Faculty of Mathematics and Physics
Charles University

20.12.2021

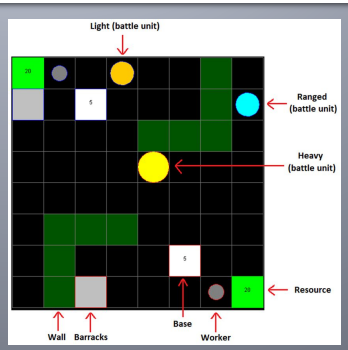
GAME DEVELOPMENT @CUNI.CZ

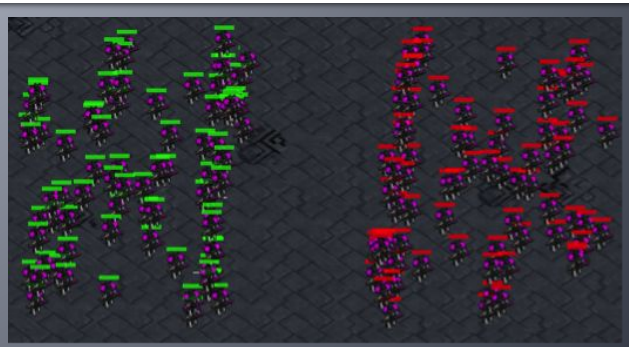
matfyz

NAIL122 AI for Games

Modeling RTS Combat Situations II

Jakub Gemrot



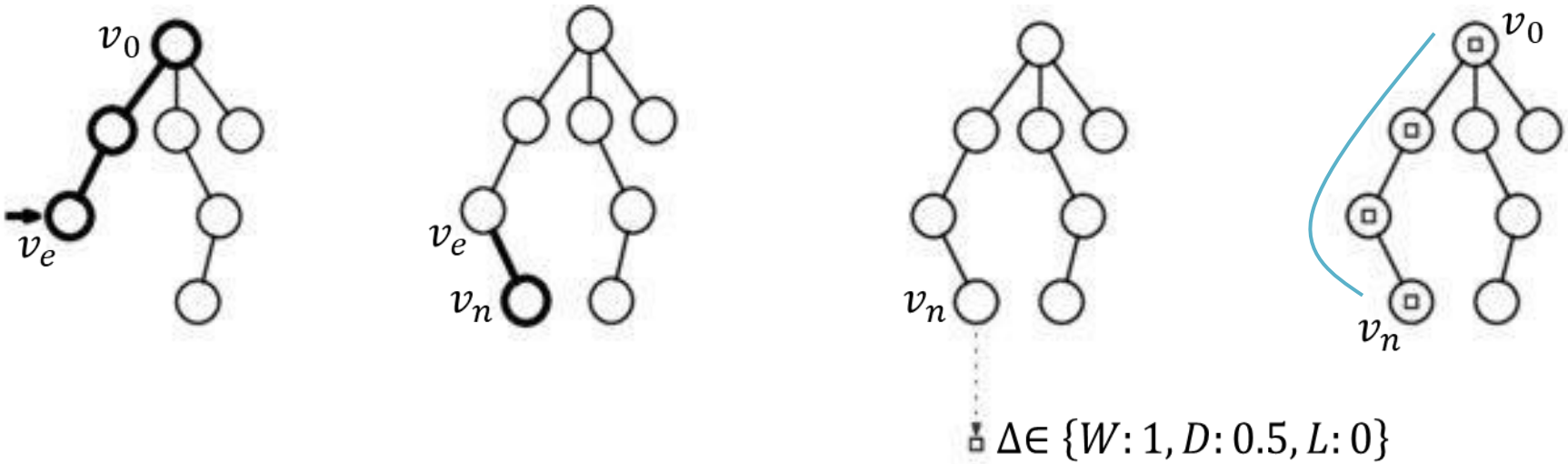


MCTS – Reminder

MCTS
High-level description

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Selection	Expansion	Simulation	Back-propagation
Tree traversed using *tree policy*	New node added to the tree (selected using the *tree policy*)	Rollouts are played from new node using *default policy*	Final state value is backpropagated to parent nodes

Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS formulated for sequential games. How about simultaneous ones?

=> UCT considering durations (UCTCD)

Churchill, D., & Buro, M. (2013, August). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1-8). IEEE.

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
procedure UCTCD(State s)
    root ← new Node
    for i ← 1 to maxTraversals do
        Traverse(root, Clone(s))
        if timeElapsed > timeLimit then break
    return most visited move at root

procedure TRAVERSE(Node n, State s)
    if n.visits = 0 then
        UpdateState(n, s, true)
        score ← s.eval()
    else
        UpdateState(n, s, false)
        if n.isTerminal() then
            score ← s.eval()
        else
            if !n.hasChildren() then
                generateChildren(s, n)
            score ← Traverse(SelectNode(n), s)
    n.visits++
    n.updateTotalScore(score)        ▷ w.r.t. player to move
    return score

procedure SELECTNODE(Node n)
    bestScore ← −∞
    for child c in n.getChildren() do
        if c.visits = 0 then return c
        score ← c.totalScore / c.visits +
                K · √(log (n.visits)/c.visits)
        if score > bestScore then
            bestScore ← score
            bestNode ← c
    return bestNode

procedure UPDATESTATE(Node n, State s, bool leaf)
    if (n.type ≠ FIRST) or leaf then
        if n.type = SECOND then
            s.makeMove(n.parent.move)
        s.makeMove(n.move)
```

Main idea: do not store states, only actions; mark nodes as FIRST, SECOND or SOLO. Do not apply actions in FIRST, wait for SECOND.

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
procedure UCTCD(State s)
    root ← new Node
    for i ← 1 to maxTraversals do
        Traverse(root, Clone(s))
        if timeElapsed > timeLimit then break
    return most visited move at root

procedure TRAVERSE(Node n, State s)
    if n.visits = 0 then
        UpdateState(n, s, true)
        score ← s.eval()
    else
        UpdateState(n, s, false)
        if n.isTerminal() then
            score ← s.eval()
        else
            if !n.hasChildren() then
                generateChildren(s, n)
            score ← Traverse(SelectNode(n), s)
    n.visits++
    n.updateTotalScore(score)        ▷ w.r.t. player to move
    return score
```

Top-level of MCTS, limited both by max number of iterations and time.
MCTS iteration == `Traverse`.
At the beginning of each iteration, we do `Clone(s)` to always work with fresh state clone.

```
            bestNode ← c
    return bestNode

procedure UPDATESTATE(Node n, State s, bool leaf)
    if (n.type ≠ FIRST) or leaf then
        if n.type = SECOND then
            s.makeMove(n.parent.move)
        s.makeMove(n.move)
```

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
1:  procedure UCTCD(State s)
2:      root ← new Node
3:      for i ← 1 to maxTraversals do
4:          Traverse(root, Clone(s))
5:          if timeElapsed > timeLimit then break
6:      return most visited move at root

8:  procedure TRAVERSE(Node n, State s)
9:      if n.visits = 0 then
10:         UpdateState(n, s, true)
11:         score ← s.eval()
12:     else
13:         UpdateState(n, s, false)
14:         if n.isTerminal() then
15:             score ← s.eval()
16:         else
17:             if !n.hasChildren() then
18:                 generateChildren(s, n)
19:             score ← Traverse(SelectNode(n), s)
20:     n.visits++
21:     n.updateTotalScore(score)        ▷ w.r.t. player to move
22:     return score

24: procedure SELECTNODE(Node n)
25:     bestScore ← −∞
26:     for child c in n.getChildren() do
27:         if c.visits = 0 then return c
28:         score ← c.totalScore / c.visits +
29:                 K · √(log (n.visits)/c.visits)
30:         if score > bestScore then
31:             bestScore ← score
```

`Traverse` method contains both selection, expansion and back-propagation; it is formulated as a recursive procedure (row 19).

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
1:  pr
2:
3:
4:
5:
6:

8:  procedure TRAVERSE(Node n, State s)
9:      if n.visits = 0 then
10:         UpdateState(n, s, true)
11:         score ← s.eval()
12:     else
13:         UpdateState(n, s, false)
14:         if n.isTerminal() then
15:             score ← s.eval()
16:         else
17:             if !n.hasChildren() then
18:                 generateChildren(s, n)
19:             score ← Traverse(SelectNode(n), s)
20:     n.visits++
21:     n.updateTotalScore(score)        ▷ w.r.t. player to move
22:     return score
```

`SelectNode` implements MCTS tree-policy using standard UCB a.k.a. UCB-1.

```
24: procedure SELECTNODE(Node n)
25:     bestScore ← −∞
26:     for child c in n.getChildren() do
27:         if c.visits = 0 then return c
28:         score ← c.totalScore / c.visits +
29:                 K · √(log (n.visits)/c.visits)
30:         if score > bestScore then
31:             bestScore ← score
32:             bestNode ← c
33:     return bestNode

35: procedure UPDATESTATE(Node n, State s, bool leaf)
36:     if (n.type ≠ FIRST) or leaf then
37:         if n.type = SECOND then
38:             s.makeMove(n.parent.move)
39:         s.makeMove(n.move)
```

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
procedure UCTCD(State s)
    root ← new Node
    for i ← 1 to maxTraversals do
        Traverse(root, Clone(s))
        if timeElapsed > timeLimit then break
    return most visited move at root

procedure TRAVERSE(Node n, State s)
    if n.visits = 0 then
        UpdateState(n, s, true)
        if n.hasChildren() then
            generateChildren(s, n)
        score ← Traverse(SelectNode(n), s)
    n.visits++
    n.updateTotalScore(score)        ▷ w.r.t. player to move
    return score

procedure SELECTNODE(Node n)
    bestScore ← −∞
    do

procedure UPDATESTATE(Node n, State s, bool leaf)
    if (n.type ≠ FIRST) or leaf then
        if n.type = SECOND then
            s.makeMove(n.parent.move)
        s.makeMove(n.move)
```

1. `UpdateState` is used to advance the (cloned) state using "move" information that is maintained in each node; i.e. a node is associated with a move that was used to get to it.

3. SOLO nodes „not created"; sequence of SOLO node moves is aggregated within single node move of corresponding player.

2. Each node is also labeled as:
FIRST – simultaneous node, first player to decide
SECOND – simultaneous node, second player to decide

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
1:  procedure UCTCD(State s)
2:      root ← new Node
3:      for i ← 1 to maxTraversals do
4:          Traverse(root, Clone(s))
5:          if timeElapsed > timeLimit then break
6:      return most visited move at root

8:  procedure TRAVERSE(Node n, State s)
9:      if n.visits = 0 then
10:         UpdateState(n, s, true)
11:         score ← s.eval()
12:     else
13:         UpdateState(n, s, false)
14:         if n.isTerminal() then
15:             score ← s.eval()
16:         else
17:             if !n.hasChildren() then
18:                 generateChildren(s, n)
19:             score ← Traverse(SelectNode(n), s)
20:     n.visits++
21:     n.updateTotalScore(score)        ▷ w.r.t. player to move
22:     return score

24: procedure SELECTNODE(Node n)
25:     bestScore ← −∞
26:     for child c in n.getChildren() do
27:         if c.visits = 0 then return c
28:         score ← c.totalScore / c.visits +
29:                 K · √(log (n.visits)/c.visits)
30:         if score > bestScore then
31:             bestScore ← score
```

9-11:
We're at leaf we visited for the first time; do the move (10) and perform playout (11).

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
procedure UCTCD(State s)
    root ← new Node
    for i ← 1 to maxTraversals do
        Traverse(root, Clone(s))
        if timeElapsed > timeLimit then break
    return most visited move at root

procedure TRAVERSE(Node n, State s)
    if n.visits = 0 then
        UpdateState(n, s, true)
        score ← s.eval()
    else
        UpdateState(n, s, false)
        if n.isTerminal() then
            score ← s.eval()
        else
            if !n.hasChildren() then
                generateChildren(s, n)
            score ← Traverse(SelectNode(n), s)
    n.visits++
    n.updateTotalScore(score)        ▷ w.r.t. player to move
    return score

procedure SELECTNODE(Node n)
    bestScore ← −∞
    for child c in n.getChildren() do
        if c.visits = 0 then return c
        score ← c.totalScore / c.visits +
                K · √(log (n.visits)/c.visits)
        if score > bestScore then
            bestScore ← score
            bestNode ← c
    return bestNode

procedure UPDATESTATE(Node n, State s, bool leaf)
```

Otherwise, update the state with the move, and…

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
1:  procedure UCTCD(State s)
2:      root ← new Node
3:      for i ← 1 to maxTraversals do
4:          Traverse(root, Clone(s))
5:          if timeElapsed > timeLimit then break
6:      return most visited move at root

8:  procedure TRAVERSE(Node n, State s)
9:      if n.visits = 0 then
10:         UpdateState(n, s, true)
11:         score ← s.eval()
12:     else
13:         UpdateState(n, s, false)
14:         if n.isTerminal() then
15:             score ← s.eval()
16:         else
17:             if !n.hasChildren() then
18:                 generateChildren(s, n)
19:             score ← Traverse(SelectNode(n), s)
20:     n.visits++
21:     n.updateTotalScore(score)        ▷ w.r.t. player to move
22:     return score

24: procedure SELECTNODE(Node n)
25:     bestScore ← −∞
26:     for child c in n.getChildren() do
27:         if c.visits = 0 then return c
28:         score ← c.totalScore / c.visits +
29:                 K · √(log (n.visits)/c.visits)
30:         if score > bestScore then
31:             bestScore ← score
32:             bestNode ← c
33:     return bestNode

35: procedure UPDATESTATE(Node n, State s, bool leaf)
36:     if (n.type ≠ FIRST) or leaf then
37:         if n.type = SECOND then
```

14-15:
For terminal node, just return the value of the node.

UCT Considering Durations

a.k.a. UCTCD

Algorithm 1 UCT Considering Durations

```
1:  procedure UCTCD(State s)
2:      root ← new Node
3:      for i ← 1 to maxTraversals do
4:          Traverse(root, Clone(s))
5:          if timeElapsed > timeLimit then break
6:      return most visited move at root

8:  procedure TRAVERSE(Node n, State s)
9:      if n.visits = 0 then
10:         UpdateState(n, s, true)
11:         score ← s.eval()
12:     else
13:         UpdateState(n, s, false)
14:         if n.isTerminal() then
15:             score ← s.eval()
16:         else
17:             if !n.hasChildren() then
18:                 generateChildren(s, n)
19:             score ← Traverse(SelectNode(n), s)
20:     n.visits++
21:     n.updateTotalScore(score)        ▷ w.r.t. player to move
22:     return score

24: procedure SELECTNODE(Node n)
25:     bestScore ← −∞
26:     for child c in n.getChildren() do
27:         if c.visits = 0 then return c
28:         score ← c.totalScore / c.visits +
29:                 K · √(log (n.visits)/c.visits)
30:         if score > bestScore then
31:             bestScore ← score
32:             bestNode ← c
33:     return bestNode

35: procedure UPDATESTATE(Node n, State s, bool leaf)
36:     if (n.type ≠ FIRST) or leaf then
37:         if n.type = SECOND then
38:             s.makeMove(n.parent.move)
39:         s.makeMove(n.move)
```

17-19:
Otherwise ensure children by enumerating possible actions (17-18), i.e., expand the node "fully", and continue traversal (19).

UCT Considering Durations

a.k.a. UCTCD

Open questions

1. Generating all children at node (row 18) must be costly.
2. Recomputing the state each traversal must be costly.
3. Is it necessary to keep FIRST / SECOND node separate?

Question is, whether the alternative approaches would be better in terms of performance.

Portfolio Greedy Search

Combining scripts and playouts while avoiding tree-search.

Churchill, D., & Buro, M. (2013, August). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1-8). IEEE.

Portfolio Greedy Search

a.k.a. PGS

Motivation

Extreme solution for RTS combat is utilization of fixed scripts, e.g., NOK-AV or KITER. However, using single script is too restrictive while some may be decomposed to sub-scripts (KITER == NOK-AV + RETREAT). Can we extend this?

⇒ Portfolio Greedy Search

Use set of scripts; try to find a good assignments to units using greedy approach, by simulating possible exploitation by the opponent.

Portfolio Greedy Search

a.k.a. PGS

Algorithm sketch

Input: your units U_o, enemy units U_e, portfolio of scripts P

Goal: assign some script $s \in P$ to each of your unit $u \in U_o$.

[A] Use some default script and assign it to all enemy units.

[B] Check, which script (assigned to all your units) is the best response; use playouts to determine that.

[C] Check, which script (assigned to all enemy units) is the best response to [B].

[D] As long as you have the budget, try to improve on the assignments by alternating following steps:

- [D1] Try to find better assignments for the enemy given current assignments of scripts to your units
- [D2] Try to find better assignments for yourself given current assignments of scripts to enemy units

Portfolio Greedy Search

Algorithm 2 Portfolio Greedy Search

```
Portfolio P                                   ▷ Script Portfolio
Integer I                                     ▷ Improvement Iterations
Integer R                                     ▷ Self/Enemy Improvement Responses
Script D                                      ▷ Default Script

procedure PortfolioGreedySearch(State s, Player p)
    Script enemy[s.numUnits(opponent(p))].fill(D)
    Script self[] ← GetSeedPlayer(s, p, enemy)
    enemy ← GetSeedPlayer(s, opponent(p), self)
    self = Improve(s, p, self, enemy)
    for r = 1 to R do
        enemy = Improve(s, opponent(p), enemy, self)
        self = Improve(s, p, self, enemy)
    return generateMoves(self)

procedure GetSeedPlayer(State s, Player p, Script e[])
    Script self[s.numUnits(p)]
    bestValue ← −∞
    Script bestScript ← ∅
    for Script c in P do
        self.fill(c)
        value ← Playout(s, p, self, e)
        if value > bestValue then
            bestValue ← value
            bestScript ← c
    self.fill(bestScript)
    return self

procedure Improve(State s, Player p, Script self[],
                                                Script e[])
    for i = 1 to I do
        for u = 1 to self.length do
            if timeElapsed > timeLimit then return
            bestValue ← −∞
            Script bestScript ← ∅
            for Script c in P do
                self[u] ← c
                value ← Playout(s, p, self, e)
                if value > bestValue then
                    bestValue ← value
                    bestScript ← c
            self[u] ← bestScript
    return self
```

`PortfolioGreedySearch` inputs:

- Portfolio P, set of scripts
- Improvement iterations I
- Improvement response R
- Initial (default) script D

Portfolio Greedy Search

Algorithm 2 Portfolio Greedy Search

```
1:  Portfolio P                                   ▷ Script Portfolio
2:  Integer I                                     ▷ Improvement Iterations
3:  Integer R                                     ▷ Self/Enemy Improvement Responses
4:  Script D                                      ▷ Default Script
5:
6:  procedure PortfolioGreedySearch(State s, Player p)
7:      Script enemy[s.numUnits(opponent(p))].fill(D)
8:      Script self[] ← GetSeedPlayer(s, p, enemy)
9:      enemy ← GetSeedPlayer(s, opponent(p), self)
10:     self = Improve(s, p, self, enemy)
11:     for r = 1 to R do
12:         enemy = Improve(s, opponent(p), enemy, self)
13:         self = Improve(s, p, self, enemy)
        return generateMoves(self)
14:
15: procedure GetSeedPlayer(State s, Player p, Script e[])
16:     Script self[s.numUnits(p)]
17:     bestValue ← −∞
18:     Script bestScript ← ∅
19:     for Script c in P do
20:         self.fill(c)
21:         value ← Playout(s, p, self, e)
22:         if value > bestValue then
23:             bestValue ← value
24:             bestScript ← c
25:     self.fill(bestScript)
26:     return self
```

```
28: procedure Improve(State s, Player p, Script self[],
29:                                          Script e[])
30:     for i = 1 to I do
31:         for u = 1 to self.length do
32:             if timeElapsed > timeLimit then return
33:             bestValue ← −∞
34:             Script bestScript ← ∅
35:             for Script c in P do
36:                 self[u] ← c
37:                 value ← Playout(s, p, self, e)
38:                 if value > bestValue then
39:                     bestValue ← value
40:                     bestScript ← c
41:             self[u] ← bestScript
42:     return self
```

`PortfolioGreedySearch` sections:

Row 7 <-> [A] … default enemy

Row 8 <-> [B] … seed yourself

Row 9 <-> [C] … seed enemy

Rows 10-13 <-> [D] … improve

Portfolio Greedy Search

Algorithm 2 Portfolio Greedy Search

```
1:  Portfolio P                                   ▷ Script Portfolio
2:  Integer I                                     ▷ Improvement Iterations
3:  Integer R                                     ▷ Self/Enemy Improvement Responses
4:  Script D                                      ▷ Default Script
5:
6:  procedure PortfolioGreedySearch(State s, Player p)
7:      Script enemy[s.numUnits(opponent(p))].fill(D)
8:      Script self[] ← GetSeedPlayer(s, p, enemy)
9:      enemy ← GetSeedPlayer(s, opponent(p), self)
10:     self = Improve(s, p, self, enemy)
11:     for r = 1 to R do
12:         enemy = Improve(s, opponent(p), enemy, self)
13:         self = Improve(s, p, self, enemy)
        return generateMoves(self)
14:
15: procedure GetSeedPlayer(State s, Player p, Script e[])
16:     Script self[s.numUnits(p)]
17:     bestValue ← −∞
18:     Script bestScript ← ∅
19:     for Script c in P do
20:         self.fill(c)
21:         value ← Playout(s, p, self, e)
22:         if value > bestValue then
23:             bestValue ← value
24:             bestScript ← c
25:     self.fill(bestScript)
26:     return self
28: procedure Improve(State s, Player p, Script self[],
29:                                                Script e[])
30:     for i = 1 to I do
31:         for u = 1 to self.length do
32:             if timeElapsed > timeLimit then return
33:             bestValue ← −∞
34:             Script bestScript ← ∅
35:             for Script c in P do
36:                 self[u] ← c
37:                 value ← Playout(s, p, self, e)
38:                 if value > bestValue then
39:                     bestValue ← value
40:                     bestScript ← c
41:             self[u] ← bestScript
42:     return self
```

`GetSeedPlayer` sections:

16: we need to choose script for every unit of the player p

19: check on each possible script c from portfolio P

20: assign c to each unit of player p

21: check how good that is

22-24: save the best

Portfolio Greedy Search

Algorithm 2 Portfolio Greedy Search

```
1:  Portfolio P                                   ▷ Script Portfolio
2:  Integer I                                     ▷ Improvement Iterations
3:  Integer R                                     ▷ Self/Enemy Improvement Responses
4:  Script D                                      ▷ Default Script
5:
6:  procedure PortfolioGreedySearch(State s, Player p)
7:      Script enemy[s.numUnits(opponent(p))].fill(D)
8:      Script self[] ← GetSeedPlayer(s, p, enemy)
9:      enemy ← GetSeedPlayer(s, opponent(p), self)
10:     self = Improve(s, p, self, enemy)
11:     for r = 1 to R do
12:         enemy = Improve(s, opponent(p), enemy, self)
13:         self = Improve(s, p, self, enemy)
        return generateMoves(self)
14:
15: procedure GetSeedPlayer(State s, Player p, Script e[])
16:     Script self[s.numUnits(p)]
17:     bestValue ← −∞
18:     Script bestScript ← ∅
19:     for Script c in P do
20:         self.fill(c)
21:         value ← Playout(s, p, self, e)
22:         if value > bestValue then
23:             bestValue ← value
24:             bestScript ← c
25:     self.fill(bestScript)
26:     return self
```

```
28: procedure Improve(State s, Player p, Script self[],
29:                                          Script e[])
30:     for i = 1 to I do
31:         for u = 1 to self.length do
32:             if timeElapsed > timeLimit then return
33:             bestValue ← −∞
34:             Script bestScript ← ∅
35:             for Script c in P do
36:                 self[u] ← c
37:                 value ← Playout(s, p, self, e)
38:                 if value > bestValue then
39:                     bestValue ← value
40:                     bestScript ← c
41:             self[u] ← bestScript
42:     return self
```

`Improve` sections:

30: according to number of "attempts" I

31: find better script for each unit u of p greedy way …

35: … that means to try each script c

37-40: whether it would yield better result

41: use (assign) the best found

Experiments and Results

Comparing PGS to UCTCD, ABCD, and scripts.

Churchill, D., & Buro, M. (2013, August). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1-8). IEEE.

Portfolio Greedy Search

Results

Scenario setup

SparCraft; number of units $n \in \{8,16,32,50\}$; 50 is the largest battle in SCBW observed *(in pro-matches even more!)*

Symmetric vs. **Se**parated states

Sy-state -> symmetric around middle point, units can be in range
Se-state -> similar but put into distant so no unit can use attack first

Sy-state – artificial! Battle always start from Se-states.

Portfolio Greedy Search

Results

Scenario setup

Groups used

- n Protoss Dragoons (Strong Ranged)
- n Zerg Zerglings (Weak Melee)
- $n/2$ Protoss Dragoons (Strong Ranged) with $n/2$ Protoss Zealots (Strong Melee)
- $n/2$ Protoss Dragoons (Strong Ranged) with $n/2$ Terran Marines (Weak Ranged)
- $n/2$ Terran Marines (Weak Ranged) with $n/2$ Zerg Zerglings (Weak Melee)

100 random battles per group configuration per sy/se-state per number of units.

Portfolio Greedy Search

Results

Search settings

Alpha-Beta search:

- Time Limit: 40 ms
- Max Children: 20
- Evaluation: NOK-AV vs. NOK-AV Playout
- Transposition Table Size: 100000 (13.2 MB)

UCT search:

- Time Limit: 40 ms
- Max Children: 20
- Evaluation: NOK-AV vs. NOK-AV Playout
- Final Move Selection: Most Visited
- Exploration Constant: 1.6
- Child Generation: One-at-leaf
- Tree Size: No Limit (6 MB largest seen in 40 ms)

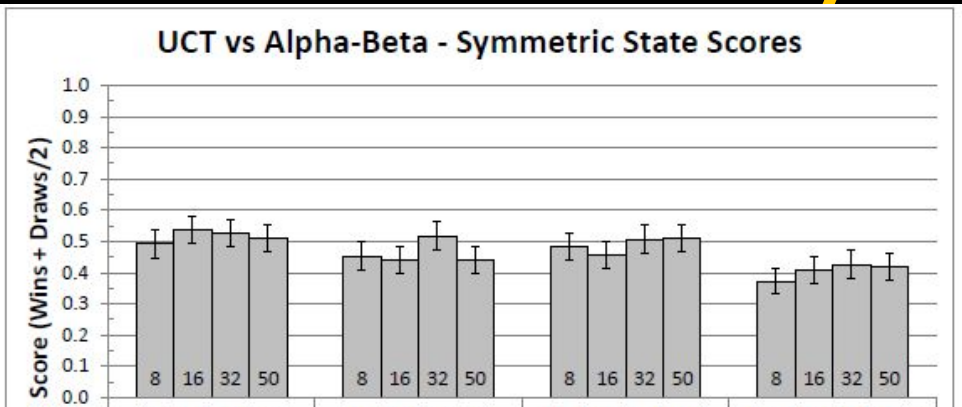
Portfolio Greedy search:

- Time Limit: 40 ms
- Improvement Iterations I: 1
- Response Iterations R: 0
- Initial Enemy Script: NOK-AV
- Evaluation: Improved Playout
- Portfolio Used: (NOK-AV, Kiter)

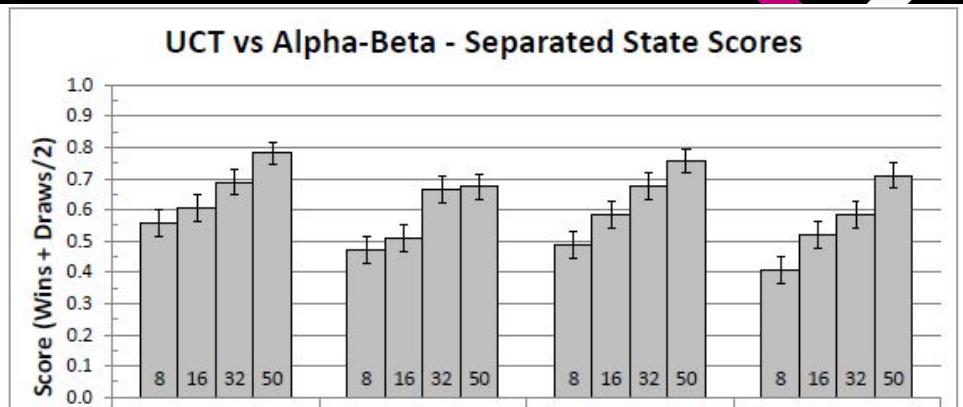
Fun fact: ABCD and UCTCD is working with low-level actions while PGS with scripts only.

More details in the paper…

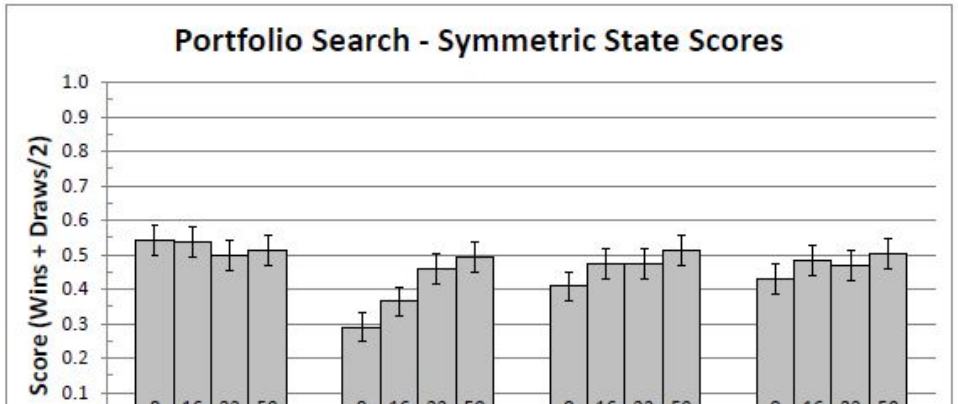
Portfolio Greedy Search



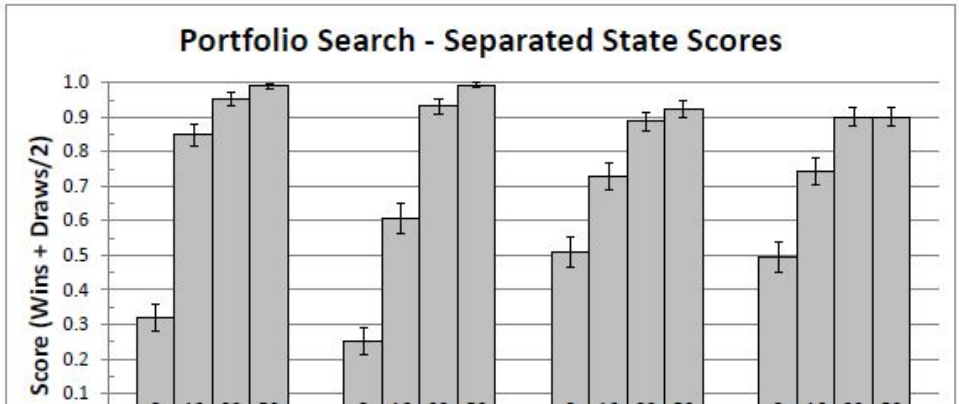
# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.492	0.454	0.482	0.373
16	0.539	0.439	0.456	0.409
32	0.526	0.518	0.507	0.427
50	0.509	0.44	0.51	0.419



# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.556	0.472	0.488	0.405
16	0.605	0.510	0.586	0.519
32	0.689	0.666	0.676	0.585
50	0.782	0.674	0.756	0.709

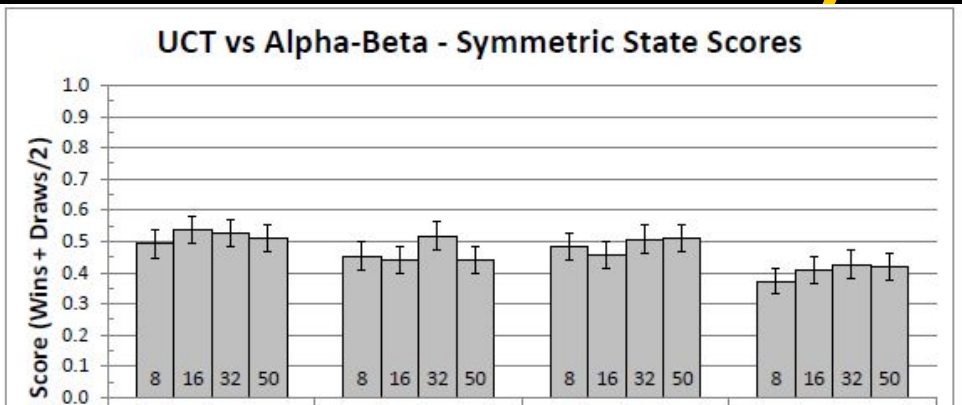


# Units	AB None	AB NOK-AV	UCT None	UCT NOK-AV
8	0.541	0.290	0.408	0.431
16	0.536	0.364	0.473	0.482
32	0.496	0.460	0.472	0.469
50	0.514	0.495	0.511	0.501

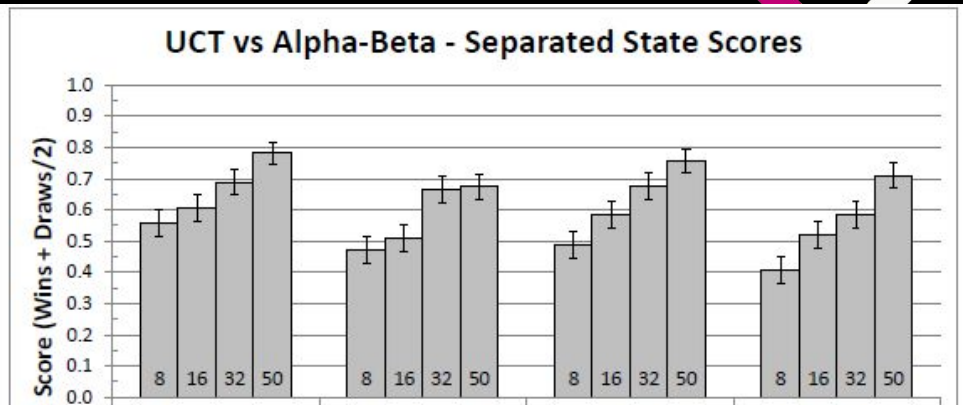


# Units	AB None	AB NOK	UCT None	UCT NOK
8	0.319	0.253	0.511	0.493
16	0.848	0.608	0.729	0.742
32	0.952	0.932	0.888	0.900
50	0.990	0.992	0.924	0.900

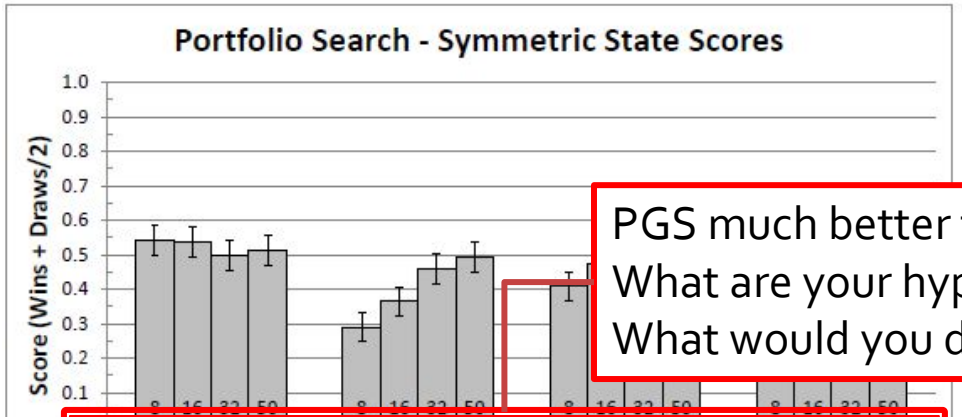
Portfolio Greedy Search



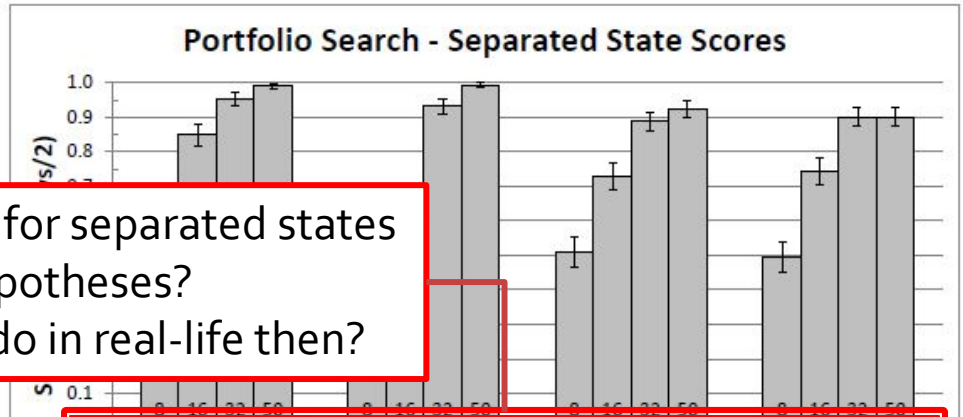
# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.492	0.454	0.482	0.373
16	0.539	0.439	0.456	0.409
32	0.526	0.518	0.507	0.427
50	0.509	0.44	0.51	0.419



# Units	UCT None vs AB None	UCT None vs AB NOK-AV	UCT NOK-AV vs AB None	UCT NOK-AV vs AB NOK-AV
8	0.556	0.472	0.488	0.405
16	0.605	0.510	0.586	0.519
32	0.689	0.666	0.676	0.585
50	0.782	0.674	0.756	0.709



# Units	AB None	AB NOK-AV	UCT None	UCT NOK-AV
8	0.541	0.290	0.408	0.431
16	0.536	0.364	0.473	0.482
32	0.496	0.460	0.472	0.469
50	0.514	0.495	0.511	0.501



# Units	AB None	AB NOK	UCT None	UCT NOK
8	0.319	0.253	0.511	0.493
16	0.848	0.608	0.729	0.742
32	0.952	0.932	0.888	0.900
50	0.990	0.992	0.924	0.900

PGS much better for separated states
What are your hypotheses?
What would you do in real-life then?

Limitation of PGS

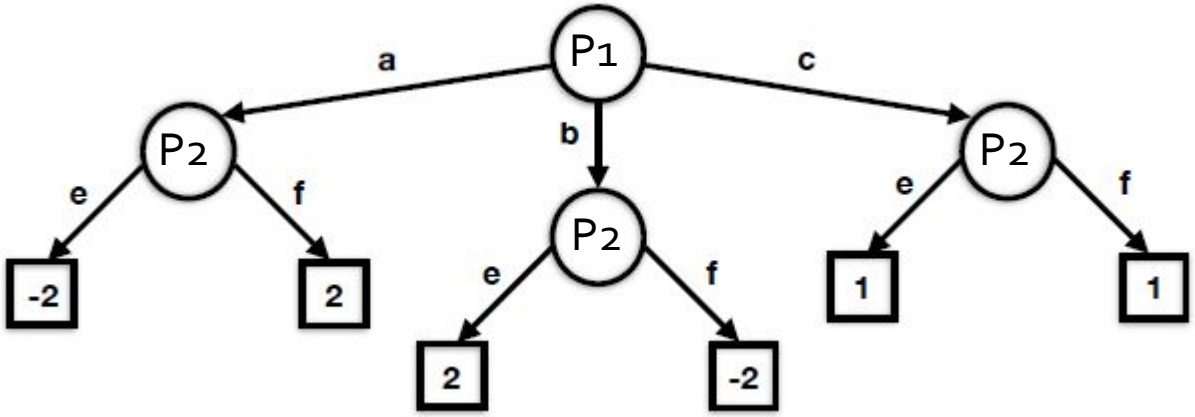
Possibility of non-convergence.

Churchill, D., & Buro, M. (2013, August). Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1-8). IEEE.

Portfolio Greedy Search

Limitations

PGS is having a problem with convergence to min-max.



At first, P2 default action was selected as f. P1 is therefore playing a first. P2 `Improves` by playing e. In which case P1 `Improves` by playing b. In which case, P2 `Improves` by playing f. Which means… PGS will never converge to play c, which would be the optimal solution MinMax would find.

Nested Greedy Search

Using PGS in hierarchy

Moraes, R., Mariño, J., & Lelis, L. (2018, September). Nested-greedy search for adversarial real-time games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 14, No. 1).

Nested Greedy Search

a.k.a. NGS

Algorithm sketch

Input: in round r, your units U_i^r, enemy units U_{-i}^r, portfolio of scripts P, state evaluation function Ψ.

Goal: assign some script $s \in P$ to each of your unit $u \in U_i^r$.

[A] Use some default script and assign it to all enemy units.
[B] Use some default script and assign it to all your units.
[C] As long as you have the budget, try to improve your assignment

- [C1] For each possible change in the assignment
- [C2] Check if the enemy cannot reassign

Nested Greedy Search

a.k.a. NGS

Algorithm 3 Nested-Greedy Search (NGS)

Require: state s, ready units $\mathcal{U}_i^r = \{u_i^1, \cdots, u_i^{n_i}\}$ and $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, evaluation function Ψ, and time limit t.

Ensure: action a for player i's units.

1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$
3: $a_i \leftarrow \{\bar{\sigma}_i(u_i^1), \cdots, \bar{\sigma}_i(u_i^{n_i})\}$
4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_{-i}^1), \cdots, \bar{\sigma}_{-i}(u_{-i}^{n_{-i}})\}$
5: **while** time elapsed is not larger than t **do**
6: **for** $k \leftarrow 1$ to $|\mathcal{U}_i^r|$ **do**
7: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
8: $a_i' \leftarrow a_i$; $a_i'[k] \leftarrow \bar{\sigma}(s, u_i^k)$
9: **if** $\mathrm{GS}(s, a_{-i}, a_i', \Psi|) > \mathrm{GS}(s, a_{-i}, a_i, \Psi)$ **then**
10: $a_i \leftarrow a_i'$
11: **if** time elapsed is larger than t **then**
12: **return** a_i
13: **return** a_i

Algorithm 4 GREEDY SEARCH (GS)

Require: state s, ready units $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, action vector a_{-i} for player $-i$, action vector a_i for player i, and evaluation function Ψ.

Ensure: the best action value by player $-i$ in response a action a_i.

$\mathcal{B} \leftarrow \infty$
for $k \leftarrow 1$ to $|\mathcal{U}_{-i}^r|$ **do**
  for each $\bar{\sigma} \in \mathcal{P}$ **do**
    $a_{-i}' \leftarrow a_{-i}$; $a_{-i}'[k] \leftarrow \bar{\sigma}(s, u_{-i}^k)$
    if $\Psi(\mathcal{T}(s, a_i, a_{-i}')) < \mathcal{B}$ **then**
      $a_{-i} \leftarrow a_{-i}'$; $\mathcal{B} \leftarrow \Psi(\mathcal{T}(s, a_i, a_{-i}'))$
return $\mathcal{B}$

`NestedGreedySearch` sections:

Row 1,3 <-> [A] … default me
Row 2,4 <-> [B] … default enemy
Row 5 <-> [C] … while we have time
Rows 6-10 <-> [C1] … check on reassignments
Row 9 <-> [C2] … should we reassign?

BTW, notation could not have been more confusing here…

Nested Greedy Search

a.k.a. NGS

Algorithm 3 Nested-Greedy Search (NGS)

Require: state s, ready units $\mathcal{U}_i^r = \{u_i^1, \cdots, u_i^{n_i}\}$ and $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, evaluation function Ψ, and time limit t.

Ensure: action a for player i's units.

1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$
3: $a_i \leftarrow \{\bar{\sigma}_i(u_i^1), \cdots, \bar{\sigma}_i(u_i^{n_i})\}$
4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_{-i}^1), \cdots, \bar{\sigma}_{-i}(u_{-i}^{n_{-i}})\}$
5: **while** time elapsed is not larger than t **do**
6: **for** $k \leftarrow 1$ to $|\mathcal{U}_i^r|$ **do**
7: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
8: $a_i' \leftarrow a_i$; $a_i'[k] \leftarrow \bar{\sigma}(s, u_i^k)$
9: **if** $\mathrm{GS}(s, a_{-i}, a_i', \Psi|) > \mathrm{GS}(s, a_{-i}, a_i, \Psi)$ **then**
10: $a_i \leftarrow a_i'$
11: **if** time elapsed is larger than t **then**
12: **return** a_i
13: **return** a_i

Algorithm 4 GREEDY SEARCH (GS)

Require: state s, ready units $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, action vector a_{-i} for player $-i$, action vector a_i for player i, and evaluation function Ψ.

Ensure: the best action value by player $-i$ in response a action a_i.

1: $\mathcal{B} \leftarrow \infty$
2: **for** $k \leftarrow 1$ to $|\mathcal{U}_{-i}^r|$ **do**
3: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
4: $a_{-i}' \leftarrow a_{-i}$; $a_{-i}'[k] \leftarrow \bar{\sigma}(s, u_{-i}^k)$
5: **if** $\Psi(\mathcal{T}(s, a_i, a_{-i}')) < \mathcal{B}$ **then**
6: $a_{-i} \leftarrow a_{-i}'$; $\mathcal{B} \leftarrow \Psi(\mathcal{T}(s, a_i, a_{-i}'))$
7: **return** $\mathcal{B}$

`GreedySearch` sections:

2: for each enemy unit u_{-i}^k
3: for each script $\bar{\sigma}$ in portfolio P
4: temporarily assign $\bar{\sigma}$ to be used by u_{-i}^k
5: if yields better result
6: permanently assign $\bar{\sigma}$ to be used by u_{-i}^k

Nested Greedy Search

a.k.a. NGS

Algorithm 3 Nested-Greedy Search (NGS)

Require: state s, ready units $\mathcal{U}_i^r = \{u_i^1, \cdots, u_i^{n_i}\}$ and $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, evaluation function Ψ, and time limit t.

Ensure: action a for player i's units.

$\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$
$\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$
$a_i \leftarrow \{\bar{\sigma}_i(u_i^1), \cdots, \bar{\sigma}_i(u_i^{n_i})\}$
$a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_{-i}^1), \cdots, \bar{\sigma}_{-i}(u_{-i}^{n_{-i}})\}$
while time elapsed is not larger than t **do**
  for $k \leftarrow 1$ to $|\mathcal{U}_i^r|$ **do**
    for each $\bar{\sigma} \in \mathcal{P}$ **do**
      $a_i' \leftarrow a_i$; $a_i'[k] \leftarrow \bar{\sigma}(s, u_i^k)$
      if $\mathrm{GS}(s, a_{-i}, a_i', \Psi|) > \mathrm{GS}(s, a_{-i}, a_i, \Psi)$ **then**
        $a_i \leftarrow a_i'$
    if time elapsed is larger than t **then**
      return a_i
return a_i

Algorithm 4 GREEDY SEARCH (GS)

Require: state s, ready units $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, action vector a_{-i} for player $-i$, action vector a_i for player i, and evaluation function Ψ.

Ensure: the best action value by player $-i$ in response a action a_i.

$\mathcal{B} \leftarrow \infty$
for $k \leftarrow 1$ to $|\mathcal{U}_{-i}^r|$ **do**
  for each $\bar{\sigma} \in \mathcal{P}$ **do**
    $a_{-i}' \leftarrow a_{-i}$; $a_{-i}'[k] \leftarrow \bar{\sigma}(s, u_{-i}^k)$
    if $\Psi(\mathcal{T}(s, a_i, a_{-i}')) < \mathcal{B}$ **then**
      $a_{-i} \leftarrow a_{-i}'$; $\mathcal{B} \leftarrow \Psi(\mathcal{T}(s, a_i, a_{-i}'))$
return $\mathcal{B}$

`GreedySearch` notes:
We're not trying all possible assignments $|P|^n$
only greedy-style $|P|.n$.

Nested Greedy Search

a.k.a. NGS

Algorithm 3 Nested-Greedy Search (NGS)

Require: state s, ready units $\mathcal{U}_i^r = \{u_i^1, \cdots, u_i^{n_i}\}$ and $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, evaluation function Ψ, and time limit t.

Ensure: action a for player i's units.

1: $\bar{\sigma}_i \leftarrow$ choose a script from $\mathcal{P}$
2: $\bar{\sigma}_{-i} \leftarrow$ choose a script from $\mathcal{P}$
3: $a_i \leftarrow \{\bar{\sigma}_i(u_i^1), \cdots, \bar{\sigma}_i(u_i^{n_i})\}$
4: $a_{-i} \leftarrow \{\bar{\sigma}_{-i}(u_{-i}^1), \cdots, \bar{\sigma}_{-i}(u_{-i}^{n_{-i}})\}$
5: **while** time elapsed is not larger than t **do**
6: **for** $k \leftarrow 1$ to $|\mathcal{U}_i^r|$ **do**
7: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
8: $a_i' \leftarrow a_i$; $a_i'[k] \leftarrow \bar{\sigma}(s, u_i^k)$
9: **if** $\mathrm{GS}(s, a_{-i}, a_i', \Psi|) > \mathrm{GS}(s, a_{-i}, a_i, \Psi)$ **then**
10: $a_i \leftarrow a_i'$
11: **if** time elapsed is larger than t **then**
12: **return** a_i
13: **return** a_i

Algorithm 4 GREEDY SEARCH (GS)

Require: state s, ready units $\mathcal{U}_{-i}^r = \{u_{-i}^1, \cdots, u_{-i}^{n_{-i}}\}$ in s, set of scripts $\mathcal{P}$, action vector a_{-i} for player $-i$, action vector a_i for player i, and evaluation function Ψ.

Ensure: the best action value by player $-i$ in response a action a_i.

1: $\mathcal{B} \leftarrow \infty$
2: **for** $k \leftarrow 1$ to $|\mathcal{U}_{-i}^r|$ **do**
3: **for** each $\bar{\sigma} \in \mathcal{P}$ **do**
4: $a_{-i}' \leftarrow a_{-i}$; $a_{-i}'[k] \leftarrow \bar{\sigma}(s, u_{-i}^k)$
5: **if** $\Psi(\mathcal{T}(s, a_i, a_{-i}')) < \mathcal{B}$ **then**
6: $a_{-i} \leftarrow a_{-i}'$; $\mathcal{B} \leftarrow \Psi(\mathcal{T}(s, a_i, a_{-i}'))$
7: **return** $\mathcal{B}$

`NestedGreedySearch` notes:
Having GS on the second ply of a game tree effectively means, that we move non-convergence down by one ply.

`GreedySearch` improvement:
We could do another GS call on row 5 (another nesting) instead of calling state evaluation function Ψ.

Experiments and Results

Comparing NGS to other searches.

Moraes, R., Mariño, J., & Lelis, L. (2018, September). Nested-greedy search for adversarial real-time games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 14, No. 1).

Nested Greedy Search

Results

Scenario setup

μRTS; maps size $x \times x$ where $x \in \{8, 12, 16, 24\}$

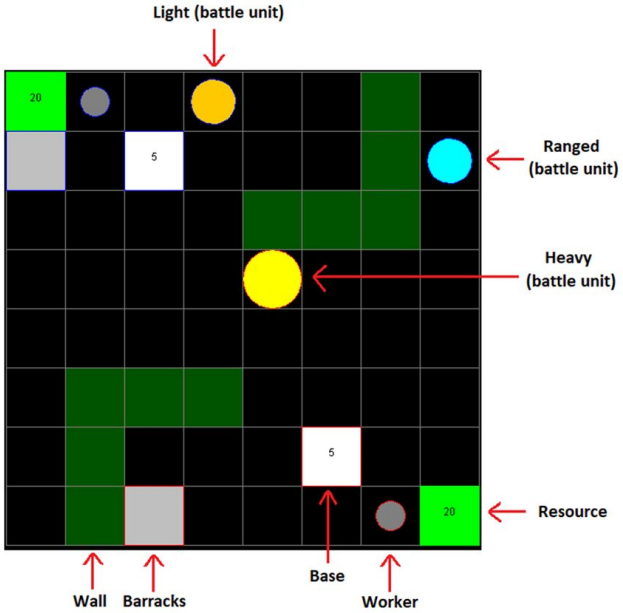
Max game cycles 3000, 4000, 4000, 5000 to corresp. map size.

Players are swapping starting locations based on odd/even match number.

Algorithm details

Ψ chosen as playout for 100 game cycles, Then taken the same as μRTS default eval. Function scoring resources, units, etc.

Portfolio: Worker rush, NOKAV, Kiter



https://github.com/santiontanon/microrts

Nested Greedy Search

Results

Map 8×8	PGS	PGS_R	NGS	Avg.
PGS	-	75.0	43.8	59.4
PGS_R	25.0	-	12.5	**18.8**
NGS	56.2	87.5	-	71.9

Map 12×12	PGS	PGS_R	NGS	Avg.
PGS	-	87.5	46.2	66.9
PGS_R	12.5	-	3.8	**8.1**
NGS	53.8	96.2	-	75.0

Map 16×16	PGS	PGS_R	NGS	Avg.
PGS	-	78.7	32.5	55.6
PGS_R	21.3	-	12.5	**16.9**
NGS	67.5	87.5	-	77.5

Map 24×24	PGS	PGS_R	NGS	Avg.
PGS	-	73.8	47.5	60.6
PGS_R	26.3	-	3	**18.8**
NGS	52.5	88.8	-	70.6

NGS always better than PGS, but not much!

No time limit set; PGS: $I = 1; R = 0$; PGS_R: $I = 1; R = 1$

Nested Greedy Search

Results

Map 8 × 8

	PS	AHT	STT	NAV	SCV	PGS_R	PGS	NGS	Avg.
PS	-	0.0	5.0	0.0	0.0	0.0	0.0	0.0	0.7
AHT	100.0	-	25.0	12.5	2.5	57.5	33.8	25.0	36.6
STT	95.0	75.0	-	46.3	21.3	92.5	91.3	26.3	63.9
NAV	100.0	87.5	53.8	-	35.0	97.5	96.3	20.0	70.0
SCV	100.0	97.5	78.8	65.0	-	92.5	68.8	12.5	73.6
PGS_R	100.0	42.5	7.5	2.5	7.5	-	63.8	10.0	33.4
PGS	100.0	66.3	8.8	3.8	31.3	36.3	-	25.0	38.8
NGS	100.0	75.0	73.8	80.0	87.5	90.0	75.0	-	83.0

Map 12 × 12

	PS	AHT	STT	NAV	SCV	PGS_R	PGS	NGS	Avg.
PS	-	0.0	2.5	70.0	40.0	7.5	0.0	0.0	17.1
AHT	100.0	-	23.8	10.0	81.3	78.8	46.3	50.0	55.7
STT	97.5	76.3	-	45.0	100.0	100.0	100.0	53.8	81.8
NAV	30.0	90.0	55.0	-	57.5	82.5	72.5	60.0	63.9
SCV	60.0	18.8	0.0	42.5	-	5.0	0.0	0.0	18.0
PGS_R	92.5	21.3	0.0	17.5	95.0	-	21.3	0.0	35.4
PGS	100.0	53.8	0.0	27.5	100.0	78.8	-	17.5	53.9
NGS	100.0	50.0	46.3	40.0	100.0	100.0	82.5	-	74.1

Map 16 × 16

	PS	AHT	STT	NAV	SCV	PGS_R	PGS	NGS	Avg.
PS	-	0.0	65.0	100.0	0.0	12.5	0.0	0.0	25.4
AHT	100.0	-	45.0	0.0	60.0	75.0	70.0	12.5	51.8
STT	35.0	55.0	-	76.3	48.8	65.0	65.0	31.3	53.8
NAV	0.0	100.0	23.8	-	25.0	82.5	53.8	17.5	43.2
SCV	100.0	40.0	51.3	75.0	-	77.5	62.5	0.0	58.0
PGS_R	87.5	25.0	35.0	17.5	22.5	-	17.5	0.0	29.3
PGS	100.0	30.0	35.0	46.3	37.5	82.5	-	5.0	48.0
NGS	100.0	87.5	68.8	82.5	100.0	100.0	95.0	-	90.5

Map 24 × 24

	PS	AHT	STT	NAV	SCV	PGS_R	PGS	NGS	Avg.
PS	-	27.5	52.5	100.0	100.0	85.0	87.5	92.5	77.9
AHT	72.5	-	0.0	5.0	16.3	62.5	50.0	0.0	29.5
STT	47.5	100.0	-	81.3	67.5	86.3	77.5	45.0	72.1
NAV	0.0	95.0	18.8	-	2.5	52.5	40.0	30.0	34.1
SCV	0.0	83.8	32.5	97.5	-	55.0	43.8	50.0	51.8
PGS_R	15.0	37.5	13.8	47.5	45.0	-	20.0	0.0	25.5
PGS	12.5	50.0	22.5	60.0	56.3	80.0	-	8.8	41.4
NGS	7.5	100.0	55.0	70.0	50.0	100.0	91.3	-	67.7

NGS was the usual winner.

AHT – Adversarial HTN, NAV – Naive MCTS, PS – Puppet search, STT – Strategy tactics

… all adversaries are search based.

PGS / NGS

Final thoughts

Is it all but smoke and mirrors?

Using low-level actions in searches are probably overwhelming to script-based approaches.

Should we really be comparing stuff across "time limits"? How about using "iterations" instead?

Where is the breaking point?

What if we put scripts into MCTS instead of low-level actions (psst, we actually did and it outperformed PGS clearly)?

Or otherwise used scripts to prune MCTS so we get to the "breaking point"?

=> Any of this could effectively become your master thesis topic!

That’s it for today!

Faculty of Mathematics and Physics
Charles University
7.12.2022

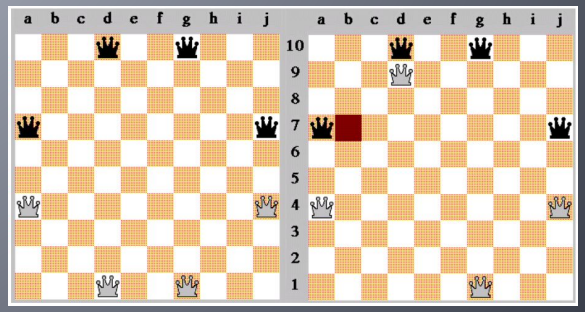
GAME DEVELOPMENT @CUNI.CZ

matfyz

NAIL122 AI for Games

MCTS improvements – part 2

Jakub Gemrot, Peter Guba



MCTS – Reminder

MCTS
High-level description

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

MCTS iteration

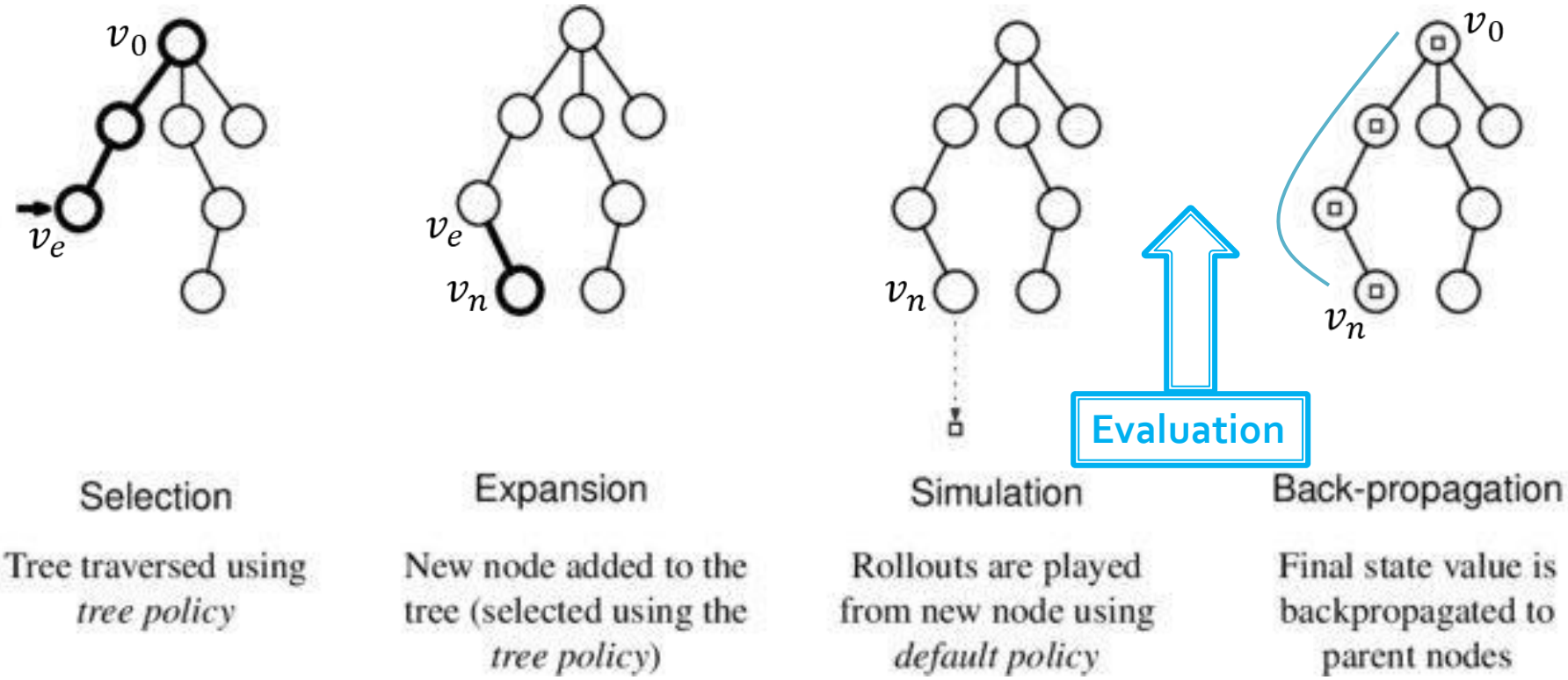


Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS domain of applicability

MCTS

Games

Games where MCTS is applicable

- Discrete

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum
- 2-player

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum
- 2-player
- Sequential

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum
- 2-player
- Sequential
- Perfect Information

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum
- 2-player
- Sequential
- Perfect Information
- Deterministic

MCTS

Games

Games where MCTS is applicable

Next lecture

- Discrete
- Zero-sum
- 2-player
- Sequential
- Perfect Information
- Deterministic

MCTS

Games

Games where MCTS is applicable

- Discrete
- Zero-sum
- 2-player
- Sequential
- Perfect Information
- Deterministic

MCTS in non-zero-sum games

MCTS in non-zero-sum games

What if one's win isn't the other's loss?

We only need the game to be zero-sum so that we can simplify how we deal with rewards.

Solution: Evaluate states separately for both players, backpropagate both their rewards and switch based on which rewards you make selection in the first step.

MCTS

Games

Games where MCTS is applicable

- Discrete
- ~~Zero-sum~~
- 2-player
- Sequential
- Perfect Information
- Deterministic

MCTS in other than 2-player games

Source paper:
Sturtevant, N. (2008). An analysis of UCT in multi-player games. *ICGA Journal*, *31*(4), 195-208.

MCTS in other than 2-player games

What if we can't just alternate between the players?

Single-player games

Adapting MCTS to single-player games that otherwise satisfy all the other conditions for games where MCTS can be applied is probably quite useless, but theoretically, all that needs to be done is to always maximise the players reward instead of alternating between minimising and maximising.

Multi-player games

Similarly to non-zero-sum games, this means we can no longer infer the rewards of other players from those of one player, so we backpropagate rewards for all players.

MCTS

Games

Games where MCTS is applicable

- Discrete
- ~~Zero-sum~~
- ~~2-player~~
- Sequential
- Perfect Information
- Deterministic

MCTS in simultaneous-moves games

Source paper:
Lanctot, M., Wittlinger, C., Winands, M. H., & Den Teuling, N. G. (2013, November). Monte Carlo tree search for simultaneous move games: A case study in the game of Tron. In *BNAIC* (Vol. 2013, pp. 104-111).

MCTS in simultaneous-moves games

How to model simultaneous moves?

First idea – just use normal MCTS

This means that one player will know the other's moves, which will most likely completely mess with planning.

Example – if this were applied to rock paper scissors, the second player could always choose the move to win.

MCTS in simultaneous-moves games

How to model simultaneous moves?

Better idea – change the datastructure

Basically, the difference to normal MCTS is that, instead of each node having a one-dimensional list of children (which can be represented as a tree), it should have a two-dimensional one.

MCTS in simultaneous-moves games

How to model simultaneous moves?

Better idea – change the datastructure

Basically, the difference to normal MCTS is that, instead of each node having a one-dimensional list of children (which can be represented as a tree), it should have a two-dimensional one.

