AIC-4301C Artificial Intelligence

Yasmina Abdeddaïm

e-mail: yasmina.abddeddaim@esiee.fr
Bureau: 4254
2022 - 2023

Plan

This course

Inspired from:

- Book: Artificial Intelligence: A Modern Approach, Stuart Russell and Peter Norvig
- Artificial Intelligence course, university of Berkeley

Turing Test

- Proposed by Alan Turing (1950), designed to provide a satisfactory operational definition of intelligence.
- A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.

Turing Test

The computer should have the following capabilities:

- **Natural language processing**: to communicate successfully.
- **Knowledge representation**: to store what it knows or hears.
- **Automated reasoning**: to use the stored information to answer questions.
- **Machine learning**: to adapt to new circumstances.
- **Computer vision**: to perceive objects.
- **Robotics**: to manipulate objects and move.

Rational Agent

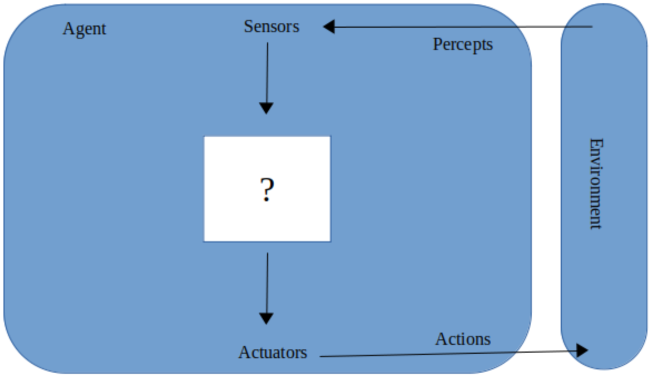
- In artificial intelligence, a central problem is that of the creation of a rational agent.
- **Rational agent** = an entity that has goals or preferences and tries to perform a series of actions that yield the best/optimal expected outcome given these goals.

Rational Agent

- In artificial intelligence, a central problem is that of the creation of a rational agent.
- **Rational agent** = an entity that has goals or preferences and tries to perform a series of actions that yield the best/optimal expected outcome given these goals.

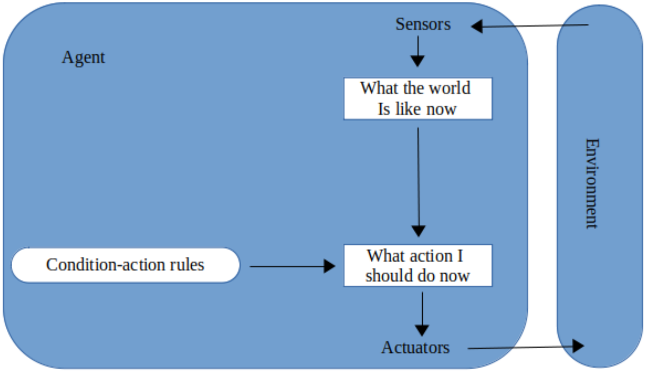
The main challenge of AI is to determine how to write programs that produce a rational behavior using a small program rather than from a large table that stores all the possible actions.

Agent



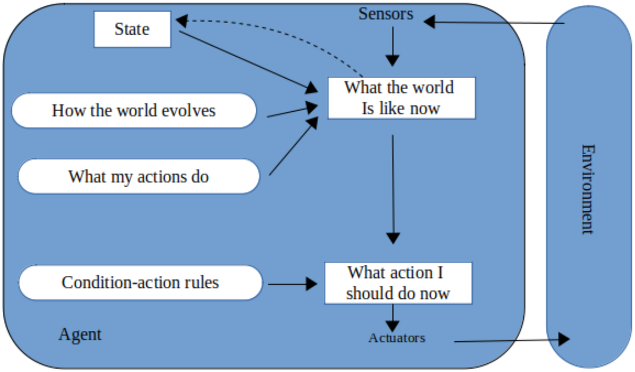
- Agent **perceiving** its **environment** through **sensors** and **acting** upon that environment through **actuators**.
- **Percept**: agent's perceptual inputs at any given instant.

Simple Reflex Agent



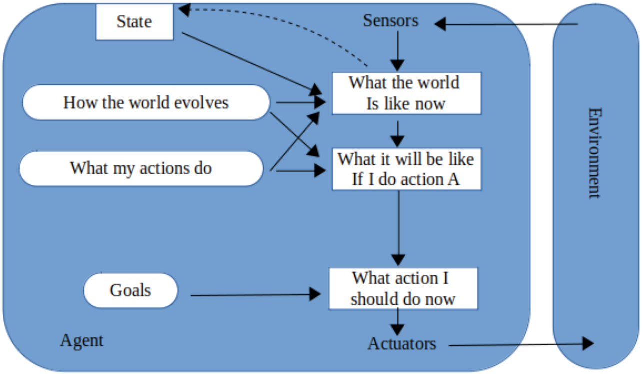
- Select actions on the basis of the current percept, ignoring the rest of the percept history.
- Acts according to a rule whose condition matches the current state, as defined by the percept.

