Tentamen i EDAF25

23 april, 2025

Skrivtid: 14-19

- Skriv bara på ena sidan av pappret
- Skriv tydligt – om texten inte går att läsa kan du inte få några poäng.
- Sätt identitet och sidnummer på varje inlämnat blad, kontrollera att sidnumret på din sista sida är samma som det antal blad du markerar på omslagspappret.

Tentamen består av uppgifter med totalt 40 poäng. Dessa poäng är fördelade på två avdelningar: *Grafteori* (10p) och *Design* (30p). För godkänt betyg kommer att krävas högst hälften av den sammanlagda poängen för de två avdelningarna.

Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet.

Hjälpmedel:
- Andersson: UML–syntax – Enstaka exemplar finns att låna. Lämna tillbaka när tittat klart så andra kan låna dem.
- Holm: Java snabbreferens – Finns att låna.

Grafteori

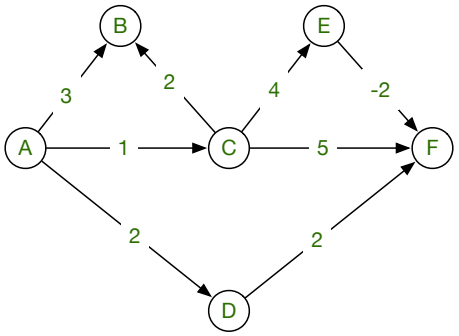
Uppgift 1

Pseudokoden nedan beskriver hur man kan räkna ut det maximala flödet genom en graf med hjälp av Edmond-Karps algoritm. Dock finns det totalt fem (5) rader med fel i koden. Ange för varje felaktighet vilket/vilka radnummer felet finns på, vad felet består i och hur det borde se ut i stället. Observera att det kan finnas mer än ett fel på en rad. *(5p)*

```
1  findpath(r, s, t)
2      find a path from s to t in the residual graph r:
3          - only edges with capacity > 0 must be used
4          - search for the path depth first
5      return the path or null if no path was found
6
7
8  edmonds_karp(g, s, t)
9      // g = a graph, s = start node, t = end node
10     r = create a residual graph for g and copy capacities:
11         for each edge in g the initial residual graph should
12         contain an equal edge with the same capacity
13         but in the opposite direction
14     flow = 0
15     while true:
16         path = findpath(r, s, t)
17         if path is null:
18             break
19         else:
20             bottleneck = smallest capacity along path
21             flow += bottleneck
22             traverse path and update the residual values:
23                 - add the flow value in the forward direction
24                 - subtract the value in the backward direction
25     return bottleneck
```

Uppgift 2

Betrakta följande graf där siffrorna anger kostnaden för respektive båge:



(a) Antag att vi använder Dijkstras algoritm för att hitta det billigaste avståndet från nod *A* till nod *F*. Vilket värde kommer algoritmen i så fall att ge oss? *2 p*

(b) Motivera varför algoritmen kommer fram till detta värde. *2 p*

(c) Vilket värde borde Bellman-Fords algoritm ge för det minsta avståndet från nod *A* till nod *F*? *1 p*

Design

Uppgift 3

I inlämningsuppgift 1 (Computer) skulle man kunna implementera en `Factorial`-klass som beräknar $n!$ för $n = 5$ med typen `LongWord` med:

```
1  public class Factorial extends Program {
2      public Factorial() {
3          Address n = new Address(0),
4                  fac = new Address(1);
5          add(new Copy(new LongWord(5), n));
6          add(new Copy(new LongWord(1), fac));
7          add(new JumpEq(6, n, new LongWord(1)));
8          add(new Mul(fac, n, fac));
9          add(new Add(n, new LongWord(-1), n));
10         add(new Jump(2));
11         add(new Print(fac));
12         add(new Halt());
13     }
14 }
```

Här har vi dock hårdkodat både värdet på n och typen `LongWord` – vi vill dock kunna använda klassen även för andra värden på n, och andra `Word`-typer (subklasser till `Word`).

