Measuring the Performance of an Agent

- The rational agent that we are aiming at should be successful in the task it is performing
- To assess the success we need to have a performance measure
- What is rational at any given time depends on
  - The performance measure that defines the criterion of success.
  - The agent’s prior knowledge of the environment.
  - The actions that the agent can perform.
  - The agent’s percept sequence to date.

For each possible percept sequence a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Junk mail filtering has to classify e-mail messages as junk or relevant messages
- A physician has to decide whether to operate on a patient or not
- The number of correctly classified instances is not the best possible measure of performance, because right and wrong decisions have different weights
- Moving an important message to the junk folder is worse than letting through some junk mails occasionally

<table>
<thead>
<tr>
<th>True Positive</th>
<th>False positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>False negative</td>
<td>True negative</td>
</tr>
</tbody>
</table>
Properties of Environments

Task environments can be classified at least by the following properties:

**Fully vs. partially observable**
- In a fully observable environment, the agent's sensors give it all relevant aspects affecting its choice of performance.
- Hence, the agent does not need to maintain any internal state (understanding of the state of the world).
- An environment might be partially observable because of noisy and inaccurate sensors or simply due to its basic nature.

**Deterministic vs. stochastic**
- If the next state of the environment is completely determined by the current state and the action executed, then we say that it is deterministic. Otherwise, it is stochastic.
- A deterministic environment may appear stochastic if it is partially observable.

**Episodic vs. sequential**
- In an episodic task environment, the next episode does not depend on the actions taken in previous episodes.
- In sequential environments, the current decision could affect all future decisions.

**Static vs. dynamic**
- In a static environment, the agent may stop to deliberate its actions without fearing that the world changes.
- In a dynamic environment, the agent has to keep looking for the state of the environment.
- In a semidynamic environment, the environment itself does not change with the passage of time, but the agent's performance score does (the agent is penalized for the time required to plan its actions).
Task Environments /3

• **Discrete vs. continuous**
  - The distinction can be made with respect to the state of the environment, time, and the percepts and actions of the agent

• **Single agent vs. multiagent**
  - Depends on which entities one wants to view as agents or just simply as stochastically behaving objects
  - In the multiagent environment one can compete or cooperate

• **Known vs. unknown**
  - Whether the “laws of physics” of the environment are known
  - In a known environment the outcomes (or outcome probabilities in a stochastic environment) for all actions are given
  - A known environment can be partially observable

---

Real World from the Perspective of a Robot

• **Partially observable**
  - Sensors are not perfect and perceive only the close environment (holds also for humans)

• **Stochastic**
  - Wheels slip, batteries run out, parts break – one can never be certain that the intended action is fulfilled

• **Sequential**
  - The effects of actions change over time \( \Rightarrow \) a robot has to manage sequential decision problems and be able to learn

• **Dynamic**
  - When to deliberate, when to act

• **Continuous/Infinite**
  - Algorithms must work in this environment, not, e.g., in a finite discrete search space

• **Single/multiagent environment**
  - Depending on whether one wants to look at other objects as agents or stochastic parts of the environment
2.4 The Structure of Agents

- Because the agent’s history of percept-action pairs stored in a table describes the external behavior of the agent, in principle the control program of the agent could be based on table lookup.
- Obviously, this only works for very small environments.
- In more realistic situations tabulating percept-action pairs is not a viable solution.
- Instead, the agent program has to be able to decide the desired action on any percept history without tabulating all possible alternatives.

- The following agent types are the most common solutions to this problem.

Simple / Model-based Reflex Agents

- The simplest possible control program makes the agent operate solely on the current percept discarding the percept history.
- Reflexes are used in emergency situations by humans as well as by robots.
- Reflexive behavior yields correct decisions only when the task environment is fully observable.

- One can choose a set of current rules based on the agents internal state.
- Hence one can maintain a model of the world covering some information that is not directly observable.
2.4.4 Goal-based Agents

- In addition to its percepts the agent possesses knowledge of its goal.
- The goal is some assertion concerning the environment which should be satisfied.
- By combining the goal and knowledge of the effects of available actions the agent can try to satisfy the goal.
- If the goal cannot be satisfied directly through one action, one has to find out a sequence of actions to satisfy it.
- The agent may resort to search algorithms (next section) or planning (later on).

