Lecture 2:
Search in Games

- ECS7002P – Artificial Intelligence in Games
- Raluca D. Gaina – r.d.gaina@qmul.ac.uk
- Office: CS.335

http://gameai.eecs.qmul.ac.uk

Queen Mary University of London

Outline

- ❑ Game Trees and Uninformed Search
- ❑ Best-First Search (A*)
- ❑ Monte Carlo Tree Search
- ❑ Evolutionary Computation and RHEA

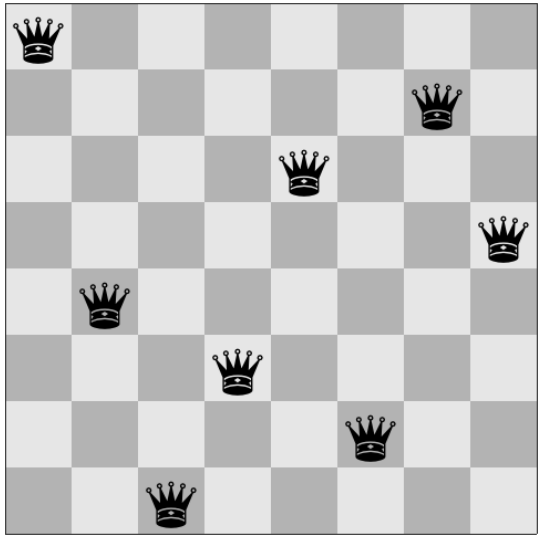
Search in Games

Game Trees and Uninformed Search

Search

Almost all AI problems can be seen as Search. If you can represent solutions to a problem in any form (i.e. an array of numbers) you "just" need to find the values for that representation that provide a good solution. In fact, you'll typically be looking for the **optimal** solution (although in many cases a **sub-optimal** solution would be just fine).

Example: the N-Queens problem. The goal is to place N queens in a NxN board so that none of them threatens any other. For instance, for N = 8:



A possible representation of potential solutions could be an array of 8 numerical values that indicate the positions of the queens (where values $\in$ [1; 64]). For instance:

[1; 15; 21; 32; 34; 44; 54; 59] is a **valid** solution.
[2; 15; 21; 32; 34; 44; 54; 59] is **not** a valid solution.

Q? What's the search space size?

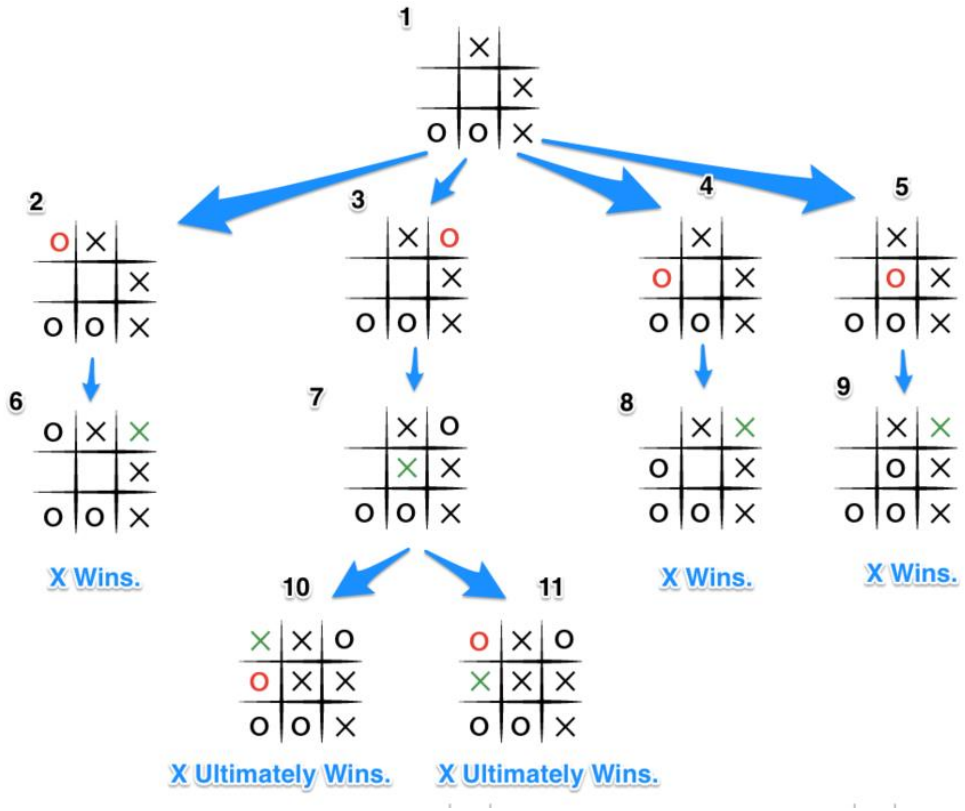
Tree Search

Tree Search is a category of search methods that uses a tree structure to navigate through possible solutions. In Tree Search, the root of the tree represents the state where the search starts. Each edge from this state is a decision (or, in a game, an action a player takes) that leads to another state in the search space.

On the right, a game tree of Tic Tac Toe:

Typically, more than one action can be taken from each state (unless it's terminal) and it's possible that there are different routes to arrive at the same state.

Tree Search algorithms mainly differ in which branches are explored and in what order.



The Eight or Sliding Puzzle

A puzzle in which the objective is to sort all 8 numbers in a 3 3 grid, from left to right and top to bottom, leaving the empty grid on the bottom right corner. The initial state is random.

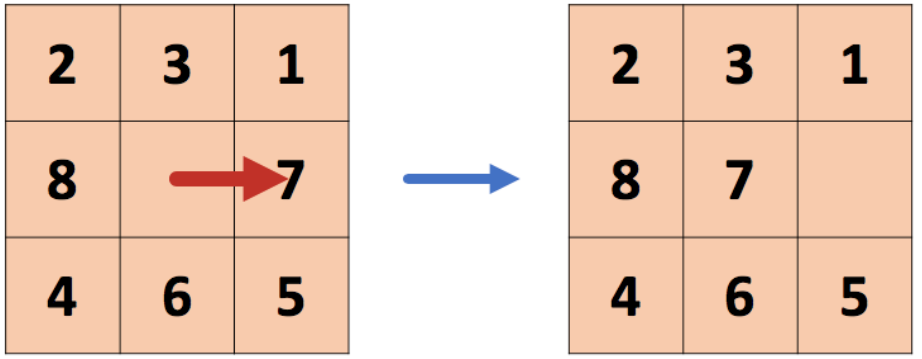
2	3	1
8		7
4	6	5

(One of) 8-puzzle starting position

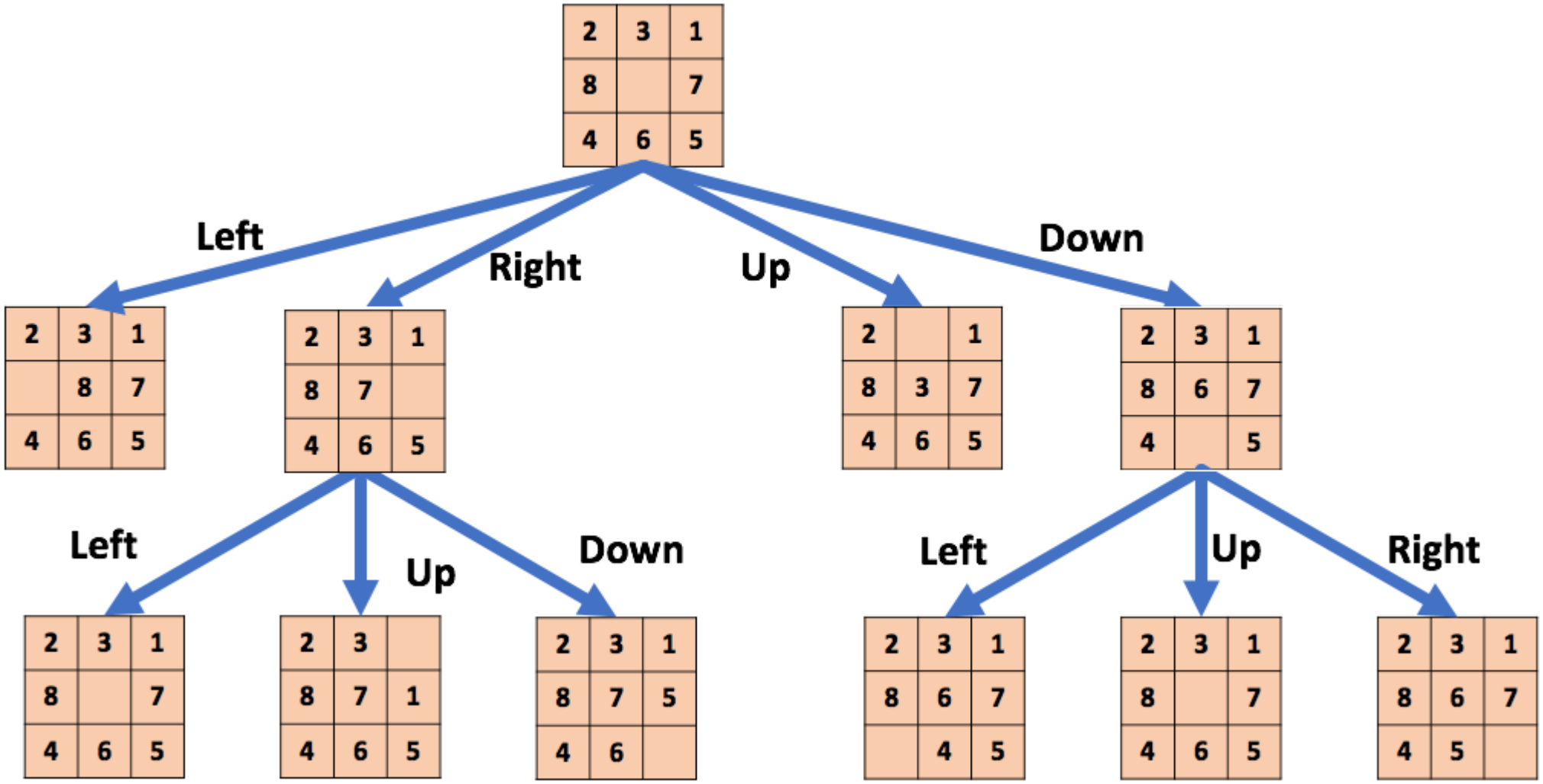
1	2	3
4	5	6
7	8	

8-puzzle goal state

At each decision point, there are up to 4 available actions: move the empty cell *Up*, *Right*, *Down* or *Left*.



