PERBANDINGAN ALGORITMA SHORTEST PATH DALAM PEMROSESAN CITRA DIGITAL SEAM CARVING

F. Alvin Sebastian[1] *florentinus.alvin@ti.ukdw.ac.id*

R. Gunawan Santosa[2] *gunawan@staff.ukdw.ac.id*

Theresia Herlina R.[3] *herlina@staff.ukdw.ac.id*

Abstract

Seam carving is a method of content aware image resizing. As solutions shortest path algorithms are used to find images seams. Seam is a horizontal or vertical path of animage that has minimum energy. There are two (2) shortest path algorithms that will be discussed in this paper. This paper contains the results of shortest path algorithms comparison between Dijkstra and Directed Acyclic Graph to see which one is better than another in case of efficiency. The precomputed and recomputed methods will be compared to find the more efficient method for executing the seam carving transformation. A web application has been built for this purpose. This web app is capable of transforming image size with seam carving method. The complexity of Dijkstra and Acyclic will be compared to find which one is better. The result is Dijkstra has been won, with the O(4V) with Acyclicis O(5V). The use of precomputed and recomputed is evaluated by the conditions. If the preparation is evaluated then recomputed is more efficient, but if the preparation is not evaluated then the precomputed method is the better one and has faster performance.

Keywords: *seam carving*, *shortest path*, *minimum energy*, *gradient magnitude*, perbandingan algoritma, Dijkstra, *acyclic*

1. Pendahuluan

Image scaling dan *cropping* merupakan metode yang kerap dipakai dalam melakukan *image resizing* atau pengubahan ukuran citra. Sebagian besar metode yang digunakan adalah *scaling*. *Scaling* merupakan pengubahan ukuran citra berdasarkan skala tanpa mempertimbangkan proporsi panjang dan lebarnya. *Scaling* juga tidak mempertimbangkan isi dari citra. *Cropping* terbatas karena hanya menghilangkan piksel pada citra dalam batasan area tertentu saja. *Image Resizing* yang lebih efektif dapat dicapai dengan mempertimbangkan isi dari citra dan bukan hanya batasan geometri saja (Avidan & Shamir, 2007).

Seam carving adalah sebuah metode untuk mengubah ukuran citra dalam pengolahan citra digital yang memperhatikan dan mempertimbangkan obyek yang terdapat dalam citra tersebut. Secara garis besar, proses pengolahan citra digital ini akan mencari bagian yang kurang penting dari citra berupa rute piksel (*path of pixels*) dan menghapusnya untuk mengecilkan ukuran citra atau menduplikasinya untuk memperbesar ukuran citra. Rute piksel yang kurang penting merupakan rute piksel dengan energi yang paling kecil. Rute piksel bisa didapatkan dari atas sampai bawah citra untuk mengubah panjang citra atau dari kiri ke kanan untuk mengubah tinggi citra. Proses untuk menentukan rute piksel atau seam akan diterangkan pada bagian landasan teori.

Algoritma untuk menentukan *seam* mana yang akan diproses yaitu Dijkstra dan *Acyclic Vertex -Weighted Graph*. Penelitian ini mengimplementasikan algoritma Dijkstra dan *Acyclic Vertex -Weighted* dalam memproses citra digital dan membandingkan algoritma-algoritma tersebut untuk mencari algoritma yang efisien. Penelitian ini juga akan membahas mengenai perbandingan antara *pre-computed seams* dan *recomputed seam*. *Pre-computed seam* disini didefinisikan sebagai pemakaian struktur data yang digunakan untuk menyimpan *seams of energy*. *Recomputed* adalah sistem akan mencari ulang seam baru untuk diproses. Jadi untuk *recomputed* sistem akan melakukan pencarian seam setiap kali ia melakukan proses *resize*.

[1] Program Studi Teknik Informatika, Universitas Kristen Duta Wacana, Yogyakarta
[2] Program Studi Teknik Informatika, Universitas Kristen Duta Wacana, Yogyakarta
[3] Program Studi Teknik Informatika, Universitas Kristen Duta Wacana, Yogyakarta

Menurut Low (1991) dan Teuber (1993), *seam carving* merupakan bagian dari pengolahan citra dan pemrosesan sinyal digital yang mempertahankan sifat penting pada obyek dalam suatu citra digital. Beberapa penelitian yang berhubungan dengan *seam carving* adalah Sarkar, Nataraj, dan Manjunath (2009) yang melakukan deteksi *seam carving* dan lokalisasinya. Sedangkan Thilagam dan Karthikeyan (2012) melakukan optimisasi ukuran citra dengan menggunakan *piece seam carving*.

