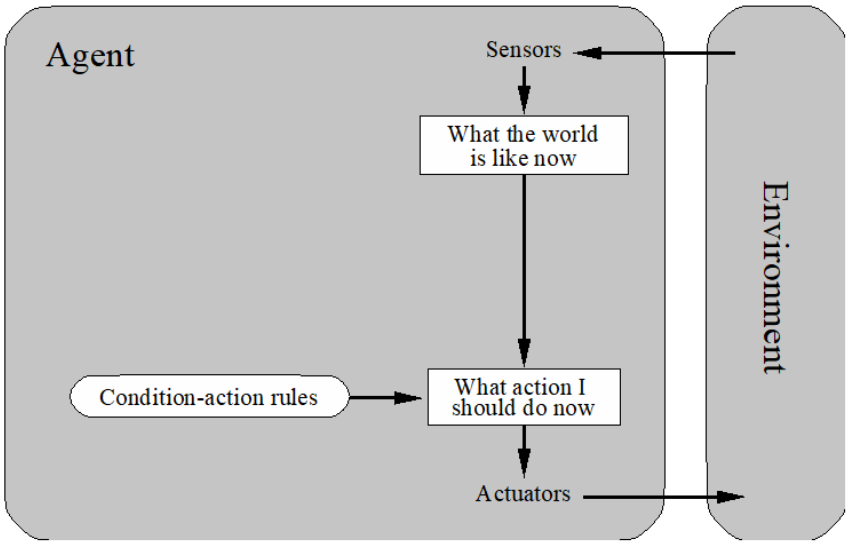
2.3 Agent Types

There are 4 basic types of agents in order of *increasing* generality:

- Simple reflex agent
- Model-based reflex agent
- Goal-based agent
- Utility-based agent

2.3.1 Simple Reflex Agent

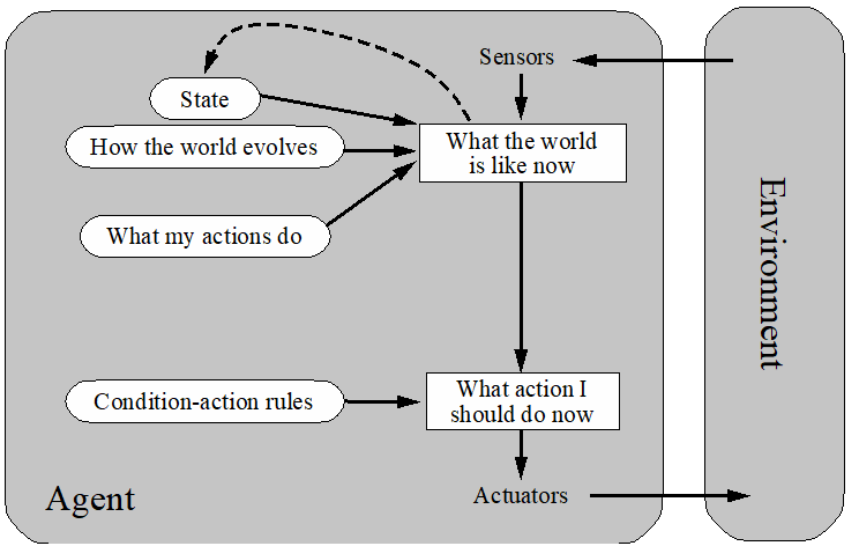
A simple reflex agent selects actions based on the **current** percept, ignoring rest of percept history.
What the world is like now: `interpret_input`; what action I should do now: `rule_match` It



is considered a computationally cheap agent program, used to achieve instantaneous reaction.

