Optimized Block Based Disparity Estimation in Stereo Systems Using a Maximum-Flow Approach

IMIR THOMO, SOTIRIS MALASIOTIS, MICHAEL G. STRINTZIS *

Department of Electrical and Computer Engineering,
University of Thessaloniki,
Thessaloniki 540 06, GREECE
Tel: +30-31-996359 Fax: +30-31-996398
alex,malasiot@panorama.ee.auth.gr, strintzi@eng.auth.gr

Abstract. A novel disparity estimation method is presented that increases the robustness of the estimator by replacing the classical approach of dynamic programming with finding the maximum flow in a graph. Once solved, the minimum cut associated to the maximum flow yields a disparity surface for the whole image at once. The results show improved depth estimations as well as better handling of depth discontinuities. Although the running time for solving the maximum flow problem is higher than dynamic programming, experiments have shown that the special topology of the graph, the position of the source and sink and the capacity structure of the edges tend to make the problem easier to solve. However, the main drawback of this approach is the large amount of memory resources required by the classical implementations of the maximum flow algorithm, which make impossible in practice to apply this approach even for the small stereo images with small disparity resolution. Taking advantage of the special topology of the graph and the position of the source and sink we propose an efficient data structure for drastically reducing the amount of memory resources used.

Keywords. disparity estimation, maximum flow, lift-to-front algorithm

1 Introduction

The determination of homologous image points between the left and right image views, is a crucially important step in stereoscopic image analysis. Perspective effects, occlusions, photometric variation and inherent ambiguities make disparity estimation a difficult task. The dynamic programming approach casts the problem of matching a pair of stereo images as an optimization problem, which is solved by making the following assumptions [1, 2, 3]. The first assumption is the well known epipolar constraint. According to this constraint the depth related displacements in stereo pairs always occur along the epipolar lines. This constraint reduces the stereo correspondence problem to one dimension. The second assumption made is the ordering constraint. According to this, a set of points on an epipolar line of the left image and their corresponding points in the right image appear in the same order. These constraints along with other details of the camera geometry regarding stereo imagery are fully covered in the particularly readble textbook of [4]. However the reduction to one dimensional space is an over-simplification of the problem and is made primarily for reducing significantly the computational complexity. The solutions obtained on consecutive epipolar lines can vary significantly and create artifacts across epipolar lines, especially affecting object boundaries that are perpendicular to the epipolar lines.

There have been several earlier approaches to smooth the artifacts created at vertical edges relating the solution of consecutive epipolar lines matched with dynamic programming [1, 2]. These approaches in practice do not perform well in every case and in general are not very efficient and not optimal.

In this paper we propose a new approach based on the method of [5]. In this method the traditional ordering constraint is replaced by the more general local coherence constraint and the stereo correspondence problem is cast as a maximum flow problem in a graph. The minimum cut associated to the maximum flow can be interpreted as a disparity surface for the whole image. A robust optimization cost function for the the stereo matching problem is introduced improving in this way the disparity estimation accuracy in [5] by at least 100%.

However, the main contribution of our approch is the adaptation of the well known "Lift-to-Front" method of [6] for solving the maximum flow problem in a random graph to the specific case of the disparity 3D graph. This method belongs to the "Preflow-Push" methods which are the fastest up to date methods for solving the maximum

*This work was supported by the EU projects ACTS 092 PANORAMA (Package for New Autostereoscopic Multiview Systems and Applications).

flow problem in a graph with significant improvement over the running time of the Ford-Fulkerson method. The main drawback of the "Lift-to-Front" method is the large amount of memory resources required by the classical implementations, which make impossible in practice to apply this approach even for the small stereo images with small disparity resolution. Taking advantage of the special topology of the graph and the position of the source and sink we use an efficient data structure reducing drastically the amount of memory resources used.

This paper is organized as follows. Section 2 describes the geometry of a general stereo scene and its associated epipolar constraint. In Section 3 a robust optimization cost function is defined. In Section 4, details of casting the stereo matching problem as a maximum flow problem are presented. Also the possibilities of graph degeneration and the method used to eliminate them are discussed there. The class of the fastest to date preflow-push algorithms and a variant of them, the lift-to-front algorithm are described in Section 5. The analysis of the lift-to-front algorithm for the specific 3D input graphs is done in Section 6. In this section it is shown that using an efficient data structure, appropriate for the specific topology of the graph a significant memory reduction can be achieved. Experiments and results are presented and discussed in Section 7.