Sedangkan pada penelitian ini akan dilihat perbandingan kompleksitas antara algoritma Djikstra dan Acyclic pada proses pencarian seam-seam yang mempunyai energi rendah. Metodologi penelitian dalam perbandingan kompleksitas dilakukan dengan menganalisis program dan *peseudocode* dari kedua algoritma tersebut sebab perbandingan pengamatan secara visual dari kedua algoritma tersebut sukar dilakukan.

2. Tujuan dan Manfaat Penelitian

Tujuan utama dari penelitian ini adalah untuk mengetahui hasil perbandingan implementasi algoritma shortest path Dijkstra dan *Acylic Vertex Weighted* dalam pemrosesan citra digital *seam carving* serta perbandingan implementasinya dalam *pre-computed seams* dan *recomputed seam*, sehingga dapat diperoleh metode yang lebih optimal. Penelitian ini juga bertujuan untuk membuat aplikasi berbasis web yang dapat digunakan sebagai alat pengolah citra digital sesuai dengan konteks penelitian.

3. Rumusan Masalah

1) Apakah Algoritma Dijkstra ataukah *Acyclic Vertex -Weighted Graph* yang lebih baik digunakan dalam proses *seam carving* dilihat dari kompleksitas algoritma tersebut?
2) Agar performa run-time sistem lebih cepat, diantara *precomputed seams* dan *recomputed seam* metode apakah yang lebih efisien untuk diimplementasikan?

4. Hipotesis

Ada dua hal yang menjadi hipotesis penulis perihal kompleksitas algoritma, hipotesis penulis adalah *Acyclic Vertex Weighted Graph* akan memiliki algoritma yang lebih kompleks karena algoritma tersebut akan mencari *topological order* terlebih dahulu sebelum melakukan proses pencarian *shortest path*. Adapun keterangan lebih lanjut mengenai *topological order* dapat dilihat dalam bagian 5.4 dalam landasan teori.

Dalam perbandingan *precomputed seams* dan *recomputed seam*, penulis memiliki hipotesis bahwa precomputed seams akan memiliki performa yang lebih cepat dibandingkan recomputed seams. Alasannya adalah karena precomputed seam akan mencari dan mensortir *seam-seam* yang akan digunakan sebagai acuan dalam melakukan *resize* sehingga tidak perlu melakukan pencarian ulang *shortest path* karena tiap seam sudah disimpan dalam sebuah struktur data yang akan disediakan untuk prosesini.

5. Landasan Teori

5.1. Seam Carving

Menurut penemunya yaitu Avidan dan Shamir (2007) *seam carving* adalah sebuah operator citra yang melakukan resize, baik itu mengecilkan atau membesarkan citra, dengan memperhatikan isi dari citra tersebut. Tujuan *seam carving* adalah mendapatkan sebuah citra berukuran baru dengan obyek yang menonjol dalam citra dalam citra masih jelas atau bahkan tidak terpengaruh oleh proses *resize*. Artinya dalam proses ini, identifikasi obyek merupakan hal yang vital. *Seam* sendiri adalah 8 *connected-path* yang optimal dari piksel-piksel dalam sebuah citra yang diambil dari atas ke bawah atau dari kiri ke kanan, dimana keoptimalan ditentukan oleh fungsi energi citra (Avidan & Shamir, 2007). Energi yang digunakan sebagai acuan dalam *seam carving* didapatkan dengan metode *gradient magnitude*. Citra akan dikonversi menjadi sebuah graf yang memiliki bo*bot* yang berupa bilangan energi terebut. Untuk mencari seam dengan bo*bot* terendah akan digunakan algoritma *shortest path* Djikstra dan *Acyclic Vertex Weighted Graph*.

Setelah *seam* didapatkan maka seam tersebut digunakan sebagai acuan untuk menghilangkan atau menduplikasi piksel-piksel untuk proses *resize*. Menghilangkan jika akan mengecilkan ukuran citra. Menduplikasi jika akan membesarkan ukuran citra. Setelah itu proses diulangi hingga didapatkan ukuran citra yang diinginkan.

5.2. Gradient Magnitude

Berdasarkan catatan dari David (2005), gradient dari sebuah citra mengukur perubahan dari citra. Perubahan tersebut memberikan dua macam informasi. *Gradient magnitude* dari tiap piksel akan disimpan dalam sebuah *vector* untuk kemudian dikomputasikan sebagai bo*bot* dari graf. *Gradient magnitude* dapat diperoleh dengan rumus:

$$I(x,y) = \sqrt{dx^2 + dy^2} \qquad [1]$$

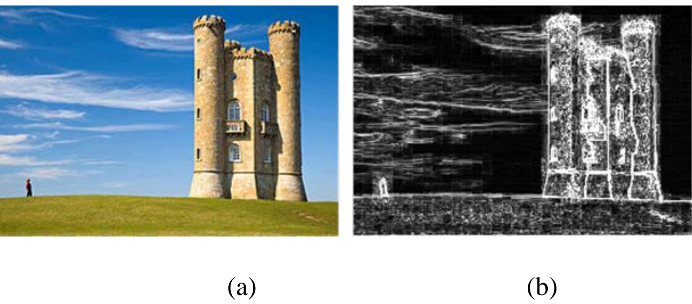
Dalam persamaan tersebut dx merupakan perubahan nilai *grayscale* piksel dengan arah x (*horizontal*), sedangkan dy perubahan pada arah y (vertikal). Sedangkan untuk mendapatkan nilai *grayscale* dari sebuah piksel citra RGB digunakan rumus luminosity:

$$I(r,g,b) = (0.2126 \times r) + (0.7152 \times g) + (0.0722 \times b) \qquad [2]$$

Rumus diatas akan dipakai untuk mendapatkan nilai *grayscale* yang digunakan untuk menghitung dx dan dy. Setelah nilai *grayscale* didapatkan, dx dan dy dapat diperoleh dengan rumus berikut:

$$\begin{aligned} dx &= I(x-1\,,y) - I(x+1,y) \\ dy &= I(x,y-1) - I(x,y+1) \end{aligned} \qquad [3]$$

Dalam persamaan tersebut dapat dilihat bahwa nilai dx dari piksel I(x,y) didapatkan dari nilai *gradient* piksel I(x-1, y) yaitu piksel di sebelah kirinya dan dikurangi dengan *gradient* piksel di sebelah kanannya yaitu I(x+1, y). Nilai dari dy didapatkan dengan mengurangkan piksel di atasnya (I(x, y-1)) dan piksel di bawahnya (I(x, y+1)). Gambar 1 ini merupakan contoh *gradient* magnitude citra:



(a) (b)

Gambar 1. (a) Gambar citra asli, (b) Gambar citra yang dikonversi menjadi citra *gradient magnitude*.

5.3. Dijkstra

Algoritma Dijkstra ditemukan oleh Edsger Dijkstra pada tahun 1956 dan dipublikasikan pada tahun 1959. Menurut Even (2011), algoritma Dijkstra ini adalah algoritma yang mirip dengan algoritma Prim's tetapi digunakan untuk menghitung Shortest Path Tree (SPT). Sebagai algoritma shortest path algoritma ini sering dipakai untuk memecahkan masalah minimalisasi. Gambar 2 di bawah ini merupakan bagan *pseudocode* untuk algoritma Dijkstra:

```
Graf berbobot G = (E, V) // E=himpunan Edge; V=himpunan vertex
s adalah elemen dari V yang merupakan sumber ke semua vertex v elemen V.
set distTo[s] <= 0 // jarak dari vertex s ke s = 0
untuk semua v elemen V – {s} lakukan
      distTo[v] = ∞ // set semua elemen distTo[v] menjadi tak terhingga
      edgeTo[v] = null

S <= {} // S merupakan himpunan vertex yang dilalui.
Q <= V // Q merupakan himpunan semua vertex V
Saat Q ≠ {} lakukan
      vertex x <= pilih elemen Q dengan bobot distTo[] yang paling kecil //Relax
                  // distTo[s] = 0
      masukkan x ke dalam S <= S ∪ { x}
      Untuk semua v Adjacent x lakukan
            Jika distTo[v] > dist[x] + weight(x, v) maka
                  distTo[v] <= dist[x] + weight(x, v)
                  edgeTo[v] <= x
      Q <= Q - x
Kembalikan distTo dan edgeTo
```

Gambar 2. Bagan pesudocode Algoritma Dijkstra

Dalam *pseudocode* tersebut dijelaskan bahwa langkah pertama adalah menyediakan sebuah graf berbo*bot* G dengan E yang merupakan himpunan edge, dan V merupakan himpunan *vertex* . Variabel s merupakan starting point atau sumber ke semua *vertex* v dalam elemen V. Vektor distTo[v] merupakan *vector* yang digunakan untuk menyimpan jarak terpendek menuju *vertex* v. edgeTo[v] adalah edge yang menuju v dengan jarak terpendek.distTo[v] untuk setiap *vertex* v bernilai tak terhingga, supaya dapat dibandingkan dengan jarak yang lebih pendek.

