IDATT2101 Øving 7

ALT og Dijkstra

Trang Minh Duong

Denne oppgaven har jeg benyttet ALT og Dijkstra algoritmer få å finne raskeste rute, antall distansenoder og tiden det tar mellom to noder.

Dijkstras algoritme

Veinettet modelleres som en rettet, vektet graf der nodene representerer veikryss og kantene representerer veistrekninger. Kantvektene er kjøretid i hundredels sekund, som gitt i datasettet. Dijkstras algoritme benytter en min-heap prioritetskø der prioriteten til hver node er den kjente avstanden fra startnoden.

```
priority_queue<
        pair<int32_t,uint32_t>, vector<pair<int32_t,uint32_t
            >>,
        greater<pair<int32_t,uint32_t>>
> pq;
dist[s] = 0;
pq.push({0,s});
while(!pq.empty()){
    auto [d,u] = pq.top();
    pq.pop();
    if(used[u]) continue;
    used[u]=1;
    for(const auto &e: adj[u]){
        if(dist[e.to] > d + e.w) {
            dist[e.to] = d+e.w;
            pq.push({dist[e.to], e.to});
        }
    }
}
return dist;
```

Listing 1: priority queue i `djikstra_all` (preprossesering)

For hver utplukket node relakseres alle tilstøtende kanter, og distansen til en nabo oppdateres dersom en kortere rute er funnet. I implementasjonen teller vi `queue_pops` (hvor mange ganger en node tas ut av køen) og `relaxations` (hvor mange ganger en avstand forbedres). Disse brukes til å sammenligne arbeidsmengden mellom Dijkstra og ALT.

```
priority_queue<FrontierItem, vector<FrontierItem>, PQCmp> pq
    ;
auto t0 = chrono::high_resolution_clock::now();
g[s] = 0;
h[s] = alt_heuristic(alt, s, t);
parent[s]=s;
pq.push({g[s]+h[s], s});
SearchStats st;
while(!pq.empty()){
    auto [f,u] = pq.top();
    pq.pop();
    if(used[u]) continue;
    used[u]=1;
    st.pq_pops++;
    if(u==t) break;
    for(const auto &e: G.adj[u]) {
        int32_t ng = (g[u] == INF ? INF : g[u] + e.w);
        if(ng < g[e.to]) {
            g[e.to] = ng;
            if(h[e.to] == 0 && e.to!=s) {
                h[e.to] = alt_heuristic(alt, e.to, t);
            }
            parent[e.to] = u;
            pq.push({g[e.to] + h[e.to], e.to});
            st.relaxations++;
        }
    }
```

Listing 2: queue_pops og relaxations i `alt_search`

ALT-algoritme (A*, Landemerker og Trekantulikheten)

ALT (A*, Landmarks, Triangle inequality) er en utvidelse av A*-algoritmen som bruker forhåndsberegnede avstander til og fra et sett med landemerker. Landemerker er strategisk utvalgte noder som ligger spredt langs grafens ytterkanter, for eksempel den nordligste, sør- og østligste noden. Målet er å velge punkter som ligger langt unna de fleste ruter, slik at de gir sterke nedre estimater på gjenværende avstand.

Forhåndsberegning. For hvert landemerke L kjøres Dijkstras algoritme to ganger:

- på originalgrafen for å beregne avstander *fra* L,
- på den omvendte grafen for å beregne avstander *til* L.

Dette lagres i to tabeller:

$$\text{fromL}[L][v] = d(L, v), \qquad \text{toL}[v][L] = d(v, L).$$

Bruk av trekantulikheten. ALT-heuristikken bygger direkte på trekantulikheten:

$$d(A, C) \leq d(A, B) + d(B, C).$$

Ved å omforme denne får vi to gyldige nedre estimater for avstanden fra n til målnoden t:

$$d(n,t) \geq d(L,t) - d(L,n),$$
$$d(n,t) \geq d(n,L) - d(t,L).$$

Den første følger fra

$$d(L,t) \leq d(L,n) + d(n,t),$$

og den andre fra tilsvarende uttrykk i den omvendte grafen. Siden avstander ikke kan være negative, tar vi også maksimum med 0.

