BAB 3

ANALISA DAN PERENCANAAN

Bab ini menjelaskan mengenai analisa yang dibutuhkan dalam implementasi algoritma. Selain itu juga menjelaskan mengenai perencanaan langkah-langkah yang akan dilakukan dalam implentasi. Hasil dari analisa dan perancangan ini akan digunakan dalam implementasi sehingga dapat membuat implementasi lebih mudah dan terencana.

3.1 Analisa Kebutuhan

Seperti yang telah dijelaskan sebelumnya, WEKA digunakan sebagai acuan dalam implementasi algoritma *data mining* dalam laporan ini. Oleh karena itu, analisa kebutuhan ini mengacu juga kepada WEKA. Adapun algoritma yang mengacu pada WEKA hanya algoritma *Cobweb*, sedangan WEKA tidak mengimplementasi algoritma *Iterate*.

Analisa kebutuhan ini dibagi menjadi 2 yaitu analisa *input* dan analisa *output.* Analisa *input* menjelaskan mengenai analisa terhadap *input*, sedangkan analisa *output* menjelaskan mengenai analisa *output* yang sesuai dengan kebutuhan.

3.1.1 Analisa *Input*

Input yang digunakan untuk aplikasi ini adalah *input* hasil *preprocessing*. *Input preprocessing* tersebut bisa berupa *database* dan *file*. *File* dan *database* tersebut berisikan informasi dari setiap transakasi yang terjadi. Transaksi tersebut disimpan dalam bentuk angka maupun karakter. Proses *preprocessing* akan mengubah bentuk tersebut menjadi bentuk yang dapat diproses oleh aplikasi yang akan dibuat ini. Hasil *preprocessing* dari *database* dan *file* berupa kumpulan data yang tiap *attribute*nya berupa angka dan dipisahkan dengan karakter koma (,).

Angka-angka tersebut merupakan perwakilan dari tiap nilai yang berasal dari *file* maupun *database*. Namun, proses *preprocessing* bukan merupakan bagian dari implementasi yang dilakukan oleh penulis.

Sebagai contoh pada subbab 2.3.4 digunakan tabel untuk mengubah *attribute-attribute* “berparu-paru atau tidak”, “tempat hidup” dan “jenis makanan” menjadi angka-angka. Berikut ini adalah contoh hasil *preprocessing* dari contoh pada subbab 2.3.4

2,1,2

2,3,2

2,2,3

2,3,2

1,1,2

Dari hasil *input* tersebut, setiap barisnya merupakan sebuah *instance*. Setiap *instance* merupakan data yang berisikan sejumlah nilai. Banyaknya nilai suatu baris menandakan banyaknya *attribute* dari *instance* tersebut. Dan setiap *attribute* pada suatu baris harus memiliki 1 buah nilai saja.

Tiap *instance* akan menyimpan nilai-nilai tersebut dalam suatu *attribute*. Yang perlu diperhatikan adalah apakah menyimpan *instance* tersebut dalam bentuk implementasi *instance* atau tidak. Mengingat jumlah data yang biasa digunakan dalam *data mining* sangat besar, berarti implementasi *instance* memerlukan memori yang besar dan akan menyebabkan proses menjadi lebih lambat karena memori untuk proses menjadi sedikit jika memori yang disediakan terbatas jumlahnya.

Oleh karena itu, tiap baris tidak diimplementasikan dalam struktur data *instance*, namun hanya dibaca dari *file* hasil *preprocessing* dan disimpan dalam bentuk *array* sehingga lebih menghemat *memory*.

3.1.2 Analisa *Output*

Analisa *output* untuk algoritma *Cobweb* sama dengan analisa *output* untuk algoritma *Iterate*. Hal ini dikarenakan alogritma *Cobweb* dan *Iterate* memiliki kesamaan dalam implementasinya, yaitu menggunakan klasifikasi *tree*.

Output yang dihasilkan menggambarkan suatu klasifikasi *tree* dengan menampilkan tiap *instance-instance* pada tiap *cluster* dalam klasifikasi *tree*. Tiap *instance* ditampilkan agar tiap *cluster* memberikan informasi lebih jelas mengenai *cluster* tersebut anggota dari *cluster* mana. Tiap *cluster* juga menampilkan berapa persentase kontribusinya terhadap seluruh *cluster* dalam klasifikasi *tree* tersebut.

Contoh:

0 100%

[1,2,6,3]

[4,6,9,0]

[2,6,2,5]

[4,6,9,0]

1 50%

[4,6,9,0]

[4,6,9,0]

2 25%

[4,6,9,0]

2 25%

[4,6,9,0]

1 25%

[1,2,6,3]

1 25%

[2,6,2,5]

3.2 Analisa dan Perancangan Algoritma

Subbab ini akan dijelaskan mengenai analisa algoritma *Cobweb* dan algoritma *Iterate*. Analisa dilakukan pada algoritma sehingga dapat diketahui bagaimana algoritma *Cobweb* dan *Iterate* diimplementasi dengan baik dan menghemat memori. Dalam subbab ini juga dijelaskan mengenai proses penyimpanan nilai dari suatu *attribute* sehingga dapat diproses.

3.2.1 Membaca Data dan Analisa Penyimpanan

Seperti yang dijelaskan sebelumnya, tiap baris di *file* hasil *preprocessing* merupakan sebuah *instance* berisikan sejumlah *attribute*. *Instance* ini dianggap sebagai sebuah obyek yang akan di*cluster*. Oleh karena jumlah dari *instance* dalam sebuah *file* sangat banyak, perlu cara penyimpanan yang baik.

Ketika *file* hasil *preprocessing* dibaca, kemudian setiap nilai dari *attribute* disimpan ke dalam sebuah *array*. Penyimpanan dalam sebuah *array* lebih menghemat memori daripada mengimplentasikan *instance* untuk menyimpan nilai *attribute*. Jumlah *attribute* untuk disimpan dari sebuah *array* sebelumnya telah ditentukan sehingga tidak perlu lagi proses menentukan jumlah *attribute* pada pembacaan baris pertama dari *file*.

Demikian juga dengan jumlah baris dari *file* tersebut. Semua baris dalam *file* dibaca hingga selesai. Kemudian *instance-instance* dalam bentuk *array* tersebut disimpan dan dibuat *pointer-pointer* yang menunjuk ke masing-masing *instance*. Seluruh *pointer* yang menunjuk kepada *instance-instance* tersebut juga disimpan dalam sebuah *array*. Sehingga sebenarnya keseluruhan *instance* membentuk sebuah *array* multidimensional dengan ukuran n x m, dimana n adalah jumlah data dan m adalah jumlah *attribute.* Bentuk penyimpanan ini digunakan pada kedua algoritma. Berikut contoh penyimpanan *preprocessing*

Data:

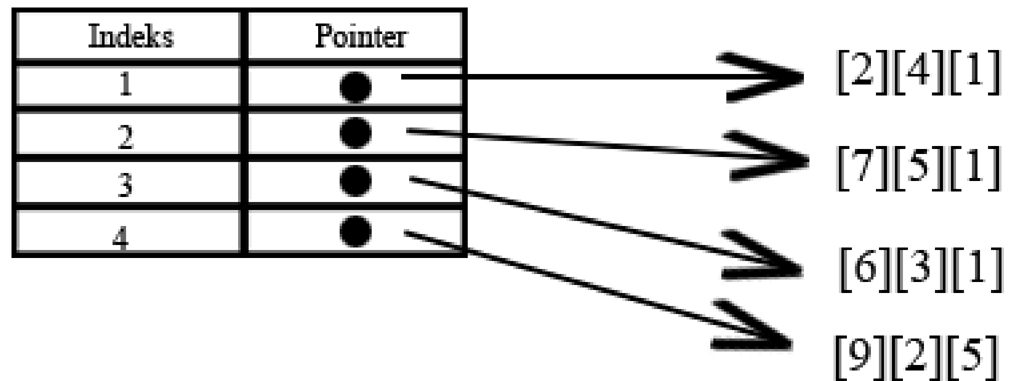
2,4,1

7,1,5

6,1,3

9,2,5

maka *instance-instance* tersebut disimpan seperti pada tabel 3.1



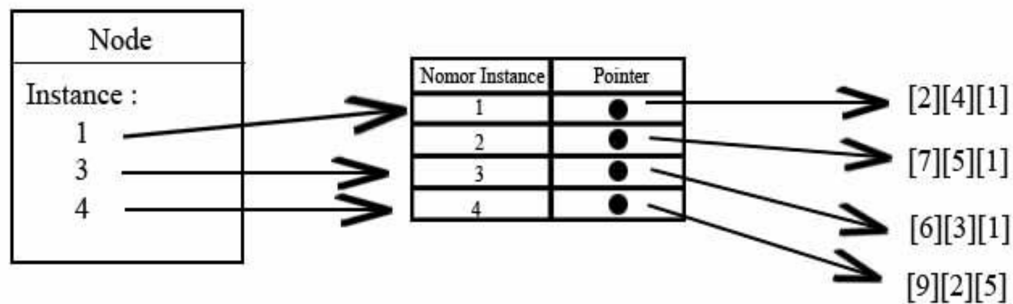
Gambar 3.1 Contoh penyimpanan *instance*.

3.2.2 Analisa *Node*

Node mewakili sebuah *cluster* pada sebuah klasifikasi *tree.* Sebuah *node* dapat menjadi anak dan orang tua dari *node y*ang lain sehingga dapat membentuk sebuah klasifikasi *tree*. Setelah mencari berbagai sumber informasi, disimpulkan bahwa ada beberapa hal yang perlu diperhatikan mengenai sebuah *node,* antara lain:

a. Dalam sebuah *node*, disimpan sejumlah *instance* yang menjadi anggota dari *node* tersebut. *Instance* yang disimpan berupa *pointer* ke kumpulan *instance-instance* yang telah disimpan. Lebih jelasnya lihat gambar 3.2.

b. Sebuah *node* memiliki seluruh *instance node* anaknya sehingga *node root* memiliki semua *instance* dan merupakan *node y*ang paling umum, sedangkan *node* daun (*leaf*) merupakan *leaf* yang paling khusus. Untuk menghemat *memory*, setiap anak dari *node* disimpan dengan menggunakan *pointer*. Lebih jelasnya lihat gambar 3.3. *Node* 1 memiliki 3 anak yaitu *node* 2, *node* 3, dan *node* 4. Sedangkan *node* 3 memiliki 2 anak yaitu *node* 5 dan *node* 6. Karakteristik *node* inilah yang membentuk klasifikasi *tree*.



Gambar 3.2 Contoh *instance-instance* yang disimpan *node.*

c. Sebuah *node* juga memiliki probabilitas dalam klasifikasi *tree*. Nilai probabilitas *node* juga disimpan dalam *node* tersebut dan nilai ini akan digunakan untuk menghitung nilai *CU* sebuah *node.* Lebih jelasnya lihat subbab 2.2.

d. Jumlah minimal *node yang* ada dalam sebuah klasifikasi *tree* akhir sebanyak n+1 *node*. Hal ini terjadi jika semua *instance* yang di*cluster* berada pada *cluster* yang berbeda satu sama yang lain.

Node akan digunakan baik dalam algoritma *Cobweb* maupun algoritma *Iterate*.