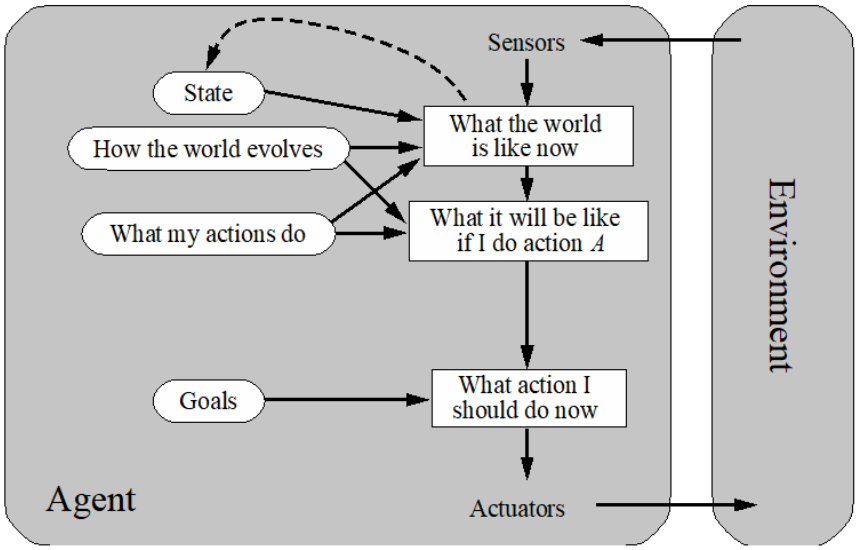
2.3.2 Model-based Reflex Agent

A model-based reflex agent handles partial observability by keeping track of environment state it cannot perceive now. Therefore, it depends on percept history.
What the world is like now: `update_state`; what action I should do now: `rule_match`



2.3.3 Goal-based Agent

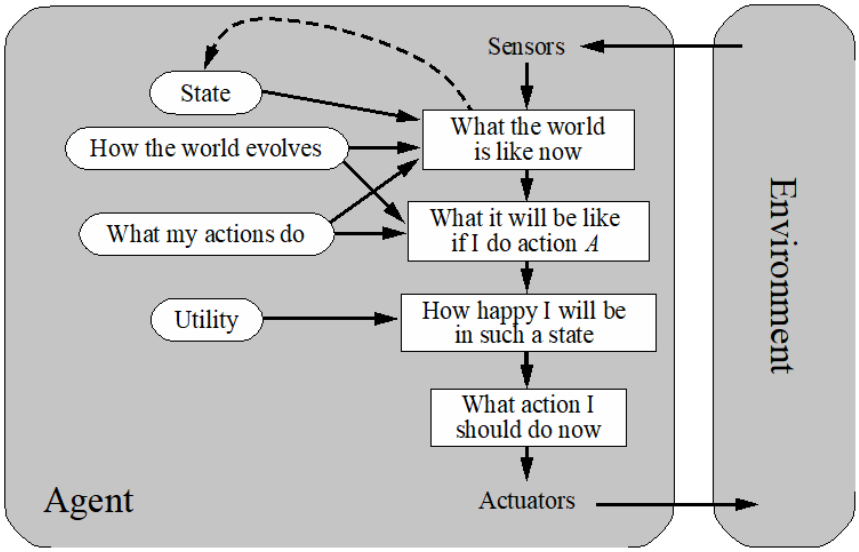
A goal-based agent has a goal which is used to describe which states are desirable. It also considers next state: whether action A result in a next state which achieves the goal? While



it is less efficient, it is more flexible since the goal can be easily changed without recoding all condition-action rules.

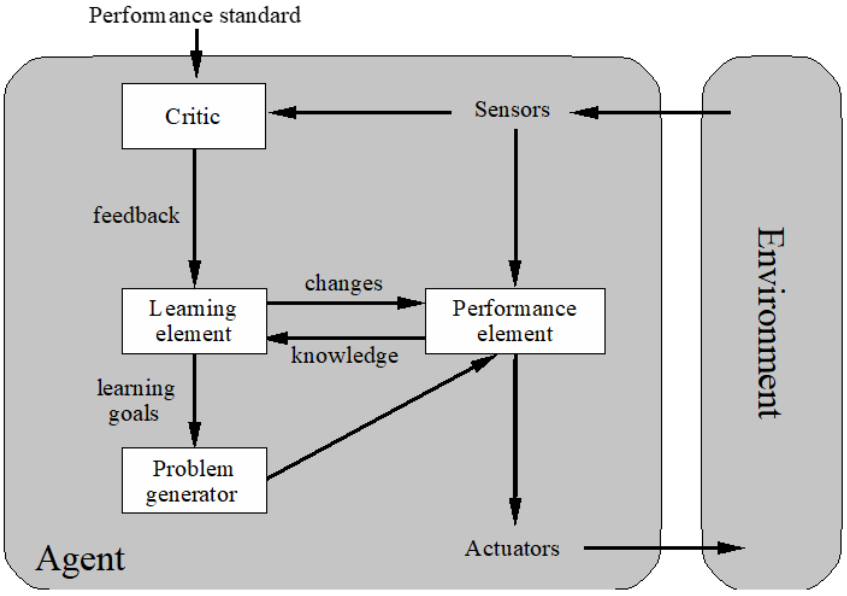
2.3.4 Utility-based Agent

A utlity-based agent can handle partially observable and stochastic properties very well using expectation.



