Proboscidea Volcanium

Programmering II - Elixir

Kenan Dizdarevic

16 Februari, 2023

Inledning

Vi skall i denna uppgift lösa ett av problemen som *Advent of Code* erbjuder. Problemet vi skall lösa kommer från dag 16.

Detta problem går att lösa på ett otroligt smärtsamt vis, men det skall vi inte göra. För att undvika långa exekveringstider kommer vi att använda oss av *dynamisk programmering*. Det är en generell metod för att lösa kombinatoriska optimeringsproblem som dag 16 gladligen erbjuder.

För att lösa detta problem kommer vi behöva utföra flertal beräkningar. Vi kommer att dela upp problemet i mindre beståndsdelar som är enklare att lösa. Det som gör denna lösning dynamisk är att dessa beståndsdelar delar samma subproblem. Detta i sin tur innebär att vi kommer behöva utföra samma beräkningar flertal gånger. Istället för att utföra dem flera gånger kan vi memorera lösningarna och återanvända dem.

Implementering

Problemformulering

Vi är fast i en vulkan med flertal elefanter. Vi har endast 30 minuter på att ta oss ut med elefanterna innan vulkanen får ett utbrott, vi hinner inte ta oss ut där vi kom från. I vulkanen finns det ett nätverk som består av rör och tryckavlastningsventiler.

Varje ventil har en flödeshastighet samt tunnlar som vi kan nyttja för att ta oss mellan ventilerna. Det tar en minut för oss att öppna en ventil och det tar en minut att gå igenom en tunnel. Alla ventiler är stängda när tiden börjar. Vi skall beräkna hur mycket tryck vi kan släppa ut om vi tar den optimala vägen genom nätverket.

Minnesmodulen

Vi måste ha möjligheten att lägga till nya lösningar till minnet samt kolla upp om vi redan har löst ett problem tidigare. Detta implementeras i modu-

len `Memory` med hjälp av Elixirs egna `Map()` som har en bra tidskomplexitet. Sökningarna är ganska krävande, där av vill vi ha ett snabbt minne. Minnesmodulen har tre funktioner `new/0` som skapar ett nytt minne, `store/3` som lägger till en lösning och `lookup/2` som returnerar lösningen om den existerar. Minnesmodulen implementeras som följande:

```
def new() do %{} end

def store(key, value, memory) do
    Map.put(memory, key, value)
end

def lookup(key, memory) do
    Map.get(memory, key)
end
```

Minne

Minnet är relativt enkelt att implementera. Vi behöver hela tiden ha i åtanke hur vi bör nyttja det. Vi skall implementera funktionen `memory/8` som antingen returnerar den valda vägen och minnet om tiden är slut. Om vi redan har öppnat alla ventiler skall vi returnera totala trycket, vägen vi tog och minnet. Minnets huvudfunktion är att kolla om vi redan har gjort en specifik sökning. Om vi har gjort detta skall vi returnera den, annars får vi utföra en ny sökning och spara den i minnet. Det är viktigt att vi returnerar det nuvarande minnet i varje steg av exekveringen, eftersom vi alltid vill ha det senast uppdaterade minnet inom räckhåll. Detta implementeras i koden nedan:

```
def memory(valve, time, closed, open, rate, map, path, cache) do
    case Memory.lookup({valve, time, open}, cache) do
      nil ->
        {max, path, cache} = search(valve, time, closed,
                                    open, rate, map, path, cache)
        cache = Memory.store({valve, time, open}, {max, path}, cache)
        {max, path, cache}
      {max, path} ->
        {max, path, cache}
end
```

Koden är relativt enkel att implementera. Vi utnyttjar funktionen `lookup/2` som vi implementerade tidigare. Detta kan resultera i två fall, antingen finns lösningen i vårt minne eller så finns den inte. Om `lookup/2` returnerar `nil` skall vi utföra en ny sökning med de nya parametrarna. Sedan skall vi spara den nya lösningen och uppdatera det gamla minnet. Om `lookup/2`

returnerar en flödeshastighet och en väg skall vi endast returnerar det och vårt minne som inte behöver uppdateras i detta fall.

Våra nycklar till minnet är vilken ventil det är, hur mycket tid som återstår och om ventilen är öppen eller ej. Detta ger oss åtkomst till värdet som består av vad trycket är och vilken väg vi har valt.

Sökning

Vi har tyvärr ingen generell sökningsalgoritm som vi kan tillämpa, därav är minnet mycket viktigt för oss. Utan minnet hade vi fått vänta flera timmar tills en lösning har genererats.

Sökningen kommer att bestå av funktionen `search/8` som tar emot två listor av alla ventiler som är öppna och stängda. Den tar även emot minnet, vår karta, vilken väg vi har valt hittills, hur mycket tid som är kvar och vilken ventil vi är på.

Det första vi gör i funktionen är att ta reda på vad flödeshastigheten från valvet är samt vilka andra ventiler vi kan gå till. Vi kommer inte att ta hänsyn till de ventiler som inte har någon flödeshastighet. Det vi istället gör är att vi ökar längden mellan de ventiler som har bindningar som består av ventiler som inte har någon flödeshastighet.

