Dynamisch programmeren voor het bepalen van het optimaal pad

Ferre Van der Vieren - 0851455

27 mei 2021

Prof. Dutré

1 Inleiding

Dynamisch programmeren is een programmeermethode waarbij men een complex probleem recursief zal onderverdelen in zogenaamde 'subproblemen' die eenvoudiger op te lossen zijn. Hierbij zal men - om de ongunstige tijdscomplexiteit van opgestapelde recursie-oproepen te vermijden - een oplossing 'bottom-up' opbouwen, waarbij men de verkregen tussenresultaten steeds opslaat om later eventueel te hergebruiken. Hierdoor wordt de negatieve invloed van overlappende oproepen bij gewone recursie vermeden.

In dit verslag zullen we het principe van dynamisch programmeren toepassen om een grid-pathfinder efficiënt te implementeren. We zullen eerst een naïeve recursieve methodiek bespreken. Vervolgens wordt dynamisch programmeren toegepast op hetzelfde probleem om tot een efficiënt algoritme te komen. Tot slot bekijken we een complexe uitbreiding van het gestelde probleem waarbij we opnieuw dynamisch programmeren zullen gebruiken om zo een dieper begrip te verkrijgen van de programmeermethodiek.

Dit verslag werd geschreven voor het opleidingsonderdeel Gegevensstructuren & Algoritmen aan de KU Leuven, en is gedeeltelijk gebaseerd op het tekstboek *Algorithms (4th edition)* van Robert Sedgewick en Kevin Wayne. Een elementaire kennis van gegevensstructuren en algoritmen wordt verondersteld. Alle code werd geschreven in Java en is beschikbaar gesteld aan de lezer.

2 Het probleem

2.1 Beschrijving

We beschouwen het volgend algemene probleem dat - mits aanpassing van enkele details - erg relevant is bij de toepassing van kortstepad-algoritmen. Gegeven een nxm bord, wat is het optimale pad voor een hardloper die begint op startpositie P in $0x0$ en moet eindigen in positie G in $n-1xm-1$. De loper begint met een bepaald energieniveau, en kan ofwel naar rechts of naar onder bewegen. Op het bord zijn er verschillende soorten tegels waarlangs de loper kan passeren, elk met een bepaalde invloed op het huidige energieniveau van de loper. Een gewone tegel (notatie: .) kost één energiepunt, een modderpoel (notatie: M) twee punten, een heuvel (notatie: H) drie punten, en bij een energietegel (notatie E) krijgt de loper netto één energiepunt. Er bestaan ook muren (notatie: X) waardoor de loper niet kan passeren. Het doel is nu om een geldig pad te vinden zodanig dat de loper zo veel mogelijk energie overhoudt bij het bereiken van de eindpositie. Een oplossing is geldig indien de energie van de loper in elke tegel (inclusief de finish) strikt groter is dan nul. Een oplossing wordt gegeven als een opeenvolging van 'RIGHT' of 'DOWN' instructies. Beschouw ter illustratie onderstaand voorbeeld:

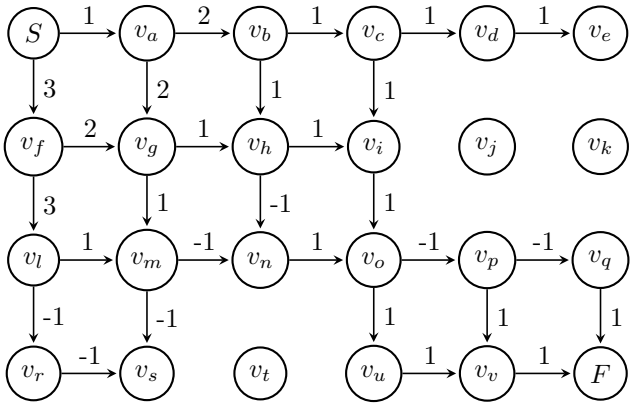
·	·	M	·	·	·
H	M	·	·	X	X
H	·	E	·	E	E
E	E	X	·	·	·

waarbij voor een energieniveau van 5 de sequentie [RIGHT, RIGHT, DOWN, DOWN, RIGHT, RIGHT, RIGHT, DOWN] een geldige oplossing zou zijn. De loper komt hierbij aan met een energie-niveau van 2. We definiëren hiervoor de hoofdfuncties $isSolvable()$, $getFinishTileEnergy()$ en $getPathSequence()$ die we in het verdere verloop van het verslag zullen bespreken.

2.2 Abstractie

Voordat we oplossingsmethoden kunnen bespreken zullen we eerst het problemen moeten abstraheren. We kunnen dit met behulp van een gewogen, gerichte graaf. Hierbij stellen de knopen posities voor en is de weging van een boog gelijk aan de kost in energieniveau om van de vertrekknoop naar de aankomstknoop te lopen. Muren kunnen we implementeren door geen bogen te laten aankomen en vertrekken in de knoop die de muur voorstelt. De graaf zal dus niet enkelvoudig zijn en uiteenvallen in verschillende componenten; we verkrijgen echter wel steeds één algemene hoofdcomponent (de andere componenten zijn namelijk gewoon alleenstaande knopen die de muren voorstellen) die enkel van belang is.

Een energiepunt stellen we voor door een boog naar dat punt met een weging van -1. Bovenstaand voorbeeld kunnen we dan als volgt voorstellen:



We kunnen nu eenvoudig een oplossing definiëren: we zoeken namelijk het optimale pad (met de laagste eindenergie) van knoop S naar knoop F. Het energieniveau waarmee we eindigen berekenen we dan simpelweg door de kost van het optimale pad af te trekken van het energieniveau waarmee de loper gestart is. Merk op dat we het optimale pad zoeken, en niet het kortste pad; elk pad is namelijk automatisch ook het kortste pad.

Merk op dat de abstractie van het probleem naar grafen niet noodzakelijk is; we kunnen ook de knopen voorstellen als elementen in een grote matrix, en een tweede matrix die de kost bijhoudt om van een knoop v_i naar een verbonden knoop v_{i+1} te lopen. Het gebruik van grafen acht ik echter duidelijk voor de bespreking/implementatie van het probleem: knopen stellen eenduidig posities voor en de bogen de kost (= verandering in energie van de loper) om van de ene positie naar de andere te lopen. In feite is men hier vrij in; hoe men een probleem wil abstraheren voor de bespreking ervan is uiteraard niet bindend voor de analyse ervan.

