Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Managing and Mining Multiplayer Online Games

im Sommersemester 2016

Kapitel 5: Künstliche Intelligenz in Spielen

Skript © 2012 Matthias Schubert

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

Kapitelübersicht

- Steuerung von Computer Controlled Entities
- Regelbasiertes Verhalten
- Zustandsautomaten
- nicht deterministisches Verhalten
- Algorithmen für AI-Engines
  - Wegewahl in offenen Umgebungen
  - Antagonistisches Verhalten

Aktionserzeugung für NPCs

Arten von Computer Controlled Entities:

NPCs (Non-Player Characters), **MOBs** (Mobile Objects) oder **Bots** (automatische Steuerung von Spielercharakteren) müssen Aktionen erzeugen, um handeln zu können.

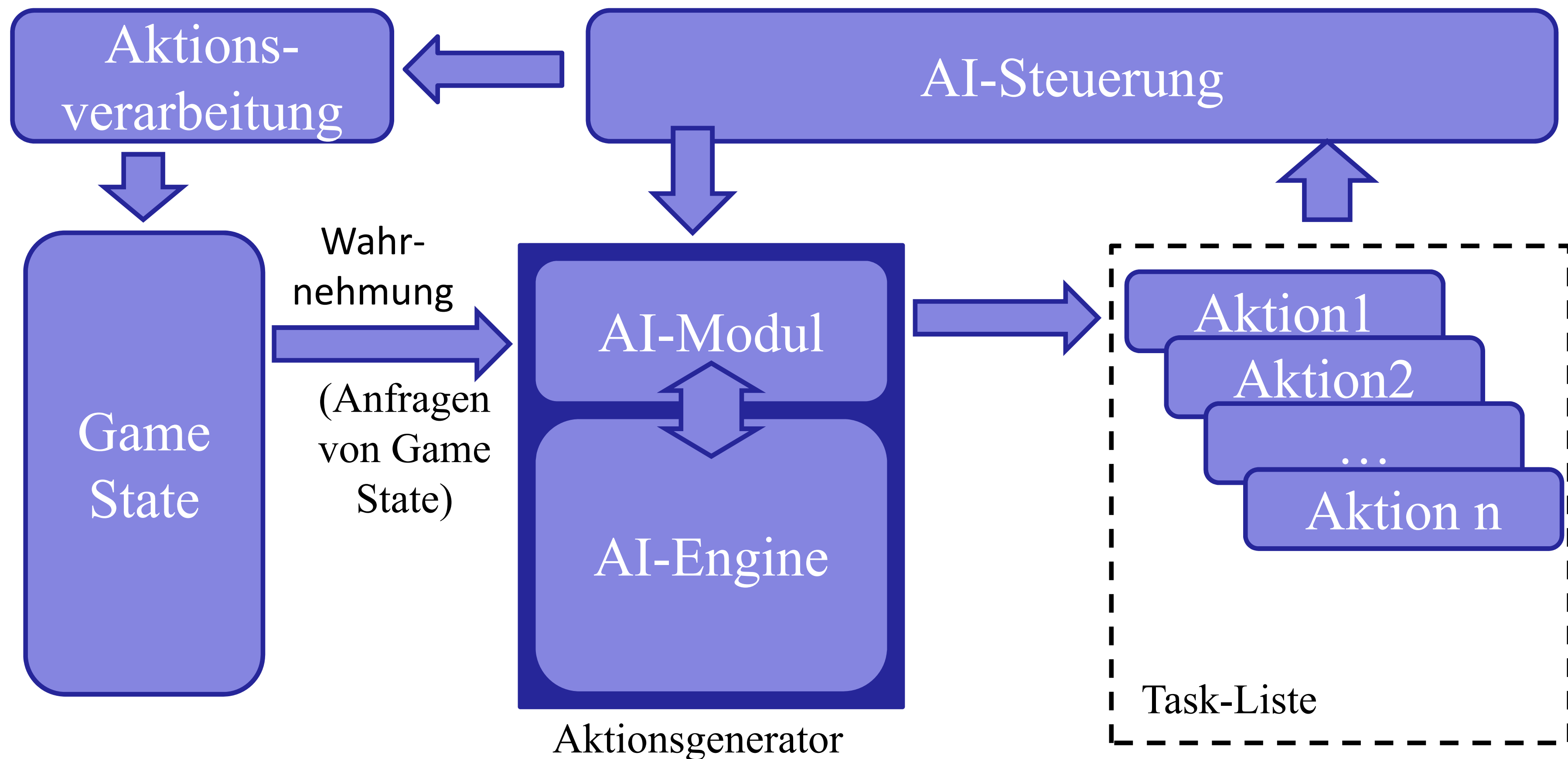
wichtige Aspekte:

- Granularität: Steuerung einer oder mehrerer Game Entities
- Berechnungsaufwand vs. Verhaltenskomplexität
- Wo werden NPCs berechnet (Server/Client-seitig)?
- deterministisches vs. probabilistisches Verhalten
- Realitätsnähe des Verhaltens

Grundaufbau eine AI

Aufgabe:

- Generiere eine/mehrere Handlungen
- die dem jetzigen Zustand angemessen sind



AI-Steuerung

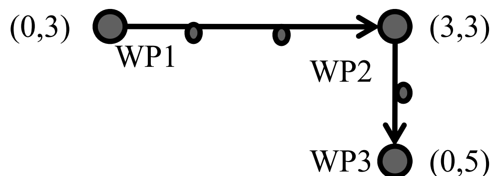
Synchronisation von Aktionsverarbeitung und Aktionserzeugung:

- Aufruf der Aktionserzeugung
  - **Zeitgesteuert**: Handlungswünsche werden regelmäßig erfragt
  - **Event-gesteuert**: Bei bestimmten Änderungen des GS können Handlungswünsche erfragt werden.
    (Spieler kommt in Aggro-Reichweite)
- Durchführung von Tasklisten
  - **Verarbeitungszeitpunkte** einzelner Aktionen in der Task-Liste
    => Durchführen komplexer Handlungen (Sequenzen von Aktionen)
  - Aktion die über einen **Zeitraum** ausgeführt werden und Management von Cool Downs (CDs)

Beispiel:

Bewegung entlang einer Trajektorie:

(WP1,WP2,WP3)



Tick	X	Y	State	
1	0	3	Gehe zu WP2	aus Task-Liste
2	1	3	Gehe zu WP2	
3	2	3	Gehe zu WP2	
4	3	3	Gehe zu WP3	aus Task-Liste
5	3	2	Gehe zu WP3	
6	3	1	Gehe zu WP3	

Aktionsgenerierung

- Wann und was wird vom Game State angefragt?
  Eine zentrale Anfrage an Game State („alle Wahrnehmungen“)
  vs. mehrere Anfragen (selektive Wahrnehmungen)
- Wie werden Handlungen generiert?
  - Organisation über Regeln oder Zustandsautomaten
  - Deterministische Handlungen oder zufallsbasiert?
- Welche Möglichkeiten soll es beim Einfügen
  in die Taskliste geben?
  - Löschen alter Handlungen
  - Einfügen in die Taskliste (Anfang, Ende, …)
- Umsetzung
  - Programmierung in der Sprache der Game Engine
  - Programmierung über Skriptsprache (z.B. LUA)

Beispiel: „Autocamp 2000“

Beispiel eines Bots: **(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."