2.4.5 Utility-based Agents

- The agent may achieve its goal in many different ways – different solutions may have differences in quality.
- Setting a goal alone does not suffice to express more complex settings.
- If the possible states of the environment are assigned an order through an utility function, then the agent can try to improve its value.
- In partially observable and stochastic environments we choose the action that maximizes the expected utility of the outcomes.
- The utility function maps a state (or a sequence of states) onto a real number.
- In order to take advantage of this approach, the agent does not necessarily have to possess a utility function explicitly.
2.4.6 Learning Agents

- Programming agents for all possible tasks by hand appears to be a hopeless task
- Already Turing (1950) proposed machine learning as a method of creating intelligent systems
- Learning allows the agent to operate in initially unknown environments and become more competent than its initial knowledge alone might allow
- The learning element of an agent has to be kept distinct from the actual performance element

- We come back to the techniques of machine learning towards the end of the course

3 SOLVING PROBLEMS BY SEARCHING

- A goal-based agent aims at solving problems by performing actions that lead to desirable states
- Let us first consider the uninformed situation in which the agent is not given any information about the problem other than its definition
- In blind search only the structure of the problem to be solved is formulated
  - The agent aims at reaching a goal state
  - The world is static, discrete, and deterministic
- The possible states of the world define the state space $\Sigma$ of the search problem
In the beginning the world is in the *initial state* $s_1 \in \Sigma$

The agent aims at reaching one of the *goal states* $G \subseteq \Sigma$

Quite often one uses a *successor function* $S: \Sigma \rightarrow \mathcal{P}(\Sigma)$ to define the agent’s possible actions.

Each state $s \in \Sigma$ has a set of legal successor states $S(s)$ that can be reached in one step.

Paths have a non-negative *cost*, most often the sum of costs of steps in the path.

The definitions above naturally determine a directed and weighted graph.

The simplest problem in searching is to find out whether any goal state can be reached from the initial state $s_1$.

The actual *solution* to the problem, however, is a path from the initial state to a goal state.

Usually one takes also the costs of paths into account and tries to find a cost-optimal path to a goal state.

Many tasks of a goal-based agent are easy to formulate directly in this representation.
For example, the states of the world of 8-puzzle (a sliding-block puzzle) are all $\frac{9!}{2} = 181,440$ reachable configurations of the tiles.

- Initial state is one of the possible states.
- The goal state is the one given on the right above.
- Possible values of the successor function are moving the blank to left, right, up, or down.
- Each move costs one unit and the path cost is the total number of moves.

Donald Knuth’s (1964) illustration of how infinite state spaces can arise.

**Conjecture:** Starting with the number 4, a sequence of factorial, square root, and floor operations will reach any desired positive integer.

$$\sqrt[3]{\sqrt[2]{\sqrt{4!}}} = 5$$

- **States** $\Sigma$: Positive numbers.
- **Initial state** $s_0$: 4.
- **Actions**: Apply factorial, square root, or floor operation.
- **Transition model**: As given by the mathematical definitions of the operations.
- **Goal test**: State is the desired positive integer.
Search Tree

- When the search for a goal state begins from the initial state and proceeds by steps determined by the successor function, we can view the progress of search in a tree structure.
- When the root of the search tree, which corresponds to the initial state, is expanded, we apply the successor function to it and generate new search nodes to the tree — as children of the root — corresponding to the successor states.
- The search continues by expanding other nodes in the tree respectively.
- The search strategy (search algorithm) determines in which order the nodes of the tree are expanded.
The node is a data structure with five components:

- The state to which the node corresponds,
- Link to the parent node,
- The action that was applied to the parent to generate this node,
- The cost of the path from the initial state to the node, and
- The depth of the node

Global parameters of a search tree include:

- \( b \) (average or maximum) branching factor,
- \( d \) the depth of the shallowest goal, and
- \( m \) the maximum length of any path in the state space

3.4 Uninformed Search Strategies

- The search algorithms are implemented as special cases of normal tree traversal
- The time complexity of search is usually measured by the number of nodes generated to the tree
- Space complexity, on the other hand, measures the number of nodes that are maintained in the memory at the same time
- A search algorithm is complete if it is guaranteed to find a solution (reach a goal state starting from the initial state) when there is one
- The solution returned by a complete algorithm is not necessarily optimal: several goal states with different costs may be reachable from the initial state
3.4.1 Breadth-first search

- When the nodes of the search tree are traversed in level-order, the tree gets searched in breadth-first order.
- All nodes at a given depth are expanded before any nodes at the next level are expanded.
- Suppose that the solution is at depth $d$.
- In the worst case we expand all but the last node at level $d$.
- Every node that is generated must remain in memory, because it may belong to the solution path.
- Let $b$ be the branching factor of the search.
- This the worst-case time and space complexities are

$$b + b^2 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$
In general, the weakness of breadth-first search is its exponential (in depth) time and space usage.

In particular, the need to maintain all explored nodes causes problems.

For example, when the tree has branching factor 10, nodes are generated 1 million per second and each node requires 1,000 bytes of storage, then:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>11 μsec</td>
<td>107 kB (10^3)</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>11 μsec</td>
<td>10.6 MB (10^6)</td>
</tr>
<tr>
<td>6</td>
<td>10^6</td>
<td>1.1 sec</td>
<td>1 GB (10^9)</td>
</tr>
<tr>
<td>8</td>
<td>10^8</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>10^10</td>
<td>3 h</td>
<td>10 teraB (10^12)</td>
</tr>
<tr>
<td>12</td>
<td>10^12</td>
<td>13 days</td>
<td>1 petaB (10^15)</td>
</tr>
<tr>
<td>14</td>
<td>10^14</td>
<td>3.5 years</td>
<td>99 petaB</td>
</tr>
<tr>
<td>16</td>
<td>10^16</td>
<td>350 years</td>
<td>1 exaB (10^18)</td>
</tr>
</tbody>
</table>

Breadth-first search is optimal when all step costs are equal, because it always expands the shallowest unexpanded node.

