**Algoritmes**

**bounds (alles op nationaal niveau)**

*theoretische upperbound*

**nationaal**

niet kritiek: 9778.9

kritiek: 9664.9

**Holland**

niet kritiek: 9911.3

kritiek: 9881.9

**Greedy**

*structuur:*

greedy versie<- lookup table (voorlopig in helper)

\

greedy-zoek algoritme (alg\_base\_greedy)

\

invalid functies

***greedy\_search***

Dit is het standaard greedy zoek algoritme dat voor een gegeven station de greedy route uitrekent. Hierbij kijkt hij telkens naar de randen van de huidige route. Welke verbindingen zijn daar en welke ervan de beste is. Welke de beste is wordt bepaald aan de hand van de lookup table die wordt meegeven. Voor zijn constraints leunt het zoek algoritme op de invalid functies, die bekijken of een route voldoet aan de gegeven constraints. Standaard moet de route voldoen aan de volgende twee constraints: 1. hij mag niet langer zijn dan de maximaal opgegeven tijdsduur 2.

Deze functie kan worden aangeroepen door andere functies. Dezen kunnen de volgorde van aanroepen en de gebruikte lookup table gebruiken, zoals het geval is in greedy\_random en recalculating\_greedy of simpelweg een enkele route opvragen zoals het geval is bij de hill\_climber\_random. Voor de beschrijving kijken we voornamelijk naar de eerste categorie, de abstractere greedy algoritmes. Hierbij zijn er twee versies aan te duiden:

***versies:***

**random greedy**

Recalculating\_greedy roept het zoek algoritme voor een random gekozen start station en rekent vervolgens de beste route hiervoor uit. De gebruikte tracks van de beste route verliezen vervolgens in de lookup table (een deel van) hun score. Immers dragen zij geen punten meer bij voor het overkoepelende resultaat. Hierna wordt er weer random een nieuw station gekozen. Dit proces herhaalt zich net zo lang tot het maximale aantal routes is bereikt of alle kritieke sporen zijn bereden.

**recalculating greedy**

Recalculating\_greedy roept het zoek algoritme aan voor elk station om vervolgens de beste route te kiezen uit de gevonden routes. De gebruikte tracks van de beste route verliezen vervolgens in de lookup table (een deel van) hun score. Immers dragen zij geen punten meer bij voor het overkoepelende resultaat. Vervolgens wordt vanuit deze nieuwe situatie opnieuw voor elk station routes uitgerekend. Dit proces herhaalt zich net zo lang tot het maximale aantal routes is bereikt of alle kritieke sporen zijn bereden.

***resultaten:***

\*algoritmes met random elementen zijn 10000 keer gerund

**greedy\_random :**

*nationaal*

kritiek: 9572.1

niet-kritiek: 9678.7

*Holland*

kritiek: 9881.0

niet\_kritiek: 9887.5

**recalculating\_greedy:**

*nationaal*

kritiek: 9549.6

niet kritiek: 9661.2

*Holland*



kritiek: 9549.6



*Heuristieken*

**-** het trimmen van de lijn, vooral nuttig bij niet alles kritiek. Hierbij wordt gekeken of er aan de randen van de routes puntenloze tracks zitten die er afgehaald kunnen worden. Levert daarop een goede winst **resultaat**

- inkorten van maximale duratie lijn met het idee dat korte routes beter zijn. Dit leverde niet beter resultaten op hoewel het opvallend was dat langer niet altijd beter betekende, zo was 165 beter dan 170. 180 moeten bleef echter het beste met *9661.2.*

- voorkomen van lange tracks midden in een route met het idee dat het inefficient is om heel lange tracks midden in je route te hebben. Hiervoor werd er geexpirimenteerd met maximale duur van een track en waar in de route hij mocht staan. Dus een track van 25 min of langer mag alleen aan de rand van een route liggen of een route van 15 min mag maximaal twee tracks afstand tot de rand van de route hebben. Met afstand tot de rand en lengte van de route werd gevarieerd. De beste combinatie bleek echter te zijn dat geen enkele track meer dan 12 van een rand van een route mochten liggen. Oftewel iedere route mag is idealiter maximaal 12 tracks. Dit leidde tot de volgende heurestiek.

- ideale maximale aantal tracks in een route. Er wordt gekeken bij maximaal hoeveel tracks per route het puntenaantal maximaal is.

*resultaat:*

kritiek: maximale lengte: 14, 9569





niet-kritieke: maximale lengte: 12, 9692.8





**Datastructuur**

Drie opties:

* node-centered
* line-centered
* Nodes and lines

1.

Hierbij stoppen wij alle data in de nodes en bestaan lijnen eigenlijk alleen binnen de station objects. Er wordt dus een netwerk van nodes gecreeerd die naar elkaar verwijzen. Hiervoor zouden dan de verschillende aanliggende stations moeten worden meegegeven, de reistijd ertussen en of de verbinding kritiek is.

Voordelen:

* Een enkel object met alle data, dus minder overhead
* Het doorzoeken van de objecten is net even wat makkelijker en sneller
* Stations kunnen makkelijk als startpunt gebruikt worden

Nadelen:

* Het is niet intuïtief
* Eigenlijk zijn er sporen er wel maar weggewerkt in de nodes, dus niet overzichtelijk
* Zeker bij het kritiek of niet kritiek van sporen is het een beetje omslachtig

2.