3.2.3 Analisa *Attribute*

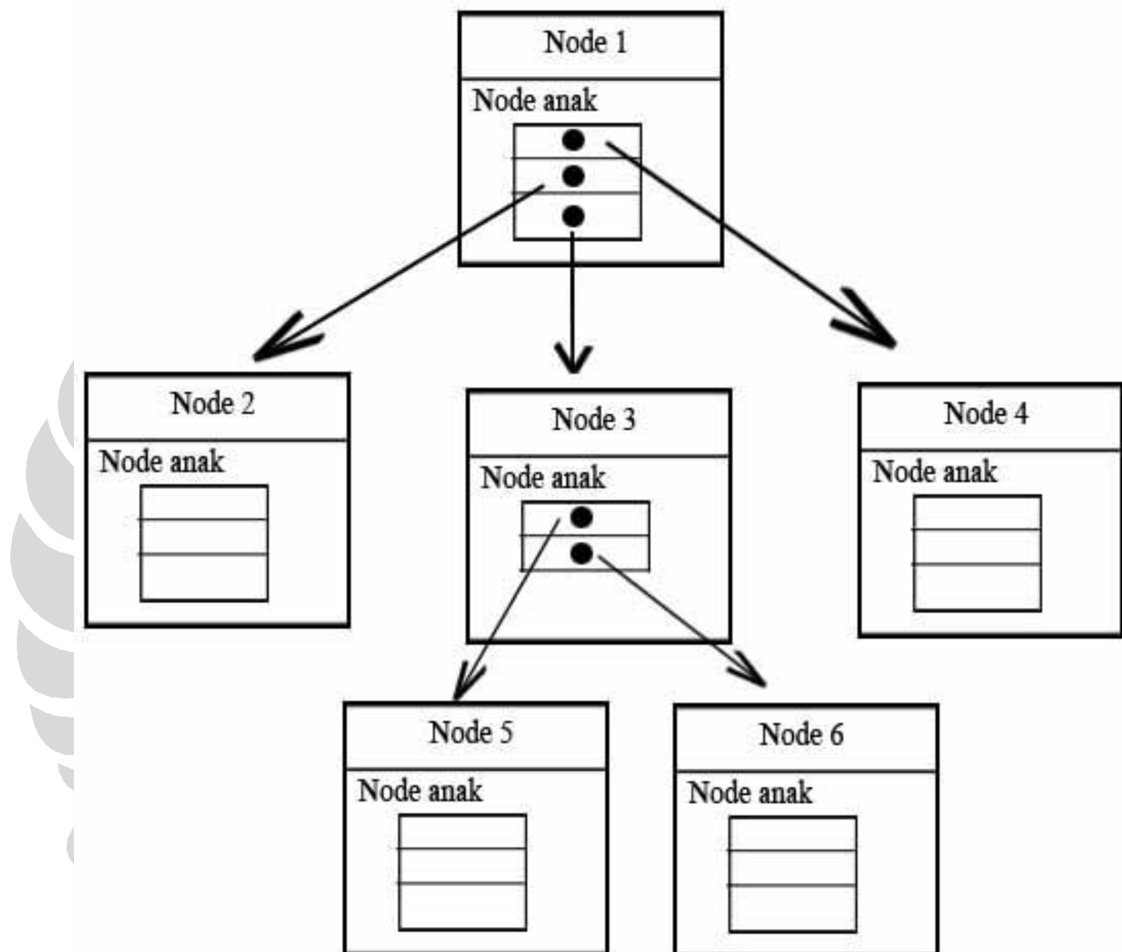
Dalam menyimpan nilai-nilai dari sebuah *attribute* pada sebuah *node*, ada beberapa hal perlu diperhatikan, antara lain:

a. Sebuah *attribute* memiliki nilai dan memiliki jumlah kemunculan nilai tersebut. Semua pasangan nilai-jumlah kemunculan pada suatu *attribute*, disimpan pada *attribute* yang bersesuaian. Tiap nilai yang disimpan pada *attribute* juga memiliki nilai probabilitas, nilai ini digunakan untuk perhitungan *CU* dan *PU* suatu *node*. Untuk lebih jelasnya lihat contoh penyimpanan *attribute*, data pertama, *attribute* pertama bernilai 3. Data ketiga *attribute* pertama bernilai 3. Total kemunculan nilai 3 ada 2 dari 3

Implementasi algoritma clustering..., Adolf Pandapotan

buah *instance*, sehingga yang disimpan pada *attribute* 1 yaitu [3][2][0,66]. Begitu juga dengan nilai dari *attribute-attribute* lainnya.

b. Sebuah *attribute* dalam sebuah *node* harus dibedakan agar dapat diketahui apa nilai sebenarnya dari *attribute* tersebut. Pada contoh dibawah, *attribute* 1 beda dengan *attribute* 2. Begitu juga dengan *attribute* 3.



Gambar 3.3 ***Node-node*** **yang memiliki anak dan membentuk klasifikasi *tree.***

Contoh untuk penyimpanan *attribute*

Data:

3,4,1,8

2,6,2,7

3,6,2,6

Attribute 1 = [3][2][0,66], [2][1][0,33]

Attribute 2 = [4][1][0,33], [6][2][0,66]

Attribute 3 = [1][1][0,33], [2][2][0,66]

Attribute 4 = [8][1][0,33], [7][1] [0,33],[6][1] [0,33]

3.2.4 Analisa Algoritma *Cobweb*

Analisa algoritma *Cobweb* pada Tugas Akhir ini menggunakan referensi utama dari pembuat algoritma ini (Fisher, 1987). Oleh karena itu, algoritma *Cobweb* yang akan diimplementasi merupakan algoritma murni.

Berikut ini analisa mengenai algoritma *Cobweb*

a. Menyiapkan *attribute*

Tujuan dari proses ini yaitu menentukan jumlah varian dari tiap *attribute*. Setiap *attribute* memiliki sejumlah varian. Perhitungan jumlah varian dari tiap *attribute* ini sangat penting untuk mengefisiensikan *memory*. Sebagai contoh

Data:

2,1,4

3,1,2

10,7,2

5,2,6

10,2,5

2,2,5

Implementasi algoritma clustering..., Adolf Pandapotan

5,7,6

Dari data diatas jika menggunakan interval 1 dari tiap *attribute* maka perlu menyiapkan 9 buah kemungkinan data untuk *attribute* pertama, 7 buah untuk *attribute* kedua dan 5 buah untuk *attribute* ketiga. Namun jika menggunakan varian dari *attribute*, untuk *attribute* pertama hanya membutuhkan 4 buah kemungkinan data saja, 3 buah dan 4 buah untuk *attribute* kedua dan ketiga. Perhitungan jumlah *attribute* ini menjadi *input* untuk pembentukan *attribute* baru.

b. Analisa *Cobweb*

Dalam algoritma *Cobweb*, proses pembentukan klasifikasi *tree* dimulai pada menentukan *root* awal dari sebuah klasifikasi *tree*. *Node root* awal berisikan *instance* yang terdapat pada baris pertama *file* hasil *preprocessing*. Baris pertama dari *file* disimpan langsung dalam *node root* tanpa harus melihat baris yang lain. Selain itu ada 4 buah fungsi yang digunakan dalam algoritma *Cobweb* untuk membentuk klasifikasi *tree*, yaitu

- Memasukan *instance* baru ke suatu *node*

  Fungsi ini dilakukan ketika *instance* baru dimasukan suatu *node y*ang menyebabkan *node* orang tuanya memiliki nilai *PU* terbesar. Lebih lengkapnya lihat pada subbab 2.3.3.

- Menggabungkan *node* (*merge node*)

  Fungsi ini dilakukan ketika suatu 2 *node* anak terbaik digabungkan dan dimasukan *instance* yang baru sehingga menyebabkan nilai *PU node* orang tuanya terbesar dari kemungkinan fungsi yang lain. Namun ketika suatu *node* hanya memiliki 2 buah anak dan ternyata dilakukan *merge* terhadap 2 *node* anaknya, bisa terjadi *loop* yang tak berhenti untuk iterasi selanjutnya karena *node y*ang diiterasi selanjutnya merupakan *node merge*. Untuk lebih jelasnya lihat gambar 3.4. Ketika sebuah *node* mempunyai 2 anak yaitu *node x* dan *node y*, nilai *PU* tertinggi terjadi jika kedua *node*

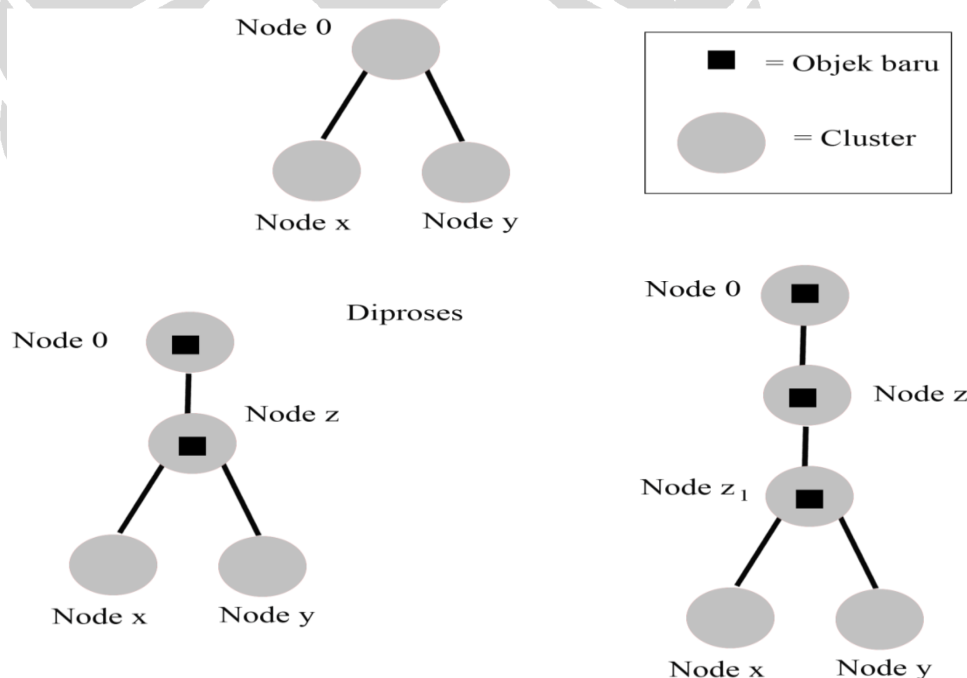
anak digabung dan menghasilkan *node z* , namun iterasi selanjutnya, pada *node z* juga terjadi nilai tertinggi ketika *node x* dan *node y* digabung, dilakukan proses *merge* menghasilkan *node* z_1, begitu seterusnya. Oleh karena itu, ketika terjadi sebuah *merge* dalam suatu *node*, untuk iterasi selanjutnya operasi *merge* tidak boleh dilakukan, sedangkan selanjutnya lagi operasi *merge* boleh dilakukan.

- Memecah *node* (*split node*)

  Fungsi ini digunakan untuk membuang sebuah *node* dan menggantinya dengan *node* anak-anaknya. Yang perlu diperhatikan adalah jika sebuah *node* memiliki tidak memiliki anak, maka *split* tidak dapat dilakukan.

- Membuat *node* baru (*new node*)

  Fungsi ini dilakukan untuk membuat sebuah *node* baru dan memasukan *instance* baru kedalamnya.



Gambar 3.4. Proses *loop* yang mungkin terjadi pada operasi *merge node*.

Dalam menentukan apakah suatu *instance* disimpan pada suatu *node* anak, *node* hasil gabungan 2 *node* terbaik (*merge*), *node y*ang dipecah (*split*) atau pada

node baru, dipilih berdasarkan nilai *PU* terbesar dari keempat kemungkinan tersebut. Lebih jelasnya lihat subbab 2.3.4.