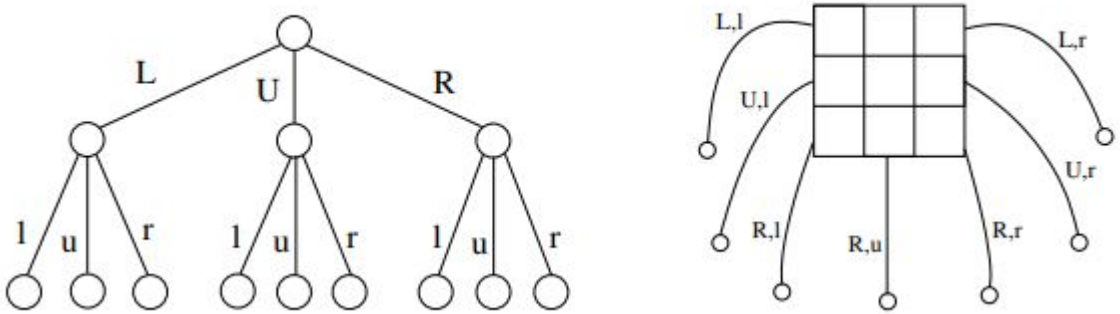


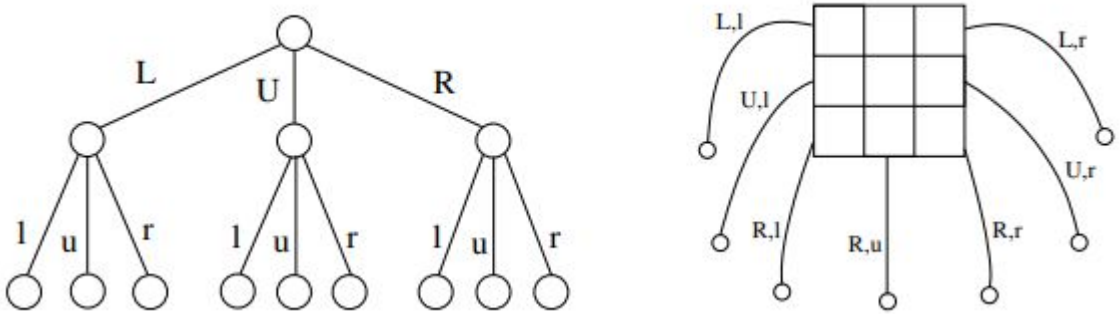
Image taken from: Schaeffer, M. S. N. S. J., & Shafiei, N. (2009). Comparing UCT versus CFR in simultaneous games. In *IJCAI Workshop on General Game Playing*.

MCTS in simultaneous-moves games

How to model simultaneous moves?

Better idea – change the datastructure

Basically, the difference to normal MCTS is that, instead of each node having a one-dimensional list of children (which can be represented as a tree), it should have a two-dimensional one.



When we apply this change to the structure we are building, we can perform selection by computing the UCB values of actions for both players.

Image taken from: Schaeffer, M. S. N. S. J., & Shafiei, N. (2009). Comparing UCT versus CFR in simultaneous games. In *IJCAI Workshop on General Game Playing*.

MCTS in simultaneous-moves games

How to model simultaneous moves?

Better idea – change the datastructure

This algorithm will not necessarily converge to a Nash Equilibrium however which means it can be exploited.

The solution to this is using a different tree policy than standard UCB – these are a bit complex however, so we won't go over them here.

MCTS

Games

Games where MCTS is applicable

- Discrete
- ~~Zero-sum~~
- ~~2-player~~
- ~~Sequential~~
- Perfect Information
- Deterministic

MCTS in imperfect information games

Source paper:
Teytaud, O., & Flory, S. (2011, April). Upper confidence trees with short term partial information. In *European Conference on the Applications of Evolutionary Computation* (pp. 153-162). Springer, Berlin, Heidelberg.

MCTS in imperfect information games

How to treat unobservable information?

Here we will allow ourselves a little simplification – we will only consider games for which there exists an integer n, such that that there is no sequence of n steps in the game which aren't fully observable.

Example of such a game – Texas Hold 'em Poker. After all the five cards that were dealt a revealed, the players must reveal their cards as well, making everything fully observable. (Caveat – if only all players except one folded, the remaining player can choose not to reveal their cards, so technically, this isn't quite true.)

Counterexample – UNO, Canasta and many other card games as there is no limit on how long players can keep their cards hidden.

MCTS in imperfect information games

How to treat unobservable information?

If we restrict ourselves to the games mentioned in the previous slide, we can actually transform them to games with simultaneous moves.

We treat every sequence of unobservable states as a single state and pick a strategy for that sequence of states instead of just a single move. This strategy is picked simultaneously for the two players.

We can therefore apply the approach discussed in the previous section.

MCTS

Games

Games where MCTS is applicable

- Discrete
- ~~Zero-sum~~
- ~~2-player~~
- ~~Sequential~~
- ~~Perfect Information~~
- Deterministic

MCTS in stochastic games

Source paper:
Bjarnason, R., Fern, A., & Tadepalli, P. (2009, October). Lower bounding Klondike solitaire with Monte-Carlo planning. In *Nineteenth international conference on automated planning and scheduling*.

MCTS in stochastic games

How to take chance into accout?

First idea – take all possibilities into account

For example, randomness in the game comes from throwing a six-sided die, you could treat each outcome as a different move in the search tree (you would then select from among these moves randomly instead of using some sophisticated tree policy).

This approach falls apart when there is too much randomness. For example, in a game of Klondike Solitaire, this would mean trying all different configurations of the hidden cards.

MCTS in stochastic games

How to take chance into accout?

Improvement – take only some possibilities into account

You could limit the branching factor in the random nodes.

In experiments done on Klondike Solitaire, this didn't seem to improve the performance of UCT. It did improve its speed however, so it should be able to explore more options in the same amount of time, making it more useful in practical applications.

MCTS in stochastic games

How to take chance into accout?

A different approach – building multiple trees

Run MCTS multiple times, each on a different fully determinized game, then average the results.

Performed better than either of the two previous approaches on Klondike Solitaire.

MCTS in stochastic games

How to take chance into accout?

Randomness and partial information

The difference between the case discussed in this section and in the previous one is that the hidden information considered here is provided by the environment hence one does not need to model an opponent's decision making over it.

Would the approach discussed here be applicable to partial-information games?

It is possible, though there don't seem to be any papers where this was tried.

MCTS

Games

Games where MCTS is applicable

- Discrete
- ~~Zero-sum~~
- ~~2-player~~
- ~~Sequential~~
- ~~Perfect Information~~
- ~~Deterministic~~

Some other adjustments

Transposition tables and DAG

Used by, e.g.:

Kozelek, T. (2009). Methods of MCTS and the game Arimaa. Diploma thesis, Charles University, Prague, Czech Republic.

Méhat, J., & Cazenave, T. (2010). Combining UCT and nested Monte Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 271-277.

Thorougly discussed by:

Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS is building a tree, games can be DAG

When game is known to form DAG often, MCTS will not share information between nodes representing the same game states.

The idea of transposition tables is to track such a nodes, e.g., by maintaining a dictionary using a node state hash as a key, and then share state values between such a nodes.

Which leads to an idea to maintain DAG instead of a tree.

Where to store MCTS data?
How to treat selection?
How to treat backpropagation?

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Storing data

Two DAGs, same game same MCTS iterations, different outcome.

In the right tree, mean can be reconstructed (no information is lost).

$$\mu = \frac{4 \times 0.75 + 2 \times 0.5}{4 + 2} = 0.67$$

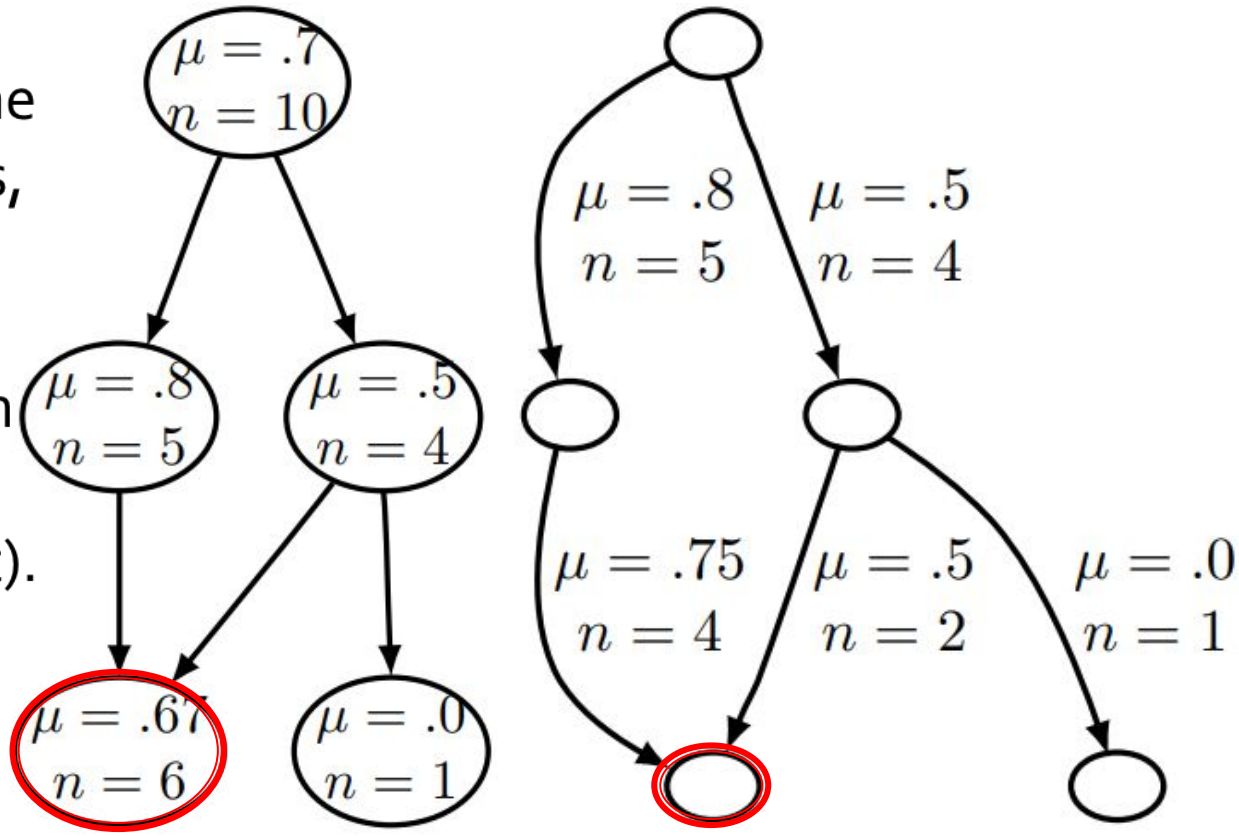


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Storing data

The difference lies in the selection phase.

Storing data with nodes skews UCB selection towards exploit.

We can take different path from the red node reaching new subgames!

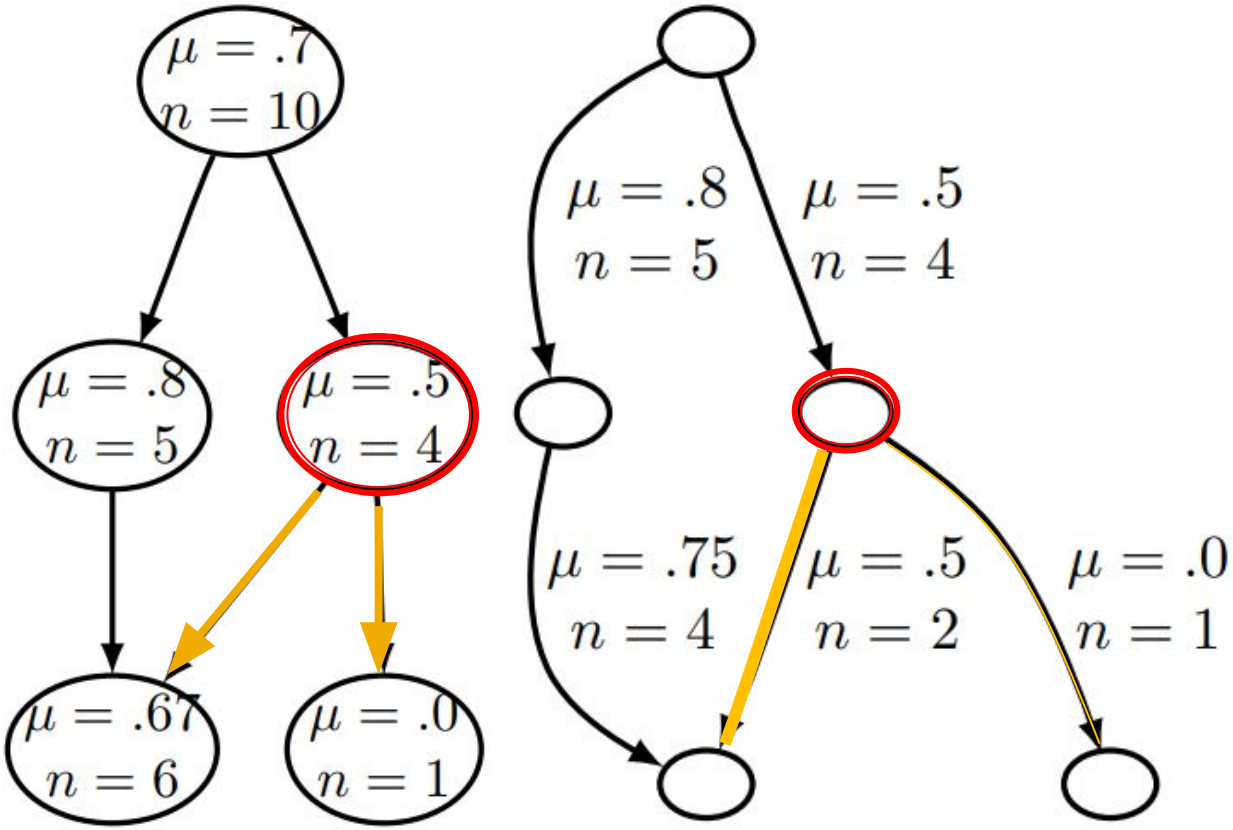


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

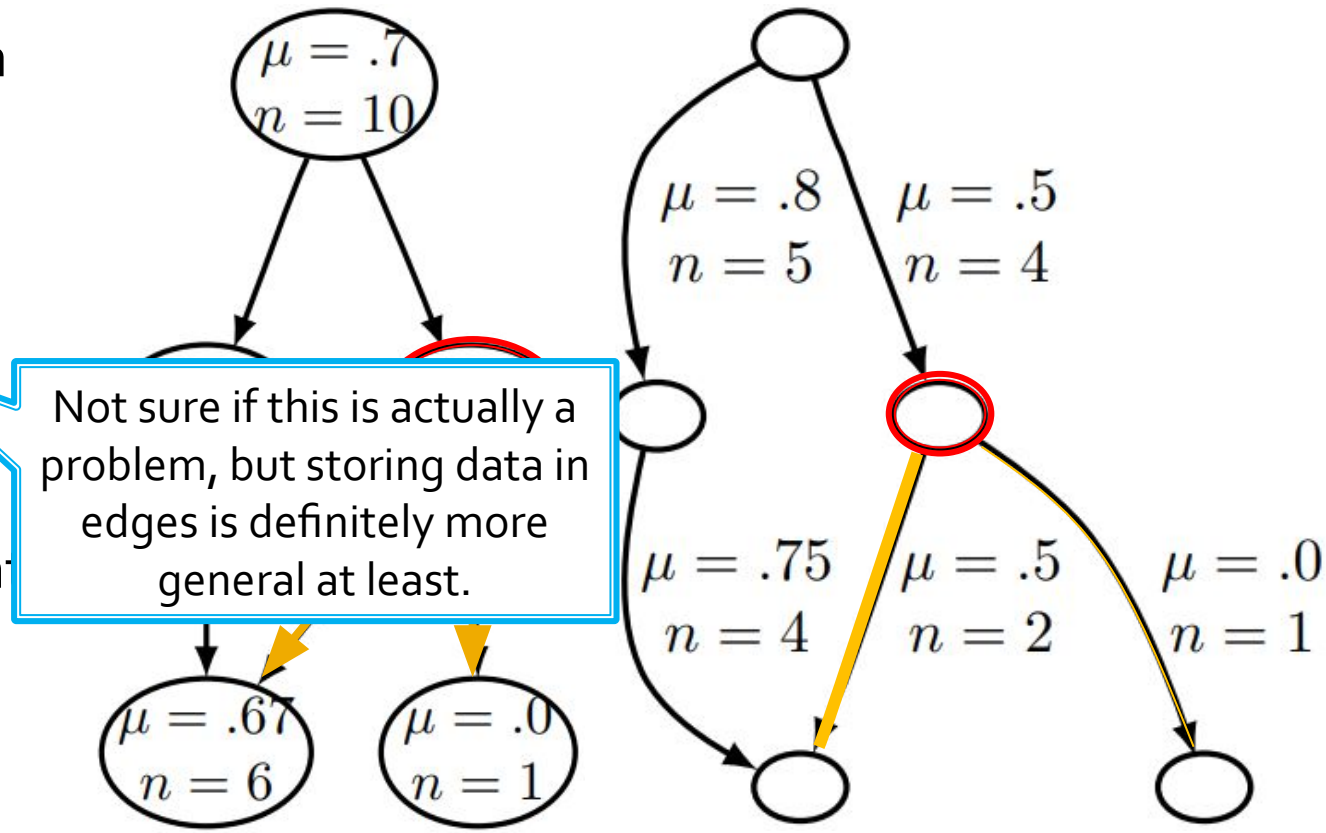
How to work with graphs?

MCTS with DAG – Storing data

The difference lies in the selection phase.

Storing data with nodes skews UCB selection towards exploit.

We can take different path from the red node reaching new subgames!



Not sure if this is actually a problem, but storing data in edges is definitely more general at least.

Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Backpropagation

We're storing data with edges, all good.

How about the backpropagation then?

Should we backpropagate:

- along all paths to the root?
  a.k.a. *update-all*
- only along the descent path?
  a.k.a. *update-descent*

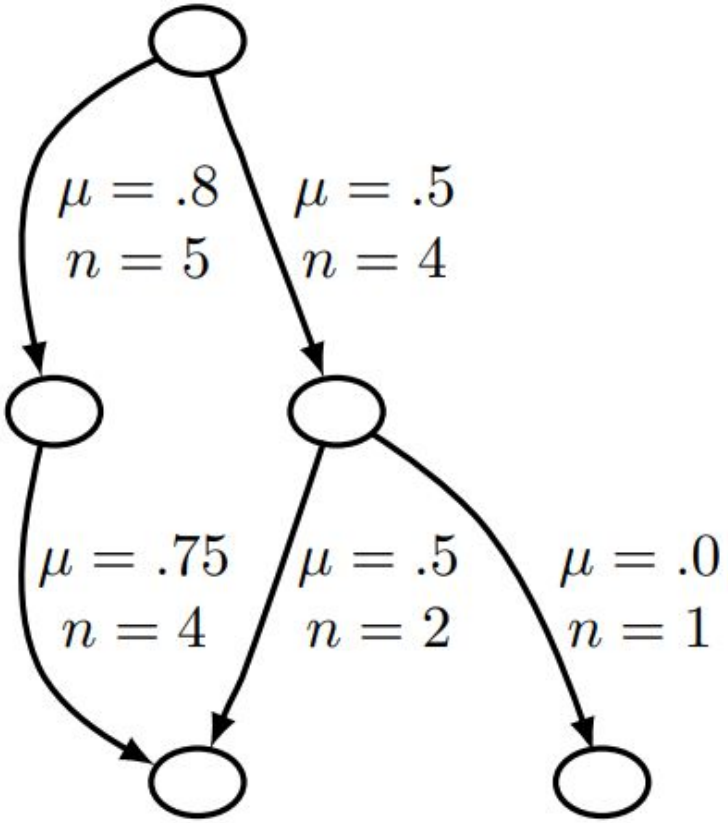


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Backpropagation

We're storing data with edges, all good.

How about the backpropagation then?

Should we backpropagate:

- along all paths to the root?
  a.k.a. *update-all*
- only along the descent path?
  a.k.a. *update-descent*

Tempting, as it would propagate information from the playout through all branches leading to the expanded node.

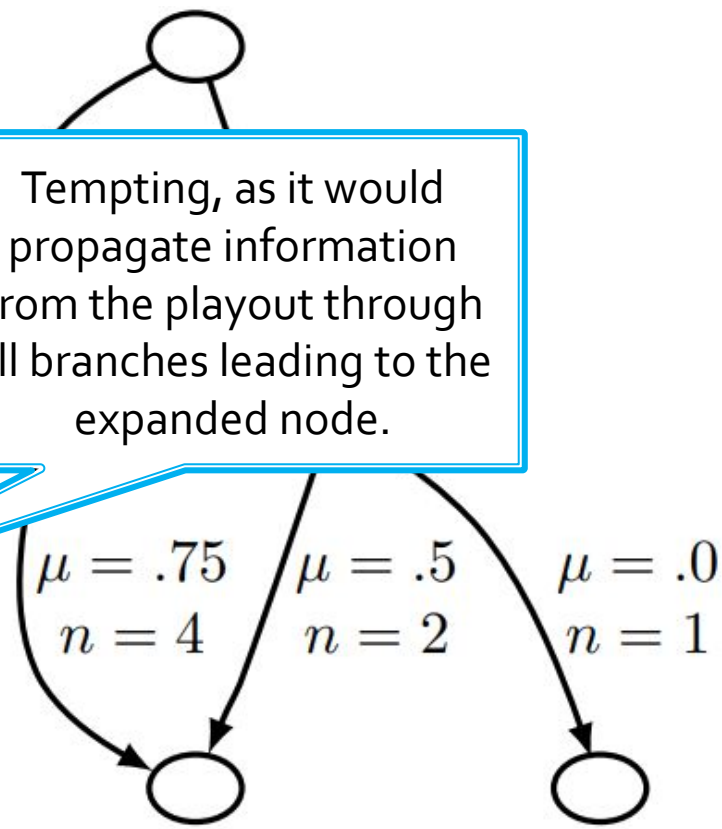


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Backpropagation w/ update-all

Let's consider the following game after initial 4 iterations where we use update-all paths to the root strategy.

Bottom edges ware taken 1x, 1x, 2x but update-all result in seemingly taking 6 simulations from the root.

What happens next?

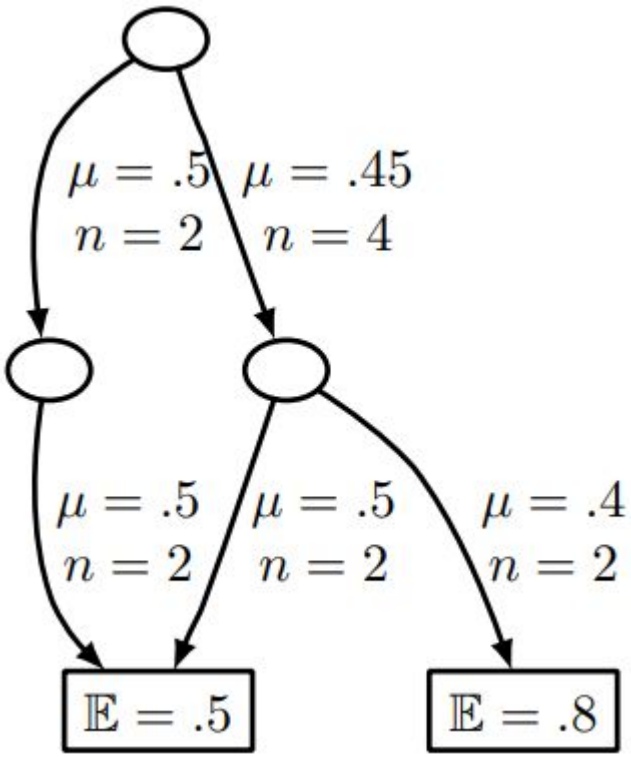


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Backpropagation w/ update-all

We will start selecting left edge from the root which will lead into ignoring the right one! Never arrive to optimum.

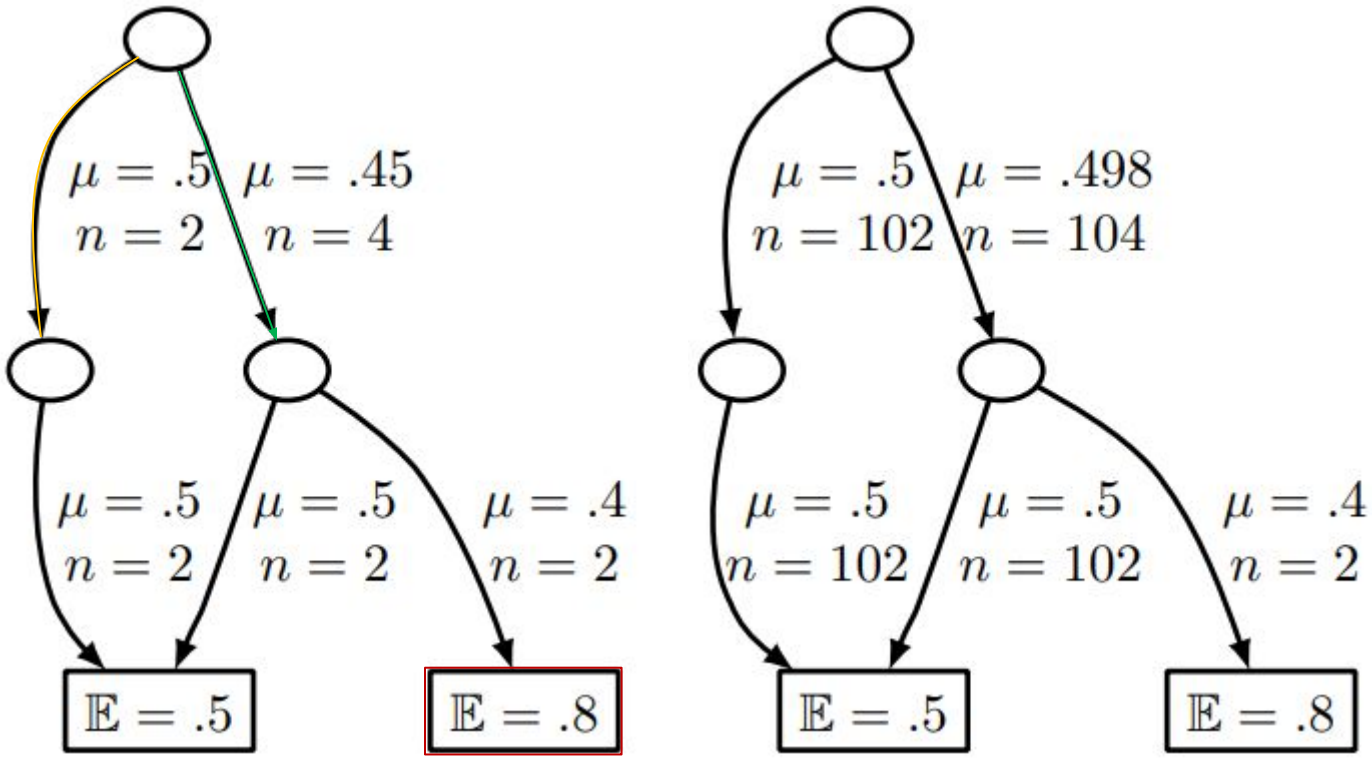


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Backpropagation w/ update-all

The problem lies in backprop. where we update-all paths to root.

Life's not that easy…

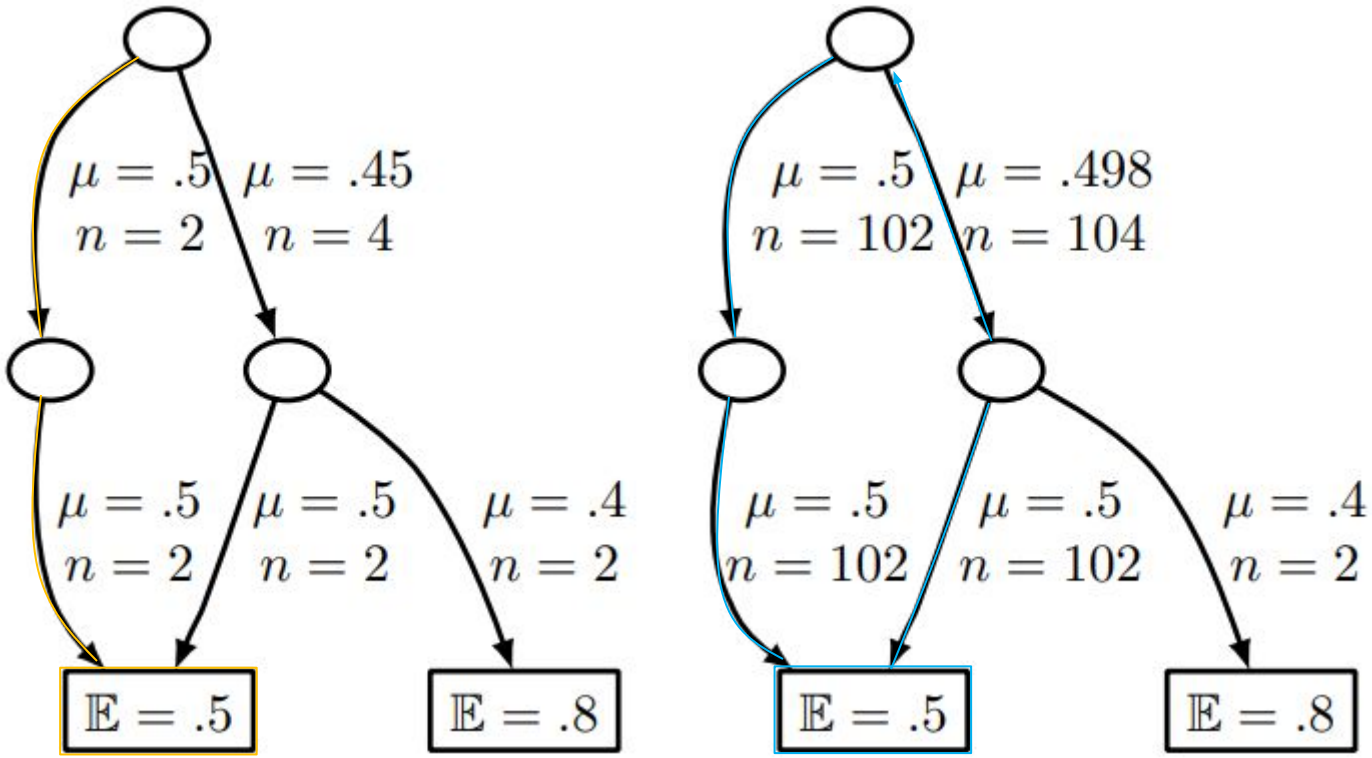


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Selection problem

How about selection now?

Constructing DAG, we might expand an edge (an action) that will lead to a node that is well understood already…

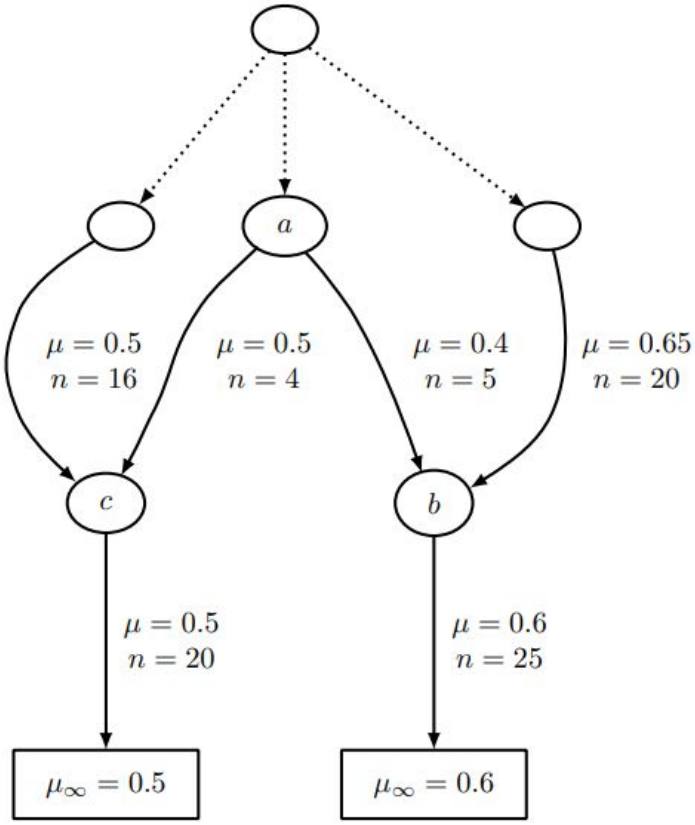


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Selection problem

Consider this tree; looking at the tree globally, we have enough information to conclude the node b is better.

However, facing local selection in node a, we do not know that if standard UCB is used.

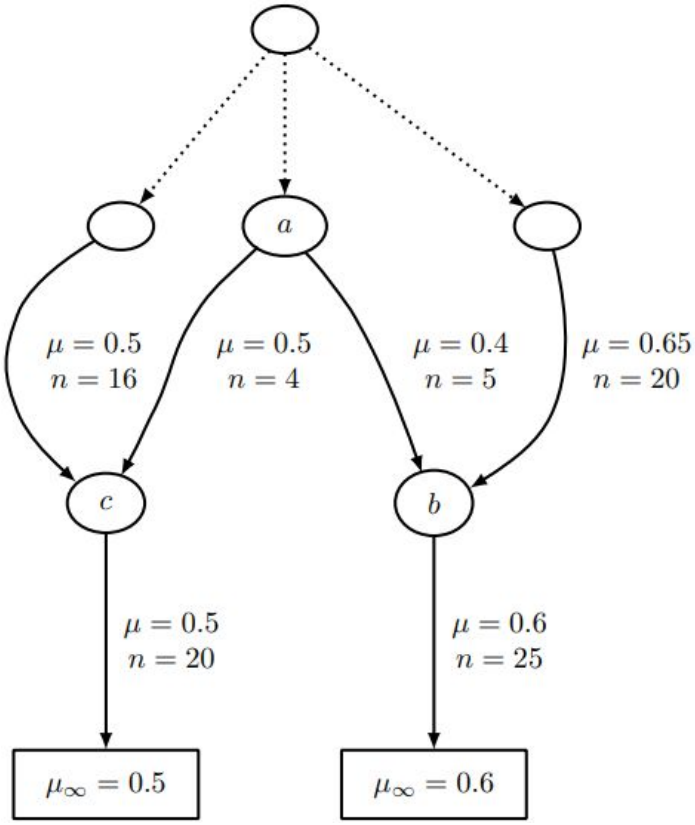


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Selection problem

Aggregated "look-ahead" MCTS data:

$c(e)$ … set of all outgoing edges of the destination node of edge e
$b(e)$ … set of all siblings + e
$\mu'(e)$ and $n'(e)$ … the mean reward and number of playouts measured in an edge before it gets any children

$$\mu_0(e) = \mu(e)$$

$$\mu_d(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$

$$n_0(e) = n(e)$$

$$n_d(e) = n'(e) + \sum_{f \in c(e)} n_{d-1}(f)$$

$$p_d(e) = \sum_{f \in b(e)} n_d(f)$$

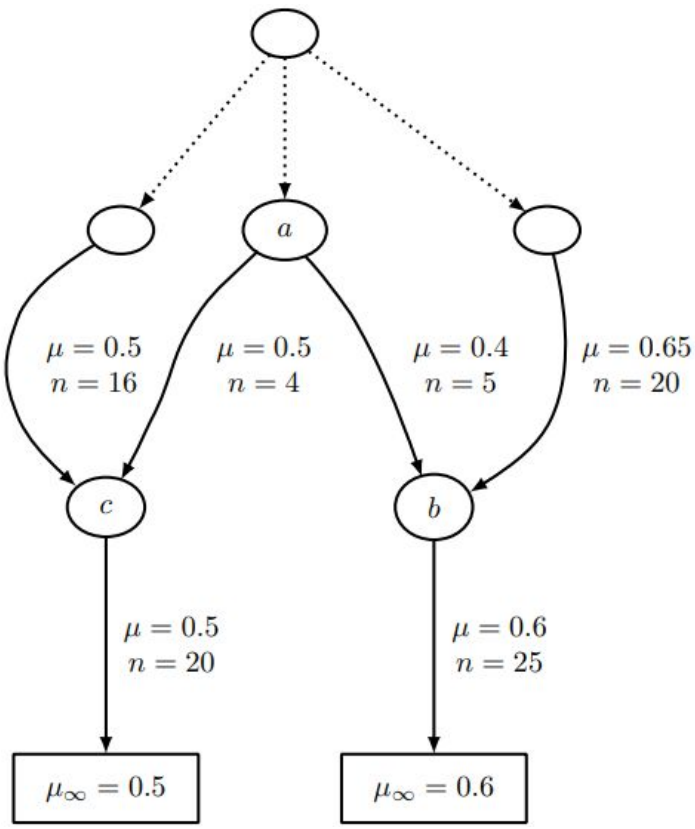


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Selection problem

Aggregated "look-ahead" MCTS data:

$c(e)$ … set of all outgoing edges of the destination node of edge e
$b(e)$ … set of all siblings + e
$\mu'(e)$ and $n'(e)$ … the mean reward and number of playouts measured in an edge before it gets any children

$$\mu_0(e) = \mu(e)$$

$$\mu_d(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$

$$n_0(e) = n(e)$$

$$n_d(e) = n'(e) + \sum_{f \in c(e)} n_{d-1}(f)$$

$$p_d(e) = \sum_{f \in b(e)} n_d(f)$$

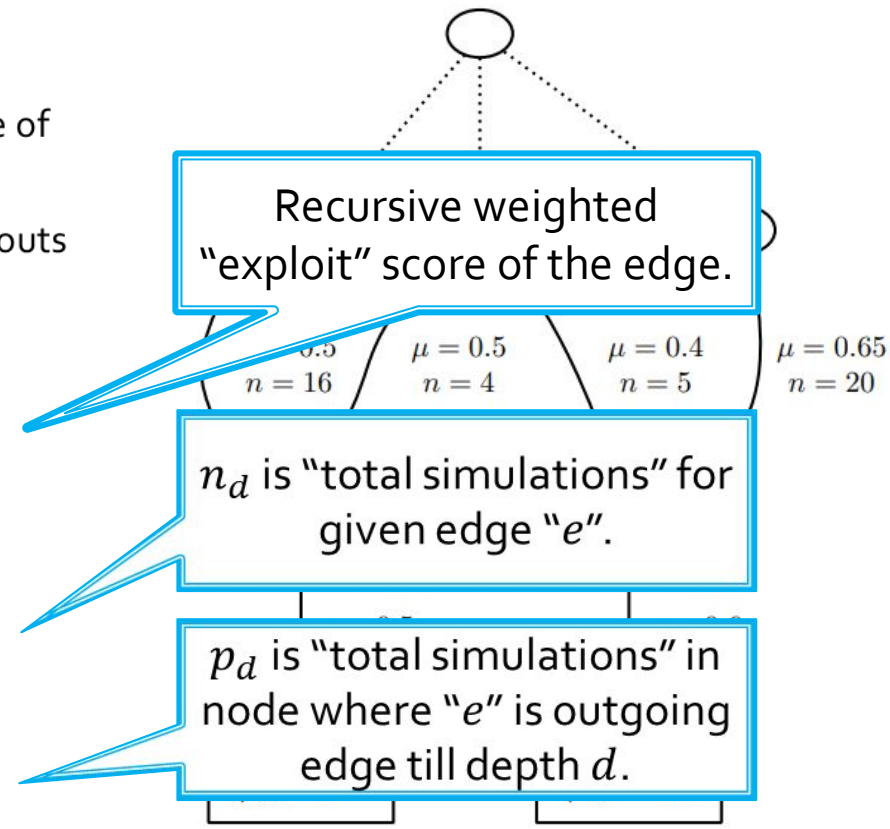


Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

MCTS with DAG – Selection problem

Aggregated "look-ahead" MCTS data:

$c(e)$ … set of all outgoing edges of the destination node of edge e
$b(e)$ … set of all siblings + e
$\mu'(e)$ and $n'(e)$ … the mean reward and number of playouts measured in an edge before it gets any children

$$\mu_0(e) = \mu(e)$$

$$\mu_d(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$

$$n_0(e) = n(e)$$

$$n_d(e) = n'(e) + \sum_{f \in c(e)} n_{d-1}(f)$$

$$p_d(e) = \sum_{f \in b(e)} n_d(f)$$

We can now define aggregated "look-ahead" UCB value:

$$\boldsymbol{u_{d_1,d_2,d_3}(e) = \mu_{d_1}(e) + c \times \sqrt{\frac{\ln p_{d_2}(e)}{n_{d_3}(e)}}}$$

For $(d_1, d_2, d_3) = (0,0,0)$ the formula is equal to plain UCB1.

Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Transposition tables / DAG

How to work with graphs?

UCD vs. UCT; UCT had c set to 0.4, which was found to be the best.

MCTS with DAG – Results on Go 6x6

d_1	d_2	d_3	$c = 0.2$	$c = 0.4$	$c = 0.6$
0	0	0	42.5%	52.0%	46.5%
1	0	0	40.5%	53.0%	48.0%
2	0	0	34.5%	37.0%	33.5%
0	0	1	47.5%	48.5%	53.0%
1	0	1	45.0%	48.0%	44.5%
2	0	1	36.0%	31.0%	31.0%
0	1	0	44.0%	53.0%	44.0%
1	1	0	45.5%	56.0%	51.0%
2	1	0	37.5%	33.0%	33.5%
0	1	1	40.5%	52.0%	49.0%
1	1	1	51.5%	48.5%	49.5%
2	1	1	39.0%	37.5%	31.5%

Best UCD results found.

Image adapted from: Saffidine, A., Cazenave, T., & Méhat, J. (2012). UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems*, *34*, 26-33.

Parallel MCTS

Source paper:
Chaslot, G. M. B., Winands, M. H., & Herik, H. J. V. D. (2008, September). Parallel monte-carlo tree search.
In *International Conference on Computers and Games* (pp. 60-71). Springer, Berlin, Heidelberg.

Parallel MCTS

How to properly parallelize our computations?

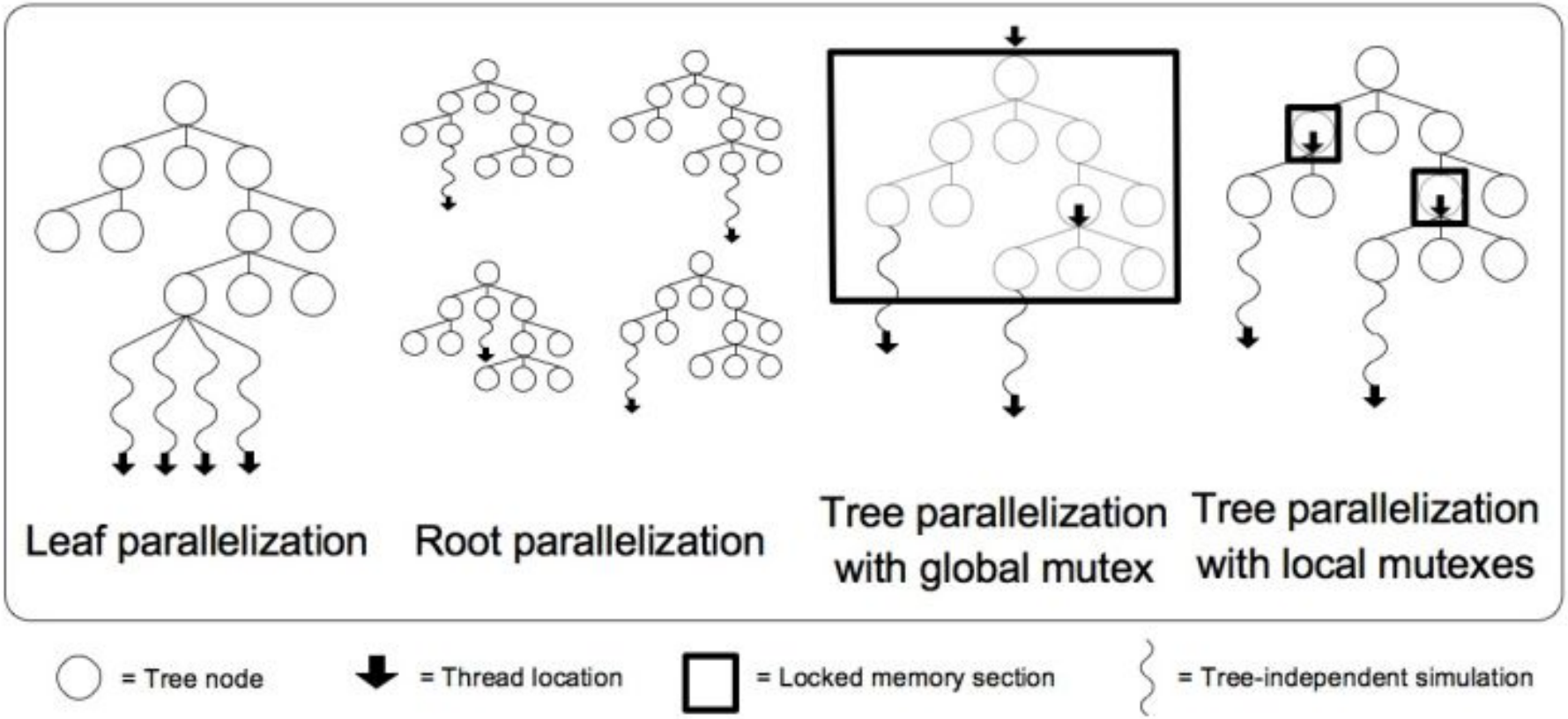


Image taken from: Chaslot, G. M. B., Winands, M. H., & Herik, H. J. V. D. (2008, September). Parallel monte-carlo tree search. In *International Conference on Computers and Games* (pp. 60-71). Springer, Berlin, Heidelberg.

Parallel MCTS

How to properly parallelize our computations?

Leaf parallelization

The main thread performs the selection and expansion steps and multiple simulations are then run in different threads. Once they all finish, the results are backpropagated.

Very easy to perform, as running simulations doesn't require any shared information.

Drawback – simulation time is unpredictable, the simulations will always have to wait for the longest one to finish.

Parallel MCTS

How to properly parallelize our computations?

Root parallelization

Run n instances of MCTS in parallel, one per thread.

Once time runs out, add up the scores of the moves computed in all the threads and choose the best one.

Again, no shared information is required which makes it quite straightforward to implement.

Parallel MCTS

How to properly parallelize our computations?

Tree parallelization with global mutex

One global mutex – only one thread can access the tree at any given time.

In the meantime, the other threads can run simulations from different leaf nodes (this makes it very different from leaf parallelization discussed earlier).

Tree parallelization with local mutexes

Each node has its own lock, allowing multiple threads to access the tree simultaneously.

Parallel MCTS

How to properly parallelize our computations?

Virtual loss

If multiple threads start traversing the tree one after another, they are likely to choose the same trajectory. To prevent this, a virtual loss can be assigned to nodes visited by a single thread.

Parallel MCTS

How to properly parallelize our computations?

Comparison

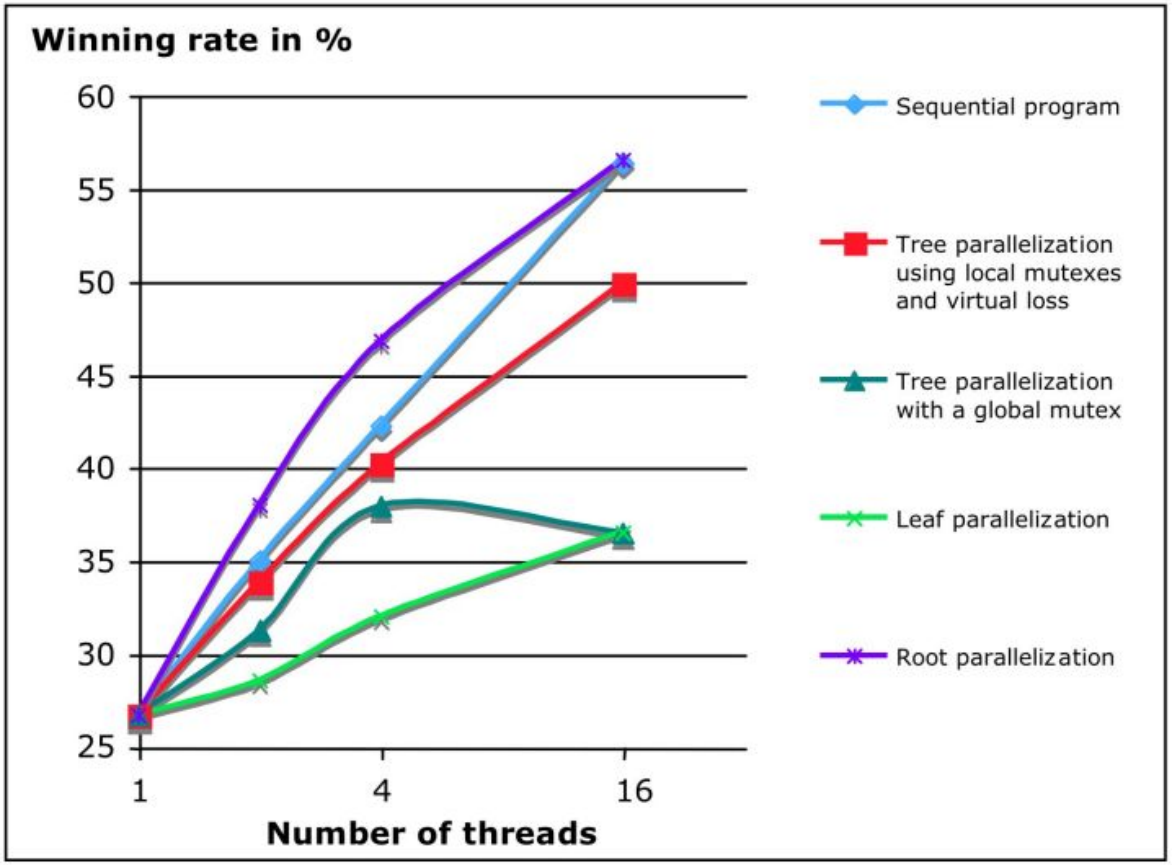


Image taken from: Chaslot, G. M. B., Winands, M. H., & Herik, H. J. V. D. (2008, September). Parallel monte-carlo tree search. In *International Conference on Computers and Games* (pp. 60-71). Springer, Berlin, Heidelberg.

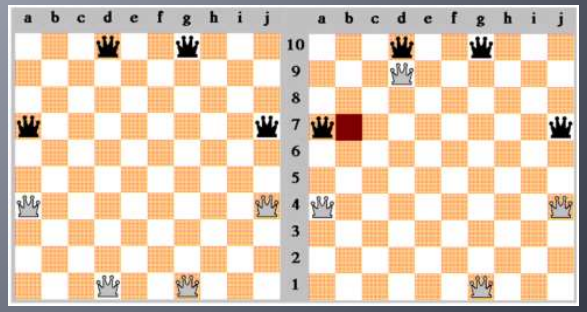
That’s it for today!

Faculty of Mathematics and Physics
Charles University
7.12.2022

NAIL122 AI for Games

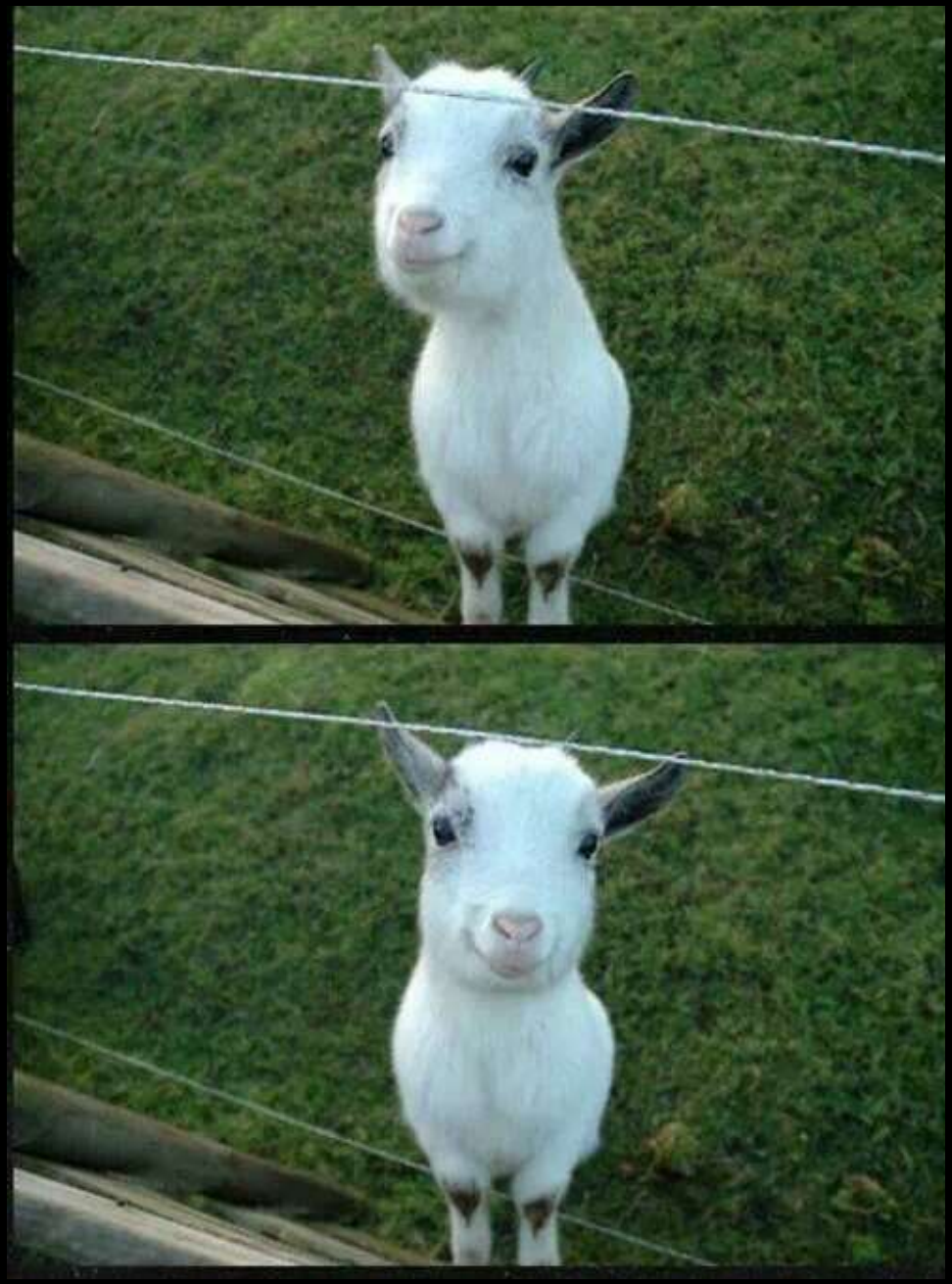
MCTS adjustments – part 1

Jakub Gemrot, Peter Guba



MCTS improvements that we'll go over today

1. Selection enhancements
    - UCB1-Tuned tree policy
    - SR+CR MCTS
    - First play urgency
    - Progressive bias
    - Move groups
2. Simulation enhancements
    - Decisive moves
    - Last good reply
    - MAST
3. Evaluation enhancements
    - Relative Bonus MCTS
    - FAP MCTS
    - WP MCTS
4. Other enhancements
    - MCTS_HP
    - AMAF
    - MCTS Solver
    - Score Bounded MCTS



Refresher

MCTS
High-level description

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

MCTS iteration

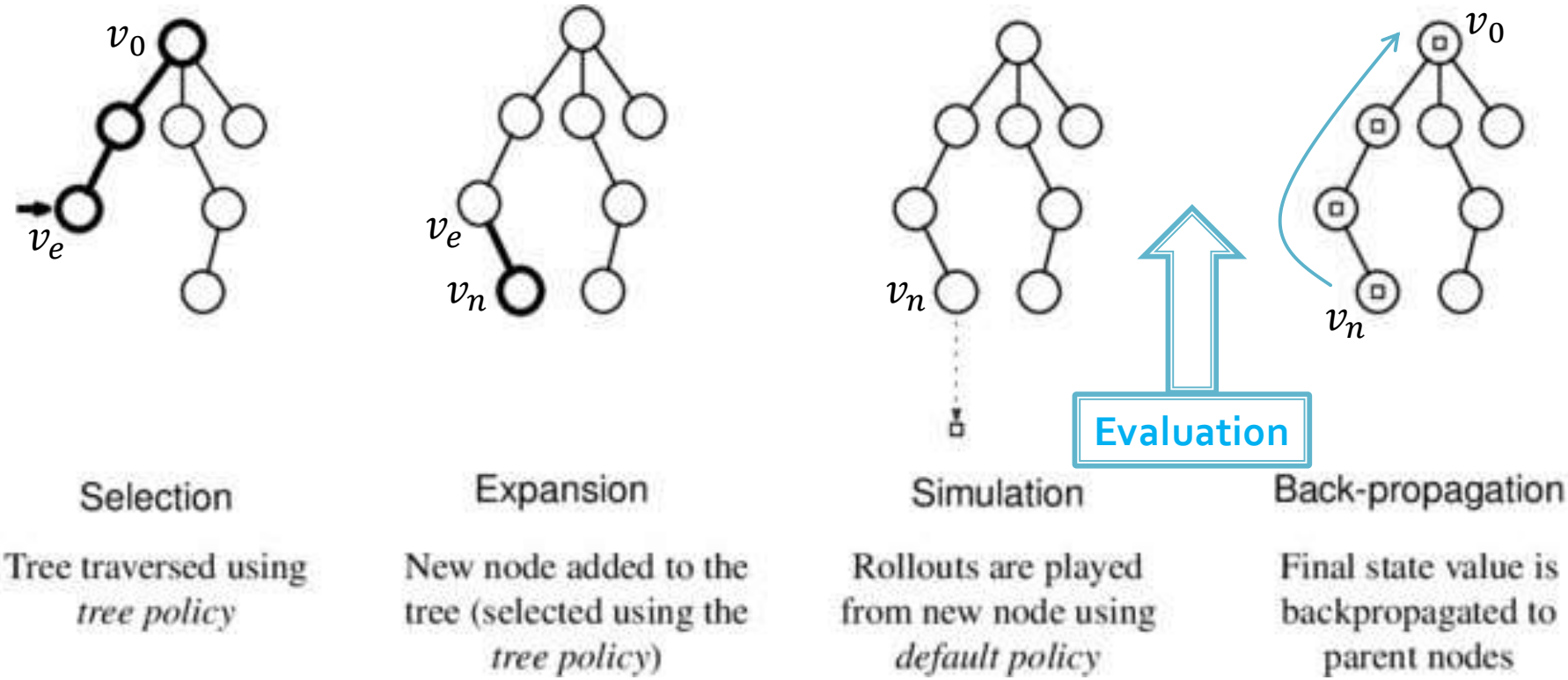


Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

Multi-armed Bandit

How to gamble properly

We kinda expect to have **a simulator** we can interact with in order to solve the problem before we pull an arm in real-life.

Multi-armed bandit game

There are K slot-machines, each having an arm.
You can choose an arm and pull it, i.e. you have $A = \{a_1, \dots, a_k\}$.
Choosing a slot-machine i and executing a_i will give you a reward $R_i(a_i)$.
Each reward $R_i(a_i)$ follows some unknown probability distribution.

The objective – Probably Approximately Correct (PAC) solution

Design an optimal policy π for the player in order to uncover an arm with the best expected reward.

That is, in the end, select an arm that probably (w/ high probability) has approximately the best expected reward.

Additional criterium

Use as fewest actions as possible to fulfill the objective.

How to select which arm to pull?

Upper Confidence Bounds

Upper confidence bounds (UCB) adaptive bandit algorithm

Tracked values:

$Q(a_i)$ mean of observed rewards after pulling a_i

n_i number of performed pulls

n number of total pulls so far

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$; always pull arm with the highest UCB

Exploitation part

Exploration part

UCB assumes payoffs to be in range $[0; 1]$! Otherwise, custom tweaks to c is required.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

Possible complications of applying MCTS to modern games

MCTS
Problems

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

High branching factor / severe time limits

MCTS
Problems

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

High branching factor / severe time limits

Long games (e.g. RTS combat)

⇒ Requires lot of time to reach the terminal state.

⇒ Once we do reach it, its value may be inconsequential.

MCTS
Problems

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

High branching factor / severe time limits

Long games (e.g. RTS combat)

⇒ Requires lot of time to reach the terminal state.

⇒ Once we do reach it, its value may be inconsequential.

Games might not end with 0/1

⇒ Some outcomes can be better then others though 0/1 will mask that out.

MCTS
Problems

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

High branching factor / severe time limits

Long games (e.g. RTS combat)

⇒ Requires lot of time to reach the terminal state.

⇒ Once we do reach it, its value may be inconsequential.

Games might not end with 0/1

⇒ Some outcomes can be better then others though 0/1 will mask that out.

Difficulty identifying trap states

MCTS improvements that we'll go over today

1. Selection enhancements
    - UCB1-Tuned tree policy
    - SR+CR MCTS
    - First play urgency
    - Progressive bias
    - Move groups
2. Simulation enhancements
    - Decisive moves
    - Last good reply
    - MAST
3. Evaluation enhancements
    - Relative Bonus MCTS
    - FAP MCTS
    - WP MCTS
4. Other enhancements
    - MCTS_HP
    - AMAF
    - MCTS Solver
    - Score Bounded MCTS

Selection enhancements

UCB1-Tuned Tree Policy

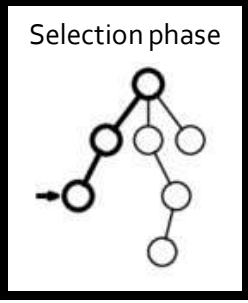
Selection enhancement

Source paper:
Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

Notes:
https://webdocs.cs.ualberta.ca/~games/go/seminar/notes/2007/slides_ucb.pdf

UCB1-Tuned

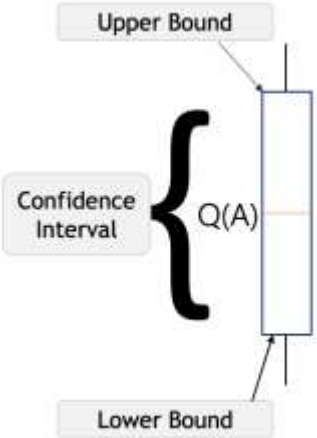
Tree policy



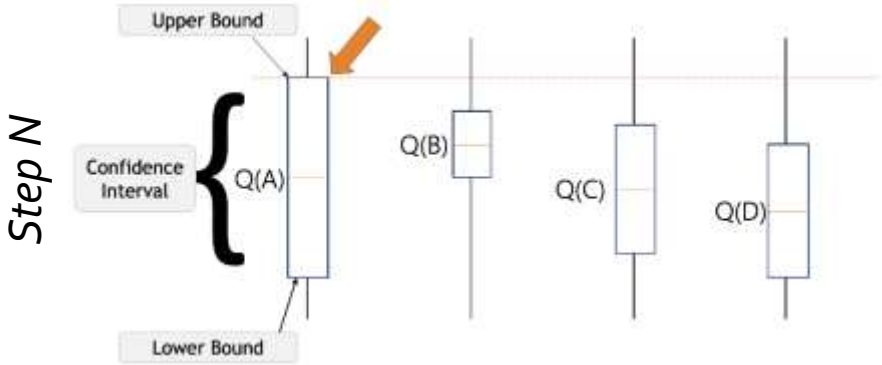
UCB revisited

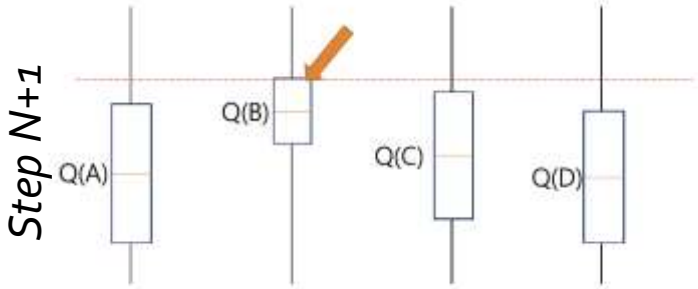
$Q(a_i)$ mean of observed rewards after pulling a_i
n_i number of performed pulls
n number of total pulls so far

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$



UCB is adding so-called upper confidence bound to the mean of observed rewards, in other words, we are buffing the observed rewards with a value that is trying to estimate uncertainty of our observations.

