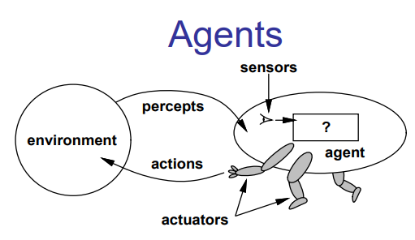
**Lesson 0**

State Spaces - Agents

Agents:

|  |  |
| --- | --- |
| **Agent** | Something that acts  *humans, robots, softbots (software agents), thermostats, etc.* Perceives the environment through sensors and acts on it through actuators |
| **Environment** | The performance measure evaluates the environment sequence |
| **Percepts** | Agent’s perceptual input (the basis for its actions) |
| **Percept Sequence** | complete history of what has been perceived. |
| **Actions** |  |
| **Actuators** |  |
| **Sensors** |  |

Agent Function vs. Agent Programs

|  |  |
| --- | --- |
| **Agent Function** | **Agent Programs** |
| An agent’s behavior is described by the agent function that maps any given **percept Sequence** to an **action** | Agent programs implement (some) agent functions.  The agent program runs on the physical architecture to produce |
| Example: Vacuum-cleaner world with two locations | Example: Vacuum-cleaner world with two locations    Percepts: location and contents, e.g. [A;Dirty]  Actions: Left, Right, Suck, NoOp **Agent Program**     |  |  | | --- | --- | | Percept Sequence | Action | | [A,Clean] | Right | | [A,Dirty ] | Suck | | [B,Clean] | Left | | [A,Clean], [A,Clean] | Suck | | [A,Clean], [A,Dirty] | Right | | … | … | |

**Problem Solving agent**

*Goal-based Agent*

A problem-solving agent is goal driven and focuses on satisfying the goal.

When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a problem-solving agent, and the computational process it undertakes is called search.

Four phase problem solving-process for agent **G.P.S.E**

|  |  |
| --- | --- |
| Goal Formulation | Organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal |
| Problem Formulation | Decides what actions should be taken to achieve the formulated goal. |
| Search | identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output |
| Execution | Executes the best optimal solution from the searching algorithms to reach the goal state from the current state |

**Lesson 0**

State Spaces – Agents, Rational Agents and Perfect Agents

Agents:

|  |  |  |
| --- | --- | --- |
| **Agent** | **Rational Agent** | **Omniscient/Perfect Agent** |
| **NOT AUTONOMOUS** | **AUTONOMOUS** | **AUTONOMOUS** |
| Something that acts | Does the right thing | Knows the actual outcome of its actions and can act accordingly |
|  | Maximizes expected performance | Perfection maximizes actual performance  *Perfection is impossible in reality* |
| Relies on prior knowledge of its designer rather than on its own percepts. | Does not only gather info, but learn as much as possible from what it percieves.  **Rational = exploration, learning, autonomy** |  |
|  | To design a rational agent, we need a task environment:  **PEAS - Performance measure, Environment, Actuators, Sensors** | NB: Rationality is NOT perfection |

**Look Up Table (Dr.)**

- Benefits: Easy to implement

- Drawbacks: Huge table, Take a long time to build the table,

No autonomy, Even with learning, need a long time to learn

the table entries

**Rational Agent**

To design a rational agent, we need to specify a task environment.

A rational agent is an agent that Acts in order to achieve the best outcome, or where there is uncertainty, the best-expected outcome. Conceptually speaking, it does the “right thing”.

|  |  |
| --- | --- |
| **Single Agent environment** | An agent solving a problem by itself is a single agent environment. |
| **Rational Behavior** | doing the right thing   * A sequence of actions causes the environment to go through a sequence of states. * If the sequence is desirable, then the agent has performed well.   *Doesn't necessarily involve thinking* |
| **Autonomy** | if its behaviour is determined by its own experience (ability to learn and adapt). |

**Lesson 0**

Types of Agents: Can all be turned into learning agents

<http://csit.ust.edu.sd/wp-content/uploads/2018/07/AI_lecture4_2018-_Intelligent-Agent-Part-2.pdf>

**Simple reflex agents (SRA)**

Select actions on the basis of the current percept, ignoring the rest of the percept history

**Goal: implicit in the condition-action rules**

**Environment:** fully observable, deterministic, static, episodic, discrete and single agent.

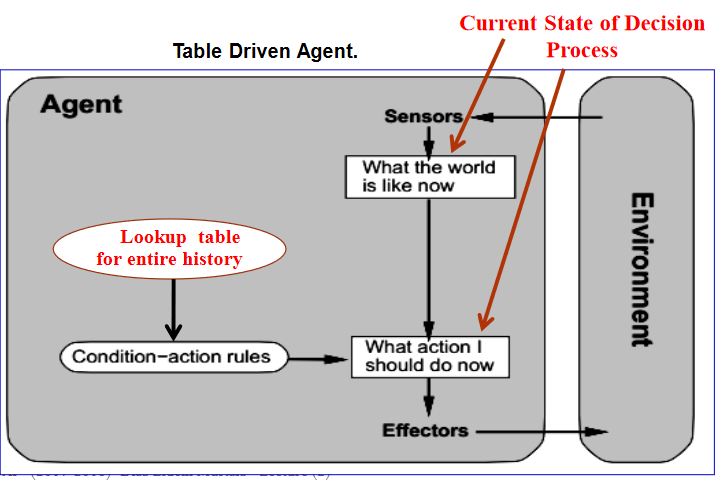
Issues:

- Size: Possible condition-action rules too big to generate and to store

- Time: Takes long to build table

- Looping: can’t make actions based on multi-level conditions

- having a model of the world does not determine what to do (rationally)



**Model-based Reflex Agents (MRA)**

*Reflex agents with State*

**Internal Model:** knowledge about “how the world works”.

Encoded internal state of the world to remember past.