2.3.5 Learning Agent

Learning agent is to operate in initially unknown environments and perform better than other agents with prior knowledge. Components in the learning agent include



- Performance element: selects the external actions
- Learning element: improves agent to perform better
- Critic: provides feedback on how well the agent is doing
- Problem generator: suggests explorative actions that will lead to new, informative (but not necessarily better) experiences

2.4 Exploitation vs. Exploration

An agent operating in the real world must choose between

1. maximizing its expected utility according to its current knowledge about the world; and
2. trying to learn more about the world since this may improve its future gains.

This problem is known as the trade-off between exploitation and exploration.

3 Uninformed Search

In this section, searching is conducted in fully **observable**, **deterministic** and **discrete** environment.
Under these conditions, the solution to any problem is a fixed sequence of actions.

3.1 Problem Formulation

Usually the problem needs to be modelled before searching can be conducted.
A problem can be defined formally by five components:

- **Initial state** that the agents starts in
- A description of possible **actions** available to the agent
- **Transition model**, i.e., a description of what each action does
- A goal test which determines whether a state S is a goal state. The goal states can be
  - explicitly specified as a set, e.g., $\{InWork\}$, or
  - implicit function, e.g., $isCheckMate(s)$.
- A **Path cost** function that assigns a numeric cost to each path.

Definition 3.1 (Cost Function and Path Cost)**.**
A cost function from state s to s' under action a is denoted as

$$c(s, a, s')$$

For this section, cost is non-negative, i.e., $c \geq 0$.
Path cost is usually additive and is calculated from $\sum_{c \in \text{Path}} c$.

Below is a simple problem-solving agent.
function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
  persistent: *seq*, an action sequence, initially empty
    state, somne description of the current world state
    goal , a goal, initially null
    problem, a problem formulation
  state← UPDATE-STATE(*state*, *percept*)
  if *seq* is empty **then**
    goal ←FORMULATE-GOAL(*state*)
    problem ←FORMULATE-PROBLEM(*state*, *goal*)
    seq ←SEARCH(*problem*)
    if *seq* = *failure* **then return** a null action
  action ←FIRST(*seq*)
  seq ←REST(*seq*)
  return action
In consideration of limited computation resources, state space has to be abstracted, where

- state is abstracted to set of real states
- action is abstracted to complex combination of real actions
- solutions is abstracted to set of real paths

Such abstraction needs to be both **valid** and **useful**.

3.2 Tree Search Algorithms

Definition 3.2 (Tree Search Algorithm)**.**
Below is the general tree search algorithm.
function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Remark: In tree search, the exploration is **memoryless**, so **redundant paths** are unavoidable.

3.3 Graph Search Algorithms

Definition 3.3 (Graph Search Algorithm)**.**
Below is the general graph search algorithm. It augments the tree search algorithm with a **explored set**.
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Nodes in graph search algorithms usually contain the following information:

- **State** to which the node corresponds
- **Parent** node which generates this node
- **Action** that was applied to the parent to generate this node
- **Path Cost** $g(n)$ of the path from the initial state to the node

Remark: Search strategies are mostly determined by the **order of node expansion**. The performance of search algorithms can be measured in

- Completeness: Solution guaranteed
- Optimality: Optimal solution found
- Time complexity
- Space complexity

Definition 3.4 (Parameters for Analysis)**.**
We define

- b: maximum number of successors of any node
- d depth of shallowest goal node, and
- m: maximum depth of search tree

to assist the measurement of performance.

Next, we look into 4 **uninformed search**, mearning that the strategies have no additional information about states beyond that provided in the problem definition.

