Foundations of Artificial Intelligence

Lorenzo Bozzoni

September 27, 2023

Contents

1 Course introduction

There will be no online streaming. The course topics are basically the same as the previous editions of the course so you are welcome to watch the existing recordings that will be available throughout this course. The list of links to the recordings is available on the course page.

For the topics that are not covered by previous year recordings, or they have substantially updated, we will provide additional recordings. Note that these are not new topics. They were already included in the course plan last year, but we were unable to include them during the previous edition.

The course calendar is available on the instructors' WeBeep pages (https://webeep.polimi.it/)

1.1 Evaluation

Evaluation is based on closed-book written exams with open questions and numerical problems. The evaluation assigns up to 32 points The laude (30 e lode) is assigned when students receive 32 points in the written exam.

Sample exams are available on the WeBeep pages of the course. The course is quite new so older exams are partially representative of the questions you should expect in the exam. The textbook is also a good source of problems and exercises. Students can reject the assigned grade and repeat the exam. All the five exams will be in presence only. There is no other way to pass the exam or increase the grade assigned to the written exam

2 Lecture 2

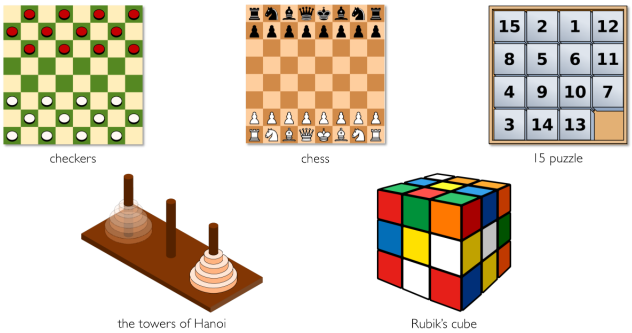
3 Problem solving by search

Problem formulation: Several real-world problems can be formulated as search problems. In a search problem, the solution can be found by exploring different alternatives.

Problem-solving agents are examples of goal-based agents

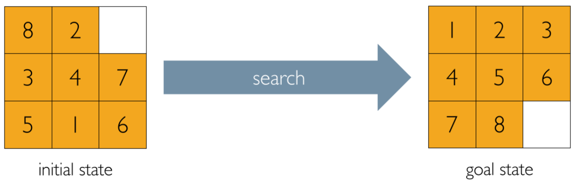
- Problem formulation
- Searching the solution
- Execute the solution

You have to start from a feasible situation, for example in the 8 problem not all configurations are solvable.



Examples of search problems.

3.1 Eight puzzle



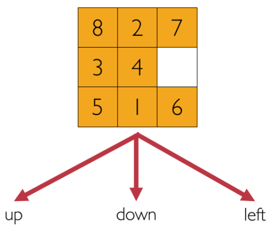
The eight puzzle

3.1.1 The states

We define a state as a feasible configuration of the 8 tiles on the 3x3 grid. The search for a solution is represented by a sequence of states in the state space.

3.1.2 The action() function

For the 8-puzzle we can define a function actions(s). The function actions(s) returns, given a state s, the actions that are applicable in that state. An action is represented by the movement of the blank position. In this case, actions(s) returns {up, down, left}



3.1.3 The result() function

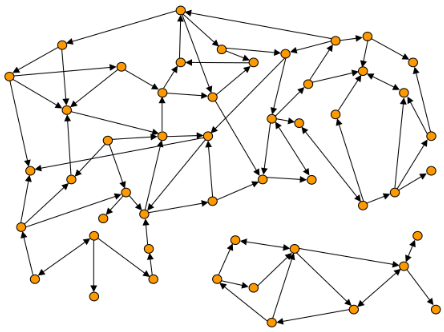
For the 8-puzzle, we can define a function result(s,a) that given a state s and an action a applicable in s, returns the state s' reached by executing a in s. State s' is a successor of s.

3.2 Search problems

A set of states S and an initial state s_0. The function actions(s) that given a state s, returns the set of feasible actions. The function result(s,a) that given a state s and an action a returns the state reached. A goal test that given a state s return true if the state is a goal state. A step cost c(s,a,s') of an action a from s to s'.

3.2.1 The state space

The space state is a directed graph with nodes representing states, arcs representing actions. There is an arc from s to s' if and only if s' is a successor of s



The solution to a search problem is a path in the state space from the initial state to a state that satisfies the goal test

3.2.2 The optimal solution

The optimal solution is the solution with the lowest cost. The cost of a path (path cost) is the sum of the costs of the arcs that compose the path (step costs) **A problem could have no solution!** Consider, for example, a case in which the starting and the ending point are in two different "island" like in the figure (the graph) above.

How many states has the state space of the n-puzzle?

- The 8-puzzle has 9! (362880) states
- The 15-puzzle has 16! (1.3x1012) states
- The 24-puzzle has 25! (1025) states

It is usually impossible to build an explicit representation of the entire state space. Consider the n-puzzle problem: suppose to generate 100 millions of states per second, it would requite 0.036 seconds to generate the 8-puzzle state space. The 15-puzzle would require a little less than 4 hours, the 24-puzzle would require more than 109 years

This is also an issue of memory not just time: the number of states in the game of chess is 10^{43}-10^{50}, the game of go has 10^{170} states, the estimated number of atoms in the universe is between 10^{79} and 10^{81}.

A problem-solving agent must find (build) a solution by exploring only a small portion of the state space.

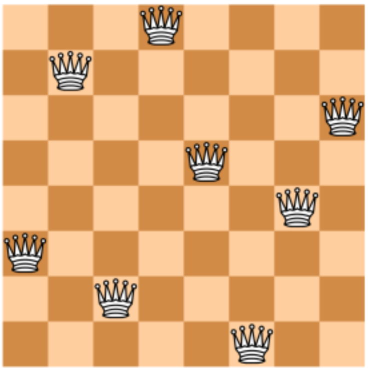
Search problems are typical of agents that operate in environments that are fully observable, static, discrete, and deterministic.

3.3 Abstraction

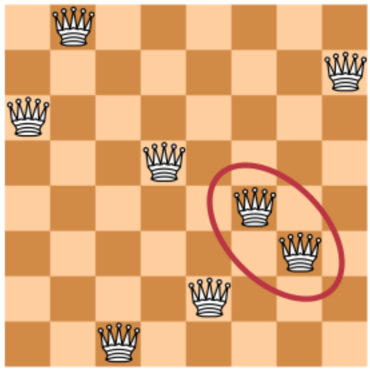
A state is an abstract representation of possible physical situations that share the same fundamental properties and differ in some details. A state of the 8-puzzle represents a set of possible physical situations that share the relative positions of the tiles but might differ in the colors of the tiles, in the materials of the tiles, etc.

3.4 The Eight-Queens Puzzle

The objective is to position 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal.