Keeps track of the current state of the world (*of the parts it cannot always see*), using an internal model.

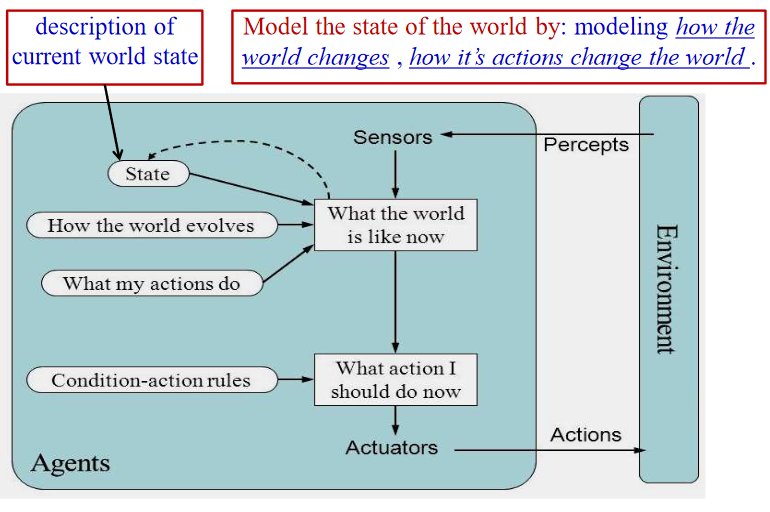
**Goal: implicit in the condition-action rules**

Issues:

- Sensors do not give entire state of the world at each input

- It then chooses an action in the same way as the reflex agent

- requires model information to be updated if changes occur



**Goal-based Agents (GA)**

*Problem Solving Agent*

keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

Search is the heart of a goal-based agent

**Goal: explicit in the condition-action rules**

- This allows us to change the goal easily (behavior can be easliy changed)

- Use search and planning to decide the best action at any given time

Issues:

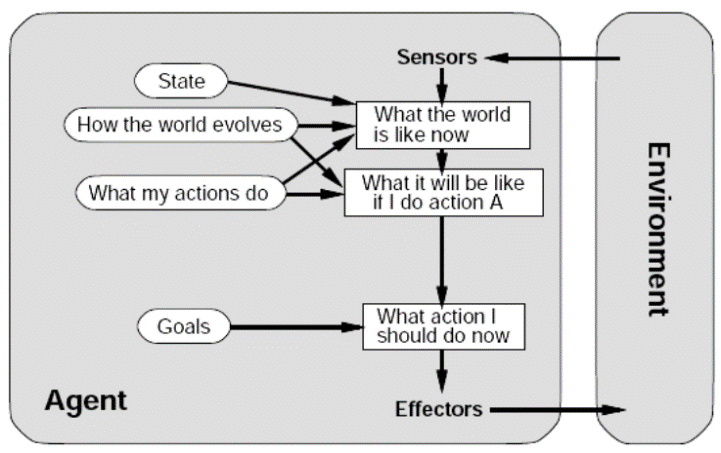
- Can be less efficient (although flexible)

- Requires search and planning

- Requires some sort of goal informations that describes desireable situations

- Knowing about the current state of the env is not always enough to decide what to do

*Goals alone are not really enough to generate high quality behavior in most environments*



**Utility-based Agents (UA)**

Uses a model of the world, along with a utility function that

measures its preferences among states of the world.

*- Utility function:*

*✓ Maps state (sequence of states) into a real number*

*✓ Resolves contradictions through trade-offs*

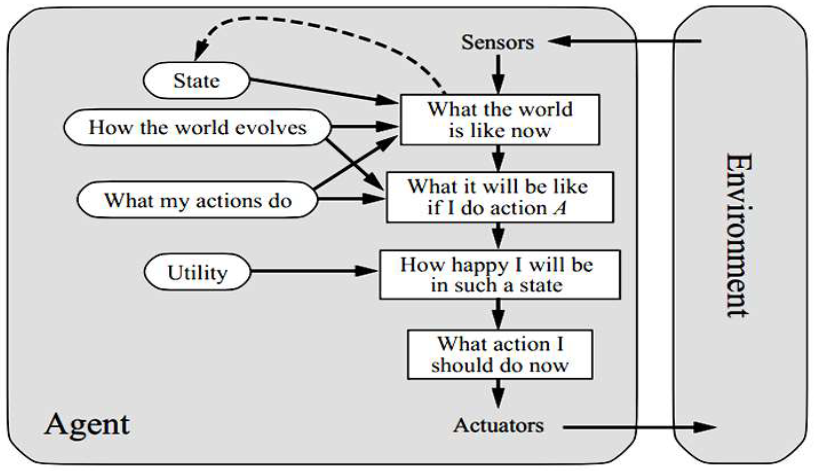
*✓ Resolves uncertainty through measure for likelihood of success*

Remember, Goal-based (GA) goals just provide a binary distinction between happy and unhappy states.