3 Oplossingsmethoden

3.1 Naïeve recursie

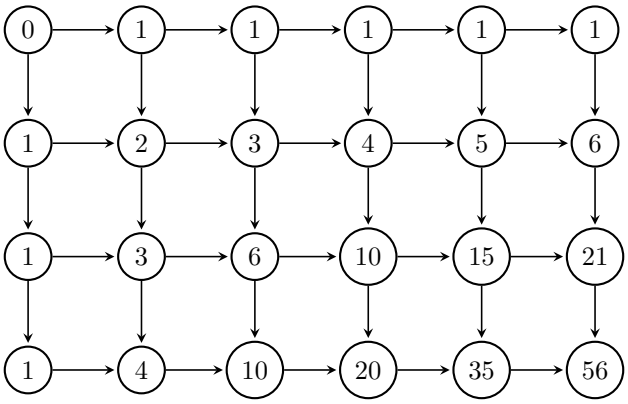
3.1.1 Beschrijving

We kunnen het probleem oplossen door - beginnend bij de startpositie - vanuit een tegel steeds recursief te kijken welke paden er mogelijk zijn vanuit de naburige tegels. Men verkrijgt dus het patroon $kostPositie + recursief(rechterVak) + recursief(onderVak)$. Men zou dit bijvoorbeeld kunnen implementeren met behulp van diepte-eerst zoeken (DFS); we beginnen bij de wortel en kiezen een boog en doorzoeken deze zo ver mogelijk voordat we terugkomen op de eerder

gemaakte stappen. Indien het gevolgde pad de eindpositie (knoop F) bereikt kunnen we het pad opslaan in een datastructuur, en dan terugkeren op de eerder gemaakte stappen om zo (eventueel) een ander mogelijk pad te vinden. Indien we alle mogelijkheden doorlopen hebben, overlopen we de kost voor elk pad en kiezen hetgeen met de laagste kost. Dit geeft uiteraard een ongunstige tijds- en ruimtecomplexiteit; in feite 'bruteforcen' we een optimale oplossing door alle mogelijke paden recursief te zoeken. Zo ontstaan er veel overlappende recursieoproepen; we berekenen dus meerdere keren dezelfde tussenoplossingen. Met een tussenoplossing bedoelen we hier het optimale pad en benodigde energieniveau naar een gegeven knoop die niet de eindpositie is.

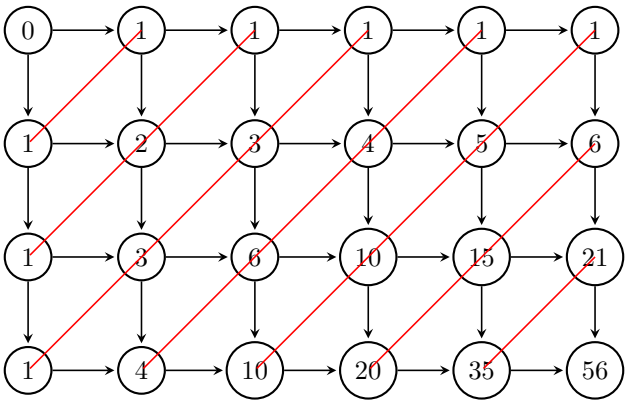
3.1.2 Complexiteit

Voor het bepalen van de complexiteit van dergelijke 'naïeve recursie' is het van belang om eerst het totaal aantal mogelijke verschillende paden te bepalen, aangezien we die recursief zullen berekenen. Het aantal verschillende paden tot een bepaalde knoop (positie) is gelijk aan de som van het aantal verschillende paden naar de twee knopen met invallende bogen op die knoop. We kunnen hiermee het totaal aantal paden voor een willekeurig bord bepalen. Neem als voorbeeld een gewoon $4x6$ bord, met de waarde van een knoop v het totaal aantal mogelijke paden tot die knoop:



Voor een $4x6$ bord vinden we dus 56 mogelijke paden voor de loper[1]. We merken hierin een patroon op de diagonalen, namelijk de driehoek van Pascal. Voor een NxN matrix is dit perfect symmetrisch, maar voor een NxM matrix geldt dit principe nog steeds ook al zal niet elke rij van de driehoek van Pascal integraal aanwezig zijn. Beschouw onderstaande figuur die dergelijk patroon aantoont:

[1]Een muur is nog steeds een knoop met ofwel geen boog ernaar toe, ofwel met een boog van gewicht oneindig. Beide opties geven uiteindelijk hetzelfde resultaat, en men zou dus bij deze recursieve methode muren kunnen beschouwen als knopen met bogen van een oneindig gewicht ernaar toe. Dergelijke paden moeten dus wel nog steeds berekend moet worden.



Merk op dat de eerste drie diagonalen volledige rijen voorstellen in de driehoek van Pascal. Daarna stopt de diagonaal vroegtijdig, maar men kan dit denkbeeldig doortrekken zodat het principe blijft gelden. Het nut van dit patroon is dat we nu in staat zijn het totaal aantal mogelijke paden van de loper te bepalen enkel op basis van de dimensies van het bord. Volgens de driehoek van Pascal is het element $T(x, y)$ in rij x en positie y gelijk aan

$$T(x, y) = \binom{x}{y} = \frac{x!}{y!(x-y)!} \tag{1}$$

zodat we nu het totaal aantal paden kunnen voorstellen door de uitdrukking

$$\binom{N+M-2}{N-1} = \binom{N+M-2}{M-1} \tag{2}$$

Deze gelijkheid geldt wegens de symmetrie die optreedt. Verder uitgewerkt krijgen we dan de uitdrukking

$$\frac{(N+M-2)!}{(M-1)!(N+M-2-(M-1))!} = \frac{(N+M-2)!}{(M-1)!(N-1)!} \tag{3}$$

voor het totaal aantal mogelijke paden in functie van de dimensies van het NxM bord. Nu resteert er zich enkel nog het bepalen van het aantal bewerkingen dat de "naïeve recursie methode"gebruikt voor de berekening van elk pad. Dit is echter eenduidig; het aantal bewerkingen is lineair ten opzichte van het lengte van het pad. Aangezien elk pad in de graaf van lengte $N + M - 2$ is, verkrijgen we een algemene tijdscomplexiteit van

$$(N+M-2)\frac{(N+M-2)!}{(M-1)!(N-1)!} \tag{4}$$

Nu kunnen we ook kort het geheugengebruik bespreken. Aangezien we steeds de matrix $tiles[][]$ moeten bijhouden, hebben we minstens al een complexiteit van $\sim NM$. Bovendien kunnen we stellen dat we worst-case niet meer extra geheugen moeten bijhouden als het voorlopig optimale pad, alsook het pad dat

we op dit moment recursief opstellen. Aangezien een (voorlopig) pad nooit langer is dan $N + M - 2$ zal dit geen invloed hebben op de tilde-notatie voor de geheugencomplexiteit. Het geheugengebruik voor deze methode is bijgevolg dus $\sim NM$.