The Eight or Sliding Puzzle



Uninformed Search

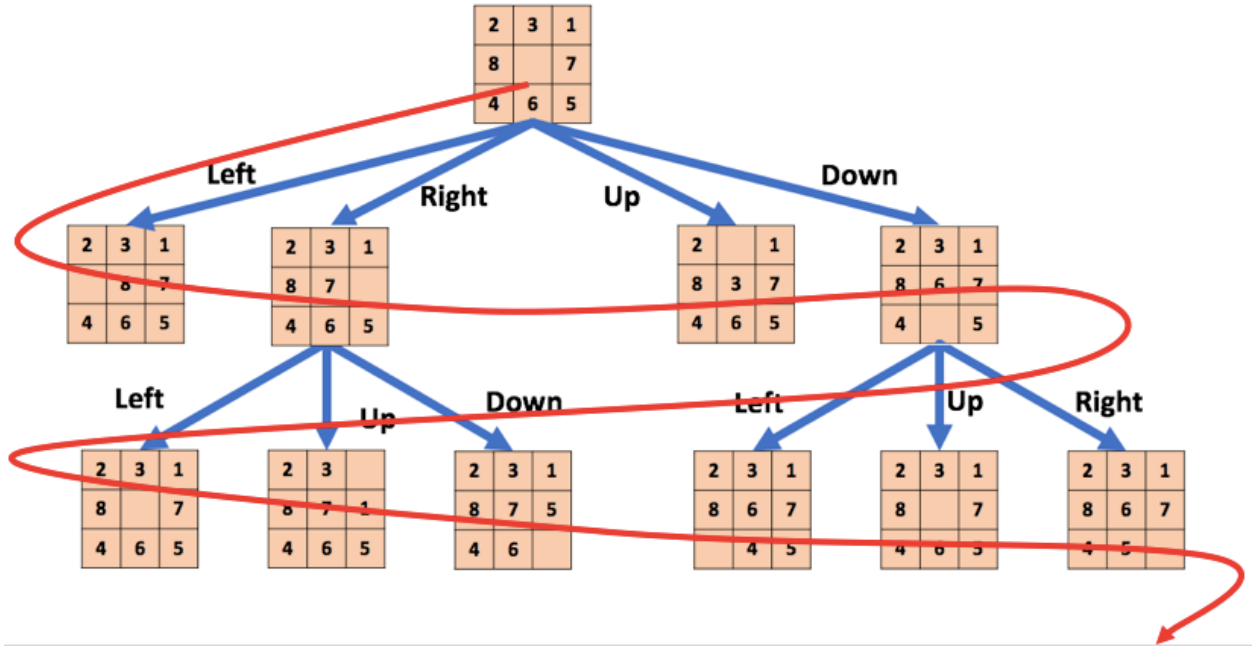
In **Uninformed Search** (also called **Blind Search**), the algorithm performs search without any information about the goal. They are brute force algorithms: they explore the search space in a predetermined way until the goal is found.

This is one of the most basics approaches that can be used. So basic they're often not even seen as AI at all. The most known are:

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-first search (BFS)

BFS explores all the actions from a single node before exploring any of the nodes after taking those actions. Therefore, all nodes of level *L-1* need to have been explored before *L* is reached.

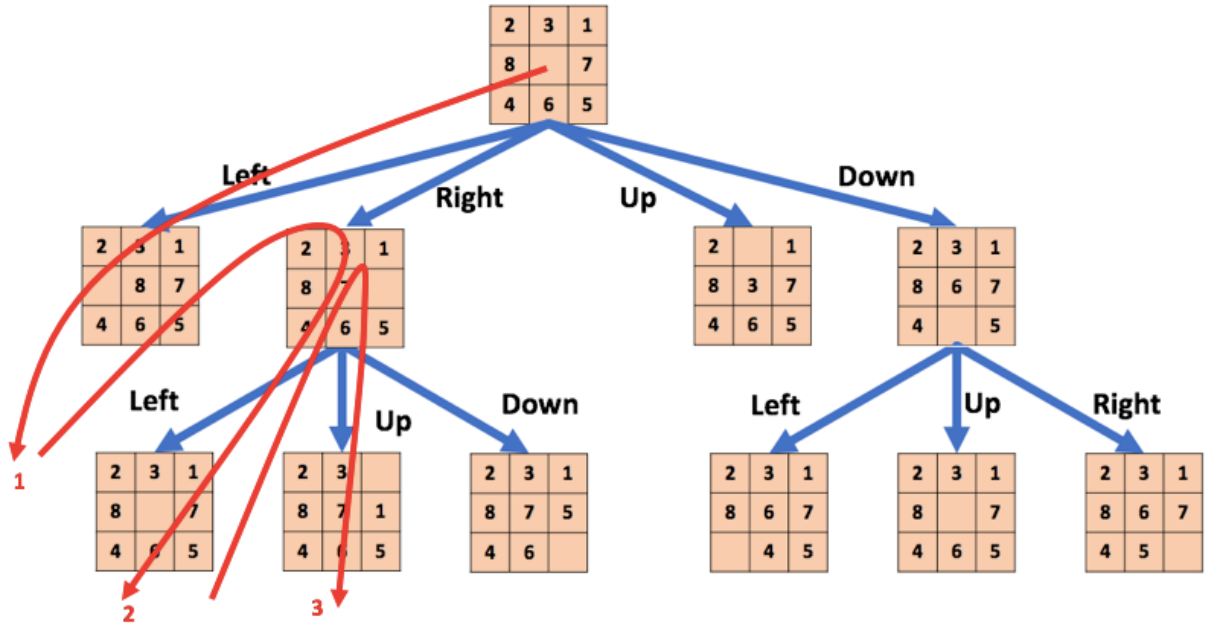


- **Advantage**: Very simple to implement. Complete.
- **Disadvantage**: Heavy memory requirements.

Q? Will this algorithm find the solution to the 8-puzzle?

Depth-first search (DFS)

DFS explores each branch as far as possible before backtracking to explore the next branch. 'As far as possible' is determined by a terminal state (which can be the goal or not).



- **Advantage**: Light memory requirements.
- **Disadvantage**: Not complete, can enter infinite loops.

Q? Will this algorithm find the solution to the 8-puzzle?

Uninformed Tree Search Variants

Depth-limited search:

- DFS with a depth limit on search.
- **Advantage**: No infinite loops.
- **Disadvantage**: It will not find the goal if it's farther than the limit.

Iterative deepening depth-first search:

- Performs iterations of depth-limited searches with increasing depth limits.
- **Advantage:** Complete. It will find the goal.

Bidirectional search:

- Two simultaneous searches: from original state towards the goal, and from the goal state to the original state.
- **Advantage**: Good time and memory requirements.
- **Disadvantage**: It's not always possible to trace back from the goal.

Outline

- ✓ Game Trees and Uninformed Search
- ❑ Best-First Search (A*)
- ❑ Monte Carlo Tree Search
- ❑ Evolutionary Computation and RHEA

Search in Games

Best-First Search (A*)

Best-First Search

Contrary to uninformed tree search, **Best-First Search** uses knowledge about the goal to conduct search. This is implemented in the form of a **heuristic**, which provides a way of establishing a preference in the order in which actions are explored from a given state.

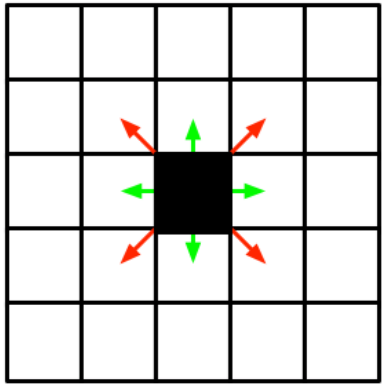
One of the most common Best-First Search algorithms is A* (pronounced 'A star'), which is commonly used in games for pathfinding.

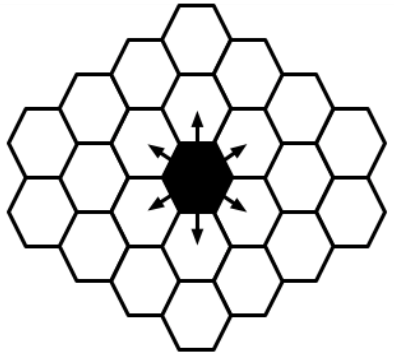
Use-case of A*: pathfinding.

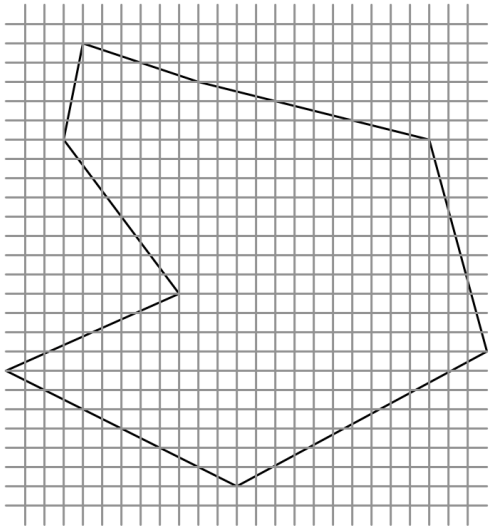
A* is used in games (mostly) to determine the shortest path between two locations in a level. However, pathfinding algorithms can't work directly with the geometry that makes up the game level. We need to construct a **graph** that we can *search*:

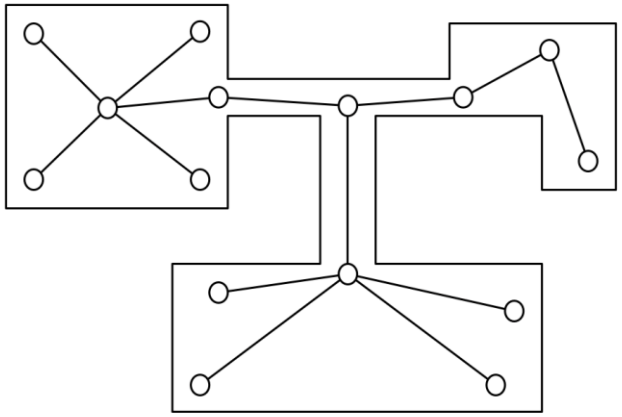
- Nodes represent points in 2D or 3D space.
- Nodes have neighbours (adjacent nodes).
- Neighbouring nodes may have different (positive) distances / costs.
- Nodes (i.e. their locations) can be constructed manually or automatically.

Pathfinding Graphs









Dijkstra

Once a grid is in place, we need to compute how to get from A to B using a pathfinding algorithm. An example of an **uninformed** search algorithm is **Dijkstra**.

Dijkstra's algorithm is a classical pathfinding algorithm that is very efficient (polynomial runtime). It finds the shortest path from a node to any other node in the graph. It is a variation of breadth first search, which orders the next nodes to be visited based on their distance to the **start node**. This distance measure is often denoted as ***g.***

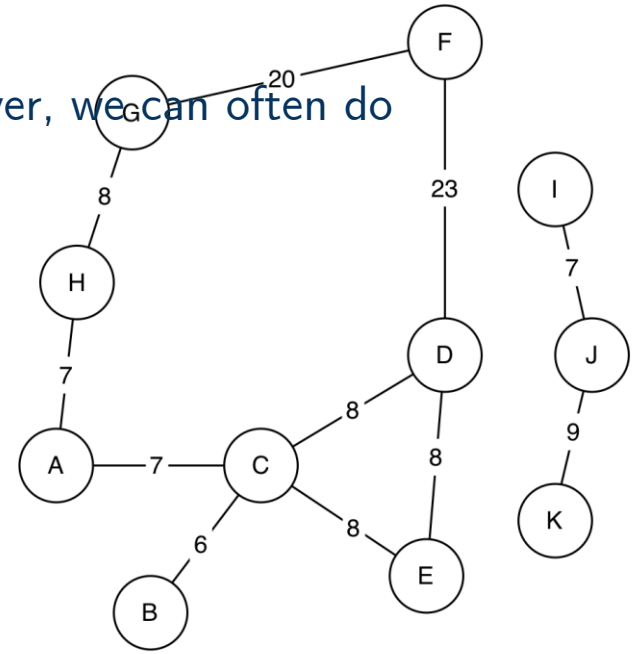
Dijkstra's algorithms makes use of a **priority queue**: a list that holds items with an associated priority. The list if always ordered such that the highest priority items are on top. The algorithm's efficiency depends crucially on the implementation of the priority queue used.