A better performance measure should allow comparison of different world states according to how happy they would make the agent if they could be achieved

*Happy = Utility, the quality of being useful*

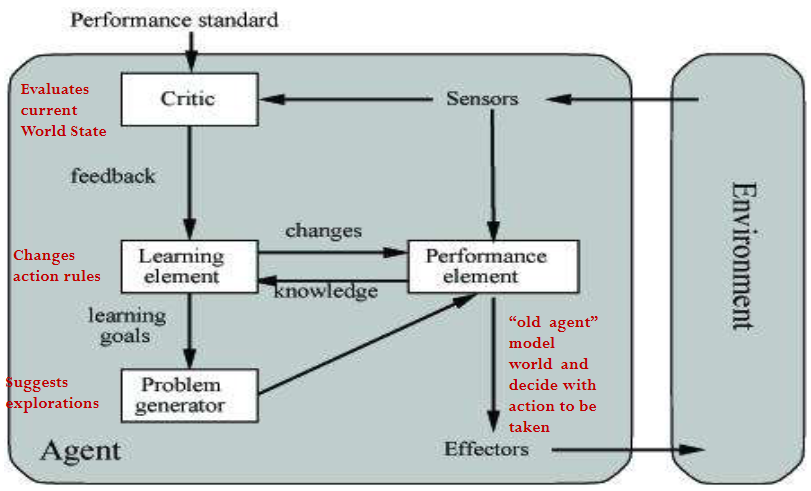
UA chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.



**Lesson 0**

Types of Agents: Learning Agents

|  |  |
| --- | --- |
| Learning element | responsible for making improvements, |
| Performance element | responsible for selecting external actions. The learning element uses feedback from critic on how agent is doing and determines how performance element should be modified to do better in future. |
| Problem Generator | Is responsible for suggesting actions that will lead to new and  informative experiences. |
| Critic | gives feedback on how agent is going and determines  how performance element should be modified to do better  in the future |



**Lesson 0**

State Spaces – Environments

To design a rational agent, we need a task environment:

**P.E.A.S.**

Performance measure Environment

Actuators

Sensors

Rational Agent

**Task Environment**

a problem specification for which the agent is a solution

Example: Automated taxi

|  |  |
| --- | --- |
| **Performance Measure** | Safe, fast, legal, comfortable trip, maximaize profits |
| **Environment** | Roads, other traffic, pedestrains, customers |
| **Actuators** | Steering, accelerator, brake, signal, horn, display |
| **Sensors** | Cameras, snar, speedometer, GPS, odometer, engine sensors, keyboard |

Environment types

<http://www.cs.stir.ac.uk/courses/ITNP4A/lectures/2%20-%20Environments.pdf>

**Fully Observable vs. partially Observable**

|  |  |
| --- | --- |
| **Fully observable** | **Partially observable** |
| Agent’s sensors give it access to complete state of environment at each point in time *Complete state means aspects relevant to*  *action choice* | Might have noisy and inaccurate sensors |
| Example: global dirt sensor | Example: local dirt sensor |

Example: Consider a game of chess. Is this a fully observable, partially observable, or unobservable environment?

[ASS1]

Fully observable. The entire state can be observed at each distinct state in the state space.

[ASS1] **Deterministic vs. Sotchastic**

|  |  |
| --- | --- |
| **Deterministic** | **Stochastic** |
| Next state is completely determined by the current state and the agent’s action. | One cannot completely determine the next state based solely on the current environment and on the agent’s actions. |

**Episodic vs. Sequential**

|  |  |
| --- | --- |
| **Episodic** | **Sequential** |
| The agent’s experience is divided into atomic episodes.  - In each episode the agent receives a percept and then performs a single action.  - Next episode does not  depend on the actions taken in previous episodes. | Current action may affect all future actions. *E.g. playing chess, taxi (both have long term consequences)* |

*short-term actions have long-term effects*

*must think ahead in choosing an action*

**Static vs. Dynamic**

|  |  |
| --- | --- |
| **Satic** | **Dynamic** |
| Cannot change while the agent is deliberating. | Can change while the agent is deliberating. |
|  | The environment is semi-dynamic if the environment itself does not change with the passage of time but the agent’s performance score does |
| Example: crossword puzzle | Example: Taxi Driver |

**Discrete vs. Continuous**

|  |  |
| --- | --- |
| **Discrete** | **Continuous** |
| Applies to state of the environment | Applies to state of the environment |
|  | percepts and actions (taxi driving continuous steering angles) |
| Example: Chess | Example: Taxi Driving (spee and location of taxi sweep through range of continuous values) |