Merk op dat we voor de recursieve methode een zeer hoge tijdscomplexiteit vinden en een relatief laag geheugengebruik; we zullen later zien dat voor dynamisch programmeren we wel degelijk meer geheugen nodig zullen hebben, maar dat dit meer dan verkiesbaar is vanwege de enorme verbetering in tijdscomplexiteit. Dit is dan ook een algemeen kenmerk van dynamisch programmeren: men kan met wat extra geheugengebruik de tijdscomplexiteit enorm opdrijven.

3.2 Dynamisch programmeren

In deze sectie beschrijven we een oplossingsmethode dat steunt op het principe van dynamisch programmeren. Ik heb hier gekozen om verder te werken met grafen, en het probleem te herleiden tot het vinden van het optimale pad in een gewogen graaf met eventuele negatieve gewichten. Merk op dat dit niet noodzakelijk de meeste efficiënte aanpak is voor dit specifiek probleem. We zouden ook gebruik kunnen maken van een tabel en deze systematisch opvullen steunend op bepaalde specifieke eigenschappen van het probleem, waaronder bijvoorbeeld de aanname dat de loper enkel naar rechts en onder kan bewegen (zie later). Op die manier hoeven we niet een graafstructuur te initialiseren op basis van de inputmatrix, en verkrijgen we dus een licht efficiënter algoritme. Deze methode wordt dan ook kort besproken op het einde van dit hoofdstuk.

Ik heb echter twee redenen waarom ik deze invalshoek niet uitvoerig besproken en geïmplementeerd heb. Ten eerste steunt dergelijke oplossingsmethode erg op de aard van dit probleem, zoals hierboven reeds gesteld. Mocht de lezer het probleem willen uitbreiden met bijvoorbeeld de mogelijkheid om een omweg te maken ('LEFT' of 'UP') om zo te kunnen eindigen met een hoger energieniveau, dan zou dergelijke oplossingsmethode al tekort schieten (zie later), terwijl de gekozen oplossingsmethode die we zullen bespreken hier - mits een kleine uitbreiding - compleet voor geschikt is. Men zou zelfs eenvoudig extra functionaliteiten kunnen toevoegen aangezien het probleem geïmplementeerd is als graaf, wat uiteraard talloze applicaties kent. Men zou bijvoorbeeld eenvoudig het optimale pad kunnen berekenen voor een loper die, gegeven een startpositie, alle posities moet aflopen met minimale energie (namelijk het vinden van een minimaal opspannende boom van de graaf). Ten tweede zou ik willen beargumenteren dat deze methode een dieper inzicht kan geven in dynamisch programmeren. Het gebruik van een simpele matrix om tussenoplossingen op te slaan (en de overige waarden in de matrix berekenen op basis van deze tussenoplossingen) is namelijk een welbekend schoolvoorbeeld, terwijl het principe van dynamisch programmeren ook te zien toegepast worden op een complexer graafprobleem dat talloos voorkomt in de informatica (optimale path-finding algoritmes) naar mijn mening toch nog een interessantere invalshoek is.

Hoewel het dus uiteindelijk tot een iets slechtere tijdscomplexiteit zal leiden, heb ik toch gekozen om dergelijke implementatie uitgebreid te bespreken op basis van bovenstaande voordelen. Voor de volledigheid zal ik - zoals gesteld - op het einde van het hoofdstuk wel de klassieke oplossingsmethode op basis van dynamisch programmeren kort bespreken.

3.2.1 Beschrijving

Zoals gesteld zullen we het probleem nu oplossen met behulp van dynamisch programmeren. We hanteren nu een efficiënte bottom-up werking; tussenoplossingen voor subproblemen worden tijdelijk opgeslagen, en hierop wordt verder opgebouwd tot we een optimale oplossing bekomen (indien die bestaat). Dit steunt op de volgende eigenschap: indien $w(v_1)$ en $w(v_2)$ optimaal zijn, dan geldt voor de knoop $v+1$ die bereikt kan worden uit ofwel v_1, ofwel v_2 dat

$$w(v+1) = min\{w(v_1) + w(e), w(v_2 + w(e)\} \tag{5}$$

met w(v) de benodigde energie om naar de knoop (v) te lopen. Dit geldt specifiek voor deze graaf aangezien elke knoop hoogstens twee invallende bogen heeft ('RIGHT' of 'DOWN'). Dit is een recursiebetrekking met als triviale gevallen de twee (of één indien een muur) knopen die direct verbonden zijn door een boog met de beginknoop. Hiervoor geldt logischerwijs $w(v_{s_1}) = w(e_1)$ en $w(v_{s_2}) = w(e_2)$. Zo kunnen we dus elke knoop uitdrukken in termen van de knopen ervoor; we bouwen steeds verder op tussenoplossingen totdat we de eindknoop bereikt hebben. Uit (1) volgt dan dat deze oplossing direct ook de optimale oplossing is (minste energieverbruik).

We maken hier nog een belangrijke opmerking bij: de vergelijking (1) drukt enkel de benodigde energie uit om van de startpositie naar een bepaalde knoop te lopen. Vanwege de aard van het probleem moeten we nog met twee andere zaken rekening houden. Ten eerste willen we niet enkel de minst benodigde energie maar ook het gevolgde pad dat deze oplossing geeft. Dit kan men echter eenvoudig implementeren. Een typische 'dynamische programmatie' wijze zou zijn dat we voor elke knoop een array bijhouden met de sequentie van zetten voor de loper om die knoop optimaal te bereiken. Indien we dan uit (1) weten uit welke vorige knoop we het beste de desbetreffende knoop bereiken kunnen we het pad van die vorige knoop dan gelijkstellen aan het pad van de huidige knoop met de bijhorende instructie er achteraan aan toegevoegd ('RIGHT' of 'DOWN'). Dit zorgt echter voor veel extra geheugengebruik: we zijn enkel geïnteresseerd in het (eventueel) pad naar de eindknoop, en dit kunnen we dan ook achterwaarts construeren als we alle tussentijdse energieniveaus hebben. Indien we in knoop v zijn met $D[v]$ de afstand van de start, ga dan naar de buurknoop u zodat $D[v] = D[u] + w(u, v)$. We moeten dus enkel de energieniveaus van de subproblemen hebben om het pad hieruit te kunnen construeren. Ten tweede moeten we garanderen dat het energieniveau van de loper op elk moment positief blijft.