Feasible solution



Infeasible solution

A first formulation could be:

- States: all arrangements of 0, 1, 2, ..., 8 queens on the board. The state space contains 64 x 63 x ... x 57 $\sim 1.8 \times 10^{14}$ states
- Initial state: 0 queens on the board
- Function actions(): all the possible ways to add one queen in an empty square
- Function result()
- Goal test: 8 queens are on the board, with no queens attacking each other
- Step cost: irrelevant, unitary

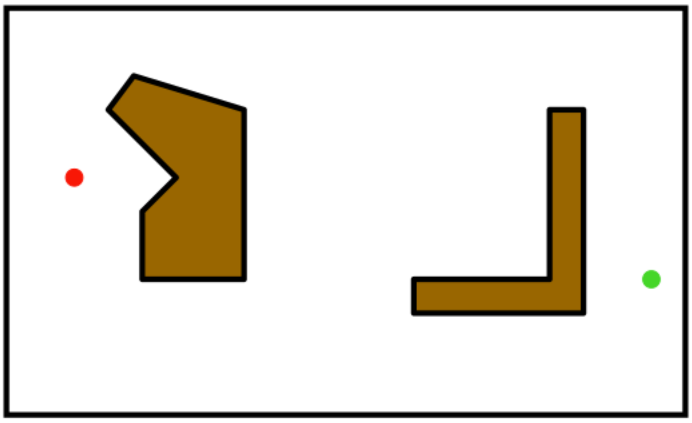
A second formulation could be:

- States: all arrangements of $k = 0, 1, 2, ..., 8$ queens in the k leftmost columns with no two queens attacking each other
- The state space is made of 2057 states
- Initial state: 0 queens on the board
- Function actions(): all the possible ways to add one queen in any square that is not attacked by any queen already in the board, in the leftmost empty column
- Function result():
- Goal test: 8 queens are on the board
- Step cost: irrelevant, unitary

In **automatic assembly problems**, the aim is to find an order in which to assemble the parts of some object. A good automatic assembly sequence should allow to add parts later in the sequence without undoing some of the work already done.

3.5 Path planning problems

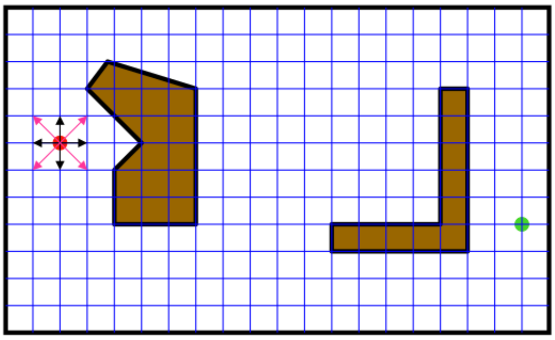
An example of path planning problem is the cleaner robot which, from a starting point, has to reach another point while avoiding obstacles.

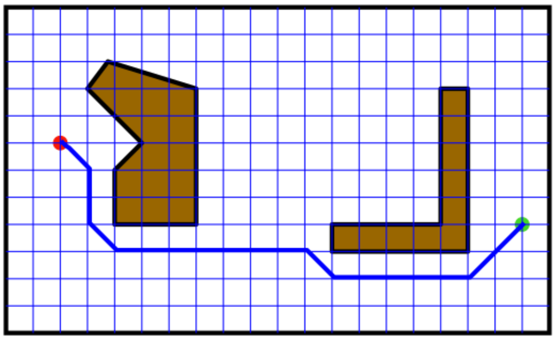


What are the possible states? How is the function actions(s) defined? And the function result(s,a)? And the goal test?

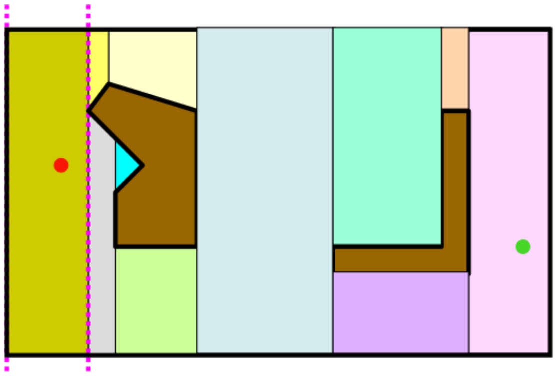
3.5.1 First formulation

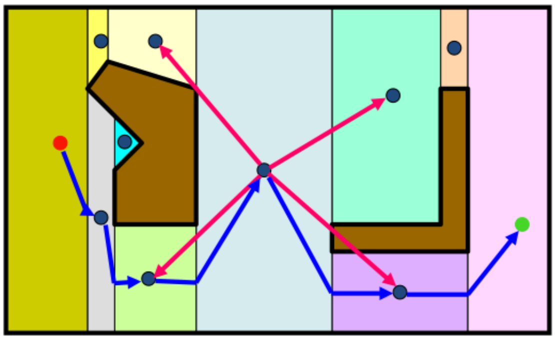
First formulation of path planning. The cost for a horizontal or vertical movement is1; 1.41 for a diagonal movement.



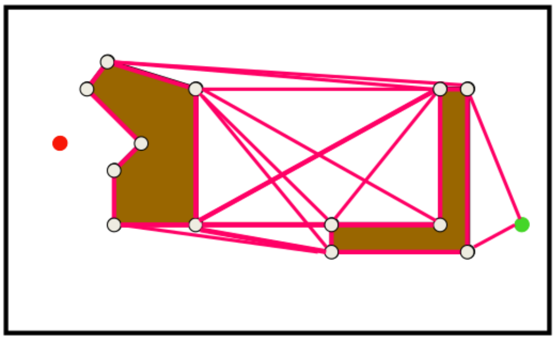


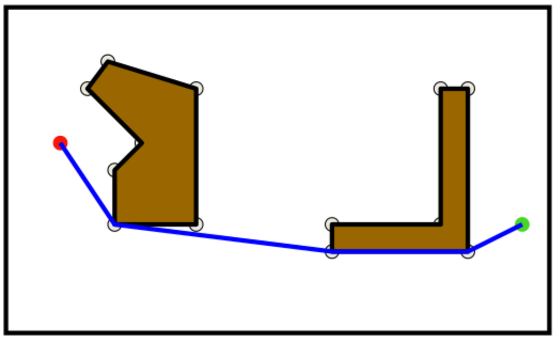
3.5.2 Second formulation





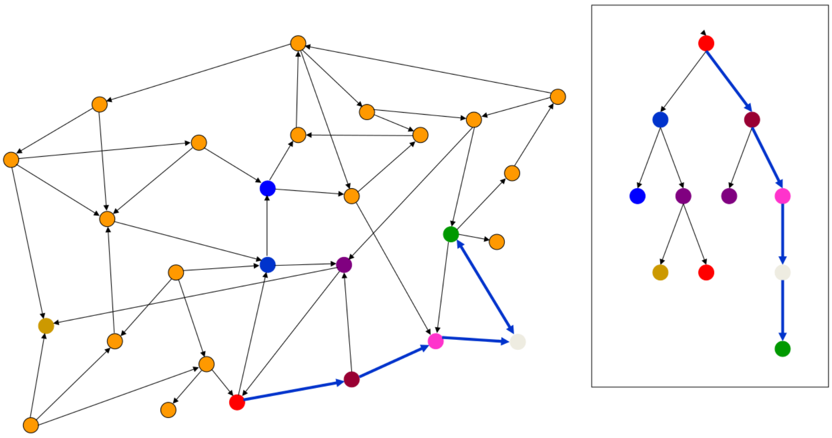
3.5.3 Third formulation





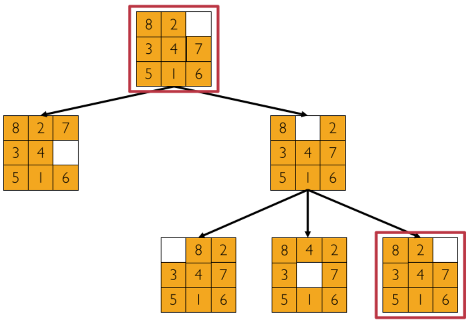
3.6 Searching for solution

Solutions to a search problem are found by building a search tree from the space state graph.