5.4. Acyclic

Tidak jauh berbeda dengan Dijkstra, menurut Even (1979) metode ini mencari shortest path dengan mengunjungi tiap *vertex* x yang ada dan mencari nilai distTo[v] dan edgeTo[v], v merupakan tetangga dari x. Perbedaan algoritma ini dengan Dijkstra yaitu algoritma ini mengunjungi setiap *vertex* dengan secara *topological order*. Artinya sebelum proses pencarian *shortest path*, sistem akan mencari terlebih dahulu *topological order* dari graf tersebut yang kemudian dijadikan acuan dalam mengunjungi tiap *vertex* yang ada. Gambar 3 di bawah ini merupakan bagan *pseudocode* untuk algoritma mendapatkan *topological order*:

```
[X = himpunan vertex V]
   list L = { } //himpunan kosong
   ketika X tidak kosong lakukan:
         cari vertex v yang tidak memiliki edges yang menuju ke v
         hapus v dari X
        tambahkan v ke L
   return L
```

Gambar 3. Bagan *pesudocode* Algoritma *topological order*

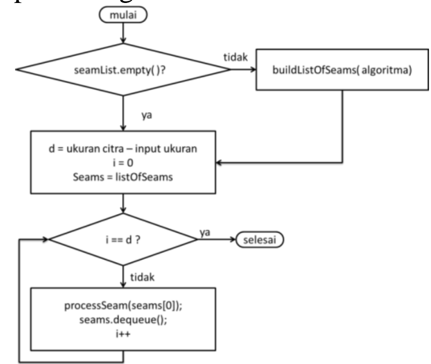
Untuk mendapatkan *topological order* L yang yang perlu dilakukan adalah dengan memencari *vertex* v dalam *vector* X yang tidak memiliki *edge* yang menuju ke dirinya, dan kemudian memasukkan v tersebut ke dalam *topological order* L. L akan digunakan sebagai acuan urutan *vertex* yang akan diproses dalam proses pencarian *shortest path*.

Setelah urutan *topological* L didapatkan, maka lakukan pencarian *shortest path* hanya saja pada Q <= V diganti dengan Q = L. Algoritma ini akan menggunakan *vector* L yang merupakan *topological order* dalam melakukan proses *relax* untuk setiap elemen dalam *vector* tersebut. Kembalian dari proses dalam *pseudocode* tersebut adalah *vector* distTo dan *vector* edgeTo.

5.5. *Precomputed* dan *Recomputed*

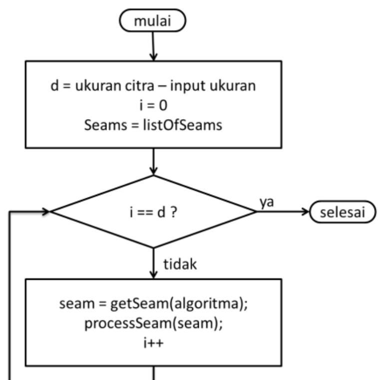
Precomputed seams adalah sebuah metode yang menggunakan sebuah struktur data yang menyimpan *seams* yang ditemukan. Artinya sebelum proses transformasi citra

dilakukan, sistem sudah terlebih dahulu mencari dan menyimpan *seams* yang digunakan sebagai acuan dari proses transformasi citra. Sedangkan *recomputed seam* adalah sebuah metode yang hanya akan mencari seam sesuai dengan kebutuhan proses transformasi. Dalam *recomputed seam*, sistem akan melakukan proses pencarian dan *resize* dalam waktu yang bersamaan. Berikut ini merupakan bagan flowchart untuk metode *precomputed*:



Gambar 4. Flowchart precomputed seams

Proses *precomputed* yang terdapat pada Gambar 4 diawali dengan pengecekan listOfSeams. Jika listOfSeams kosong maka proses akan melakukan pembentukan listOfSeams. Setelah pembentukan selesai, maka akan dilanjutkan dengan pemrosesan listOfSeams yang sudah dibangun terhadap citra. Variabel d merupakan jumlah perubahan ukuran citra. Berikut ini merupakan *flowchart* untuk algoritma *recomputed*:



Gambar 5. Flowchart proses *recomputed seam*

Untuk proses *recomputed* yang tertuang pada Gambar 5, sistem akan melakukan perulangan sebanyak d kali. Perulangan tersebut digunakan untuk mencari *seam* berdasarkan algoritma yang dipakai dan kemudian memproses *seam* tersebut terhadap citra.

6. Hasil Penelitian

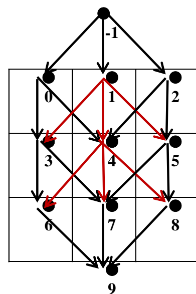
6.1. Perbandingan Kompleksitas Algoritma Dijkstra dan Acyclic