We zouden namelijk een optimale oplossing kunnen vinden die eigenlijk geen geldige oplossing is vanwege een energieniveau van nul of lager in één van de tussenposities. Dit is ook opnieuw een kwestie van implementatie. Indien we het pad beschouwen naar een knoop v_i vanuit een knoop v_{i-1}, en $w(v_i) \leq 0$ op dat pad, dan beschouwen we dat pad als ongeldig en updaten we de waarde van de knoop v_i niet.

3.2.2 Algoritme

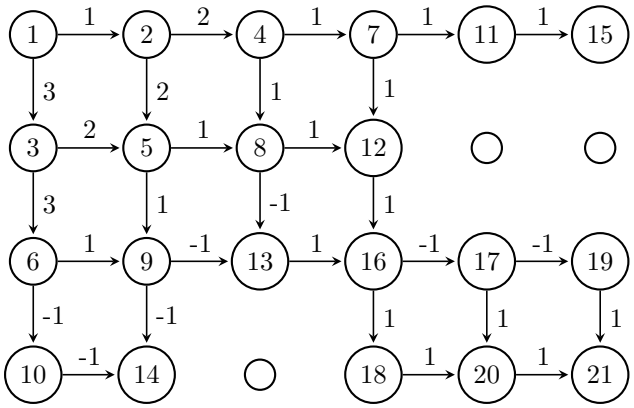
Voor het bepalen van het optimale pad kunnen we echter niet het efficiënte algoritme van Dijkstra gebruiken: het is namelijk 'greedy', waardoor het faalt bij grafen met bogen met een negatieve waarde. Volgens (1) kan men het optimale pad van een knoop v bepalen indien we het optimale pad kennen van alle knopen met een vertrekkende boog die invalt op v. De vraag is nu echter; hoeveel moeten we dit recursief herhalen? Voor de soort grafen van dit problemen weten we dat er geen negatieve kringen bestaan. Uit de grafentheorie weten we dat een kringvrij pad hoogstens $V - 1$ bogen bevat, dus het kortste pad kan nooit meer dan $V - 1$ bogen bevatten (want anders bevat het pad een kring, en aangezien de graaf nooit een negatieve kring kan bevatten is deze kring nadelig en kan dus verwijderd worden om een optimaler pad te vinden). Men verkrijgt zo de volgende pseudo-code:

Algorithm 1: Standaard Bellman-Ford implementatie

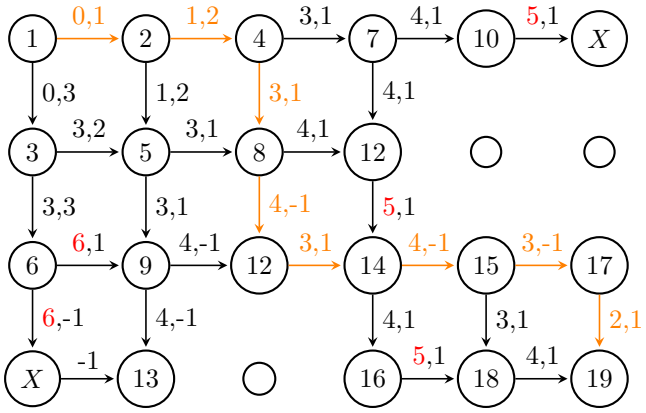
```
D[S] = 0
for elke knoop v ≠ S do
|  D[v] = ∞
end
for elke knoop i = 1 tot |V| − 1 do
|  for elke boog (v1, v2) ∈ E do
|  |  Relax(v1, v2);
|  end
end
Function Relax(v1, v2):
|  for elke boog (v1, v2) ∈ E do
|  |  if D[v2] > D[v1] + W[v1, v2] then
|  |  |  D[v2] = D[v1] + W[v1, v2]
|  |  end
|  end
```

We kunnen deze standaardvorm nog optimaliseren en aanpassen aan het gegeven probleem, maar het principe blijft hetzelfde. We maken de observatie dat alle bogen in de graaf steeds opnieuw te 'relaxeren' vaak tot geen verandering leidt. Meer specifiek kan men stellen dat de enigste bogen die voor een verandering in de waarde van het optimale pad naar die knoop kunnen zorgen de bogen zijn waarbij de waarde van de beginknoop aangepast is in een eerdere

iteratie. Beschouw opnieuw het voorbeeld ter illustratie: we zullen nooit de waarde van de eindknoop F kunnen aanpassen indien de waarde van het optimale pad tot de knopen v_v en v_q nog steeds oneindig is (door de initialisatie). Voor de implementatie in de code gebruiken we hiervoor een queue, die initieel enkel de startknoop bevat, waar we steeds de knopen aan toevoegen waarnaar verwezen wordt door een boog die succesvol gerelaxeerd is. Dit gaat ten koste van wat extra geheugen, maar zal resulteren in een veel betere uitvoeringstijd. We vermijden namelijk naïeve oproepen van $Relax()$ waarbij we op voorhand weten dat de if-statement niet zal gelden. Ter verduidelijking beschouwen we de werking van het algoritme op het eerder besproken voorbeeld. Indien we steeds eerst de rechterknoop in plaats van de knoop onderaan beschouwen en een ongelimiteerde energie veronderstellen, dan geeft volgende figuur aan hoeveel en in welke volgorde de $Relax()$ functie opgeroepen wordt (het nummer in elke knoop stelt de chronologische volgorde voor):



In totaal wordt de $Relax()$ functie dus 21 keer opgeroepen voor dit specifiek probleem. Laten we nu de vereenvoudigingen wegwerken; we houden in elke knoop de benodigde energie om tot die knoop te geraken bij, en voeren in de $Relax()$ functie een if-statement in waarbij we nakijken of deze waarde strikt kleiner is dan de beginenergie. Indien de waarde groter is dan de beginenergie dan voert de functie niks uit. De bogen vertrekkende uit die knoop zijn dus 'nutteloos' en worden niet gebruikt bij berekeningen. Op die manier bepalen we of de graaf oplosbaar is voor een bepaalde startenergie en ook de (eventuele) eindenergie en gevolgde pad. We verkrijgen dan de volgende figuur, met het paar (x, y) bij een boog als x die benodigde energie van de knoop waaruit de boog vertrekt, en y de kost van dat pad te nemen:



Enkele knopen krijgen dus geen waarde meer (aangeduid met X), aangezien die niet bereikbaar zijn door een geldige oplossing. We vinden met deze implementatie dus een optimale oplossing met als eindenergie $5 - min\{2+1, 4+1\} = 2$. Het gevolgde pad is weergegeven in het oranje op de figuur.