**Single Agent vs. Multi agent**

|  |  |
| --- | --- |
| **Single agent** | **Multi agent** |
| One agent | More than one agent |
| [ASS1]  The key distinction is if an agent X’s behavior is best described as maximizing a performance measure whose value depends on agent Y’s behavior. | percepts and actions (taxi driving continuous steering angles) |
| Communication is non-existent in this env. | Communication is rational behavior in this env. |
| Example: Solving a crossword puzzle | Example: Chess (competitive multi agent environment)  [ASS1] the opponent agent X is trying to maximize its performance measure, which by the rules of chess minimises agent Y’s performance measure |
|  | Example: Taxi driving (cooperative multi agent environment) *avoiding collisions maximizes all agent’s performances* |

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | Chess with a clock | Chess without a clock | Taxi driving |
| Fully observable | Yes | Yes | No |
| Deterministic | Strategic | Strategic | No |
| Episodic | No | No | No |
| Static | Semi | Yes | No |
| Discrete | Yes | Yes | No |
| Single agent | No | No | No |

The environment type largely determines the agent design

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

**Agent Programs**

Need to develop agents – programs that take the current percept as input from the sensors and return an action to the actuators

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a small amount of code.

**Lesson 0**

State Spaces - Well-defined problems

<https://www.cpp.edu/~ftang/courses/CS420/notes/uninformed%20search.pdf>

<https://www.pearsonhighered.com/assets/samplechapter/0/1/3/6/0136042597.pdf>

A well-defined problem can be described by:

|  |  |
| --- | --- |
| **A start or initial state** | initial state that the agent starts in |
| **Actions** | A description of the possible actions available to the agent. |
| **Transition model** | This is specified by a function . A transition model is a description of what each action does. A successor is any state reachable from a given state by a single action. |
| **Path cost** | function that assigns a numeric cost to a path. Cost of a path is the sum of costs of individual actions along the path |
| **Goal test** | test to determine if at goal state |

**Lesson 0**

Searching - Search

A search algorithm takes a search problem as input and returns a solution, or an indication of failure. In this chapter we consider algorithms that superimpose a search tree over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each node in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

**Search space (vs. Goal Space)**

[ASS1]

The search space is the set of states that have to be searched for a solution.

A goal space is a set of goal states

**State space**

The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another.

**Search tree**

The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

The root node of the search tree is at the initial state.

We can expand the node, by considering the available ACTIONS for that state.

We use the RESULT function to see where those actions lead to.

We generate a new node (called a child node or successor node) for each of the resulting states.

Each child node has the initial state as its parent node.

Unexpanded nodes (reached) are called the frontier of the search tree.

**Lesson 6**

Searching- Search data structures

Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with four components:

node.STATE: the state to which the node corresponds;

node.PARENT: the node in the tree that generated this node;

node.ACTION: the action that was applied to the parent’s state to generate this node;

node.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, we use as a synonym for PATH-COST.

We need a data structure to store the frontier. The appropriate choice is a queue of some kind, because the operations on a frontier are:

IS-EMPTY(frontier) returns true only if there are no nodes in the frontier.

POP(frontier) removes the top node from the frontier and returns it.

TOP(frontier) returns (but does not remove) the top node of the frontier.

ADD(node, frontier) inserts node into its proper place in the queue.

Three kinds of queues are used in search algorithms:

[ASS1]

**Priority queue**

Nodes are added and sorted based on some key, this ensures

that certain states take priority over others during the expansion phase

**FIFO (First-in-first-out)**

used in DFS searches, nodes are added in reverse order to ensure that

the last node added will be the first node to be explored

**LIFO (Last-in-first-out)** queue/stack

Typically used in BFS searches: nodes are added in the order they are

Generated

Redundant Paths

A cycle (or loopy path) is a special case of a redundant path. So even though the state space could have only 20 states, the complete search tree is infinite because there is no limit to how often one can traverse a loop.

**Lesson 7**

Problem solving performance.

A certain list of criteria is used and considered to evaluate an algorithm’s performance:

[ASS1]

**Completeness**

*Will the algorithm find a solution if it exists?*

A complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. The algorithm should be guaranteed to find a solution when there is one, and to correctly report failure when there is not. A search algorithm must be systematic in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state.

**Cost optimality**

*Will the algorithm find the best solution (optimal path cost among*

*all solutions)?*

A solution should be guaranteed to be optimal. The algorithm should find a solution with the lowest path cost of all existing solutions.

**Time Complexity**

*How long does the algorithm take to find a solution?*

This attribute also considers the measure of difficulty of the problem. The algorithm should take the least time to find a solution, measured in

seconds, or more abstractly by the number of states and actions considered.

**Space Complexity**

*How much memory is needed to perform the search for a solution?*

This attribute also considers the measure of difficulty of the problem. The algorithm should utilize the least amount of memory needed to perform

A strategy is defined by picking the order of node expansion

- Strategies are evaluated along the following dimensions:

• Completeness: Does it always find a solution if one exists?

• Time complexity: Number of nodes generated/expanded

• Space complexity: Maximum number of nodes in memory

• Optimality: Does it always find a least-cost solution?