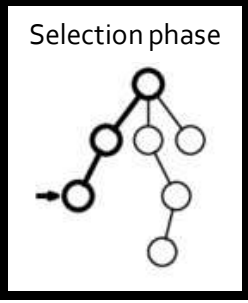




Images source: https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/

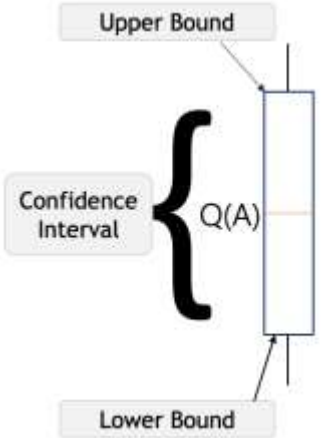
UCB1-Tuned

Tree policy



UCB1-tuned

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
$\boldsymbol{X_i}$	**observed rewards from arm $\boldsymbol{a_i}$**
$\boldsymbol{X_{i,s}}$	**observed reward from arm $\boldsymbol{a_i}$ when pulled for the $\boldsymbol{s}$-time**



$$UCB1_{tuned}(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}.\min\left\{\frac{1}{4}; V_i(n_i)\right\}}$$

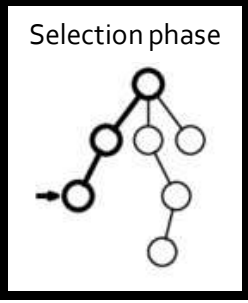
Where V_i is based on real observed variance of rewards for arm a_i.

$$V_i(n_i) = \sigma_{a_i} + \sqrt{\frac{2.\ln(n)}{n_i}} = E[X_i^2] - E[X_i]^2 + \sqrt{\frac{2.\ln(n)}{n_i}} =$$

$$= \frac{1}{n_i}\sum_{t=1}^{n_i} X_{i,t}^2 - Q(a_i)^2 + \sqrt{\frac{2.\ln(n)}{n_i}}$$

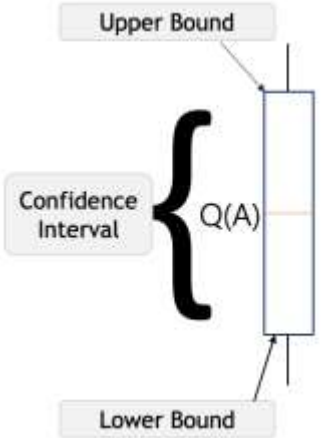
UCB1-Tuned

Tree policy



UCB1-tuned

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
X_i	observed rewards from arm a_i
$X_{i,s}$	observed reward from arm a_i when pulled for the s-time



$$UCB1_{tuned}(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}.\min\left\{\frac{1}{4};V_i(n_i)\right\}}$$

Where V_i is based on real observed variance o

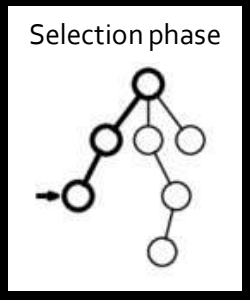
$$V_i(n_i) = \sigma_{a_i} + \sqrt{\frac{2.\ln(n)}{n_i}} = E[X_i^2] - E[X_i]$$

$$= \frac{1}{n_i}\sum_{t=1}^{n_i} X_{i,t}^2 - Q(a_i)^2 + \sqrt{\frac{2.}{n_i}}$$

Bounding the variance with $\frac{1}{4}$ as that is the highest variance for rewards [0;1]. If rewards are different, adapt accordingly.

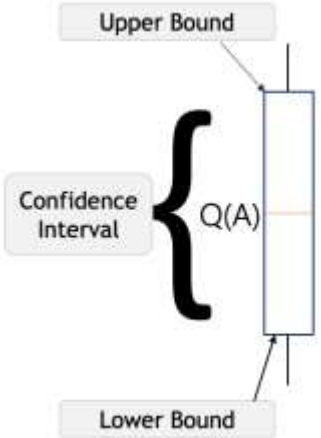
UCB1-Tuned

Tree policy



UCB1-tuned

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
X_i	observed rewards from arm a_i
$X_{i,s}$	observed reward from arm a_i when pulled for the s-time



$$UCB1_{tuned}(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}.\min\left\{\frac{1}{4}; V_i(n_i)\right\}}$$

Regret bounds not proved but found to perform better in practice than $UCB1$; used in variety of implementations including real-time game Tron.

SR+CR MCTS

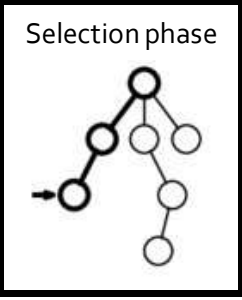
Selection enhancement

Source paper:

Tolpin, D., & Shimony, S. (2012). MCTS based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 26, No. 1, pp. 570-576).

SR+CR MCTS

Tree policy



Expected Regret

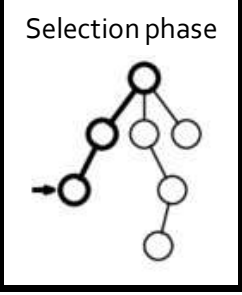
Put simply, expected regret is the difference between the expected reward of the best possible course of action and of the chosen course

Simple regret (SR) – computed over a single action

Cumulative regret (CR) – computed over multiple actions

SR+CR MCTS

Tree policy



Regret in MAB

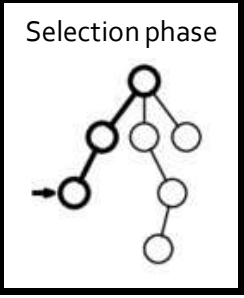
Regular UCB minimizes cumulative regret. This can make sense in the MAB setting when you get a reward for every arm pull.

Regret in MCTS

In MCTS, we only get a real reward for actually choosing an action at the end, so it makes more sense to minimize the simple regret at the root. In the rest of the nodes, cumulative regret should be minimized however, as they contribute to the evaluation of their parent nodes.

SR+CR MCTS

Tree policy



Policies for minimizing simple regret

ε-greedy: choose the action with the highest mean observed reward with probability ε, choose any other action with probability $\frac{1}{n-1}$ where n is the number of possible actions

UCB$_{\sqrt{\ }}$: $Q(a_i) + c\sqrt{\frac{\sqrt{n}}{n_i}}$

First Play Urgency

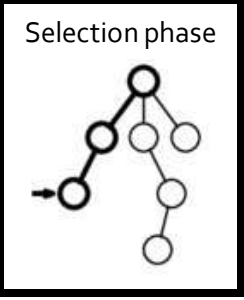
Selection enhancement

Source paper:

Gelly, S., & Wang, Y. (2006, December). Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*.

First Play Urgency

Tree policy



UCB revisited

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
X_i	observed rewards from arm a_i
$X_{i,s}$	observed reward from arm a_i when pulled for the s-time

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$

Initially, $n_i = 0$ for not-yet-tried arm a_i making $\mathrm{UCB}(a_i) = \infty$. We need to try each arm first before resort to exploitation.

First play urgency circumvent this by using tree policy:

$$\mathrm{FPU}(a_i) = \begin{cases} x & \text{for } n_i = 0 \\ \mathrm{UCB}(a_i) & \text{for } n_i > 0 \end{cases}$$

where x is a hand-picked problem-dependent constant. Allows for early exploitations.

Progressive Bias

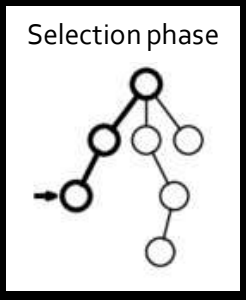
Selection enhancement

Source paper:

Chaslot, G. M. J., Winands, M. H., HERIK, H. J. V. D., Uiterwijk, J. W., & Bouzy, B. (2008). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(03), 343-357.

Progressive Bias

Tree policy



Marrying heuristic with UCB (domain dependent technique)

Not using well-known and sound heuristic for a given game would be a waste.

Instead of using UCB directly, we add "heuristic value".

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
$a_i.s$	state in which we apply a_i

$$PB(a_i) = UCB(a_i) + \alpha(a_i, n_i, n).h(a_i.s)$$

First play urgency can be modeled by this.

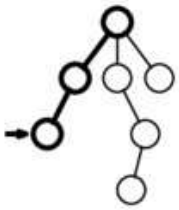
Heuristic discount factor possibly dependent on the arguments.

Heuristic value of the state; scale accordingly to the range of UCB.

Progressive Bias

Tree policy

Selection phase



Marrying heuristic with UCB (domain dependent technique)

Not using well-known and sound heuristic for a given game would be a waste.

Instead of using UCB directly, we add "heuristic value".

$Q(a_i)$	mean of observed rewards after pulling a_i
n_i	number of performed pulls
n	number of total pulls so far
$a_i.s$	state in which we apply a_i

$$PB(a_i) = UCB(a_i) + \alpha(a_i, n_i, n).h(a_i.s)$$

Paper experimented with $\alpha(a_i, n_i, n) = \frac{1}{n_i}$.

Beware of the computational cost of heuristic function h.

Move Groups

Selection enhancement

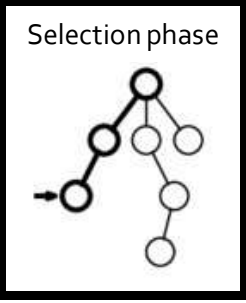
Source papers:

Childs, B. E., Brodeur, J. H., & Kocsis, L. (2008, December). Transpositions and move groups in Monte Carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games* (pp. 389-395). IEEE.

Saito, J. T., Winands, M. H., Uiterwijk, J. W., & Van Den Herik, H. J. (2007). Grouping nodes for Monte-Carlo tree search. In *Computer Games Workshop* (pp. 276-283).

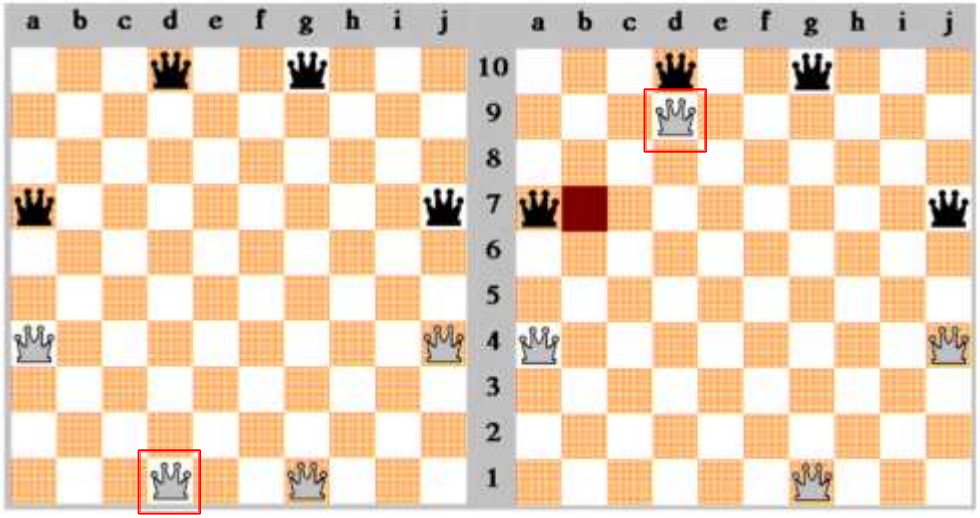
Move Groups

Tree policy



The Amazons Game

Initial position: White selects Amazon in red Rectangle.



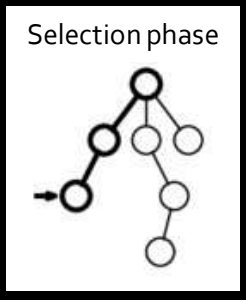
Amazon is moved and red square was shot at; this square becomes blocked.

Each player begins with 4 Amazons (move like Queens in Chess).
Player move: select Amazon, move, then shoot arrow to block some tile forever (shoot follows the same logic as movement, i.e., you can shoot to a place you could have moved to with the Amazon after the first move you make).
Loss condition: a player is unable to move.
Score: number of moves a winner can make after the loss condition is triggered.

Images from: Kloetzer, J., Iida, H., & Bouzy, B. (2007). The Monte-Carlo approach in Amazons. In *Proceedings of the Computer Games Workshop* (pp. 185-192).

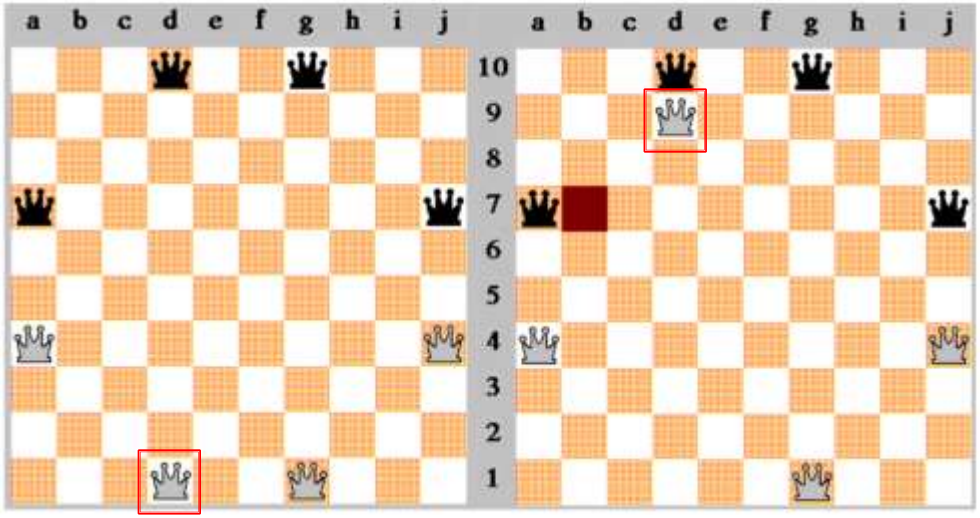
Move Groups

Tree policy



The Amazons Game

Initial position: White selects Amazon in red Rectangle.



Amazon is moved and red square was shot at; this square becomes blocked.

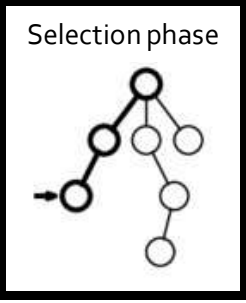
Initial MCTS idea: model move and shoot as single node, i.e., a player's move in Amazons is modelled by single nodes within MCTS tree.

… is that wise?

Images from: Kloetzer, J., Iida, H., & Bouzy, B. (2007). The Monte-Carlo approach in Amazons. In *Proceedings of the Computer Games Workshop* (pp. 185-192).

Move Groups

Tree policy



The Amazons Game

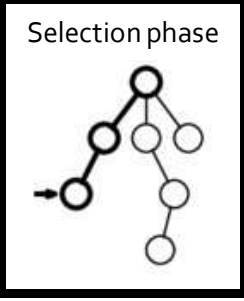
For the move let's denote for Amazon a_i:
n… number of legal positions to move to
m… number of legal positions to shoot to (after any move)

Upper bound of number of actions for $a_i \sim n.m$ in a given MCTS node.

UCB is going to be computed for every node from the total of $n.m$ => there is single optimal action to be exploited against $\sim n.m - 1$ sub-optimal actions in a given MCTS node.

Move Groups

Tree policy



The Amazons Game

Now consider the case where we split the player move action into two steps. I.e., for selected Amazon a_i first consider only move actions and then consider shoot actions.

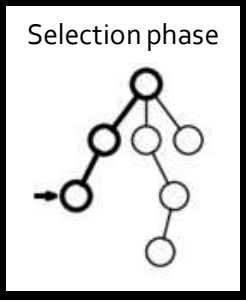
On the level of move action selection, there will be only n competing action, we would possibly select sub-optimal move action $n - 1$ times. For optimal move action selected there would be less then $m - 1$ sub-optimal actions to select.

We are weighing approx. $n.m$ sub-optimal selection against $n + m$.

⇒ Splitting the move action into two steps is effectively grouping "shoot actions" with initial move action in the eyes of the tree policy at the cost of space and applying a full player's move.

Move Groups

Tree policy



Other Games

The game of Amazons easily lends itself to the approach we have described in the previous slides. However, the same approach can be applied other games as well if one comes up with conditions for separating moves in the groups that are relevant to decision making.

For example, a group of researchers has successfully applied this approach to Go. They separated moves into three groups:

1. Legal moves in proximity of the last move
2. Legal moves on the border of the game board that do not belong to the previous group
3. Other moves

They found that their variant of the algorithm beat standard UCT in 836 games out of 1000.

Simulation enhancements

Decisive Moves

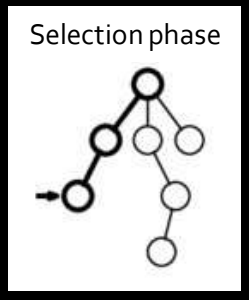
Selection and simulation enhancement

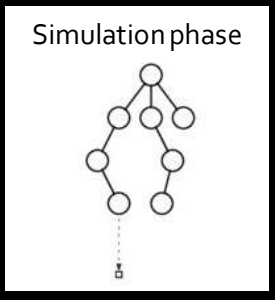
Source paper:

Teytaud, F., & Teytaud, O. (2010, August). On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games* (pp. 359-364). IEEE.

Decisive moves

Tree / Default policy





Decisive move

A move that immediately leads to winning.

Anti-Decisive move

A move that prevents the opponent to perform a decisive move.

Tree / Default policy modification

Use (anti-)decisive move if possible, otherwise use original tree / default policy.

Shown to increase playing strength despite extra computational cost of (anti-) decisive moves in Go and Hex.

Last Good Reply

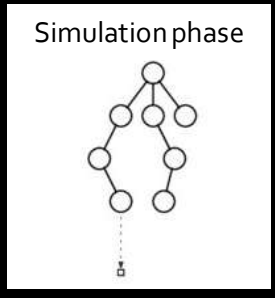
Simulation enhancement

Source paper:

Drake, P. (2009). The last-good-reply policy for Monte-Carlo Go. *Icga Journal*, *32*(4), 221-227.

Last Good Reply

Default policy



Every move is considered a reply to the previous move. For every one of the opponents moves, the last reply which led to a win is stored and played in the playout if it is legal.

More replies can be stored this way so that the algorithm has some back up moves to choose from in case the first one to be tried isn't legal anymore.

In an extension of this approach called Last Good Reply with Forgetting, the reply is also forgotten once it leads to a loss.

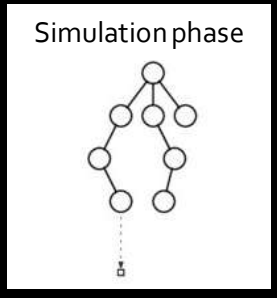
MAST

Simulation enhancement

Source paper:
Finnsson, H., & Björnsson, Y. (2008, July). Simulation-based approach to general game playing. In *Aaai* (Vol. 8, pp. 259-264).

MAST

Default policy



MAST – Move Average Sampling Technique

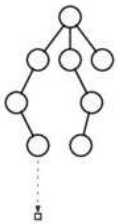
A table is kept which stores the average reward for each move. This table is updated every iteration and values in it are used to bias the selection of moves in playouts using the Gibbs distribution.

$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\sum_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

MAST

Default policy

Simulation phase



MAST – Move Average Sampling Technique

A table is kept which stores the average reward for each move. This table is updated every iteration and values in it are used to bias the selection of moves in playouts using the Gibbs distribution.

The probability of an action *a* being chosen.

$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\sum_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

MAST
Default policy

MAST – Move Average Sampling Technique

A table is kept which stores the average reward for each move. This table is updated every iteration and values in it are used to bias the selection of moves in playouts using the Gibbs distribution.

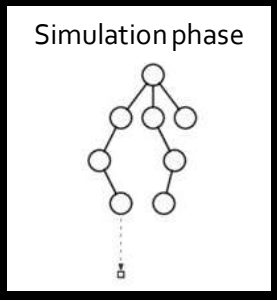
$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\sum_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

The average reward for action *a*.

The probability of an action *a* being chosen.

MAST

Default policy



MAST – Move Average Sampling Technique

A table is kept which stores the average reward for each move. This table is updated every iteration and values in it are used to bias the selection of moves in playouts using the Gibbs distribution.

$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\sum_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

The average reward for action *a*.

The probability of an action *a* being chosen.

A tuning parameter.

Evaluation enhancements

Relative Bonus MCTS

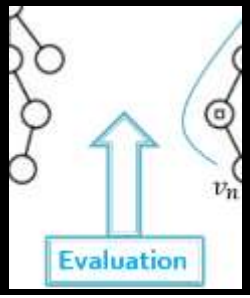
Evaluation enhancement

Source paper:

Pepels, T., Tak, M. J., Lanctot, M., & Winands, M. H. (2014). Quality-based rewards for Monte-Carlo tree search simulations. In *ECAI 2014* (pp. 705-710). IOS Press.

Relative Bonus MCTS

Evaluation update

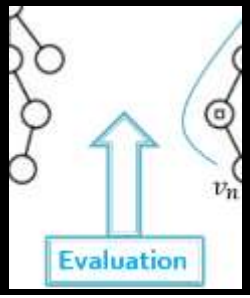


Relative Bonus

A bonus computed from the length of the playout and the depth at which it starts. The idea is that the shorter the playout, the more relevant the information it provides is. This bonus is used as a control variate (explained in the following slides).

Relative Bonus MCTS

Evaluation update



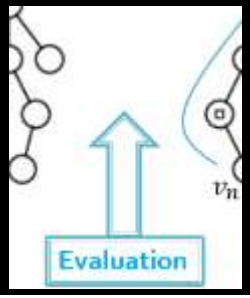
Control Variates

A variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

$$Z = X + a * (Y - \mathrm{E}[Y])$$

Relative Bonus MCTS

Evaluation update



Control Variates

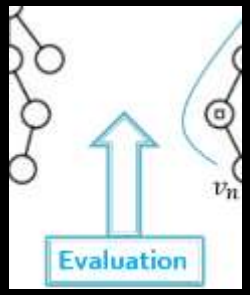
A variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

$$Z = X + a * (Y - \mathrm{E}[Y])$$

The original estimator of some value E[X].

Relative Bonus MCTS

Evaluation update



Control Variates

A variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

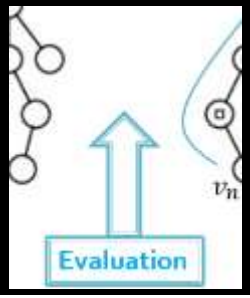
$$Z = X + a * (Y - \mathrm{E}[Y])$$

The original estimator of some value E[X].

A correlated variable with known mean.

Relative Bonus MCTS

Evaluation update



Control Variates

A variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

$$Z = X + a * (Y - \mathrm{E}[Y])$$

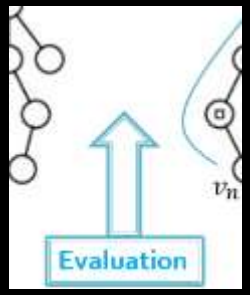
New estimator of E[X].

The original estimator of some value E[X].

A correlated variable with known mean.

Relative Bonus MCTS

Evaluation update



Control Variates

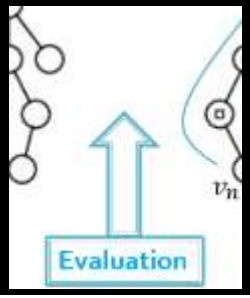
A variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

$$Z = X + a * (Y - \mathrm{E}[Y])$$

If X and Y are correlated, it can be shown that setting a to $-\mathrm{Cov}(X, Y)/\mathrm{Var}(Y)$ minimizes $\mathrm{Var}(Z)$.

Relative Bonus MCTS

Evaluation update



Control Variates

A variance [illegible] ge of correlatio[illegible] that one of them has [illegible] ed estimator with a red[illegible]

As mentioned before, the control variate in our case is the length of the playout. Since the mean value of the length of playouts isn't known in advance however, an online-computed estimate is used.

$$Z = X + a * (Y - \mathrm{E}[Y])$$

If X and Y are correlated, it can be shown that setting a to $-\mathrm{Cov}(X, Y)/\mathrm{Var}(Y)$ minimizes $\mathrm{Var}(Z)$.

FAP MCTS

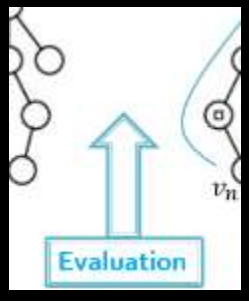
Evaluation enhancement

Source paper:

Xie, F., & Liu, Z. (2009, November). Backpropagation modification in monte-carlo game tree search. In *2009 Third International Symposium on Intelligent Information Technology Application* (Vol. 2, pp. 125-128). IEEE.

FAP MCTS

Evaluation update



FAP – Feedback Adjustment Policy

Basic Idea – Playouts performed later in the search should be treated as more important.

Divides playouts into groups based on when they are performed. Each group is then assigned a multiplicative factor *n* which makes the playouts in the group worth *n* normal playouts. The later the playouts are performed, the larger the factor.

This variant seems to be able to handle large numbers of units better than other MCTS variants.

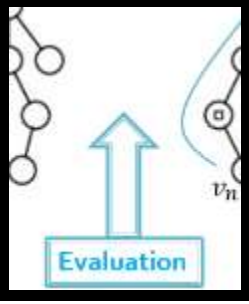
Disadvantage – needs to know the number of playouts.

WP MCTS

Evaluation enhancement

WP MCTS

Evaluation update



WP – Weighted Propagation

Requires a heuristic for estimating how good a state is.

Basic Idea – One should be able to take into account the whole trajectory of the playout, not just the final state.

Computes the heuristic for every state encountered during a playout, averages them and uses that as the backpropagated value.

Other enhancements

MCTS_HP

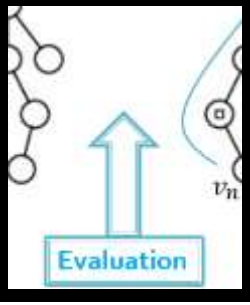
Evaluation and Backpropagation enhancement

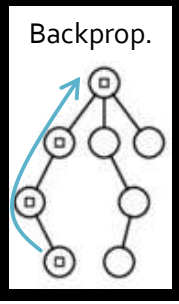
Source paper:

Šmejkal, P., & Gemrot, J. (2018, September). Engaging turn-based combat in the children of the Galaxy videogame. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 14, No. 1, pp. 229-235).

MCTS_HP

Evaluation and Backpropagation update

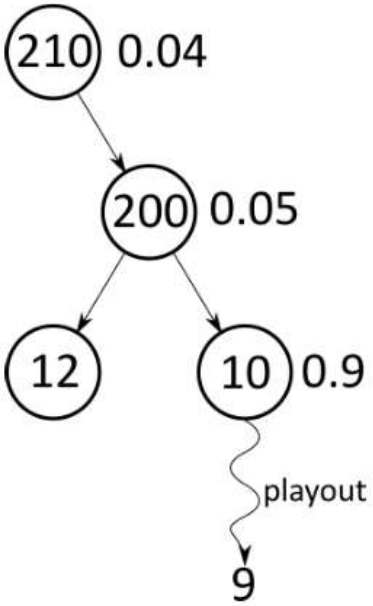




Primarily applicable to games where the player fights using multiple units, so mostly RTS games.

Evaluation adjustment – the difference between the sums of remaining hit points of the two teams is used as the score

Backpropagation adjustment – the backpropagated score is normalized (divided by the sum of hit points of the team that won in the playout)



All moves as first (AMAF)

Backpropagation enhancement

First mention:

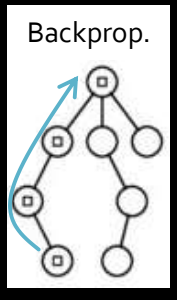
Gelly, S., & Silver, D. (2007, June). Combining online and offline knowledge in UCT. In *Proceedings of the 24th international conference on Machine learning* (pp. 273-280).

Follow-up paper:

Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, *175*(11), 1856-1875.

All moves as first (AMAF)

Backpropagation policy update



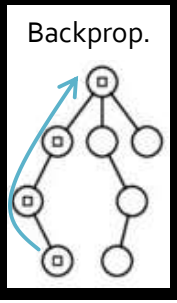
AMAF idea

Picture the game Go. During playout, we might be playing moves, which are either viable on top of the tree or as siblings to nodes selected during tree descent (selection phase).

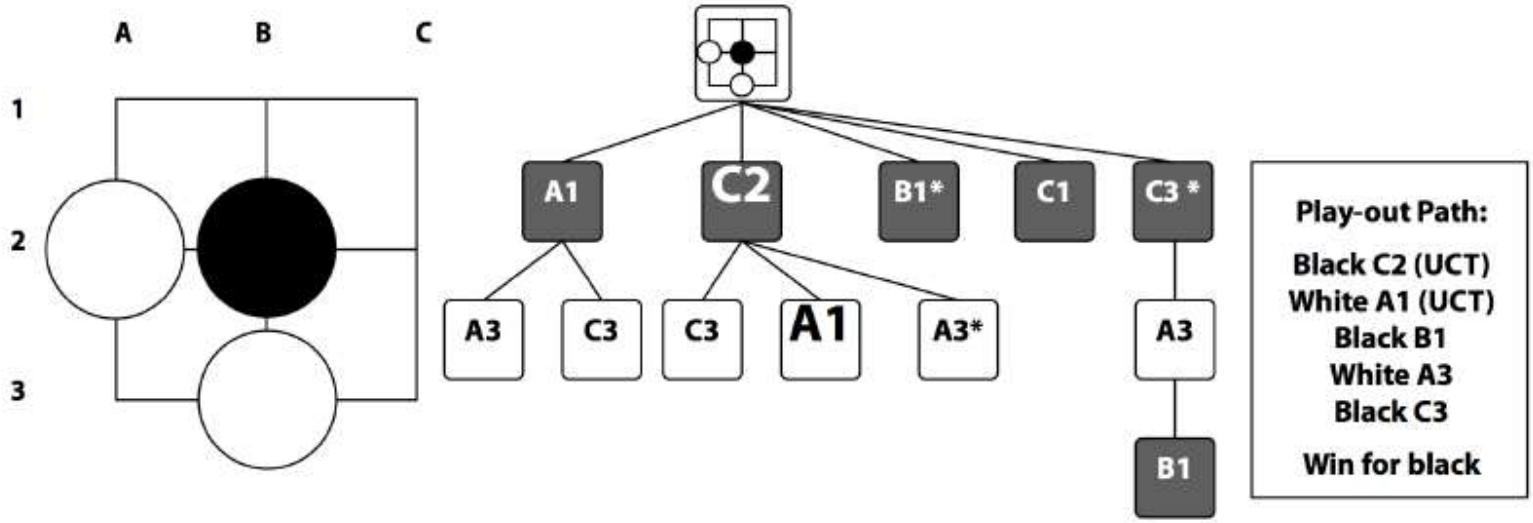
As we know the result of the playout for these moves, we can update values of these nodes; if we reorder moves within the playout, these move would be a valid playout for such nodes.

All moves as first (AMAF)

Backpropagation policy update



AMAF example



- Nodes marked with * have their MCTS stats updated given the playout (which we would not done with standard backpropagation)
- Such score is sometimes tracked separately and called AMAF score (or similarly according to concrete AMAF variation)

Image adapted from: Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1-43.

MCTS Solver

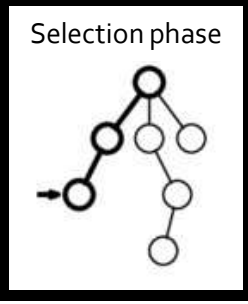
Selection & Backpropagation enhancement

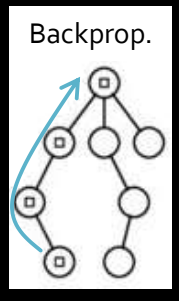
Source paper:

Winands, M. H., Björnsson, Y., & Saito, J. T. (2008, September). Monte-Carlo tree search solver. In *International Conference on Computers and Games* (pp. 25-36). Springer, Berlin, Heidelberg.

MCTS Solver

Selection & Backpropagation update





MCTS has a problem discovering specific sequences of moves that are only good when performed one after another (i.e. a sequence of moves that surely leads to checkmate in check).

One way to help alleviate this disadvantage is to try to prove which nodes are sure wins and which are sure losses. This can be done by additionally backpropagating values ∞ and -∞.

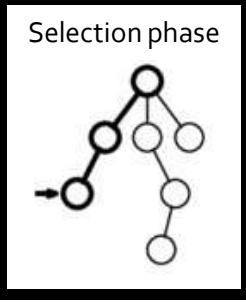
Terminal node that is a win => ∞

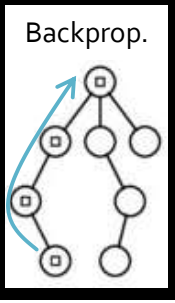
Terminal node that is a loss => -∞

Nonterminal node => ∞ if at least one of its children leads to a win, -∞ if all of its nodes lead to losses

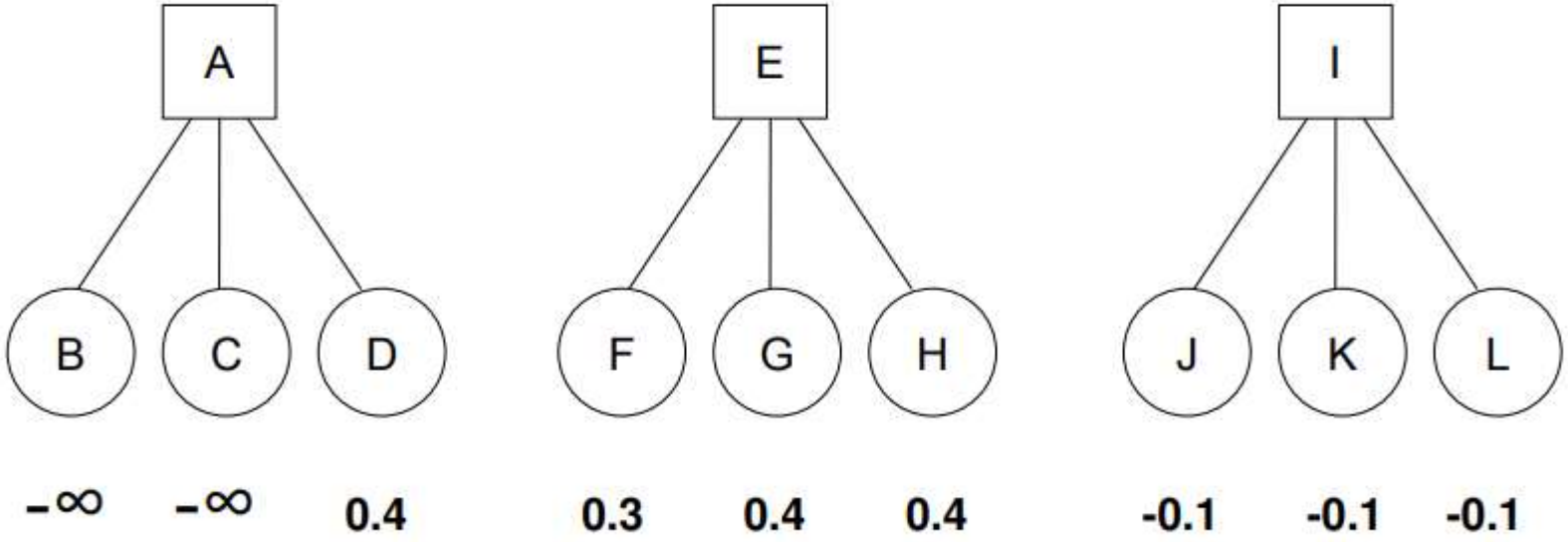
MCTS Solver

Selection & Backpropagation update



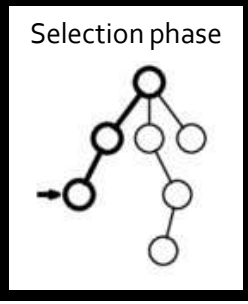


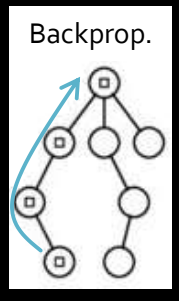
How should we treat -∞ during selection?



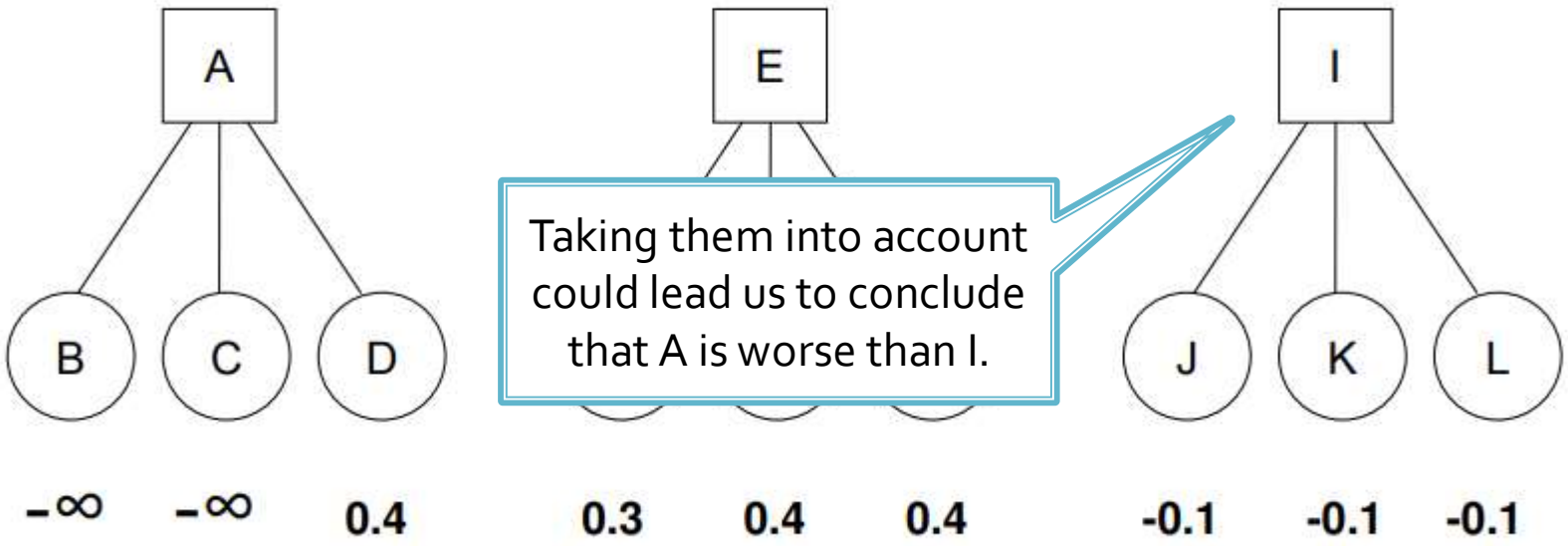
MCTS Solver

Selection & Backpropagation update



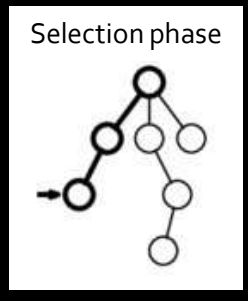


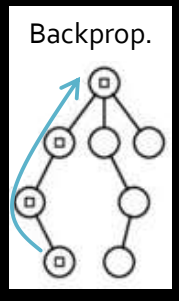
How should we treat -∞ during selection?



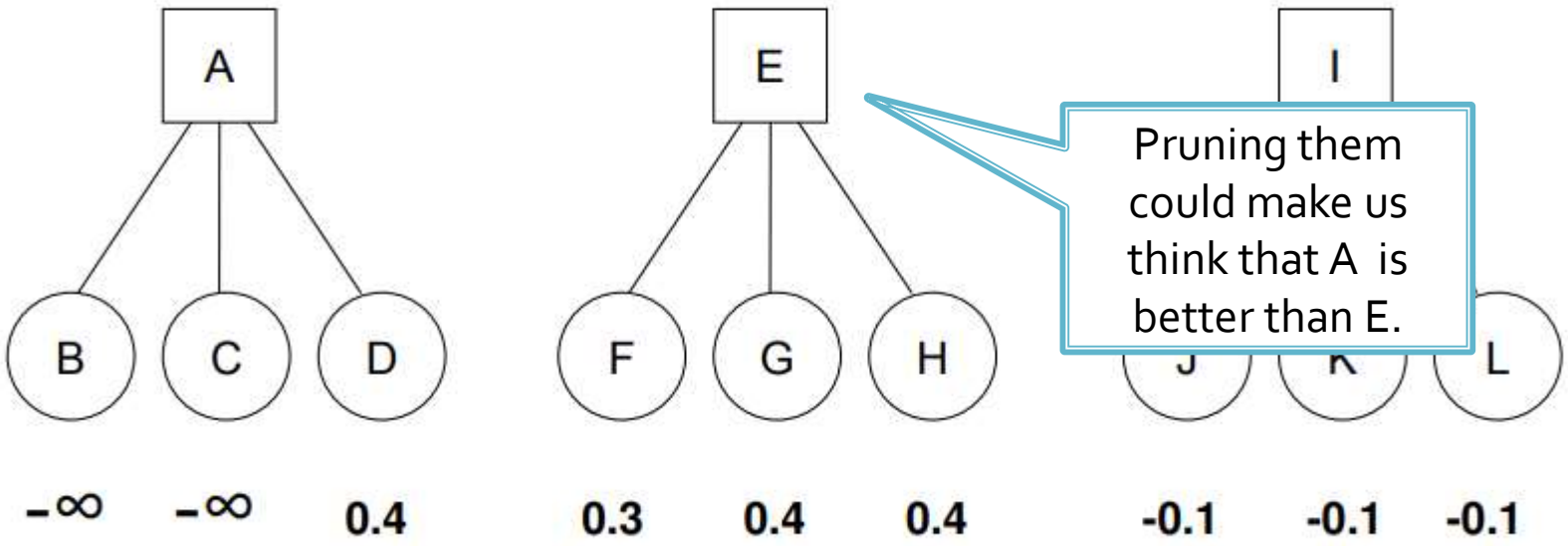
MCTS Solver

Selection & Backpropagation update



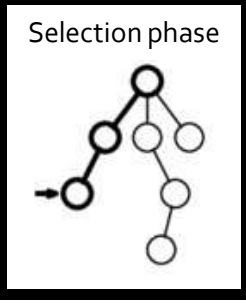


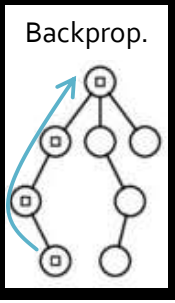
How should we treat -∞ during selection?



MCTS Solver

Selection & Backpropagation update



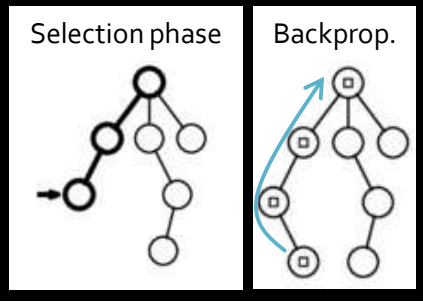


How should we treat -∞ during selection?

Essentially an open question. The authors of the original paper perform selection using their default policy for the first N times a node is visited, then start using UCB and disregard the nodes which are proven losses.

MCTS Solver

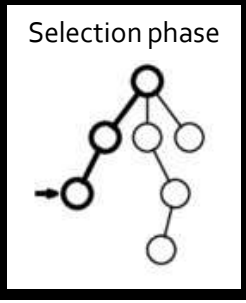
Selection & Backpropagation update

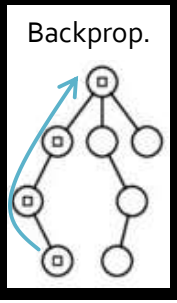


This points to a broader question with respect to MCTS:

MCTS Solver

Selection & Backpropagation update



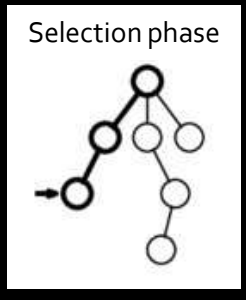


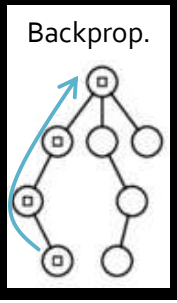
This points to a broader question with respect to MCTS:

How to take into account the number of available moves?

MCTS Solver

Selection & Backpropagation update





This points to a broader question with respect to MCTS:

How to take into account the number of available moves?

There are games where having few moves available can be indicative of being in a bad position – chess, for example. How to take that into account?

(Also an open question.)

Score Bounded MCTS

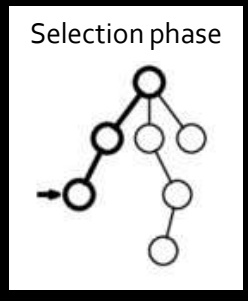
General enhancement

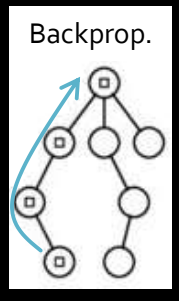
Source paper:

Cazenave, T., & Saffidine, A. (2010, September). Score bounded Monte-Carlo tree search. In *International Conference on Computers and Games* (pp. 93-104). Springer, Berlin, Heidelberg.

Score Bounded MCTS

General update





Score Bounding idea

The game may end with different results then "win/draw/lose".
Track "pessimistic" $pess(n)$ and "optimistic" $opti(n)$ score values for nodes; propagate them as well.

We label optimal play in node n as $real(n)$.

It holds that: $pess(n) \leq real(n) \leq opti(n)$.

Initialize $pess(n) = minscore$; $opti(n) = maxscore$.

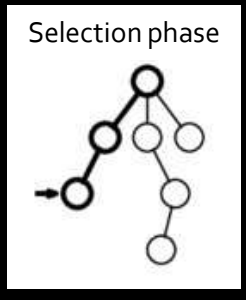
For max/min nodes we compute:

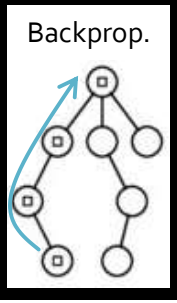
$$pess(n) \coloneqq \max/\min_{s \in children(n)} pess(n)$$

$$opti(n) \coloneqq \max/\min_{s \in children(n)} opti(n)$$

Score Bounded MCTS

General update





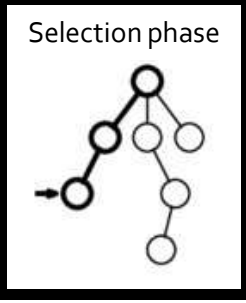
Score Bounding propagation (pessimistic bounds)

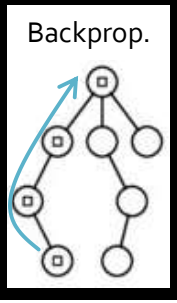
Algorithm 1 Pseudo-code for propagating pessimistic bounds

```
procedure prop-pess
arguments node s
if s is not the root node then
    Let n be the parent of s
    Let old_pess := pess(n)
    if old_pess < pess(s) then
        if n is a Max node then
            pess(n) := pess(s)
            prop-pess(n)
        else
            pess(n) := min_{s' ∈ children(n)} pess(s')
            if old_pess > pess(n) then
                prop-pess(n)
            end if
        end if
    end if
end if
```

Score Bounded MCTS

General update





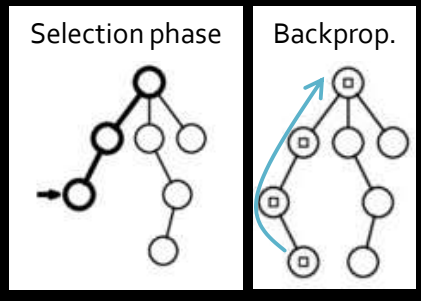
Score Bounding propagation (optimistic bounds)

Algorithm 2 Pseudo-code for propagating optimistic bounds

```
procedure prop-opti
arguments node s
if s is not the root node then
    Let n be the parent of s
    Let old_opti := opti(n)
    if old_opti > opti(s) then
        if n is a Max node then
            opti(n) := max_{s' ∈ children(n)} opti(s')
            if old_opti > opti(n) then
                prop-opti(n)
            end if
        else
            opti(n) := opti(s)
            prop-opti(n)
        end if
    end if
end if
```

Score Bounded MCTS

General update

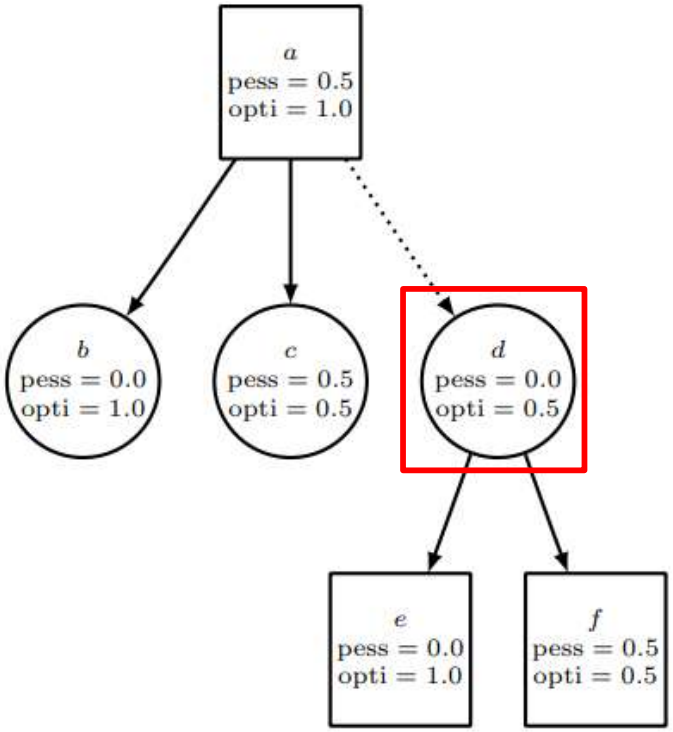


Alpha-beta like cuts

Once we have information about pess/opti values for nodes, we can perform pruning similar to alpha/beta improvement of min/max search.

Node c is solved as pess==opti;
and it seems node d cannot give better results. We prune d.

The problem is, when to update pess/opti values of leafs, which are not terminals.



That’s it for today!

Faculty of Mathematics and Physics
Charles University
23.11.2022

NAIL122 AI for Games

Assignment 1

Peter Guba

The Assignment

Implement MCTS in chess

Steps:

1. Download this project:
https://github.com/peter-guba/MCTS_chess_assignment

2. Open it in Unity and adjust the MCTSSearch class so that it performs MCTS.

3. Profit!

The Project

The Project

I used this project as the basis: https://github.com/SebLague/Chess-AI

The original project was made by this guy:
https://www.youtube.com/@SebastianLague

Here is a video explaining the project:
https://www.youtube.com/watch?v=U4ogKoMIzqk
(You don't need to watch it to do the homework, but it's very interesting.)

The Project

I used this project as the basis: https://github.com/SebLague/Chess-AI

The original project was made by t[illegible]uy
https://www.youtube.com/@Sebastian[illegible]

Here is a video explaining the project:
https://www.youtube.com/watch?v=U4og[illegible]
(You don't need to watch it to do the homework, but it's very interesting.)

I had to jerry-rig 2 projects together, so it's sadly not going to be the prettiest piece of software you've ever seen.

How to work with it

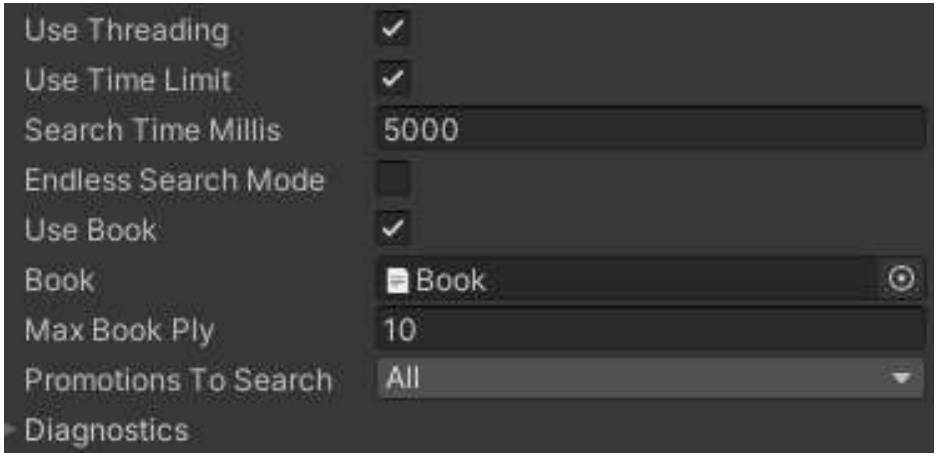
I made the adjustments to the project in Unity 2021.2.6f1, but the original project was made in Unity 2019.4.15f1, so it should work fine in versions from 2019 onwards.

After you run it, you'll be presented with a chess board and 6 buttons, the functions of which should be quite self-explanatory.

You can change the game and AI settings in the editor by clicking on the Game Manager object and then adjusting the values in its scripts.

AI Settings

Besides the MCTS algorithm that you are supposed to implement, the project already has a functioning implementation of Alpha-Beta Search. Both of these algorithms have some common settings. Here are the important ones:

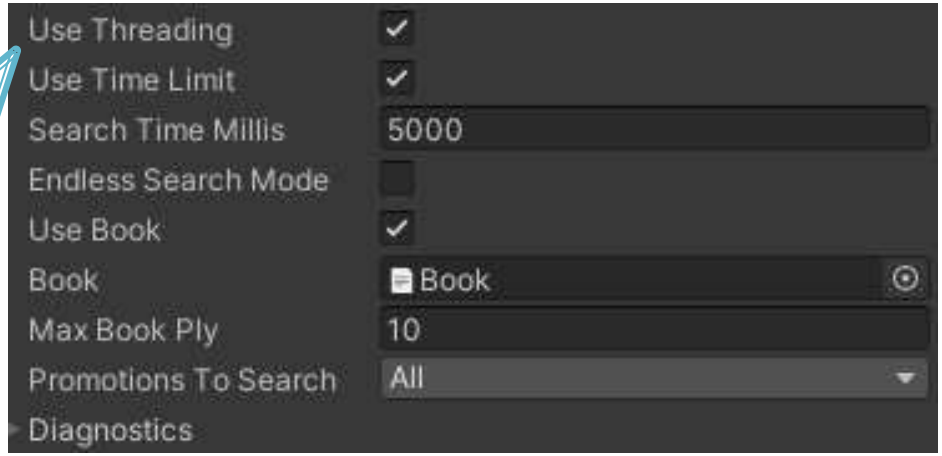


AI Settings

Besides the MCTS algorithm that you are supposed to implement, the project already has a functioning implementation of Alpha-Beta Search. Both of these algorithms have some common settings. Here are the important ones:

If threading is used, then the AI algorithm will be run in a different thread than the main one meaning the project won't freeze while waiting for the algorithm to finish.

DO NOT USE THIS UNTIL YOU THINK YOUR ALGORITHM WORKS!!! The reason is that, if an error occurs within the AI thread, you won't be notified, the thread just won't finish.

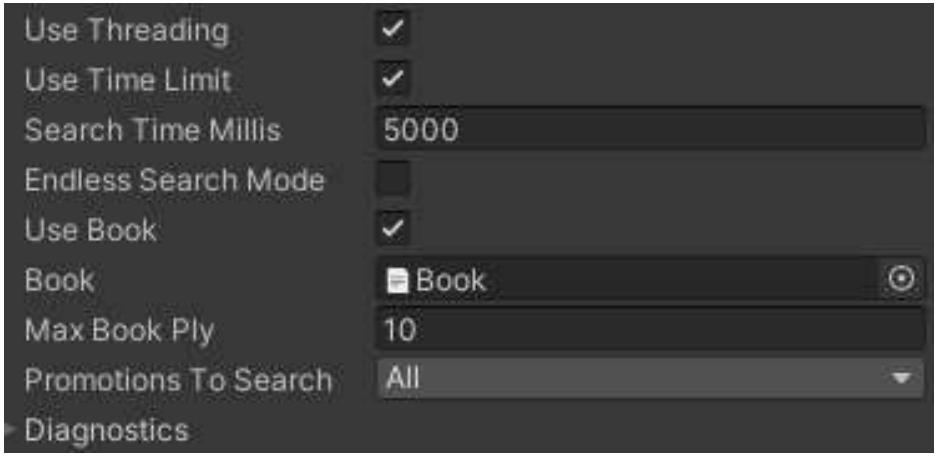


AI Settings

Besides the MCTS algorithm that you are supposed to implement, the project already has a functioning implementation of Alpha-Beta Search. Both of these algorithms have some common settings. Here are the important ones:

If time limit is used, the AI will have a given number of milliseconds to make a decision. This number is entered through the "Search Time Millis" parameter.

THIS ONLY WORKS WHEN THREADING IS USED!!!



Tests

The tests consist of 15 problems where your program has to find the correct sequence of moves to checkmate its opponent. The problems are divided into three groups of five, based on how many moves the sequence has – 1, 2 or 3, more than that turned out to be too complex. I left the problems for 4- and 5-move sequences as sequences, in case some of you'd like to give them a try.

After the tests finish running, the program will tell you how many were completed successfully. There is also a log file (simply called log.txt) to which the results of the individual tests will be written – it is located at ./Chess-AI-main.

API

Important classes

MCTSSearch – The class that you're supposed to expand.

Board – Represents the current state of the chess board.

Move – Represents a single move.

MoveGenerator – A class for generating moves.

Evaluation – A class for evaluating states.

FenUtility – Contains methods for storing and loading board configurations.

SimMove, SimPiece, SimPieceType – Classes used in simulation – lightweight equivalents of their counterparts that lack the sim- prefix.

Board

important methods & attributes

WhiteToMove – A boolean variable that indicates which player is on the move.

Clone() – Creates a clone of the board.

GetLightWeightClone() – Creates a lightweight clone of the board (a 2D array of SimPieces). This lightweight clone is used in simulations.

MakeAMove(Move move, bool inSearch) – Applies a given move to the board. The second parameter is set to false by default and isn't something you need to worry about (it is used by AlphaBetaSearch).

MoveGenerator

important methods & attributes

GenerateMoves(Board board, bool isRoot, bool includeQuietMoves) – Generates possible moves for the given board. The second argument determines whether the node for which the moves are to be generated is the root node. The third parameter is set to true by default and you should leave it that way (it determines whether non-capturing moves are generated too).

GetPossibleSimMoves(SimPiece[,] state, bool team) – Generates possible moves for a state given as a 2D array of SimPieces. Used during simulations. The second parameter determines the team for which the moves are to be generated.

MoveGenerator

important methods & attributes

GenerateMoves(Board board, bool isRoot, bool includeQuietMoves) – Generates possible moves for the given board. The second argument determines whether the node for which the moves are to be generated is the root node. The third parameter is set to true by default and you should leave it that way (it determines whether non-capturing moves are generated too).

There is a similar function called GenerateMoves_DO_NOT_USE.
DO NOT USE THAT ONE!!!

GetPossibleSimMoves(SimP[illegible]nerates possible moves for a state given as a [illegible]luring simulations. The second parameter determines the team for which the moves are to be generated.

Evaluation

important methods & attributes

EvaluateSimBoard(SimPiece[,] board, bool team) – Evaluates how good a state given as a 2D array of SimPieces is.

FenUtil

important methods & attributes

startFen – A string that determines the configuration of the chess board when the program is turned on (more how the configuration is encoded later). Useful for debugging.

SimPiece

important methods & attributes

type – The type of the piece.

MCTS implementation notes

Guidelines

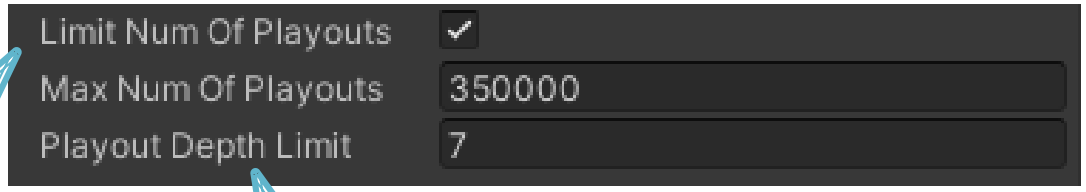
For tree policy, use UCB.

In the Expansion step, pick actions starting from the last one.

When performing simulations, first clone the state in the node using the Board.GetLightWeightClone method and use the lightweight clone.

When evaluating states, use rewards 0 = loss, 1 = win, Evaluation.EvaluateSimBoard() for anything else. (Alternatively, you can use different rewards if you find it easier, but remember to adjust UCB and the evaluation function accordingly in order to achieve the same results.)

MCTS Settings



If set to true, the number of playouts that the algorithm can perform is limited by the following parameter.

The maximum depth of the playouts.

(The values used here are taken from one of the tests – for an actual game, something like 70 000 and 100 might be better.)

Caveats

The move generators use some tricks, so you aren't actually getting the vanilla version of MCTS, but since you aren't implementing those, it doesn't really matter. Just be aware that your algorithm will perform somewhat better than the basic version of MCTS would.

As the lightweight clone of the current state is just a 2D array, it has no special methods, so there is no method for applying moves to it. However, as doing so just requires changing the position of one piece according to the data in a SimMove object, I thought I could leave this for you to implement where you see fit.

Caveats

When implementing MCTS, you don't need to fill out the bestEval variable. AlphaBetaSearch uses this variable for logging, so I left it there but you can safely ignore it if you want to.

After the search finishes, the content of the bestMove variable is passed to the GameManager, so make sure it is filled out whenever the search terminates (possibly by updating it every iteration).

The diagnostics class is present for logging purposes (to see how it's used, look at AlphaBetaSearch). Again, it can be useful for debugging stuff but feel free to ignore it. If you *are* going to use it, alter its contents however you want.

Resources

Resources

https://www.dailychess.com/chess/chess-fen-viewer.php

http://www.ee.unb.ca/cgi-bin/tervo/fen.pl

https://www.sparkchess.com/mate-in/five

https://www.chesspuzzles.com/mate-in-two

Resources

https://www.dailychess.com/chess/chess-fen-viewer.php

http://www.ee.unb.ca/cgi-bin/tervo/fen.pl

https://www.sparkchess.com/mate-in/five

https://www.chesspuzzles.com/mate-in-two

Allows you to convert board positions to FEN notation.

Resources

https://www.dailychess.com/chess/chess-fen-viewer.php

http://www.ee.unb.ca/cgi-bin/tervo/fen.pl

https://www.sparkchess.com/mate-in/five

https://www.chesspuzzles.com/mate-in-two

Allows you to convert FEN notation to board positions.

Resources

https://www.dailychess.com/chess/chess-fen-viewer.php

http://www.ee.unb.ca/cgi-bin/tervo/fen.pl

https://www.sparkchess.com/mate-in/five

https://www.chesspuzzles.com/mate-in-two

Offers some more chess puzzles in case you need them.

Resources

https://www.dailychess.com/chess/chess-fen-viewer.php

http://www.ee.unb.ca/cgi-bin/tervo/fen.pl

https://www.sparkchess.com/mate-in/five

https://www.chesspuzzles.com/mate-in-two

Even more puzzles.

End Notes

End Notes

Once you're done, send me your solution to my e-mail: guba@ksvi.mff.cuni.cz. DO NOT TRY TO SEND ME THE ENTIRE PROJECT, just the scripts that you have changed/added.

There is no strict deadline, however, you have to have all the assignments approved before attending the exam.

Don't forget to use UnityEngine.CoreModule.Mathf for math operations instead of System.Math.

Let me know if you encounter any bugs that I should fix or if you come up with some extensions of the project that would make working on the assignment easier.

Also, don't forget that you have one more week for the bonus homework mentioned in the previous lecture.

Faculty of Mathematics and Physics
Charles University
23.11.2022

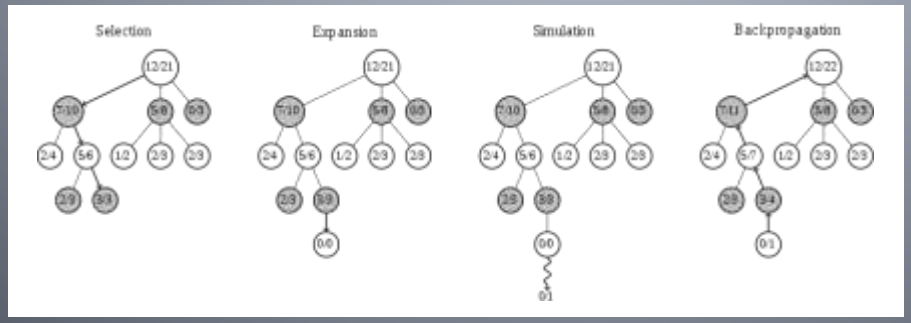
GAME DEVELOPMENT @CUNI.CZ

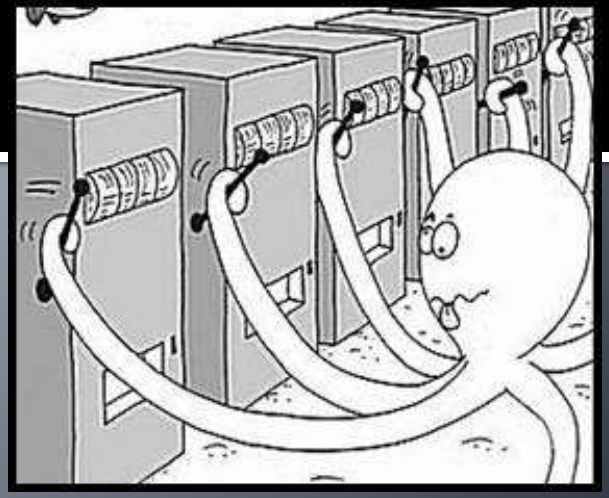
matfyz

NAIL122 AI for Games

Monte Carlo Tree Search

Jakub Gemrot, Peter Guba





Motivation

Motivation

Why should you care?

Motivation

Why should you care?

Motivation

Why should you care?

Motivation

Why should you care?

Motivation

Why should you care?