Dijkstra's C# code

```
1 public static List <N> Dijsktra (N start , N target ) {
2       PQ pq = new PQ();
3       start. visited = true;
4       start.g = 0;  // We only use g in Dijkstra
5       pq.Add (start);
6
7        while ( ! pq.IsEmpty () ) {
8             N currentNode = pq.Get ();  // Removes the node from PQ.
9
10          if ( currentNode.isEqual(target) )
11               break;
12
13          foreach ( E next in currentNode.adj ) {
14              double currentDistance = next.cost ;
15
16               if ( ! next.node.visited ) {
17                    next.node.visited = true ;
18                    next.node.g = currentDistance + currentNode.g;
19                    next.node.parent = currentNode ;
20                    pq.Add ( next.node );  // Sorting according to g
21               }
22               else if ( currentDistance + currentNode.g < next.node.g ) {
23                         next.node.g = currentDistance + currentNode .g;
24                         next.node.parent = currentNode ;
25                    }
26               }
27          }
28      }
29      return ExtractPath(target);
30 }
31
```

From Dijkstra to A*

Dijkstra is complete and always finds the shortest path. Also, it is efficient. However, we can often do better.



- **Q?** How could we improve Dijkstra?
- By adding knowledge about the goal.

Dijkstra uses the distance travelled so far and utilises a priority queue to consider nodes in the graph with the smallest distance from the start found so far. We would expect to perform better if we could also guess how close a node is to the target (for point-to-point distances).

A* uses a **heuristic** (often denoted as ***h***) to estimate the utility of a node with respect to the target. This allows the algorithm to explore the graph towards the more promising regions. For this to work, we require the following:

- The heuristic must be **admissible**: An admissible heuristic never over-estimates the distance from A to B.
- The closer the heuristic is to the actual distance, the better A* performs.

A* (part 1)

A* keeps a list of "open" nodes, reachable from the already explored nodes, but which that have not yet been visited. For each one of these "open" nodes ***N***, an estimate of the distance to the goal is made. The next node/state to visit is the one with the smaller aggregated cost, which is the sum of the cost from the origin (***g***) and the cost to the destination (***h***).

$$cost_N = g + h$$

This estimation is the **heuristic**: we use information about the goal to guide the search.

```
1 public enum Heuristics { STRAIGHT, MANHATTAN };
2
3 public static List <N> AStar (N start, N target, Heuristics heuristic)
4 {
5       PQ open = new PQ();
6       List <N> closed = new List <N >();
7       start.g = 0;
8       start.h = GetHValue(heuristic, start, target);
9       open.Add(start);
10
```

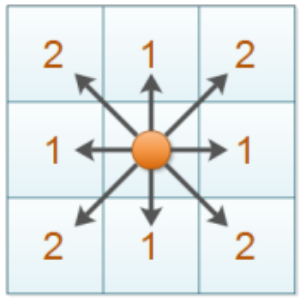
A* (part 2)

```
11      while ( ! open.IsEmpty () ) {
12           currentNode = open.Get ();
13           closed.Add(currentNode);
14           if ( currentNode.isEqual(target) ) break;
15
16           foreach ( E next in currentNode.adj ) {
17                double currentDistance = next.cost;
18
19                if ( ! open.Contains(next.node) && ! closed.Contains(next.node) ) {
20                     next.node.g = currentDistance + currentNode.g;
21                     next.node.h = GetHValue(heuristic, next.node, target);
22                     next.node.parent = currentNode ;
23                     open.Add(next.node);  // Sorting according to (g+h)
24                }
25                else if ( currentDistance + currentNode.g < next.node.g ) {
26                          next.node.g = currentDistance + currentNode.g;
27                          next.node.parent = currentNode;
28
29                          if ( closed.Contains(next.node) )
30                               closed.Remove(next.node);
31
32                          if ( open.Contains(next.node) )  // Updates position in PQ
33                               open.Remove(next.node);
34
35                          open.Add(next.node);
36                     }
37           }
38      }
39      return ExtractPath ( target );
40 }
41
```

Heuristics for A*

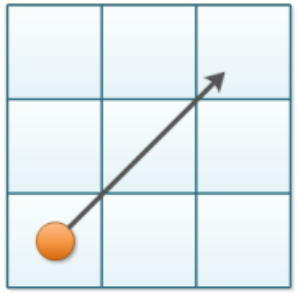
Some types of heuristics:

Manhattan Distance



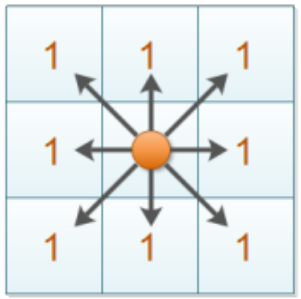
$$|x_1 - x_2| + |y_1 - y_2|$$

Euclidean Distance



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Chebyshev Distance



$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

(lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance)

To use A*, we need a heuristic. In the simplest case, we can say $h = 0$. A* then behaves identical to Dijsktra. In order to improve the algorithm, we should choose an admissible heuristic as close as possible to the real distance.

Q?

- Rectangular grid, 4-way connectivity: which are admissible?
- Rectangular grid, 8-way connectivity: which are admissible?
- What other attribute is important for a heuristic to be useful?

Some Facts about A*

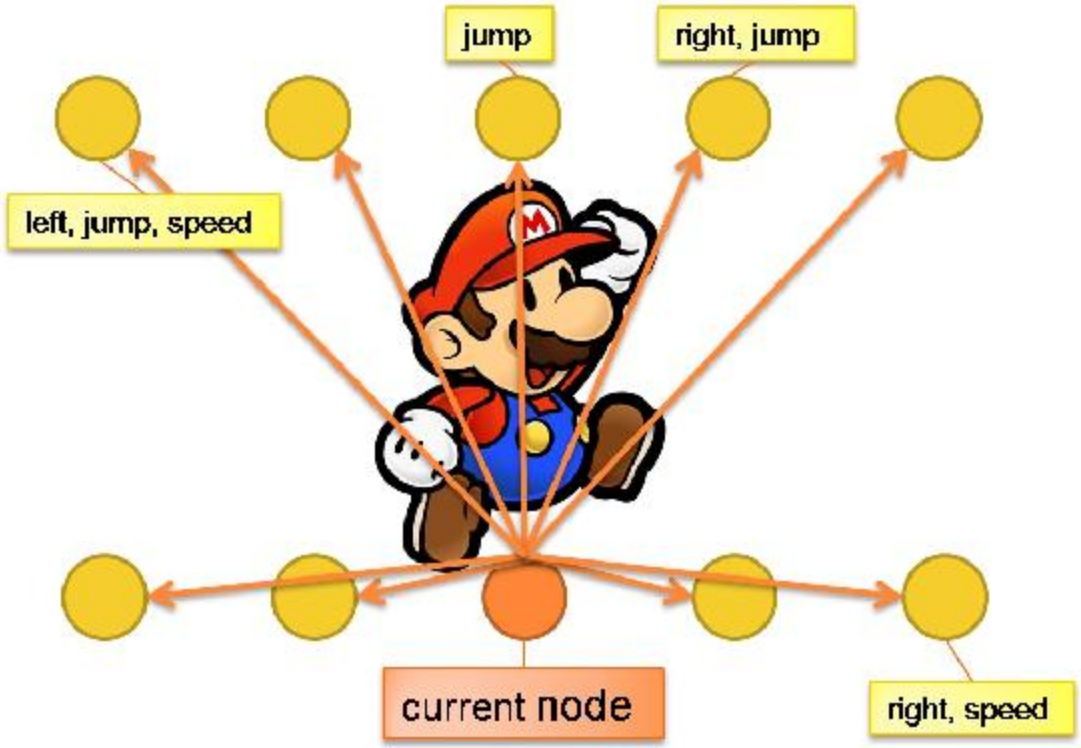
Some facts from theory.stanford.edu/~amitp/GameProgramming/Heuristics.html:

- If h(n) is 0, only g(n) matters, and A* equals Dijkstra's algorithm.
- If h(n) is never more than the actual distance, then A* is guaranteed to find a shortest path. The lower h(n), the more node A* expands.
- If h(n) is exactly equal to the actual distances, then A* will only follow the best path and never expand anything else.
- If h(n) is sometimes greater than the cost of moving from n to the goal, then A* is not guaranteed to find a shortest path.
- If h(n) is very high relative to g(n), then only h(n) plays a role, and A* turns into Best-First-Search.

http://en.wikipedia.org/wiki/A*_search_algorithm

A*

A* is not only used to navigate worlds, however. As the previous methods, it can also be used to search in the game tree. That is, A* can be used for **planning**.



See it in action: `https://youtu.be/DlkMs4ZHHr8`

Outline

- ✓ Game Trees and Uninformed Search
- ✓ Best-First Search (A*)
- ❑ Monte Carlo Tree Search
- ❑ Evolutionary Computation and RHEA

Search in Games

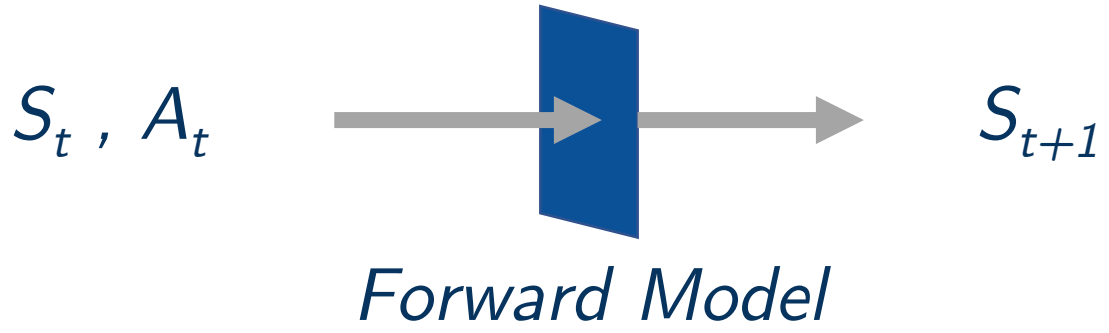
Monte Carlo Tree Search

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a highly selective best-first search method. MCTS builds an asymmetric tree search exploring the most promising part of the search space. It's an algorithm that works considerably well with *high branching factors* (many actions available from a given state).

But, if the game tree is large, how does it manage to search efficiently? An important part of the algorithm is based on sampling **randomly** actions from states: it requires a **Forward Model (FM)**.

