UNIVERSITEIT GENT

Project AD3: OpenSeekMap

Arno Vermote - 01806792

Academiejaar 2020-2021

1 Probleemstelling

Er moet een programma opgesteld worden dat zoekopdrachten kan uitvoeren. Er wordt een databank van locaties en een zoekterm gegeven aan het programma. Het programma geeft dan de 5 beste zoekresultaten uit de databank terug.

2 Implementatie zoekalgoritme

Om deze probleemstelling aan te pakken, maken we gebruik van een variant van algoritme 7 uit de cursus. Dit algoritme neemt 2 strings als input en geeft de editeerafstand terug. De aanpassingen die gebeurd zijn, hebben een invloed op de resultaten, geheugengebruik en snelheid.

2.1 Letters verwisselen

Het originele algoritme laat niet toe om letters te verwisselen met kost 1. Dit wordt oorspronkelijk gezien als 2x een letter vervangen, dus de kost 2 wordt toegekend. Door een kleine aanpassing aan het algoritme kan ondersteuning voor verwisselen toegevoegd worden. We controleren of een verwisseling mogelijk is aan de hand van een extra controle: we kijken of de vorige letter van tekst 1 overeenkomt met de huidige letter van tekst 2, en omgekeerd. Indien dit waar is, is er dus een verwisseling mogelijk. We doen dan de huidige kost - 1, omdat we oorspronkelijk 2 hadden betaald voor de dubbele substitutie, en dan nu dus in totaal 1 betaald hebben voor een verwisseling.

Er is echter wel een probleem: stel text1 = 'aba' en text2 = 'bab'. Dan zou volgens bovenstaande aanpassing er 2x een verwisseling optreden. Dit is fout. Daarom wordt er bijgehouden of er in de vorige iteratie een verwisseling is gedetecteerd. Indien dit zo is, kan er in de huidige iteratie zeker geen verwisseling optreden: er kan echter maar 1 keer verwisseld worden met 3 letters!

```
De relevante code staat in
src/Searcher/Searcher.c, regels 51:62
```

2.2 Optimalisatie geheugen

In algoritme 7 van de cursus wordt steeds de volledige matrix bijgehouden. We hebben echter enkel de vorige kolom nodig om de huidige te bereken. We kunnen dus de rest van de matrix weggooien. De gebruikte implementatie is analoog aan die van oefening 68, met enkel de returnwaarde aangepast.

```
De relevante code staat in
src/Searcher/Searcher.c, regel 25
```

2.3 geheugenimpact UTF-8

Aangezien het gebruikte algoritme geen karakteristieke vectoren opstelt, is er dus ook geen merkbare geheugenimpact.

2.4 Slechte oplossingen detecteren

Het zal vaak voorkomen dat een databanklijn niet binnen de editeerafstand van 3 valt ten opzichte van een gegeven zoekterm. Daarom kan het voordelig zijn om tijdens de uitvoering van het algoritme te controleren of de maximaal toegelaten editeerafstand niet overschreden is. Dit wordt gedaan door na het opstellen van de huidige kolom te kijken of iedere waarde van de huidige kolom groter is dan de maximaal toegelaten editeerafstand. Indien dit zo is, kan de uitkomst na het uitvoeren van het volledige algoritme nooit kleiner zijn dan de toegelaten waarde, dus zal het een slechte match zijn. We kunnen dus stoppen met het algoritme.

Het uitvoeren van deze check zal natuurlijk ook impact hebben op de snelheid van het algoritme. Echter zijn de databanklijnen vaak zo verschillend dat er bijna altijd onmiddellijk kan onderbroken worden, en er dus toch tijd bespaard wordt.

```
De relevante code staat in
src/Searcher/Searcher.c, regel 64
```

2.5 Preventie dubbel zoeken

Er is nog een andere manier om de snelheid te verbeteren. Stel als zoekterm 'De Sterre Gent'. Deze zoekterm wordt dan gesplitst in verschillende delen. Echter kunnen 2 opsplitsingen wel een zelfde 'subopsplitsing' hebben: bijvoorbeeld ['De Sterre', 'Gent'] en ['De', 'Sterre', 'Gent'] hebben de opsplitsing 'Gent' gemeen. Door het zoekresultaat van 'Gent' op te slaan, voorkomen we dat we 2x dezelfde term moeten zoeken. Dit brengt een duidelijke verbetering in snelheid.

```
De relevante code staat in
src/main.c, regels 31:38, 66:75
```

3 Sortering databank

Bij het inlezen van de databank worden de databanklijnen gesorteerd op lengte van de plaatsnaam: lijnen met een gelijke lengte worden in dezelfde lijst gestoken. Door deze lijsten op een gestructureerde manier op te slaan, kunnen we dus alle plaatsnamen met lengte n opvragen. Dit blijkt zeer handig te zijn als we later moeten zoeken: stel lengte van zoekterm $= x$. Door de voorwaarde dat er niet 3 of meer letters mogen toegevoegd/verwijderd worden, kunnen we de zoekopdracht beperken tot databanklijnen van lengte $x - 2$ tot en met $x + 2$. Dit heeft natuurlijk een enorme impact op de snelheid van het programma.

```
De relevante code staat in
src/DatabaseReader/DatabaseReader.c, regels 39:40
```

4 Tijdscomplexiteit

4.1 Programmastructuur

Voor we de complexiteit gaan berekenen, zullen we eerst de structuur van het programma verkennen. Sterk vereenvoudigd, is deze als volgt:

```
readDatabase();
for every searchterm {
    best_matches = [];
    for {
    for every possible split to subsets {
        results = [];
        for every word in this split {
             results.append(searchInDatabase(word));
        }
        for every total_match {
            updateBestMatches(total_match);
        }
    }
}
return best_matches;
```

4.2 Complexiteit berekenen

Stel:
z: het aantal zoektermen
l_z: de lengte van de langste zoekterm
w: het aantal woorden in de zoekterm met de meeste woorden
n: het aantal elementen in de databank
l_d: de lengte van het langste plaatsnaam in de databank

- De eerste for-loop zal z keer worden uitgevoerd.
- De tweede for-loop zal $2^{(w-1)}$ keer worden uitgevoerd.
- De derde for-loop zal maximaal w keer worden uitgevoerd.
- In het slechtste geval zal de volledige databank moeten doorzocht worden. Dit zijn dus n zoekopdrachten.
- Iedere zoekopdracht heeft een complexiteit van $\mathcal{O}(l_z * l_d)$ (Lemma 21 uit cursus).
- Bij het bepalen van de beste matches moeten eerst de totale matches opgesteld worden. In het geval dat alle databanklijnen een match zijn voor ieder woord, zal results w keer n zoekresultaten bevatten. Er zijn dus n^w mogelijke combinaties als totale matches.

- Het berekenen van de score is w^2, want de gegeven formule van synergie is kwadratisch.

Hieruit besluiten we dan dat de complexiteit gelijk is aan $\mathcal{O}(z * 2^{(w-1)} * w * ((n * l_z * l_d) + (n^w * w)))$.

5 Testen

De testen bestaan uit 2 delen: het eerste deel test individuele functies. Het principe is gelijkaardig aan dat van Unit-testen. Indien alle testen succesvol zijn, wordt exitcode 0 teruggegeven. Het tweede deel van de testen is vergelijkbaar met de Docker-testen die op SubGit stonden. De output van het programma zelf wordt vergeleken met wat volgens mij het correcte antwoord is. Dit kan doordat de input en databank zodanig zijn opgesteld dat het voorspelbaar is wat de output zal zijn.

```
De relevante code staat in
tests/runTests.sh
tests/FunctionTests/test_main.c
```