An intuitive explanation

MCTS – High-level description

MCTS
High-level description

MCTS iteration

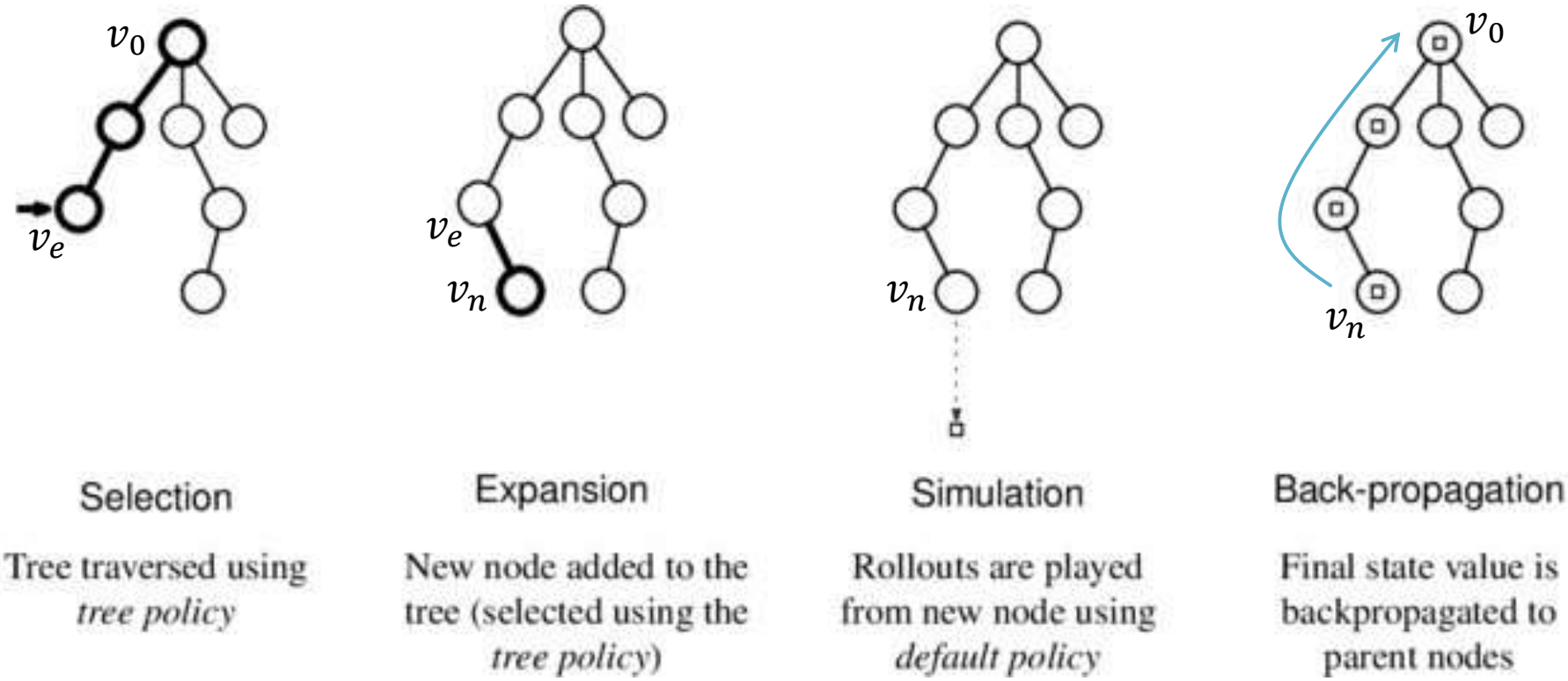


Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS
High-level description

MCTS iteration

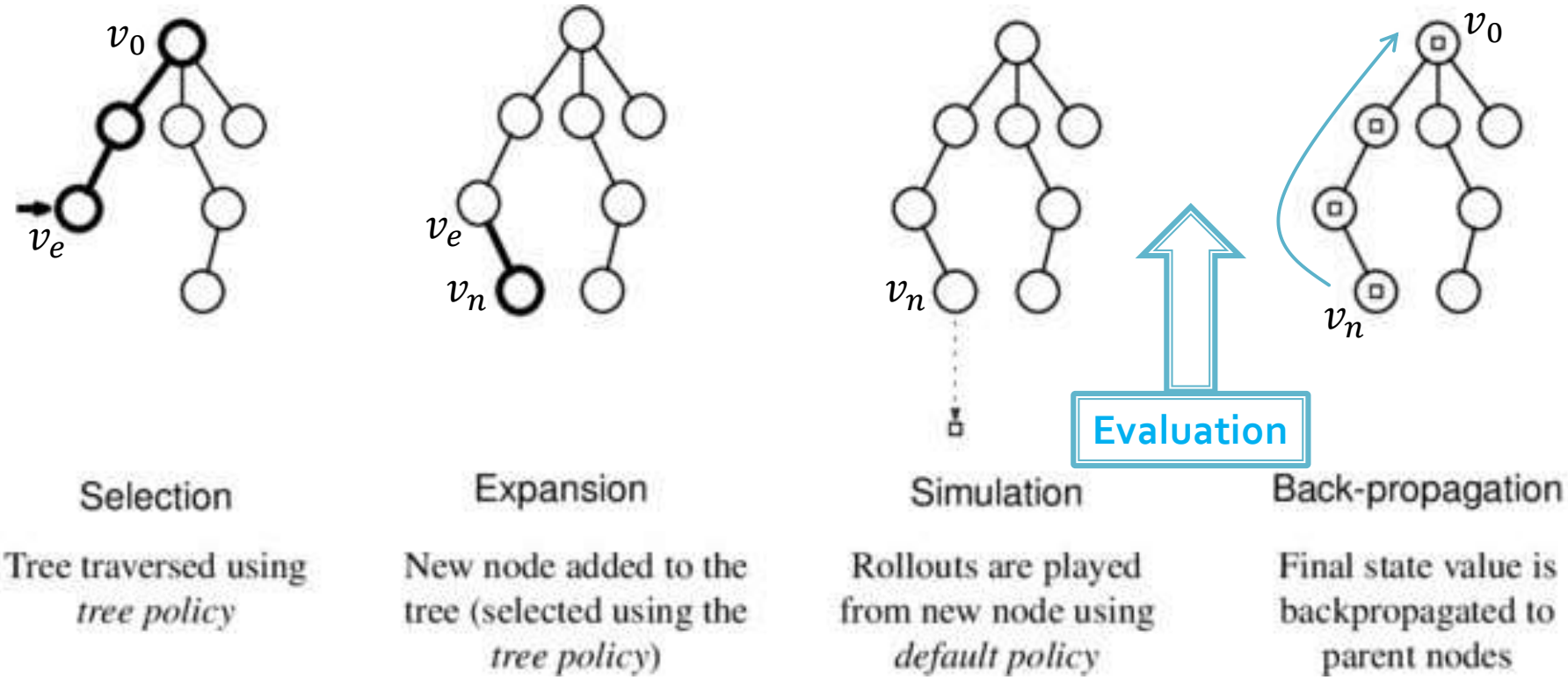


Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS
High-level description

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

MCTS iteration

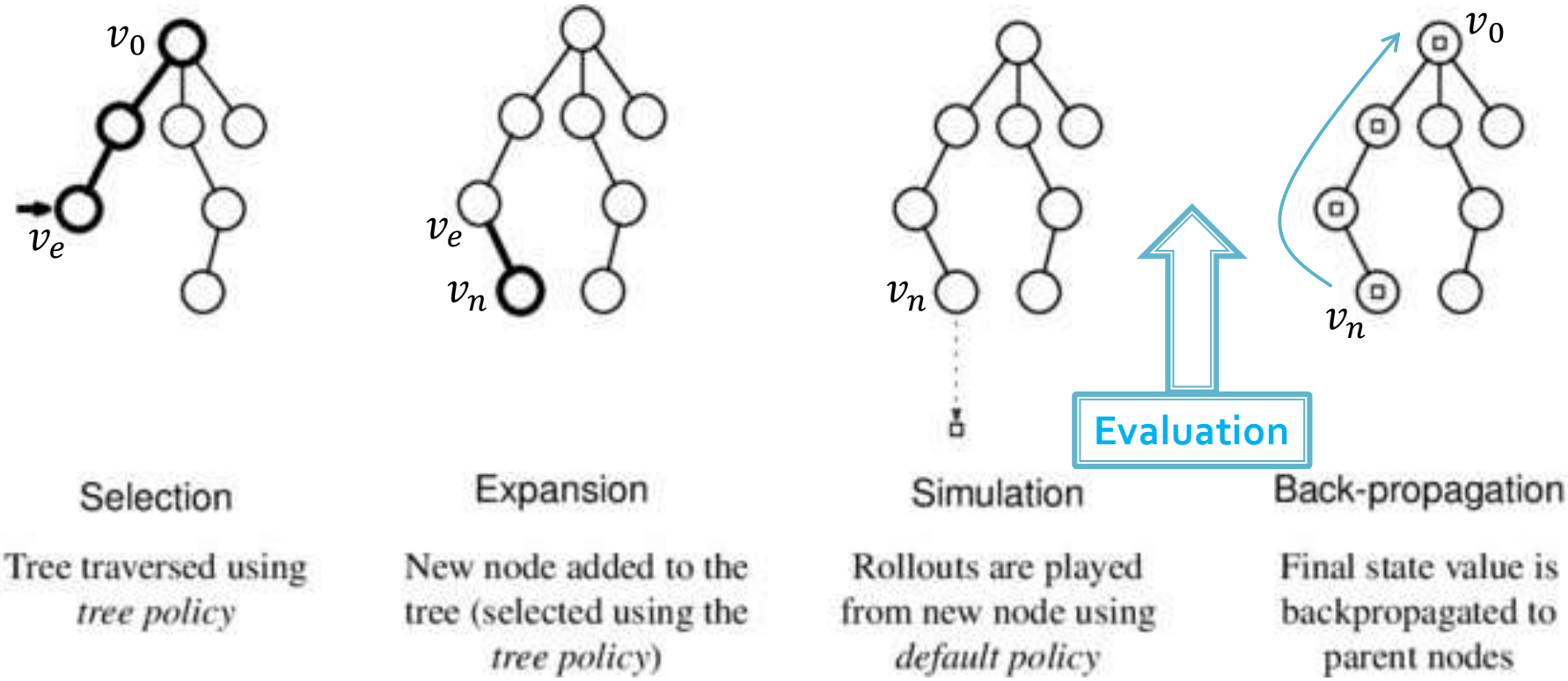


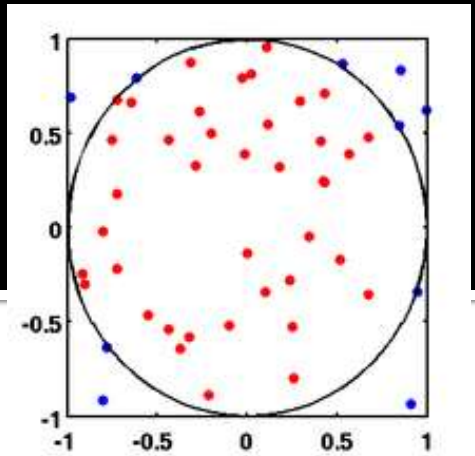
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

Monte Carlo Tree Search Roots

Monte Carlo Method

Monte Carlo Method

Approximating integrals



Estimating π

```
int i, throws = 99999, insideCircle = 0;
double randX, randY, pi;

srand(time(NULL));

for (i = 0; i < throws; ++i) {
  randX = rand() / (double) RAND_MAX;
  randY = rand() / (double) RAND_MAX;
  if (randX * randX + randY * randY < 1)
    ++insideCircle;
}

pi = 4.0 * insideCircle / throws;
```

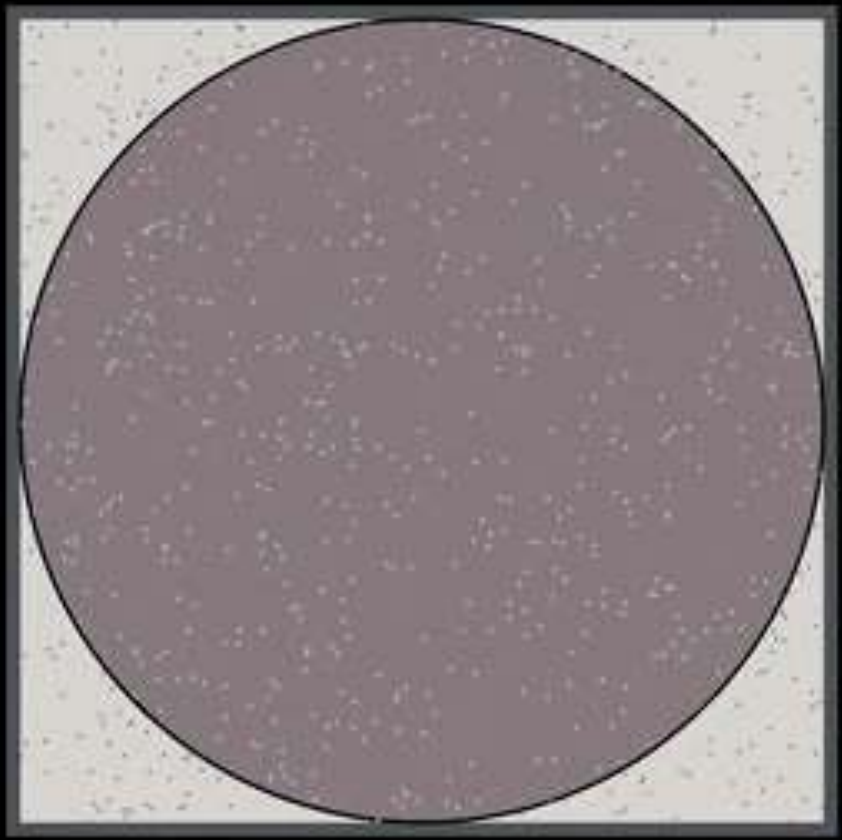
$$\frac{\pi}{4} = \frac{points\ inside\ circle}{points\ in\ square}$$

We are assuming uniform distribution of points $X_i = (randX_i, randY_i)$

Source: https://en.wikipedia.org/wiki/Monte_Carlo_integration

$$\pi \approx 4 \times \frac{\text{points within a circle}}{\text{total number of points}}$$

$$3.034285 = 4\left(\frac{531}{700}\right)$$



Monte Carlo Integration

Approximating integrals

When integrating f(x) is hard

$$I = \int_{\Omega} f(x)\, dx$$

Integral we want to estimate.

$$X_i \sim p(x)$$

Random variable following some distribution, e.g., uniform.

$$I \approx F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

Integral estimation using N points.

Source: https://en.wikipedia.org/wiki/Monte_Carlo_integration

Monte Carlo Tree Search Roots

Multi-armed bandit problem

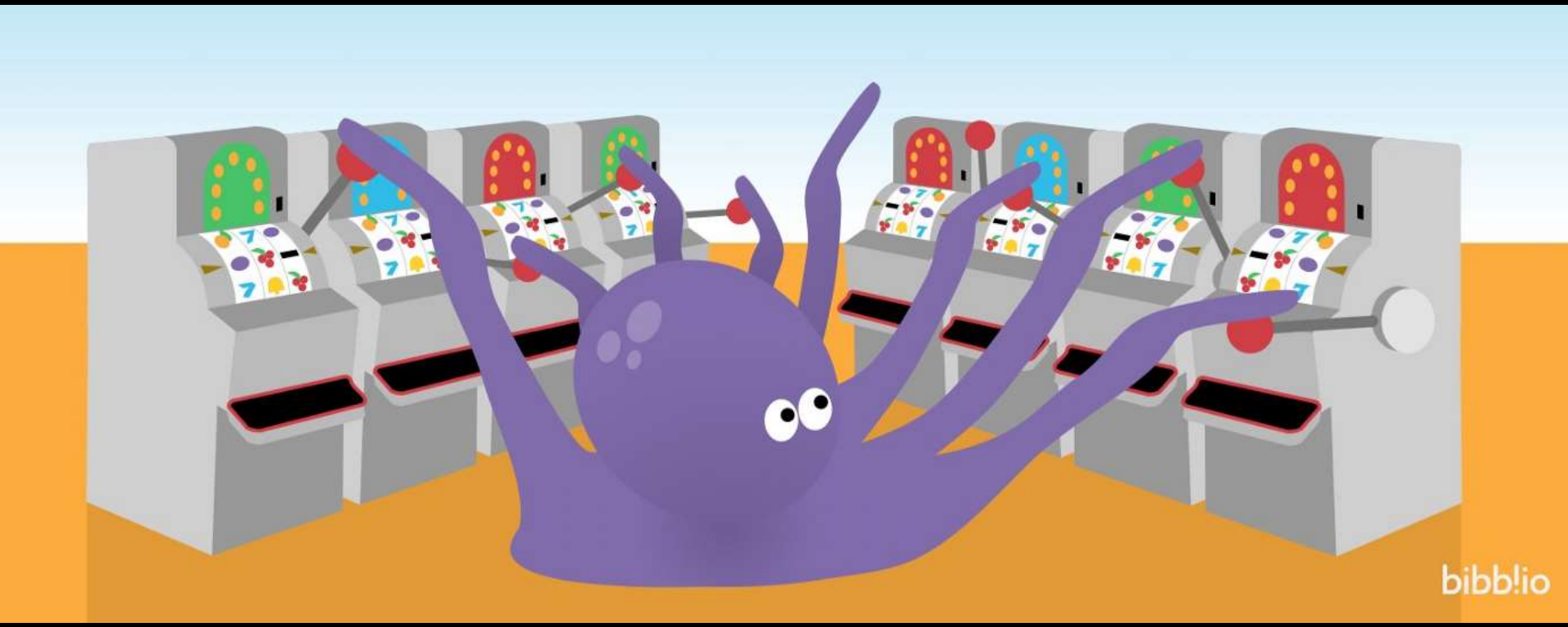


Image source: https://blog.lightningai.com/multi-armed-bandits-are-the-new-a-b-tests-27dd7b48765b?gi=fd1a99234662
Great tutorial chain: https://towardsdatascience.com/multi-armed-bandits-part-1-b8d33ab80697

Multi-armed Bandit

How to gamble properly

We kinda expect to have **a simulator** we can interact with in order to solve the problem before we pull an arm in real-life.

Multi-armed bandit game

There are K slot-machines, each having an arm.
You can choose an arm and pull it, i.e. you have $A = \{a_1, \dots, a_k\}$.
Choosing a slot-machine i and executing a_i will give you a reward $R_i(a_i)$.
Each reward $R_i(a_i)$ follows some unknown probability distribution.

The objective – Probably Approximately Correct (PAC) solution

Design an optimal policy π for the player in order to uncover an arm with the best expected reward.

That is, in the end, select an arm that probably (w/ high probability) has approximately the best expected reward.

Additional criterium

Use as fewest actions as possible to fulfill the objective.

How to select which arm to pull?

Upper Confidence Bounds

Regret minimization bandit problem:

Multi-armed bandit problem, but this time we are given N moves, after which we have to select an arm that is close to the best possible.

NaiveBandit solution (dividing arm pulls equally) is a poor choice

⇒ It is wasting time on arms, which seems to be bad already

The problem

How to balance exploration and exploitation, i.e., how to balance exploitation of good arms with exploration of poorly understood arms, in order to minimize the regret?

Regret: $\rho_n = \mu^* N - \sum_{i=1}^{|A|} \mu_i n_i$

where μ^* is the best expected payoff, μ_i is expected payoff of arm *i* and n_i is number of pulls made on arm *i*.

How to select which arm to pull?

Upper Confidence Bounds

Upper confidence bounds (UCB) adaptive bandit algorithm

For each arm (action) a_i maintain:

$Q(a_i)$ mean of observed rewards after pulling a_i

n_i number of performed pulls

Also track:

n number of total pulls so far

And for each arm (action) a_i compute:

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$

c is confidence value that controls the explore/exploit. To be hand-tuned for a given problem at hand.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

How to select which arm to pull?

Upper Confidence Bounds

Upper confidence bounds (UCB) adaptive bandit algorithm

For each arm (action) a_i maintain:

$Q(a_i)$ mean of observed rewards after pulling a_i

n_i number of performed pulls

Also track:

n number of total pulls so far

And compute:

UCB

In MCTS, this will correspond to the number of pulls in the parent node, not the overall total number of pulls!!!

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

How to select which arm to pull?

Upper Confidence Bounds

Upper confidence bounds (UCB) adaptive bandit algorithm

Tracked values:

$Q(a_i)$ mean of observed rewards after pulling a_i

n_i number of performed pulls

n number of total pulls so far

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$; always pull arm with the highest UCB

Exploitation part

Exploration part; note that at the beginning this has value $+\infty$! UCB will try to pull (explore) all arms once before resorting to exploitation.

UCB assumes payoffs to be in range $[0; 1]$! Otherwise, custom tweaks to c is required.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

How to select which arm to pull?

Upper Confidence Bounds

$$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$$

Where does this part come from?

How to select which arm to pull?

Upper Confidence Bounds

Chernoff/Hoeffding bound:
For iid samples $x_1 \ldots x_m \in [0, 1]$ with $\mathrm{E}[x_i] = \mu$,

$$P\left(\left|\frac{\sum_{i=1}^{m} x_i}{m} - \mu\right| \geq \delta\right) \leq 2e^{-2m\delta^2}$$

for all $\delta > 0$.

How to select which arm to pull?

Upper Confidence Bounds

Chernoff/Hoeffding bound:
For iid samples $x_1 \ldots x_m \in [0, 1]$ with $\mathrm{E}[x_i] = \mu$,

$$P\left(\left|\frac{\sum_{i=1}^{m} x_i}{m} - \mu\right| \geq \delta\right) \leq 2e^{-2m\delta^2}$$

for all $\delta > 0$.

After substituting $\sqrt{\frac{\ln n}{n_i}}$ for δ and n_i for m, we get

$$P\left(\left|\frac{\sum_{i=1}^{n_i} x_i}{n_i} - \mu\right| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

How to select which arm to pull?

Upper Confidence Bounds

Chernoff/Hoeffding bound:
For iid samples $x_1 \ldots x_m \in [0,1]$ with $\mathrm{E}[x_i] = \mu$,

$$P\left(\left|\frac{\sum_{i=1}^{m} x_i}{m} - \mu\right| \geq \delta\right) \leq 2e^{-2m\delta^2}$$

for all δ

This is our mean observed reward, $Q(a_i)$.

After su and n_i for m, we get

$$P\left(\left|\frac{\sum_{i=1}^{n_i} x_i}{n_i} - \mu\right| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

Contains both the cases when $Q(a_i)$ is larger than μ and when it is smaller, so we can restrict ourseves to just one of these by getting rid of the absolute value without breaking the inequality.

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

$$P\left(\mu - Q(a_i) \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq 2e^{-2n_i\left(\sqrt{\frac{\ln n}{n_i}}\right)^2}$$

$$P\left(|Q(a_i) - \mu| \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

$$P\left(\mu - Q(a_i) \geq \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

$$P\left(\mu \geq Q(a_i) + \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(\mu \geq Q(a_i) + \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

$$P\left(\mu < Q(a_i) + \sqrt{\frac{\ln n}{n_i}}\right) > 1 - \frac{2}{n^2}$$

How to select which arm to pull?

Upper Confidence Bounds

$$P\left(\mu \geq Q(a_i) + \sqrt{\frac{\ln n}{n_i}}\right) \leq \frac{2}{n^2}$$

$$P\left(\mu < Q(a_i) + \sqrt{\frac{\ln n}{n_i}}\right) > 1 - \frac{2}{n^2}$$

This shrinks rapidly with increasing number of pulls, so the UCB formula provides an upper bound on the true mean value with a high degree of confidence (hence the name).

How to select which arm to pull?

Upper Confidence Bounds

Upper confidence bounds (UCB) adaptive bandit algorithm

$UCB(a_i) = Q(a_i) + c\sqrt{\frac{\ln(n)}{n_i}}$; if all arms have been pulled at least once, pull arm with the highest UCB (otherwise pull one of the arms that haven't been tried yet)

Why to pull an arm with the highest UCB?

Observations:

- Exploitation part is high, if the arm is good.
- Exploration part is high, if the number of tries is low.
- If $c \cong 1$ and rewards are not from $[0; 1]$ but $[0; n]$, exploration part could become negligible for bigger n.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, *47*(2), 235-256.

How to select which arm to pull?

Upper Confidence Bounds

More resources

https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f

- Other strategies (e.g. ε-greedy) described as well

https://medium.com/analytics-vidhya/multi-armed-bandit-analysis-of-upper-confidence-bound-algorithm-4b84be516047

- Excellent comparison on arms of high/low differences of payoff means
- High difference => UCB is good, low difference => ε-greedy is better

Monte Carlo Tree Search

Going step by step

MCTS
High-level description

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ←
Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration

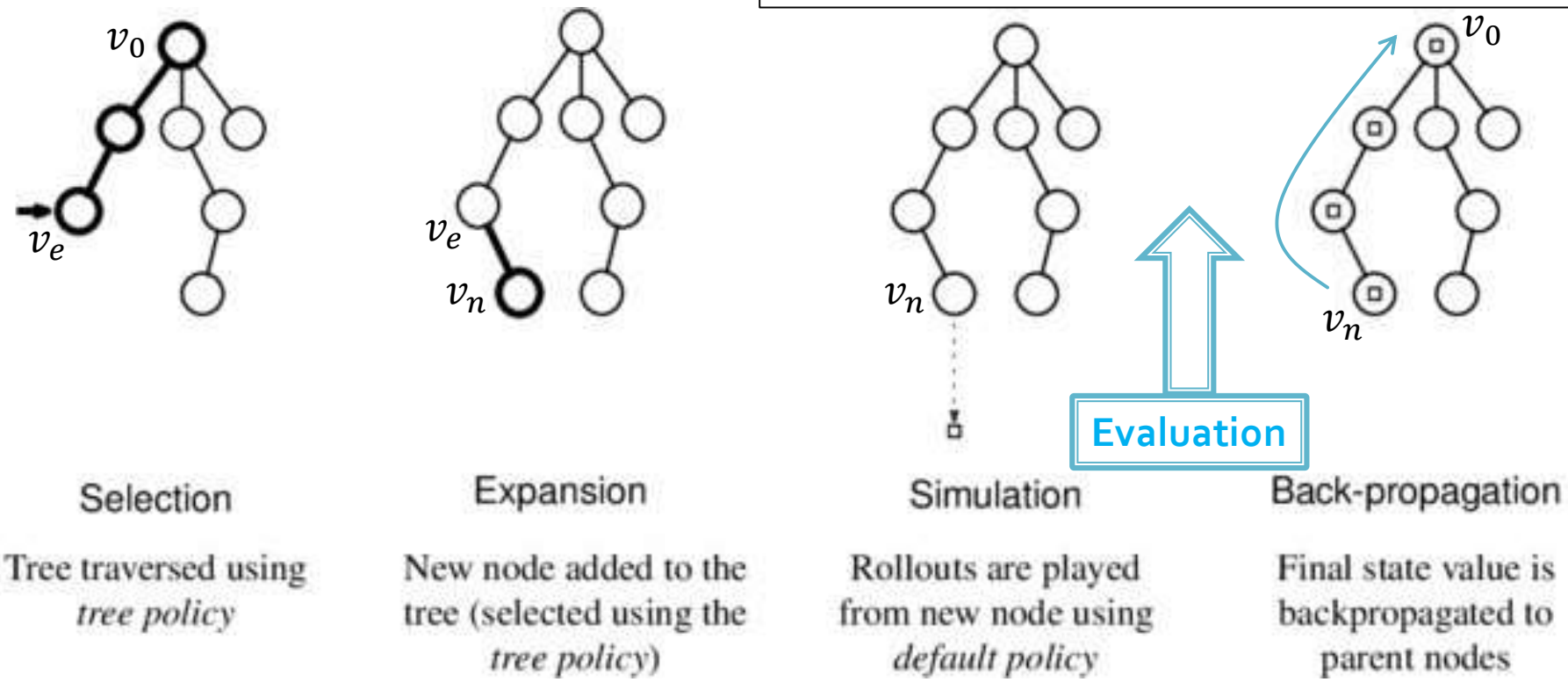


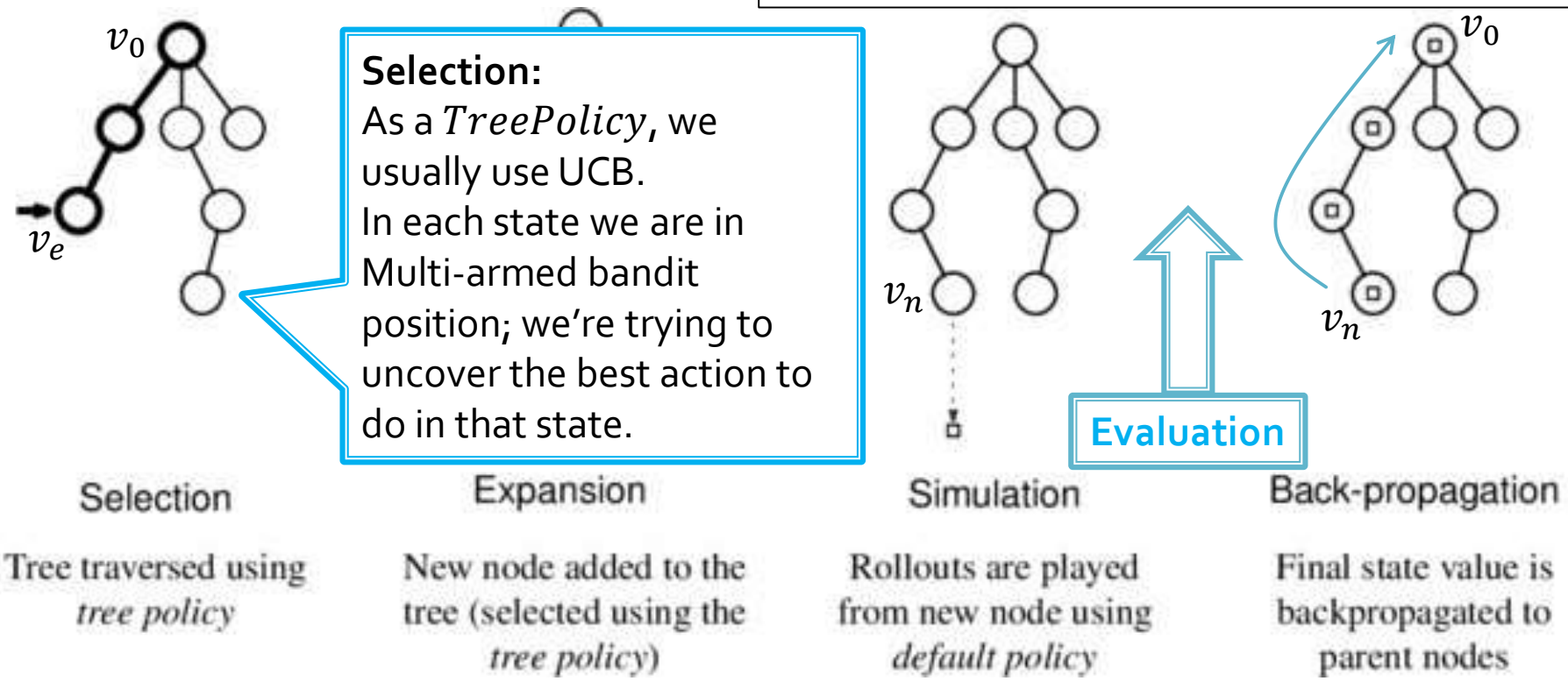
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Step 1: Selection

```
function MCTS(s_0)
    create root node v_0 with state s_0
    while has computation budget do
        v_e ← Select(v_0, TreePolicy)
        v_n ← Expand(v_e, ExpandPolicy)
        Δ ← Simulate(v_l.s, DefaultPolicy)
        BackPropagate(Δ, v_n)
    return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Selection:
As a $TreePolicy$, we usually use UCB.
In each state we are in Multi-armed bandit position; we're trying to uncover the best action to do in that state.

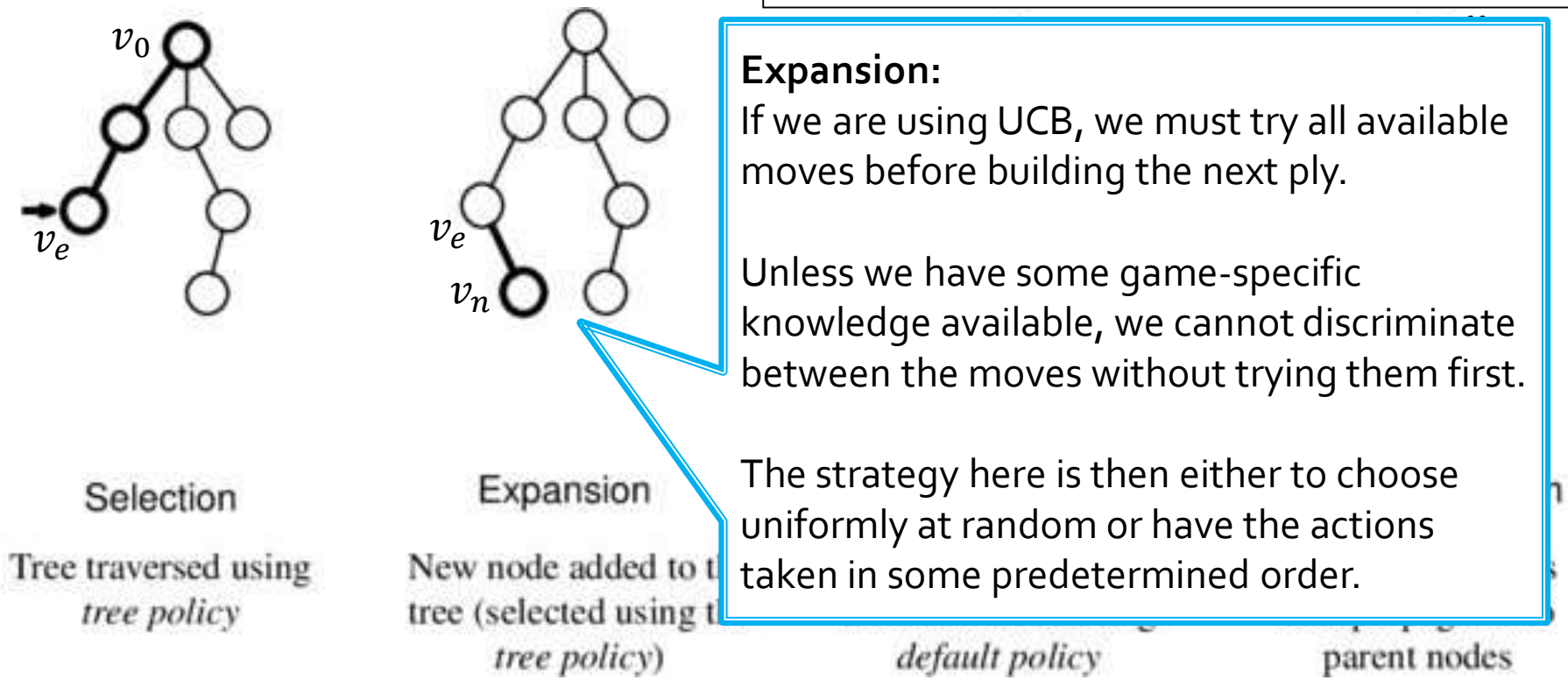
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Step 2: Expansion

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ←
Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Expansion:
If we are using UCB, we must try all available moves before building the next ply.

Unless we have some game-specific knowledge available, we cannot discriminate between the moves without trying them first.

The strategy here is then either to choose uniformly at random or have the actions taken in some predetermined order.

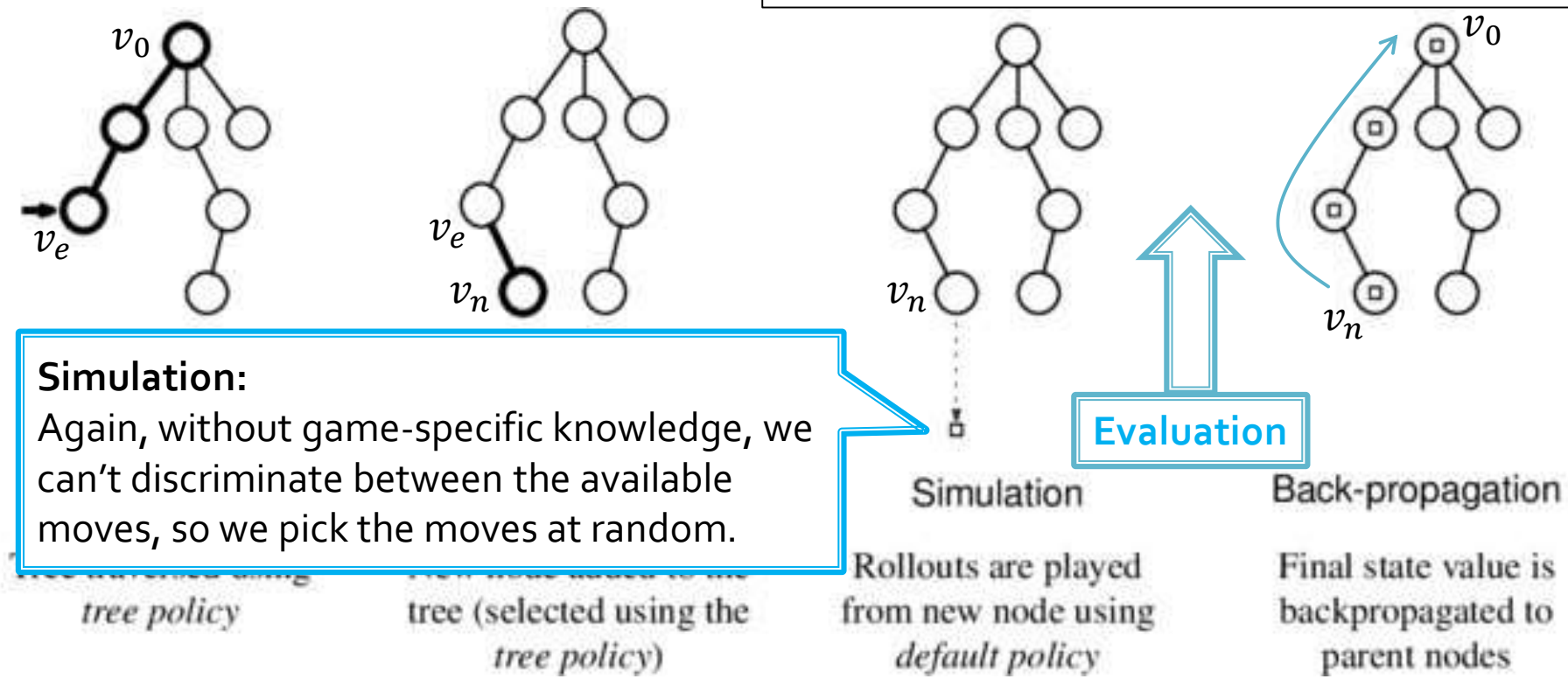
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Step 3: Simulation

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ←
Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Simulation:
Again, without game-specific knowledge, we can't discriminate between the available moves, so we pick the moves at random.

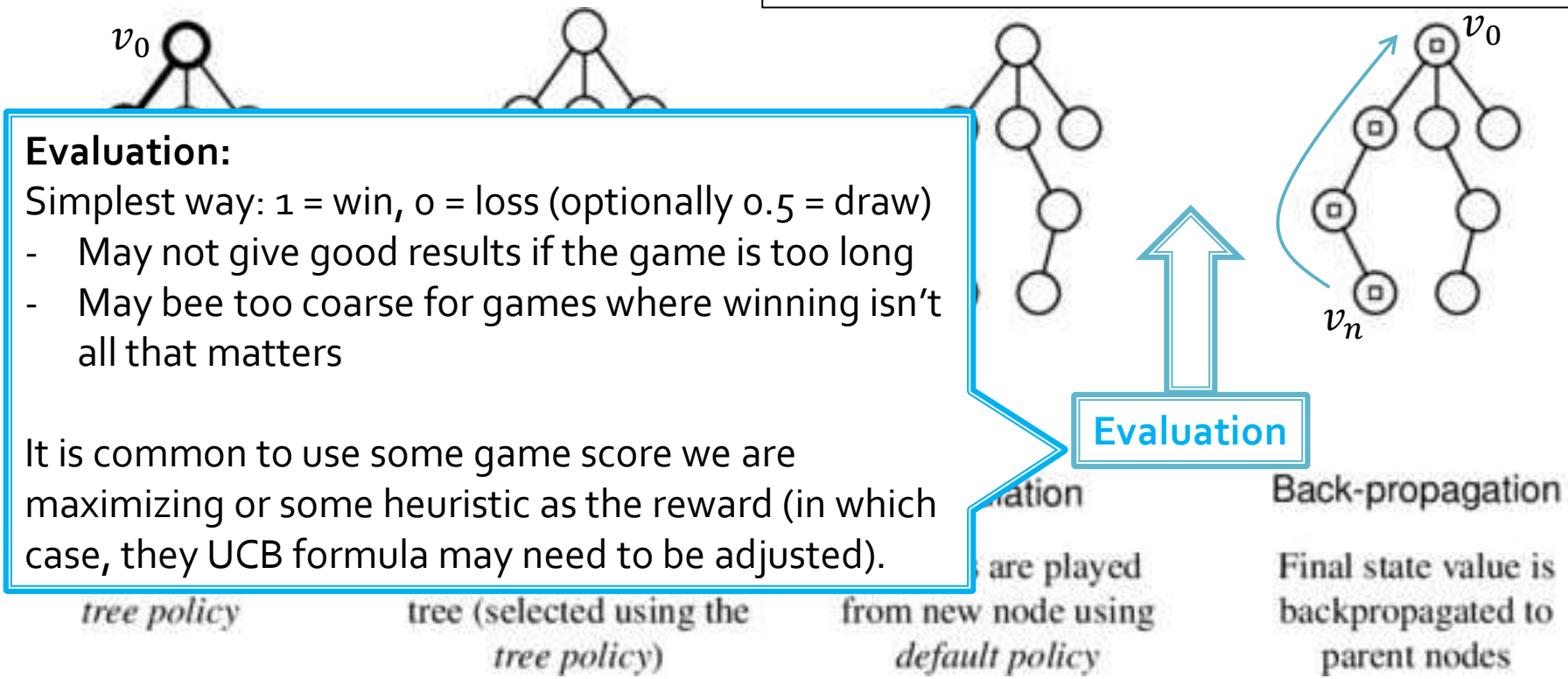
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Step 3.5: Evaluation

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ←
Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Evaluation:

Simplest way: 1 = win, 0 = loss (optionally 0.5 = draw)

- May not give good results if the game is too long
- May bee too coarse for games where winning isn't all that matters

It is common to use some game score we are maximizing or some heuristic as the reward (in which case, they UCB formula may need to be adjusted).

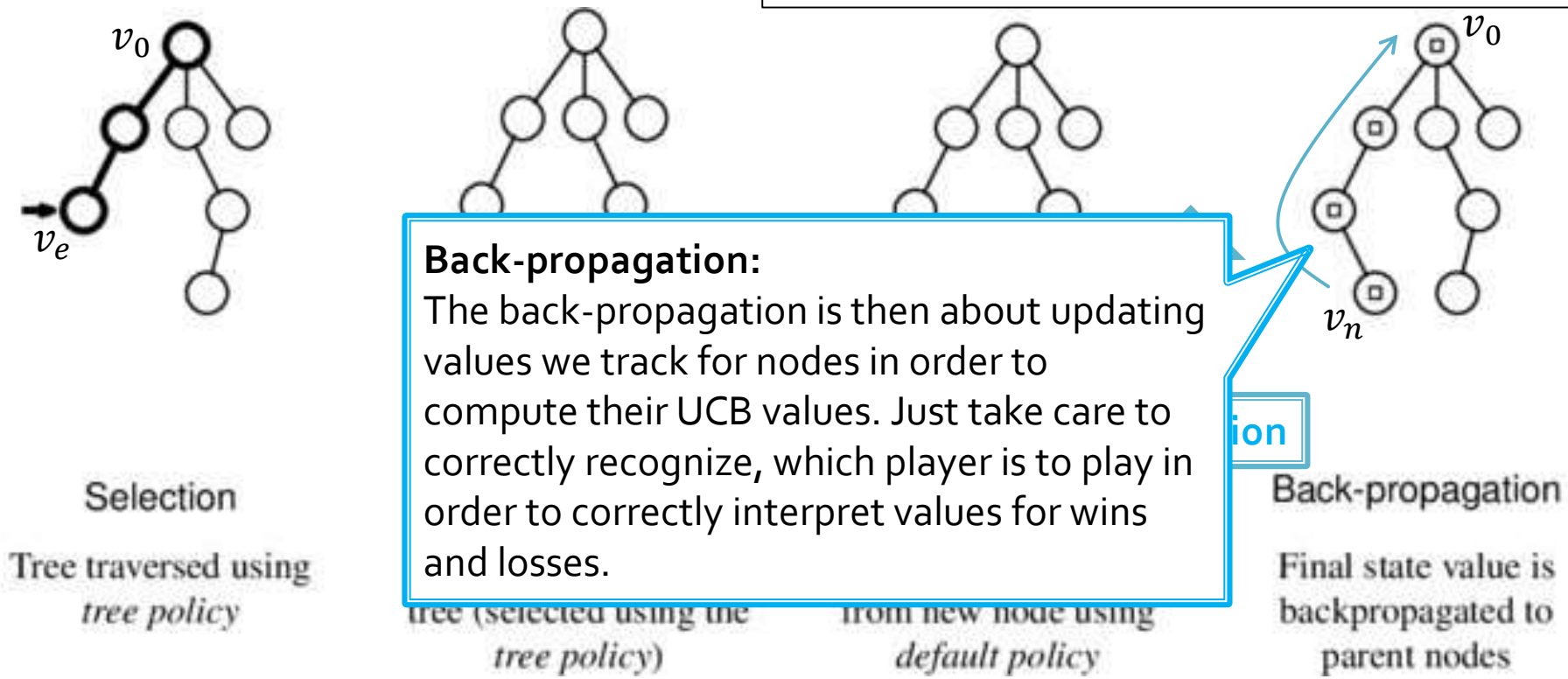
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Step 4: Backpropagation

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ←
Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration



Back-propagation:
The back-propagation is then about updating values we track for nodes in order to compute their UCB values. Just take care to correctly recognize, which player is to play in order to correctly interpret values for wins and losses.

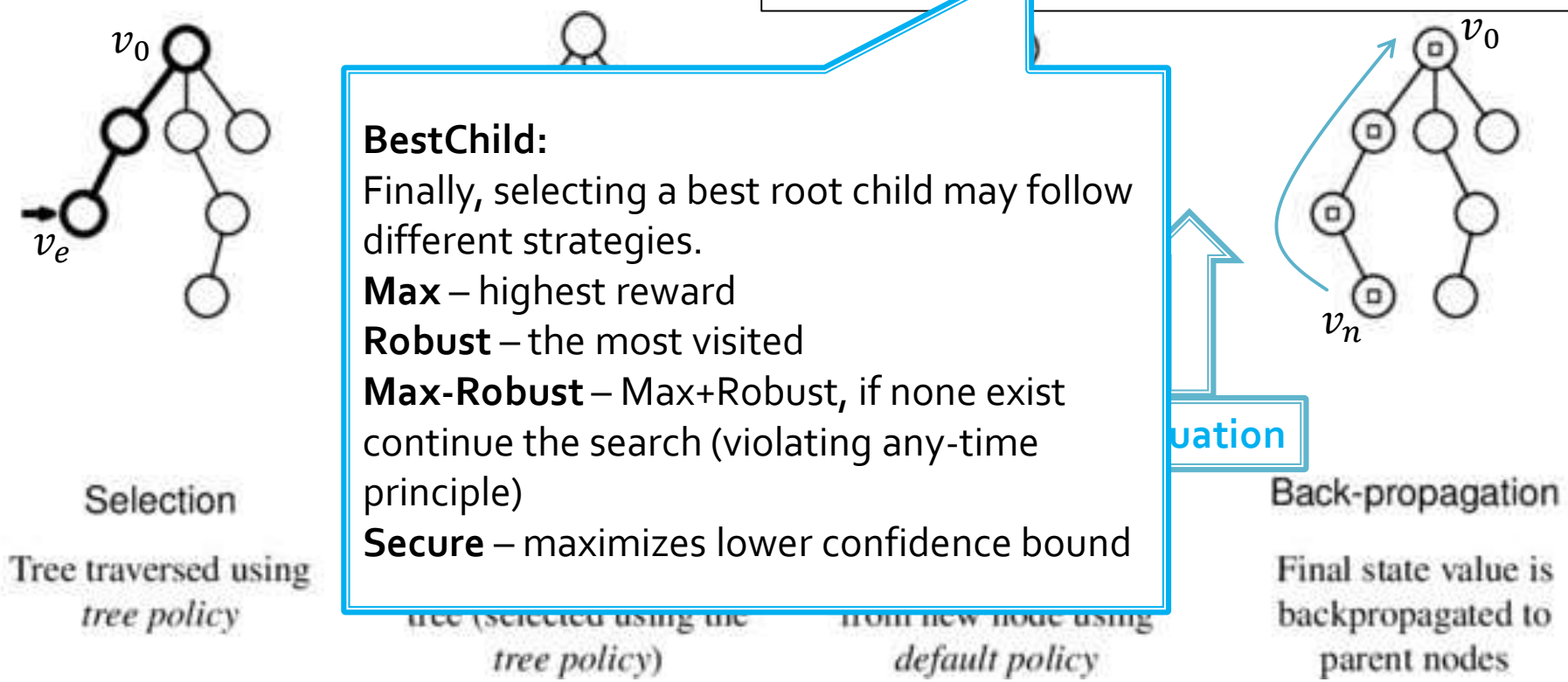
Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

MCTS

Choosing a move

