Artificial Intelligence

1 What is an AI?

There are various definitions of an AI, ranging from thinking humanly and rationally to acting humanly and rationally. The *turing test*, is test in which a human interrogator interacts with a machine, sending it messages back and forth, and a machine passes if it fools the human into thinking that the messages are being sent to them by a human. For this a computer needs: **natural language processing, knowledge representation, automated reasoning and machine learning**. To pass the *total turing test* a computer would additionally need **computer vision and robotics.**

1.1 Intelligent Agents

An **agent** is just something that acts and a **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. **Percept** means the agent's perceptual inputs at any given time, and a **percept sequence** is the complete history of everything the agent has ever perceived. The **agent function** is an abstract mathematical description that maps a given percept to an action; an **agent program** is a concrete implementation of the agent function, running within some physical system. *It is better to design a performance measure according to what one wants in an environment, then how one wants an agent to behave.*

The proper definition of a rational agent is *for each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.* An **omniscient** agent knows the actual outcomes of its actions.

1.2 The Nature of Environments

An environment is defined as **PEAS**: performance measure, environment, actuators and sensors. There are different types of environments, namely:

- **Observable vs Partially-Observable**: it is observable when the agents' sensors have complete access to the environment's state at all times

- **Single agent vs Multi agent**: there could be multiple agents in an environment. There is also a question of what must be considered an agent. This gives way to the concept of **competitive** vs **cooperative** environments.

- **Deterministic vs Stochastic**: If the next state can be completely determined by the current state and the action executed by the agent, then it is deterministic; and stochastic otherwise. An environment is **uncertain** if it is not fully observable or not deterministic. Note that a **nondeterministic** environment is one where each action is characterized by its possible outcomes, but no probabilities are attached to them.

- **Episodic vs Sequential**: In an episodic environment the agent's experience is divided into atomic episodes. The next episode doesn't depend on the action taken in the previous episode.

- **Static vs Dynamic**: If an environment can change when an agent is deliberating, then it's dynamic, and is static otherwise. If the environment doesn't change when deliberating but the performance score does, then we call it **semi-dynamic.**

- **Discrete vs Continuous**: The distinction here applies to the state of the environment, the way time is handled and the percepts and actions of the agent. For example, chess having a discrete set of states; the same doesn't apply for taxi driving.

- **Known vs Unknown**: This applies to the agent's state of knowledge about the "laws of physics" of the environment. Note that it's possible that a known environment is partially observable like solitaire. Conversely, an environment can also be unknown and fully observable, like in a video game, one can see the state but one doesn't know the control until one tries to play.

1.3 The Structure of Agents

There are four basic types of agent programs:

- **Simple Reflex Agents**: Agents that select the current action based on the current precepts and ignoring the rest of the precept history. It is also important to note that these types of agents are usually implemented in a fully-observable environment.

- **Model-Based Reflex Agents**: The best way to handle a partially observable environment is to keep some sort of an internal representation of the aspects of the environment not currently observable. Therefore, an agent should have some sort of knowledge about how the world works and the agents who have said knowledge are called model-based agents.

- **Goal-Based Agents**: These types of agents consider how close they get to a goal, in addition to have a model of how their environment works. These types of agents are also quite flexible as they can update their actions on-the-fly depending on their goals and the feedback they get from the environment.

- **Utility-Based Agents**: Since the previous model does not differentiate between how it gets to its goal, and which state would make it more happy, it is not efficient. Therefore a utility function is needed to determine just that i.e. it is an internalization of its performance measure. The previous model will also fail when there are conflicting goals or when there are several goals the agent can aim for, none of which can be achieved with certainty; in both cases, a utility function can dictate which action to take to maximize expected utility.

There are different ways an agent can represent the world around it:

- **Atomic**: Each state is of the world is indivisible: it has no internal structure.

- **Factored**: Each state is split up into a fixed set of variables and attributes, each of which can have a value. Uncertainty can also be represented in this representation.

- **Structured**: This type of representation has objects and their relationship with each other.

2 Problem Solving by Searching

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. **Problem formulation** is the process of deciding what actions and states to consider, given a goal. In general, *an agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.* The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase. Note while the agent is executing, it *ignores its percepts* when choosing its actions because it knows in advance what they will be.