Namun nilai *PU* terbaik yang didapat dalam proses pembentukan klasifikasi *tree* dari keempat kemungkinan diatas terkadang sangat kecil, oleh karena itu untuk menjaga agar nilai *PU* cukup besar maka digunakan suatu nilai batas yaitu nilai *cutoff*. Hal ini diperlukan karena jika nilai *PU* suatu *cluster* sangat kecil, maka *cluster* tersebut memiliki kualitas yang buruk. Jika nilai *PU* terbesar lebih kecil dari pada nilai *cutoff*, maka *instance* tersebut disimpan pada *node* terbaik karena menandakan hubungan *instance* tersebut dengan *instance-instance* yang ada pada *node* terbaik lebih dekat daripada *instance-instance* pada *node y*ang lain.

3.2.5 Analisa Algoritma *Iterate*

Algoritma *Iterate* yang diterapkan pada Tugas Akhir ini adalah algoritma murni *Iterate*. Adapun sampai saat ini penulis belum menemukan implementasi algoritma ini. Referensi yang digunakan hanya paper yang dibuat oleh pembuat algoritma *Iterate* (Biswas, Weinberg, Fisher, 1995).

Berikut ini adalah analisa algoritma *Iterate*

- **Menyiapkan *attribute***

  Tujuan dari proses ini adalah menentukan jumlah varian dari tiap *attribute*. Setiap *attribute* memiliki sejumlah varian. Adapun analisa ini sama seperti yang analisa pada algoritma *Cobweb*.

- **Analisa *Iterate***

  Pada awal algoritma *Iterate*, *node root* memiliki semua *instance* yang akan di*cluster*. Dari *instance-instance* dalam *node root* inilah dibuat klasifikasi *tree*. Begitu juga *node-node* anak dari setiap *node* dalam proses pembentukan klasifikasi *tree*. Pada subbab 2.9, ada 3 langkah dalam algoritma *Iterate*. Berikut adalah analisa untuk tiap langkah

a. Pembentukan klasifikasi *tree*

Pada pembentukan klasifikasi *tree* ini, jika *node* memiliki 1 *instance*, maka tidak ada pengembangan klasifikasi *tree* untuk *node* tersebut. Sebaliknya, jika lebih dari 1 *instance*, maka dilakukan pengembangan klasifikasi *tree* untuk *node* tersebut. Jika nilai *PU* untuk menyimpan *instance* ke *node* anak sama dengan nilai *PU* jika dimasukan ke *node* baru, maka yang dipilih adalah menyimpan pada *node* anak.

Dalam pengembangan klasifikasi *tree*, ada fungsi ADO untuk mengurutkan *instance-instance* dalam sebuah *node*. Penulis mencari sumber tentang ADO di internet dan menggirim *email* kepada penemu algoritma tersebut, tapi penulis tidak mendapatkan informasi apapun. Oleh karena itu, analisa hanya menggunakan definisi dari paper sumber algoritma *Iterate* tentang ADO untuk membentuk fungsi ADO (Biswas, Weinberg, Fisher, 1995). Untuk mengurutkan *instance-instance* tersebut berdasarkan nilai Manhattan *distance* dengan ADO, penulis menggunakan algoritma *heap sort* untuk mengurutkan *instance-instance* tersebut. Penulis menggunakan *heap sort* untuk mengurutkan *instance-instance* karena *heap sort* memiliki kompleksitas O (*n* log *n*) (Lang, 2000) dan menurut penulis mudah diimplementasikan. Dibandingkan dengan algoritma mengurutkan yang lain, *heap sort* cukup baik untuk data yang sangat besar.

Algoritma *Iterate* sama seperti *Cobweb* yaitu menggunakan klasifikasi *tree*. Oleh karena itu *Iterate* juga menggunakan *node* sebagai representasi *cluster*.

b. Pengumpulan partisi awal yang baik dari klasifikasi *tree* untuk membentuk *cluster-cluster* yang diinginkan

Input untuk langkah kedua ini adalah klasifikasi *tree* hasil dari langkah pertama. Klasifikasi *tree* menghasilkan nilai *CU* tiap levelnya seperti pada gambar 2.11. Analisa pada tahap ini yaitu jika *node* yang didapat pada langkah kedua ini memiliki 1 buah *instance*, atau tidak

memiliki anak, maka *node* tersebut tidak dimasukan ke dalam *initial partition* karena *node* tersebut adalah *node* daun, *node* ini merupakan *node y*ang paling khusus sehingga tetap berada pada klasifikasi *tree*. *Node-node* ,yang membentuk grup-grup *initial node*, yang diperoleh disimpan dalam bentuk *pointer*. Hal ini dilakukan untuk menghemat penggunaan memori daripada menyimpan *node y*ang sebenarnya. Hasil *output* dari langkah kedua ini adalah sebuah *array* yang berisi *pointer-pointer* yang setiap *pointer*nya mengacu ke *node-node y*ang ingin diperbaiki nilai *CU*nya. Untuk penjelasan mengenai langkah ini pada subbab 2.9.

c. Proses iterasi terhadap obyek-obyek untuk memaksimalkan pemisahan antara *cluster*

Input untuk langkah ketiga ini adalah *array* yang berisi *pointer-pointer* hasil dari langkah kedua. Setiap *node y*ang ada pada *array pointer* tersebut diperbaiki nilai *CU*nya dengan cara mengiterasi *instance-instance*nya. Seperti yang dijelaskan pada subbab 2.4.3, setiap *instance* tersebut dimasukan ke dalam grup-grup lain lalu dihitung nilainya *CM*nya. Pada langkah ini kemungkinan terjadi pemindahan *instance* yang sebenarnya lebih baik jika pada *node* awalnya karena walaupun nilai *CM node* lain lebih besar dari *CM node* awal, belum tentu *instance* tersebut baik berada di *node* itu. Oleh karena itu sebelum dilakukan pemindahan, *instance-instance* pada *node y*ang akan diperbaiki terlebih dahulu diurutkan dengan fungsi ADO.

Jika pada proses ini suatu *node* akhirnya memiliki 1 buah *instance* dan *instance* tersebut paling baik jika ada pada grup lain, mana *node* tersebut dibuat dari *tree*. Pada saat pemindahan *instance* dari suatu grup dilakukan, maka *instance* tersebut dihilangkan oleh *node* orang tua grup tersebut hingga *node root* agar tidak ada penggandaan *instance*, kemudian pada grup yang baru *instance* tersebut ditambahkan pada *node* orang tua grup tersebut hingga pada *node root*. Selain itu dilakukan iterasi dari

initial grup (*node* paling atas dari grup) hingga level terbawah dari grup untuk menentukan dimana *instance* tersebut disimpan.

3.3 Permasalahan Analisa dan Perencanaan

Penulis dalam menganalisa algoritma *Iterate* dan *Cobweb* mendapat masalah dalam mencari sumber-sumber informasi. Paper acuan yang digunakan penulis terbit pada tahun 1987 untuk *Cobweb* dan tahun 1998 untuk *Iterate*. Kesulitan dalam mencari sumber *Cobweb* adalah sulitnya pemahaman penulis mengenai *machine learning* dan keterbatasan *paper* pendukung dari penemunya serta contoh aplikasinya. Adapun pada WEKA, algoritma *Cobweb* yang diimplementasikan telah dimodifikasi terlebih dahulu. Namun penulis mengatasi keterbatasan masalah ini dengan mencari paper lain yang secara implisit menjelaskan mengenai *Cobweb* serta penulis melakukan percobaan dengan WEKA. Permasalahan pada *Iterate* juga sama. Untuk mengatasi hal tersebut, penulis mencari *paper* lain yang berhubungan dan berdiskusi dengan pembimbing Tugas Akhir.

BAB 4

IMPLEMENTASI ALGORITMA

Pada bab ini akan dijelaskan mengenai implementasi dari algoritma *Cobweb* dan algoritma *Iterate*. Implementasi algoritma *Cobweb* dilakukan pada 4 bagian dari algoritma tersebut antara lain fungsi evaluasi heuristik, representasi *state*, operator, dan strategi kontrol. Sedangkan pada algoritma *Iterate*, implementasi dilakukan pada 3 buah langkah dalam algoritma tersebut, yaitu pembentukan klasifikasi *tree* (*classified*), pengambil partisi awal untuk perbaikan klasifikasi *tree*(*extract*), dan proses iterasi perbaikan klasifikasi *tree*(*iterative redistribution*). Implementasi ini juga diharapkan penulis menghasilkan *output* yang sesuai yang diharapkan. Selain implementasi 2 algoritma tersebut dijelaskan mengenai implementasi membaca data dan tampilan sistem.

4.1 Membaca data

Pada subbab ini akan dijelaskan bagaimana membaca data dari *file* hasil *preprocessing*. Pembacaan data dari *file* ini sama-sama digunakan dalam kedua algoritma. Ada dua ide yang menurut penulis dapat dilakukan dalam pembacaan data dari *file*, yaitu setiap baris dibaca kemudian diproses dengan algoritma atau semua baris dalam *file* dibaca dan disimpan *instance*-*instance*nya kemudian diproses. Ide pertama hanya bisa dilakukan pada algoritma *Cobweb*, namun tidak untuk algoritma *Iterate* karena sangat sulit untuk membentuk klasifikasi *tree* pada langkah pertama dan juga tidak bisa dilakukan perhitungan jumlah *instance* serta varian dari seluruh *attribute*. Oleh karena itu, penulis memilih langkah kedua. Dalam sebuah *file* terdapat baris-baris yang menandakan sebuah transaksi. Pembacaan data dilakukan setiap baris. Setiap membaca baris, data-data tersebut disimpan dalam sebuah *array*. Besar dari sebuah *array* sesuai dengan banyaknya *attribute* yang ingin disimpan. Kemudian sebuah *pointer* akan menunjuk kepada *array* tersebut. *Pointer* inilah yang akan disimpan pada sebuah *array pointer*. Menurut sumber ("*Vector"*, 1993), penggunaan *vector* untuk menyimpan kumpulan *array* akan berpengaruh pada performa dari implementasi karena

array-array tersebut akan digunakan selama proses algoritma berlangsung dan vector tersebut akan memakan memori yang cukup besar. Besar *array pointer* yang digunakan diasumsikan sebesar 100.000. Menurut penulis, jumlah ini cukup besar untuk banyaknya baris dalam sebuah *file*.

Setiap *array* dianggap sebuah *instance* atau obyek yang akan di*cluster*. Berikut ini adalah algoritma membaca data dari *file* hasil *preprocessing* pada dua algoritma tersebut:

```
1  ListInstance = new array of pointer;
2  SumOfAttribute = n;
3  SumOfInstance = 0;
4  ListVarian = new array of pointer;
5  for int i = 0 up to SumOfAttribute
   a. ListVarian[i] = new array of integer;
   b. set all value of ListVarian[i] as 0;
6  end for
7  while File != EOF
   a. index = 0;
   b. row = read line of File;
   c. instance = new array of integer with size n;
   8  while index is not equal to SumOfAttribute
   a. instance[index] = row[index];
   b. index = index + 1;
   c. ListVarian[row[index]] = ListVarian[row[index]]+1;
   d. end while
9  ListInstance[sumOfInstance] = pointer to instance;
10 SumOfInstance = SumOfInstance + 1;
11 end while
12 SumOfVarianAttribute = array of integer;
```

```
13  for int i = 0 up to SumOfAttribute
    a.  SumOfVarianAttribute [i] = 0;
    b.  for int j = 0 up to end of listVarian[i]
        1 if ListVarian[i][j] is not equal to 0
        2 SumOfVarianAttribute[i]= SumOfVarianAttribute[i]+1;
        3 end if
    c.  end for
14  end for
```

Misalkan *dataset* seperti berikut

3,1,2,5

3,6,2,8

3,5,6,2

7,2,4,9

Jumlah *attribute* sebanyak 4 buah. Pertama-tama dibuat *array pointer* serta properti yang menyimpan jumlah *pointer*, misalkan ListInstance dan SumOfInstance (nomor 1 dan 3 pada algoritma membaca *file*) dan menyiapkan properti untuk menyimpan varian dari data (nomor 4-5). Baris pertama yaitu [3,1,2,5] di loop untuk disimpan dalam sebuah *array*, misalkan *array* tersebut *ArrayIns*, maka

ArrayIns[0] bernilai 3

ArrayIns[1] bernilai 1

ArrayIns[2] bernilai 2

ArrayIns[3] bernilai 5.

