Foundations of Artificial Intelligence

Lorenzo Bozzoni

November 2, 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

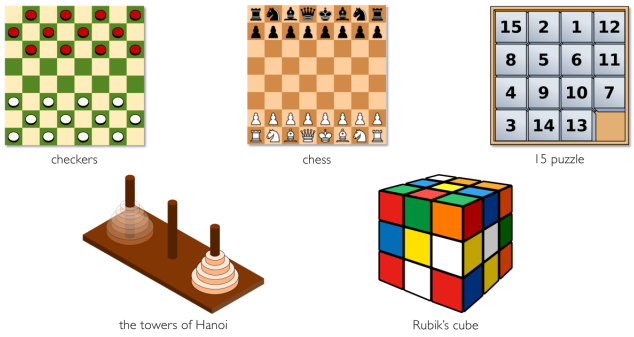
3 Problem solving by search

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

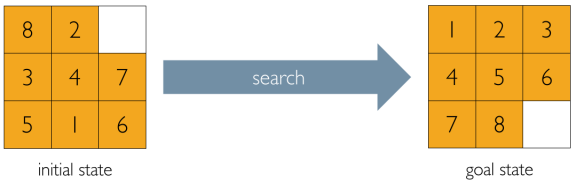
- Problem formulation
- Searching the solution
- Execute the solution

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

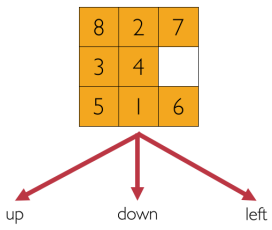


3.1.1 The states

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

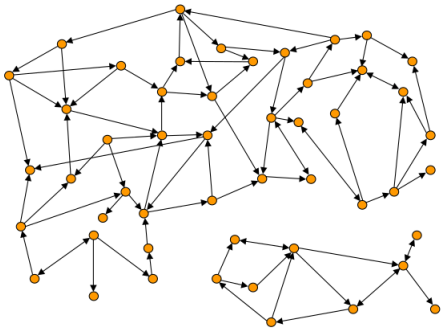
For the 8-puzzle, we can define a function result(s,a) that given a state s and an action a applicable in s, returns the state s' reached by executing a in s. State s' is a successor of s.

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.

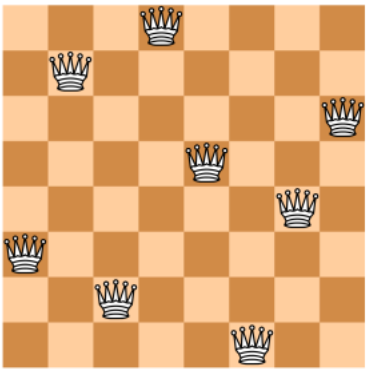
Search problems are typical of agents that operate in environments that are fully observable, static, discrete, and deterministic.

3.3 Abstraction

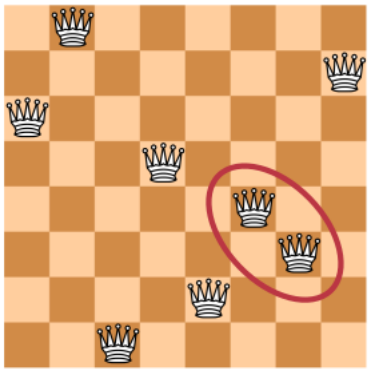
A state is an abstract representation of possible physical situations that share the same fundamental properties and differ in some details. A state of the 8-puzzle represents a set of possible physical situations that share the relative positions of the tiles but might differ in the colors of the tiles,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

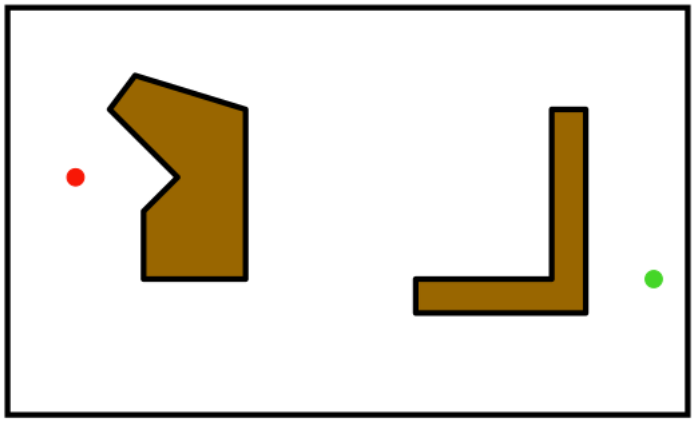
A second formulation could be:

- 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

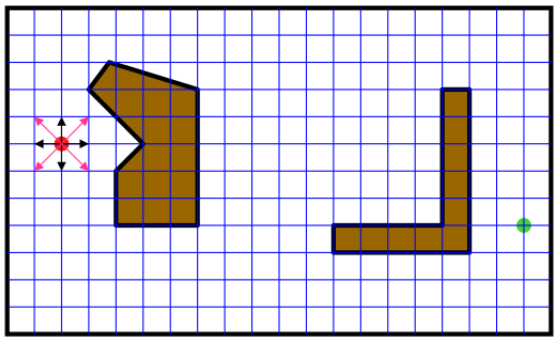
An example of path planning problem is the cleaner robot which, from a starting point, has to reach another point while avoiding obstacles.

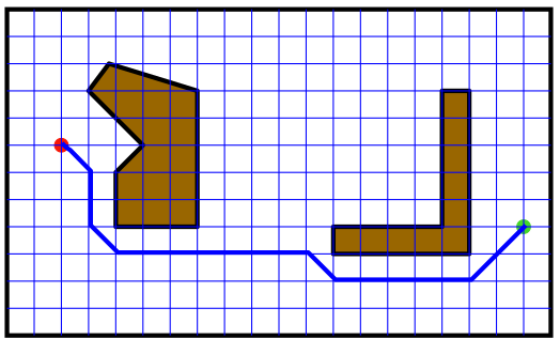


What are the possible states? How is the function actions(s) defined? And the function result(s,a)? And the goal test?

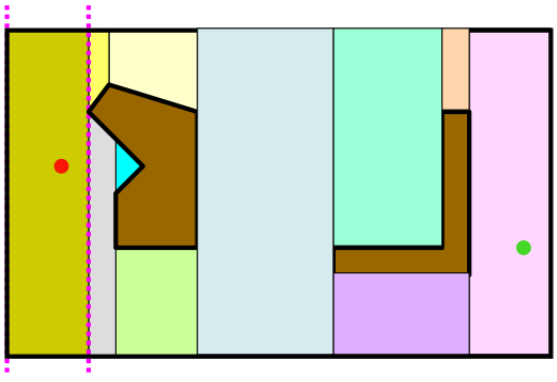
3.5.1 First formulation

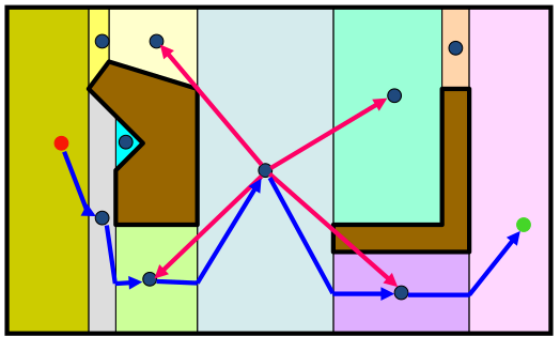
First formulation of path planning. The cost for a horizontal or vertical movement is1; 1.41 for a diagonal movement.



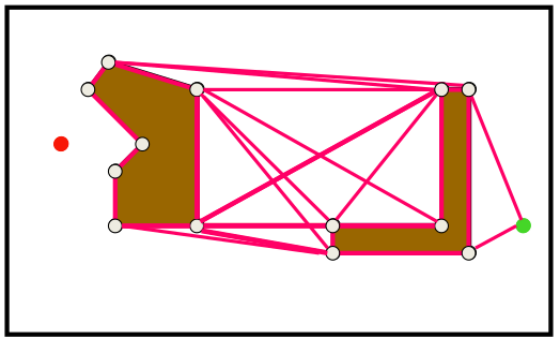


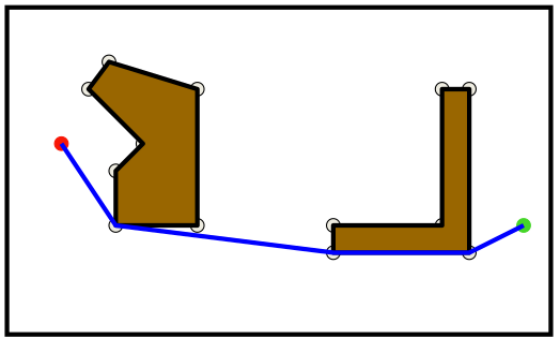
3.5.2 Second formulation





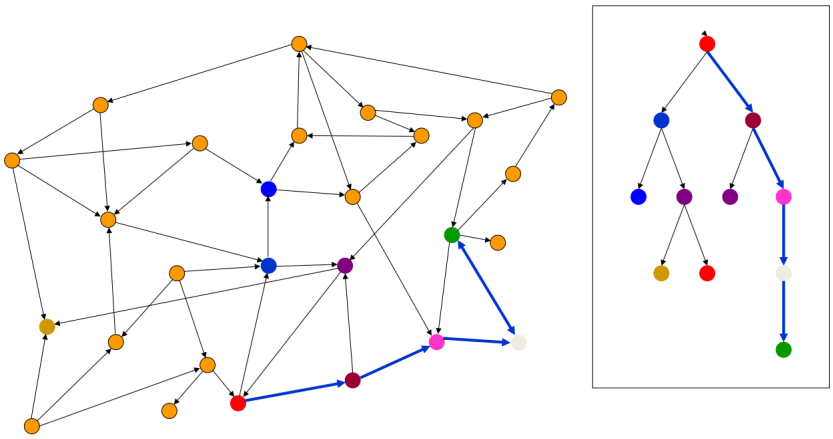
3.5.3 Third formulation





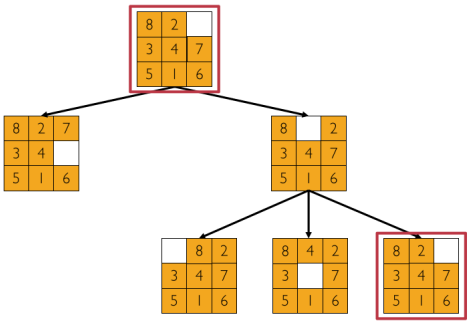
3.6 Searching for solution

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
```

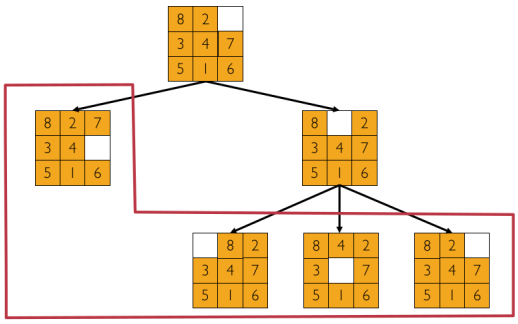
```
    self.action = action
    self.path_cost = path_cost
    self.depth = 0
    if parent:
        self.depth = parent.depth + 1
```

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:

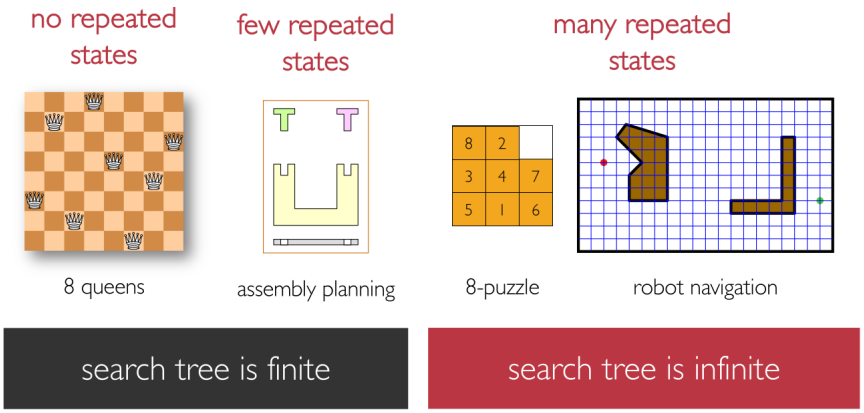
```
def expand(self, problem):
    """List the nodes reachable in one step from this node."""
    return [self.child_node(problem, action) for action in problem.actions(self.state)]
def child_node(self, problem, action):
    """[Figure 3.10]"""
    next_state = problem.result(self.state, action)
   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

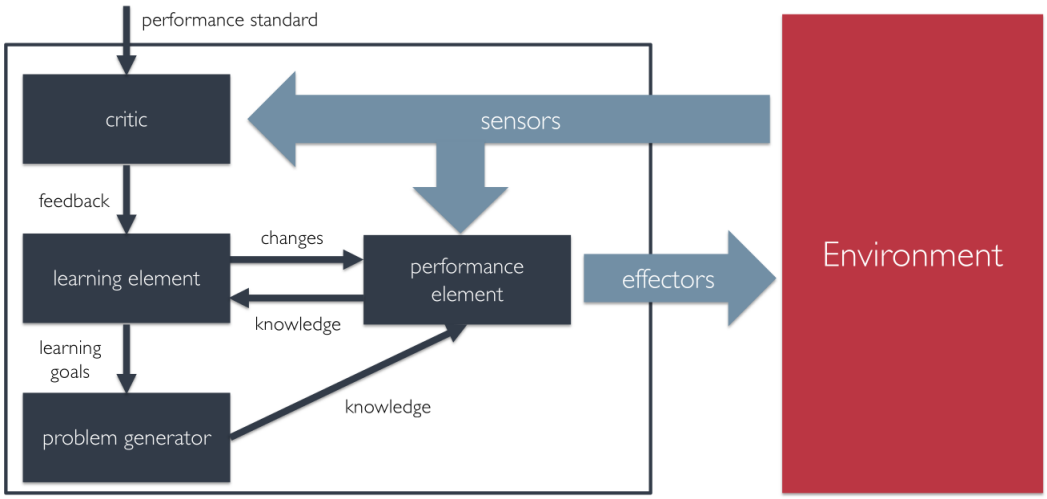
4 Learning Agents

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:

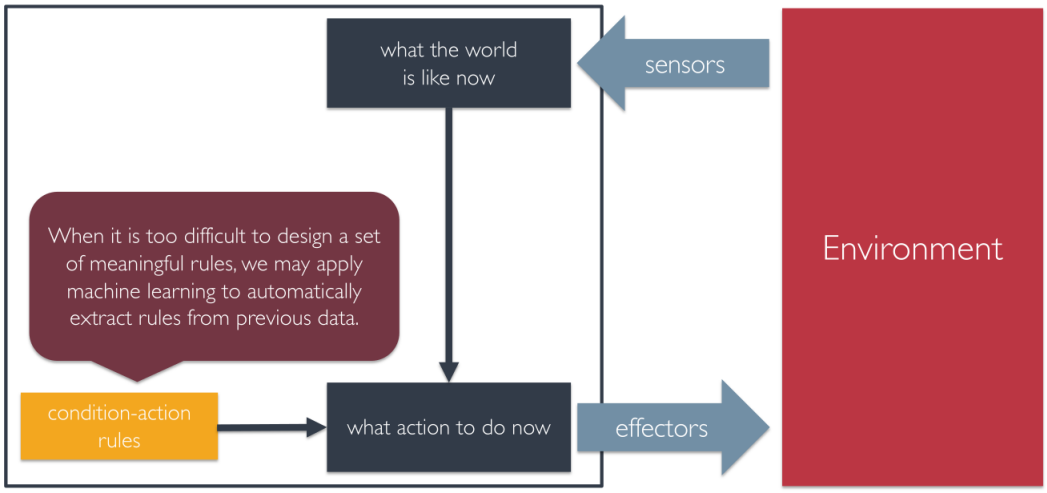
- Learning Element: it is responsible for making improvements.
- Performance Element: it selects external actions.
- Critic: it provides feedback on the agent is doing and determines how the performance element should be modified to improve future performance
- Problem Generator: it suggests exploratory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

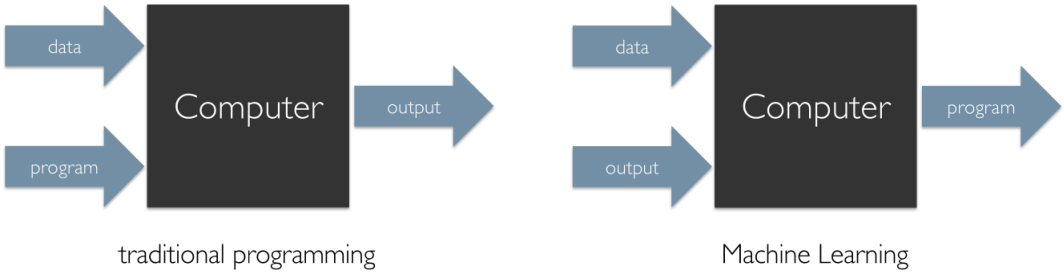
4.1 Machine learning

"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:

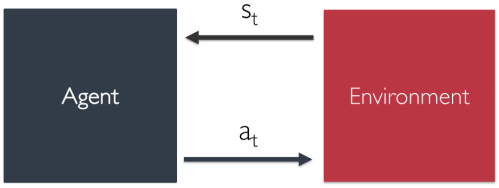


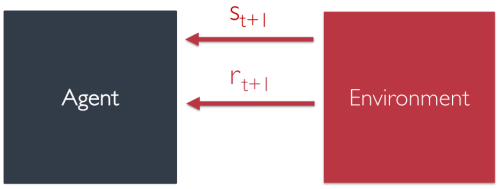
Suppose we have the experience we collected encoded as a dataset $D = x_1, \ldots, x_N$:

4.1.1 Reinforcement learning

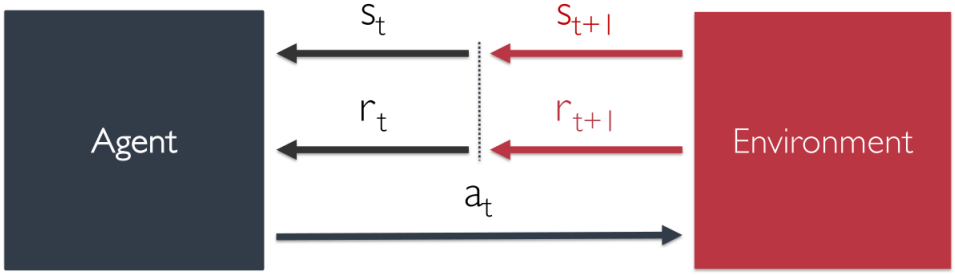
In sequential decision making:

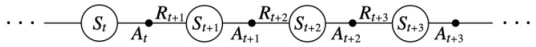
- we take a sequence of decisions (or actions) to reach the goal
- the optimal actions are context-dependent
- we generally do not have examples of correct actions
- actions may have long-term consequences
- short-term consequences of optimal actions might seem negative





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:





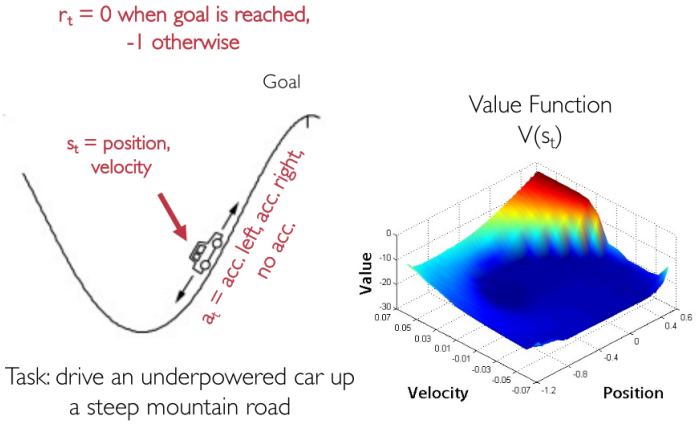
The agent's goal is to maximize the total amount of reward received How much future reward will it get when it performs a_t in s_t and then continues to do its best from there on? What is the expected payoff from s_t and a_t? The agent needs to compute an **action-value function** mapping state-action 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 $p(s_{t+1}, r_{t+1}|s_t, a_t)$.