3.3.1 Breadth First Search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ←a FIFO queue with node as the only element
  explored ←an empty set
  loop do
    if EMPTY?( frontier) then return failure
    node←POP( frontier ) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ←CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child )
        frontier ←INSERT(child , frontier )
```

Above is breadth-first search on a graph.
Remark: Breadth-first search is optimal only if step costs are all identical.

3.3.2 Uniform-cost Search

Uniform-cost search expands on the idea of BFS, with frontier implemented by a priority queue.
With this modification, UCS is in general optimal. **function** UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure
node ←a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ←a priority queue ordered by PATH-COST, with *node* as the only element
explored ←an empty set
loop do
if EMPTY?(*frontier*) **then return** failure
node←POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
add *node*.STATE to explored
for each *action* in *problem*.ACTIONS(*node*.STATE) **do**
child ←CHILD-NODE(*problem*,*node*, *action*)
if *child*.STATE is not in *explored* or *frontier* **then**
frontier ←INSERT(*child* , *frontier*)
else if *child*.STATE is in *frontier* with higher PATH-COST then
replace that *frontier* node with *child*

3.3.3 Depth First Search

Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search backs up to the next deepest node that still has unexplored successors.
Its frontier is implemented using a stack.

3.3.4 Depth-limited Search

DFS may fail if there is infinite state spaces. This can be solved using Depth Limited Search, by supplying DFS with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors.
function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
else if *limit* = 0 *then return* cutoff
else
cutoff_occurred?←false
for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child ←CHILD-NODE(*problem*, *node*, *action*)
result ←RECURSIVE-DLS(*child* , *problem*, *limit* − 1)
if *result* = *cutoff* **then** *cutoff_occurred*?←true
else if *result* ≠ *failure* **then return** *result*
if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*

3.3.5 Iterative Deepening DFS

Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.
It does this by gradually increasing the limitfirst 0, then 1, then 2, and so onuntil a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
for *depth* = 0 to ∞ **do**
result ←DEPTH-LIMITED-SEARCH(*problem*, *depth*)
if *result* ≠ cutoff **then return** *result*

3.3.6 Comparing Uninformed Seaerch Strategies

Criterion	BFS	UCS	DFS	DLS	IDS	Bidirectional
Complete	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Time	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{\frac{d}{2}})$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{\frac{d}{2}})$
Optimal	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]

Remark:

1. Complete if b is finite
2. Complete if step costs $\geq \epsilon$ for positive ϵ.
3. Optimal if step costs are all identical
4. If both direction use BFS

Choosing the search strategy really depends on the problem:

- Finite/infinite depth of search tree
- Known/unknown solution depth
- Repeated States
- Identical/Non-identical step costs
- Completeness and Optimality Requirement
- Resource constraints(time, space)

4 Informed Search

Informed search exploits problem-specific knowledge and obtains heuristics to guide the search.

4.1 Best First Search

Best first search uses an **evaluation function** $f(n)$ for each node n. $f(n)$ will be the cost **estimate** to reach the goal from n.
The strategy is to expand node with lowest evaluation cost first. In other words, the strategy is identical to UCS except we use f instead of g for cost evaluation.

4.2 Greedy Best First Search

Greedy best-first search expands the node that appears to be closest to goal.
The evaluation function $f(n) := h(n)$ here $h(n)$ is the **heuristic** function that estimates the cost of cheapest path from n to goal.
The performance of Greedy Best First Search is as below.

Complete?	Yes, if b is finite
Optimal?	No
Time	$O(b^m)$
Space	Max size of frontier $O(b^m)$

4.3 A^* search

A^* search uses the information g to avoid expanding paths that are already expensive.
The evaluation fucntion for A^* search is $f(n) = g(n) + h(n)$. f can be interpreted as the estimated cost of cheapest path through n to goal.

Definition 4.1 (Admissible Heuristics)**.**
A heuristic $h(n)$ is **admissible** if, for every node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost to reach the goal state from n.

An admissible heuristic is **optimistic** in the sense that it *never* overestimates the cost to reach the goal.

Theorem 4.1.
If $h(n)$ is admissible, then $A^\star$ using Tree-Search is optimal.

Definition 4.2 (Consistent Heuristics)**.**
A heuristic is **consistent** if, for every node n and every successor n' of n generated by any action a,

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

If h is consistent, then $f(n)$ is non-decreasing along any path.

Theorem 4.2.
Consistency implies admissibility. This can be shown easily using proof by contradiction.

Theorem 4.3.
If $h(n)$ is consistent, then A^* using GRAPH-SEARCH is optimal.

