Modern Game AI Algorithms
Monte Carlo Tree Search

Mike Preuss | LIACS

Universiteit Leiden
The Netherlands

liacs Leiden Institute of Advanced Computer Science

roadmap

lecture plan (Thursdays 14:15), https://smart.newrow.com/#/room/rzm-514

Feb	18	introduction	Apr	8	procedural content generation II
	25	learning and optimization		15	player experience modeling
Mar	4	procedural content generation I	Apr	22	Georgios: spatial data analysis
	11	Monte Carlo tree search		29	believable behavior
	18	experimentation / ANN / GAN	May	6	team AI and mass behavior
	25	Vanessa: 1 year game industry…		13	realtime strategy AI
Apr	1	learning from pixels	May	20	free project presentations 1

labs plan (Thursdays 16:15)

Feb	18	no lab	Apr	8	free projects (assignment 3)
	25	single assignment: map generation		15	free projects (assignment 3)
Mar	4	single assignment (week 2)	Apr	22	free projects (assignment 3)
	11	single ass. deadline / start ass 2		29	free projects (assignment 3)
	18	assignment 2	May	6	free projects (assignment 3)
	25	assignment 2		13	deadline free projects
Apr	1	deadline ass. 2 / start free proj.	May	20	free project presentations 2

assessment

what is graded:

- single assignment 25%, smaller group assignment 30%,
  free project assignment 45% (including presentation), no written exam

the single assignment is issued today (shall be on Brightspace already):

- simple programming task requiring a bit creativity, Python

smaller group project (largely March):

- group work, most likely in a competition-based fashion on the GVGAI (new version)
- groups of 2 intended

free group project (free means you choose the topic):

- working in small groups (4-5 people, exceptions possible)
- during the second half of the tutorials, and in your preparation time

topics of today

- game trees, MIN-MAX and alpha-beta
- Monte Carlo Tree Search
- using it for „real“ problems (example)
- GGP and GVGAI
- Rolling Horizon EA

picture from janeb13 on Pixabay

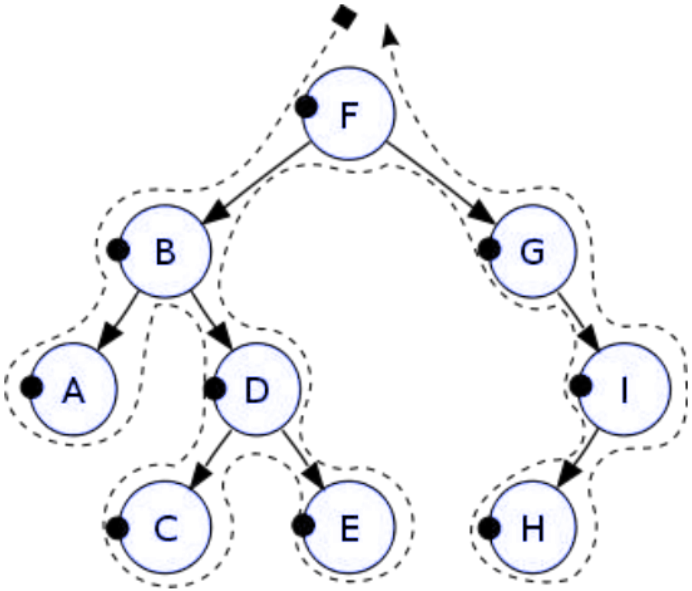
MCTS in Total War: Rome II (Creative Assembly)

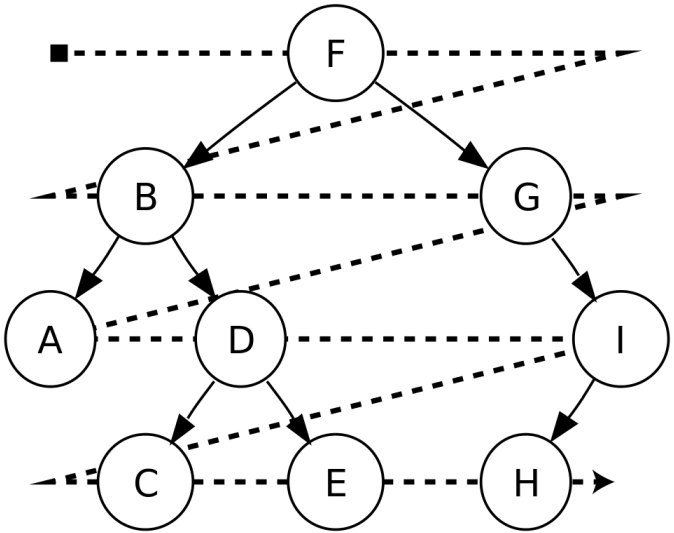
- since the ROME II edition, MCTS is used for planning
- what are the advantages? randomness (surprise), avoid mistakes (alternatives tried), anytime algorithm
- what is it used for? resource allocation to tasks selection of actions (according to resource availability)
- domain knowledge leads to aggressive pruning and biased search (balancing/design)
- industry reluctant to take over, but many will follow

see Tommy Thompson's report:
https://www.gamasutra.com/blogs/TommyThompson/20180212/314399/Revolutionary_Warfare__The_AI_of_Total_War_Part_3.php

(game) tree search

- any method that traverses (examines or updates), expands (creates) and approximates a (game) tree
- result of search: a strategy (optimal hopefully) that wins the game
- it requires end state outcomes and/or utilities/heuristics
- best-first, A*, Dijkstra, breadth-first are examples…





defeating Kasparov

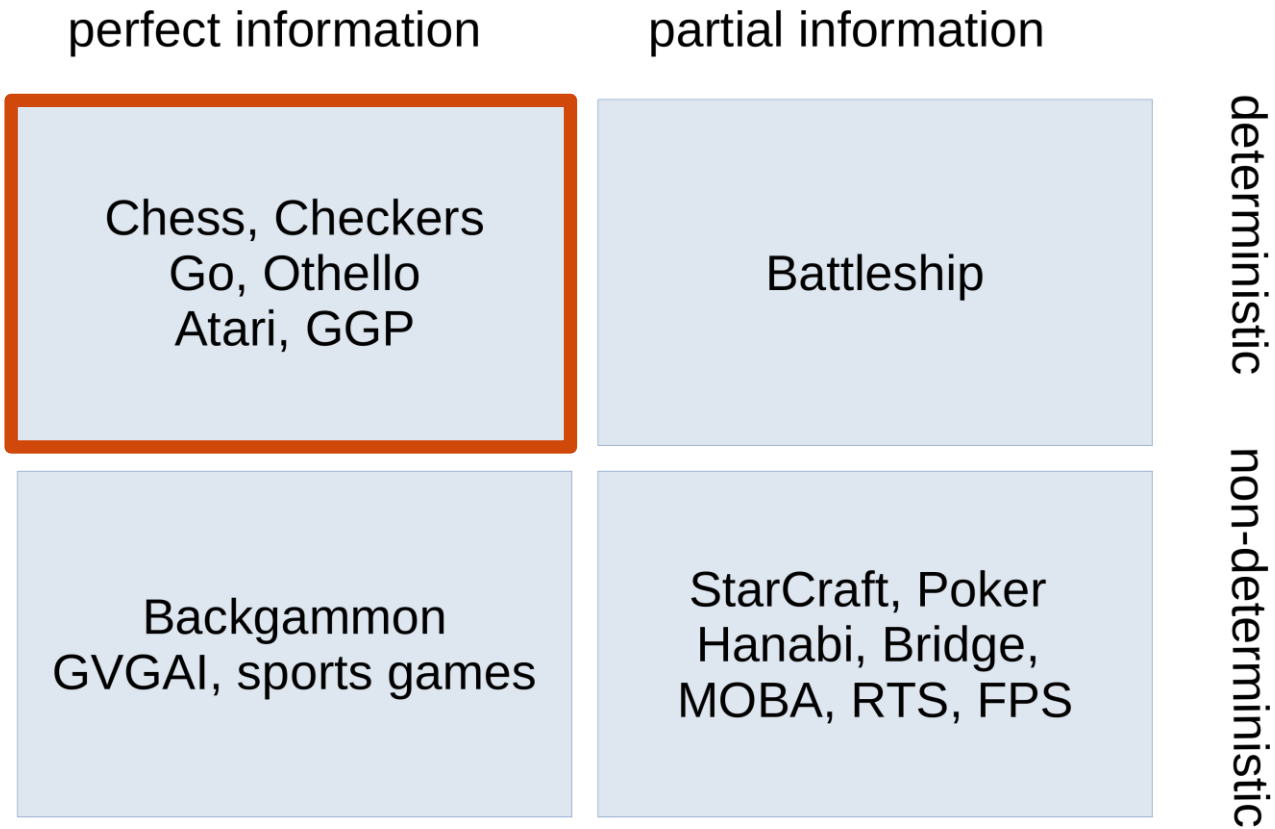
- the Chess world champion was defeated by a min-max tree search algorithm with alpha-beta pruning in 1997
- seemingly partly due to a bug :-)
- can't we just transfer this to other types of games? not so easy
- Kasparov has interesting position toward AI nowadays:

https://www.wired.com/story/defeated-chess-champ-garry-kasparov-made-peace-ai/

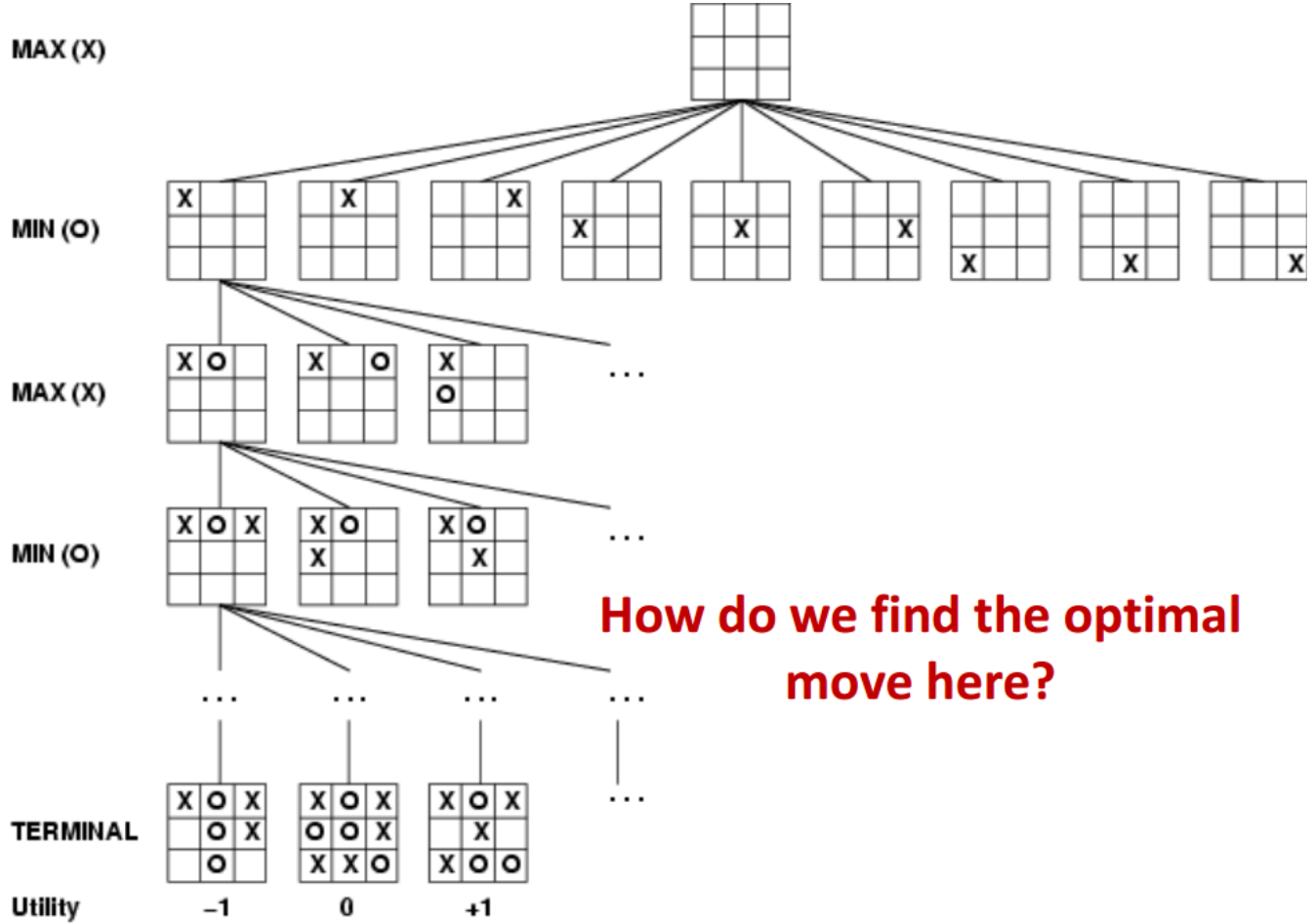
"Technology is the main reason why so many of us are still alive to complain about technology."

