Sammanfattning

I denna labb undersöktes en övärld med ett nätverk av delvis sammanlänkade städer. En av huvuduppgifterna var att jämföra prestandan hos olika algoritmer för att hitta den kortaste, respektive billigaste vägen mellan två städer i övärlden. Resultaten visade att algoritmen *billigaste vägen*, en implementering av Dijkstra's algoritm, presterade bättre än algoritmen *bästa-först* i 3 av 5 fall, när det gällde att hitta den lägsta kostnaden mellan två städer. Den största skillnaden i pris för en specifik väg mellan de båda algoritmerna var 57 sestertier. *Bredden-först* genererade generellt kortare vägar än *billigaste-vägen*. Den kortaste vägen mellan två städer hittades av den sistnämnda algoritmen och innefattade 6 st. städer; sträckan var mellan städerna *Veritas* och *Pineland.*

Reflektioner

I den här labben har jag bl.a. lärt mig att:

- Implementera en prioritetskö med hjälp av en min-heap
- Förstå bättre hur en heap är konstruerad
- Använda nya bibliotek och funktioner i Python såsom random & pickle
- Spara data & variabler till en extern fil som sedan kan användas av ett annat program
- Använda ascii-värden för att generera Emojis
- Lagra & extrahera data ur listor i listor
- Använda jämförelser av typen ”if x in y” för tuples.

Formalia

Namn: Fabian Revilla
Personnummer: 19950308-7471
Kurs: DD1321

Uppgiftsbeskrivning

Du ska implementera några algoritmer för att undersöka en övärld.
Filen Fabian.py innehåller en klass MapEdges som definierar en eller flera grafer av förbundna städer. I klassen finns några metoder som du ska använda för att lösa uppgiften.

- def getCityName(self, idx): - returnerar stadsnamn givet index
- def getCityCoord(self, idx): - returnerar kartkoordinaten för en stad givet index. Kartan består av en sträng med radbrytningar. Första raden så är y=0, andra y=1 o.s.v.
- def getNumberOfCities(self): - totalt antal städer i övärlden
- def getNeighborsTo(self, city): - returnerar en lista av grannar givet ett stadsindex
- def hasEdge(self, city1, city2): - returnerar om det finns en direktförbindelse
- def getCostBetween(self, city1, city2): - returnerar kostnaden att ta sig från ena staden till den andra

1 Antal öar

Uppgiftsbeskrivning

Konstruera en algoritm för att ta reda hur många öar som finns med hjälp av metoderna ovan. Beskriv din algoritm utförligt samt hur du kommit fram till algoritmen i rapporten. Var noga med att inte specialisera din algoritm till det data du fått, algoritmen ska fungera generellt med metoderna ovan. Klistra in en körning med lämpliga beskrivande utskrifter.

Kort beskrivning av algoritmen och hur den implementerades

I denna algoritm används noder för att representera de olika öarna. Funktionen getNeighborsTo() används för att generera en nods grannstäder & getNumberOfCities() används för att få ut det totala antalet städer. Algoritmen utvecklades genom att analysera hur klassen MapEdges() var uppbyggd för att första hur data över städerna lagrades i den och medlemsfunktionerna kunde användas för att extrahera nödvändig data såsom totalt antal städer och grannstäder. För att utforma algoritmen behövdes det först bestämmas vad som definierar en ö och vilka antagandet som skulle gälla. Till exempel gjordes definitionen att när alla möjliga vägar inom ett nätverk av städer har besökts, så har en ö hittats. Vidare bygger algoritmen på antagandet att alla städer har minst en granne och två godtyckliga städer från två olika öar inte har en direktförbindelse.

