Løsningsforslag NIO 2022 - runde 2

Kryptering

For hver bokstav i den krypterte meldingen kan den dekrypteres ved å finne plasseringen til den bokstaven i nøkkelen, og hente ut bokstaven på plassen én til venstre i nøkkelen. Husk at den aller første bokstaven i nøkkelen ikke har noe til venstre for seg, så da må du ta nøkkelens bakerste bokstav i stedet.

Virussmitte

Gruppe 1 (20 poeng)

Det seneste noen kan bli smittet er dag 1000, og 2000 dager senere vil alle være friske, så det er mulig å simulere alle 3000 dager. Hver dag er det 100 mennesker å simulere, og opptil 100 hendelser. Det vil gå raskt nok å sjekke hver person og hvert møte hver dag. For hver dag som går kan du telle antall syke mennesker på den dagen, og huske det høyeste slike tallet du har sett.

Gruppe 2 (23 poeng)

Folk er syke kun på dagen etter de blir smittet. Det er mulig å simulere alle dagene, eller kun simulere dager med møter. Ved å sortere møtene etter dag, kan man gå gjennom listen med møter i kronologisk rekkefølge, og for hvert møte, sjekke at person A ble smittet for første gang i går, og person B aldri har blitt smittet. I såfall blir B smittet den dagen. Det største antall syke på én dag vil være det samme som det største antall som blir smittet på én dag, siden sykdom er nøyaktig én dag forskjøvet fra smitte. (Hvis 12 stykker blir smittet på dag 2, vil det være nøyaktig de 12 som er syke på dag 3)

Gruppe 3 (57 poeng)

Folk kan nå ha lang inkubasjonstid, og lang sykdomstid. Det er ikke lenger mulig å simulere hver dag noen er syke. Det vi kan gjøre i stedet er å bruke en prioritetskø, og legge inn alle møtene som hendelser. Simuleringen går gjennom prioritetskøen i kronologisk rekkefølge. Når noen møtes, sjekker vi om én av dem er syke, og den andre aldri har blitt smittet. I såfall skal den andre personen markeres som smittet, og to nye hendelser legges inn i køen. Om I dager skal

personen bli syk, og om *I*+*S* dager skal personen bli frisk. Vi fortsetter å simulere gjennom prioritetskøen, og teller maks antall syke på én gang. Merk at hvis person A blir frisk på samme dag som peron B blir syk, kan vi ikke si at de to har vært syke på samme dag.

En annen måte å gjøre det på, som ikke bruker noen kø, er å huske smittetidspunkt for alle personer, og deretter gå gjennom møtene i kronologisk rekkefølge (sorter møter etter dag). Hvis en person ble smittet på dag x, vet vi at vedkommende er syk i intervallet [x+I, x+I+S-1]. Dermed kan vi for alle møter enkelt sjekke om én av personene er syke, basert på tidspunktet de eventuelt ble smittet. Hvis den éne personen er syk, og den andre aldri har blitt smittet, har den andre personen blitt smittet i det møtet, på den dagen. Etter vi har gjort dette for alle treffene, så vet vi for hver person i hvilket intervall den personen var syk på (dersom personen ble syk i det hele tatt). Vi kan sortere disse intervallene etter starttidspunkt og gå gjennom de, for å finne ut av på hvilket tidspunkt det var flest syke på.

Bysykkel

Gruppe 1 (15 poeng)

Siden alle høydene er 0 så vet vi at reiseveien mellom to nabonoder er 50 hvis man går og 20 hvis man sykler. Det vil dermed aldri lønne seg å levere fra seg en sykkel på en stasjon, gå litt, og så hente en ny sykkel på en annen stasjon, ettersom det uansett vil være raskere å sykle den avstanden mellom de to stasjonene. Altså vil man enten ikke bruke sykkel i det hele tatt, eller bruke sykkel på nøyaktig ett strekke. Hvis man bruker en sykkel så ser man at det vil alltid lønne seg å plukke den opp fra den nordligste stasjonen med en sykkel i en eller annen kolonne, for så å parkere den på den sørligste stasjonen i en eller annen kolonne. Det er dermed maks 250×250 mulige par av stasjoner vi trenger å teste. Tiden for et slikt par av stasjoner vil være 50×manhattan_avstand(start, stasjon1) + 20×manhattan_avstand(stasjon1, stasjon2) + 50×manhattan_avstand(stasjon2, slutt). manhattan_avstand mellom to punkter (x1,y1) og (x2,y2) er gitt ved |x1-x2| + |y1-y2| , hvor | | betegner absoluttverdi.