Heuristikk. ALT kombinerer alle landemerker ved å ta maksimum av alle del-estimater:

$$h(n) = \max_{L} \left(d(L,t) - d(L,n),\ d(n,L) - d(t,L),\ 0 \right).$$

```
1  static inline int32_t alt_heuristic(const ALTData &alt,
       uint32_t n, uint32_t goal){
2      if(!alt.loaded) return 0;
3      int32_t best = 0;
4      const size_t L = alt.landmark_ids.size();
5      for(size_t l = 0; l < L; l++){
6          int32_t a = alt.fromL[l][goal]; // d(L, t)
7          int32_t b = alt.fromL[l][n]; // d(L, n)
8          if(a < INF && b < INF) best = max(best, max(0, a - b
               )); // max(0, d(L,t) - d(L,n))
9          int32_t c = alt.toL[n][l]; // d(n, L)
10         int32_t d = alt.toL[goal][l]; // d(t, L)
11         if(c < INF && d < INF) best = max(best, max(0, c - d
               )); // max(0, d(n,L) - d(t,L))
12     }
13     return best;
14 }
```

Listing 3: Heuristikk over landemerker

Korrekthet og ytelse. Siden ALT-heuristikken alltid er en nedre grense på den faktiske avstanden, er den en gyldig nedre grense. Dermed finner ALT alltid en optimal rute. I praksis gir den sterke heuristikken meget store besparelser i antall `queue_pops` og `relaxations`, spesielt på lange ruter som Trondheim–Oslo eller Gol–Lakselv.

Preprosessering

Før ALT kan brukes, kjøres:

- én Dijkstra for hvert landemerke på originalgrafen,
- én Dijkstra for hvert landemerke på den omvendte grafen.

Dette gir to matriser av størrelse $L \times N$ og $N \times L$, som lagres i binærfiler for å unngå mange minutters beregning ved senere kjøringer. Landemerkene velges som geografiske ytterpunkter i grafen (min/max breddegrad og lengdegrad). (Se listing 1 for eksempel)

Nærmeste interessepunkt (k-nærmeste)

For å finne de k nærmeste interessepunktene (POI-er) av en gitt type kjøres Dijkstra fra startnoden, men søket stoppes så snart det er funnet k POI-er som matcher bitmasken. Dette gir i verste fall samme kompleksitet som full Dijkstra, men terminerer vanligvis svært tidlig.

```
1 vector<NearbyPOI> nearest_pois(const Graph &G, uint32_t src,
      uint32_t mask, int k){
2     const uint32_t N = (uint32_t)G.nodes.size();
3     vector<int32_t> dist(N, INF);
4     vector<char> used(N,0);
5     vector<vector<pair<uint32_t,uint32_t>>> poiIndex(N); //
          node -> list of (code,name_idx)
6     for(size_t i = 0; i < G.pois.size(); i++) {
7         const auto &p = G.pois[i];
8         if(p.node < N) poiIndex[p.node].push_back({(uint32_t
              )i, p.code});
9     }
```

Listing 4: Funksjon for å finne næremest POI

Kompleksitet

Dijkstra

Med N noder og M kanter og en heap-basert prioritetskø er tiden:

$$O\big((N+M)\log N\big)$$

Minnet er $O(N+M)$ for grafen og $O(N)$ for hjelpearrayer som `dist`, `parent` og `used`.

ALT

Alt består av to faser:

Preprosessering For L landemerker:

$$O\big(L\,(N+M)\log N\big)$$

siden det kjøres én Dijkstra for hver retning og hvert landemerke. Minnet er:

$$O(LN)$$

for heuristikk-tabellene.

Søk Worst case er lik Dijkstra, men i praksis reduseres antall `queue_pops` og `relaxations` dramatisk.

