Foundations of Artificial Intelligence

G5. Board Games: Monte-Carlo Tree Search Framework

Malte Helmert

University of Basel

May 22, 2024

Board Games: Overview

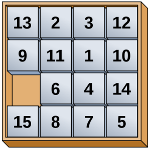
chapter overview:

- G1. Introduction and State of the Art
- G2. Minimax Search and Evaluation Functions
- G3. Alpha-Beta Search
- G4. Stochastic Games
- G5. Monte-Carlo Tree Search Framework
- G6. Monte-Carlo Tree Search Variants

Introduction

Monte-Carlo Tree Search

algorithms considered previously:

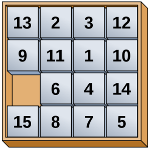


systematic search:

- systematic exploration of search space
- computation of (state) quality follows performance metric

Monte-Carlo Tree Search

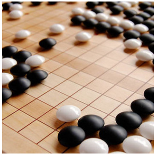
algorithms considered previously:



systematic search:

- systematic exploration of search space
- computation of (state) quality follows performance metric

algorithms considered today:

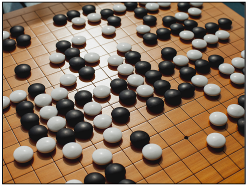


search based on Monte-Carlo methods:

- sampling of game simulations
- estimation of (state) quality by averaging over simulation results

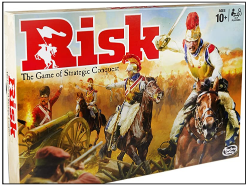
Game Applications

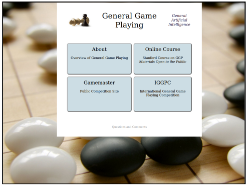
board games



hidden information games

stochastic games





general game playing

real-time strategy games

dynamic difficulty adjustment

Świechowski et al., Monte Carlo Tree Search: a review of recent modifications and applications (2023)

Applications Beyond Games

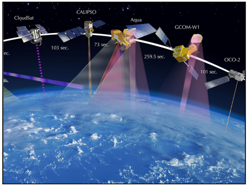
story generation

chemical synthesis

UAV routing

coast security

forest harvesting



Earth observation

Świechowski et al., Monte Carlo Tree Search: a review of recent modifications and applications (2023)

MCTS Environments

MCTS environments cover entire spectrum of properties.

We study MCTS under the same restrictions as before, i.e.,

- environment classification,
- problem solving method,
- objective of the agent and
- performance measure

are identical to Chapters G1–G3.

MCTS extensions exist that allow us to drop most restrictions.

Monte-Carlo Tree Search

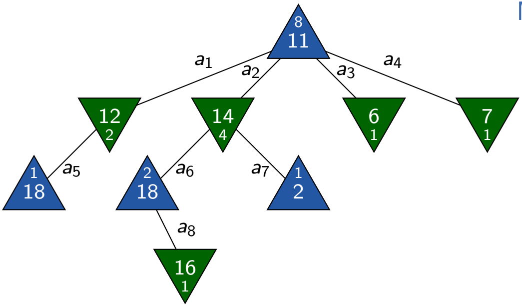
Data Structures

Monte-Carlo tree search

- is a tree search variant
  $\rightsquigarrow$ no closed list
- iteratively performs game simulations from the initial position (called trial or rollout)
  $\rightsquigarrow$ no (explicit) open list

$\rightsquigarrow$ MCTS nodes are the only used data structure

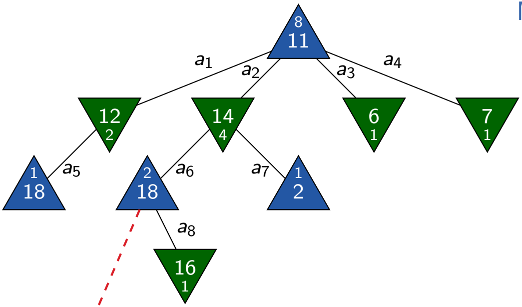
Data Structure: MCTS Nodes



MCTS nodes store

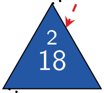
- a reached position
- how it was reached
- its successors
- a utility estimate ($\hat{v}$)
- a visit counter (N)
- possibly additional information

Data Structure: MCTS Nodes



MCTS nodes store

- a reached position
- how it was reached
- its successors
- a utility estimate ($\hat{v}$)
- a visit counter (N)
- possibly additional information



position:	not displayed
move:	a_6
successors:	[**none**, 16 1]
$\hat{v}$:	18
N:	2
...:	...