Together, the **initial state, actions and transition model** implicitly define the **state space** of the problem - the set of all states reachable from the initial state by any sequence of actions. A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest

path cost among all solutions. The process of removing detail from a representation is called **abstraction**.

It is quite important to distinguish between a node and a state: a node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world. Furthermore, two different states can contain the same world state if that state is generated via two different search paths. We can evaluate the performance of an algorithm in four ways:

- **Completeness**: Is the algorithms guaranteed to find a solution if there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

In AI, the graph is often represented implicitly by the initial state, actions and transition model and is frequently infinite. Complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node; d, the **depth** of the shallowest goal node; and m, the maximum length of any path in the state space.

2.1 Uninformed Search Strategies

2.1.1 Breadth-First Search

This is a simple strategy in which all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The algorithm is given in figure 1.

BFS is *complete* - if the shallowest goal node is at some finite depth BFS will eventually find it after generating all the shallower nodes. Note that the *shallowest* is not always the *optimal* one. BFS is only optimal if the path cost is a non-decreasing function of the depth of the node. The time complexity of BFS is:

$$b + b^2 + b^3 + ... + b^d = O(b^d) \tag{1}$$

For the space complexity there will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, giving space complexity as $O(b^d)$. *Memory requirements are a bigger problem for BFS than execution time and time is still a major factor. Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

2.1.2 Uniform-Cost Search

This is similar to BFS, but it expands the node with the lowest path cost, the goal test is applied to a node *not when it's generated, but when it's chosen for expansion* and a test is added in case a better path is found to a node currently

```
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)
```

Figure 1: Breadth-First Search.

on the frontier. The algorithm is given in figure 2. First, we note that whenever this algorithm selects a node for expansion, the optimal path to that node is found and second, paths never get shorter as nodes are added. Thus, *uniform-cost search expands nodes in order of their optimal path cost.* This search is complete given that the cost of every step exceeds some small positive constant ϵ. The space and time complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the optimal solution.

2.1.3 Depth-First Search

Depth-First Search always expands upon the deepest node in the frontier, this is accomplished using a LIFO queue. The graph-search version of the algorithm is complete in finite spaces but the tree-search version could potentially fall into a infinite loop. Although, it must be noted that both versions fail if there is an infinite state space, with an infinite non-goal path e.g. Knuth's 4 problem. Both versions are also not optimal. The time complexity of DFS is $O(b^m)$ and space complexity is $O(bm)$.

2.1.4 Depth-Limited Search

The problem of DFS failing in infinite spaces can be solved if we apply a limit ℓ to the depth we expand up till. Although, it introduces incompleteness if we choose $\ell < d$ and non-optimality if $\ell > d$. Its time complexity is $O(b^\ell)$ and its space complexity is $O(b\ell)$. Its pseudocode is shown in figure 4.

2.1.5 Iterative Deepening Search

The algorithm of IDS is shown in figure 5. Its space complexity is $O(bd)$. Like BFS, IDS is complete if the branching factor is finite and it is optimal when the

```
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
```

Figure 2: Uniform-Cost Search.

A recursive implementation of DFS:[5]

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

A non-recursive implementation of DFS:[6]

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Figure 3: Depth-First Search.

```
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
```

Figure 4: Depth-Limited Search.

```
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
```

Figure 5: Iterative Deepening Search.

path cost is a non-decreasing function of the depth. The time complexity is:

$$(d)b + (d-1)b^2 + ... + (d-d+1)b^d = O(b^d) \tag{2}$$

In general, IDS is preferred if the search space is large and the depth of the solution is not known.

2.1.6 Bidirectional Search

Bidirectional search applies the idea of starting two searches: one from the initial state and one from the goal state, hoping that they meet in the middle; with the motivation that $b^{d/2} + b^{d/2} < b^d$. Thus the space and time complexity for this algorithm is $O(b^{d/2})$.

2.1.7 Comparison of Uninformed Search Strategies

Figure 6 shows comparisons for tree search versions of the algorithms discussed. For graph searches, the main difference is that DFS is complete for finite state spaces and that the space and the time complexities are bounded by size of the state space.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 6: Comparison of uninformed search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; b complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

3 Informed Search Algorithms

An informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. The general approach considered is an instance of a general graph or tree search in which a node is chosen for expansion based on its **evaluation function**, $f(n)$. Most best-first algorithms include as a component of f a **heuristic function**, denoted by $h(n)$, which is an estimated cost of the cheapest path from the state at node n to a goal state. We use the constraint that if n is the goal node then $h(n) = 0$.

