TDDD86 – Laboration #5

24 augusti 2022

I den här uppgiften får du öva på rekursiv backtracking. Filerna du behöver för att komma igång finns som `labb5.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 5 redovisning''` och en `git push`. Se till att filerna **Boggle.h**, **Boggle.cpp** och **boggleplay.cpp** är med. Informera sedan din assistant genom att maila honom/henne.

Boggle är ett spel som spelas på ett kvadratiskt rutnät på vilket man slumpmässigt fördelar en uppsättning bokstavskuber. Bokstavskuberna är 6-sidiga tärningar, förutom att de har en bokstav på varje sida snarare än en siffra. Målet är att hitta ord på spelbrädet via stigar genom närliggande bokstäver. Två bokstäver är **grannar** om de finns bredvid varandra horisontellt, vertikalt eller diagonalt. Det finns upp till åtta bokstäver nära en kub. Varje kub får användas maximalt en gång i ett ord.

I den riktiga versionen av det här spelet arbetar alla spelare samtidigt och listar orden de hittar på ett papper. När tiden tagit slut tas dubbletter bort och spelarna får en poäng för varje unikt ord, d.v.s, för varje ord ingen annan spelare lyckats hitta.

Din uppgift är att skriva ett program som spelar en konsollbaserad version av spelet anpassad för en mänsklig spelare mot en datormotståndare. Oturligt nog vet datorn hur man använder rekursiv backtracking, så den kan hitta alla möjliga ord på brädet och krossa dig varje gång.

För att starta spelet skakar man bokstavskuberna och lägger ut dem på brädet. Den mänskliga spelaren spelar först, genom att skriva in orden ett efter ett. Din kod verifierar först att ordet är giltigt, sedan lägger du in det i spelarens ordlista och ger spelaren poäng beroende på ordets längd (en poäng per bokstav ≥ 4). Ett ord är giltigt om det uppfyller följande kriterier:

- minst 4 bokstäver långt
- är ett giltigt engelskt ord i ordlistan
- kan bildas genom att binda samman närliggande bokstavskuber på brädet (genom att använda varje given kub som mest en gång)
- har inte redan bildats av spelaren under denna spelomgång (även om det finns flera möjliga sätt att bilda samma ord räknas ordet som mest en gång)

När spelaren har hittat så många ord hen kan är det datorns tur. Datorn söker i brädet för att hitta samtliga kvarvarande ord och tilldelar sig själv poäng för dessa ord. Datorn slår typiskt spelaren eftersom den hittar *alla* ord.

Ditt program bör efterlika exekveringsloggen till höger. I resursmappen bland startfilerna finns filer med kompletta exempelkörningar.

(introtext borttrimmad)

```
Do you want to generate a random board? y
It's your turn!
FYCL
IOMG
ORIL
HJHU

Your words (3): {"FOIL", "FORM",
                 "ROOF"}
Your score: 3
Type a word (or press Enter
             to end your turn): room
You found a new word! "ROOM"
...

Your words (5): {"FOIL", "FORM",
         "ROOF", "ROOM", "ROOMY"}
Your score: 6
Type a word (or press Enter
             to end your turn):

It's my turn!
My words (16): {"COIF", "COIL", "COIR",
"CORM", "FIRM", "GIRO", "GLIM", "HOOF",
"IGLU", "LIMO", "LIMY", "MIRI", "MOIL",
"MOOR", "RIMY", "ROIL"}
My score: 16
Ha ha ha, I destroyed you. Better luck
next time, puny human!

Play again (Y/N)? n
Have a nice day.
```

Att sätta upp spelbrädet

Det verkliga bogglespelet har sexton bokstavskuber med vissa bestämda bokstäver på sidorna. Bokstäverna på kuberna är inte slumpmässiga utan utvalda så att vanliga bokstäver ska dyka upp oftare och att det ska vara lättare att få en bra blandning av vokaler och konsonanter. Vi vill att ditt bogglespel ska matcha detta. Följande tabell listar samtliga bokstäver på all sex sidor av de sexton kuberna i Boggle. DU bör bestämma dig för ett lämpligt sätt att representera denna information i ditt program och deklarera det därefter.