Model Based Reflex Agent



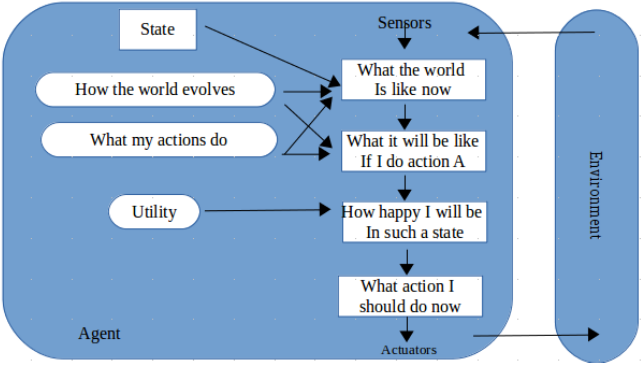
- Keeps track of the current state of the world, using an internal model.
- Chooses an action in the same way as the reflex agent.

Goal Based Agent


State
Sensors
How the world evolves
What the world Is like now
What my actions do
What it will be like If I do action A
Goals
What action I should do now
Agent
Actuators
Environment

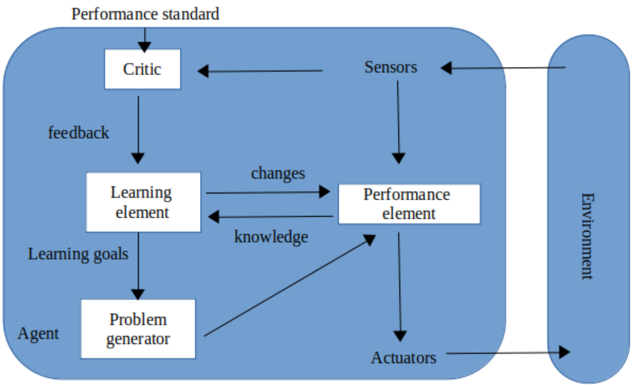
- Keeps track of the world state as well as a set of goals it is trying to achieve.
- Chooses an action that will (eventually) lead to the achievement of its goals.

Utility Based Agent


State
Sensors
What the world Is like now
How the world evolves
What my actions do
What it will be like If I do action A
Utility
How happy I will be In such a state
What action I should do now
Agent
Actuators
Environment

Uses a model of the world and a utility function that measures its preferences among states of the world.

Learning Agent



- Operates in initially unknown environments and becomes more competent than its initial knowledge.
- Learning element: makes improvements of the knowledge.
- Performance element: selects external actions.
- Problem generator: suggests actions that lead to new knowledge.

Plan

Search Problem

- A goal based agent needs a mathematical representation of the environment in which the agent will exist.

Search Problem

- A goal based agent needs a mathematical representation of the environment in which the agent will exist.
- Formally expressed a **search problem**: given an agent's current state how can we reach a new state that satisfies its goals in the best possible way.

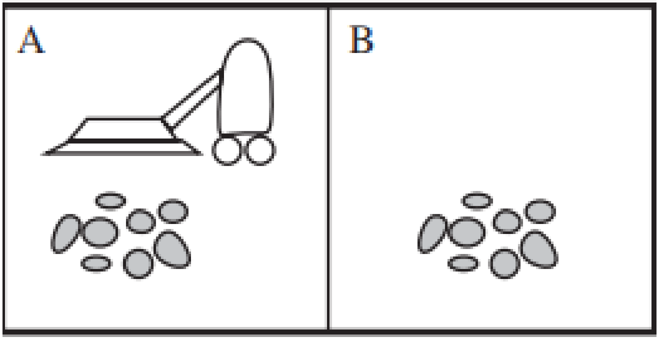
Search Problem

- A goal based agent needs a mathematical representation of the environment in which the agent will exist.
- Formally expressed a **search problem**: given an agent's current state how can we reach a new state that satisfies its goals in the best possible way.
- Formal search problem definition:
  - **State space**: set of all possible states in your given world.
  - **Initial state**: state in which the agent is initially.
  - **Successor function**: function that takes in a state and an action and returns the successors of the state given that action.
  - **Actions**: set of actions that can be executed in a state.
  - **Goal test**: function that takes a state as input and determines whether it is a goal state.
  - **Path cost**: assigns a numeric cost to each path. We assume that the cost of a path can be described as the sum of the costs of the individual actions (**step cost**) along the path.

Search Problem

- **A solution** to a search problem is an action sequence that leads from the initial state to a goal state.
- **An optimal solution** has the lowest path cost among all solutions.

Toy problems: vacuum world

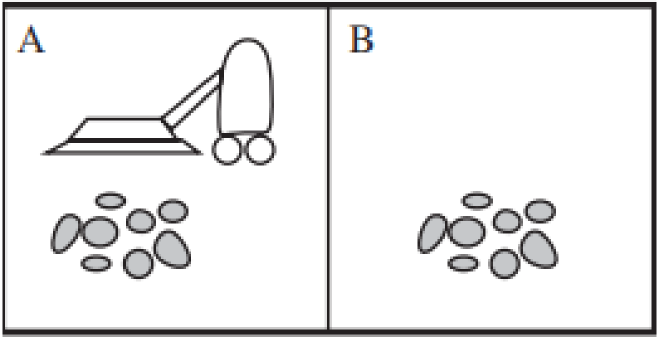


- Two locations: squares A and B.
- The vacuum agent perceives which square it is in and whether there is dirt in the square.
- It can choose to move left, move right, suck up the dirt, or do nothing.