3.1 Greedy Best-First Search

This algorithm uses $f(n) = h(n)$. Due to its greedy nature it is not optimal and is also incomplete even in a finite state space, like DFS. The graph version of this algorithm is complete in finite spaces, but not infinite ones. The worse case space and time complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search tree.

3.2 A* Search

This search evaluates nodes by using:

$$f(n) = g(n) + h(n) \tag{3}$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, $f(n)$ is the estimated cost of the cheapest solution through n. Provided $h(n)$ satisfies certain conditions, it's both complete and optimal.

3.2.1 Conditions for Optimality: Admissibility and Consistency

The first condition we require is $h(n)$ be an **admissible heuristic**, meaning it is one that *never overestimates* the cost to reach the goal. Thus we have

that $f(n)$ never overestimates the true cost of a solution along the current path through n. The second condition required is **consistency**, or **monotonicity** for applications of A* to graph search. A heuristic $h(n)$ is consistent if:

$$h(n) \leq c(n, a, n') + h(n') \tag{4}$$

Where n' is the successor of n generated by action a. This is a form of the general **triangle inequality**, which says that each side of a triangle can't be longer than the sum of the other two sides. Here the triangle is formed by n, n' and the goal G_n closest to n.

3.2.2 Optimality of A*

We know that *the tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent.* First let us establish: *if $h(n)$ is consistent then the values of $f(n)$ along any path are non-decreasing.* Suppose n' is a successor of n; then $g(n') = g(n) + c(n, a, n')$ thus we have:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) \tag{5}$$

The next step would be to prove that *whenever A* selects a node for expansion, the optimal path to that node has been found.* The fact that f costs are non-decreasing along any path allows us to draw **contours** in the state space. If C^* is the cost of the optimal solution path, then:

- A* first expands all nodes with $f(n) < C^*$.
- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting the goal node.

Completeness requires that there be only finitely many nodes wih cost less than or equal to C^*, a condition only met if all step costs exceed some finite ϵ and if b is finite. One final observation is A* is **optimally efficient** for any given consistent heuristic i.e. no other optimal algorithm is guaranteed to expand fewer nodes than A* because any algorithm that doesn't expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution. The space complexity is $O(b^d)$ and the time complexity is $O(b^{\Delta})$, with $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal; and this is the formula for **absolute error**. For constant step costs we have $O(b^{\epsilon d})$, with $\epsilon \equiv (h^* - h)/h^*$ which is the **relative error**.

3.3 Heuristic Functions

There are a few good heuristics used for the 8-puzzle like the number of misplaced tiles and the **manhattan distance** which is the sum of horizontal and vertical distances. One way to characterize the quality of a heuristic is the **effective branching factor**, b^*, which is conventionally defined as an average number of nodes revisited of the current iteration N as compared to the previous

iteration $N-1$. 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 in order to contain $N+1$ nodes. Thus:

$$N+1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d \tag{6}$$

If $\forall$ nodes $n, h_2(n) \geq h_1(n) \implies h_2$ **dominates** h_1.

3.3.1 Relaxed Problems

A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph. Because edges are added to the state-space, any optimal solution in the original problem is also a solution in the relaxed problem, but the relaxed problems may have better solutions. Hence, *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.* Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it muse obey the triangle inequality and is therefore consistent. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to compute. Since a collection of admissible heuristics are obtained, we can use an appropriate heuristic on each node individually using:

$$h(n) = \max\{h_1(n), ..., h_m(n)\} \tag{7}$$

3.4 Local Search Algorithms

These algorithms operate using a single **current node** and generally move only to neighbors of that node and the paths followed aren't retained. Thus they have the advantages:

- They use very little memory - usually a constant amount
- They can often find reasonable solutions in large or infinite state spaces

3.4.1 Hill-Climbing Search

The algorithm is shown in figure 7. It is comparative to "trying to find the top of Mount Everest in a thick fog while suffering from amnesia". It is also called **greedy local search**. Hill-Climbing gets stuck in **local maxima/minima, ridges** and **plateaux**.