A **search tree** is composed of search nodes. Each node corresponds to a path from the initial state to a state in the state space. Each state of the space state can correspond to multiple nodes, when a state can be reached, from the initial state, following multiple paths.

If the states can be visited multiple times, then the search tree can be infinite even if the state space is finite!



In the figure you can see that the starting state is obtained again during the search. This would end up in a infinite loop of the search tree.

3.6.1 Nodes data structure

```
class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0):
    """Create a search tree Node, derived from a parent by an action."""
    self.state = state
    self.parent = parent
    self.action = action
    self.path_cost = path_cost
```

```
    self.depth = 0
    if parent:
        self.depth = parent.depth + 1
```

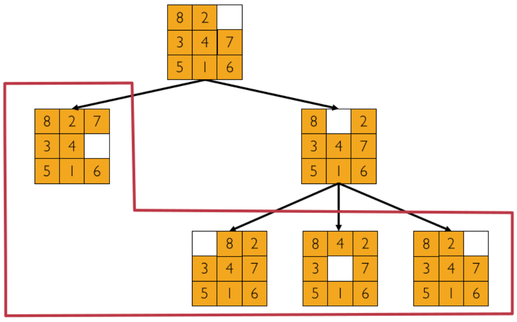
3.6.2 Tree search algorithm

```
Initialize the root of the tree with the initial state of problem

LOOP DO
    IF there are no more nodes candidate for expansion
        THEN RETURN failure
    select a node not yet expanded
    IF the node corresponds to a goal state
        THEN RETURN the corresponding solution
        ELSE expand the chosen node, adding the resulting nodes to the tree
```

3.6.3 Frontier

The frontier is the set of nodes of the search tree that have been generated, but not yet chosen. The frontier is implemented as a priority queue.



- Search Strategies: A search strategy determines the ordering of nodes in the frontier
- Node Selection: The first node in the frontier is usually selected

The expansion of a node in the search tree involves two steps:

1. Apply function actions() to the state s of node n to compute all the available actions
2. For each action a, compute the successor state s' using result(s,a) and generate a child node for each successor state

The new generated nodes are inserted in the frontier. The node expansion code from the textbook's notebook is:

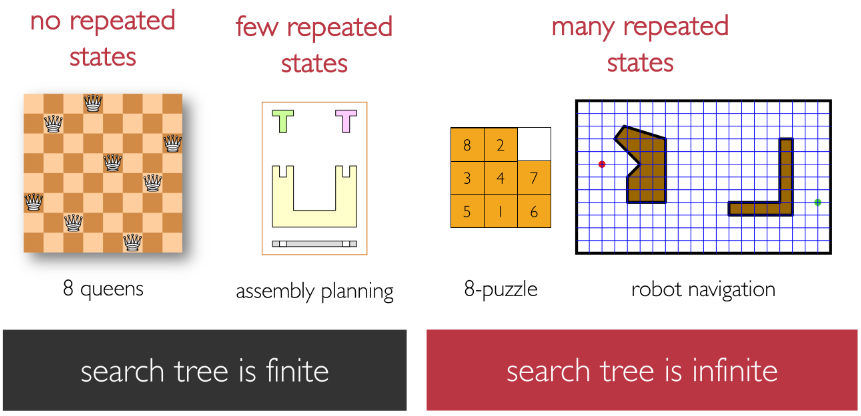
```
def expand(self, problem):
    """List the nodes reachable in one step from this node."""
    return [self.child_node(problem, action) for action in problem.actions(self.state)]
def child_node(self, problem, action):
    """[Figure 3.10]"""
    next_state = problem.result(self.state, action)
    next_node = Node(next_state, self, action, \
        problem.path_cost(self.path_cost, self.state, action, next_state))
    return next_node
```

The search strategies discussed so far can generate many nodes (in the search tree) corresponding to the same state (in the state space). This is unavoidable in problems with reversible actions. Such "repeated states" (revisited states) can generate infinite search trees even if the state space is finite. The search thus becomes inefficient.

To avoid repeated states, we need to be able to compare the states corresponding to nodes. We use a closed list containing the states from nodes already selected from the frontier. When a node is chosen for expansion, its state is checked against the closed list. When a node is expanded, its state is added to the closed list.

3.6.4 Graph search algorithm

```
initialize the root of the tree with the initial state of problem
initialize the closed list to the empty set
LOOP DO
    IF there are no more nodes candidate for expansion
        THEN RETURN failure
    choose a node not yet expanded
    IF the node corresponds to a goal state
        THEN RETURN the corresponding solution
        ELSE IF the state corresponding to the node is not in the list
            THEN add the corresponding state to the closed list
                expand the chosen node, adding the resulting nodes to the tree
```



3.6.5 Best first search

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

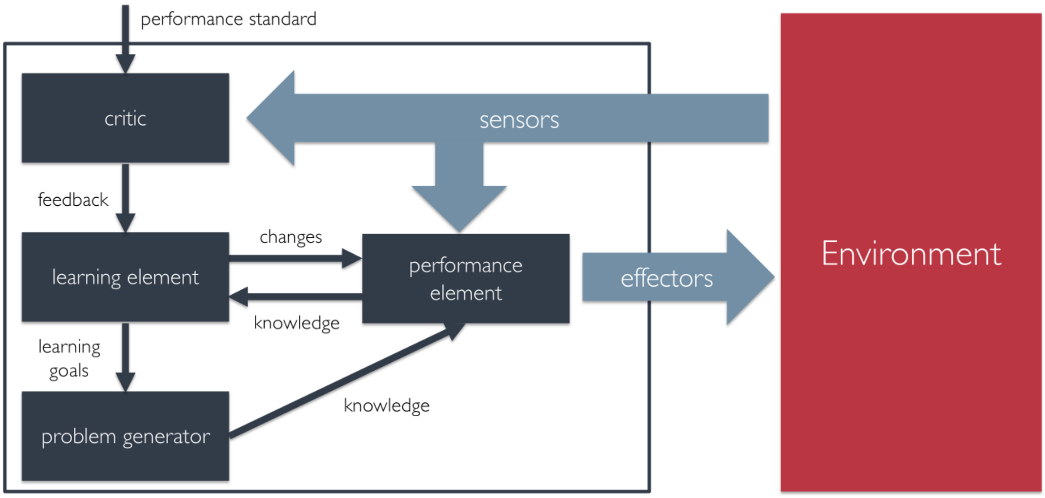
4 Learning Agents

So far we have described the following agents:

- simple reflex agents
- model-based reflex agents
- goal-based agents
- utility-based agents

Check the lecture 3 for more details. All these agents seen so far can improve their performance with learning. Every component of the agent's decisional process can be modified in order to perform better.

Learning allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. "What kind of performance element will my agent use to do this once it has learned how?"