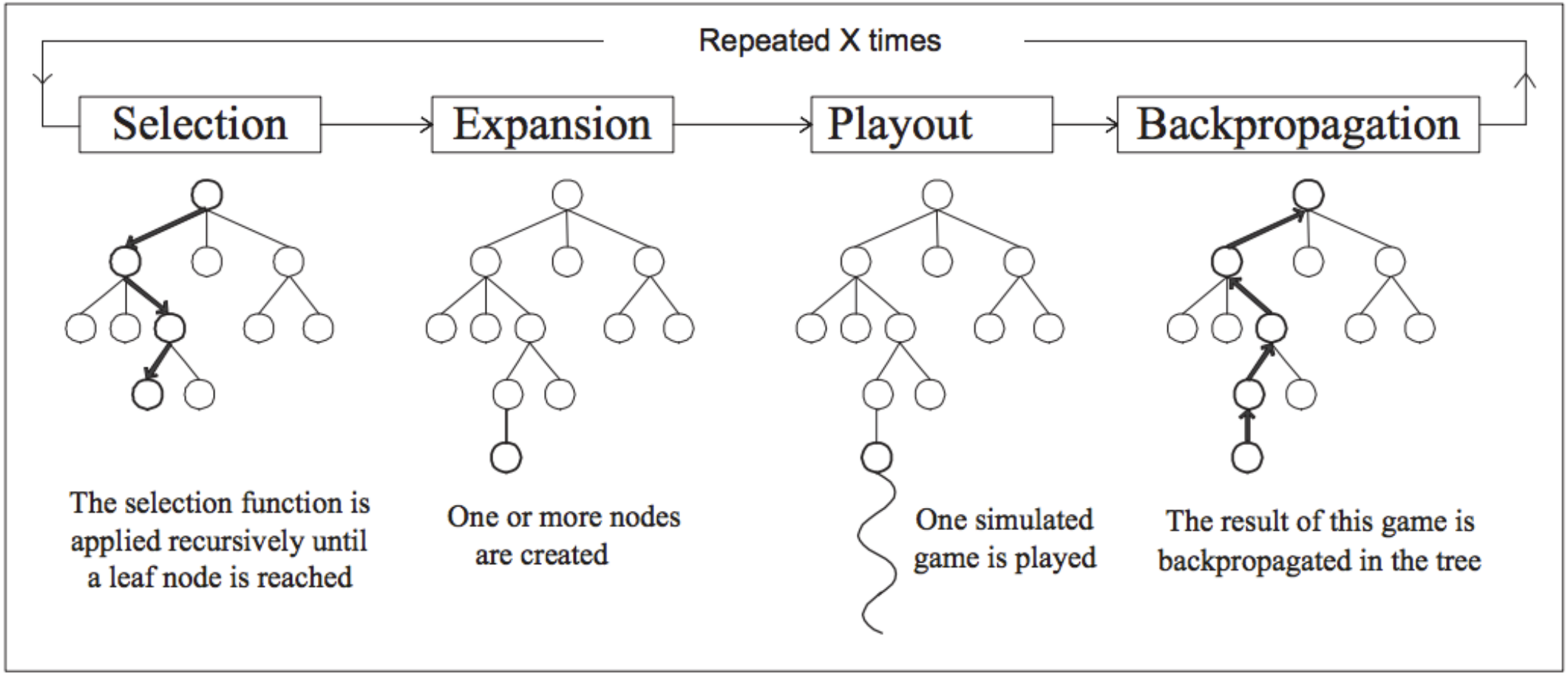
A **Forward** or **Predictive Model** is an internal simulator of the game that the AI player has access to. The agent uses this FM to simulate the effect (i.e. the next game state) of applying certain actions on a given state



Given a current state S_t and an action A_t applied from S_t, the FM will provide the next state S_{t+1}.

Monte Carlo Tree Search

MCTS is an algorithm that consists of repeating a sequence of 4 steps:



Monte Carlo Tree Search

Each node of the tree represents a game state. Each node holds the following statistics:

- *N(s)*: Number of times the state *s* has been visited by the algorithm.
- *N(s,a)*: Number of times the action *a* has been played from state *s*.
- *Q(s,a)*: An estimation of how good it is to play action *a* from state *s*.

These statistics are updated, in certain nodes, at every iteration.

Wait, *how good it is to play action a from s*? *Q(s,a)* is the average (i.e. an estimation) of rewards seen when playing action *a* from *s*.

Rewards (R) indicate how good a state is:

- If the state is **terminal** (the game is over), this is simple: did I win? R = 1. Did I lose? R = 0.
- If the state is **not terminal**, we need a function that tells us how good this is. This is typically a **heuristic**, designed with knowledge of the goal state in mind, R = *f(s)*.

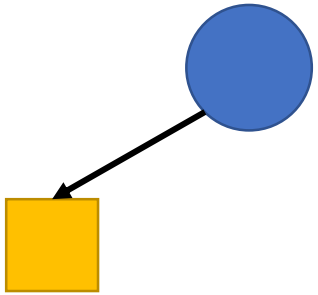
Monte Carlo Tree Search – Step by Step

The algorithm starts with the current game state. Only one node in the tree. In this example, we assume there are **3 possible actions** from each state.

During the agent's thinking time, it will have a certain budget (shorter if it's a real-time game, longer if it's not) to run as many iterations as possible, before giving a final action **recommendation**.

Monte Carlo Tree Search – Step by Step

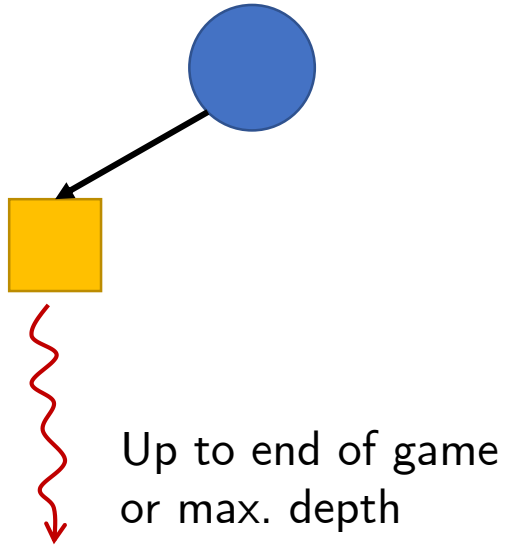
In the original algorithm, the **selection** step can only be applied when all children nodes from the current state have been visited. This is not the case at the start, so we move to the next step: **expansion**.



We choose one action (at random, or in a predefined order), use the Forward Model to roll the state forward to the next state, and add it as a new node to the tree.

Monte Carlo Tree Search – Step by Step

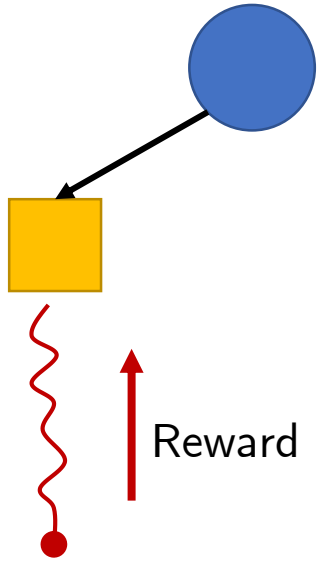
The **playout** (also called **simulation** or **rollout**) step plays actions uniformly at random from the state added at the previous step. This is known as the **Default Policy**.



Until when? If it's able to find a terminal state, random actions are executed until then. Otherwise, a **maximum depth** is established and the rollout ends after a number of random moves.

Monte Carlo Tree Search – Step by Step

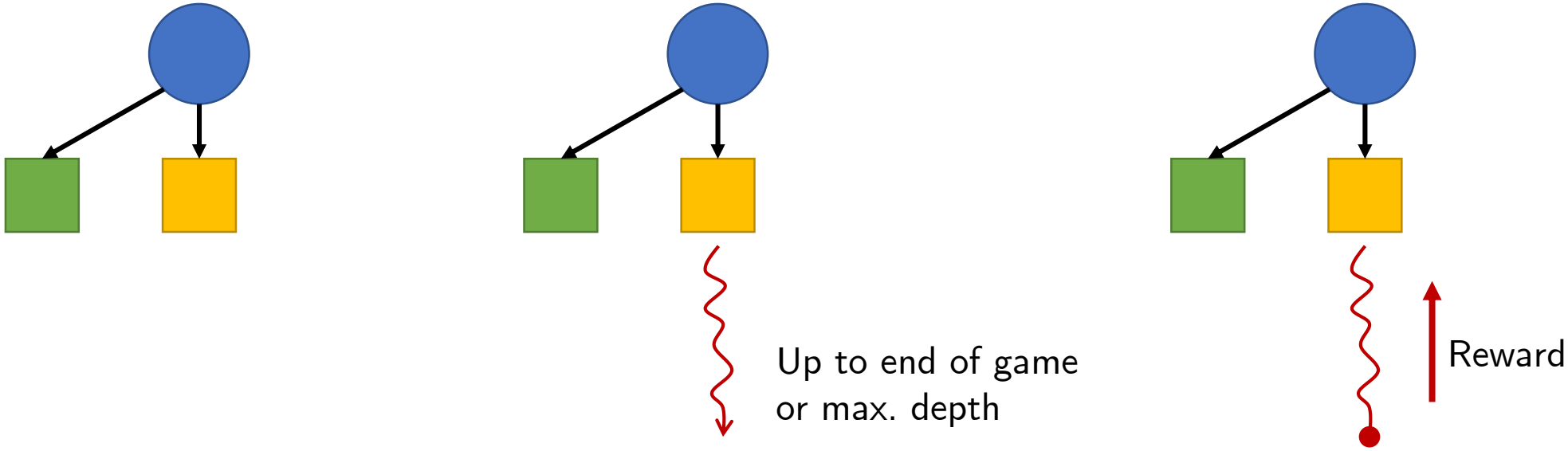
We capture the **reward** of the state at the end of the rollout. Either a victory condition or a **heuristic** on the state. The **backpropagation** phase takes this reward to the nodes **visited** in this iteration.



This reward updates the value of $Q(s,a)$ of all visited state-action pairs. It also updates $N(s)$ and $N(s,a)$ in these nodes.

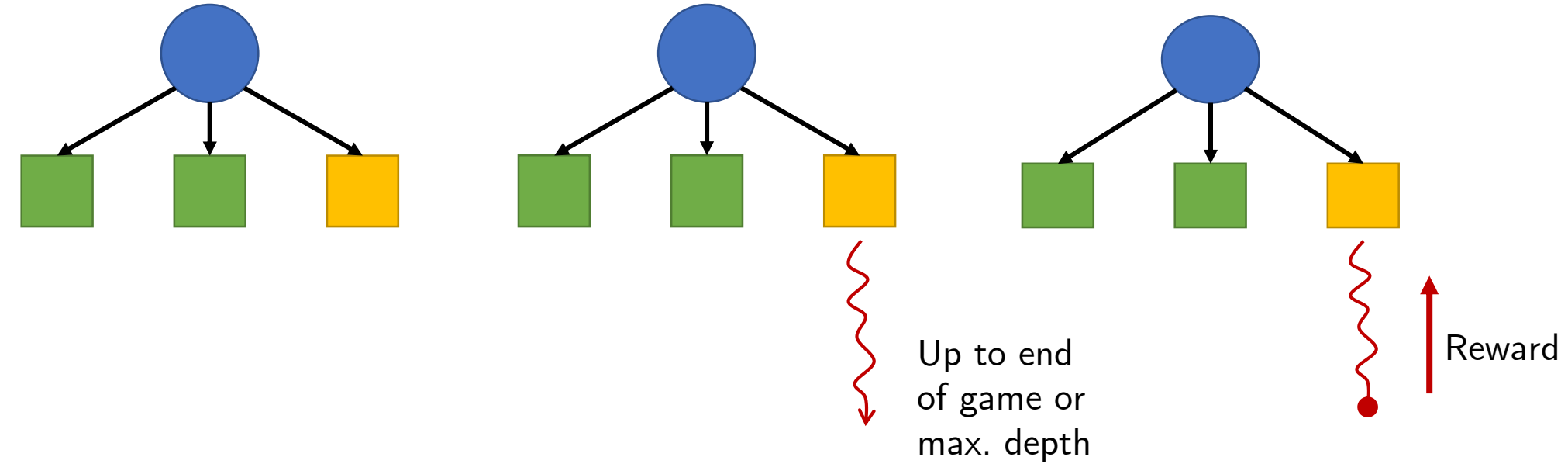
Monte Carlo Tree Search – Step by Step

The first iteration is now over. We still can't do selection from the root node (only one action has been explored from this state). We expand, simulate and backpropagate in this second iteration.