Algoritmens struktur:

1. Skapa en kö & en lista för obesökta noder(där en nod representerar en stad)
2. Om alla noder besökts – bryt, annars gå till steg 3
3. Gå till nästa obesökta nod
4. Om noden inte har besökts:
    a. Räkna upp ö-variabeln med 1
    b. Lägg in noden i kön
    c. Så länge kön inte är tom:
        i. Plocka ut första noden ur kön
        ii. Generera alla obesökta barn till noden m.h.a. getNeighborsTo()
        iii. För varje obesökt barn:
            1. Lägg till barnet i kön
            2. Lägg till barnet i listan med besökta noder
            3. När alla obesökta barn har gåtts igenom – gå tillbaks till steg 2

Utskrift

```
FABs-MBP-5:alabb fabian$ python3 1.py
den 1:a ön har 81 städer
den 2:a ön har 25 städer
den 3:a ön har 26 städer
den 4:a ön har 52 städer
den 5:a ön har 25 städer
Totalt finns det 5 öar i övärlden
```

Figur 1. Antalet städer i de olika öarna

2 En ö-karta och dess koordinater

Uppgiftsbeskrivning

En av öarna är utritad och finns även given i filen karta.py som en lång sträng island_map med radbrytningar. Förutom radbrytningar består kartan av minus, punkter, mellanslag samt flera höga (> 2000) utf-8 tecken som representerar städer. Skriv ett program som skriver ut stadsnamnen och dess koordinater på denna ö. Koordinater ges som (x, y) där y räknas uppifrån och första staden påträffas på y=2 (under den streckade linjen). Observera att städer på olika öar kan ha samma koordinater. Beskriv din algoritm kortfattat i rapporten.

Algoritmbeskrivning och implementation

Uppgiften löstes genom att stega igenom kartan som är en lång textsträng, tecken för tecken. För varje ny tecken beräknas ett nytt koordinatpar (x,y) och för varje tecken som har ett ASCII-värde över 2000 koordinaterna ned i en lista. Mer detaljerad beskrivning av algoritmen följer nedan:

1. Initiera två koordinatvariabler, x & y och en lista *L* för lagring av koordinater
2. För varje tecken i strängen *island_map*:
    a. Om tecknet är en radbrytning(”\n”)
        i. Gå till nästa rad; y = y + 1
        ii. Nollställ x-variabeln
    b. Om tecknet har ett utf-8 värde över 2000
        i. Spara de aktuella koordinaterna i *L*
    c. Gå vidare till nästa tecken i strängen; x = x + 1

Utskrift

Figur 2. Städernas koordinater

3 Bredden först

Uppgiftsbeskrivning

Utöka *MapEdges* med en medlemsfunktion som givet ett stadsnamn returnerar dess index. Vad som händer om stadsnamnet inte finns bestämmer du själv.
Skriv en bredden-först algoritm som tar reda på kortaste vägen, mätt i antal städer på vägen, på sträckorna nedan. Algoritmen ska även räkna ut kostnaden att gå denna kortaste väg. Om det inte finns en väg ska algoritmen rapportera detta (och inte krascha). Hur många noder finns i kön när algoritmen kört färdigt? Beskriv din algoritm i rapporten.

1. Nearon -> Doveseal
2. Pella's Wish -> Snowmelt
3. Dimbar -> Firecrow
4. Thales -> Anfauglith
5. Veritas -> Pineland

Algoritmbeskrivning och implementation

I algoritmen representeras varje stad av en nod. Stadsdata såsom namn, koordinater och grannar hämtas från klassen *MapEdges*. Algoritmen implementerar bredden-först sökning för att hitta den kortaste vägen mellan två städer. Metodens olika steg beskrivs i detalj nedan.

1. Definiera en startnod
2. Lägg in startnoden i kön

Så länge kön inte är tom:

1. Plocka ut översta noden ur kön
2. Ifall noden är det sökta värdet – bryt. Annars gå vidare till steg 3
3. Generera alla obesökta barn(grannstäder) till noden
4. Lägg in alla obesökta barn i kön
5. Gå tillbaks till steg ett

Resultat

Väg	Kostnad	Antal städer	Antal noder i kön
1. Nearon -> Doveseal	77	11	0
2. Pella's Wish -> Snowmelt	84	6	1
3. Dimbar -> Firecrow	151	8	2
4. Thales -> Anfauglith	165	8	2
5. Veritas -> Pineland	84	6	2

Tabell 1. Resultat av bredden-först algoritmen

4 Bästa först

Uppgiftsbeskrivning

Byt ut kön i bredden-förstsökningen mot en prioritetskö och låt barnen från en föräldranod läggas in i prioritetsordning. Använd getCostBetween för att ta reda på kostnaden att gå mellan städerna (högre siffra är dyrare). Vad blir (den billigaste) totalkostnaden för rutterna ovan? Blev det någon skillnad i väg? Hur många noder innehåller prioritetskön när algoritmen kört färdigt?