Beispiel

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!

Beispiel

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 ...]

Steuerung durch Regeln

- **Rumpf**: Prädikatenlogischer Ausdruck auf dem Game State
- **Kopf**: Liste von Aktionen, die in die Task-Liste eingefügt werden
- Steuerung besteht aus einer Liste von Regeln
- Auswahl der Regel über Sortierung nach Priorität
  => erste Regel, die feuert, erzeugt Handlung

Nachteile:

- Regelrümpfe sind häufig redundant
- hoher Aufwand beim finden der anzuwendenden Regel

Beispiel:

- 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()

Steuerung mit Zustandsautomaten

Idee: Fasse Regeln nach Zustand des NPCs zusammen.

⇒ nur die Regeln, die für den aktuellen Zustand gelten, werden ausgewertet

⇒ aktueller Zustand des NPCs ist Teil des Game States

⇒ Zustände können bei komplexen MOBs auch weiter unterteilt werden

⇒ Änderungen des Zustands über Aktionsverarbeitung
(z.B. Monster bricht die Verfolgung ab => Übergang von *battle* zu *base*)

battle

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

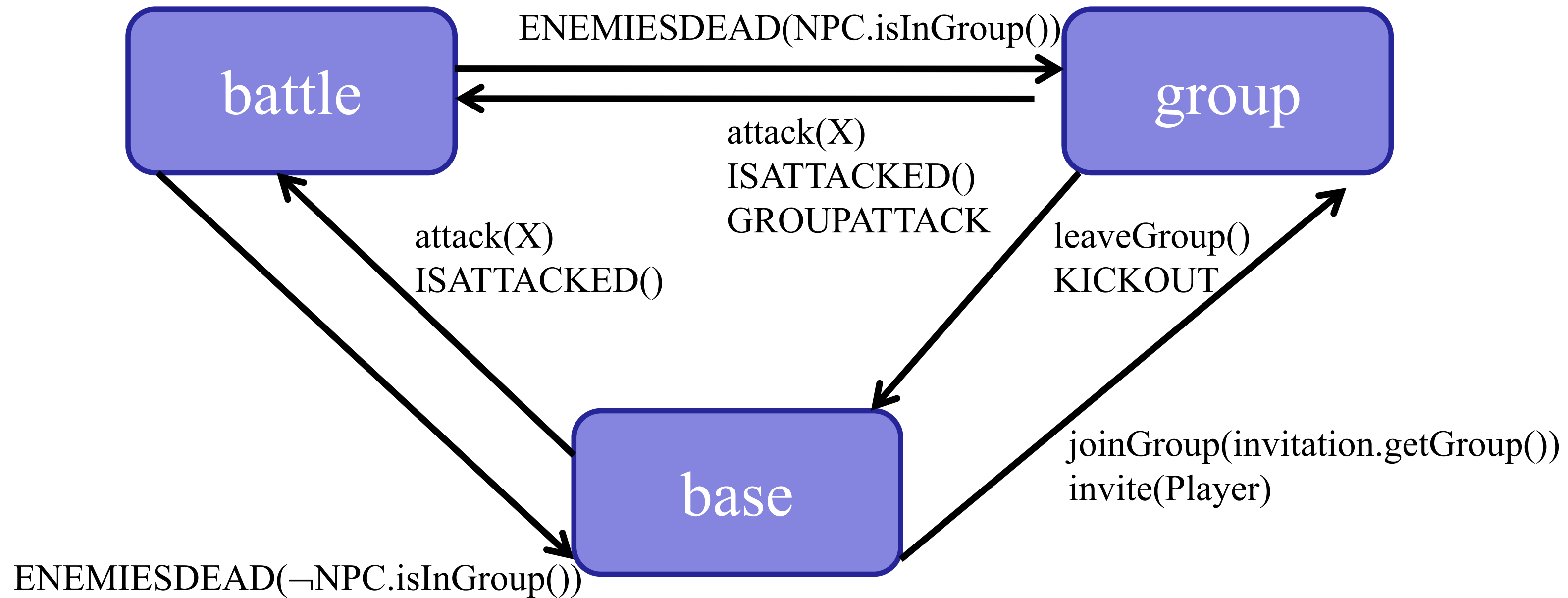
group

- __=>Follow(NPC.GroupLeader)

base

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

Beispiel: Zustandsautomat



- GROSSBUCHSTABEN: eventbasierter Übergang
- Kleinbuchstaben: aktionsbasierter Übergang

Verarbeitung mit Zustandsautomaten

Ablaufschema:

1. Anfrage des ***GS***
   (zustandsbestimmende Informationen)
2. Bestimmen des aktuellen Zustands
   - eindeutige Aufteilung über disjunkte Zustände (z.B. Krieg oder Frieden)
   - eindeutige Bestimmung über Prioritäten (z.B. 1:battle, 2:group, 3:base)
3. Anfrage des ***GS***
   (zustandsspezifische Informationen)
4. Aktionsgenerierung
   (Anwendung der lokalen Regeln)
5. Manipulation der Task-Liste
   - Abbruch der alten Aktion:
     Löschen der TL und Neueinfügen der neuen Handlungen
   - Einfügen der neuen Aktionen am Anfang der Liste
   - Einfügen der neuen Aktionen am Ende der Liste