Resultat

Setter:

```
DATA='C:/Users/duong/OneDrive - NTNU/Documents/cpp/algdat/Oeving7/'
```

Gløshaugen–Otilienborg og Fosnavåg–Espoo

```
$ ./oeving7 --data "$DATA" --route 2001238 1987066 --algo dijkstra --csv d_glos_otilienborg.csv
./oeving7 --data "$DATA" --route 2001238 1987066 --algo alt      --csv a_glos_otilienborg.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
ALGO = Dijkstra
start = 2001238 goal = 1987066
travel_time(h:mm:ss.mmm) = 0:03:56.960
queue_pops = 3565 relaxations = 3955 time_ms = 1
path_nodes = 53
Wrote d_glos_otilienborg.csv (lat, lon) for plotting
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
ALT cache loaded.
ALGO = ALT
start = 2001238 goal = 1987066
travel_time(h:mm:ss.mmm) = 0:03:56.960
queue_pops = 1338 relaxations = 1575 time_ms = 9
path_nodes = 53
Wrote a_glos_otilienborg.csv (lat, lon) for plotting
```

(a) Log Gløshaugen–Otilienborg

```
$ ./oeving7 --data "$DATA" --route 2486870 5394165 --algo dijkstra --csv d_fosnavag_espoo.csv
./oeving7 --data "$DATA" --route 2486870 5394165 --algo alt      --csv a_fosnavag_espoo.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
ALGO = Dijkstra
start = 2486870 goal = 5394165
travel_time(h:mm:ss.mmm) = 20:01:06.970
queue_pops = 6755846 relaxations = 7047790 time_ms = 11330
path_nodes = 5133
Wrote d_fosnavag_espoo.csv (lat, lon) for plotting
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
ALT cache loaded.
ALGO = ALT
start = 2486870 goal = 5394165
travel_time(h:mm:ss.mmm) = 20:01:06.970
queue_pops = 919806 relaxations = 987005 time_ms = 4719
path_nodes = 5133
Wrote a_fosnavag_espoo.csv (lat, lon) for plotting
```

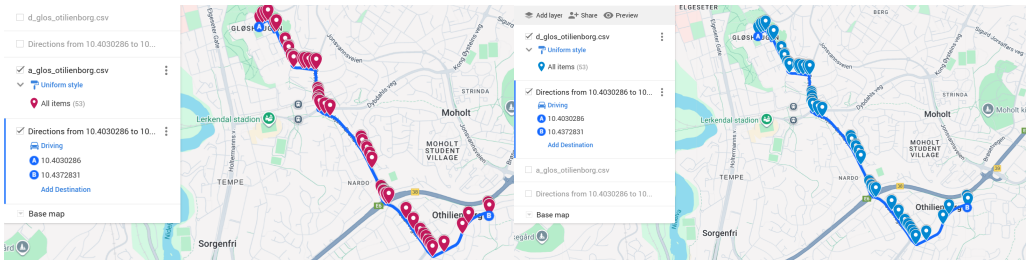
(b) Log Forsnavåg–Espoo

Figur 1: Log for kjøringen

Rute	Algoritme	Reisetid (tt:mm:ss.mmm)	queue_pops	relaxations	tid [ms]
Gløshaugen–Otilienborg	Dijkstra	0:03:56.960	3565	3955	∼0
	ALT	0:03:56.960	1338	1575	∼0
Fosnavåg–Espoo	Dijkstra	20:01:06.970	6755846	7047790	1570
	ALT	20:01:06.970	919806	987005	611

Tabell 1: Sammenligning av arbeidsmengde mellom algoritmene.

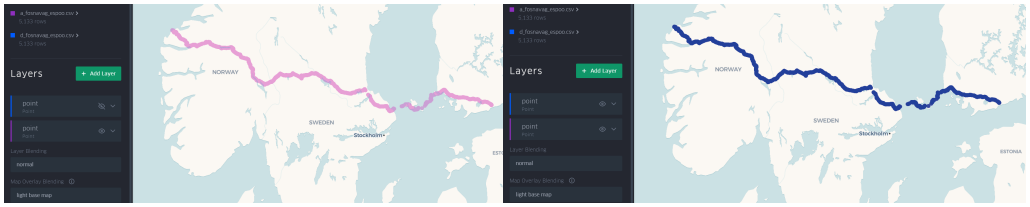
Vi kan se at tiden for algoritmen å regne ruten Gløshaugen – Otilienborg er ~ 0 pga. det er bare 53 noder mellom start og slutt. Det er 5133 noder mellom Fosnavåg og Espoo, og dermed tok algoritmen mer enn 0 ms (1570 ms for Dijkstra og 611 ms for ALT)