The agents seen till now are inside the "performance element" block.
Components:

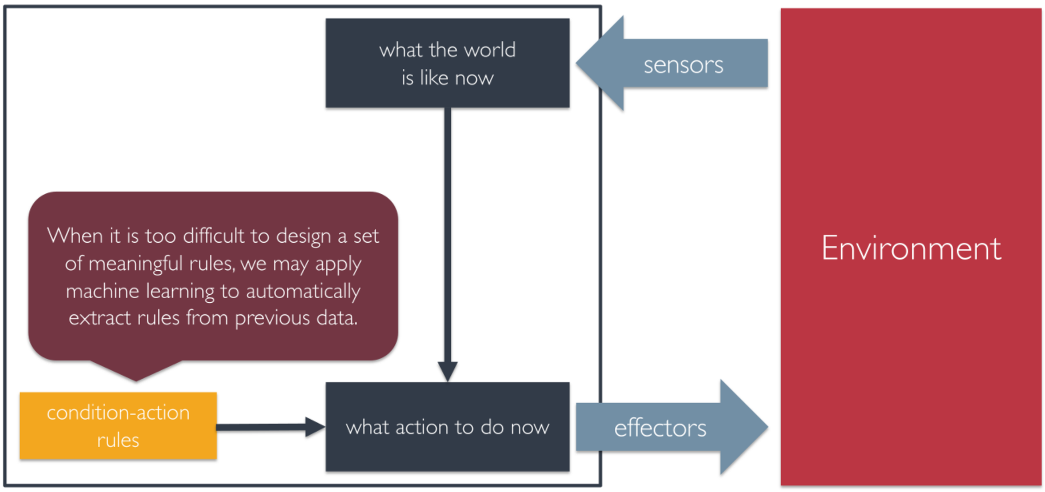
- Learning Element: it is responsible for making improvements.
- Performance Element: it selects external actions.
- Critic: it provides feedback on the agent is doing and determines how the performance element should be modified to improve future performance
- Problem Generator: it suggests exploratory actions that will lead to new and informative experiences

The performance element analyses the sensor inputs and decide what action to perform. Thus it is what we have previously considered to be the entire agent.

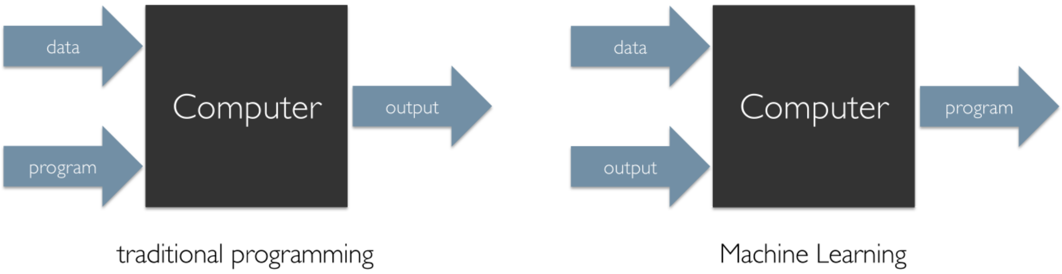
4.1 Machine learning

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, improves with experience E" Mitchell (1997).
It is an area of Artificial Intelligence focused on building algorithms capable of learning, extracting knowledge from experience. Machine Learning algorithms extract knowledge, they cannot create it. The goal is to build programs that can make informed decisions on new unseen data.

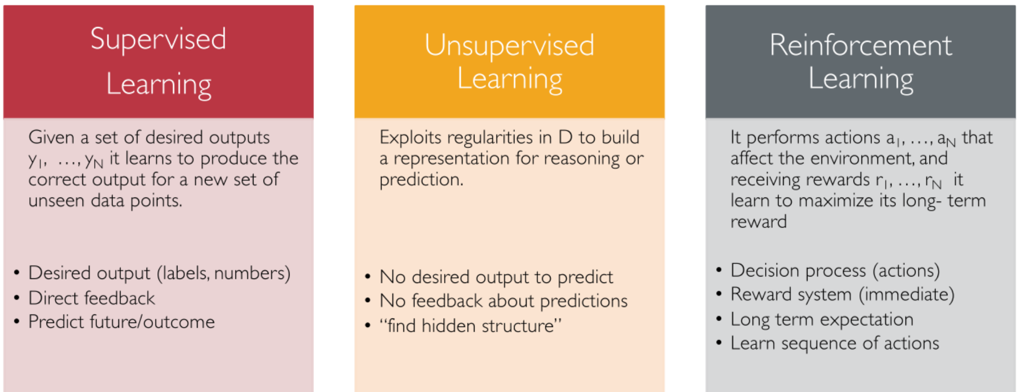
Example of application of machine learning to extract rules for a simple reflex agent:



Differences between programming and machine learning:



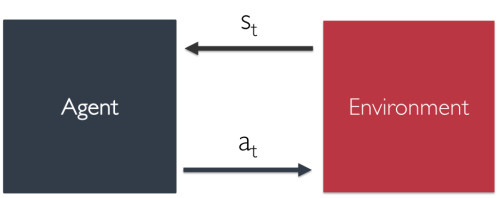
Suppose we have the experience we collected encoded as a dataset $D = x_1, \ldots, x_N$:

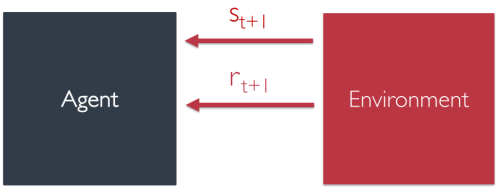


4.1.1 Reinforcement learning

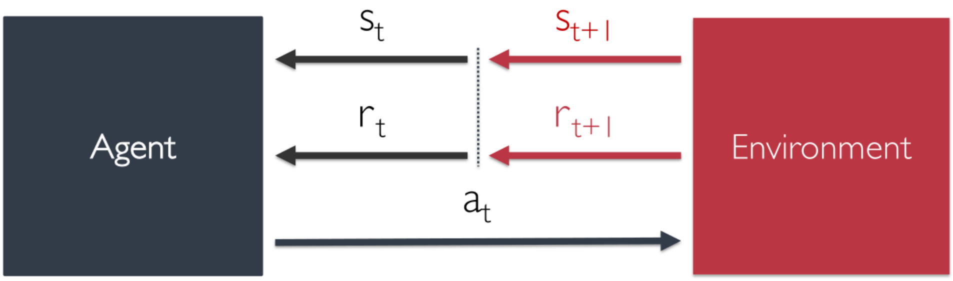
In sequential decision making:

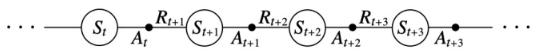
- we take a sequence of decisions (or actions) to reach the goal
- the optimal actions are context-dependent
- we generally do not have examples of correct actions
- actions may have long-term consequences
- short-term consequences of optimal actions might seem negative





At time t, the agent perceives the environment to be in state s_t and decides to perform action a_t. As a result, in the next time step $t+1$ the environment state changes to s_{t+1} and the agent receives a reward r_{t+1}. This is the generic agent-environment interaction in reinforcement learning:





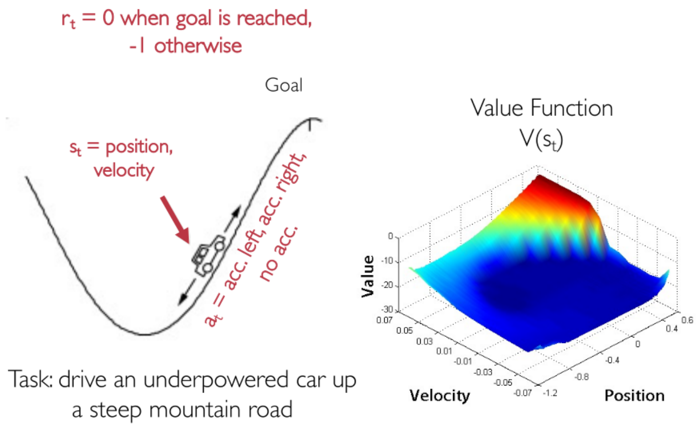
The agent's goal is to maximize the total amount of reward received How much future reward will it get when it performs a_t in s_t and then continues to do its best from there on? What is the expected payoff from s_t and a_t? The agent needs to compute an **action-value function** mapping state-action pairs to expected payoffs:

$$Q(a_s, s_t) \rightarrow \text{payoff}$$

or a **state-value functions** mapping to expected payoffs

$$V(s_t) \rightarrow \text{payoff}$$

Reinforcement learning assumes that $Q(s_t, a_t)$ is represented as a table But the real world is complex, the number of possible inputs can be huge! We cannot afford to compute an exact $Q(s_t, a_t)$ (more about this later).



What the car should do is to swing back and forth in order to achieve enough speed to climb the mountain.

Action selection

At each time step, the agent must decide what action to take in step t based on its current evaluation of the expected payoff in st using a policy function. At any given point in time, a policy $\pi(s_t)$ select what actions the agent should perform. The policy defines the behavior of an agent based on its payoff evaluation. The policy can be deterministic or stochastic.

- Deterministic policy

- In the simplest case the policy can be modeled as a function $\pi : \mathcal{S} \to \mathcal{A}$.
  - For example, the policy might simply select the action with the largest expected payoff
  - This type of policy can be conveniently represented using a table

- Stochastic policy
  - It maps each state to a probability distribution over the actions $\pi : \mathcal{S}, \mathcal{A} \to \mathcal{R}$
  - $\pi(s, a)$ returns the probability of selecting a in s
  - Since $\pi(s, a)$ is a probability distribution, it always return value greater or equal to zero and the sum over all the actions is 1
  - A stochastic policy can be used to represent also a deterministic policy

To obtain a lot of reward, the agent must prefer actions that it has tried in the past and found to lead to high payoff. However, to discover such actions, it has to try actions that it has not selected before. The agent needs to find a trade-off between the exploration of new actions and the exploitation promising actions. This is called exploration-exploitation dilemma.

- Greedy Policy: for each state, it deterministically selects an action with maximal value
- ϵ-Greedy Policy: with probability ϵ it performs a random action, with probability $1 - \epsilon$ it performs the action promising the highest payoff

The environment
The environment must satisfy the Markov property. The next state s_{t+1} and reward r_{t+1} depend only on the current state s_t and action a_t. The environment can thus be modeled as a **Markov Decision Process (MDP)** that has a one-step dynamic described by the probability distribution $p(s_{t+1}, r_{t+1}|s_t, a_t)$.

- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$
- $\sum\limits_{s \in \mathcal{S}} \sum\limits_{r \in \mathcal{R}} p(s', r|s, a) = 1 \qquad \forall s \in \mathcal{A}, \forall a \in \mathcal{A}(s)$

Expected payoff
In reinforcement learning, the agent has to maximize the reward it receives in the long run

$$G_t \doteq r_{t+1} + r_{t+2} + r_{t+3} + \dots r_{t+k} + \dots \stackrel{?}{=} \infty$$

To provide an upper bound to the payoff, we introduce a discount the future rewards by a factor $\gamma \in (0, 1)$:

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \gamma^{k-1} r_{t+k} + \dots < \infty$$

Thus, the expected reward to maximize will be defined as:

$$\mathbb{E}[G_t] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] \leq R_{max} \frac{1}{1 - \gamma}$$

The reward hypothesys
"That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)." (Rich Sutton)

In reinforcement learning, the agent learns how to maximize the expected future payoff. We must design a reward function that adequately represents our learning goal. Examples

- Goal-reward representation returns 1 when the goal is reached, 0 otherwise
- Action-penalty representation returns -1 when the state is not the goal, 0 once the goal is reached

Challenges to reward hypothesis: how to represent risk-sensitive behavior?How to capture diversity in behavior?

The value function The action-value function $Q(s_t, a_t)$ estimates the expected future payoff when performing action at in state s_t The state-value function $V(s_t)$ estimates the expected future payoff starting from s_t(in the former case) They can be both decomposed as the sum of the immediate reward received r_{t+1} and the future rewards.

The state-value function can again be decomposed into immediate reward plus discounted value of successor state (Bellman Expectation Equation):

$$V(s) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})|s_t = s]$$

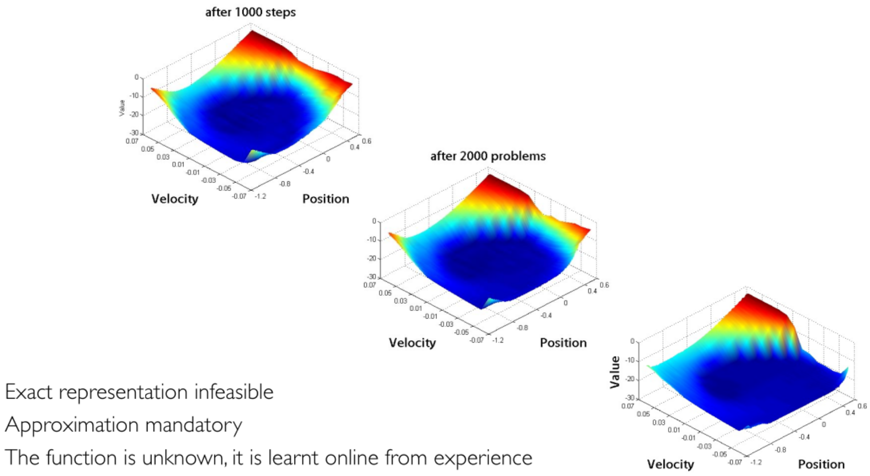
The action-value function can be similarly decomposed:

$$Q(s,a) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1}) | s_t = s, a_t = a]$$

At the beginning the table $Q(\cdot,\cdot)$ is initialized with random values. At time t,

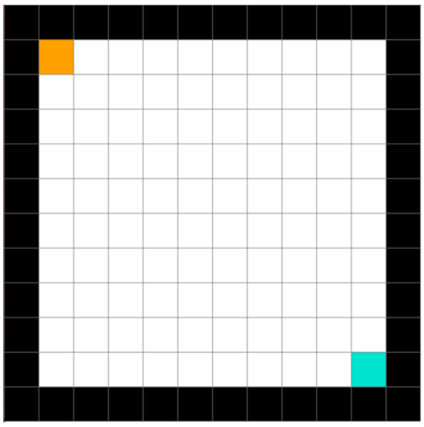
$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \beta \left[r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1},a) - Q(s_t,a_t) \right]$$

The parameters are the discount factor γ, the learning rate β, the action selection strategy $\pi(s_t,a_t)$; ϵ-Greedy is the most common choice during learning but sufficient exploration must be guarantee to tackle the exploration-exploitation dilemma.