Determinismus und Verhaltensänderung

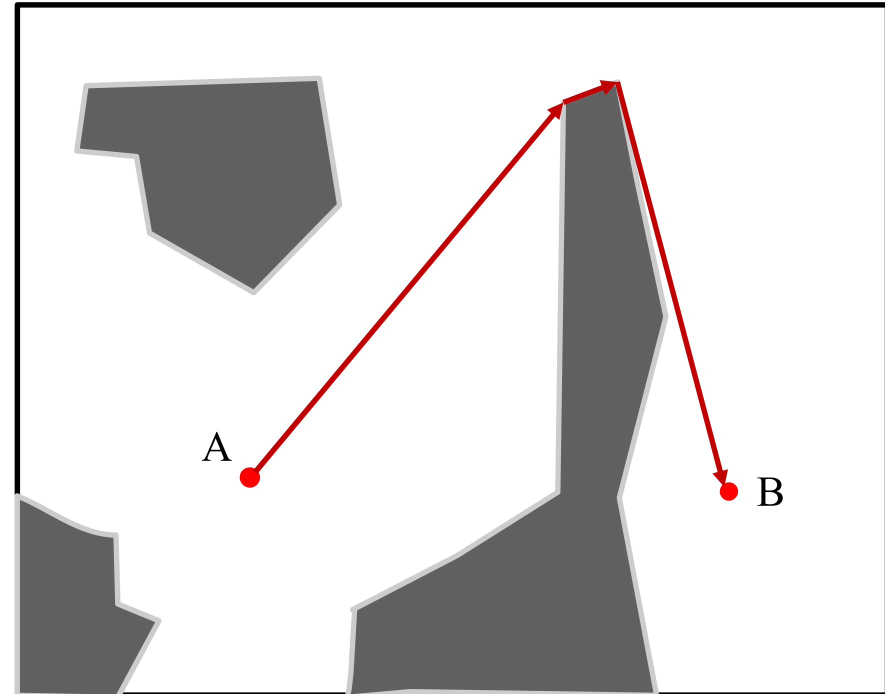
- ***Deterministisches Verhalten***:
  MOBs generieren in der gleichen Situation immer die gleichen Aktionen
- ***Probabilistisches Verhalten:***
  - Regeln haben im Kopf eine Menge möglicher Aktionen
    => Auswahl mittels Zufallsprozess
  - in einer Situation können mehrere Regeln anwendbar sein
    => zufällige Auswahl der Regeln
  - die Bestimmung des aktuellen Zustands oder Unterzustands wird zufällig ausgewählt
- ***Lernendes Verhalten:***
  - spezielle Parameter im Kopf oder Rumpf der Regeln werden auf Basis von Trainingsbeispielen angepasst
    (Verwendung von Klassifikations- und Regressionsmethoden)
    Beispiel: Optimierung der Wahrscheinlichkeitsverteilung verschiedener Angriffe bei bestimmten Gegnern
- ***Evolutionäres Verhalten:***
  - Einfügen neuer Regeln und Zustände im Rahmen der allgemeinen Spielregeln
  - Training über z.B. über genetische Algorithmen
  - Problematisch, da viele neue Regeln erfolglos sein werden
    (sehr experimentell und in den meisten Spielen kaum umsetzbar)

AI-Engine und AI-Dienste

- Aktionsgenerierung und AI-Steuerungen benötigen Basisoperationen, die bestimmte Aufgaben effizient und generisch erledigen:
  - Finden von Ein- und Ausgängen
  - Finden kürzester Wege
  - antagonistisches Verhalten
  - Schwarmverhalten
  - …
- AI-Engine: Sammlung von Diensten, die für die Umsetzung der AIs nützlich sind

Wegewahl in offenen Umgebungen

- offene Umgebung: 2D Raum ($\subseteq IR^2$)
- MOBs dürfen sich frei bewegen
- Hindernisse versperren direkte Verbindungen
- Darstellung der Hindernisse durch:
  - ***Polygone***
  - Pixeldarstellungen
  - beliebige geometrische Flächen (Kreise, Ellipse, …)

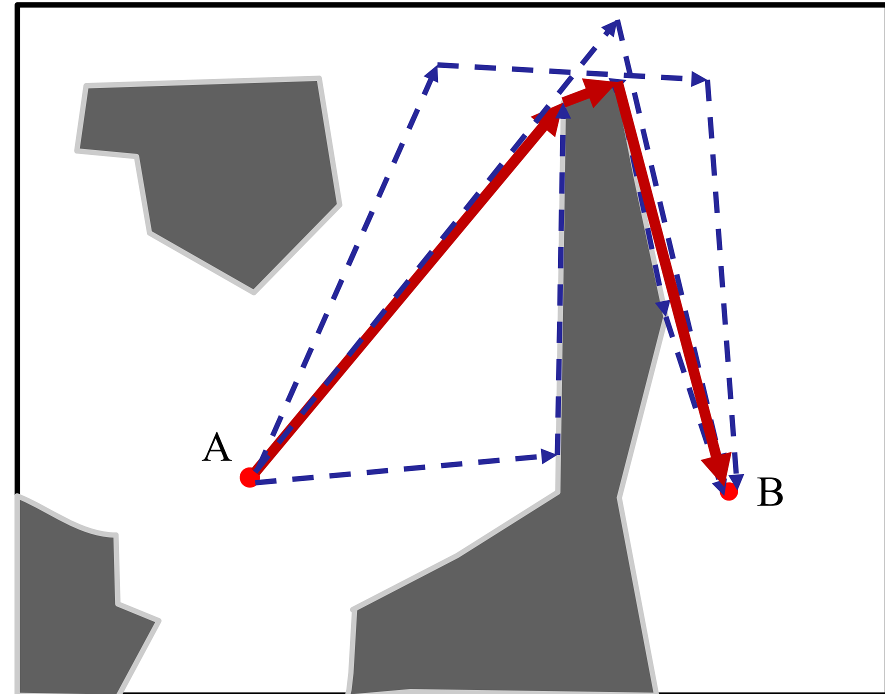


Lösung für Polygon-Darstellung:

- Ableiten eines Graphen für eine Karte auf dem die kürzesten Wege verlaufen (Sichtbarkeitsgraphen)
- Integrieren von Start und Ziel
- Verwendung von Wegewahl-Algorithmen wie Dijkstra, A* oder IDA*

Sichtbarkeitsgraphen

- Finden des kürzesten Pfads in einer offenen Umgebung ist eine Suche in einem unendlichen Suchraum
- **Lösung**: Einschränken des Suchraums durch folgende Eigenschaften von optimalen Pfaden:
  - **Die Wegpunkte jedes kürzesten Pfads sind entweder Start, Ziel oder Ecken der Hindernis-Polygone.**
  - **Pfade dürfen die Polygone nicht schneiden.**



- Der kürzeste Pfad in der offenen Umgebung *U* ist auch Teil des Sichtbarkeitsgraphen $G_U(V,E)$.