(a) Kart Gløshaugen–Otilienborg med ALT

(b) Kart Gløshaugen–Otilienborg med Dijkstra

Figur 2: Kart Gløshaugen–Otilienborg ved bruk av Google My Maps



(a) Kart Forsnavåg–Espoo med ALT **(b)** Kart Fosnavåg–Espoo med Dijkstra

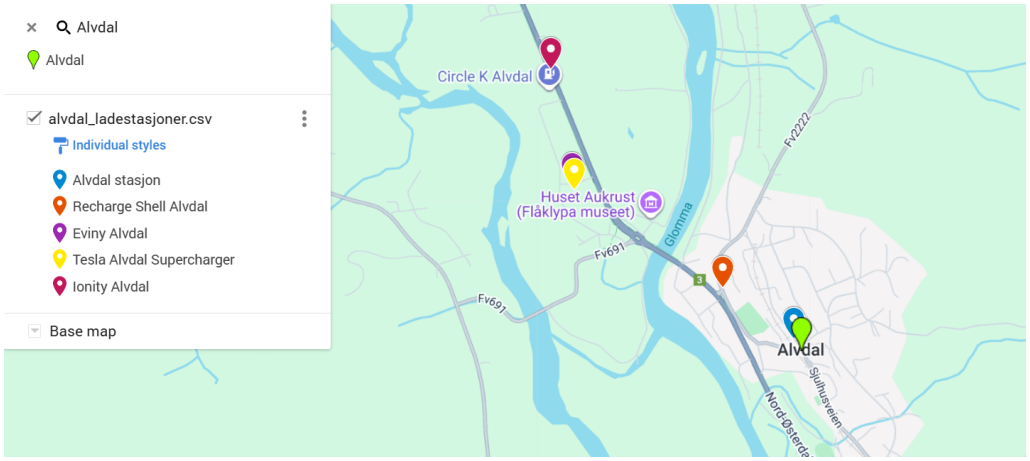
Figur 3: Kart Forsnavåg–Espoo ved bruk av kepler.gl

5 ladestasjoner nær «Alvdal» (node 3414169)

5 spisesteder:

```
$ ./oeving7 --data "$DATA" --nearest 3414169 --mask 4 --k 5 --csv alvdal_ladestasjoner.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
nearest k = 5 from node = 3414169 mask = 4
1. node = 3884449 time = 0:00:08.240 code = 4 name = "Alvdal stasjon" lat = 62.109 lon = 10.6334
2. node = 6516590 time = 0:00:23.250 code = 4 name = "Recharge Shell Alvdal" lat = 62.1109 lon = 10.
6276
3. node = 5387052 time = 0:01:34.020 code = 4 name = "Eviny Alvdal" lat = 62.1148 lon = 10.6156
4. node = 7869370 time = 0:01:37.440 code = 4 name = "Tesla Alvdal Supercharger" lat = 62.1146 lon =
 10.6157
5. node = 2186419 time = 0:01:44.360 code = 4 name = "Ionity Alvdal" lat = 62.1191 lon = 10.6138
nearest_pois_time_ms = 194
```

Figur 4: 5 nærmeste ladestasjoner



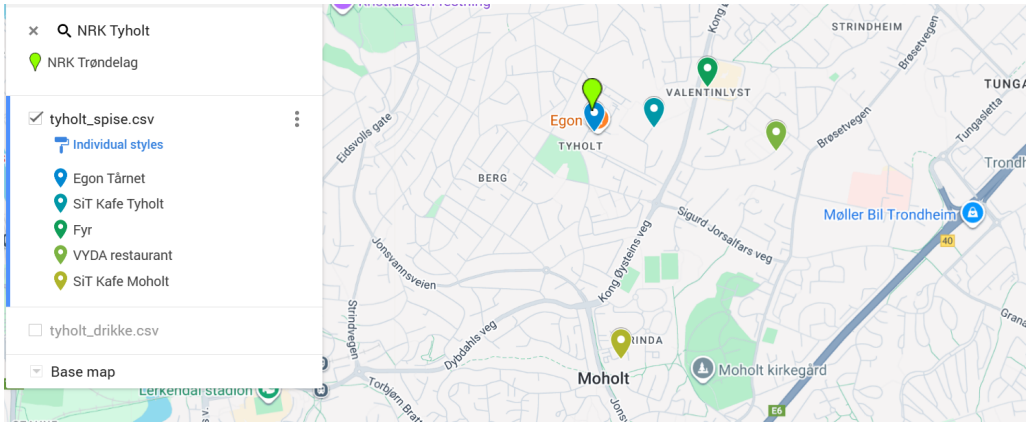
Figur 5: Kart av de 5 ladestasjonene nær Alvdal (grønn) ved bruk av Google My Maps

