Quoridor-AI

Graf-Konstruktion (Riktad-Viktad)

Noder

Varje ruta på spelbrädet måste läggas till i en lista för användning, detta betyder att antalet noder som kommer läggas till är bredden multiplicerat med höjden vilket är antalet noder. Det ger en tidskomplexitet på O(N). Utrymmeskomplexiteten blir detsamma eftersom den kommer behöva använda (N × nod) mängd utrymme för att lagra alla noder vilket blir O(N).

Bågar

För varje nod som lags till skall det finnas bågar som går emellan varandra som indikerar för spelaren/motståndare vart de kan gå eller var det är mer lämpligt att gå. För att lagra information om alla bågar kan en närhetslista användas för varje nod. I en nod lagras bågar vilket vardera innehåller information om vilken nod den går till samt vad den har för vikt.

I ett kvadratiskt fält beroende på om rutan ligger i ett hörn eller vid kanten kan de antingen ha 4, 3 eller 2 bågar som går emellan varandra. I fallet av Quoridor kommer antalet bågar maximalt bli $((4 \times 2) + ((bredd - 2) \times 2 \times 3) + ((höjd - 2) \times 2 \times 3) + ((bredd - 2) \times (höjd - 2) \times 4)$ | *alt.* $2(2 * bredd * höjd - bredd - höjd)$ | *alt.* $4(N - \sqrt{N})$

Ex. Quoridor har 9x9 rutor som ger $4(81 - \sqrt{81}) = 288$ bågar.

Däremot har Quoridor väggar som kan sättas ut på spelplanen vilket kommer ta bort bågar som går mellan noder där väggen blivit placerad. För att konstruera en graf med aktuella bågar krävs en algoritm vilket kollar om det finns en vägg placerad mellan denna nod till en granne, och i så fall, lägg inte till en båge mellan dessa. I Quoridor kommer varje nod ha medelvärdet fyra grannar som kommer behöva kollas för varje ruta. Eftersom varje nod kollas och dess grannar kommer algoritmen få en tidskomplexitet på ($\sim 4 \times N$) som ger O(N). För att lagra alla bågar krävs en utrymmeskomplexitet på O(E) där $E \leq 4(N - \sqrt{N})$, N är dominerande faktor vilket således betyder att O(E) kan skrivas om till O(N). För mängden noder som växer kommer antalet bågar växa lika snabbt i utrymme vilket gör de proportionella mot varandra.

Prioritetskö (Max-Min Heap)

För att snabbt hitta en nod med max/min prioritet, kan en prioritetskö användas vilket är en max eller min heap. Den snabbt lagrar det största/minsta värdet på toppen av heapen vilket gör det effektivt för exempelvis A* att alltid välja noden med minst värde i open. Vid insättning och utdrag har den en tidskomplexitet på O(log n). Utrymmeskomplexitet för de operationer är O(1) då de inte kräver något extra utrymme för att utföras.

(Källa: https://www.geeksforgeeks.org/binary-heap/)

A*

Eftersom vi arbetar på ett kvadratiskt fält kan vi ge ett mer noggrant komplexitets-uttryckt. I värsta fall kommer A* besöka varje nod och kolla genom dess bågar för närmaste väg. Utan väggar placerade är medelvärdet för graden per nod lika med 288/81 = 3.55556. Det är det maximala medelvärdet för graden per nod vilken gör det således försumbar. Därför kan tidskomplexiteten för A* skrivas om till O(N) istället för det mer generella $O(b^d)$. Utrymmeskomplexiteten är detsamma O(N) eftersom den kommer i värsta fall lagra mängden noder proportionellt till N.

Problemet med Quoridor är att spelet har flera mål som spelaren kan gå mot. Vanlig A* är designad för att hitta den kortaste vägen mellan start och mål där båda oftast bara finns en av. För att få A* att fungera med flera mål kan en omvänd A* användas. Där istället startpunkten är målet och alla mål är startpunkter för algoritmen. Det betyder att den kommer i början utvärdera för varje start dess grannar och sedan därifrån ta den mest optimala noden till det satta målet. Detta betyder att tidskomplexitet kommer växa linjärt för antalet mål det finns. I Quoridor är det hur lång en sida är eller √N antal noder. I värsta fall är M alltid mindre än N vilket betyder att tidskomplexiteten O(N) inte ändras. Utrymmeskomplexiteten är fortfarande O(N) eftersom efter varje mål utvärderas tas det bort från kön och därefter fortsätter det som normalt. Samma som innan kan storleken för M aldrig bli större än vad det blir vid sökningen i värsta fall vilket betyder att N dominerar över M i utrymme.