Untuk membandingkan efisiensi algoritma digunakan perbandingan kompleksitas algoritma tersebut yang didasarkan pada Gacs dan Lovasz (1999). Gambar 6 merupakan potongan *code* algoritma Dijkstra yang diimplementasikan di dalam sistem:

```
946 function dijkstra(G, weight, width, mode){
947     Graph.distTo = new Array(weight.length);
948     Graph.edgeTo = new Array(weight.length);
949     Graph.G = G;
950     Graph.w = width;
951     Graph.weight = weight;
952     Graph.minDistTo = Number.MAX_VALUE;
953     Graph.minSeamIndex = 0;
954     setInf(Graph.distTo);
955     setMinOne(Graph.edgeTo);
956     if(mode == 0){
957         for(var s = 0; s < width; s++){
958             Graph.distTo[s] = Graph.weight[s];
959         }
960     }else{
961         for(var s = 0; s < weight.length; s+=width){
962             Graph.distTo[s] = Graph.weight[s];
963         }
964     }
965     for(var s = 0; s < G.length; s++){
966         relax(s);
967     }
968 }
```

Gambar 6. Potongan code Dijsktra

Dalam potongan code di atas V adalah G.length pada baris 965 dan W merupakan width dalam parameter fungsi tersebut. Nilai 3 didapatkan dari proses relax() yang dalam konteks ini hanya akan memproses maksimal 3 *vertex* . Dijkstra sendiri memiliki *worst case* $O(|V|^2)$ ketika jumlah *edge* adalah semua elemen lainnya. Namun dalam sistem ini performa algoritma Dijkstra adalah O(3V + W). Berikut ini adalah gambar tiap *vertex* dalam *vector* 3 x 3



Gambar 7. Gambar ilustrasi *vertex* dan *edge* dalam graf *vector* citra

Dari Gambar 7 dapat disimpulkan bahwa maksimal tiap *vertex* memiliki tiga (3) *edge* (garis panah merah). Berarti jika algoritma Dijkstra dijalankan, maka ketika tiap *vertex* i dilalui, algoritma akan memproses maksimal tiga (3) *vertex* yang adjacent dengan *vertex* i. Dengan begitu didapatkan kompleksitas Dijkstra dalam konteks sistem ini adalah O(3V + W). Kondisi terburuk adalah W = V sehingga algoritma ini bisa memiliki kompleksitas O(4V) atau dapat dinotasikan dengan O(V) yang merupakan kompleksitas dengan kecepatan linear.

Dalam implementasi dalam sistem algoritma ini memiliki kompleksitas yang hampir sama dengan algoritma Dijkstra. Berbeda dengan algoritma Dijkstra, algoritma ini memerlukan proses *sort* yang memilki kompleksitas O(V) karena dalam implementasinya dalam menentukan *vertex* yang terlebih dahulu dikerjakan tidak perlu membandingkan, hanya memasukkan index dari *vertex* secara horizontal atau vertical dengan perhitungan modulus dengan acuan W atau *width* dari citra. Berikut ini merupakan potongan code untuk menentukan *topological order*:

```
985   function topologyOrder(G, width, dir){
986       //dir = 0 : vertical
987       //dir = 1 : horizontal
988       alert(dir + "G" + G.length);
989       L = [];
990       switch(dir){
991           case 0:{
992               for(var i = 0; i < G.length; i++){
993                   L.push(i);
994               }
995               break;
996           }
997           case 1:{
998               for(var i = 0; i < width; i++){
999                   for(var j = i; j < G.length; j+=width){
1000                      L.push(j);
1001                  }
1002              }
1003              break;
1004          }
1005      }
1006      return L;
1007  }
```

Gambar 8. Potongan code fungsi topologyOrder

Pada Gambar 8, V dalam fungsi ini adalah G.length (baris 992 dan 999) yang merupakan jumlah elemen dari graf dan juga jumlah elemen *vector* citra. Dalam potongan code tersebut perulangan dilakukan selama V kali, sehingga di dapatkan waktu T(V). Berikut ini adalah potongan code fungsi acyclic untuk mencari *shortest path* dalam graf:

```
1009  function acyclic(G, weight, width, mode){
1010      Graph.distTo = new Array(weight.length);
1011      Graph.edgeTo = new Array(weight.length);
1012      Graph.G = G;
1013      Graph.w = width;
1014      Graph.weight = weight;
1015      Graph.minDistTo = Number.MAX_VALUE;
1016      Graph.minSeamIndex = 0;
1017      setInf(Graph.distTo);
1018      setMinOne(Graph.edgeTo);
1019
1020      var L = topologyOrder(G, width, mode);
1021      alert(G.length + " " +L.length);
1022      //set distTo
1023      for(var i = 0; i < width; i++){
1024          Graph.distTo[L[i]] = Graph.weight[L[i]];
1025      }
1026
1027      for(var i=0; i< L.length; i++){
1028          relax(L[i]);
1029      }
1030
1031  }
1032
```

Gambar 9. Potongan code fungsi acyclic.

Dari potongan code pada Gambar 9, L.length merupakan yang memiliki nilai sama dengan jumlah elemen *vector* citra sehingga didapatkan T(W + 3V) + T(V) atau O(W + 4V) dengan W adalah *width* yang merupakan parameter (baris 1009) dan T(V) adalah pemanggilan fungsi topologyOrder pada baris ke 1020. Dengan kondisi terburuk W = V, maka bisa kompleksitas ini bisa dinotasikan dengan O(5V) atau O(V).