Sichtbarkeitsgraphen

Umgebung: U

- Menge von Polygonen $U=(P_1, \ldots, P_n)$ **(Hindernisse)**
- Polygon P: Planarer zyklischer Graph: $P = (V_P, E_P)$

Sichtbarkeitsgraph: $G_U(V,E)$

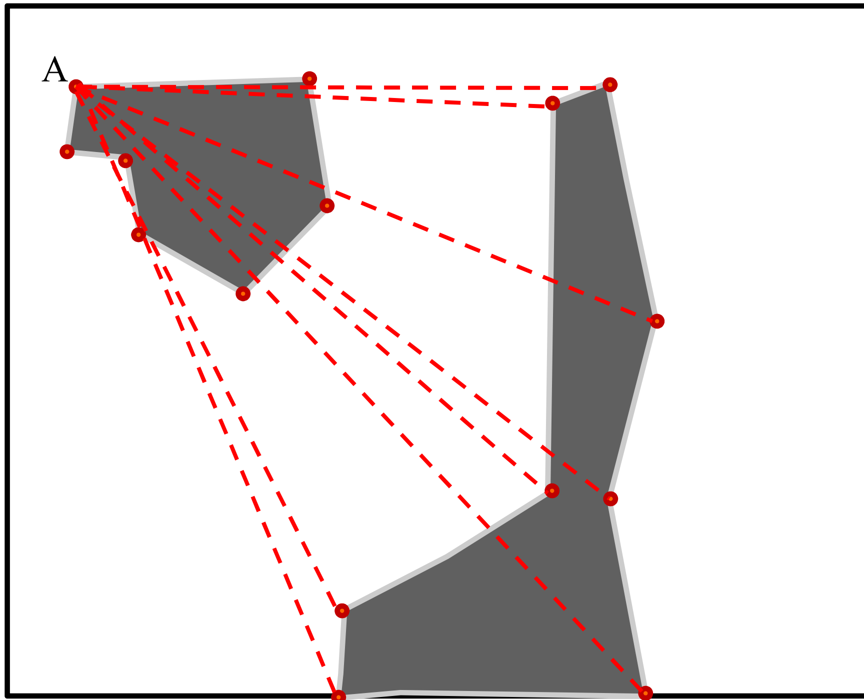
- *Knoten:* Ecken der Polygone P = $\{V_1, \ldots, V_l)$ in U: $V_U = \bigcup_{P \in U} V_P$
- *Kanten:* Alle Kanten der Polygone mit allen Kanten zwischen Knoten aus unterschiedlichen Polygonen, die keine Polygonkanten schneiden.

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

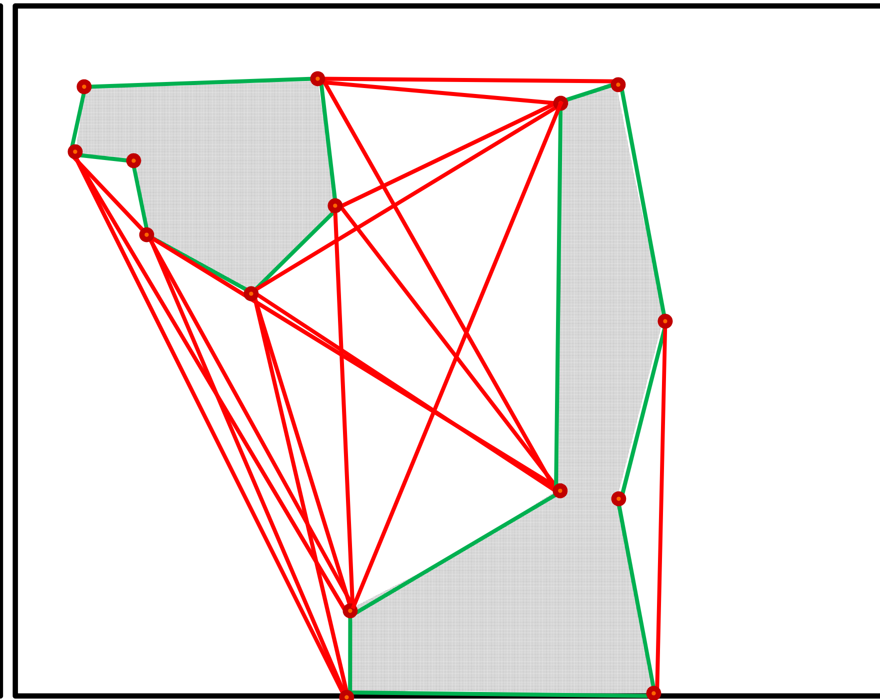
Anmerkungen:

- Diese Definition gilt für konvexe Hindernisse. Bei konkaven Polygonen muss zusätzlich die konvexe Hülle berechnet werden. Für die so berechneten zusätzlichen Kanten ist wieder ein Schnitttest mit Polygonkaten notwendig.
- Diese Definition beinhaltet einen naiven Algorithmus ($O(n^3)$) zur Konstruktion von Sichtbarkeitsgraphen. Die Ableitung von Sichtbarkeitsgraphen kann mehrfach optimiert werden. (O‘Rourke 87: $O(n^2)$)

Beispiel: Sichtbarkeitsgraph



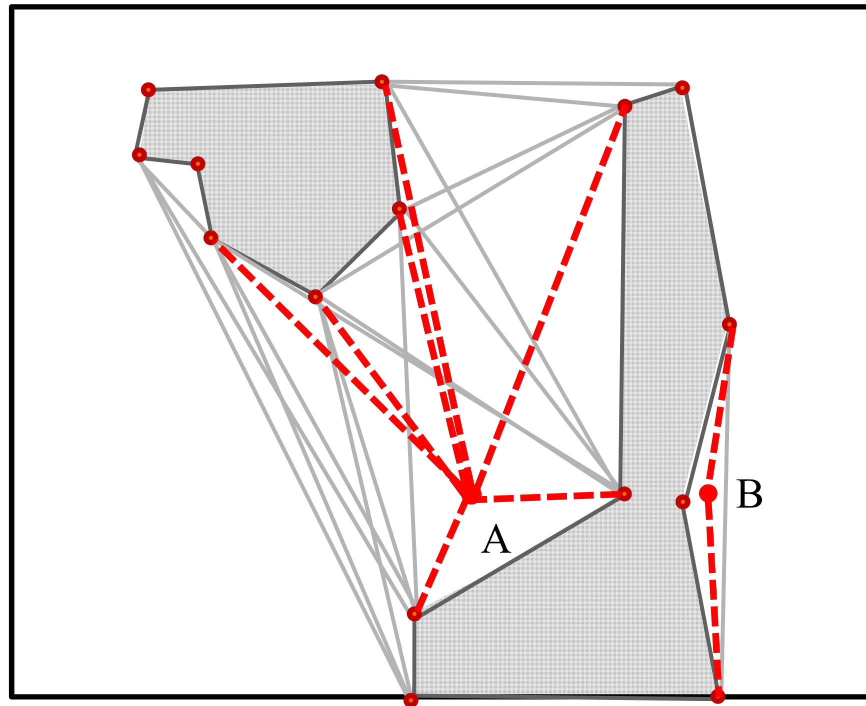
Kanten für den Knoten A, die getestet und verworfen werden.



Sichtbarkeitsgraph: Rote Segmente verlaufen zwischen den Polygonen. Grüne Segmente markieren die Grenzen der Polygone.