```
function MCTS(s_0)
        create root node v_0 with state s_0
        while has computation budget do
                v_e ← Select(v_0, TreePolicy)
                v_n ← Expand(v_e, ExpandPolicy)
                Δ ← Simulate(v_l.s, DefaultPolicy)
                BackPropagate(Δ, v_n)
        return BestChild(v_0, BestPolicy).a
```

MCTS iteration



BestChild:
Finally, selecting a best root child may follow different strategies.
Max – highest reward
Robust – the most visited
Max-Robust – Max+Robust, if none exist continue the search (violating any-time principle)
Secure – maximizes lower confidence bound

Image adapted from: Santos, A., Santos, P. A., & Melo, F. S. (2017, August). Monte carlo tree search experiments in hearthstone. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 272-279). IEEE.

Monte Carlo Tree Search

End Notes

MCTS

Pros and Cons

Pros:

- Simple, anytime and asymmetric
- Doesn't require game-specific knowledge but can easily be augmented to take advantage of it
- Proven to converge to perfect play in infinity

Cons:

- Doesn't deal too well with large branching factors
- It may need to run for a large number of iterations which can be costly both in terms of memory and time
- Problem identifying trap states and making sacrifices

MCTS

Observations

MCTS is a heurisctic anytime asymmetric search.

For rewards from $[0; 1]$, set $c = \frac{1}{\sqrt{2}}$.

L. Kocsis, C. Szepesv´ari, and J. Willemson, “Improved Monte-Carlo Search,” Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

Probability of selecting wrong action converges to zero at polynomial rate as the number of games simulated grows to infinity; MCTS converges to minimax. L. Kocsis and C. Szepesv´ari, “Bandit based Monte-Carlo Planning,” in Euro. Conf. Mach. Learn. Berlin, Germany: Springer, 2006, pp. 282–293.

If domain-specific heuristic is available, minimax algorithm is still viable option.

UCB can be a poor choice if $|A|$ is high, due to the requirement of trying all actions in given state first; you might want to try $\varepsilon - greedy$ instead.

Simulating game till the end via random moves can be problematic. Easy for Go, harder for Chess, even harder for e.g. RTS combat (moving around will not resolve the combat).

Comparison to AlphaBeta: MCTS does not require domain-dependent state evaluation function, builds the search tree in
an asymetric way and can be terminated at any time.

MCTS

History till 2009

1990 Abramson demonstrates that Monte Carlo simulations can be used to evaluate value of state [1].
1993 Brugmann [31] applies Monte Carlo methods to the field of computer Go.
1998 Ginsberg's GIB program competes with expert Bridge players.
1998 MAVEN defeats the world scrabble champion [199].
2002 Auer et al. [13] propose UCB1 for multi-armed bandit, laying the theoretical foundation for UCT.
2006 Coulom [70] describes Monte Carlo evaluations for tree-based search, coining the term Monte
Carlo tree search.
2006 Kocsis and Szepesvari [119] associate UCB with tree-based search to give the UCT algorithm.
2006 Gelly et al. [96] apply UCT to computer Go with remarkable success, with their program MOGO.
2006 Chaslot et al. describe MCTS as a broader framework for game AI [52] and general domains [54].
2007 CADIAPLAYER becomes world champion General Game Player [83].
2008 MOGO achieves dan (master) level at 9 9 Go [128].
2009 FUEGO beats top human professional at 9 9 Go [81].
2009 MOHEX becomes world champion Hex player [7].

Taken from: Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1-43.

MCTS

Literature and other resources

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, *4*(1), 1-43.

Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2021). Monte Carlo Tree Search: A Review of Recent Modifications and Applications. *arXiv preprint arXiv:2103.04931*.

Bandit algorithm and various strategies:
https://towardsdatascience.com/bandit-algorithms-34fd7890cb18#b390

UCB:
https://towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f

Bonus homework

Bonus Homework

(Completely voluntary)

Come up with an some improvement of this algorithm.

Your idea should be general, not game specific, however, you can assume that we have a heuristic for evaluating how good a state is, as that is pretty common.

No need to code anything, just write it down and send it to me (pseudocode is welcome though). My e-mail is guba@ksvi.mff.cuni.cz and I am also on Discord.

If I will find you ideas reasonable, they will shine a favourable light on you during the exam.

Deadline: 6.12.2022 23:59:59

That’s it for today!

Faculty of Mathematics and Physics
Charles University
16.11.2021

GAME DEVELOPMENT @CUNI.CZ

matfyz

NAIL122 AI for Games

Modeling RTS Combat Scenarios

Jakub Gemrot

How to tackle combat in RTS games?

We're going to build up knowledge (almost) ground up towards understanding:
Churchill, D., Saffidine, A., & Buro, M. (2012, October). Fast heuristic search for RTS game combat scenarios. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Modelling RTS combat

States, units and moves

The Problem

How to create combat AI for real-time strategy game? Whom to target, where to move in order to minimize own losses while maximizing damage to the opponent?

Modelling RTS combat

States, units and moves

The Problem

How to create combat AI for real-time strategy game? Whom to target, where to move in order to minimize own losses while maximizing damage to the opponent?

Psst.. It can be surprisingly hard even if AI controls 1 unit and faces the opposition of N enemy units!

Mathematical Games

(peeking into NDMI098 / NUMV090)

Games

Mathematically speaking

Informal Definition

Game is any set of circumstances that has a result dependent on the actions of two or more decision-makers (players).

Games

Mathematically speaking

Informal Definition

Game is any set of circumstances that has a result dependent on the actions of two or more decision-makers (players).

⇒ Differs from (deterministic) simulation.

In simulations, the outcomes are dependent on the given (initial) state and its rules only, in games, the result depends on the (initial) state and players choices.

In (deterministic) simulations, rules are fixed, they have single outcome, players have multiple choices and can affect the game's end.

Games

Mathematically speaking

Concepts of theoretical games

N	finite set of players
Actions	or Moves available to players at certain moments of the game
Nodes	moments of the game where players can perform actions (== make moves)
(Game) Tree	order of players' moves
Payoffs	gains (or losses) of players at game tree <u>leaves</u>
Information set	"fog of war", a set of nodes, which are indistinguishable for a given player, i.e., a given player lacks information to recognize what concrete (game) node from the set they are in right now
Nature	or Environment or Chance is a player $o \in N$ used to model randomness

Extensive form games

(peeking into NDMI098 / NUMV090)

Extensive form games

Describing the (Tic-tac-toe) game in its entirety

Tic-tac-toe game tree

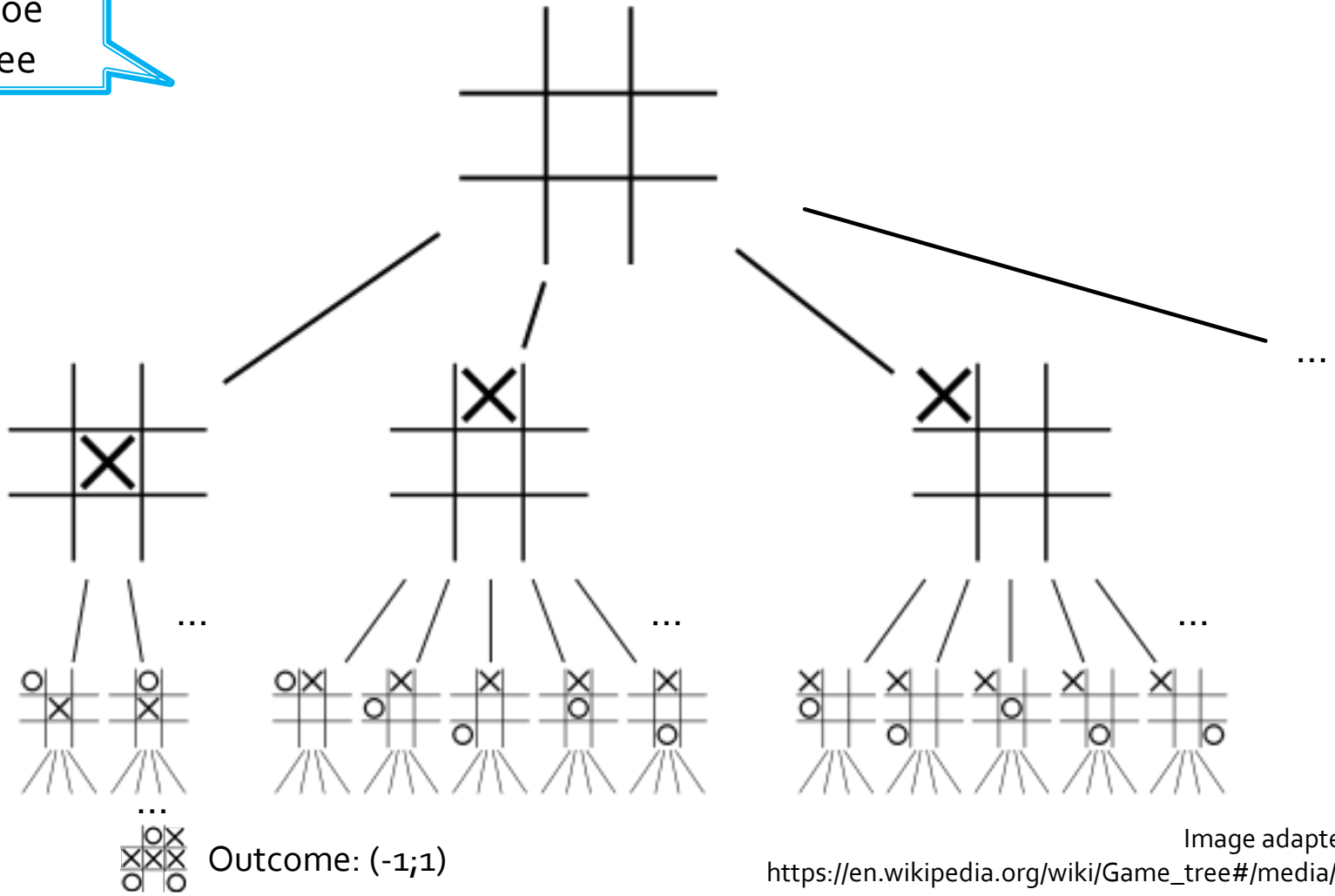
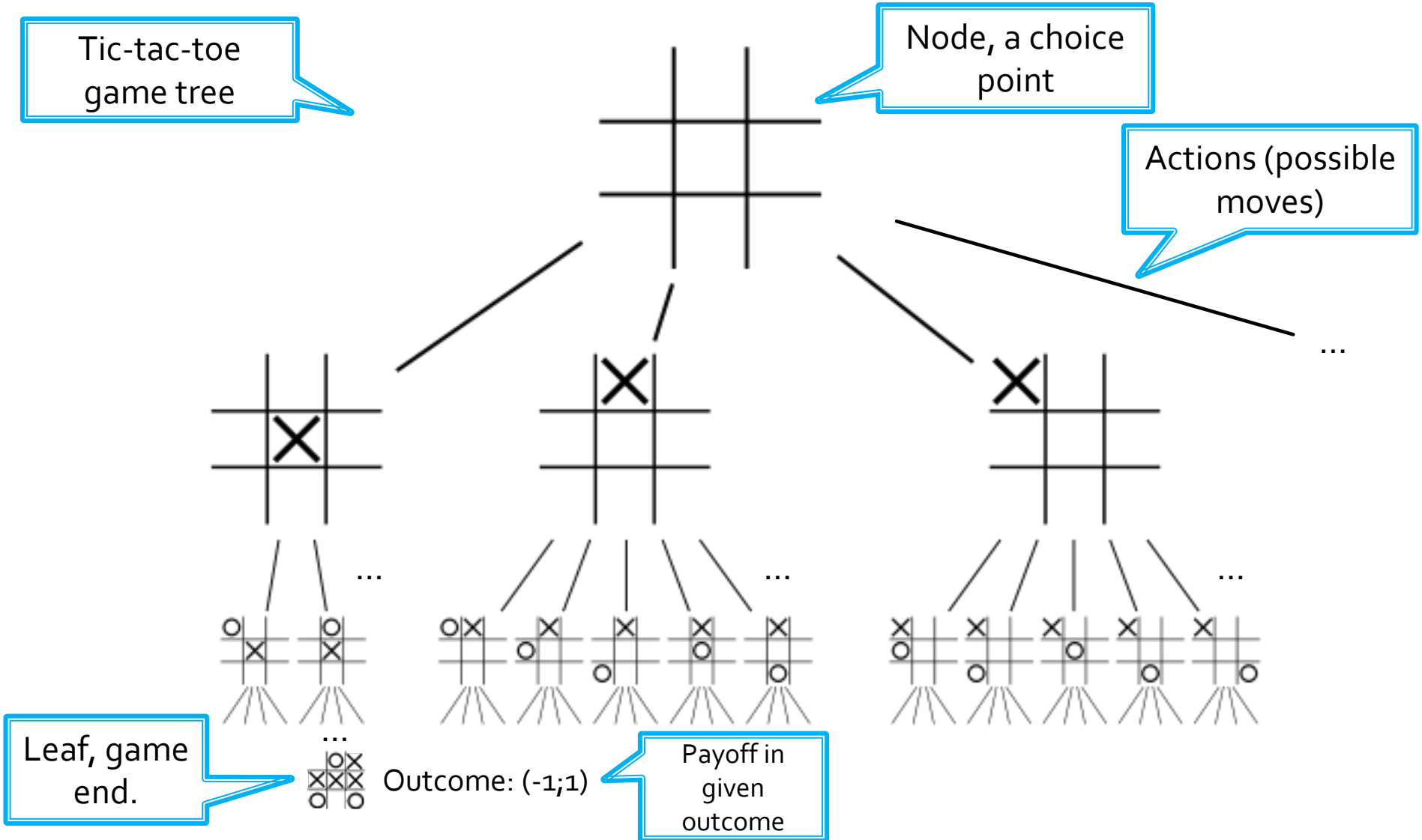


Image adapted from: https://en.wikipedia.org/wiki/Game_tree#/media/File:Tic-tac-toe-game-tree.svg

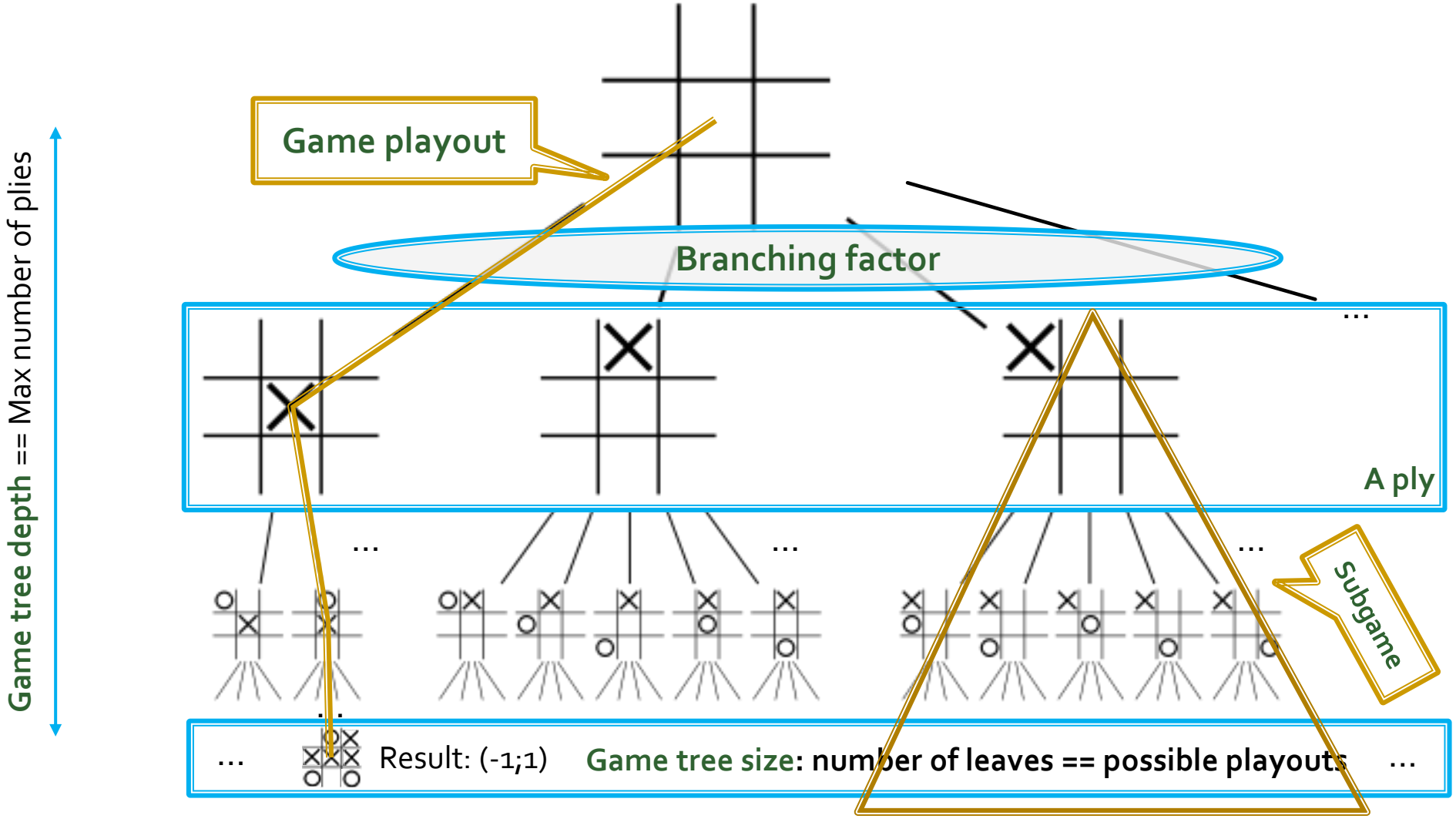
Extensive form games

Describing the (Tic-tac-toe) game in its entirety



Extensive form games

Describing the (Tic-tac-toe) game in its entirety



Games

Mathematically speaking

Main features of Games important to us

Sequential / Simultaneous
Perfect / Imperfect information

Additional type of Games

Zero-sum / Non-zero sum
Cooperative / Non-cooperative
Symmetric / asymmetric

Normal form games

(peeking into NDMI098 / NUMV090)

Normal form games

Theoretical background

Also called strategic form games

Concepts of theoretical games

Strategy	a complete „algorithm" for a player how to play the game
Strategy profile	set of strategies, one for each player
Pure strategy	a strategy, where the player has exactly one action planned for every game tree node where the player can make a move
Strategy set	all pure strategies available to a player to choose from
Mixed strategy	a probability distribution over a strategy set; a player chooses a strategy at the beginning of a game; associated with expected payoff
Behavior strategy	a strategy, where for any given information set of a player, we have a probability distribution defined over actions available in the information set
Dominated strategy	a strategy that is inferior in terms of payoffs to another strategy a player may select (<u>regardless other player actions</u>)

Normal form games

Theoretical background

Normal form game definition

$$\{Q; S_1, \dots, S_n; u_1(s_1, \dots, s_n), \dots, u_n(s_1, \dots, s_n)\}$$

Q list of players, indexed 1 .. N

S_i strategy set for a player *I*

u_i payoff function: $S_1 \times \cdots \times S_n \rightarrow R^n$
i.e., for strategies $s_i \in S_i$ chosen by players
the function returns players' payoffs $r_i \in R$
(+ … gains, - … losses)

Normal form games

Theoretical background

Normal form game definition

$$\{Q; S_1, \dots, S_n; u_1(s_1, \dots, s_n), \dots, u_n(s_1, \dots, s_n)\}$$

Q list of players, indexed 1 .. N

S_i strategy set for a player *I*

u_i payoff function: $S_1 \times \cdots \times S_n \rightarrow R^n$
i.e., for strategies $s_i \in S_i$ chosen by players
the function returns players' payoffs $r_i \in R$
(+ … gains, - … losses)

Many games and even social situations falls under the notion of normal form games!

Normal form games

Theoretical background

Normal form game matrix representation

Prisoners' dilemma

		Player 2	
		confess	don't confess
Player 1	confess	(-6, -6)	(0, -10)
	don't confess	(-10, 0)	(-1, -1)

Image source: https://www.thoughtco.com/the-prisoners-dilemma-definition-1147466

Nash equilibrium

(peeking into NDMI098 / NUMV090)

Nash equilibrium

Theoretical background

Informally speaking

Nash equilibrium is an outcome reached that, once achieved, means no player can increase payoff by changing decisions unilaterally.

In other words…

If each player has chosen a strategy, and no player can increase their own expected payoff by changing their strategy while the other players keep theirs unchanged, then the current strategy profile constitutes a Nash equilibrium.

Nash equilibrium

Theoretical background

Nash's Theorem

Every finite, non-cooperative game for two or more players is having a Nash equilibrium either in pure or mixed strategies.

In other words…

If the proceeding of the game is in Nash equilibrium, then even if I know, what strategies others are playing, I cannot do better then to stick with strategy I'm currently playing.

Normal form games

Theoretical background

Pure strategy Nash equilibrium example

Prisoners' dilemma

		Player 2	
		confess	don't confess
Player 1	confess	(-6, -6)	(0, -10)
	don't confess	(-10, 0)	(-1, -1)

Image source: https://www.thoughtco.com/the-prisoners-dilemma-definition-1147466

Normal form games

Theoretical background

Pure strategy Nash equilibrium example

Prisoners' dilemma

Pure (Nash) equilibrium

		Player 2	
		confess	don't confess
Player 1	confess	(-6, -6)	(0, -10)
	don't confess	(-10, 0)	(-1, -1)

Image source: https://www.thoughtco.com/the-prisoners-dilemma-definition-1147466

Normal form games

Theoretical background

Mixed strategy Nash equilibrium example

Rock paper scissors

		Player 2		
		Rock	Paper	Scissors
Player 1	Rock	0, 0	-1, 1	1, -1
	Paper	1, -1	0, 0	-1, 1
	Scissors	-1, 1	1, -1	0, 0

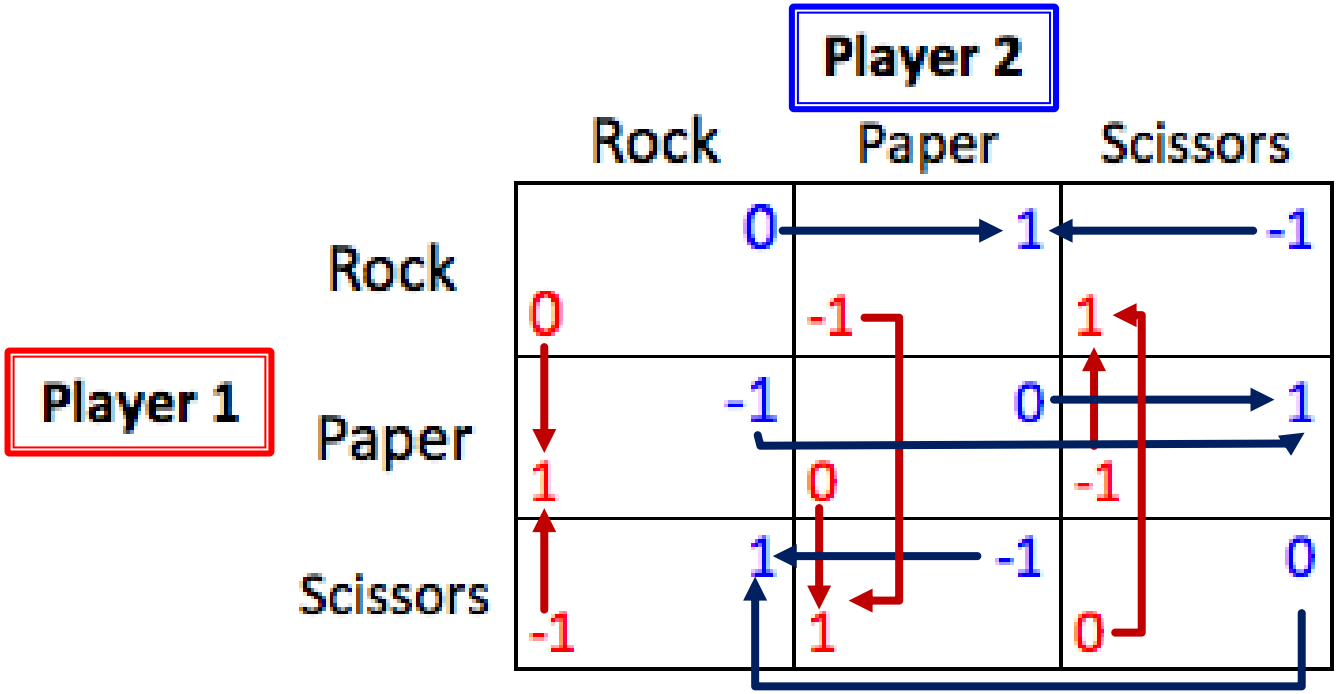
Image source:
https://upload.wikimedia.org/wikipedia/commons/c/cb/Rockpaperscissorspayoff.png

Normal form games

Theoretical background

Mixed strategy Nash equilibrium example

Rock paper scissors



Mixed strategy is a probability distribution over a strategy set; a player chooses a strategy at the beginning of a game; associated with expected payoff

Normal form games

Theoretical background

		Player 2		
		Rock	Paper	Scissors
Player 1	Rock	0, 0	-1, 1	1, -1
	Paper	1, -1	0, 0	-1, 1
	Scissors	-1, 1	1, -1	0, 0

Mixed strategy Nash equilibrium example

Rock paper scissors – Linear programming

P2 (expected) utilities for different P1 strategies

P1 plays rock: $u = r.0 + p.1 + s.-1 = p - s$

P1 plays paper: $u = r.-1 + p.0 + s.1 = s - r$

P1 plays scissor: $u = r.1 + p.-1 + s.0 = r - p$

And we have probability distribution: $r + p + s = 1$

Now P1 is trying to minimize P2 utilities, so we have
$u < p - s; u < s - r; u < r - p$

P2 is trying to maximize the utility, which yields result $r = p = s = \frac{1}{3}$

Battle of Sexes

And subgame perfection
And backward induction
(Sort of "maxmax" algorithm…)

Battle of Sexes

The Game

Normal form of the game

A boy and a girl are spending night together, though they forgot, where they agreed to meet. The boy would like to go and see a football match, whereas the girl would like to go to the opera.

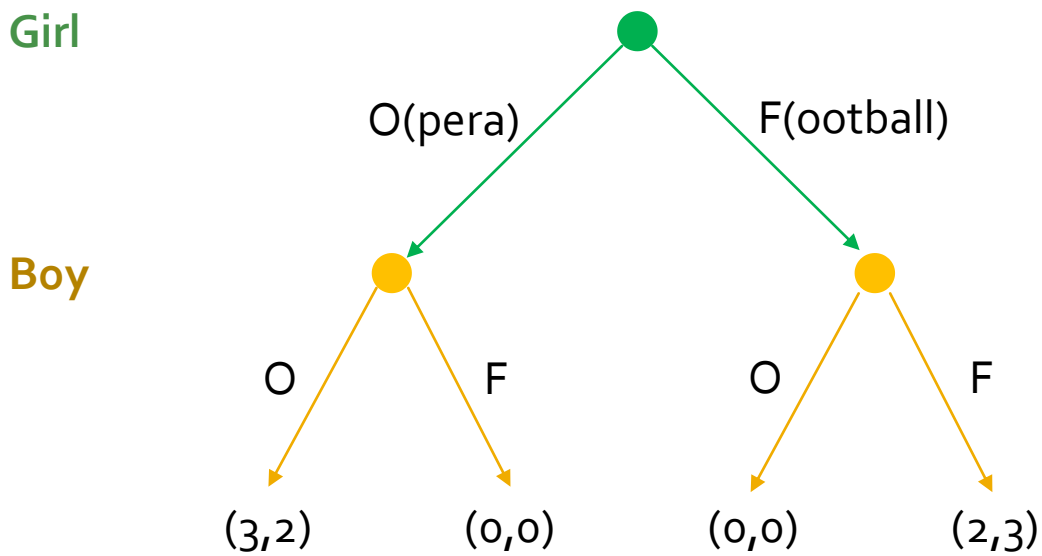
	Opera	Football
Opera	3,2	0,0
Football	0,0	2,3

Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she going. The simultaneous game turns into sequential.

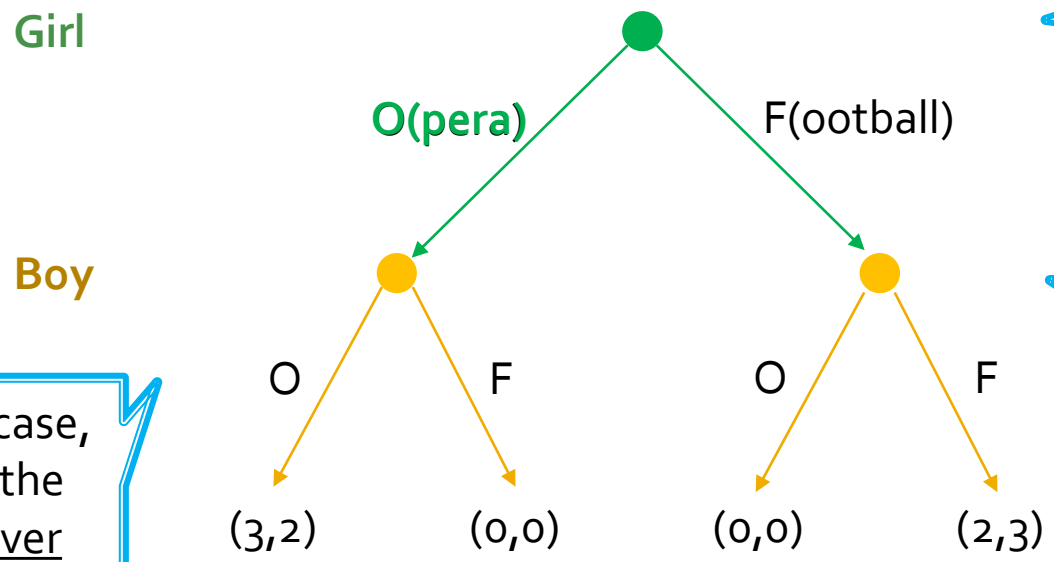


Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she going. The simultaneous game turns into sequential.



2. Knowing that and applying backward induction, the girl knows what to do ☺

1. In each subgame, the boy (if rational) has to adopt "copy" strategy. The subgame perfection.

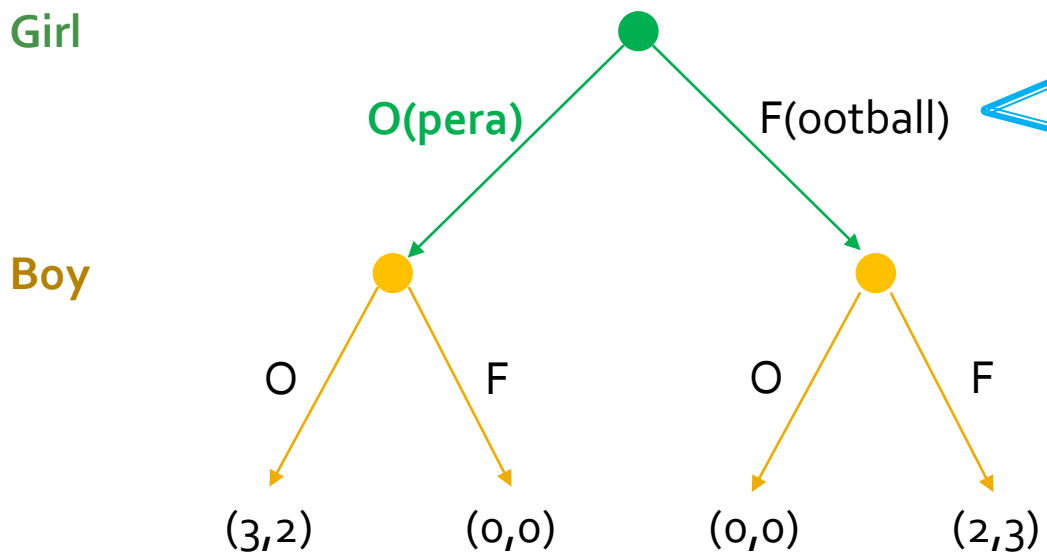
3. In this case, there is the first mover advantage.

Battle of Sexes

The Game

Altered game

Suppose the girl has a chance to text the boy where is she going. The simultaneous game turns into sequential.



4. Finally, perfect information simultaneous games are (sometimes) referred to as **stacked matrix games**.

5. That's because each subgame can be described using a normal form, i.e., the payoff matrix.

Back to RTS combat! … Sort of.

Suikoden, Konami (1995), game featuring classical JRPG combat system

Before taking on RTS combat, we're going to check on: Attrition games played on graphs

Furtak, T., & Buro, M. (2010, October). On the complexity of two-player attrition games played on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 5, No. 1).

Attrition games on graphs

And its intricacies

Attrition Game on Graph (AGG)

Two-player perfect information simultaneous game on a graph.
Every node belongs to a single player (either white and black here)
Each node described by $\langle h; a \rangle$

h	…	health
a	…	attack value

Graph is directed, an $x \to y$ edge means x can attack y.
Discrete version: all nodes choose their target, then simultaneously attack, all nodes with zero or lower health are removed.
Players may have various objectives (e.g. minimize damage taken).

Attrition games on graphs

And its intricacies

1 vs N units; minimize dmg taken

White: 1 unit$\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle, \ldots, \langle h_n; a_n \rangle$
White objective: minimize damage taken by its unit

Theorem

In discrete version, to minimize white's total sustained damage it is sufficient to order its targets by decreasing value of $a_i / \lceil h_i / a_0 \rceil$ and never change targets until they have been destroyed.

Intuitively, white wants to neutralize a threat with high attack value and low health while not over overkilling it.

Proof by contradiction, order units, try to swap two black units j, j+1 (check the paper for details).

Attrition games on graphs

And its intricacies

1 vs N units; white is maximizing reward (AGG-1:N-Rew)

White: 1 unit $\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle, \ldots, \langle h_n; a_n \rangle$
White objective: maximize reward from neutralizing black units; each black unit associated with reward $r_i \geq 0$.

Theorem

Given a discrete AGG scenario with n black units with health h_i, attack a_i, and kill reward $r_i \geq 0$ for white, and a single white unit with health h_o and attack a_o, it is <u>NP-hard</u> for white to decide what the reward-maximal target ordering is, in case white <u>does not survive</u>.

Attrition games on gr

And its intricacies

1. In SC or AoE, consider facing multiple enemies of different kinds including special power units.

1 vs N units; white is maximizing reward (AGG-1:N-Rew)

White: 1 unit $\langle h_0; a_0 \rangle$; Black: N units $\langle h_1; a_1 \rangle, \ldots, \langle h_n; a_n \rangle$
White objective: maximize reward fror
each black unit associated with reward

2. We will show how to encode the 0-1 knapsack problem as AG-1:N-Rew.

Theorem

Given a discrete AGG scenario with n black units with health h_i, attack a_i, and kill reward $r_i \geq 0$ for white, and a single white unit with health h_o and attack a_o, it is <u>NP-hard</u> for white to decide what the reward-maximal target ordering is, in case white <u>does not survive</u>.

Attrition games on graphs

And its intricacies

0-1 knapsack -> AGG-1:N-Rew

For 0-1 knapsack problem instance of n items $\langle w_i; v_i \rangle$ and a limit of a bag w_{max}, we define AGG-1:N-Rew input as follows:

White:
single unit $\langle w_{max}; 1 \rangle$

Black:
n units $\langle w_i; 0 \rangle$ and reward v_i
1 unit $\langle \infty; 1 \rangle$

Fully connected graph.

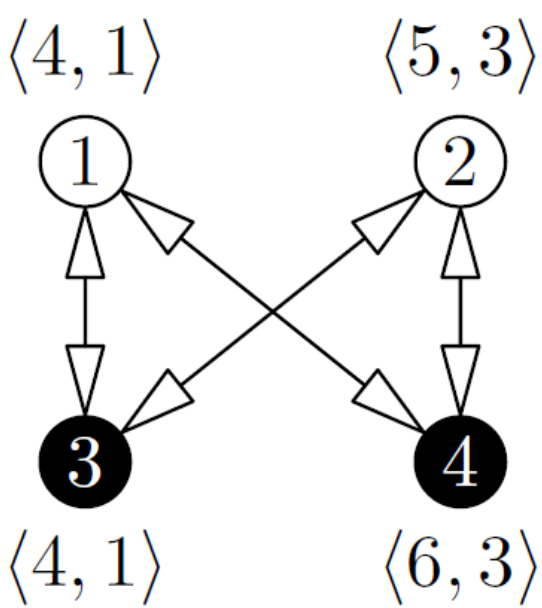
=> White unit is destroyed in w_{max} steps, may choose to eliminate at max w_{max} health, which can be distributed between black units. Corresponding knapsack items to destroyed units are the ones we're putting into the bag.

Attrition games on graphs

And its intricacies

Specific 2 vs 2 units, total annihilation AGG; requires mixed strategy

White: $\langle h = 4; a = 1\rangle, \langle 5; 3\rangle$; Black: $\langle 4; 1\rangle, \langle 6; 3\rangle$; fully conn.graph



Normal form representation

White

Black

	0	1	0	1
	-1	1	1	1
	1	1	**0**	1
	0	1	**1**	1

Black payoffs: 1 win, 0 draw, -1 lose

Bold == non-dominated strategies

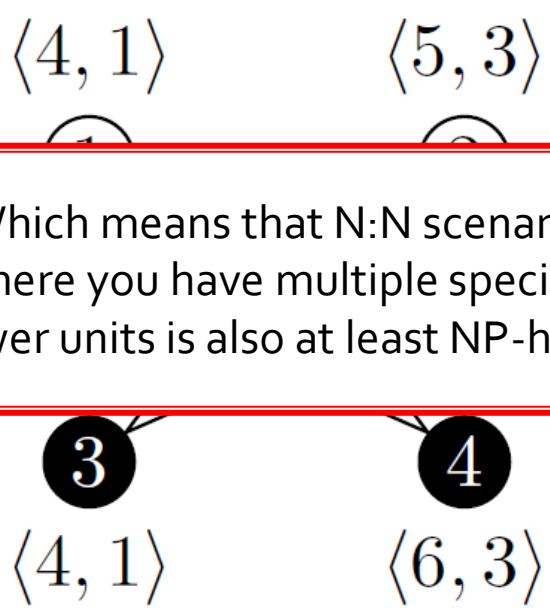
Attrition games on graphs

And its intricacies

Specific 2 vs 2 units, total annihilation AGG: requires mixed strategy

White: $\langle h = 4; a = 1\rangle, \langle 5; 3\rangle$; Blac aph

We have two non-dominated strategies here => equilibrium is in mixed strategies, in this concrete case 50:50.



Which means that N:N scenario where you have multiple special-power units is also at least NP-hard.

White

Black

	0	1	0	1
	-1	1	1	1
	1	1	**0**	1
	0	1	**1**	1

Black payoffs: 1 win, 0 draw, -1 lose

Bold == non-dominated strategies

Finally, the RTS combat!

Now we can safely discuss the paper:

Churchill, D., Saffidine, A., & Buro, M. (2012, October). Fast heuristic search for RTS game combat scenarios. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Modelling RTS combat

RTS Combat Game - States, units and moves

Units

Unit $u = \langle p, \mathrm{hp}, \mathrm{hp}_{\max}, \mathrm{t}_a, \mathrm{t}_m, v, w \rangle$

- Position $p = (x, y)$ in $\mathbb{R}^2$
- Current hit points hp and maximum hit points $\mathrm{hp}_{\max}$
- Time step when unit can next attack t_a, or move t_m
- Maximum unit velocity v
- Weapon properties $w = \langle \mathrm{damage}, \mathrm{cooldown} \rangle$

We also track damage-per-frame:

- $u.dpf = \dfrac{u.w.damage}{u.w.cooldown}$

Modelling RTS combat

RTS Combat Game - States, units and moves

Moves

Move $m = \{a_0, \ldots, a_k\}$ which is a combination of unit actions $a_i = \langle u, \text{type}, \text{target}, t \rangle$, with

- Unit u to perform this action
- The type type of action to be performed:

  Attack unit target

  Move u to position target

  Wait until time t

States

State $s = \langle \text{t}, p, m, U_1, U_2 \rangle$

- Current game time t
- Player p who performed move m to generate s
- Sets of units U_i under control of player i

Modelling RTS combat

RTS Combat Game - States, units and moves

Legal moves for units

Given a state s and unit u, its legal actions are:

$u.t_a \leq s.t$	… u may attack anything in range
$u.t_m \leq s.t$	… u may move in any direction
$u.t_m \leq s.t \leq u.t_a$	… u may wait
$s.t \leq u.t_m \ \& \ s.t \leq u.t_a$	… u must perform no-action

To illuminate the last condition, think of, e.g., a siege tank from Star Craft that can change between move and siege modes, which takes time.

Modelling RTS combat

RTS Combat Game - States, units and moves

Game terminal condition

All units of a player reaches zero HP.

Further assumptions

Zero-sum game (i.e., no asymmetric rewards)

Limitations wrt real RTS games

- no special powers or spells
- no hit point or shield regeneration
- no travel time for projectiles
- no unit collisions
- no unit acceleration, deceleration or turning
- no fog of war

=> Yet the game is harder than AGG == at least NP-hard => backward induction is unfeasible => we need to resort to (heuristic) searches

Scripted Behaviors

The „obvious“ heuristic

Scripted behaviors

Defining strategy through reactive behavior

Different types of scripts

Random	[RND]	Picks a legal move with uniform prob. distribution
Attack-Closest	[AC]	1. Attack closest in range if able 2. If reloading, wait 3. If not in range, move to range
Attack-Weakest	[AW]	Dtto AC, but: 1. attacks weakest in range
Kiting	[Kit]	Dtto AC, but: 2. move away from closest enemy
Attack-Value	[AV]	Dtto AC, but: 1. attacks u w/ highest $u.dpf$ / $u.hp$
No-OverKill-Attack-Value	[NOK-AV]	Dtto AV, but: will no try to attack unit that was already dealt with lethal damage
Kiting-AV	[Kit-AV]	Dtto Kit, but: 1. attacks u w/ highest $u.dpf$ / $u.hp$

Scripted strategy w/ script X: assign script X to all units a player controls

Search approximations

Adapting alpha-beta search to simultaneous games

Using searches

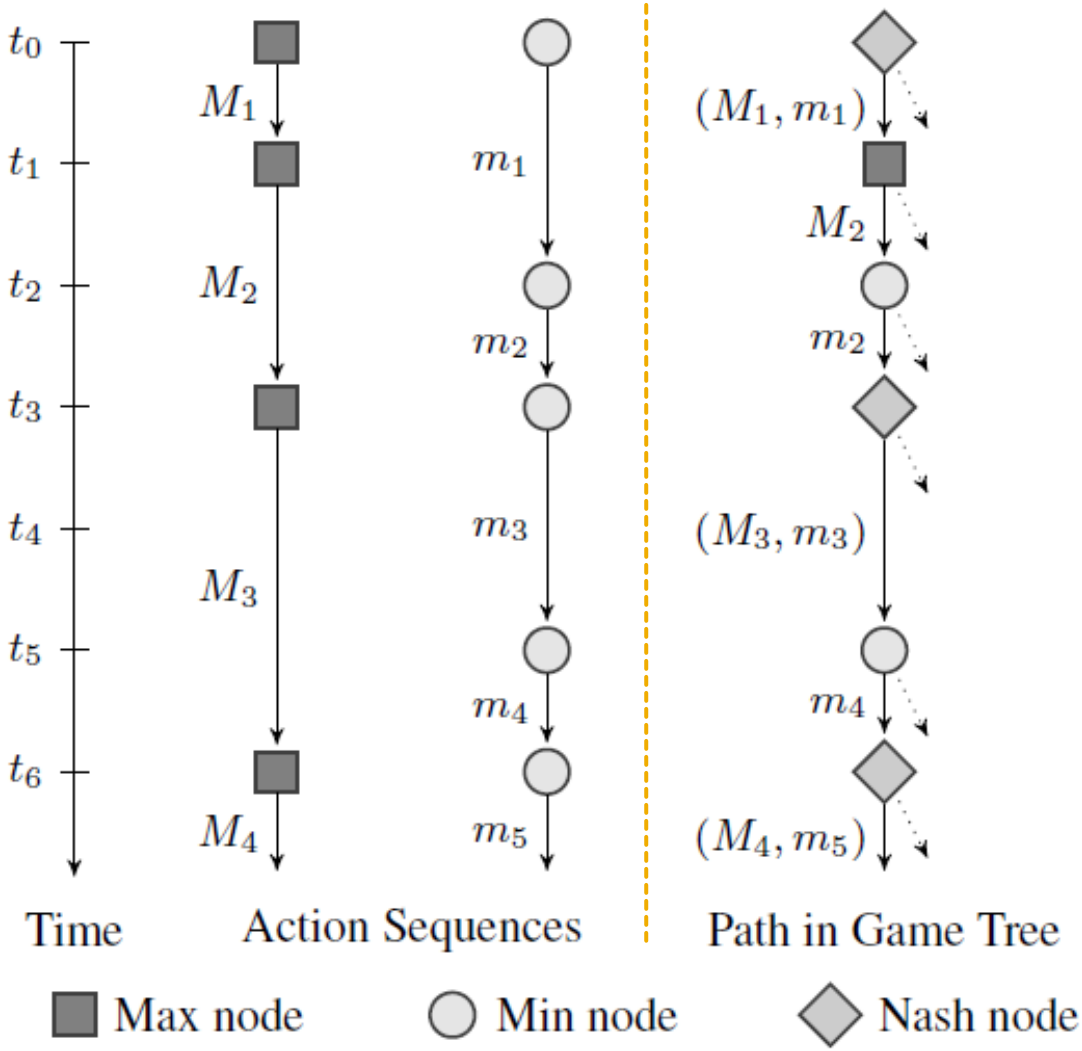
As an approximation of Nash equilibrium

Quick recap

Minimax (or maximin)	recursive algorithm to find the best move in <u>sequential</u> non-cooperative games
Alpha-beta pruning	technique improving minimax by pruning game tree of branches, which cannot bring better results than already found
Move ordering	heuristically ordering possible moves from (seemingly) better to worse to improve AB pruning
Evaluation function	a function evaluating a state used at depths where we stop searching
Iterative deepening	search technique where we incrementally increasing search depths until time runs out; allows for any-time results
Transposition tables	cache to maintain previously seen states that allows to reuse results especially during iterative deepening

Durative actions

The problem of simultaneous games



In sequential games, players alternate; in RTS combat, they might not. The same player might decide multiple times (easy to handle) and there are points in time where players decides on moves simultaneously (troublesome).

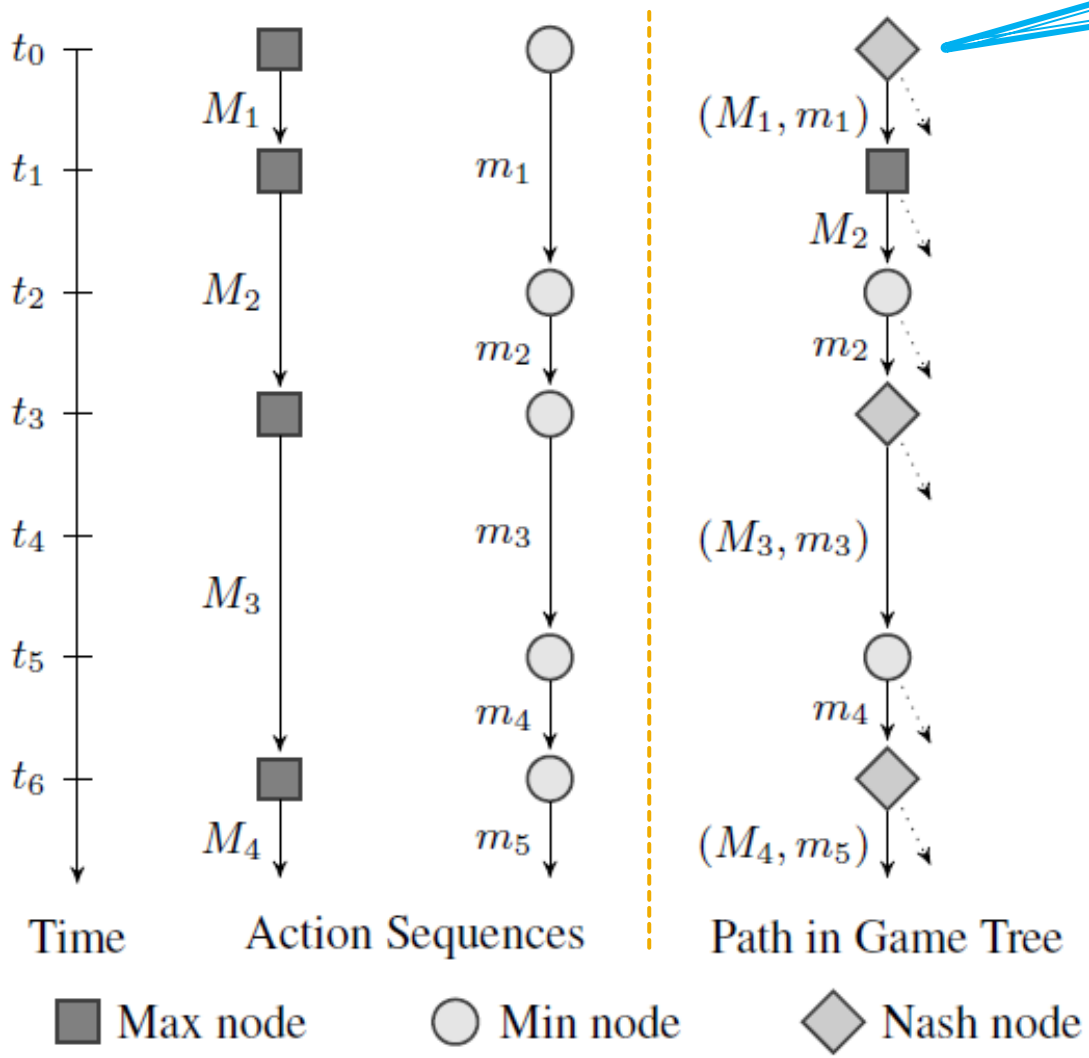
Deciding on moves first may disclose information (see 2v2 case for AGG).

We need to adapt standard MINMAX (alpha-beta) algorithm for durative actions.

Durative actions

The problem of simultaneous g

How to deal with nodes, where players are deciding on actions simultaneously (labeled as Nash nodes)?



In sequential games, players alternate; in RTS combat, they might not. The same player might decide multiple times (easy to handle) and there are points in time where players decides on moves simultaneously (troublesome).

Deciding on moves first may disclose information (see 2v2 case for AGG).

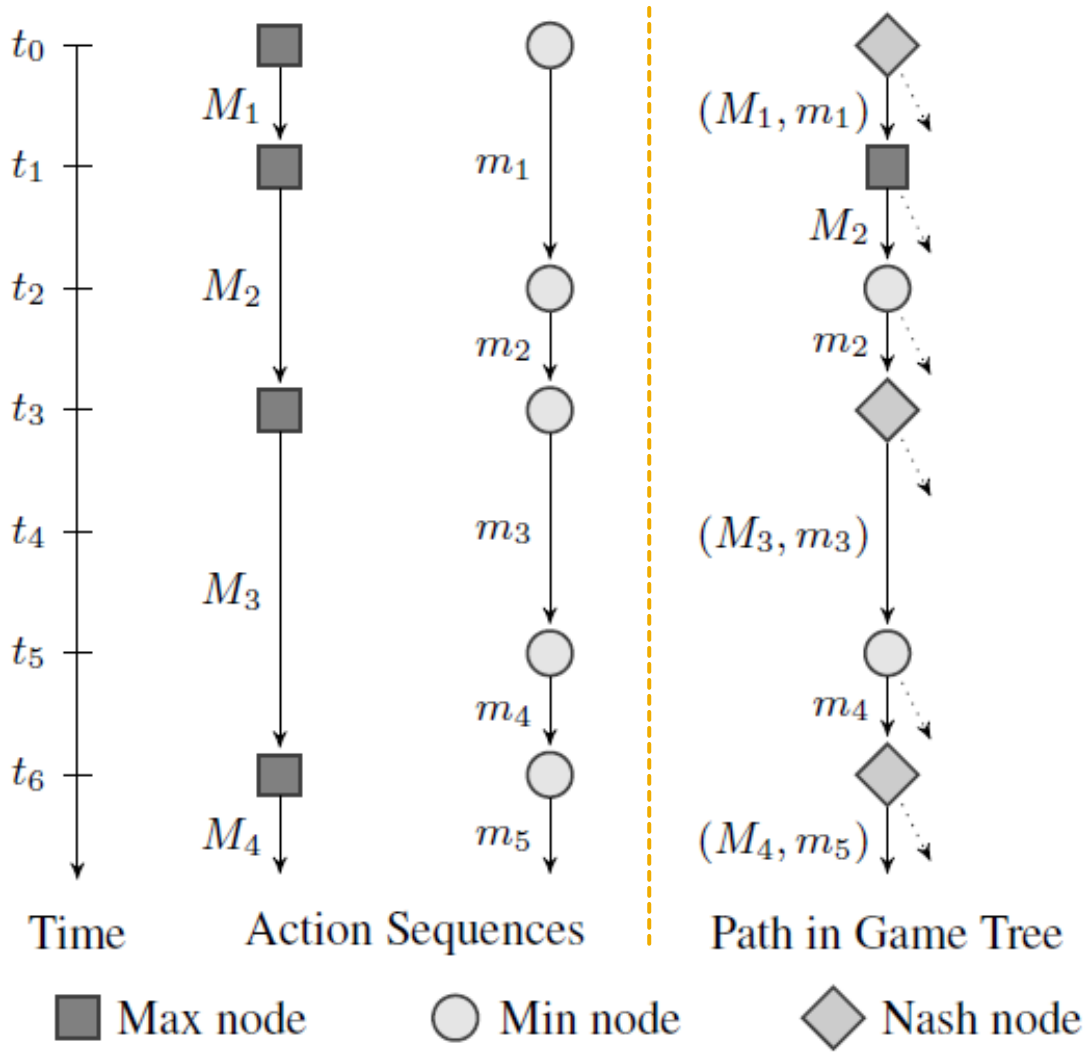
We need to adapt standard MINMAX (alpha-beta) algorithm for durative actions.

Stacked matrix games

(theoretically sound model)

Stacked matrix games

Dealing with "simultaneous move" node



In simultaneous nodes, one need to employ normal form of description for the node, i.e., both players decides at ones, which leads to the full matrix of decisions and computing their payoffs.

Such payoffs cannot be determined immediately, and we need to continue the search to determine them.

Randomized Alpha-Beta (RAB)

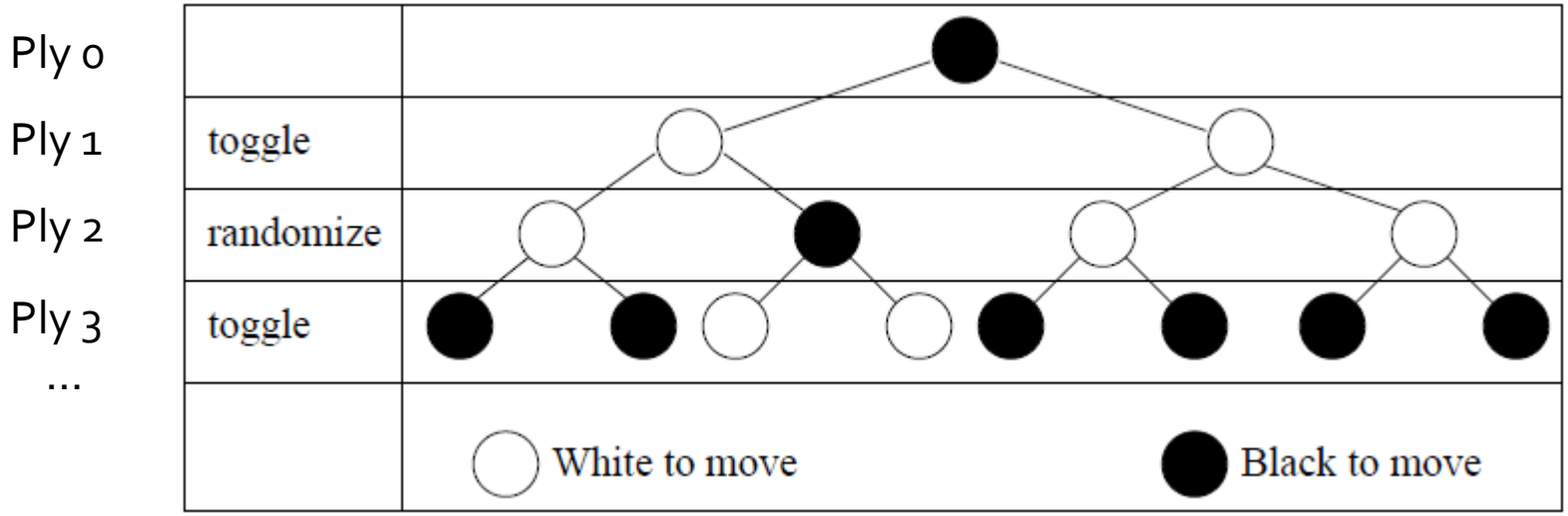
(policy for serialization of simultaneous nodes)

Detour to:

Kovarsky, A., & Buro, M. (2005, May). Heuristic search applied to abstract combat games. In *Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 66-78). Springer, Berlin, Heidelberg.

Randomized Alpha-Beta (RAB)

Serializing the simultaneous node



Assumption: game with simultaneous only; search is containing only simultaneous nodes.

Idea: Randomize order of players on even plies; toggle on odd.

Algorithm: see the paper

Alternate Alpha-Beta (ALT)

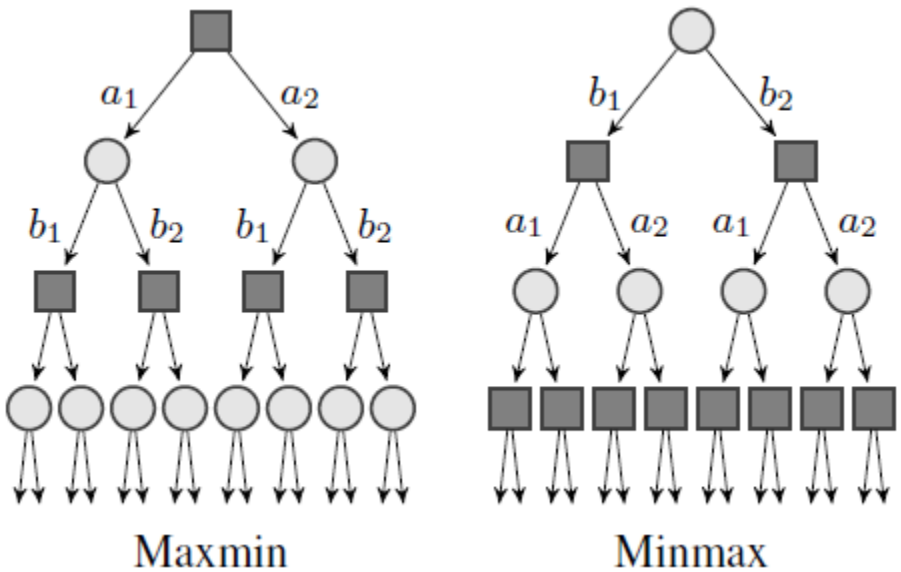
(policy for serialization of simultaneous nodes)

Getting back to:

Churchill, D., Saffidine, A., & Buro, M. (2012, October). Fast heuristic search for RTS game combat scenarios. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Alternate Alpha-Beta (RAB)

Serializing the simultaneous node



Assumption: not really, simultaneous node can be encountered anywhere along the search

Idea: Once simultaneous node is reach, alternate between maxmin and minimal; e.g. on the first sim. node use maxmin, on the second minmax, etc.

Alpha-Beta Considering Durations

(ABCD)

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

s – state

d – depth to search

m_0 – delayed action effect used for simultaneous nodes

α, β – bounds

Meant to be used with iterative deepening.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

First, mind the real-time constraints. Note that this ABCD should be run in "iterative deepening" manner, thus timeout means "do not use this result at all".

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

If we are in terminal node (either depth reaches zero or maximal time for scenario is reached), we return valuation of the state.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

This line condense a lot of stuff.

[A] If none unit can perform actions (different then "pass"), advance the time to the point some unit may perform an action first.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

This line condense a lot of stuff.

[B] Given the current state, i.e., state of units, determine which players can move.

If at this stage, we are in simultaneous node, use "policy" to determine, which player will make its decision first.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

Next, we are iterate over moves player "toMove" can do, save current batch of moves we're going to investigate into ***m***.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

If we are in simultaneous node, and there is nothing in m_0 buffer, i.e., this ABCD has not been called from simultaneous node …

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

... then we continue with next ply of ABCD, but (!) passing actions "m" as an argument.

This "m" will act as m_0 in next invocation, so we will not get into this branch next time.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

Additionally, if we are near the end of the search depth, we do not bother resolving simultaneous node as we're terminating anyway.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

The else branch is then about solving the "delayed action" effect.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

First, this version of ABCD is not using reversible action. So even though we are performing DFS, we clone the state.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

Then, we are solving the delayed action effect, if m_0 is containing some actions, we apply them here …

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

... before applying currently selected actions.

Alpha-Beta Considering Durations

a.k.a. ABCD

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

The rest of the algorithm is standard alpha-beta prunning.

State Evaluation Function

(eval)

State evaluation function

eval

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

The state evaluation function is used here to evaluate nodes at certain depths.

State evaluation function

eval

3: **else if** terminal(s, d) **then return** eval(s)

$$\mathrm{LTD}(s) = \sum_{u \in U_1} \mathrm{hp}(u) \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \mathrm{hp}(u) \cdot \mathrm{dpf}(u)$$

$$\mathrm{LTD2}(s) = \sum_{u \in U_1} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u) - \sum_{u \in U_2} \sqrt{\mathrm{hp}(u)} \cdot \mathrm{dpf}(u)$$

LTD3(s): playout; instead of evaluation finish the game by performing a playout, which means, on each unit decision point, use preselected script to select an action. Play until either or both sides are annihilated.

Move Ordering
in
ABCD Iterative Deepening

Move Ordering

Sequence of trying out the actions

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1 then
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

If move ordering is done right, it improves the effect of alpha-beta pruning as better state values are found faster.

Move Ordering

Sequence of trying out the actions

Algorithm 1 Alpha-Beta (Considering Durations)