2 Epipolar Geometry and Matching

As can be seen from Figure 1, given m_1 in the image plane I_1, all possible physical points M that may have produced m_1 are on the infinite half line (m_1, C_1). As a direct consequence, all possible matches m_2 of m_1 in the plane I_2 are located on the projection, through the second imaging system, of this infinite half line.

In the traditional approach to stereo matching, a single epipolar line in the first image is matched with its corresponding epipolar line in the second image. The established matching between two lines is a path in the grid of all possible matches (Figure 2). The matching grid between the epipolar line in the first image and the epipolar line in the second can be transformed into the equivalent formulation on the right where only the line in the first image appears directly. In this case each potential match has the form (m, d), where m is the position along the epipolar line and d is its associated disparity. In Figure 3 all minimum cost paths defining the matching of epipolar lines are now assembled into a single minimum cost surface. The goal of this construction is to take advantage of one very important property of disparity fields, which is local coherence, by which is meant that disparities tend to be locally very similar, in any and all directions. This property is exploited in dynamic programming based methods along epipolar lines by enforcing the ordering constraint. However, local coherence occurs in all directions. By putting all epipolar lines together and solving globally for a disparity surface, it becomes possible to take full advantage of local coherence and to improve the resulting depth map.

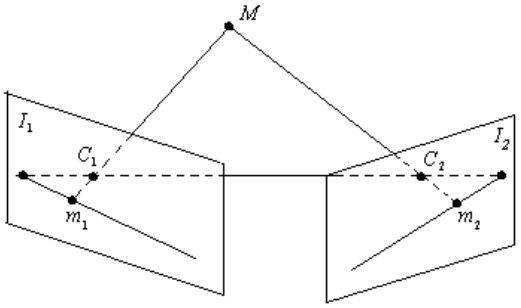


Figure 1: Epipolar geometry

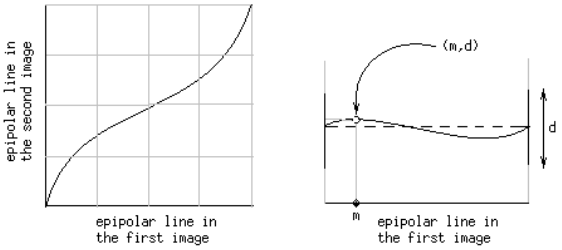


Figure 2: Epipolar matching

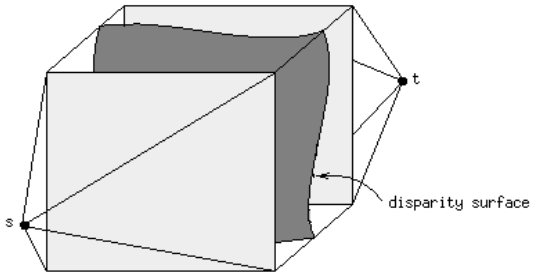


Figure 3: Putting all epipolar lines together and casting the problem of finding the disparity surface in a maximum flow problem in a graph

3 Definition of a Useful Cost Function

In order to perform the actual stereo matching, the definition of a matching cost function is needed. It should be small for a likely match and large for an unlikely one.

By assuming that the surfaces are Lambertian (i.e. their intensity is independent of viewing direction) the

intensity values of the projections of one $3D$ point M on the camera planes should be identical and thus these projection are a valid match. Therefore, we can define the matching cost as the variance of the pixel intensities as

$$cost(m,d) = \tfrac{1}{2}((I_1(m) - \bar{I}(m,d))^2 + (I_2(m+d) - \bar{I}(m,d))^2) \quad (1)$$

where $I_k(m)$ is the luminace value of pixel m in the image plane I_k, $k = 1, 2$ and $\bar{I}(m,d)$ is the mean of the luminance values of pixel m in image plane I_1 and pixel $m+d$ in image plane I_2. However, as can be seen also from the experimental results, the disparity estimation based on this cost function is not stable and produces many outlier disparity values. A natural alternative would be to use as cost function the Displace Frame Difference (DFD):

$$DFD(m,d) = \sum_{(x,y)\in W} ||I_1(m_x + x, m_y + y) - I_2(m_x + d_x + x, m_y + d_y + y)|| \quad (2)$$

where W is a square window centered at pixel m.

