Luleå Tekniska Universitet **19 mars 2014**
Laborationsrapport

Laboration 3

Datastrukturer och Algoritmer D0041D

Primms Algoritm

Namn Magnus Björk
E-mail magbjr-3@ltu.student.se

Handledare
Felix Hansson

Innehåll

1 Problemspecifikation **1**

2 Åtkomst och användarhandledning **1**

3 Systembeskrivning **2**

4 Algoritmbeskrivning **4**

5 Testkörningar **6**

6 Diskussion **6**

7 Lösningens begränsningar **7**

8 Problem och reflektioner **7**

1 Problemspecifikation

Uppgiften gick ut på att implementera Primms algoritm, som med en d-ary heap skulle skapa ett minimalt uppspännande träd från en sammanhängande, viktad och oriktad graf.

Sedan skulle körtiden av algoritmen undersökas. Resultatet skulle visa hur tidskomplexiteten blir påverkad av förhållandet mellan D (antal barn noder per förälder nod i d-ary heap) och E (antal kanter i grafen.)

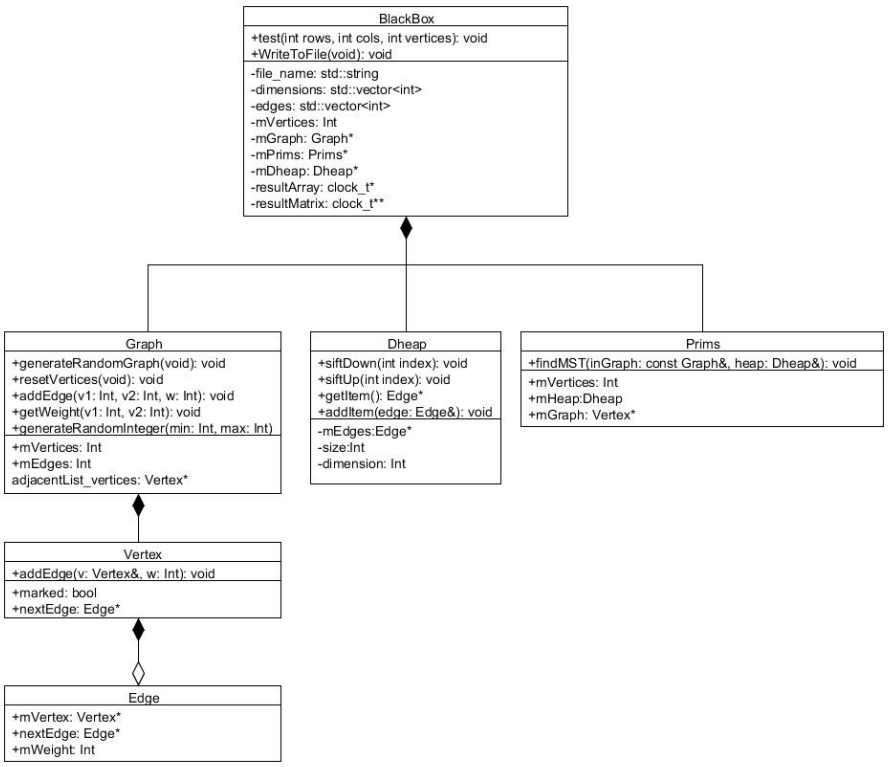
2 Åtkomst och användarhandledning

Laborationen är skriven i Visual Studio 2012 och skall följa med i samma zip-fil som denna rapport.

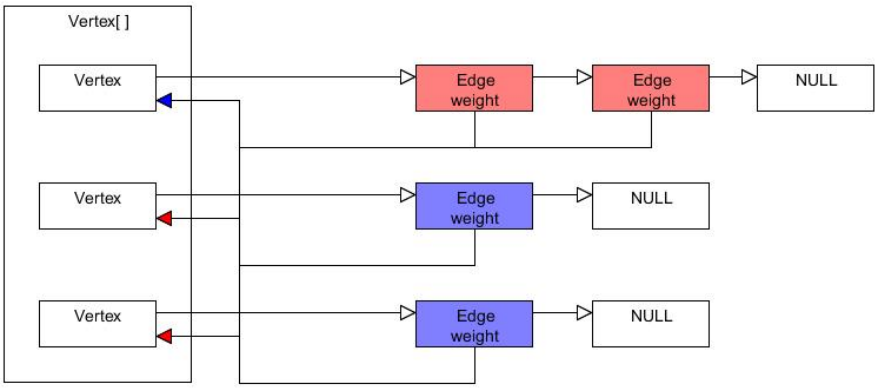
I filen main.cpp finns main funktionen som kör programmet. Vill man göra ett test måste man först initiera ett objekt BlackBox sedan köra metoden BlackBox::test(int rows, int cols, int vertices).