Gruppe 2 (30 poeng)

Enten bruker vi ikke sykkel i det hele tatt, og får en avstand på 50×2×(N-1), eller så må vi sykle hele veien. Tiden det tar å sykle vil være avhengig av hvilken vei man tar, så her må man bruke

f.eks. https://no.wikipedia.org/wiki/Dijkstras_algoritme. Vi betrakter hver rute i kartet som en node, som har kanter til sine nabonoder. Lengden / vekten av en slik kant avhenger av høydeforskjellen mellom rutene, slik gitt i oppgaven. Når Djikstras algoritme har gitt oss den raskeste måten å sykle på så trenger vi bare å sammenligne det med tiden det tar å gå, og skrive ut det minste av de to.

Gruppe 3 (55 poeng)

Her kan vi også bruke Dijsktra, men det er litt vanskeligere siden det her kanskje vil være lønnsomt å gå deler av veien. Når vi betrakter tilstanden vi kan befinne oss i, så ser vi at den er gitt av hvilken rute vi befinner oss på, og hvorvidt vi har en sykkel eller ikke. Vi får dermed $2 \times N^2$ noder i grafen vår, som vi kan betegne med (0, 0, tilstand), hvor x og y er koordinater og tilstand er 0 hvis man går eller 1 hvis man har en sykkel. I tillegg til kantene som representerer å gå eller sykle mellom nabonoder så får vi også kanter som represeterer overgangene fra en tilstand til en annen. Hvis det er en sykkelstasjon av type F eller * på posisjon (x,y), så lager vi en kant fra (x, y, 0) til (x, y, 1) med lengde 0 i grafen vår. Hvis det er en sykkelstasjon av type T eller * på posisjon (x,y), så lager vi en kant fra (x, y, 1) til (x, y, 0) med lengde 0.
Raskeste reisevei blir da gitt av Dijkstra som avstanden fra (0, 0, 0) til (N-1, N-1, 0).

Togtransport

Gruppe 1 (7 poeng)

Siden man bare kan bygge 1 togskinne så kan man prøve alle N-1 mulighetene. Merk at det kan være at det er flere ordre som gjelder den samme strekningen, så svaret er ikke nødvendigvis det samme som den mest kostbare orderen med lengde 1.

Gruppe 2 (12 poeng)

Siden alle ordrene har utgangspunkt i en av endestasjonene, så vil det alltid lønne seg å bygge V togskinner fra venstre og T-V togskinner fra høyre. Man kan prøve alle mulige verdier av V fra 0 til T-V for å se hvilken som gir best løsning.

Gruppe 3 (15 poeng)

Siden N er såpass liten, så kan vi prøve ut alle mulige måter å bygge maksimalt T togskinner på, og se hvilke verdier vi får ut av de.

Gruppe 4 (15 poeng)

Litt likt som med gruppe 3, men her er det K som er veldig liten. Vi kan prøve ut alle delmengder av ordrene (det finnes 2^K slike delmengder), og sjekke om det er mulig å tilfredsstille alle disse ordrene. Det optimale svaret er da den totale verdien av den mest verdifulle delmengden som det var mulig å tilfredsstille.