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

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):

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

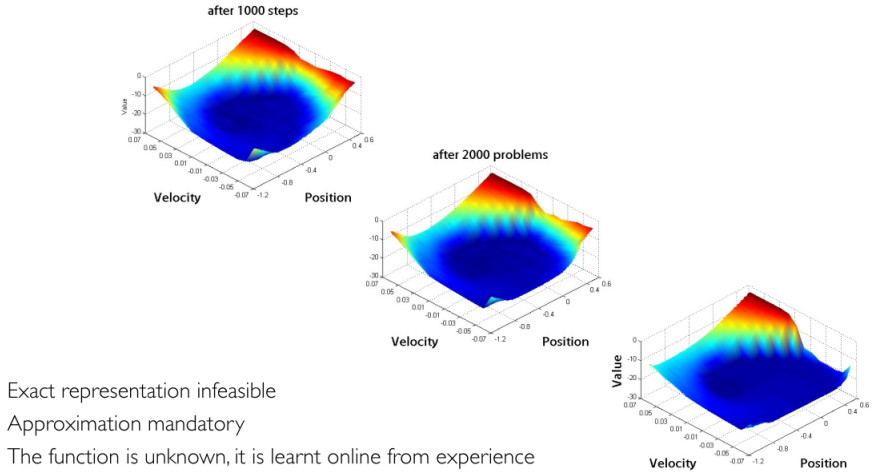
The action-value function can be similarly decomposed:

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

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

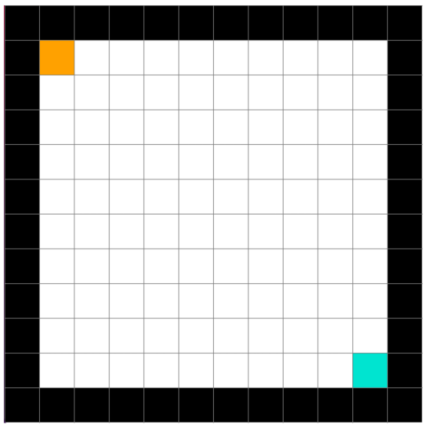
$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \beta \left[r_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1},a) - Q(s_t,a_t) \right]$$

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.

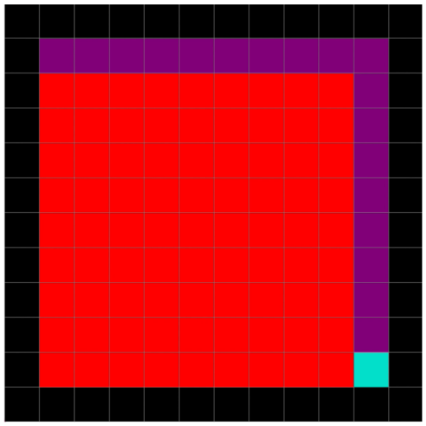


Tabular representation is infeasible in practice and approximators must be used for interesting problems. Reinforcement learning computes an unknown value function while also trying to approximate it. Approximator works on intermediate estimates while also providing information for the learning. Convergence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

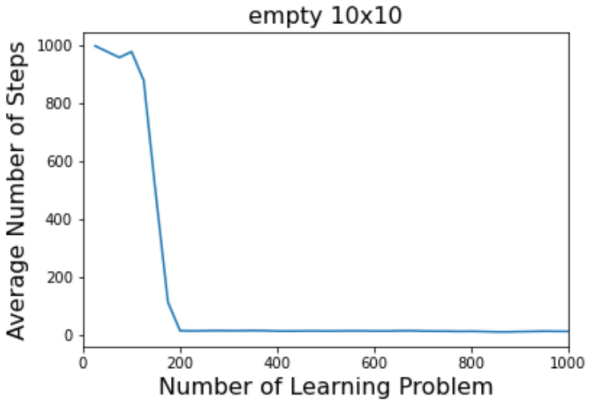
Let's now consider this simple empty environment with one start position (yellow) and one goal position (blue):



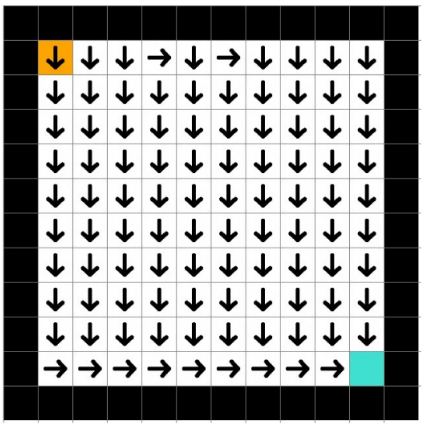
Here is the solution using a search algorithm:



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.

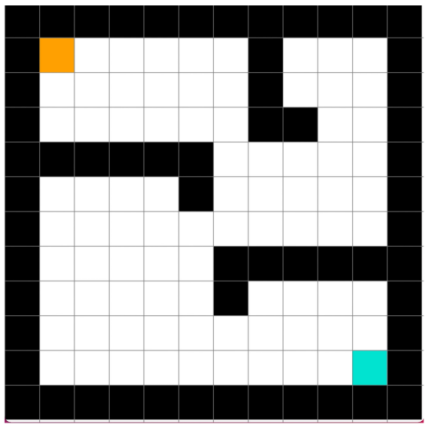


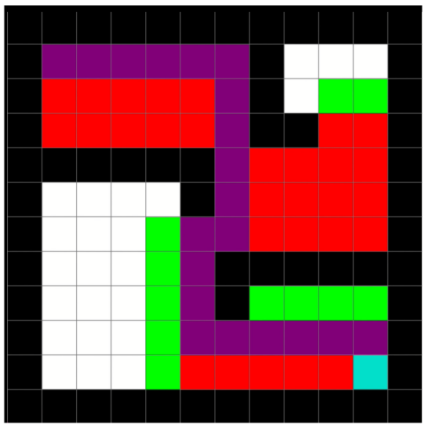
Search finds a solution, reinforcement learning an action value function. Here in the figure below is represented the best action for every position based on the action-value function on the left, while on the right the computed action value function:



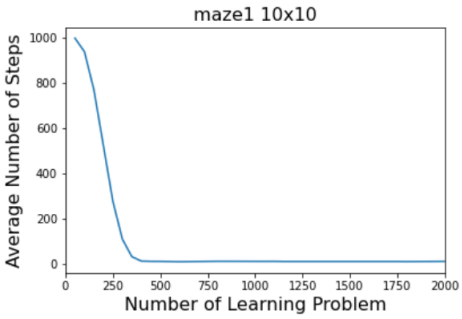
State	Up	Right	Down	Left
(1, 2)	0.397	0.440	0.440	0.000
(1, 3)	0.418	0.463	0.463	0.000
(1, 4)	0.440	0.488	0.488	0.000
(1, 5)	0.463	0.513	0.513	0.000
(1, 6)	0.488	0.540	0.540	0.000
(1, 7)	0.513	0.569	0.569	0.000
(1, 8)	0.540	0.599	0.599	0.000
(1, 9)	0.569	0.630	0.630	0.000
(1, 10)	0.599	0.663	0.000	0.000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

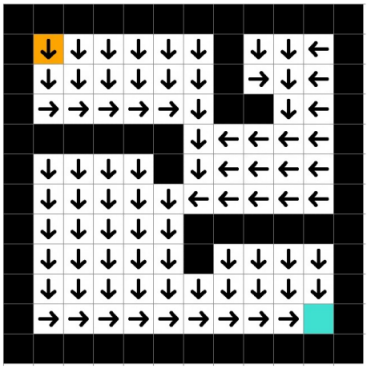
Let's consider another example. On the left is the new problem to solve while on the right there is the solution found by using the A* search algorithm:





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

Here is introduced the closed list in order to avoid visiting the same node multiple times.

```
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 nodes to the tree
```

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

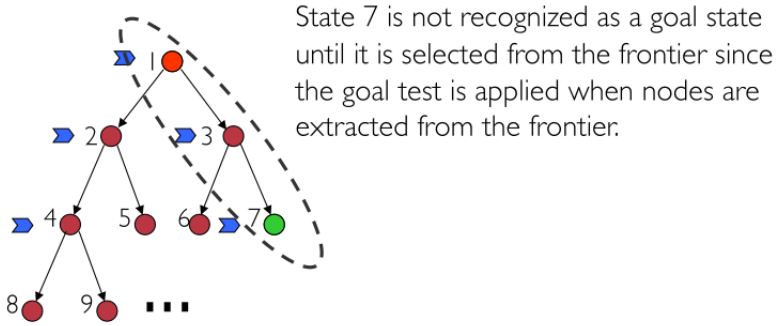
When the branching factor is large, the problem is difficult, the number of nodes will grow quickly. When the value of d is large the closest solution will require more exploration of the search tree. Sometimes, the depth d is fixed for any solution (like for example, in the eight-queen puzzle)

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 $k+1$.

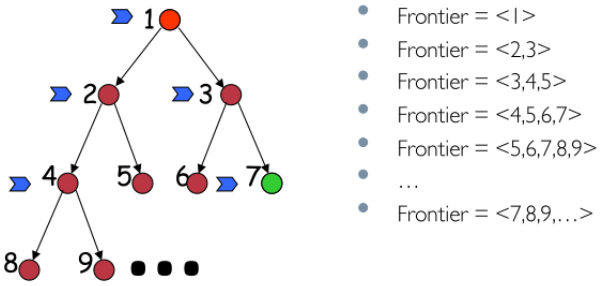
Breadth-first search always selects the node with minimum depth (the shallowest one). When more nodes are at the same depth, a tie-breaker is applied determined by the insertion order.

Example:



The node "2" is expanded before the node "3" so in a certain istant the frontier is: 3,4,5 as it is written also in the figure below. The node "7" is not immediately recognized as a goal, it is inserted in the frontier before. Once the goal node is reached, the path to the root is selected as solution.

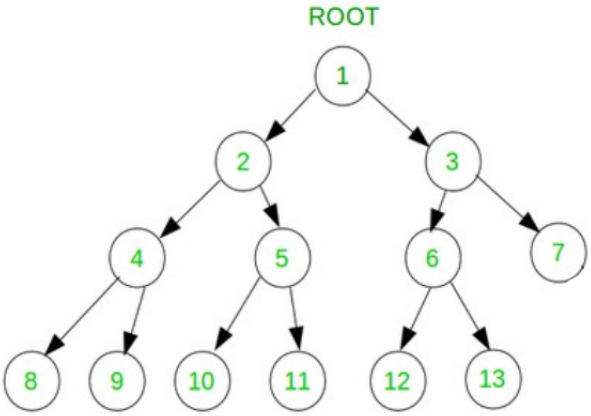
Breadth-first search can be implemented by considering the frontier a FIFO (First In First Out) queue. The new nodes are appended at the end of the frontier. Thus, the nodes generated first are selected first.



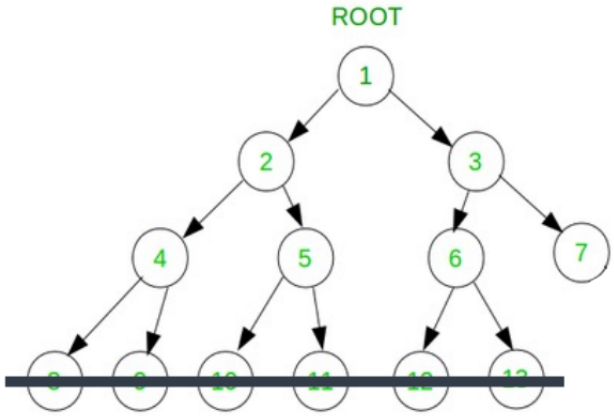
5.4.1 Evaluation of Breadth-First Search

- 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. The cost for the second level is greater than the first, and so on.
- 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.

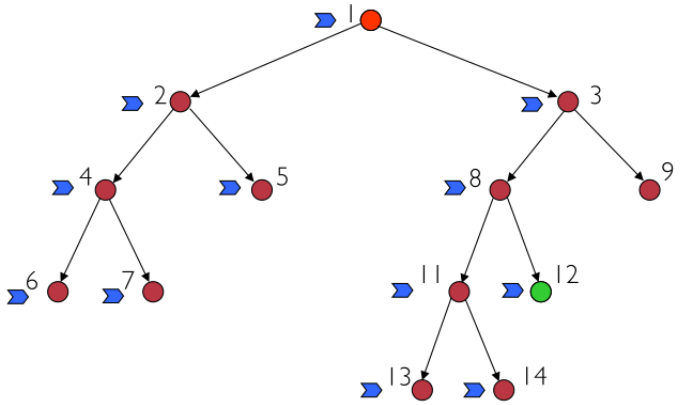
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

5.5.1 Evaluation of Uniform-Cost Search

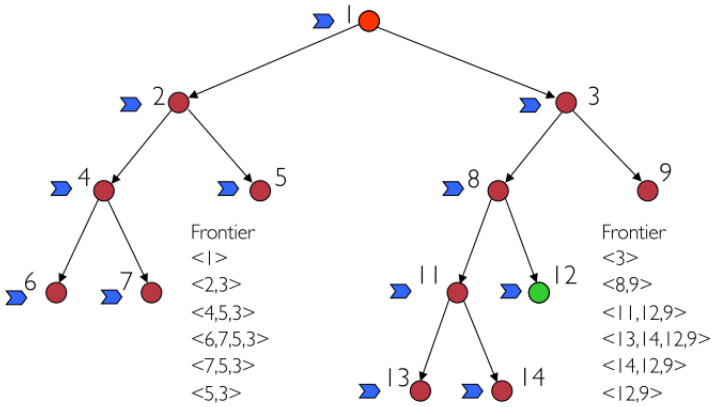
- Uniform-cost search is complete: assuming b finite and step costs larger than a small positive ϵ
- Uniform-cost search is optimal: uniform-cost search expands nodes in order of their optimal path costs
- Complexity: Uniform-cost search has temporal and spatial worst-case complexity of $O(b^{(1+\lfloor C^*/\epsilon \rfloor)})$ nodes, where C^* is the cost of the optimal solution. The fraction at the exponent represents the depth of the tree since ϵ is very small will always be choosen by the algorithm.

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. Since it is a stack, it can be implemented both explicitely by a data structure or implicitely by means of recursive calls. It is like a breadth-first search with LIFO instead of FIFO. 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, might go very deeply in a wrong direction.
- 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)
```

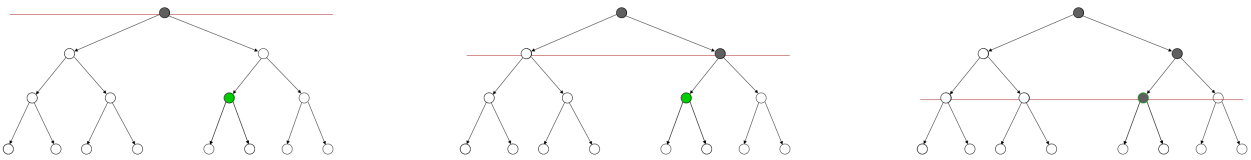
```
        if result != cutoff:
    return result
```

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:



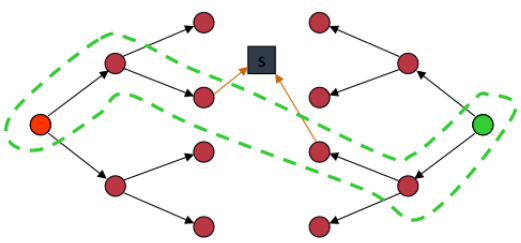
Every time the depth is increased, the entire tree is regenerated (optimization exists)

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	$O(bd)$	$O(b^d)$

6 Informed Search Strategies

They exploit specific knowledge that is not contained in problem formulation. They select a node from the frontier according to an evaluation function f(n). The evaluation function f(n) provides an estimate of how much a node is "promising". There are different ways to calculate the evaluation function, we will focus on: Greedy best-first search and A* search.

Conventionally, best (most promising) nodes have small values of evaluation function f(n). Informed search strategies implement the frontier as a priority queue ordered in increasing order of f(n). Thus, nodes with minimum f(n) are chosen first.

```
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**.

6.1 Best-first greedy search

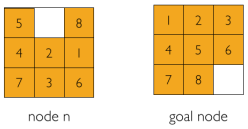
Best-first greedy search uses an evaluation function that is equal to a heuristic function h(n). Thus, the evaluation f(n) corresponds to the heuristic function h(n) that evaluates the estimated cost of the shortest path from a node n to a goal node. To apply best-first greedy search, h(n) must be known. For example, it is sufficient to know the heuristic value for each state.

The heuristic h(n) is an estimate, not the actual cost If we knew the actual cost, the problem would have already been solved.

6.1.1 Examples of best-first greedy search

Let's consider two examples of heuristics for the 8-puzzle:

- $h_1(n)$ is the number of misplaced tiles
- $h_2(n)$ is computed as the sum of Manhattan distances of each tile to its final position (computed only considering horizontal and vertical movements)



Considering the node N we have:

- $h_1(n) = 6$ because only 4 and 7 are in the right position
- $h_2(n) = 2$ (for tile 5) + 3 (for tile 8) + 0 (for tile 4) + 1 (for tile 2) + 3 (for tile 1) + 0 (for tile 7) + 3 (for tile 3) + 1 (for tile 6) = 13

What are the underlying intuitions behind $h_1(n)$ and $h_2(n)$?

- $h_1(n)$: if a tile is misplaced, I will need at least one move to place it in its correct position
- $h_2(n)$: i am counting how many moves each tile would require if no other tiles were placed on the board

$h_1(n)$ and $h_2(n)$ are defined considering simpler problems (that is, by relaxing some of the original constrains).

6.2 Heuristic function accuracy

Given two (consisten) heuristic funtions h_1 and h_2 such that $h_1(n) \leq h_2(n)$ fon any node n, h_2 **dominates** h_1 since each node expanded by A^* search using h_1 is also expanded by h_2. In the 8-puzzle h_2 dominates h_1. **When h_2 dominates h_1, h_2 we say it is more accurate or more informed than h_1.**

Best-first Greedy Tree Search python code (it still checks for cycles but has no reached table):

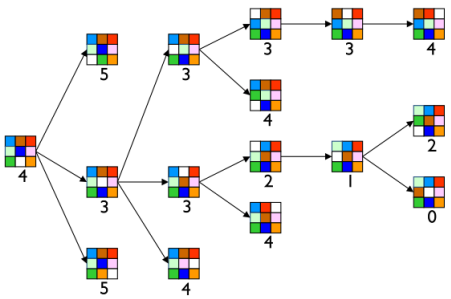
```
def best_first_tree_search(problem, f):
    "A version of best_first_search without the ‘reached‘ table."
    frontier = PriorityQueue([Node(problem.initial)], key=f)
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            if not is_cycle(child):
                frontier.add(child)
    return failure
```

If f(n)=g(n) then it is uninformed search algorithm since g(n) is a given data corresponding to the path cost from root to the node.

