Lecture Notes for

Managing and Mining Multiplayer Online Games

Summer Semester 2017

Chapter 10: Artificial Intelligence

http://www.dbs.ifi.lmu.de/cms/VO_Managing_Massive_Multiplayer_Online_Games

Chapter Overview

- managing computer controlled entities
- modeling decision processes
- rule based behavior and state machines
- optimizing behavior
- non-deterministic behavior
- algorithms for AI engines and special decision problems
  - pathfinding in open environments
  - antagonistic search

Formal Decision Processing

- State space S: set of all possible situations.
  (e.g. positions of all figures on the chess board + who is able to make a term)
- Actions A_s: Given a situation s, *A_s describes the set of all actions an entity might perform. (e.g. all valid moves for the current player)*
- *Selecting action $a \in A_s$ in situation $s \in S$ might lead to a transition to a new state s*.*
  - *Case I: there is only on outcome s**
  - *Case II: s* is part of a subset $S^* \subseteq S$. It might hold that $s \subseteq S$.*
- *Given S* we have to consider the likelihood Pr(s*|s,a) for each $s^* \in S^*$.*

=> *the same action in the same situation might lead to different results (luck, behavior reacts differently...)*

Formal Decision Processing

What is the best action in a situation?

- reward: reaching a state s might yield a reward R(s) (negative rewards = cost)
- a state might be terminal (no action left, target reached)
- utility of a sequence of states *$(s_0,..,s_n)$ being generated by a sequence of actions $(a_0,..,a_{n-1})$*.
  - finite horizon: $U_h = \sum_{i=0}^{n} R(s_i)$
  - infinite horizon: $U_h = \sum_{i=0}^{n} \lambda^i R(s_i)$ where $0<\lambda_i<1$
- under uncertainty state transitions:
  - *multiple $(s_0,..,s_n)$ might be observed when starting at s_0 (n is not fixed)*
    => policy π: for each $s \in S$, select an action $a \in A_s$.
  - the utility of a policy π is the expected utility over all possible sequences starting in s: $U^{\pi}(s) = E\left[\sum_{i=0}^{\infty} \lambda^i R(s_i)\right]$

Formal Decision Processing

- an optimal policy maximizes the utility:
  $\pi^* = argmax_{\pi \in \Pi} U^{\pi}(s)$
- Bellman equation for iterative computation

$$U(s) = R(s) + \lambda \, max_{\, a \in A(s)} \sum_{s'} P(s'|s,a) U(s')$$

- simple optimization based on value iteration:

repeat
  U:=U',δ := 0
  for each state s in S do
    U'(s):= R(s) + $\lambda \, max_{\, a \in A(s)} \sum_{s'} P(s'|s,a) U(s')$
    if |U'(s) – U(s)|> δ then
      δ := |U'(s) – U(s)|
until δ < ε(1- λ)/ λ
return

Action-Generation for NPCs

Types of Computer Controlled Entities:

NPCs (Non-Player Characters), **MOBs** (Mobile Objects) or **Bots** (automated control for player characters) must create actions to be able to act.

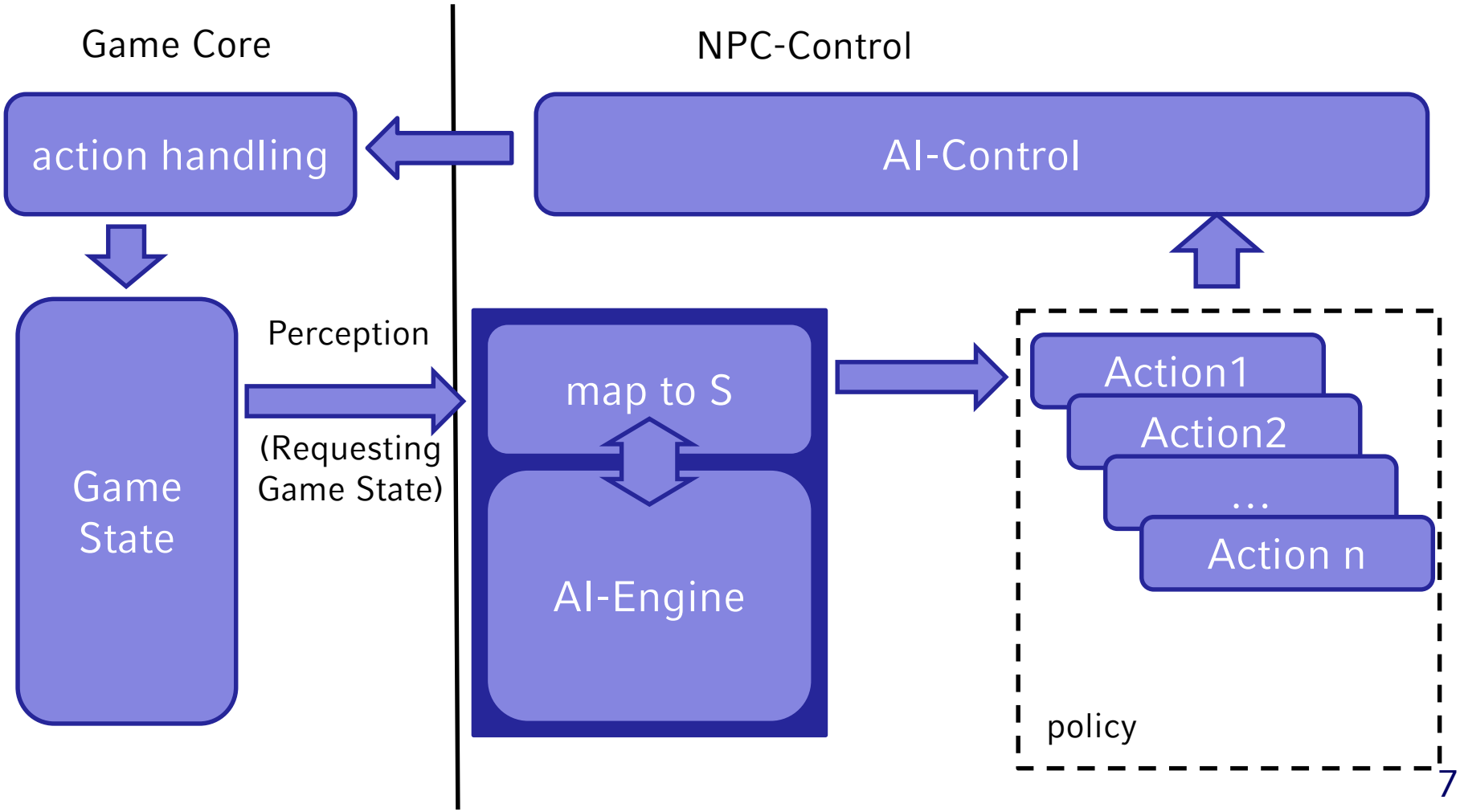
Important Aspects:

- granularity: control over one or several game Entities
- computational cost vs. behavioral complexity
- where are NPCs controlled (server or client)
- deterministic vs. probabilistic behavior
- how realistic is the behavior/what is good behavior

Basic Build of an AI (Autonomous Agents)

Task:

- generation of the next action
- appropriate to the actual state



AI-Control

Synchronization of action processing and action creation:

- calling action creation
  - **Time-controlled**: Action requests are inquired regularly
  - **Event-controlled**: Certain changes of GS can inquire an action request. (Player enters Aggro-Range)
- the action creation:
  - map game state observation to an internals state
  - output the action from current policy
- open question: how are the policies created
  - for a small state space the complete policies can be precomputed and stored
  - the granularity of an action might be to high to put into a particular action. (e.g. run after enemy requires a policy for the best path)
  - this low level policies/plans might can be computed on the fly

Action Generation

- What is requested when from the game state?
- One central request from Game State („all perceptions“)
  vs. several requests (selective perception)
- How are actions generated?
  - Organized with rules or state machines?
  - Deterministic or randomized actions?
- Implementation
  - Programming in game engine language
  - Programming in script language (e.g. LUA)

Example: „Autocamp 2000“

example policy for a RPG farm bot:

(http://www.gamespy.com/articles/489/489833p1.html)

1) If invited by any group => join group
2) If in a group => follow behind the leader
3) If sees a monster => attack
4) If someone says something ending in a question mark
   => respond by saying "Dude?"
5) If someone says something ending in an exclamation point
   => respond by saying "Dude!"
6) If someone says something ending with a period
   => respond by randomly saying one of three things: "Okie“, "Sure“, or "Right on"
7) EXCEPTION: If someone says something directly to you by mentioning your name, respond by saying "Lag."

Example

KillSwitch: [Shouting] Does anyone want to join our hunting party?

Farglik: [Powered by the Autocamp 2000] Dude?

[KillSwitch invites Farglik to join the group.]

[Farglik joins the group]

KillSwitch: We're gonna go hunt wrixes.

Farglik: Right on.

[The group of players runs out, Farglik following close behind. Farglik shoots at every little monster they pass.]

KillSwitch: Why are you attacking the durneys?

Farglik: Dude?

KillSwitch: The durneys, the little bunny things -- why do you keep shooting at them?

Farglik: Dude?

KillSwitch: Knock it off guys, I see some wrixes up ahead. Let's do this.

Farglik: Right on.

[The group encounters a bunch of dangerous wrixes, but they gang up and shoot every one of them.]

KillSwitch: We rock!

Farglik: Dude!

Troobacca: We so OWNED them!

Farglik: Dude!

Example

KillSwitch: Uh oh, hang on. Up ahead are some Sharnaff bulls. We can't handle them, so don't shoot.

Farglik: Okie.

[Farglik shoots one of the Sharnaff bulls.]

[The bull attacks; Trobacca and several other party members are killed before they beat it.]

KillSwitch: You IDIOT! Farglik why did you shoot at them?

Farglik: Lag.

KillSwitch: Well don't do it again.

Farglik: Sure.

[Farglik shoots at another Sharnaff bull.]

[The entire party is slaughtered except for Farglik.]

[... Farglik stands there, alone, for several hours ...]

Control with Rules

- **Body**: predicate logical expression on the game state
- **Head**: List of actions to be inserted into the task-list
- Control consists of a list of rules
- Rule is chosen by priority sorting
  => first applicable rule creates action

Disadvantages:

- Rule bodies are often redundant
- High expense of finding the applicable rule

Example:

- NPC.inbattle() $\wedge$ (NPC.health < 50) => NPC.runAwayFrom(NPC.battle().getMaxAggroMob())
- NPC.inbattle() $\wedge$ (NPC.health > 50) => hit (NPC.battle().getMaxAggroMob())
- (NPC.health < 50) $\wedge$ (NPC.inv.contains(Melone)) => NPC.eat(Melone)
- NPC.inGroup() => Follow(NPC.GroupLeader)
- NPC.hasInvite() => NPC.join(NPC.invitation.getGroup())
- __=> NPC.wait()

Control with hierarchical state machines

Idea: Combining rules by state of the NPC

⇒ Only the rules applicable to the actual state are being evaluated

⇒ Actual NPC state is part of the game state

⇒ For complex MOBs states can be further differentiated

⇒ Changing state through action handling
(e.g. Monster breaks off pursuit=> Change from *battle* to *base*)

battle

- (NPC.health < 50) => NPC.runAwayFrom(NPC.battle().getMaxAggroMob())
- (NPC.health > 50) => hit (NPC.battle().getMaxAggroMob())

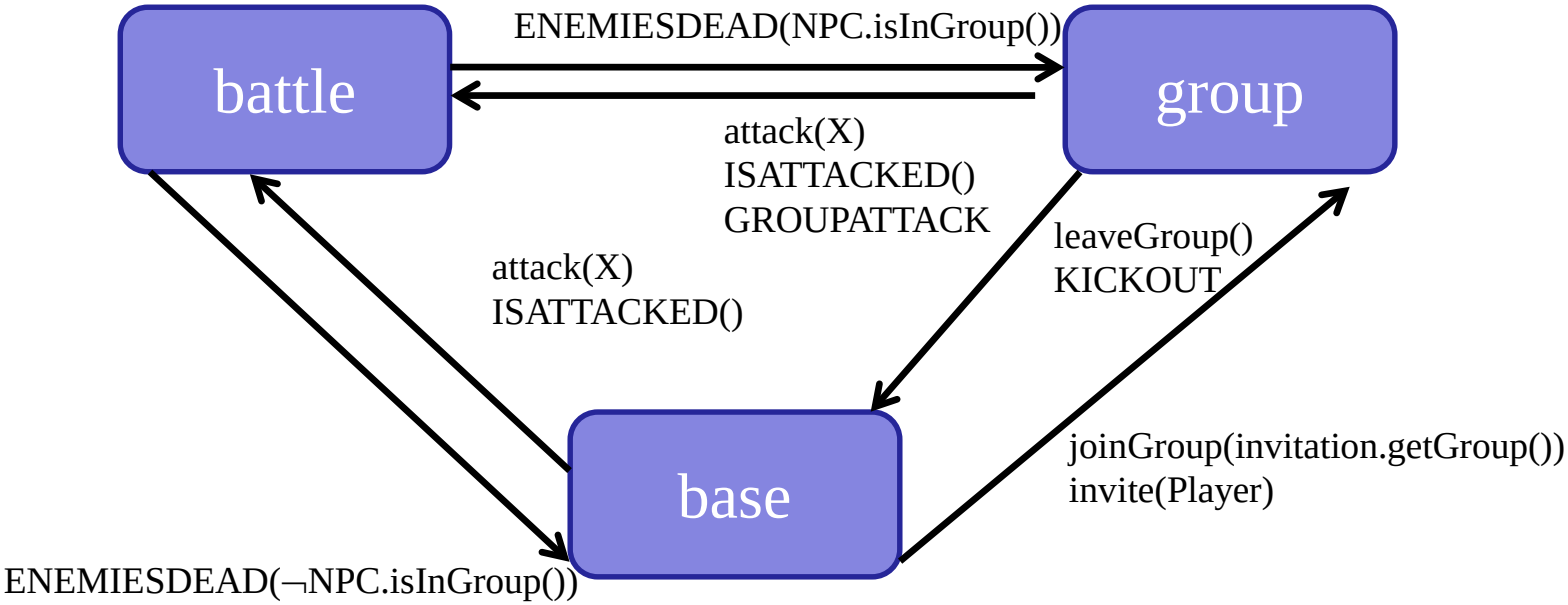
group

- __=>Follow(NPC.GroupLeader)