Gruppe 5+6 (26+27 poeng)

Her må vi finne en lur måte å dele opp problemet på. Vi kan ta for oss strekningene fra venstre til høyre. For hver slik strekning så har vi kun to muligheter - enten bygger vi en togskinne der eller ikke. (Hvis vi er tomme for togskinner så har vi ikke noe valg i det hele tatt). Men hva som lønner seg vil avhenge av hvilke togskinner vi allerede har bygd, og hvor mange togskinner som vi har igjen. Antall togskinner som vi har igjen vil alltid være et heltall mindre enn N, så der er det ikke så mange muligheter, men det kan være veldig mange måter å ha valgt ut hvilke togskinner som vi allerede har bygd på. Heldigvis er det kun relevant hvor langt vi har sammenhengende bygd rett til venstre for strekningen vi ser på, noe som også kan representeres som et ganske lite heltall!

La `optimal[posisjon][start_intervall][antall_skinner]` uttrykke hvor mye penger man kan tjene på å fortsette å utvide seg til høyre fra den gitte posisjonen dersom man har sammenhengende togskinner mellom `start_intervall` og `posisjon`, og `antall_skinner` med skinner til rådighet. (Vi inkluderer altså ikke pengene man allerede har tjent på skinnene som man allerede har bygd.) Vi får da formelen

```
optimal[posisjon][start_intervall][antall_skinner] =
    0 dersom posisjon = N-1 (ikke mulig å bygge videre)
    0 dersom antal_skinner = 0 (ingen skinner igjen å bygge med)
    max(optimal[posisjon+1][posisjon+1][antall_skinner],
        optimal[posisjon+1][start_intervall][antall_skinner-1] +
```

`direkte_fortjeneste(start_intervall, posisjon+1)`

`)` i alle andre tilfeller

hvor `direkte_fortjeneste(u,v)` er hvor mye penger man tjener på å utvide intervallet med sammenhengende togskinner fra å være [u,v-1] til å være [u,v]. Dette kan beregnes som summen av verdiene på alle ordrene hvor $\texttt{min}(A_i, B_i) \geq u$ og $\texttt{max}(A_i, B_i) = v$.

Dersom vi itererer over alle mulige verdier av `posisjon`, `start_intervall` og `antall_skinner` (og bare passer på at når vi itererer over `posisjon` så går vi fra N-1 og ned til 0), så kan vi fylle ut alle verdiene i denne tabellen veldig raskt. Det endelige svaret finner vi da i `optimal[0][0][T]`.

Denne teknikken for å dele opp et problem i mindre problemer kalles for *dynamisk programmering* (DP) og er en veldig verdifull teknikk for å løse mange vidt forskjellige problemer.

Den eneste forskjellen mellom subtask 5 og 6 er at man må være litt mer forsiktig med hvor mange ganger man kaller `direkte_fortjeneste` funksjonen, og heller forhåndslagre de verdiene den gir, slik at man slipper å regne de ut for det samme intervallet flere ganger.

Datavirus

Gruppe 1 (11 poeng)

Ingen av datamaskinene er koblet til mer enn 2 andre datamaskiner. Siden vi vet at alle datamaskinene kan nå hverandre i utgangspunktet, betyr dette at vi enten har en kjede med datamaskiner (M=N-1), eller en sirkel med datamaskiner (M=N). Hvis vi har en sirkel vil det ikke ha noe å si hvilken datamaskin som infiseres, alle de andre vil fremdeles kunne nå hverandre, og knytningsgraden blir (N-1)(N-2)/2. Dersom nettverket vårt er en kjede bør vi infisere den maskinen som er mest mulig i midten. Da får vi 2 grupper med knytningsgrad floor((N-1)/2)*floor((N-1)/2-1)/2 og ceil((N-1)/2)*ceil((N-1)/2-1)/2.

Gruppe 2 (18 poeng)

N <= 1000.

Gruppe 3 (27 poeng)

M = N-1. Dette betyr at det opprinnelige nettverket danner et tre. Det finnes altså nøyaktig én vei fra A til B, for hvert par av datamaskiner (A,B). Hvis den veien går via datamaskin C, og C blir infisert, vil A og B ikke lenger være knyttet.