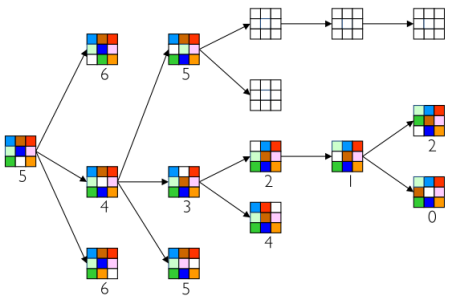
Best-first Greedy Search python code (with reached table)

```
def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    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 or child.path_cost < reached[s].path_cost:
            reached[s] = child
            frontier.add(child)
    return failure
```

Example of best-first greedy search using $h_1(n)$ and graph-search (because of repeated states):

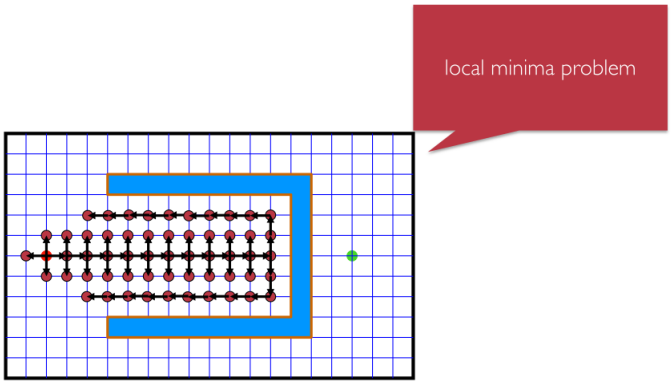


Example of best-first greedy search using $h_2(n)$ and graph-search (because of repeated states)



$h_1(n)$ estimated that we would need four moves to reach the goal $h_2(n)$ estimated that we needed five moves. They are both estimates, but some estimates might be more accurate than others (more later).

Another example of best-first greedy search. In this case $h(n)$ is computed as the straight distance from the goal. We apply graph-search since we might have repeated nodes.



6.2.1 Evaluation of Best-First Greedy Search

- Best-first greedy search is not complete: since it can get easily stuck in local optima (and cannot backtrack)
- Best-first greedy search, in general, is not optimal: for the same reason
- Complexity: it has temporal and spatial (worst-case) complexity of $O(b^m)$ nodes, m is the maximum depth of the search tree (and could be infinite)

6.3 A* search

The evaluation function $f(n)$ of a node n is computed as the sum of two components: the cost to reach n from the root $g(n)$ and an heuristic function $h(n)$.

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

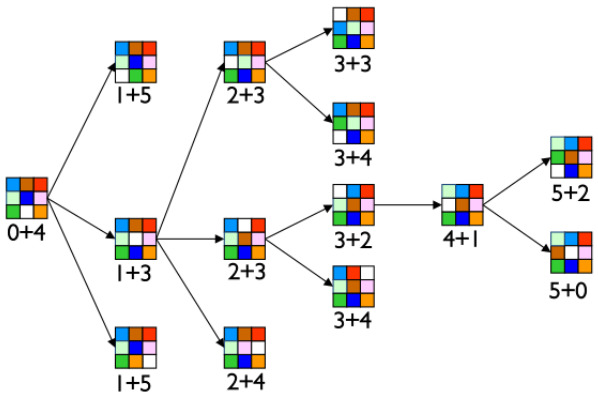
$f(n)$ estimates the cost of a solution that passes through node n:

- $g(n)$ is the path cost from the root to n (what we know for sure)
- $h(n)$ estimates the cost of the shortest path from n to a goal node (what we are guessing)

A* Seach python code (with reached table):

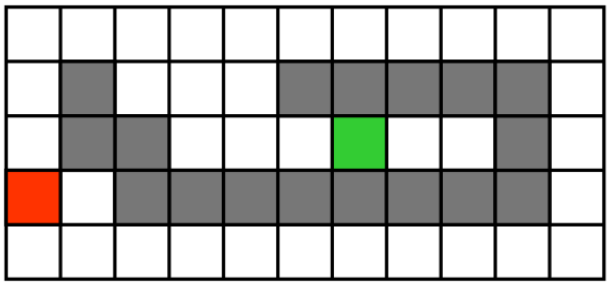
```
def g(n): return n.path_cost

def astar_search(problem, h=None):
    """Search nodes with minimum f(n) = g(n) + h(n)."""
    h = h or problem.h
    return best_first_search(problem, f=lambda n: g(n) + h(n))
```

We must know $h(n)$ to apply A* search. Example of A* search using graph-search. $h(n)$ is computed as the number of misplaced tiles:



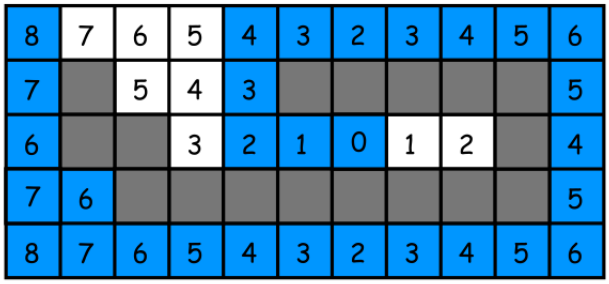
6.3.1 Advantages of A* search

Navigation example: the red square is the starting position, the green square is the goal, grey squares are obstacles. Best-first greedy search using h(n) computed as the Manhattan distance from the goal (without considering obstacles) on the right.

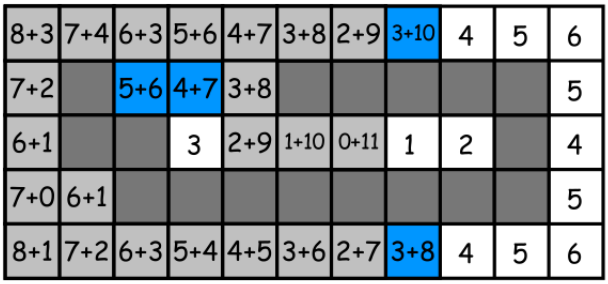


8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

The values of the distance can be stored or computed on the fly depending on the problem. Best-first greedy search using h(n) computed as the Manhattan distance from the goal (without considering obstacles) on the left while on the right A* search using h(n) computed as the Manhattan distance from the goal (without considering obstacles).



8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6



8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

By using both the cost so far g(n) and the estimate of the cost of reaching the goal h(n), A* can lead the agent away from suboptimal paths.

6.3.2 Evaluation of A* search

- A* search is complete and optimal for tree-search: when h(n) is admissible
- A* search is complete and optimal for graph-search: when h(n) is consistent
- Complexity: it has a temporal and spatial (worst-case) complexity that is exponential in the length of the solution

6.4 Admissible heuristic functions

A heuristic function h(n) is admissible when, for each node n

$$0 \leq h(n) \leq h^*(n)$$

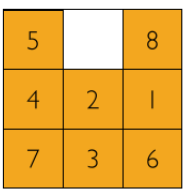
where $h*(n)$ represents the actual cost from node n to the solution. When n is a goal node, h(n) should be zero. h(n) is "optimistic" since it always underestimates the true cost to reach the goal.

Example

- h(n) computed as the estimated cost of the shortest path from n to a goal node
- h*(n) returning the actual cost of the shortest path from n to a goal node

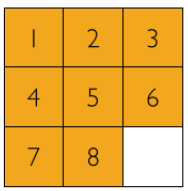
How can I define an admissible h(n) without knowing h*(n)?

Considering



5		8
4	2	1
7	3	6

node n

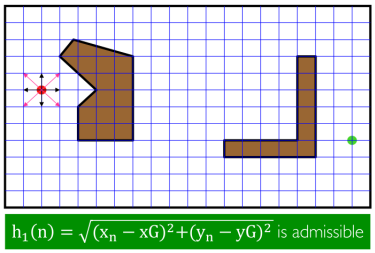


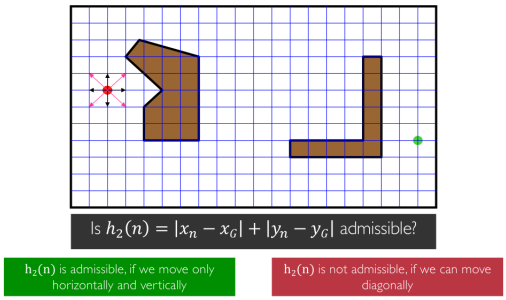
1	2	3
4	5	6
7	8	

goal node

Is h1(n) admissible? (the number of misplaced tiles). Is h2(n) admissible? (the sum of Manhattan distances of each tile to its final position).

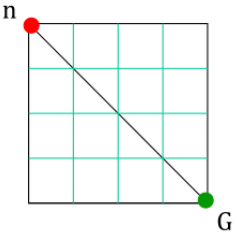
Step cost of horizontal/vertical actions is 1 and $\sqrt{2}$ for a diagonal actions. (x_G, y_G) is the goal position.





Why is $h_2(n)$ not admissible, when moving diagonally?

$h^*(n) = 4\sqrt{2}$
$h_2(n) = 8$



When moving diagonally, the actual $h^*(n)$ is 5.66 while $h_2(n)$ is 8 (thus, it overestimates the cost). How to create an admissible heuristic function? **An admissible heuristic can usually be seen as the cost of an optimal solution to a relaxed problem, obtained by removing constraints.**

In the robot navigation example,

- The Manhattan distance corresponds to the optimal solution when we remove the obstacles but keep the constraints of moving in four directions
- The Euclidean distance corresponds to the optimal solution when we remove both the obstacles and allow diagonal movements

In the 8-puzzle,

- $h_1(n)$ was based on removing the constraint of moving only by sliding the tiles
- $h_2(n)$ considered the moves needed to position each tile as if it was alone on the board

6.5 First optimality theorem for A^* search

A^* seach with tree-search is complete and optimal when $h(n)$ is admissible (when all step costs are larger than a small positive ϵ).

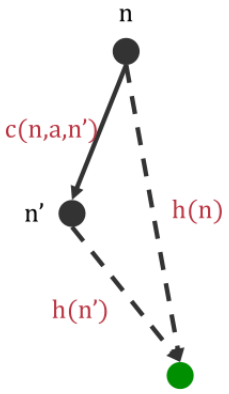
Completeness: if a solution exists, it will have a finite cost C^*. Since the step costs are at least ϵ, then $g(n)$ will always increase and exceed the cost of the solution. Thus, A^* will expand forever the tree. **Thus, if a solution exists, A^* search terminates with the solution**.

Optimality: call C^* the cost of the optimal solution. Consider a sub-optimal goal node G' in the frontier $f(G') = g(n) + h(n) \leq C^*$. Consider a node n in the frontier, which is on the path corresponding to the optimal solution; thus, $f(n) = g(n) + h(n) \leq C^*$. From the two equations we derive that $f(n) \leq C^* \leq f(G')$ and thus the suboptimal goal G' will not be chosen from the frontier. **Thus, when A^* search selects a node from the frontier; the path to the corresponding state is optimal**.

6.6 Consistent Heuristic Functions

A heuristic function $h(n)$ is consistent when, for each node n and each one of its successors n':

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



(triangular inequality)

A consistent heuristic function is also admissible; vice versa is not true in general. If n is a goal node, then $h(n) = 0$.

When is a heuristic function not consistent?

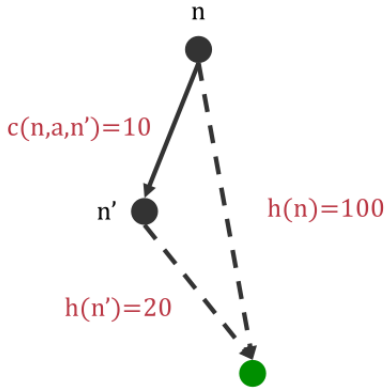
Consider a heuristic function for which:

- the estimated cost to the goal node from n is 100
- the cost of moving from n to n' is 10
- the estimated cost to the goal node from n' is 20

In this case, the triangular inequality does not hold:

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

$$(100 \not\leq 10 + 20)$$



(triangular inequality)

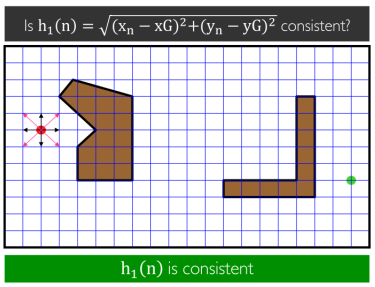
Let's consider two examples of heuristics for the 8-puzzle:

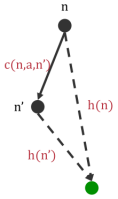
- $h_1(n)$ is the number of misplaced tiles
- $h_2(n)$ is computed as the sum of Manhattan distances of each tile to its final position (computed only considering horizontal and vertical movements)

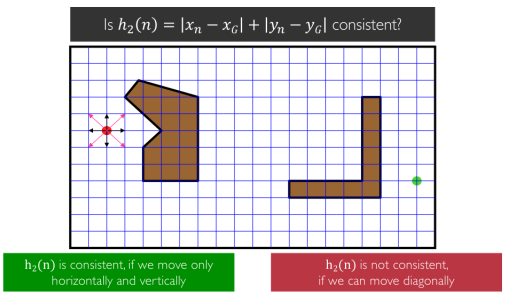
Considering the node N we have:

- $h_1(n) = 6$ because only 4 and 7 are in the right position
- $h_2(n) = 2$ (for tile 5) + 3 (for tile 8) + 0 (for tile 4) + 1 (for tile 2) + 3 (for tile 1) + 0 (for tile 7) + 3 (for tile 3) + 1 (for tile 6) = 13

Both heuristics are consistent. Now let's get back on the path problem with obstacles:







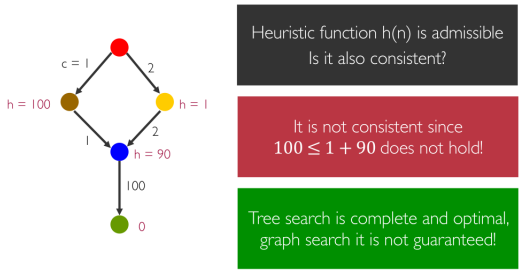
6.7 Second optimality theorem for A^* search

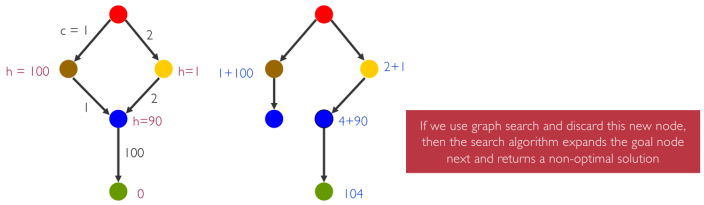
$A*$ search with graph-search is complete and optimal when $h(n)$ is consistent (when all step costs are larger than a small positive ϵ).

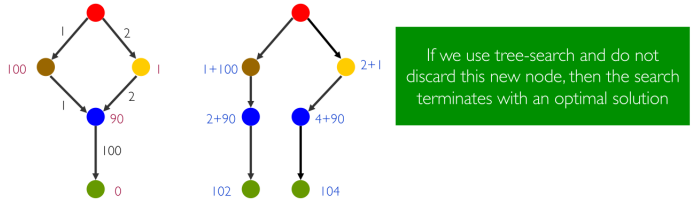
Completeness: it can be proved that in the same way we did for tree search with an admissible $h(n)$.

Optimality: given n' successor of n, $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$. Thus, A^* search chooses (and expands) nodes in non-decreasing order of $f(n)$. When A^* selects a node from the frontier, the current path to the corresponding state is optima. Thus, when A^* selects the first goal node from the frontier, this is the optimal solution.

Consistency is a stronger property than admissibility. In fact, there are more admissible than consistent functions Graph-search is more constrained since it does not consider nodes corresponding to the same state. Thus, it requires a stronger property to guarantee completeness and optimality.







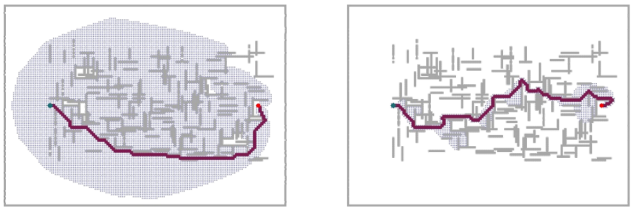
Other properties of A^* search:

- A^* search expands all nodes with $f(n) < C^*$
- A^* search expands some nodes with $f(n) = C^*$
- A^* search expands no nodes with $f(n) > C^*$
- A^* search is **optimally efficient**: there exists no other optimal search algorithm that, for any heuristic function, expands less nodes than A^* search (except some nodes with $f(n) = C^*$)

So, if we have a consistent heuristic, we are ready to apply A^* search, correct? Not really... There are very dumb consistent heuristic functions The heuristic $h_0(n)$ that returns 0 for all nodes n is consistent and thus admissible, but uninteresting. A* search with $h_0(n)$ implements uniform-cost search. Uniform-cost search is thus a special case of A* search.

6.8 Weighted A*

It introduces a weight factor $w(1 \leq w < \infty)$ over the heuristics so that, f$(n) = g(n) + wh(n)$. The theoretical properties do not apply anymore, but this extension of A* can be more efficient. A* search (left) compared to weighted A* search with $w = 2$ (right):

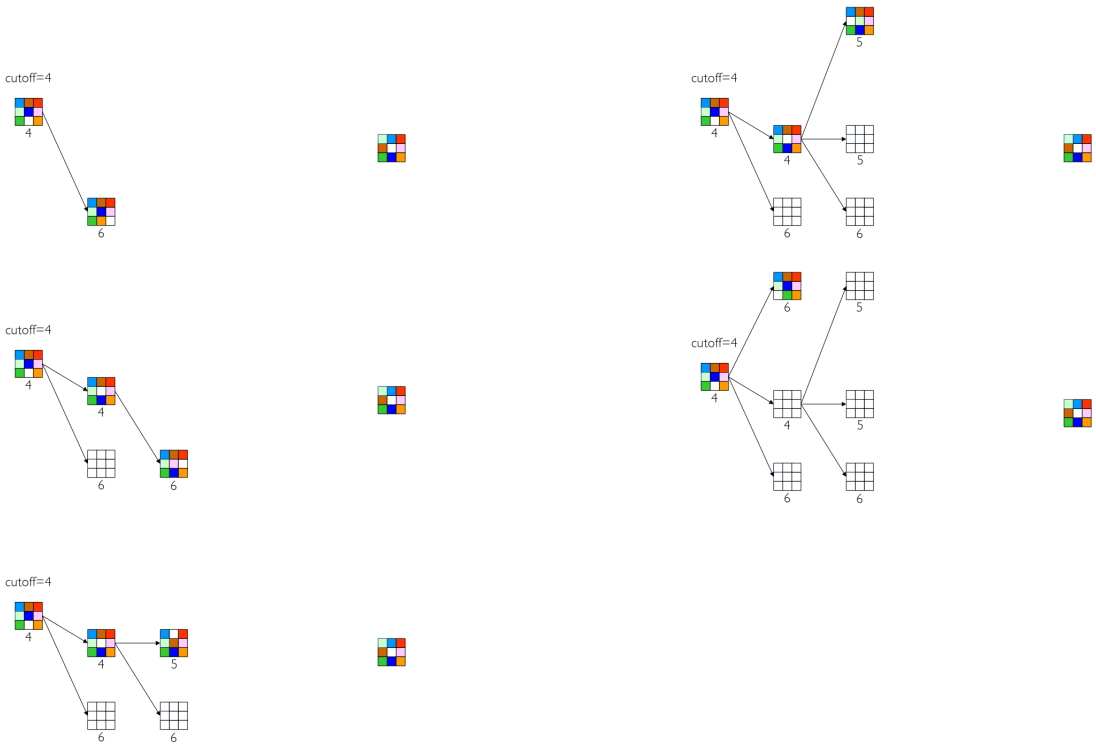


6.9 Iterative deepening A* (IDA*) search

IDA* reduces memory requirement of A* search by applying a limit to the values of f(n). It assumes to have a consistent heuristic function $h(n)$. Iterative Deepening A* (IDA*) Search:

```
cutoff = f(root-node)
repeat
    1. perform depth-first search by expanding all nodes n such that f(n) <= cutoff
    2. cutoff = smallest value f() of non-expanded (leaf) nodes until
    (a solution is found) or (a termination condition on time or memory is met)
```

Example of IDA* search. 8-puzzle with $h(n)$ computed as the number of misplaced tiles and cutoff equal to 4:



Cutoff set to 5:

6.9.1 Evaluation of IDA* search

- Completeness: IDA* search is complete when $h(n)$ is admissible
- Optimality: IDA* search is optimal when $h(n)$ is admissible
- Complexity: IDA* search requires less memory than A* search and avoids to sort the frontier. However, IDA* cannot avoid to revisit states not on the current path, because it uses too little memory (it just remembers the current cutoff). Thus, it is very efficient memory-wise but not time-wise since it revisits states
- Several other variants of A* search exist and are widely used

7 Adversarial search strategies

Adversarial search considers multiagent environments in which agents compete. Board games are a typical scenario for adversarial search strategies.

