# 1 Uninformed en informed search

**Vul volgende tabel aan. De tabel bevat voor verschillende zoekalgoritmes de tijds- en ruimtecomplexiteit, of het algoritme al dan niet compleet is (steeds een doelnode vindt als die er is), en al dan niet optimaal is (steeds het kortste pad naar de meest dichtstbijzijnde doelnode vindt). Veronderstel dat de doelnode gezocht wordt in een eindige boom met diepte m, waarbij elke node die geen blad is, b kinderen heeft:**

| Algoritme | Tijdscomplexiteit | Ruimtecomplexiteit | Compleet | Optimaal |
|---|---|---|---|---|
| Breadth-First Search | $O(b^m)$ | $O(b^m)$ | Ja | Ja |
| Depth-First Search | $O(b^m)$ | $O(bm)$ | Nee | Nee |
| Iterative Deepening | $O(b^d)$ | $O(bd)$ | Ja | Ja |

## Is het uniform cost search algoritme optimaal (vindt het steeds het kortste pad tot een doelnode)? Bewijs of geef een tegenvoorbeeld.

Ja, het **Uniform Cost Search (UCS)** algoritme is optimaal, mits aan bepaalde voorwaarden is voldaan. UCS vindt steeds het kortste pad naar een doelnode in een zoekprobleem waar de kosten van de randen (edge costs) niet negatief zijn. Hier is een bewijs voor de optimaliteit en de reden waarom negatieve gewichten problemen kunnen veroorzaken:

### Bewijs voor optimaliteit van UCS

1. **Eigenschap van UCS:** UCS is een variatie van een breedte-eerst zoekalgoritme dat prioriteit geeft aan de kosten van het pad. Het gebruikt een **prioriteitsqueue** waarin nodes worden gesorteerd op basis van de cumulatieve kosten vanaf de startnode tot de huidige node.
2. **Optimaliteit door prioriteitsqueue:**
   - Bij UCS wordt altijd de node met de laagste cumulatieve kosten als eerste verwerkt.
   - Wanneer een doelnode wordt bereikt, is het pad naar die doelnode gegarandeerd minimaal. Dit komt doordat UCS de kosten van alle mogelijke paden naar elke node in oplopende volgorde verwerkt. Zodra een doelnode wordt geselecteerd, zijn er geen kortere paden meer mogelijk.
3. **Geen negatieve gewichten:** UCS veronderstelt dat de gewichten van de randen (edge costs) niet negatief zijn. Dit zorgt ervoor dat eenmaal verwerkte nodes niet opnieuw verwerkt hoeven te worden, omdat een pad niet korter kan worden na het toevoegen van extra positieve kosten.
4. **Conclusie:** Aangezien UCS altijd de goedkoopste optie kiest en dit proces voortzet tot de doelnode wordt bereikt, is het gegarandeerd dat UCS het kortste pad vindt.

### Tegenvoorbeeld met negatieve gewichten

Als er negatieve gewichten in de graaf zijn, kan UCS falen. Dit komt doordat een pad dat eerder als optimaal werd beschouwd, later overschreven kan worden door een pad met een lagere cumulatieve kosten.

**Voorbeeld:** Overweeg de volgende graaf met negatieve gewichten:

- Start: $A$
- Doel: $D$
- Randen en kosten:
  - $A \to B$ met kosten 1
  - $B \to C$ met kosten 2
  - $C \to D$ met kosten $-4$
  - $A \to D$ met kosten 5

**Stap-voor-stap UCS:**

1. UCS begint bij $A$ en zet $B$ (kosten 1) en $D$ (kosten 5) in de prioriteitsqueue.

2. $B$ wordt verwerkt, en $C$ wordt toegevoegd met cumulatieve kosten $1 + 2 = 3$.
3. $C$ wordt verwerkt, en $D$ wordt toegevoegd met cumulatieve kosten $3 + (-4) = -1$.
4. Uiteindelijk kiest UCS $D$ met de negatieve kosten $-1$, wat het correcte resultaat lijkt.

**Probleem:** Als $A \to B \to C \to D$ meerdere keren verwerkt moet worden vanwege negatieve gewichten, kan UCS in een oneindige lus terechtkomen. Hierdoor wordt het algoritme inefficiënt en mogelijk niet optimaal.

#### Conclusie

Uniform Cost Search is **optimaal** zolang de graaf geen negatieve gewichten bevat. In het geval van negatieve gewichten is UCS niet gegarandeerd optimaal en kan het falen. In dergelijke gevallen is een ander algoritme, zoals **Bellman-Ford**, beter geschikt.