```
procedure ABCD(s, d, m_0, α, β)
    if computationTime.elapsed then return timeout
    else if terminal(s, d) then return eval(s)
    toMove ← s.playerToMove(policy)
    while m ← s.nextMove(toMove) do
        if s.bothCanMove and m_0 = ∅ and d ≠ 1
            val ←ABCD(s, d − 1, m, α, β)
        else
            s′ ← copy(s)
            if m_0 ≠ ∅ then s′.doMove(m_0)
            s′.doMove(m)
            v ←ABCD(s′, d − 1, ∅, α, β)
        if toMove = MAX and (v > α) then α ← v
        if toMove = MIN and (v < β) then β ← v
        if α ≥ β then break
    return toMove = MAX ? α : β
```

As we are running ABCD using iterative deepening, we can store information about promising moves from previous runs. This can be used to sort actions in consecutives ABCD invocations, allowing it to run faster. If no such information is available, we can still use a script to suggest "first move".

Paper Results

Setup

Paper Results

N vs. N combats in Star Craft; N ranged 2-8

Four different army types: Marine only, Marine+Zergling, Dragoon + Zealot, Dragoon + Marine (all combinations, up-to 4 of each unit type)

Symmetric starting locations

Max 500 actions, after that LTD was used to determine the winner.

Final player score: score = (#wins + #draws / 2) / #matches

Scripts vs. Searches

Paper Results

Alt': The difference from described Alt is, that in simultaneous nodes, we pick the player to move first as the one who moved last.

Opponent	ABCD Search Setting				
	Alt LTD	Alt LTD2	Alt NOK-AV	Alt' NOK-AV	RAB' Playout
Random	0.99	0.98	1.00	1.00	1.00
Kite	0.70	0.79	0.93	0.93	0.92
Kite-AV	0.69	0.81	0.92	0.96	0.92
Closest	0.59	0.85	0.92	0.92	0.93
Weakest	0.41	0.76	0.91	0.91	0.89
AV	0.42	0.76	0.90	0.90	0.91
NOK-AV	0.32	0.64	0.87	0.87	0.82
Average	0.59	0.80	0.92	0.92	0.91

Searches vs. Searches

Paper Results

Table 2: Playout-based ABCD performance

Opponent	Alt NOK-AV	Alt' NOK-AV	RAB' Playout
Alt-NOK-AV		0.47	0.46
Alt'-NOK-AV	0.53		0.46
RAB'-NOK-AV	0.54	0.54	
Average	0.54	0.51	0.46

Scripts exploitability

Paper Results

Table 3: Real-time exploitability of scripted strategies.

Random	Weakest	Closest	AV	Kiter	Kite-AV	NOK-AV
1.00	0.98	0.98	0.98	0.97	0.97	0.95

Results of search with opponent modeling, i.e., the other player in the search was modeled by the exact script the search has been playing against.

We can see, that scripts are highly exploitable!

That’s it for today!

Faculty of Mathematics and Physics
Charles University
12.10.2022

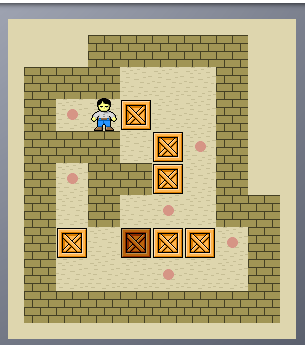
GAME DEVELOPMENT @CUNI.CZ

matfyz

NAIL122 AI for Games

Modeling (A-based) AI Player*

Jakub Gemrot



How to think about AI player?

Modeling AI Player

High-level abstraction

Game as the process of information exchange.

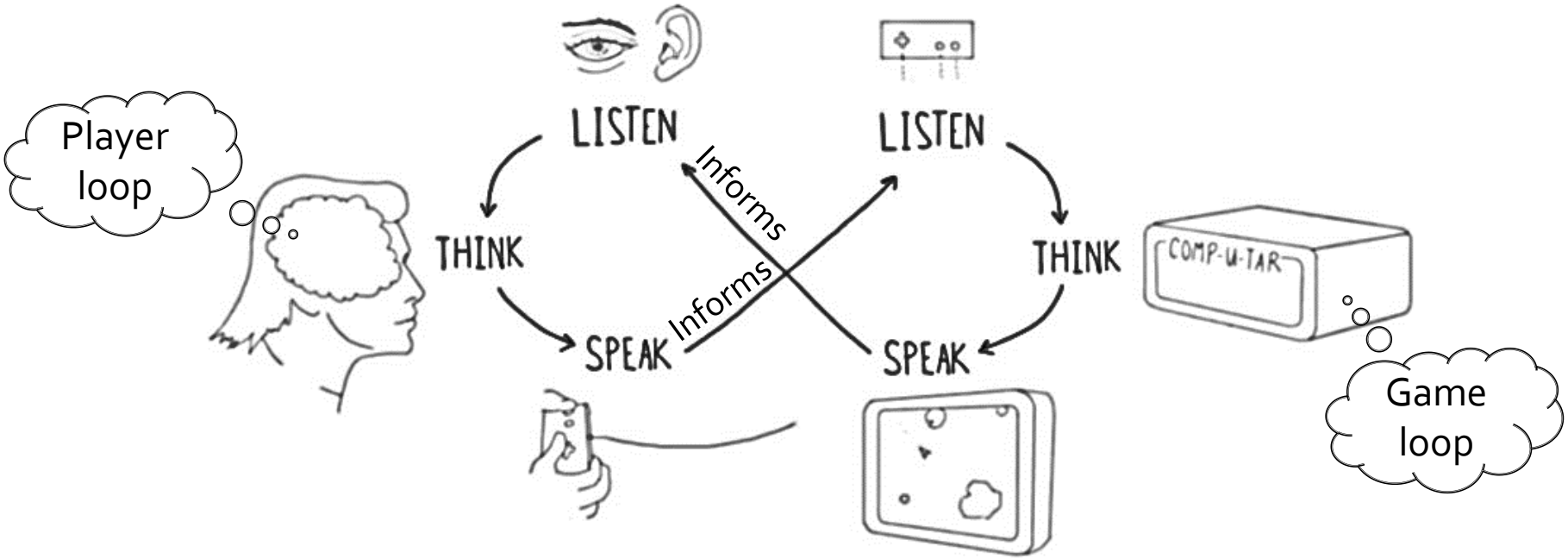


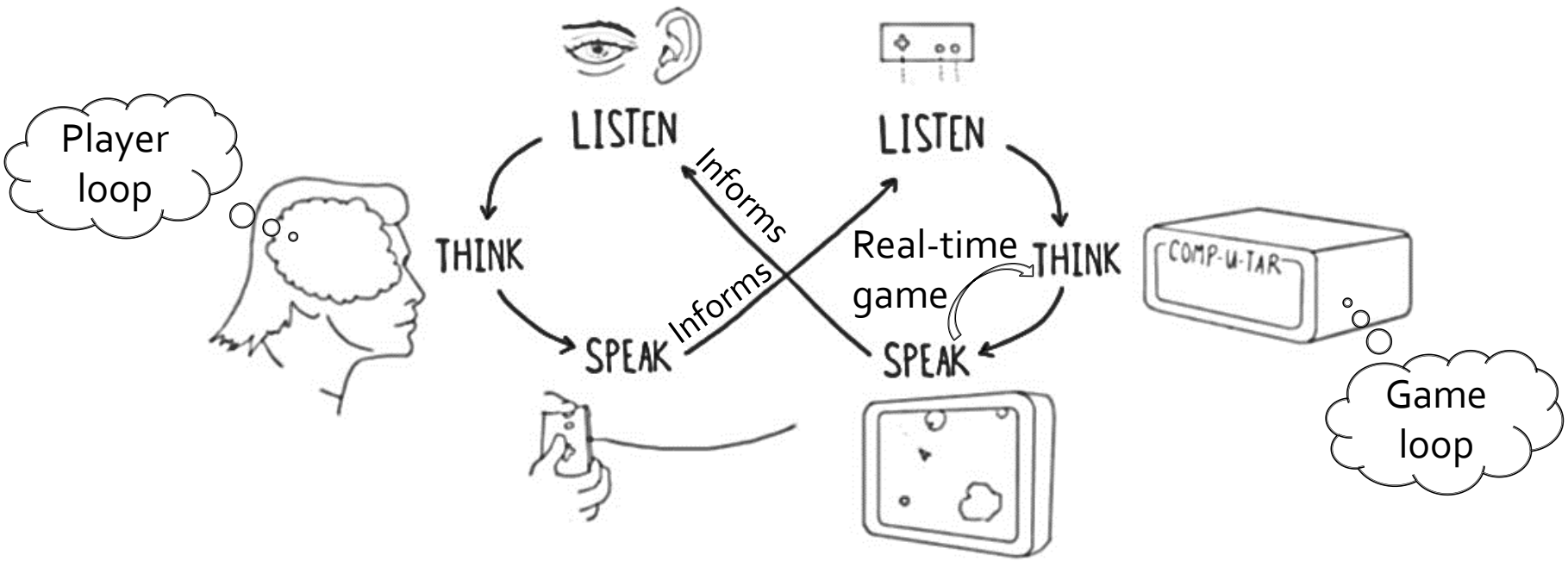
Image: Swink, S. (2008). Game feel: a game designer's guide to virtual sensation. CRC Press.

Modeling AI Player

High-level abstraction

Game as the process of information exchange.

Mind the real-time backloop!



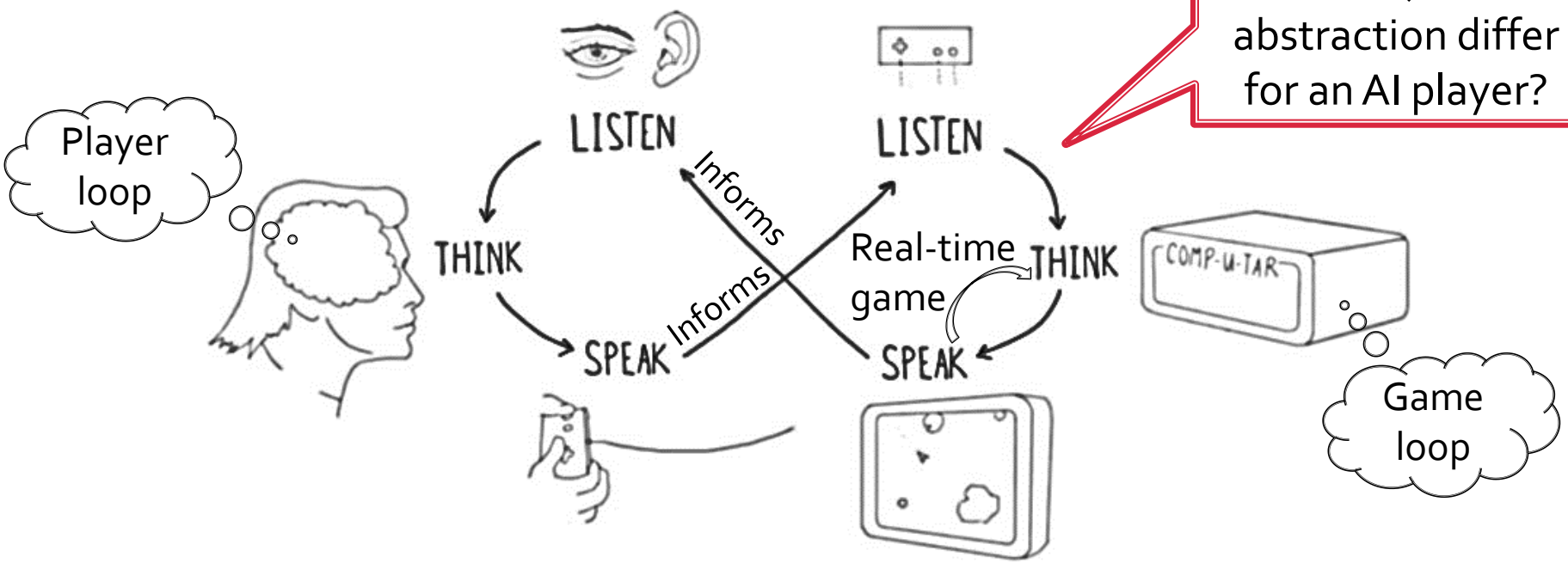
Modeling AI Player

High-level abstraction

Game as the process of information exchange.

Mind the real-time backloop!

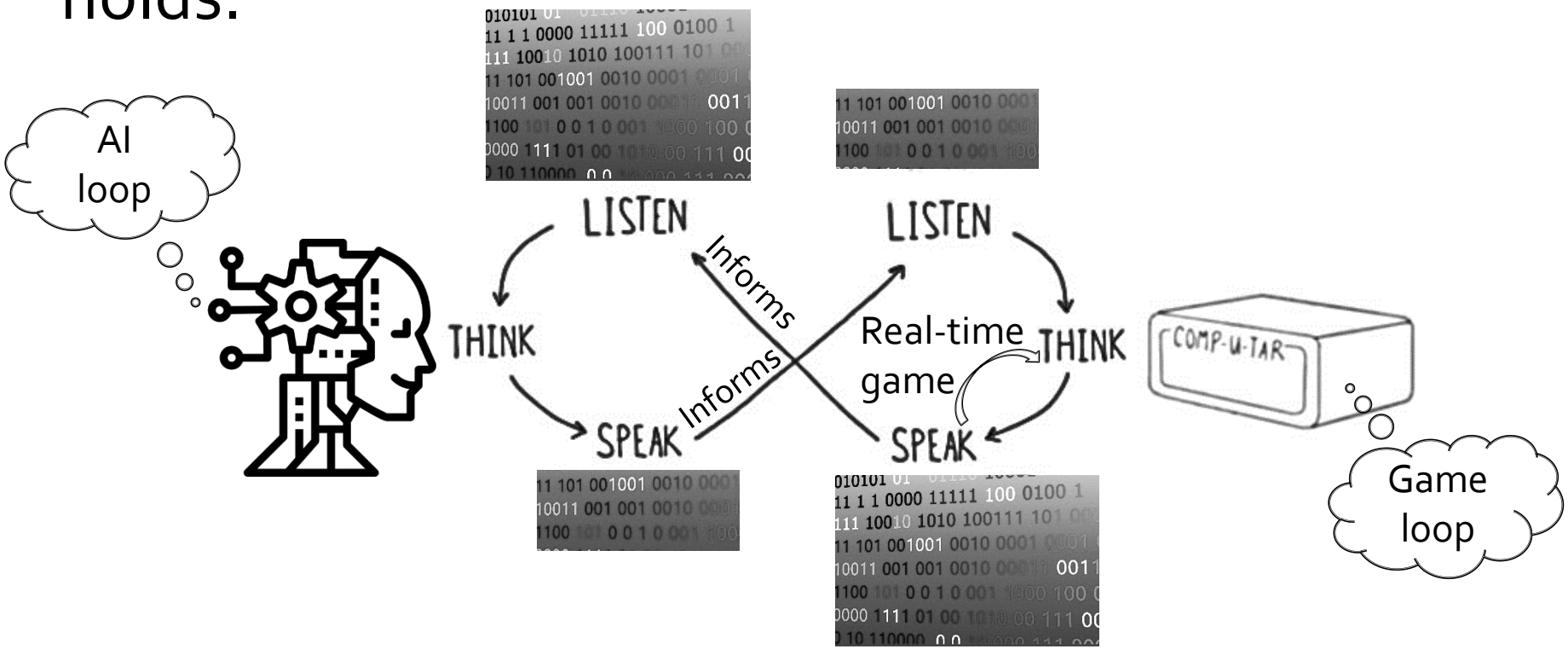
Does (and how much) the abstraction differ for an AI player?



Modeling AI Player

High-level abstraction

"Only" the interface is different, but abstraction holds.

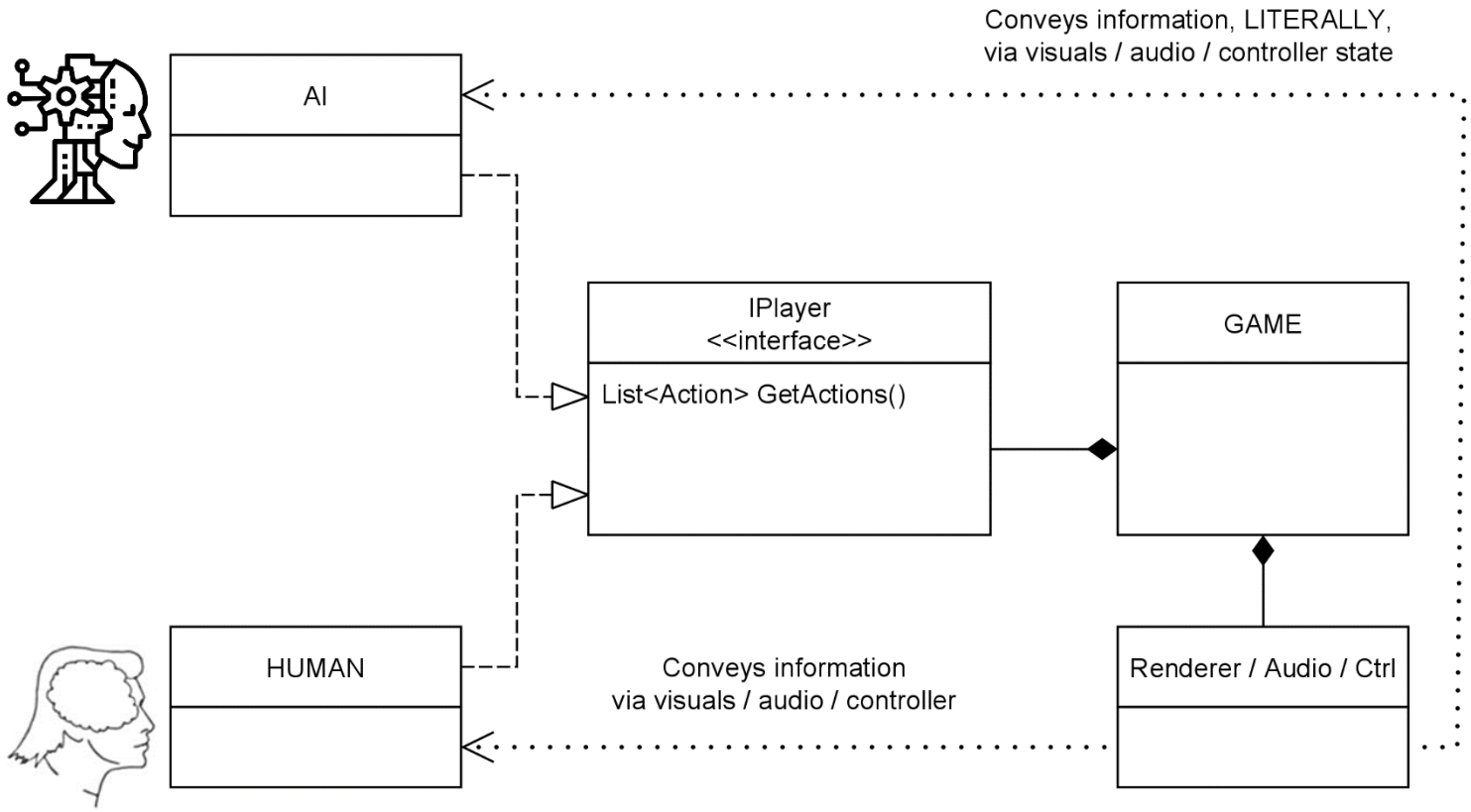


AI Image: photo3_idea, https://www.flaticon.com/free-icon/ai_1693746

Modeling AI Player

UML-like abstraction

Version A



Modeling AI Player

UML-like abstraction

Version A

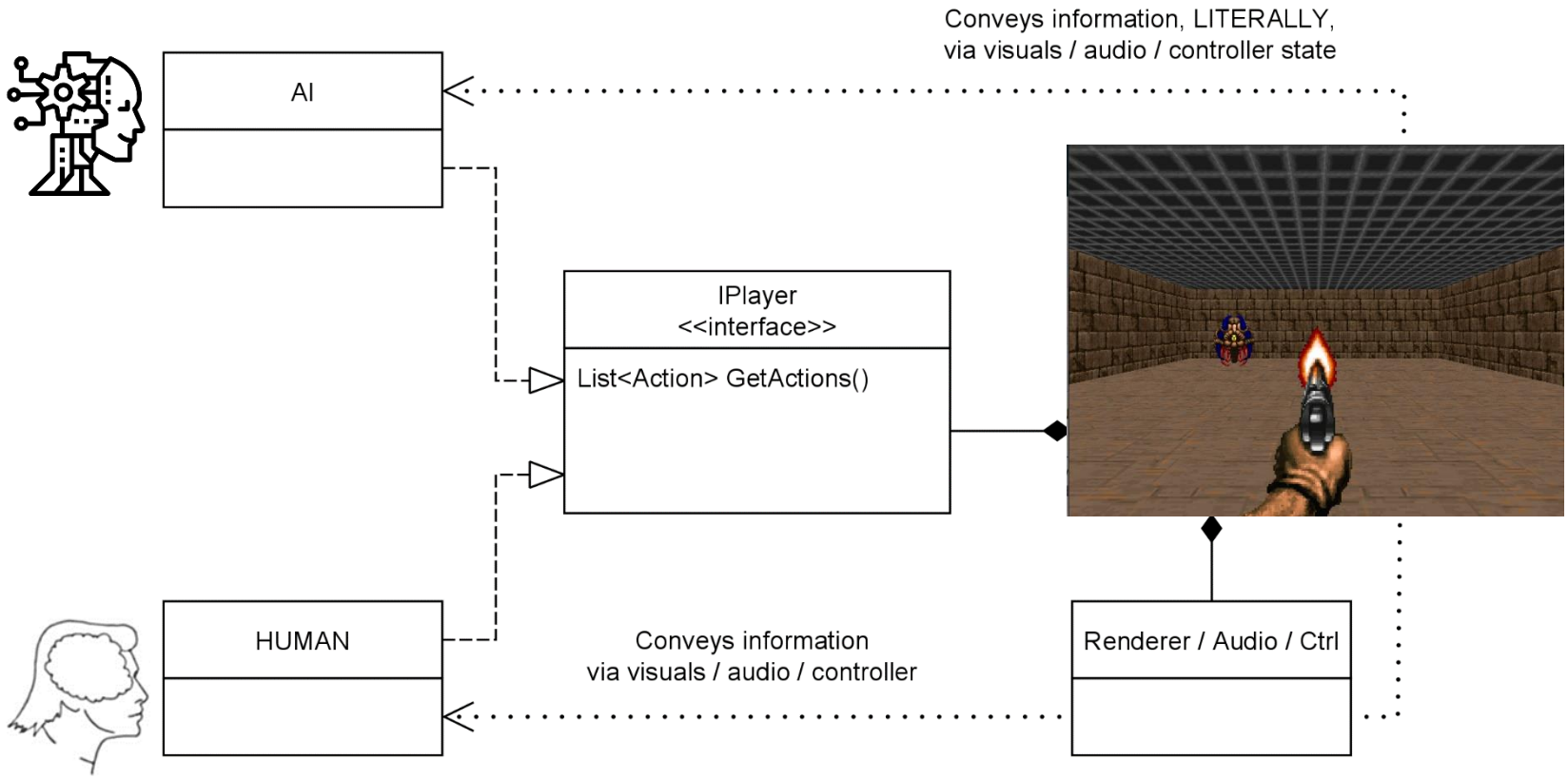
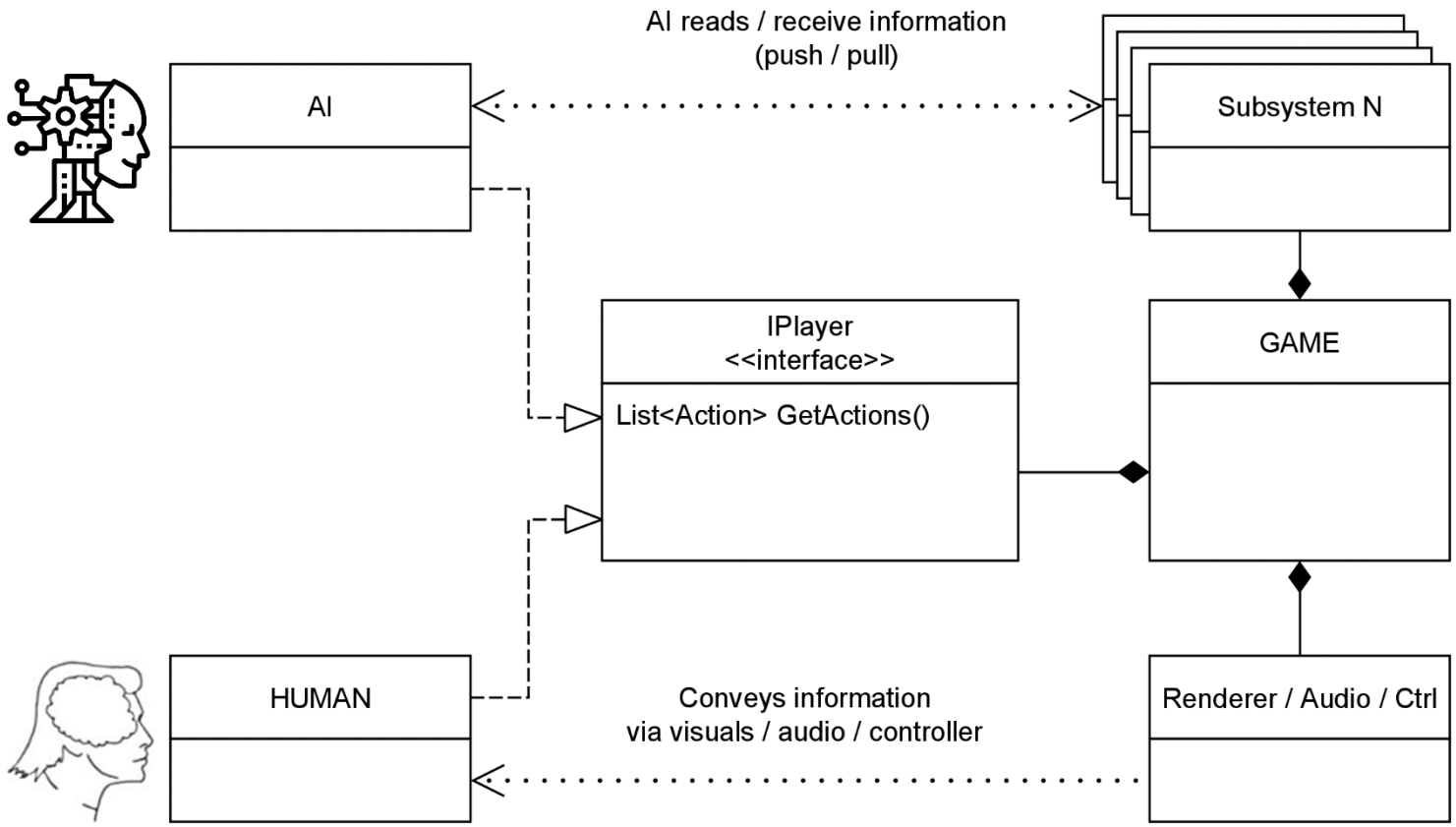


Image: VizDoom, http://vizdoom.cs.put.edu.pl/tutorial

Modeling AI Player

UML-like abstraction

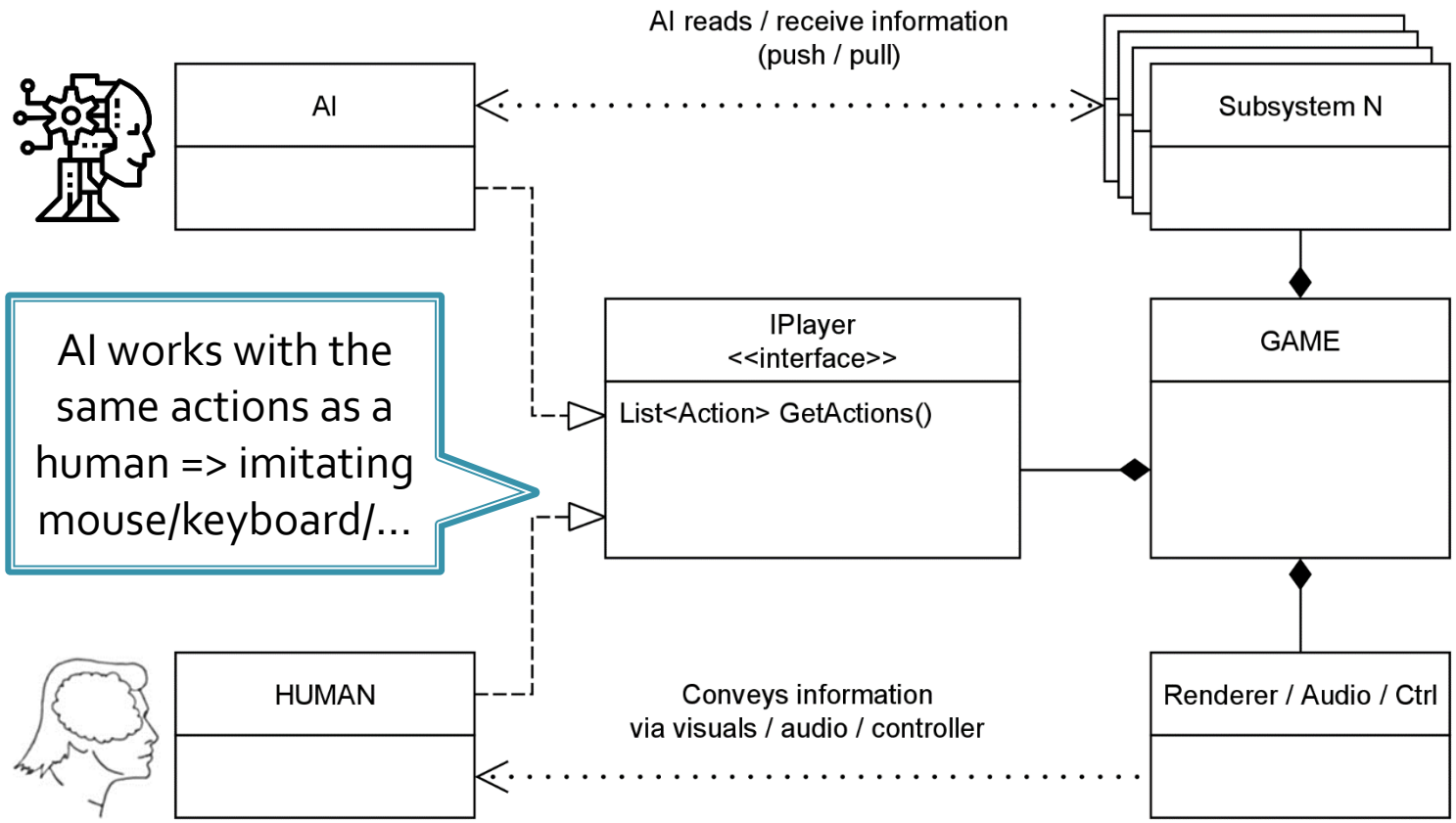
Version B



Modeling AI Player

UML-like abstraction

Version B



AI works with the same actions as a human => imitating mouse/keyboard/...

Modeling AI Player

UML-like abstraction

Version B

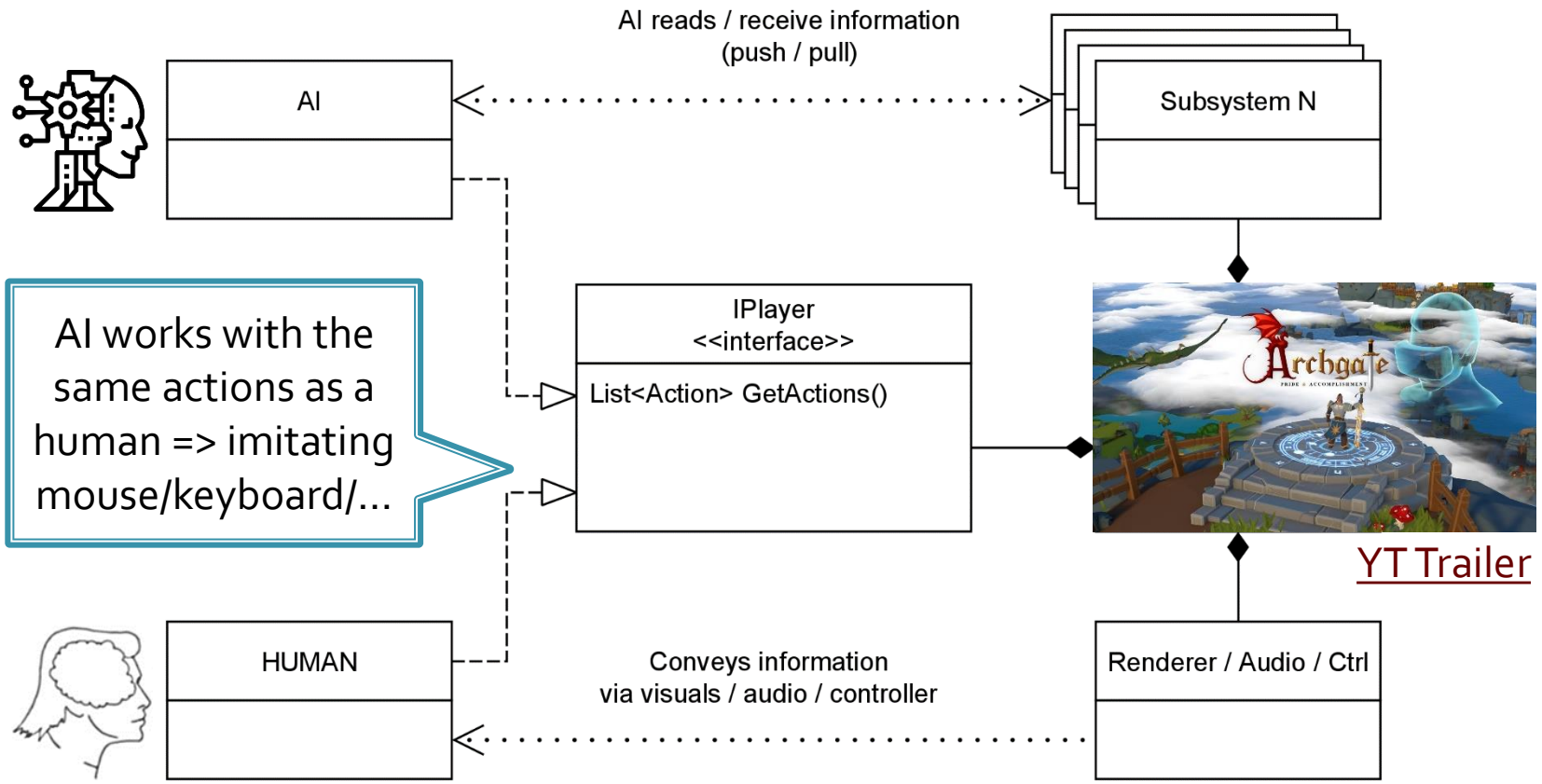


Image: Samdosoft, Archgate Pride & Accomplishment, https://www.samdosoft.com/archgate/

Modeling AI Player

UML-like abstraction

Version B

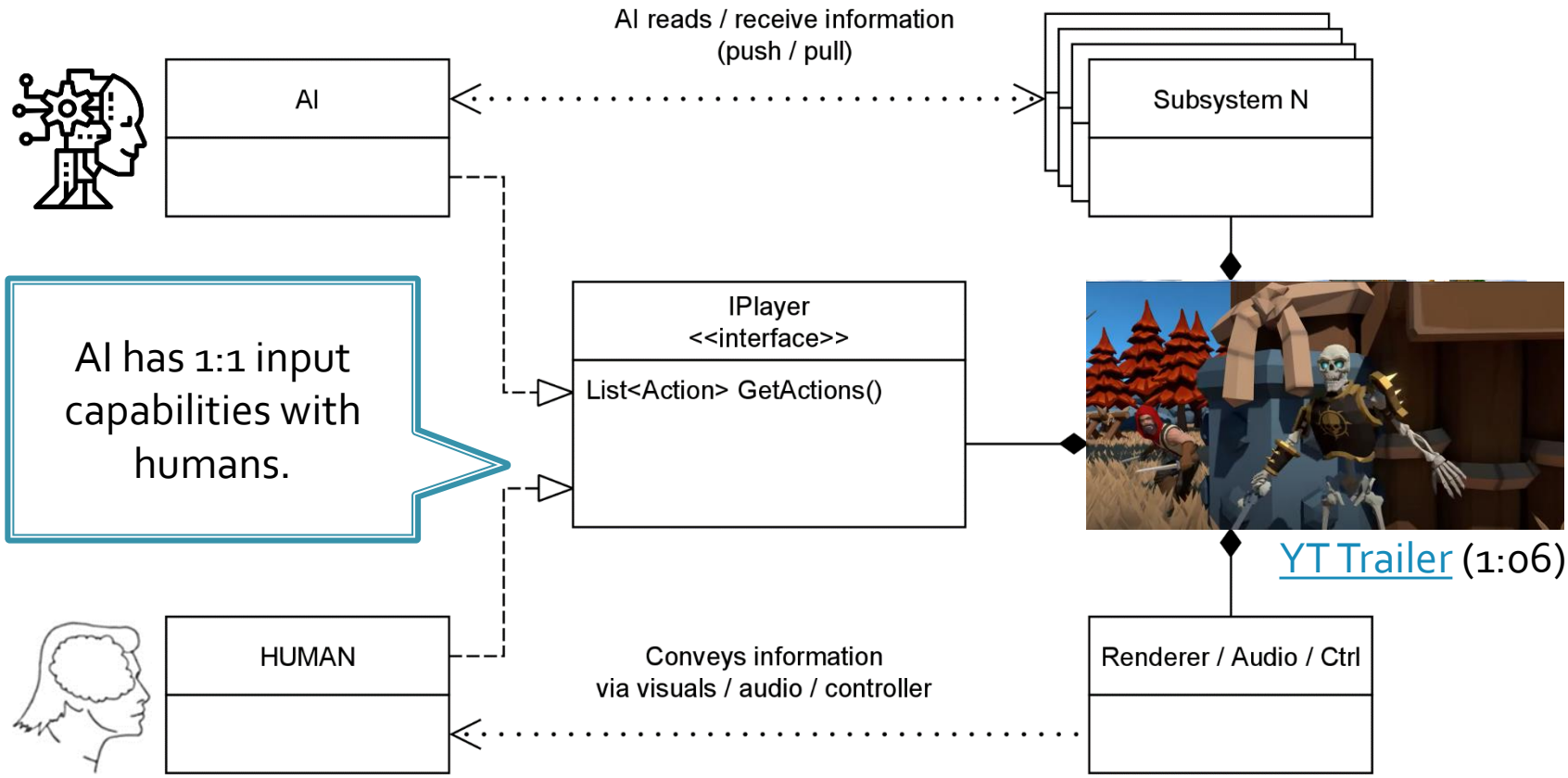
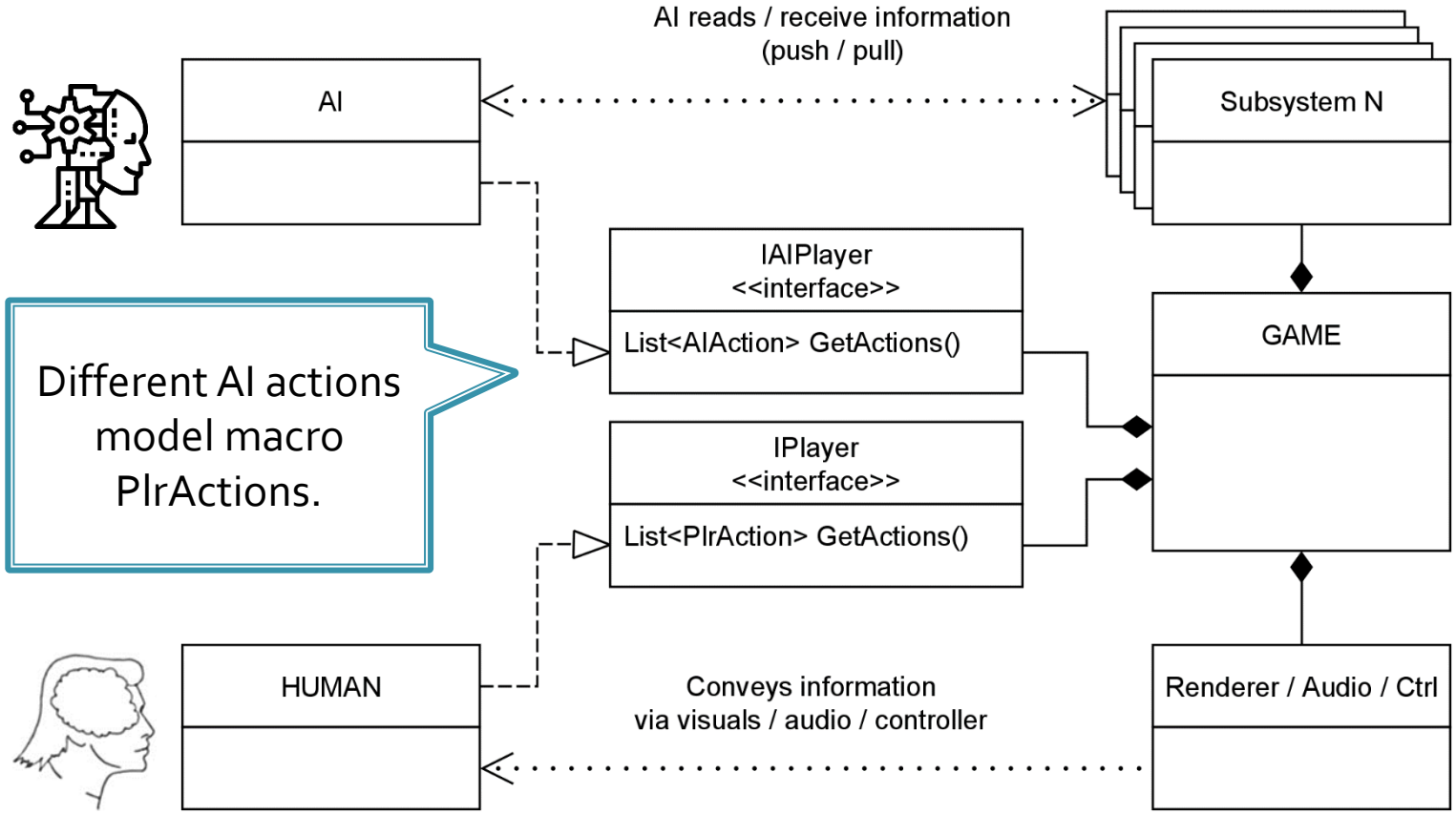


Image: Samdosoft, Archgate Pride & Accomplishment, https://www.samdosoft.com/archgate/

Modeling AI Player

UML-like abstraction

Version C



Different AI actions model macro PlrActions.

Modeling AI Player

UML-like abstraction

Version C

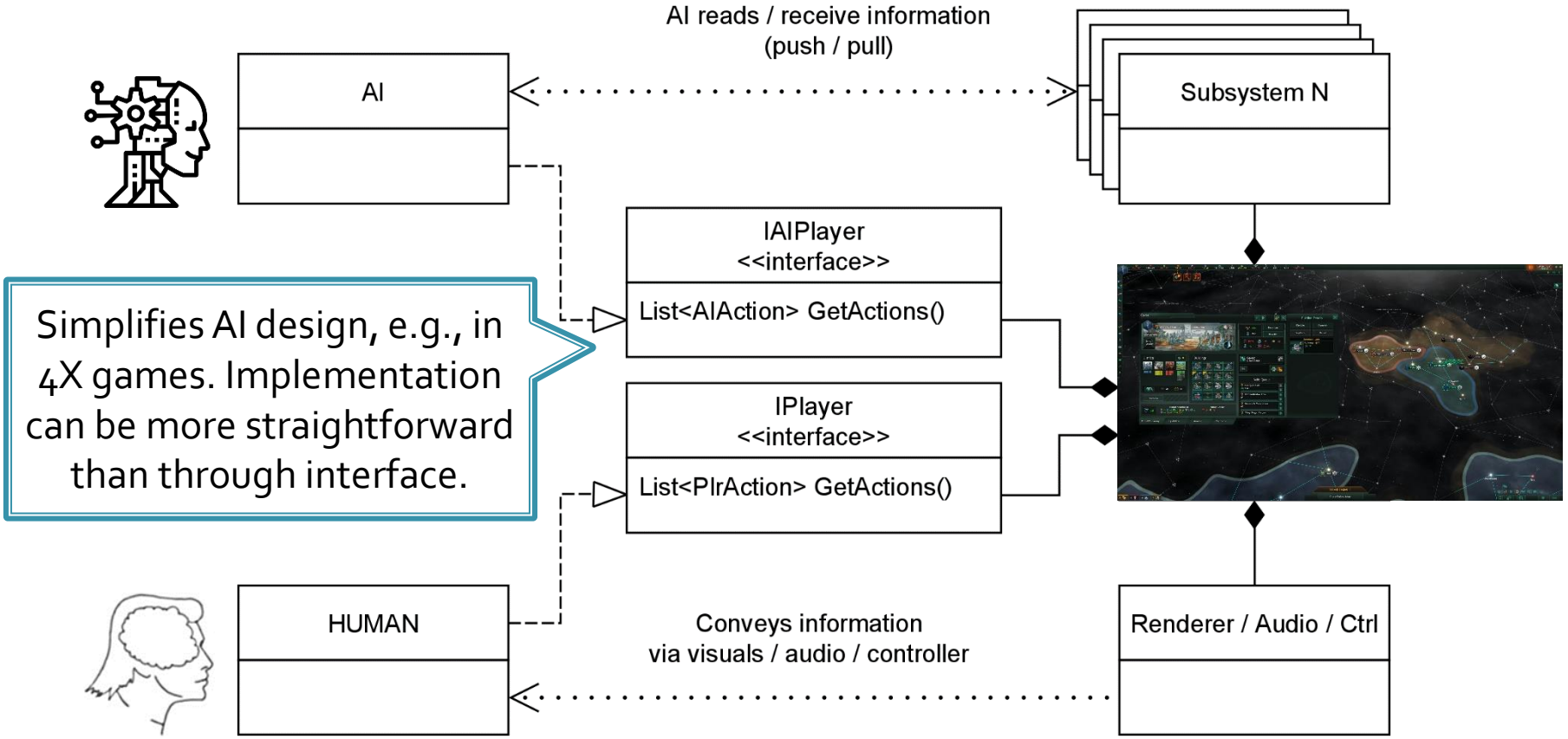
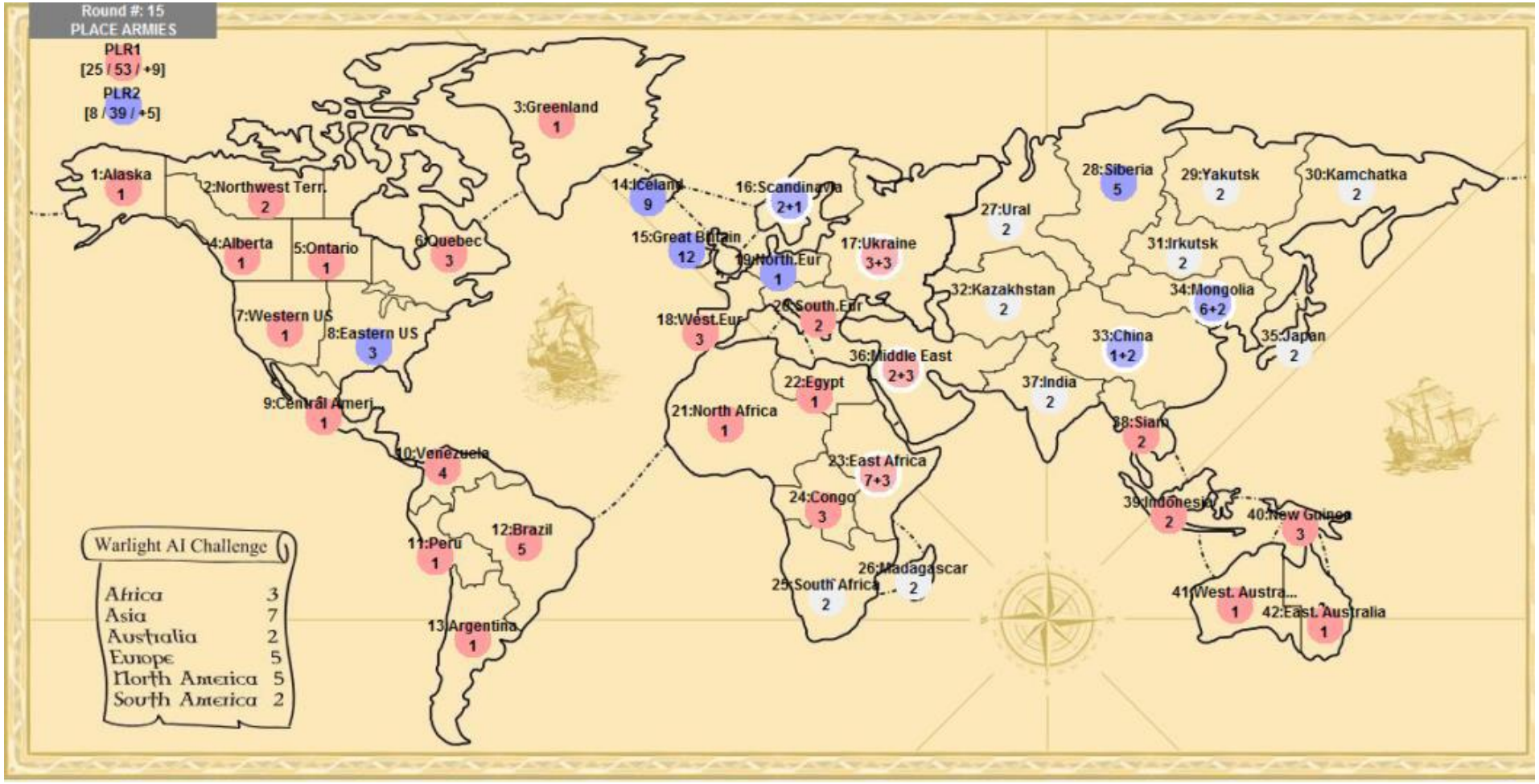


Image: Paradox Interactive, Stellaris, https://store.steampowered.com/app/281990/Stellaris

Modeling AI Player

UML-like abstraction

Version C – Example of AI Macro action different from Human actions



Modeling AI Player

UML-like abstraction

Version C

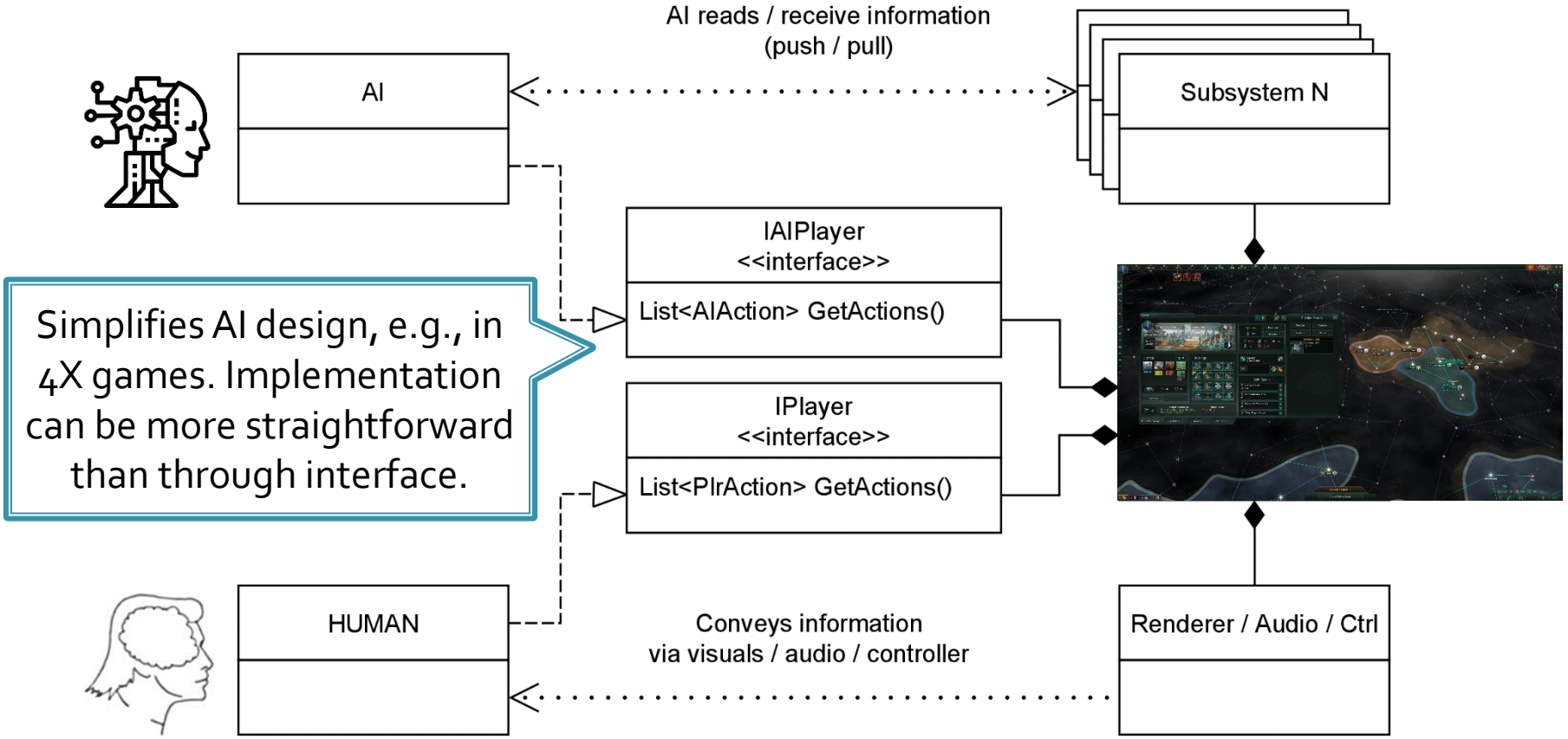


Image: Paradox Interactive, Stellaris, https://store.steampowered.com/app/281990/Stellaris

Modeling AI Player

UML-like abstraction

Version D

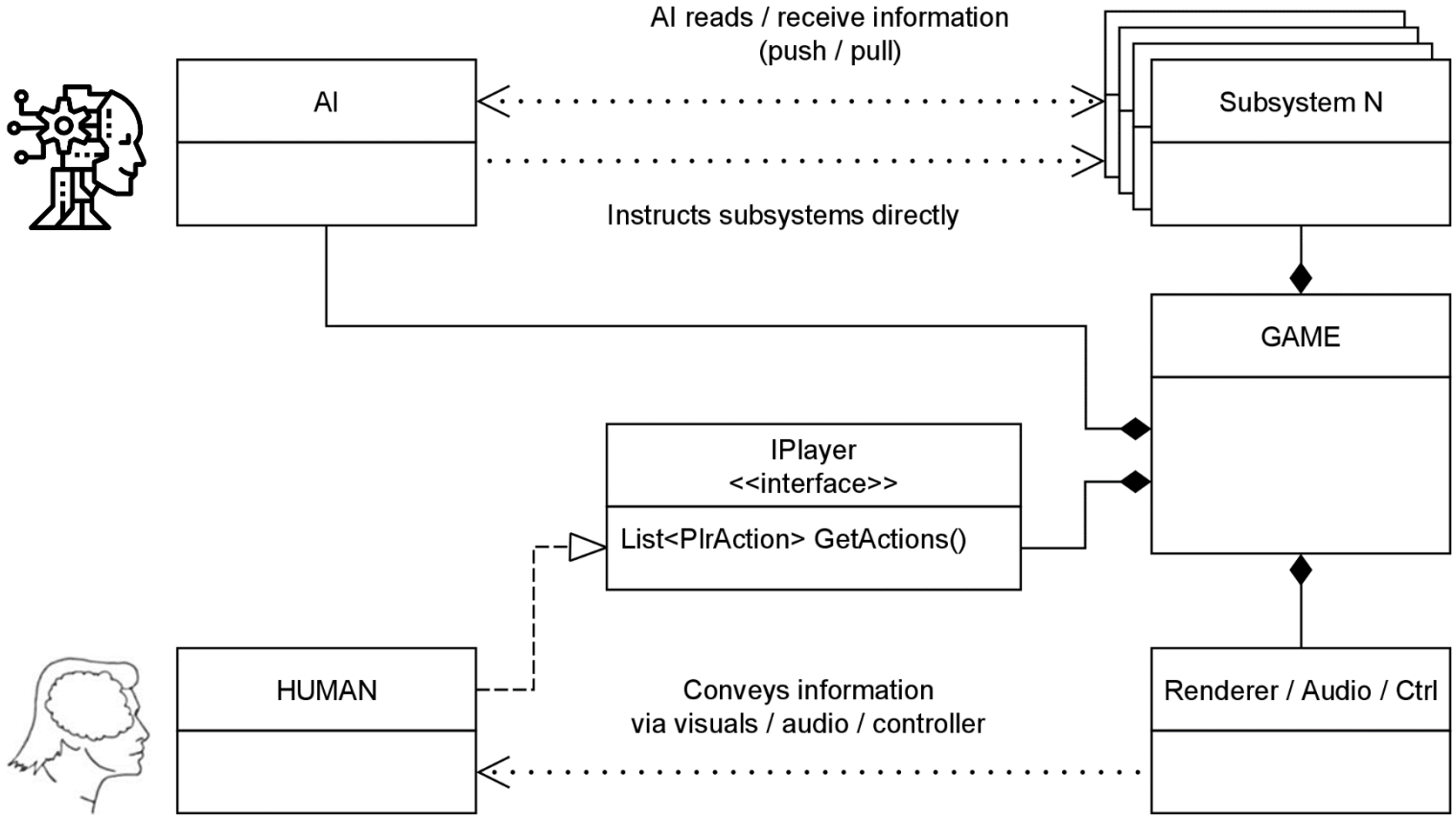


Image: Paradox Interactive, Stellaris, https://store.steampowered.com/app/281990/Stellaris

This AI Player model is sufficient for <u>REACTIVE</u> agents only!

Modeling AI Player

Short peek into intelligent agents

Simple reflex agent

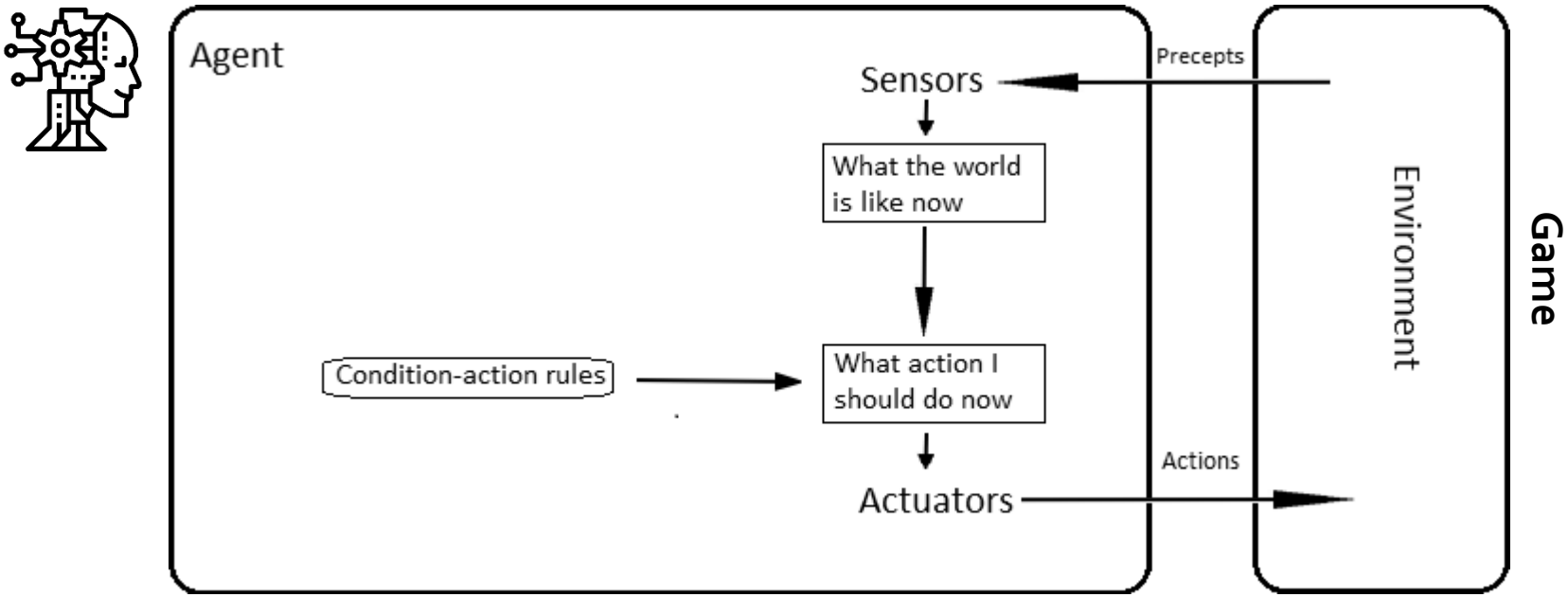
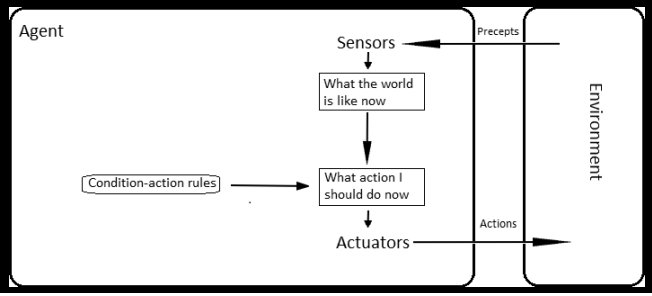


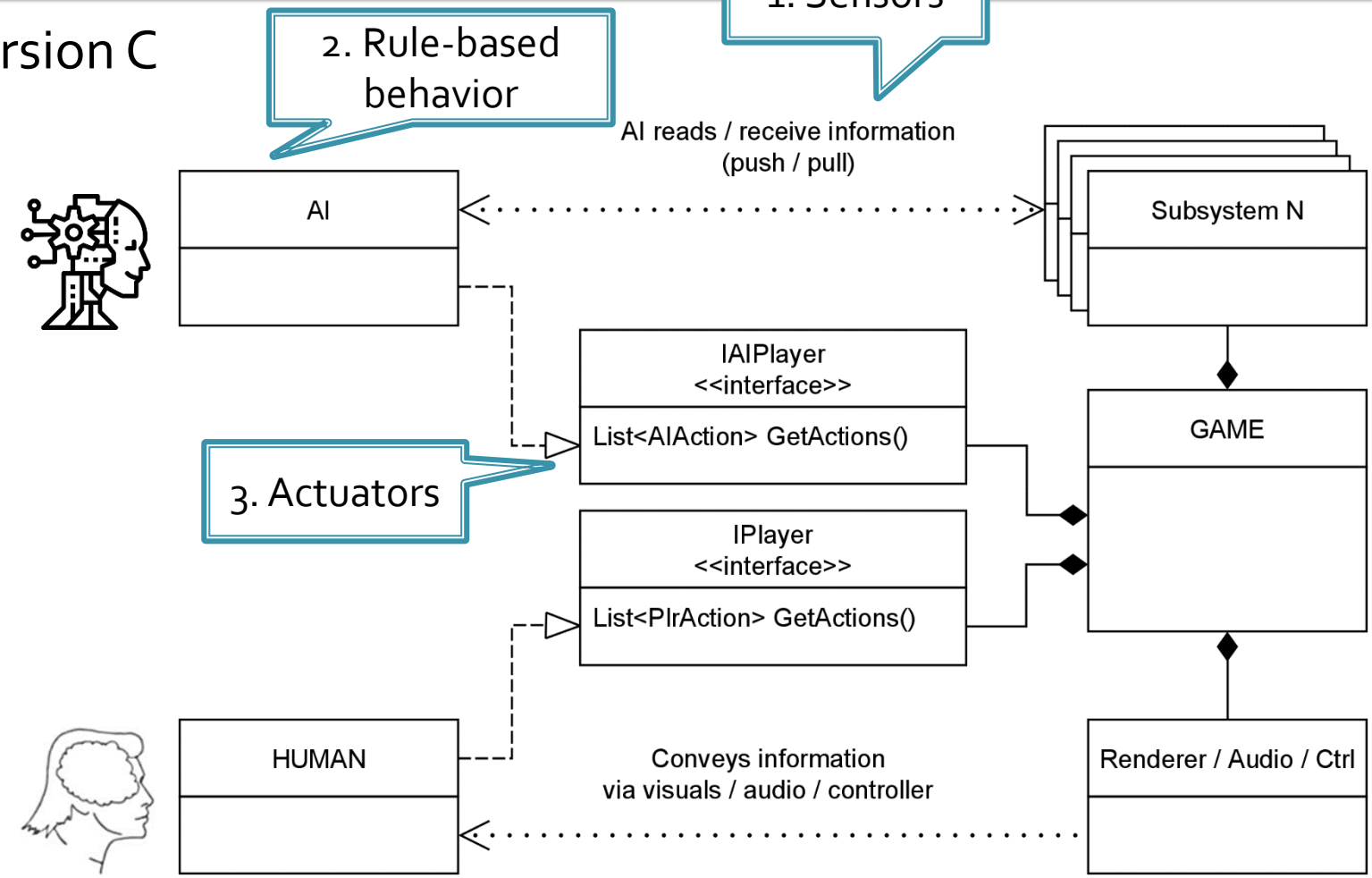
Image: By DDSniper - Originally fromStuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach p.49; with additions by me, CC0, https://commons.wikimedia.org/w/index.php?curid=14666503

Modeling AI Player

UML-like abstraction



Version C



Modeling AI Player

Short peek into intelligent agents

Simple reflex agent

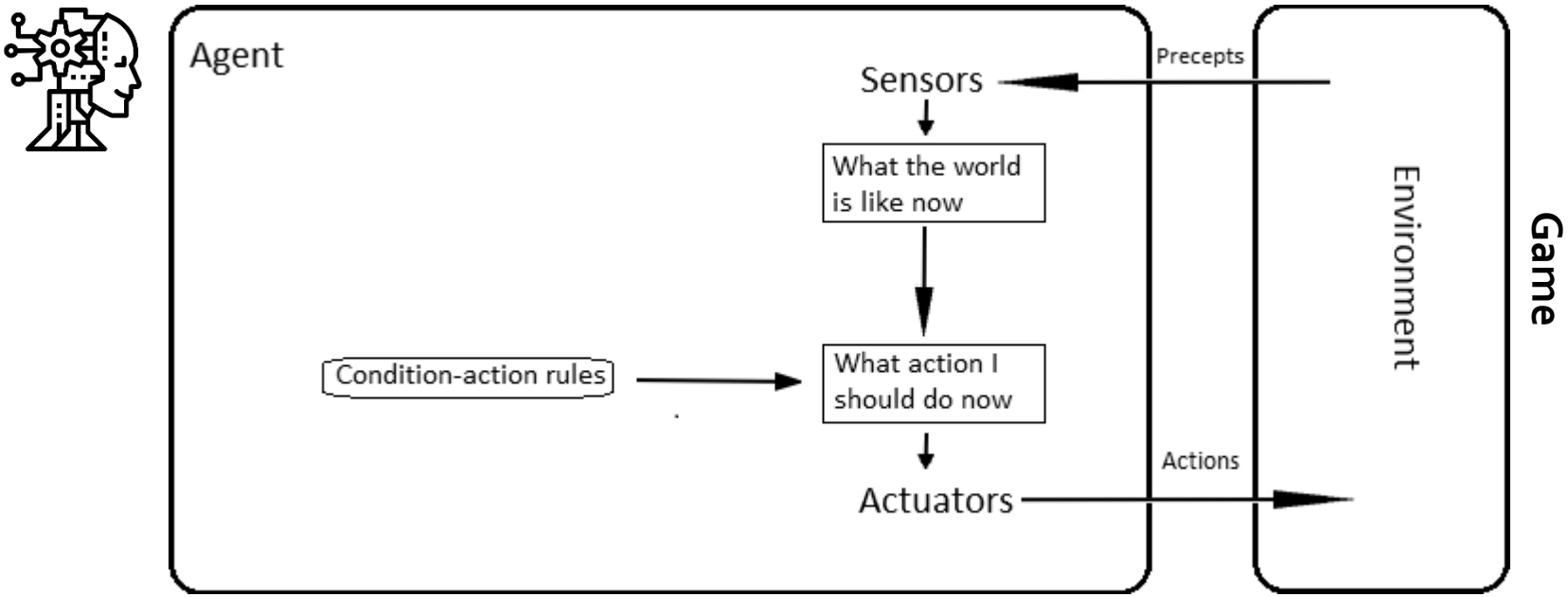


Image: By DDSniper - Originally fromStuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach p.49; with additions by me, CC0, https://commons.wikimedia.org/w/index.php?curid=14666503

Modeling AI Player

Short peek into intelligent agents

Model based, goal (driven) agent

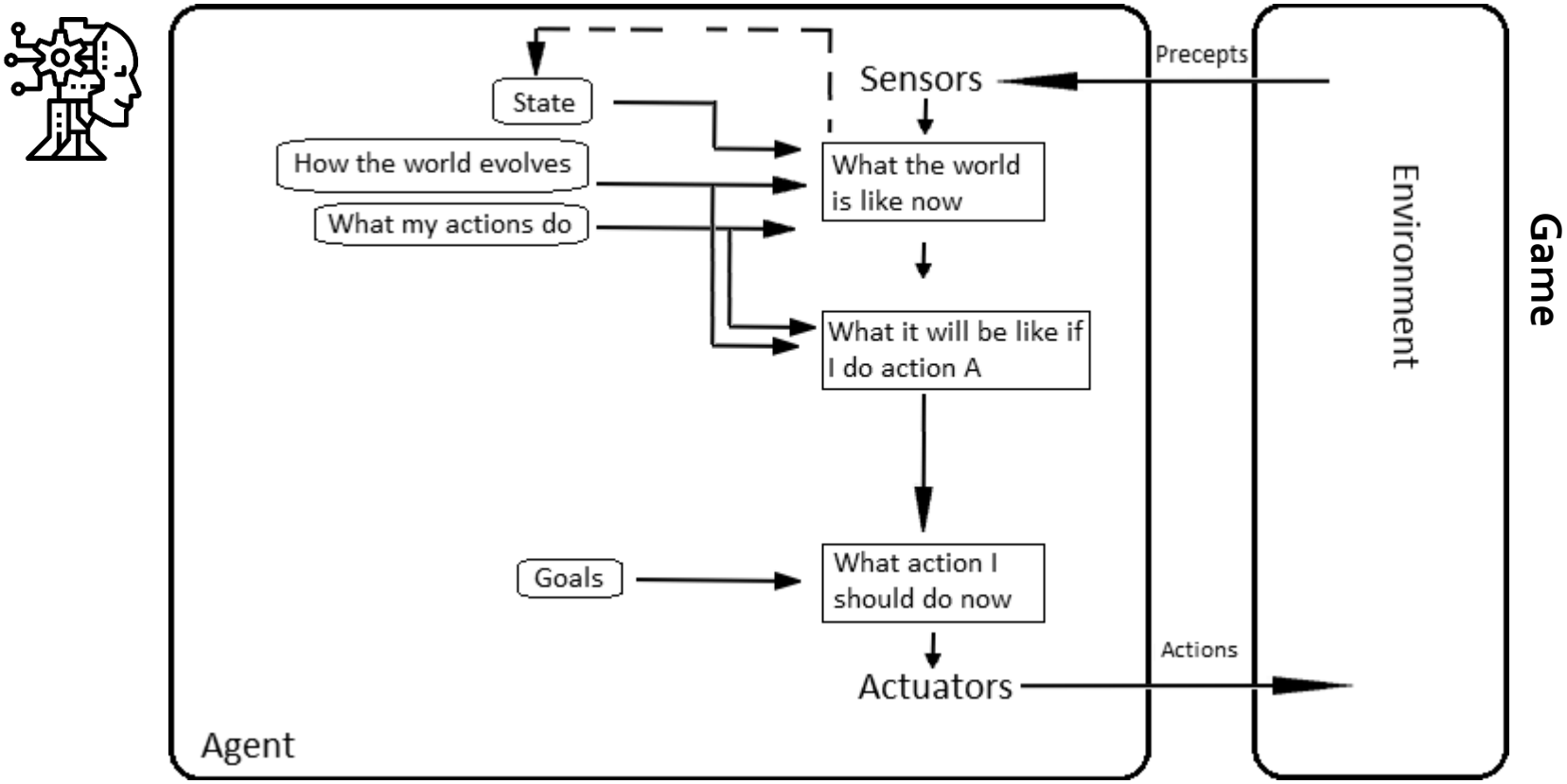
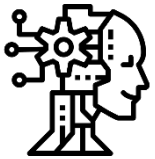


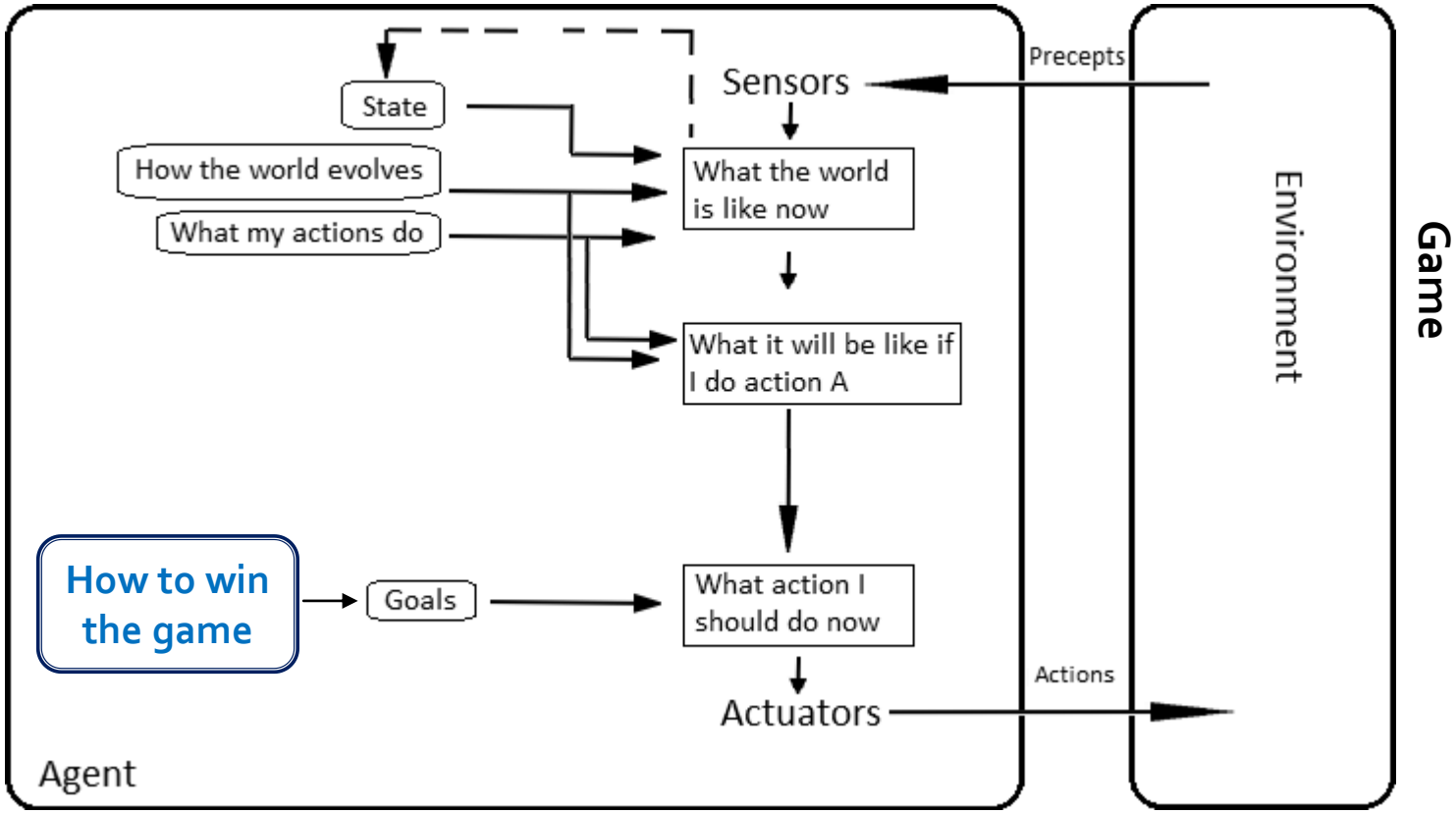
Image: By DDSniper - Originally from Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach p.51; with additions by me, CCo, https://commons.wikimedia.org/w/index.php?curid=14666524

Modeling AI Player

Short peek into intelligent agents

Model based, goal (driven) agent

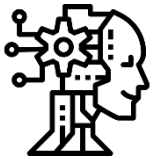


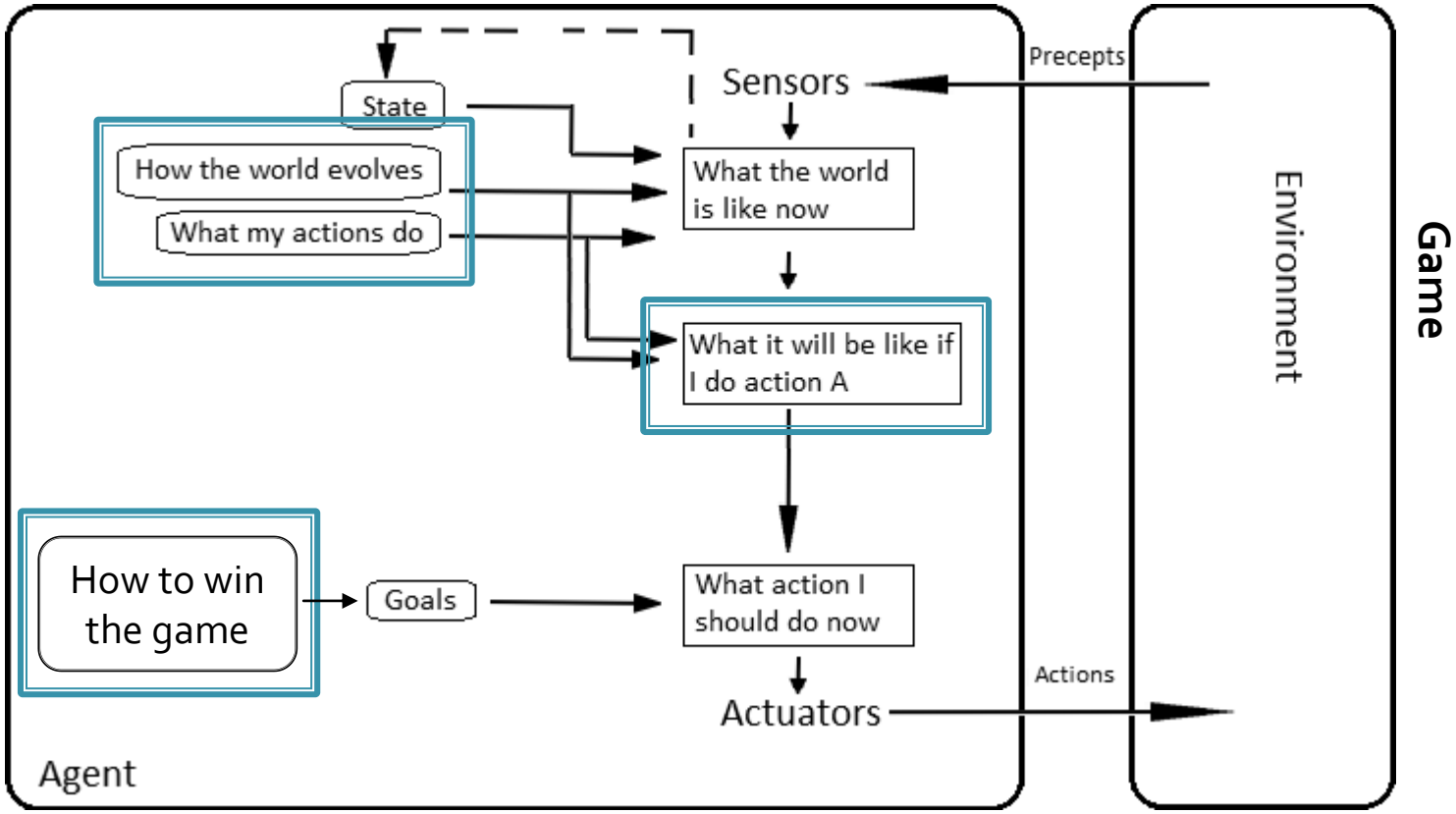


Modeling AI Player

Short peek into intelligent agents

Model based, goal (driven) agent





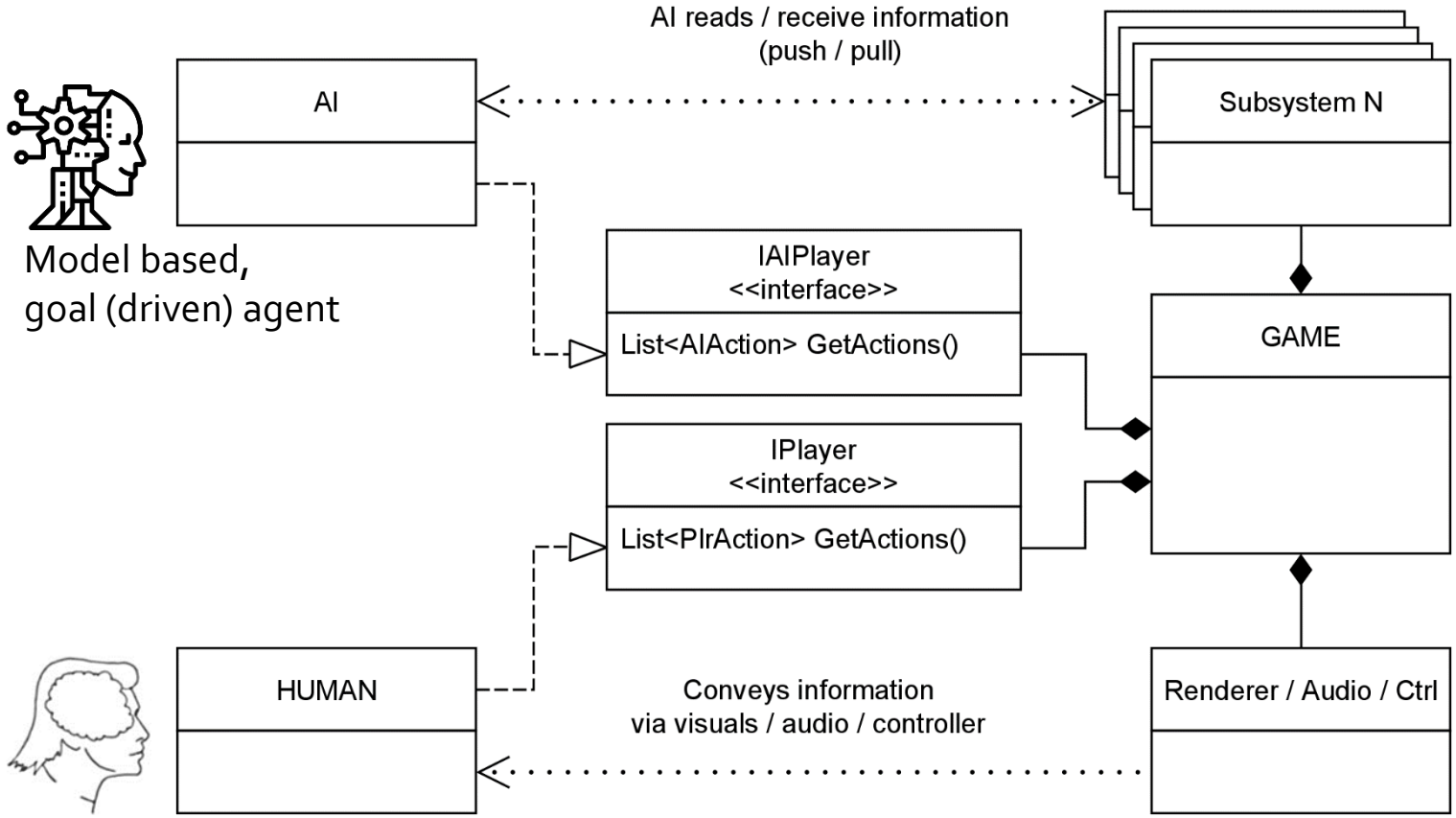
These are hard to answer / implement in general.

Modeling AI Player

UML-like abstraction

Why is it hard to implement blue boxes from previous slide in here?

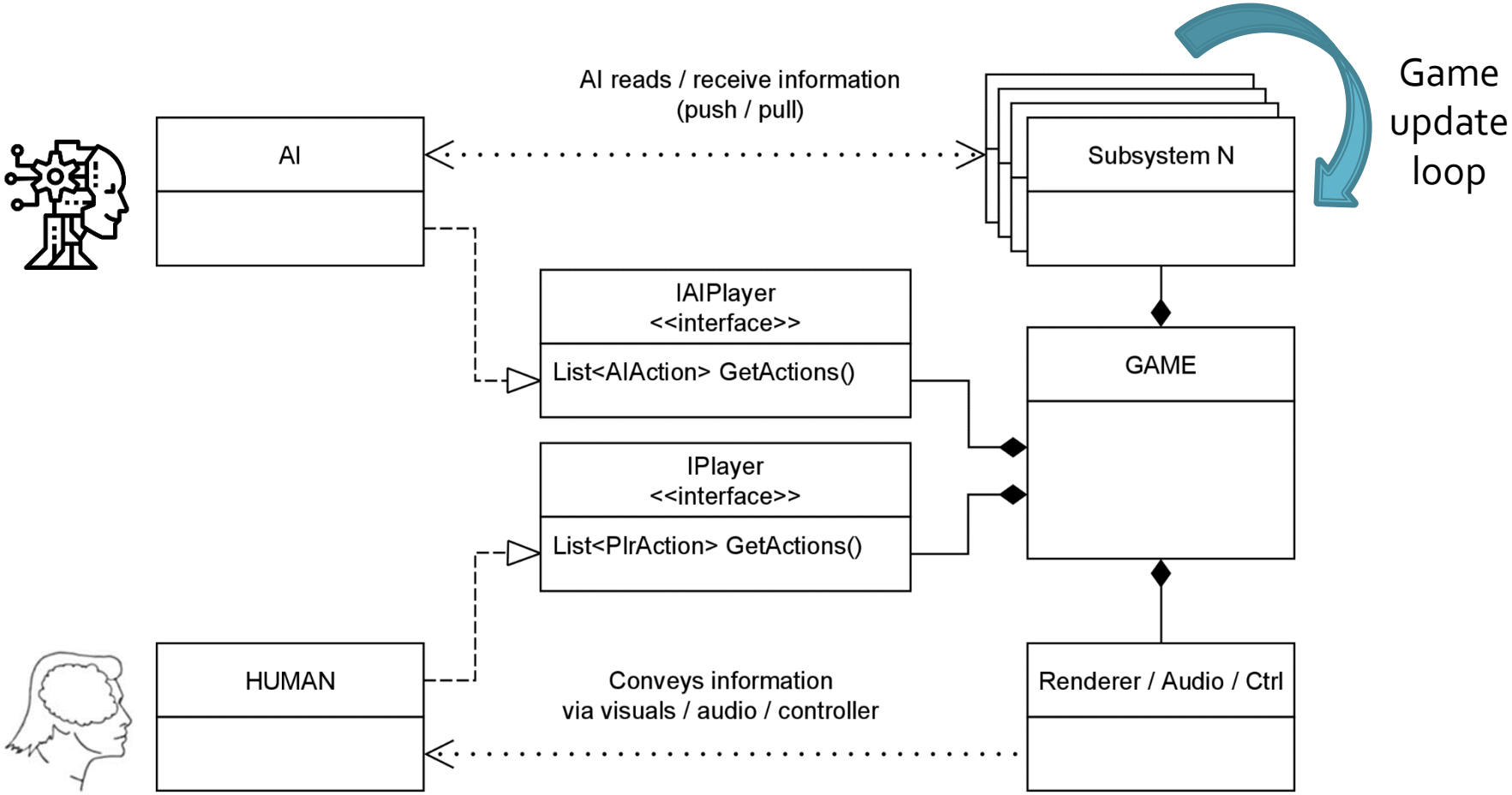
Version C



Modeling AI Play

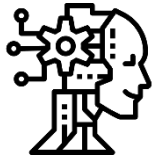
UML-like abstraction

The game update loop is complex!
Many subsystems, many state update functions in recurrent forms (== based on previous frame numbers), physics simulation in some games, etc. all which are not giving us any "predictions".



Mo
UML

Version C.2



GameModel
void Simulate(List<AI/PlrActions> actions)

Consults

Simulates

AI reads / receive information
(push / pull)

AI

Subsystem N

IAIPlayer
<<interface>>
List<AIAction> GetActions()

GAME

IPlayer
<<interface>>
List<PlrAction> GetActions()

HUMAN

Conveys information
via visuals / audio / controller

Renderer / Audio / Ctrl

Human Brain
void Simulate(List<AI/PlrActions> actions)

Learns and simulate

Modeling AI Player

Game Models

Game Model Types

- Simplified
  - In order to be able to search for at least something
- Forward
  - Complete simulation of the game, in its ideal form it is 100% accurate
- Learned
  - Based on observation, the model of the game is learned by the agent on its own

The Forward Model is the key!

Modeling AI Player

Forward model

Forward model can be conceptualized as functions:

Fwd.actions: $G \rightarrow P(A)$
Fwd.advance: $G \times P(A) \rightarrow G$
Fwd.state.clone

- G … set of all possible game states
- A … set of all actions (all players)

Modeling AI Player

Forward model

Forward model can be co
functions:

Given the game state, the forward model enumerates actions that can be taken by players.

Fwd.actions: $G \rightarrow P(A)$

Fwd.advance: $G \times P(A) \rightarrow G$

Fwd.state.clone

- game
- all pla

And for technical reason, we also need to be able to clone a game state.

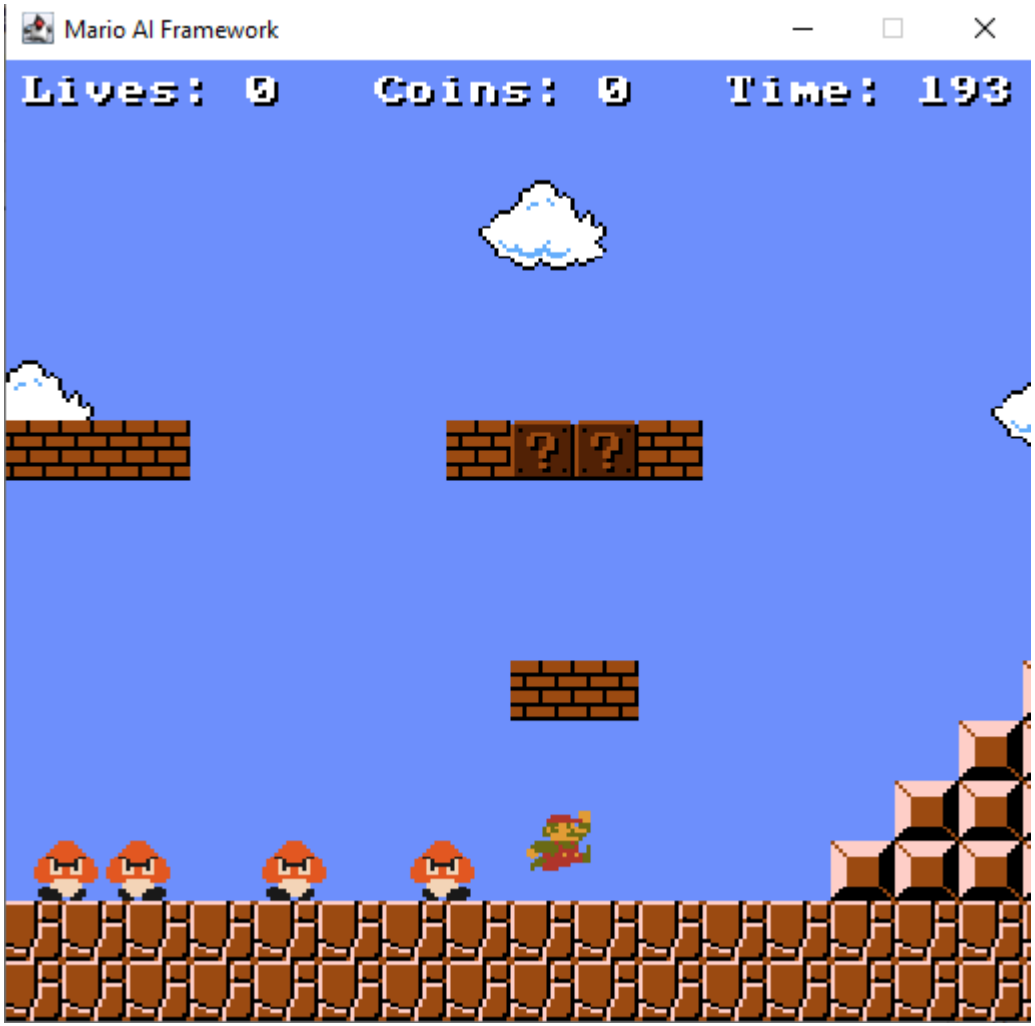
Given the game state and actions to be taken by the players, the forward model computes the next state.

Modeling AI Player

Forward model

Game Model Types

- Simplified
- Forward
- Learned



Super Mario is a single-player game!
All other entities have simple rule-based behavior and thus they can be encoded into the game's forward model "next" function.

A* Algorithm reminder (sort of)

https://www.youtube.com/watch?v=GC-nBgi9roU

A* Algorithm using Forward Model

Algorithm 1 A* algorithm using forward model