Properties of A^* search is as below. **Remark**: There are more efficient heuristic search

Complete?	Yes, if there is finite # nodes with $f(n) \leq f(G)$
Optimal?	Yes
Time	$O(b^{h^*(s_0)-h(s_0)})$ where $h^*(s_0)$ is actual cost of getting from root to goal
Space	Max size of frontier $O(b^m)$.

algorithms in 3.5.3 of the book to tackle the exponential memory usage drawback. One algorithm is **recursive best-first search** and another one is (simplifed) Memory Bounded A^*.

4.4 Formation of Heuristics

As the time complexity of A^* depends on the strength of heuristic, forming good heuristics is critical to A^*.

Definition 4.3 (Dominance)**.**
If two admissible heuristics h_1, h_2 admits $h_2(n) \geq h_1(n)$ for all n, then h_2 **dominates** h_1.

It follows that h_2 incurs lower search cost than h_1.

Definition 4.4 (Relaxed Problem)**.**
A problem with fewer restriction on all actions is called a **relaxed problem**.

We can use relaxed problem to generate admissible heuristics.

Theorem 4.4.
The cost of an optimal solution to a relaxed problem is an admissible heuristics for the original problem.

Usually, the more relaxed the problem, the worse the heuristics.

4.5 Local Search

In local search, the path to goal is *irrelevant*; the goal state *itself* is the solution.
Therefore, the state space for local search is the set of **complete** configurations.
Local search algorithms maintain single "current best" state and try to improve it.
It has the advantage of

- very little memory usage
- find reasonable solutions in large state space

4.5.1 Hill-Climbing Search

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current←MAKE-NODE(*problem*.INITIAL-STATE)
  loop do
    neighbour← a highest-valued successor of *current*
    if *neighbour*.VALUE≤*current*.VALUE **then return** *current*.STATE
    current←*neighbour*

The above is the algorithm for hill-climbing search.
The problem with hill-climbing search is that, depending on initial state, it may end in local maxima, not global maxima.
The **non-guaranteed** fixes include sideway moves or random restarts.

4.5.2 Simulated Annealing

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  current← MAKE-NODE(*problem*.INITIAL-STATE)
  for $t = 1$ to ∞ **do**
    $T \leftarrow$*schedule*(t)
    if $T = 0$ **then return** *current*
    next→ a randomly selected successor of *current*
    $\Delta E \leftarrow$*next*.VALUE-*current*.VALUE
    if $\Delta E > 0$ **then** *current*←*next*
    else *current*←*next* only with probability $e^{\frac{\Delta E}{T}}$

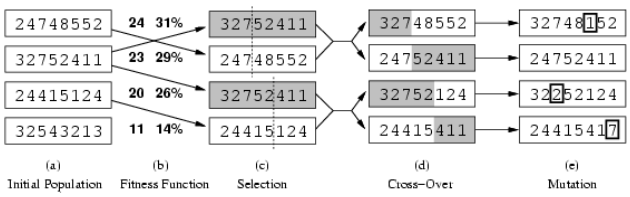
Theorem 4.5 (Properties of Simulated Annealing)**.**
If T decreases slowly enough, then simulated annealing search will find a global optimumm with probability approacing 1.

4.5.3 Local Beam Search

Local Beam Search adopts the idea to keep several best states during each iteration of local search.
The problem is it may quickly concentrate in small region of state space.
Solution to the above problem is stochastic beam search.

4.5.4 Genetic Algorithms

The idea of Genetic Algorithms is that a successor state is generated by combining two parent states.
The algorithm starts with k randomly generated states(population), and there is a evaluation fucntion(fitness function) whose higher value indicates better states. The algorithm produces the next generation of states by selection, crossover and mutation. Local search is very useful in space exploration problem.


24748552
32752411
24415124
32543213
24 31%
23 29%
20 26%
11 14%
32752411
24748552
32752411
24415124
32748552
24752411
32752124
24415411
32748152
24752411
32252124
24415417
(a)
Initial Population
(b)
Fitness Function
(c)
Selection
(d)
Cross–Over
(e)
Mutation

5 Adversarial Search

For this section, we mainly cover deterministic games.

5.1 Game: Adversarial Search Problems

Games is a variant of search problems but there are two unique features in games.

1. Utility Maximising opponent
   Solution is a strategy specifying a move for every possible opponent response.
2. Time Limit
   Unlikely to find goal, must approximate

For the two-player fixed sum game, we label them as

- MAX player, who wants to maximise value
- MIN player, who wants to minimise value

The existence of two players with opposite goals makes the searching problem adversarial.

Definition 5.1 (Game)**.**
A game is defined by 7 components:

1. Initial State s_0
2. States
3. Players: PLAYER(s) defines which player has the move in state s.
4. Actions: ACTIONS(s) returns set of legal moves in state s
5. Transition model: RESULT(s, a) returns state that results from the move a in state s.
6. Terminal test TERMINAL(s) returns `true` if game is over and `false` otherise
7. Utility function UTILITY(s, p) gives final numeric value for a game that ends in terminal state s for a player p

Definition 5.2 (Constant-sum Game)**.**
A game is a constant-sum game if total utility score over all agents sum to constant.

Remark: Player strategy still needs to specify behaviour in states that will never be reached.

Definition 5.3 (Winning/Non-losing Strategy)**.**
A strategy s_1^* for player 1 is called **winning** if for any strategy s_2 by player 2, the game ends with player 1 as the winner.