Beskrivning av algoritm

I denna uppgift används en liknande algoritm som i uppgift 3. Skillnaden är att kön är utbytt mot en prioritetskö och att barnen läggs in i prioritetsordning där barnet med lägst prioritet först läggs in i kön.

Resultat

Väg	Kostnad	Antal städer	Antal element i prio-kön
Nearon -> Loveseal	67 sestertier	11 st	3
Pella's Wish -> Snowmelt	31 sestertier	7 st	2
Dimbar -> Firecrow	85 sestertier	9 st	0
Thales -> Anfauglith	29 sestertier	9 st	5
Veritas -> Pineland	22 sestertier	7 st	3

Tabell 2. Bästa först-algoritmens resultat för de 5 olika sträckorna

Billigast totalkostnad: 22 sestertier

Algoritmen väljer en annan väg än bästa först i 4 av 5 fall.

5 Billigaste vägen

Uppgiftsbeskrivning

Implementera en girig bästaförst sökning enligt föreläsningsanteckningarna. I princip ska du använda en prioritetskö där du redan från början lägger in samtliga städer med absurt hög kostnad. Där du i de föregående algoritmerna lade in nya noder ska du istället omprioritera noder i prioritetskön. Omprioritera antingen genom att, som i föreläsningsanteckningarna, plocka ur och stoppa in igen eller omprioritera direkt i prioritetskön.
Mät kostnaderna för sträckorna ovan. Hur lång blir vägen? När är algoritmen klar? Hur många noder har behandlats? Hur många noder innehåller prioritetskön när algoritmen är klar? Hur har du implementerat algoritmen? Beskriv din algoritm i rapporten.

Beskrivning algoritm och implementation

I detta fall representeras varje stad av en nod. Noderna innehåller stadsdata såsom namn, prioritet, koordinater och namn. Algoritmen bygger på att den nod som har lägst prioritet plockas ut först för att generera grannstäder(barn till noden). Algoritmen implementerar en heap i form av en min-prioritetskö. I detta fall betyder det att noden med minst prioritet alltid ligger högst upp i heapen och alltid blir det element som plockas ut först. En nods prioritet beräknas genom kostnaden att gå till staden och kan uppdateras under programmets körning i takt med att billigare vägar hittas. Algoritmen fortsätter på detta sätt tills alla noder har behandlats och prioritetskön är tom. För att endast obesökta noder ska genereras sparas redan besökta noder ned i en lista *visited.* Algoritmen bygger på Dijkstra's algoritm och inleds på följande sätt:

1. Sätt alla noders prioritetsvärde till ett maxvärde
2. Sätt startnodens prioritet till 0.
3. Stoppa in startnoden i en min-prioritetskö.

Den fortsatta algoritmen för att hitta billigaste vägen blir sedan:

1. Plocka ut en nod *p* från prioritetskön.
2. Generera alla obesökta barn till noden m.h.a. funktionen *getNeighborsTo()*
3. Undersök noden p:s barn ett och ett
    1. Räkna ut kostnaden att gå från p till barnet och addera med p:s prioritet
    2. Om den sammanlagda kostnaden är billigare än barnets nuvarande prioritet
        1. Sätt om barnets prioritet till den nya kostnaden
        2. Sätt föräldrapekaren att peka på p
        3. Omprioritera barnet genom att
            1. plocka bort barnet ur prioritetskön
            2. stoppa in barnet i prioritetskön
4. Upprepa från 1. tills prioritetskön är tom (det går att avbryta tidigare när bättre vägar inte finns)
5. Skriv ut billigaste vägen genom att följa föräldrapekarna från slutnoden till startnoden.