```
procedure A*(start, goal, fwdModel)
    fringe ← {start}
    visitedCosts ← init
    while fringe.size ≠ 0 and !timeout do
        curr ← fringe.popLowestCostState()
        if curr.isSatisified(goal) then
            return path
        end if
        for action : fwdModel.actions(curr) do
            next ← curr.clone()
            fwdModel.advance(next, action)
            next.cost ← pathCost(next) + heur(next)
            if visitedCosts[next] ≠ unknown then
                if next.cost >= visitedCosts[next] then
                    continue
                end if
            end if
            visitedCosts[next] ← next.cost
            fringe.add(next)
        end for
    end while
    if timeout then
        return path to the most promising state
    end if
    return fail                    ▷ Path to goal not found.
end procedure
```

Here we have a formulation of an A* search given a game forward model.

Algorithm 1 A* algorithm using forward model

```
1: procedure A*(start, goal, fwdModel)
2:     fringe ← {start}
3:     visitedCosts ← init
4:     while fringe.size ≠ 0 and !timeout do
5:         curr ← fringe.popLowestCostState()
6:         if curr.isSatisified(goal) then
7:             return path
8:         end if
9:         for action : fwdModel.actions(curr) do
10:            next ← curr.clone()
11:            fwdModel.advance(next, action)
12:            next.cost ← pathCost(next) + heur(next)
13:            if visitedCosts[next] ≠ unknown then
14:                if next.cost >= visitedCosts[next] then
15:                    continue
16:                end if
17:            end if
18:            visitedCosts[next] ← next.cost
19:            fringe.add(next)
20:        end for
21:    end while
22:    if timeout then
23:        return path to the most promising state
24:    end if
25:    return fail                    ▷ Path to goal not found.
26: end procedure
```

1: The search is initialized with initial position *start* (current state of the game), description of the *goal* state and simulator *fwdModel* that can advance the game state.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

2: Similar to BFS or Dijkstra, A* is holding so-called fringe (or an open list) of nodes that were not probed yet and constitutes nodes, through which the path to the goal can lead.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

3: A* maintains information about node's cost; in A* it's not only a distance from the *start* but also an estimation how far is the node from the *goal*.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

4: While we have not exhausted the options and we still have time to search (when game state is advanced because of real-time, the search might become obsolete!), perform respective A* iterations.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

5: Similar to Dijkstra's, A* always selects a node with the "least" cost. As the cost contains also an estimation of the distance to the goal, it directs A* to probe "promising" nodes.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

6: If selected node satisfies the *goal…*

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

7: … we won, and returns the path leading from *start* to the selected node *curr* (which is the *goal* we were looking for).

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

9: Otherwise, we iterate through all possible actions we can do in such a state.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

10: First, we clone the *curr* state of the game into *next*. That's because we're going to modify *next*.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

11: Then, we advance *next*, applying selected *action* we currently iterating with.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

12: Then, we compute (path) cost of such a state as combination of real cost reaching *next* from *start* adding an estimation of the cost reaching *goal* from *next*. We use some so-called heuristic function, specifically tailored for our problem.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

13: If we have already seen the *next* state before (i.e., we already know some path cast to *next*)...

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

14: … and the cost we currently computed for the *next* state is actually higher then previously found… meaning we have found a worse path to the *next* state…

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

15: ... we ignore it and continue with another action.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

18: Otherwise, it means we either see the *next* state for the first time or we have found a better path to the *next* state… so we note its cost …

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

19: … and add it into the fringe as a candidate to probe next.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

22: If we ever exit the main while cycle because of the timeout...

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

23: ...we return path to the most promising state (with the lowest cost) we found.

Algorithm 1 A* algorithm using forward model

```
1:  procedure A*(start, goal, fwdModel)
2:      fringe ← {start}
3:      visitedCosts ← init
4:      while fringe.size ≠ 0 and !timeout do
5:          curr ← fringe.popLowestCostState()
6:          if curr.isSatisified(goal) then
7:              return path
8:          end if
9:          for action : fwdModel.actions(curr) do
10:             next ← curr.clone()
11:             fwdModel.advance(next, action)
12:             next.cost ← pathCost(next) + heur(next)
13:             if visitedCosts[next] ≠ unknown then
14:                 if next.cost >= visitedCosts[next] then
15:                     continue
16:                 end if
17:             end if
18:             visitedCosts[next] ← next.cost
19:             fringe.add(next)
20:         end for
21:     end while
22:     if timeout then
23:         return path to the most promising state
24:     end if
25:     return fail                    ▷ Path to goal not found.
26: end procedure
```

25: Otherwise, we've managed to search through all possible states and have not found the *goal*.

Important Forward Model Functionalities

Algorithm 1 A* algorithm using forward model