4 Adversarial Search

A **utility function** is defined as the final numeric value for a game that ends in terminal state s for a player p. A **zero-sum game** is defined as one where the total payoff to every player is the same in every instance of the game.

```
function HILL-CLIMBING( problem) returns a solution state
   inputs: problem, a problem
   local variables: current, a node
                    next, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      next ← a highest-valued successor of current
      if VALUE[next] < VALUE[current] then return current
      current ← next
   end
```

Figure 7: Hill-Climbing Search.

```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

Figure 8: The Minimax Algorithm.

4.1 The Minimax Algorithm

The algorithm is shown in figure 8. The utility values of the leaf nodes are computed and the corresponding minimax values are backed up the tree recursively. The time complexity of this algorithm is $O(b^m)$, where m is the maximum depth of the tree and there are b legal moves at each point. The space complexity is $O(bm)$ for an implementation that generates all the actions at once and $O(m)$ for an implementation that generates actions one at a time.

4.2 Alpha-Beta Pruning

A way to improve the abysmal complexity of Minimax is to use alpha-beta pruning, which prunes away branches that can't possibly influence the final decision. α is the best choice we have found so far for Max and β is the best

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 9: Alpha-Beta Pruning.

choice for Min. The algorithm is applied in figure 9. **Move ordering** can be applied to improve complexity to $O(b^{m/2})$ and the branching factor becomes $\sqrt{b}$. If the successors are examined at random and not best-first the complexity increases to $O(b^{3m/4})$. The best moves in a game are called **killer moves** and a killer move heuristic is used to find them. The hash table of previously seen moves is called a **transposition table**.

4.3 Evaluation Functions

An evaluation function returns an estimate of the expected utility from a given position. This is useful as sometimes we want to stop our search at a depth limit and evaluate the leaf nodes using this function. Most evaluation functions compute separate numerical contributions from each feature then combine them to find the total value. These types of functions are also called **weighted linear functions** because:

$$E(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{8}$$

In **cutting off search** `isTerminal` and `utility` in minimax are replaced by `cutoff` and `eval`. In stochastic games, chance nodes are added in addition to min/max nodes. The minimax values are also replaced by **expected values**: the average over all possible outcomes of the chance nodes. This leads us to make **expecti-minimax** for games with chance nodes.

5 Machine Learning in Game Search

We can automate the fine design choices in an evaluation function by using machine learning. We can use **book learning** to learn a sequence of moves for important positions, like remembering what opening moves lead to what outcomes and remembering what moves led to a loss and avoiding them in the future. **Search control learning** is used to learn how to make search more efficient like order of move generation for $\alpha - \beta$ and learning a classifier to predict what depth we should search depending on the current state. Weights in evaluation functions can also be learned to make them agree with the true final utility.

5.1 Gradient Descent Learning

This is a a type of *supervised learning* which is like the Hill-Climbing algorithm, in that it starts at some point **w** in the weight space and moves to a neighboring lower point until we converge to the minimum possible loss. The hope that weights can be learned that closely approximate the true output. Each weight is updated according to:

$$w_i \leftarrow w_i - \alpha(z - t)f_i(s) \tag{9}$$

α is the **learning rate** which can be a constant or can decay over time. The problems with this type of learning are:

- Delayed reinforcement: reward maybe received after a few steps of applying the move
- Credit assignment: need to know which actions were responsible for the outcome

5.2 Temporal Difference Learning

Since supervised learning is for single-step prediction, TD is for multi-step prediction. The correctness of the prediction is not known until a few steps later, but intermediate steps can provide information about correctness. This is a type of supervised learning. **TDLeaf**(λ) combines TD and minimax to update evaluation function to reduce differences in rewards predicted at different levels of the search. The weight update rule applied is:

$$w_j \leftarrow w_j + \alpha \sum_{i=1}^{N-1} \frac{\partial r(s_i^l, w)}{\partial w_j} \sum_{m=1}^{N-1} \lambda^{m-i} d_m \tag{10}$$

Where $r(s_i^l, w)$ is the reward for of a state (which is found at max cut-off depth using minimax starting at s_i) using the weight w. And d_i is defined as difference between successive states:

$$d_i = r(s_{i+1}^l, w) - r(r_i^l, w) \tag{11}$$

6 Constraint Satisfaction Problems

We use a **factored representation** here for each state and when each value from the vector of variables of the states satisfies certain constraints, we say a solution has been found. Problems such as this are called **constraint satisfaction problems**. These problems are defined by:

- X is a set of variables $\{X_1, ..., X_n\}$.
- D is a set of domains $\{D_1, ..., D_n\}$, one for each variable. Each D_i contains a set of allowable values, $\{v_i, ..., v_n\}$ for X_i.
- C a set of constraints that specify allowable combination of values. Each C_i consists of a pair $\langle scope, rel \rangle$ where *scope* is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

A CSP is solved by **assignment** of variables and if no constraints are violated, the assignment is **consistent**, if it assigns a value to each variable then it is **complete** and **partial** otherwise. Solving problems this way is useful as it eliminates large areas from the state space. Domains can range from **continuous** to **discrete** and from **finite** to **infinite**. Continuous domain problems can be solved using **linear programming**. A **unary constraint** restricts the value of one variable and a **binary** one restricts two variables. A binary CSP (one with only binary constraints) can be represented by a constraint graph. Confusingly, a constraint which involves an arbitrary number of variables is called a **global constraint**. An example is $Alldiff$ which says that every variable involved in the constraint must have a different value.

Cryptarithmetic puzzles, which are essentially arithmetic operations on number represented in their alphabetic form, can also be solved using CSPs. A **constraint hypergraph** can be used to represent n-ary constraints, using hypernodes (drawn as squares). Every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced. Another way to convert is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and one binary constraint for each pair fo constraints in the original graph that share variables. Global constraint are preferred because they are less error-prone and allow us to design special-purpose inference algorithms not available for a set of more primitive constraints. There are also **preference constraints** indicating which solutions are preferred, which can be realized by penalizing lesser-preferred solutions; such problems are called **constraint optimization problems**.

6.1 Constraint Propagation

In regular state space an algorithm can only search, but with CSPs, in addition to search, we can do an inference called **constraint propagation**: using con-