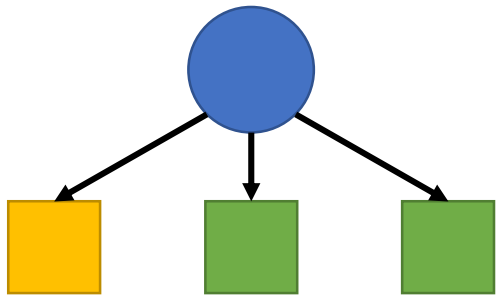
Monte Carlo Tree Search – Step by Step

And again for the third iteration:



Monte Carlo Tree Search – Step by Step

On the next iteration we can apply the **selection** step. This selection is made according to a **Tree Policy**. This tree policy is key in the MCTS algorithm. The objective is to balance between exploiting the best action seen so far and exploring actions that haven't been seen often enough (thus our estimate could be inaccurate).



[Tree policy]

The most popular, default in MCTS, is Upper Confidence Bounds (UCB1). The action selected is the one that maximizes the following expression:

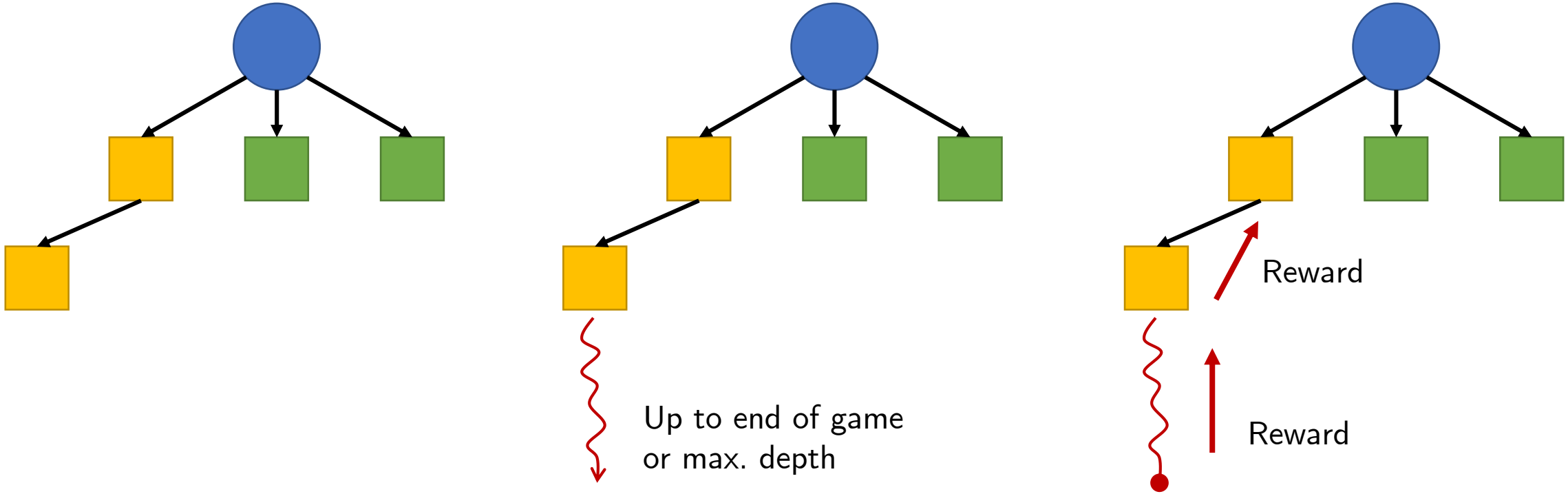
$$a = \underset{a \in A(s)}{argmax}\, Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \quad (1)$$

$A(s)$: set of available actions from state s.

C: exploration constant. $C = \sqrt{2}$ if $R \in [0,1]$.

Monte Carlo Tree Search – Step by Step

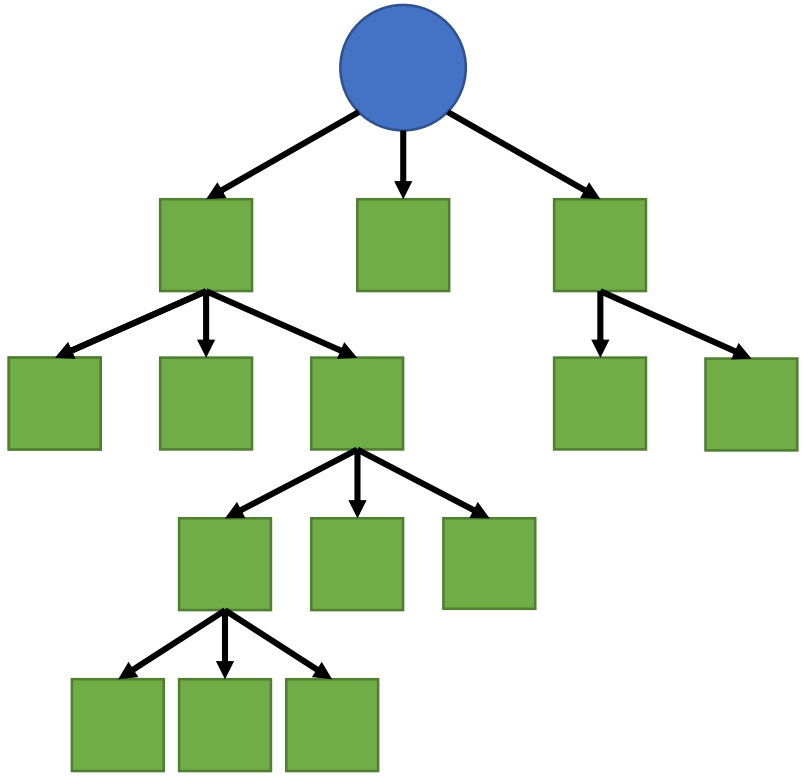
After selecting a node, we continue with the next steps: expand a new node from there, rollout and backpropagation.



Note that this backpropagation updates the statistics of **3 nodes** (including root node).

Monte Carlo Tree Search – Step by Step

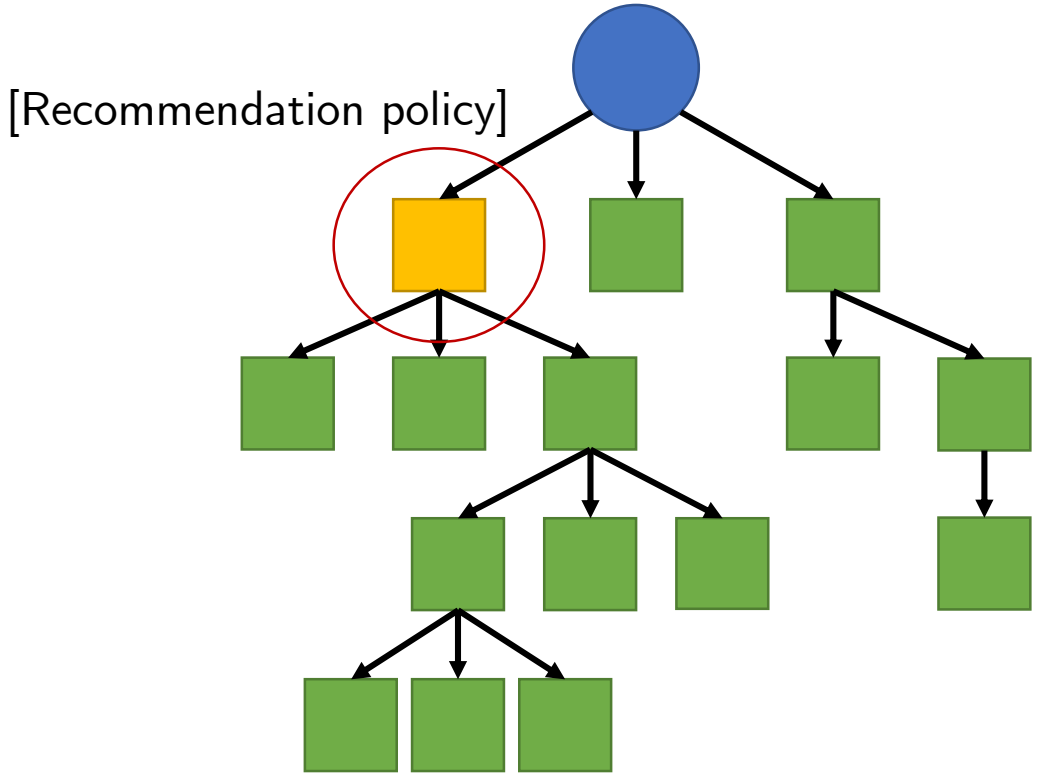
We repeat this procedure iteration after iteration. Because of the tree policy, the tree grows asymmetrically, exploring the most promising parts of the search space.



Where to stop? When the budget is over. The more iterations, the better, but MCTS is an **anytime** algorithm: it can be stopped at anytime and it'll provide a sensible action (in contrast, for instance, A* *must* end).

Monte Carlo Tree Search – Step by Step

When time or a number of iterations is up, MCTS stops and recommends one action *a* to execute in the real game.



This is the **Recommendation Policy**, and it can take several forms: the *action* that maximizes *Q(s,a)* or *N(s,a)* or others (even, UCB1 again).

Outline

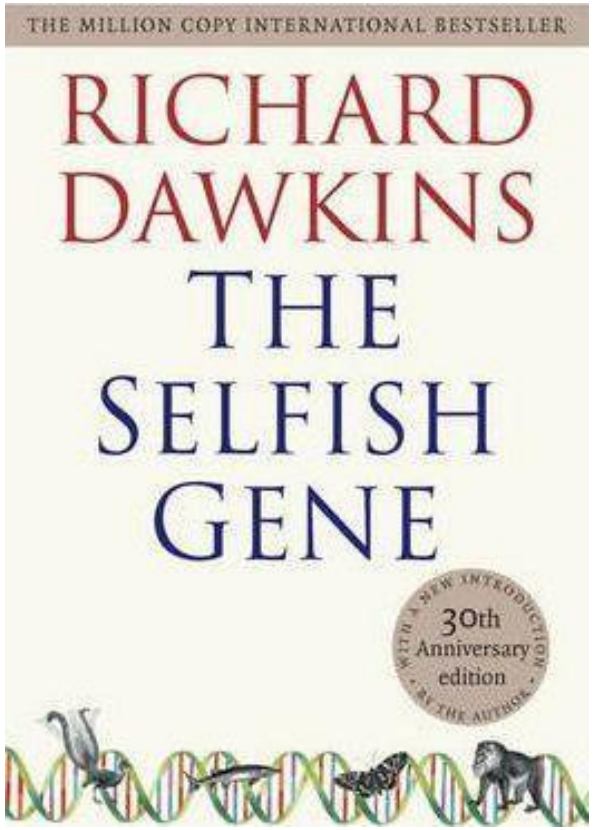
- ✓Game Trees and Uninformed Search
- ✓Best-First Search (A*)
- ✓Monte Carlo Tree Search
- ❑Evolutionary Computation and RHEA