Datat sparas i en matris av typen std::clock_t, därför måste man ange rader (int rows) och kolumner (int cols) när man skall köra test metoden. Parametern int vertices bestämmer storleken på den graf som skall testas. Antal kanter räknas ut utifrån hur många hörn grafen har.

3 Systembeskrivning



Figur 1: *Klassdiagram över programmet.*



Figur 2: *Exempel på graf med två kanter och tre hörn. En kant representeras av två Edge objekt (en till och en tillbaka).*

Programmet består av sex stycken klasser:

- Graph

  Representation av graf i form av en så kallad adjacent-list.

  Viktiga metoder:

  - void Graph::generateRandomGraph(void)
    Metod för att populera grafen med kanter med slumpade vikter. Grafen är sammansat, oriktad och viktad. Vikterna tar inte hänsyn till triangel olikheten.

- Vertex

  Klass som representerar hörnen i grafen. Har en pekare till ett Edge objekt, detta skall då vara hörnets första kant. Vertex har även en bool som används av Prims Algoritm för att avgöra om hörnet ingår i MST (minimum spanning tree).

- Edge

  Denna klass representerar kanterna i grafen. Har två viktiga pekare: En Vertex pekare som refererar till den Vertex kanten går till. Samt finns en Edge pekare som pekar på nästa kant, förutsatt att det finns fler kanter, annars är det en noll-pekare.

- Prims

  I denna klass finns metoden som utför Prims Algoritm. Implementationen finns utförligen beskriven under Algoritmbeskrivning.

  Viktiga metoder:

  - void findMST(const Graph& inGraph, Dheap& heap)
    Metod som skall finna minsta uppspännande träd i en graf. Använder sig av en prioritetskö i form av typen Dheap.
  - void addEdgesToHeap(Vertex& Vertex)
    Metod som används av Prims::findMST(...) för att lägga till alla kanter av ett hörn till prioritetskön.

- Dheap

  Implementering av en så kallad d-ary heap. En d-ary heap liknar en binärheap men skiljer sig på grund av att den kan ha d antal barn per förälder istället för två. En binär heap är en d-ary heap med d = 2.

  Vid initiering av Dheap så krävs parametrar för d och även antal kanter som finns i grafen. Antal kanter krävs av anledningen att jag allokerar en array av Edge objekt som potentiellt skall kunna rymma alla kanter i grafen.

  Dheap har en Integer som håller reda på hur många element som finns i ovannämnda array.

- BlackBox

  Klass som innehåller testmetoden som testar Prims Algoritm. Den mätar tiden algoritmen tar för att köra och skriver även ut resultat till fil.

  Viktiga metoder:

  - void test(int rows, int cols, int vertices)
    Testdatat sparas i en matris av typen std::clock_t. Därför måste man ange int rows och int cols som parametrar. Parametern int vertices anger storleken på den graf som skall skapas i testet.

4 Algoritmbeskrivning

- Graph::generateRandomGraph()
  1. Skapa en array av Vertex objekt
  2. Slumpa ordningen i array och Skapa Edge objekt mellan index och index + 1. Nu skall det finnas array.size() - 1 stycken Edge's
  3. Slumpa ut två unika Vertex ur array som inte redan har en Edge tillsammans och skapa en Edge mellan dessa
  4. Utför 2 och 3 tills dess att önskat antal Edge uppnåtts.
- Dheap::siftDown(int index) (Dheap::siftUp(int index)) siftDown används när man tar bort ett element ur Dheap. siftUp när man lägger till ett element. Bägge bygger på samma princip därför går jag bara igenom sift-Down().
  1. Parameter int index är förälder.
  2. Ta minsta element av barn och jämför med förälder.
  3. Är förälder lika eller mindre, avbryt rekurssion.
  4. Byt plats på förälder och minstabarn och utför algoritmen igen fast med minsta barnets föregående position som index.
- BlackBox::test(int rows, int cols, int vertices)

  Tider av test sparas i en matris av storlek (rows * cols). I kolumnerna i matrisen skiljer sig d av d-ary heapen och i raderna av matrisen antal kanter av grafen som testas. När max_edges nämns menas (vertices * (vertices - 1)) / 2.
  1. Skapa matris av std::clock_t objekt, med storlek: int cols * int rows.
  2. Initiera och slumpa en graf med *vertices* antal hörn och $\frac{max\_edges}{rows}$ antal kanter.
  3. Initiera en Dheap med d = 2.
  4. Starta en klocka och kör Prims algoritm på grafen med skapad Dheap som prioritetskö.
  5. Stoppa klockan och lägg till tiden i matris.
  6. Gå tillbaka till steg 3. men öka d.
  7. Har en kolumn i matrisen fyllts med resultat så gå till steg 2. och öka antal kanter med $\frac{max\_edges}{rows}$ antal kanter.
  8. .När hela matrisen är full är testet slut.