base

- ((NPC.health < 50) (NPC.inv.contains(Melone)) => NPC.eat(Melone)
- NPC.hasInvite() => NPC.join(NPC.invitation.getGroup())
- __=> NPC.wait()

Example: State Machine



- UPPER CASE: event based transition
- lower case: action based transition

Processing with State Machines

Process description:

1. Requesting ***GS***
   (information determining state)
2. Determining present state
   - clear allocation with disjunctive states (e.g. War and Peace)
   - unambiguous determination with priorities(e.g. 1:battle, 2:group, 3:base)
3. Requesting ***GS***
   (state specific information)
4. Generating action
   (Use of local rules)
5. perform action

Determinism and behavior change

- ***Deterministic behavior:***
  MOBs generate identical actions in identical situations
- ***Probabilistic behavior:***
  - Rules' heads contain several possible actions
    => Choosing with a random process
  - Several rules may be applicable in a situation
    => randomly selecting the rules
  - Determining current state or sub state is random

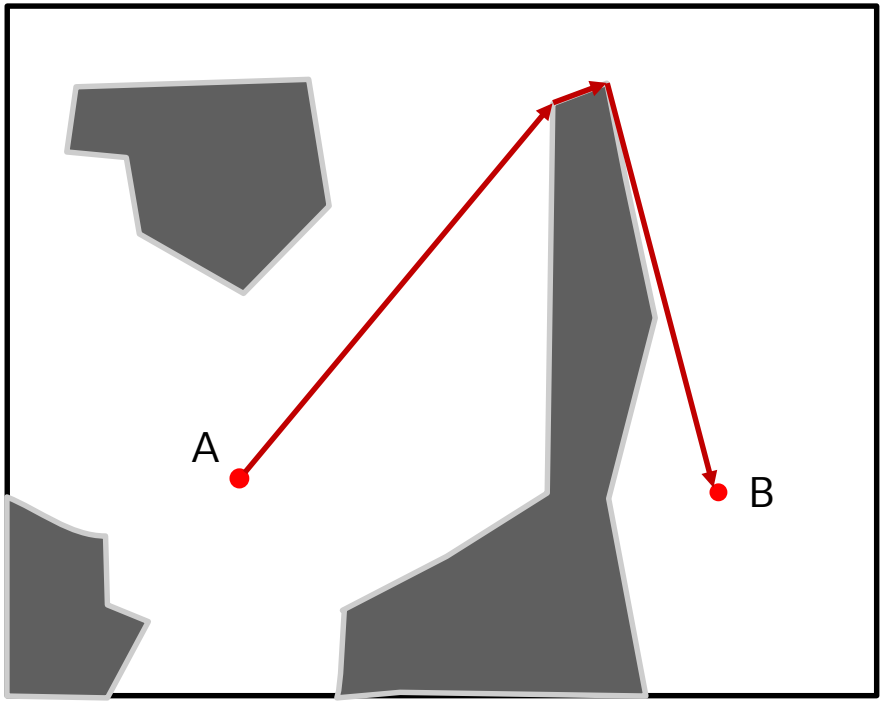
- ***Learning behavior:***
  - usually the state space S and the corresponding actions A are determined by the game. (S might be an abstraction of possible game states)
  - learning corresponds to selecting the best action for each situation
  - Markov decision processes: cf. formal view in the beginning
    - transition probabilities for each a_s must be known.
    - several algorithms for finding optimal policies
  - Reinforcement learning:
    - transition probabilities are unknown but samples are available (simulation!)
    - learn the expected future rewards directly from the samples

AI-Engine and AI-Services

- Action generation and AI control make base operations necessary that efficiently and generically complete certain tasks:
  - Finding entry and Exit points
  - Finding the shortest route
  - Antagonistic behavior
  - Swarm behavior
  - …
- AI-Engine: Collection of services useful for implementing AI

Pathfinding in Open Environments

- open environment: 2D Space ($\subseteq IR^2$)
- MOBs can move freely
- obstacles block direct connections
- presenting obstacles with:
  - ***polygons***
  - pixel-presentation
  - any geometric form (Circle, Ellipse, …)

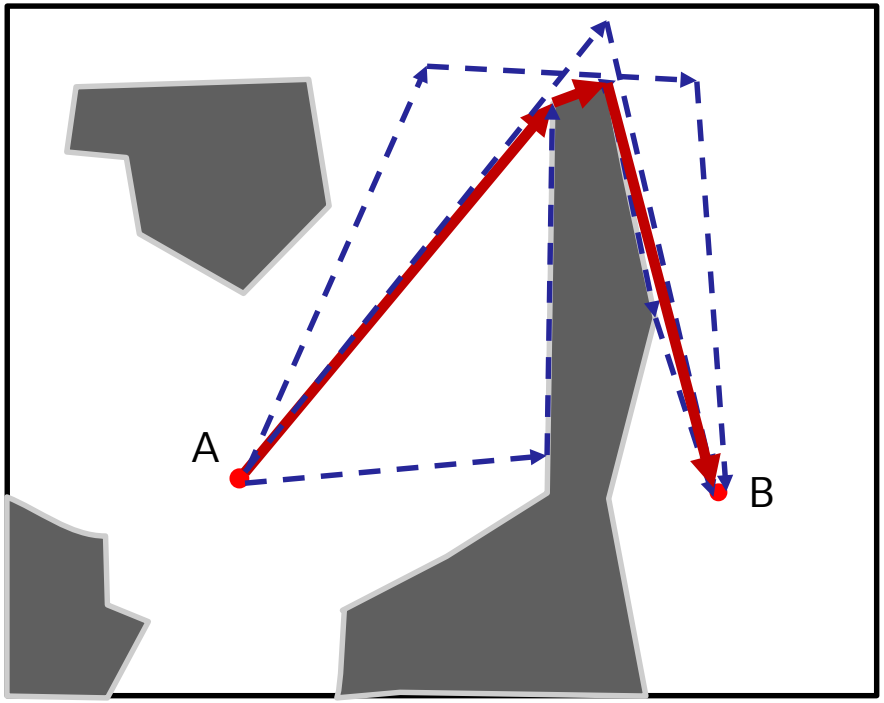


solution for polygon presentation:

- deriving a graph for the map containing the shortest routes (visibility graph)
- integrate start and target points
- use of pathfinding algorithms like Dijkstra, A* or IDA*

Visibility Graph

- finding the shortest path in an open environment is a search over an infinite search area
- **solution**: restricting the search area with following properties of optimal paths:
  - **waypoints of every shortest path are either start, target or corners of an obstacle-polygon.**
  - **paths cannot intersect polygons.**



- the shortest path in the open environment *U* is also part of the visibility graph $G_U(V,E)$.