Search in Games

Evolutionary Computation and RHEA

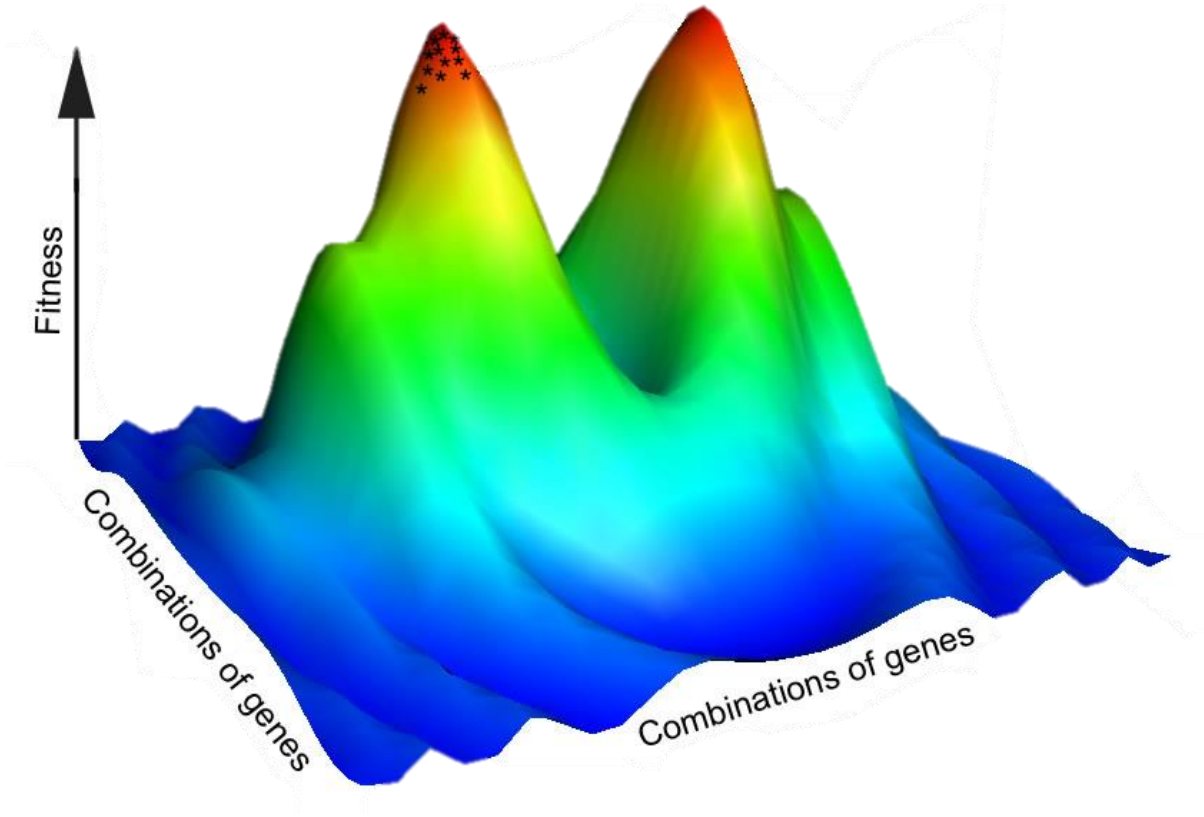
Evolutionary Computation

One of the biggest sub-fields of AI search methods is **evolutionary computation**. These methods are based on the concept of **Natural Selection**, that evolves a series of solutions, applying **genetic operators** such as reproduction, mutation, recombination and selection.



Evolutionary Algorithms

An Evolutionary Algorithm (EA) is an **optimization** algorithm that considers complete solutions to a problem. As such, it requires an **objective** function (also referred to as utility, evaluation or *fitness* function) that can assign a numeric value to a solution, which can then be maximized or minimized. An optimization algorithm can be seen as a process that searches, in the space of solutions, for those with the highest (or lowest) value of that utility.



Evolutionary Algorithms

An important concept of EAs (and optimization in general) is the **representation** of the solutions. In general, these solutions are represented as a vector or array of numerical values, or as a string of characters or enumerators. The mapping from these variables to the actual solution has a big impact on the efficiency and efficacy of the search algorithm.

- The representation of a solution (i.e. array of numerical values) is called **Genotype**.
- The *expression* of that solution in the problem (i.e. behaviour of an agent) is called **Phenotype**.

Evolutionary Algorithms, in a nutshell:

- Representation of solutions to a given problem as individuals. If multiple individuals are considered, they form a population.
- Each individual is evaluated in the problem, and assigned a *fitness* value.
- The *fitness* indicates *how good* this solution is to the given problem.
- The solutions *evolve* during several *generations*, creating new individuals through reproduction, mutation, recombination and selection.
- Elitism: the population could be forced to not discard the best individual(s) of a generation.

Hill Climber

Possibly the simplest optimization algorithm is the **Hill Climber**, which executes a type of **Local Search**. It works as follows:

1. **Initialization**: create a solution *s* by choosing a random point in the search space and evaluate its fitness *f(s)*.

2. **Expansion**: generate and evaluate **all** possible neighbours of *s*. This is, all possible single modifications that you can do to *s* so you obtain an *s'* that differs from *s* by at most a certain given **distance**.

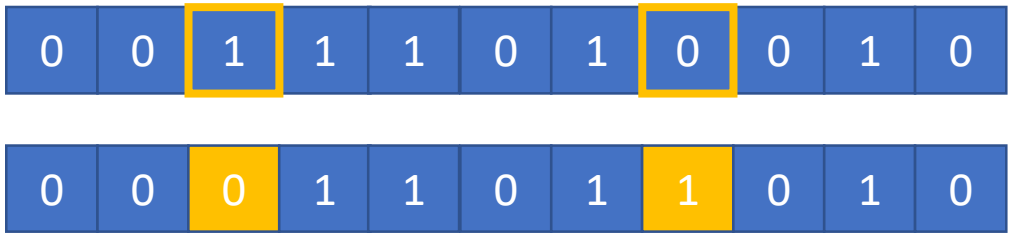
3. **Iterate:** if at least 1 neighbour fitness *f(s')* is better than *f(s)*, replace *s* with *s'* that has the best *f(s')* observed, and return to **step 2**.

4. **Return**: if none of the neighbours fitness *f(s')* is better than *f(s)*, exit the algorithm and return *s*.

Stochastic Hill Climber: Mutation

Local Search (*Deterministic* Hill Climber) is only applicable if the representation only produces a small number of neighbours per point. If that's not the case, it's recommended to use variants of hill climber.

Stochastic (or *Randomized*) Hill Climber relies on one of the most basic EA operators: **mutation**. Mutation consists of a small, random change of a solution.



Given a solution of length n, mutation is typically applied as follows:

1. Each position has a probability of $1/n$ to be modified. OR Choose 1 position to be mutated.
2. The position to be mutated changes to a different value.

Stochastic Hill Climber

The algorithm for Stochastic Hill Climber follows these steps:

1. **Initialization**: create a solution *s* by choosing a random point in the search space and evaluate its fitness *f(s)*.

2. **Mutation**: by using mutation, generate *s'*, one of the possible neighbours of *s*, and evaluate it to obtain *f(s')*.

3. **Iterate:** if *f(s')* is better than *f(s)*, replace *s* with *s'* and return to **step 2**.

4. **Return:** if the computational budget is over, or *f(s)* is good enough, exit.

The main limitation (again!) is that it can get stuck in **local minima**. Random re-starts is a typical solution to this problem, and also adding with a small probability the possibility of moving to a solution with a slightly worse fitness (*simulated annealing*). However, a very popular response is the use of a **population**.

Evolutionary Algorithms (EA)

A typical Evolutionary Algorithm:

1. **Initialize** the population of N individuals, t = 0.
2. For each generation t, until convergence or max T:

    2.1. Evaluate the population P_t (i.e. evaluate all its individuals).
    2.2. Promote the best E individual(s) from P_t to the next generation's population (P_{t+1}).
    2.3. For each $N - E$ position in P_{t+1}:

        2.3.1. Select two parents from population P_t.
        2.3.2. Create new individual(s) from them with **crossover**.
        2.3.3. Mutate the individual(s).
        2.3.4. Insert the new individual in the next generation P_{t+1}.

    2.4. Advance to the next generation ($P_t \rightarrow P_{t+1}$).

3. **Return** best individual from the final population P_T.

Evolutionary Algorithms (EA)

Initialization:

- Each individual is a sequence (array) of values (genes), which cold be initialized as 0, or random, or biased with some prior knowledge.

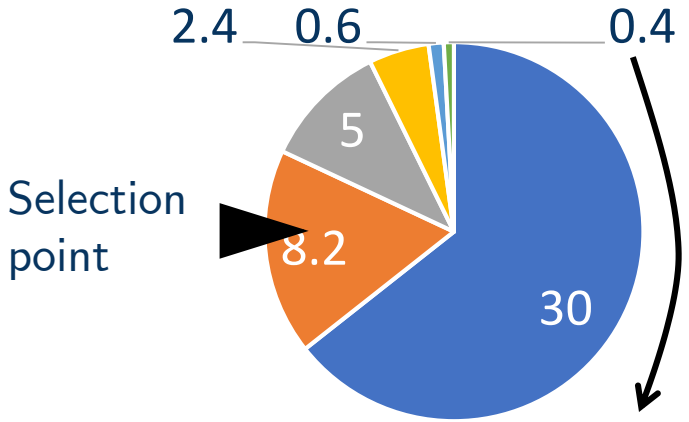
Evaluation of the population:

- Each individual represents a solution to the problem.
- A function takes this individual and assigns a numeric score to it (*fitness*) to be minimized or maximized.

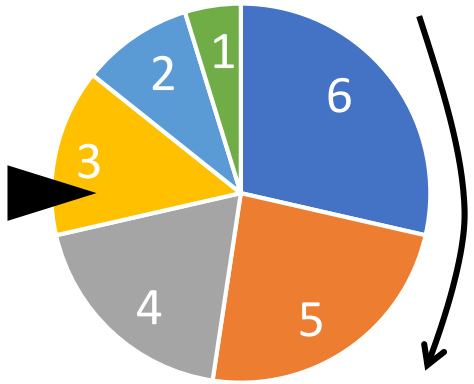
Evolutionary Algorithms (EA)

Selection:

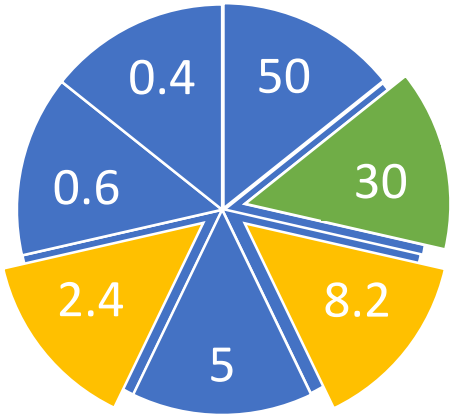
- **Roulette**: select an individual from the population at random, proportional to its fitness.
- **Rank**: like roulette, but each fitness value is assigned a rank, and the selection is proportional to the associated rank for each individual. Reduces the effect of large differences in fitness between individuals.
- **Tournament**: select *M* individuals at random from the population. Then, select the best one out of the *M* individuals (highest fitness). Smaller values of *M* (= 2, 3, 4) provide *stronger evolutionary pressure*, and *tend* to work better.



