# Verslag labo 1

1. **Question 1: DFS**

In de eerste opdracht werd er gekozen om het algoritme recursief te implementeren. Omdat het algoritme reeds gekend was, werd er online gezocht naar een oplossing in pseudocode. Hierna werd deze pseudocode omgezet in python en toegepast op de specifieke situatie. Het idee achter DFS is om eerst een volledige tak (branch) te volgen en het eerste pad dat naar het doel gevonden wordt als oplossing te nemen. Het spreekt voor zich dat deze oplossing niet ideaal zal zijn. Het recursieve algoritme begint bij de root node en zal iedere bekeken node als bezocht markeren zodat dezelfde locatie geen tweede keer bezocht kan worden. Vervolgens wordt de juiste bewegingsrichting toegevoegd aan het pad. Indien de goal state (lees: locatie met een dot) bereikt is, zal de recursieve functie True returnen. Indien dit niet het geval is, zal de functie opgeroepen worden op al zijn successors die nog niet bekeken zijn. De successors zijn alle locaties (en oa. de richting ten opzichte van de huidige locatie) waar PacMan naar zou kunnen bewegen als hij op de huidige node stond. Als een onderliggende node de goal gevonden heeft, zal deze True returnen en zal bijgevolg de node erboven (zijn parent node dus) ook True returnen. Als het einde van een tak bereikt wordt, dan wordt er False gereturned en zal de parent node de bovenste richting van het pad poppen zodat onnodige richtingen verwijderd worden. In de code zelf worden de stappen in meer detail toegelicht als commentaar. Er werd gebruik gemaakt van een Stack om het gewenste LIFO gedrag te verkrijgen.

1. **Question 2: BFS**

Het Breadth-first search (BFS) algoritme zal, startend van de source node, elk niveau controleren tot de bestemming is berreikt. Na het checken van niveau 1, springen we naar niveau 2; Daarna naar niveau 3, enzovoort. De volgorde bij het doorlopen van nodes in hetzelfde niveau kan verschillen van tijd tot tijd. Het processen van alle nodes, voor het springen naar een dieper niveau, is in dit algoritme uiterst belangrijk.

Om hierin te slagen, maakt het algoritme gebruik van een queue. Deze queue werkt volgens het first in, first out (FIFO) principe. De eerste stap in een while-lus is het fetchen van een node uit deze queue. In elke iteratie zijn er twee opties mogelijk. De eerste optie gaat als volgt: “Zijn we bij de bestemming aangekomen?”. Zo niet, voegen we de successors van de node toe aan de queue. Elke successor wordt automatisch toegevoegd aan de ‘visited’ stack. Deze stack werd ook gebruikt in het voorgaande DFS algoritme. Het zorgt ervoor dat geen enkele node twee keer wordt bezocht.

Stel dat we de bestemming hebben gevonden. Dit is niet genoeg. Net zoals bij DFS, moet er een lijst met directions teruggeven worden. Anders blijft Pacman stil staan op het scherm. In het BFS algoritme zoeken we een weg terug naar de source node (functie “backtrace”). Dit is mogelijk dankzij het bijhouden van een directionary ‘parent’. De weg naar boven omkeren, geeft ons een correct pad voor Pacman.

1. **Question 3**

In deze opdracht zal steeds gezocht worden naar de action die de laagste cost heeft om naar een volgende node te gaan om zo het path met de minste cost te verkrijgen. In elke node wordt opgeslagen in welke state ze zijn, welke actions moeten gebeuren om er te geraken en de cost om er te geraken. Dit is nodig om deze variablen te kunnen volgen. Er wordt ook gewerkt met een priorityqueue => deze zal steeds het item in zijn queue poppen met de laagste priority (wat in mijn code overeenkomt met de stepCost) zodat steeds naar de node met de laagste stepCost wordt gekeken. De code werkt als volgt: er wordt steeds geloopt met als voorwaarde dat de priorityque niet leeg is (als die leeg is, gaat die uit de loop). In de loop wordt de node met de laagste priority(stepCost) gepopt en daaruit worden “node, actions, stepCost” gehaald. Als deze node nog niet was bezocht, wordt deze in visited[] gezet en worden van deze node de mogelijke succesors bepaald (newAction,newStepCost,newNode). Deze nieuwe node wordt samen met de stepCost als priority in de priorityque gezet en dan begint de loop opnieuw tot uiteindelijk het hele path (dus alle actions) zijn bepaald en deze wordt gereturned.