The value of the cost function is considered reliable only if the pixel m is located over highly textured regions. For the detection of texture we used a variant of the technique proposed in [7] which is based on the observation that the highly textured regions present high local variation of the luminance in all directions, while on edges the variation of the luminance is higher in the edge direction. The cost function is computed for every disparity layer and the corresponding cost image is low pass filtered. The low pass filtering replaces each unreliable value with a weighted mean of neighboring reliable pixel values.

4 Casting the Stereo Matching Function as a Maximum Flow Problem

We solve globally for the disparity surface by adding a source and a sink as in Figure 3, and treat it as flow problem in a graph. The vertex set is defined as

$$V = V^\star \cup \{s, t\} \quad (3)$$

where s is the source, t is the sink and $V^\star$ is the $3D$ mesh:

$$V^\star = \{(x,y,d) : x \in [0 \dots x_{max}], y \in [0 \dots y_{max}], d \in [0 \dots d_{max}]\} \quad (4)$$

where $(x_{max}+1, y_{max}+1)$ is the base image size and the $d_{max}+1$ is the disparity resolution. Internally the mesh is six-connected and the source s connects to the front plane while the back plane is connected to the sink t. We have:

$$E = \begin{cases} (u,v) \in V^\star \times V^\star & ||u - v|| = 1 \\ (s,(x,y,0)) & x \in [0 \dots x_{max}] \\ ((x,y,d_{max}),t) & y \in [0 \dots y_{max}] \end{cases} \quad (5)$$

Being six-connected instead of four connected, each vertex of the new problem is not only connected to its neighbors along the epipolar line, but also across adjacent epipolar lines. We can compute the maximum flow between the source and the sink. The set of edges that are saturated by the maximum flow represent the minimum cut of the graph. This cut separates the source and the sink and effectively represents the disparity surface. The edge capacities are defined in a straightforward way. The matching cost is used directly as a capacity. Since a likely match has a low matching cost, the corresponding edge capacity will be low and that edge is likely to be saturated by the maximum flow. Conversely, a high matching cost yields a high capacity edge, which is unlikely to be saturated. Since a vertex in the graph corresponds to a potential match, we can derive its matching cost by the above cost equation. The capacity of an edge is derived from the matching cost of the two vertices that it links:

$$c(u,v) = \frac{cost(m_u, d_{m_u}) + cost(m_v, d_{m_v})}{2} \quad (6)$$

where the $m_{\{u,v\}}$ are the corresponding reference image points and the $d_{\{m_u,m_v\}}$ are their hypothetical disparities. We can write the above equation in the following equivalent form:

$$c(u,v) = \frac{cost(u) + cost(v)}{2} \quad (7)$$

where $cost(u)$ is used for simplicity instead of $cost(m_u, d_{m_u})$ and $cost(v)$ instead of $cost(m_v, d_{m_v})$ since u and v are matches and defined by their associated point and disparity.

Sometimes there can be edges with zero capacity along which no flow passes. These cases can frequently occur in the homogeneous regions or in the ideal case of a perfect match. As no flow pass through these edges, they do not belong to the set of saturated edges and so they are not included in the disparity surface. This problem can be solved by a well-known method used in linear programming which avoids the graph degeneration. Namely instead of zero capacities ε edge capacities are used where ε is a small number near zero but not zero. In this way the undesired possibility of non passing flow through the "very good" matching edges is eliminated. These edges now, are likely to be saturated by the flow and the resulting disparity surface is more reliable.