Visibility Graph

Environment: U

- Set of polygons $U=(P_1, \ldots, P_n)$ **(Obstacles)**
- Polygon P: planar cyclic graph: $P = (V_P, E_P)$

Visibility graph: $G_U(V,E)$

- *Nodes:* Corners of polygons P = $\{V_1, \ldots, V_l)$ in U: $V_U = \bigcup_{P \in U} V_P$

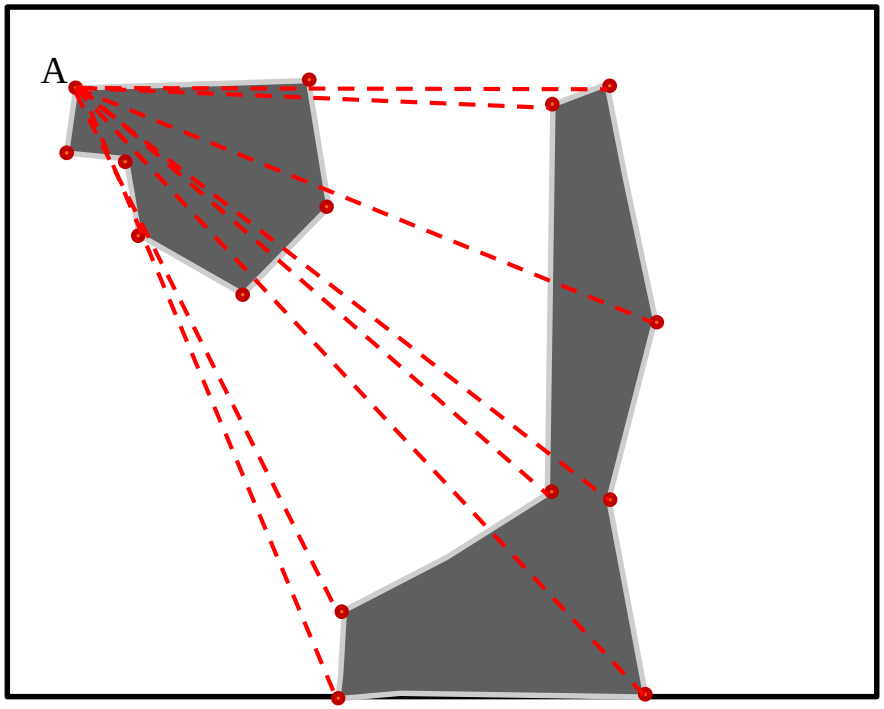
- *Edges:* All edges of polygons with all edges of nodes from different polygons that do not intersect another polygon-edge.

$$E_U = \bigcup_{P \in U} E_P \cup \{(x,y) \mid x \in P_i \wedge y \in P_j \wedge i \neq j \wedge \forall_{P \in U} \forall_{e \in E_P} : (x,y) \cap e = \{\}\}$$

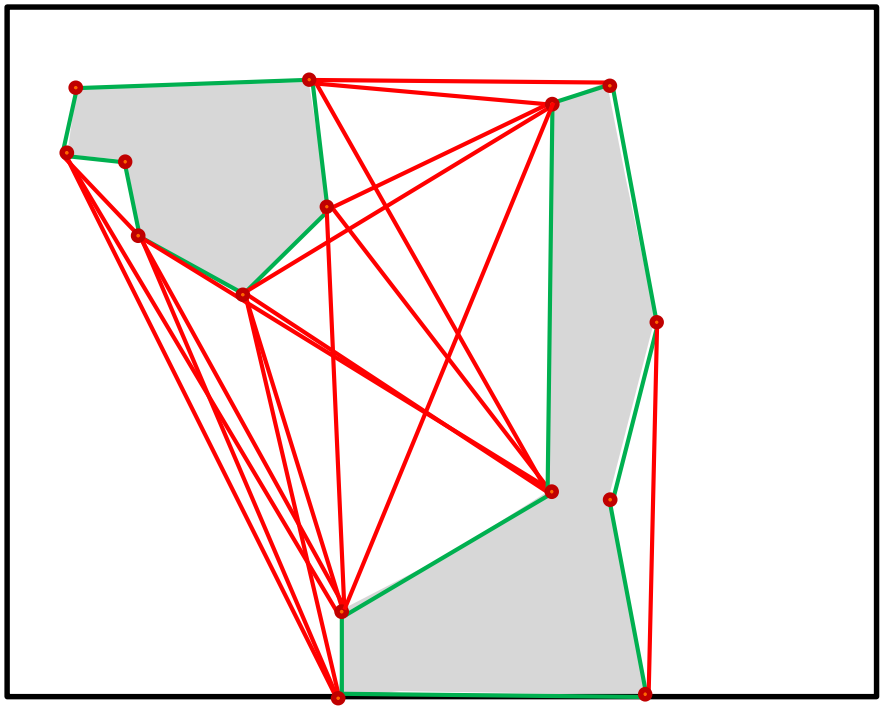
Remarks:

- This definition applies to convex obstacles. For concave polygons the convex shell must be calculated additionally. For any thus calculated additional edges testing for intersection with polygon edges becomes necessary.
- This definition includes a naive algorithm ($O(n^3)$) to construct a visibility graph. Die Derivation of the visibility graph can be optimized. (O'Rourke 87: $O(n^2)$)