We consider games with:

- two players
- turn-taking
- zero sum
- perfect information
- deterministic and stochastic (they include elements of chance)

Zero sum means that if a player wins (+1) the other player loses (-1). Thus sum, thus, is zero. Perfect information means that the agent has perfect information about the game state which is fully observable (like chess, unlike poker).

Games can be formulated as search problems with an element of uncertainty due to the action (moves) of the oppponent. In general, there is no sequence of actions that makes a player win independently of the actions of the opponent. At each turn, a player must choose the action to play without the possibility to explore the whole state space.

7.1 Formulation of search problem for a game

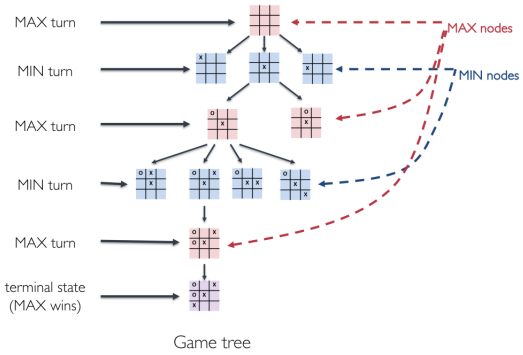
- **Players**: two players, MAX and MIN, that take turn. MAX moves first
- **Initial state**: intial set up of the game (initial configuration of the board)
- **Function Actions(s)**: given a state s, returns the set of legal moves in s
- **Function Result(s,a)**: defines the state resulting from taking action a in state s
- **Goal test**: given a state, returns true if the game is over
- **Utility(s,p)**: returns the numerical value to player p in terminal state s (e.g., +1, 0, or -1)

Python formulation:

```
    class Game:
def actions(self, state):
"""Return a collection of the allowable moves from this state."""
raise NotImplementedError
def result(self, state, move):
"""Return the state that results from making a move from a state."""
raise NotImplementedError
def is_terminal(self, state):
"""Return True if this is a final state for the game."""
return not self.actions(state)
def utility(self, state, player):
"""Return the value of this final state to player."""
raise NotImplementedError
```

Playing a game

```
def play_game(game, strategies: dict, verbose=False):
    """Play a turn-taking game. ‘strategies‘ is a {player_name: function} dict,
    where function(state, game) is used to get the player’s move."""
    state = game.initial
    while not game.is_terminal(state):
    player = state.to_move
    move = strategies[player](game, state)
    state = game.result(state, move)
    if verbose:
    print(’Player’, player, ’move:’, move)
    print(state)
    return state
```



Game tree

Solving the game corresponds to finding the best action for MAX (or for MIN) in a state.

7.2 Action selection - Minimax Search

It considers the game from MAX's perspective. Starting from the current state (corresponding to the root), MAX builds the game tree. The game tree can be built completely or up to a maximum depth h (which could depend on the available time to perform a move). The game tree is built using a depth-first search. MAX evaluates the leaf nodes of the game tree using Utility(s) or by an evaluation function e(s). MAX "backs up" the minimax values from the leaves to the root. MAX chooses the best move (the move that maximizes the minimax value).

7.3 The evaluation function: e(s)

When the state is not terminal, we cannot compute utility. In non-terminal state, we can compute an evaluation function e(s) that, given a state s and a player p, returns a real number estimating the expected utility of s to p. Evaluation function:

- $e(s) > 0$ means that s is good for MAX
- $e(s) < 0$ means that s is good for MIN
- $e(s) = 0$ means that s is neutral

7.4 The utility function

The Utility function returns the numerical value to player p in terminal state s (e.g., +1, 0, or -1) Utility:

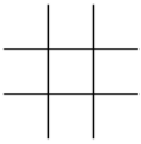
- Utility$(s) > 0$ means that MAX wins
- Utility$(s) < 0$ means that MIN wins
- Utility$(s) = 0$ means a tie

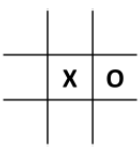
Convention Utility(s) and e(s) return values relative to MAX. The values for MIN are obtained by taking the opposite (i.e., multiplying by -1) because the games are zero sum. Utility(s) and e(s) are positive when they are good for MAX, negative when they are good for MIN.

Example of evaluation function for Tic-Tac-Toe e(s) is computed as the difference between:

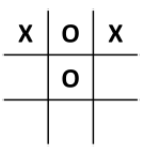
- The number of rows, columns and diagonals that are open for MAX and
- The number of rows, columns, and diagonals open for MIN

e(s) could be normalized between -1 and +1 to keep values consistent with utility.


e(s) = 8 – 8 = 0



e(s) = 6 – 4 = 2



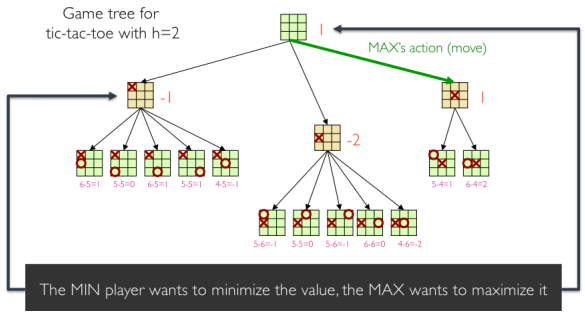
e(s) = 3 – 3 = 0

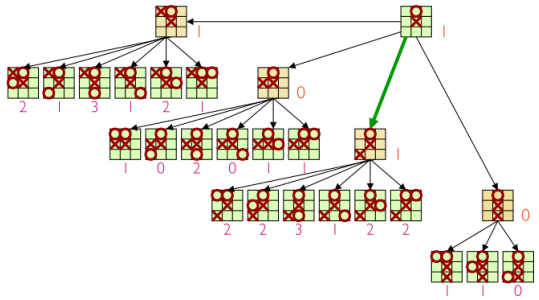
Starting from the values assigned (by function Utility() or by the evaluation function) to the leaves and going up to the root, a minimax value is assigned to each node of the game tree. The minimax value of a node is the estimate of the expected utility for a player in the state corresponding to that node, if both players play optimally. The minimax value of a leaf node is equal to the value returned by:

- The function Utility(s), if the leaf node is terminal
- The evaluation function e(s), if the leaf node is not terminal

The minimax value of a MAX node is the maximum of the minimax values of its children. The minimax value of a MIN node is the minimum of the minimax values of its children.

Examples of Minimax Search:

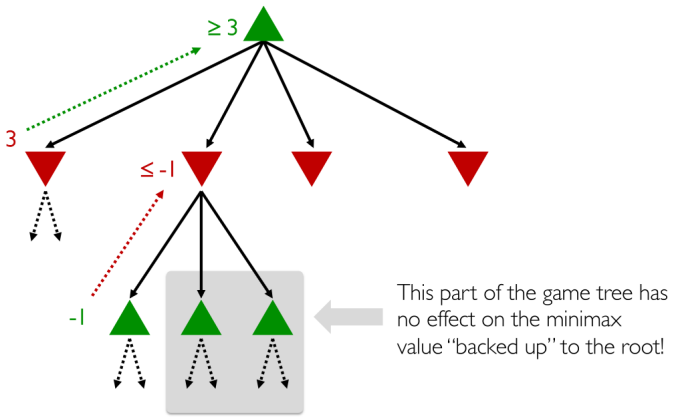




Minimax search python implementation:

```
def minimax_search(game, state):
    """Search game tree to determine best move; return (value, move) pair."""
    player = state.to_move
    def max_value(state):
        if game.is_terminal(state):
            return game.utility(state, player), None
        v, move = -infinity, None
        for a in game.actions(state):
            v2, _ = min_value(game.result(state, a))
            if v2 > v:
                v, move = v2, a
    return v, move

def min_value(state):
    if game.is_terminal(state):
        return game.utility(state, player), None
    v, move = +infinity, None
    for a in game.actions(state):
        v2, _ = max_value(game.result(state, a))
        if v2 < v:
            v, move = v2, a
    return v, move
    # the Minimax search plays for MAX
    return max_value(state)
```

Minimax is optimal when using the utility function (when the entire game tree is built). However, Minimax search builds a game tree in which nodes grow exponentially with the maximum depth h. Can MAX choose the optimal action without examining all the nodes in the game tree?

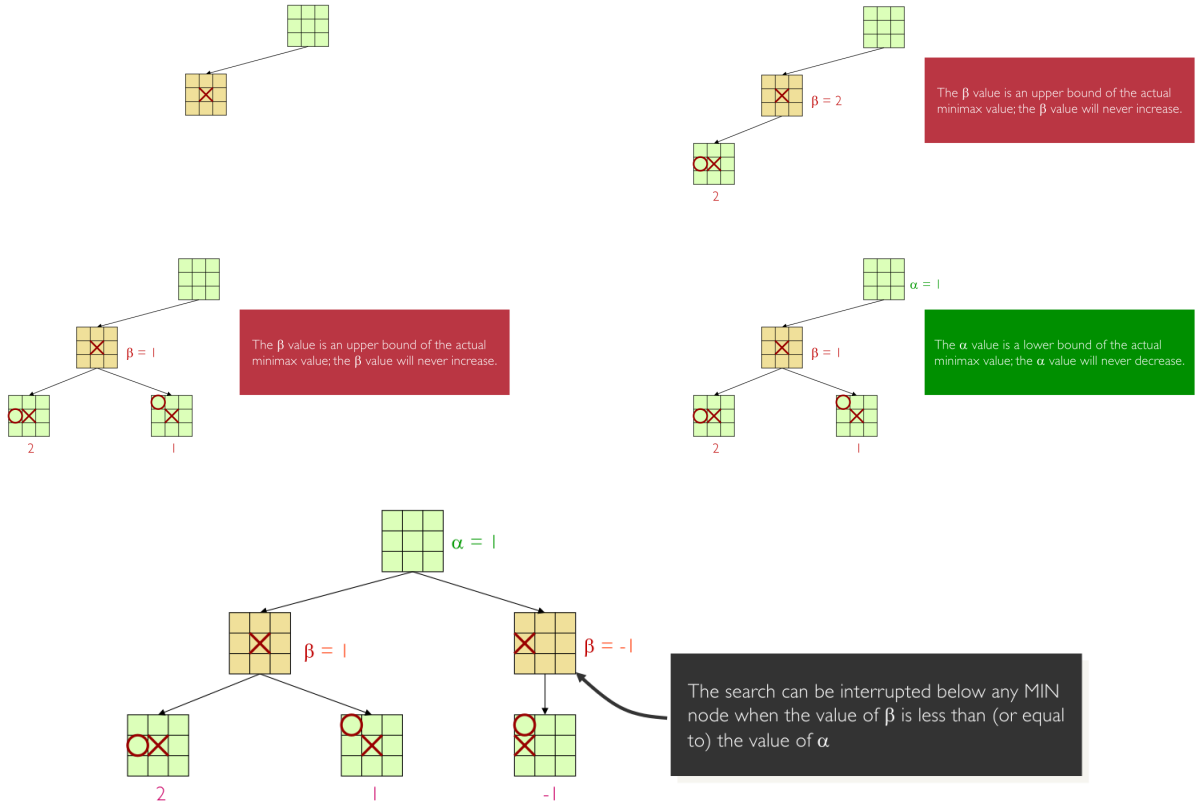


7.5 Alpha-Beta Pruning

Reduces the complexity of the minimax search by not considering branches of the game tree that cannot influence the final decision. Alpha-beta pruning returns the same outcome of the "vanilla" minimax search. Alpha-beta pruning uses two parameters α and β:

- Parameter α: it represents the value of the best move found so far for MAX (i.e. with the highest minimax value). α can be interpreted as "at least"
- Parameter β: it represents the value of the best move found so far for MIN (i.e. with the lowest minimax value). β can be interpreted as "at most"

Alpha-beta pruning for Tic-Tac-Toe



Here is a sketch of the algorithm of Alpha-Beta pruning:

1. Update the α and β values as soon as possible
2. Interrupt the search below a MAX node when the α value is larger or equal to the β value
3. Interrupt the search below a MIN node when the β value is smaller or equal to the α value

7.5.1 Evaluation of Minimax Search and Alpha-Beta pruning

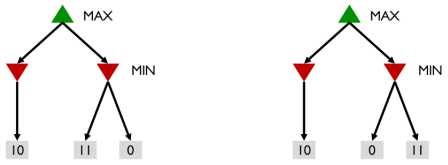
Given the maximum number of legal moves in a state is b, minimax search will expand $O(b^h)$ nodes. The efficiency of alpha-beta pruning depends on the order in which nodes are examined. In the worst case, alpha-beta pruning expands O(b^h) nodes.

In the best case, alpha-beta pruning expands $O(b^{h/2})$ nodes

- MIN nodes that are children of a MAX node should be considered in increasing order of minimax values
- MAX nodes that are children of a MIN node should be considered in decreasing order of minimax values
- This cannot be guaranteed in practice

In the average case (nodes considered in random order), alpha-beta prunig expands $O(b^{3h/4})$ nodes.

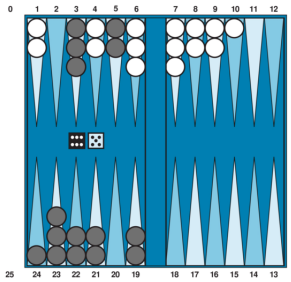
Move ordering example:



Move ordering example.

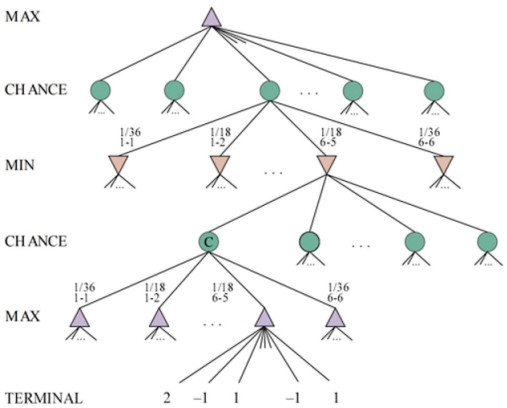
8 Stochastic games

Many games include unpredictable stochastic events, like throwing of dice in backgammon. We can apply an approach like minimax search using a game tree with chance nodes and expectiminimax values.

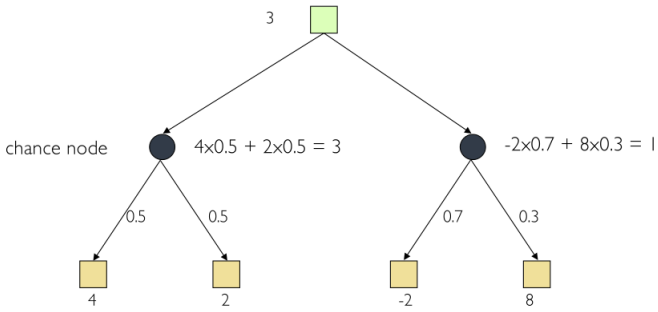


Black has rolled and must choose among four legal moves: (5-11,5-10), (5-11,19-24), (5-10,10-16), and (5-11,11-16), where the notation (5-11,11-16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

A typical backgammon position. The goal of the game is to move all one's pieces off the board. Here is a schematic game tree for a backgammon position:

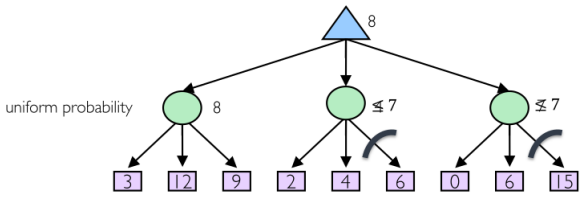


Stochastic games example for chance nodes with only two possible outcomes:

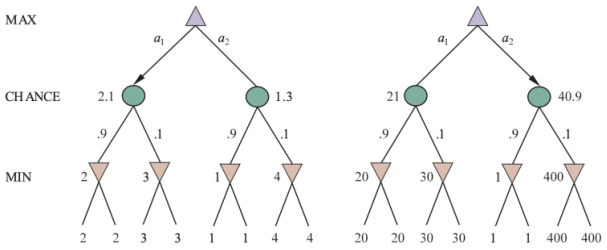


8.1 Alpha-Beta pruning for stochastic games

Alpha-beta pruning can be applied to stochastic games. Consider the following game tree:



What happens if you don't know the range of Utility() and e()? What happens if, instead, you know the range [0, 15]? An order-preserving transformation of leaves values can change the best move at the root. Do values matter also in deterministic games (minimax)? Why?



9 Monte Carlo Tree Search

Alpha-Beta pruning is effective in many games, however it cannot play some games, like Go. Why Go is difficult?

- Its branching factor is very large
- The game comprises more than 200 moves (about 250 moves on average)
- Order of magnitude greater than the branching factor of 20 for Chess
- It lacks a good evaluation function
- It is difficult to model, similar looking positions can have completely different outcomes

Alpha-beta pruning reached a proficiency level weak to intermediate.

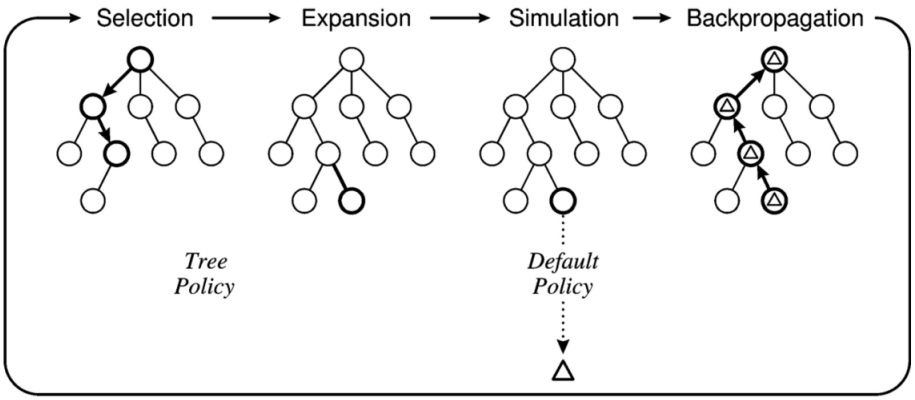
Monte Carlo Tree Search (MCTS) builds a "sparse" game tree to choose the best move at the root. Monte Carlo for random simulation MCTS runs random simulations and builds a search tree from the results. Markovian Decision Problem (MDP) with sequence of decisions, problem are phrased as state, action pairs.

Monte Carlo tree search has different phases:

1. Selection: it descends the game tree selecting successors using a selection policy until either a node with an unexpanded branch or a leaf is reached
2. Expansion: if the node is not a goal, the tree is expanded by generating one or more children nodes.

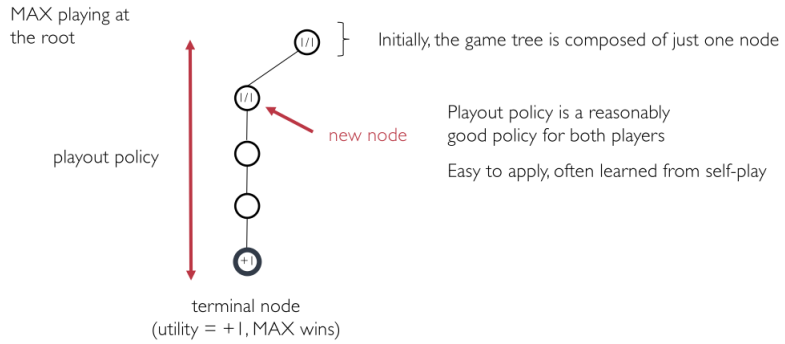
3. Simulation: a simulation (a playout or roll-out) is run from the current node to evaluate the game outcome by following a default playout policy (typically random); This results in a reward computed according to a reward function.