Time and space complexity are measured in terms of

• b: Maximum branching factor of the search tree (maximum number of successors of any node)

• d: Depth of the least-cost solution (depth of the shallowest goal node)

• m: Maximum depth of the state space, may be ∞ (maximum length of any path in the state space)

**Lesson 8**

Types of Search

**Uninformed**

The agent has no information about the underlying problem other than its definition.

*our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step.*

e.g. BFS, DFS UCS, IDS, Depth-limited, Best-first, Bidirectional

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criterion | Complete | Time | Space | Optimal? |
| BFS | Yes, if is finite |  |  | Yes, if cost = 1 per step |
| UCS | Yes, if step cost |  |  | Yes, nodes expanded in increasing order of |
| DFS | No, fails in infinite-depth spaces |  |  | No |
| DLS | No |  |  | No |
| IDS | Yes |  |  | Yes |

The evaluation function is the depth of the node—that is, the number of actions it takes to reach the node.

**Informed**

The agent has some idea of where to look for solutions.

*Our agent who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.*

e.g. A\*, bidirectional A\*, Weighted A\*, IDA (iterative Deepening A\*), RBFS (recursive best-first search) and SMA\* (simplified memory-bounded A\*) , Greedy best-first,

This type of search uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy.

The hints are in the form of a heuristic function and are denoted as . The performance of heuristic search algorithms depends on the quality of the heuristic function.

**Lesson 8**

Uninformed Search

Formal State-Space Model

<https://courses.cs.washington.edu/courses/cse415/06wi/notes/Search.pdf>

Problem = (S,s,A,f,g,c)

S = state space

s = initial state

A = actions

f = state change function f: S x A -> S

g = goal test function g: S -> {true,false}

c = cost function c: S x A x S -> R

Example: 3 coins problem

There are 3 (distinct) coins: coin1, coin2, coin3.

- The initial state is

H H T

- The legal operations are to turn over exactly one coin.

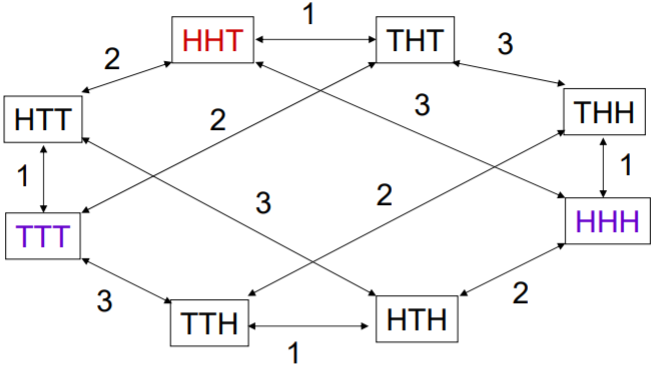
*1 (flip coin1), 2 (flip coin2), 3 (flip coin3)*

- Two goal states:

H H H

T T T

State-Space Graph

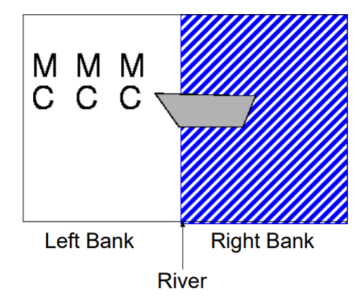


Example: Missionaries and Cannibals Problem

- Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people.

- If there are ever more cannibals than missionaries on one side of the river, the cannibals will eat the missionaries.

*(We call this a “dead” state.)*

Find a way to get everyone to the other side, without anyone getting eaten.

Define your state as (M,C,S)

M: number of missionaries on left bank

C: number of cannibals on left bank

S: side of the river that the boat is on

**Objects of the State World:**

<https://www-users.cs.umn.edu/~gini/4511/oldexams/mid1key-s09.pdf>

M M M C C C B

3 missionaries, 3 cannibals, 1 boat, a left and a right river bank.

C represents a cannibal, M represents a missionary, and B represents the location of the boat.

To describe a state, we need to know how many hikers and children are on either side of the river, and which side of the river the boat is. To achieve this, we keep 3 values per state , where:

**Representation of a State of the World:**

A state of the world is represented as 2 lists:

is the left bank.

is the right bank.

represents the location’s number of children

represents the location's number of hikers

is 1 when the boat is on the shore and 0 when it is on the opposite shore

Initial State

L<3 3 1> R<0 0 0>

Goal State:

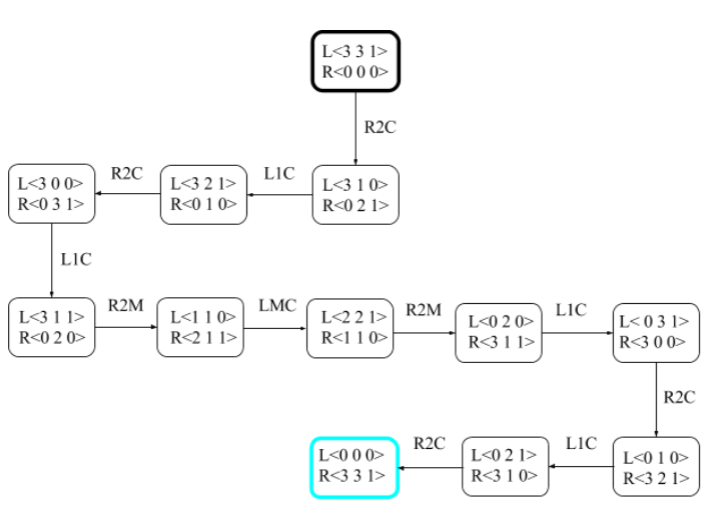
L<0 0 0> R<3 3 1>

Other solution:

<https://cs.brynmawr.edu/Courses/cs372/spring2012/slides/03_UninformedSearch.pdf>

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | L | R |
| 0 | Initial setup | MMMCCC B | - |
| 1 | Two cannibals cross over | MMMC | B CC |
| 2 | One comes back | MMMCC B | C |
| 3 | Two cannibals go over again | MMM | B CCC |
| 4 | One comes back | MMMC B | CC |
| 5 | Two missionaries cross | MC | B MMCC |
| 6 | A missionary & cannibal return | MMCC B | MC |
| 7 | Two missionaries cross again | CC | B MMCC |
| 8 | A cannibal returns | CCC B | MMM |
| 9 | Two cannibals cross | C | BMMMCC |
| 10 | One returns | CC B | MMMC |
| 11 | And brings over the third | - | B MMMCCC |

State-Space Graph



**Lesson 8.1: Uninformed Search**

Breadth-first Search

When all actions have the same cost, an appropriate strategy is breadth-first search, 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. This is a systematic search strategy that is therefore complete even on infinite state spaces.

We could implement breadth-first search as a call to BEST-FIRST-SEARCH where the evaluation function is the depth of the node—that is, the number of actions it takes to reach the node.

**Lesson 8.2: Uniform Cost Search**

Uniform Cost Search (Dijkstra’s algorithm)

The Uniform Cost Search (UCS) is an uninformed search algorithm. It used to move around a directed weighted search tree to go from a start node to one of the ending nodes with a minimum cumulative cost. It is used to find the path with the lowest cumulative cost in a weighted graph where nodes are expanded according to their cost of traversal from the root node. This is implemented using a priority queue where lower the cost higher is its priority.

Priority queue = data structure in which data of the form (item, value) can be inserted and the item of minimum value can be retrieved efficiently

• Operations:

– Init (PQ): Initialize empty queue

– Insert (PQ, item, value): Insert a pair in the queue

– Pop (PQ): Returns the pair with the minimum value

• In our case:

– item = state value = current cost g()

Uniform Cost Search Algorithm

**Basic iteration:**

1. Pop the state s with the lowest path cost from PQ

2. Evaluate the path cost to all the successors of s

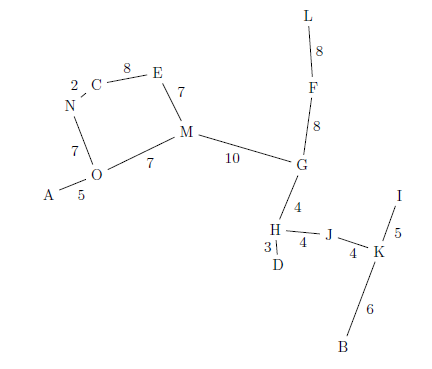
3. Add the successors of s to PQ

Redundant Paths

A cycle (or loopy path) is a special case of a redundant path.

In UCS, if a path with a child already in the frontier (PQ) is generated, then replace the existing node with the new one only if the new path’s path cost is less than that of the existing one.

Example: Perform a Uniform Cost Search (UCS) on the graph. The start node is M and the goal (11) node is F. Provide a step-wise explanation of the search as it progresses. At each step, provide the frontier, and show which node is selected for expansion. Provide the final path from the start to the goal.

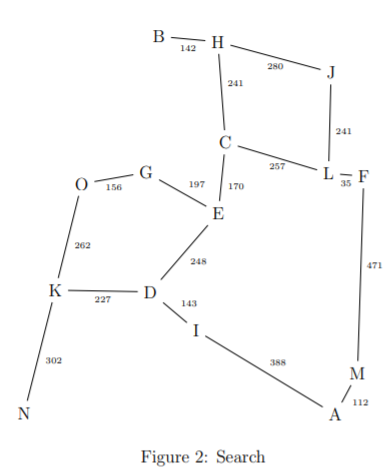


|  |  |  |
| --- | --- | --- |
| **Step** | **Node Expanded** | **Frontier (priority queue)** |
| 1 |  | M(^g = 0) |
| 2 | M | E-M(7), O-M(7), G-M(10) |
| 3 | E-M | O-M(7), G-M(10), A-O-M(12), N-O-M(14), C-E-M(15) |
| 4 | O-M | G-M(10), A-O-M(12), N-O-M(14), C-E-M(15) |
| 5 | G-M | A-O-M(12), H-G-M(14), N-O-M(14), C-E-M(15), F-G-M(18) |
| 6 | A-O-M | H-G-M(14), N-O-M(14), C-E-M(15), F-G-M(18) |
| 7 | H-G-M | N-O-M(14), C-E-M(15), D-H-G-M(17), F-G-M(18), J-H-G-M(18) |
| 8 | N-O-M | C-E-M(15), D-H-G-M(17), F-G-M(18), J-H-G-M(18) |
| 9 | C-E-M | D-H-G-M(17), F-G-M(18), J-H-G-M(18) |
| 10 | D-H-G-M | F-G-M(18), J-H-G-M(18) |
| 11 | F-G-M | J-H-G-M(18) |

Note at step 8, another path to C is found, and ignored, since it is higher than the existing path. At step 9, N is already in the explored list, so we don't add M-E-C-N to the frontier.