A strategy t_1^* for player 1 is called **non-losing** if for any strategy s_2 by player 2, the game ends in either a tie or a win for player 1.

Theorem 5.1.
In the game of chess, only one of the following is true:

1. White has a winning strategy s_W^*
2. Black has a winning strategy s_B^*
3. Each player has a non-losing strategy

5.2 Minimax

Definition 5.4 (Minimax)**.**
The optimal strategy at every node for the two player can be derived using the minimax algorithm, as it achieves subperfect nash equilibrium.

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal}(s) \\ \max_{\alpha \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \text{MAX} \\ \min_{\alpha \in \textsc{Actions}(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \text{MIN} \end{cases}$$

Complete?	Yes(if game tree is finite)
Optimal?	Yes
Time	$O(b^m)$
Space	$O(bm)$

5.3 $\alpha - \beta$ Pruning

The game tree of Minimax algorithm is very huge in practice and $\alpha - \beta$ pruning is required to optmise the search.

Definition 5.5 (α, β)**.**
For each MAX node n, $\alpha(n)$ is the highest observed value found on all paths from n; initially $\alpha(n) = -\infty$.
For each MIN node n, $\beta(n)$ is the lowest observed value found on all paths from n; initially $\beta(n) = +\infty$.

Theorem 5.2 (Alpha/Beta Prune)**.**
Alpha prune: Given a MIN node n, stop searching below n if there is some MAX ancestor i of n with $\alpha(i) \geq \beta(n)$.
Beta prune: Given a MAX node n, stop searching below n if there is some MIN ancestor i of n with $\beta(i) \leq \alpha(n)$.

Remark:

- When we prune a branch, it **never** affects final outcome.

- Good move ordering improves effectiveness of pruning
- "Perfect" ordering improves time complexity to $O(b^{\frac{m}{2}})$.
- Random ordering improves time complexity to $O(b^{\frac{3}{4}m})$ for $b < 1000$.

5.4 Evaluation Function

$\alpha - \beta$ pruning does not resolve the problem of the large depth of the game tree.
Standard solution to the problem is

- Evaluation function, which estimates the expected utility of the state
- Cutoff test, which limits the depth of the traversal

Definition 5.6 (Heuristic Minimax Value)**.**
Heuristic Minimax algorithm makes uses of the evaluation function and cutoff test.

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{\alpha \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{\alpha \in \text{ACTIONS}(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

The above algorithm will run MINIMAX until depth d, and use evaluation function for nodes of depth greater than d.

Definition 5.7 (Evaluation Function)**.**
An evaluation function is a mapping from game states to real values $f : S \to \mathbb{R}$.

The evaluation function should observe the following properties:

- Cheap to compute
- For non-terminal states, it must be strongly **correlated** with actual chances of winning

It would be good if the evaluation function maintains the relative ordering of game states.

Definition 5.8 (Modern Evaluation Functions)**.**
Modern evaluation functions are usually in the form of weighted sum of position features:

$$f(n) = \sum_{i \in I} w_i \text{feat}_i(n)$$

The weights are usually determined dynamically.

Remark: Cutting off search can be combined with iterative deepening.

5.5 Stochastic Games

Stochastic games are games involved with uncertainty. Minimax needs to add a chance layer for the calculated of the **expected** value of a state.

6 Constraint Satisfaction Problems

6.1 Constraint Satisfaction Problems

Previously, we have covered standrad search problem where state is atomic.

Definition 6.1 (Constraint Satisfaction Problem)**.**
A constraint satisfaction problem consists of the following components:

- Variables $\mathbf{X} = (X_1, \ldots, X_n)$; each variable X_i has a domain D_i.
- Constraints $\mathbf{C}$, which involves
  - Constraint scope: which variables are involved
  - Constraint relation: what is the relation between them
- Objective: find a legal assignment $(y_1, \ldots, y_n)$ of values to variables $(X_1, \ldots, X_n)$, such that
  - $y_i \in D_i$ for all $i \in [n]$.
  - Constraints are all satisfied.

Constraint satisfaction problem(CSP) is a broad modelling framework. Examples of CSP include graph coloring and job-shop scheduling.
Different aspects of CSP include

- Discrete variables of finite domains and infinite domains **vs** continuous variables
- Unary constraints **vs** Binary constraints **vs** Global(i.e. higher-oredr) constraints

CSP can be solved using standard search formulation:

- Each state is a partial variable assignment
- **Initial State**: the empty assignment []
- **Transition Function**: assign a *valid* value to an unassigned variable; fail if no *valid* assignments
- **Goal Test**: all variables have been assigned.

One very useful properties of CSP is that *every* solution appears at depth n, where n variables have been assigned. Therefore, the best search technique is DFS.

Definition 6.2 (Backtracking Search)**.**
DFS for CSPs with single-variable assignment is called **backtracking search**. It performs backtracking when there is no legal assignments before reaching any goal.
function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
  return BACKTRACK({}, *csp*)