Toy problems: vacuum world

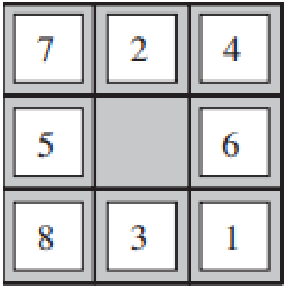
- **State space**: A state is determined by the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt $\Rightarrow 2 \times 2^2 = 8$ possible states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: Left, Right, and Suck.
- **Successor function**: Expected effects of actions except moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- **Goal test**: Checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy problems: vacuum world

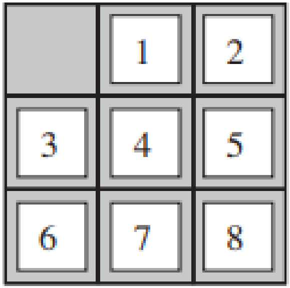


- **Simple goal agent**: if the current square is dirty, then suck; otherwise move to the other square.
- **Solution**: suck, right, suck.

Toy problems: The 8-puzzle



Start State



Goal State

- A 3×3 board with 8 numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space.
- The objective is to reach a specified goal state.

Toy problems: The 8-puzzle

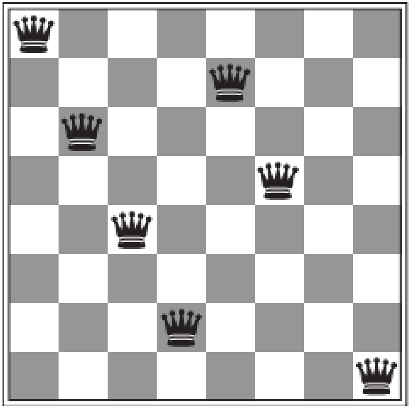
- **State space**: the location of each of the 8 tiles and the blank in one of the 9 squares.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: movements of the blank space Left, Right, Up, or Down.
- **Successor function**: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy problems: The 8-puzzle

- **State space**: the location of each of the 8 tiles and the blank in one of the 9 squares.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: movements of the blank space Left, Right, Up, or Down.
- **Successor function**: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

- The 8-puzzle belongs to the family of sliding-block puzzles (NP-complete), which are often used as test problems for new search algorithms in AI.
- The 8-puzzle has $9!/2 = 181.440$ states, the 15-puzzle has around 1.3 trillion states, the 24-puzzle has around 10^{25} states

Toy problems: 8-queens problem



- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- A queen attacks any piece in the same row, column or diagonal.

Toy problems: 8-queens problem

- **State space**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Successor function**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

In this formulation $A_{64}^{8} = 64!/56! = 64 \times 63 \ldots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate.

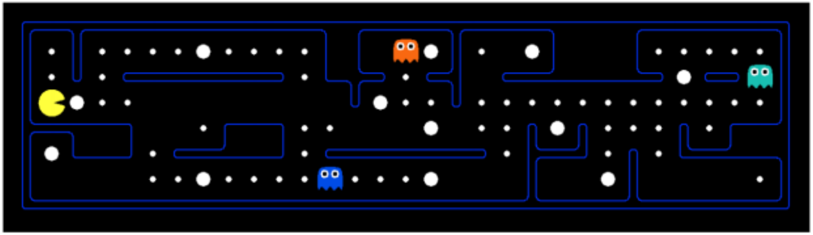
Toy problems: 8-queens problem

Better formulation prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

Reduces the 8-queens state space from 1.8×10^{14} to just $2,057$.

Toy problems: Pacman

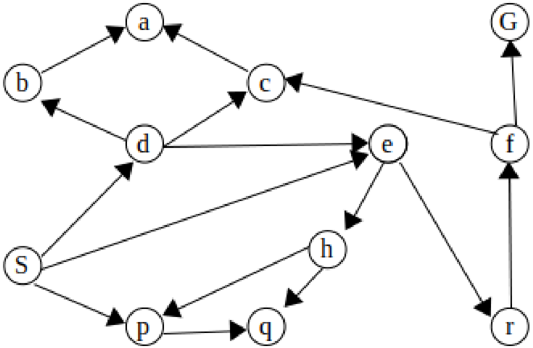


Pathing	Eat-all-dots
-States: (x,y) locations	-States: (x,y) location, dot booleans
-Actions: North, South, East, West	-Actions: North, South, East, West
-Successor: Update location only	-Successor: Update location and booleans
-Goal test: Is (x,y)=END?	-Goal test: Are all dot booleans false?

State Space Graph

- A mathematical representation of a search problem
  - States are (abstracted) world configurations.
  - Arcs represent successors (action results).