Nästa steg i funktionen är att kolla om ventilen som vi är på är stängd. Detta görs med följande kod:

```
{currentMax, currentPath, currentCache} = if Enum.member?(closed, valve)
```

Nu kan det uppstå två olika scenarion, antingen är ventilen öppen eller stängd och vi skall behandla båda fall. Om ventilen är öppen kan vi stå kvar på samma plats, det är ingen mening att öppna en ventil som redan är öppen. Vi väntar på nästa skede där vi skall förflytta oss istället. Om ventilen är stängd skall vi öppna den. Detta gör vi genom att först ta bort den från listan som innehåller alla ventiler som är stängda. Sedan måste vi uppdatera vårt minne. Vi uppdaterar minnet med följande anrop:

```
memory(valve, time - 1, removed, Memory.insert(valve, open),
       rate + currentRate, map, [valve | path], cache)
```

Vi uppdaterar minnet genom att ta bort en minut, eftersom det tar en minut att öppna en ventil. Sedan lägger vi till ventilen som vi öppnade till listan som innehåller de öppna ventilerna. Vi ökar även vår nuvarande flödeshastighet genom att addera det vi hade innan med det som valvet bidrar med. Sedan utökar vi även vår väg som vi har tagit genom nätverket. Slutligen uppdaterar vi flödeshastigheten som kommer användas i en senare del av funktionen.

```
currentMax = currentMax + rate
{currentMax, currentPath, currentCache}
```

Nästa steg i funktionen är att använda oss av `Enum.reduce()` för att iterera över varje ventil i nätverket och uppdatera dem. För varje ventil i nätverket kommer funktionen att kolla om distansen till nästa ventil är mindre än den återstående tiden. Om vi hinner gå till nästa ventil innan tiden tar slut förflyttar vi oss till nästa nod och anropar `memory/8` för att uppdatera minnet. Koden för detta är följande:

```
fn({next, distance}, {currentMax, currentPath, cache}) ->
   case distance < time do
     true ->
       {newMax, newPath, cache} = memory(next, time - distance,
                                         closed, open, rate, map, path, cache)
```

Sedan kollar vi om vår nya flödeshastighet är högre än den föregående, om detta stämmer returnerar vi den nya flödeshastigheten. Koden presenteras nedan:

```
case newMax > currentMax do
  true ->
    {newMax, newPath, cache}
  false ->
    {currentMax, currentPath, cache}
```

Funktionen som söker kombinerat med minnet som vi implementerade leder till en avsevärd förbättring av exekveringstiden, den är fortfarande inte snabb. Optimeringsproblemem medför dessvärre flertal problem som måste lösas än enbart själva uppgiften.

Den maximala flödeshastigheten som vi kunde uppnå i vårt nätverk uppgick till 2253.

Kloka elefanter

I den andra delen av problemet har elefanterna blivit otroligt kloka. Det tar oss endast 4 minuter att lära en elefant att öppna korrekt ventil i korrekt ordning. Vår uppgift är att beräkna den maximala flödeshastighete som vi kan uppnå under 26 minuter.

Det första man tänker på är att dela ventilerna på det vis att elefanten får en halva och vi tar resten. Men detta kommer inte generera en optimal lösning, det kan finnas fall då den optimala vägen innebär att vi korsar varandras ventiler.

Initialt tänker vi att man kan nyttja *flertrådad programmering*. Man utför beräkningarna för de optimala vägarna för oss och elefanten. Till detta behöver vi ett minne. Nu står vi in för ett val. Skall vi ha ett gemensamt minne eller två olika? Vi kan ha två olika trådar som ständigt kommunicerar med varandra för att inte hamna i de fall där vi försöker öppna samma ventil. Det viktigaste i denna uppgift är koordination. Vi kan ständigt leta

efter optimala vägar samtidigt som vi kollar om vi öppnar samma valv. Om vi öppnar samma valv kan vi gå till ventilen innan och välja en annan väg.

Slutsats

Slutligen ser vi att dynamisk programmering är en otroligt kraftfull metod när vi skall lösa optimeringsproblem. Detta problem verkar invecklat till en början. Det finns inget mönster som vi har sett tidigare som vi kan applicera på nätverket. Detta medför att vi noggrant måste utveckla en funktion som uppfyller kraven.

Funktionen skulle inte klara sig på egen hand. Utan det dynamiska minnet skulle vi få vänta väldigt länge på en lösning. Nätverkets olika ventiler har samma subproblem vilket innebär att vi kan hitta lösningar och sedan spara dem i minnet. Sedan när vi är på samma plats tar vi hjälp av minnet och i bästa fall slipper vi utföra beräkningarna.

Optimeringsproblem medför tyvärr komplexare lösningar. Dessa lösningar är otroligt kreativa och kraftfulla vilket ger oss ett helt annat perspektiv på programmering.