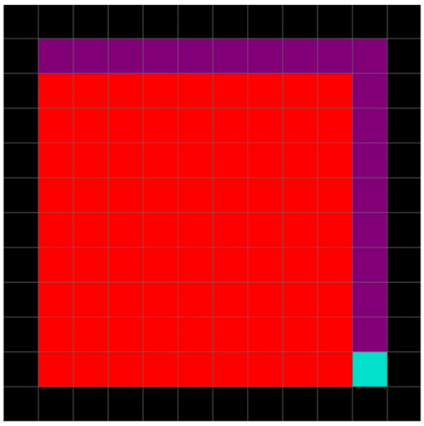


Tabular representation is infeasible in practice and approximators must be used for interesting problems. Reinforcement learning computes an unknown value function while also trying to approximate it. Approximator works on intermediate estimates while also providing information for the learning. Convergence is not guaranteed.

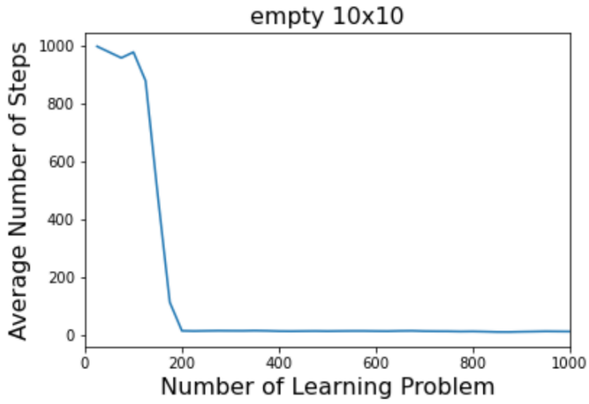
Let's now consider this simple empty environment with one start position (yellow) and one goal position (blue):



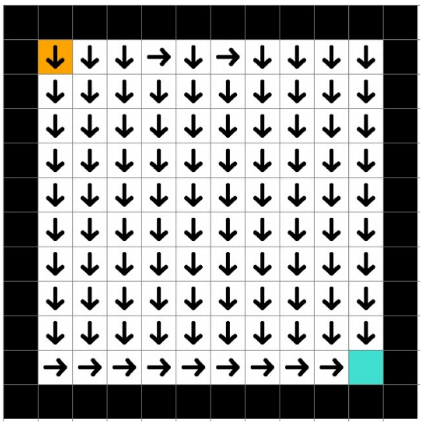
Here is the solution using a search algorithm:



When applying reinforcement learning we need to select a reward function. Let's keep it simple, zero everywhere, except when we reach the goal when we reach the goal, we receive one. Let's measure the performance as the average number of steps in the last 100 problems.

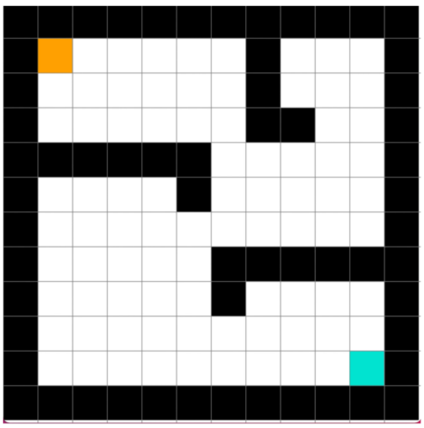


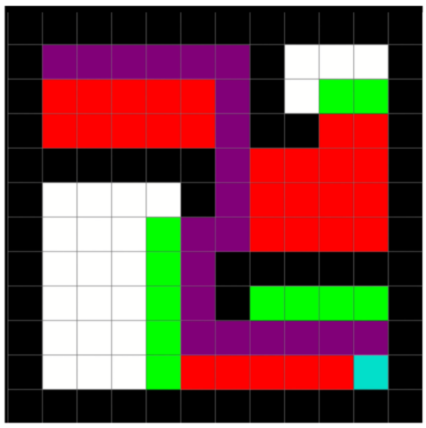
Search finds a solution, reinforcement learning an action value function. Here in the figure below is represented the best action for every position based on the action-value function on the left, while on the right the computed action value function:



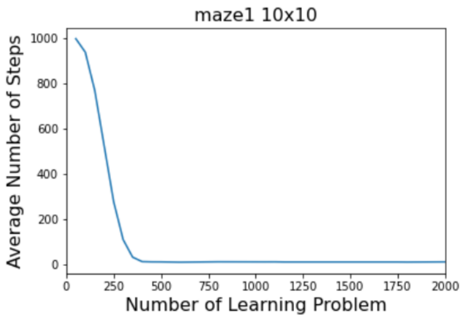
State	Up	Right	Down	Left
(1, 2)	0.397	0.440	0.440	0.000
(1, 3)	0.418	0.463	0.463	0.000
(1, 4)	0.440	0.488	0.488	0.000
(1, 5)	0.463	0.513	0.513	0.000
(1, 6)	0.488	0.540	0.540	0.000
(1, 7)	0.513	0.569	0.569	0.000
(1, 8)	0.540	0.599	0.599	0.000
(1, 9)	0.569	0.630	0.630	0.000
(1, 10)	0.599	0.663	0.000	0.000
(2, 1)	0.000	0.440	0.440	0.397
(2, 2)	0.418	0.463	0.463	0.418
(2, 3)	0.440	0.488	0.488	0.440
(2, 4)	0.463	0.513	0.513	0.463
(2, 5)	0.488	0.540	0.540	0.488
(2, 6)	0.513	0.569	0.569	0.513
...	...	...	...	...

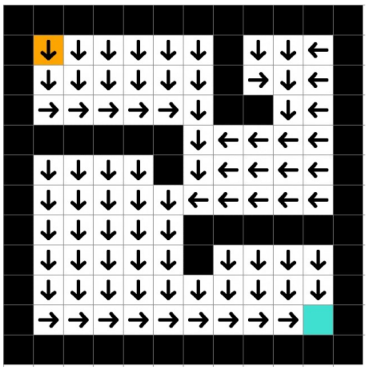
Let's consider another example. On the left is the new problem to solve while on the right there is the solution found by using the A* search algorithm:





Down below the solution obtained using reinforcement learning:





On the slides there is also the computed action value function for this last example problem.

4.1.2 Supervised learning

Given a set of inputs-output pairs $(x_1, y_1) \ldots, (x_N, y_N)$ it learns to produce the correct output for a new set of unseen data points. When the target values are real values, we have a regression problem. When they are discrete or symbolic, we have a classification problem.

5 Uninformed search strategies

Search Strategies: a search strategy determines the ordering of nodes in the frontier. Selecting a node amounts to decide the ordering of the frontier and to select the first node.
Uninformed Search Strategies: they use only the information contained in the problem formulation.