```
function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var←SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences←INFERENCE(csp, var, value)
      if inferences≠failure then
        add inferences to assignment
        result←BACKTRACK(assignment, csp)
        if result≠failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

6.2 General Purpose Heuristics

To improve backtracking efficiency, we utilise general-purpose heuristics.

6.2.1 Most Constrained Variable

This strategy, a.k.a. minimum-remaining-values(MRV) heuristic, chooses the variable with *fewest legal* values.

6.2.2 Most Constraining Variable

This strategy, a.k.a. degree heuristic, chooses the variable with most constraints on remaining unassigned variables.

6.2.3 Least Constraining Value

This strategy, given a variable, chooses the least constraining value. Equivalently, it chooses te one value that rules out the fewest values for the neighbouring unassigned variables.

6.3 Forward Checking

Forward checking works by

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Despite forward checking propogates information from assigned to unassigned variables, it does not necessarily provide early detections for all failures.

6.4 Constraint Propagation

Constraint propagation improves on forward checking, by repeatedly locally enforces constraints.
Constraint propagation generate inferences in CSPs by looking at

- Node consistency for unary constraints
- Arc consistency for binary constraints
- Other constraints

Constraint propagation can be either interleaved with search, or run as a preprocessing step.

6.4.1 Arc Consistency

Simplest form of propagation will make each arc consistent.

Definition 6.3 (Arc Consistent)**.**
X_i is arc-consistent with respect to X_j if and only if for every value $x \in D_i$, there exists some value $y \in D_j$ that satisfies the binary constraint on the arc (X_i, X_j).

The propagation by arc consistency works by looking at the updated codomain, and trims the domain until no elements in teh domain maps outside the updated codomain.
This will usually cause a chain reaction, and it detects failure earlier than forward checking.

Theorem 6.1 (Arc Consistency Algorithm AC-3)**.**
function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBOURS$-\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*
function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised$\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised$\leftarrow$ *true*
  return *revised*

The above algorithm gives time complexity of $O(n^2d^3)$, as

- CSP has at most n^2 directed arcs

- each arc can be inserted at most d times, as X_i has at most d values to delete
- Checking the consistency of an arc takes $O(d^2)$ times.

To maintain Arc Consistency, we can proceed the search as follows:

- Establish AC at the root
- When AC-3 terminates, choose a new variable
- re-establish AC given the new variable choice(Maintain AC)
- repeat;;
- backtrack if AC gives *empty* domain

6.4.2 k-consistency

Arc consistency on binary constraints can be generalised to k consistency on global constraints. However, k consistency need to consider not only pairwise constraints, but also constraints of more variables.

6.5 Local Search for CSPs

Hill-climbing, simulated annealing can be modified to be applied to CSP, by

- allow states that violate constraints
- operators reassign variable values

The the variable value selection, we apply the **min-conflict** heuristic, which chooses value that violates the fewest constraints.

Theorem 6.2 (Min-Conflict Algorithm)**.**
function MIN-CONFLICT(*csp*, *max_steps*) **returns** a solution or failure
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current← an initial complete assignment for *csp*
  for $i = 1$ to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var← a randomly chosen conflicted variable from *csp*.VARIABLES
    value← the value v for *var* that minimises CONFLICTS(*var*, *v*, *current*, *csp*)
    set *var*=*value* in **current**
  return *failure*

7 Logical Agents

7.1 Knowledge-based Agents

Knowledge-based agents represent agent domain knowledge using logical formulas.

