TDDD86 – Laboration #2

1 september 2020

Syftet med den här laborationen är att öva användandet av standardbibliotekets containrar. Du kommer att använda vector, stack, queue, set och map. Eftersom det är svårt att hitta på ett enskilt problem som kräver alla dessa containrar är uppgiften tvådelad. Filerna du behöver för att komma igång finns som `labb2.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 2 redovisning''` och en `git push`. Se till att filerna **wordchain.cpp**, och **evilhangman.cpp** är med. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.

Del A: Ordkedja

En *ordkedja* är en förbindelse från ett ord till ett annat bildad genom att byta ut en bokstav i taget under villkoret att ett giltigt ord bildas i varje delsteg och att alla ord i kedjan är lika långa. Till exempel är följande en ordkedja som binder samman ordet `"code"` med ordet `"data"` på engelska. Varje utbytt bokstav är understruken för tydlighets skull:

$$\texttt{code} \rightarrow \texttt{c}\underline{\texttt{a}}\texttt{de} \rightarrow \texttt{ca}\underline{\texttt{t}}\texttt{e} \rightarrow \underline{\texttt{d}}\texttt{ate} \rightarrow \texttt{dat}\underline{\texttt{a}}$$

Det finns många ordkedjor som binder samman dessa två ord, men vår är den kortaste. Det kan finnas fler kedjor av samma längd, men ingen med färre steg än denna.

Du ska nu skriva ett program som hittar en ordkedja av minimal längd mellan två ord som matats in av användaren. Din kod måste använda STL `stack` och `queue` tillsammans med en given algoritm för att hitta en kortaste sådan sekvens.

Här är en interaktionslogg mellan ditt program och en användare (med användarens indata understruken):

```
Welcome to TDDD86 Word Chain.
If you give me two English words, I will transform the
first into the second by changing one letter at a time.

Please type two words: code data
Chain from data back to code:
data date cate cade code
Have a nice day.
```

Notera att ordkedjan skrivs ut i omvänd ordning, från det andra ordet till det första. Om det finns flera giltiga ordkedjor av samma längd mellan ett givet start- och slutord behöver ditt program inte generera exakt den kedja som visas i loggen ovan, men du måste generera en av minimal längd.

Du kan anta att indata är giltigt. Till exempel får du anta att användaren skriver exakt två ord och att bägge orden är giltiga ord i den engelska ordlistan samt att de inte är samma ord. Du får också anta att filen med den engelska ordlistan finns och är läsbar av ditt program. Om ogiltigt indata förekommer är ditt programs beteende ospecificerat; det kan göra vad du vill, inklusive krascha.

Du kommer att behöva slå upp engelska ord. Vi tillhandahåller en fil, `dictionary.txt` som innehåller dessa ord, ett per rad. Läs denna fil som indata och välj en effektiv container för att lagra och slå upp ord. Notera att du ska loopa över ordlistan bara en gång; nämligen när du läser och lagrar den i en effektiv container. Du ska aldriga loopa över alla element i containern för att lösa den här uppgiften.

Del A, implementationsdetaljer:
Att hitta en ordkedja är en specialfall av ett **kortaste vägenproblem** där vi vill hitta en väg från en startposition till en slutposition. Kortaste vägen-problem dyker upp inom routingproblem på Internet, när man vill jämföra muterade gener och så vidare. Strategin vi kommer att använda för att hitta en kortaste väg heter **breddenförstsökning (BFS)**, en sökprocess som expanderar ut från en startposition, undersöker alla möjligheter som ligger ett steg bort, sedan två steg och så vidare, till en lösning hittats. BFS garanterar att den första lösningen som hittats är lika kort som någon annan lösning. *(Breddenförstsökning är inte den mest effektiva algoritmen för att generera minimala ordkedjor men vi kommer att beröra bättre sökalgoritmer senare i kursen.)*

För ordkedjor börjar vi med att undersöka kedjor som är ett steg bort från originalordet, där endast en bokstav ändrats. Sedan kontrolleras alla kedjor som är två steg bort, där två bokstäver har bytts ut. Sedan tre, fyra och så vidare. Vi implementerar algoritmen för breddenförstsökning genom att använda en **kö** för att lagra partiella kedjor som representerar möjligheter att utforska. Varje partiell kedja är en **stack**, vilket betyder att den övergripande datastrukturen är en **kö av stackar**.

