CSE473 Notes

Eric Xia

Last Updated 29 March 2021

Contents

1 Introduction

Rational: maximally achieving pre-defined goals; maximizing your expected utility
Rationality: only concerns what decisions are made and not the thought process behind them

Artificial intelligence is often observed in two dimensions, human vs. rational and thought (internal reasoning) vs. behavior (external characterization)

Turing Test: designed as a thought experiment to sidestep the philosophical vagueness of the question "can a machine think?", would need the following capabilities:
• **Natural language processing** to communicate successfully in a human language
• **Knowledge representation** to store what it knows or hears
• **Automated reasoning** to answer questions and to draw new conclusions
• **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns

Total Turing Test: requires interaction with objects and people in the real world, proposed because Turing viewed the *physical* simulation of a person as unnecessary to demonstrate intelligence, the robot would need:
• **Computer vision:** and speech recognition to perceive the world
• **Robotics** to manipulate objects and move about
Agent: an entity that operates autonomously, perceives its environment, persists over a prolonged time period, adapts to change, and creates and pursues goals
Rational agent: selects actions that maximize its (expected) utility
Limited rationality: acting appropriately when there is not enough time to do all the computations one might like
Value alignment problem: the values or objectives put into the machine must be aligned with those of the human
Incompleteness theorem: in any formal theory as strong as Peano arithmetic (the elementary theory of natural numbers), there are necessarily true statements that have no proof within the theory
Computability: capable of being computed by an effective procedure
Tractability: a problem is intractable if the time required to solve instances of the problem grows exponentially with the size of the instances
NP-completeness: provides a basis for analyzing the tractability of problems: any problem class to which the class of NP-complete problems can be reduced is likely to be intractable
Decision theory: combines probability theory with utility theory and provides a formal and complete framework for individual decisions made under uncertainty
Satisficing: making decisions that are "good enough" rather than laboriously calculating an exact optimal decision

2 Intelligent Agents

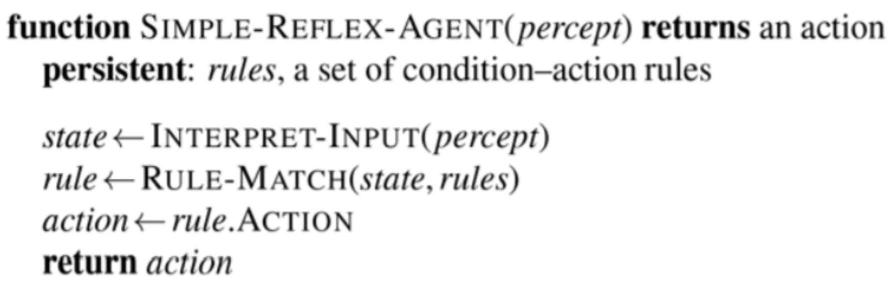
Agent: anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**
Percept: the content an agent's sensors are perceiving
Percept sequence: the complete history of everything an agent has ever perceived
Agent function: describes an agent's behavior and maps any given percept sequence to an action
• an abstract mathematical description
Agent program: a concrete implementation of an agent function, running within some physical system
Rational agent: selects actions that are expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has **Consequentialism:** we evaluate an agent's behavior by its consequences
Performance measure: evaluates any given sequence of environment states and captures the notion of desirability of results **Omniscience:** knows the *actual* outcome of its actions and can act accordingly
• rationality measures the *expected* outcome of an agent's outcomes and can act accordingly
Task environment: the problems to which rational agents are the solutions
• **PEAS: P**erformance, **E**nvironment, **A**ctuators, **S**ensors
Fully observable: if an agent's sensors give it access to the complete state of the environment at each point in time
• a task environment is effectively fully observable if the sensors detect all aspects *relevant* to the choice of action
Competitive: multiagent example is chess because opponent entity B is trying to maximize its performance measure which by the rules of chess minimize agent A's performance measure
Co-operative: multiagent example is avoiding collisions on the road maximizes the performance measures of all agents
Deterministic: the next state of the environment is completely determined by the current state and the action executed by the agent(s) such that the agent need not worry about uncertainty
Nondeterministic: not deterministic
• **stochastic:** is sometimes used as a synonym for nondeterministic however the convention is to use "stochastic" when dealing explicitly with quantifiable probabilities and "nondeterministic" when not
Agent architecture: some computing device with physical sensors and actuators that implements the agent program