- The central component of a knowledge-based agent is its **knowledge base** (KB), which is a set of sentences.
- Each sentence is expressed in a **knowledge representation language**.
- Declarative approach to build an agent by TELL it what it needs to know
- Then it infers by ASKING itself what to do, from knowledge in the KB
- The agent must be able to
  - Represent states, actions, etc.
  - Incorporate new percepts
  - Update internal world representations
  - Deduce hidden world properties, and deduce actions

The following algorithm is one general algorithm for Knowledge Based Agents:
function KB-AGENT(*percept*) **returns** an *action*
  persistent: *KB*, knowledge base
    t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, t))
  action$\leftarrow$ ASK(*KB*, MAKE-ACTION-QUERY(t))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, t))
  $t \leftarrow t + 1$
  return *action*

7.2 Logic in Genral: Models and Entailment

Following are some terms used in logic:

- **Logic**: formal language for KB, infer conclusions
- **Syntax**: defines the sentences in the language
- **Semantics**: define the "meaning" of sentences

Definition 7.1 (Modelling)**.**
m models α if α is true under m.
Usually, we let $M(\alpha)$ denote the set of all models for α.

Definition 7.2 (Entailment)**.**
Entailment means that one thing **follows from** another. Formally,

$$\alpha \models \beta \Leftrightarrow M(\alpha) \subseteq M(\beta)$$

Definition 7.3 (Inference)**.**
We use $KB \vdash_i \alpha$ to mean "sentence α is derived from KB by inference algorithm i".

- **Soundness**: i is sound if $KB \vdash_i \alpha$ implies $KB \models \alpha$.
- **Completeness**: i is complete if $KB \models \alpha$ implies $KB \vdash_i \alpha$.

Inference is done by deriving knowledge out of percepts. Specifically, given KB and α, we want to know if $KB \vdash \alpha$.

7.3 Propositional Logic

7.3.1 Syntax

Sentence in propositional logic are represented by symbols, like S_i. Logical connectives are used to construct complex sentences from simpler ones:

- If S is a sentence, $\neg S$ is a sentence.
- If S_1, S_2 are sentences,
  - $S_1 \wedge S_2$ is a sentence.
  - $S_1 \vee S_2$ is a sentence.
  - $S_1 \Rightarrow S_2$ is a sentence.
  - $S_1 \Leftrightarrow S_2$ is a sentence.

7.3.2 Semantics

A model is a **truth assignment** to the basic variables. All other sentences' truth value is derived according to logical rules.

7.3.3 Inference by Truth Table Enumeration

We adopt depth-first enumeration of all models. This guarantees soundness and completeness.
For n symbols, time complexity is $O(2^n)$ and space complexity is $O(n)$.
Below is the algorithm:
function TT-Entails?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence is propositional logic
    α, the query, a sentence in propositional logic

symbols← a list of the proposition symbols in KB and α **return** TT-Check-All(KB, α, *symbols*, {})

function TT-Check-All((KB, α, *symbols*, *model*) **returns** *true* or *false*
  if Empty?(*symbols*) **then**
    if PL-True?(KB, *model*) **then return** PL-True?(α, *model*)
    else return *true*
  else do
    P ←First(*symbols*)
    rest←Rest(*symbols*)
    return (TT-Check-All(KB, α, *rest*, *model*$\cup\{P = true\}$)
        and
        TT-Check-All(KB, α, *rest*, *model*$\cup\{P = false\}$)

7.4 Equivalence, Validity, Satisfiability

Definition 7.4 (Validity)**.**
A sentence is **valid** if it is true in *all* models.

Theorem 7.1 (Deduction Theorem)**.**
Validity is connected to entailment through deduction theorem:

$$KB \models \alpha \Leftrightarrow (KB \Rightarrow \alpha) \text{ is valid}$$

Definition 7.5 (Satisfiability)**.**
A sentence is satisfiable if it is true in **some** model.
A sentencxe is unsatisfiable if it is true in *no* models.

Theorem 7.2 (Satisfiability and entailment)**.**
Satisfiability is connected to entailment via the following:

$$KB \models \alpha \Leftrightarrow (KB \wedge \neg\alpha) \text{ is unsatisfiable}$$

7.5 Proof Methods

There are two ways of proving an proposition, either by applying inference rules, or by model checking.
Examples of inference rules include

- And elimination: $a \wedge b \models a$.
- Modus Ponens: $a \wedge (a \Rightarrow b) \models b$.
- Logical Equivalence: $(a \vee b) \models \neg(\neg a \wedge \neg b)$.

7.5.1 Resolution for Conjunctive Normal Form

Suppose we have a conjunction of disjunction of literals(clauses), and if a literal x appeas in C_1 and its negation $\neg x$ in C_2, it can be resolved as:

$$\frac{(x_1 \vee \cdots \vee x_m \vee x) \wedge (y_1 \vee \cdots \vee y_n \vee \neg x)}{(x_1 \vee \cdots \vee x_m \vee y_1 \vee \cdots \vee y_n)}$$

Resolution is sound and complete for propositional logic.