Example: Visibility Graph



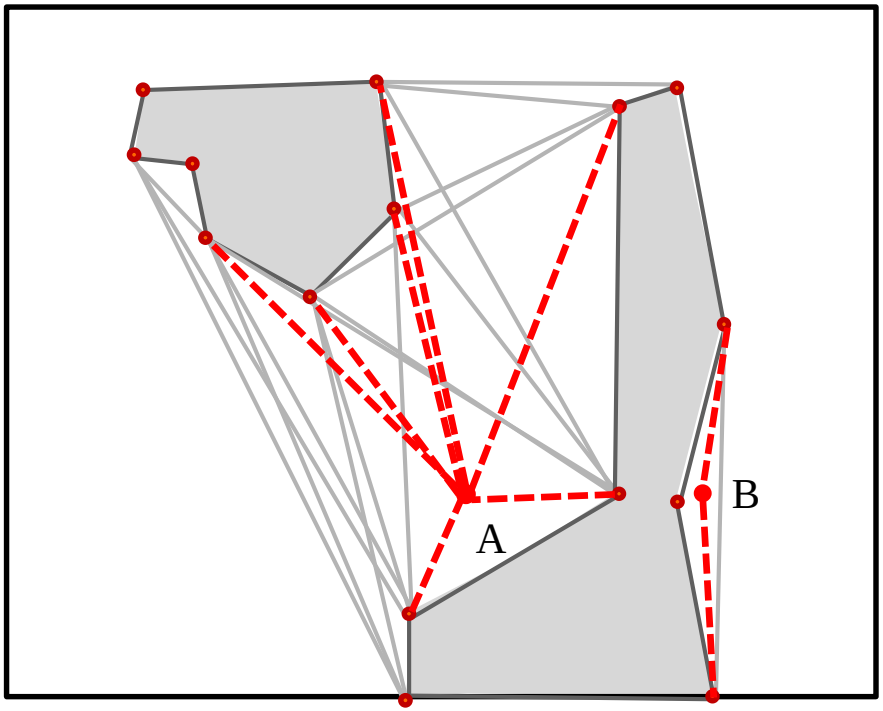
Edges for the node A being tested and discarded.



Visibility Graph: Red segments run between polygons. Green segments mark the polygons' borders.

Expansion with Start- and Target-Nodes

- visibility graph can be pre-calculated for static environments
- mobile objects must be integrated into the graph before calculation
- insert start S and taget Z as point polygons
- connect the new nodes to with all edges unless an intersection with a polygon occurs



Dijkstra

Used Data Structures:

- PriorityQueue Q (Contains paths sorted descending by cost)
- cost table T (contains cost for the currently best path for all visited nodes)

Pseudo-Code:

```
FUNCTION Path shortestPath(Node start, Node target)
    Q.insert(new Path(start,0))
    WHILE(Q.notIsEmpty())
        Path aktPath = Q.getFirst()
            IF aktPath.last()  == target THEN                    //result found
                return aktPath
            ELSE
                FOR Node n  in aktPath.last().nachfolger() DO    //extending of current path
                    Path newPath = aktPath.extend(n)
                    IF newPath.cost()<T.get(newPath.last()) THEN //update if optimal for now
                        T.update(newPath.last,newPath.cost)
                        Q.insert(newPath,newPath.cost)
                    ENDIF
                ENDDO
            ENDIF
    ENDWHILE
    RETURN NULL //there is no path
ENDFUNCTION
```

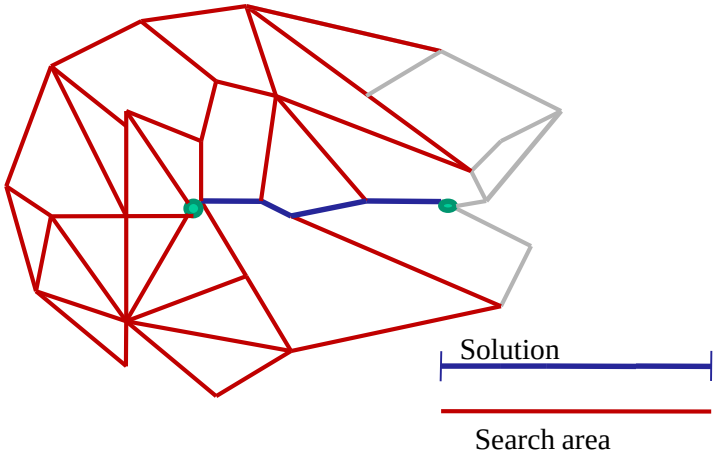
A*-Search

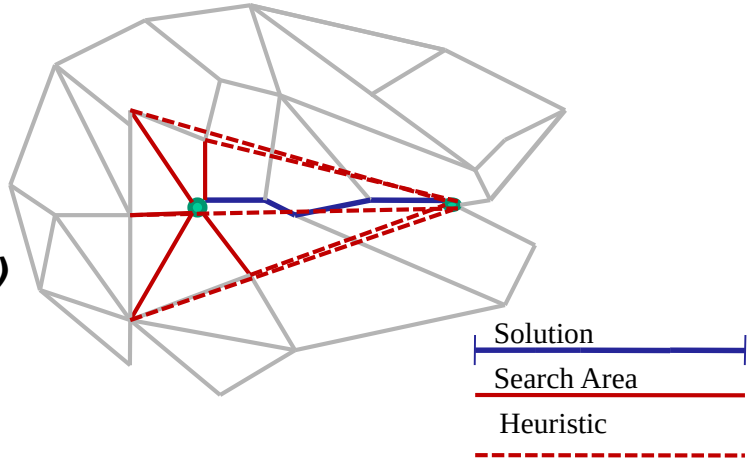
- Dijkstra's algorithm contains no information of the direction to the target

  => expansion of the search into all directions until the target is found
- But indications exist for the direction the search should head
- A*-Search formalizes these "indications" into an optimistic forward estimation:

  h(n, target) for a node n
- h(n,target) indicates a lower bound for the minimum cost to reach the target
- improves the order of the search through sorting the priority queue by minimal total cost to target
- allows to prune path P as soon as it's costs plus the heuristic are greater than the best result up to now:

  P.cost()+h(pfad1.last(),target) > bestPath.cost()





- standard heuristic to estimate the route:

  Euclidian distance between actual position and target position

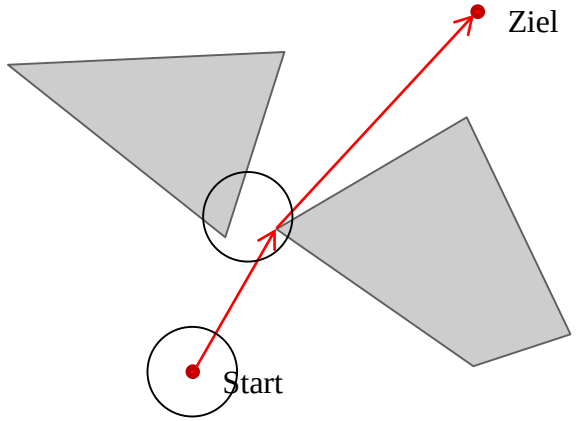
Pseudo Code: A*-Search

Pseudo Code:

```
FUNCTION Path shortestPath(Node start, Node target)
    Q.insert(new Path(start),0)
     WHILE(Q.notIsEmpty())
          Path aktPath = Q.getFirst()
          IF aktPath.last()  == target THEN  //found result
               return aktPath
          ELSE
               FOR Node n  in aktPath.last().successor() DO      //expanding the current path
                    Path newPath = aktPath.extend(n)
                    IF newPath.cost()<T.get(newPath.last()) THEN   //update if optimal so far
                         T.update(newPath.last, newPath.cost())
                         Q.insert(newPath, newPath.cost() +h(newPath.getLast(), target))
                    ENDIF
               ENDDO
          ENDIF
    ENDWHILE
    RETURN NULL         //there is no path
ENDFUNCTION
```