$$\text{agent} = \text{architecture} + \text{program}$$

Simple reflex agent: agents that select actions on the basis of the *current* percept and ignore the rest of the percept history
• works only if the environment is fully observable **Condition-action rule**:

if *car-in-front-is-braking* **then** *initiate-braking*

A simple reflex agent:



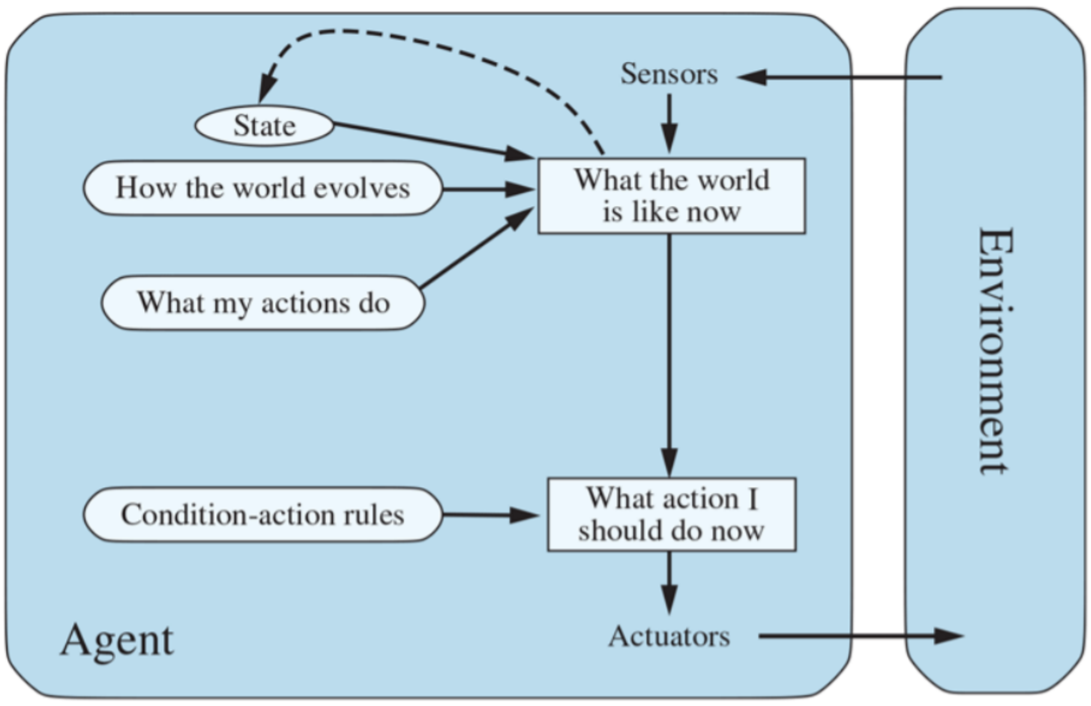
```
function SIMPLE-REFLEX-AGENT(percept) returns an action
    persistent: rules, a set of condition–action rules

    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action
```

Escaping from infinite loops is possible if the agent can randomize its actions.
Internal state: depends on percept history and thereby reflects at least some of the unobserved aspects of the current state
Transition model: the knowledge about "how the world works" whether implemented in simple Boolean circuits or in complete scientific theories
Sensor model: the type of knowledge in which the state of the world is reflected in the agent's percepts
Model-based agent: an agent that uses both transition and sensor models