Erweiterung um Start-und Zielknoten

- Sichtbarkeitsgraph kann für statische Umgebungen vorberechnet werden
- Bewegliche Objekte müssen vor der Berechnung in den Graphen integriert werden
- Einfügen von Start S und Ziel Z als Punkt-Polygone
- Verbinden der neuen Knoten mit allen Ecken, falls kein Polygonschnitt auftritt



Dijkstra

Verwendete Datenstrukturen:

- PriorityQueue Q (enthält Pfade absteigend nach Kosten sortiert)
- Kostentabelle T (enthält Kosten für aktuell besten Pfad für alle besuchten Knoten)

Pseudo-Code:

```
FUNCTION Path shortestPath(Node start, Node ziel)
    Q.insert(new Path(start,0))
     WHILE(Q.notIsEmpty())
          Path aktPath = Q.getFirst()
          IF aktPath.last()  == ziel THEN                      //Ergebnis gefunden
             return aktPath
          ELSE
             FOR Node n  in aktPath.last().nachfolger() DO        //erweitern des aktuellen Pfades
                 Path newPath = aktPath.extend(n)
                 IF newPath.cost()<T.get(newPath.last()) THEN     //update falls bisher optimal
                      T.update(newPath.last,newPath.cost)
                      Q.insert(newPath,newPath.cost)
                 ENDIF
             ENDDO
          ENDIF
    ENDWHILE
    RETURN NULL                                               //es gibt keinen Pfad
ENDFUNCTION
```

A*-Suche

- Dijkstras Algorithmus hat keine Information über die Richtung in der das Ziel liegt

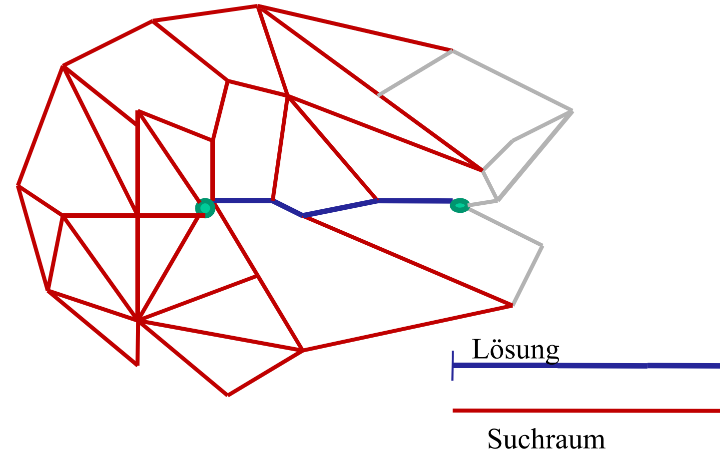
  => Expansion der Suche in alle Richtungen bis das Ziel gefunden wird
- Aber es gibt Hinweise darauf in welche Richtung man suchen sollte
- A*-Suche formalisiert diese „Hinweise“ in einer optimistischen Vorwärtsabschätzung: h(n,Ziel)

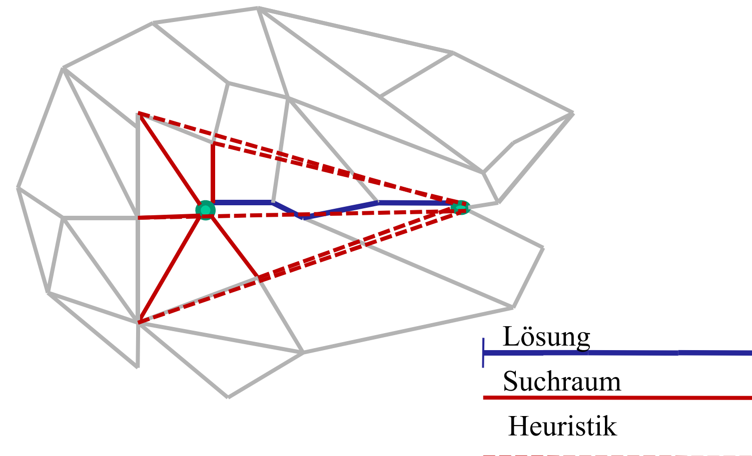
  für einen Knoten n
- h(n,Ziel) gibt eine untere Schranke der Kosten an, die zum Erreichen des Ziels mindestens noch benötigt werden
- Verbessert Suchreihenfolge durch Sortierung nach

  minimalen Gesamtkosten zum Ziel
- Ermöglicht das Abschneiden von Pfad P (Pruning) sobald seine Kosten zuzüglich der Heuristik größer sind als das beste bisherige Ergebnis:

  P.cost()+h(pfad1.last(),Ziel) > bestPath.cost()
- Standardheuristik für Abschätzung der Strecke:

  Euklidischer Abstand zwischen der jetzigen Position und des Ziels





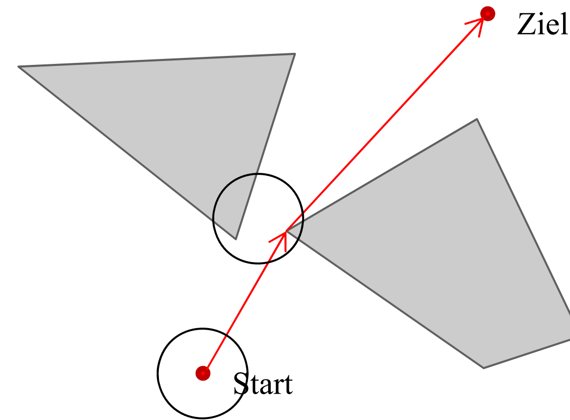
Pseudo-Code: A*-Suche

Peseudo-Code:

```
FUNCTION Path shortestPath(Node start, Node ziel)
   Q.insert(new Path(start),0)
   WHILE(Q.notIsEmpty())
      Path aktPath = Q.getFirst()
      IF aktPath.last()  == ziel THEN                    //Ergebnis gefunden
        return aktPath
      ELSE
        FOR Node n  in aktPath.last().nachfolger() DO     //erweitern des aktuellen Pfades
           Path newPath = aktPath.extend(n)
           IF newPath.cost()<T.get(newPath.last()) THEN   //update falls bisher optimal
              T.update(newPath.last, newPath.cost())
              Q.insert(newPath, newPath.cost() +h(newPath.getLast(), Ziel))
           ENDIF
        ENDDO
      ENDIF
   ENDWHILE
   RETURN NULL                                            //es gibt keinen Pfad
ENDFUNCTION
```