The Five Elements of Uninformed Search Strategies

1. The set of states (and an initial state)
2. Function actions()
3. Function result()
4. Goal test
5. Step cost

5.1 Tree-search algorithm

```
Initialize the root of the tree with the initial state of problem
LOOP DO
    IF there are no more nodes candidate for expansion
        THEN RETURN failure
    select a node not yet expanded
    IF the node corresponds to a goal state
        THEN RETURN the corresponding solution
        ELSE expand the chosen node, adding the resulting nodes to the tree
```

5.2 Graph-search algorithm

```
initialize the root of the tree with the initial state of problem
initialize the closed list to the empty set
LOOP DO
    IF there are no more nodes candidate for expansion
        THEN RETURN failure
    choose a node not yet expanded
    IF the node corresponds to a goal state
        THEN RETURN the corresponding solution
        ELSE IF the state corresponding to the node is not in the closed list
        THEN add the corresponding state to the closed list expand the chosen node, adding the resulting
```

5.3 Evaluation of search strategies

- **Completeness**: is the search strategy guaranteed to find a solution when there is one?
- **(Cost) Optimality**: does the search strategy find the optimal solution (minimum cost solution)?
- **(Time and Space) Complexity**: how much time and memory does the search strategy take to perform the search?
- **Parameters**:
  - **b**, the branching factor of the search tree (the maximum number of successors of any node)
  - **d**, the depth d of the shallowest goal node

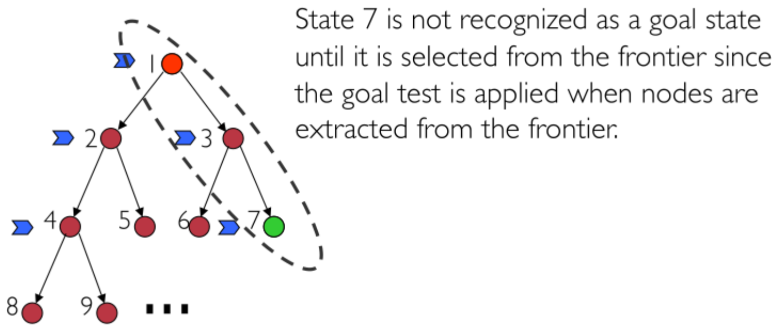
When the branching factor is large, the problem is difficult, the number of nodes will grow quickly. When the value of d is large the closest solution will require more exploration of the search tree. Sometimes, the depth d is fixed for any solution (like for example, in the eight-queen puzzle)

5.4 Breadth-first search

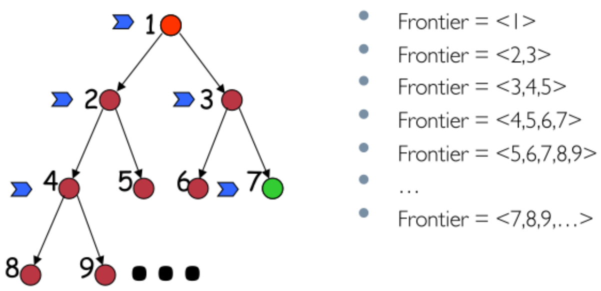
First, it selects the root, then all the root's successors, then all their successors, and so on. In breadth-first search, all nodes at level k of the search tree must be selected before selecting any node at level $k + 1$.

Breadth-first search always selects the node with minimum depth (the shallowest one). When more nodes are at the same depth, a tie-breaker is applied determined by the insertion order.

Example:



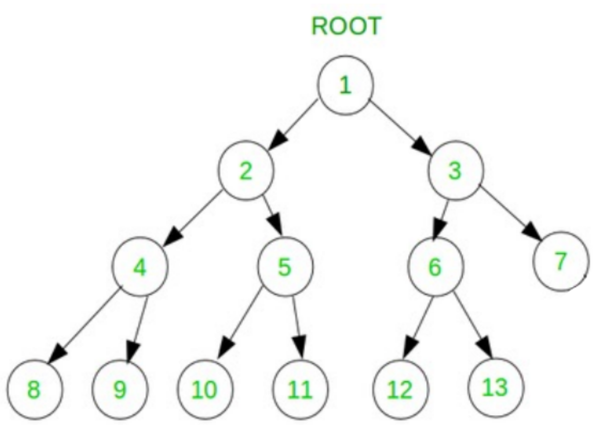
Breadth-first search can be implemented by considering the frontier a FIFO (First In First Out) queue. The new nodes are appended at the end of the frontier. Thus, the nodes generated first are selected first.



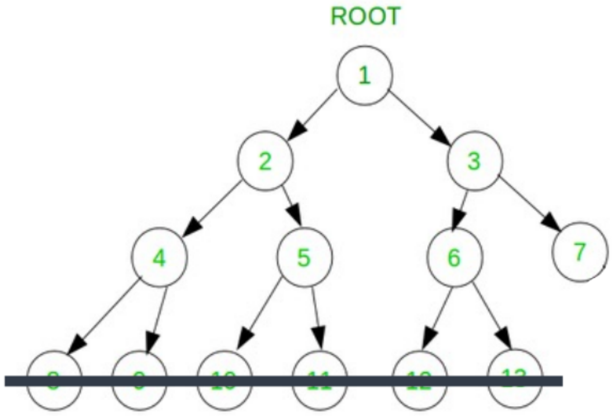
5.4.1 Evaluation of Breadth-First Search

- Breadth-first search is complete: assuming b is finite
- Breadth-first search is optimal: it finds the shallowest solution when step costs are all equal (e.g., they are unitary). When the path cost to any node is a non-decreasing function of the depth of the node
- Complexity: breadth-first search has temporal and spatial (worst-case) complexity of $O(bd + 1)$ nodes. The root of the search tree generates b nodes at level 1. Each one of them generates b nodes, totaling b^2 nodes at level 2. Thus, at level d+1 there will be $b^{d+1} - b$ nodes

Worst case of Breadth-First Search:



In breadth-first search, we can perform the goal test when a node is generated, instead of when it is selected from the frontier. This variant is still complete and optimal (when the path cost to any node is a non-decreasing function of the depth of the node). This variant has temporal and spatial (worst-case) complexity of $O(b^d)$ nodes since we avoid expanding a node before adding it to the frontier.



Here is the breadth-first search algorithm from the textbook:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

And the python version with the goal test when a node is selected from the frontier is:

```
def breadth_first_search(problem, verbose=False):
    node = Node(problem.initial)
    frontier = FIFOQueue([node])
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure,visited_states
```

A similar python version with the goal test when a node is generated:

```
def breadth_first_search_early_goal_test(problem):
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
```

```
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
               return child
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure
```

We could implement breadth-first search as a call to Best-First-Search where the evaluation function f(n) is the depth of the node (the number of actions it takes to reach the node). However, the native implementation is more efficient since:

- A first-in-first-out queue will be faster than a priority queue
- 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
- Thus, breadth-first search can perform 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

5.5 Uniform cost search

It generalizes breadth-first search: it sorts the nodes in the frontier according to their increasing path cost from the root and it selects the node n with the smallest path cost from the root. It does not necessarily choose the shallowest node! It is implemented as best-first-search where the evaluation function f(n) returns the path cost.