Visibility Graph for extended objects

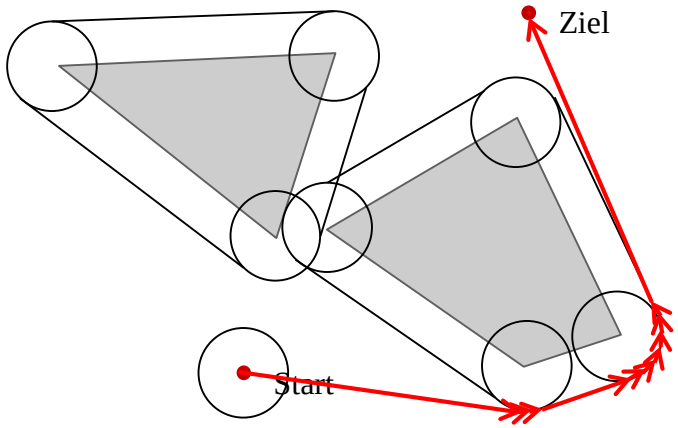
- MOBs usually have a spatial expansion: Circle or Polygon
- Visibility graph is only feasible for point-routing
- Adjusting the visibility graph regarding the spatial expansion of the object: Polygons are being expanded by the spatial extension (Minkowski Sum)



Problem with this solution:

- For circular expansion: Circles have an infinite number of corners
  => Visibility graph is not derivable
- For Polygon-Environment: object rotation should be considered

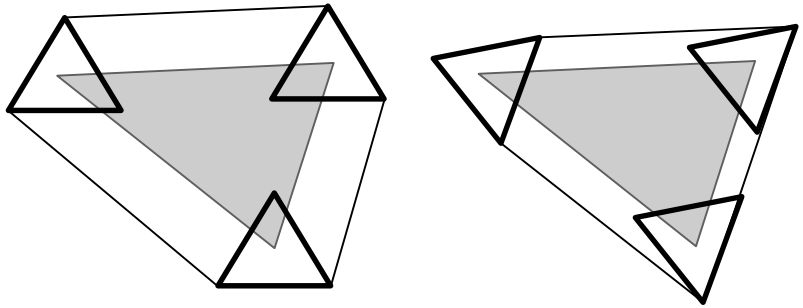
  => Every rotation requires a separate extension



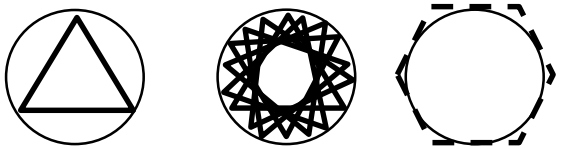
Visibility Graph for extended Objects

Solution Approach:

- Polygons are approximated by surface of revolution
  => Circle
- Circles are approximated by minimal surrounding polygons (MUP)
  => e.g. hexagon, octagon
- Form Minkowski sum with the MUPs and derive visibility graph.



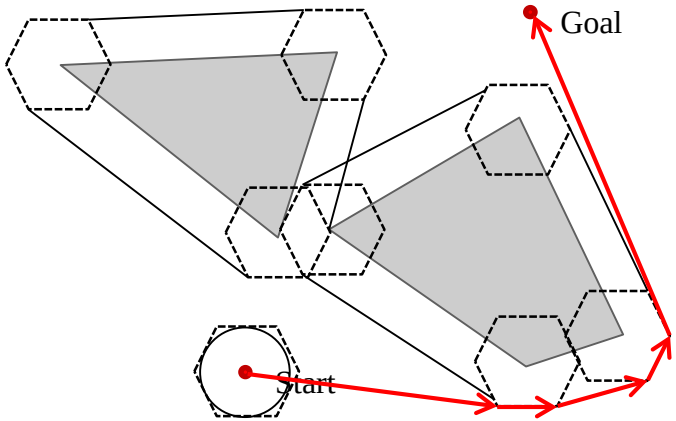
Minkowski-Sum in different rotations of the same shape



Double approximation of surface of revolution and minimal surrounding Hexagon

Remarks:

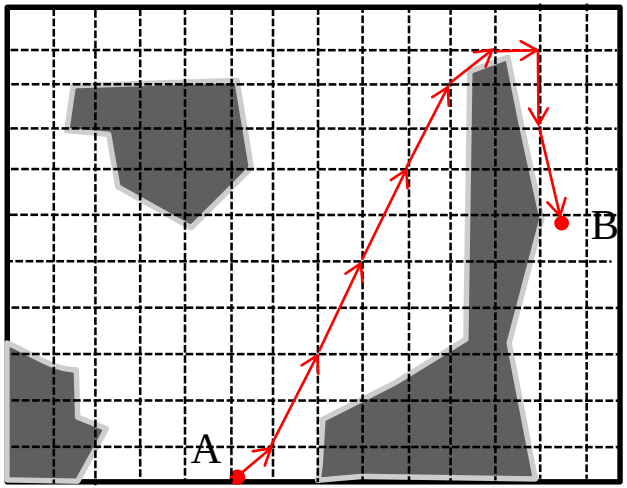
- Paths are not optimal
- Passages are considered conservatively
- Curves are taken angular
- MMO should only have a limited amount of bounding box sizes because each requires it's own graph

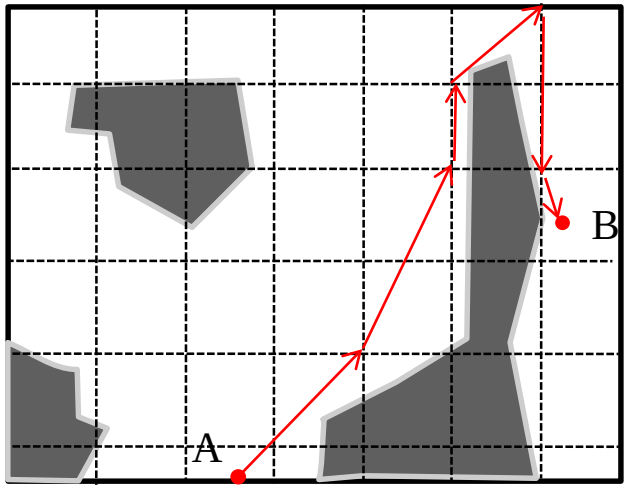


More Pathfinding Methods

Other Methods:

- approximation polygons through polygons with less corners
- hierarchic Routing for longer routes
- precalculate and store shortest paths in dedicated data structures
- grid based graphs: overlay the map with a grid and route over cell centers.





Conclusion:

Routing is an old but still active field of research in computer sciences.

Antagonistic Player Behavior

How is an AI capable to react to a player's behavior?