## Is het best search first algoritme optimaal (vindt het steeds het kortste pad tot een doelnode)? Bewijs of geef een tegenvoorbeeld.

Het **Best First Search** (BFS) algoritme is niet gegarandeerd optimaal, tenzij specifieke voorwaarden worden voldaan. Hieronder wordt dit uitgelegd, gevolgd door een tegenvoorbeeld om de niet-optimaliteit aan te tonen.

#### Best First Search (BFS) Algoritme

Best First Search selecteert de volgende te verkennen knoop op basis van een **evaluatiefunctie** $f(n)$. Deze evaluatiefunctie kan gebaseerd zijn op verschillende heuristieken, bijvoorbeeld de geschatte afstand tot de doelknoop.

- **Optimaliteit hangt af van de gebruikte heuristiek**. Als de heuristiek niet consistent of niet minimaal is, kan BFS niet garanderen dat het het kortste pad vindt.

#### Voorwaarden voor Optimaliteit

1. **Heuristiek is consistent**: Een heuristiek is consistent als voor elke knoop $n$ en een buurknop $n'$, geldt dat:
$$h(n) \leq c(n, n') + h(n')$$
waarbij $h(n)$ de geschatte kost is van $n$ naar het doel, en $c(n, n')$ de werkelijke kost tussen $n$ en $n'$.
2. **Heuristiek is minimaal**: Dit betekent dat $h(n) \leq h^*(n)$, waarbij $h^*_(n)$ de werkelijke minimale kost is van $n$ naar het doel.

Als de heuristiek inconsistent is of overschat, kan BFS een suboptimaal pad kiezen omdat het verkeerde knopen prioriteert.

#### Tegenvoorbeeld

Laten we een graaf beschouwen waarin de heuristiek niet consistent is.

#### Graaf:

- Knopen: $A, B, C, D$
- Paden:
  - $A \to B$ met kost 1
  - $A \to C$ met kost 5
  - $B \to D$ met kost 10
  - $C \to D$ met kost 1
- Heuristiek ($h$):
  - $h(A) = 6$
  - $h(B) = 4$
  - $h(C) = 2$
  - $h(D) = 0$ (doelknoop)

**Verkenning met BFS:**

1. Begin bij $A$, kies knoop met laagste $h$:
   - $h(B) = 4$, $h(C) = 2$
   - Knoop $C$ wordt gekozen (laagste $h$).
2. Ga naar $C$, kies volgende stap:
   - Vanuit $C \to D$, totale kost $5 + 1 = 6$.

**Kortste pad:** Het kortste pad is $A \to B \to D$ met totale kost $1 + 10 = 11$. Echter, BFS kiest $A \to C \to D$ met totale kost 6, wat suboptimaal is.

**Conclusie**

Het Best First Search algoritme is niet optimaal tenzij een consistente en minimale heuristiek wordt gebruikt. In het bovenstaande tegenvoorbeeld toont de inconsistente heuristiek aan dat BFS een suboptimaal pad kiest.

# 2 Adversarial search

## Beschrijf hoe je een AI-speler voor het spel "Mens erger je niet" zou implementeren. Punten worden bepaald door de efficiëntie van jouw oplossing en volledigheid van jouw beschrijving.

Om een AI-speler voor het spel **"Mens erger je niet"** te implementeren, maken we gebruik van een combinatie van beslissingslogica, heuristieken en simulatie om optimale zetten te bepalen. De AI moet keuzes maken die de kans vergroten om de pionnen naar de eindcirkels te brengen, terwijl ze tegelijkertijd de voortgang van tegenstanders probeert te dwarsbomen. Hieronder wordt stap voor stap uitgelegd hoe dit kan worden gerealiseerd.

### 1. Representatie van de speltoestand

Om de AI-speler te laten werken, moet de huidige speltoestand volledig worden gemodelleerd:

- **Spelbord**: Een array van 40 velden (voor de gemeenschappelijke baan) plus 4 sets van 4 velden (eindcirkels) per speler.
- **Thuisbasis**: Elke speler heeft een aparte "thuisbasis" waar 0 tot 4 pionnen kunnen staan.
- **Actieve pionnen**: Pionnen die zich op het gemeenschappelijke bord of in de eindcirkels bevinden.
- **Dobbelsteenworp**: Het resultaat van de worp (1–6).

De speltoestand wordt bijgehouden als een data-structuur, bijvoorbeeld een Python-object of een dictionary.