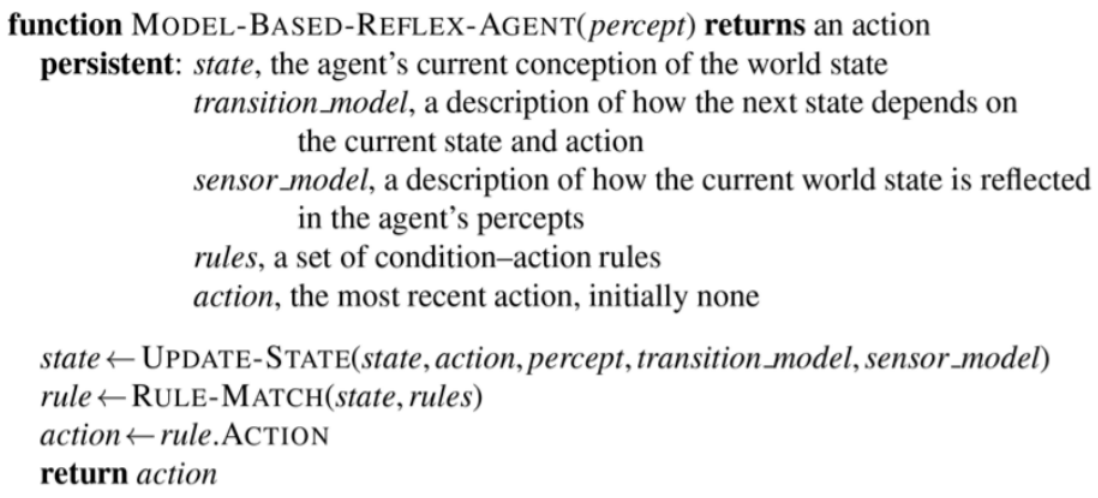
Search and *planning* are the subfields that attempt to find action sequences that achieve an agent's goals. Decision making for reflex agents and goal-based agents are different in that reflex agent designs do not consider the future - questions such as "what will happen if I do X" and "will that make me happy" are not explicitly represented. The reflex agent will brake when it sees brake lights from the car ahead. It does not know why, whereas the goal-based agent will brake when it sees brake lights ahead because that's the action it predicts that will achieve its goal of not hitting other cars. The goal-based agent may appear less efficient however it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

Utility function: an internalization of a performance measure
Model-free agent: learns what action is best in a particular situation without ever learning exactly how that action changes the environment
Learning element: responsible for making improvements
• uses feedback from the **critic** on how the agent is doing and determines how the performance element should