5 spise- eller drikkestedene som er nærmest «NRK Tyholt» (node 2374446)

5 spisesteder:

```
$ ./oeving7 --data "$DATA" --nearest 2374446 --mask 8 --k 5 --csv tyholt_spise.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
nearest k = 5 from node = 2374446 mask = 8
1. node = 2374445 time = 0:00:07.020 code = 8 name = "Egon TÅ¥rnet" lat = 63.4225 lon = 10.4315
2. node = 1957368 time = 0:01:02.330 code = 8 name = "SiT Kafe Tyholt" lat = 63.4226 lon = 10.4365
3. node = 1998113 time = 0:01:08.160 code = 8 name = "Fyr" lat = 63.4242 lon = 10.4409
4. node = 1943205 time = 0:01:21.560 code = 8 name = "VYDA restaurant" lat = 63.4218 lon = 10.4467
5. node = 1944945 time = 0:01:42.590 code = 8 name = "SiT Kafe Moholt" lat = 63.414 lon = 10.4337
nearest_pois_time_ms = 680
wrote tyholt_spise.csv
```

Figur 6: 5 nærmeste spisesteder

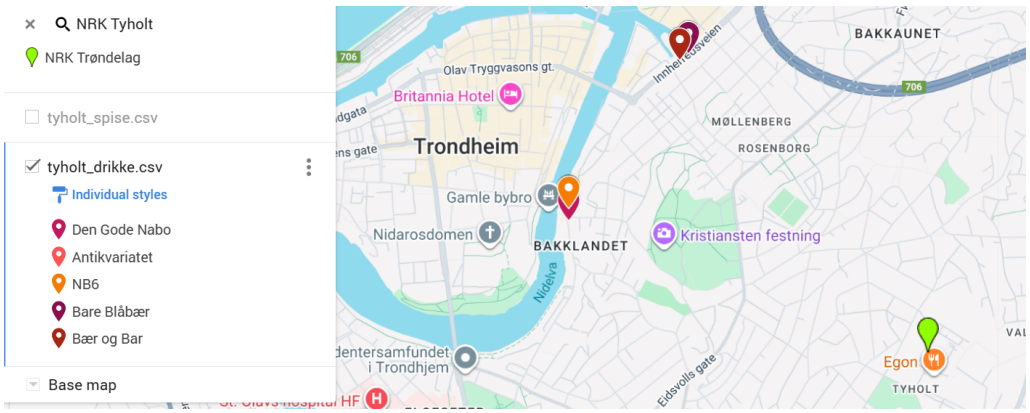


Figur 7: Kart av de 5 spisesteder nær NRK Tyholt (grønn) ved bruk av Google My Maps

5 drikkesteder:

```
$ ./oeving7 --data "$DATA" --nearest 2374446 --mask 16 --k 5 --csv tyholt_drikke.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
nearest k = 5 from node = 2374446 mask = 16
1. node = 2004122 time = 0:03:08.220 code = 24 name = "Den Gode Nabo" lat = 63.4279 lon = 10.403
2. node = 1999122 time = 0:03:13.240 code = 24 name = "Antikvariatet" lat = 63.4283 lon = 10.403
3. node = 1999122 time = 0:03:13.240 code = 16 name = "NB6" lat = 63.4283 lon = 10.403
4. node = 7938421 time = 0:03:15.330 code = 24 name = "Bare BlÅ¥bÅ¦r" lat = 63.4338 lon = 10.4125
5. node = 7938422 time = 0:03:18.490 code = 24 name = "BÅ¦r og Bar" lat = 63.4335 lon = 10.4118
nearest_pois_time_ms = 523
wrote tyholt_drikke.csv
```