Array ini akan disimpan dalam bentuk *pointer* dalam ListInstance dan karena baris pertama, maka disimpan dalam ListInstance[0]. Properti SumOfInstance bertambah 1. Kemudian untuk tiap *attribute* pada *index* nilai *attribute* tersebut maka ListVarian[*index attribute*][nilai *attribute*] ditambahkan 1. Sebagai contoh untuk *instance* pertama

ListVarian[0][3] = 1

ListVarian[1][1] = 1

ListVarian[2][2] = 1

ListVarian[3][5] = 1.

Dan jika dimasukan *instance* kedua maka

ListVarian[0][3] = 2

ListVarian[1][1] = 1

ListVarian[1][6] = 1

ListVarian[2][2] = 2

ListVarian[3][5] = 1

ListVarian[3][8] = 1.

Begitu seterusnya hingga baris terakhir. Setelah selesai maka dilakukan perhitungan jumlah varian *attribute*, pada nomor 14-15 jika nilai pada ListVarian [i][j] tidak sama dengan 0 (nol) maka jumlah varian *attribute* I (SumOfVarianAttribute) bertambah 1, dimana pada ListVarian[i][j], i adalah *attribute* ke I, dan j adalah *index* ListVarian[i].

Hasil akhirnya seluruh didapat seperti pada tabel 4.1. Dan jumlah varian untuk *attribute* pertama berjumlah 2, *attribute* kedua berjumlah 4, *attribute* ketiga berjumlah 3, dan *attribute* keempat berjumlah 4. *Array pointer* ini nantinya akan

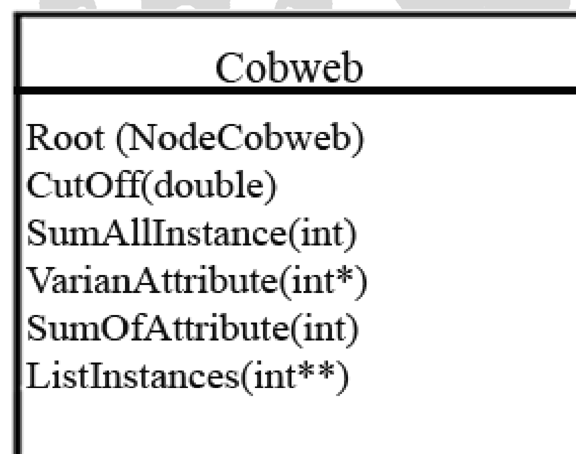
diimplementasikan sebagai ListInstances pada struktur data *Cobweb* dan *Iterate*. Dimana jika ListInstances[0] berarti menunjuk pada *instance* [3][1][2][5].

Tabel 4.1 Hasil akhir *array pointer*.

Pointer	*Instance*
0	[3][1][2][5]
1	[3][6][2][8]
2	[3][5][6][2]
3	[7][2][4][9]

4.2 Algoritma *Cobweb*

Seperti yang dijelaskan sebelumnya bahwa algoritma *Cobweb* terbagi menjadi 4 bagian. Namun sebelum dijelaskan mengenai 4 bagian tersebut, terlebih dahulu dijelaskan mengenai struktur data dari *Cobweb*. Gambar 4.1 menunjukan struktur data dari *class Cobweb*.



Gambar 4.1 Struktur data *Cobweb*.

Berikut penjelasan struktur data *Cobweb*

- *Root*: *node* awal yang memiliki semua *instance*.

- SumAllInstance: jumlah semua *instance*.
- SumOfAttribute: jumlah *attribute*.
- VarianAttribute: jumlah varian dari tiap *attribute*. VarianAttribute ini merupakan sumOfVarianAttribute pada algoritma membaca data pada subbab 4.1.
- ListInstances: kumpulan seluruh *instance*.
- *Cutoff:* nilai *cutoff*.

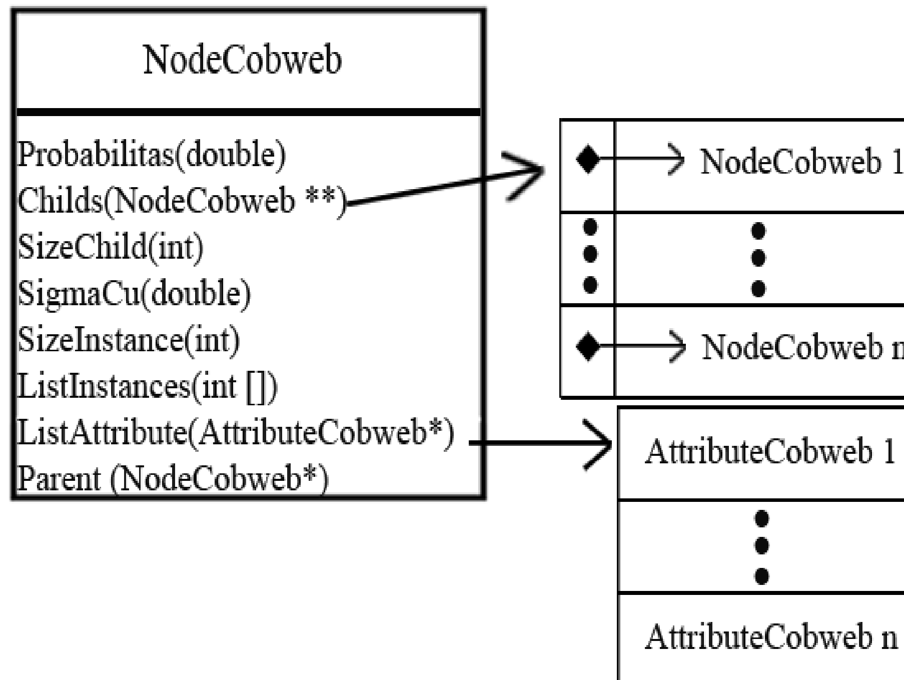
Berikut adalah implementasi 4 bagian dari *Cobweb*

4.2.1 Representasi *State*

Representasi *state* diimplementasikan sebagai *node*. Seperti dijelaskan pada subbab analisa 3.2.2, *node* harus dapat menyimpan *instance, node* anaknya, memiliki *attribute* dan *probabilitas*, maka *node* diimplementasi dengan *struct*. Gambar 4.2 merupakan struktur data dari *node Cobweb* yaitu NodeCobweb.

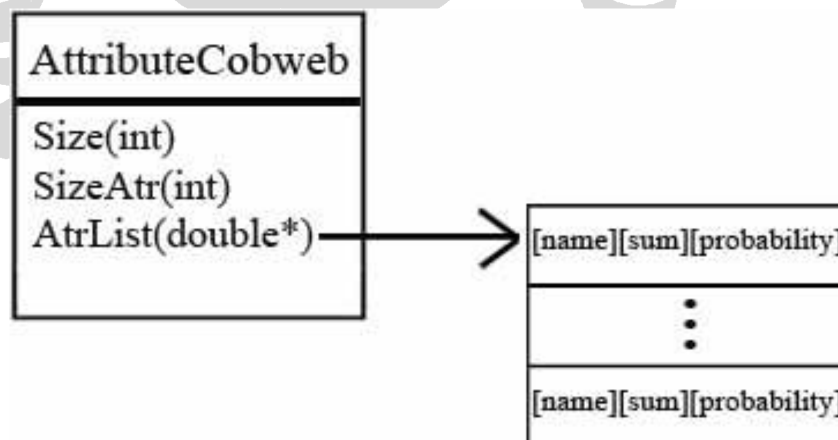
Berikut ini adalah penjelasan mengenai struktur data *node*

- Probabilitas: probabilitas *node* pada struktur klasifikasi. Pada fungsi *CU*, probabilitas ini dilambangkan dengan $P(C_k)$, dimana C_k adalah *cluster k.*
- Childs: kumpulan *pointer-pointer* yang menunjuk kepada *node-node* anak.
- SizeChilds: jumlah *node* anak.
- SigmaCu: nilai dari *CU node* sebelum dikali dengan probabilitas *node*. Ini digunakan untuk menghitung *PU* ketika suatu *instance* sementara disimpan pada *node* anak maupun *node* baru.
- SizeInstances: jumlah semua *instance*.
- ListInstances: kumpulan *instance-instance*, berisikan posisi (*index*) *instance* dalam ListAllInstance pada struktur data *class Cobweb*.



Gambar 4.2 Struktur data NodeCobweb.

- ListAttribute: berisikan *pointer-pointer* yang menunjuk kepada AttributeCobweb-AttributeCobweb *node*.
- *Parent*: *pointer* yang menunjuk *node* orang tuanya.



Gambar 4.3 Struktur data AttributeCobweb.

Sedangkan untuk *attribute*, implementasinya yaitu struktur data AttributeCobweb. Gambar 4.3 merupakan struktur data AttributeCobweb. Berikut penjelasannya

- Size: jumlah dari semua nilai. Digunakan untuk menghitung probabilitas dari suatu nilai.
- SizeAtr: jumlah dari varian nilai yang ada pada AttributeCobweb.
- AtrList: kumpulan *pointer-pointer* yang menunjuk kepada *array* anggota AttributeCobweb.

Tabel 4.2 ***Node*** **yang dimasukan 2** ***instance*** **yaitu [3][1][2][5] dan [3][5][6][2].**

<table>
<tr><td colspan="4">NodeCobweb</td></tr>
<tr><td colspan="4">Probabilitas(1), hanya contoh</td></tr>
<tr><td colspan="4">Childs{ }</td></tr>
<tr><td colspan="4">Sizechilds(0)</td></tr>
<tr><td colspan="4">SigmaCu(0), hanya contoh</td></tr>
<tr><td colspan="4">Parent{ }</td></tr>
<tr><td colspan="4">ListAttribute</td></tr>
<tr><td>Attribute pertama</td><td>Attribute kedua</td><td>Attribute ketiga</td><td>Attribute keempat</td></tr>
<tr><td>Size = 2
SizeAtr = 1
AtrList = ([3][2][1])</td><td>Size = 2
SizeAtr = 2
AtrList = ([1][1][0,5]), ([5][1][0,5])</td><td>Size = 2
SizeAtr = 2
AtrList = ([2][1][0,5]), ([6][1][0,5])</td><td>Size = 2
SizeAtr = 2
AtrList = ([2][1][0,5]), ([5][1][0,5])</td></tr>
<tr><td colspan="4">ListInstance {0,2}</td></tr>
<tr><td colspan="4">SizeInstance(2)</td></tr>
</table>

Contoh 2 buah *instance* pada tabel 4.1, [3][1][2][5] dan [3][5][6][2], dengan jumlah *attribute* 4 buah dimasukan ke dalam *node*. *Instance* pertama, [3][1][2][5] dimasukan, nilai (*name*) *attribute* pertama yaitu 3, jumlahnya (*sum*) 1 buah, probabilitasnya (*probability*) 1, disimpan sebagai [3][1][1]. Jumlah *attribute* (Size) dan SizeAtr sebanyak 1. Begitu juga dengan nilai 1 pada *attribute* kedua, nilai 2 pada *attribute* ketiga dan nilai 5 pada *attribute* keempat. Hasil akhir setelah *instance* kedua dimasukan seperti pada tabel 4.2.