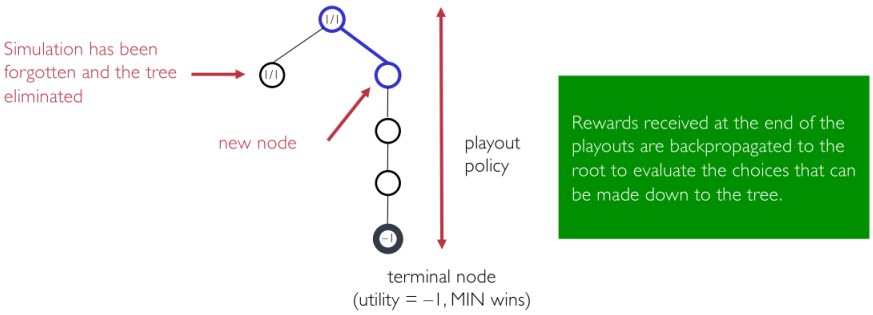
4. Backpropagation: the reward is used to update all the nodes visited during the previous steps.

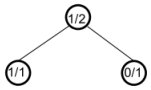


MCTS builds a "sparse" game tree by sampling the game tree focusing on the most promising paths. MCTS have more difficulty in problems in which sampling might not be effective. For example, problems with very few good moves with respect of the number of available ones.

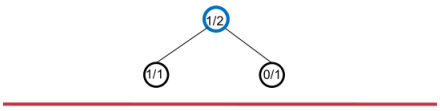
When does MCTS stop? It is an **anytime method**. Whenever it stops, it can provide an answer. Typically, it can be stopped when a time limit has been reached or a number of rollouts have been performed.

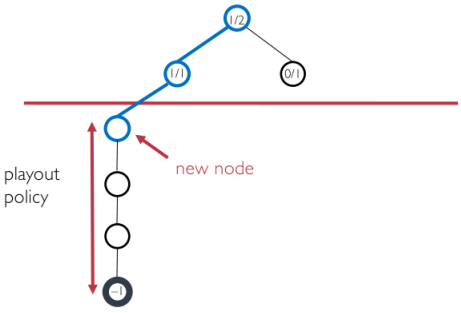


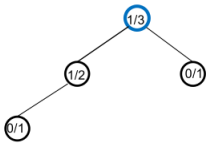


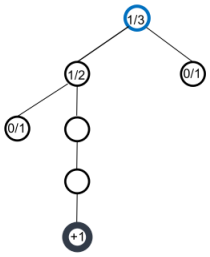


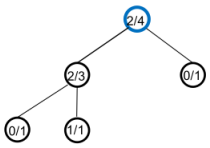
- Each node n stores a pair of values
  - Number of wins for the player at the root
  - Number of performed simulations that involve n
- Alternatively, each node n stores a pair of values
  - Number of wins for the player at the node
  - Number of performed simulations that involve n
- In general, the utilities $U(\mathbf{n})$ are stored in nodes











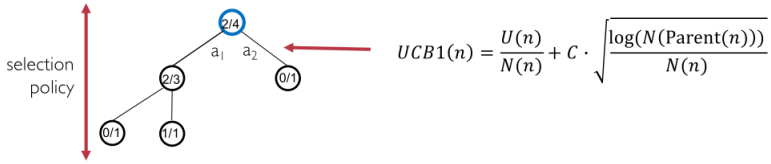
How do we select the successors nodes? Selection policy should balance exploration and exploitation. I wish to visit nodes that promise the highest chance of winning. I wish to try less explored nodes to learn what might happen if I take those actions. **UCB1(n)** is a common solution to balance exploration and exploitation in MCTS.

$$UCB1(n) = \frac{U(n)}{N(n)} + C \cdot \sqrt{\frac{\log(N(Parent(n)))}{N(n)}}$$

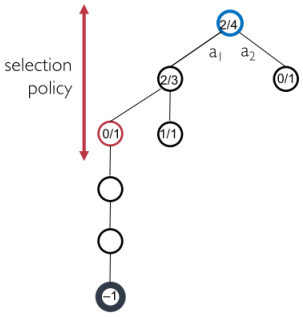
The node with the largest UCB1(n) is selected. Where:

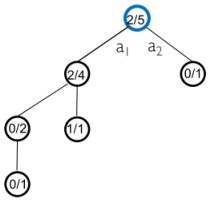
- $U(n)$ is the utility of node n (number of wins)
- $N(n)$ is the number of simulations involving node n
- C is the exploration parameter
- $N(Parent(n))$ is the number of simulations involving the parent of node n

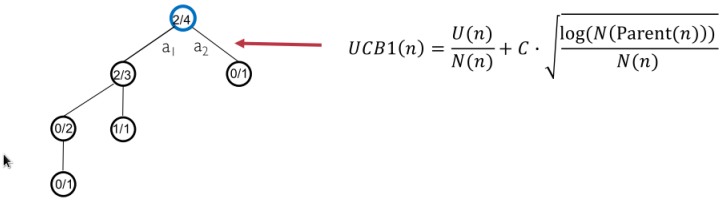
And, in general, the first term is the exploitation term and the second term is the exploration term. When C is infinity, then we have pure exploration. When C is zero, we have pure exploitation.

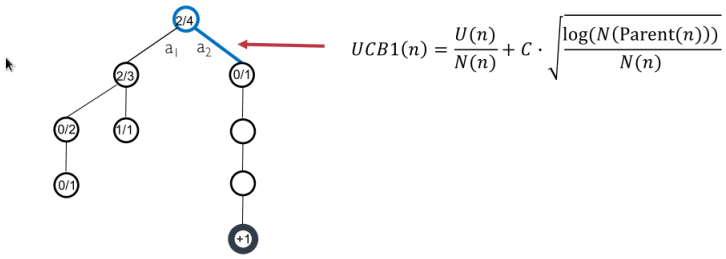


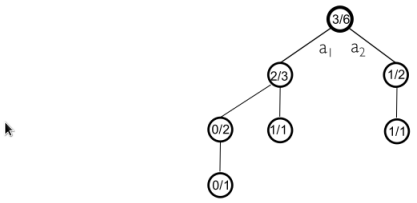

selection
policy
2/4
a1
a2
2/3
0/1
0/1
1/1
$UCB1(n) = \frac{U(n)}{N(n)} + C \cdot \sqrt{\frac{\log(N(\text{Parent}(n)))}{N(n)}}$


selection
policy
2/4
a1
a2
2/3
0/1
0/1
1/1
-1


2/5
a1
a2
2/4
0/1
0/2
1/1
0/1


2/4
a1
a2
2/3
0/1
0/2
1/1
0/1
$UCB1(n) = \frac{U(n)}{N(n)} + C \cdot \sqrt{\frac{\log(N(\text{Parent}(n)))}{N(n)}}$





MCTS does not need an explicit evaluation function. Random playouts are enough to explore the search space. Thus, it can be employed an evaluation function, or a substantial corpus of knowledge, is unavailable.

10 Adversarial search and Machine Learning

Alpha-beta pruning improves minimax by reducing the number of visited states. Some Basic Statistics - Tic-Tac-Toe First Move:

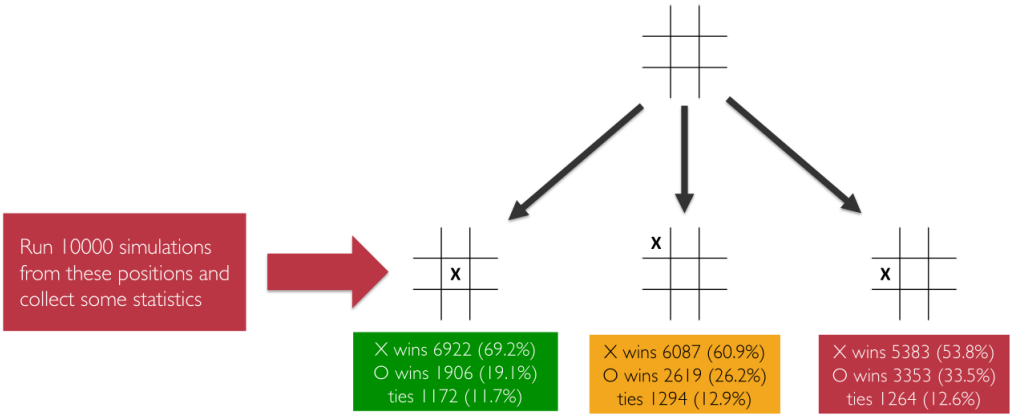
- Minimax Search
  - # visited states = 255168
  - # distinct states = 958 (0.4%)
  - 100 matches take 54.9 s
- Minimax Search with Repeated States
  - # visited states = 958
  - # distinct states = 958
  - 100 matches take 5.51 s
- Alpha-Beta Pruning
  - # visited states = 9221
  - # distinct states = 662 (7.2%)
  - 100 matches take 4.46 s
- Alpha-Beta Pruning with Repeated States
  - # visited states = 626
  - # distinct states = 626
  - 100 matches take 2.35 s

10.1 Beyond alpha-beta pruning

Alpha-Beta pruning is effective in many games; however, it cannot play game with large state spaces and very large branching factors like for example Go. Why Go is difficult? Its branching factor is very large, the game comprises more than 200 moves (about 250 moves on average), order of magnitude greater than the branching factor of 20 for Chess, it lacks a good evaluation function, it is difficult to model, similar looking positions can have completely different outcomes. Alpha-beta pruning reached a proficiency level weak to intermediate.

We can stop search at a given depth, but an adequate evaluation function becomes mandatory. That we cannot create unless we know the problem well. What could we do if we do not know the domain that well?

We could simulate how the match will develop after I perform a given action.



Monte Carlo Tree Search develops an asymmetric game tree while evaluating states using random simulations Simulations are run incrementally, one after the other, while deeper nodes are added to the tree.

The more complex the game, the more simulations we need and... The more expensive the simulation becomes. Here is the algorithm of MCTS:

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 6.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

Example of tree:

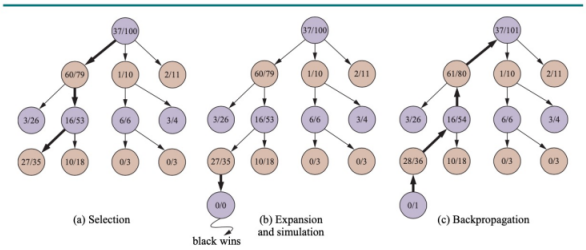


Figure 6.10 One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

10.2 Alpha zero

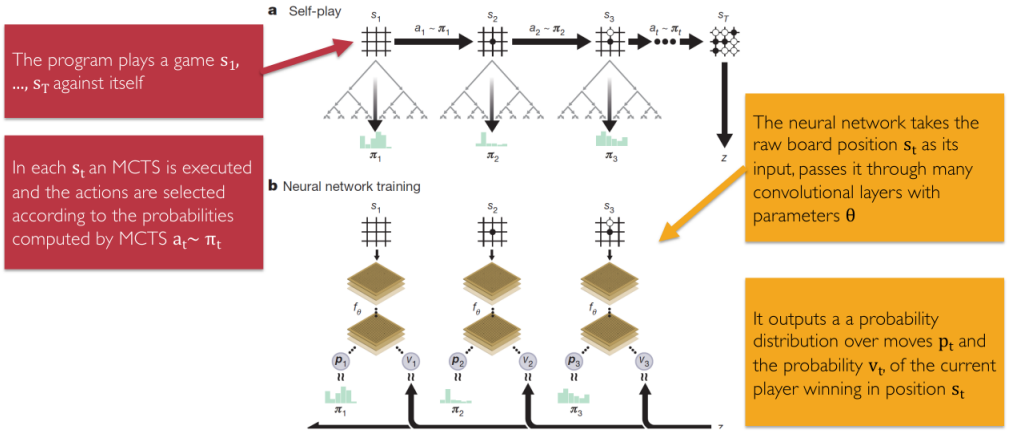
Learns from scratch by self-play reinforcement learning. It uses only the board as input features. It use a single neural network to evaluate positions and sample moves (compute their probabilities).

Deep neural network f_θ with parameters θ that takes as input a raw board representation s and its history and produces as output:

- the vector of move probabilities p for each action such that $p_a = Pr(a|s)$
- a scalar value v estimating the probability of the current player winning from position s
- $f_\theta(s) = (p, v)$

The neural network is trained from games of self-play using reinforcement learning. In each position s, an MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities π of playing each move. These search probabilities usually select much stronger moves than the raw neural network probabilities p computed by f_θ. MCTS can be viewed as a powerful policy improvement operator.

It is based on a new reinforcement learning algorithm with lookahead search inside the training loop.



Now let's go back to the beginning: we want to develop an artificial player for a game, let's say tic-tac-toe. What are out options? We can implement a player using

10.2.1 Approach #1

We can implement a player using what we know about the game. Here is the sketch of pseudo-code for a tic-tac-toe player:

- if there is a winning position, take it and win
- if there is a loosing position, block it
- select the best position, considering that:
  - the center is the best

- next most important positions are the corners near one of our pieces
- finally, choose one of the side positions

What are the limitations of this approach?

1. We need to be proficient in the game
2. We do not have a general strategy to explore the variety of situations that might occur
3. In fact, we cannot be sure that we considered all the possibilities
4. We have to start from scratch if we consider a new game

We can have games with very similar rules that require very different playing strategies

10.2.2 Approach #2

What do we do as players? We usually simulate what might happen ahead. Suppose they teach you to play a new game, something completely different from what you played so far... Here is a naive human player approach:

- Consider the current state of the game and the moves I can do
- If I do move X what will the opponent do?
- If I am not proficient, I am forced to consider all options and possible match developments
- How much look ahead can I do? One-step ahead? Two steps? How many options can I evaluate?
- If I am very good, I might be able to simulate all the possibilities

This is exactly what minimax using a utility function does. Is minimax good?

- Pros:
  - It provides a general strategy to check moves and situations (it is optimal)
  - We can use it with any game, we are just using knowledge about the available moves and the terminal condition
  - We do not need to be proficient in the game or previous experience
- Cons:
  - When applying minimax search using utility (not an evaluation function) we explore all the possible matches
  - This is ok for tic-tac-toe but not for more complex games like chess
  - But even simpler games can become infeasible for plain minimax using utility experience

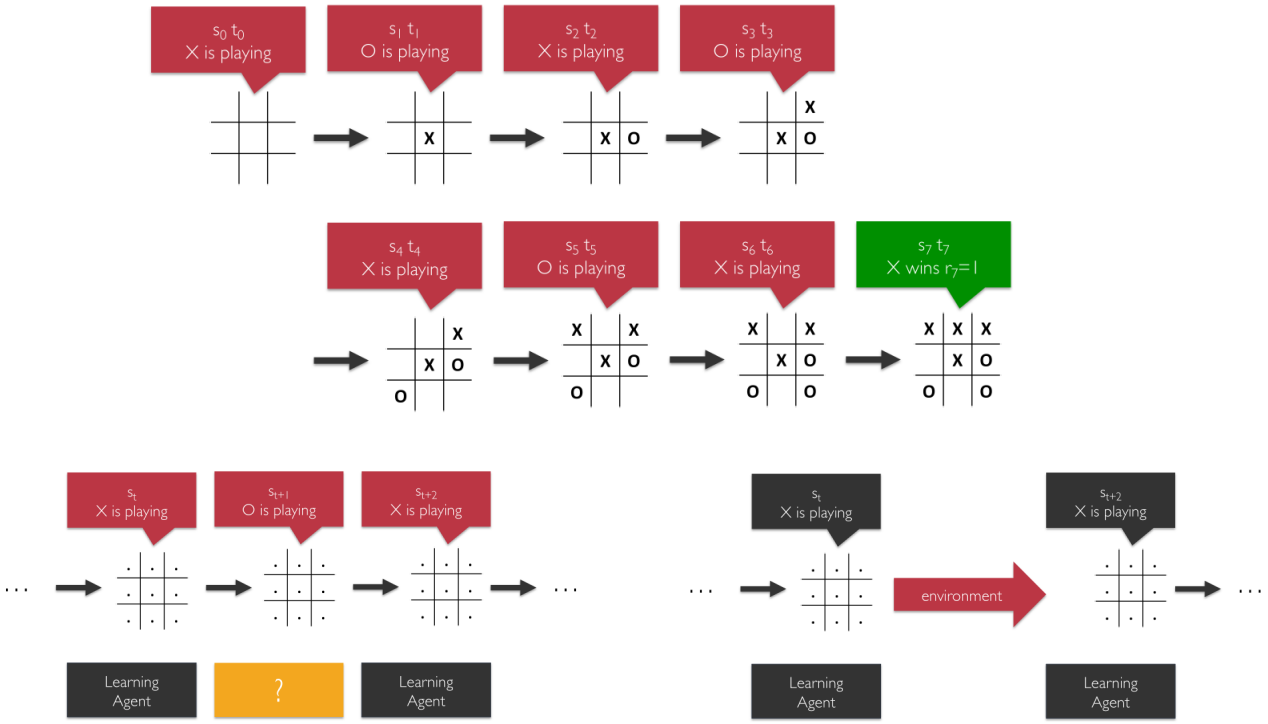
We can improve it by avoiding exploring situations that we know will lead to worse solutions. This is exactly what alpha-beta pruning using a utility function does.

10.2.3 Approach #3

All the previous approaches only use the game rules and no additional domain knowledge. We could add it by introducing an evaluation function. But before we do that, we will try other approaches that do not need domain knowledge.

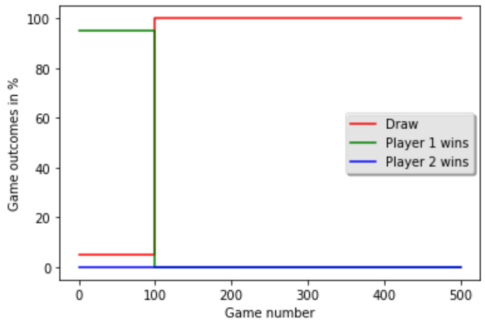
We assume that the agent must learn how to play first (X). The state s_t is represented by the current board. The agent has potentially 9 possible actions corresponding to the 9 positions. The agents receives a reward of 1 if it wins, -1 if it looses and 0 if it draws. The other parameters are:

- the learning rate β
- the discount factor γ
- the value q_0 to initialize the Q-table

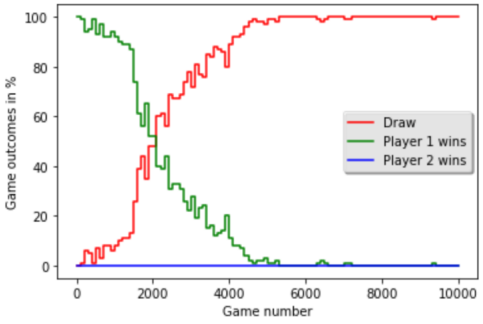


How do we track performance? Who is playing the opponent?

Option #1 - Minimax, the optimal one
Player #1 (Minimax) vs. Player #2 (Q-learning)
Minimax is deterministic. It always returns the optimal move. How is it to learn from such an opponent?

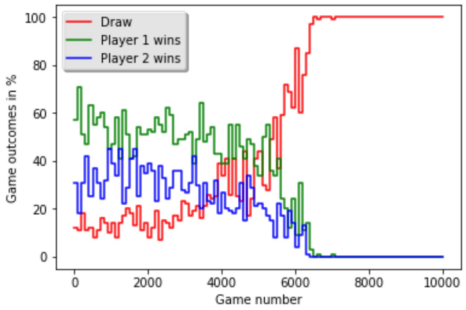


Option #2 - "random" Minimax
Player #1 (RndMinimax) vs. Player #2 (Q-learning)
It works like Minimax and when there are more moves with utility 1 for a given position it randomly chose between them. RndMinimax still returns an optimal move. However, it is stochastic since O selects an optimal action randomly among the available ones. Thus, the same state (board), and the same action by X, might return a different next state (board), since O is selecting an action randomly. How is it to learn from such an opponent?



Option #3 - Tabular Q-learning
Player #1 (Q-learning) vs. Player #2 (Q-learning)

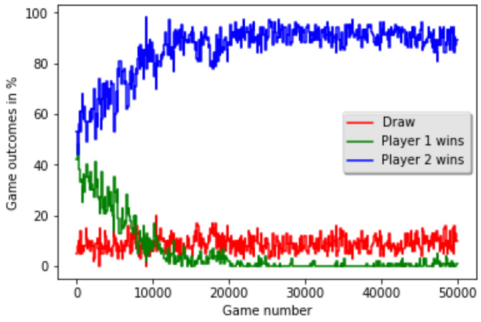
Both players are trying to learn how to play. They are learning different roles, one as the starting player, one as the second player. Initially they don't know anything, will they be able to learn how to play?