Här följer en beskrivning i pseudokod av algoritmen för att lösa ordkedjeproblemet:

```
function wordChain(w1, w2):
    create an empty queue of stacks
    create/add a stack containing {w1} to the queue
    while the queue is not empty:
        dequeue the partial-chain stack from the front of the queue
        if the word at the top of the stack is the destinaction word:
            hooray!  output the elements of the stack as the solution
        else:
            for each valid English word that is a neighbour (differs
            by 1 letter) of the word at the top of the stack:
                if that neighbour word has not already been used in a ladder before:
                    create a copy of the current chain stack
                    put the neighbour word at the top of the copy stack
                    add the copy stack to the end of the queue
```

Delar av pseudokoden svarar nästa direkt mot faktisk C++-kod. En del som är mer abstrakt är delen som instruerar dig att undersöka varje **"granne"** till ett givet ord. En granne till ett givet ord w är ett ord av samma längd som w som skiljer sig i exakt en bokstav från w. Till exempel är `date` och `data` grannar.

Det är *inte* tillräckligt att leta efter grannar genom att loopa över hela ordlistan varje gång; detta är alldeles för långsamt. Använd istället två nästlade loopar för att hitta alla grannar till ett givet ord: En som går igenom varje bokstavsposition i ordet och en som loopar igenom bokstäverna i alfabetet från `a-z` och byter ut bokstaven i den bokstavspositionen med var och en av de 26 bokstäverna. När du, till exempel, undersöker grannar till "`date`", skulle du testa:

- `aate`, `bate`, `cate`, ..., `zate` ← alla möjliga grannar där endast första bokstaven ändrats
- `date`, `dbte`, `dcte`, ..., `dzte` ← alla möjliga grannar där endast andra bokstaven ändrats
- ...
- `data`, `datb`, `datc`, ..., `datz` ← alla möjliga grannar där endast fjärde bokstaven ändrats

Notera att många möjliga ord längs vägen (`aate`, `dbte`, `datz`, etc.) inte är giltiga engelska ord. Din algoritm har tillgång till en engelsk ordlista och varje gång du genererar ett ord i den här processen behöver du slå upp det i ordlistan för att vara säker på att det faktiskt är ett giltigt ord.

En lite mer subtil sak är att du inte ska återanvända ord som inkluderats i en tidigare kedja. Anta, till exempel, att du har lagt till den partiella kedjan `cat` → `cot` → `cog` till kön. Senare, om din kod behandlar kedjan `cat` → `cot` → `con`, så är en granne till `con` faktiskt `cog`, vilket gör att du kanske skulle vilja undersöka `cat` → `cot` → `con` → `cog`. Men, att göra det är onödigt. Om det finns en ordkedja med dessa fyra ord så måste det finnas en kortare som tar bort mellanhanden genom att utesluta det onödiga ordet `con`. Så fort du har köat en kedja som slutar med ett specifikt ord har du hittat en väg av minimal längd från startordet till slutordet i kedjan, så du behöver *aldrig* köa det slutordet igen.

För att implementera denna strategi, håll reda på orden som redan använts i någon kedja. Ignorera dessa ord om de dyker upp igen. Genom att hålla reda på orden du redan använt undviker du också möjligheten att fångas i en oändlig lopp genom att råka bygga en cirkulär kedja, som till exempel `cat` → `cot` → `cog` → `bog` → `bag` → `bat` → `cat`.

Del B: Elak hänga gubbe

Det är svårt att skriva datorprogram som spelar spel. När vi människor sätter oss ned för att spela spel kan vi använda oss av tidigare erfarenheter, anpassa oss till motståndarens strategi och lära oss från våra misstag. Datorer, å andra sidan, följer blint en förutbestämd algoritm som (förhoppningsvis) får den att bete sig intelligent. Trots att datorer har slagit sina mänskliga skapare i vissa spel, som schack och dam, använder sig ofta deras program av hundratals år av samlad mänsklig erfarenhet och utomordentligt komplicerade algoritmer och optimeringar för att slå sina motståndare med beräkningskraft.

Det finns många sätt att bygga bra datormotståndare, men ett sätt som inte är så väl utforskat i modern forsk-

ning är — att fuska. Varför lägga möda på att försöka lära en dator subtila strategiska nyanser när man helt enkelt kan skriva ett program som inte spelar rent och därför kan vinna lätt? I den här uppgiften kommer du att bygga ett elakt program som böjer på reglerna i *Hänga gubbe* för att överlista den mänskliga motståndaren gång på gång.

Om du inte är bekant med Hänga gubbe går spelet till som följer:

1. En spelare väljer ett hemligt ord och ritar sedan ett antal streck som motsvarar ordets längd.
2. Den andra spelaren börjar gissa bokstäver. Så fort hen gissar en bokstav som ingår i det hemliga ordet avslöjar den första spelaren varje instans av den bokstaven i ordet. Annars räknas gissningen som felaktig.
3. Spelet är slut när antingen alla bokstäver i ordet avslöjats eller när den andra spelaren fått slut på gissningar.