Hierbij zijn het de lijnen die alle informatie met zich meedragen. Er wordt een netwerk van lijnen gevormd die naar elkaar wijzen. In dit geval moeten de stations aan beiden uiteinden worden meegegeven, net als de aanliggende lijnen aan beide zijden.

Voordelen:

Ligt dicht bij de nodes.

* Een enkel object met alle data, dus minder overhead
* Het doorzoeken van de objecten is net even wat makkelijker en sneller
* Kritieke sporen zijn uiteindelijk van belang en dit maakt dat snel inzichtelijk

Nadelen:

* Ook niet intuïtief
* Nog onoverzichtelijker door de twee zijden waar weer lijnen aanliggen

3.

Hierbij wordt de intuitieve indeling tussen node en lijn in stand gehouden. Lijn verwijzen naar hun twee nodes en nodes verwijzen naar hun aanliggende lijnen. Er ontstaat dus een netwerk van nodes en lijnen ertussen.

Voordelen:

* Het meest intuitief en overeenkomende met de kaart
* Informatie wordt bewaard op logische en gescheiden plekken

Nadelen:

* Meer overhead, want meer objecten
* Om data op te vragen moet je ook meer objecten in
* Langzamer, maar O(n)?

**Algorithmes**

Twee stappen bij dit probleem. Overkoepelende aanpak oftewel welke filosofie volgen we op het grotere plaatje. Kiezen we beginpunten, maken we mogelijke routes gebaseerd op tijd etc. Hierbij hoort ook het verkleinen van de toestandsruimte. Dan op een kleiner niveau een pathing algorithme om routes te zoeken met onze gezette restricties.

**Algemeen**

* Het kiezen van een beperkt aantal beginpunten

**Intuitieve ideeen**

* Begin bij kritieke stations en zoek verder op andere kritieke

Je begint bij een van de kritieke stations en gaat vanaf die positie op zoek naar routes naar andere kritieke stations en dus sporen. Je rekent echter alle kanten op dus uiteindelijk kan het begin station ook midden in de route komen.

* Voordeel: Bij kleine hoeveelheden kritieke stations werkt dit waarschijnlijk vrij goed, omdat je zoveel mogelijk kritieke stations wil bereiden.
* Nadeel: Als alle sporen kritiek zijn of zelfs als er substantieel meer kritieke stations zijn dan toegestane routen, dan moet je of stations gaan selecteren of alsnog veel doorrekenen
* Begin bij ver uit elkaar liggende stations

Een manier om het probleem op te lossen. Je kiest ver uiteen liggende stations, het liefst kritiek, om zo een brede dekking te hebben.

* Begin bij routes die in korte tijd veel stations aandoen

Kijk naar een locatie waar relatief veel sporen bij elkaar liggen (dus tijdsafstanden) en begin vanaf daar een routes te maken die zoveel mogelijk stations aandoen in de gegeven tijd

* Minimaliseer dubbele bereiding sporen

Meer een manier om de toestandsruimte te verkleinen.

* Kritieke stations met meer dan 4 sporen, krijgen twee routes.

**Depth-first**

Depth-first is recursief geimplementeerd. Simpelweg bekijkt het algortime alle mogelijk routes van 120 min of minder om een bepaalde station. Hij kijkt per station naar de verbindende tracks en voegt hun bestemmingen een voor een toe aan een nieuwe potentiele route. Als de nieuwe route minder dan de maximaal toegestane duur heeft dan wordt de bestemming toegevoegd aan de huidige route en opnieuw naar de verbindende routes gekeken. Dit waaiert dus uit in een boom van vele mogelijke opties. Als een van de vele potentiele routes zijn tijdslimiet bereikt (een station extra zou over de maximale lengte gaan) dan wordt de score van die potentiele route teruggeven en steeds weer vergeleken met de andere vertakking, net zo lang tot de hoogste score uit de boom overblijft.

*Versie 1 (depth\_fjrst1 en depth\_search1)*

Deze is de meest basis versie. Hij kijkt voor elk station simpelweg naar de best mogelijke route voor elk station en kiest de hoogst scorende binnen het maximale aantal lijnen.

*Versie 2 (depth\_fjrst2 en depth\_search1) 9514.8*

Deze versie houdt bij welke kritieke tracks al door de beste route van voorgaande stations zijn gereden en geeft hier geen punten meer voor. Hierdoor wordt er dus gezocht om het aantal resterend tracks.

*Versie 3 (depth\_fjrst3 en depth\_search2) best resultaat 9514.8*

Deze versie checkt ook op mogelijke loops door te kijken of duos eerder voorkomen, dit geeft voornamelijk een runtime boost

*Versie 3,5 (depth\_first (licht aangepast) en depth\_search 2) resultaat 9498*

Tevens kwamen we er hier achter dat er niet rekening werd gehouden met de mogelijkheid dat een kortere route beter kan zijn. Nadat geimplementeerd te hebben ging de score echter naar 9498. Dit legt de vinger op deze zere plek van dit algoritme. Misschien kan een route die door het eerst bekeken station wordt bereden wel nog efficienter bereden worden door een andere route bij een later station.

Problemen:

* kan niet inschatten of routes uit latere stations efficienter zouden zijn
* startvolgorde van de stations maakt uit voor de score

**Literatuur**

Train Timetabling Problem (TTP)

* Flow algorithmes
* Dantzig and Fulkerson?