```
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) ← 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.NEIGHBORS - {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 ← 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 ← true
  return revised
```

Figure 10: The Arc-Consistency Algorithm AC-3.

straints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable.

6.1.1 Node Consistency

A single variable (a node in the CSP network) is node-consistent if all the values in the variable's domain satisfy the variable's unary constraints. A network can be node-consistent if every variable in the network in node-consistent.

6.1.2 Arc Consistency

A variable is arc-consistent if every value in its domain satisfies the variable's binary constraint i.e. X_i is arc-consistent with respect to another variable X_j if:

$$\forall v_i \in D_i, \exists v_j \in D_j \text{ s.t. the binary constraint } (X_i, X_j) \text{ is satisfied} \tag{12}$$

A network is arc-consistent if every variable is arc-consistent with every other variable. The arc-consistency algorithm is shown in figure 10. The time complexity of AC-3 is $O(cd^3)$, where c are the binary constraints (arcs), d is the the maximum domain size of one of n variables and also note that checking consistency of an arc can be done in $O(d^2)$ time.

6.2 Backtracking Search

A problem is **commutative** if the order of application of any given set of actions has no effect on the outcome. We use **backtracking search** for a depth-first

```
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
```

Figure 11: Backtracking search algorithm.

search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 11. The complexity of this algorithm is $O(d^n)$ where n is the number of variables. The algorithm can be improved by thinking about questions like:

- Which variable should be assigned next and what order should its values be tried?
- What inferences should be performed at each step in the search?
- When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

When we choose an unassigned variable in the CSP, we can choose the most constrained one thereby reducing the size of the search tree, doing so is called the **minimum-remaining-values** heuristic. Another technique used is the **degree heuristic** which chooses the variable involved in the largest number of constraints. Once a variable has been selected, the algorithm must choose which of its values to examine first, the **least-constraining-value** heuristic can be used for this. Doing this would leave maximum flexibility for subsequent variable assignments. One of the simplest form of inference is called **forward checking**; whenever a value is chosen for X, the forward checking process establishes arc consistency for it.

Maintaining Arc Consistency, MAC, is used to maintain arc consistency throughout the graph. So after variable X_i is assigned a value, `inference` calls AC-3 but only starting with arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i.

```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 12: The Tree CSP Solver algorithm.