The cut obtained by solving the maximum flow problem is the optimal way to separate the source and the sink for the particular cost function. As it is known, any cut must break each path connecting the source with the sink. From the graph structure for any (x,y), there is a path $s \rightsquigarrow t$ of the form:

$$s \to (x,y,0) \to (x,y,1) \to \dots \to (x,y,d_{dmax}) \to t \quad (8)$$

Since the capacities of the edges that connect the source and sink with other vertices are infinity, it follows that the cut will break this path in at least one edge $(x, y, d) - (x, y, d + 1)$ where $d \in [0, d_{max} - 1]$. According to the above, a disparity map can be constructed from the minimum cut as follow. For each point (x, y), the disparity is the largest d such that the edge $(x, y, d) - (x, y, d + 1)$ belongs to the minimum cut.

5 Preflow-push Algorithms

In this section, we present the "preflow-push" approach to computing maximum flows. The fastest maximum-flow algorithms to date are preflow-push algorithms. This section describes a refinement of the Goldberg's "generic" maximum-flow algorithm, the "Lift to Front" approach which has a running time of $O(V^3)$. The intuition behind the preflow-push methods is best understood in terms of fluid flows. We consider a flow network $G = (V, E)$ to be a system of interconnected pipes of given capacities. Vertices which are pipe junctions, have two properties. First to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrary large reservoir that can accumulate fluid. Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses.

Vertex heights determine how flow is pushed: we only push flow downhill, that is from a higher vertex to a lower vertex. There may be positive net flow from a lower vertex to a higher vertex, but operations that push flow always push it downhill. The height of the source is fixed at $|V|$, and the height of the sink is at 0. All other vertex heights start at 0 and increase with time. The algorithm first sends as much flow as possible downhill from the source toward the sink. The amount it sends is exactly enough to fill each outgoing pipe from the source to the capacity. When flow first enters an intermediate vertex, it collects in the vertex's reservoir. From there, it is eventually pushed downhill.

It may eventually happen that the only pipes that leave a vertex u and are not already saturated with flow connect to vertices that are on the same level as u are uphill from u. In this case, to rid an overflowing vertex u of its excess flow, we must increase its heights. This operation is called "lifting" vertex u. Its height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. After a vertex is lifted, therefore, there is at least one outgoing pipe through which more flow can be pushed.

A capacity edge usually is considered equivalent with two arcs inverse to each other with the same capacity equal to the edge capacity. We note $f(u, v)$ for the current net flow over the arc (u, v), $h[u]$ for the height of vertex u, and $e[u]$ for the excess accumulated at the reservoir of vertex u. Also we define the residual capacity of an arc (u, v) to be the difference $c_f = c(u, v) - f(u, v)$. The flow satisfies the following well known properties:
Capacity constraint: For all $u, v \in V$, we require

$$f(u, v) \leq c(u, v) \tag{9}$$

Skew symmetry: For all $u, v \in V$, we require

$$f(v, u) = -f(u, v) \tag{10}$$

Flow conservation: For all $u \in V - s, t$, we require

$$\sum_{v \in V} f(u, v) = 0 \tag{11}$$

5.1 The basic operations

From the preceding discussion, we see that there are two basic operations performed by a preflow- push algorithm: pushing flow excess from a vertex to one of its neighbors and lifting a vertex. The basic operation $PUSH(u, v)$ can be applied if u is an overflowing vertex, $c_f(u, v) > 0$, and $h(u) = h(v) + 1$.

```
PUSH(u,v)
temp = min(e[u],cf(u,v))
f(u,v) = f(u,v)+temp
f(v,u) = -f(u,v)
e[u] = e[u] - temp
e[v] = e[v] + temp
```

The basic operation $LIFT(u)$ applies if u is overflowing and if and if $c_f(u, v) > 0$ implies $h[u] \leq h[v]$ for all vertices v. In other words, we can lift an overflowing vertex u if for every vertex v for which there is residual capacity from u to v, flow cannot be pushed from u to v because v is not downhill from u.

```
LIFT(u)
h[u] = 1+min{h[v] : cf(u,v)>0}
```

The generic preflow-push algorithm uses the following subroutine to create an initial preflow in the flow network.

```
INITIALIZE-PREFLOW(G,s)
for each vertex u in V[G]
   do h[u] = 0
      e[u] = 0
for each edge (u,v) in E[G]
   do f(u,v) = 0
      f(v,u) = 0
h[s] = |V[G]|
for each vertex u adjacent to s
   do f(u,s) = c(s,u)
      f(s,u) = -c(s,u)
      e[u] = c(s,u)
```

After initializing the flow, the generic algorithm repeatedly applies, in any order, the basic operations wherever they are applicable. It is shown that the generic algorithm runs in $O(V^2E)$ time.

