DA3018 Projekt

Anton Wallgren, Albin Niva Printz, Dennis Uygur Andersson

June 1, 2017

1 Inledning

Vi angrep problemet genom att tolka den samlade datan ur ett matematiskt perspektiv för att därefter utveckla ett datorprogram som delar upp inläsningen av datasamlingen i mindre delproblem.

2 Metod

2.1 Problemmodellering

Vi valde att tolka contigerna och deras överlappningar som en graf där hörnen representerar contigerna och kanterna representerar deras överlappningar. Utifrån bakgrunden gjorde vi antagandet att de contiger som har många överlappningar antagligen kommer från ett lågkomplexitetsområde och borde därför kunna sorteras bort. Detta kommer förhoppningsvis göra att graferna dessa noder befinner sig i bryts upp i flera mindre grafer. Uppgiften blir då i vår modell att hitta de hörn i grafen som har många grannar, ta bort dessa och låta de sammanhängande delgraferna utgöra delproblemen som blir vår utdata.

2.2 Vår lösning

Vi har använt oss av en kombination av Python- och shellscript. Först rensas contiger med fler än ett visst antal överlappningar ut med hjälp av Bash och Python. Sedan skapas ett grafobjekt i Python av resterande contiger, och grafens sammanhängande delgrafer skrivs som strängar bestående av hörnnamnen till en resultatfil, i vilken alltså varje rad är ett delproblem.

2.3 Användning

För att köra mjukvaran startar man ”main.sh” med Bash från huvudmappen med sökväg till indatafilen som första argument och högst tillåtna antal överlappningar som andra argument.

Exempel:

```
$bash main.sh resources/Spruce_fingerprint_2017-03-10_16.48.olp.m4 10
```

Efter att mjukvaran har körts en gång på utdatan, oavsett värde på det andra argumentet, kan mjukvaran nästa gång köras (med samma indata) med ett tredje "skip"-argument som gör att mjukvaran hoppar över vissa steg och därmed går snabbare. Detta tredje argument kan vara vad som helst, det viktiga är att något anges.

Exempel:

```
$bash main.sh resources/Spruce_fingerprint_2017-03-10_16.48.olp.m4 10 ;

bash main.sh resources/Spruce_fingerprint_2017-03-10_16.48.olp.m4 15 skip ;

bash main.sh resources/Spruce_fingerprint_2017-03-10_16.48.olp.m4 20 1
```

I exemplet ovan kommer mjukvaran att köra en kortare version i det andra samt tredje anropet. Utdatan skrivs till "Results/Result_n.txt", där n är högst tillåtna antal överlappningar. Varje rad i filen representerar ett delproblem.

2.4 Github

Vi har kontinuerligt använt oss av Github för att smidigt dela kod med varandra. I vår förvaringsplats har vi delat upp filerna i olika mappar. Så att skripten ligger i respektive bash-/pythonmapp, resultaten ligger i resultatmappen o.s.v. De viktigaste anledningarna till användandet av git/github var att se till att alla använder den senaste versionen av programmet, kunna göra nya grenar för att testa idéer, samt tillåta parallell utveckling av projektet.

2.5 Unixverktyg

Bash används dels för att koppla ihop alla delscript till ett huvudscript, "main.sh", samt i vissa delscript som löser delar av bortsorteringen av contiger med fler än ett visst antal grannar, exempelvis scriptet "over_n_neighbours.sh" som tar indata och ger en fil med de contiger som har över n överlappningar.

2.6 Python

Python används dels för att ta bort contiger med fler än n överlappningar i scriptet "scripts/python/social_contig_remover.py", dels för att sedan dela upp resterande contiger i delproblem i scriptet "scripts/python/read_and_divide.py".

2.7 Algoritmer

När vi delar upp grafen i delgrafer använder vi oss av bredden-först-sökning för att att hitta de sammanhängande delgraferna.

2.8 Komplexitet

Tidskomplexiteten för python funktionerna

__init__.py

- Parse - Eftersom denna funktion måste gå igenom n rader i indata, samt använder sig av `str.split()`, som är $O(m)$ på en sträng av längd m. Hela `Graph.parse()` blir alltså $O(nm)$.
- Connect - Eftersom `self._nodes`, och grannmängderna sparas som hashtabeller, blir `in` en $O(1)$ operation. Ty detta, samt tilldelning, är det enda vi gör i denna funktion blir hela funktionen $O(1)$.
- Remove - Som tidigare nämnt sparas `self._nodes` som en hashtabell, blir `in` en $O(1)$ operation. Dock behöver vi även iterera över hörn-som-skatas-borts grannar, vilka i värsta är $|V|$ stycken (där V är hörnmängden för grafen). Komplexiteten för `self.remove()` blir alltså $O(|V|)$.
- Subgraph - Detta är egentligen bara en bfs-algoritm. I värsta fall är den $O(|v| + n)$, eftersom vi kommer behöva iterera över varje hörn och kant om hela grafen är sammanhängande.
- csg_ify - Den börjar med att skapa ett dictionary, med nycklarna från `self._nodes` men med 1 som värde. Detta är en $O(|V|)$ operation. Efter det gör vi bfs på hela grafen, men tar bort hörn från grafen i takt med att de upptäcks. Bfs är som bekant $O(|V| + n)$, och med tillägget att vi kör `self.remove()` på varje hörn (som är en $O(|V|)$ operation) blir alltså `self.csg_ify()` $O(|V|^2 + n)$.

  Om vi betecknar platsen ett hörns namn tar i minnet som en minnesenhet blir minneskomplexiteten för grafklassen i värsta fall $O(|V|^2)$. Ty om varje hörn i grafen kommunicerar med varje annan nod i grafen kommer grannmängden till de $|V|$ hörnen i grafen vara $O(|V|)$ stora, och alltså blir den sammanlagda minneskomplexiteten $O(|V|^2)$. Den enda metoden till klassen som gör något speciellt minnesmässigt är `Graph.csg_ify()`, som utöver grafen sparar ett dictionary med alla hörnnamn, samt en deque med hörnnamnen. Både dessa är $O(|V|)$ i minnet, men eftersom grafen redan är $O(|V|^2)$ gör detta ingen påverkan på minneskomplexiteten.

  Bashscripts

- Over_n_neighbours - Alla delar är $O(n)$ förutom sorteringen som görs genom unixverktyget 'sort' som är en parallelliserad version av mergesort, där mergesort har komplexitet O(n log n). Sammantaget har `over_n_neighbours` tidskomplexiteten O(n log n).
- create_new_indata - kallar funktionen `social_contig_remover.py` som är $O(n)$ och kallar split som också har komplexiteten $O(n)$. Sammantaget har alltså `create_new_indata` tidskomplexiteten $O(n)$.

Bash- och pythonskripten som förbereder den nya indatafilen har alla $O(n)$ i minneskomplixtet. Detta beror på att de skapar delvisa nya kopior av indata.

3 Resultat

Nedan respresenteras resultatet för antalet delgrafer samt snittet för antalet contiger/delgraf. Detta är givetvis olika beroende på vad man definierar som en sociala nod.

Tabell för olika gränser av sociala noder			
Gränser på antal överlappningar	Antal delgrafer	Antalet kontiger	Medel antalet contiger per graf
10	1498702	8183658	5.46
15	1034242	9720297	9.40
20	758105	10376863	13.69
25	567308	10903177	19.22

Som förväntat så erhåller man fler delgrafer desto lägre gränsen är på antalet överlappningar.