Sichtbarkeitsgraphen für ausgedehnte Objekte

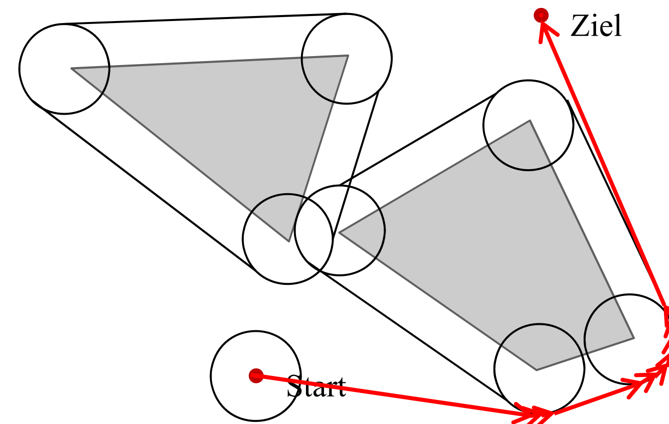
- MOBs haben in der Regel eine räumliche Ausdehnung: Kreis oder Polygon
- Sichtbarkeitsgraph ist nur für Punkt-Routing geeignet
- Anpassung des Sichtbarkeitsgraphen bzgl. der Ausdehnung des Objekts:
  Polygone werden um die räumliche Ausdehnung erweitert (Minkowski Summe)



Probleme bei der Lösung:

- Bei kreisförmiger Ausdehnung: kreisförmige Umgebungen haben unendlich viele Ecken
  => Sichtbarkeitsgraph so nicht ableitbar
- Bei Polygon-Umgebung: Drehung der Objekte sollte berücksichtigt werden

  => Für jede Rotation ist eine eigene Erweiterung notwendig



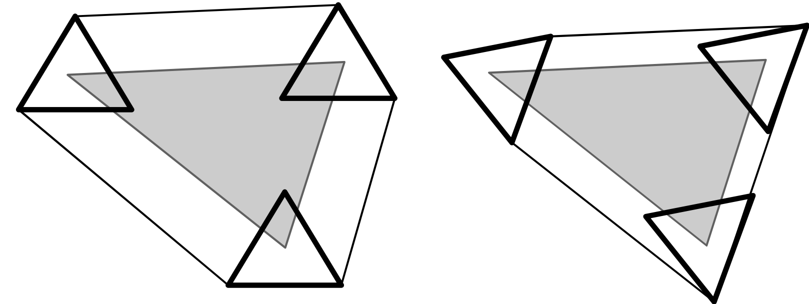
Sichtbarkeitsgraphen für ausgedehnte Objekte

Lösungsansatz:

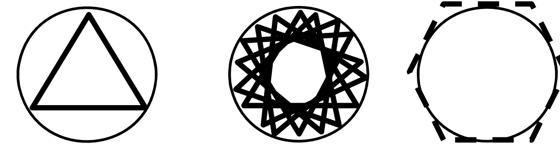
- Polygone werden durch Rotationsfläche approximiert
  => Kreis
- Kreise werden mit minimalen umgebenden Polygonen (MUP) approximiert
  => z.B. Hexagon, Oktagon
- Bilde Minkowski-Summe anhand der MUPs und leite Sichtbarkeitsgraphen ab.

Anmerkung:

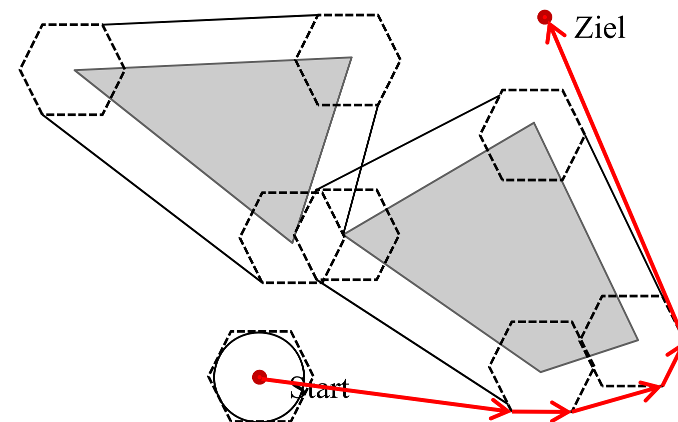
- Wege sind nicht optimal
- Durchgänge werden konservativ betrachtet
- Kurven werden eckig gelaufen
- in MMO sollte es nur eine beschränkte Auswahl an Umgebungsgrößen geben, da jede ihren eigenen Graphen benötigt



Minkowski-Summe mit unterschiedlicher Rotation derselben Ausdehnung



Doppelte Approximation über Rotationsfläche und minimales umgebendes Hexagon



Weitere Methoden zur Wegewahl

Weitere Methoden:

- Approximation der Polygone durch Polygone mit weniger Ecken
- hierarchisches Routing für längere Strecken
- Vorberechnung von kürzesten Wegen und Speicherung in Datenstrukturen
- Verwendung von Grid-basierten Graphen, die Karten in Gridzellen einteilen und das Routing über Zellenmittelpunkte laufen lassen.

A
B

A
B

Fazit:

Wegewahl ist ein altes, aber immer noch aktives Forschungsgebiet in sehr vielen Teilbereichen der Informatik.

Reaktion auf Spielerverhalten

Wie kann eine AI auf das Verhalten des Spielers reagieren?

- AI liest aktuellen Zustand des Spielers und leitet daraufhin eine angepasste Aktion ab.
- Auswahl der passenden Aktion über

  => Vorberechnen/Schätzen des Ergebnisses der Handlung

  => Bewerten des erwarteten Ergebnisses mit Bewertungsfunktion/Heuristik

Beispiel:

Option 1: Monster M schlägt Spieler S
=> Spieler S hat noch 900 HP

Option 2: Monster M flieht vor S
=> Spieler S hat 1000 HP

Bewertung: Option 1 ist besser, da der Feind 100 HP verliert.

Antagonistische Suche

Problem: Einfaches Modell berücksichtigt nicht, dass der Gegner ebenfalls handelt, um selbst seinen Vorteil zu vergrößern.

Im Beispiel:

M greift S an und schlägt M für 100 HP

$\Rightarrow$ S schlägt M für 1000 HP

Ergebnis: S hat 900 HP und M ist tot

Antagonistische Suche aus der Spieltheorie gibt einen formellen Rahmen für reaktives Verhalten.