5.2 The lift-to-front algorithm

The generic preflow-push method allows us to apply the basic operations in any order at all. By choosing the order carefully and managing the network data structure efficiently, we can solve the maximum-flow problem faster than $O(V^2E)$ bound. The lift-to-front algorithm maintains a list of the vertices in the network. Beginning at the front, the algorithm scans the list, repeatedly selecting an overflowing vertex u and then "discharging" it, that is, performing push and lift operations until u no longer has a positive excess. Whenever a vertex is lifted, it is moved to the front of the list and the algorithm begins its scan anew. It is shown [6] that the lift-to-front algorithm has running time $O(V^3)$, which is asymptotically at least as good as $O(V^2E)$.

Edges, in classical implementations, are organized into "neighbor lists." Given a flow network $G = (V, E)$, the neighbor list $N[u]$ for a vertex $u \in V$ is a singly linked list of the neighbors of u in G. The first vertex in the $N[u]$ is pointed to by $head[N[u]]$. The vertex following v in a neighbor list is pointed to by $next - neighbor[v]$; this pointer is NIL if v is the last vertex in the neighbor list. The lift to front algorithm cycles through each neighbor list in an arbitrary order that is fixed throughout the execution of the algorithm. For each vertex u, the field $current[u]$ points to the vertex currently under consideration in $N[u]$. Initially, $current[u]$ is set to $head[N[u]]$. An overflowing vertex u is discharged by pushing all of its excess flow through edges (arcs) with non-zero residual capacity to neighbors vertices, lifting u as necessary. The pseudocode is as follows.

```
DISCHARGE(u)
while e[u] > 0
  do v = current[u]
     if v = NIL
        LIFT(u)
        current[u] = head[N[u]]
     else
        if cf(u,v)>0 and h[u]=h[v]+1
          PUSH(u)
        else
          current[u] = next-neighbor[v]
```

As we said above, in the lift-to-front algorithm, we maintain a linked list L consisting of all vertices in $V - s, t$. The pseudocode for the lift-to-front algorithm assumes that $next[u]$ points to the vertex that follows u in list L and that, as usual, $next[u] = NIL$ if u is the last vertex in the list.

```
LIFT-TO-FRONT(G,s,t)

INITIALIZE-PREFLOW(G,s)
L = V[G]-{s,t}
for each vertex u in V[G]-{s,t}
   do current = head[N[u]]
u=head[L]
while u <> NIL
   do old-height = h[u]
      DISCHARGE(u)
      if h[u] > old-height
         move u to the front of L
      u = next[u]
```

It should be note that, if u was moved to the front of the list, the vertex used in the next iteration is the one following u in its new position in the list. It is shown that the lift-to-front algorithm terminates and its running time is $O(V^3)$.

6 On Analyzing the Algorithm and the Specific Topology of the Graph

Analyzing an algorithm has come to mean estimating the resources that the algorithm requires. Usually, resources such as computational time and memory are of primary concern. The dynamic programming approach on separate epipolar lines requires a total running time which might seem much better than the maximum-flow algorithm. However, the computational time required for solving the maximum-flow problem depends heavily in the input graph. After experiments with various stereo pairs we observed that the topology of the graph, the position of the source and sink, and the structure of edge capacities all tend to make the problem easier to solve, making the average running time much better than the worst case analysis. In practice, the average running time of the fully optimized lift-to-front algorithm compares favorably with the dynamic programming approach. The typical running time for 256×256 images is anywhere between 0.5 to 5 minutes, on an Indigo 2 SGI workstation, depending on the depth resolution used.