Figur 8: 5 nærmeste drikkesteder



Figur 9: Kart av de 5 drikkesteder nær NRK Tyholt (grønn) ved bruk av Google My Maps (Antikvariatet og NB6 er på samme sted)

5 spise- og drikkesteder:

```
$ ./oeving7 --data "$DATA" --nearest 2374446 --mask 24 --k 5 --csv tyholt_spise_drikke.csv
Loaded nodes: 7956886
Loaded edges: 17815613
Loaded POIs: 277800
nearest k = 5 from node = 2374446 mask = 24
1. node = 2374445 time = 0:00:07.020 code = 8 name = "Egon TÃ¥rnet" lat = 63.4225 lon = 10.4315
2. node = 1957368 time = 0:01:02.330 code = 8 name = "SiT Kafe Tyholt" lat = 63.4226 lon = 10.4365
3. node = 1998113 time = 0:01:08.160 code = 8 name = "Fyr" lat = 63.4242 lon = 10.4409
4. node = 1943205 time = 0:01:21.560 code = 8 name = "VYDA restaurant" lat = 63.4218 lon = 10.4467
5. node = 1944945 time = 0:01:42.590 code = 8 name = "SiT Kafe Moholt" lat = 63.414 lon = 10.4337
nearest_pois_time_ms = 687
wrote tyholt_spise_drikke.csv
```

Figur 10: 5 nærmeste spise- eller drikkesteder

Sammenligne med løsningsforslag

Rute	Algoritme	Reisetid (tt:mm:ss.mmm)	queue_pops	relaxations	path_nodes	tid [ms]
Kårvåg–Gjemnes	Dijkstra	0:40:47.340	13407	13962	334	2
	ALT	0:40:47.340	4297	4543	334	1
Gjemnes–Kårvåg	Dijkstra	0:40:47.340	20004	20994	334	4
	ALT	0:40:47.340	3940	4192	334	1
Malmö Brygghus–Nørrebro	Dijkstra	0:32:44.050	111607	119795	408	28
	ALT	0:32:44.050	35599	39714	408	17
Nørrebro–Malmö Brygghus	Dijkstra	0:33:35.040	203117	213499	410	46
	ALT	0:33:35.040	23534	26094	410	12
Trondheim–Oslo	Dijkstra	5:53:13.290	1027523	1066521	1981	168
	ALT	5:53:13.290	116248	125353	1981	47
Oslo–Trondheim	Dijkstra	5:53:00.770	3253843	3398916	2030	590
	ALT	5:53:00.770	489485	515273	2030	154
Jyväskylä–Kárášjohka	Dijkstra	11:03:16.930	2607364	2721688	3451	599
	ALT	11:03:16.930	840447	901236	3451	324
Kárášjohka–Jyväskylä	Dijkstra	11:03:54.550	1149379	1197750	3407	243
	ALT	11:03:54.550	318500	337439	3407	160
Gol–Lakselv Hotell	Dijkstra	21:07:07.040	7801017	8131802	7518	1796
	ALT	21:07:07.040	2554910	2680316	7518	933
Lakselv Hotell–Gol	Dijkstra	21:06:48.720	5388730	5619051	7524	1222
	ALT	21:06:48.720	1154301	1216948	7524	454
Esso Sauda–Tervo	Dijkstra	21:44:14.380	7442388	7762293	5796	1656
	ALT	21:44:14.380	1496011	1577200	5791	733
Tervo–Esso Sauda	Dijkstra	21:45:43.500	7507150	7830986	5820	1337
	ALT	21:45:43.500	2835489	3005936	5815	1244

Tabell 2: Sammenligning av reisetid, arbeidsmengde, rute-lengde og tid for Dijkstra og ALT på ulike ruter.