State Space Graph

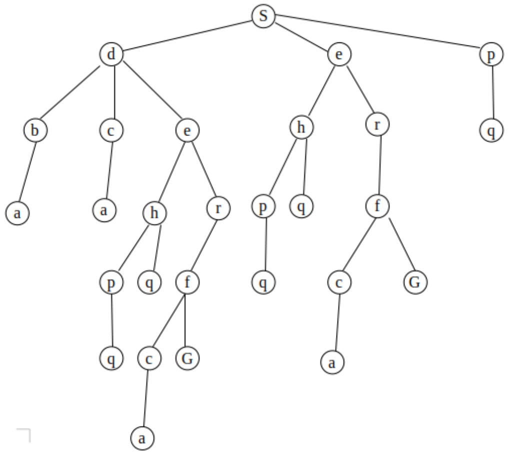


- Each state occurs only once.
- The graph is finite can rarely build this full graph in memory.

Search Tree

- A mathematical representation of a search problem.
  - A node is an entire path in the state space graph.
  - A tree, i.e, no cycle in the graph

Search Tree



- Each state can occur several times.
- The graph is infinite.

Tree Search Algorithm

```
Function TREE-SEARCH(problem) returns a solution, or failure
frontier = initial state of problem
loop
    if the frontier is empty then
        return failure
    end if
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then
        return the corresponding solution
    end if
    expand the chosen node, adding the resulting nodes to the frontier
end loop
```

Graph Search Algorithm

```
1:  Function GRAPH-SEARCH(problem) returns a solution, or failure
2:  frontier = initial state of problem
3:  explored set = empty
4:  loop
5:      if the frontier is empty then
6:          return failure
7:      end if
8:      choose a leaf node and remove it from the frontier
9:      if the node contains a goal then
10:         return the corresponding solution
11:     end if
12:     add the node to the explored set
13:     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
14: end loop
```

Measuring search problem performance

- **Completeness**: Is the algorithm guaranteed to find a solution when 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?

Measuring search problem 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 (less deep) goal.
- **m**: the maximum length of any path in the state space
- In most of the cases we will consider the **worst-case complexity**.

Reminder:

- $O(g(n)) = \{f(n) : \exists c_1 \in \mathbb{N}, \exists n_0 \in \mathbb{N}\ s.t.\ 0 \leq f(n) \leq c_1 g(n), \forall n \geq n_0\}$
- $\Theta(g(n)) = \{f(n) : \exists c_1 \in \mathbb{N}, \exists c_2 \in \mathbb{N}, \exists n_0 \in \mathbb{N}\ s.t.\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

Plan

Uninformed search

- Have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state.

Breadth-first search

- Expands the shallowest nodes first.
- Complete.
- Optimal for unit step costs.
- Has exponential space time and space complexity.

Breadth-first search

```
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)
end if
frontier = a FIFO queue 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 */
    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)
            end if
            frontier = INSERT(child,frontier )
        end if
    end for
end loop
```

Breadth-first Search is Complete

- If a solution exists, then d, the depth of the shallowest goal node must be finite.
- BFS will reach this depth, so it is complete.

Breadth-first Search Optimality

- BFS is generally not optimal because it does not take costs into consideration when determining which node to choose in the frontier.
- BFS is guaranteed to be optimal if the path cost is a non decreasing function of the depth of the node. If all edge costs are equivalent is a special case.

Breadth-first Search Complexity

- **Time complexity**: search $1 + b + b^2 + \ldots b^d$ nodes in the worst case, since we go through all nodes at every depth from 1 to d. Hence, the time complexity is $O(b^d)$.
- **Space complexity**: The frontier in the worst case, contains all the nodes in the level corresponding to the shallowest solution. Since the shallowest solution is located at depth d, the space complexity is $O(b^d)$.

Uniform-cost Search

- Expands the node with lowest path cost g(n).
- Goal test is applied to a node when it is selected for expansion rather than when it is first generate, the first goal node that is generated may be on a suboptimal path.
- A test is added in case a better path is found to a node currently on the frontier.

Uniform-cost Search

```
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
    if EMPTY?(frontier ) then
        return failure
    end if
    node = POP(frontier ) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then
        return SOLUTION(node)
    end if
    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
            end if
        end if
    end for
end loop
```

Uniform-cost Search (UCS) is Complete

- If a goal state exists, it must have some finite length shortest path; hence, UCS must eventually find this shortest length path.
- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ϵ. UCS will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions

Uniform-cost Search Optimality

UCS is optimal if we assume all edge costs are non negative.

- Whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. If this is not the case, there would have to be another frontier node n' on the optimal path from the start node to n, by definition, n' would have lower g-cost than n and would have been selected first.
- Because step costs are non negative, paths never get shorter as nodes are added.
- These two facts imply that uniform-cost search expands nodes in order of their optimal path cost.
- Hence, the first goal node selected for expansion must be the optimal solution.

Uniform-cost Search Complexity

- Let C^* be the cost of the optimal solution.
- Assume that every action costs at least ϵ.
  - **Time complexity**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  - **Space complexity**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  - **Time complexity if all step costs are the same** : $O(b^{1+d})$
  - **Space complexity if all step costs are the same**: $O(b^{1+d})$

Depth-first search (DFS)

- Expands the deepest unexpanded node first.
- Instance of the graph-search algorithm, depth-first search uses a LIFO queue instead of FIFO queue.
- Can be implemented with a recursive function that calls itself on each of its children in turn.