Garry Kasparov

what we can solve with min-max tree search



tic-tac-toe: two-player, deterministic, turn-based



MIN and MAX are the players

MAX

- wants to **maximize** the result of the utility function
- winning strategy if, on MIN's turn, a win is obtainable for MAX for all moves that MIN can make

MIN

- wants to **minimize** the result of the evaluation function
- winning strategy if, on MAX's turn, a win is obtainable for MIN for all moves that MAX can make

utility functions

a utility function:

- estimates how good the current board configuration is for a player
- utility(state)

examples:

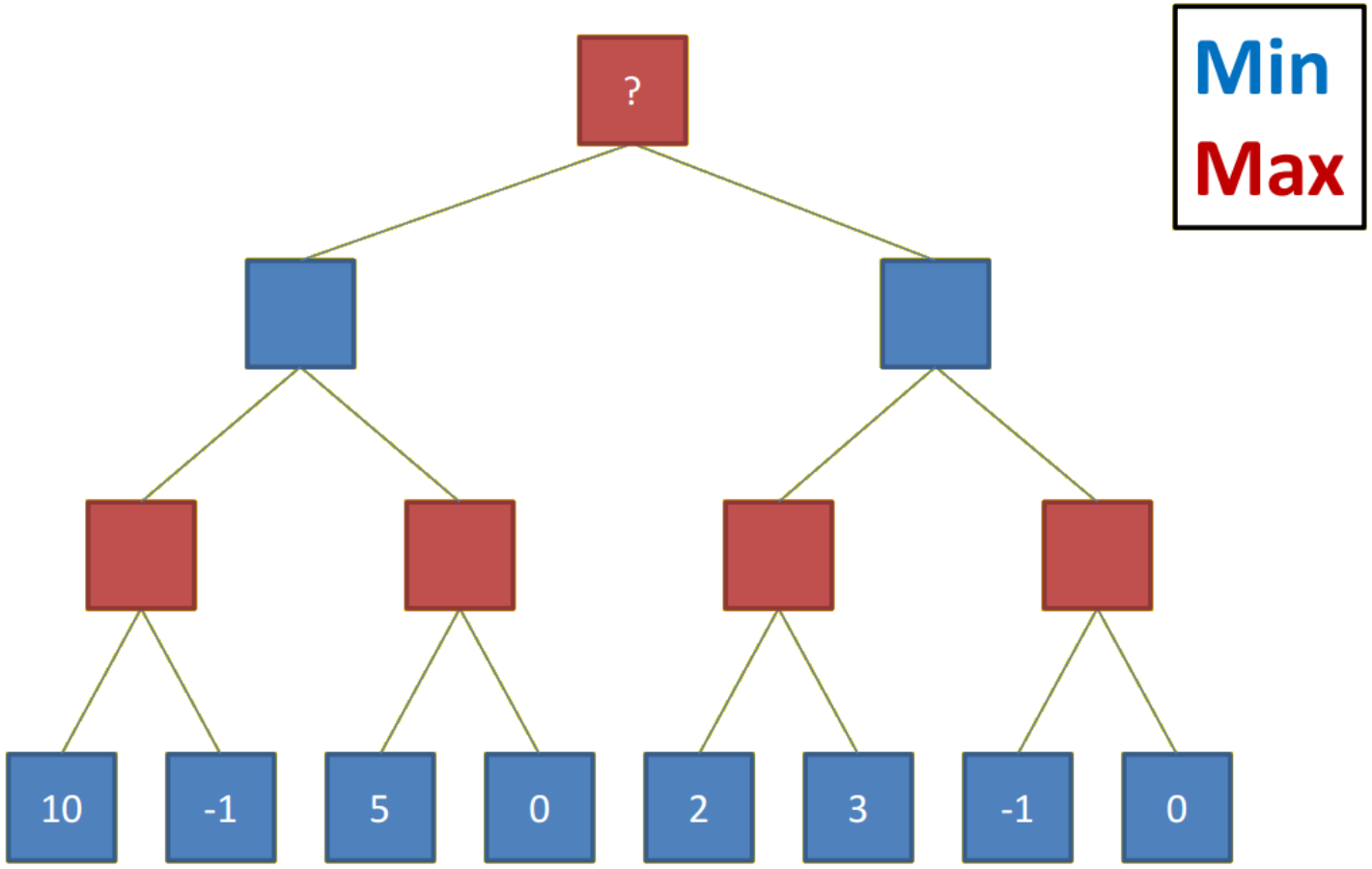
- typical values from -infinity (loss) to +infinity (win) or
  -1 (MIN wins), 0 (draw) , +1 (MAX wins)
- Othello: number of white pieces - number of black pieces
- Chess: value of all white pieces - value of all black pieces
- weighted sum of factors (e.g. Chess)
  utility(S) = $w_1 f_1(S) + w_2 f_2(S) + \ldots + w_n f_n(S)$
  – $f_1(S)$ = (number of white queens) – (number of black queens), $w_1 = 9$
  – $f_2(S)$ = (number of white rooks) – (number of black rooks), $w_2 = 5$
  – …
- if the utility is X for a player, it is -X for the opponent
  – “zero-sum game”

MIN-MAX tree search

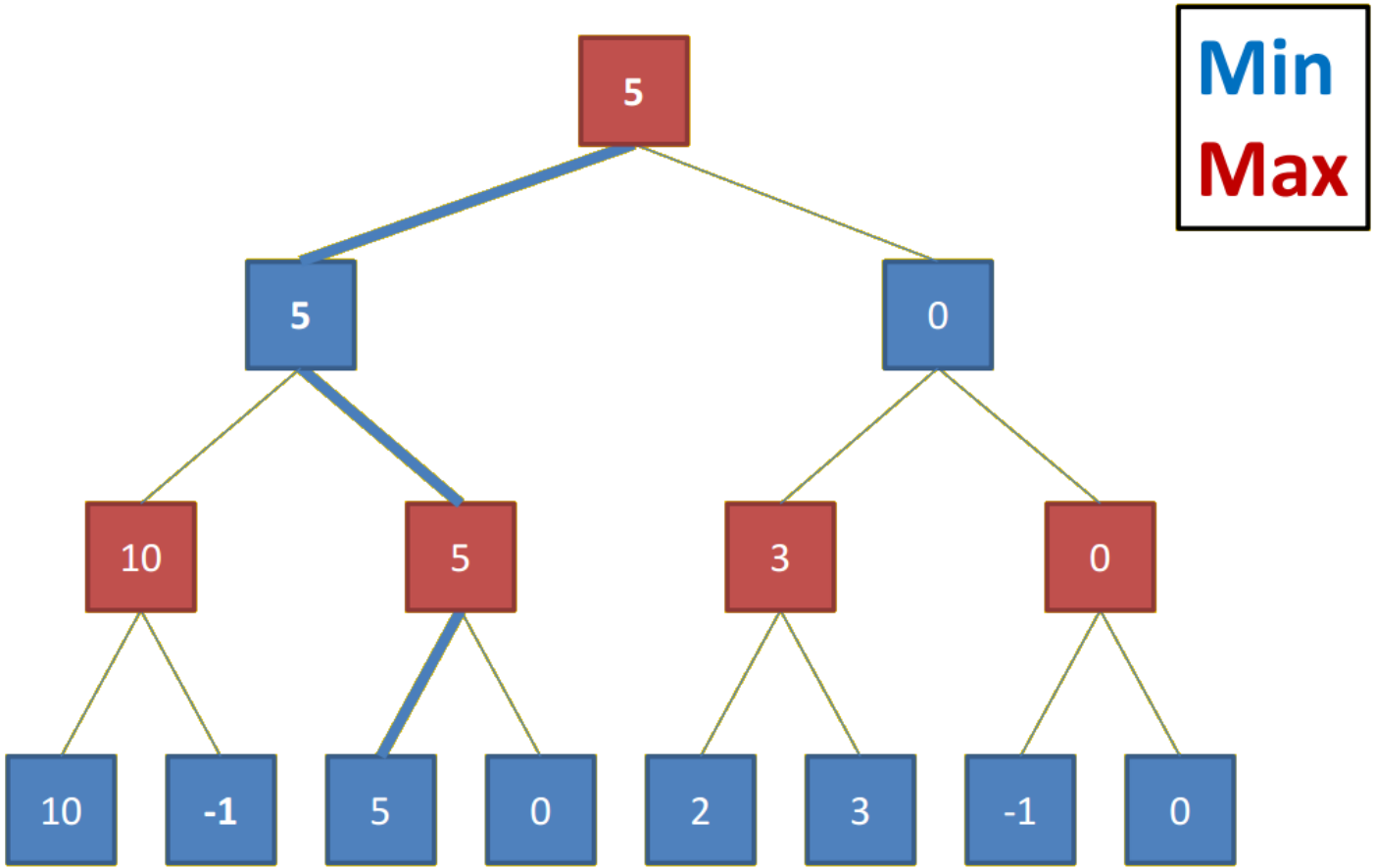
designed to find the **optimal strategy for MAX** and find **best move (policy, strategy)**:

1. generate the whole game tree, down to the leaves
2. apply utility function to each leaf
3. back-up values from leaves through branch nodes:
   – a MAX node computes the MAX of its child values
   – a MIN node computes the MIN of its child values
4. at root: choose the move leading to the child of highest value

MIN-MAX tree search: an example



MIN-MAX tree search: an example



MIN-MAX tree search algorithm

```
function MINIMAX-DECISION(state) returns action
        inputs: state   // current state in game
        return the a in actions(state) maximizing MIN-VALUE(forward(state,a))

function MAX-VALUE(state) returns utility
        if TERMINAL-TEST(state) then return UTILITY(state)
        v = -infinity
        for a, s in SUCCESSORS(state) do v = MAX(v, MIN-VALUE(s))
        return v

function MIN-VALUE(state) returns utility
        if TERMINAL-TEST(state) then return UTILITY(state)
        v = infinity
        for a, s in SUCCESSORS(state) do v = MIN(v, MAX-VALUE(s))
        return v
```

game tree sizes

- Tic-Tac-Toe

  9! = 362,880
  exact solution quite reasonable

- Chess

  b ≈ 35 (approximate average branching factor)
  d ≈ 100 (depth of game tree for “typical” game)
  $b^d \approx 35^{100} \approx 10^{154}$ nodes!!
  exact solution infeasible

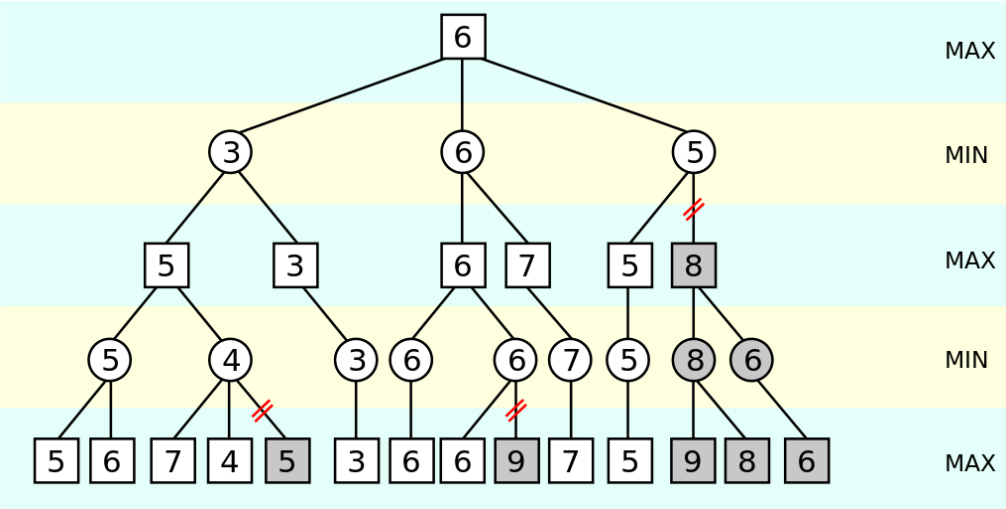
- StarCraft

  conservative estimate of state number from 400 units (2-player)
  on 128x128 map $16384^{400} \approx 10^{1685}$

- it is **usually** impossible to develop the whole search tree
  need of good utility functions and/or pruning techniques

alpha-beta pruning

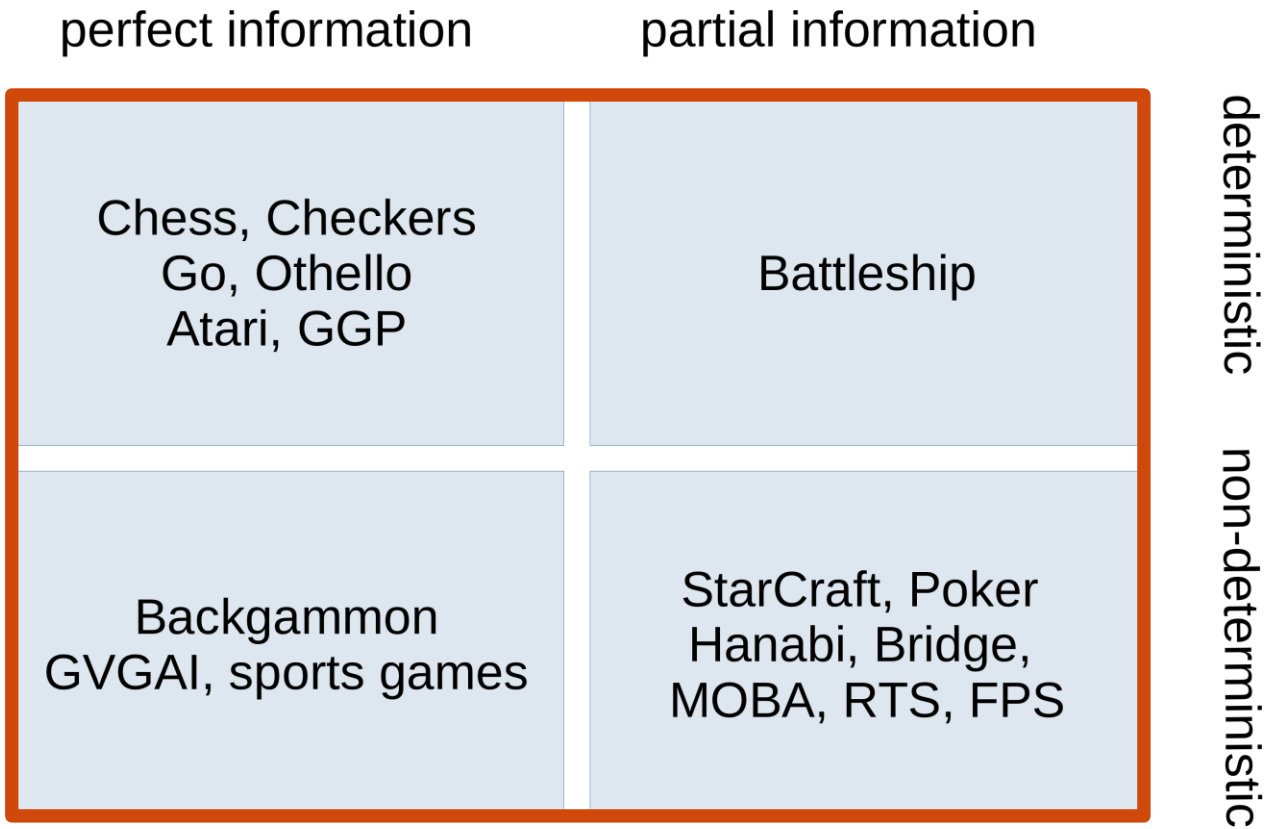
- we can disregard a part of the tree (prune)
- if a move is worse than an already evaluated one, stop evaluating it
  (we save building part of the tree and even computing some leaves)
- alpha = minimum score MAX can get, beta = maximum score MIN can get
- tree evaluated from left, if alpha >= beta we can discard that subtree (will never be played)