Monte-Carlo Tree Search: Idea

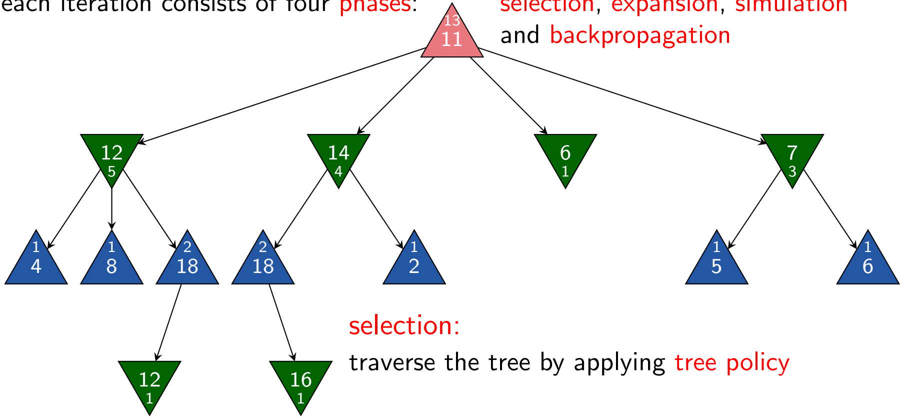
Monte-Carlo Tree Search (MCTS) ideas:

- build a partial game tree
- by performing trials as long as resources (deliberation time, memory) allow
- initially, the tree contains only the root node
- each trial adds (at most) one node to the tree

after termination, play the associated move of a successor of the root node with highest utility estimate

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation

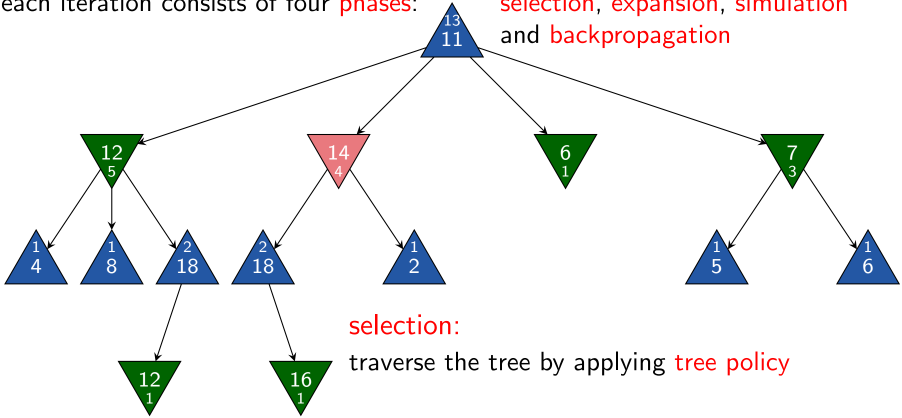


selection:

traverse the tree by applying tree policy

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation

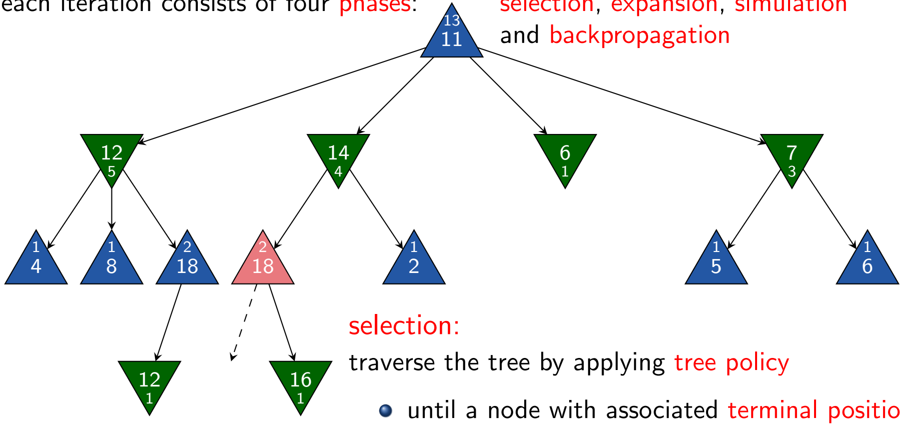


selection:

traverse the tree by applying tree policy

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



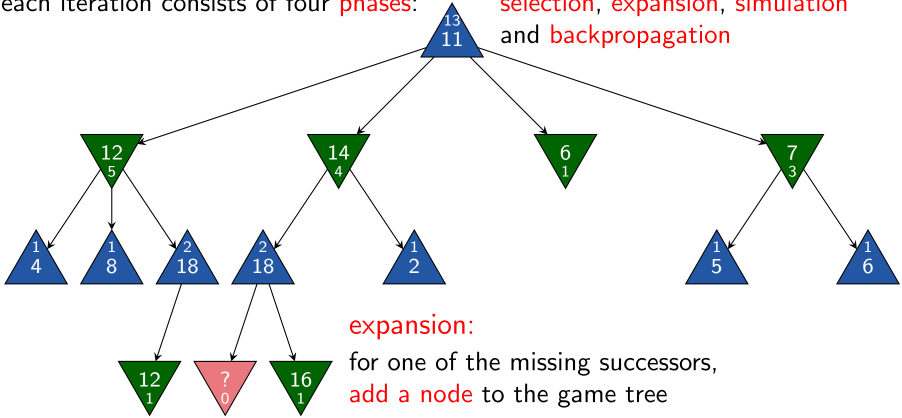
selection:

traverse the tree by applying tree policy

- until a node with associated terminal position
- or a node with missing successor is reached

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation

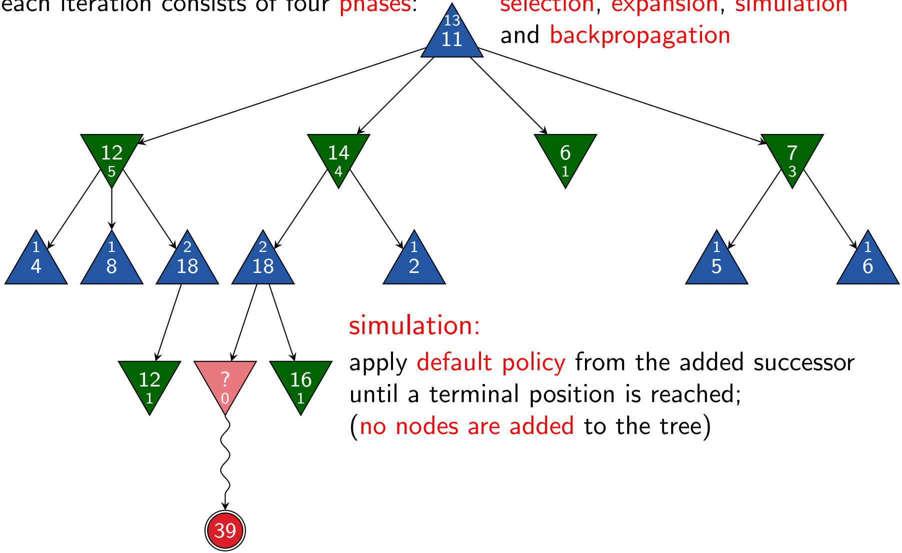


expansion:

for one of the missing successors, add a node to the game tree

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation

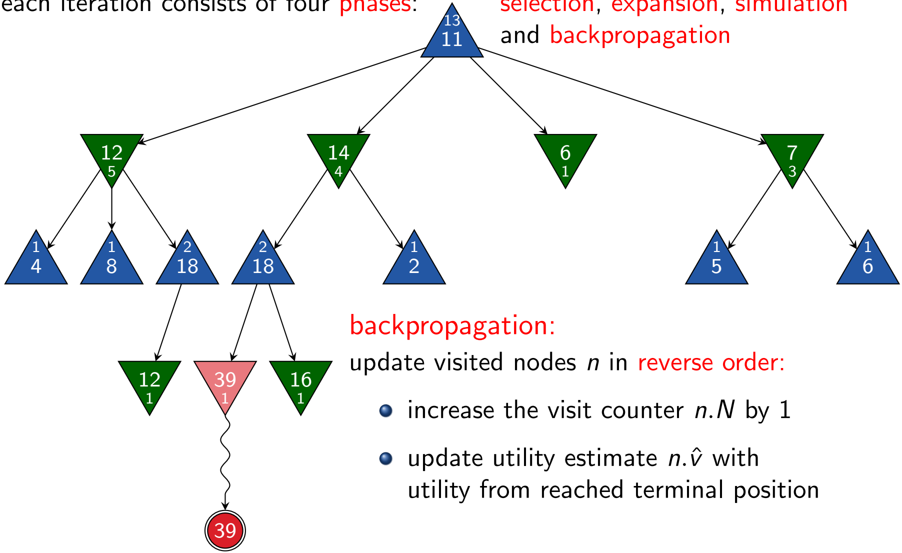


simulation:

apply default policy from the added successor until a terminal position is reached;
(no nodes are added to the tree)

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



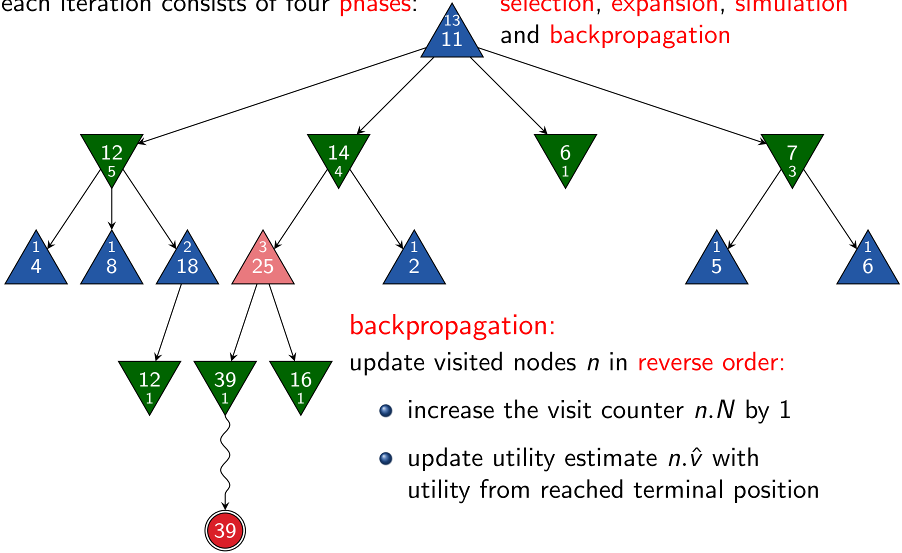
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



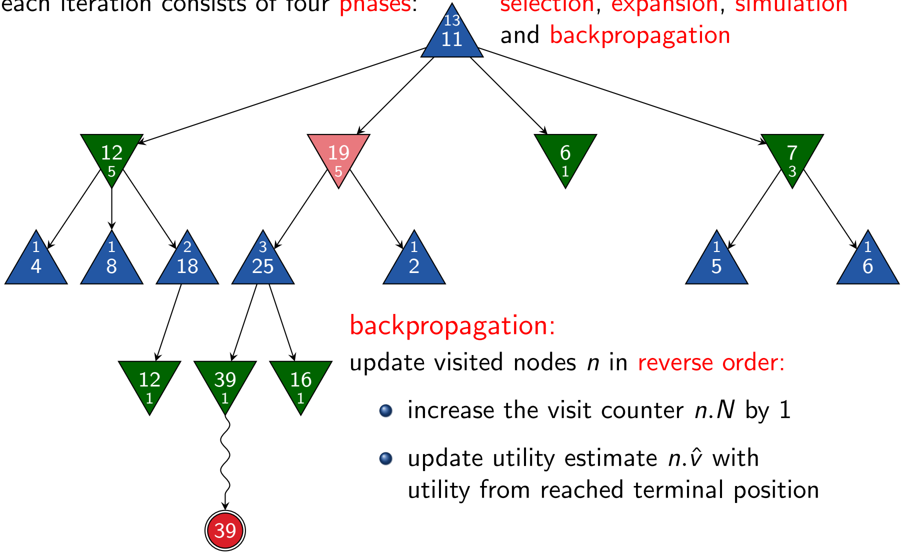
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