Depth-first search is not complete

- The graph-search version, which avoids repeated states and redundant paths, is complete because it will eventually expand every node.
- The tree-search version is not complete.
- In infinite state spaces, both versions are not complete if an infinite non-goal path is encountered.

Depth-first search is not optimal

Simply finds the "leftmost" solution in the search tree without regard for path costs.

Depth-first search Complexity

- **Time complexity**: In the worst case, depth first search may end up exploring the entire search tree. Hence, given a tree with maximum depth m, the runtime of DFS is $O(b^m)$.
- **Space complexity**: In the worst case, DFS maintains b nodes at each of m depth levels on the frontier. Hence, the space complexity of DFS is $O(bm)$.

Depth-limited Search

- Supplying depth-first search with a predetermined depth limit l in infinite state spaces.
- Nodes at depth l are treated as if they have no successors.
- Additional source of incompleteness if $l < d$
- Time complexity is $O(b^l)$.
- Space complexity is $O(bl)$.
- Depth-first search is a special case of depth-limited search with $l = \infty$.

Depth-limited Search

```
Function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
return RECURSIVE-DLS(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
                end if
            end if
        end for
        if cutoff occurred? then
            return cutoff
        else
            return failure
        end if
    end if
end if
```

Iterative deepening search

- Calls depth-first search with increasing depth limits until a goal is found.
- Combines the benefits of depth-first and breadth-first search.
- Space complexity: $O(bd)$, small like depth-first search.
- Like breadth-first search, it is complete when the branching factor is finite
- Like breadth-first search, optimal when the path cost is a non decreasing function of the depth of the node.
- Time complexity $(d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$ -asymptotically the same as breadth-first search.

```
1: Function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
2: for depth = 0 to $\infty$ do
3:     result = DEPTH-LIMITED-SEARCH(problem, depth)
4:     if result = cutoff then
5:         return result
6:     end if
```

Bidirectional search

- The idea is to run two simultaneous searches, one forward from the initial state and the other backward from the goal.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
- Not always applicable and may require too much space.

Plan

Informed search

- Uses problem-specific knowledge beyond the definition of the problem itself.
- Can find solutions more efficiently than can an uninformed strategy.

Generic Best-first search (BFS)

- Evaluation function $f(n)$: estimated cost of node n.
- An instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$.
- The implementation is identical to that for uniform-cost search except for the use of f instead of g to order the priority queue.

Greedy best-first search

- Heuristic function $h(n)$: estimated cost of the cheapest path from the state at node n to a goal state, i.e. $f(n) = h(n)$
- Always selects the frontier node n with the lowest $h(n)$ value for expansion.
- Greedy search operates identically to UCS.
- Not Complete: Greedy search is not guaranteed to find a goal state if one exists,
- Not optimal: particularly in cases where a very bad heuristic function is selected.

A^* search

- g(n): The function representing the path cost from the start node to node n.
- h(n): The heuristic value function of the estimated cost of the cheapest path from n to the goal.
- f(n): The function representing estimated cost of the cheapest solution through n.

$$f(n) = g(n) + h(n)$$

Admissibility of $h(n)$

- Definition: the value estimated by an **admissible heuristic** $h(n)$ is neither negative nor an overestimate.
- If $h^*(n)$ is the true optimal forward cost to reach a goal state from a given node n, we can formulate the admissibility constraint by $\forall n, 0 \leq h(n) \leq h^*(n)$.
- Admissible heuristics are optimistic because they think the cost of solving the problem is less than it is.
- If $h(n)$ is admissible, because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, $f(n)$ never overestimates the true cost of a solution along the current path through n.

Consistency of $h(n)$

- Definition: a **heuristic** $h(n)$ **is consistent** if for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n : $h(n) \leq c(n, a, n') + h(n')$.

Consistency of $h(n)$

- Definition: a **heuristic** $h(n)$ **is consistent** if for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n : $h(n) \leq c(n, a, n') + h(n')$.
- Required only for applications of A^* to graph search

Consistency of $h(n)$

- Definition: a **heuristic** $h(n)$ **is consistent** if for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n : $h(n) \leq c(n, a, n') + h(n')$.
- Required only for applications of A^* to graph search
- Every consistent heuristic is also admissible

Dominance

Heuristic h is dominant over heuristic h', if $\forall n : h(n) \geq h'(n)$

Optimality

1. The tree-search version of A^* is optimal if $h(n)$ is admissible.
2. The graph-search version of A^* is optimal if $h(n)$ is consistent.