3.2.3 Complexiteit

Voor het bepalen van de tijds- en geheugencomplexiteit te bepalen moeten we onze specifieke implementatie bespreken. Allereerst zullen we van de gegeven NxM matrix een graaf maken; we initialiseren eerst de gelinkte lijst $LinkedList < Graph.Edge > []adjacencylist$, en de array $Node[]nodes$, wat $2|V| = 2NM$ bewerkingen vraagt met N en M de dimensie van de matrix. Daarna zullen we de bogen tussen de knopen toevoegen in $adjacencylist$. Dit vraagt $2(N+M-2)+N+M = 3(N+M)-4$ array accesses (op $String[]tiles$), aangezien we voor elke rij en kolom behalve de onderste en meest rechtse twee mogelijke bogen hebben. Indien we geen representation exposure willen moeten we ook nog eens de 2D-array diep kopiëren, wat voor een extra NM array accesses zorgt. Voor onze implementatie doet dit er echter niet toe. De initialisatie - het opzetten van de juiste datastructuren voor het algoritme op basis van de $String[][]tiles$ array - heeft dus een complexiteit van $\sim 2NM$.

Nu zullen we het algoritme oproepen (klasse $BellmanFord.java$). Hierbij initialiseren we de afstand nodig om tot een knoop te geraken op oneindig (behalve de startknoop, die start met waarde nul) in de $distTo[]$ array, wat resulteert in een complexiteit van $\sim|V| = NM$. Nu zetten we de startknoop op de eerder besproken queue, en zullen we opeenvolgend met een while-lus de eerste knoop van de queue ophalen en die 'relaxeren'. We passen deze $Relax()$ functie efficiënt aan zodat het niet meer alle bogen in de graaf relaxeert (zoals in de pseudocode), maar enkel de bogen verbonden met die knoop. Bovendien zullen we de aankomstknoop van een succesvol gerelaxeerde boog aan de queue toevoegen; dit zorgt voor een efficiëntere complexiteit zoals eerder besproken. Zo verkrijgen

we de volgende Java implementatie[2]:

```
private void relax(Graph G, int v){
  for (Graph.Edge e : G.adjacencylist[v]){
    int w = e.destination;
    if (distTo[w] > distTo[v] + e.weight &&
        G.nodes[v].costFromStart < startingEnergy){
      distTo[w] = distTo[v] + e.weight;
      edgeTo[w] = e;
      G.nodes[w].costFromStart =
          G.nodes[v].costFromStart + e.weight;
      if (!onQueue[w]){
        queue.add(w);
         onQueue[w] = true;
      }
    }
  }
}
```

Indien we enkel array accesses tellen en de resterende bewerkingen als verwaarloosbaar beschouwen, dan hebben we in totaal zes array accesses per succesvol gerelaxeerde boog; twee waardes in $distTo[]$ worden opgevraagd/aangepast, twee waardes in $G.nodes[]$, één waarde in $edgeTo[]$ en $onQueue[]$. Voor een onsuccesvolle relaxatie is dit de helft; $distTo[v]$, $distTo[w]$ en $G.nodes[v]$. Worst-case hebben we dus zes array accesses per boog, en elke knoop v heeft maximaal twee bogen in $adjacencylist[v]$ ('RIGHT' en 'DOWN'). Aangezien men nooit meer keren de functie $Relax()$ oproept dan het totaal aantal knopen in de graaf wordt deze hoogstens $|V|$ keer opgeroepen. We krijgen dus een worst-case complexiteit van $\sim 12|V| = \sim 12NM$ met $|V|$ het aantal knopen in de graaf en N en M de dimensie van de NxM matrix.

Eens het algoritme volledig klaar is moeten we echter nog het gevolgde pad bepalen; dit doen we, zoals eerder vermeld, van achter naar voren op basis van ge gevonden tussenoplossingen voor het benodigd energieniveau. Deze bewerking is lineair. Aangezien de lengte van elk pad van de beginknoop naar de eindknoop voor dit type van grafen gelijk is aan $N + M - 2$, verkrijgen we een complexiteit van $\sim N + M$ voor de bepaling van de sequentie van zetten voor het optimale pad. Hier merken we waarom we het pad pas later van achter naar voor bepalen in plaats van het bijhouden van een array van het beste pad naar elke knoop; de complexiteit is nu maar een lage orde term in vergelijking met de complexiteit van $Relax()$, en de constante factor voor de complexiteit van die functie zou hoger zijn indien we bij elke succesvolle relaxatie nog eens een array moet kopiëren en aanvullen. Het volledige probleem is nu opgelost; zowel de eindenergie, als de oplosbaarheid en sequentie van een eventuele oplossing is

[2]Zie de appendix voor de interface van de klasses voor meer duidelijkheid over de verschillende variabelen.

berekend.

We hebben voor onze specifieke implementatie van het algoritme een totale worst-case tijdscomplexiteit van $\sim 13NM$ gevonden: $\sim NM$ voor de initialisatie en $\sim 12NM$ voor het relaxeren (de complexiteit voor het bepalen van het optimale pad na de afloop is slechts een lage orde term en valt dus weg). Samen met de creatie van de benodigde graaf (complexiteit $\sim 2NM$) resulteert dit in een totale complexiteit van $\sim 15NM$.We hebben hier een aantal bewerkingen als verwaarloosbaar gerekend (zie bv. de $Relax()$ functie), dus de exacte constante factor kan verschillen afhankelijk van implementatie/berekening. Bovendien zal de gemiddelde complexiteit gunstiger zijn; niet elke boog zal succesvol gerelaxeerd worden, en niet elke knoop heeft twee uitgaande bogen (bv. de knopen op de laatste rij of een muur). We merken wel op dat de aard van het probleem ervoor zorgt dat de complexiteit altijd evenredig zal zijn met de term NM (dimensie van de matrix, of ook wel het aantal knopen $|V|$). Zoals uitgebreid behandeld in deze sectie kunnen we enkele optimalisaties in de implementatie aanbrengen waardoor de constante factor daalt. Begrijpen hoe en waarom we bepaalde implementatiekeuzes maken is hierbij crucialer dan het exact kunnen bepalen van de constante factor in de tilde-notatie. We kunnen zeker en vast concluderen dat we de algemene complexiteit van het algoritme drastisch verbeterd hebben ten opzichte van de naïeve pseudo-code van het Bellman-Ford algoritme zoals eerder vermeld. Deze aanpak zou namelijk evenredig zijn met $|V||E|$, in plaats van enkel met $|V| = NM$ op een constante factor na zoals nu.