Annars fungerar A* som den skall där den använder viktade bågar som längd från start (standard = 1) och distansen från längd till mål som heuristik. Eftersom den söker optimalt till mål blir det oftast mindre än N och vanligen istället längden av ’kortaste väg’. Dessutom, hädanefter benämns ’kortaste väg’ för ’*Pth*’ (kort för path) som en kort förenkling.

Flytta över motståndaren

Kort algoritm som i fall om avancerade regler försöker gå över motståndaren enligt reglerna. Sker när spelarens nästa position är på motståndarens nuvarande position. Fungerar så här,

- Ifall ingen vägg bakom motståndaren.
    - Förflytta som normalt över motståndaren i riktning till där vi försöker gå.
- Ifall vägg/utanför banan bakom motståndaren.
    - Först, kollar om vi kan förflytta diagonalt till en av motståndarens grannar.
        - Om förflyttningen till granne finns inom *Pth*, sätt det som nästa position.
        - Om inte, välj granne som är närmst till mål. (Osannolikt)
    - Annars, betyder det att motståndaren inte har några grannar. (Osannolikt)
        - Välj därför att gå till bästa granne till spelaren med hjälp av samma princip som förra.

För att kolla om vald granne finns inom *Pth* behöver vi bara kolla om grannen finns i position två i *Pth* (0 = spelare, 1 = motståndare, 2 = granne till motståndare).

Eftersom mängden grannar är maximalt fyra kan det ses som försumbart och varje iteration av grannar sker därför på O(1) tid. Således har algoritmen tidskomplexiteten O(1) och utrymmeskomplexiteten O(1) (behöver inget extra utrymme, *Pth* är redan förinitialiserad).

Förutsäg motståndaren

Att förutsäga motståndaren är grundat på att försöka förhindra motståndaren från att försöka placera väggar som kan stoppa dig avsevärt från att nå målet. Det görs genom att hitta positionen som är mest lämplig för motståndaren att placera en vägg och för att förhindra det, lägga en motsatt vägg på den positionen. Exempelvis, motståndaren kan placera en horisontell vägg på [1, 2] som stoppar dig från att nå målet, lägg då en vertikal vägg på [1, 2].

Algoritmen fungerar så att den går igenom varje nod i *Pth* och beroende på riktningen till nästa nod i *Pth* bestäms vilken vägg som skall placeras. Höger-vänster blir vertikal, upp-ner blir horisontell. Sen testas denna placering genom att kolla på hur mycket längre blir spelarens *Pth* och motståndarens *Pth* om en vägg placeras på den positionen. Om *Pth* blir mycket längre för spelaren jämfört med motståndaren så är det en lämplig position att placera en vägg på. Positionen och väggen blir tillsammans ett **Drag**. För draget bestäms även någon poäng (spelaren *Pth* längd) och om denna poäng är större än nuvarande bästa draget, sätts det som nya bästa draget.

I slutet används det bästa draget och vi försöker placera en motsatt vägg på den positionen och ifall om det inte går, avbryt.

Tidskomplexiteten för denna algoritm är tidskomplexiteten för A* upphöjt till 2. Vilket i absolut värsta fall är $O(N^2)$. För varje nod i *Pth* beräknar vi en ny väg *Pth* för att testa den nya placeringen. Extremt kostsamt fast är oftast mycket snabbare om A* följer en optimal väg till mål. Utrymmeskomplexiteten blir högst $O(2 \times N)$ till att lagra den nya *Pth* för spelaren och motståndaren vid varje testning av placering.

Placera vägg

Fungerar i princip exakt likadant som att förutsäga motståndaren fast omvänt. Vi försöker förhindra motståndaren genom att testa placera en vägg genom *Pth*. Om *Pth* blir mycket längre för motståndaren än för spelaren så är det en lämplig position att placera en vägg på. Baseras på samma poängsystem där istället det bestäms genom längden av motståndarens S. Det bästa draget bestämmer var väggen skall placeras. Har samma tid- och utrymmeskomplexitet som förra.