Proof Optimality for Tree-search (1)

- Assume an optimal goal A and a suboptimal goal B.
- Some ancestor n of A (can include A) must currently be on the frontier, since A is reachable from the initial state.
- We claim n will be selected for expansion before B, using 3 statements:
  1. $g(A) < g(B)$. Because A is optimal and B suboptimal, we conclude that A has a lower backwards cost to the start state than B.
  2. $h(A) = h(B) = 0$, because we are given that our heuristic satisfies the admissibility constraint. Since both A and B are both goal states, the true optimal cost to a goal state from A or B is simply $h^*(n) = 0$ hence $0 \leq h(n) \leq 0$
  3. $f(n) \leq f(A)$, because, through admissibility of h, $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(A) = f(A)$. The total cost through node n is at most the true backward cost of A, which is also the total cost of A.
- Statements 1. and 2. imply that $f(A) < f(B)$ as follows: $f(A) = g(A) + h(A) = g(A) < g(B) = g(B) + h(B) = f(B)$
- $f(A) < f(B)$ and statement 3. imply that $f(n) < f(B)$ as follows: $f(n) \leq f(A) \wedge f(A) < f(B)$

Proof Optimality for Tree-search (2)

We proved that $f(n) < f(B)$
Hence, we can conclude that n is expanded before B. We have proven this for arbitrary n, we can conclude that all ancestors of A (including A itself) expand before B.

Proof Optimality for Graph-search (1)

Claim 1: the values of $f(n)$ for nodes along any graph are nondecreasing.

- Let be two nodes, n and n', where n' is a successor of n.
  Then:

  $$f(n') = g(n') + h(n') = g(n) + cost(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

  If for every n, n' along a path, $f(n') \geq f(n)$, then the values of $f(n)$ are nondecreasing along that path.

Using claim 1, we can now show that whenever a node n is removed for expansion, its optimal path has been found.

Proof Optimality for Graph-search (2)

Claim 2: if $h(n)$ is a consistent heuristic, whenever we remove a node for expansion, we've found the optimal path to that node.

- Assume Claim 2 is false i.e. when n is removed from the frontier, the path found to n is suboptimal.
- Then, there must be an ancestor of n, n'', on the frontier that was never expanded but is on the optimal path to n. Contradiction as we've shown that values of f along a path are nondecreasing, and so n'' would have been removed for expansion before n.

Proof Optimality for Graph-search (3)

Claim 3: an optimal goal A will always be removed for expansion and returned before any suboptimal goal B.

- Since $h(A) = h(B) = 0$, we have $f(A) = g(A) < g(B) = f(B)$

Hence, we can conclude that A^* graph search is optimal under a consistent heuristic.

Plan

Adversarial Search (Games)

- Agents have one or more adversaries who attempt to keep them from reaching their goal(s).
- Agents can no longer run the search algorithms.
- Need to run a new class of algorithms to solve adversarial search problems = games.

Games

A game can be defined by:

- **S0**: The initial state.
- **PLAYER(s)**: Defines which player has the move in a state.
- **ACTIONS(s)**: Returns the set of legal moves in a state.
- **RESULT(s, a)**: The transition model = the result of a move.
- T**ERMINAL-TEST(s)**: true when the game is over and false otherwise.
- **UTILITY(s, p)**: A utility function, defines the final numeric value for a game that ends in terminal state s for a player p.

Game Tree: A tree where the nodes are game states and the edges are moves.

Adversarial Search (Games)

- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information?

Plan

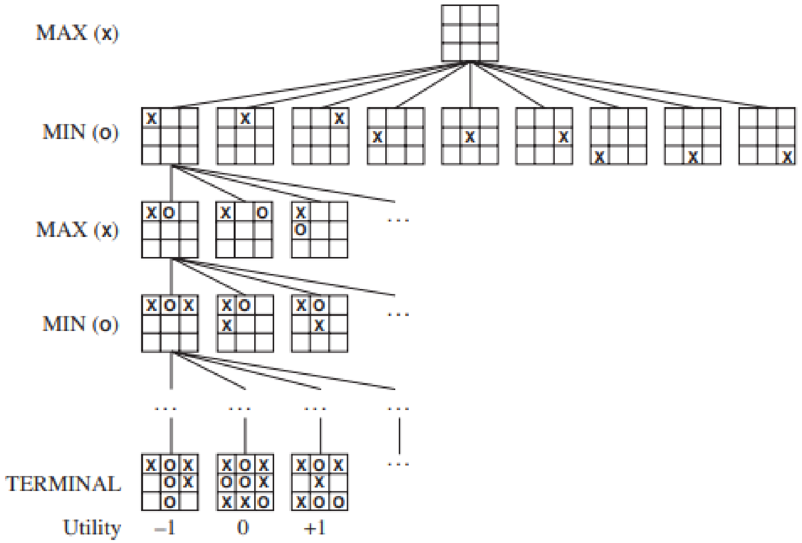
Deterministic two-player zero-sum games with perfect information

- **Deterministic**: the next state of the environment is completely determined by the current state and the action executed by the agent.
- **Perfect information**: both players have all the information about the state of the game at all times. As a result, each player knows all the information available to his opponent at the time he has to play.
- **Zero-sum game**: the total utility function to all players is the same for every instance of the game.
- Chess is a deterministic two-player zero-sum games with perfect information, the outcome is a win, loss, or draw, with values $+1$, 0, or $1/2$, the utility function is $0+1$, $1+0$ or $1/2 + 1/2$.