- Prims::addEdgesToHeap(Vertex& vertex)

  Hjälpmetod till Prims::findMST. Lägger till alla kanter av ett hörn till prioritetskön.

  1. Markera att givet hörn ingår i trädet.
  2. Om kantpekaren av givet hörn är noll, avbryt.
  3. Om det hörn som kantpekaren pekar på *INTE* ingår i trädet, lägg då till denna kant till prioritetskön.
  4. Gå till 2 men använd kantpekaren av kanten i 3.

- Prims::findMST(Graph& graph, Dheap& dheap)

  Min implementering av algoritmen finnes i klassen Prims. I klassen Graph finns en array över alla hörn som finns i grafen. Denna array används i Prims för att få tillgång till alla kanter.

  1. Välj ett hörn ur array given från Graf.
  2. Lägg till kanter av hörnet till prioritetskön med hjälp av Prims::addEdgesToHeap.
  3. Ta ut minsta kant ur prioritetskön.
  4. Om hörnet denna kant pekar på redan är med i trädet ta då nästa kant ur prioritetskön. Upprepa tills ett hörn utanför trädet finnes.
  5. Nu har den kant med minst som pekar till ett hörn utanför trädet funnits. Om det inte finns kvar hörn i grafen är algoritmen färdig. Annars gå till 2. och använd funna hörn.

5 Testkörningar

Samtliga tider är i millisekunder. Kolumner visar antal barn i d-heap och raderna visar hur många kanter som fanns i den graf som testades.Notera att det inte är samma graf som testats i tabell 1 och tabell 2.

Tabell 1: Graf med 1000 hörn

	2	5	8	11	14	17	20	23	26
49950	12	9	9	9	9	9	10	12	10
99900	20	19	19	19	19	18	20	20	20
149850	34	30	31	28	28	28	28	29	31
199800	44	40	39	39	39	40	40	40	40
249750	60	53	51	53	52	52	54	54	53
299700	68	59	63	58	62	63	62	63	61
349650	82	74	69	73	71	72	71	76	69
399600	97	85	87	87	87	83	89	87	88
449550	105	90	91	86	95	91	91	89	92

Tabell 2: Graf med 1000 hörn

	2	4	8	16	32	64	128	256	512
49950	12	11	8	10	11	13	14	16	24
99900	21	20	20	19	20	21	23	27	34
149850	31	29	30	29	30	33	36	40	48
199800	47	43	41	40	44	45	48	53	58
249750	53	52	47	48	51	55	59	64	71
299700	73	69	66	64	67	68	71	80	83
349650	79	72	72	72	74	75	79	86	94
399600	89	81	78	79	81	83	86	91	102
449550	111	97	94	93	106	95	96	112	121

6 Diskussion

Testerna visade att för Prims Algoritm så lämpar sig dary heap bättre som prioritetskö än en vanlig binary-heap.

Fördelen med en dary heap över en binary heap är att det går snabbare att sätta in element i heapen, det krävs färre platsbyten helt enkelt. Nackdelen är när man ska ta bort element ur heapen.

När det gäller Prims Algoritm i täta grafer så sätter man in betydligt fler element än man tar bort och där kan man då tjäna in lite tid.

7 Lösningens begränsningar

Inte modulär. Ville från början göra en ADT Graph och göra både adjacent list och adjacency matrix för att jämföra skillnaden mellan dessa grafer. Tänkte även göra en ADT Algorithm för att kunna testa både Kruskals och Prim.

Det slutade med att jag slopade hela den tankegången och gjorde denna lösning istället.

Programmet visar inte något konkret bevis på att det faktiskt hittar MST. Nedan följer ett screenshot där jag kört algoritmen på en graf med 6 hörn och 15 kanter.

Figur 3: *Hur man tyder trädet: Var rad av trädet är en kant som går ut från trädet. 103 « 2 betyder till exempel att det finns en kant mellan T = {0} och hörn 2 med vikt 103. 31 « 3: Det finns en kant mellan T = {0,2} och 3 med vikt 31 etc.*

8 Problem och reflektioner

Som jag nämnt under delen *Lösningens begränsningar* så hade jag en annan tankegång från utgångsskedet. Detta visade sig vara för ambitiöst i förhållande till hur mycket tid som fanns tillgodo.

Förutsatt att denna lösning är korrekt så är jag nöjd med laborationen.

Hade jag ändrat något hade det varit hur jag genererar grafen. Jag skulle ha gjort en metod som alltid genererar maximalt antal kanter i förhållande till hörn. Detta skulle ha bidragit till att graferna genererades snabbare och jag skulle ha kunnat testa snabbare.

Skrivit kommentarer i koden på svenska. Skrev allt färdigt och körde sedan Doxygen för att upptäcka att vårat kära alfabet inte översattes korrekt. Bytte därför bara å,ä till a och ö till o. Hädan efter blir det att skriva kommentarer på engelska.