4.2.2 Fungsi evaluasi heuristik

Seperti yang telah dijelaskan sebelumnya, fungsi evaluasi heuristic digunakan untuk menghitung nilai *CU* dan *PU* dari sebuah *node*. Fungsi *CU* dihitung terhadap *attribute-attribute* yang dimiliki oleh sebuah *node*. Sedangkan fungsi *PU* menghitung nilai rata-rata *CU* dari sebuah *node*. Penjelasan mengenai fungsi *CU* dan *PU* ada pada subbab 2.3.1. Berikut ini implementasi fungsi *CU* dan PU

```
11  cu_function(Node node){
12  n = SumOfAttribute;
13  Result = 0;
14  Boolean exist = false;
15   For i = 0 up to n
16     For k = 0 up to Size of each attribute of parent of node
17       For j = 0 up to Size of each attribute of node
18            If AtrList[j][0] of Attribute[i] in node equals to AtrList[k][0] of
    Attribute[i] in parent of  node
19         X = probability in AtrList[j][2] of Attribute[i]in node;
20         Y = probability in AtrList[k][2] of Attribute[i]in parent of node;
21         exist = true;
22         Result = Result + (X^2 – Y^2);
23        End if
24       End for
```

```
25      If exist is false
26        Y = AtrList[j][0] of Attribute[i] in parent of node;
27        Result = Result + (0^2 – Y^2);
28      End if
29      exist = false;
30     End for
31   End for
32  SigmaCu of node = Result;
33  Return probability of node * Result;
34  }
35  pu_function(Node node){
36  Result = Minimum of Double;
37  For each child of node;
38  Result = Result + cu_function(child node);
39  End For
40  Return Result / sum of child of node;
41  }
```

4.2.3 Operator

Seperti yang telah dijelaskan sebelumnya pada subbab 2.3.3, operator dalam algoritma ini ada 4 buah. Keempat operator ini diimplementasikan dalam fungsi-fungsi. Berikut adalah fungsi-fungsi operator

- Memasukan obyek ke dalam *node*

  Operator ini menyimpan *pointer*nya saja dan memperbaharui *attribute-attribute*nya dari memasukan *instance* baru.

```
1  addInstance(Node thisNode, Instance ins){
2  Create pointer to Instance ins and save that pointer in Node thisNode;
3  Update attribute for adding Instance ins;
```

```
4   }
```

- Membuat *node* baru (*new node*)

    Operator ini membuat *node* baru sebagai anaknya dan memasukan *instance* baru kedalamnya. *Node* anak ini disimpan sebagai *pointer*.

```
1   newNode(Node thisNode, Instance ins){
2   Create new node as child of thisNode;
3   Add Instance ins to new node and update attribute;
    }
```

- Mengkombinasikan 2 *node* menjadi 1 *node* (*merge node*)

    Operator ini membuat *node* baru kemudian menyimpan semua *instance* dari 2 *node* parameternya, dan *instance* baru serta memperbaharui *attribute-attribute*nya.

```
1   Node mergeNode(Node first, Node second, Instance ins){
  2       Create fusionNode as new Node third as parent of Node first and
          Node second;
  3       Update attribute of fusionNode for add all Instance of Node first,
          Node second and Instance ins;
  4       Return fusionNode;
5   }
```

- Membagi sebuah *node* menjadi beberapa *node* (*split node*)

```
1   splitNode(Node bestNode){
2            Remove Node bestNode from tree;
3            Promote children of Node bestNode to parent of Node node;
4   }
```

Selain keempat fungsi diatas, diimplementasi juga fungsi untuk mengekspansi sebuah *node*. Operator ini membuat 2 *node* anak baru, *node* pertama memiliki semua *instance node* orang tuanya, sedangkan *node* kedua memiliki *instance* baru.

```
1  expandNode(Node thisNode,Instance ins){
2  Create firstNode as new node that contains thisNode's instances;
3  Add firstNode as child of thisNode;
4  Create secondNode as new node that contains Instance ins;
5  Add secondNode as child of thisNode;
6  }
```

4.2.4 Strategi Kontrol

Seperti yang sudah dijelaskan, strategi kontrol digunakan untuk membentuk klasifikasi *tree*. Berikut ini implementasi dari strategi kontrol *Cobweb*

```
1  Input: New Instance newIns and nodes of the actual concept hierarchy N
2  Algorithm_COBWEB (Instance newIns, Node N){
3  IsUnlimitLoop = false;
4  IsLowerThanCutoff = false;
5  While(N is not NULL){
6  If Size of children N is 2
       a. IsUnlimitLoop = true;
7  End if
8  If N is a leaf node THEN
       a. PUExpand = PU-value fo expand Node N
       b. If PUExpand > CutOff
              1 ekspandNode(N,newIns);
       c. End if
       d. N = NULL;
9  Else
       a. BestPUValue = Minimum double;
       b. For all child of N
              1 Compute TempPU as PU-value of Node N when child i-th
```

add *instance* newIns;

2 If TempPU < BestPUValue

3 BestPUValue = TempPU;

4 End if

c. End for

d. Suppose BestNode is the successor *node* with highest *PU*-value PUBestValue;

e. Suppose SecondBestNode is the successor *node* with second highest *PU*-value;

f. Suppose PUNewNode is the PU-value for insertion of *Instance* newIns in some new *node* S;

g. Suppose PUMergingNode is the *PU*-value for merging P and R in a new *node* S and add *Instance newIns,* but if IsUnlimitLoop is true, PUMergingNode = Minimum double;

h. Suppose PUSplitingNode is the *PU*-value for splitting P;

i. BestAllPUValue is the best *PU*-value of PUBestValue, PUNewNode, PUMergingNode, PUSplitingNode;

j. If BestAllPUValue < CutOff;

1 IsLowerThanCutoff = true;

k. End if

l. If BestNode is the BestAllPUValue Or IsLowerThan*Cutoff* is true

1 addInstance(BestNode, newIns);

2 Insert the *Instance* newIns in *N* and update *attribute* of *N*;

3 If IsLowerThanCutoff is true

a. Return;

4 End if

5 *N* = BestNode;

6 IsUnlimitLoop = false;

7 BestAllPUValue = false;

m. Else if PUNewNode is the BestAllPUValue

1 newNode(*N*, newIns);

2 Insert the *Instance* newIns in *N* and update *attribute* of *N*;

3 *N* = NULL;

n. Else if PUMergingNode is the BestAllPUValue

1 FusionNode = mergeNode(BestNode, SecondBestNode, newIns);

2 Insert the *Instance* newIns in *N* and update *attribute* of *N*;

3 *N* = FusionNode;

o. Else if PUSplitingNode is the BestAllPUValue

1 splitNode(BestNode);

2 IsUnlimitLoop = false;

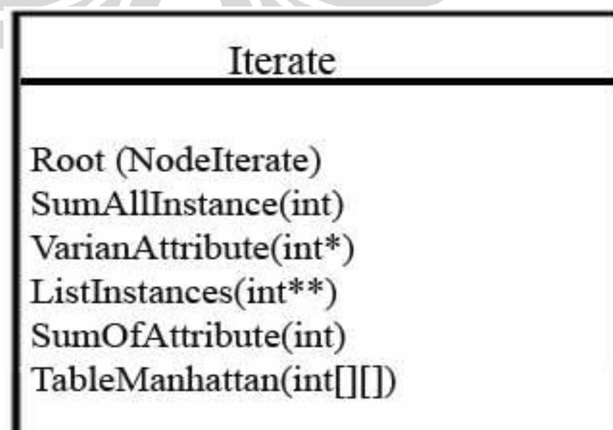
10 End if

11 }

12 }

4.3 Algoritma *Iterate*

Implementasi algoritma ini mirip dengan *Cobweb*. Pada gambar 4.4 merupakan struktur data *Iterate*. Seperti yang sudah dijelaskan sebelumnya pada subbab 2.4, algoritma *Iterate* menggunakan *node* sebagai *cluster*. Implementasi *node* pada *Iterate* hampir sama dengan *Cobweb* namun ada perbedaan yaitu properti UnsortInstance dan SumUnsortInstance. Untuk lebih jelasnya lihat struktur data NodeIterate pada gambar 4.5.



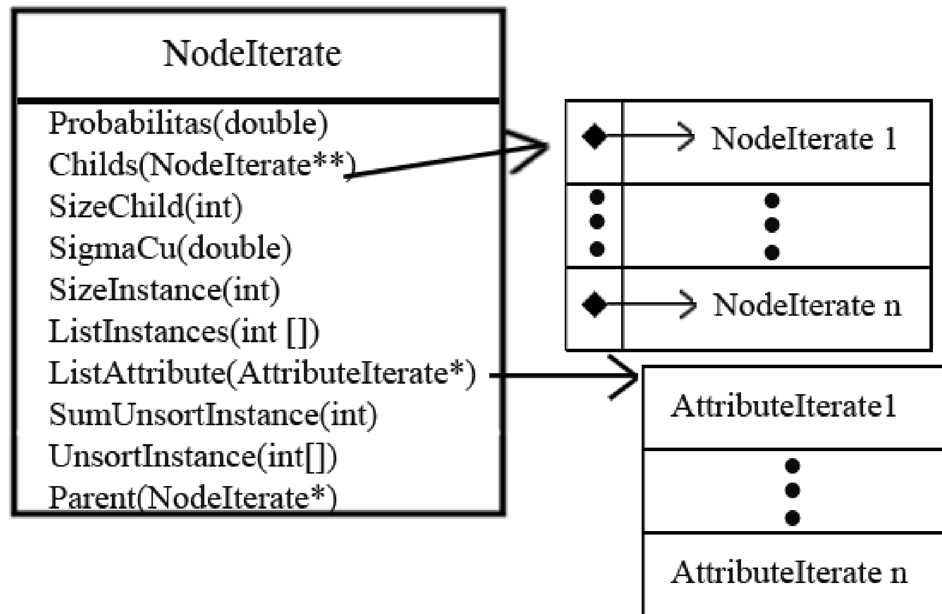
Gambar 4.4 Struktur data *Iterate*.

Berikut penjelasan struktur data *Iterate*

- *Root*: *node* awal yang memiliki semua *instance*.
- SumAllInstance: jumlah semua *instance*.
- SumOfAttribute: jumlah *attribute*.
- VarianAttribute: jumlah varian dari tiap *attribute*. VarianAttribute ini merupakan sumOfVarianAttribute pada algoritma membaca data pada subbab 4.1.
- ListInstances: kumpulan seluruh *instance*.
- TableManhattan: berisikan nilai-nilai dari jarak antara 2 buah *instance*. Sebagai contoh jika TabelManhattan[i][j] berarti jarak antara *instance* ke *i* dengan *instance* ke *j*. Lihat subbab 4.1.