Basisfall:

- Zugbasierte Spiele: Handlungen lassen sich sequentialisieren
- endliche Anzahl an Handlungsalternativen bei jedem Zug

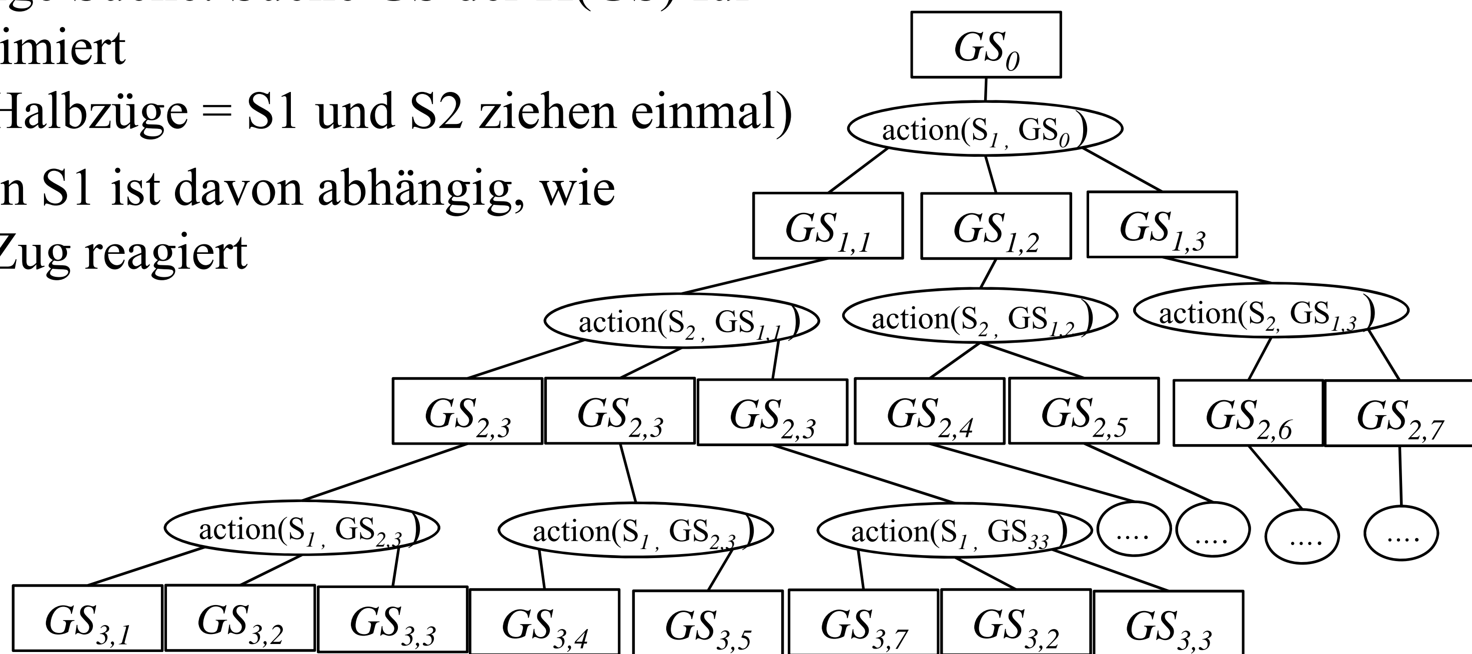
Antagonistische Suche

Gegeben:

- GS_i: Spielstand vor Zug i. Spieler: S1 und S2.
- Aktionen von Spieler S_j für GS :action(S_{j}, GS_i) = {A_1, …, A_k)
- Bewertungsfunktion $H: GS \rightarrow IR$ (je höher, desto besser für Spieler S)

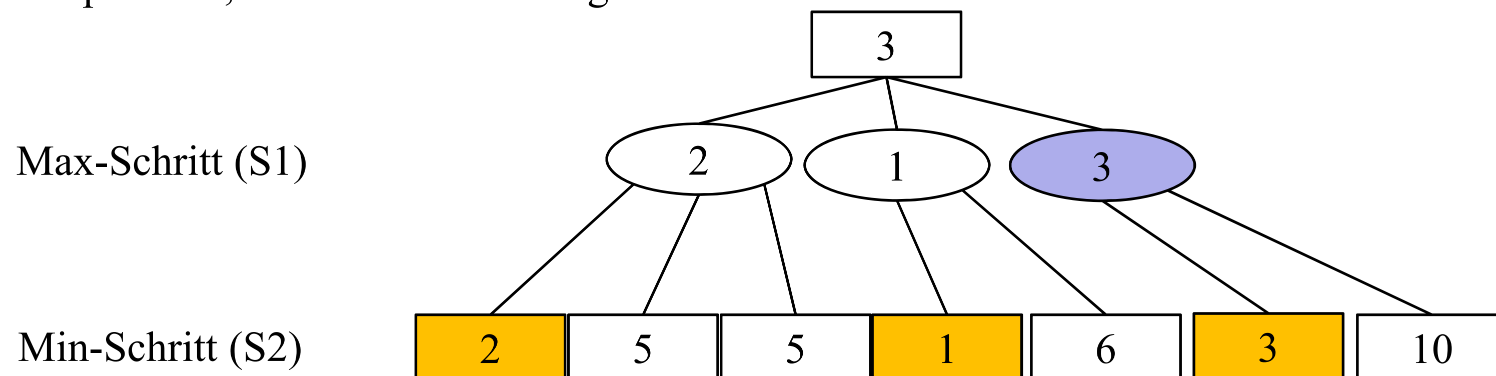
Suchbaum:

- vollständiger Baum: enthält alle möglichen Spielverläufe (i.d.R zu groß)
- unvollständige Suche: Suche GS der H(GS) für h Züge maximiert (1 Zug = 2 Halbzüge = S1 und S2 ziehen einmal)
- Auswahl von S1 ist davon abhängig, wie S2 auf den Zug reagiert



Min-Max Suche in antagonistischen Suchbäumen

- Bewerte einen Zug A so, dass H(GS) nach Reaktion von S2 (versucht H zu minimieren) maximal wird.
- Suchtiefe:
  - Vorgegebene Anzahl an Zügen
  - ⇒ Zeit kann variieren und ist schwer abschätzbar
  - ⇒ unruhige Stellungen machen abschneiden einiger Äste unvorteilhaft
  - Iterative Deepening:
    - mehrfache Berechnung mit steigender Suchtiefe
    - bei Time-Out: Abbruch und Verwendung der letzten vollständigen Bewertung (da sich Aufwand im Schnitt verdoppelt, ist Aufwand ca. doppelt so hoch)
  - unruhige Stellungen: einzelne Äste werden weiter expandiert, wenn Blätter unruhig sind.



Alpha-Beta Pruning

Idee: Wenn es schon einen Zug gibt, der selbst nach Gegenreaktion noch mit α bewertet wird, können alle Äste, die weniger als α Wert erzeugen abgeschnitten werden.