Fra begge tabellene kan vi bekrefte at begge algoritmene fungerer som forventet. Reisetiden stemte med løsningsforslaget. I tillegg er antall avstandsnoder, queue-pops, og relaxations for ALT er lavere enn for Dijkstra som forventet. Forskjellen i arbeidsmengden mellom Dijkstra og ALT skyldes at algoritmene utfører helt ulik mengde søkearbeid. Selv om begge alltid finner den optimale ruten, bruker ALT langt færre operasjoner underveis. Dette er på grunn av:

Dijkstra har ingen retningsinformasjon Dijkstra må utforske grafen i alle retninger samtidig, som en sirkulær bølge som brer seg utover fra startnoden. For å garantere optimalitet må alle noder med avstand $\leq d(s,t)$ poppes fra køen. På lange ruter (flere hundre eller tusen kilometer) betyr dette at algoritmen kan måtte traversere store deler av grafen. Dette gir ofte millioner av `queue_pops` og `relaxations`, slik resultatene viser.

ALT bruker trekantulikheten til å lage en nedre grense på avstanden som nevnt tidligere, bygger ALT på A*-algoritmen og bruker landemerker for å beregne en heuristikk som estimerer hvor langt det minst kan være igjen til målet

$$h(n) = \max_{L} \left(d(L,t) - d(L,n),\ d(n,L) - d(t,L),\ 0 \right).$$

Dette uttrykket er en matematisk nedre grense på avstanden $d(n,t)$, utledet fra trekantulikheten. Dersom en node ligger geografisk "feil vei" i forhold til målet, vil heuristikken

bli stor, og noden får høy $f = g+h$-verdi i prioritetskøen. Dermed unngår ALT å utforske store deler av grafen.

ALT snevrer inn søket til en smal "tunnel" mot målet A* (og dermed ALT) prioriterer noder nærmest den optimale ruten. Søkeområdet endres fra en stor "sirkel" (Dijkstra) til en smal "korridor":

- få noder blir tatt ut av køen,
- få naboer blir vurdert,
- få relaxations skjer,
- men ruten blir identisk med Dijkstra.

Dette stemte med resultatene, der ALT typisk har 10–20 ganger færre queue_pops og relaxations sammenlignet med Dijkstra, selv om reisetiden og antall avstandsnoder er identisk.

path_nodes alltid er likt Begge algoritmene finner samme optimale rute. Forskjellen ligger utelukkende i hvor mye "unødvendig arbeid" som gjøres før den riktige ruten nås. Dijkstra må utforske alt innenfor avstandsringen rundt start, mens ALT leder søket direkte inn mot korrekt rute. Dette gir dramatisk bedre ytelse, som resultatene i tabellene viser.

Begrensninger

- **Statisk datasett:** Grafen gjenspeiler ikke trafikk, kø, vær eller midlertidige stengninger. Reisetiden er derfor teoretisk og ikke realistisk i et virkelig trafikkscenario.
- **Asymmetri i rutene:** Veinettet er modellert som en rettet graf. En rute $A \to B$ kan ha helt annen reisetid enn $B \to A$ på grunn av enveiskjøringer og rundkjøringer.
- **Minnebruk i ALT:** Heuristikk-tabellene krever betydelig minne. For $N \approx 8$ millioner og $L = 6$ landemerker utgjør dette flere hundre MB. Preprosesseringen tar i tillegg flere minutter.
- **Encoding-problemer:** Interessepunktdataene er i UTF-8, men terminalen som brukes kan ende opp med feil visning av norske tegn (f.eks. "Tårnet" vises som "TÃ¤rnet"). Selve datastrukturen lagrer navnene korrekt.

```
$ grep -i "Tyholt" interessepkt.txt
2374446 1       "Tyholt"
1957368 8       "SiT Kafe Tyholt"
1955665 4       "NRK Tyholt"
1852826 4       "Tyholt HVFS"
7062709 1       "Tyholt"
2393808 1       "Tyholt"
7441350 1       "Tyholt"
6383008 1       "Tyholt"
```

Figur 11: grep-sjekk

- **Avbrytning i nearest-POI:** Søket avbrytes når k treff er funnet. Dette gir rask respons, men gir ikke garanti for global oversikt over alle POI-er som ligger på samme avstand.