Option #4 - Random player

Player #1 (Random player) vs. Player #2 (Q-learning)

We are learning to play against an opponent that does not anything about playing. It is stochastic since O selects actions randomly among the available ones. How is it to learn from such an opponent?

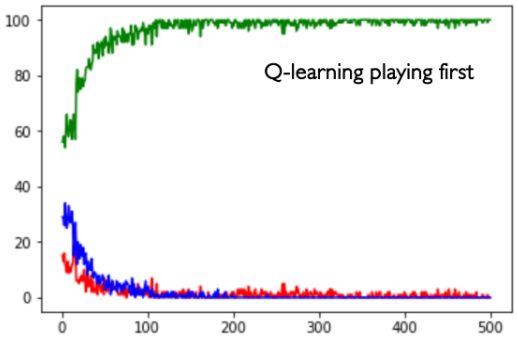


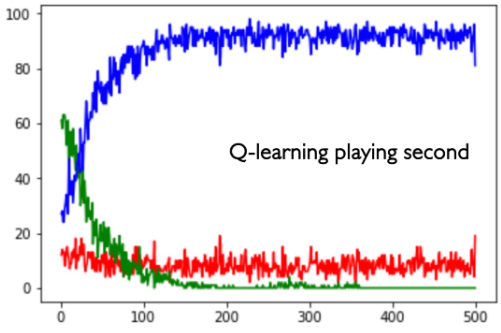
Some Thoughts about Minimax and Tabular Q-learning

Like Minimax and Alpha-beta pruning, tabular Q-learning is feasible for small number of states. Minimax explores the entire game tree which is infeasible even for Connect-4. Alpha-beta pruning explores only the relevant part of the game tree. Tabular Q-learning needs to store the Q values for every state-action pair. Minimax/Alpha-beta pruning assume that the opponent plays optimally. If it does not, they can exploit its weaknesses (minimax values should be interpreted as "at least"). Minimax/Alpha-beta pruning play optimally right from the start while Q-learning needs to be trained and its performance is influence by the opponents used for training Q-learning must be trained against an opponent whose quality can affect the learning process However, it can learn an optimal strategy against any opponent, even a random one.

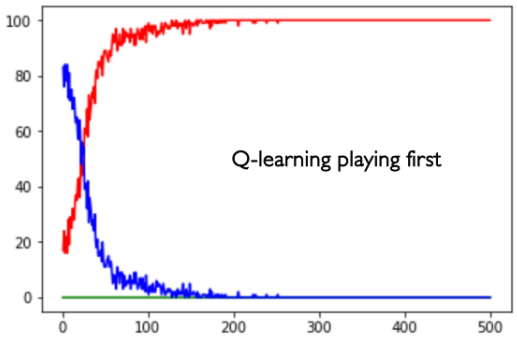
10.2.4 Approach #4

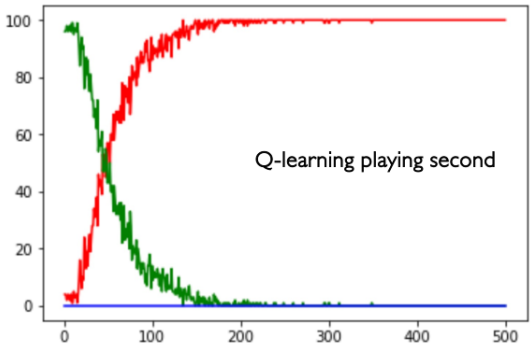
The Q-table can be viewed as a function mapping state-action pairs into q-values. We can use a deep neural network to approximate the Q-table. There is a caveat though, we are simultaneously learning both the q-values and how to approximate them. Q-learning implemented using deep neural networks (green when playing first, blue when playing second) vs a random player (draws are reported in red):





Q-learning implemented using deep neural networks (green when playing first, blue when playing second) vs a random minimax (draws are reported in red):





Is there anything else we can do without using any domain knowledge? We can use simulations to assess the state values and apply MCTS.

11 Constraint Satisfaction Problems: Formulation and Solving

Let's start from the search trees. We have seen that the search trees are composed by 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.

But, considering the implementation we have made of the node, we can easily see that the state of a node is written in a very generic manner. **We have not specified any structure for it so the states can be anything. Plain search algorithms only compare states using $=$ and $\neq$.**

What we want to do now is to specialize the representation of the states in a search problem. Now it rises a new tradeoff:

$$\text{expressiveness} \leftrightarrow \text{efficiency}$$

When the structure of the state is fixed, more efficient solving techniques could be developed. In constraint satisfaction problems, the state has a **factored representation**, namely it is represented as a set of pairs variable - value.

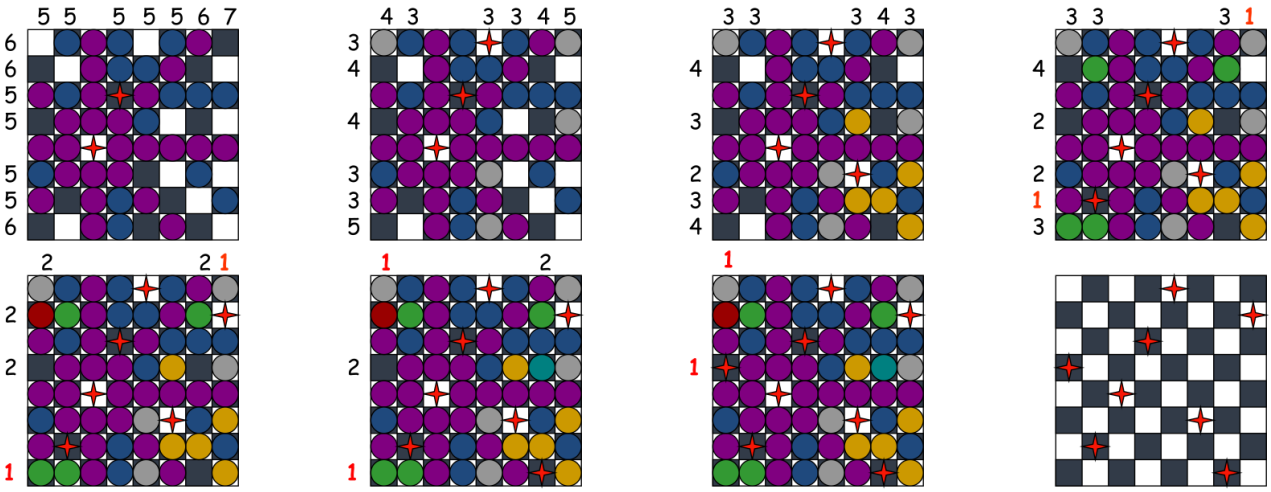
11.1 8-queens problem

We consider once again this problem in which your goal is to position 8 queens on a chessboard so that no queen can attack any other queen. A naive formulation is:

- Initial state: empty board
- Actions(): add a queen to an empty cell
- Result(): ...
- Goal test: 8 queens that do not attack each other
- Step cost: 1

A more efficient formulation is:

- Place a queen on a cell
- Do not consider attacked cells
- Coun the number of cells that are not attacked in each row and column
- Place a queen in the cell corresponding to the row and column with the smallest number
- Do not consider attacked cells
- Repeat ...



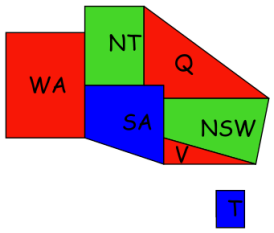
What can an efficient formulation provide? Empowering function Actions() and goal test, "propagate the constraints" in order to limit the actions applicable in a state and identifying failures as soon as possible (failure test).

11.2 Constraint Satisfaction Problems (CSPs)

They are defined as follows:

- Set of **variables**: $\{X_1, \ldots, X_n\}$
- **Domains**: $D_1, \ldots, D_n$ of possible values for each variable
- Set of **Constraints**: $\{C_1, \ldots, C_p\}$. Each constraint involves a subset of variables and specifies the valid (or invalid) combinations of their values

The goal is to assign a value to each variable in such a way that all the constraints are satisfied. **Solution**: **complete** (all variables are assigned a value) and **consistent** (all constraints are satisfied) assignments.
Example: Map Coloring



- 7 variables: $\{WA, NT, Q, NSW, V, SA, T\}$
- Each variable has the same domain: $D_i = \{red, green, blue\}$
- Variables corresponding to adjacent regions cannot have the same color: $WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V$

Example: 8-queens

- 8 variables $X_i, i = 1, 2, \ldots, 8$, each one representing the row of the queen on the column i (or vice versa)
- Each variable has the same domain $D_i = \{1, 2, \ldots, 8\}$
- Constraints are of the form:
  - no two queens on the same row: $X_i \neq X_j, i, j = 1, 2, \ldots, 8, i \neq j$
  - no two queens on the same diagonal: $|X_i - X_j| \neq |i - j|, i, j = 1, 2, \ldots, 8, i \neq j$

Example: Zebra puzzle There are five houses.

- The Englishman lives in the red house
- The Spaniard owns the dog
- Coffee is drunk in the green house
- The Ukrainian drinks tea
- The green house is immediately to the right of the ivory house
- The Old Gold smoker owns snails
- Kools are smoked in the yellow house
- Milk is drunk in the middle house
- The Norwegian lives in the first house
- The man who smokes Chesterfields lives in the house next to the man with the fox
- Kools are smoked in the house next to the house where the horse is kept
- The Lucky Strike smoker drinks orange juice
- The Japanese smokes Parliaments
- The Norwegian lives next to the blue house

Who drinks water? Who owns the zebra?

- 5 variables, the "houses"
- Each variable is a vector of 5 values: nationality, color, beverages, cigarettes, animals
  - N_i = {Englishman, Spanish, Ukrainian, Japanese, Norwegian}
  - C_i = {red, green, ivory, yellow, blue}
  - B_i = {coffee, tea, milk, orange juice, water}
  - S_i = {Old Gold, Kools, Chesterfields, Lucky Strike, Parliaments}
  - A_i = {dog, snalis, fox, horse, zebra}

- Constraints:
  - The nationality of a house is different from the nationality of the other houses → if $X_i = [v, \dots]$ then $X_j = [v', \dots]$ with $v \neq v'$ for all $j \neq i$
  - The color of a house is different from the color of the other houses
  - ...
  - The Englishman lives in the red house → if $X_i = [\text{Englishman}, \dots]$ then $X_i = [\text{Englishman}, \text{red}, \dots]$
  - The Norwegian lives in the first house → $X_1 = [\text{Norwegian}, \dots]$
  - ...

11.3 Assignments

Consider n variables: $X_1, X_2, \dots, X_n$.

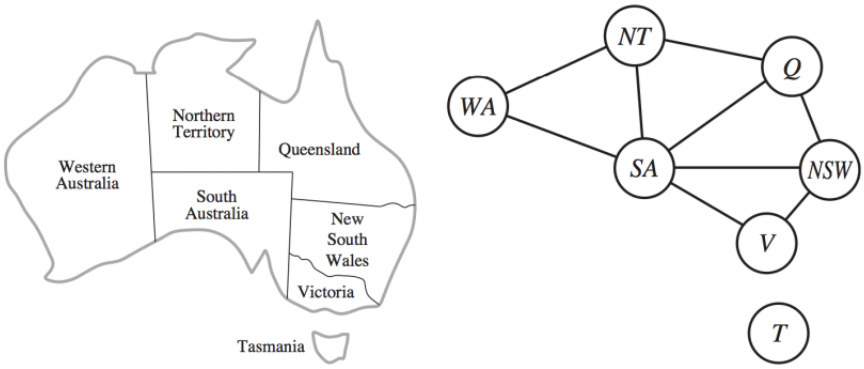
Definition of **consistent assignment**: $\{X_{i1} \leftarrow v_{i1}, X_{i2} \leftarrow v_{i2}, \dots, X_{ik} \leftarrow v_{ik}\}$ with $0 \leq k \leq n$ such that values $v_{i1}, v_{i2}, \dots, v_{ik}$ satisfy all constraints involving variables $X_{i1}, X_{i2}, \dots, X_{ik}$.

Definition of **complete assignment**: $\{X_{i1} \leftarrow v_{i1}, X_{i2} \leftarrow v_{i2}, \dots, X_{ik} \leftarrow v_{ik}\}$ with $k = n$.

If the domains of the variables have size d, there are $O(d^n)$ complete assignments.

11.4 Binary constraints

We usually consider binary constraints, namely with constraints involving two variables. Any generic CSP can be transformed in a CSP with only binary constraints. The structure of a CSP with binary constraints is represented by a constraint graph.



Example - Binary constraints Oridinal problem: $X = \{x, y, z\}$, $D = \{D_y = \{1, 2\}, D_x = \{3, 4\}, D_z = \{5, 6\}\}$, $C = \{C_1 = [x + y = z]\}$. Constraint C_1 is not binary → binarization. **Goal**: formulate a new problem equivalent to the original one but with only binary constraints. Hint: introduce a new variable with a complex domain...

Introduce a new variable u whose domain is the Cartesian product of the domains of the variables involved in C_1: $D_u = \{(1,3,5), (1,3,6), (1,4,5), (1,4,6), (2,3,5), (2,3,6), (2,4,5), (2,4,6)\}$. Reduce D_u keeping only the valid combinations according to C_1: $D_u = \{(1,4,5), (2,3,5), (2,4,6)\}$ ("u encapsulates C_1"). Add consistency constraints: $u[1] = x, u[2] = y, u[3] = z$.

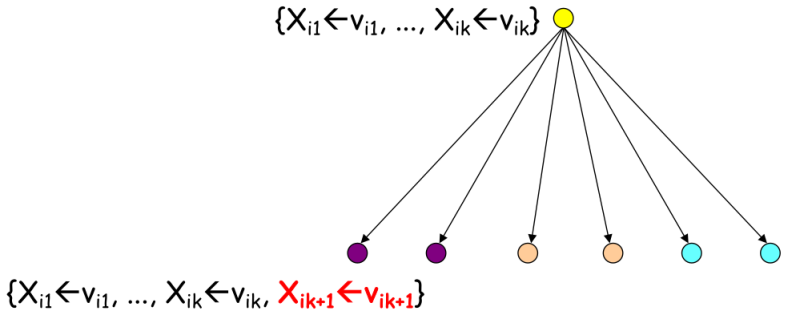
Implementation of CSP:

```
class CSP(search.Problem):
    """This class describes finite-domain Constraint Satisfaction Problems.
    A CSP is specified by the following inputs:
        variables A list of variables; each is atomic (e.g. int or string).
        domains A dict of {var:[possible_value, ...]} entries.
        neighbors A dict of {var:[var,...]} that for each variable lists
            the other variables that participate in constraints.
        constraints A function f(A, a, B, b) that returns true if neighbors
            A, B satisfy the constraint when they have values A=a, B=b

    def MapColoringCSP(colors, neighbors):
        """Make a CSP for the problem of coloring a map with different colors
        for any two adjacent regions. Arguments are a list of colors, and a
```

```
        dict of {region: [neighbor,...]} entries. This dict may also be
        specified as a string of the form defined by parse_neighbors."""
        if isinstance(neighbors, str):
            neighbors = parse_neighbors(neighbors)
        return CSP(list(neighbors.keys()), UniversalDict(colors), neighbors, different_values_constraint)

    Australia_csp = MapColoringCSP(list('RGB'), """SA: WA NT Q NSW V; NT: WA Q; NSW: Q V; T: """)
```

11.5 CSPs as search problems

- States: consistent assignments
- Initial state: empty assignment $\{\}$, namely $k = 0$
- Actions($\{X_{i1} \leftarrow v_{i1}, X_{i2} \leftarrow v_{i2}, \ldots, X_{ik} \leftarrow v_{ik}\}$): all possible consistent assignment for X_{ik+1}
- Result($\{X_{i1} \leftarrow v_{i1}, X_{i2} \leftarrow v_{i2}, \ldots, X_{ik} \leftarrow v_{ik}\}$, $X_{ik+1} \leftarrow v_{ik+1}$) = $\{X_{i1} \leftarrow v_{i1}, X_{i2} \leftarrow v_{i2}, \ldots, X_{ik} \leftarrow v_{ik}, X_{ik+1} \leftarrow v_{ik+1}\}$
- Goal test: $k = n$
- Step cost: unitary



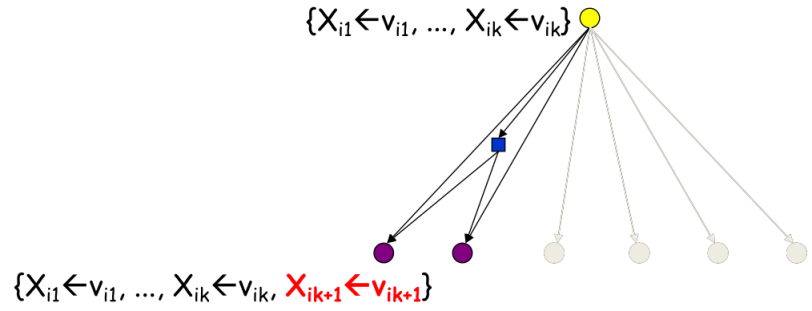
$r = n - k$ variables that are not yet assigned with d values $\rightarrow$ branching factor $r \times d$.

11.5.1 Commutativity

The order in which values are assigned to the variables has no impact on the possibility of reaching consistent and complete assignments. Expand a node N by choosing one variable X not included in the assignment associated to N and by assigning it a value from its domain (the new assignment must be consistent) $\rightarrow$ reduce branching factor.

Example - Commutativity Consider 4 variables X_1, X_2, X_3, X_4. Consider a node N with (consistent) assignment $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2\}$. Consider, for instance, variable X_4 with domain $\{v_{4,1}, v_{4,2}, v_{4,3}\}$. Successors of N are all the nodes with consistent assignments:

- $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_4 \leftarrow v_{4,1}\}$
- $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_4 \leftarrow v_{4,2}\}$
- $\{X_1 \leftarrow v_1, X_2 \leftarrow v_2, X_4 \leftarrow v_{4,3}\}$

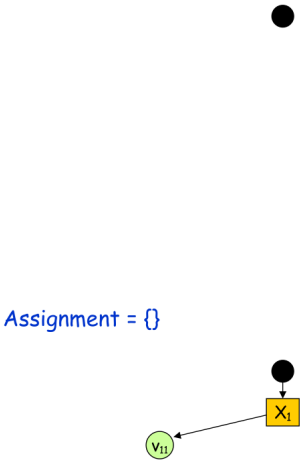


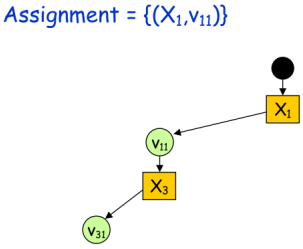
The order in which values are assigned to the variables has no impact on the possibility of reaching consistent and complete assignments.