Resultat

Väg	Kostnad	Antal städer	Antal element i priokön
Nearon -> Loveseal	66 sestertier	12 st	0
Pella's Wish -> Snowmelt	27 sestertier	7 st	0
Dimbar -> Firecrow	28 sestertier	9 st	0
Thales -> Anfauglith	30 sestertier	10 st	0
Veritas -> Pineland	22 sestertier	7 st	0

Tabell 3. Billigaste vägens resultat för de 5 olika sträckorna

6 Jämför bästa först och billigaste vägen

Uppgiftsbeskrivning

När får de båda algoritmerna samma kostnad och när får de olika kostnad? Är någon mer effektiv än någon annan om det inte finns en väg? Gör en analys. Använd ön nedan eller konstruera en egen ö med egna tillrättalagda data så att du kan illustrera (rita) och beskriva med vägexempel när och varför algoritmerna får dels lika, dels olika vägar

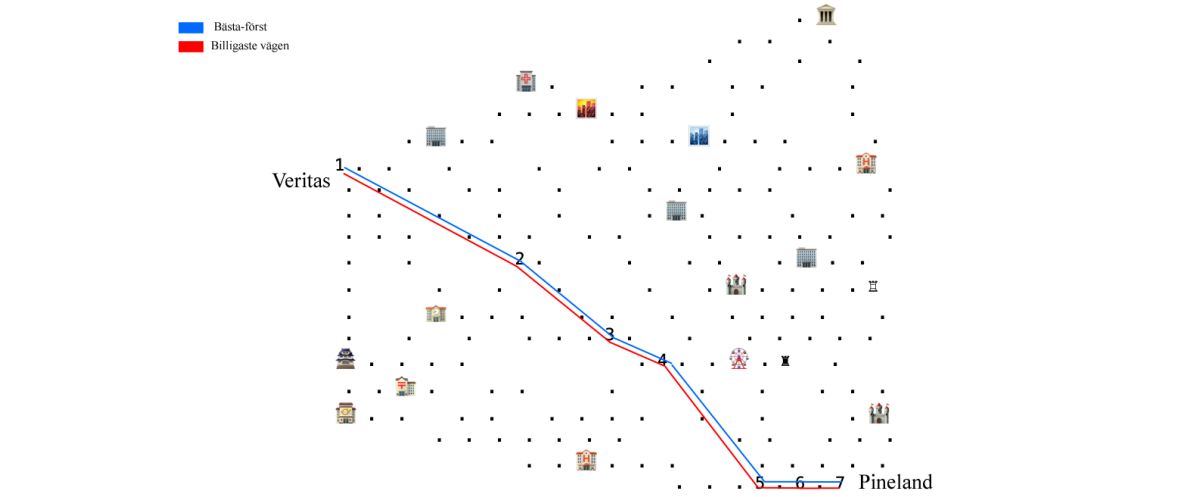
Analys

En väsentlig skillnad mellan de två algoritmerna, är att en nod alltid behåller sitt prioritetsvärde i bästa först-algoritmen. Detta skiljer sig från billigaste vägen, där en nods prioritetsvärde kan uppdateras under programmets gång, i takt med att bättre väger hittas. I praktiken leder det till att billigaste först ofta genererar billigare vägar, när bättre alternativ finns. Detta speglas även i resultaten, där billigaste vägen genererade samma eller lägre kostnad än bästa först i 4 av de 5 fallen som undersöktes.

Om ingen väg finns mellan två städer kommer bästa först komma fram till det svaret snabbare, eftersom noder som redan har fått ett prioritetsvärde aldrig behandlas igen. I billigaste vägen kan en nod generera barn fler gånger om prioritetsvärdet ändras, vilket ökar tidsåtgången för att gå igenom alla noder. I båda fallen måste dock alla noder behandlas, men bästa först skulle teoretiskt sett göra det snabbare.

Vägarna sammanfaller

I sällsynta fall händer det att den billigaste vägen sammanfaller med den genererad från bästa först-algoritmen. Detta var fallet för rutten Veritas -> Pineland, vilket illustreras i Figur 3 Detta kan antingen bero på att det är den enda möjliga vägen mellan städerna, eller att den först hittade vägen i bästa först-algoritmen helt enkelt av slump också är den billigaste. I Tabell 3 går det att tyda att städerna hade samma prioritetsvärde i de båda algoritmerna, vilket visar att inga bättre vägar hittades för någon av städerna i bästa först-algoritmen. Värt att notera är att vägen innehåller relativt få stopp(7 st) med låga prioritetsvärden(0-22), vilken ökar sannolikheten att vägen ska tas av bästa först algoritmen. Dessutom innehåller ön få städer vilket minskar sannolikheten att alternativa billigare vägar ska genereras.