At step 4, expanding G-M actually generates F-G-M (a goal path), but since there may still be a shorter path to F, the algorithm carries on.

Ties were broken using lexicographical name ordering of the node name in the state space. The final path is thus: M-G-F with a path cost of 18.

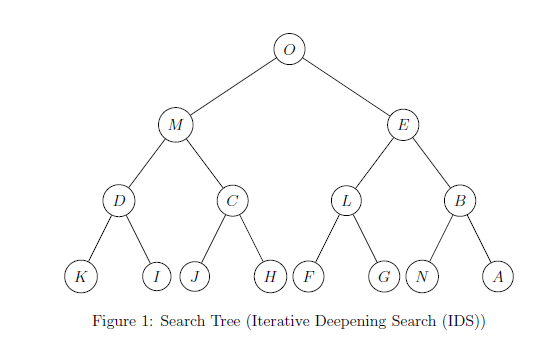
Example: Perform a UCS on the graph. The start node is H and the goal node is K.

|  |  |  |
| --- | --- | --- |
| **Step** | **Node Expanded** | **Frontier (priority queue)** |
| 1 |  | H |
| 2 | H | H-B(142), H-C(241), H-J(280) |
| 3 | H-B | H-C(241), H-J(280), H-C-E(411), H-C-L(498) |
| 4 | H-C | H-J(280), H-C-E(411), H-C-L(498), |
| 5 | H-J | H-C-E(411), H-C-L(498), |
| 6 | H-C-E | H-C-L(498), H-C-E-G(608), H-C-E-D(659) |
| 7 | H-C-L | H-C-L-F(533), H-C-E-G(608), H-C-E-D(659) |
| 8 | H-C-L-F | H-C-E-G(608), H-C-E-D(659), H-C-L-F-M(1004) |
| 9 | H-C-E-G | H-C-E-G-O(764), H-C-E-D(659), H-C-L-F-M(1004) |
| 10 | H-C-E-G-O | H-C-E-D(659), H-C-L-F-M(1004), H-C-E-G-O-K(1026) |
| 11 | H-C-E-D | H-C-E-D-K(886), H-C-L-F-M(1004), H-C-E-G-O-K(1026) |
| 12 | H-C-E-D-K | H-C-L-F-M(1004), H-C-E-G-O-K(1026) |
| 13 | H-C-L-F-M | H-C-E-G-O-K(1026) |
| 14 | H-C-E-G-O-K |  |

**Lesson 8.5: Uninformed Search**

Iterative Deepening Search

Example: ASS1 - Consider the search tree in Figure 1. Show the order in which the nodes will be expanded at each level (start with level 0 and continue until the goal test is successful), given that IDS is used. Assume the goal node is G, and that nodes are expanded from left to right (M is expanded before E and so on). (Hint: make sure you understand the difference between expansion and generation, and also that you understand when goal checks occur.)

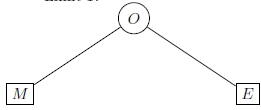


Expansion means we apply legal actions to a chosen node - this, by definition, means that our order looks somewhat different from what one may expect. Also, note that the IDS is a repeated invocation of the depth limited search. In the depth limited search, once we generate a node, we recursively call the depth limited search on that node. Thus, goal state checking happens immediately after generation for each node. In general, the children of nodes that are expanded are thus goal checked.

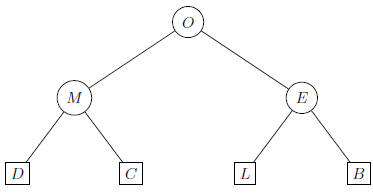
1. Limit 0: (No expansion - O is just goal tested)



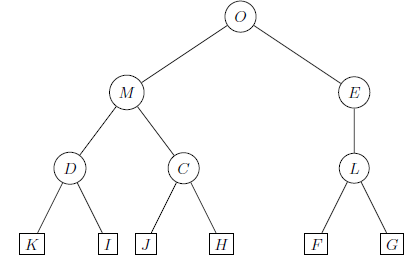
2. Limit 1: O (M and E are just goal tested, not expanded)



3. Limit 2: O M E



4. Limit 3: O M D C E L



5. Limit 4: None, search terminates once L is expanded - once L is expanded, G will be generated and goal tested on the recursive call.