Roulette



Rank

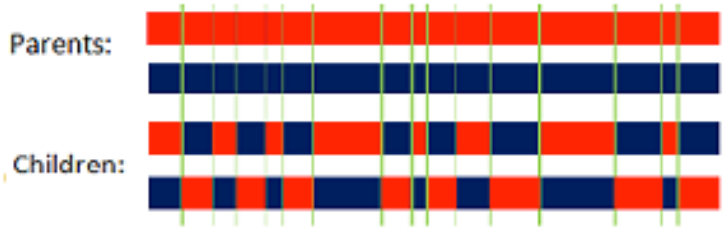


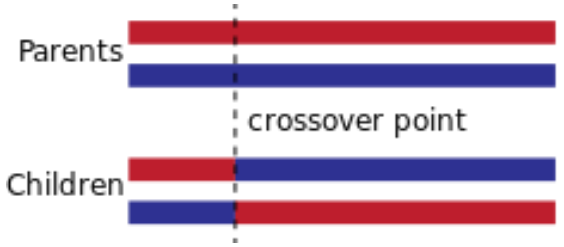
Tournament

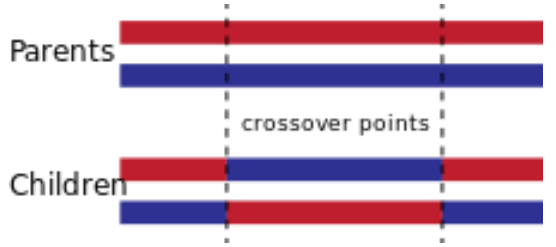
Evolutionary Algorithms (EA)

Crossover: combining genes from 2 (or more) individuals to create a new one (or more).

- Uniform: taking a gene from each parent, at random.
- Single-point: choose a position in the individual at random. Then, take the first half from one parent, and the second half from another.
- N-point: choose N positions at random and repeat as above.







Elitism: select the best E individuals from the population and directly promote them to the next generation directly. Typically, E takes a small value to avoid a narrow convergence of the population.

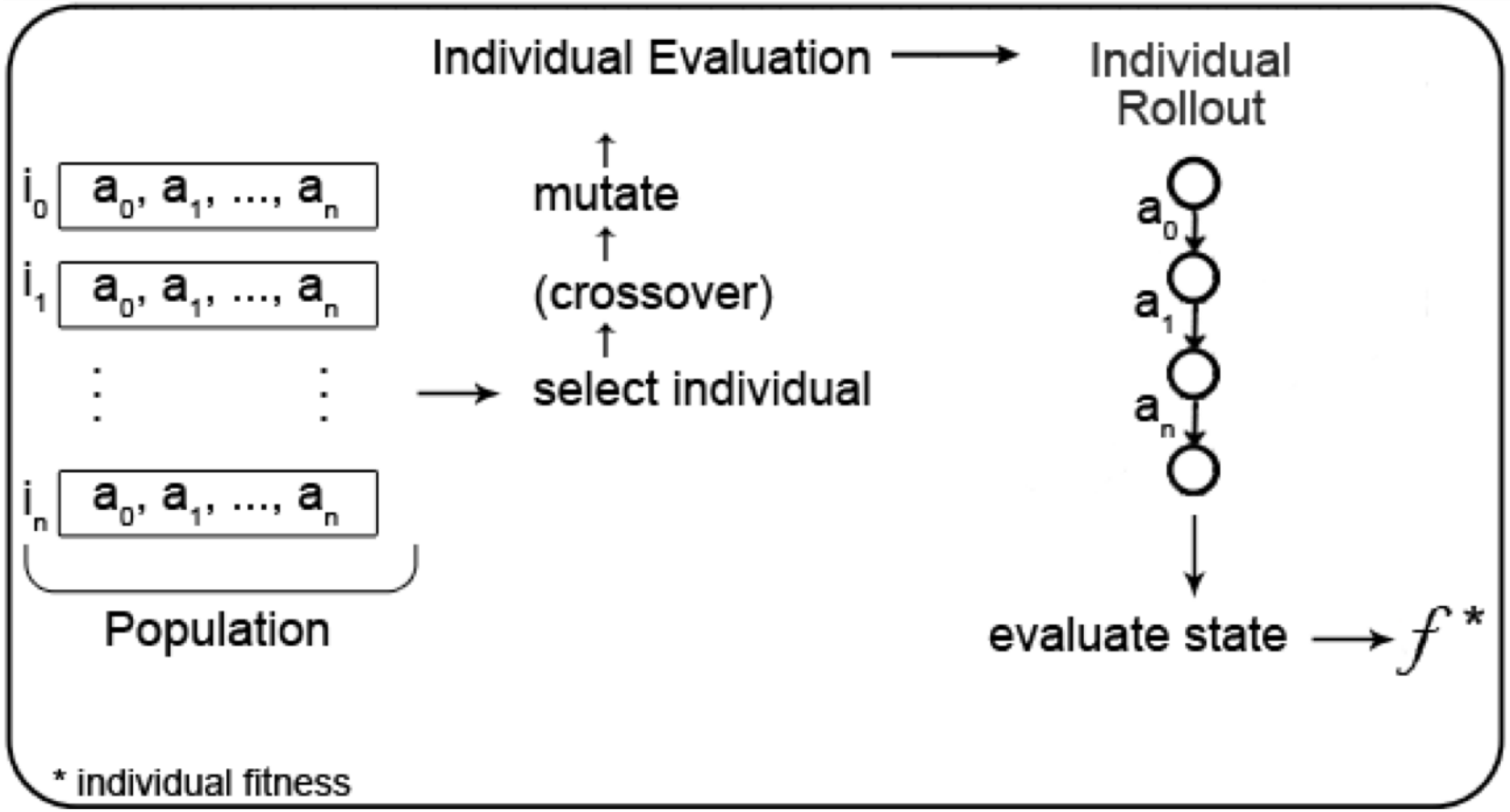
Evolutionary Computation for Decision Making

Rolling Horizon Evolutionary Algorithms (RHEA) is a family of evolutionary computation methods for decision making in real-time. Like MCTS, it is **anytime**: it can be stopped after several iterations (again, the more, the better) and it returns a sensible action for the game.

RHEA is based on any Evolutionary Algorithm, repeated at every game tick:

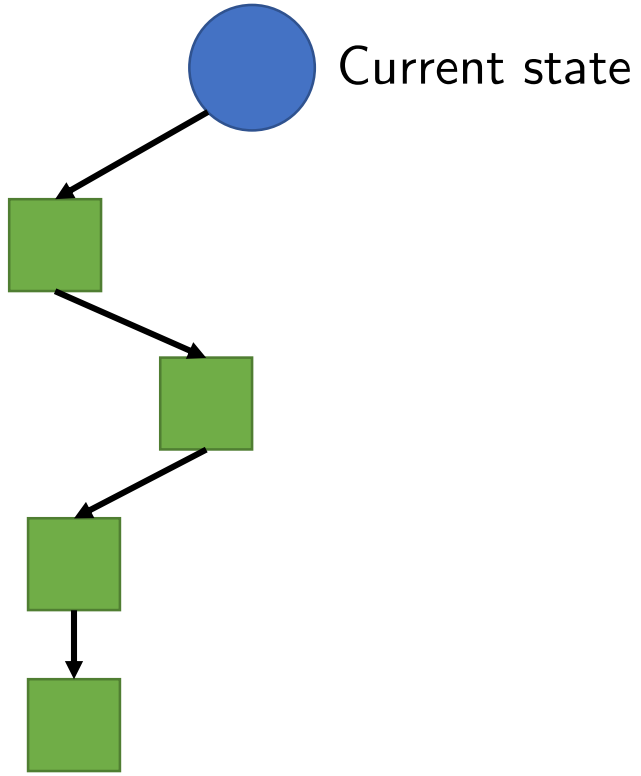
- **Each individual**:
  - Represents an **action plan** (sequence of actions).
  - Is evaluated by executing the plan using the **Forward Model**, starting from the current state.
  - **Fitness**: value returned by a heuristic function applied to the state reached after executing the action plan.
- **Plays** the first action from the best individual found at the end of the evolutionary process.

RHEA


Individual Evaluation
Individual Rollout
mutate
(crossover)
select individual
i_0 $a_0, a_1, ..., a_n$
i_1 $a_0, a_1, ..., a_n$
i_n $a_0, a_1, ..., a_n$
Population
a_0
a_1
a_n
evaluate state
f^*
* individual fitness

RHEA – Step by Step

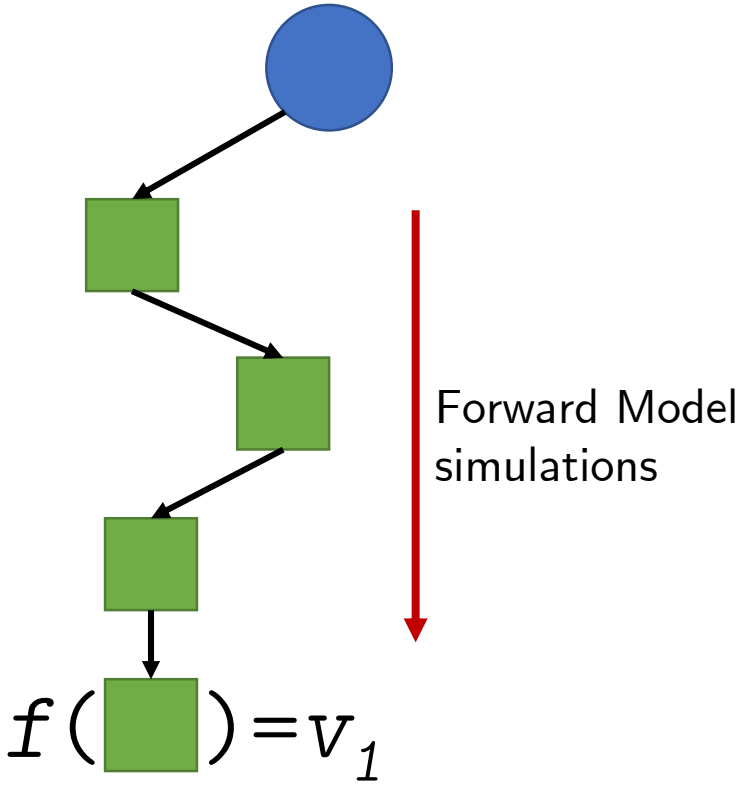
We can visualize RHEA on a game tree as well. If we use a Stochastic Hill Climber as the EA, we'd start with a random full plan of length L ($L = 4$ in this example) ...



This is the equivalent of genotype $LRLD$ in a game with three actions, left (L), right (R) and down (D).