Deterministic two-player zero-sum games with perfect information

- **Two players**: Min and Max.
- **MAX**: maximizes result.
- **MIN**: minimizes result.
- Players alternate turns.
- **Graph Tree**:
    - Two kind of states: states under the control of MAX and states under the control of MIN
    - Each leaf state indicates UTILITY(s, MAX) of the terminal state s

Tic-Tac-Toe Game Tree



Solution in a Game

MAX must find a strategy, which specifies:

- MAX's moves in the initial state
- MAX's moves in the states resulting from every possible response by MIN.
- and so on.

Minimax value

- **Optimal strategy**: best achievable utility (for MAX) against an optimal adversary.
- Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as **MINIMAX(n)**.
- The minimax value of a terminal state is its utility.
- MAX prefers to move to a state of maximum value.
- MIN prefers a state of minimum value.

Minimax value

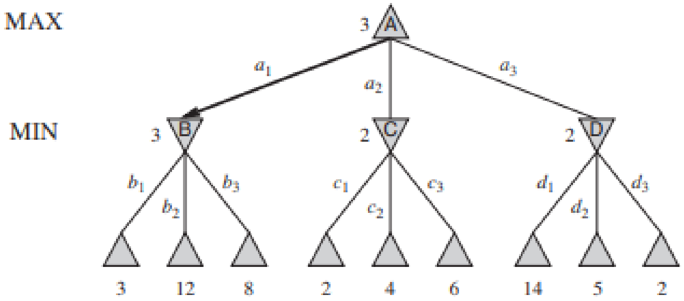
- **Optimal strategy**: best achievable utility (for MAX) against an optimal adversary.
- Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as **MINIMAX(n)**.
- The minimax value of a terminal state is its utility.
- MAX prefers to move to a state of maximum value.
- MIN prefers a state of minimum value.

$$MINIMAX(s) = \begin{cases} UTILITY(s) & if \quad TERMINAL - TEST(s) \\ max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & if \quad PLAYER(s) = MAX \\ min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & if \quad PLAYER(s) = MIN \end{cases}$$

Plan

Minimax algorithm

The Minimax algorithm can select optimal moves by a **depth-first search** of the game tree.



Minimax algorithm

```
Function MINIMAX-DECISION(state) returns an action
return $argmax_{a \in ACTIONS(s)}$ MIN-VALUE(RESULT(state, a))

Function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then
    return UTILITY(state)
end if
v = $\infty$
for each a in ACTIONS(state) do
    v = MAX(v, MIN-VALUE(RESULT(s, a)))
end for
return v

Function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then
    return UTILITY(state)
end if
v = $\infty$
for each a in ACTIONS(state) do
    v= MIN(v, MAX-VALUE(RESULT(s, a)))
end for
return v
```

$argmax_{a \in S}$ f(a) computes the element a of set S that has the maximum value of f(a).

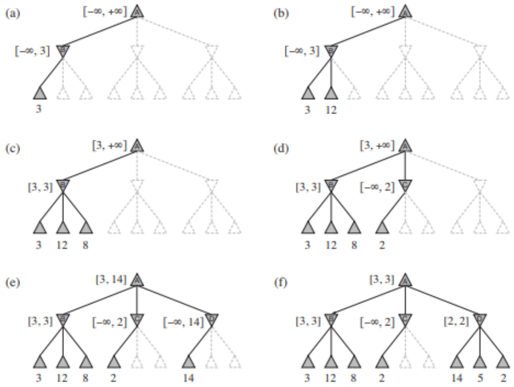
Minimax algorithm Efficiency

- Performs a complete depth-first exploration of the game tree.
- **Time Complexity**: O(b^m) with m maximum depth of the tree and b is the legal moves at each point.
- **The space complexity**: O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.
- For real games the time cost is impractical, this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Plan

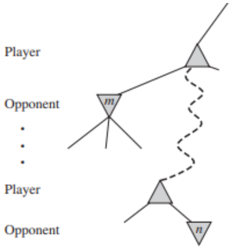
The alpha-beta search algorithm

Computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.



The alpha-beta search algorithm

The general case for alpha-beta pruning: If m is better than n for Player, we will never get to n in play.



- α = the value of the best (highest-value) choice we have found so far at any choice point along the path for MAX.
- β = the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN.

The alpha-beta search algorithm

```
Function ALPHA-BETA-SEARCH(state) returns an action
v = MAX-VALUE(state,$-\infty$,$+\infty$)
return the action in ACTIONS(state) with value v

Function MAX-VALUE(state,$\alpha$, $\beta$) returns a utility value
if TERMINAL-TEST(state) then
    return UTILITY(state)
end if
v = $-\infty$
for each a in ACTIONS(state) do
    v = MAX(v, MIN-VALUE(RESULT(s,a),$\alpha$, $\beta$))
    if v $\geq$ $\beta$ then
        return v
    end if
    $\alpha$ = MAX($\alpha$, v)
end for
return v

Function MIN-VALUE(state,$\alpha$, $\beta$) returns a utility value
if TERMINAL-TEST(state) then
    return UTILITY(state)
end if
v = $+\infty$
for each a in ACTIONS(state) do
    v = MIN(v, MAX-VALUE(RESULT(s,a) ,$\alpha$, $\beta$))
    if v $\leq$ $\alpha$ then
        return v
    end if
    $\beta$ = MIN($\beta$, v)
end for
```

Alpha-Beta Successors Ordering

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined.
- Examine first the successors that are likely to be best, if this can be done time complexity $= O(b^{m/2})$.
- Examine successors in random order the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b.