(remember we are starting with an empty tree and compute values on demand from left to right)

demo: http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html

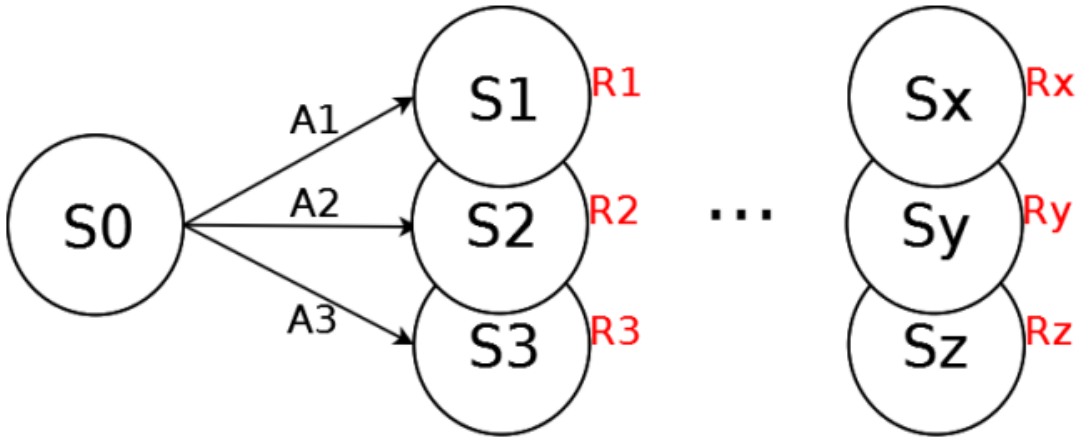
what we can solve with Monte Carlo tree search



states and actions

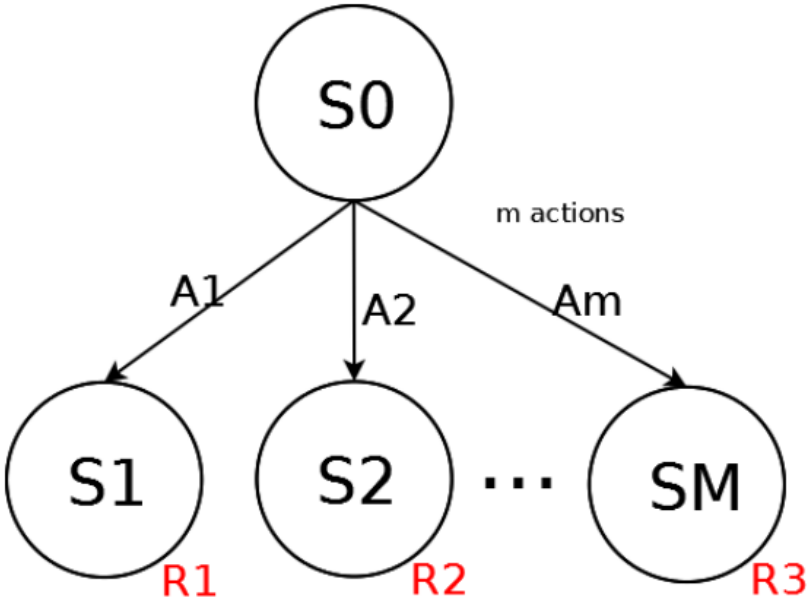
- a Markov decision process (MDP) is a tuple $< S, A, P, R, \gamma >$
  S: finite set of all possible states of the game
  A: finite set of all possible actions in the game
  P: a state transition probability matrix $P^{a}_{ss'} = P[s' | s, a]$
  R: a reward Function, $Rs\ a = E[r|s, a]$
  γ: a discount factor $\gamma \in [0, 1]$

- we need to learn a table
- "finite set" is problematic



one step look ahead

- try all actions, pick the one with the highest reward
- problem: we do not reach a goal state, need heuristics



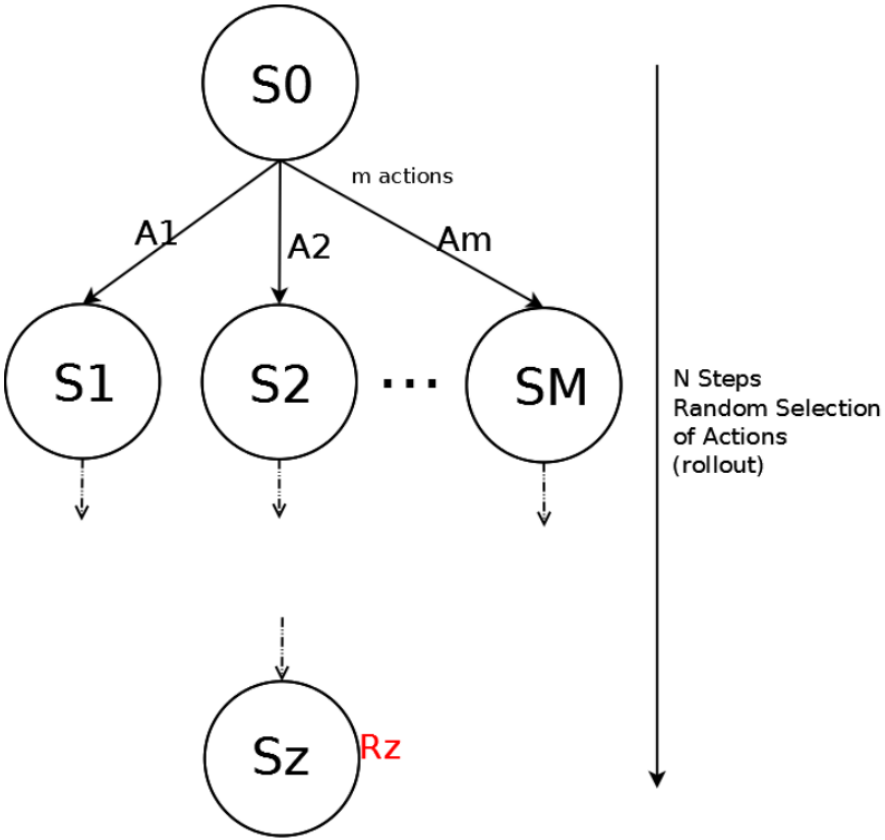
flat Monte Carlo

- iteratively, apply *N* consecutive actions picked randomly
- pick the action that leads to:
  - the highest reward after *N* steps
  - the highest **average** reward after *N* steps, etc.

problems:

- this is similar to randomized brute force
- how large is N? when do we get a non-0 reward?

can we somehow “intelligently” select interesting paths?



multi-armed bandit problem

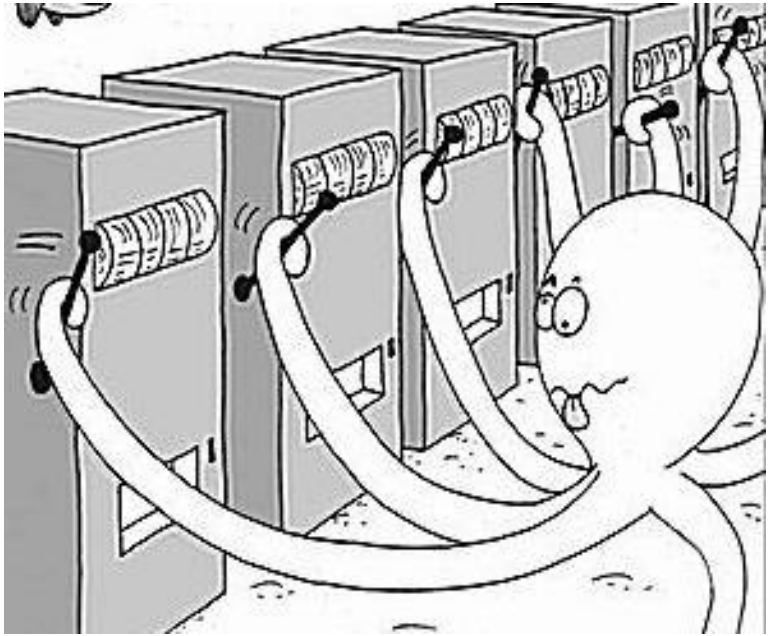
- given that there are several choices, which one do we take?
- this relates to a well-known decision making problem: the Multi-Armed Bandit Problem (MAB)

- at each step pull one arm

- noisy/random reward signal

pick the arm so as to:

- minimize regret (expected loss due to not picking the best arm)
- maximize expected return



Monte Carlo (uniform)

most simple approach: **Monte Carlo**

- share trials uniformly between arms
- don't keep any outcome information

almost as simple: **ε-greedy**

- P(1-ε) – pick the best arm so far
- P(ε) – pick an arm randomly
- exploitation vs. exploration

upper confidence bound (UCB)

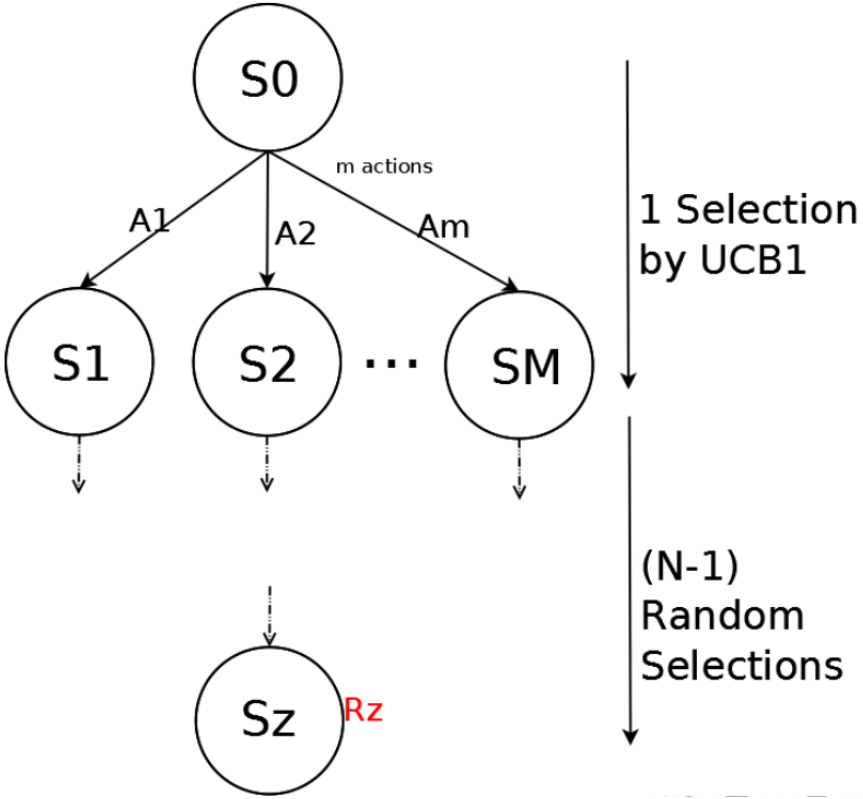
- balance exploitation and exploration
- an example: UCB1

$$a^* = \underset{\alpha \in A(s)}{\operatorname{argmax}} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s))}{N(s,a)}} \right\}$$

- Q(s,a): average of rewards after taking action a from state s
- N(s): times the state s has been visited
- N(s,a): times the action a has been picked from state s
- C: constant that balances exploitation (Q) and exploration terms
  - application dependent
  - typical value for single player games with rewards in [0,1]: $\sqrt{2}$

flat UCB

- use UCB1 to select action from the root node (or current state)
- pick (N – 1) actions at random
- repeat iteratively, return action with (one of):
  - the highest reward after N steps
  - the highest **average** reward after 1 step (Q (s, a))
  - the most visited node after 1 step (highest a for N (s, a)).
  - the highest UCB1 value after 1 step, etc.