We kunnen nu ook de geheugencomplexiteit van onze implementatie bepalen; we houden de NxM matrix $tiles[][]$ bij, alsook een array van knopen van lengte NM, en als slot een gelinkte lijst - ook van lengte NM - met de bijhorende bogen van elke knoop. Aangezien elke knoop in de graafabstractie van ons probleem maximaal twee uitgaande bogen kan hebben, verkrijgen we een maximaal benodigd geheugen van $\sim 2NM$ voor $adjacencylist[]$. Bovendien zijn er nog twee arrays van lengte NM gedefinieerd voor het Bellman-Ford algoritme; één array die de afstand tot elke knoop bijhoudt ($distTo[]$) en één die bijhoudt of een knoop al dan niet op de queue geplaatst is ($onQueue[]$). Ten slotte gebruiken we nog een queue om het algoritme te implementeren, die per definitie nooit groter kan worden dan het aantal posities (namelijk NM). In totaal is de geheugencomplexiteit dus evenredig met NM, en meer specifiek gelijk aan $\sim 6NM$.

3.2.4 Alternatieve oplossing

Nu we uitgebreid de oplossing van het probleem geïnterpreteerd als graafprobleem hebben besproken, zullen we nog even kort - zoals eerder gesteld - de klassiekere oplossingsmethode bespreken. We maken hierbij een NxM matrix A aan met de waarde van $A[i][j]$ gelijk aan het benodigde energieniveau van het optimale pad tot positie (i, j) (definitie van het subprobleem, redelijk analoog aan onze andere oplossingsmethode). In plaats van de overgangen tussen

posities te coderen als bogen van een graaf, zullen we nu de waarde van $A[i][j]$ definiëren in termen van de andere posities in de matrix en de gegeven input matrix $tiles[][]$, namelijk:

$$A[i][j] = min\{A[i-1][j] + value(tiles[i,j]), A[i][j-1] + value(tiles[i,j])\} \quad (6)$$

We komen tot deze vergelijking analoog als aan de uitdrukking (5). Aangezien de loper enkel naar rechts en onder kan bewegen, zal de eerste rij en de eerste kolom van de matrix A eenvoudig opgevuld kunnen worden (want er is maar één mogelijk pad tot dergelijke posities, zie de bespreking van de naïeve recursie). Nu kunnen we de matrix in row-major order beginnen opvullen: de waarden in de tweede rij kunnen van links naar rechts bepaald worden op basis van de eerste rij (en eerste kolom), de waarden in de derde rij op basis van de tweede rij, enz. Hier ziet men duidelijk het principe van dynamisch programmeren in: we slaan tussenoplossingen op en bouwen hierop steeds verder. Zo bereiken we uiteindelijk de eindpositie en dus de eindenergie, en kunnen we - analoog als bij de graafimplementatie - van achter naar voren terugwerken om het gevolgde pad te bepalen. We zullen ook opnieuw bij elke iteratie checken of de huidige energie van de loper wel boven de nul blijft.

Deze implementatie is duidelijk efficiënter; we gebruiken minder geheugen (de matrix $tiles[][]$ en één enkele extra matrix A, dus complexiteit $\sim 2NM$), en we voeren minder bewerkingen uit (de tijdscomplexiteit is nog steeds evenredig met NM, maar de extra bewerkingen voor het omspringen met de datastructuren voor de graafimplementatie vallen weg). De tijds- en geheugencomplexiteit is dus nog steeds evenredig met de factor NM, maar nu wel met een lagere constante factor.

3.3 Uitbreiding van het probleem

Beschouw nu volgende uitbreiding van het probleem; er is een tweede loper P2 die van linksonder naar rechtsboven beweegt en dus enkel maar 'RIGHT' of 'UP' kan lopen. Voor de eerste loper P1 verandert niks. We willen nu dat beide lopers een baton doorgeven aan elkaar. Concreet betekent dit dat ze elkaars afgelegd pad exact één keer kruisen; hun afgelegde wegen moeten dus in één, en hoogstens één tegel kruisen. We zijn opnieuw geïnteresseerd in de optimale paden van P1 en P2 zodat de som van de eindenergieën van beide loopers maximaal is. Eerdere regels, zoals de voorwaarde dat elke loper steeds een positief energieniveau heeft, blijven gelden.

De code en redenering vanuit de vorige sectie gelden hier niet langer meer. We kunnen namelijk niet het optimale pad voor P1 en P2 afzonderlijk berekenen, want dit kan mogelijks een ander kruispunt opleveren. Ook werkt het bepalen van het optimale pad voor een bepaalde speler om dan het pad van de andere speler hierop aan te passen niet. Er kan namelijk een globaal optimalere oplossing zijn die afzonderlijk niet optimaal is voor P1 en P2, maar wel optimaal in

termen van de som van de eindenergieën. We moeten dus een andere manier vinden om te redeneren over het probleem.

Een belangrijk concept in deze uitbreiding is het kruisen van de twee lopers. Het probleem is namelijk analoog aan de vorige sectie totdat we de kruising in rekening moeten brengen. Dit kunnen we uitbuiten door het probleem te ontbinden in het vinden van het optimale pad van P1 en P2 van hun respectievelijke startposities tot het kruispunt en dan van het kruispunt tot de respectievelijke eindposities:

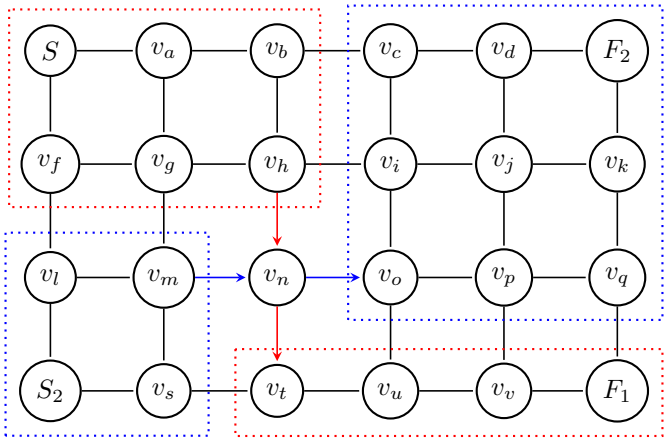
$$P(start \rightarrow finish) = P(start \rightarrow kruispunt) + P(kruispunt \rightarrow finish) \quad (7)$$

Het vinden van het kruispunt voor een optimale uitkomst is hier de uitdaging. We kunnen alvast een belangrijkste vaststelling maken: het kruispunt zal nooit gelegen zijn op één van de randen van het bord of naast een muur. Dit is omdat de vier tegels rond het kruispunt steeds geldige posities moeten zijn (dus niet gelegen uit het bord of een muur) aangezien de lopers anders gegarandeerd meerdere tegels gemeenschappelijk hebben (ofwel kruist men beide horizontaal, ofwel beide verticaal). De vier buurtposities van een kruispunt moeten dus steeds geldige posities zijn; zo kan de ene speler het punt horizontaal kruisen, en de andere speler verticaal. Dit legt ook een beperking van de dimensie van het spelbord op; men zal voor kleine borden met veel muren vaak geen mogelijk geldig kruispunt vinden en dus geen oplossing.