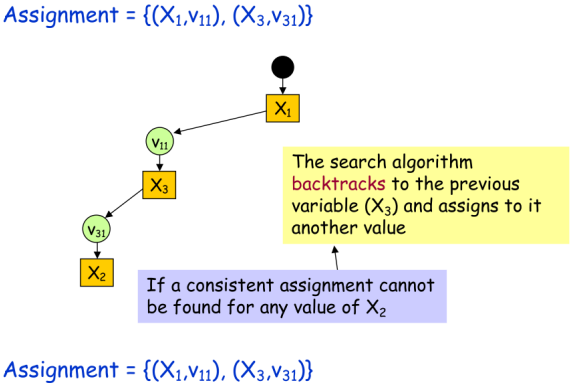
1. Expand a node N by choosing one variable X not included in the assignment associated to N and by assigning it a value from its domain (the new assignment must be consistent) $\rightarrow$ reduce branching factor
2. The path for reaching a node is not relevant and **all solutions are at depth n $\rightarrow$ depth-first search with backtracking**

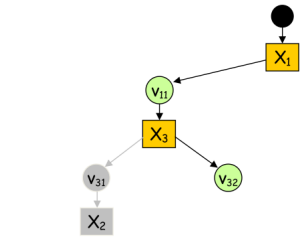
11.5.2 Backatracking: consider a single value

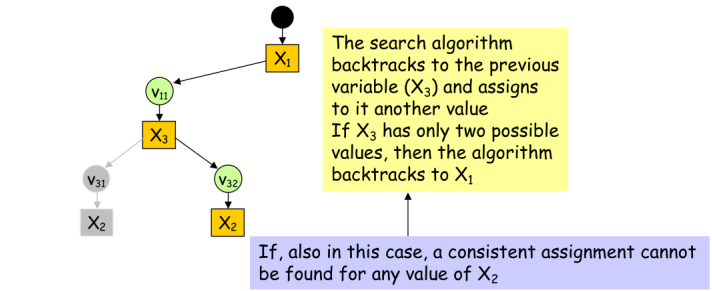
Example with 3 variables: (x_1, x_2, x_3) and domains: $D_1 = \{v_{11}, v_{12}, \dots\}$, $D_2 = \{v_{21}, \dots\}$, $D_3 = \{v_{31}, v_{32}\}$.

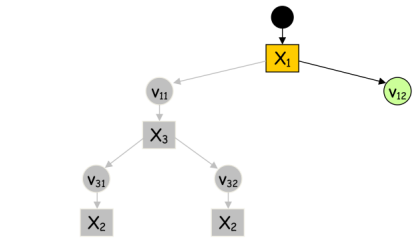


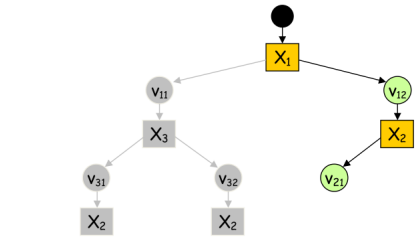


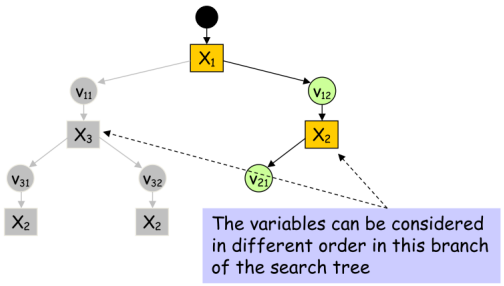




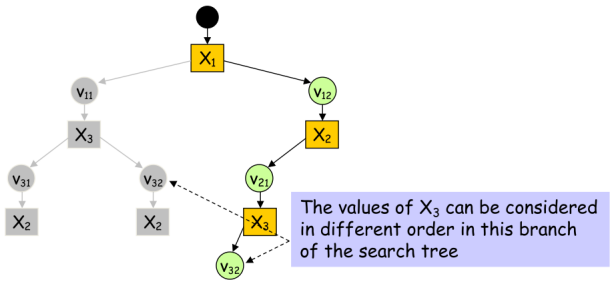




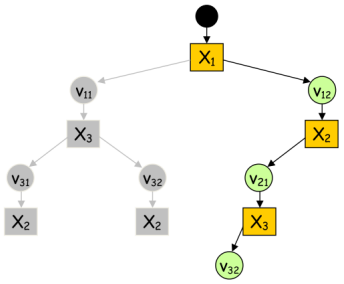




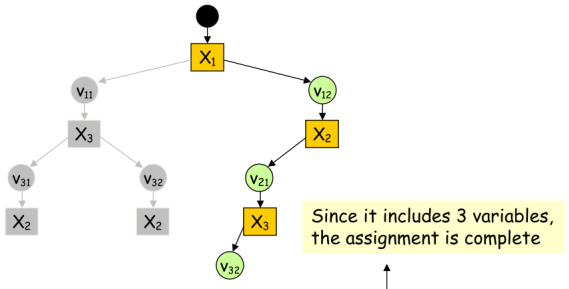
Assignment = {(X_1,v_{12}), (X_2,v_{21})}



Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}



Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}



Assignment = {(X_1,v_{12}), (X_2,v_{21}), (X_3,v_{32})}

Here is the backtracking algorithm:

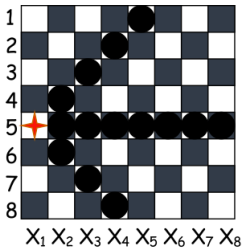
```
CSP-BACKTRACKING(A)
    IF assignment A is complete, THEN RETURN A
    X ← choose a variable not in A
    D ← choose an order of values in the domain of X
    FOR EACH v in D DO
        add (X ← v) to A
        IF A is consistent THEN
            result ← CSP-BACKTRACKING(A)
        IF result != failure, THEN RETURN result
            remove (X ← v) from A
    RETURN failure


def backtracking_search(csp,select_unassigned_variable=first_unassigned_variable,
order_domain_values=unordered_domain_values, inference=no_inference):
    def backtrack(assignment):
        if len(assignment) == len(csp.variables):
            return assignment
        var = select_unassigned_variable(assignment, csp)
        for value in order_domain_values(var, assignment, csp):
            if 0 == csp.nconflicts(var, value, assignment):
                csp.assign(var, value, assignment)
                removals = csp.suppose(var, value)
                if inference(csp, var, value, assignment, removals):
                    result = backtrack(assignment)
                    if result is not None:
                        return result
                csp.restore(removals)
        csp.unassign(var, assignment)
        return None

    result = backtrack({})
    assert result is None or csp.goal_test(result)
    return result
```

12 Constraint Satisfaction Problems: Heuristics and Optimization

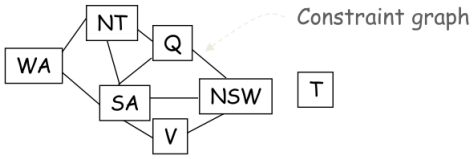
Let's start considering the backtracking algorithm:

```
CSP-BACKTRACKING(A)
    IF assignment A is complete, THEN RETURN A
    X ← choose a variable not in A
    D ← choose an order of values in the domain of X
    FOR EACH v in D DO
        add (X ← v) to A
        IF A is consistent THEN
            result ← CSP-BACKTRACKING(A)
            IF result != failure, THEN RETURN result
        remove (X ← v) from A
    RETURN failure
```

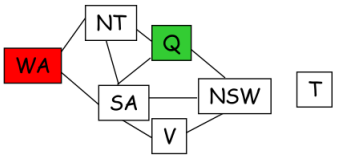
Regarding the forward checking: assigning value 5 to X_1 reduces the domains of $X_2, X_3, \ldots, X_8$.



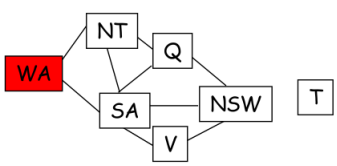
Consider now the map coloring problem:



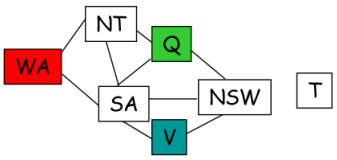
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB



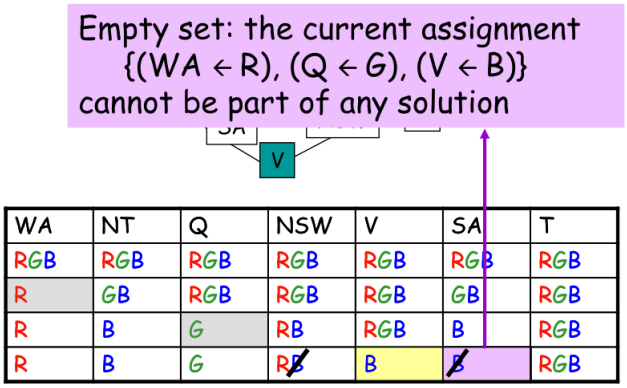
WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	~~B~~	G	~~RGB~~	RGB	~~B~~	RGB



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	~~RGB~~	RGB	RGB	RGB	~~RGB~~	RGB



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	~~RB~~	B	~~B~~	RGB



WA	NT	Q	NSW	V	SA	T
RGB	RGB	RGB	RGB	RGB	RGB	RGB
R	GB	RGB	RGB	RGB	GB	RGB
R	B	G	RB	RGB	B	RGB
R	B	G	~~R~~B	B	~~B~~	RGB

When $X \leftarrow v$ is added to an assignment A:

```
    FOR EACH variable Y not in A DO
FOR EACH constraint C that involves Y and X DO
eliminate from the domain of Y all the values that do not satisfy C (given A)
```

Python implementation:

```
def forward_checking(csp, var, value, assignment, removals):
    """Prune neighbor values inconsistent with var=value."""
    csp.support_pruning()
    for B in csp.neighbors[var]:
        if B not in assignment:
            for b in csp.curr_domains[B][:]:
                if not csp.constraints(var, value, B, b):
                    csp.prune(B, b, removals)
            if not csp.curr_domains[B]:
        return False
    return True
```

Here is the modified backtracking algorithm:

```
CSP-BACKTRACKING(A, var-domains)
IF assignment A is complete, THEN RETURN A
X ← choose a variable not in A
D ← choose an order of values in the domain of X
FOR EACH v in D DO
add (X ← v) to A
var-domains ← FORWARD-CHECKING(var-domains, X, v, A)
IF no variable has an empty domain THEN
result ← CSP-BACKTRACKING(A, var-domains)
IF result != failure, THEN RETURN result
remove (X ← v) from A
RETURN failure
```

Which variable X should be assigned next? The current assignment could not bring to any solution, but we might need time to discover this. Choose the right variable in order to discover inconsistencies in the current assignment as soon as possible. In what order should the values of X be considered? The current assignment could bring to a solution (could be parto of a solution). Choose the right value to assign to X in order to reach a solution as quickly as possible.

These aspects are domain-independent and made possibile by the factored representation of the state.

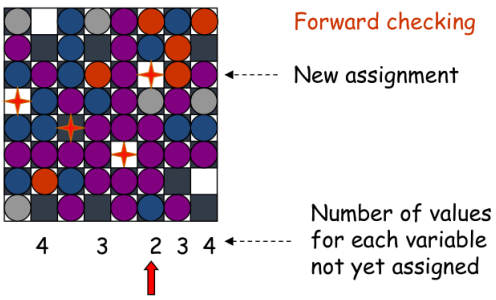
To answer the previous questions:

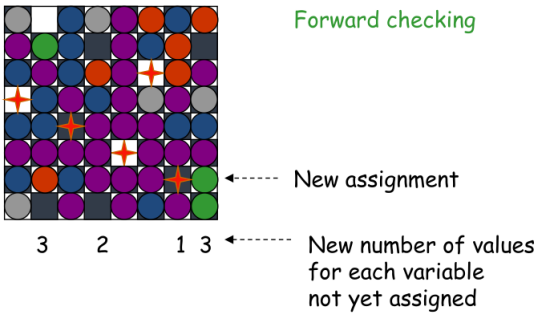
- Which variable X should be assigned next? **Minimum-Remaining-Values (MRV) heuristic and Degree heuristic**
- In what order should the values of X be considered? **Least-constraining-value heuristic**

12.1 MRV heuristic

Also called most-constrained-variable heuristic. Choose the variable with the fewest legal values remaining in its domain (after application of forward checking).

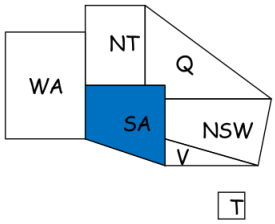
For example, in the 8 queens problem:





12.2 Degree heuristic

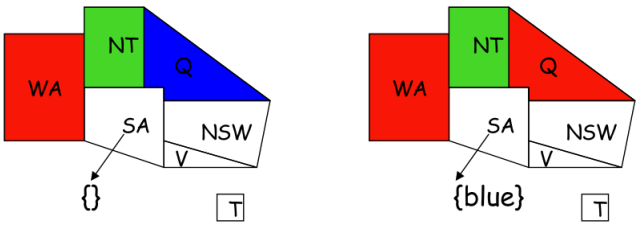
Choose the variable that is involved in the largest number of constraints with unassigned variables. Used to break ties of the MRV heuristic. Consider the map coloring problem:



The variable chosen first is SA because all the variables have domains composed of 3 values, but SA is involved in the largest number (5) of constraints with variables not yet assigned.

12.3 Least-constraining-value heuristic

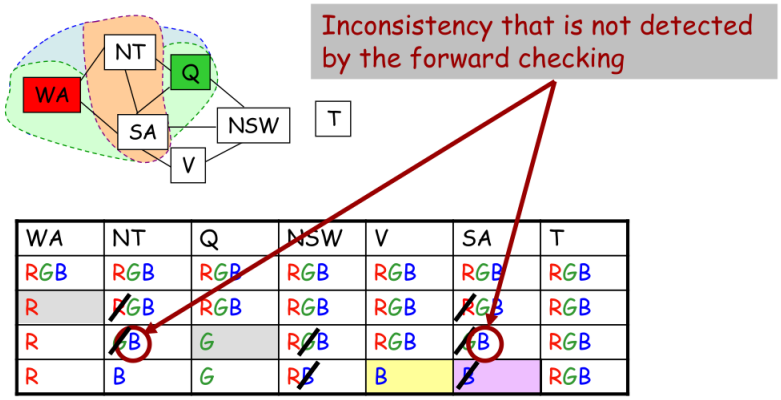
Choose the value that gives "more freedom" to the variables that are not yet assigned. Again, considering the map coloring problem:



Q has two values in its domain: red and blue. Assigning blue to Q leaves 0 values for SA, while assigning red to Q leaves 1 value for SA so the least-constraining-value heuristic assignes red to Q.

Using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable. Forward checking is a very simple form of **constraint propagation**.

Can we do better than what we have seen until now?
The answer is obviously yes, indeed:



12.4 Arc consistency

Applicable to CSPs with binary constraints (constraints that involve just two variables). Assume to have a directed constraint graph. Remove from the domain of X the values that are inconsistent with the values in the domain of Y.

```
    REMOVE-VALUES(X,Y) [consider the directed arc from X to Y]
removed ← false
FOR EACH value v in the domain of X DO
IF there is no value u in the domain of Y such that the constraints between X and Y is satisfied
THEN remove v from the domain of X
removed ← true
RETURN removed
```

12.5 AC-3 algorithm

```
initialize a queue Q with all the constraints (X,Y) and (Y,X)
WHILE Q is not empty DO
(X,Y) ← remove from Q
IF REMOVE-VALUES(X,Y) THEN
IF the domain of X is empty THEN RETURN failure
insert all (Z,X) in Q (if they are not yet in Q, do not insert (Y,X))
```

Actual python implementation:

```
    def AC3(csp, queue=None, removals=None, arc_heuristic=dom_j_up):
    if queue is None:
    queue = {(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]}
    csp.support_pruning()
    queue = arc_heuristic(csp, queue)
    while queue:
    (Xi, Xj) = queue.pop()
    if revise(csp, Xi, Xj, removals):
    if not csp.curr_domains[Xi]:
    return False
    for Xk in csp.neighbors[Xi]:
    if Xk != Xj:
    queue.add((Xk, Xi))
    return True
```

12.5.1 Temporal complexity analysis

```
    Forward checking
    When (X ← v) is added to an assignment A:
    FOR EACH variable Y not in A DO
    FOR EACH constraint C that involves Y and X DO
    eliminate from the domain of Y all the values that do not satisfy C (given A)
```

Given:

- n: number of variables:
- s: largest number of constraints that involve any given variable ($s \leq n-1$)
- d: size of domains

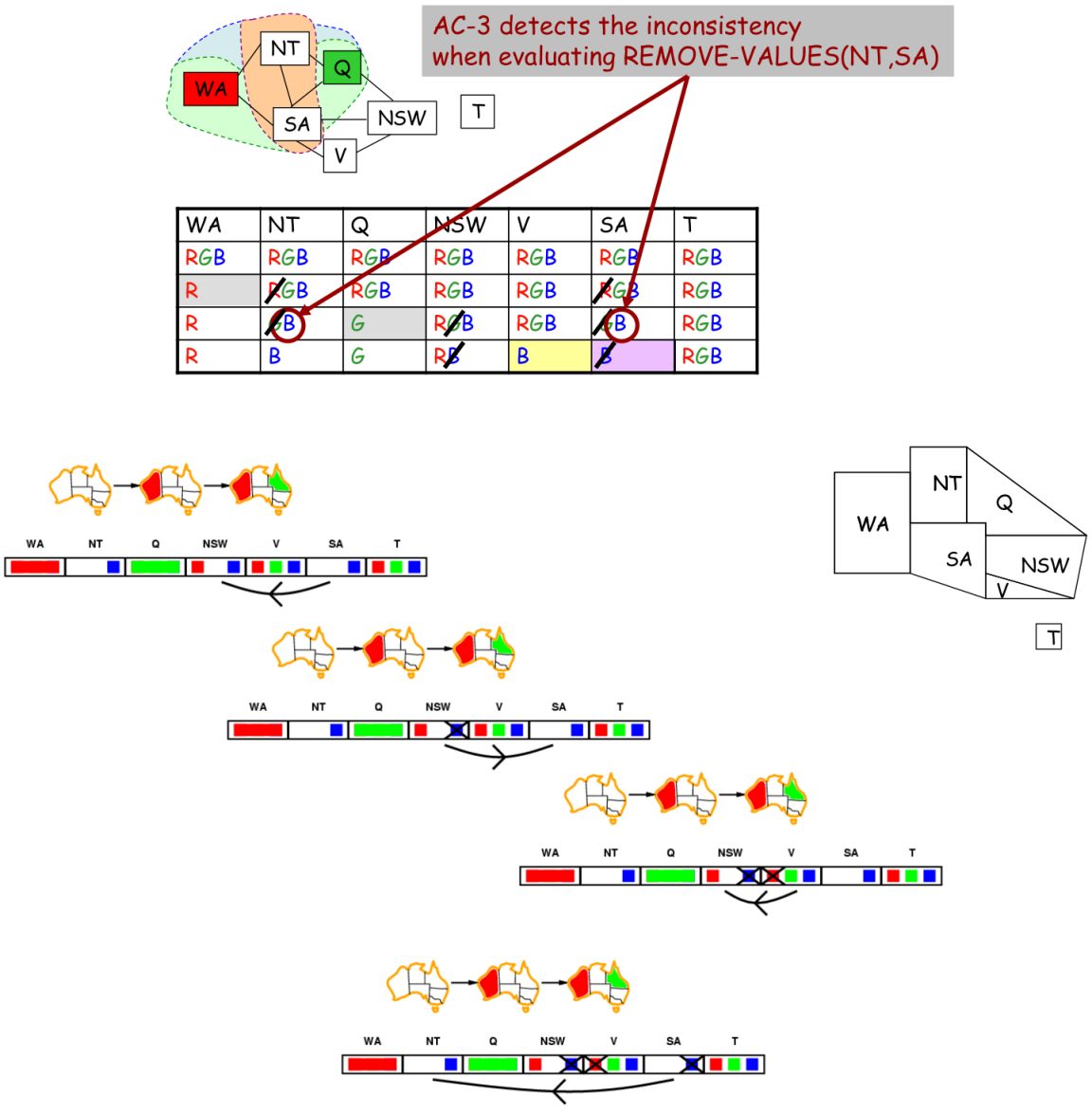
You have:

$$O(n \times s \times d)$$

```
AC-3
initialize a queue Q with all the constraints (X,Y) and (Y,X)
WHILE Q is not empty DO
(X,Y) ← remove from Q
IF REMOVE-VALUES(X,Y) THEN
IF the domain of X is empty THEN RETURN failure
insert all (Z,X) in Q (if they are not yet in Q, do not insert (Y,X))
```

Each arc is inserted in Q at most d times. REMOVE-VALUES has temporal complexity $O(d^2)$. AC-3 has complexity