Dalam konteks running time kedua algoritma ini, Dijkstra dan Acyclic memiliki kompleksitas yang sama yaitu O(V). Akan tetapi jika membandingkan running time sebelum notasi O(V) didapatkan maka algoritma Acyclic memiliki running time yang lebih lama dibandingkan Dijkstra yaitu O(5V) dibanding O(4V). Hal ini disebabkan karena dalam prosesnya, algoritma Acyclic harus membentuk *topological order* untuk menentukan urutan *vertex* yang diproses. Artinya algoritma Dijkstra memiliki *running time* yang lebih cepat daripada Acyclic.

Mengenai hasil seam dari algoritma Dijkstra dan Acyclic ditemukan bahwa seam yang didapatkan bisa berbeda, karena urutan *vertex* yang diproses berbeda. Hal ini dimungkinkan terjadi ketika ditemukan *vertex* -*vertex* yang memiliki energi yang sama, sehingga urutan pemrosesan akan mempengaruhi hasilnya juga.

6.2. Perbandingan Kompleksitas *Precomputed* dan *Recomputed*

Metode precomputed hanya dapat berjalan ketika semua *seams* vertical atau horizontal sudah tersimpan di dalam struktur data list of *seams*. *List of seams* berisi seam yang didapatkan dengan proses *shortest path*. Dalam sistem, *list of vertical* seams dinamakan dengan vseams dan hseams untuk horizontal seams. Karena proses membangun *list of seams* juga termasuk dalam proses ini, maka kompleksitas untuk proses membangun seams juga akan dimasukkan ke dalam proses ini. Gambar 10 berikut ini merupakan potongan *code* untuk membangun *list of seam*:

```
461  Pre.buildVSeams = function(pixels, alg){
462      Pre.vseams = [];
463      Pre.hseams = [];
464      changeInnerHtml("act-data", "build vertical seams");
465      var vseamdt = document.getElementById("vseam-data");
466      var tP = pixels;
467      var width = tP.width;
468      for(var i = 0; i < width - 1; i++){
469          //console.log(i);
470          var luminos = Img.getLuminosityVector(tP);
471          var gradMag = Img.getGradientMagnitudeVector(luminos, tP.width);
472          var G = Img.createVerticalGraph(gradMag.length, tP.width);
473          switch(alg){
474              case 0: {
475                  dijkstra(G, gradMag, tP.width, 0);
476                  break;
477              }case 1:{
478                  acyclic(G, gradMag, tP.width, 0);
479                  break;
480              }
481          }
482          Graph.minSeamIndex = findVerticalMinEnergy(Graph.distTo, tP.width);
483          Pre.vseams.push(Pre.getSeam(Graph.minSeamIndex, Graph.edgeTo));
484          tP = proVerticalSeam(Graph.minSeamIndex, tP, 0);
485          vseamdt.innerHTML = i+1;
486      }
487      Pre.updateStatus();
488  };
```

Gambar 10. Potongan code untuk fungsi buildVSeams

V merupakan panjang citra yang berada dalam G.length. W merupakan panjang citra (width) yang didapatkan dari tP.width (baris 467). tP merupakan sebuah object ImageData. Jika V adalah jumlah element dalam *vector* citra dan W adalah panjang citra maka untuk membangun list of seams diperlukan waktu T(W . V) dimana V merupakan kompleksitas dari algoritma *shortest path* yang dipakai. Bisa dikatakan bahwa kompleksitas didapatkan dari pemrosesan algoritma O(V) sebanyak W kali (baris 468). Dengan kondisi terburuk W = V maka kompleksitas dalam proses pembangunan list of seams dibutuhkan waktu $O(V^2)$.

Untuk memproses seams diperlukan input dw yang merupakan jumlah seam yang di akan diproses. Gambar 11, merupakan fungsi pemrosesan seam untuk precomputed:

```
582      if(dW != 0){
583          if(Pre.vseams.length == 0){
584              alert("vseams belum dibangun!");
585              document.getElementById("out-w").value = pixels.width;
586              document.getElementById("out-h").value = pixels.height;
587          }else{
588              for(var i=0; i < dW; i++){
589                  tP = Pre.processSeams(Pre.vseams[0], tP, 0, mode);
590                  Pre.vseams.splice(0, 1);
591                  vseamdt.innerHTML = i+1;
592              }
593          }
594      }
```

Gambar 11. Potongan code fungsi precomputed

.