AAEEGN	ABBJOO	ACHOPS	AFFKPS	AOOTTW	CIMOTU	DEILRX	DELRVY
DISTTY	EEGHNW	EEINSU	EHRTVW	EIOSST	ELRTTY	HIMNQU	HLNNRZ

"Skaka" kuberna i början av varje spelomgång. Vi har två slumpmässiga aspekter att ta hänsyn till:

- En slumpvis vald plats på 4×4-brädet ska väljas för varje kub.
- En slumpvis vald sida av varje kub ska väljas som den uppåtvända sidan för den kuben.

Vi skickar med en fil **shuffle.h** som inkluderar en funktion `shuffle` som kan blanda om slumpvis bland elementen i en array/ett fält, en `vector` eller en `Grid`. Om du är nyfiken på att veta hur blandningen går till så använder algoritmen följande pseudokod:

```
function shuffle(list):
  for each int i from 0 to list's size - 1:
    Choose a random index r between i and the last element index, inclusive.
    Swap the elements at positions i and r.
```

Ditt spel måste också ha valet att låta användaren mata in en brädkonfiguration manuellt. I detta val matar användaren in en sträng av 16 bokstäver, representerande kuberna från vänster till höger, topp till botten. (Detta är också användbart för att testa din kod.) Verifiera att användarens sträng är tillräckligt lång för att fylla brädet och be om en ny inmatning ifall den inte är exakt 16 tecken lång. Be också om en ny inmatning om något av de 16 tecknen inte är en bokstav från A-Z. Din kod ska inte bry sig om användaren matar in versaler eller gemener. Du ska inte kontrollera huruvida de 16 inmatade tecknen faktiskt skulle kunna bildas av de 16 bokstavskuberna; acceptera helt enkelt vilka 16 alfabetiska tecken som helst.

Den mänskliga spelarens tur

Den mänskliga spelaren matar in alla ord hen hittar på brädet. Som tidigare beskrivits måste du för varje inmatat ord kontrollera att det är minst fyra bokstäver långt, finns i den engelska ordlistan, inte redan har inkluderats i spelarens ordlista och kan bildas på brädet via närliggande kuber. Meddela användaren om något villkor inte håller. Det finns inget straff för att mata in ett ogiltigt ord, men ogiltiga ord räknas inte eller som poänggivande.

Om ordet är giltigt lägger du till ordet till spelarens ordlista och poäng. Ordets längd bestämmer poängen: ett 4-bokstavsord är värt 1 poäng; ett 5-bokstavsord är värt 2 poäng; 6-bokstavsord är värda 3 poäng; och så vidare. Spelaren matar in en tom rad när hen inte hittar fler ord, vilket signalerar slutet på den mänskliga spelarens tur.

Datorspelarens tur

När den mänskliga spelaren är färdig med att mata in ord söker datorn igenom brädet för att hitta kvarvarande ord den mänskliga spelaren missat. Datorspelaren får poäng för varje kvarvarande ord som uppfyller kriterierna. Om datorns slutpoäng är strikt större än människans vinner datorn. Om spelaren får lika många poäng eller om människans poäng är större än datorns vinner den mänskliga spelaren.

Du kan hitta alla ord på brädet med hjälp av **rekursiv backtracking**. Idén är att starta från en given bokstavskub, sedan utforska närliggande kuber runt den och pröva alla partiella strängar som kan tillverkas, sedan utforska varje grannes granne och så vidare. Algoritmen blir ungefär som följer:

```
for each letter cube c:
   mark cube c as visited.
   for each neighbouring cube next to c:
      explore all words that could start with c's letter.
   un-mark cube c as visited.
```

Du vill inte besöka samma bokstavskub två gånger under en given utforskningsstig, så för att algoritmen ska fungera behöver din `Boggle`-klass något sätt att "märka" en bokstavskub som besökt eller ej. Du skulle kunna använda en separat datastruktur för märkning, eller modifiera din existerande brädstruktur, etc. Det är upp till dig, så länge som det fungerar och är effektivt.