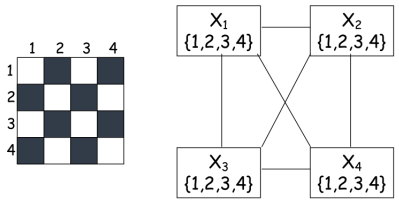
$$O(n \times d \times s \times d^2) = O(n \times s \times d^3)$$



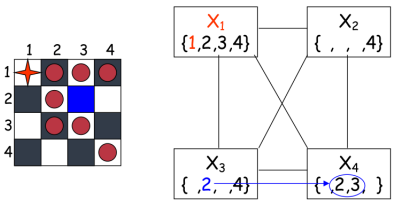
Modified backtracking algorithm:

```
    IF assignment A is complete, THEN RETURN A
    run AC-3 and reduce var-domains of unassigned variables
    IF a variable has an empty dosmain THEN RETURN failure
    X ← choose a variable not in A
    D ← choose an order of values in the domain of X
    FOR EACH v in D DO
    add (X ← v) to A
    var-domains ← FORWARD-CHECKING(var-domains, X, v, A)
    IF no variable has an empty domain THEN
    result ← CSP-BACKTRACKING(A, var-domains)
    IF result != failure, THEN RETURN result
    remove (X ← v) from A
    RETURN failure
```

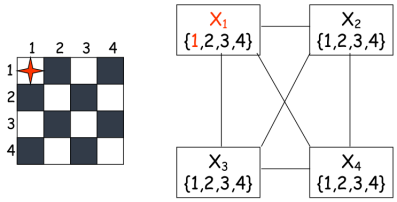
12.5.2 4-queens problem



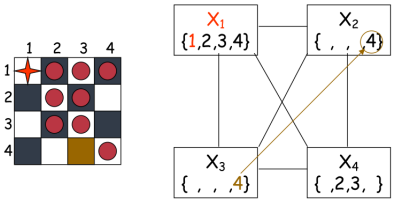
The modified[2] backtracking algorithm starts by calling AC-3, which do not remove any value from domains



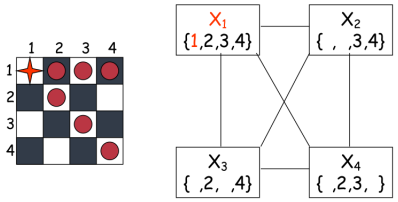
The algorithm calls AC-3, which removes 3 from the domain of X_2 and 2 from the domain of X_3



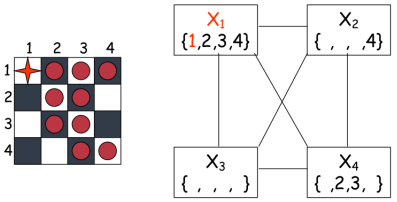
The algorithm chooses a variable and a value to assign to it. Heuristics do not help in this case: variable X_1 and value 1 are chosen randomly



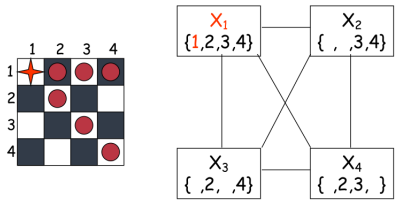
The algorithm calls AC-3, which removes 3 from the domain of X_2 and 2 from the domain of X_3 and 4 from the domain of X_3



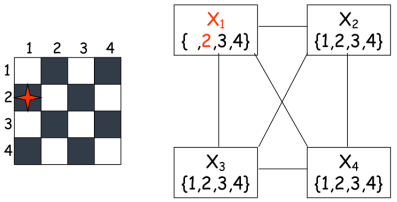
The algorithm calls forward checking, which eliminates two values in the domains of the unassigned variables



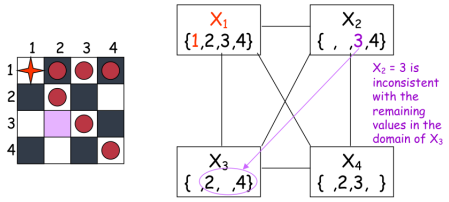
The domain of X_3 is empty → backtracking



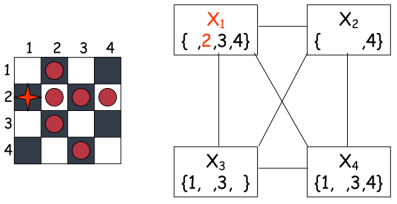
The algorithm calls AC-3



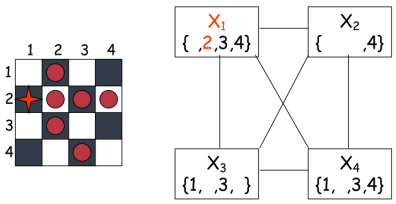
The algorithm removes 1 from the domain of X_1 and assigns 2 to X_1



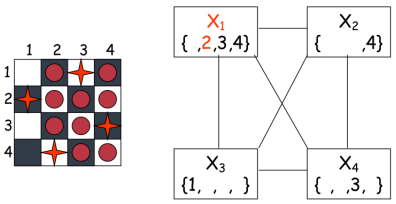
The algorithm calls AC-3, which removes 3 from the domain of X_2



The algorithm calls forward checking

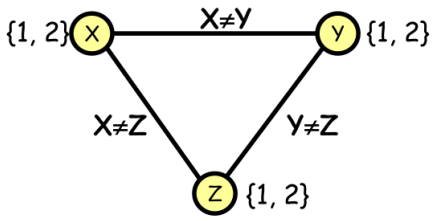


The algorithm calls AC-3



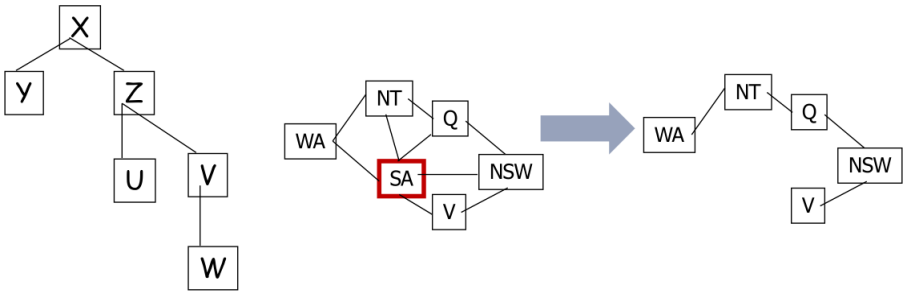
The algorithm calls AC-3, which reduces the domains to X_3 and X_4 to a single value

Is AC-3 enough? No, it is not. AC-3 does not detect all the inconsistencies among binary constraints. Consider the following example:



Generalizing constraint propagation (k-consistency) increases the computational complexity. Trade-off between time spent in search and time spent in constraint propagation.AC-3 is usually a good compromise. Other improvements:

- Backjumping
- Exploit the structure of the problem
- Independent subproblems
- Efficient solutions when the constraint graph is a tree. Tree decomposition:

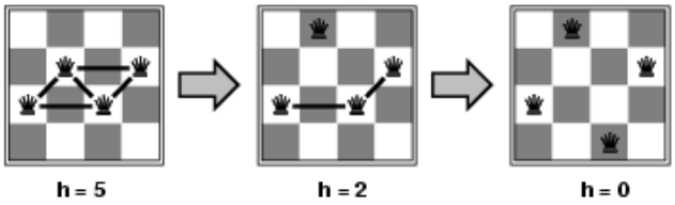



```
    IF assignment A is complete, THEN RETURN A
    run AC-3 and reduce var-domains of unassigned variables
    IF a variable has an empty dosmain THEN RETURN failure
    X ← choose a variable not in A
    D ← choose an order of values in the domain of X
    FOR EACH v in D DO
    add X ← v to A
    var-domains ← FORWARD-CHECKINGvar-domains, X, v, A
    IF no variable has an empty domain THEN
    result ← CSP-BACKTRACKINGA, var-domains
    IF result != failure, THEN RETURN result
    remove X ← v from A
    RETURN failure
```

Some CSPs can be solved only using inference, without any search. For example the game of Sudoku.

12.6 Local search

Starting from a complete assignment and trying to improve it with "local moves" using optimization techniques.

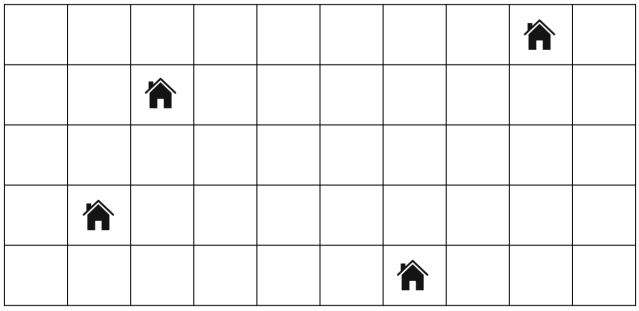


Where h represents the number of attacks between queens. The goal is to minimize h.

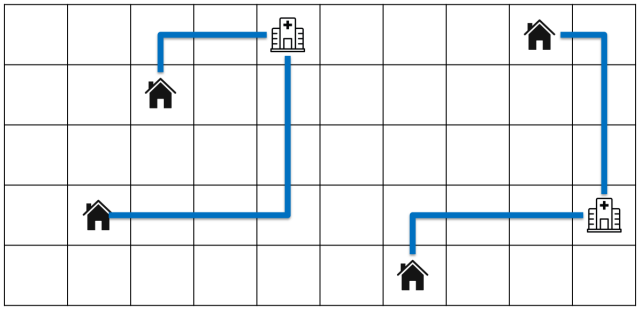
Optimization is the problem of choosing the best option from a set of options. Search algorithms solve optimization problems by maintaining a set of different paths (options) that they are simultaneously exploring. Local search algorithms maintain a single state (option) and search by moving to a neighboring state They are useful when we don't care about the path, but are simply looking for the solution (not for the path to reach it).

In local search we want to know how the solution looks like not how to reach it.

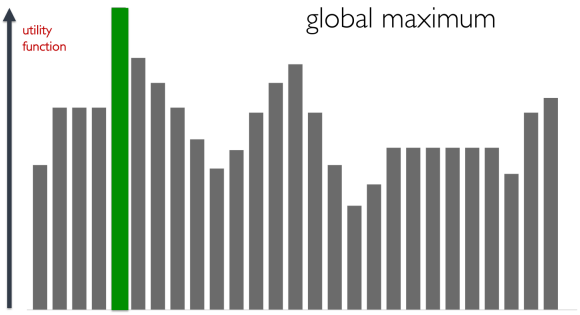
Example: homes and hospital

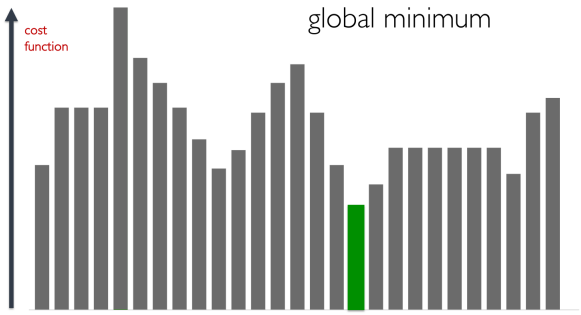


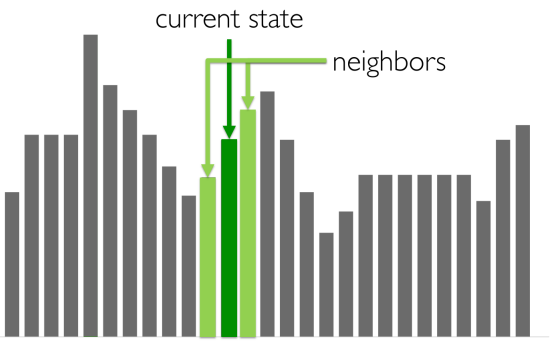
We want to find a way to place two hospitals on this map that minimizes the sum of the distances between the houses and the closest hospitals. We compute the cost using the Manhattan distance.



Cost of the configuration (state) is 4+4+3+6=17 because if we place the hospitals in these positions, the cost using the Manhattan distance is 17.



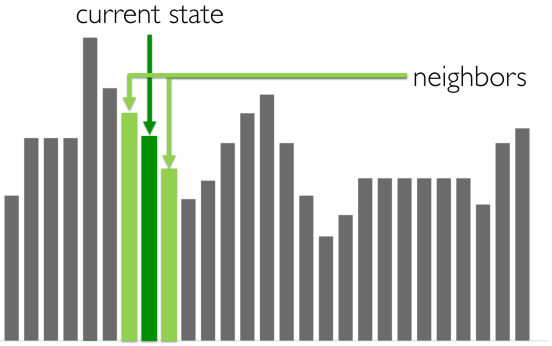


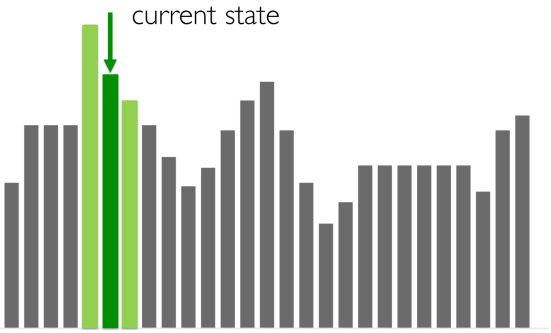


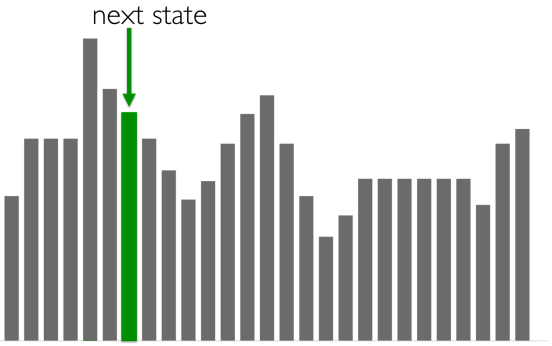
Neighbors are states that are close to the current state but slightly different and thus might have a slightly different value. In the case of the hospitals, a neighbor solution might move one or more hospitals to one position. Thus, we keep the current state and slightly move from it by exploring its neighboring states.

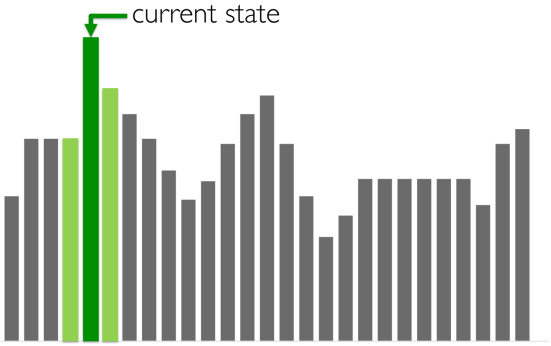
12.6.1 Hill climbing

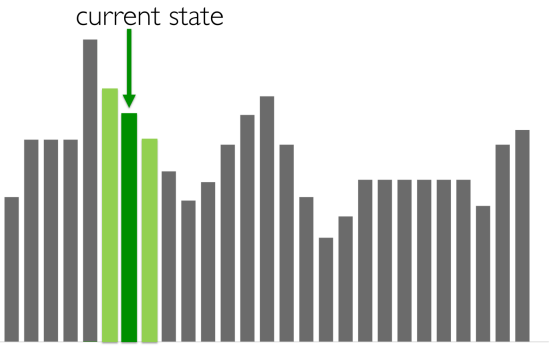
In hill climbing, if I am trying to find the global maximum, I will move to the neighbor with the highest value. Iteratively, I will evaluate the new neighbors and select that with the highest value. Until all the neighbors represent worse options than the current state. In this case, the current state is output as the maximum found by hill climbing.

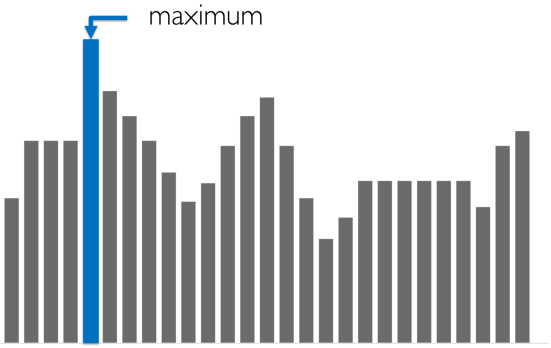










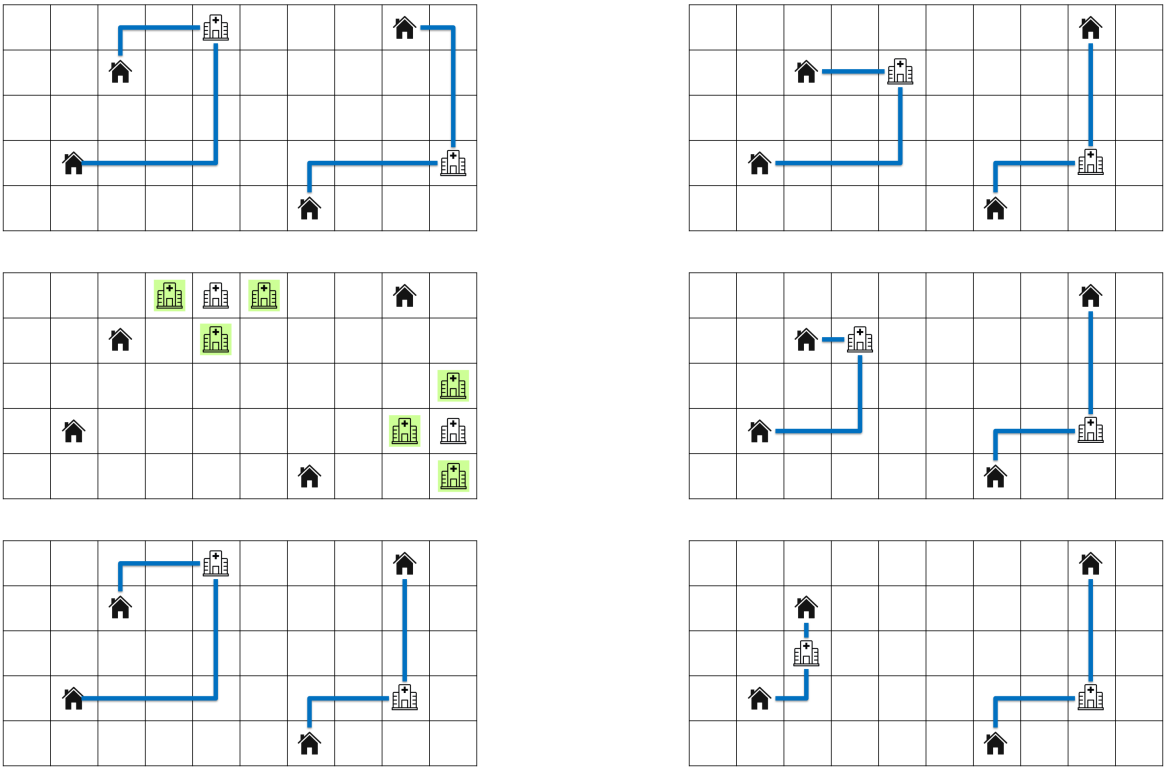


The actual algorithm of Hill-climbing is:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

Now we apply hill climbing to the hospital problem:



Hill climbing might converge to a maximum that is not the global maximum... Hill climbing variants and other optimization methods

- Steepest-ascent: Choose the highest-valued neighbor
- Stochastic: Choose randomly from higher-valued neighbors
- First-choice: Choose the first higher-valued neighbor
- Random-Restar: Conduct hill climbing multiple times
- Local-beam search: Choose the k highest valued neighbors

Other optimization approaches include: Simulated annealing, Tabu search, Genetic algorithms, Linear programming.

CSPs could be viewed as optimization problems with: objective function = number of violated constraints. Optimal solution has objective function = 0, meaning that no constraint is violated (consistency). How do you ensure that the solution of the optimization problem is also complete (all variables are assigned a value)? **Using local search, CSPs could be viewed as optimization problems with a complete-state formulation: the state of the optimization problem has all the components of a CSP solution, but not in the right place**.