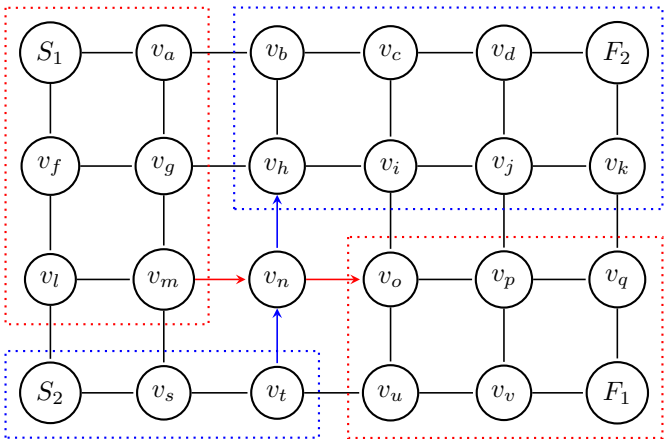
Men zou dus relatief eenvoudig alle mogelijke kruispunten kunnen bepalen voor een bepaald bord. De vraag luidt nu: hoe weten we welk kruispunt hiervan een optimale oplossing zal geven? En gegeven het optimale kruispunt, welke loper passeert er horizontaal door en welke verticaal? Deze vragen zijn niet eenvoudig te beantwoorden aangezien de optimaliteit van een kruispunt afhangt van de tegels van de subproblemen; indien er bijvoorbeeld veel energietegels onder een kruispunt liggen zal het optimaal zijn voor P2 om verticaal het kruispunt te kruisen. We zullen dus naar de omliggende tegels moeten kijken om kruispunten met elkaar te kunnen vergelijken.

We kunnen nu het principe uit (7) toepassen. De waarde van $P(start \rightarrow kruispunt)$ - de energiekost om van de startpositie naar het kruispunt te bewegen - is eenvoudig te berekenen met de implementatie vanuit de vorige sectie. We maken immers een bord voor beide spelers (met behulp van dynamisch programmeren), en kunnen hier eenvoudig de kost om naar een kruispunt te bewegen van aflezen. De tweede stap, namelijk het berekenen van $P(kruispunt \rightarrow finish)$, is moeilijker. We kunnen wel dit deel van het bord beschouwen als afzonderlijk bord, en hiervan dan de benodigde energie berekenen om tot de eindpositie te komen. Zo hebben we twee waardes voor elk van de lopers: de energiekost om zich te bewegen tot juist voor het kruispunt, en de kost om zich dan - eens door het kruispunt - te bewegen tot de eindpositie. We moeten deze berekening

tweemaal toepassen; in het eerste geval zal P1 verticaal door het kruispunt gaan en P2 horizontaal, en in het tweede geval omgekeerd. Onderstaande twee figuren[3] maken dit principe duidelijk. Neem opnieuw een willekeurig $4x6$ bord ter illustratie, en neem v_n al het kruispunt dat we moeten analyseren. We hebben nu voor elke speler respectievelijk twee 'subborden'; voor P1 is dit van de start voor P1 tot positie v_h, en daarna van v_t tot de finish van P1, voor P2 van de start van P2 tot positie v_m, en daarna van v_o tot de eindpositie voor P2. In de eerste figuur beschouwen we het geval als P1 (rood) verticaal door het kruispunt gaat, en P2 horizontaal (blauw):



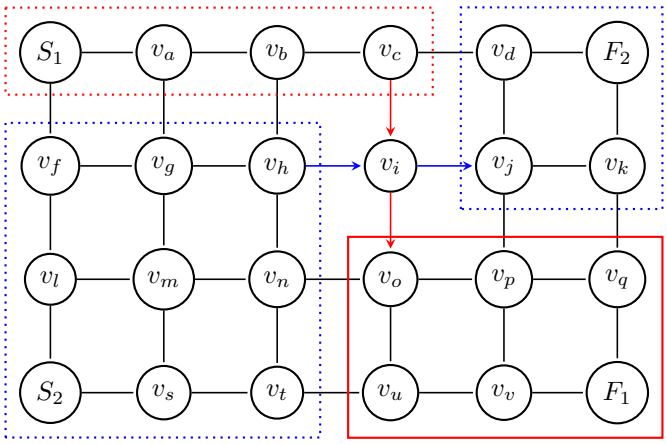
Beschouw nu voor hetzelfde kruispunt het tweede geval, namelijk als P1 horizontaal door het kruispunt gaat en P2 verticaal:



Zoals eerder gesteld kunnen de linkse twee subborden makkelijk opgelost worden, namelijk de waarde van de eindpositie in het subbord aflezen uit de volledige

[3]We maken terug gebruik van grafen voor een eenvoudigere en duidelijkere visualisatie.

NxM tabel opgesteld zoals in het vorig hoofdstuk van dit verslag behandeld. Voor de twee subborden die de situatie $P(kruispunt \rightarrow finish)$ beschrijven stellen we een compleet afzonderlijk bord op zoals gedaan in vorig hoofdstuk, maar nu voor een kleinere dimensie uiteraard, en berekenen dan de energiekost om van de start van het subbord naar de eindpositie te gaan (voor bovenstaande situatie is dat van positie v_o tot F_1 voor P1 en van positie v_h tot F_2 voor P2). We merken echter op dat we deze eindkost voor de subborden die $P(kruispunt \rightarrow finish)$ bepalen kunnen hergebruiken, want we zullen dikwijls dezelfde subborden bekomen voor andere kruispunten die we beschouwen. Ter illustratie geven we een voorbeeld; beschouw nu het mogelijks optimaal kruispunt v_i, gelegen rechtsboven het vorig beschouwde kruispunt v_n. Stel dat we het geval willen beschouwen dat P1 verticaal kruist en P2 horizontaal. Dan verkrijgen we de volgende situatie:



Merk op dat het tweede subbord voor P1 (omringd met een rode, volle lijn) volledig gelijk is aan het tweede subbord van P1 van vorige figuur (waarbij v_n het kruispunt was en P1 horizontaal kruiste). We kunnen hier dus dynamisch programmeren toepassen; in tegenstelling tot het steeds opnieuw moeten berekenen van de energiekost voor bepaalde subborden, zullen we deze 'tussenuitkomsten' bijhouden in een tabel. We creëren dus een extra tabel van integers voor zowel P1 als voor P2, waarin we voor elk subbord dat we berekenen voor die speler vanaf positie (i, j) tot de finish voor die speler de benodigde energie opslaan op positie $[i][j]$ in de tabel[4]. Voordat we een volledig subbord aanmaken beginnende van positie (i, j) voor P1 of P2 zullen we eerste elke keer kijken of de energiekost van dat subbord al eens eerder berekend is. Indien dit niet het geval is, maken we het subbord aan en berekenen zo de energiekost, en slagen dit op in de tabel.