Jika panjang *seam* adalah W maka pemrosesan seam terhadap image memiliki kompleksitas waktu T(dw·W). Nilai W didapatkan dari potongan code pemrosesan *seam* pada baris 589 (fungsi Pre.processSeams). Gambar 12 merupakan potongan code fungsi Pre.processSeams yang melakukan perulangan selama W kali:

```
367     switch(dir){
368         case 0:{
369             for(var i = 0; i<seams.length; i++){
370                 r = seams[i] * 4;
371                 switch(promode){
372                     case 0:{
373                         t.splice(r, 4);
374                         break;
375                     }
376                     case 1:{
377                         r = seams[i] * 4;
378                         //console.log(r);
379                         t.splice(r, 0, p[r], p[r+1], p[r+2], p[r+3]);
380                         break;
381                     }
382                     case 2:{
383                         r = seams[i] * 4;
384                         t[r] = 255;
385                         t[r+1] = 0;
386                         t[r+2] = 0;
387                         t[r+3] = 255;
388                         break;
389                     }
390                 }
391 
392             }
393             break;
394         }
```

Gambar 12. Potongan code fungsi Pre.processSeams

Nilai W didapatkan dari seams.length yang merupakan panjang dari *seam* sehingga total kompleksitas yang didapatkan adalah $T(WV + dw \cdot W)$. Dengan kondisi terburuk yaitu $W = V$ dan $dw = W = V$ maka kompleksitas Algoritma tersebut bisa menjadi $O(V^2 + V^2)$ menjadi $O(2V^2)$ atau $O(V^2)$. Kondisi tersebut ketika precomputed hanya mengerjakan satu (1) kali eksekusi saja.

Metode *recomputed* pada Gambar 13, tidak membutuhkan struktur data tambahan. Artinya kompleksitas ditentukan oleh dw dan kompleksitas algoritma terhadap citra. Gambar 13 ini merupakan potongan code untuk metode *precomputed*:

```
634     for(var i = 0; i < dW; i++){
635         var luminos = Img.getLuminosityVector(tP);
636         var gradMag = Img.getGradientMagnitudeVector(luminos, tP.width);
637         var G = Img.createVerticalGraph(gradMag.length, tP.width);
638         switch(alg){
639             case 0: {
640                 dijkstra(G, gradMag, tP.width, 0);
641                 break;
642             }case 1:{
643                 acyclic(G, gradMag, tP.width, 0);
644                 break;
645             }
646         }
647         Graph.minSeamIndex = findVerticalMinEnergy(Graph.distTo, tP.width);
648         tP = proVerticalSeam(Graph.minSeamIndex, tP, mode);
649     }
```

Gambar 13. Potongan kode metode *recomputed*

Kompleksitas yang didapatkan untuk metode *recomputed* adalah $T(dw \times V)$ dimana (V) merupakan kompleksitas dari algoritma Dijkstra maupun Acyclic dan dW adalah jumlah perulangan dilakukan sebanyak dW (baris 634). Sehingga didapatkan $T(dw \cdot V)$, untuk kasus terburuk $dw = W = V$ maka algoritma tersebut memiliki kompleksitas $O(V^2)$.

Menurut Sedgewick dan Wayne (2011) untuk mengerjakan sebuah task yang sama metode precomputed pasti akan memiliki waktu yang lebih lama dengan running time $O(2V^2)$ dibandingkan dengan recomputed yang memiliki running time $O(V^2)$. Untuk resize berikutnya, metode ini hanya membutuhkan waktu $O(V)$ saja, sehingga metode ini tepat untuk proses melakukan resize berulang ulang. Sedangkan untuk recomputed tetap membutuhkan waktu $O(V^2)$.

Tabel 1 berikut ini merupakan hasil dari pemrosesan citra sampel dalam ukuran 240×160 px dengan perubahan sebanyak 20 *pixel* sampai dengan 120×120 px:

Tabel 1.
Tabel hasil pemrosesan *seam carving*

No	Citra	Data
0		Tidak ada
1		Action: Seam Carving Algorithm: Acyclic Method: Recomputed VSeam(s) processed: 20 HSeam(s) processed: 0 Execution Time: 1.262 sec
2		Action: Seam Carving Algorithm: Acyclic Method: Recomputed VSeam(s) processed: 20 HSeam(s) processed: 20 Execution Time: 4.003 sec
3		Action: Seam Carving Algorithm: Acyclic Method: Recomputed VSeam(s) processed: 20 HSeam(s) processed: 0 Execution Time: 1.844 sec
4		Action: Seam Carving Algorithm: Acyclic Method: Recomputed VSeam(s) processed: 20 HSeam(s) processed: 20 Execution Time: 3.233 sec
5		Action: Seam Carving Algorithm: Acyclic Method: Recomputed VSeam(s) processed: 20 HSeam(s) processed: 0 Execution Time: 0.801 sec