(a) Vilken SOLID-princip bryter koden ovan mot, och vilket/vilka mönster kan vi använda för att göra koden bättre. *2 p*

(b) Refaktorisera koden ovan, så att vi lätt kan byta mellan olika värden på n, och använda olika `Word`-typer. Definiera de interfaces och implementera de klasser som behövs för att vi skall kunna använda din modifierade `Factorial`-klass med godtyckliga `Word`-typer, och se specifikt till att vi kan vi kan använda både `LongWord` och `VeryLongWord`. I `Factorial`-klassen behöver du bara tydligt visa vilka rader som skall ändras, och förklara hur de skall skrivas istället. De båda klasserna `LongWord` och `VeryLongWord` är färdigskrivna, och deras konstruktorer är:

- `LongWord (long value)`
- `VeryLongWord (String value)`

Här skickar vi ett `long`-värde till `LongWord`, eftersom de inte behöver kunna hantera större värden än vad som ryms i en `long`, medan vi skickar en `String` till `VeryLongWord`, eftersom den skall kunna hantera tal som är mycket större än någon numerisk typ i Java, och `String` är tillräckligt flexibel för att kunna beskriva godtyckligt stora tal[1].

Tips: Den enda standardtyp som är tillräckligt generell för att beskriva *alla slags ord* är just `String` (vi kan inte använda exempelvis `BigInteger`, eftersom vi inte kan vara säkra på att våra `Word` är heltal – vi kan inte ens vara säkra på att de är tal). *10 p*

(c) Vi har en metod:

```
void run(Computer computer, Program program)
```

där vi för att göra uppgiften enklare antar att `computer` har ett minne som automatiskt får rätt slags ord.

Skriv nu två satser som använder din refaktoriserade kod från uppgift (b) och anropar metoden `run()` ovan med ett givet `computer`-objekt för följande beräkningar:

- beräkna 10! med `LongWord`, och
- beräkna 70! med `VeryLongWord`.

Dessa beräkningar måste egentligen använda olika `computer`-objekt, eftersom de kräver minnen med olika slags `Word`-objekt, men när du anropar `run` kan du förutsätta att vi har rätt slags ord i minnet på `computer`. *2 p*

[1] Det betyder inte att klassen `VeryLongWord` nödvändigtvis använder strängar internt – hur den är implementerad spelar ingen roll för uppgiften.

Uppgift 4

Vi vill kunna beräkna värdet av reella uttryck, och vi vill använda liknande teknik som vi använt bland annat i projektuppgift 2 i denna kurs.

För att till exempel bestämma värdet av uttrycket 1,0+2,0·3,0 i Javakod vill vi kunna skriva satserna:

```
var expr = new Plus(new Number(1.0),
                    new Times(new Number(2.0),
                              new Number(3.0)));
System.out.println(expr.value());
```

och alltså få ut värdet 7,0.

(a) Skriv alla gränssnitt och klasser som behövs för att vi skall kunna beräkna uttryck som i det inledande exemplet ovan med hjälp av den givna Javakoden. Använd det traditionella mallmetodmönstret, *Template Method Pattern*, för att undvika kod-duplicering. Du behöver bara implementera den funktionalitet för uttryck som visas i exemplet (det vill säga konstanter, addition och multiplikation), men man ska lätt kunna lägga till fler operationer. Uttrycken ska kunna vara godtyckligt långa. *8 p*

(b) Rita ett sekvensdiagram som visar beräkningen av uttrycket `expr.value()` i huvudprogrammet ovan – använd implementationen i (a). *3 p*

(c) Ett alternativ till att använda mallmetodmönstret/Template Method Pattern för att undvika duplicerad kod är att använda strategimönstret, *Strategy Pattern*, för att åstadkomma samma sak. Skriv en variant av lösningen i (a) där vi istället använder strategi-objekt i form av lambda-uttryck för att undvika kod-duplicering. Du behöver inte skriva om de klasser eller interfaces i (a) som kan återanvändas utan att ändras. *4 p*

(d) Ge ett exempel på en situation då strategimönstret skulle kunna vara att föredra framför Template Method. *1 p*