Fundamentalt för spelet är faktumet att den första spelaren är ärlig med ordet hen har valt. På så sätt kan hen avslöja om en given gissad bokstav ingår i ordet eller inte. Men vad händer om den första spelaren inte är ärlig? Det skulle ge spelaren som väljer det hemliga ordet ett enormt övertag. Antag, till exempel, att du är spelaren som försöker gissa ordet och att du lyckats avslöja bokstäver så att spelet har följande tillstånd med endast en kvarvarande gissning:

```
DO-BLE
```

Det finns bara två engelska ord som matchar detta mönster “doable” och “double”. Om spelaren som valt det hemliga ordet spelar rent har du en femtio-femtiochans att vinna om du gissar att `'A'` eller `'U'` är den sista bokstaven. Om din motståndare däremot fuskar och faktiskt inte bestämt sig för något av orden är det omöjligt för dig att vinna. Oavsett vilken bokstav du gissar på kan din motståndare hävda att hen valt det andra ordet, säga att din gissning var fel och vinna spelet.

Låt oss illustrera tekniken med ett exempel. Antag att du spelar Hänga gubbe och att det är din tur att välja ett ord, vilket vi antar ska ha längd fyra. Istället för att faktiskt välja ett ord sammanställer du en lista över alla ord med fyra bokstäver du känner till. Låt oss, för enkelhets skull, anta att vi använder engelska och att vi bara kan komma på några stycken ord av längd fyra:

```
ALLY  BETA  COOL  DEAL  ELSE  FLEW  GOOD  HOPE  IBEX
```

Antag nu att din motståndare gissar bokstaven `'E'`. Du måste nu berätta för din motståndare vilka bokstäver i ordet du “valt” som är `E`:n. Nu har ju du egentligen inte valt något ord, vilket betyder att du har flera valmöjligheter när du ska avslöja `E`:na.

```
ALLY  BETA  COOL  DEAL  ELSE  FLEW  GOOD  HOPE  IBEX
```

Som du märker faller orden nu in i fem “ordfamiljer”.

- `----`, som innehåller `ALLY`, `COOL` och `GOOD`.
- `-E--`, som innehåller `BETA`, och `DEAL`.
- `--E-`, som innehåller `FLEW` och `IBEX`.
- `E--E`, som innehåller `ELSE`.
- `---E`, som innehåller `HOPE`.

Eftersom bokstäverna du avslöjar måste tillhöra *något* ord i din ordlista kan du välja att avslöja vilken som helst av de ovanstående fem familjerna. Det finns många sätt att välja vilken familj som ska avslöjas — kanske vill du styra din motståndare mot en mindre familj med mer obskyra ord, eller mot en större familj i hopp om att detta ska hålla många valmöjligheter öppna. I den här uppgiften ska vi, för enkelhets skull, använda oss av det sistnämnda sättet och alltid välja den största kvarvarande ordfamiljen. I det här fallet betyder det att du ska välja familjen `----`. Detta reducerar ner din ordlista till

```
ALLY  COOL  GOOD
```

och eftersom du inte avslöjade några bokstäver kan du säga till din motståndare att hens gissning var felaktig.

Låt oss titta på två exempel till av den här strategin. Givet denna ordlista med tre ord skulle du, om din motståndare gissar bokstaven `'O'`, dela upp ordlistan i två familjer:

- `-OO-`, som innehåller `COOL` och `GOOD`.
- `----`, som innehåller `ALLY`.

Den första av dessa familjer är större än den andra så du väljer den, avslöjar två `O` i ordet och reducerar din lista till

```
COOL  GOOD
```

Men vad händer om din motståndare gissar en bokstav som inte finns någonstans i ordlistan? Till exempel, om din motståndare nu gissar på `'T'`? Inga problem. Om du försöker dela upp de kvarvarande orden i ordfamiljer ser du att det bara finns en familj: Familjen `----`, med både `COOL` och `GOOD`. Eftersom det bara finns en ordfamilj är den trivialt den största och genom att välja den behåller du den ordlista du redan hade.

Nu kan det gå på två sätt. Antingen kan din motståndare vara smart nog att skära ner ordlistan till ett ord och sedan gissa det ordet. I det fallet borde du gratulera hen — det är imponerande spelat med tanke på vad du håller på med! Annars, och allra vanligast, kommer din motståndare att bli helt utspelad och få slut på gissningar. När detta händer kan du välja vilket ord du vill från den kvarvarande listan och hävda att det var ditt val hela tiden.