**Lesson 9**

Informed (Heuristic) Search

**Best-first search:** Expand most desirable unexpanded node

Special cases: greedy search, A\* search

**Greedy best-first search:** expands the node that appears to

be closest to goal

**A∗ Search:** avoid expanding paths that are already expensive

Evaluation function

= cost so far to reach

= heuristic, estimated cost to goal from

= estimated total cost of path through to goal

search uses an admissible heuristic

i.e., where is the true cost from

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criterion | Complete | Time | Space | Optimal? |
| Greedy | No, can get stuck in loops | ) | ) | No |
| A\* | Yes | exp | All nodes | Yes |

**Lesson 9: Informed Search**

A\* Search

Example: ASS1 S2 2020

Perform an A\* search on the graph in Figure 2. The start node is N and the goal node is F. Provide a step-wise explanation of the search as it progresses. At each step, provide the frontier, and show which node is selected for expansion. Provide the final path from the start to the goal. Use table 1 for the values for each node in the graph.

|  |  |
| --- | --- |
| Node | Estimated Cost to goals |
| A | 550 |
| B | 660 |
| C | 280 |
| D | 550 |
| E | 420 |
| F | 0 |
| G | 570 |
| H | 510 |
| I | 570 |
| J | 270 |
| K | 760 |
| L | 30 |
| M | 450 |
| N | 800 |
| O | 600 |

Evaluation function

= estimated total cost of path through to goal

= cost so far to reach

= heuristic, estimated cost to goal from

|  |  |  |
| --- | --- | --- |
| **Step** | **Node Expanded** | **Frontier** |
| 1 |  |  |
| 2 | N | K-N (1062,302,760) |
| 3 | K-N | O-K-N (1164,564,600), D-K-N (1079,529,550) |
| 4 | D-K-N *use the path with the lowest* | O-K-N (1164,564,600), E-D-K-N (1197,777,420), I-D-K-N (1242,672,570) |
| 5 | O-K-N | G-O-K-N (1290,720,570), E-DK-N (1197,777,420), I-D-K-N (1242,672,570) |
| 6 | E-D-K-N | G-O-K-N (1290,720,570), C-ED-K-N (1227,947,280), I-D-K-N  (1242,672,570) |
| 7 | C-E-D-K-N | G-O-K-N (1290,720,570), L-C-ED-K-N (1234,1204,30), H-C-ED-K-N (1698,1188,510), I-D-K-N  (1242,672,570) |
| 8 | L-C-E-D-K-N | G-O-K-N (1290,720,570), F-L-CE-D-K-N (1239,1239,0), J-L-CE-D-K-N (1715,1445,270), H-C-ED-K-N (1698,1188,510), I-D-K-N  (1242,672,570) |
| 9 | F-L-C-E-D-K-N | G-O-K-N (1290,720,570), J-L-CE-D-K-N (1715,1445,270), H-C-ED-K-N (1698,1188,510), I-D-K-N  (1242,672,570) |

**Lesson 0**

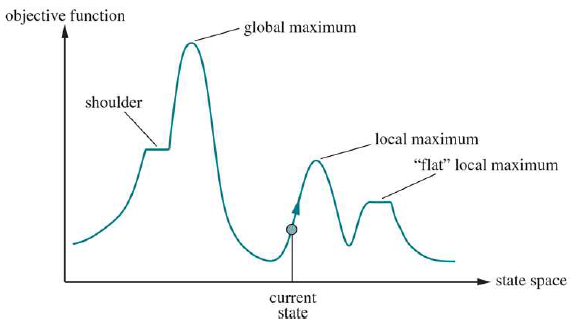
Search in Complex Environments

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages:

(1) they use very little memory; and

(2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function. Each point (state) in the landscape has an “elevation,” defined by the value of the objective function. If elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum—and we call the process hill climbing. If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum—and we call it gradient descent.



**Local Maxima**

A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will

then be stuck with nowhere else to go.

**Objective Function**

An objective function (also called either a loss, or reward function) is a function that maps the current state to some linear value which can be used to judge the fitness or goodness of the state. When we want to avoid loss, we define an objective function with respect to loss, and we try to minimize loss by minimizing the evaluation of the state using the objective function (i.e. we look for a state that results in the smallest value when evaluated using the objective function). When we want reward, we define an objective function with respect to reward and we try to maximize the objective function.

To explain the difference between hill-climbing and simulated annealing, we first need to understand what local search algorithms are.

**Local search**

The are algorithms used to solve optimization problems. They aim to find the best state by utilizing an objective function. Each state in this context has an “elevation”, which is defined by the objective function.

**Hill climbing**

In the context mentioned above, the aim is to find the highest peak (a global maximum). This process is called hill-climbing. If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum—and we call it gradient descent.

Hill climbing gradually improves a solution recursively by selecting the best neighbour based on an evaluation function recursively, until there is not a neighbour better than the current. If there is more than one best

successor, a random from the set of best successors is selected.

Hill climbing is incomplete because it gets stuck in a local maxima because “downward” moves are not allowed.

**Simulated Annealing**

Simulated annealing is technique that allows downward steps in order to escape from a local maxima. Annealing emulates the concept in metallurgy; where metals are heated to very high temperature and then gradually cooled so its structure is frozen at a minimum energy configuration.

The idea behind annealing is that, at high temperatures the algorithm

should jump out of a local maxima.

**State Spaces**

**Searching**

**Adversarial Search**

**Constraint satisfaction**

**Predicate (FOL) Logic**

**Machine Learning**