Model-based reflex agent

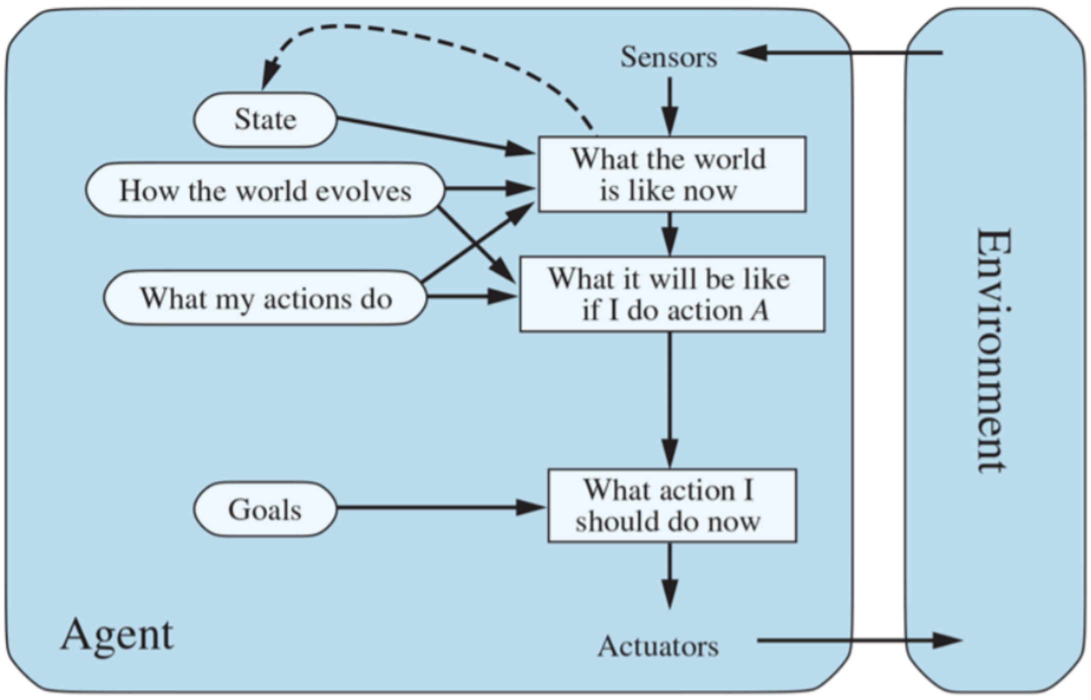


Model-based reflex agent



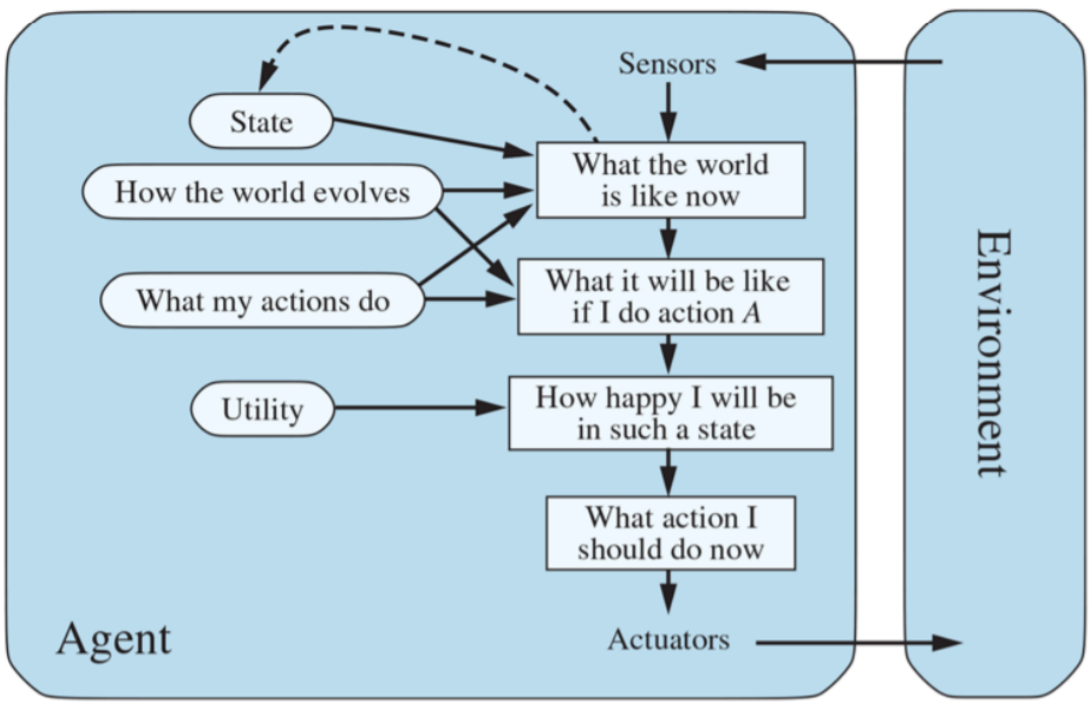
```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              transition_model, a description of how the next state depends on
                    the current state and action
              sensor_model, a description of how the current world state is reflected
                    in the agent's percepts
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, transition_model, sensor_model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Model-based, goal-based agent


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

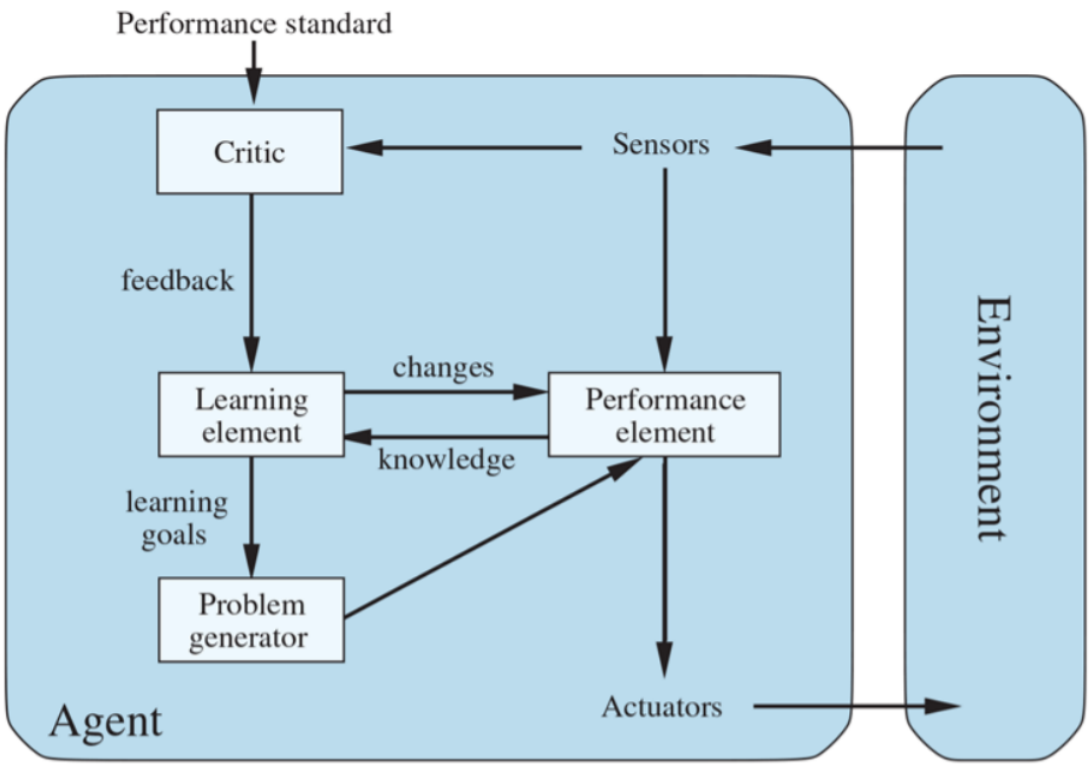
be modified to do better in the future
Performance element: responsible for selecting external actions
Problem generator: responsible for suggesting actions that will lead to new and informative experiences

Model-based, utility-based agent



Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action *A*
How happy I will be in such a state
What action I should do now
Actuators
Agent
Environment

General learning agent



Three ways to represent states and the transitions between them for components of agent programs:
Atomic representation: a state is a black box with no internal structure
Factored representation: a state consists of a vector of attribute values which can be Boolean, real-valued, or one of a fixed set of symbols
Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects

Localist representation: if there is a one-to=one mapping between concepts and memory locations
Distributed representation: the representation of a concept is spread over many memory locations and each memory location is employed as part of the representation of multiple different concepts

3 Solving Problems By Searching

3.1 Problem-Solving Agents

Planning agents: agents that use **factored** or **structured** representations of states.
Informed algorithms: the agent can estimate how far it is from the goal
Uninformed algorithms: the agent cannot estimate how far it is from the goal
Open-loop system: ignoring the percepts breaks the loop between agent and environment.
• if there is a chance that the model is incorrect or the environment is nondeterministic then the agent would be safer using a **closed-loop** approach that monitors the percepts

A search **problem** may be defined as follows:
• A set of possible **states** that the environment can be in, called the **state space**
• The **initial state** that the agent starts in
• A set of one or more **goal states**.
• The **actions** available to the agent. Given a state s, ACTIONS(s) returns a finite^2 set of actions that can be executed in s. We say that each of these actions is **applicable** in s. For example:

$$\text{ACTIONS}(\textit{Arad}) = \{\textit{ToSibiu, ToTimisoara, ToZerind}\}$$

• A **transition model** which describes what each action does. RESULT(s, a) returns the state that results from doing action a in state s. For example,