Figur 3. Bild som visar algoritmernas val av väg mellan städerna Veritas & Pineland

Stad	Priovärde - Bästa först	Priovärde - Billigaste vägen
1. Veritas	0	0
2. Pony Run	2	2
3. Mal Yaska	4	4
4. Snowforge	9	9
5. Willowdale	11	11
6. Suilwen	15	15
7. Pineland	22	22

Tabell 4. Städernas olika prioritetsvärden under körning för de båda algoritmerna

Olika vägar

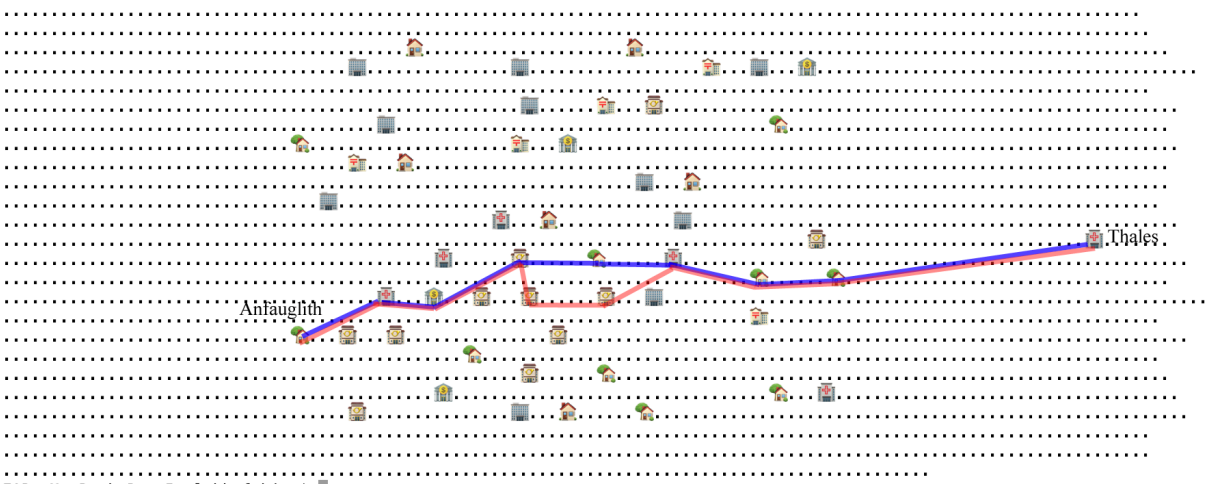
I Figur 4 nedan visas de båda algoritmernas vägar för rutten Thales -> Anfauglith. Algoritmerna följer initialt samma väger men vid det femte stoppet väljs olika vägar. Bästa först väljer att gå till staden Gandahar, medan billigaste först väljer staden Battlecliff. I Tabell 4 och 5 nedan kan vi se att staden Battlecliff har lägre prioritetsvärde än Gandhar, vilket förklarar varför billigaste vägen-algoritmen valde att gå till den förstnämnda.

Stad	Prioritetsvärde
1. Thales	0
2. Bullmar	10
3. Angband	13
4. Eden's Gulf	15
5. Gandahar	18
6. Pinnella Pass	21
7. Palanthas	24
8. Lamorra	26
9. Anfauglith	29

Tabell 5. Städernas prioritetsvärden vid användning av bästa först

Stad	Prioritetsvärde
1. Thales	0
2. Bullmar	10
3. Angband	13
4. Eden's Gulf	15
5. Battlecliff	17
6. Greenglade	20
7. Pinnella Pass	22
8. Palanthas	25
9. Lamorra	27
10. Anfauglith	30

Tabell 6. Städernas prioritetsvärden vid användning av Billigaste först-algoritmen



Figur 4. Algoritmernas val av väg mellan städerna Thales & Anlauglith