- α: S1 erreicht mindestens α auf diesem Teilbaum ($H(GS) > \alpha$)
- β: S2 erreicht maximal β auf diesem Teilbaum ($H(GS) < \beta$)

Algorithmus:

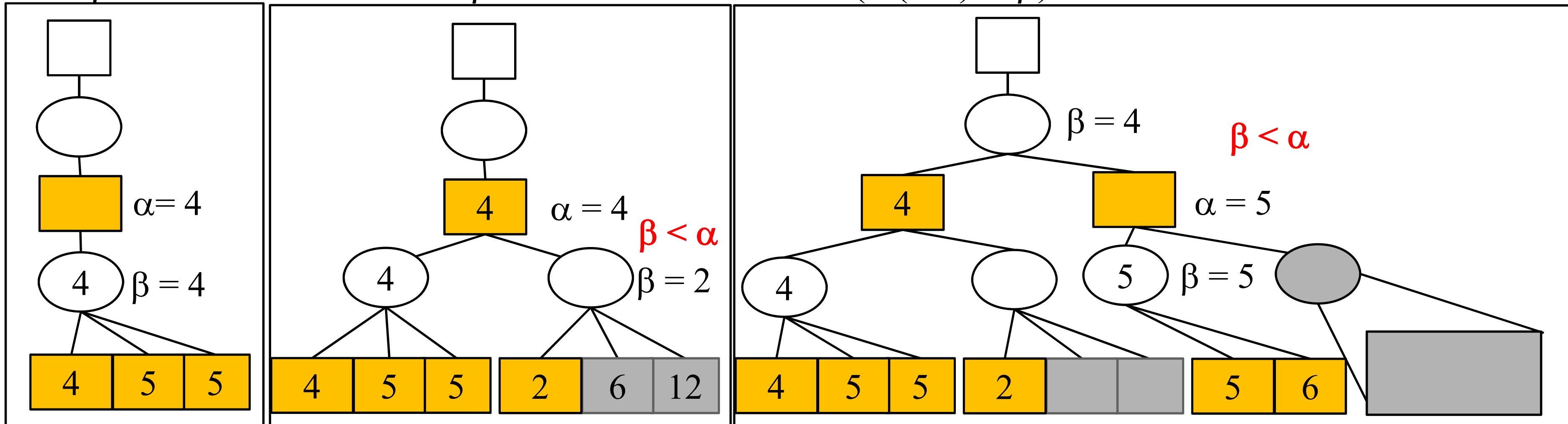
- Durchlaufe Suchbaum mit Tiefensuche und fülle innere Knoten bei Rückweg zur letzten Verzweigung
- beim Berechnen der inneren Knoten:
  Wenn $\beta < \alpha$ dann
  - schneide restlichen Teilbaum ab
  - setze β-Wert für den Teilbaum, falls seine Wurzel ein Min-Knoten ist
  - setze α -Wert für den Teilbaum, falls seine Wurzel ein Max-Knoten ist

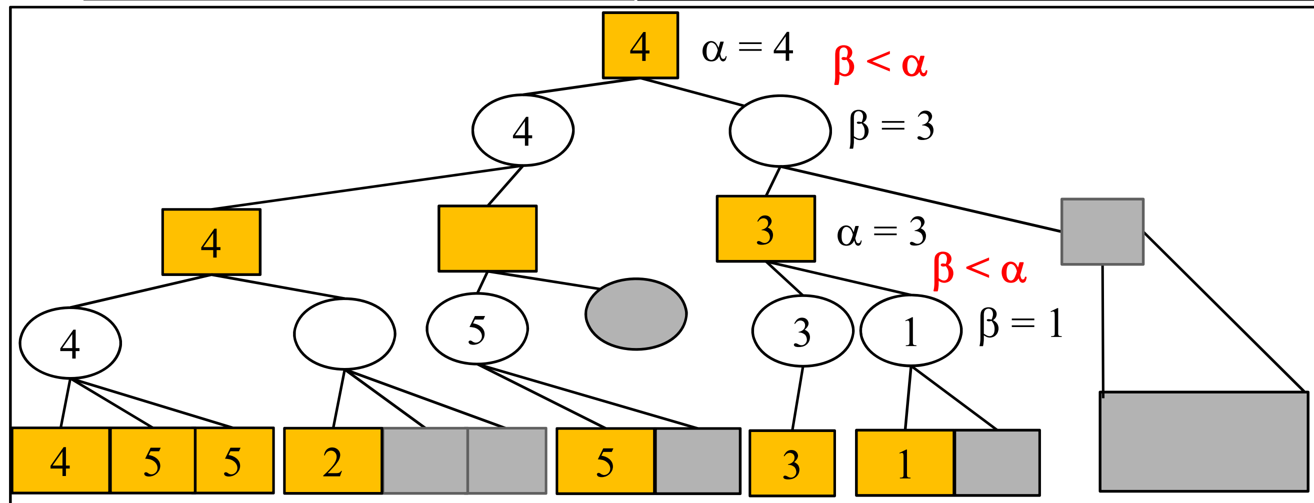
  Sonst setze β-Wert auf Minimalwert bei Min-Knoten
  setze α-Werte auf Maximalwert bei Max-Knoten

Alpha-Beta Pruning

Idee: Wenn es schon einen Zug gibt, der selbst nach Gegenreaktion noch mit α bewertet wird, können alle Äste, die weniger als α Wert erzeugen gepruned werden.

- α: S1 erreicht mindestens α auf diesem Teilbaum (H(GS) > α)
- β: S2 erreicht maximal β auf diesem Teilbaum (H(GS) < β)





Anwendbarkeit für allgemeine Spiele

Erweiterung auf allgemeinere Spiele können meist durch Anpassung der Bewertungsfunktion erreicht werden:

- probabilistische Spiele: Maximieren des Erwartungswerts (z.B. Backgammon)
- unvollständige Informationen (z.B. Stratego, Skat, …)

Erweiterung auf mehrere Spieler:

- Min-Knoten berücksichtigen die Aktionen aller Spieler
  (Hohe Zahl von Spielerreaktionen können durch Kombination mehrerer Reaktionen entstehen)

Wegfallen der Zeit-Synchronization:

- Prinzipiell möglich, aber Anzahl der möglichen Spielverläufe wächst im Allgemeinen Fall exponentiell
- Ohne Einschränkung des Suchraums schwierig zu berechnen

Fazit: Die Grundidee ist prinzipiell auf beliebige Spiele übertragbar, aber der Einsatz scheitert meist an der hohen Anzahl von Spielverläufen, die durch die Anzahl der Spieler, mögliche zeitliche Verläufe und allgemeinen Handlungsoptionen entstehen können.

Lernziele

- Aufbau der Steuerung von computergesteuerten Entitäten
- Regelbasierte Steuerung
- Zustandsautomaten für das Verhalten
- Typische Aufgaben von AI-Engines
- Wegewahl in offenen Umgebungen
- Wegewahl mit ausgedehnten Objekten
- Antagonistisches Suchen
- Min-Max Suche
- Alpha-Beta Pruning

Literatur

- 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.