Velg ut en datamaskin som rot i treet, og gjør en dybde-først-traversering, slik at hver node får 0, én eller flere sub-trær. Å infisere en datamaskin med 3 subtrær vil resultere i 4 uavhengige nett, siden hvert subtre blir adskilt fra treets rot, og adskilt fra hverandre. Da vil den totale knytningsgraden oppnådd ved å infisere datamaskinen være lik summen av knytningsgrader internt i hvert subtre, pluss knytningsgraden i resten av treet (over den infiserte datamaskinen). Merk at hvis den infiserte datamaskinen er roten i treet, vil den ikke ha noe nettverk over seg, og man får da kun sub-trær.

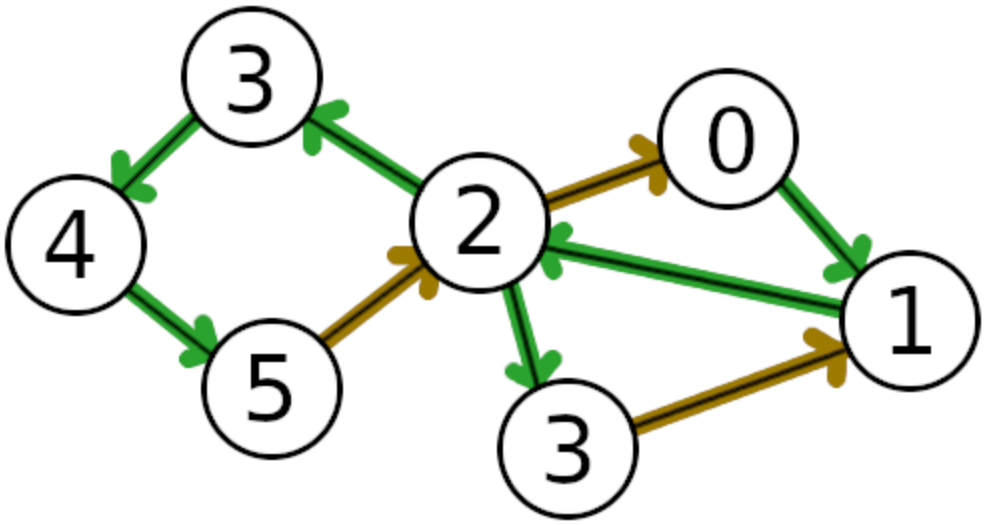
NB: Her kan det være litt ekstra vanskelig å skrive i python, siden en rekursiv dybde-først-traversering kan føre til rekursjon så dyp at python-implementasjonen krasjer. (Tenk deg at alle de 100 000 datamaskinene er koblet i én lang kjede.) I C++ kan dette gå greit, så lenge den rekursive funksjonen ikke har mange lokale variabler.

Gruppe 4 (44 poeng)

Løsningen ligner fremdeles mye på gruppe 3, men vi har nå ingen garantier om at nettverket ser ut som et tre. Når vi gjør dybde-først-traverseringen, vil vi derfor få to typer kanter. Tre-kanter er de kantene vi traverserer nedover som vanlig dybde-først. Disse tre-kentene er rettet, og etablerer en forelder-barn-relasjon mellom nodene. Når vi kun ser på tre-kanter, ser grafen ut som et tre.

Hvis noden vi utvider derimot har en kant i den opprinnelige grafen som fører til en node som allerede er besøkt lenger oppe i treet, kan vi ikke ta med den kanten i treet, siden det da ikke lenger ville vært et tre. Siden kanten peker tilbake oppover i treet kaller vi den en bakover-kant. Også den er rettet, og vil alltid peke til en direkte eller indirekte forelder i treet.

Vi definerer dybde i treet, slik at roten har dybde 0, og alle andre noder har dybde lik avstand fra roten, **målt i tre-kanter**.