UCB1 - discussion

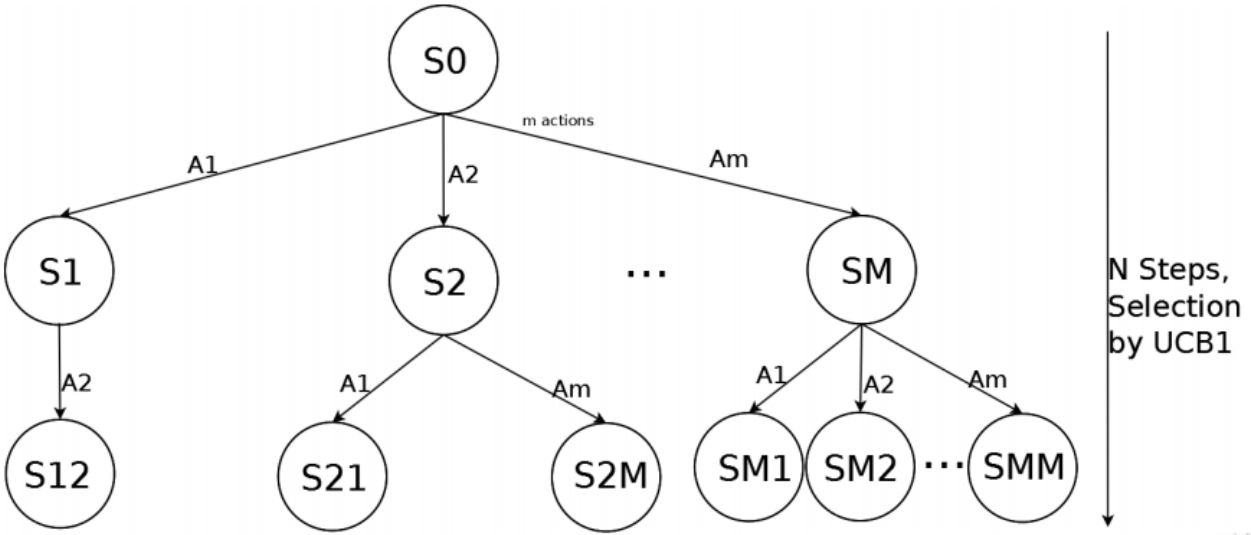
- UCB1 formula minimises regret
  - bound grows logarithmically with the total number of actions taken: **ln(N(s))**
  - bound shrinks with the number of times we have taken this particular action: **N(s,a)**
  - ensures: actions are tried infinitely often but still a good balance of exploration/exploitation

- needs limited information:
  - game rules (states, actions)
  - terminal state evaluation (reward)
- super simple to implement
- surprisingly effective, but…

… doesn't look ahead !!!

building a tree with UCB1

- build a tree: search *N* steps in the future
- the search is **not** exhaustive: tree grows asymmetrically
- repeat iteratively, return action with:
  - the highest reward after *N* steps
  - the highest **average** reward after 1 step ($Q(s, a)$)
  - the most visited node after 1 step (highest a for $N(s, a)$).
  - the highest UCB1 value after 1 step, etc.



Monte Carlo tree search (MCTS)

depends on two concepts:

1. true value of action can be approximated via random **simulations**
2. the values may be used efficiently to adjust the policy towards a **best-first** strategy (brute force is too expensive)

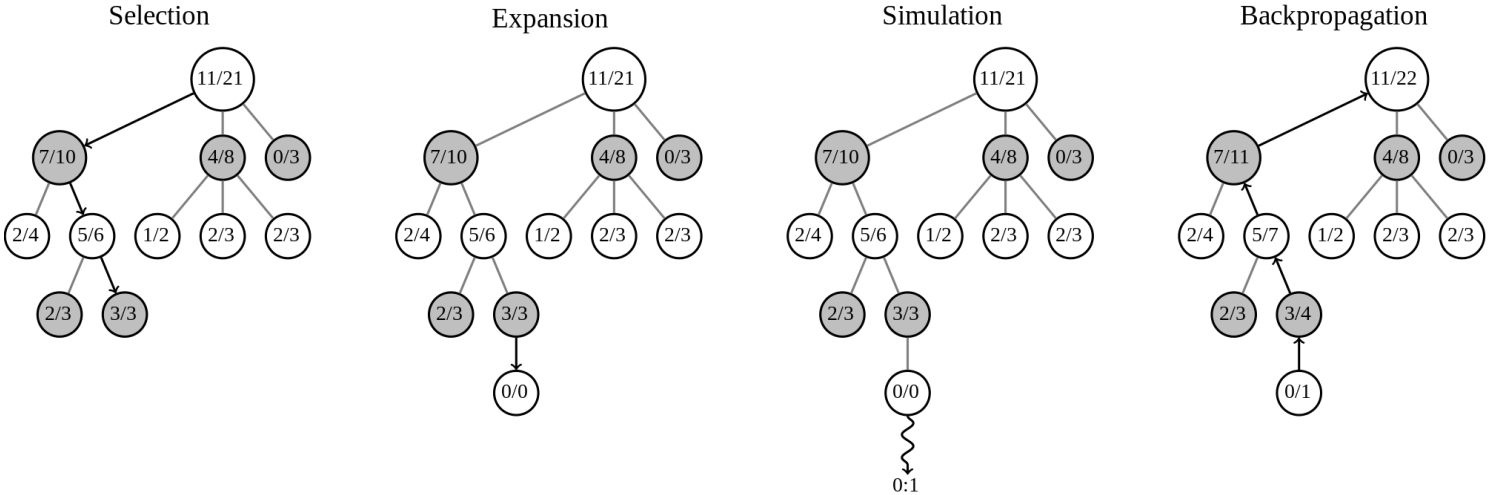
advantages:

- anytime algorithm – stop whenever you like
- needs only game rules:
  - moves
  - terminal state evaluation (win, loss, draw, score)
- no need for a heuristic function, …
  - but can be enhanced with domain knowledge in various ways
  - key advantage over MIN MAX

MCTS – the big picture

- selection: select promising node within the tree (by means of UCB1)
- expansion: add new leaf
- simulation: play out the game until we reach a terminal state (and obtain a reward)
- backpropagation: inform the “higher” nodes about move potential

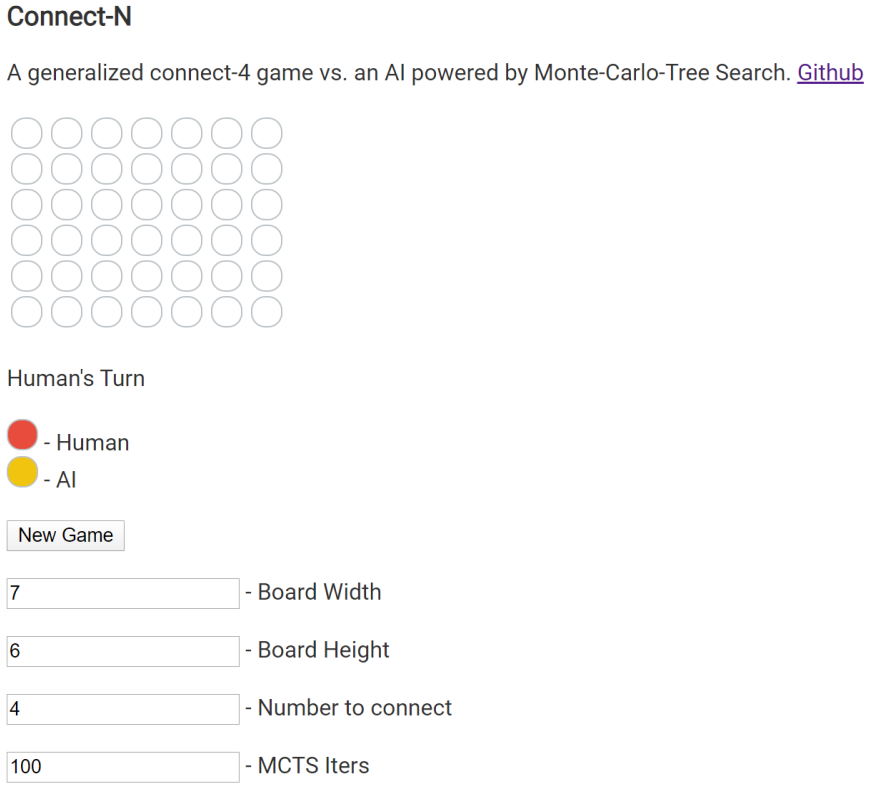
- selection is also called “tree policy”, simulation the “default policy”
- for each node, we store: incoming action a, associated state s, total simulation reward Q, visit count N



MCTS plays "connect-4" (against you)

- in connect-4 you can only "throw in" a coin in one of the columns and it falls down
- the aim is to get 4 coins in a row (any form, diagonal included)
- start to play against 100 MCTS iters (last parameter)
- if you win, proceed with 1000, then with 10000
- do you realize a difference in AI playing strength?
- btw, program does not always recognize the human winning (you have to)

https://sethpipho.github.io/monte-carlo-tree-search-js/demo/connect-4/



MCTS main algorithm

- BestChild simply picks best child node of root according to some criteria: e.g. best mean value
- in our pseudo-code BestChild is called from TreePolicy and from MctsSearch, but different versions can be used
  - e.g., final selection can be the max value child or the most frequently visited one

function MCTSSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

tree policy

- note that node selected for expansion does not need to be a leaf of the tree
  - the *nonterminal* test refers to the game state

```
function TreePolicy(v)
    while v is nonterminal do
        if v not fully expanded then
            return Expand(v)
        else
            v ← BestChild(v, Cp)
    return v
```

expand

- note: need to choose an appropriate data structure for the children (array, ArrayList, Map)

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

best child (UCT)

- this is the standard UCB equation (we rewrite the UCB1 C slightly here)
  - used in the tree
- higher values of *c* lead to more exploration
- other terms can be added, and usually are

$$\textbf{function } \textsc{BestChild}(v, c)$$

$$\textbf{return } \underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

default policy

- each time a new node is added to the tree, the default policy randomly rolls out from the current state until a terminal state of the game is reached
- the standard is to do this uniformly randomly
  - but better performance may be obtained by biasing with knowledge

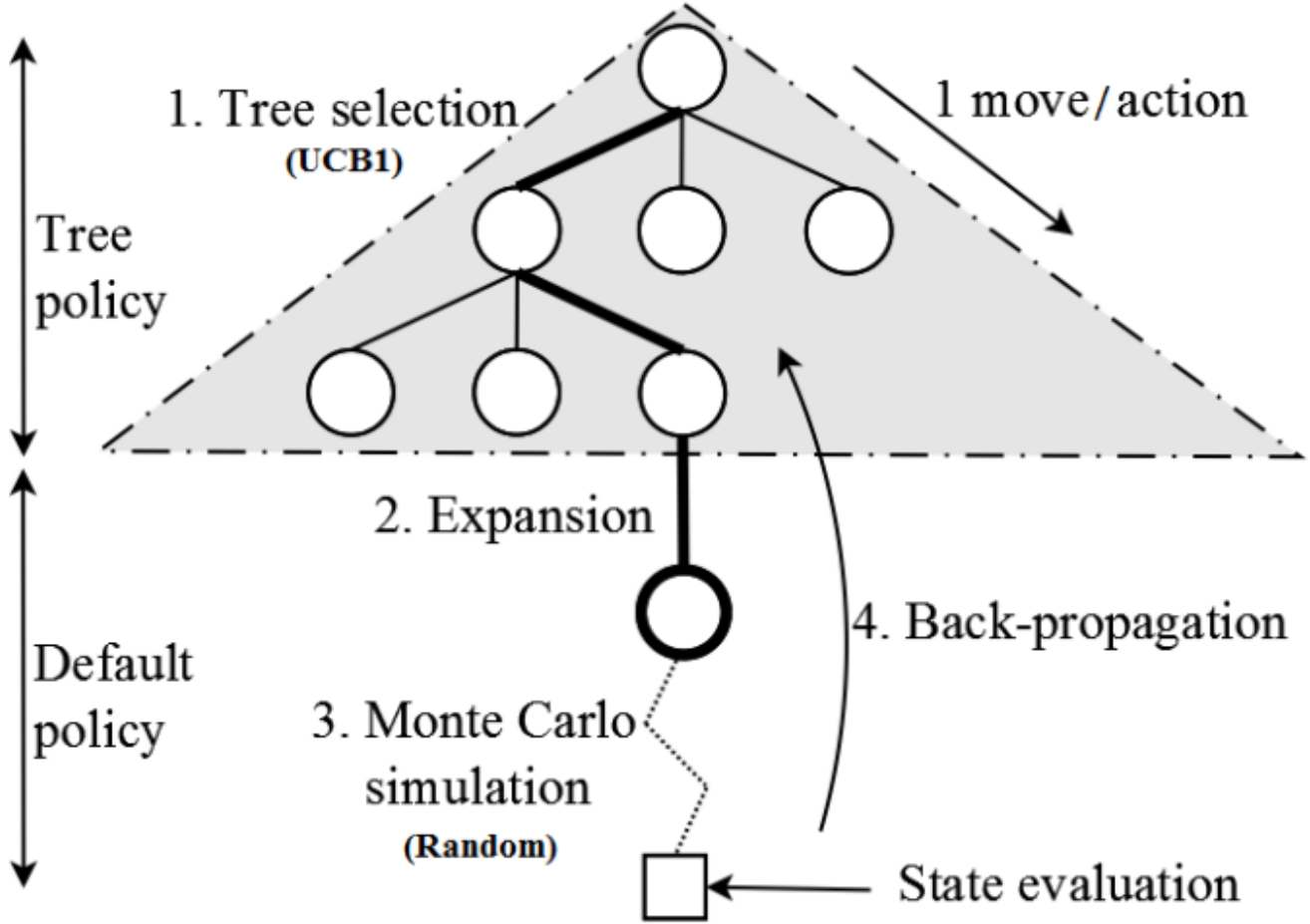
function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