```
procedure A*(start, goal, fwdModel)
    fringe ← {start}
    visitedCosts ← init
    while fringe.size ≠ 0 and !timeout do
        curr ← fringe.popLowestCostState()
        if curr.isSatisified(goal) then
            return path
        end if
        for action : fwdModel.actions(curr) do
            next ← curr.clone()
            fwdModel.advance(next, action)
            next.cost ← pathCost(next) + heur(next)
            if visitedCosts[next] ≠ unknown then
                if next.cost >= visitedCosts[next] then
                    continue
                end if
            end if
            visitedCosts[next] ← next.cost
            fringe.add(next)
        end for
    end while
    if timeout then
        return path to the most promising state
    end if
    return fail                    ▷ Path to goal not found.
end procedure
```

<u>Three requirements</u> on the forward model:

1. **Clone** – being able to duplicate its state;
2. **Actions** – given the state, enumerate all possible actions for the player;
3. **Advance** – given the state and (an) action(s) of (a) player(s), compute next game state.

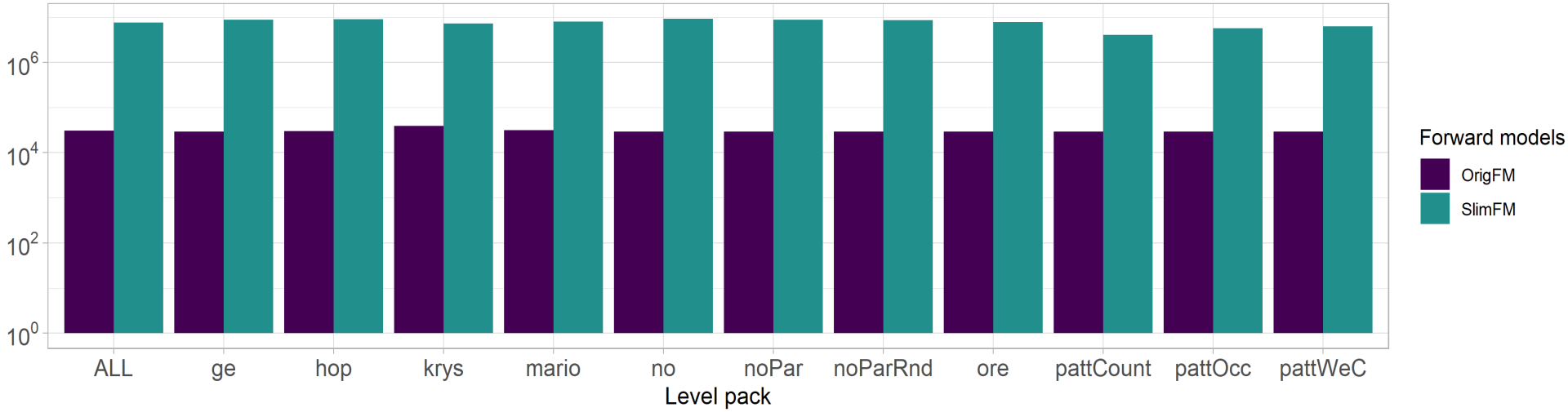
Why implementation matters?

SuperMario A* Agent

How we developed new SOTA agent

Avg. number of clones per second.

Higher is better. Log-scale!



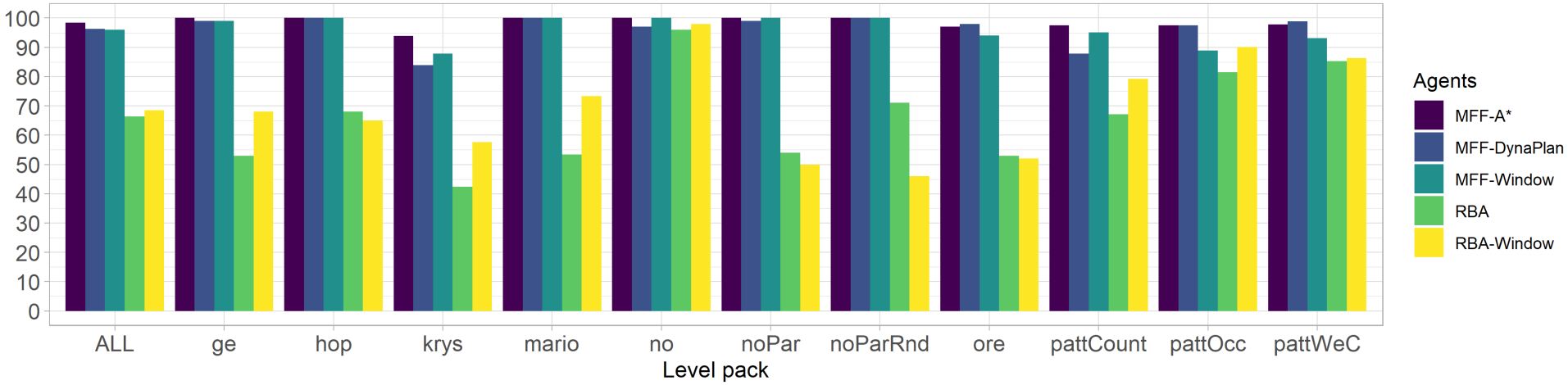
D. Šosvald, M. Topfer, J. Holan, V. Černý, J. Gemrot, "Super Mario A-Star Agent Revisited, in IEEE International Conference on Tools with Artificial Intelligence, 2021, *in press*

SuperMario A* Agent

How we developed new SOTA agent

% of levels finished given the level pack

Higher is better.



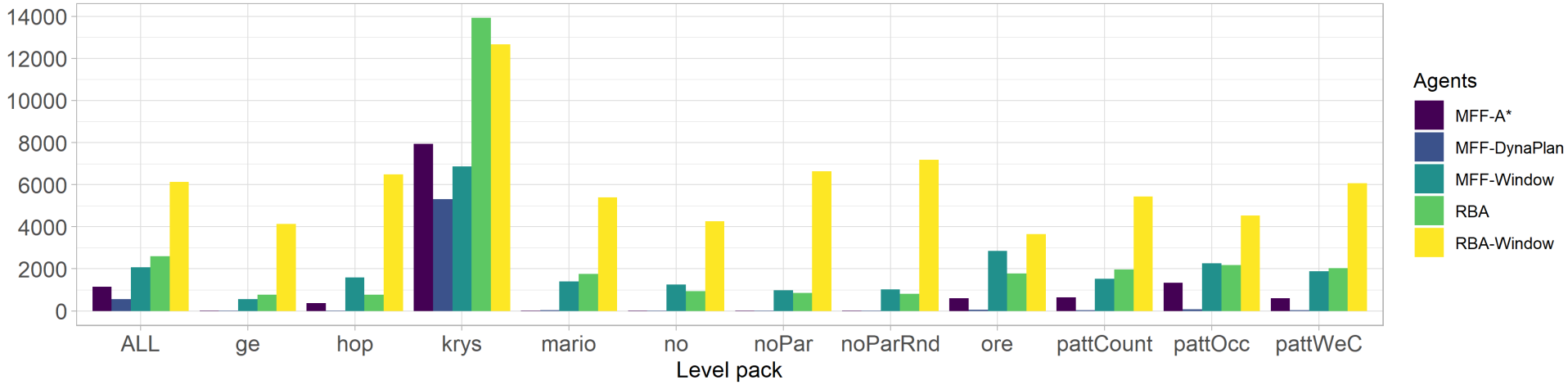
D. Šosvald, M. Topfer, J. Holan, V. Černý, J. Gemrot, "Super Mario A-Star Agent Revisited, in IEEE International Conference on Tools with Artificial Intelligence, 2021, *in press*

SuperMario A* Agent

How we developed new SOTA agent

Avg. runtime of agent in ms given the level pack

Lower is better.



D. Šosvald, M. Topfer, J. Holan, V. Černý, J. Gemrot, "Super Mario A-Star Agent Revisited, in IEEE International Conference on Tools with Artificial Intelligence, 2021, *in press*

That’s all folks!

Faculty of Mathematics and Physics
Charles University
5.10.2022

GAME DEVELOPMENT @CUNI.CZ

matfyz

NAIL122 AI for Games

How to tackle AI for (not only) 4X games? A teaser…

Jakub Gemrot

AI in Games…

Why?

Would it be a game

without …

Source: https://wall.alphacoders.com/big.php?i=324425

Would it be a game

without …

Adapted from: https://wall.alphacoders.com/big.php?i=324425

Would it be a game

if you have to ...

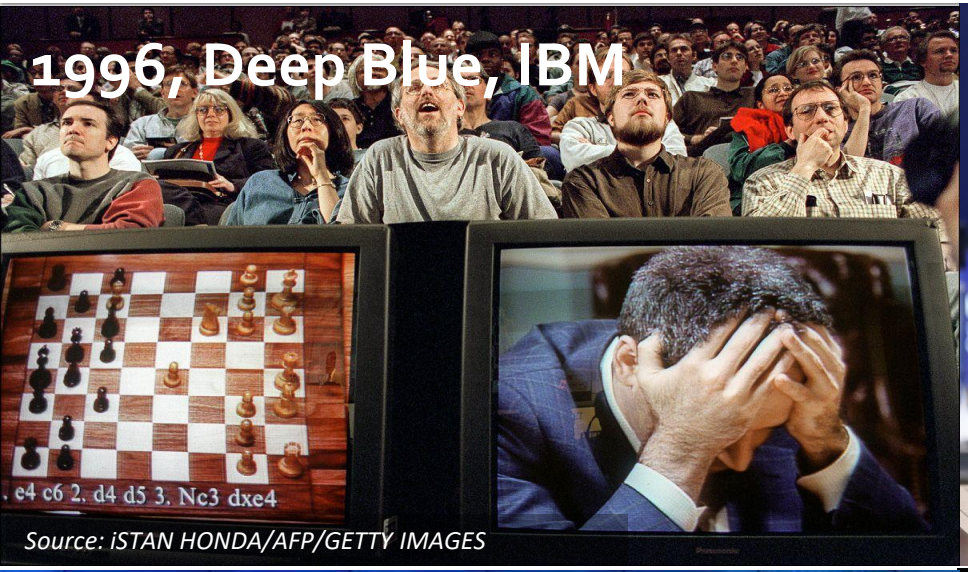
Would it be a game

if you have to …

Games are still used as a source of interesting (and hard) problems.

Mastering Games

Means to compete with humankind



Source: iSTAN HONDA/AFP/GETTY IMAGES

Source: AP/Lee Jin-man

Source: ihttps://www.inverse.com/article/13630

*Source: GETTY IMAGES/NICOLAS_*

Go 19x19

Legal positions ~ 10^{172}
Avg. game length ~ 150 plies

AlphaZero (is claimed) to require >4.6yr of TPU time to master it.
Learned in 8hr on a farm of 5064 TPUs.

Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." arXiv preprint arXiv:1712.01815 (2017).

What's on the game horizon now?

Modern Computer Games

Now for "real" problems…

Modern Computer Games

Now for "real" problems…

Stochastic
Partially observable
Simultaneous
(Real-time)
Huge game trees

Modern Computer Games

Now for "real" problems…

Stochastic
Partially observable
Simultaneous
(Real-time)
Huge game trees

=> Fun to play!

Star Craft II

Targeted by Google, OpenAI, Facebook

"… as a pro, I can still spot some of the system's weaknesses."
GRZEGORZ "MANA" KOMINCZ

Even (supervised) AlphaStar has not reached superhuman level.

(more on that later)

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*(7782), 350-354.

Even if it does… how many game development companies has access to or can afford to use
Google farms and their TPUs?

… worldwide

… in Europe

… in Czech Republic

… your future company?

Star Craft: Brood War

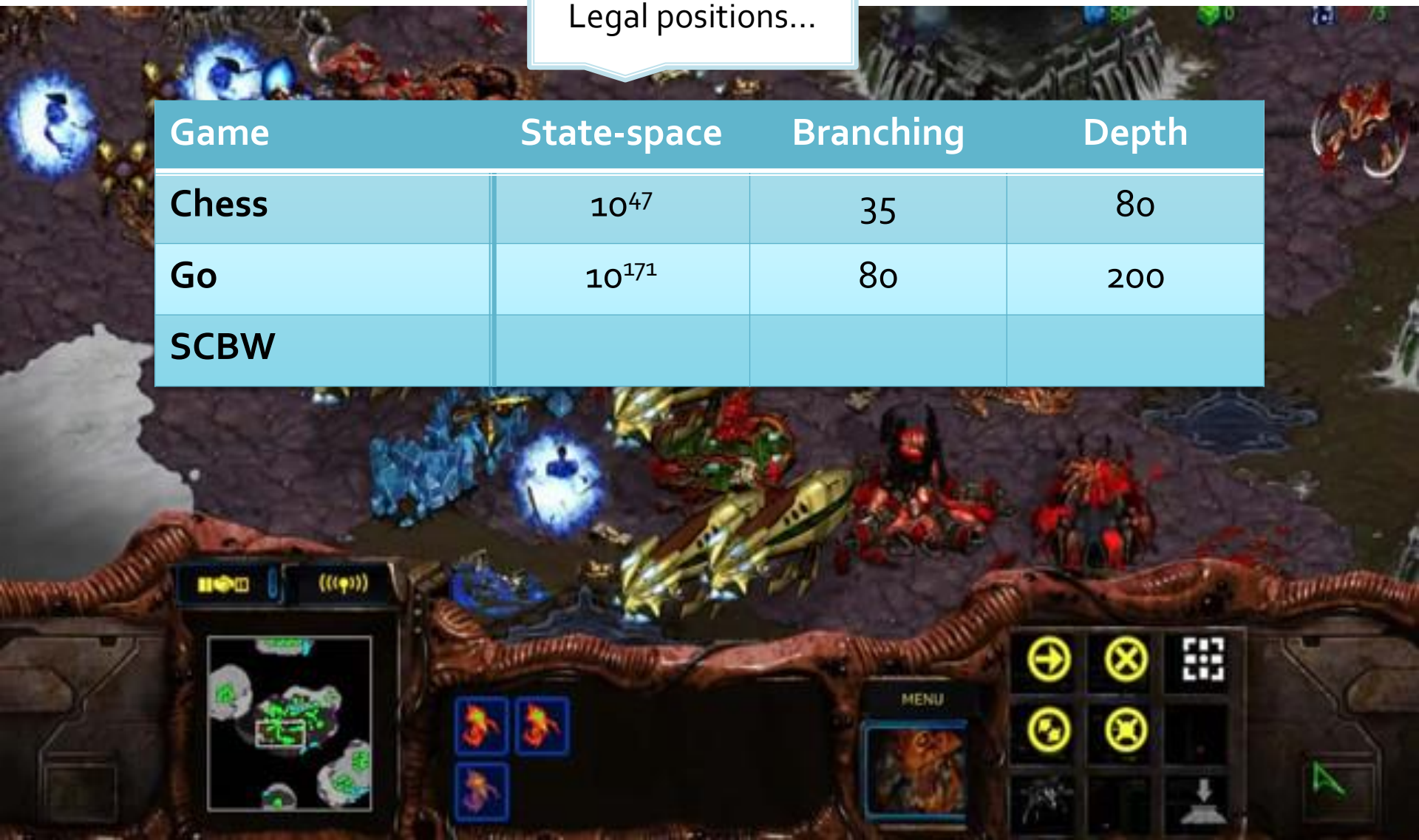
In scope of various unis for some time now

Star Craft: Brood War

Game size?

Legal positions…

Game	State-space	Branching	Depth
Chess	10^{47}	35	80
Go	10^{171}	80	200
SCBW			



Star Craft: Brood War

Game size?

Let's switch to simple/imprecise estimations to keep the comparison fairer.

Game	State-space	Branching	Depth (pl.)
Chess	10^{57}	~35	~80
Go	10^{430}	~250	~211
SCBW			

Star Craft: Brood War

Game size?

Game	State-space	Branching	Depth (pl.)
Chess	10^{57}	~35	~80
Go	10^{430}	~250	~211
SCBW	10^{1685}		

StarCraft map: 128x128
Maximum number of units: 400

Considering only unit positions:
$(128x128)^{400}=16384^{400}\approx10^{1685}$

Star Craft: Brood War

Game size?

Game	State-space	Branching	Depth (pl.)
Chess	10^{57}	~35	~80
Go	10^{430}	~250	~211
SCBW	10^{1685}	9^4 - 10^{200}	

Units: 4 – 200

Actions per unit: 10

(wait, move 8 dirs, attack, power, …)

Branching factor: 9^4 - 10^{200}

Star Craft: Brood War

Game size?

Game	State-space	Branching	Depth (pl.)
Chess	10^{57}	~35	~80
Go	10^{430}	~250	~211
SCBW	10^{1685}	9^4 - 10^{200}	36000

Length of a game: 25 minutes

25 min x 60 sec x 24 iteration/sec = 36000

How to tackle such a space?

How to tackle such a space?

Simplify the space

Star Craft: Brood War

Divide et impera

AI is not solving everything at once but splits the play into the set of subproblems (a.k.a. expert approach)

Layers of control

- Strategic – Army/Base level
  - Build, research, muster, expand, manage groups
- Tactical – Group level
  - Move, attack, siege, defend
- Reactive – Unit Level
  - Engage, withdraw, use ability

STRATEGIC LAYER (SCBW)

Star Craft: Brood War

Strategic Layer

- Not addressed much
- Partial observability is a big problem as the first encounter with the enemy is done usually after 2-4 minutes (depth 2 880 – 5 760)
- Even though we have a lot of replays, if you consider the number of maps, combination of races and different initial positions, the data set is not big enough for each bucket to really do ML
- Human players have already converged to many viable opening strategies for each map

Star Craft: Brood War

Strategic Layer

"Poor man's solution"

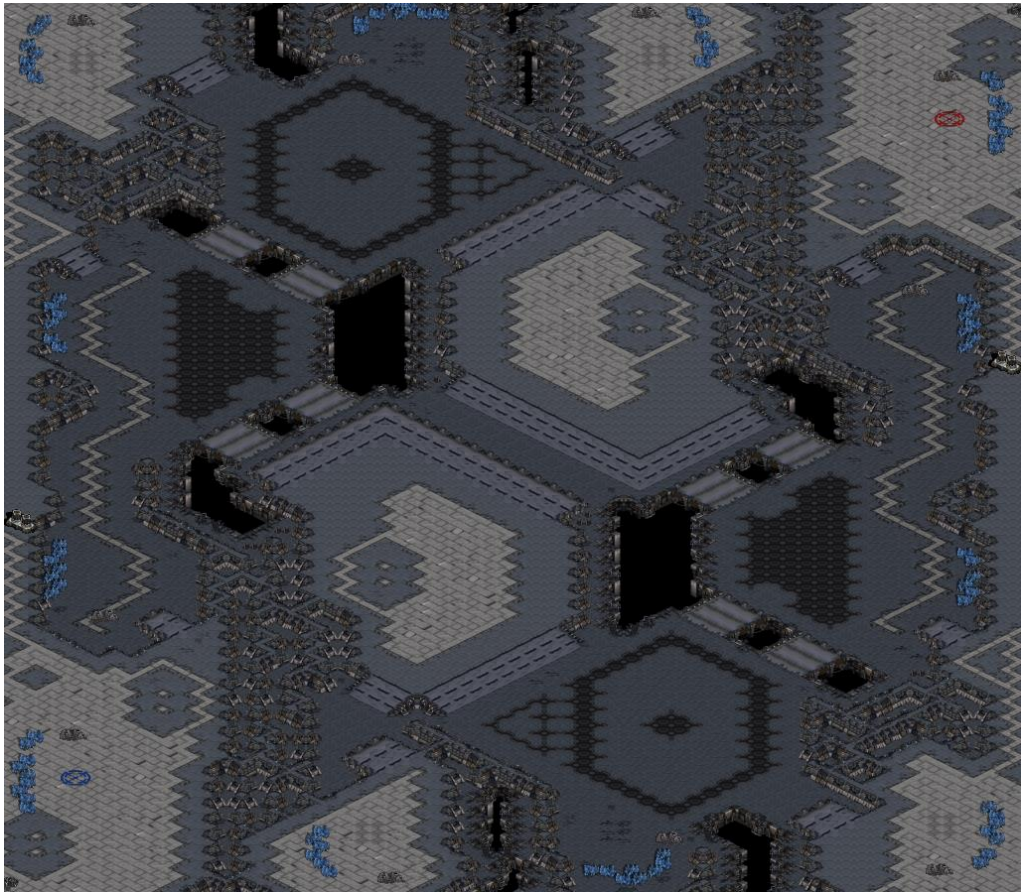
- Pick existing strategy and implement its build order via rule-based systems
- Zerg: 6-pool rush, Lurker rugh, Mutarush, …
- Terran: Bunker-push, Tank-push, …
- Protoss: Zealot rush, Photon cannon rush, …
- Suitability depends on the map and initial base positions
- Typically, each bot implements one to a few strategies (state in ~2018)

TACTICAL LAYER (SCBW)

Star Craft: Brood War

Tactical layer

Abstraction
for map Benzene



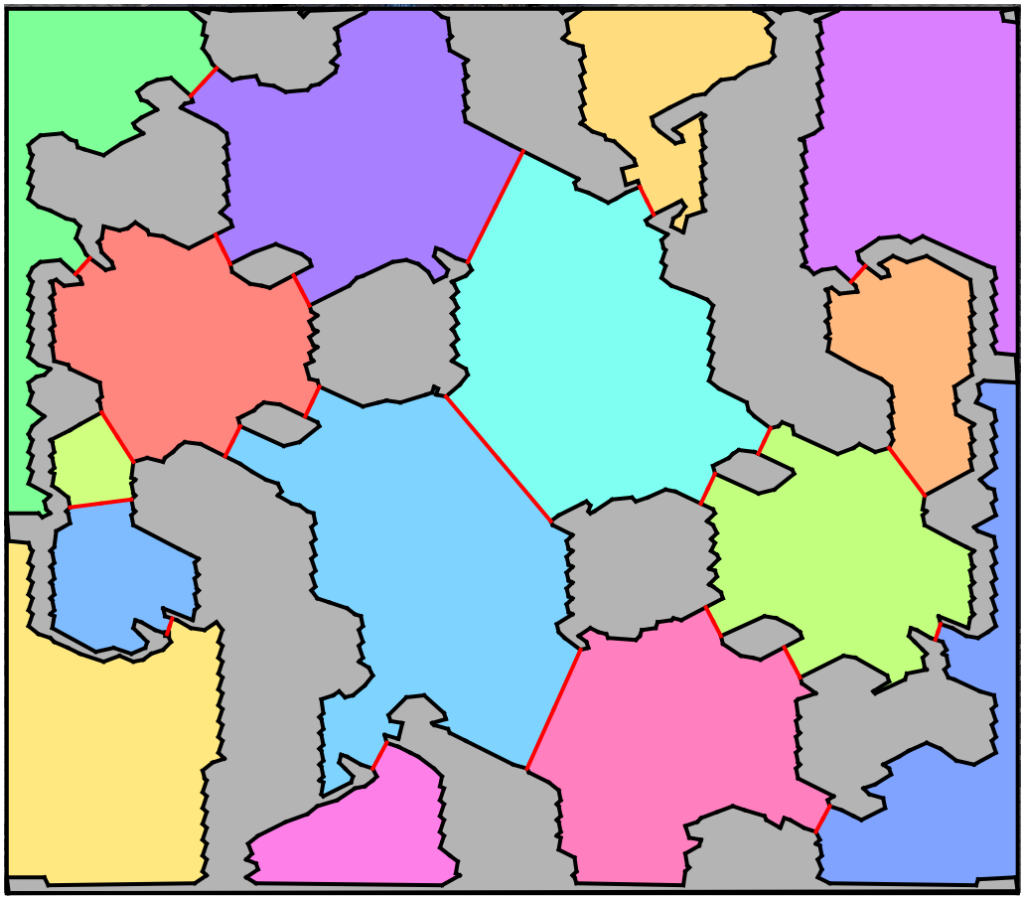
Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer

Abstraction
for map Benzene

Perkins' algorithm
to decompose
a map into
regions
and **chokepoints**.
Params sensitive.



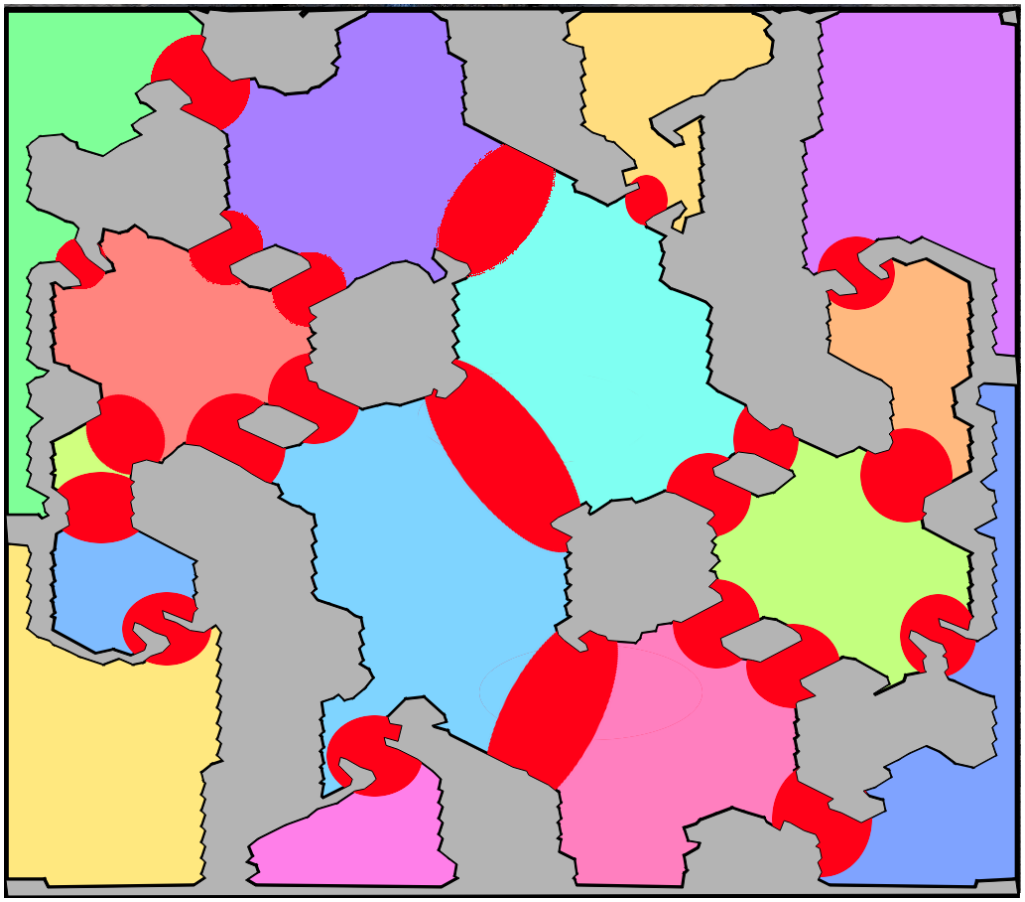
Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer

Abstraction
for map Benzene

Chokepoints (20)
are dividing
regions (15).



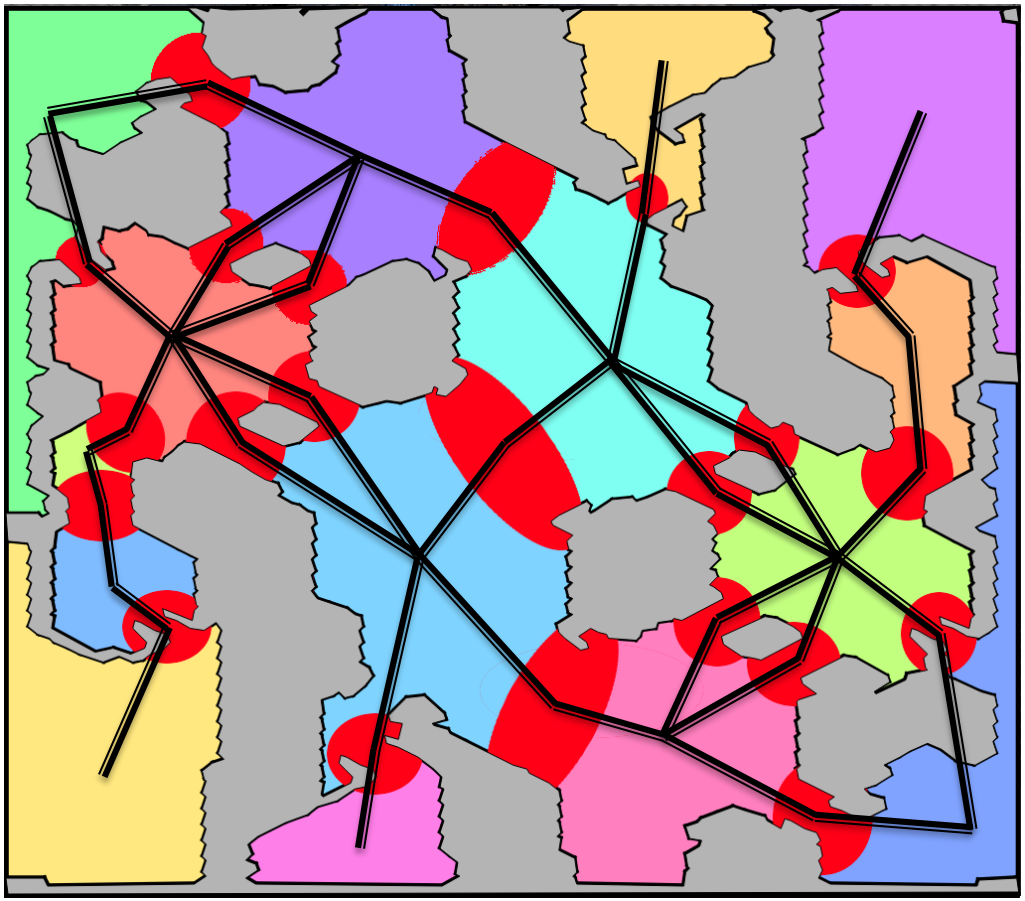
Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer

Abstraction
for map Benzene

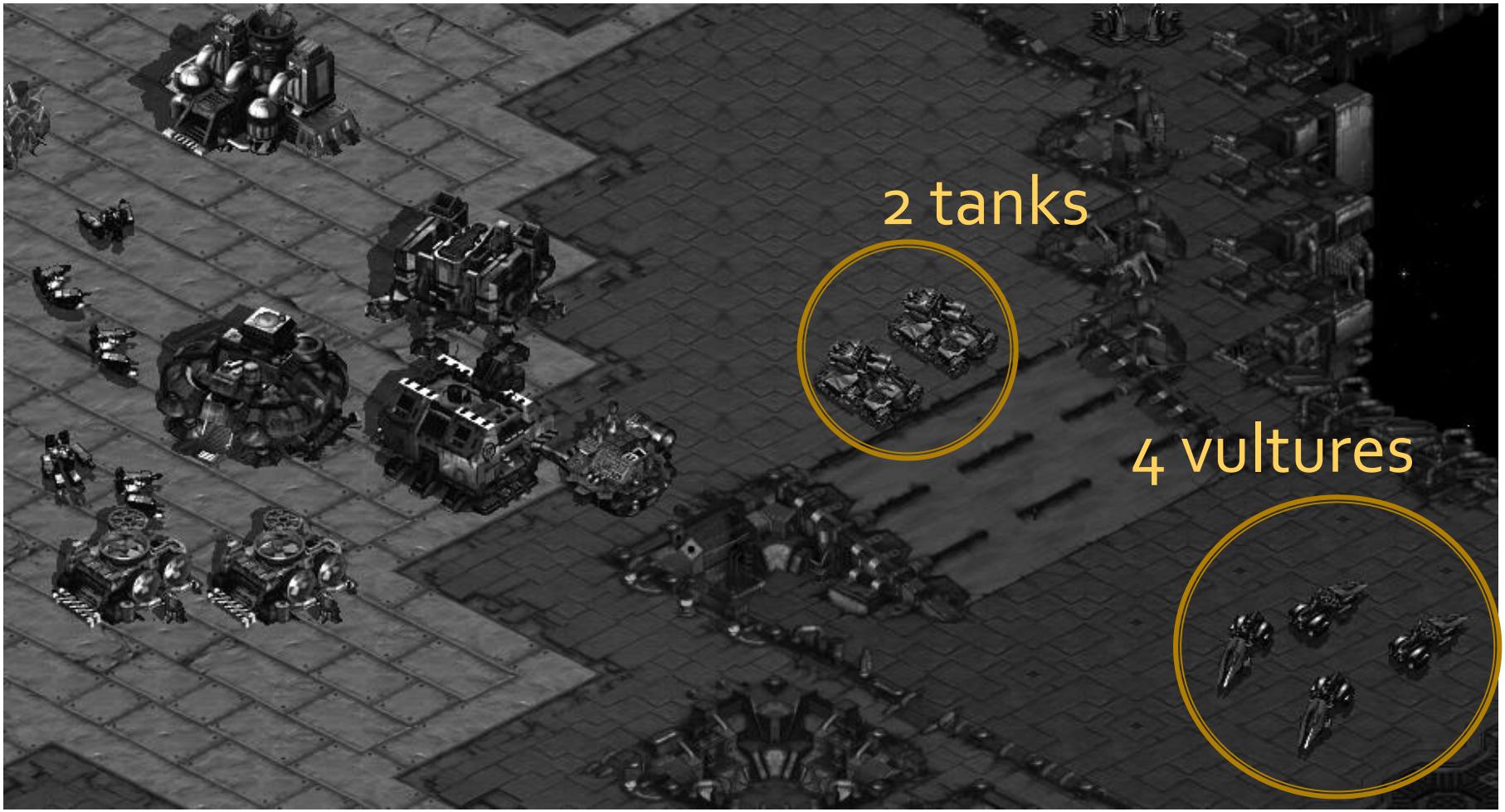
Distance matrix
precomputed
between regions.
(Mind the air units.)



Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

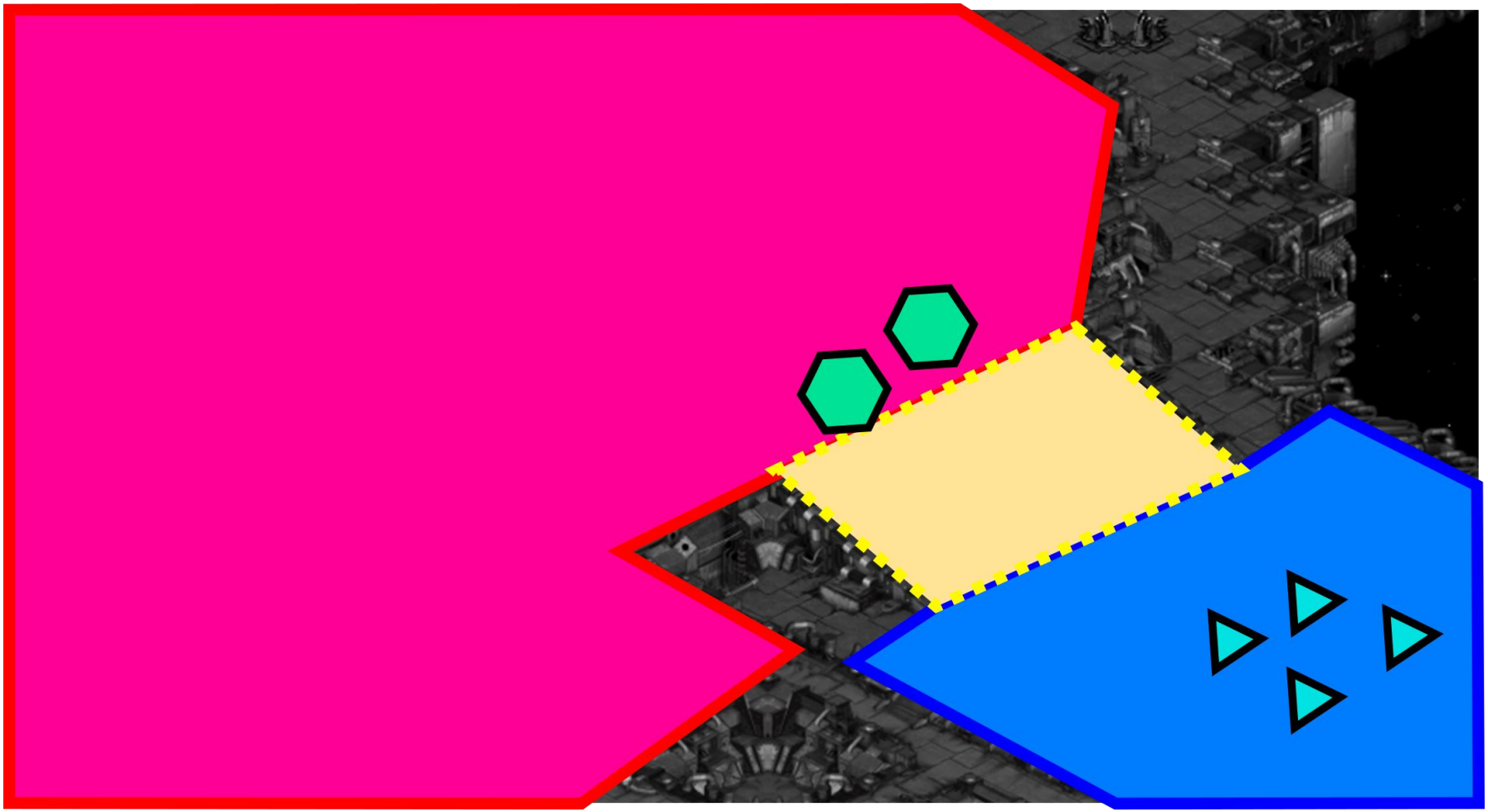
Tactical layer



Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

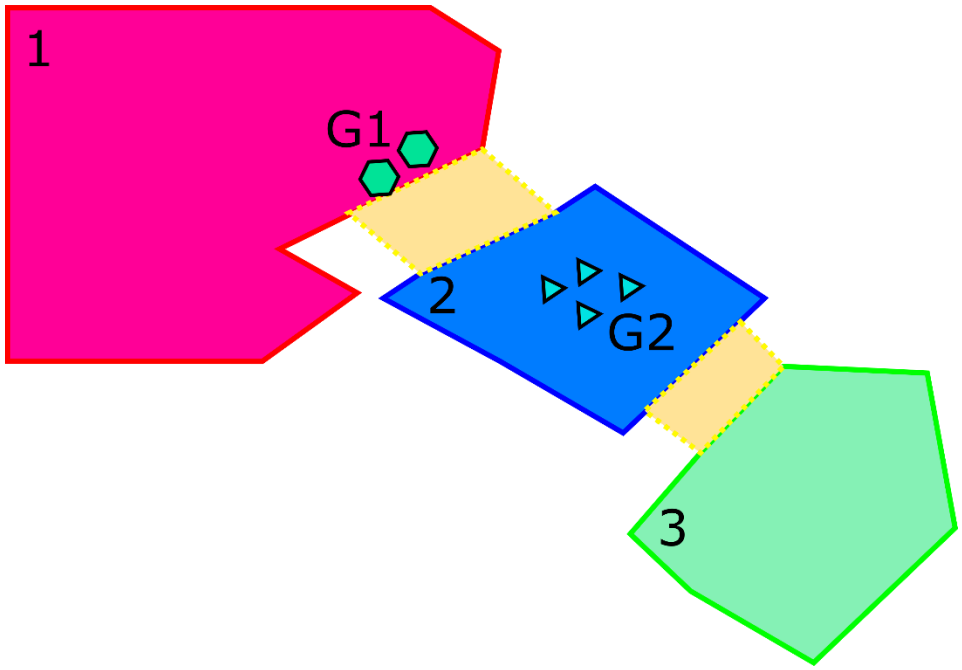
Tactical layer



Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer



G1:

move 2, idle

G2:

move 1, move 3, idle

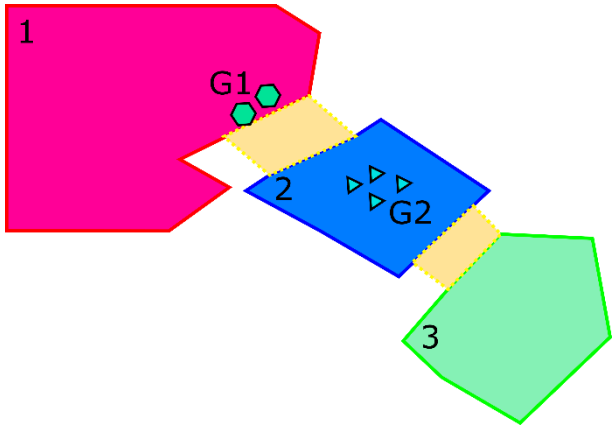
=> Branching factor 6

Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer

SCBW player is managing about 8 groups.



Avg.# of region links ~ 4

⇒ 4 move + 1 idle action

⇒ $5^8 = 4.10^5$ branching factor in a late game phase

Much smaller during early/mid game phases.

Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

Star Craft: Brood War

Tactical layer

Search is doable

- Alpha-Beta Considering Durations (ABCD)
- MCTS Considering Durations (MTCSCD)

(discussed later)

Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

REACTIVE LAYER (SCBW)

Star Craft: Brood War

Reactive layer

Star Craft: Brood War

Reactive layer

Star Craft: Brood War

Reactive layer

Red player: 22 units (4 stationary)

Possible actions (roughly):

$$9^{18} \times 8^{4} \sim 10^{20}$$

Blue payer: 47 unit

Possible actions (roug.):

$$9^{47} \sim 10^{87}$$

Local "search" to prune the action space.

Star Craft: Brood War

Reactive layer

Action space -> Script space

Instead of actions, we use scripts.

Churchill, David, Abdallah Saffidine, and Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios." *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012.

Star Craft: Brood War

Reactive layer

Action space -> Script space

Instead of actions, we use scripts.

`Script: S -> A`

For a given state $s \in S$, it gives an action $a \in A$ to perform. Usually O(N), where N is number of actions for a unit.

Churchill, David, Abdallah Saffidine, and Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios." *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012.

Star Craft: Brood War

Reactive layer

Action space -> Script space

Instead of actions, we use scripts.

`Script: S -> A`

For a given state $s \in S$, it gives an action $a \in A$ to perform. Usually O(N), where N is number of actions for a unit.

Closest	attack closest unit
Kiting	attack closest unit then escape
AV	attack highest `dpf(u)/hp(u)`
NOK-Closest	attack closest unit if not received lethal dmg already
NOK-AV	NOK but attack via AV
Kiting-AV	hit and run, choose target via AV
Kiting-NOK-AV	kiting but choose NOK-AV

Churchill, David, Abdallah Saffidine, and Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios." *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012.

Star Craft: Brood War

Reactive layer

Action space -> Script space

Instead of actions, we use scripts.

`Script: S -> A`

For a given state $s \in S$, it gives an action $a \in A$ to perform. Usually O(N), where N is number of actions for a unit.

Closest	attack closest unit
Kiting	attack closest unit then escape
AV	attack highest `dpf(u)/hp(u)`
NOK-Closest	attack closest unit if not received lethal dmg already
NOK-AV	NOK but attack via AV
Kiting-AV	hit and run, choose target via AV
Kiting-NOK-AV	kiting but choose NOK-AV

Exploitability By search

Random	Weakest	Closest	AV	Kiter	Kite-AV	NOK-AV
1.00	0.98	0.98	0.98	0.97	0.97	0.95

Churchill, David, Abdallah Saffidine, and Michael Buro. "Fast Heuristic Search for RTS Game Combat Scenarios." *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2012.

Star Craft: Brood War

Reactive layer

Red player: 22 units (4 station.)

Possible actions (**low-level**):

$$9^{18} \times 8^{4} \sim 10^{20}$$

Blue payer: 47 unit

Possible actions (**low-l.**):

$$9^{47} \sim 10^{87}$$

Local "search" to prune the action space.

Star Craft: Brood War

Reactive layer

Red player: 22 units (4 station.)

Possible actions (**2 scripts**):

$$2^{18} = 262\,144$$

Blue payer: 47 unit

Possible actions (**2 scr.**):

$$2^{47} \sim 10^{53}$$

Star Craft: Brood War

Reactive layer

Red player: 22 units (4 station.)
Possible actions (**2 scripts**):

$$2^{18} = 262\,144$$

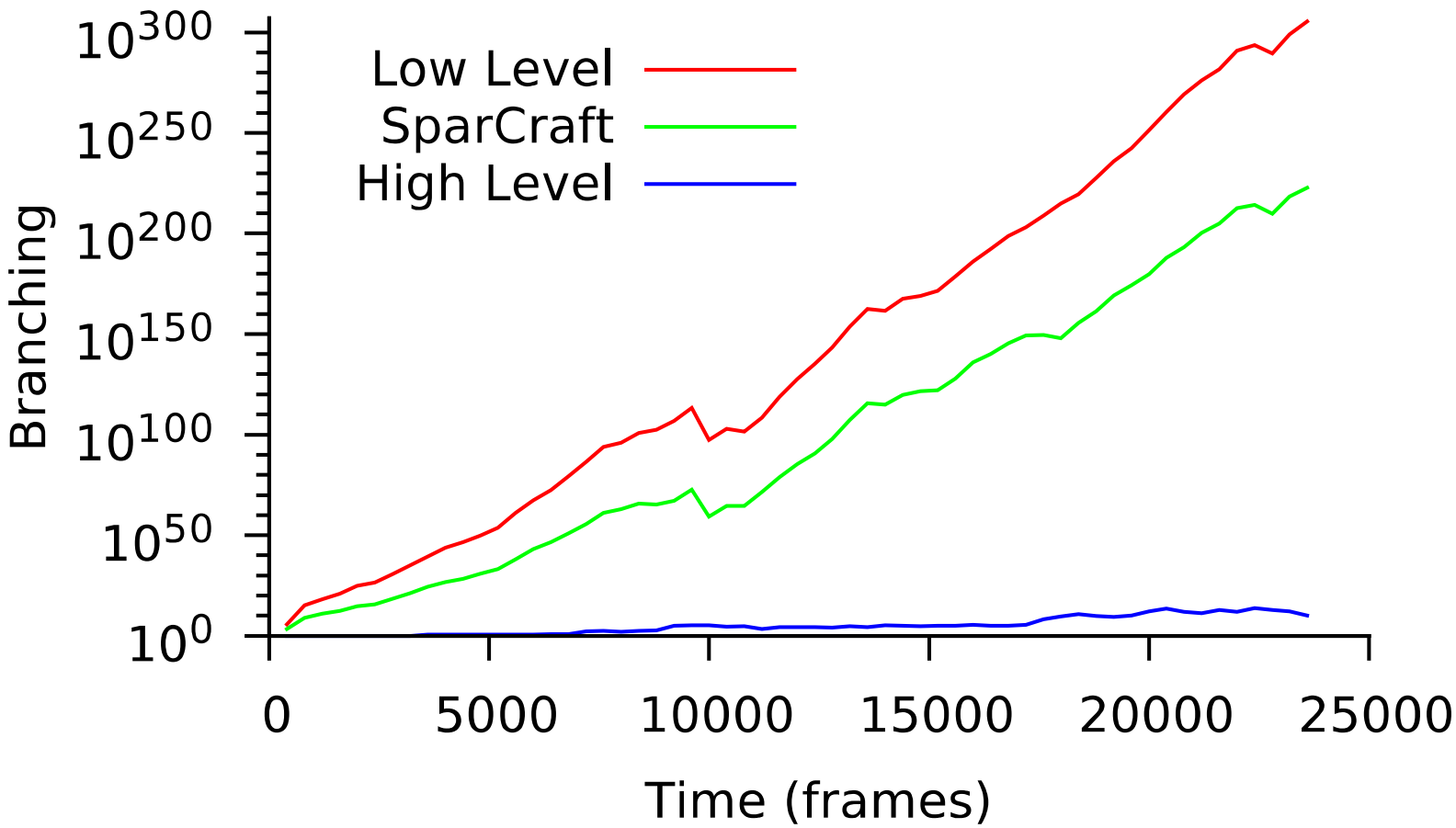
Blue payer: 47 unit
Possible actions (**1 scr.**):

1

Evaluating a script costs non-trivial time, typically O(N)!

Star Craft: Brood War

Tactical vs. Reactive Layer



Uriarte, Alberto, and Santiago Ontañón. "Game-tree search over high-level game states in RTS games." *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

So what’s this course is going to be about?

Difficult problems!

Without definite answers…

NAIL122 AI for Games

- Course web page: shorturl.at/vyORT
- Discord: https://discord.gg/c49DHBJ

- Algorithms for symbolic real-time search
  - Not only for 4X games…
- Detour … F.E.A.R. AI

- There will be an exam in the end (algos)
- Lab credit for implementation of some real-time search algorithms + its evaluation and the most probably w/ game!

NAIL122 AI for Games

Deserves your attention?

E-mail: gemrot@gamedev.cuni.cz

Faculty of Mathematics and Physics
Charles University
19.10.2022

NAIL122 AI for Games

F.E.A.R. AI and Planning

Jakub Gemrot

F.E.A.R.™
FIRST ENCOUNTER ASSAULT RECON

F.E.A.R.

Let's check on it's AI

https://www.youtube.com/watch?v=KQN3yKYkFmE

F.E.A.R.

What we have seen

AI that can:

- Advance and shoot (obviously)
- Take cover / Retreat
- Flank you
- Flush you with grenade
- Hold position
- Use environment (the cabinet)
- Detect flash-light
- Chat over radio

Orkin, J. (2006, March)
Three states and a plan: the AI of FEAR
Game developers conference (Vol. 2006, p. 4)

F.E.A.R.

Three states of AI

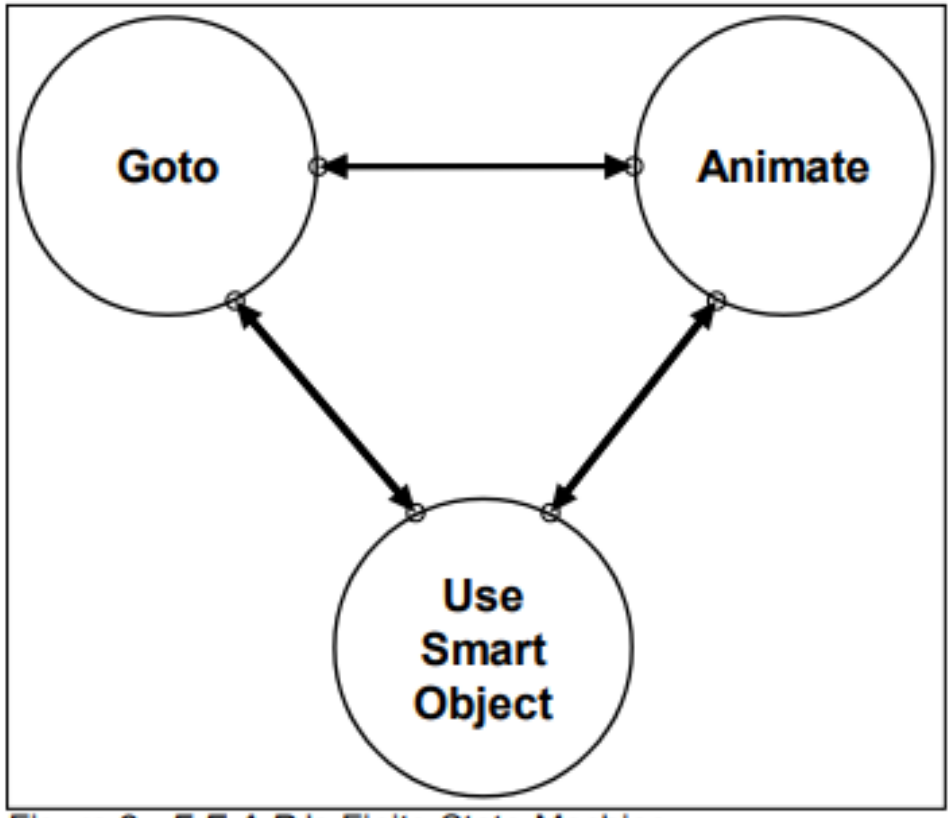


Figure 2: *F.E.A.R.*'s Finite State Machine

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

F.E.A.R.

Three states of AI

Path-finding through navigation mesh.

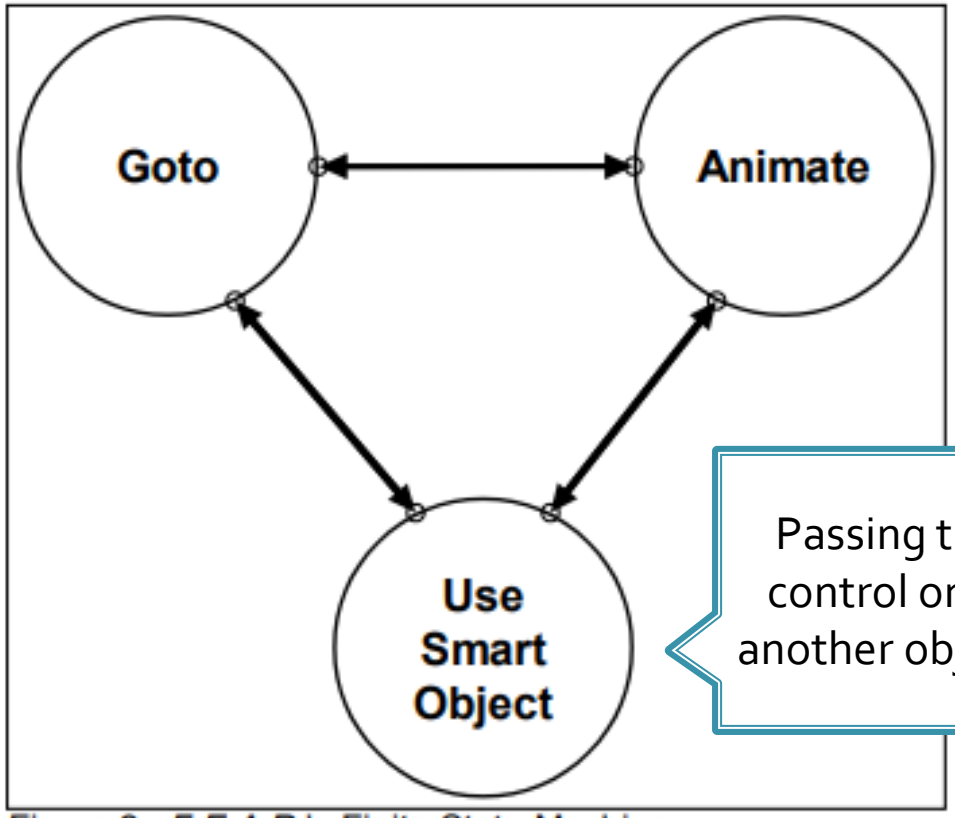


Figure 2: *F.E.A.R.*'s Finite State Machine

Running uninterruptible animation.

Passing the control onto another object.

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

Navigation Mesh

(peeking into NAIL068)

Environment Abstraction

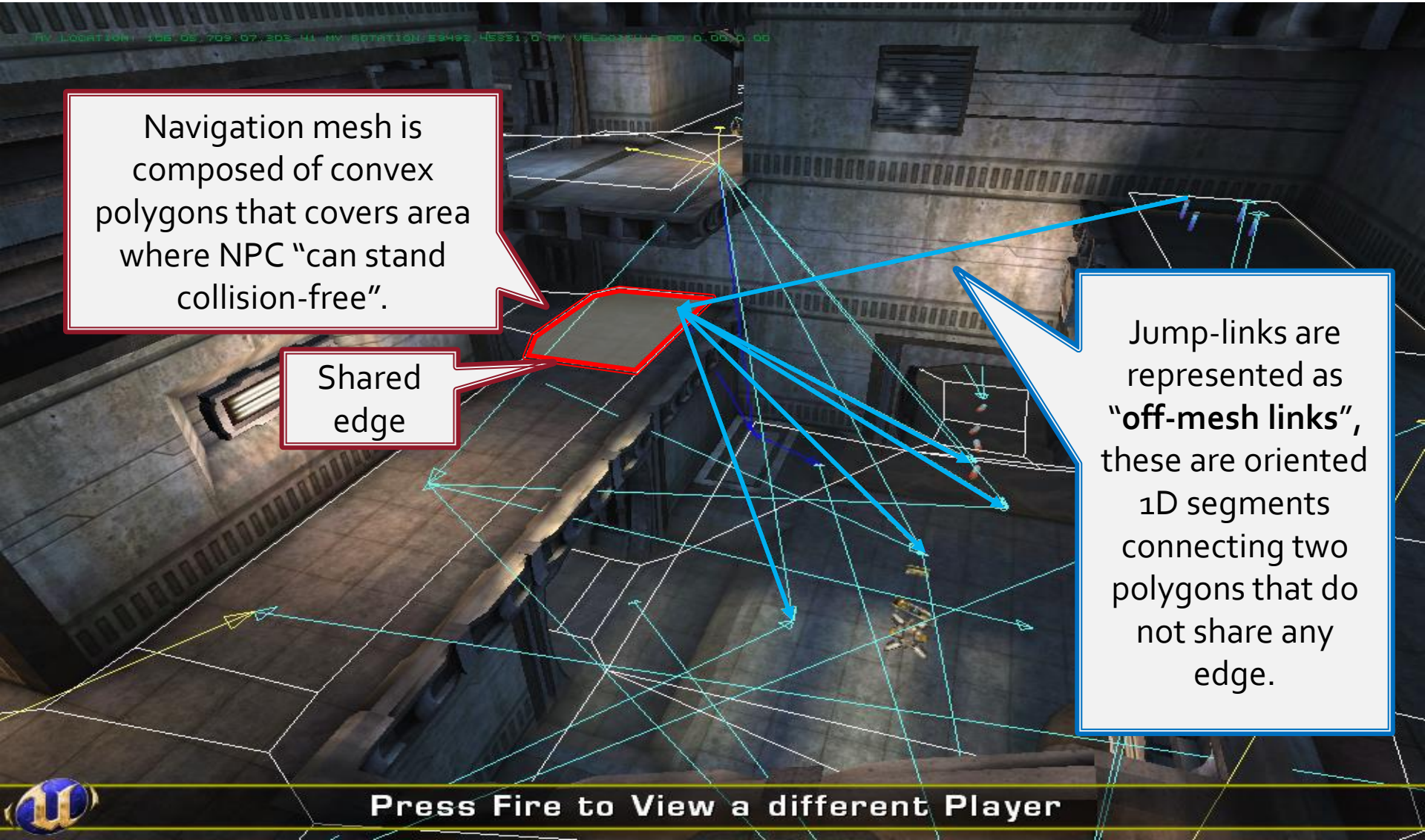
Navigation Mesh

Environment Abstraction

Navigation Mesh

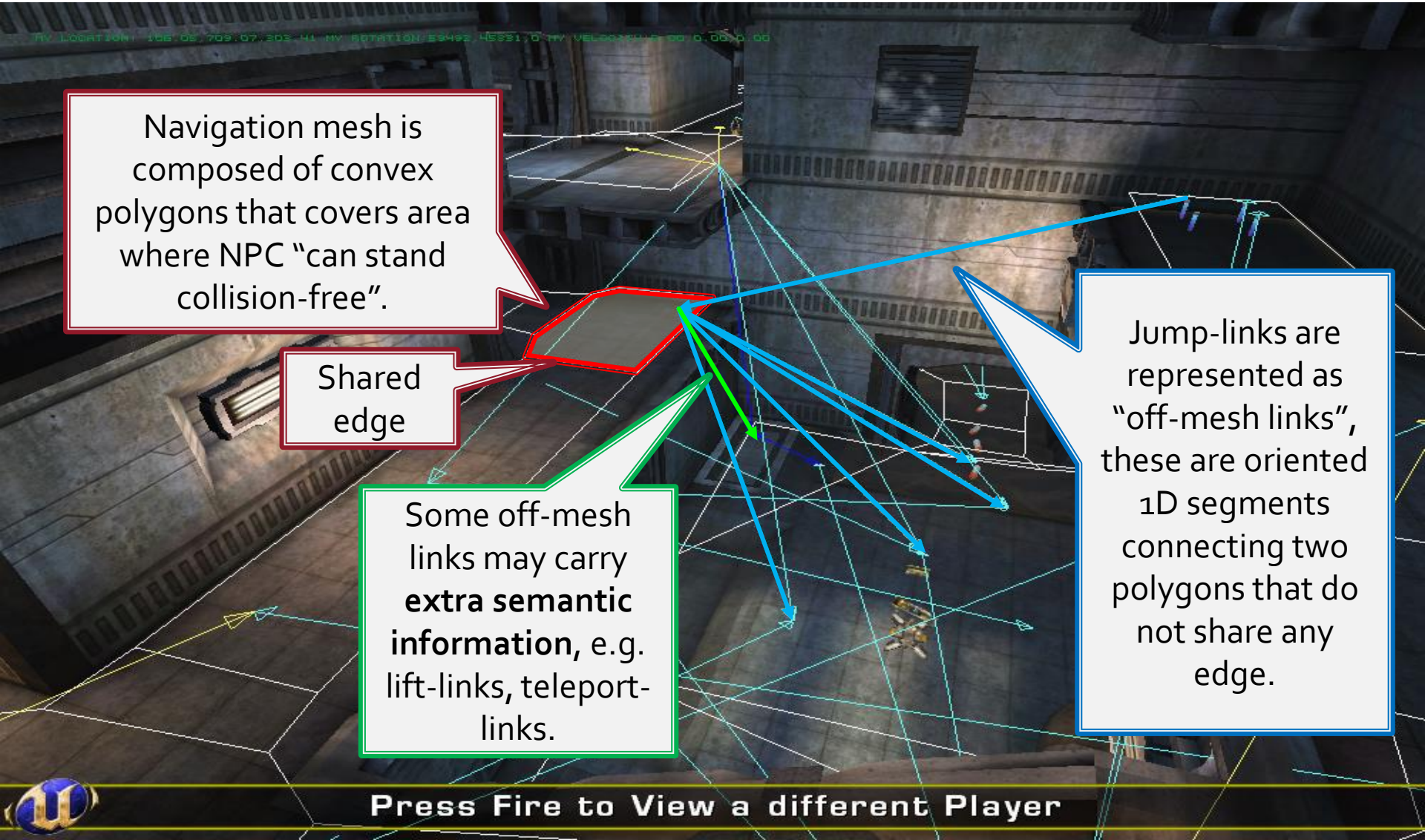
Environment Abstraction

Navigation Mesh



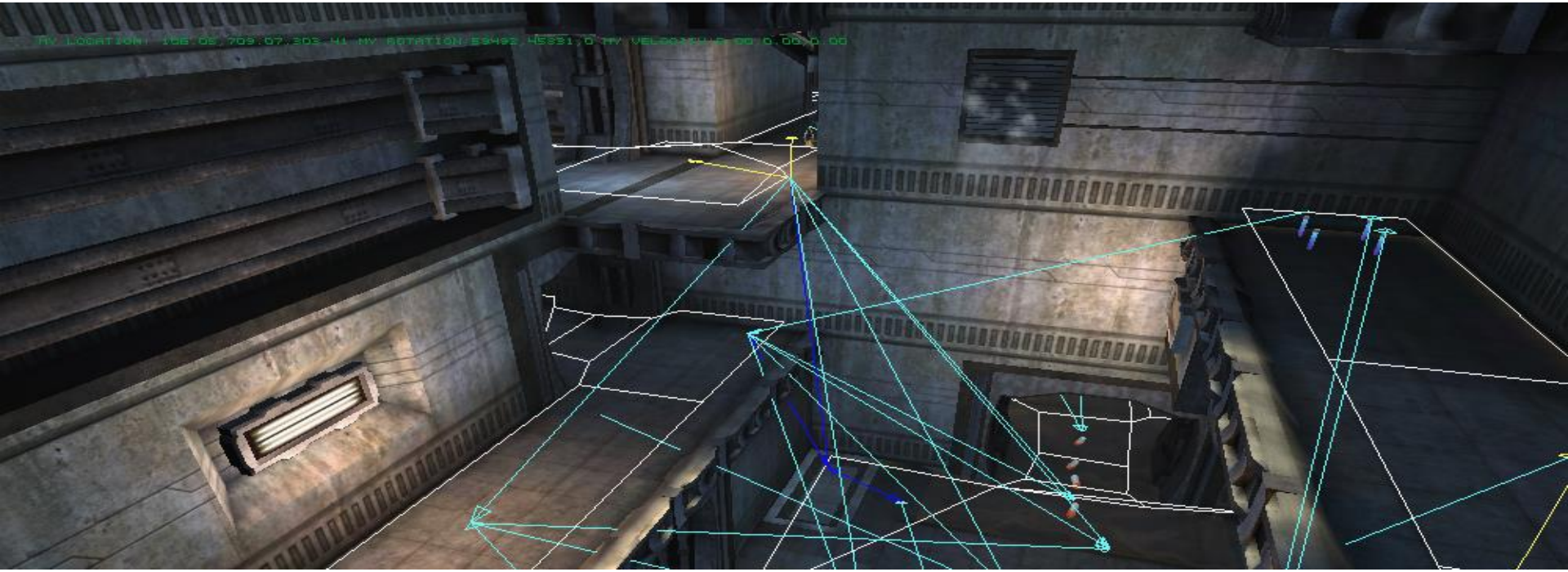
Environment Abstraction

Navigation Mesh



Environment Abstraction

Navigation Mesh



- Thanks to Mikko Mononen, we have Recast (~2009)
  - Creates navigation mesh out of triangle soup (no extra information required)
  - Permissive license - usable for commercial purposes… used both in Unity and UE4 today
- Show case: https://www.youtube.com/watch?v=5g_K6f8y9ko

Smart Objects

(peeking into NAIL068)

Sims

From horror to life simulator and back again

Sims

From horror to life simulator and back again

Sims

From horror to life simulator and back again

https://www.youtube.com/watch?v=RdVis25dXJk&t=790s

Now back to F.E.A.R. again

F.E.A.R.

Three states of AI

Path-finding through navigation mesh.

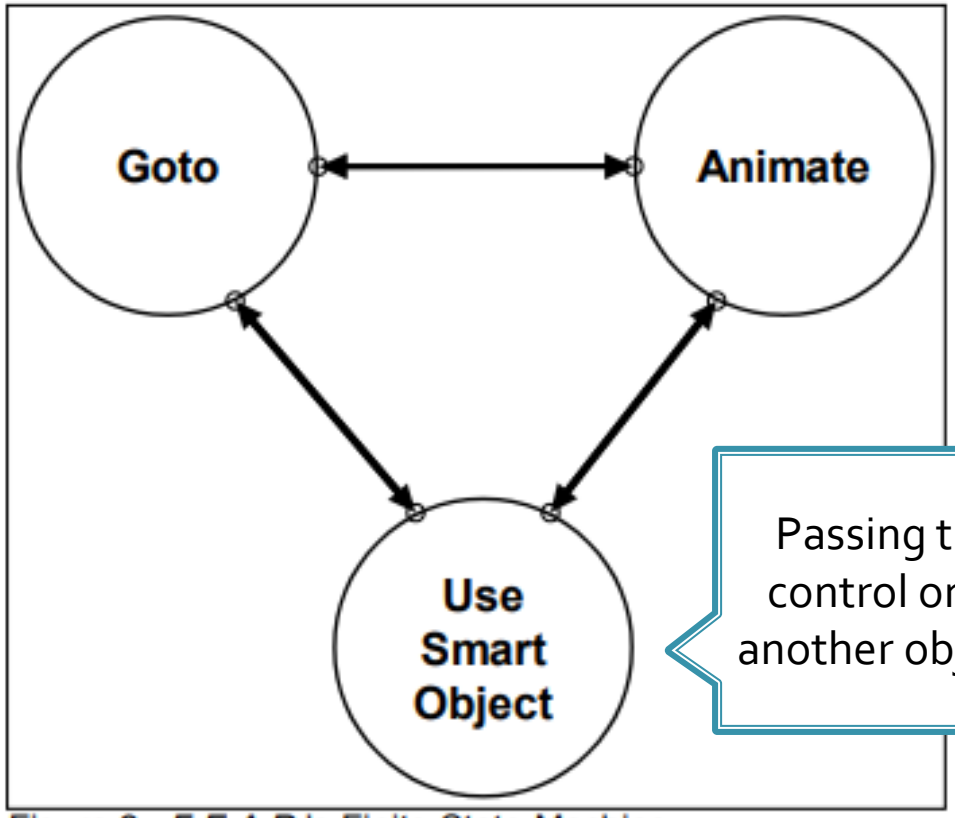


Figure 2: *F.E.A.R.*'s Finite State Machine

Running uninterruptible animation.

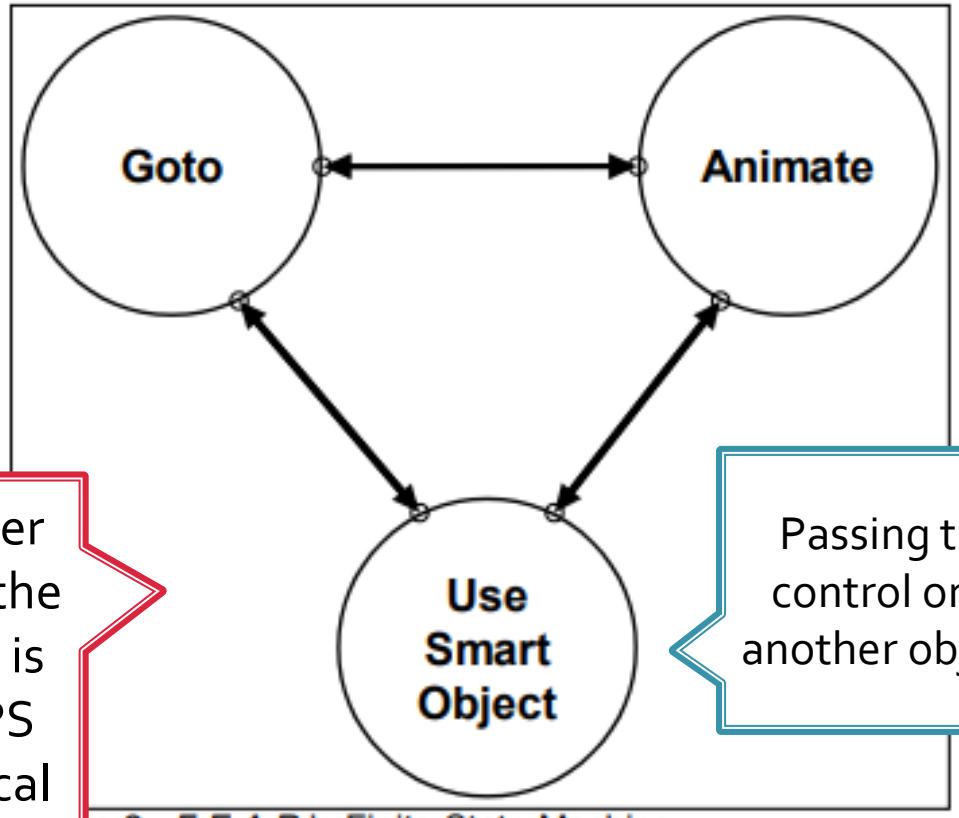
Passing the control onto another object.

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

F.E.A.R.

Three states of AI

Path-finding through navigation mesh.



re 2: *F.E.A.R.*'s Finite State Machine

Running uninterruptible animation.

A*-based planner is orchestrating the switches, which is based on STRIPS idea from classical planning.

Passing the control onto another object.

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

Classic Planning

(peeking into NAIL071)

State Space

The Definition

$$S : \langle S, A, Action(s), Result(s, a), Cost(s, a) \rangle$$

- S is the set of all possible states;
- A is the set of possible actions, not related to a particular state but regarding all the state space;
- $Action(s)$ is the function that establish which action is possible to perform in a certain state;
- $Result(s, a)$ is the function that returns the state reached performing action a in state s
- $Cost(s, a)$ is the cost of performing an action a in state s. In many state spaces is a constant, but this is not true in general.

State Transition System w/ Events

The Definition

$$\boldsymbol{\Sigma} = (\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{E}, \boldsymbol{\gamma})$$

S	set of states
A	set of (agent) actions
E	set of (external / exogenous) events
γ	transition function $S \times A \times E \rightarrow P(S)$

Planning problem

The Definition

$$\boldsymbol{P = (\Sigma, s_0, g)}$$

Σ state transition system

s_0 initial state

g goal state

A solution is then a sequence of actions $\langle a_1, \dots, a_n \rangle$, which transforms initial state into a goal state:

$$g = \gamma(\dots \gamma(\gamma(s_0, a_1), a_2) \dots), a_n)$$

Classic planning assumptions

Simplifications really…

1. System is finite
2. System is fully observable
3. System is deterministic
4. System is static (no external events)
5. Goals are given
6. Plans are sequential
   - No contingencies required
7. Time is implicit
   - No action duration is assumed
8. Offline planning
   - World does not change during planning

Planning domain

Specifying Σ further via set representation

$L = \{p_1, \dots, p_n\}$

- set of all (unique) propositions a state can be made of

Planning domain $\boldsymbol{\Sigma}$ over L is then $(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{\gamma})$

$S \subseteq P(L)$ states are made of propositions

$a \in A$ is a triple over subsets of L

$a = (precond, p.eff, n.eff)$

preconditions, positive and negative effects

$\gamma(s, a) = \big(s - n.eff(a)\big) \cup p.eff(a)$

provided a is applicable in s iff $precond(a) \subseteq s$

Operators

Adding variables to actions

Actions operate over sets of literals; they cannot be parametrized.

Operator == parametrized action.

Formally: operator $\boldsymbol{o = (name, precond, p.eff, n.eff)}$

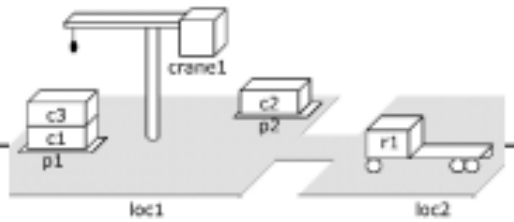
$name$	$n(x_1, \ldots, x_n)$ name of the operator including variables
$precond$	preconditions of the operator, which are now literals that can be parametrized
$p/n.eff$	positive and negative effects, again, can be parametrized

Operators

Adding variables to actions

An action is a fully instantiated operator

– substitute constants to variables



operator

take(k, l, c, d, p)

;; crane k at location l takes c off of d in pile p

precond: belong(k, l), attached(p, l), empty(k), top(c, p), on(c, d)

effects: holding(k, c), $\neg$ empty(k), $\neg$ in(c, p), $\neg$ top(c, p), $\neg$ on(c, d), top(d, p)

action

take(crane1,loc1,c3,c1,p1)

;; crane crane1 at location loc1 takes c3 off c1 in pile p1

precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)

Bartak, R. (2021), NAIL071 - Planning and Scheduling, Lecture 2. http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture02eng.pdf

Forward Planning

(peeking into NAIL071)

Forward planning

Similar to what we have seen in previous lecture

Forward-search(O, s_0, g)

$s \leftarrow s_0$

$\pi \leftarrow$ the empty plan

loop

if s satisfies g then return π

$E \leftarrow \{a | a$ is a ground instance of an operator in O,

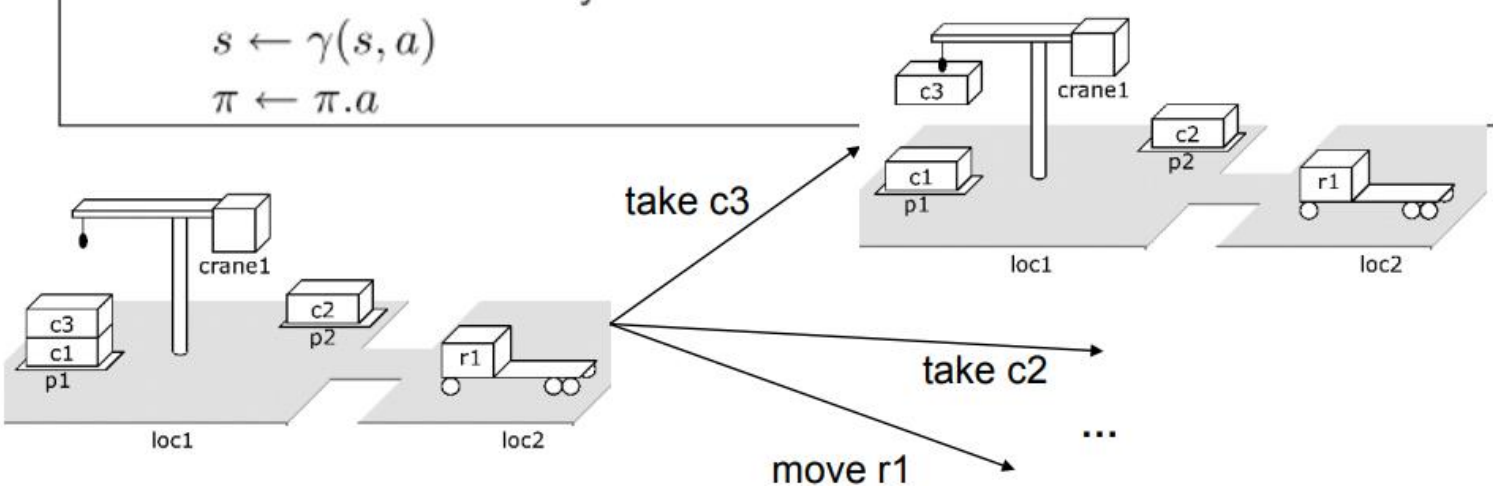
and $\text{precond}(a)$ is true in $s\}$

if $E = \emptyset$ then return failure

nondeterministically choose an action $a \in E$

$s \leftarrow \gamma(s, a)$

$\pi \leftarrow \pi.a$



Bartak, R. (2021), NAIL071 - Planning and Scheduling, Lecture 3. http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture03eng.pdf

Backward Planning

(peeking into NAIL071)

Backward planning

Start with the goal and search backwards

Action a is relevant for a goal g if and only if:

- action **a** contributes to goal **g**: $\mathbf{g} \cap \text{effects}(\mathbf{a}) \neq \emptyset$
- effects of action **a** are not conflicting goal **g**:
  - $g^- \cap \text{effects}^+(a) = \emptyset$
  - $g^+ \cap \text{effects}^-(a) = \emptyset$

A **regression set** of the goal **g** for (relevant) action **a** is

$\gamma^{-1}(g,a) = (g - \text{effects}(a)) \cup \text{precond}(a)$

Example:

goal: **{on(a,b), on(b,c)}**

action **stack(a,b)** is relevant

stack(*x*,*y*)	
Precond:	holding(*x*), clear(*y*)
Effects:	~holding(*x*), ~clear(*y*), on(*x*,*y*), clear(*x*), handempty

by backward application of the action we get a new goal:

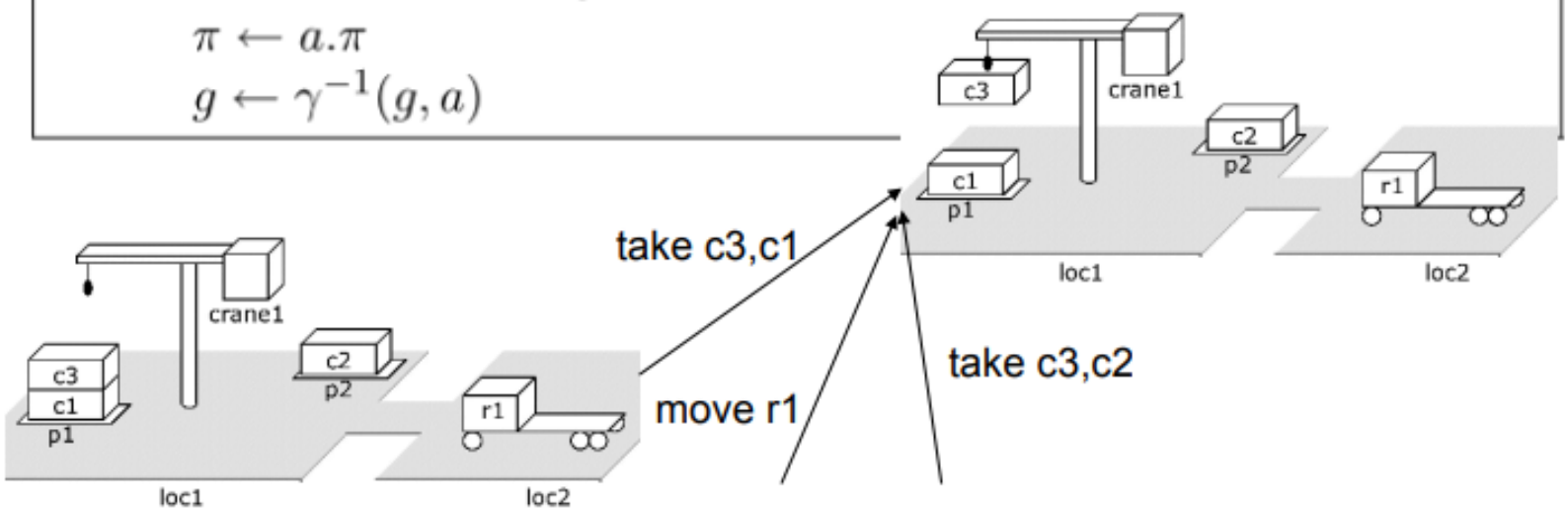
{holding(a), clear(b), on(b,c)}

Bartak, R. (2021), NAIL071 - Planning and Scheduling, Lecture 3. http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture03eng.pdf

Backward planning

Start with the goal and search backwards

Backward-search(O, s_0, g)
  $\pi \leftarrow$ the empty plan
  loop
    if s_0 satisfies g then return π
    $A \leftarrow \{a | a$ is a ground instance of an operator in O
      and $\gamma^{-1}(g, a)$ is defined$\}$
    if $A = \emptyset$ then return failure
    nondeterministically choose an action $a \in A$
    $\pi \leftarrow a.\pi$
    $g \leftarrow \gamma^{-1}(g, a)$



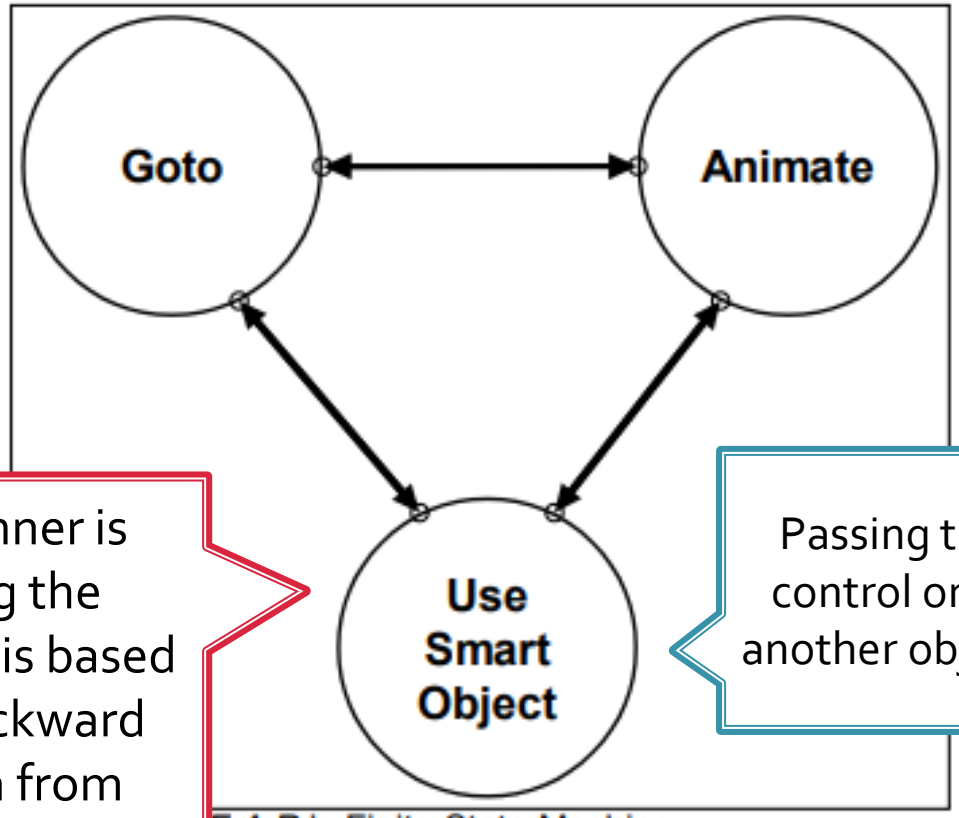
Bartak, R. (2021), NAIL071 - Planning and Scheduling, Lecture 3. http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture03eng.pdf

Now back to F.E.A.R. again

F.E.A.R.

Three states of AI

Path-finding through navigation mesh.



E.A.R.'s Finite State Machine

Running uninterruptible animation.

A*-based planner is orchestrating the switches, which is based on STRIPS (backward chaining) idea from classical planning.

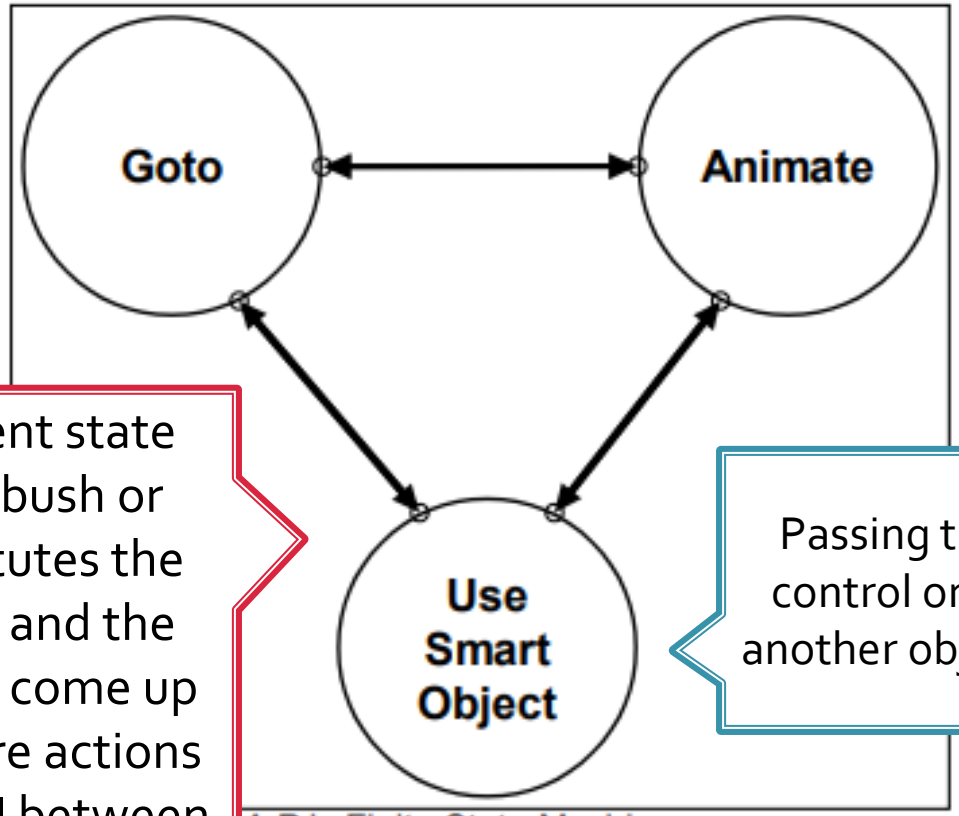
Passing the control onto another object.

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

F.E.A.R.

Three states of AI

Path-finding through navigation mesh.



A.R.'s Finite State Machine

Running uninterruptible animation.

NPC is given current state and goal (e.g. ambush or cover); this constitutes the planning problem and the planner is trying to come up with the plan, where actions are switching the AI between these three states.

Passing the control onto another object.

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

Adapting classic planning for first-person shooter game

(dealing with planning assumptions)

Classic planning vs. FPS reality

The assumptions (reminder)

1. System is finite
2. System is fully observable
3. System is deterministic
4. System is static (no external events)
5. Goals are given
6. Plans are sequential
   - No contingencies required
7. Time is implicit
   - No action duration is assumed
8. Offline planning
   - World does not change during planning

Classic planning vs. FPS reality

The problem

1. System is finite

!!! Planning domain requires full specification of the world states in literals, no numeric fluents to represent positions, weapons, ammo, health, …

Classic planning vs. FPS reality

F.E.A.R. AI Solution

1. System is finite

!!! Planning domain requires full specification of the world states in literals, no numeric fluents to represent positions, weapons, ammo, health, … or "is cover near-by?" literal.

=> Allow for the use of procedural preconditions and effects.

Classic planning vs. FPS reality

The problem

3. System is deterministic
4. System is static (no external events)

!!! The player is source of external events, aligning plans of (all) NPCs is time consuming and thus they are source of external events as well.

Classic planning vs. FPS reality

F.E.A.R. AI Solution

3. System is deterministic
4. System is static (no external events)

!!! The player is source of external events, aligning plans of (all) NPCs is time consuming and thus they are source of external events as well.

=> Validate plans (== check if the rest of the plan can be executed still) and replan if not.

Classic planning vs. FPS reality

The problem

5. Goals are given

!!! The planner itself is not able to come up with the target goal state(s)

Classic planning vs. FPS reality

F.E.A.R. AI Solution

5. Goals are given

!!! The planner itself is not able to come up with the target goal state(s)

=> Provide simple rule-based squad behaviors, which defines goals for squad members to achieve

Classic planning vs. FPS reality

The problem

6. Plans are sequential

!!! The planner does not give contingencies, due to external events may fail during the execution.

Classic planning vs. FPS reality

F.E.A.R. AI Solution

6. Plans are sequential

!!! The planner does not give contingencies, due to external events may fail during the execution.

=> Plans are checked for validity, if they cease to be valid, replanning will take care of contingencies.

Classic planning vs. FPS reality

The problem

7. Time is implicit

!!! Actions take time to execute, e.g., reloading freezes the NPC and makes it vulnerable.

!!! Actions deemed to be the same for the planner, but some actions are easier to do then the others. E.g. opening the door is easier (and makes more sense) then trying to blast it off.

Classic planning vs. FPS reality

F.E.A.R. AI Solution

7. Time is implicit

=> Work with action preconditions and priorities; use actions only when it makes the most sense and prioritize actions according to their importance, e.g., running for cover is more important then having the weapon loaded

=> Assign actions with costs, use A* to search for plans with the lowest cost

Classic planning vs. FPS reality

The problem

8. Offline planning

!!! If the planner takes too much time, fast-paced games such as FPS can move on too much so the found plan is invalid from the start

Classic planning vs. FPS reality

F.E.A.R. AI Solution

8. Offline planning

!!! If the planner takes too much time, fast-paced games such as FPS can move on too much so the found plan is invalid from the start.

=> Implement the planner as fast as possible, do not chase complicate goals thus allow for finding the solution fast.

F.E.A.R. AI Design Philosophy

Following slides are heavily based on:
Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

F.E.A.R. AI Design

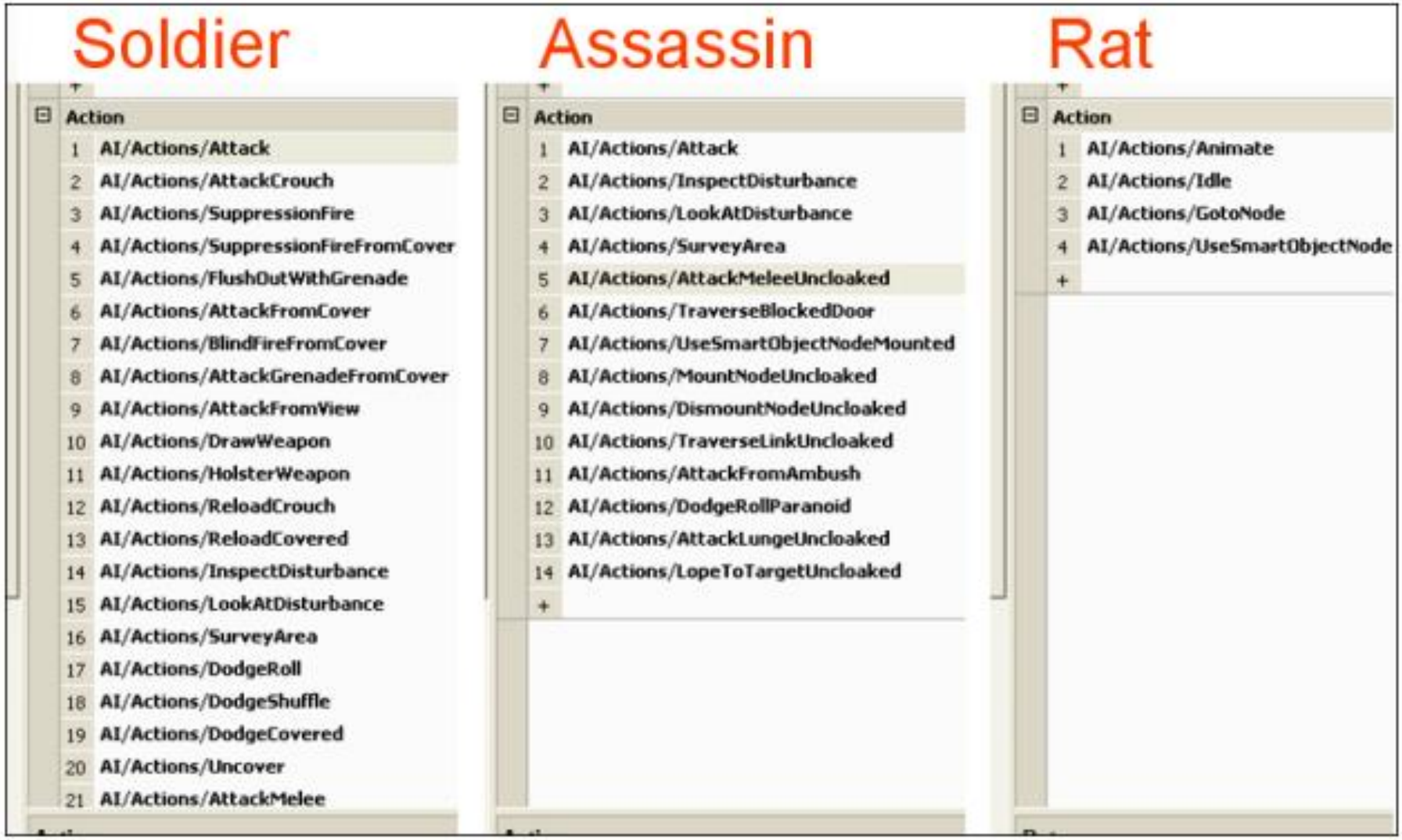
Level Design

„The design philosophy at Monolith is that the designer's job is to create interesting spaces for combat, packed with opportunities for the A.I. to exploit."

- Level design must offer situations AI can leverage
- AI actions must be customizable so level designers can create different kind of oppositions
- Level designers must not need to script AI behaviors

F.E.A.R. AI Design

Level Design



Soldier

	Action
1	AI/Actions/Attack
2	AI/Actions/AttackCrouch
3	AI/Actions/SuppressionFire
4	AI/Actions/SuppressionFireFromCover
5	AI/Actions/FlushOutWithGrenade
6	AI/Actions/AttackFromCover
7	AI/Actions/BlindFireFromCover
8	AI/Actions/AttackGrenadeFromCover
9	AI/Actions/AttackFromView
10	AI/Actions/DrawWeapon
11	AI/Actions/HolsterWeapon
12	AI/Actions/ReloadCrouch
13	AI/Actions/ReloadCovered
14	AI/Actions/InspectDisturbance
15	AI/Actions/LookAtDisturbance
16	AI/Actions/SurveyArea
17	AI/Actions/DodgeRoll
18	AI/Actions/DodgeShuffle
19	AI/Actions/DodgeCovered
20	AI/Actions/Uncover
21	AI/Actions/AttackMelee

Assassin

	Action
1	AI/Actions/Attack
2	AI/Actions/InspectDisturbance
3	AI/Actions/LookAtDisturbance
4	AI/Actions/SurveyArea
5	AI/Actions/AttackMeleeUncloaked
6	AI/Actions/TraverseBlockedDoor
7	AI/Actions/UseSmartObjectNodeMounted
8	AI/Actions/MountNodeUncloaked
9	AI/Actions/DismountNodeUncloaked
10	AI/Actions/TraverseLinkUncloaked
11	AI/Actions/AttackFromAmbush
12	AI/Actions/DodgeRollParanoid
13	AI/Actions/AttackLungeUncloaked
14	AI/Actions/LopeToTargetUncloaked

Rat

	Action
1	AI/Actions/Animate
2	AI/Actions/Idle
3	AI/Actions/GotoNode
4	AI/Actions/UseSmartObjectNode

F.E.A.R. AI Design

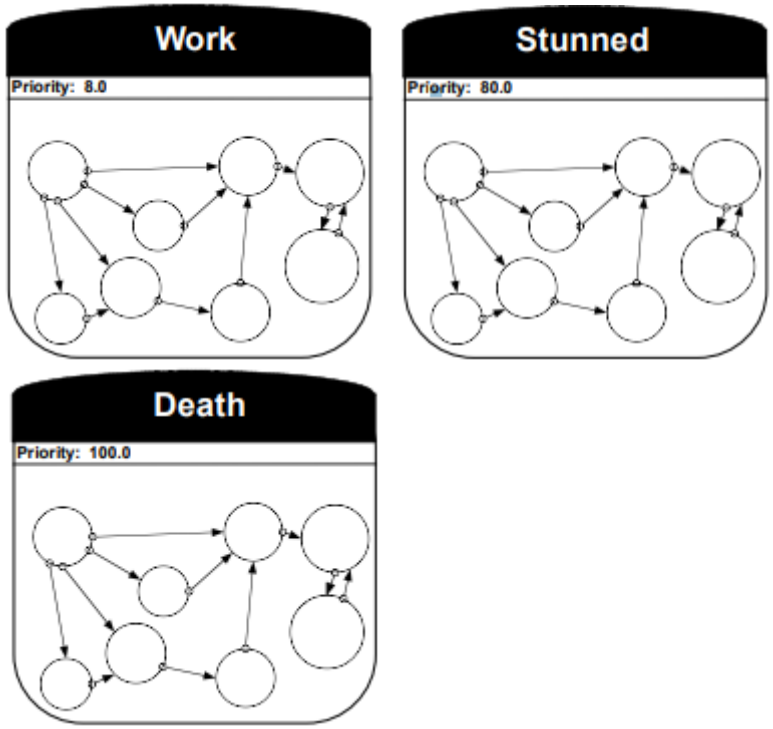
Decoupling Goals and Actions

„ … in the old system (NOLF2), each goal contained an embedded FSM. There was no way to separate the goal from the plan used to satisfy that goal. If we wanted any variation between the behavior of a mime and the behavior of a mutant, or between other character types, we had to add branches to the embedded state machines. Over the course of two years of development, these state machines become overly complex, bloated, unmanageable, and a risk to the stability of the project. "

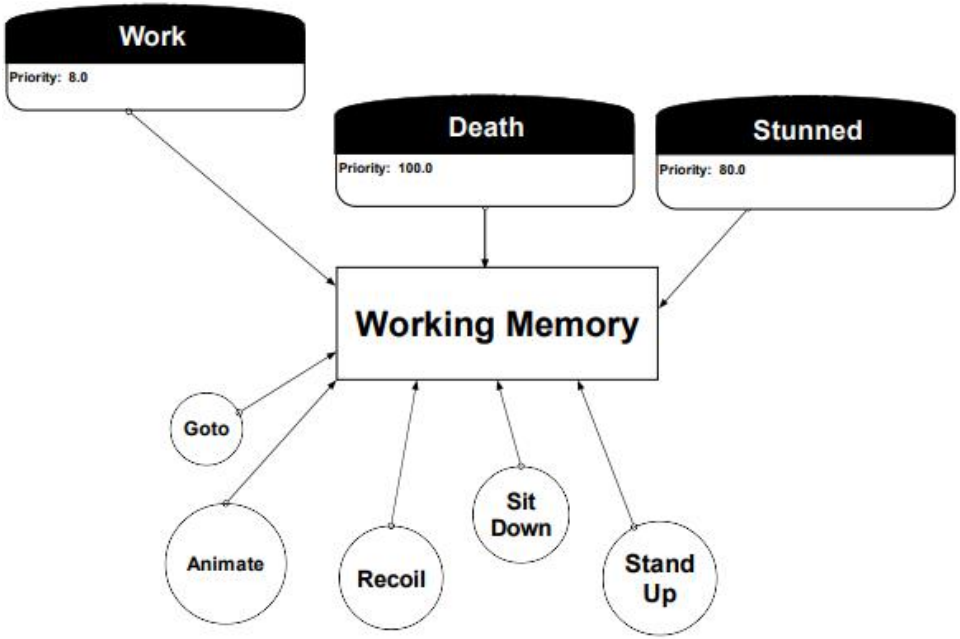
- You encode what you can do (actions), you do not encode when to execute them (concrete chain of actions)

F.E.A.R. AI Design

Decoupling Goals and Actions



In F.E.A.R., you can specify goals and actions and let the system find the plan of its own, fitting the situation.



In NOLF2, and any **reactive** planning planning, you are tight-coupling goals and their plans

F.E.A.R. AI Design

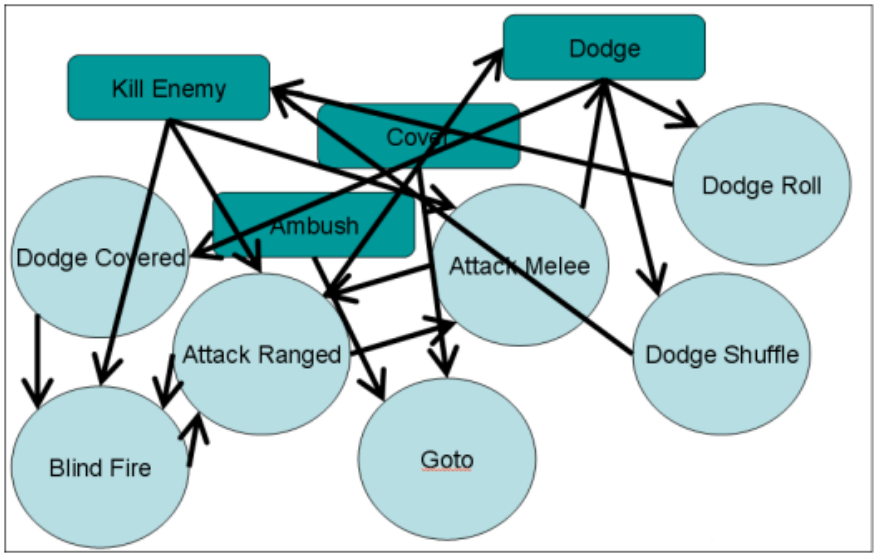
Layering Behaviors

„ The primary point we are trying to get across here is that with a planning system, we can just toss in goals and actions. We never have to manually specify the transitions between these behaviors. The A.I. figure out the dependencies themselves at run-time based on the goal
state and the preconditions and effects of actions. "

- You do not need to specify transition between behaviors
- E.g. … [A] Shoot -> [S] Got hit -> transition cnd -> [A] Take cover -> [S] Player near -> transition cnd -> [A] Melee attack -> …
  - In planning, you do not need to specify the transitions between [A]ctions given NPC [S]ensed information; but someone needs to set goals

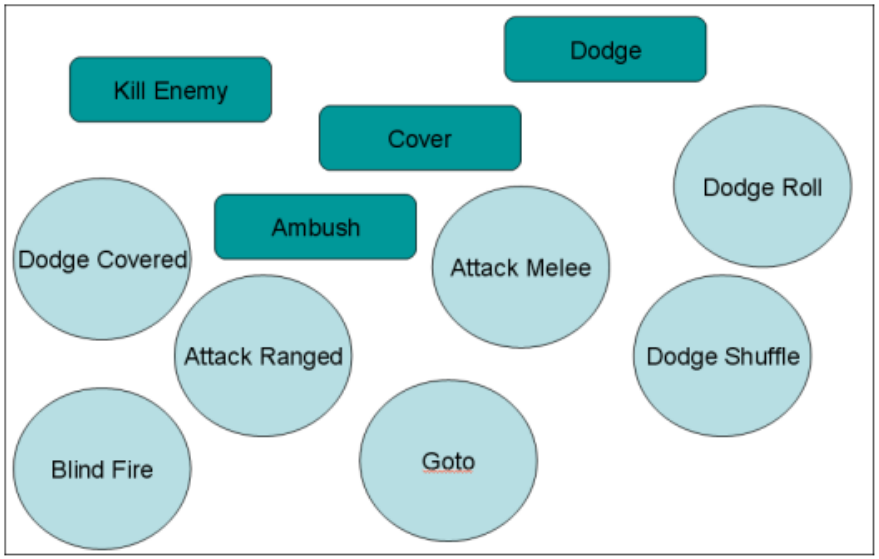
F.E.A.R. AI Design

Layering Behaviors



In F.E.A.R., the goal is always check for validity, if invalid, new goal is (contextually) selected and planned for.

In NOLF2, and any **reactive** planning, you need to manually specify all transitions between behavior states (plans).



F.E.A.R. AI Design

Dynamic Problem Solving

„ The third benefit of a planning system is the dynamic problem solving ability that re-planning gives the A.I. Imagine a scenario where we have a patrolling A.I. who walks through a door,
sees the player, and starts firing. If we run this scenario again, but this time the player physically holds the door shut with his body, we will see the A.I. try to open the door and fail. He then re-plans and decides to kick the door. When this fails, he re-plans again and decides to dive through the window and ends up close enough to use a melee attack! "

- Record failures in the working memory of NPC, when replanning, the planner tries other approach.

Goal Oriented Action Planner (GOAP)

vs.

STRIPS

(the differences)

Following slides are heavily based on:
Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).

GOAP vs. STRIPS

Difference #1: Cost per Action

Actions need to be ordered somehow because of their different degree of usefulness / sensibility.

In GOAP, there is a cost defined for each action, in STRIPS there is no such a thing.

GOAP vs. STRIPS

Difference #2: No add/delete lists

In backward chaining and classic planning in general you have separate definitions of preconditions, actions and effects.

In GOAP, preconditions and effects is sharing the same space and it is cleverly connecting actions to each other to represent the state of the world.

GOAP vs. STRIPS

Difference #3 and #4: Procedurality

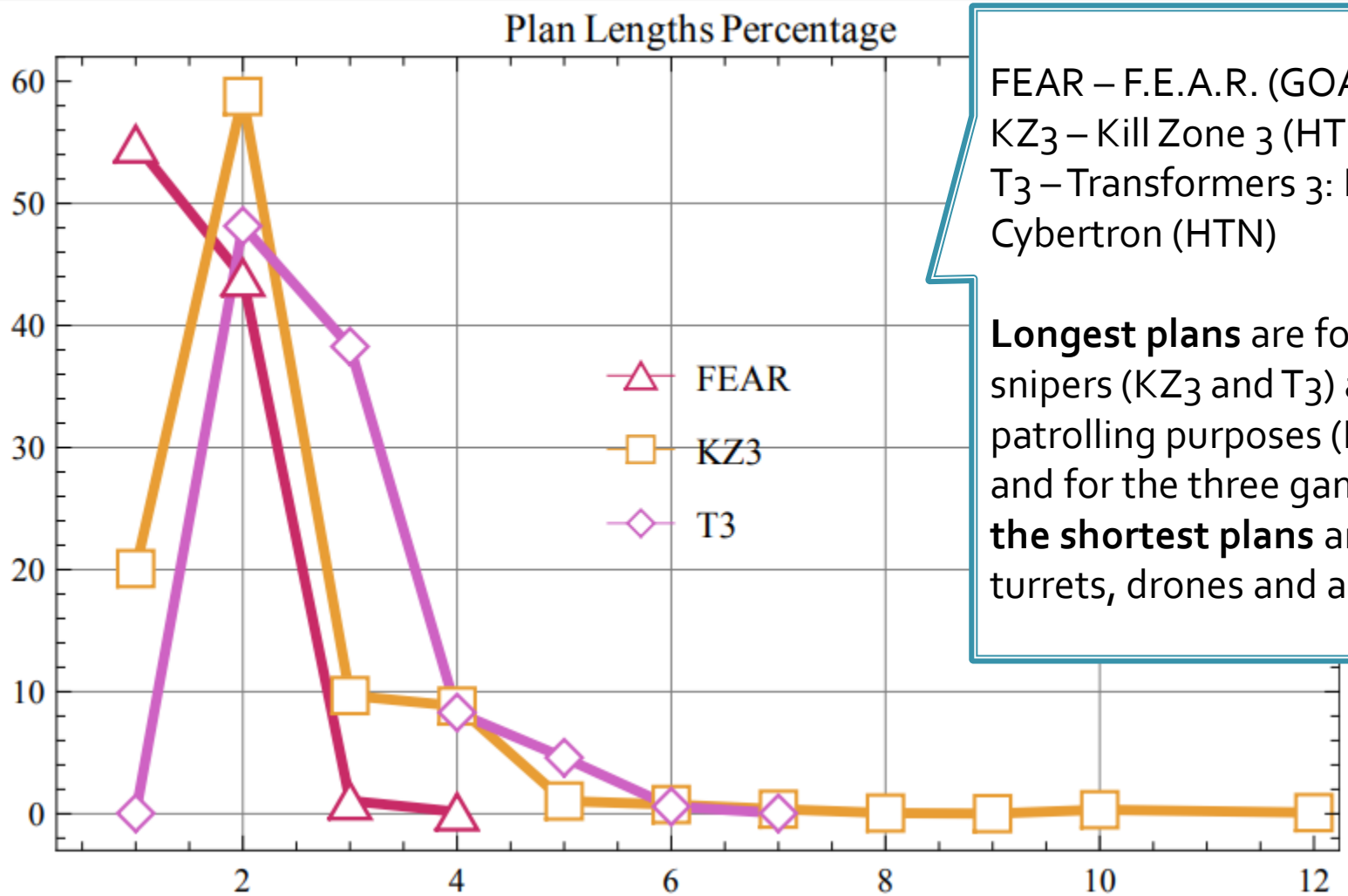
In classic planning the preconditions and effects are specified using set representation.

In GOAP, actions have hooks for procedural checks of preconditions and procedural effects applied to the state.

How big plans GOAP create in practice?

Classic planning vs. FPS reality

F.E.A.R. AI Solution



FEAR – F.E.A.R. (GOAP)
KZ3 – Kill Zone 3 (HTN)
T3 – Transformers 3: Fall of Cybertron (HTN)

Longest plans are for snipers (KZ3 and T3) and patrolling purposes (FEAR), and for the three games, **the shortest plans** are for turrets, drones and animals.

Jacopin, É. (2014, September). Game AI planning analytics: The case of three first-person shooters. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 10, No. 1).

Squad Behaviors in F.E.A.R.

Squad Behaviors

Architecture

Central coordinator that re-clusters NPCs into squads based on proximity.

Two types of squad behaviors: simple and complex.

Simple squad behaviors explicitly encoded; complex are arising emergently in cooperation with level design.

Simple squad behaviors are setting goals for GOAP planner or respective NPCs to satisfy.

Squad Behaviors

Simple (explicit) behaviors

Get-to-cover

1 NPC to execute suppression fire
The rest of the group takes cover.

Advance-cover

1 NPC to execute suppresion fire
Anoteher NPC to move closer to the threat.

Orderly-Advance

Move the squad to some position in a line.

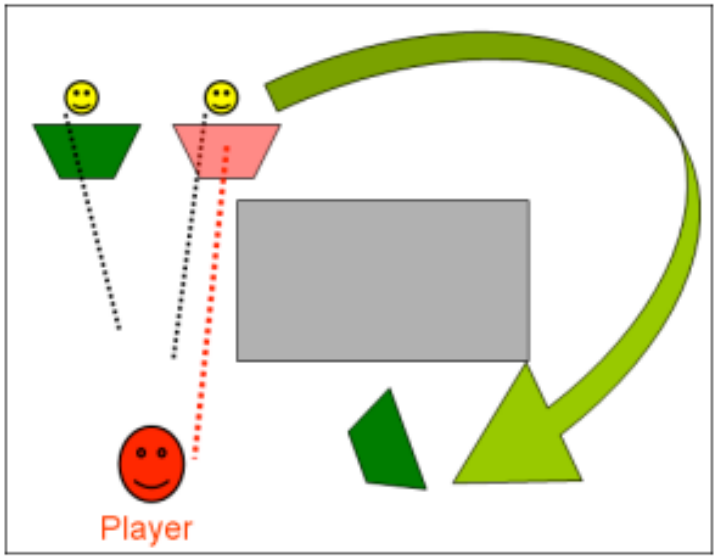
Search

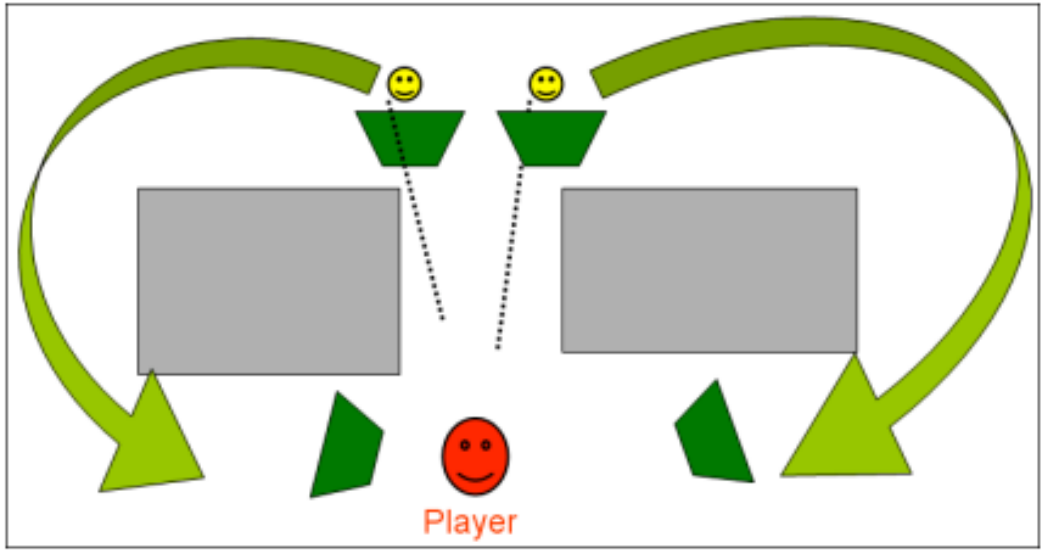
Splits the squad into pairs to search rooms in area.

Squad Behaviors

Complex (emergent) behaviors

Orkin, J. (2006, March). Three states and a plan: the AI of FEAR. In *Game developers conference* (Vol. 2006, p. 4).





Here the player attacks an NPC, which goal is set to Cover and the result plan is actually interpreted by the player as flanking.

Here the simple squad behavior Advance-Cover activates Cover goal for NPCs and as the nearest covers are different for them, it will be interpreted by the player as if NPCs are trying to surround her.

Combat Dialogues in F.E.A.R.

Combat Dialogues

Making the AI visible all the time

Once no NPCs can be seen (e.g. they take cover), they become invisible and all the AI code seems in vain + the player is not hinted what would come next.

=> Transparent combat dialogue solves both issues

=> Only clever tricks and (even planner) events

Combat Dialogues

Making the AI visible all the time

When someone is hurt, do not yell “I’m hurt” but first play other NPC voice “What is your status”.

When the player disappears from the vision, have NPC comment on the situation.

Planner fails to find a cover (which is hard to know/check for the player), do not stand there dumb, rather comment on the fact.

If nothing is happening, let soldiers at least talk, e.g. “Advance!” … “No way!”.

Have the last man standing calling for reincorcements, even though none will ever arrive.

That’s it for now!
(Almost…)

F.E.A.R.

Let's revisit the video!

This comment illustrates nicely how it (sometimes) is easy to trigger player imagination!

https://www.youtube.com/watch?v=KQN3yKYkFmE

F.E.A.R.

Sources

Gameplay
https://www.youtube.com/watch?v=KQN3yKYkFmE

F.E.A.R. SDK
https://github.com/xfw5/Fear-SDK-1.08

Relevant Roman Bartak's slides
http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture02eng.pdf
http://ktiml.mff.cuni.cz/~bartak/planovani/lectures/lecture03eng.pdf

AI and Games videos
https://www.youtube.com/watch?v=rf2T_j-FlDE
https://www.youtube.com/watch?v=PaOLBOuyswI (shorter more condensed version)

F.E.A.R.

Sources

Original 3 States of AI paper
https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf
And its GDC talk
https://www.gdcvault.com/play/1013282/Three-States-and-a-Plan

Paper on illusion of communication in F.E.A.R.
https://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter02_Combat_Dialogue_in_FEAR_The_Illusion_of_Communication.pdf

Game AI Planning Analytics paper by Eric Jacopin
https://www.researchgate.net/publication/331983913_Game_AI_Planning_Analytics_The_Case_of_Three_First-Person_Shooters