- AI reads the current state of the player and derives a matched action.
- select a matching action with

  => Precalculating/Estimating an action's result

  => Evaluation of expected results with value functions/heuristics

Example: Option 1: Monster M hits player S
=> player S has 900 HP remaining

Option 2: Monster M flees from S
=> Player S has 1000 HP remaining

Evaluation: Option 1 is preferred, since the enemy looses 100 HP.

Antagonistic Search

Problem: Simple model does not take the opponent acting to increase his own advantage into account.

in the example:

M attacks S and hits M for 100 HP

$\Rightarrow$ S hits M for 1000 HP

result: S has 900 HP and M is dead

antagonistic search from game theory offers a formal framework for reactive behavior.

Base Case:

turn based game: Action can be sequenced

=> finite number of alternative actions for every turn

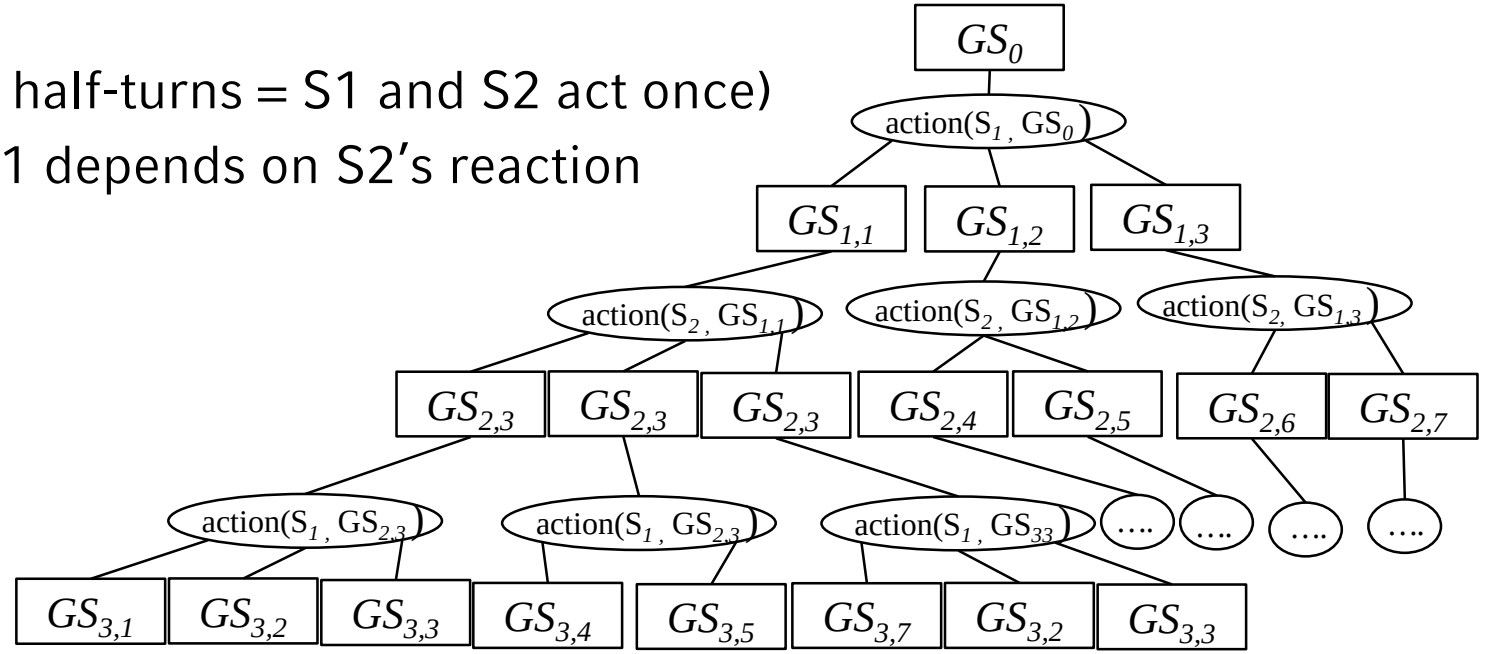
Antagonistic Search

Given:

- GS_i: score before move i. players: S1 and S2.
- actions of player S_j for GS :action(S_j, GS_i) = {A_1, …, A_k)
- valuation function $H: GS \rightarrow IR$ (the higher, the better for player S)

Search-Tree:

- complete tree: contains all possible courses of play (normally to big)
- incomplete search: search GS maximizing H(GS) for h turns
  (1 turn = 2 half-turns = S1 and S2 act once)
- action of S1 depends on S2's reaction

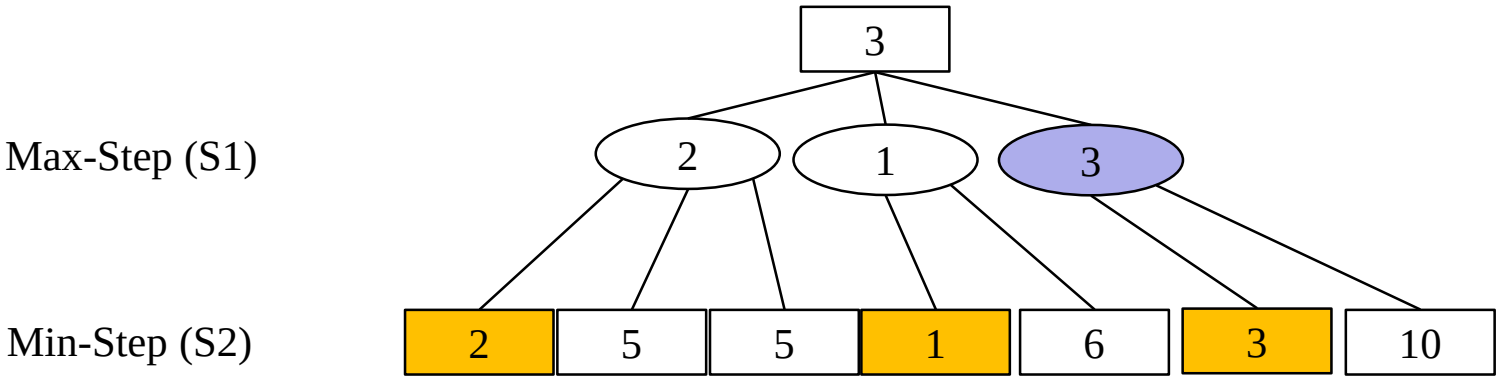


Min-Max Search in antagonistic Search Trees

- rate a move A to maximize H(GS) after S2's reaction (tries to minimize H).
- Search depth:
  - fixed depths

  ⇒ time may vary and is hard to estimate

  ⇒ turbulent positions make pruning of some branches unfavorable
  - iterative deepening:
    - multiple calculations with increasing search depth
    - on time-out: abort and use of last complete calculation (for each level the total cost doubles)
  - turbulent positions: single branches are being expanded if leaves are turbulent.