backup

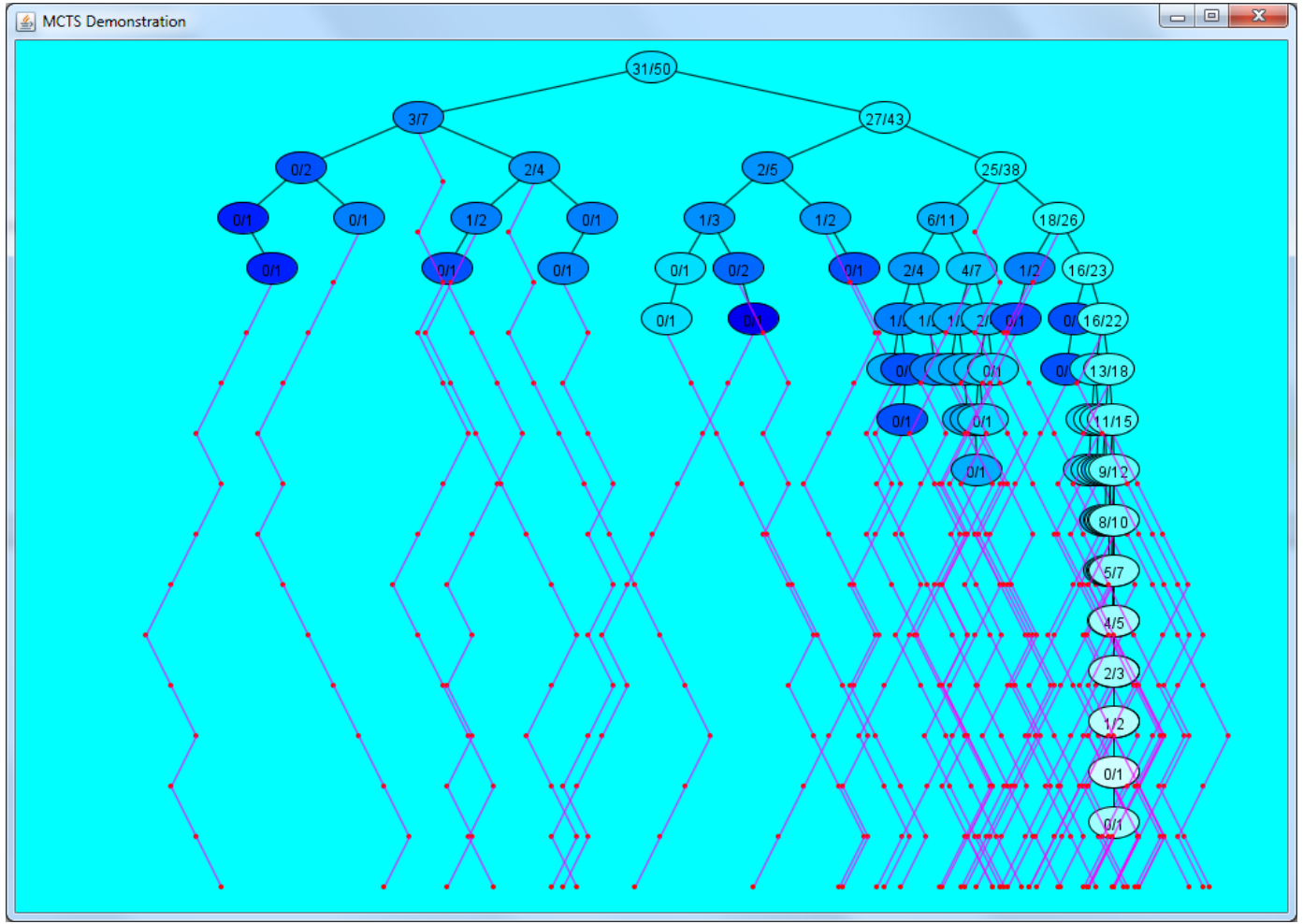
- note that v is the new node added to the tree by the tree policy
- back up the values from the added node up the tree to the root

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

MCTS: the even bigger picture

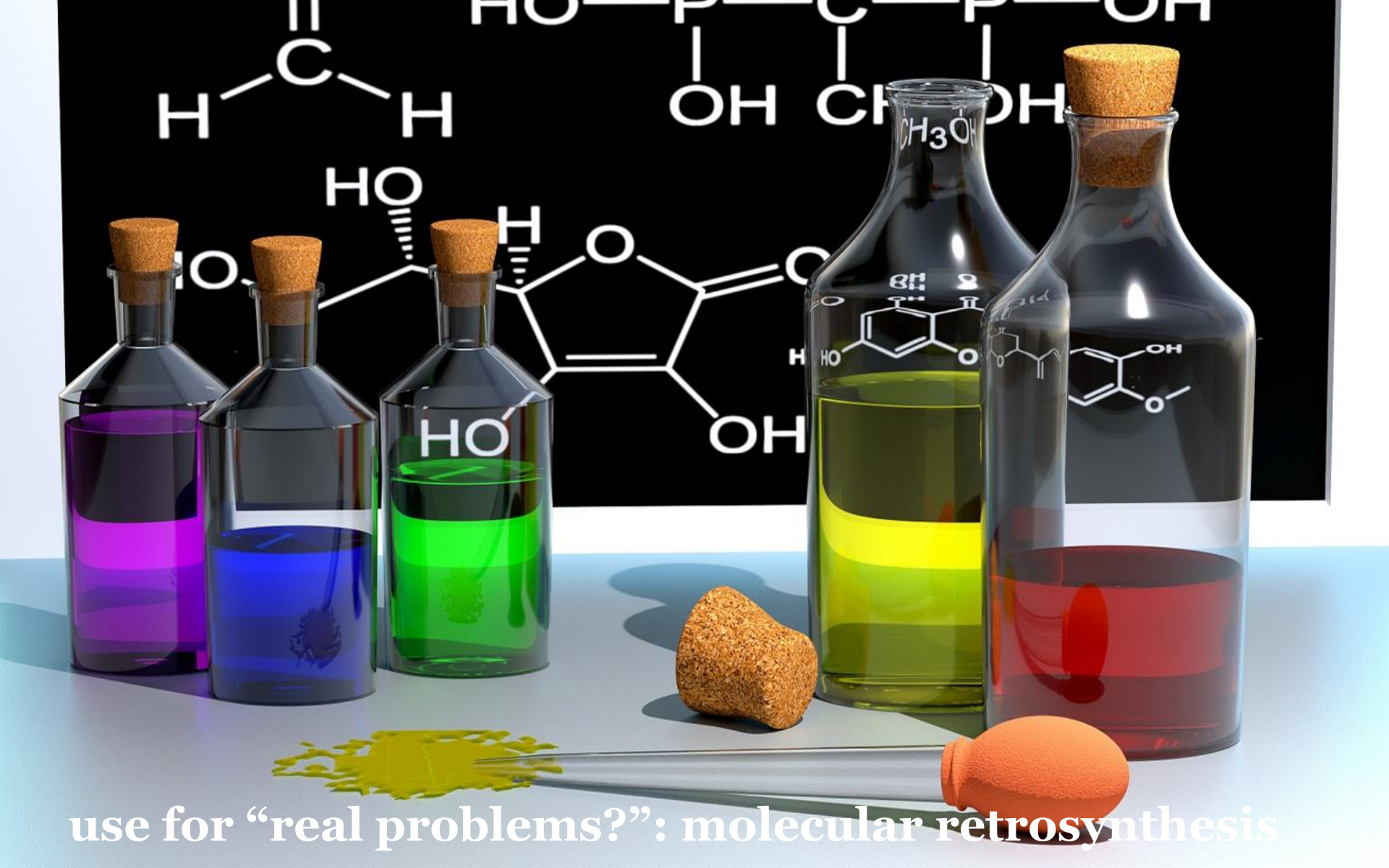


MCTS builds asymmetric trees



MCTS goes real-time

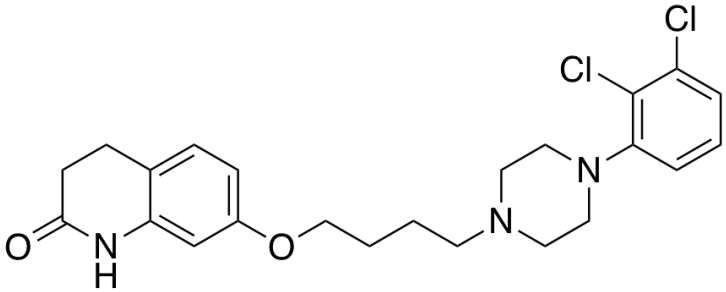
- limited roll-out budget
  - heuristic knowledge becomes important
  - we will only see a small part of the tree
- action space is fine-grained
  - take *macro actions* otherwise planning will be very short-term
  - currently unclear how to "generate" macro actions
- maybe no terminal node in sight
  - use a heuristic
  - tune simulation depth
- or use "heavy" playouts (use heuristic in playouts)?

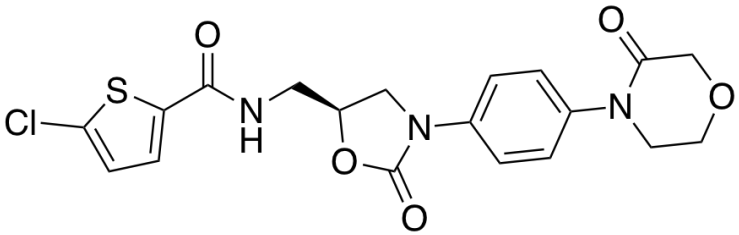


use for “real problems?”: molecular retrosynthesis

chemical synthesis

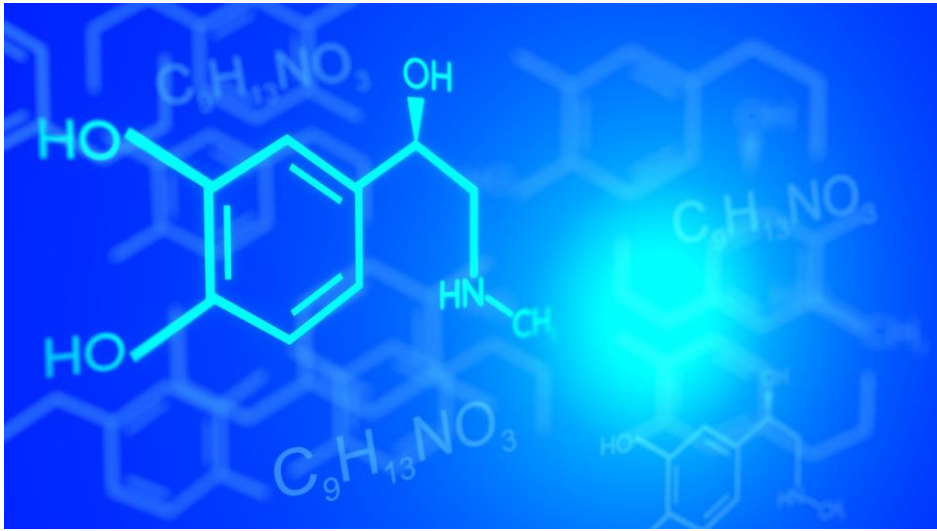
- many important drugs are small organic molecules
- synthesis is a significant bottleneck
- 60 years history in computer aided synthesis planning, not much success





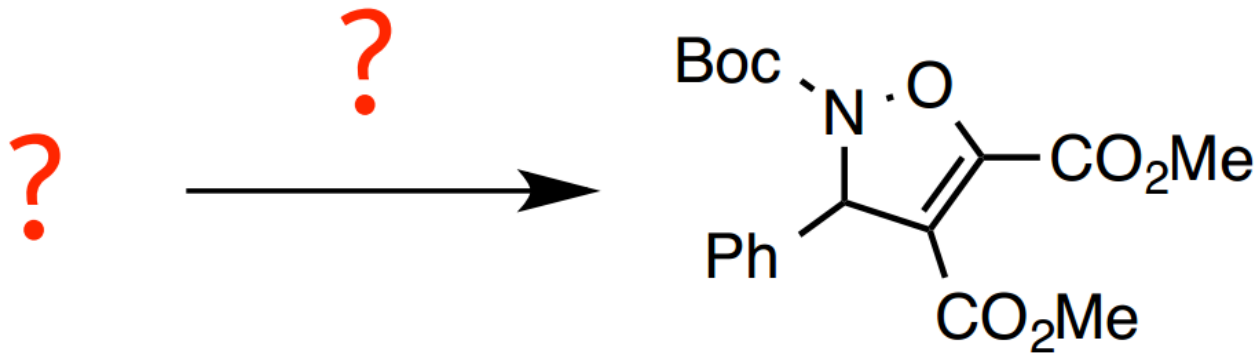
retrosynthesis is a key problem

- much of Chemistry is “theoretical” (in a computer) now
- there are tools for predicting properties of unknown substances
- but being able to model a substance does not mean that we know how actually make it
- retrosynthesis can find such ways
- this is a multi-modal problem
  we want several different solutions
- this is a multi-objective problem,
  there are multiple objectives
  - cost
  - yield
  - conflicting required properties
    (bind to x but not to y)



can we frame this in the MCTS/DNN context?