$$\text{RESULT}(\textit{Arad, ToZerind}) = \textit{Zerind}$$

• An **action cost function**, denoted by ACTION-COST(s, a, s') when we are programming or $c(s, a, s')$ when we are doing math, that gives the numeric cost of applying action a in state s to reach state s'. A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length in miles or it might be the time it takes to complete the action

A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state. An **optimal solution** has the lowest path cost among all solutions. The state space may be represented by a graph in which the *vertices are states* and the *directed edges between them are actions.*

3.2 Example Problems

Standardized problem: is intended to illustrate or exercise various problem-solving methods
• can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms
Real-world problem: example is road navigation, real-world problems are ones whose solutions people actually use and whose formulation is not standardized
Grid world problem a 2D rectangular array of square cells in which agents can move from cell to cell.
• usually the agent can move to any obstacle-free adjacent cell horizontally or vertically and in some problems diagonally
• cells contain objects which the agent can pick up, push, or otherwise act upon

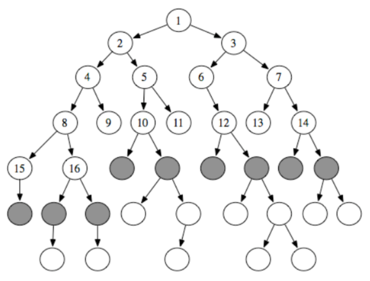
Traveling salesperson problem (TSP): a touring problem in which every city on a map must be visited. The aim is to find a tour with cost $< C$ (or in the optimization version, to find a tour with the lowest cost possible).

3.3 Search Algorithms

Search algorithm: takes a search problem as input and returns a solution, or an indication of failure
Search tree: is a superimposition of the state space graph
• each node in a search tree corresponds to a 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

When nodes in the search tree are **expanded** (visited), child nodes are *generated.* The **frontier** or **fringe** of the search tree is the set of all nodes *at the end* of all visited paths. The frontier separates two regions of the state-space graph into an interior region where every state has been expanded, and an exterior region of states that have not yet been reached.

Grey nodes make up a fringe



Best-first search: choose a node n with minimum value of some **evaluation function** $f(n)$. On each iteration we choose a node on the frontier with minimum $f(n)$ value, return it if its state is in a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path.

Nodes in search trees are 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 $g(node)$ as a synonym for PATH-COST.

Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution. The appropriate choice for a data structure to store the frontier is a *queue*, 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:
- *Priority queue*: first pops the node with the minimum cost according to some evaluation function f. It is used in best-first search