6.3 The Structure of Problems

Independent subproblems are ones which combine to yield the final solution to the problem. Independence can be ascertained by finding the **connected components** of a constraint graph, each problem corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then $\bigcup_i S_i$ is a solution of $\bigcup_i CSP_i$. Solving problems in this way reduces the complexity from $O(d^n)$ to $O(d^c n/c)$. A constraint graph is a tree if any two variables are connected by only one path and *any tree-structured CSP can be solved in linear time with respect in the number of variables.* The key is **directed arc consistency**: every X_i is arc-consistent with each X_j for $j > i$ if and only if a CSP is defined to be DAC under any ordering of variables $X_i, X_2, ..., X_n$.

An ordering of variables such that each variable appears after its parent in the tree is called **topologically sorted**. The tree-CSP-solver is given in figure 12. Now that we have an efficient algorithm for trees, we can think of converting graphs to trees: by either removing nodes or by collapsing nodes together. If a value for a node is set it can considered as not being there at all. The general algorithm is:

- Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S which is called the **cycle cutset**
- For each possible assignment to the variables in S that satisfy all the constrains in S,
  - remove from the domains of the remaining variables any values that are inconsistent with the assignment for S
  - if the remaining CSP is a solution, then return it with the assignment to S

If the cycle cutset size is c then time complexity is $O(d^c \cdot (n-c)d^2)$; we have to try each of the d^c combinations of values for the variables in S, and for each combination we must solve a tree problem of size $n-c$. Finding the *smallest* cycle cutset is NP-hard, but efficient algorithms like **cutset conditioning** can be used, which instantiates, in all ways, a set of variables such that the remaining constraint graph is a tree.

7 Quantifying Uncertainty

The problems with keeping track of a belief state - a representation of the set of all possible world states that it might be in - are:

- The agent must consider every logical possible explanation for the observations, no matter how unlikely, leading to impossible large and complex belief-state representations.
- A correct contingent plan that handles every eventuality can grow arbitrarily large and must consider arbitrarily unlikely contingencies.
- Sometimes there is no plan that is guaranteed to achieve the goal. It must have some way to compare the merits of plans that are not guaranteed.

Decision theory essentially states that an agent is rational if and only if it chooses the action that yields the highest expected utility, averages over all the possible outcomes of the action. The basic axioms of probability theory say that every possible world has a probability between 0 and 1 and that the total probability of the set of possible worlds is 1:

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1 \tag{13}$$

Conditional probabilities are defined by:

$$P(a|b) = \frac{P(a \wedge b)}{P(b)} \tag{14}$$

Note also the inclusion-exclusion principle:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b) \tag{15}$$

The following equations give marginal probability and conditioning respectively:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}, \mathbf{z}) \mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}|\mathbf{z}) P(\mathbf{z}) \tag{16}$$

Let $\mathbf{E}$ be the list fo evidence variables, let $\mathbf{e}$ be the list of observed values for them and let $\mathbf{Y}$ be the remaining unobserved variables and then we have:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \tag{17}$$

Independence can be written as:

$$P(a|b) = P(a) \text{ or } P(a \wedge b) = P(a)P(b) \tag{18}$$

And independence for random variables can be written as:

$$\mathbf{P}(X|Y) = \mathbf{P}(X) \text{ or } \mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) \tag{19}$$

7.1 Bayes' Rule

The **Bayes' rule** is given as:

$$P(b \mid a) = \frac{P(a \mid b)P(b)}{P(a)} \tag{20}$$

The Bayes' rule for multivariate variables is:

$$\mathbf{P}(Y \mid X) = \frac{\mathbf{P}(X \mid Y)\mathbf{P}(Y)}{\mathbf{P}(X)} \tag{21}$$

Gives α is is the normalization constant needed to make entries in $\mathbf{P}(Y|X)$ sum to 1, the Bayes' rule becomes:

$$\mathbf{P}(Y|X) = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y) \tag{22}$$

The **conditional independence** of two variables X and Y, given a third variable Z is:

$$\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z) \tag{23}$$

The following equation illustrates how a single cause directly influences a number of effects, all of which are conditionally independent, given the cause:

$$\mathbf{P}(Cause, Effect_1, ..., Effect_n) = \mathbf{P}(Cause) \prod_i \mathbf{P}(Effect_i|Cause) \tag{24}$$

Such a probability distribution function is called a **naive Bayes** model as it is often used in cases where the "effect" variables are not actually conditionally independent given the cause variables.