[4]Merk hierbij op dat we ook de soort tegel van de beginpositie in acht moeten brengen; de loper beweegt namelijk van het kruispunt tot die tegel, dus de waarde van die tegel heeft ook invloed op de energiekost voor het optimale pad in het subbord

Eens we voor elk mogelijk kruispunt de energiekost hebben berekend voor de bijhorende subborden, kunnen we de kruispunten onderling beginnen vergelijken. Stel dat de matrices $beginValuesP1[][]$ en $beginValuesP2[][]$ de energiekost van de startpositie tot een bepaald punt bijhouden voor de respectievelijke spelers (cfr. vorig hoofdstuk), en de matrices $endValuesP1[][]$ en $endValuesP2[][]$ de energiekosten bijhouden voor de subborden die we bekomen na het kruisen van het kruispunt (werking hiervan is reeds in bovenstaande paragraaf uitgelegd). De totale optimale energiekost voor beide spelers voor een willekeurig kruispunt gelegen op positie (i, j) is dan gelijk aan

$$\begin{aligned} P_{total} = min\{&(beginValuesP1[i-1][j] + tiles[i][j] + endValulesP1[i+1, j])+ \\ &(beginValuesP2[i][j-1] + tiles[i][j] + endValulesP2[i, j+1]), \\ &(beginValuesP1[i][j-1] + tiles[i][j] + endValulesP1[i, j+1])+ \\ &(beginValuesP2[i-1][j] + tiles[i][j] + endValulesP2[i+1, j])\} \end{aligned} \tag{8}$$

Hoewel het een lange vergelijking is, is het principe vrij eenvoudig. We zullen voor elk kruispunt bepalen welke manier het beste is; ofwel gaat P1 verticaal door het kruispunt en P2 horizontaal, ofwel omgekeerd. Vandaar de min{} operatie. Indien P1 verticaal door het kruispunt gaat, berekenen we de kost van het pad van P1 door de kost om tot de positie boven het kruispunt te lopen startend vanaf de beginpositie op te tellen met de kost om door het kruispunt te gaan (tiles[i][j]) en de kost om na de kruising tot de eindpositie te lopen. Dit doen we analoog voor P2 (maar dan horizontaal). Deze beide tellen we op; dat is dan de totale kost voor beide spelers om op die manier (verticaal/horizontaal) door het kruispunt te gaan. Dan zullen we de andere manier beschouwen, en van deze twee waarden dus het minimum berekenen. Merk op dat we hierbij ook steeds nakijken of de energie van elke speler op elk moment steeds positief is. Het minimum van deze twee waarden beschouwen we dan als de 'optimale' kost die we verkrijgen indien we dat punt als kruispunt kiezen. We zullen dergelijke berekening dus doen voor alle mogelijke kruispunten. Hierbij houden we steeds bij wat het tot nu toe optimale kruispunt is, en hoe de lopers dat punt dan kruisen (een simpele boolean waarde zoals bijvoorbeeld $verticalP1$ volstaat). Elke keer we een kruispunt vinden waarbij de totale kost lager ligt (en de energie van elke loper steeds positief is), zullen we dit beschouwen als het nieuwe voorlopig beste kruispunt (en het kruispunt dat er eerder was verwijderen). Zo bekomt men uiteindelijk het optimale kruispunt, en welke loper er verticaal en welke horizontaal doorloopt. Eens we dit weten kunnen we niet alleen de eindenergieën voor P1 en P2 eenvoudig bepalen, maar ook de gevolgde paden construeren. We kennen namelijk de vier subborden die bij de kruising van dat punt op die manier horen, en voor elk (sub)bord kunnen we dan terug van achter naar voren het pad construeren, analoog aan de oplossing voor één speler uit het vorige hoofdstuk. Ook al is het duidelijk dat de tijdscomplexiteit hoger zal liggen dan bij de oplossing voor één speler, toch kunnen we stellen dat we - vanwege de afwezigheid van enige diep geneste lussen of dergelijke - toch nog steeds een acceptabele uitvoeringstijd krijgen voor relatief grote bor-

den. Bovendien zal de uitvoeringstijd erg afhankelijk zijn van het soort bord; indien we bijvoorbeeld een bord hebben met erg veel muren in het midden zal het aantal mogelijke kruispunten drastisch dalen, en de tijdscomplexiteit dus gunstig beïnvloeden. Zowel de dimensie van het bord, als het aantal muren en de plaatsing ervan hebben dus invloed op de tijdscomplexiteit.

4 Conclusie

In dit verslag hebben we het principe van dynamisch programmeren toegepast op het vinden van het optimale pad voor een loper, met als uitbreiding een gelijkaardig probleem maar dan voor twee lopers. Eerst hebben we vastgesteld dat het oplossen van het probleem door middel van een 'naïeve' recursieve aanpak erg inefficiënt is vanwege overlappende recursie-oproepen die de tijdscomplexiteit onacceptabel verhogen. Als oplossing hiervoor hebben we dynamisch programmeren gebruikt; we verdelen het grote probleem in subproblemen en slaan de tussenoplossingen voor deze kleinere problemen op om zo bottom-up naar een algemene oplossing toe te werken. Dit principe kan men toepassen op het invullen van een tabel in row-major volgorde waarbij de waarde van een positie in de tabel uitgedrukt wordt in termen van de vorige posities, maar kan - zoals we bemerkt hebben - evengoed gebruikt worden voor graafalgoritmen of complexere problemen zoals ook geïllustreerd bij de bespreking van de uitbreiding van het probleem. De efficiëntie van dergelijke aanpak is dan ook duidelijk gebleken; bij een recursieve aanpak voor het basisprobleem verkregen we faculteiten in de tijdscomplexiteit, terwijl door dynamisch programmeren we een oplossing gevonden hebben waarvan zowel de tijdscomplexiteit als het geheugengebruik evenredig zijn met NM, het product van de dimensies van het bord.