Dari Tabel 1 tersebut dapat dilihat bahwa sampel memiliki batas optimal pada perubahan ukuran ke 4, karena pada perubahan ke 5 terlihat bahwa bagian leher jerapah terpotong. Hal ini dapat terjadi melihat bagian yang dilindungi dari citra ini adalah bagian badan jerapah yang cenderung memiliki energi *gradient magnitude* yang besar. Bagian leher dan kepala dari jerapah dapat dilindungi dengan cara memberikan energi yang lebih besar pada daerah tersebut. Dari hasil tersebut juga dapat dilihat bahwa ukuran badan jerapah tidak mengalami perubahan, perubahan yang banyak terjadi adalah mengecilnya *background*. Dengan demikian proses *seam carving* ini berhasil melakukan transformasi ukuran citra secara *content aware* dengan 1 objek didalamnya.

7. Kesimpulan

Berdasarkan hasil penelitian di atas, algoritma Dijkstra memiliki kompleksitas yang lebih kecil yaitu O(4V) dibandingkan Acyclic yang memiliki kompleksitas waktu O(5V). Artinya algoritma Dijkstra memiliki performa yang lebih cepat dibandingkan dengan Acyclic meskipun pada dasarnya notasi Big Oh kedua algoritma ini sama yaitu O(V).

Efisiensi metode *precomputed* dam *recomputed* tergantung pada konteks pemakaiannya, meskipun sama-sama mempunyai kompleksitas $O(V^2)$. Jika proses mempertimbangkan proses awal atau hanya dilakukan sekali transformasi saja, metode recomputed lebih efisien, sedangkan jika proses tidak mempertimbangkkan proses awal dan akan dipakai secara kontinyu, maka metode precomputed lebih efisien.

8. Saran

Dalam penelitian ini, penulis menemukan beberapa kendala dalam pembuatan sistem yang dapat menjadi saran untuk penelitian selanjutnya. Berikut ini merupakan poin-poin yang menjadi kendala dalam penelitian:

1) Ketika melakukan pemrosesan citra berukuran besar, jika proses memakan waktu yang terlalu lama browser akan memunculkan dialog unresponsive script. Selain itu, pada waktu pemrosesan citra, proses tersebut tidak dapat di interupsi, sehingga tidak memungkinkan menjalankan proses yang asynchronous seperti menampilkan loading bar atau persentase pemrosesan citra, agar lama selesai proses dapat di estimasikan oleh pengguna. Solusi untuk masalah ini adalah dengan menggunakan multithreading dengan menggunakan ajax atau worker.js.
2) Untuk melindungi bagian citra yang memiliki energi lebih rendah, dapat ditambahkan metode lain untuk hasil citra yang sesuai keinginan pengguna. Metode seperti face detecting dapat digunakan untuk kasus melindungi wajah pada citra foto manusia. Energy coloring secara interaktif juga dapat digunakan untuk menentukan objek yang akan dilindungi atau dihilangkan.
3) Agar performa lebih cepat dapat dicari algoritma *shortest path* dengan metode lain atau dengan model pemrograman dynamic programming.

Daftar Pustaka

Avidan, S., & Shamir, A. (2007). Seam Carving for Content-Aware Image Resizing.

Danziger, P. (n.d.). *Big O Notation.* Retrieved January 19, 2015, from Department of Computer Science - Ryerson University: http://www.scs.ryerson.ca/~mth110/Handouts/PD/bigO.pdf

David, J. (2005). *Image Gradients* . Retrieved 4 1, 2014, from http://www.cs.umd.edu/: http://www.cs.umd.edu/~djacobs/CMSC426/ImageGradients.pdf

Even, S. (1979). *Graph Algorithms.* Rockville, Maryland: Computer Science.

Gacs, P., & Lovasz, L. (1999). *Complexity of Algorithm Lecture Notes.*

Low, A. (1991). *Introductory Computer Vision and Image Processing.* Singapore: McGraw-Hill Book Company (UK) Limited.

MIT. (n.d.). *http://web.mit.edu/16.070/www/lecture/big_o.pdf.* Retrieved January 19, 2015, from MIT - Massachusetts Institute of Technology: http://web.mit.edu/16.070/www/lecture/big_o.pdf

Sarkar, A., Nataraj, L., & Manjunath, B. S. (2009). Detection of Seam Carving and Localization of Seam. *Proceedings of the 11th ACM workshop on Multimedia and Security*, 107-116.

Sedgewick, R., & Wayne, K. (2011). *Algorithms.* Massachusetts: Pearson Education, Inc.

Teuber, J. (1993). *Digital Image Processing.* Hempstead: Prentice Hall International (UK) Ltd.

Thilagam, K., & Karthikeyan, S. (2012). Optimized Image Resizing using Piecewise Seam Carving. *International Journal of Computer Applications*, 24-30.