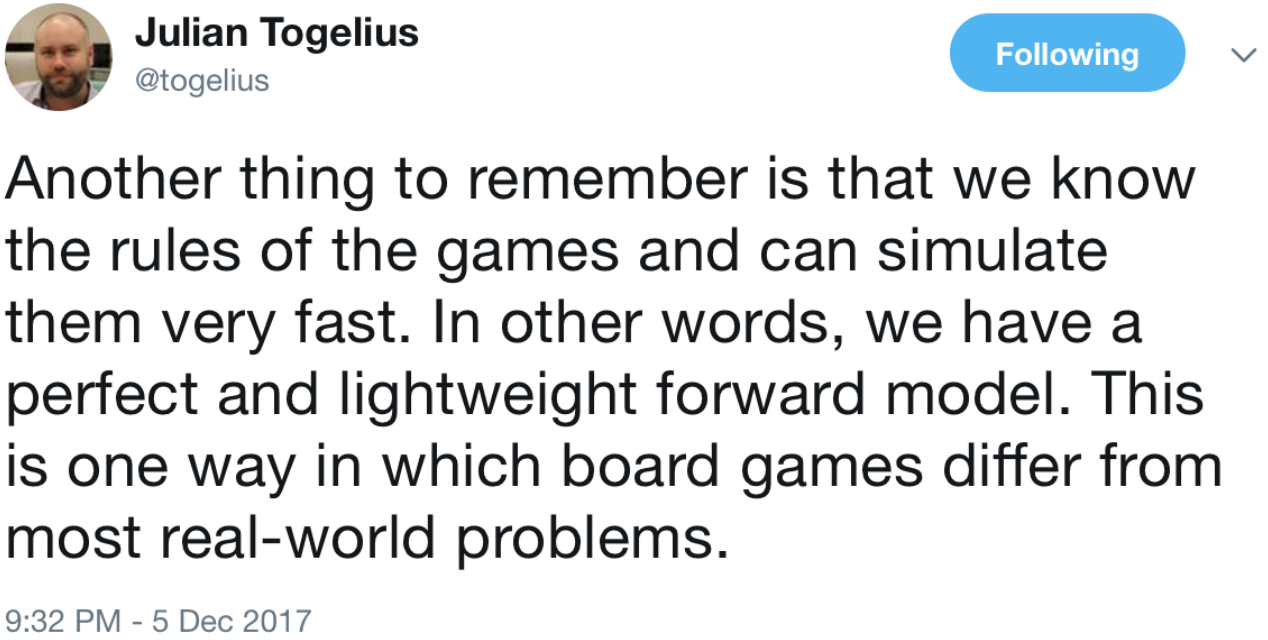
- the problem is not to take the right move
- but to take the right decisions for synthesizing a molecule
- both share a lot:
  - high branching factor
  - sequence length not exactly known
  - full tree cannot be enumerated
  - no strong heuristics



solving different problems

	Chess	Go	Synthesis
System	DeepBlue	AlphaGo	?
Scoring	Heuristics	MCTS+DNN, RL	?
Rules	Trivial & Fixed	Trivial & Fixed	Complex, unknown, expanding
Branching Factor	very small	small	large
Experiments	cheap & fast	cheap & fast	takes days (lab/simulation)

games vs the real world

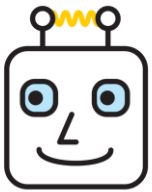


similar but different

AlphaGo (Zero):
take perfect world simulator (forward model), then RL/planning with lots of compute

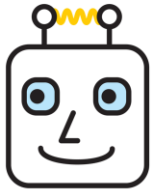
This work:
Learn generalizing(!) world simulator (forward model), then RL/planning

RL



Use agent in real world

Use agent in real world



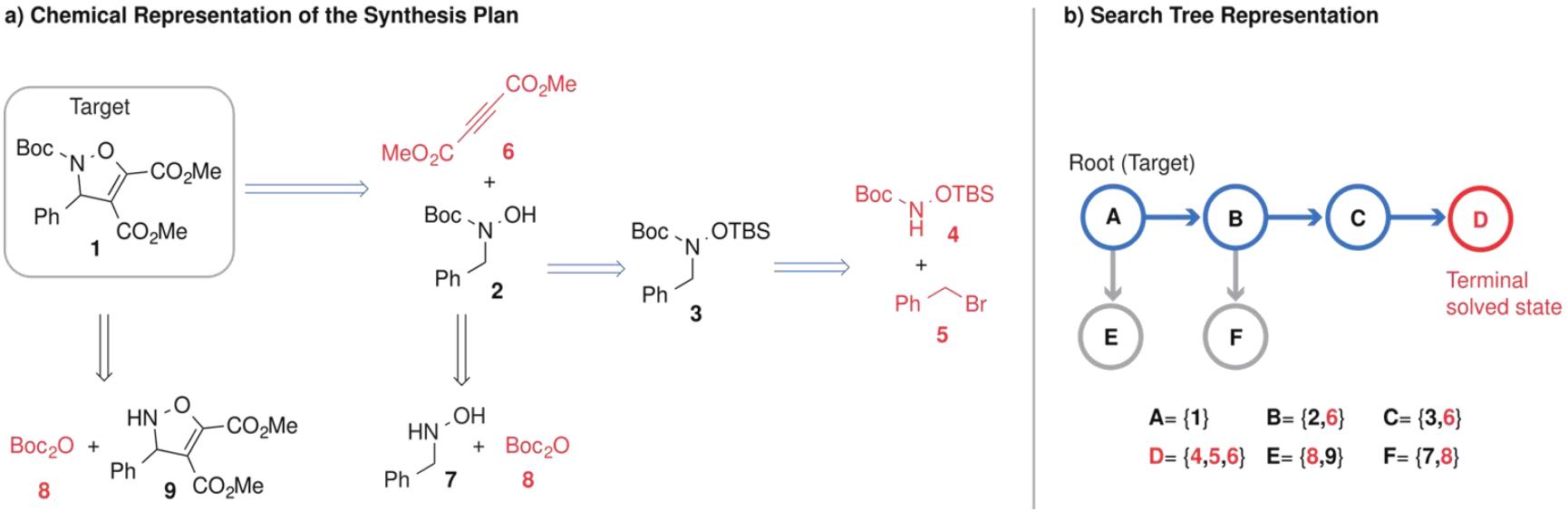
RL

Learn from past experiments

what problems need to be solved?

1. we need a way to encode the available molecules
2. we need a way to encode reaction rules
3. we need to **obtain** existing reaction rules
4. we need to **find out** which reaction rules are most suitable for a specific molecule (part)

finally, it should look like this:



obtaining reaction rules

- we have representations for 1) and 2)
- for 3), there have been manual attempts, but this does not scale:
- chemical knowledge doubles every decade, currently around 11 million reactions known
- solution: we extract reaction rules from an existing database (Reaxys),
  this results in 301,671 rules

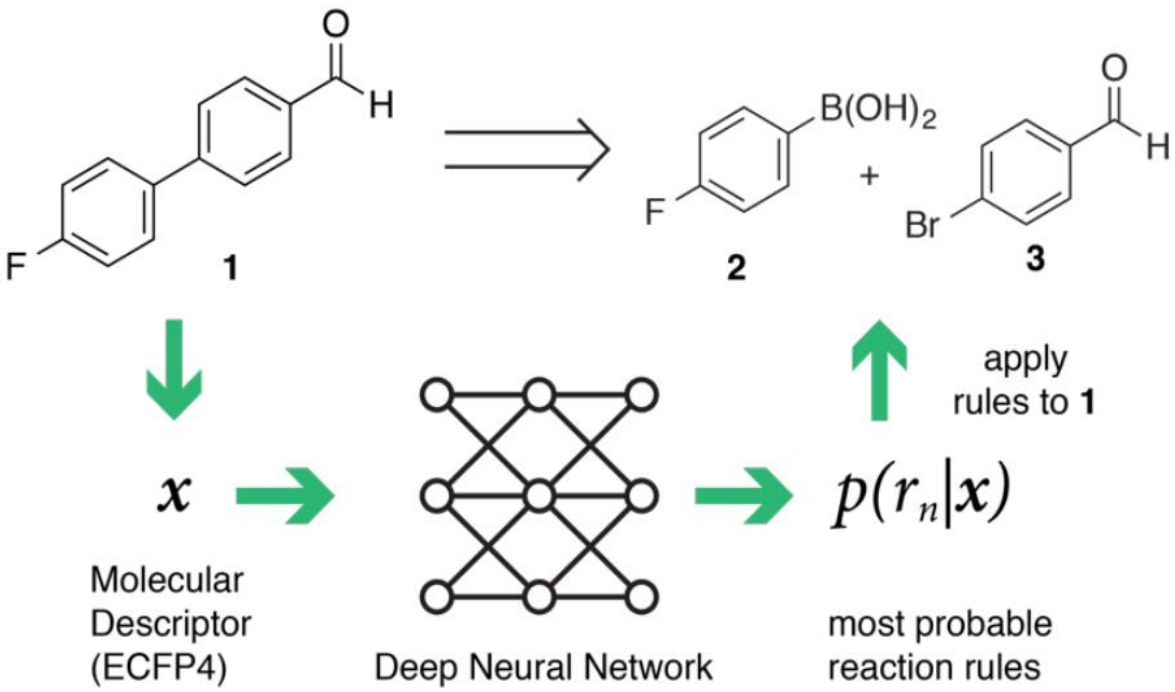
thus we can construct search trees with

- depth from around 5 up to around 20
- branching factors of approx. 46,000 (avg. applicable rules)
- for 5 steps, the node count is already a number with 23 digits
- **this is far too much, even for MCTS!**

help! how do we get the branching factor down?

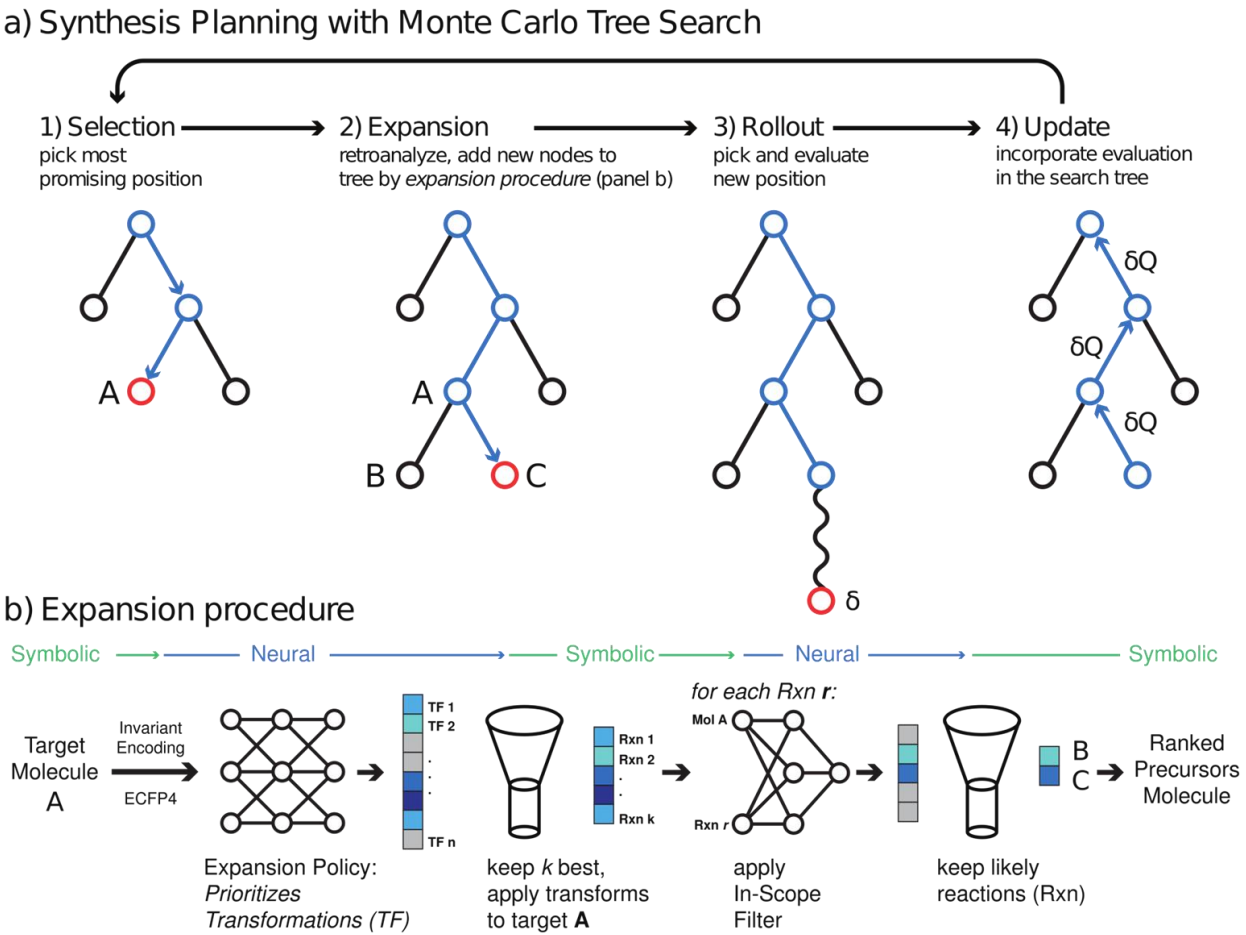
deep neural network for reaction preferences

- from all reaction rules we learn a preference for suggesting most likely ste
- the top50 accuracy is already 0.73 for the whole ruleset
- this reduces the branching factor to 50

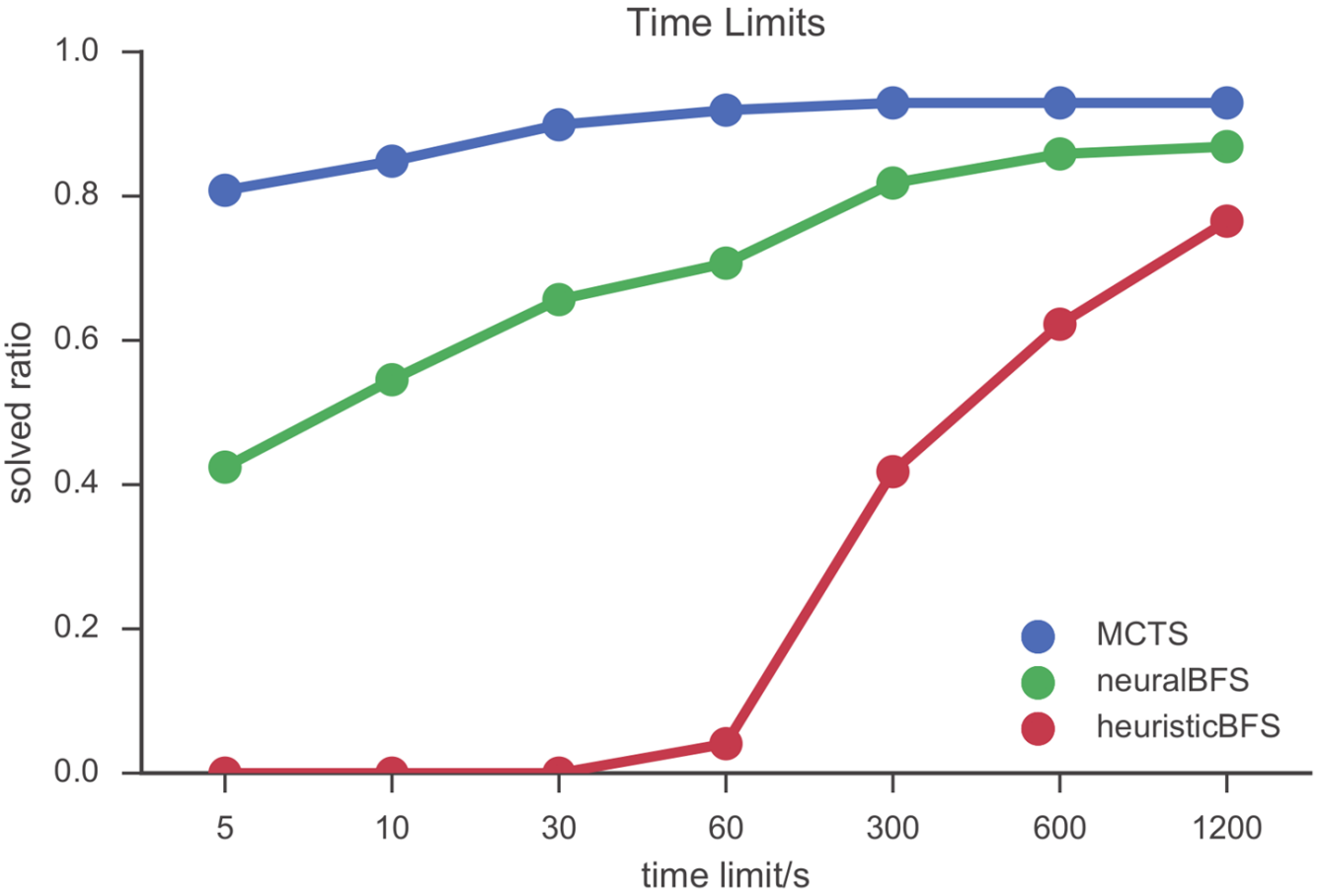


Segler, Waller, Chem. Eur. J. 2017, DOI: 10.1002/chem.201605499

putting everything together



how does it perform?