Untuk menghitung nilai Manhattan *distance* antara dua buah *instance* digunakan fungsi manhattanDist dengan parameter 2 buah *instance*. Fungsi ini digunakan pada saat membaca *file*.

```
1  Int manhattanDist(Instance firstIns, Instance secondIns){
2  Result = 0;
3    For int i = 0 up to  SumOfAttribute
4      If firstIns[i] > secondIns[i]
5         Result = Result + (firstIns[i] - secondIns[i]);
6      Else
7         Result = Result + (secondIns[i] - firstIns[i]);
8      End if
9    End for
10 Return Result;
11 }
```



Gambar 4.5 Struktur data NodeIterate

Berikut penjelasan struktur data NodeIterate

- Probabilitas: probabilitas *node* pada struktur klasifikasi. Pada fungsi *CU*, probabilitas ini dilambangkan dengan $P(C_k)$, dimana C_k adalah *cluster k*.
- Childs: kumpulan *pointer-pointer* yang menunjuk kepada *node-node* anak.
- SizeChilds: jumlah *node* anak.
- SigmaCu: nilai dari *CU node* sebelum dikali dengan probabilitas *node*. Ini digunakan untuk menghitung *PU* ketika suatu *instance* disimpan pada *node* baru.
- SizeInstances: jumlah semua *instance*.
- ListInstances: kumpulan *instance-instance*, berisikan posisi (*index*) *instance*.
- ListAttribute: berisikan *pointer-pointer* yang menunjuk kepada *AttributeIterate-AttributeIterate node*.
- SumUnsortInstance: jumlah *instance* yang belum diurutkan fungsi ADO.

- UnsortInstance: kumpulan *instance* yang belum diurutkan fungsi ADO.
- *Parent*: *pointer* yang menunjuk ke *node* orang tuanya.

Untuk *attribute* pada *Iterate*, struktur datanya sama seperti AttributeCobweb hanya nama saja yang berbeda. Nama dari *attribute* tersebut yaitu AttributeIterate. AttributeIterate diimplementasikan dengan *struct*.

Implementasi algoritma ini dilakukan pada 3 buah langkahnya. Berikut implementasi tiap langkah

4.3.1 Langkah pertama: Pembentukan klasifikasi *tree*(*classified*)

Pada langkah ini, pembentukan klasifikasi *tree* dibentuk dari *node-node*. Berikut implementasi langkah pertama algoritma *Iterate*

```
1   classified(Node root){
2   L = new array of Node;
3   L[0] = root;
4   While L is empty
        a. Node thisNode = first element in L;
        b. If size of instances in n bigger than 1
        c. Sort instances of n that sorted by function ADO;
        d. For int i = size of instance in Node thisNode up to 0
                1 Ins = instance[i];
                2 If last instance
                3 Create new node as child of n and add the instance to that
                  node;
                4 Else
                5 BestNode = 0;
                6 BestPuNode = 0;
                7 For int i = 0 up to size of children Node thisNode
                    a. PuTemp = PU Node thisNode for add Ins to
                       child[i];
```

```
                    b.  If PuTemp > BestPuNode
                            1   BestPuNode = PuTemp;
                            2   BestNode = i;
              8 End for
              9 BestNewNode = PU score for add instance to a new child
                of Node thisNode;
             10 If BestPuNode >= BestNewNode
             11 addInstance(child[i],Instance);
             12 addInstance(n,Instance);
             13 Else
             14 newNode(n,Instance);
             15 Update attribute of Node n;
             16 End if
        e.  End for
        f.  Remove L[0];
        g.  Add new children in L;
5   End while
6   }
```

Dalam pembentukan klasifikasi diatas, diperlukan fungsi ADO, fungsi *CU* dan PU. Untuk fungsi *CU*, dalam pembentukan klasifikasi *tree* ini sama dengan fungsi *CU* pada *Cobweb*. Berikut fungsi ADO untuk pembentukan klasifikasi *tree* ini

```
1   Function_ADO(Instances){
2   List = Array of integer;
3   ListPosition = Array of integer;
4   For i = 0 up to size of Instances
        a.  Result = 0;
        b.  Instance pivot = Instances[i];
```

```
        c. For j = 0 up to size of Instances
              1 For k = 0 up to last attribute
              2 Result = Result + manhattanDistance(pivot,Instances[j]);
              3 End for
        d. End for
        e. List[i] = Result;
        f. ListPosition[i] = i;
   5  End for
   6  Sort ListPosition with heap sort according to List;
   7  Return ListPosition;
}
```

Fungsi berikut digunakan membuat *node* baru sebagai anaknya dan memasukan *instance* baru kedalamnya. *Node* anak ini disimpan sebagai *pointer*.

```
1  newNode(Node thisNode, Instance ins){
2  Create new node as child of thisNode;
3  Add Instance ins to new node and update the attribute;
   }
```

Fungsi berikut digunakan untuk menyimpan *instance* baru ke sebuah *node*. *Instance* disimpan sebagai *pointer*.

```
1  addInstance(Node thisNode, Instance ins){
2  Create pointer to Instance ins and save that pointer in Node thisNode;
3  Update Attribute for adding Instance ins;
4  }
```

4.3.2 Langkah kedua: Pengumpulan partisi awal yang "baik" dari klasifikasi *tree* (*extract*)

Berikut implementasi langkah kedua

```
1  extract(){
2  ListNode = Array of Node;
3  ListNode[0] = Root;
4  ListInitNode = Array of Node;
5  SizeInit = 0;
6  InitialNode = root;
7  While ListNode is empty
      a. InitialNode = ListNode[0];
      b. If CU of ListNode[0] > CU of all child of ListNode[0]
           1 ListInitNode[SizeInit] = ListNode[0];
           2 SizeInit = SizeInit + 1;
      c. Else
           1 For int i = 0 up to size child of ListNode[0]
                a. If CU of ListNode[0] ≤ CU child [i] of ListNode[0]
                      1   Add children of CU child [i] to ListNode
                b. Else
                      1   ListInitNode[SizeInit] = ListNode[0];
                      2   SizeInit = SizeInit + 1;
                c. End if
           2 End for
           3 Remove ListNode[0];
8  End while
9  }
```

4.3.3 Langkah ketiga: *Iterative redistribution*

Untuk langkah ketiga, *input*nya adalah ListInitNode dan SizeInit dari langkah kedua. Berikut adalah hasil implementasinya

```
1   iterativeRedistribution(){
2   if SizeInit is equal to 2
3   return;
4   End if
5   For int i = 0 up to SizeInit
6   PivotNode = ListInitNode[i];
7   Sort instances of PivotNode with Function_ADO;
8   BestNode = i;
9   BestCu = CU of PivotNode;
10  For int j = 0 up to size of PivotNode's instances
11    For int k = 0 up to SizeInit
12       If k is  not equal to i
13          CUTemp = CU for add instance[j] in ListInit[k];
14          If CUTemp > BestCu
15              BestNode = k;
16              BestCu = CUTemp;
17       End if
18     End For
19   If i is not equal to BestNode
20       Remove instance j from parent PivotNode until root;
21       Remove instance j from PivotNode until leaf, if leaf only have 1
    instance, remove leaf from tree;
22      Add instance j from parent ListInit[BestNode] until root;
23      InitNode = ListInit[BestNode];
24      IsEnd = false;
25      While size of childs InitNode is not equal to 0 And IsEnd is true
```

```
26            Compute PU for add instance j to all child of InitNode;
27            Compute PU for add instance j to a new node of InitNode;
28            Assign instance j to node for which the PU score is highest and
     InitNode, and update attribute for InitNode;
29          If instance is added to a new node
30              IsEnd = true;
31          End if
32             End for
33        End while
34    End if
35  End for
36  End for
37  }
```

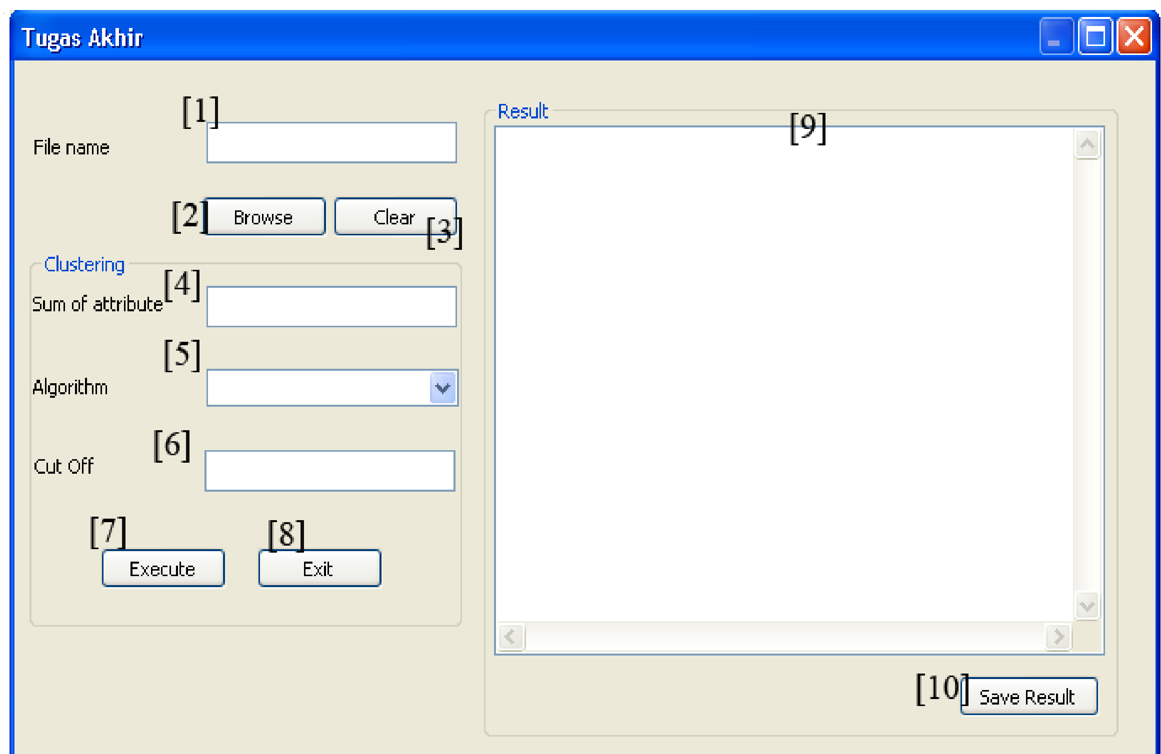
4.4 Tampilan sistem

Program dibuat dengan *input* berupa *file txt* dan menghasilkan klasifikasi *tree*. Gambar 4.6 adalah tampilan sistem.

Berikut penjelasan gambar 4.6

- Angka 1: Nama *file input*.
- Angka 2: Tombol untuk mencari *file input.*
- Angka 3: Tombol untuk menghapus nama *file.*
- Angka 4: Jumlah *attribute*.
- Angka 5: Jenis algoritma. Ada 2 pilihan yaitu *Cobweb* dan *Iterate*.
- Angka 6: Nilai *cutoff*. Nilai ini hanya untuk *Cobweb*.
- Angka 7: Tombol untuk memproses data.
- Angka 8: Tombol untuk keluar program.

- Angka 9: Layar untuk menampilkan hasil.
- Angka 10: Tombol untuk menyimpan hasil. Format untuk menyimpan yaitu *txt*.



Gambar 4.6 Tampilan sistem.