Alpha-Beta Pruning

Idea: If a move already exists, that is evaluated with α even after a counter reaction, all branches creating a value less than α can be pruned.

- α: S1 reaches at least α on this sub-tree ($H(GS) > \alpha$)
- β: S2 reaches at most β on this sub-tree ($H(GS) < \beta$)

Algorithm:

- traverse search tree with depth-first search and fill inner nodes on the way back to the last branching
- for calculating inner nodes:

  if $\beta < \alpha$ then
  - prune the remaining sub-tree
  - set β-value for the sub-tree if it's root is a min-node
  - set α -value for the sub-tree if it's root is a max-node

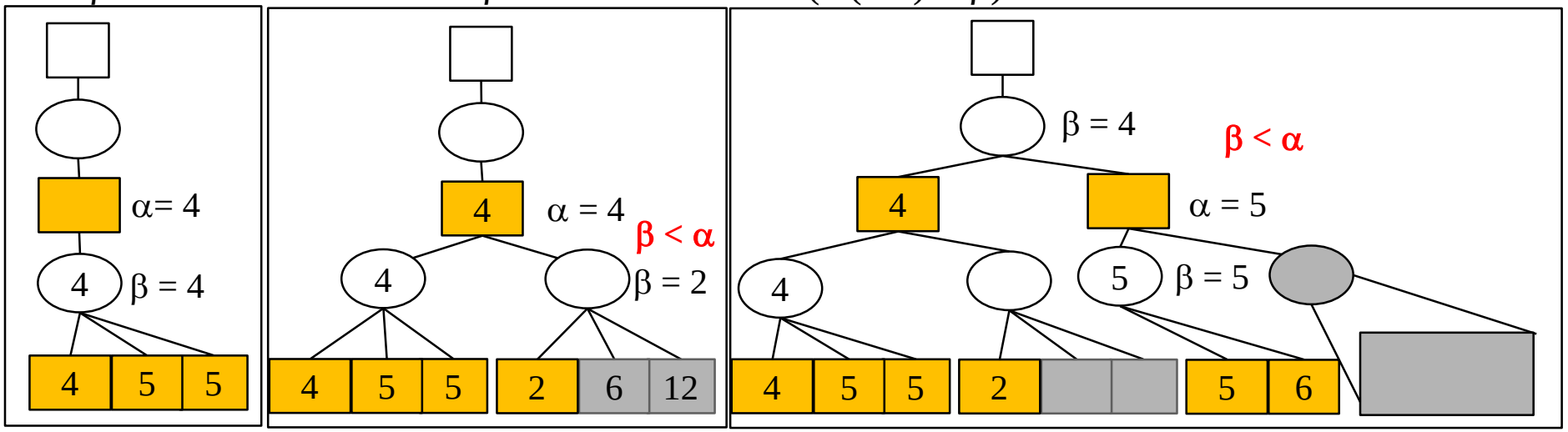
  else set β-value to the minimum of min-nodes

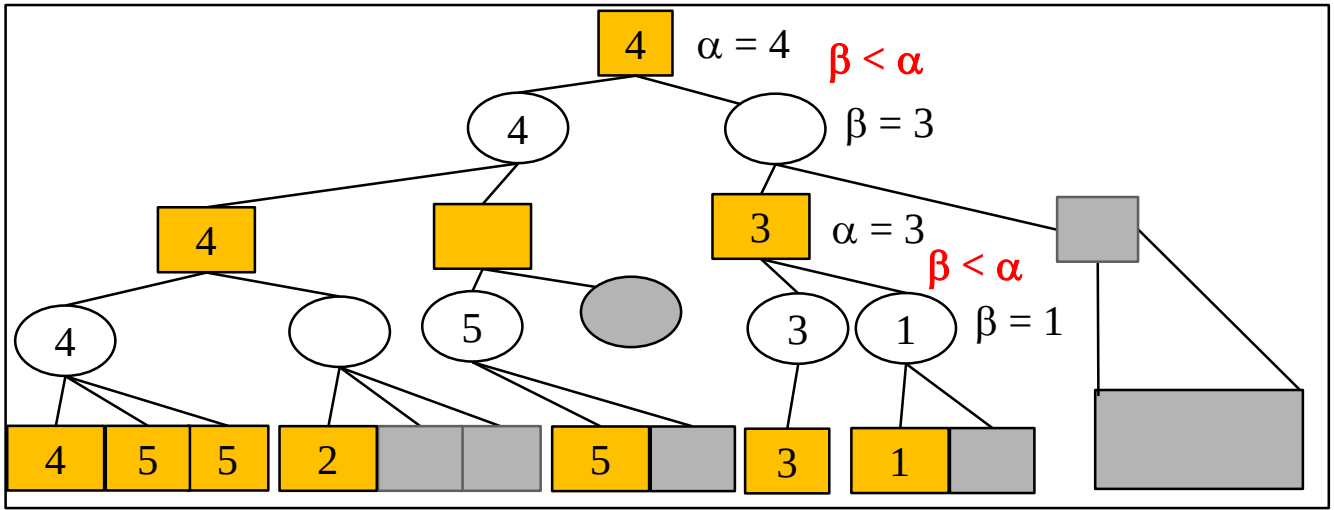
  set α-value to the maximum of max-nodes

Alpha-Beta Pruning

Idea: If a move already exists, that is evaluated with α even after a counter reaction, all branches creating a value less than α can be cut.

- α: S1 reaches at least α on this sub-tree ($H(GS) > \alpha$)
- β: S2 reaches at most β on this sub-tree ($H(GS) < \beta$)





Usability for general Games

expansion to more general games can be reached by adjusting the evaluation function:

- probabilistic games: maximize the expected value (e.g. Backgammon)
- incomplete information (e.g. Hearth Stone, …)

expansion to several players:

- min-nodes consider all players' actions
  (large number of player reactions might generate a large state space)

ceasing time synchronization:

- possible in principle, but the number of possible state transitions generally grows exponentially
- hard to calculate without restricting the search area

conclusion: The basic idea is applicable to any game, but in practice the state space and the state transitions strongly increases with a large number of players, possible temporal sequences and available actions.

Learning Goals

- modeling decision processes
- rules based policies
- state machines
- typical tasks for AI-Engines
- pathfinding in open environments
- pathfinding with expanded objects
- antagonistic Search
- Min-Max Search
- Alpha-Beta Pruning

Literature

- Nathan R. Sturtevant
  Memory-Efficient Abstractions for Pathfinding
  In Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), 2007.
- Nathan R. Sturtevant, Michael Buro
  Partial pathfinding using map abstraction and refinement
  In Proceedings of the 20th national Conference on Artificial Intelligence, 2005.
- Joseph O'Rourke:
  Computational Geometry in C.
  2nd Edition, Cambridge University Press, 1998.
- S. Russel, P. Norvig: **Aritificial Intelligence: a modern approach,**
  Pearson, 2016 (third edition)