paper

nature
International journal of science

PDF

Article | Published: 28 March 2018

Planning chemical syntheses with deep neural networks and symbolic AI

Marwin H. S. Segler ✉, Mike Preuss & Mark P. Waller ✉

Nature **555**, 604–610 (29 March 2018) | Download Citation ⬇

Abstract

To plan the syntheses of small organic molecules, chemists use retrosynthesis, a problem-solving technique in which target molecules are recursively transformed into increasingly simpler precursors. Computer-aided retrosynthesis would be a valuable tool but at present it is slow and provides results of unsatisfactory quality. Here we use Monte Carlo tree search and symbolic artificial intelligence (AI) to discover retrosynthetic routes. We combined Monte Carlo tree search

Editorial Summary

Computers teach themselves to make molecules

Chemical reaction databases that are automatically filled from the literature have made the planning of chemical syntheses, whereby... show more

Sections | Figures | References

Abstract

Main

Training the expansion and rollout policies

Prediction with the in-scope filter network

Integrating neural networks and MCTS

Evaluating the performance characteristics of 3N-MCTS

Quantitative evaluation

General Game Playing (GGP)

General Game Playing

General Artificial Intelligence

- one player can play several games
- without actually knowing much about the game
- considered important step toward AGI (or GAI? :-))
- however, games very simple

- important publications:

Gamemaster
Leaderboard

Sign In

Player	Matches	Score
mozart_the_cat	10492	40
indy	14436	35
lara	14337	35
alloy	2491	21
sancho	628	18
drop_table_teams	315	16
ocminimax	591	15
ocmcs	1037	4
aaa	0	0
afflach	0	0
algue_rythme	0	0
ares	0	0
blinky	0	0
dfadsfadfadc	0	0
egghead	0	0
perceval_ens	0	0
ravar	0	0

NB: Scores and counts do not include matches with more than 3 recorded errors.

Comments and complaints to genesereth@stanford.edu.

Gamemaster
Games

Sign In

Id	Name	Roles	Creator
alquerque	Alquerque	2	genesereth@stanford.edu
asylum	Asylum	2	genesereth@stanford.edu
badconnectfour	Bad Connect Four	2	genesereth@stanford.edu
badskirmish	Bad Skirmish	2	genesereth@stanford.edu
badtictactoe	Bad Tic Tac Toe	2	genesereth@stanford.edu
battleofnumbers	Battle of Numbers	2	dustinsfink@gmail.com
bestbuttonsandlights	Best Buttons and Lights	1	genesereth@stanford.edu
bestknightstour	Best Knights Tour	1	genesereth@stanford.edu
bestsukoshi	Best Sukoshi	1	genesereth@stanford.edu
breakthrough	Breakthrough	2	genesereth@stanford.edu
breakthroughsmall	Breakthrough Small	2	genesereth@stanford.edu
buttonsandlights	Buttons and Lights	1	genesereth@stanford.edu
checkersonabarrelnokings	Checkers on a Barrel No Kings	2	genesereth@stanford.edu
chinesecheckers	Chinese Checkers	6	genesereth@stanford.edu
chinesecheckers2	Chinese Checkers 2	2	genesereth@stanford.edu
chinesecheckers3	Chinese Checkers 3	3	genesereth@stanford.edu
chinesecheckers4	Chinese Checkers 4	4	genesereth@stanford.edu
chinook	Chinook	2	genesereth@stanford.edu
connectfour	Connect Four	2	genesereth@stanford.edu
connectfour3p	Connect Four 3 Player	3	genesereth@stanford.edu
connectfourteam	Connect Four Team	4	chuchro3@stanford.edu
donttouch	Dont Touch	2	bhz@stanford.edu
dualhunter	Dual Hunter	2	genesereth@stanford.edu
dualrainbow	Dual Rainbow	2	genesereth@stanford.edu
duidoku	Duidoku	2	genesereth@stanford.edu
duikoshi	Duikoshi	2	genesereth@stanford.edu
eightpuzzle	Eight Puzzle	1	genesereth@stanford.edu
farmingquandries	Farming Quandries	2	ptr.pham@gmail.com
firesheep	Fire Sheep	2	ptr.pham@gmail.com
freeforall	Freeforall	2	genesereth@stanford.edu
hamilton	Hamilton	1	genesereth@stanford.edu
hex	Hex	2	genesereth@stanford.edu
horseshoe	Horseshoe	2	genesereth@stanford.edu
hunter	Hunter	1	genesereth@stanford.edu
hunterbig	Hunter Big	1	bhz@stanford.edu
jointbuttonsandlights	Joint Buttons and Lights	1	genesereth@stanford.edu

Pell, Barney. "METAGAME: A new challenge for games and learning." (1992).

Genesereth, Michael, Nathaniel Love, and Barney Pell. "General game playing: Overview of the AAAI competition." AI magazine 26.2 (2005): 62-62.

http://ggp.stanford.edu/ggp

GVGAI environment

- general video-game playing (GVGP) succeeds general game-playing
- targeted at more complex, non-deterministic games
- language first shaped during Dagstuhl seminar CI and AI in games, 2012

- important publications:

John Levine, Clare B. Congdon, Michal Bída, Marc Ebner, Graham Kendall, Simon Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General Video Game Playing. Dagstuhl Follow-up, 6:1–7, 2013.

Tom Schaul. A Video Game Description Language for Model-based or Interactive Learning. In Proceedings of the IEEE Conference on Computational Intelligence in Games, Niagara Falls, 2013. IEEE Press

http://www.gvgai.net still exists, but is not actively used any more (last competition 2019)

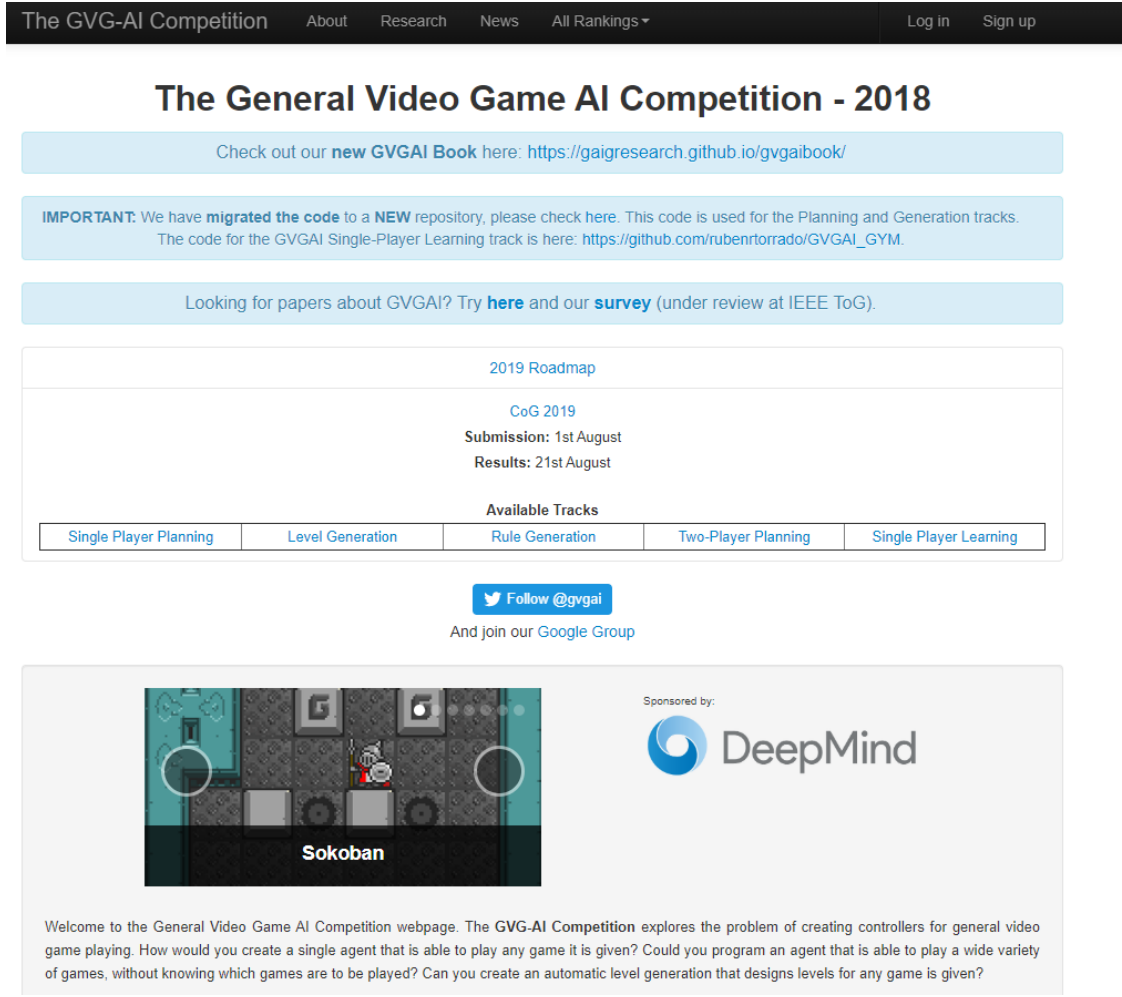
the new implementation is here: https://github.com/SUSTechGameAI/GVGAI_GYM

„old“ GVGAI website

- competitions ran for 5 years at major games conferences
- currently discontinued but we are using the “old” Java code with the new sprites and run only locally.
- learning (not planning) competitions are still running

http://www.aingames.cn/
https://github.com/SUSTechGameAI

- new sprite set:



learning or planning?

- boundaries between these two sometimes a bit fuzzy
- in game playing, we usually mean learning when we have no means to look into the future
- more specifically, when we have no "forward model" that can be used as simulation
- if we have a forward model we usually speak of planning
- here, we can compare the consequences of different actions and then roll back
- however, the forward model could be "learned" from past experiences
- or it is just a very sophisticated heuristic? ;-)

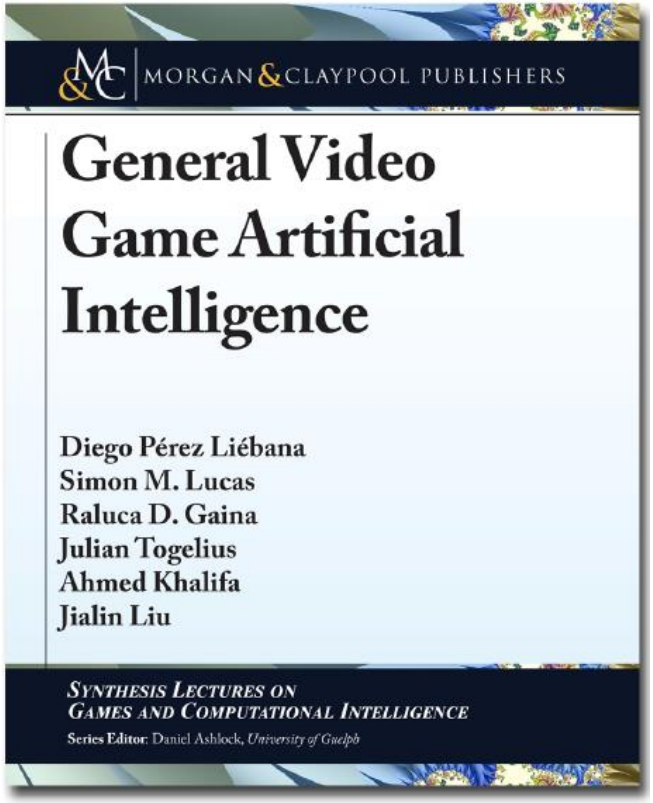
picture from Chuck Underwood on Pixabay

GVGAI book

- summarizes what happened in GVGAI from around 2013 to 2019
- different types of competitions:
  - planning
  - learning
  - procedural content generation