BAB 5

HASIL UJI COBA IMPLEMENTASI

Pada bab ini akan dijelaskan hasil uji coba terhadap program implementasi. Pengujian dilakukan dengan menggunakan *dataset* kecil dan besar. Hasil pengujian program ini akan dibanding dengan hasil yang didapat dengan WEKA. Namun hasil yang dibandingkan dengan WEKA hanya hasil dari program *Cobweb* saja karena *Iterate* tidak diimplementasi oleh WEKA. Untuk uji coba program *Iterate*, hasilnya dibandingkan dengan hasil program *Cobweb*. Untuk *dataset* kecil, dilakukan perbandingan kualitas dari *cluster-cluster* yang terbentuk pada *Cobweb* dengan WEKA dan *Cobweb* dengan *Iterate*, sedangkan untuk *dataset* yang besar, dilakukan perbandingan waktu eksekusi program *Cobweb* dengan WEKA dan program *Iterate*.

Hasil pengujian dilakukan dengan menggunakan komputer dengan spefisikasi sebagai berikut:

Processor : Intel (R) Core2Duo T5500 @1.66 Ghz

Memory : 1024 MB

Harddisk : 80 GB

Operating System : Microsoft Windows XP Professional

5.1 Penjelasan Umum Hasil Pengujian

Dataset yang digunakan dalam uji coba *Cobweb* dan *Iterate* merupakan data hasil *preprocessing* namun karena *preprocessing* bukan bagian dari Tugas Akhir ini, maka *dataset* yang digunakan dibuat oleh penulis. Sedangkan *dataset* untuk WEKA merupakan *dataset* awal yang bukan hasil *preprocessing* karena WEKA dapat melakukan *preprocessing*. Oleh karena itu, agar hasil dari WEKA dapat dibandingkan dengan *Cobweb*, maka *dataset* untuk *Cobweb* merupakan hasil *preprocessing dataset* untuk WEKA.

Seperti yang telah dijelaskan sebelumnya, pengujian dilakukan dengan menggunakan 2 jenis data yaitu *dataset* kecil dan *dataset* besar. *Dataset* kecil yaitu *dataset* yang dibuat oleh penulis. *Dataset* kecil untuk uji coba *Cobweb* dengan WEKA memiliki jumlah 10 data dengan 4 buah *attribute*. Pada pengujian *Cobweb* dengan WEKA, *dataset* kecil *Cobweb* terdiri dari 10 buah varian urutan data, hal ini dikarenakan WEKA menggunakan fungsi acak terhadap urutan data sebelum diproses.

Tabel 5.1 Isi *file* data_kecil.*arff*.

```
% Mengelompokan mahluk hidup
@relation weather
@attribute cara_bernapas {paru-paru, insang }
@attribute tempat_hidup {darat, air,udara}
@attribute berdarah {panas, dingin}
@attribute jenis_makanan {herbivora,karnivora,omnivora}
@data
paru-paru, udara, panas, karnivora
paru-paru, darat, panas, herbivora
paru-paru, darat, panas, karnivora
paru-paru, darat, panas, omnivora
paru-paru, air, panas, karnivora
insang, air, dingin, karnivora
paru-paru, darat, panas, karnivora
paru-paru, udara, panas, herbivora
insang, air, dingin, karnivora
paru-paru, darat, panas, herbivora
```

Dataset kecil pada uji coba perbandingan *Cobweb* dengan WEKA digunakan untuk mengecek kebenaran algoritma dan kualitas partisi. Pengecekan kebenaran algoritma dilakukan dengan membandingkan salah satu hasil uji coba *dataset* kecil pada *Cobweb* dengan WEKA, sedangkan untuk kualitas partisi hasil uji coba, digunakan rata-rata nilai *PU* untuk semua *cluster* yang memiliki *sub-sub cluster* (*node* anak).

Perhitungan rata-rata nilai *PU* juga dilakukan untuk membandingkan kualitas partisi-partisi *Cobweb* dan *Iterate.* Untuk uji coba *Iterate* dengan *Cobweb*, *dataset* kecil yang digunakan juga berjumlah 10 data dengan 4 buah *attribute*. Jumlah *dataset* kecil yang sedikit ini dipakai penulis karena jika lebih banyak, maka klasifikasi *tree* yang terbentuk semakin besar sehingga sulit untuk dicek kualitas partisinya. Untuk *dataset* yang besar memiliki jumlah data 1000 buah, 2000 buah, 3000 buah dan 4000 buah, dengan 4 buah *attribute*.

Hasil uji coba pada WEKA juga berbentuk klasifikasi *tree* namun tidak menampilkan *instance-instance* yang disimpan pada setiap *cluster*, oleh karena itu agar dapat dibandingkan dengan *Cobweb* maka penulis menampilkan *instance-instance* di setiap *cluster*nya.

Pada tabel 5.1 ditampilkan *file* "data_kecil.*arff*" sebagai *dataset* kecil untuk WEKA. Untuk mengetahui cara membaca format *file arff* dapat dilihat pada subbab 2.5. Urutan hewan pada baris 2 dan seterusnya sama dengan urutan data.

Dengan menggunakan tabel 5.2 maka didapat tabel 5.3 adalah *file* "data_kecil.*txt*". *Dataset* ini yang bersesuai dengan "data_kecil.*arff*" untuk algoritma *Cobweb* dan *Iterate*. Untuk varian *dataset* kecil *Cobweb*, dapat dilihat pada lampiran A hingga J. *File* data_kecil.*txt* merupakan *dataset* A_1 pada lampiran A.

Tabel 5.2 Tabel acuan untuk “data_kecil.*txt*” dan “data_kecil.*arff*”.

Attribute pertama	*Attribute* kedua	*Attribute* ketiga	*Attribute* keempat
Paru-paru = 1, Insang = 2.	Darat = 1, Air = 2, Udara = 3.	Panas = 1, Dingin = 2.	Herbivora = 1, Karnivora = 2, Omnivora = 3.

Tabel 5.3 Isi *file* data_kecil.*txt*.

1,3,1,2 1,1,1,1 1,1,1,2 1,1,1,3 1,2,1,2 2,2,2,2 1,1,1,2 1,3,1,1 2,2,2,2 1,1,1,1

5.2 Hasil Pengujian *Dataset* Kecil

5.2.1 Algoritma *Cobweb*

Pada uji coba ini, nilai *cutoff* yang digunakan sebesar 0, agar tidak terjadi pemotongan *cluster*. Untuk lampiran A hingga I, klasifikasi *tree*nya tidak ditampilkan karena akan sangat banyak. Pada lampiran K memuat salah satu hasil *dataset* kecil (*dataset* A_{10} pada lampiran J) pada implementasi *Cobweb* untuk mengecek kebenaran algoritma *Cobweb* yang diimplementasi. Sedangkan lampiran L merupakan hasil uji coba *dataset* kecil pada WEKA. Untuk lampiran L dan K, ada tambahan penamaan *cluster* untuk digunakan pada analisa hasil uji

coba. Dari kedua hasil tersebut dapat disimpulkan bahwa algoritma *Cobweb* pada Tugas Akhir ini berbeda dengan algoritma *Cobweb* pada WEKA. Algoritma *Cobweb* pada WEKA bukan algoritma murni, karena mengimplementasikan operator-operator lain selain 4 operator pada algoritma *Cobweb* murni. Operator-operator tersebut antara lain *split-plus new leaf* (proses *split* terhadap *node* terbaik dan menambah *node* baru) dan *split plus merge best* (proses *split node* terbaik ,menggabungkan 2 *node* terbaik dari *node-node* anaknya menjadi suatu *node* dan menambahkan *instance* ke *node* tersebut).

Nilai rata-rata *PU* untuk 10 varian *dataset* kecil adalah sebagai berikut

Tabel 5.4 Rata-rata *PU dataset* A_1, A_2, A_3, A_4, A_5.

Dataset	A_1	A_2	A_3	A_4	A_5
Rata-rata *PU*	**0,1682**	**0,1709**	**0,1703**	**0,1612**	**0,1682**

Tabel 5.5 Rata-rata *PU dataset* A_6, A_7, A_8, A_9, A_{10}.

Dataset	A_6	A_7	A_8	A_9	A_{10}
Rata-rata *PU*	**0,18467**	**0,1601**	**0,17029**	**0,18467**	**0,1708**

Dari tabel 5.4 dan 5.5, nilai rata-rata *PU* untuk seluruh *dataset* adalah 0,1709. Sedangkan nilai rata-rata *PU* WEKA sebesar 0,2036. Dari kedua nilai tersebut, dapat disimpulkan kualitas partisi WEKA lebih baik dari *Cobweb*.

Walaupun secara menyeluruh kualitas partisi WEKA lebih baik, nilai *PU* untuk *root* pada WEKA (0.1667) lebih kecil dibandingkan pada program *Cobweb* (0.444). Ini menandakan bahwa partisi *root* pada *Cobweb* lebih baik dibandingkan WEKA. Hal ini disebabkan karena pada hasil WEKA, *instance* hewan [1, 2, 1, 2] (paru-paru, air, panas, karnivora) dikelompokan pada *cluster* 13, padahal akan

lebih baik jika dimasukan kedalam *cluster* 2 seperti pada hasil program *Cobweb* (*cluster* 2 dan *cluster* 5).

5.2.2 Algoritma *Iterate*

Seperti yang telah dijelaskan sebelumnya, uji coba pada *Iterate* dilakukan untuk membandingkan kualitas partisi program *Iterate* dengan *Cobweb*. Nilai *cutoff* untuk *Cobweb* sebesar 0. Dari uji coba dengan *dataset* A_{10} diperoleh hasil seperti pada lampiran M. Kemudian dibandingkan kualitas partisi dari hasil uji coba *Cobweb* dan *Iterate* dengan *dataset* A_{10}. Diperoleh perbandingan seperti pada tabel 5.6.

Tabel 5.6 Perbandingan nilai-nilai *PU cluster* pada uji coba *Cobweb* dan *Iterate*.

Perbandingan	*Cobweb*	*Iterate*
Jumlah *cluster* yang terbentuk	18 *cluster*	12 *cluster*
Nilai *PU cluster-cluster* yang bukan *node* daun	*PU*(*Cluster* 1) = 0,45 *PU*(*Cluster* 2) = 0 *PU*(*Cluster* 5)= 0,2222 *PU*(*Cluster* 6)= 0,2222 *PU*(*Cluster* 7) = 0 *PU*(*Cluster* 11)= 0,25 *PU*(*Cluster* 14) = 0,2222 *PU*(*Cluster* 15) = 0	*PU*(*Cluster* 1) = 0,45 *PU*(*Cluster* 3)= 0,1958 *PU*(*Cluster* 4)= 0,2963 *PU*(*Cluster* 8)= 0,1867 *PU*(*Cluster* 9)= 0,222

Dari tabel 5.6 dilakukan perhitungan rata-rata nilai *PU* untuk *Cobweb* dan *Iterate* berturut turut sebesar 0,1708 dan 0.27016. Sehingga dapat disimpulkan

bahwa kualitas partisi program *Iterate* lebih baik dari program *Cobweb*. Selain itu, kekurangan *Cobweb* yaitu urutan mempengaruhi bentuk klasifikasi *tree*. Ini dapat dilihat pada uji coba dengan *dataset* lain. Berdasarkan tabel 5.5 dan tabel 5.4, nilai rata-rata *PU* pada *dataset* A_1 hingga A_{10} berbeda, sedangkan pada *Iterate* untuk *dataset* A_1 hingga A_{10} nilai rata-rata *PU* tetap sama. Hal ini dikarenakan fungsi ADO telah terlebih dahulu mengurutkan *instance-instance* sebelum diproses. Kestabilan partisi inilah yang merupakan tujuan dari algoritma *Iterate*. Lebih jelasnya lihat subbab 4.3.