### 2. Doelen van de AI

De AI heeft de volgende prioriteiten:

1. Breng pionnen naar de eindcirkels.
2. Bescherm eigen pionnen tegen slaan.
3. Sla pionnen van tegenstanders als dat mogelijk is.
4. Minimaliseer het aantal zetten dat nodig is om alle pionnen naar de eindcirkels te krijgen.

### 3. Beslissingslogica

De AI moet beslissingen nemen op basis van de huidige toestand van het spel en de dobbelsteenworp. De logica kan worden geordend in de volgende stappen:

#### a. Opties genereren

- Voor elke pion bepalen welke bewegingen mogelijk zijn op basis van de dobbelsteenworp.
- Mogelijke scenario's:
  - Pion uit de thuisbasis brengen (indien de worp 6 is en er een pion in de thuisbasis staat).

- Een pion verplaatsen op het gemeenschappelijke bord.
- Een pion in de eindcirkels verplaatsen (indien toegestaan door de worp).
- Een pion slaan als deze op het doelveld van de worp staat.

**b. Prioriteiten toewijzen** Elke mogelijke zet krijgt een prioriteitsscore. De prioriteit wordt bepaald door een reeks heuristieken:

1. **Slaan van een tegenstander**: Als een zet resulteert in het slaan van een tegenstander, krijgt deze hoge prioriteit.
2. **Beschermen van eigen pionnen**: Vermijd zetten die een eigen pion blootstellen aan slaan.
3. **Voortgang naar eindcirkels**: Bewegingen die een pion dichter bij de eindcirkels brengen, krijgen een hogere prioriteit.
4. **Gebruik van een pion in de thuisbasis**: Als een pion in de thuisbasis staat en de worp 6 is, moet deze pion worden verplaatst.
5. **Blokkeren van tegenstanders**: Zet een pion op een veld dat strategisch belangrijk is voor tegenstanders (bijvoorbeeld hun thuisbasis).

**c. Beslissing nemen**

- Sorteer alle mogelijke zetten op basis van hun prioriteitsscores.
- Kies de zet met de hoogste prioriteit.
- Bij gelijke prioriteit: kies willekeurig of op basis van aanvullende regels.

**4. Simulatie en evaluatie**

Een meer geavanceerde AI kan zet-opties simuleren en evalueren:

- **Monte Carlo-simulatie**: Simuleer verschillende scenario's voor een zet en kies de optie met het beste gemiddelde resultaat.
- **Minimax-algoritme met heuristieken**: Gebruik een boomstructuur om toekomstige speltoestanden te evalueren en bepaal de optimale zet (eventueel met een dieptebeperking).

**5. Implementatiedetails**

**a. Data-structuren**

- **Spelbord**: Een array met statusinformatie voor elk veld (leeg, bezet door een specifieke speler).
- **Pionnen**: Objecten met attributen zoals huidige positie, status (op het bord, in eindcirkel, in thuisbasis).
- **Speltoestand**: Object dat alle bovenstaande informatie bevat, inclusief de dobbelsteenworp en de actieve speler.

**b. Heuristieken** Een voorbeeld van een scorefunctie om prioriteiten te berekenen:

$$\text{Score} = w_1 \cdot \text{Voortgang naar eindcirkels} + w_2 \cdot \text{Slaan tegenstander} - w_3 \cdot \text{Risico eigen pion}$$

Waar $w_1, w_2, w_3$ gewichten zijn die het belang van elke factor bepalen.

**c. Iteratieve ontwikkeling**

- Begin met een eenvoudige regelgebaseerde AI.
- Voeg simulaties toe om toekomstige zetten te evalueren.
- Optimaliseer de AI door de gewichten en prioriteiten empirisch aan te passen.

**6. Voorbeeldscenario**

1. **Starttoestand**:
   - Pionnen: Eén pion in de thuisbasis, twee op het bord, één in de eindcirkel.
   - Worp: 6.
2. **Opties**:
   - Pion uit de thuisbasis brengen.

- Eén van de pionnen op het bord verplaatsen.

3. **Evaluatie**:
   - Prioriteit 1: Breng de pion uit de thuisbasis.
   - Prioriteit 2: Verplaats een pion op het bord dichter naar de eindcirkels.
4. **Beslissing**: Breng de pion uit de thuisbasis.

### 7. Conclusie

De AI-speler wordt geïmplementeerd met een combinatie van eenvoudige regels en strategische simulaties. Door gebruik te maken van heuristieken en evaluatiemethoden kan de AI flexibel reageren op de veranderende speltoestand en zowel offensieve als defensieve strategieën toepassen.