- https://gaigresearch.github.io/gvgaibook/

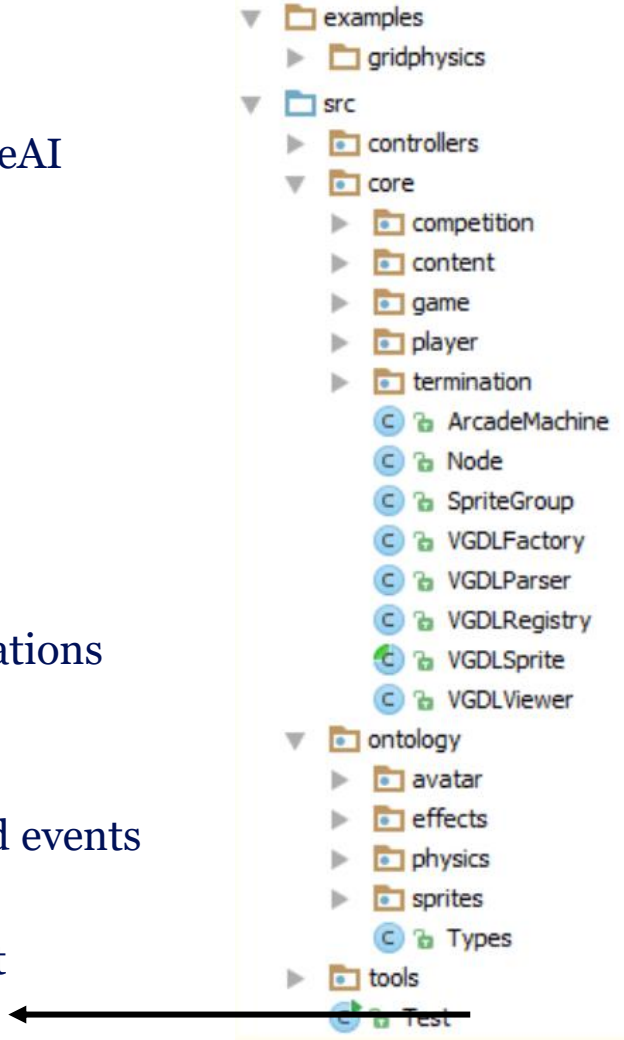
(chapters available for download)



code overview

on the github server https://github.com/SUSTechGameAI
go into GVGAI_GYM/gym_gvgai/envs/gvgai/

- examples → gridphysics: game and levels files
- controllers: sample working controllers
- core: core codebase of the framework
- core.competition: competition parameters
- core.content: game and sprite creation
- core.game: game engine, forward model and observations
- core.player: abstract class for players
- core.termination: game termination conditions
- ontology: definitions for sprites, avatars, physics and events
- tools: several useful classes
- **Test.java**: entry point to run the framework, now at src/tracks/singlePlayer/Test.java



VGDL – game definitions

```
BasicGame
    SpriteSet
        base    > Immovable    color=WHITE img=base
        avatar  > FlakAvatar   stype=sam
        missile > Missile
            sam  > orientation=UP    color=BLUE singleton=True img=spaceship
            bomb > orientation=DOWN  color=RED  speed=0.5 img=bomb
        alien   > Bomber       stype=bomb   prob=0.01  cooldown=3 speed=0.8 img=alien
        portal  >
            portalSlow  > SpawnPoint   stype=alien  cooldown=16   total=20 img=portal
            portalFast  > SpawnPoint   stype=alien  cooldown=12   total=20 img=portal

    LevelMapping
        0 > base
        1 > portalSlow
        2 > portalFast

    TerminationSet
        SpriteCounter      stype=avatar               limit=0 win=False
        MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True

    InteractionSet
        avatar  EOS  > stepBack
        alien   EOS  > turnAround
        missile EOS  > killSprite
        missile base > killSprite
        base bomb > killSprite
        base sam > killSprite scoreChange=1
        base   alien > killSprite
        avatar alien > killSprite scoreChange=-1
        avatar bomb  > killSprite scoreChange=-1
        alien  sam   > killSprite scoreChange=2
```

VGDL – level definitions

```
1  wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
2  w                                    w
3  w1                                   w
4  w000                                 w
5  w000                                 w
6  w                                    w
7  w                                    w
8  w                                    w
9  w                                    w
10 w    000        000000      000      w
11 w   00000     00000000    00000      w
12 w   0   0     00    00    00000      w
13 w                 A                  w
14 wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

a sample (random) controller

- realtime conditions important:
  - constructor max 1 sec
  - every action max 40 msec

- we do not need to have an idea of what actions actually mean
- but we get the action list and have to decide which to take

```java
package random; //The package name is the same as the username in the web.

public class Agent extends AbstractPlayer {

    protected Random randomGenerator;

    //Constructor. It must return in 1 second maximum.
    public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)
    {
        randomGenerator = new Random();
    }

    //Act function. Called every game step, it must return an action in 40 ms maximum.
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

        //Get the available actions in this game.
        ArrayList<Types.ACTIONS> actions = stateObs.getAvailableActions();

        //Determine an index randomly and get the action to return.
        int index = randomGenerator.nextInt(actions.size());
        Types.ACTIONS action = actions.get(index);

        //Return the action.
        return action;
    }
}
```

StateObservation (I)

- allows the agent to query the state of the game:
  - `StateObservation.getGameScore();`
  - `StateObservation.getGameTick();`
  - `StateObservation.getGameWinner();`
  - `StateObservation.isGameOver();`
  - `StateObservation.getWorldDimension()`
- . . . the state of the avatar:
  - `StateObservation.getAvatarPosition();`
  - `StateObservation.getAvatarSpeed();`
  - `StateObservation.getAvatarOrientation();`
  - `StateObservation.getAvatarResources();`
- . . . the available actions in the game:
  - `StateObservation.getAvailableActions();`
- . . . the history of events (collisions) in the game:
  - `StateObservation.getEventsHistory();`

StateObservation (II)

- . . . Observations in the game:
  - `StateObservation.getObservationGrid();`
  - `StateObservation.getNPCPositions(position?);`
  - `StateObservation.getImmovablePositions(position?);`
  - `StateObservation.getMovablePositions(position?);`
  - `StateObservation.getResourcesPositions(position?);`
  - `StateObservation.getPortalsPositions(position?);`
  - `StateObservation.getFromAvatarSpritesPositions(position?);`

what is an *Observation*? it is an object that contains:

- int itype: type of sprite of this observation
- int category: category of this observation (static, resource, npc, etc.)
- int obsID: unique ID for this observation
- Vector2 position: position of the observation
- Vector2 reference: reference (pivot) position
- double sqDist: distance from this observation to the reference

StateObservation (III)

in StateObservation:

- `1 ArrayList <Observation>[][] getObservationGrid();`

- bi-dimensional array, matching the level grid
- each ArrayList contains list of observations in that position

- `1 ArrayList <Observation>[] getNPCPositions();`
- `2 ArrayList <Observation>[] getNPCPositions(Vector2d reference);`
- returns a list of observations of NPC in the game
- as there can be NPCs of different type, each entry in the array corresponds to a sprite type
- every ArrayList contains a list of objects of type Observation
- if “reference” is given, observations are sorted by distance to the reference’s position

the forward model

also, in StateObservation:

```
StateObservation copy();
```

- create a copy of the StateObservation object

```
void advance(Types.ACTIONS action);
```

- advances the StateObservation object, applying the action supplied
- allows to simulate the effects of applying actions
- the StateObservation updates itself to reflect the next state
- important: the games are stochastic in nature!
  - the next state must be considered as a possible future state when applying a certain action.
  - the agent has the responsibility to deal with these inaccuracies.

evolutionary controller for the GVGAI

following a “classical” approach for General Video Game Playing:

- each individual is a player: determines how action decisions are made

this is offline training:

- EA determines the best agent first
- the submitted agent does not use online evolution

this is the GA-Eater approach:
train a rule set, apply the learned rules

closed loop – open loop

- until now we have always represented a state with a node
- this is called **closed loop**
- we can also represent the whole path that leads to a specific state in a node
- this is called **open loop**
- especially useful if our game is non-deterministic, and much smaller trees
- however, usually performing worse than closed loop if enough time

- another way to cope with chance events is to use AMAF value estimates / RAVE / GRAVE
- the key idea here is to generalize samples over more moves to get reasonable values quicker

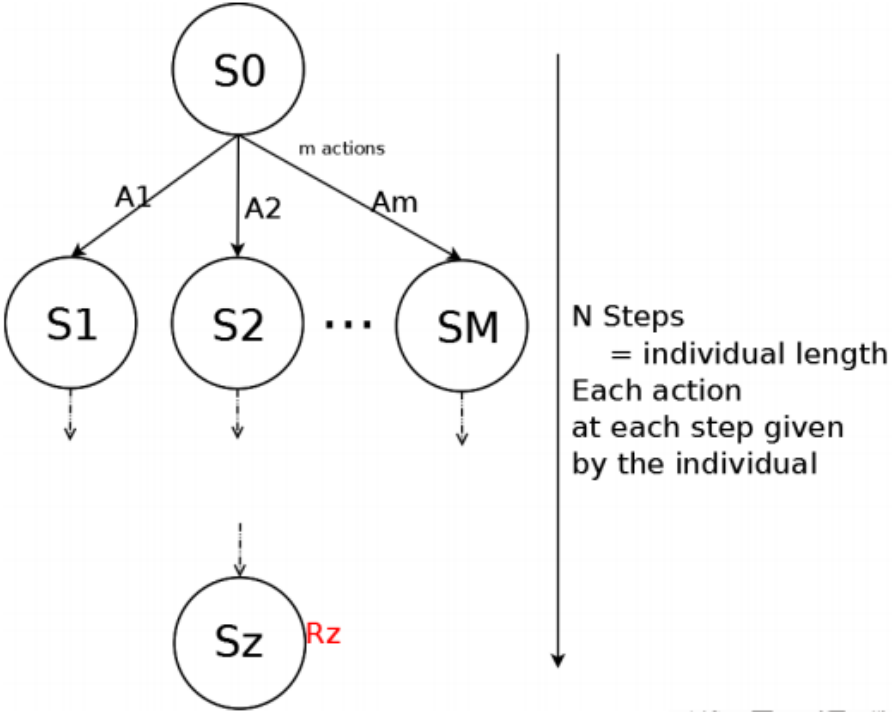
picture from Free Photos on Pixabay

rolling horizon EA

basic idea:

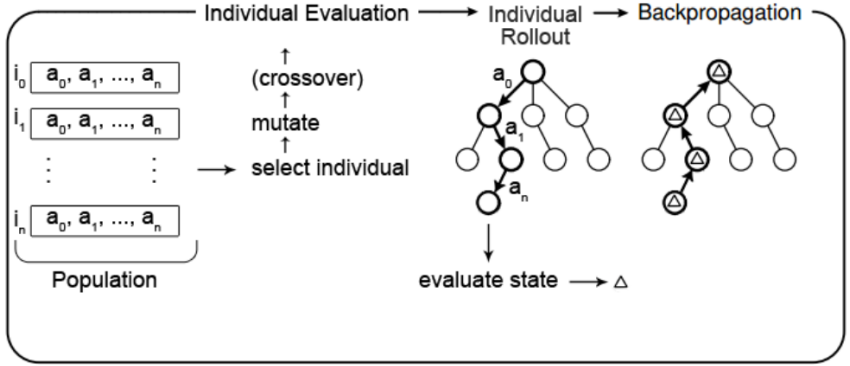
no offline training, evolution is used online, while playing the game

- each individual is a sequence of actions to apply from the current state (open loop control)
- fitness: evaluation of the state reached after applying the sequence of actions
- within the real-time constraints, the RH-EA evolves the best sequence of actions
- evolve the population normally: mutation, crossover, elitism, etc.
- when the process is over, apply first action of the best individual
- SampleGA sample controller



combine RHEA and MCTS (or other enhancements)

- can't we use a "statistical tree" to memorize our experiences?
- or just add rollouts as in MCTS?
- or shift the buffer (move with applied action) and keep the rest of the precomputed path?



- yes, shift buffer and added rollouts work well, on par with MCTS on 20 GVGAI test games, but cannot say in advance which will work for which game (not very well understood)

#	Algorithm	Points	Avg. Wins	G-0	G-4	G-13	G-15	G-18	G-22	G-25	G-29	G-36	G-46	G-49	G-50	G-58	G-60	G-61	G-67	G-75	G-77	G-84	G-91
1	EA-shift-roll	430	42.05 (2.48)	18	25	18	25	18	18	18	25	25	25	18	25	18	25	25	25	18	18	18	25
2	MCTS	430	41.30 (1.76)	25	18	25	18	25	25	25	18	18	18	25	18	25	18	18	18	25	25	25	18

Table IV: Configuration 10-14, $R = 5$. Best algorithm found compared with MCTS. In this order, the table shows the rank of the algorithms, their name, total F1 points, average of victories and F1 points achieved on each game.

Raluca D. Gaina, Simon M. Lucas, Diego Perez Liebana: Rolling horizon evolution enhancements in general video game playing. CIG 2017: 88-95

readings

very good intro paper that already explains a lot of variants:

- Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton, **A Survey of Monte Carlo Tree Search Methods**, *IEEE Transactions on Computational Intelligence and AI in Games* (2012), pp. Vol. 4:1 143-191.

multi-objective MCTS:

- Diego Perez Liebana, Sanaz Mostaghim, Spyridon Samothrakis, Simon M. Lucas: Multiobjective Monte Carlo Tree Search for Real-Time Games. IEEE Trans. Comput. Intell. AI Games 7(4): 347-360 (2015)

self-adapting RHEA:

- Raluca D. Gaina, Diego Perez Liebana, Simon M. Lucas, Chiara F. Sironi, Mark H. M. Winands: Self-Adaptive Rolling Horizon Evolutionary Algorithms for General Video Game Playing. CoG 2020: 367-374

- start here for more: https://dblp.org/pid/164/9035.html
  (Diego is probably the guy who did most GVGAI papers)

take home

- game trees as primary tools for board game planning/learning
- Monte Carlo tree search necessary for larger games where min-max is too slow
- MCTS can be combined with heuristics
- for very large state spaces, MCTS is too slow: merge states (macro states), focus on subareas
- for planning, a forward model is necessary; we have to learn one if we do not have one
- Rolling Horizon EA competitive to plain MCTS in GVGAI (surprisingly)

picture from OpenClipart-Vectors on Pixabay