- *FIFO queue*: used in BFS
- *LIFO queue (Stack)*: used in DFS

Repeated states: are generated by a **cycle** (**loopy path**)

Search algorithms are **graph searches** if they check for redundant paths and **tree-like searches** if they do not check. In many AI problems, the graph is represented only implicitly by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of d, the **depth** or number of actions in an optimal solution, m, the maximum number of actions in any path, and b, the **branching factor** or number of successors of a node that need to be considered.

3.4 Uninformed Search Strategies

Breadth-first search (BFS) is an appropriate strategy when all actions have the same cost. We could implement breadth-first serach as a call to BEST-FIRST-SEARCH where the evaluation function $f(n)$ is the depth of the node - that is, the number of actions it takes to reach the node. We can get additional efficiency with a couple of tricks. A FIFO queue will be faster than a priority queue and will give us the correct order of nodes: new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes which are shallower than the new nodes get expanded first. In addition, *reached* can be a set of states rather than a mapping from states to nodes, because once we've reached a state, we can never find a better path to the state. This means that we can do an **early goal test**, checking whether a node is a solution as soon as it is generated, rather than the **late goal test** that best-first search uses, waiting until a node is popped off the queue.

BFS always finds a solution with a minimal number of actions, because when it is generating nodes at depth d, it has already generated all the nodes at depth $d-1$, so if one of them were a solution, it would have been found. This means it is cost-optimal for problems where all action shave the same cost. For BFS, the *memory requirements are a bigger problem than the runtime.* In general, *exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.* The total number of nodes generated in a search tree where every state has b successors is given by $O(b^d)$ which is the time and space complexity of BFS. Using BFS where the evaluation function is the cost of the path from the root to the current node is called Dijikstra's algorithm by the theoretical CS community and **uniform-cost search** by the AI community. The complexity of uniform-cost search is characterized in terms of C^*, the cost of the optimal solution, and ϵ, a lower bound on the cost of each action, with $\epsilon > 0$. The algorithm's worst-case time and space complexity is

$$O\left(b^{1+\frac{C^*}{\epsilon}}\right)$$

which can be much greater than b^d. This is because uniform-cost search can explore large trees of action with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, $b^{1+\frac{C^*}{\epsilon}} = b^{d+1}$, and uniform-cost search is similar to BFS. Uniform-cost search is cost-optimal because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