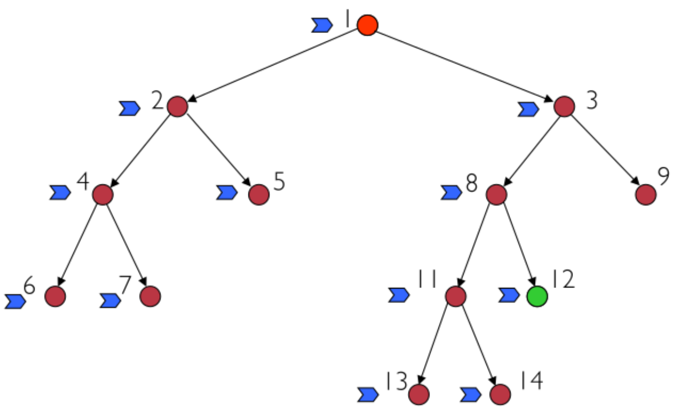
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

5.5.1 Evaluation of Uniform-Cost Search

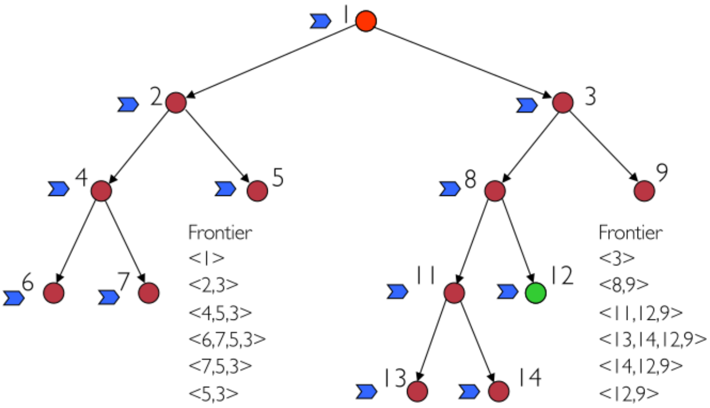
- Uniform-cost search is complete: assuming b finite and step costs larger than a small positive ϵ
- Uniform-cost search is optimal: uniform-cost search expands nodes in order of their optimal path costs
- Complexity: Uniform-cost search has temporal and spatial worst-case complexity of $O(b^{(1+\lfloor C^*/\epsilon \rfloor)})$ nodes, where C^* is the cost of the optimal solution

5.6 Depth-first search

First, it selects the root node, then, it expands one of its successors, then one of its successors, and so on. Depth-first search always selects the node at maximum depth (the deepest one). When it reaches a node without successors, it "backtracks" and chooses one of the deepest nodes not yet chosen.



Depth-first search can be implemented by representing the frontier as a Last-In-First-Out (LIFO) queue. The generated successor nodes are inserted at the front of the frontier. The most recently generated nodes are selected first.



The python *recursive implementation* of depth-first search is:

```
def depth_first_recursive_search(problem, node=None):
    if node is None:
        node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    elif is_cycle(node):
        return failure
    else:
        for child in expand(problem, node):
            result = depth_first_recursive_search(problem, child)
            if result:
                return result
    return failure
```

The python *iterative implementation* of depth-first search is:

```
def depth_first_search_iterative(problem):
    "Search deepest nodes in the search tree first."
    frontier = LIFOQueue([Node(problem.initial)])
    result = failure
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        elif not is_cycle(node):
            for child in expand(problem, node):
                frontier.append(child)
    return result

def is_cycle(node, k=30):
    "Does this node form a cycle of length k or less?"
    def find_cycle(ancestor, k):
        return (ancestor is not None and k > 0 and
        (ancestor.state == node.state or find_cycle(ancestor.parent, k - 1)))
    return find_cycle(node.parent, k)
```

5.6.1 Evaluation of Depth-First Search

- Depth-first search is not complete: it can follow path of infinite length
- Depth-first search is not optimal: it does not find the shallowest solution

- Complexity: given the maximum depth of the search tree m, depth-first has spatial (worse-case) complexity of $O(bm)$ nodes. It has temporal (worst-case) complexity of $O(b^m)$ nodes; m could be infinite; $m \geq d$ (often $m \gg d$). At each step of the search, only a single path (from the root to a leaf node) must be stored in memory; also, the nodes not yet expanded at each level should be stored

Also depth-first search could be implemented as a call to best-first search where the evaluation function f(n) is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states.

5.7 Depth-limited search

Depth-limited search is a depth-first search with a predetermined depth limit L. Nodes at level L are assumed to have no successors. Depth-limited search is complete when $L \geq d$, however, d is often unknown.
Depth-limited search is not optimal. Complexity: Depth-limited search has spatial (worst-case) complexity of $O(bL)$ nodes, it has temporal (worst-case) complexity of $O(b^L)$ nodes.

Python recursive implementation of depth-limited search:

```
def depth_limited_search(problem, limit=40):
    "Search deepest nodes in the search tree first."
    frontier = LIFOQueue([Node(problem.initial)])
    result = failure
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        elif len(node) >= limit:
            result = cutoff
        elif not is_cycle(node):
            for child in expand(problem, node):
                frontier.append(child)
    return result
```

5.8 Iterative deepening search

It performs repeated depth-limited searches, increasing the depth limit $L : L = 0, L = 1, L = 2, \ldots$. Recursive python implementation of iterative deepening search:

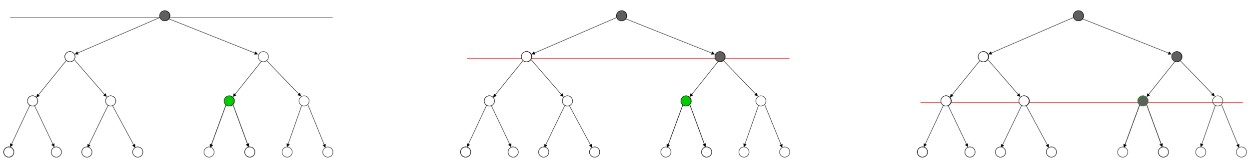
```
def iterative_deepening_search(problem):
    "Do depth-limited search with increasing depth limits."
    for limit in range(1, sys.maxsize):
        result = depth_limited_search(problem, limit)
        if result != cutoff:
    return result
```

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Figure 3.12 Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

Example of iterative deepening search:



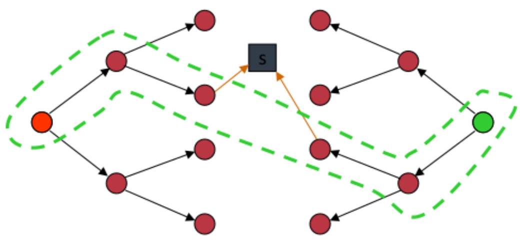
Every time the depth is increased, the entire tree is regenerated (optimization exists)

5.8.1 Evaluation of Iterative Deepening Search

- Iterative deepening search is complete (assuming b finite): if a solution exists, eventually $L = d$
- Iterative deepening search is optimal: when the step costs are all equal (as in breadth-first search)
- Complexity: iterative deepening search has spatial (worst-case) complexity of $O(bd)$ nodes. It has temporal (worst-case) complexity of $O(b^d)$ nodes

5.9 Bidirectional search

It performs two searches in parallel a "forward" search from the initial state to a goal state a "backward" search from a goal state to the initial state. Example of bidirectional search:



5.10 Tree-search Vs. Graph-search algorithms

Tree-search algorithms work well when the state space is a tree. Graph-search algorithms work well when the state space is a graph (general case).

Several variants are possible. For example, repeated states can be checked when a node is generated (not when a node is expanded). Trade-off between memory usage for storing the closed list and time spent to check the closed list.

5.11 Summary of uninformed search strategies

	with repeated states		**without** repeated states			
Search strategy	Complete	Optimal	Complete	Optimal	S(n)	T(n)
Breadth-First Search	yes	yes	yes	yes	O(b^d)	O(b^{d+1})
Uniform-Cost Search	yes	yes	yes	yes	O($b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}$)	O($b^{1+\lfloor \frac{C^*}{\varepsilon} \rfloor}$)
Depth-First Search	no	no	yes	no	O(bm)	O(b^m)
Limited-Depth Search	no	no	no	no	O(bL)	O(b^L)
Iterative Deepening	yes	yes	yes	yes	O(bd)	O(b^d)