Del B, implementationsdetaljer:
Uppgiften är att skriva ett program som spelar Hänga gubbe enligt strategin vi skissat ovan. Utgå från filen `evilhangman.cpp` och implementera följande beteende:

1. Läs filen `dictionary.txt` som har en stor ordlista. För testning finns också `di.txt`.
2. Be användaren att mata in en ordlängd. Upprepa detta så länge det behövs till hen matar in ett tal sådant att det finns åtminstone ett ord som har den längden.
3. Be användaren mata in ett antal gissningar, vilket måste vara ett heltal större än noll. Oroa dig inte för ovanligt stora antal gissningar — trots allt kommer din motståndare inte ha glädje av fler än 26 gissningar!
4. Be användaren välja om hen vill få se antalet kvarvarande ord i ordlistan efter varje gissning. Detta förstör förstås illusionen av att du spelar rent, men är mycket användbart för testning (och rättning).
5. Spela en omgång Hänga gubbe enligt följande:
   (a) Konstruera en lista av alla ord i engelska språket vars längd matchar den önskade längden.
   (b) Skriv ut hur många gissningar användaren har kvar tillsammans med eventuella bokstäver spelaren har gissat och den nuvarande versionen av ordet (med ej avslöjade bokstäver som `'-'`). Om användaren tidigare valt att få se antalet kvarvarande ord, skriv ut detta också.
   (c) Be användaren gissa en bokstav och upprepa till hen gissar en bokstav hen ej gissat förut. Säkerställ att exakt en bokstav matats in och att det är en bokstav i alfabetet.
   (d) Partitionera orden i ordlistan i grupper baserat på ordfamilj.
   (e) Identifiera den vanligaste ordfamiljen bland de kvarvarande orden, ta bort alla ord i ordlistan som inte är med i den familjen och avslöja bokstävernas positioner (om några) för användaren. Dra bort en kvarvarande gissning om ordfamiljen inte innehåller den gissade bokstaven i någon position.
   (f) Välj ett ord från ordlistan och avslöja det som det "valda" ordet om spelaren fått slut på .
   (g) Gratulera spelaren om hen korrekt gissat ordet.
6. Fråga spelaren om hen vill spela igen och loopa därefter.

Det är upp till dig att fundera på hur du ska partitionera ord i ordfamiljer. Fundera på vilka datastrukturer som skulle vara bäst för att hålla reda på ordfamiljer och den stora ordlistan. Skulle en `vector` fungera? Kanske en `map`? En `stack` eller `queue`? Tänk igenom din design innan du börjar koda så sparar du mycket tid och huvudvärk.

Det program du ska skriva försöker upprätthålla en illusion: Det låtsas spela Hänga gubbe, men gör i själva verket något mycket mer elakt bakom kulisserna. Följdaktligen måste du se till att få ditt program så responsivt som möjligt. Om spelaren måste vänta flera sekunder efter att ha matat in en bokstav kommer hen säkerligen att misstänka att något inte står rätt till. Illusionen kommer att brytas och skönheten i ditt program att gå förlorad. Optimera dock inte i förtid; se till att få ditt program att fungera först, innan du bryr dig om dess effektivitet.

Möjliga utökningar

E1 — lite elakare (1 poäng):
Algoritmen för Elak hänga gubbe som skissats här är inte på något sätt optimal och det finns flera fall där den gör riktigt dåliga val. Antag till exempel att motståndaren har exakt en gissning kvar och att datorn har

följande ordlista:

```
DEAL   TEAR   MONK
```

Om motståndaren nu gissar bokstaven `'E'` upptäcker datorn att ordfamiljen `-E--` har två element och att ordfamiljen `----` bara har ett. Alltså väljer datorn familjen med `DEAL` och `TEAR`, avslöjar ett `E` och ger motståndaren en chans till att gissa. Men, eftersom motståndaren bara hade en gissning kvar, hade ett mycket bättre beslut varit att välja familjen `----` med `MONK` som enda medlem och därmed få motståndaren att förlora spelet direkt.

Skapa en ny branch som heter E1 och implementera ett bättre beteende när motståndaren endast har en gissning kvar. Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 E1 redovisning''` och en `git push`. Se till att filen **evilhangman.cpp** är med. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] E1 redovisning`.

E2 — mycket elakare (3 poäng):
Av resonemanget i E1 ser vi att strategin att alltid välja den största ordfamiljen inte nödvändigtvis är bäst i alla lägen. Fundera på möjliga förbättringar av algoritmen. Kanske kan du vikta ordfamiljerna med något annat mått än storlek. Kanske kan du få datorn att "titta framåt" ett steg eller två genom att ta hänsyn till de möjliga val som kan komma att uppträda i framtiden. Skapa en ny branch som heter E2 för ditt mycket elakare program. Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 E2 redovisning''` och en `git push`. Se till att filen **evilhangman.cpp** är med. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] E2 redovisning`.