Effektivitet är väldigt viktigt för den här delen av programmet. Precis som med många exponentiella sökalgoritmer är det här viktigt att hitta sätt att begränsa sökningen för att vara söker på att processen kan slutföras inom rimlig tid. Om den skrivs på rätt sätt ska koden för att hitta alla ord på brädet exekvera på en sekund eller mindre. För att säkerställa att din kod är tillräckligt effektiv måste du göra följande optimeringar:

- använd en `Lexicon`-datastruktur för att lagra den engelska ordlistan (finns bland den givna koden)
- skär av grenar av sökträdet genom att inte utforska partiella stigar som inte kan bilda något giltigt ord
- använd effektiva datastrukturer i övrigt i ditt program

En av de viktigaste strategierna i Boggle är att skära bort sökgrenar som inte leder någon vart. `Lexicon`-datastrukturen har en medlemsfunktion `containsPrefix` som tar in en strängparameter och returnerar `true` om något ord i ordlistan börjar med den delsträngen. Om till exempel den första kuben du väljer att börja ifrån visar bokstaven `Z` och din algoritm sedan utforskar en av dess grannar och hittar ett `X` har du en stig som börjar med `ZX`. I det här fallet skulle `containsPrefix`-funktionen informera dig om att det inte finns något engelskt ord som börjar med prefixet "`ZX`". Därför bör din algoritm avsluta utforskandet av den här stigen och ägna sig åt mer lovande kombinationer.

Implementationsdetaljer:
Vi tillhandahåller en fil **bogglemain.cpp** som innehåller programmets `main`-funktion. Koden skriver ut ett introduktionsmeddelande om spelet och startar sedan en loop som gör upprepade anrop till en funktion `playOneGame`. Efter varje anrop till `playOneGame` frågar huvudkoden om spelaren vill spela igen och avslutar när användaren till slut säger "no". `playOneGame`-funktionen är inte skriven: du måste skriva den i **boggleplay.cpp**. I samma fil kan du placera all annan logik och de hjälpfunktioner som behövs för att spela en omgång av spelet.

För att det konsollbaserade gränssnittet ska upplevas lite renare är det bra om du rensar skärmen mellan varje ord användaren matar in. Detta gör att spelets tillstånd vanligtvis är synligt på samma plats på skärmen hela tiden under spelets gång. Använd gärna funktionen `clarConsole` för detta ändamål.

Funktionen `playOneGame` ska utföra all konsollbaserad användarinteraktion så som att skriva ut spelets nuvarande tillstånd. Detta är den enda filen i vilken du ska ha några satser som läser/skriver till `cout` eller `cin`. Men **boggleplay.cpp** är inte tänkt att vara platsen för att hålla reda på spelets tillstånd, logik eller algoritmer. I stället ska huvuddelen av den koden finnas i **Boggle.h**/**Boggle.cpp**-filerna, vilka ska innehålla implementationen av en `Boggle`-klass. Ett `Boggle`-objekt representerar det aktuella spelbrädet och spelets tillstånd och ska ha medlemsfunktioner för att utföra huvuddelen av spelets funktioner som att hitta ord på brädet, hålla reda på poäng och bestämma vem som vunnit spelet.

I tidigare laborationer har vi gett dig en lista över vilka medlemmar du behövt implementera. I den här uppgiften ber vi dig att själv komma på vilka medlemmar som behövs. Kom ihåg att varje medlemsfunktion i din klass ska ha ett klart och tydligt syfte. Du måste också deklarera de privata datamedlemmarna som behövs i ett `Boggle`-objekt.