DFS is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. DFS is not cost-optimal as it returns the first solution it finds, even if it is not the cheapest. For finite state spaces that are trees DFS is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space. For problems where a tree-like search is feasible, DFS has much smaller needs for memory. A "reached" table is not kept at all and the frontier is very small. For a finite tree-shaped state-space, a depth-first tree-like search takes time proportional to the number of states, and have memory complexity of only $O(bm)$, where b is the branching factor and m is the maximum depth of the tree.

Bidirectional search: simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{\frac{d}{2}} + b^{\frac{d}{2}}$ is much less than b^d.

Comparison of uninformed search algorithms

Criterion	BFS	Uniform-Cost	DFS	Depth-Limited	Iterative Deepening	Bidirectional
Complete?	Yes	Yes	No	No	Yes	Yes
Optimal cost?	Yes	Yes	No	No	Yes	Yes
Time	$O(b^d)$	$O\left(b^{1+\frac{C^*}{\epsilon}}\right)$	$O(b^m)$	$O(b^\ell$	$O(b^d)$	$O\left(b^{\frac{d}{2}}\right)$
Space	$O(b^d)$	$O\left(b^{1+\frac{C^*}{\epsilon}}\right)$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O\left(b^{\frac{d}{2}}\right)$

3.5 Informed (Heuristic) Search Strategies

Informed search strategy: uses domain-specific hints about the location of goals to find solutions more efficiently than uninformed strategies
• these hints come in the form of a **heuristic function**, denoted $h(n)$:

$$h(n) = \text{ estimated cost of the cheapest path from the state at node } n \text{ to a goal state}$$

Greedy best-first search expands first the node with the lowest $h(n)$ value - the node that appears to be closest to the goal - on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$. Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is $O(|V|)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

A* search is the most common informed search algorithm, which is a best-first search that uses the evaluation function

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

where $g(n)$ is the path cost from the initial state to node n, and $h(n)$ is the *estimated* cost of the shortest path from n to a goal state, so we have

$$f(n) = \text{ estimated cost of the best path that continues from } n \text{ to a goal}$$

A* search is complete. Whether A^* is cost-optimal depends on certain properties of the heuristic. A key property is **admissibility**: an **admissible heuristic** is one that *overestimates* the cost to reach a goal, s.t. an admissible heuristic is *optimistic*. With an admissible heuristic, A* is cost-optimal. A slightly stronger property is called **consistency**. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by an action a, we have:

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

3.6 Heuristic Functions

Manhattan distance (city-block distance): in the sliding tile problem, the manhattan distance would be the sum of the horizontal and vertical distances.