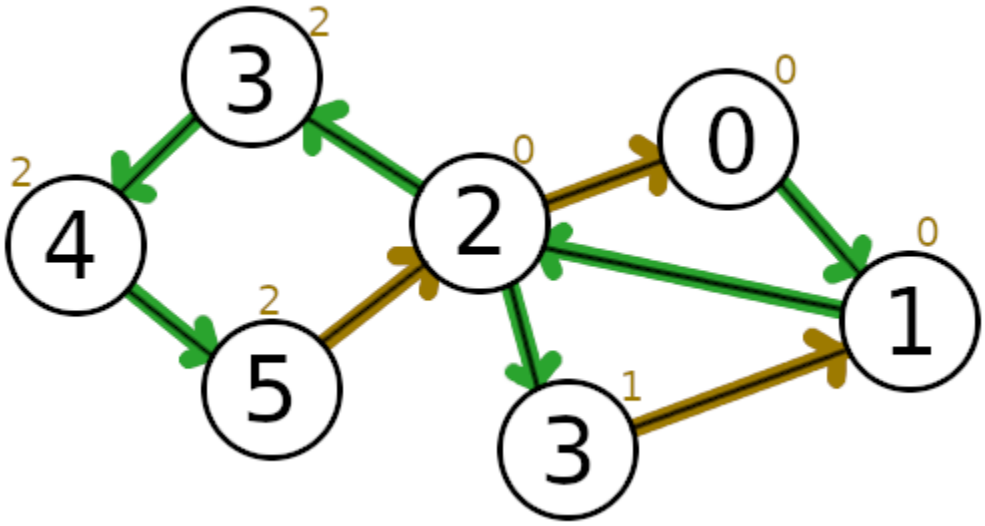
Bildet viser en mulig DFS-traversering av en graf, der tre-kanter er blitt farget grønne, og bakover-kanter er brune. Tallet inni hver node er nodens dybde, så rot-noden er den med 0. Merk at valg av rotnode og rekkefølge på utforskning av kanter er gjort vilkårlig, så samme graf kan ha veldig mange mulige DFS-trær. Dette er bare én av mulighetene.

Når vi nå skal regne ut hvilken knytningsgrad vi oppnår av å fjerne en node, ser vi igjen på sub-trær til den noden. Kun de grønne kantene er med i treet, så noden med dybde 2 har nøyakig 2 sub-trær, og én forelder. Vi anser ikke de brune kantene som subtrær, enten de peker inn eller ut av noden vår. Dette betyr til gjengjeld at de sub-trærne vi har, kun er koblet sammen via noden vår. Hadde det vært mulig å komme seg til den éne noden med dybde 3 via den andre med dybde 3, kunne ikke begge hatt dybde 3!

Vi vet altså at de to sub-trærne vi har garantert ikke har noen kanter mellom hverandre i den opprinnelige grafen. Dessverre kan de fortsatt være koblet til det øvrige treet, over noden med dybde 2. Det ideelle er at et sub-tre ikke er koblet til noen noder høyere opp enn dybde 2, for isåfall vil det å fjerne noden med dybde 2, kutte sub-treet vekk fra roten. Da vil sub-treet være umulig å nå fra resten av treet, som fjerner knytningsgrad fra grafen.

For å raskt kunne vite hvor høyt opp et sub-tre har bakover-kant til, lar vi hver node huske den minste dybden man kan komme til ved å følge grønne kanter, etterfulgt av én bakoverkant.

Denne tilleggsinformasjonen lagres samtidig som DFS-treet bygges, ved at hver node tar den minste dybden hvert av sub-trærne kan nå, og eventuelle egne bakover-kanter ut av seg selv.



Se for eksempel at noden med dybde 2 har ett sub-tre som kun har koblinger til dybde 2 eller lavere, og ett sub-tre som har en kobling til dybde 1. Dette forteller oss at kun det første subtreet vil kobles vekk fra roten, dersom vi fjerner noden med dybde 2. Dette ser vi også at stemmer.

For å regne ut den endelige knyningsgraden, regner vi ut knytningsgraden til alle slike sub-trær, og knytningsgraden hoved-treet vil sitte igjen med når sub-trærne er kappet av. Ved å telle antall noder som kappes av i hvert fullstendig separerte sub-tre, er det lett å vite hvor mange noder rot-treet sitter igjen med.

For å finne den beste noden å fjerne, forsøker vi å fjerne alle. For hver node trenger vi kun å se på de direkte etterkommerne i DFS-treet, så total kjøretid for å bygge treet blir O(M), og for å sjekke infisering av hver node O(N). Totalt O(N+M).