On the other hand, to this special case of uniform-cost search, we could also apply the greedy algorithm, which always expands the node with the lowest path cost.

If the cost of every step is strictly positive the solution returned by the greedy algorithm is guaranteed to be optimal.

The space complexity of the greedy algorithm is still high.

When all step costs are equal, the greedy algorithm is identical to breadth-first search.
3.4.3 Depth-first search

- When the nodes of a search tree are expanded in preorder, the tree gets searched in depth-first order.
- The deepest node in the current fringe of the search tree becomes expanded.
- When one branch from the root to a leaf has been explored, the search backs up to the next shallowest node that still has unexplored successors.
- Depth-first search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Depth-first search requires storage of only $bm+1$ nodes.
Using the same assumptions as in the previous example, we find that depth-first search would require 156 kB (instead of 10 exaB) at depth 16 (7 trillion times less).

- If the search tree is infinite, depth-first search is not complete.
- The only goal node may always be in the branch of the tree that is examined the last.
- In the worst case also depth-first search takes an exponential time: $O(b^m)$.
- At its worst $m \gg d$, the time taken by depth-first search may be much more than that of breadth-first search.
- Moreover, we cannot guarantee the optimality of the solution that it comes up with.

3.4.4 Depth-limited search

- We can avoid examining unbounded branches by limiting the search to depth $\ell$.
- The nodes at level $\ell$ are treated as if they have no successors.

- Depth-first search can be viewed as a special case with $\ell = m$.
- When $d \leq \ell$ the search algorithm is complete, but in general one cannot guarantee finding a solution.

- Obviously, the algorithm does not guarantee finding an optimal solution.
- The time complexity is now $O(b^\ell)$ and the space complexity is $O(b\ell)$. 
3.4.5 Iterative deepening search

- We can combine the good properties of limited-depth search and general depth-first search by letting the value of the parameter $\ell$ grow gradually.
  - E.g., $\ell = 0, 1, 2, \ldots$ until a goal node is found.

- In fact, thus we gain a combination of the benefits of breadth-first and depth-first search.
  - The space complexity is controlled by the fact that the search algorithm is depth-first search.

- On the other hand, gradual growth of the parameter $\ell$ guarantees that the method is complete.
  - It is also optimal when the cost of a path is nondecreasing function of the depth of the node.

3.4.6 Bidirectional search

- If node predecessors are easy to compute — e.g., $\text{Pred}(n) = S(n)$ — then search can proceed simultaneously forward from the initial state and backward from the goal.

- The process stops when the two searches meet.

- The motivation for this idea is that $2b^{d/2} \ll b^d$.

- If the searches proceeding to both directions are implemented as breadth-first searches, the method is complete and leads to an optimal result.

- If there is a single goal state, the backward search is straightforward, but having several goal states may require creating new temporary goal states.
An important complication of the search process is the possibility to expand states that have already been expanded before. Thus, a finite state space may yield an infinite search tree. A solvable problem may become practically unsolvable. Detecting already expanded states usually requires storing all states that have been encountered. One needs to do that even in depth-first search. On the other hand, pruning may lead to missing an optimal path.

Searching with Partial Information

Above we (unrealistically) assumed that the environment is fully observable and deterministic. Moreover, we assumed that the agent knows what the effects of each action are. Therefore, the agent can calculate exactly which state results from any sequence of actions and always knows which state it is in. Its percepts provide no new information after each action.

In a more realistic situation the agent’s knowledge of states and actions is incomplete. If the agent has no sensors at all, then as far as it knows it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.
• An agent without sensors, thus, must reason about sets of states that it might get to, rather than single states.
• At each instant the agent has a belief of in which state it might be.
• Hence, the action happens in the power set $\mathcal{P}(\Sigma)$ of the state space $\Sigma$, which contains $2^{\vert \Sigma \vert}$ belief states.
• A solution is now a path that leads to a belief state, all of whose members are goal states.
• If the environment is partially observable or if the effects of actions are uncertain, then each new action yields new information.
• Every possible contingency that might arise during execution need considering.

• The cause of uncertainty may be another agent, an adversary.
• When the states of the environment and actions are uncertain, the agent has to explore its environment to gather information.

• In a partially observable world one cannot determine a fixed action sequence in advance, but needs to condition actions on future percepts.
• As the agent can gather new knowledge through its actions, it is often not useful to plan for each possible situation.
• Rather, it is better to interleave search and execution.