Selain itu dilakukan perhitungan untuk mengetahui seberapa besar perbedaan antar *cluster* hasil program *Cobweb* dan *Iterate*. Perhitungan dilakukan dengan fungsi *variance of distribution* terhadap *cluster* 5 pada hasil *Cobweb* dan *cluster* 3 pada hasil *Iterate*. *Cluster-cluster* tersebut dipilih karena memiliki *instance-instance* yang sama.

Nilai *variance of distribution* partisi *cluster* 6 terhadap *cluster* 11 sebesar 3,055, sedangkan *cluster* 6 terhadap *cluster* 14 sebesar 1,77, dan *cluster* 11 terhadap *cluster* 14 sebesar 2,055. Rata-rata nilai *variance of distribution* untuk *cluster* 5 pada *Cobweb* sebesar 2,29. Sedangkan pada *Iterate* nilai rata-rata *variance of distribution* partisi *cluster* 4 terhadap *cluster* 8 sebesar 2,314. Terbukti bahwa algoritma *Iterate* memaksimalkan perbedaan antara 2 buah *cluster*. Lihat subbab 4.3.

5.3 Hasil Pengujian *Dataset* Besar

5.3.1 Algoritma *Cobweb*

Seperti yang dijelaskan sebelumnya, pengujian juga dilakukan dengan *dataset* yang besar. Namun, untuk *dataset* yang besar perlu diperhatikan jumlah data dan *attribute* karena akan mempengaruhi lamanya waktu eksekusi program dan beban yang berlebihan pada komputer pengujian.

Untuk pengujian dengan *dataset* besar digunakan *dataset* dengan jumlah data 1000 sampai 4000 dengan interval 1000 data dan jumlah *attribute* 4 buah. Sama seperti *dataset* kecil, *dataset* besar juga menggunakan *attribute* yang sama.

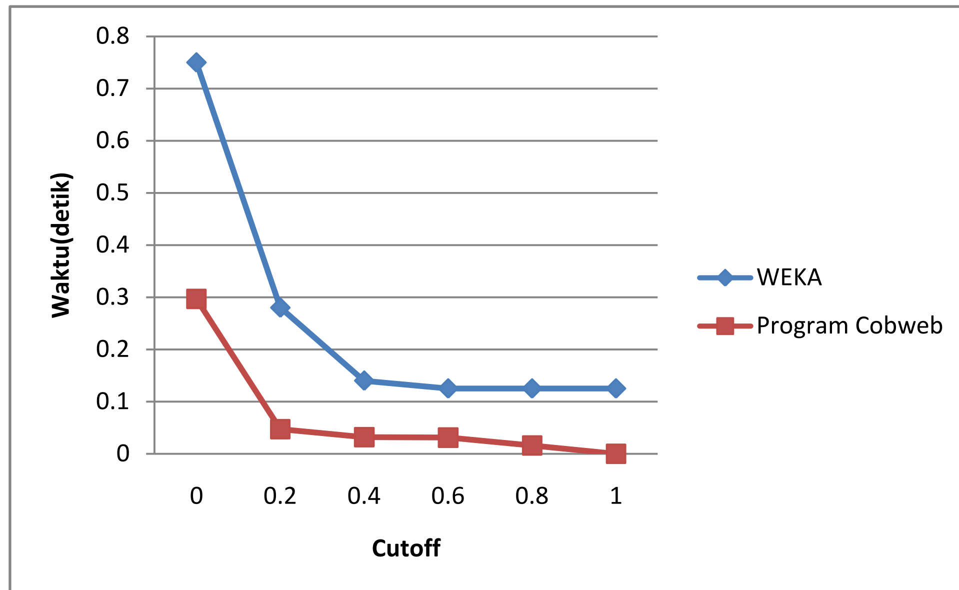
Pengujian pertama dilakukan dengan menggunakan nilai *cutoff* yang berbeda-beda. Nilai *cutoff* yang digunakan dimulai dari nilai *cutoff* 0 hingga 1 dengan interval 0,2 dan untuk setiap nilai *cutoff* dihitung lamanya waktu pemrosesan *dataset*. Tabel 5.6 merupakan hasil ujicoba *dataset* besar pada *Cobweb* dan WEKA.

Tabel 5.7 Hasil ujicoba WEKA dan *Cobweb* dengan jumlah data 1000.

Cutoff	Program *Cobweb*	WEKA
0	0,297 detik	0,75 detik
0.2	0.047 detik	0,28 detik
0.4	0,032 detik	0,14 detik
0.6	0,031 detik	0,125 detik
0.8	0,016 detik	0,125 detik
1	0 detik	0,125 detik

Dari gambar 5.1 dapat disimpulkan bahwa semakin besar nilai *cutoff* maka waktu eksekusi program WEKA maupun implementasi *Cobweb* semakin cepat. Untuk nilai *cutoff* 0 hingga 1, waktu eksekusi program *Cobweb* lebih cepat dari WEKA.

Kemudian dilakukan pengujian dengan meningkatkan jumlah data, mulai dari 1000 hingga 4000 dengan interval 1000. Nilai *cutoff* yang digunakan sebesar 0. Hasil dari pengujian ini dapat dilihat pada tabel 5.8.

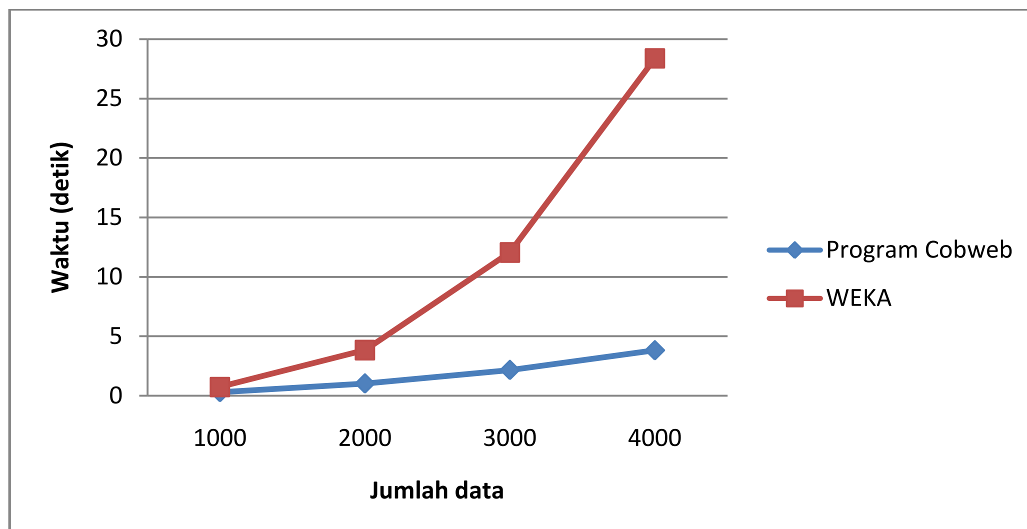


Gambar 5.1 Grafik perbandingan nilai *cutoff* dengan waktu eksekusi.

Table 5.8 Hasil ujicoba WEKA dengan *Cobweb* dengan nilai *cutoff* 0.

Jumlah Data	Program *Cobweb*	WEKA
1000	0,297 detik	0,75 detik
2000	1,031 detik	3,85 detik
3000	2,17 detik	12,062 detik
4000	3,828 detik	29,969 detik

Dari gambar 5.2 dapat dilihat bahwa untuk jumlah data yang besar, program *Cobweb* lebih cepat dari pada WEKA. Bahkan untuk besar data 3000 dibanding 4000, lama waktu eksekusi WEKA naik secara signifikan. Dari hasil tersebut didapat kesimpulan bahwa aplikasi dengan menggunakan bahasa Java lebih lambat dari pada bahasa C++.



Gambar 5.2 Grafik perbandingan jumlah data dengan waktu eksekusi dengan nilai *cutoff* 0.

5.3.2 Algoritma *Iterate*

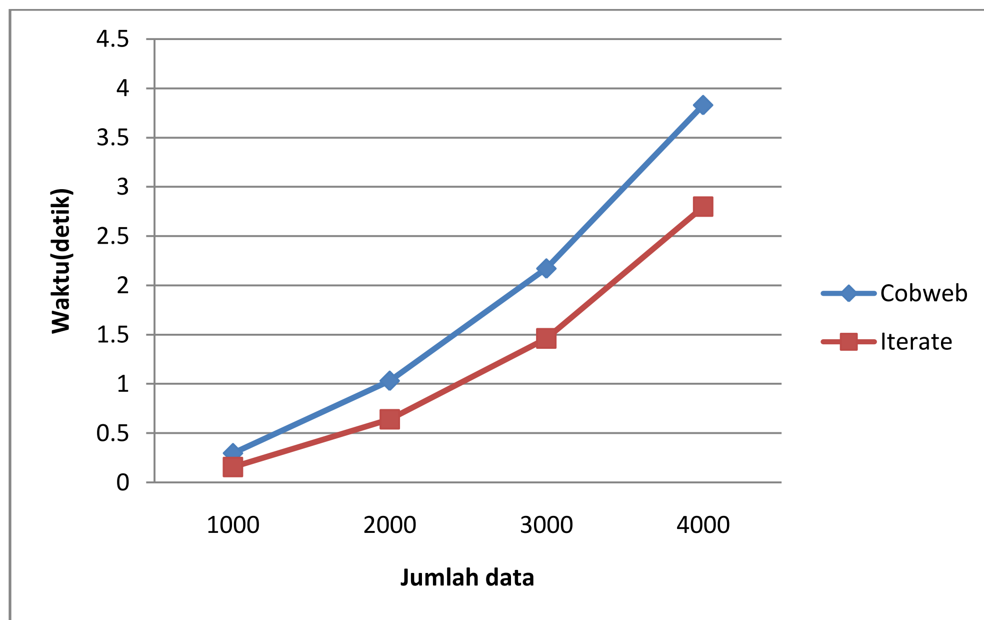
Untuk uji coba *dataset* besar pada *Iterate* menggunakan *dataset* yang sama seperti pada *Cobweb*. Namun pengujian dilakukan hanya terhadap perubahan jumlah data karena algoritma *Iterate* tidak menggunakan nilai *cutoff*. Oleh karena itu untuk dapat dibandingkan dengan program *Cobweb*, maka pengujian program *Cobweb* menggunakan nilai *cutoff* 0.

Dari pengujian diperoleh hasil seperti pada tabel 5.9.

Tabel 5.9 Hasil uji coba *dataset Cobweb* dan *Iterate*.

Jumlah data	*Cobweb*	*Iterate*
1000	0,297 detik	0,157 detik
2000	1,031 detik	0,641 detik
3000	2,17 detik	1,461 detik
4000	3,828 detik	2,787 detik

Dari gambar 5.3 diperoleh kesimpulan bahwa algoritma *Iterate* lebih cepat dari pada algoritma *Cobweb*. Hal ini dikarenakan algoritma *Iterate* pada pembentukan klasifikasi *tree* menggunakan 2 buah proses di setiap levelnya yaitu membuat *node* baru dari sebuah *instance* dan memasukan *instance* ke sebuah *node* yang sudah ada, sedangkan pada *Cobweb* terdapat 4 buah proses disetiap level *tree* untuk setiap *instance*. Untuk lebih jelasnya dapat dilihat pada subbab 2.5 tentang algoritma *Iterate*.



Gambar 5.3 Grafik perbandingan *Cobweb* dengan *Iterate*.