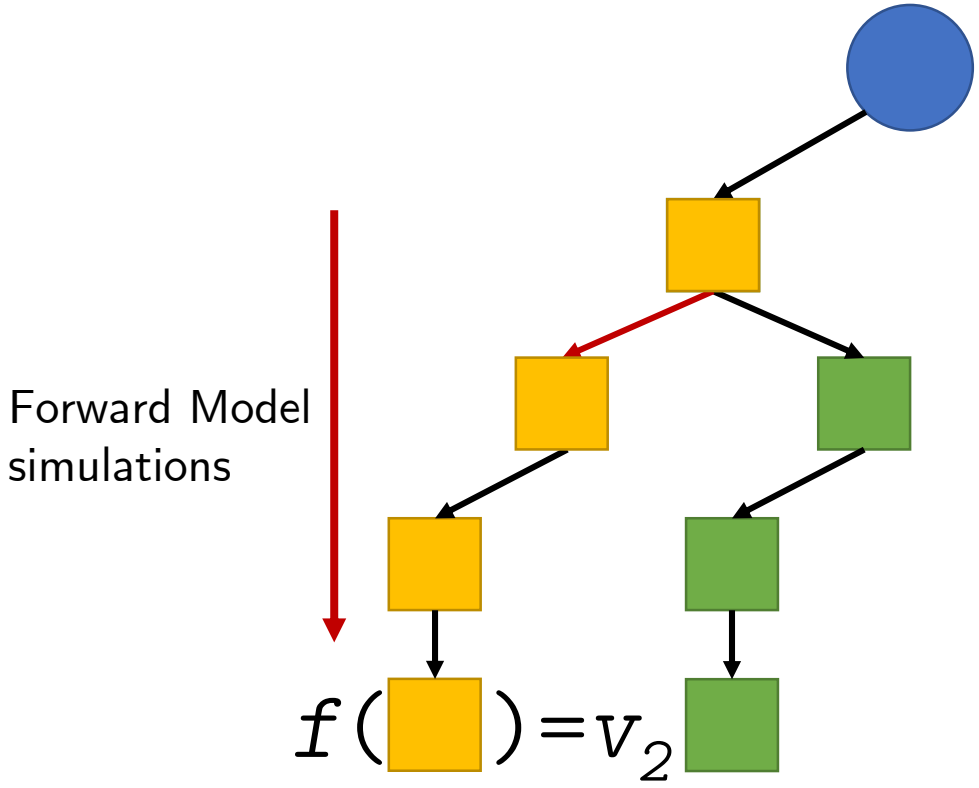
RHEA – Step by Step

We evaluate this plan by simulating with the Forward Model from the current state until the end, and apply a heuristic function on the last state reached. This is the fitness of the first individual, v_1.



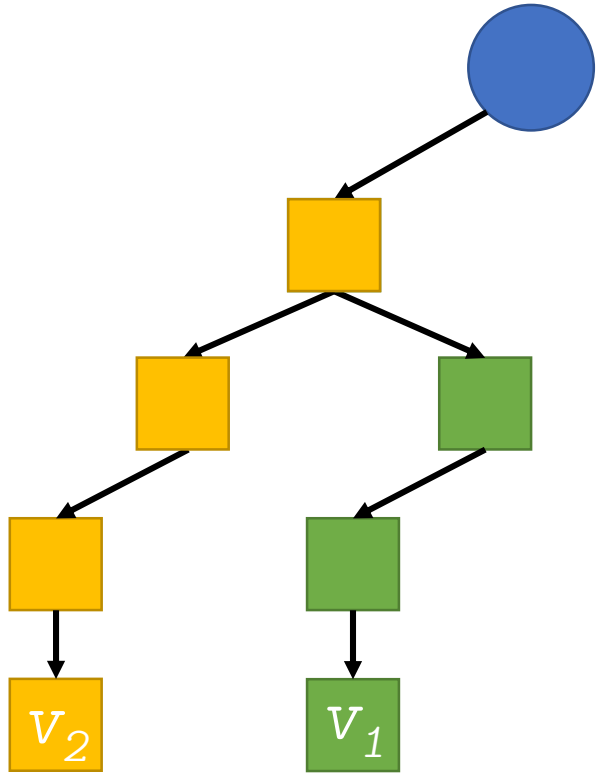
RHEA – Step by Step

Next, we mutate the individual by changing one of the genes (arrows). Let's say the new individual was mutated to ***LLLD***. We evaluate this new individual as well, to obtain v_2.



RHEA – Step by Step

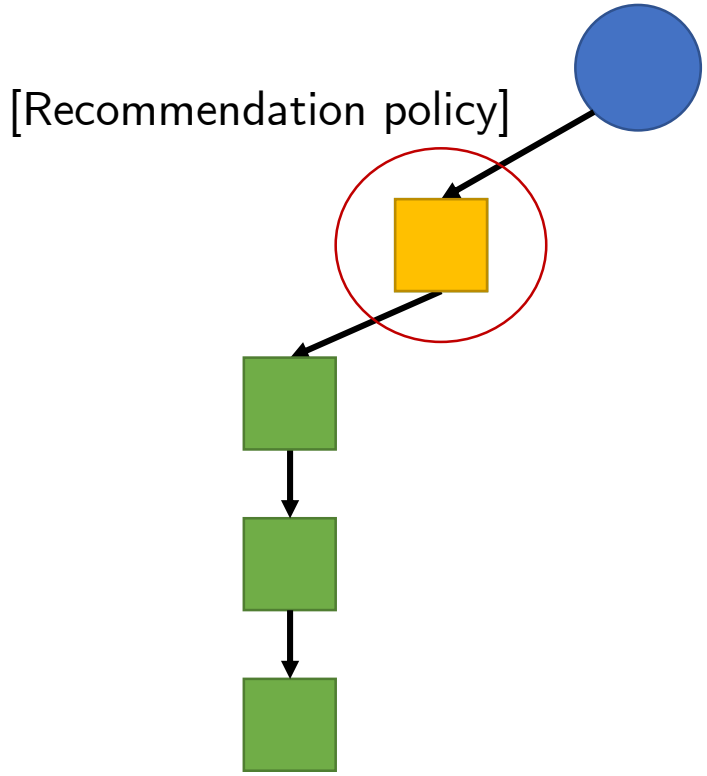
Comparing v_1 and v_2, we choose the best value and discard the individual with the lower fitness. The process repeats until a good enough solution is found, or the budget was reached.



$$v_1 < v_2$$

RHEA – Step by Step

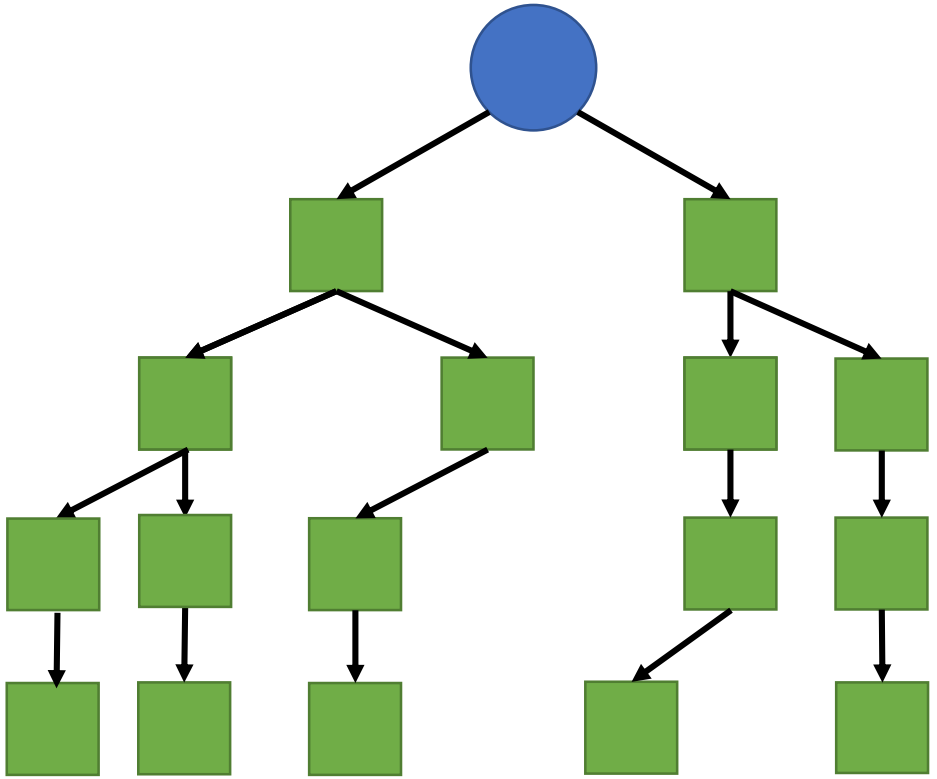
At the end of the evolutionary process, we return the first action of the best (or only) individual in the population.

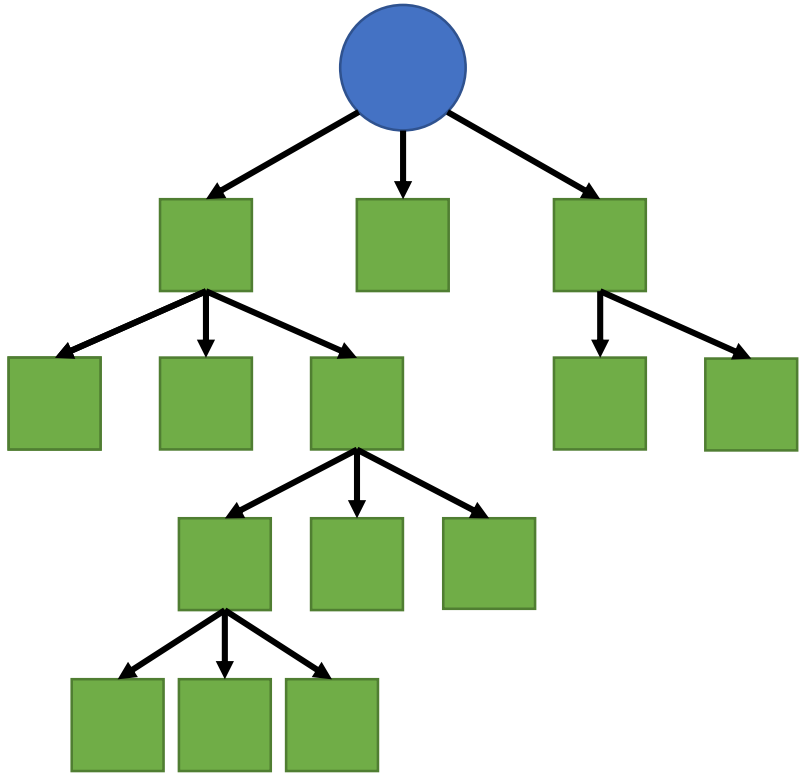


Q? What other recommendation policy could we use here?

RHEA– Step by Step

We can also put together the state space explored by RHEA during its evolution process. Compared to MCTS (on the right), the tree is on average deeper, but more sparse.





Q? Why does RHEA explore the space differently?

Outline

- ✓Game Trees and Uninformed Search
- ✓Best-First Search (A*)
- ✓Monte Carlo Tree Search
- ✓Evolutionary Computation and RHEA

Further Reading

Monte Carlo Tree Search:

- Pepels, Tom, Mark HM Winands, and Marc Lanctot. **Real-time Monte Carlo Tree Search in Ms PacMan.** IEEE Transactions on Computational Intelligence and AI in games 6.3 (2014): 245-257.
- https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6731713

Rolling Horizon Evolution

- Perez, Diego, et al. **Rolling Horizon Evolution versus Tree Search for Navigation in Single-player Real-time Games.** Proceedings of the 15th annual conference on Genetic and evolutionary computation. ACM, 2013.
- http://www.diego-perez.net/papers/GECCO_RollingHorizonEvolution.pdf

Acknowledgements

Part of the materials of this lecture are based on:

- The Game AI Book, by Georgios N. Yannakakis and Julian Togelius (2017)
  http://gameaibook.org

- Other cited references.