Plan

Games with Imperfect Decisions

- The minimax algorithm generates the entire game search space.
- The alpha-beta algorithm allows us to prune large parts of it.
- Alpha-beta still has to search all the way to terminal states for at least a portion of the search space.
- Usually not practical, because moves must be made in a reasonable amount of time.

Games with Imperfect Decisions

- Cut off the search earlier.
- Apply a heuristic evaluation function to states in the search, turning non terminal nodes into terminal leaves.
- Replace the utility function by a heuristic evaluation function **EVAL** to estimates utility.
- Replace the terminal test by a **cutoff test** that decides when to apply EVAL.

Games with Imperfect Decisions

- Cut off the search earlier.
- Apply a heuristic evaluation function to states in the search, turning non terminal nodes into terminal leaves.
- Replace the utility function by a heuristic evaluation function **EVAL** to estimates utility.
- Replace the terminal test by a **cutoff test** that decides when to apply EVAL.

For state s and maximum depth d:

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

Evaluation functions

- Returns an estimate of the expected utility of the game from a given position

Evaluation functions

- Returns an estimate of the expected utility of the game from a given position
- How do we design good evaluation functions?
  - Should order the terminal states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses.
  - The computation must not take too long.
  - For non terminal states, the evaluation function should be strongly correlated with the actual chances of winning.
  - Ideal function: returns the actual minimax value of the position.

Evaluation functions

- In practice:
  - Calculating features of the state.
    Chess features example: the number of white queens, black queens, ...
    $EVAL(s)$= weighted linear sum of features
    $EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$ with w_i is a weight and each f_i is a feature of the state s.
    Chess example: $f_1(s) =$ (num white queens - num black queens)

Evaluation functions

- In practice:
  - Calculating features of the state.
    Chess features example: the number of white queens, black queens, . . .
    $EVAL(s)$= weighted linear sum of features
    $EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$ with w_i is a weight and each f_i is a feature of the state s.
    Chess example: $f_1(s) =$ (num white queens - num black queens)
  - MONTE CARLO Simulation: have the algorithm play thousands of games against, the resulting win percentage can be a good approximation of the value of the state.

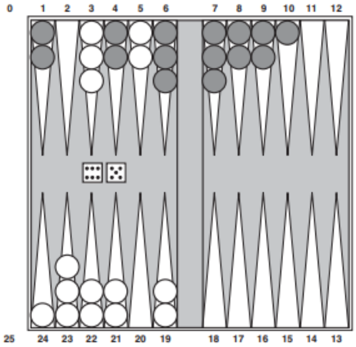
CUTOFF-TEST

- Straightforward approach: set a fixed depth d so that CUTOFF-TEST(state, depth) returns true for all depth greater than some fixed depth d. The depth d is chosen so that a move is selected within the allocated time.
- More robust approach: is to apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search. Iterative deepening also helps with move ordering
- Simple approaches can lead to errors due to the approximate nature of the evaluation function.
- The evaluation function should be applied only to positions that are unlikely to exhibit wild swings in value in the near future.

Plan

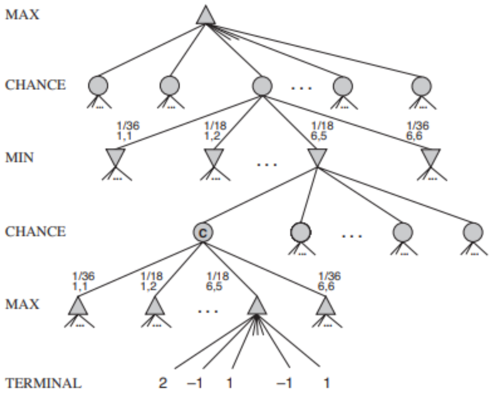
Stochastic Games

- Unpredictable external events.
- Dice are rolled at the beginning of a player's turn to determine the legal moves (Backgammon).



Stochastic Games

Include chance nodes in addition to MAX and MIN nodes



Stochastic Games

$$EXPECTIMINIMAX(s) = \begin{cases} UTILITY(s) & if \quad TERMINAL - TEST(s) \\ max_{a \in Actions(s)} EXPECTIMINIMAX(RESULT(s,a)) & if \quad PLAYER(s) = MAX \\ min_{a \in Actions(s)} EXPECTIMINIMAX(RESULT(s,a)) & if \quad PLAYER(s) = MIN \\ \sum_r P(r) EXPECTIMINIMAX(RESULT(s,r)) & if \quad PLAYER(s) = CHANCE \end{cases}$$

r represents a possible dice roll (or other chance event) and P(r) the probability of action r.

Plan

Plan

Plan