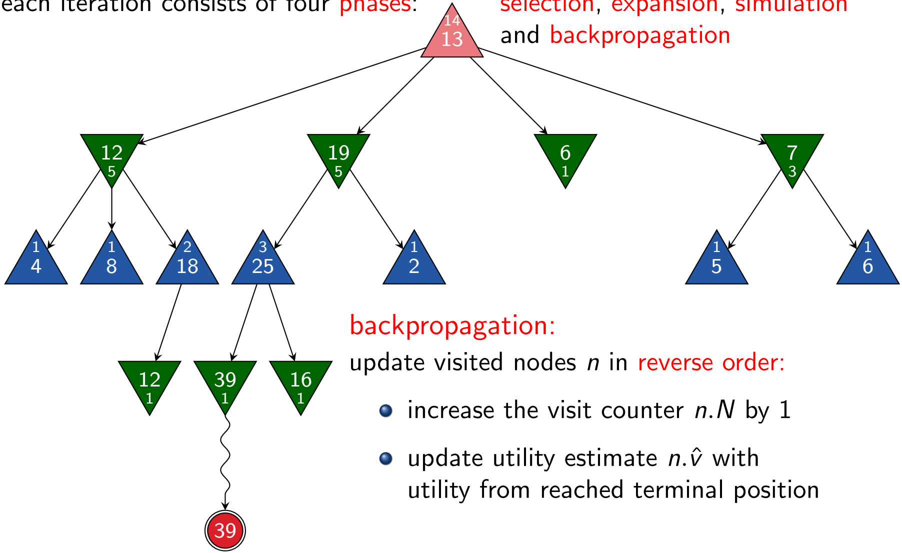
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



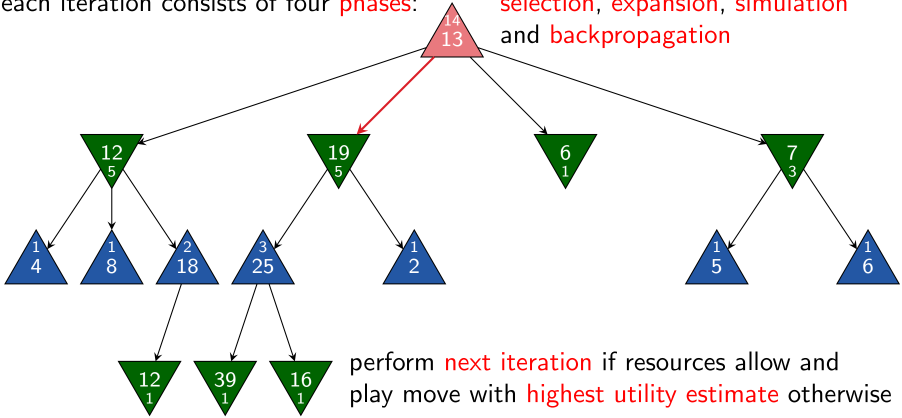
backpropagation:

update visited nodes n in reverse order:

- increase the visit counter $n.N$ by 1
- update utility estimate $n.\hat{v}$ with utility from reached terminal position

Idea and Example

each iteration consists of four phases: selection, expansion, simulation and backpropagation



perform next iteration if resources allow and play move with highest utility estimate otherwise

Monte-Carlo Tree Search: Pseudo-Code

Monte-Carlo Tree Search

$n_0 :=$ create_root_node()
while time_allows():
    visit_node(n_0)
$n_{\text{best}} := \arg\max_{n \in \text{succ}(n_0)} n.\hat{v}$
return n_{best}.move

Monte-Carlo Tree Search: Pseudo-Code

function visit_node(n)

```
if is_terminal(n.position):
    utility := utility(n.position)
else:
    s := n.get_unvisited_successor()
    if s is none:
        n' := apply_tree_policy(n)
        utility := visit_node(n')
    else:
        utility := simulate_game(s)
        n.add_and_initialize_child_node(s, utility)
n.N := n.N + 1
n.v̂ := n.v̂ + (utility − n.v̂) / n.N
return utility
```

$$n.\hat{v} := n.\hat{v} + \frac{utility - n.\hat{v}}{n.N}$$

Summary

Summary

- Monte-Carlo methods compute averages over a number of random samples.
- Monte-Carlo Tree Search (MCTS) algorithms simulate a playout of the game
- and iteratively build a search tree, adding (at most) one node in each iteration.
- MCTS is parameterized by a tree policy and a default policy.