1. **Question 4: A\* search**

A\* zal een optimaal pad zoeken en daarbij de laagste cost + heuristic als metric nemen. Op dit moment wordt er echter een nullHeuristic gebruikt, dit wil zeggen dat de heuristiek altijd nul zal zijn waardoor enkel de cost een rol speelt. De volgende node die telkens bezocht zal worden, is de node met de laagste waarde. Om dit te kunnen bepalen, zal er een PriorityQueue gebruikt worden. Bij het opvragen van een element (pop()) zal telkens de node met de laagste waarde (cost + heuristic) gegeven worden. Dit werd niet recursief opgelost waardoor er een loop gebruikt wordt totdat er een pad gereturned wordt. Ook hier werd er eerst naar een pseudo code gezocht die vervolgens geïmplementeerd moest worden in dit probleem. De cost zal telkens weergeven wat de cost is om tot een bepaalde node te geraken terwijl de heuristiek ons kan vertellen hoe wenselijk een node (vermoedelijk) is om tot bij het uiteindelijke doel te geraken.

1. **Question 5: Finding All the Corners**

Het doel van deze opdracht was het implementeren van “cornersProblem”. Dit is een “search problem” waarbij Pacman de 4 hoeken van een doolhof zo snel mogelijk probeert te zoeken.

In deze implementatie bestaat de state uit 2 delen: de start positie van Pacman en de bezochte hoeken. Corners\_visited is een 4-bit binair nummer. Elke bit kan gezien worden als een Boolean gerelateerd aan een specifieke hoek. Arriveert Pacman bij een hoek, dan zal zijn specifieke bit in corners\_visited op 1 (True) gezet worden. De hoeken waar Pacman nog naartoe moet, hebben een 0 (False) in corners\_visited. Een bit zetten op 1 is mogelijk met de bitsgewijze OF-operatie. De bitmask voor elke hoek staat in self.cornerMask.

Staan alle bits van corners\_visited op 1, dan is het programma gedaan en wordt het bijhorend resultaat getoond op het scherm. Met BFS zal Pacman een veel langer pad afleggen dan met het A\* search algoritme.

1. **Question 6**

Placeholder.

1. **Bonus question 7: Eating all the dots**

In deze opdracht wordt er gebruik gemaakt van de eerder geschreven A\* search. Het is echter de bedoeling om nu een zo goed mogelijke heuristiek op te stellen. Eerst moet isGoalState aangevuld worden. Aangezien we weten dat iedere food dot telt als een goal, kunnen we hier zeer eenvoudig gebruik maken van de food grid. Deze is toegankelijk door self.food. Deze geeft een matrix van gelijke grootte als het veld waarbij ieder veld op een boolean is ingesteld. De velden die op True staan, zijn velden met een dot op. De anderen hebben geen dot. Hierdoor kunnen we de gegeven positie gewoon returnen binnenin de food grid waardoor de velden met een dot als goal beschouwd worden maar de andere velden niet. Als heuristiek werd er eerst gekozen om de Manhattan Distance te nemen ten opzichte van de meest nabije goal state. Dit gaf echter geen goede resultaten dus werd er een moeilijkere oplossing geprobeerd: de heuristiek werd de som van de Manhattan Distance ten opzichte van alle goal states gedeeld door het aantal goal states (delen zodat de cost ook nog een relevante rol zou spelen). Dit wil zeggen dat een veld waar veel dots in de buurt zijn, een lagere heuristiek zou krijgen dan een veld waar weinig dots in de buurt zijn. Hierdoor wordt de som van cost + heuristic lager waardoor dit veld sneller bezocht zal worden en dus als beter beschouwd zal worden. Bijgevolg zullen er minder nodes bezocht moeten worden om een ideaal pad te vinden. Bij uitbreiding zou er nog gecontroleerd kunnen worden op muren tussen de state en goal state en daarbij kunnen er eventueel strafpunt(en) toegevoegd worden aan de heuristiek.

1. **Bonus question 8: Suboptimal search**

In deze opdracht ging het niet om het vinden van een ideaal pad maar om een pad te vinden dat telkens naar de meest nabije goal state verwijst. Eerst werd er geprobeerd om met een loop door alle goal states te gaan en aan de hand van Manhattan Distance de meest nabije food dot te bepalen. Daarna zou er een pad naar gezocht moeten worden dat liefst zo kort mogelijk was. Uiteindelijk bleek het efficiënter en simpeler om een eerdere opdracht te gebruiken: Breadth first search. Een [visualizer tool](https://clementmihailescu.github.io/Pathfinding-Visualizer/) op GitHub hielp erbij om dit inzicht te verkrijgen ([source code](https://github.com/clementmihailescu/Pathfinding-Visualizer)). Bovendien geeft breadth first search gegarandeerd het kortste pad.