- Du kommer säkerligen behöva en datastruktur för att representera spelbrädets tillstånd, alltså vilka bokstäver de 16 bokstavskuberna visar upp.
- Du bör också lagra ett `Lexicon` i ditt bogglespel för att representera den engelska ordlistan. Vi tillhandahåller resursfilen **EnglishWords.dat** som innehåller över 125000 ord du måste använda för att initiera ditt `Lexicon`. Den här filen är i ett effektivt binärt format som optimerar ditt `Lexicon` för sökningar efter ord och prefix.
- Det går bra att deklarera ytterligare datamedlemmar.
- Gör inte något till en privat datamedlem om det bara behöva i **en** medlemsfunktion.
- Trots att **boggleplay.cpp**-filen ska sköta all I/O bör din `Boggle`-klass ha många behjälpliga funktioner för den att anropa så att den inte behöver någon komplicerad logik. Skriv till exempel funktioner så att **boggleplay** kan skicka ett ord till din `Boggle`-klass och ställa diverse frågor om det, som om det är tillräckligt långt, om det finns i ordlistan, om det redan spelats och så vidare. **boggleplay**-klienten kan anropa alla dessa och, beroende på resultatet, skriva ut ett passande meddelande.
- **boggleplay**-koden behöver kunna visa alla ord som hittats av den mänskliga spelaren, tillsammans med spelarens poäng. `Boggle`-klassen ska hålla reda på sådana saker, inte **boggleplay**. **boggleplay**-koden ska fråga `Boggle`-klassen efter sådan information vilken bör returneras eller skickas ut till den.
- Varje gång användaren säger att hen vill spela en gång till kommer du att behöva återställa spelets tillstånd, radera eventuellt funna ord, nollställa poäng och så vidare. `Boggle`-klassen ska innehålla alla logik för att återställa detta tillstånd, så väl som att skaka bokstavskuberna och blanda de 16 bokstäverna till en ny kombination i början av varje spel. Kom ihåg att ditt bogglespel ska ha ett sätt att låta **boggleplay**-klienten initiera ett nytt spel med en fix 16-bokstävers sträng representerande bokstäverna användaren

vill ha på brädet.

- Glöm inte `const`.
  Lämplig arbetsgång:
  1. Sätt upp kuber, utritning av bräde, kubskakning. Designa datastruktur för kuber och bräde. Initiera och blanda kuberna. Lägg till ett val för att låta användaren tvinga en konfiguration av brädet.
  2. Människans tur (förutom att hitta ord på brädet). Skriv loopen som låter användaren mata in ord. Förkasta ord som redan matats in, inte är tillräckligt långa eller inte finns i ordlistan. Oroa dig inte för den rekursiva sökalgoritmen än; genomför bara övriga kontroller av ett ord.
  3. Backtrackingalgoritm för att hitta ett givet ord på brädet. Använd nu rekursion för att verifiera att ett ord kan bildas på brädet, givet spelets regler. Se till att avbryta sökningen så fort du inser att den inte kan leda till ett positivt resultat.
  4. Datorns tur (hitta alla ord på brädet). Implementera nu datorspelaren. Använd kraften hos rekursion för att traversera spelbrädet i en uttömmande sökning för att hitta alla kvarvarande ord.
     NOTERA: Programmet innehåller två rekursiva sökningar: en för att hitta ett specifikt ord inmatat av den mänskliga spelaren och en annan för att söka över hela brädet under datorspelarens tur. Du tänker kanske att dessa borde gå att kombinera till en integrerad funktion, genom att göra all ordsökning i början av spelet, precis efter att brädet initierats. För att bli godkänd på labben måste du dock implementera människan och datorn som **två separata sökfunktioner**. Det finns tillräckliga skillnader mellan de två för att de inte ska gå att kombinera rent och snyggt.
  5. Loopa för att spela flera spel och polera.