**Datastructuur**

Drie opties:

1. node-centered
2. line-centered
3. 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?