But, as can be easily observed the main drawback of the classical implementation of the lift-to-front algorithm is the large amount of memory that it requires. Let us estimate how large this amount of memory is. As were shown above, edges in the lift-to-front algorithm are organized into "neighbor lists." The best solution for an arbitrary graph would be a singly linked list of the neighbors for each vertex. Because in our graph the vertices are six-connected, their "neighbors lists" will have six elements. In these elements we have to store information about the endpoint (an edge typically is considered equivalent with two arcs inverse to each other, so can be used the term endpoint), capacity and residual capacity of the corresponding edge along with the pointer to the next element of the list. However, we must note that in order to fully optimize the lift-to-front algorithm regarding the

	Bytes
Endpoint	4
Capacity + res. capacity	2
Pointer to inverse arc	4
Pointer for linking the list	4
Total	14

Table 1: Bytes required for neighbor list element.

running time, the line

```
f(v,u) = -f(u,v)
```

in the $PUSH$ routine must be executed in $O(1)$ time. For this reason, usually in the classical implementations a pointer to the inverse arc is stored in each element of the "neighbor lists."

In this way, 14 bytes are needed per list element or $6 * 14 = 84$ bytes in total per vertex, according to Table 1.

Regarding the source and sink their "neighbor lists" have $x_{max} * y_{max}$ elements, that is $x_{max} * y_{max} * 14$ bytes are needed.

We must note that an additional amount of eight bytes for each vertex for storing the excess flow and the height value and eight bytes for maintaining the vertex list in the LIFT-TO-FRONT routine is required.

So in this representation, which is the best for arbitrary graphs, the lift-to-front algorithm needs $84 + 16 = 100$ bytes for each vertex excluding the source and sink and $2 * 100 * x_{max} * y_{max}$ bytes for the source and sink. Totally, the algorithm requires an amount of

$$100 * x_{max} * y_{max} * d_{max} + 2 * 100 * x_{max} * y_{max}$$

bytes of memory resources.

This amount of memory is in many cases very large prohibiting us to run the program for large images with large disparity resolution. For example in in a stereo pair 256×256 with disparity resolution 30 about $210Mb$ memory is needed. However, using the special structure of the graph we can drastically reduce the memory resources needed by the lift-to-front algorithm. Firstly, we show how the second term in the above sum can be eliminated. Eventually, all the flow that can possibly get through to the sink has arrived there. No more can arrive, because the pipes obey the capacity constraints. The algorithm then sends the excess collected in the reservoirs of overflowing vertices back to the source by continuing to lift vertices to the above the fixed height $|V|$ of the source. Instead of performing a push operation from the front vertices back to the source we can simply cancel the excess of the front vertices. In this way there is no need to store any information regarding the source, its "neighbor list" etc. Regarding the sink, recall that the capacities of edges that link the end vertices to the sink are infinity. In this way if eventually there is excess to the reservoirs of the end vertices it all can be pushed to the sink. Similarly, we do not perform in these cases a push operation but simply cancel the excess flow eliminating in this way any information regarding the sink. Eliminating the source and sink we now have a 3D graph with six connected vertices only. By taking advantage of this fact we can use an array of

$$x_{max} * y_{max} * d_{max} \times 6$$

dimensions where each row corresponds to a vertex and each element in the row corresponds to an edge. The elements of the array have a size of two bytes only, just for storing the capacity and residual capacity of the corresponding edge. The key property of the array is the order in which the information for each vertex is placed in it. Namely the u^{th} row of the array corresponds to the (i, j, d) vertex by the following function:

```
INDEX(u, i, j, d)
temp = u div dmax
d = u mod dmax
j = temp div xmax
i = temp mod xmax
```

Of course this function is one-to-one and its inverse is:

```
INVERSE-INDEX(i, j, d)
return j*nx*dmax + i*dmax + d
```

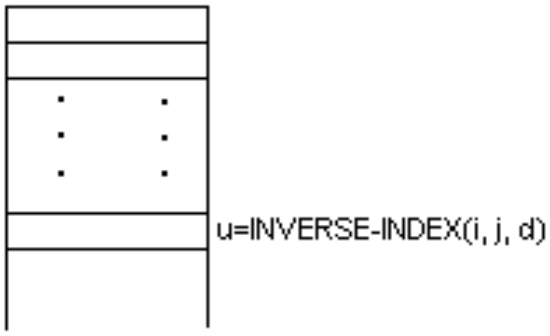


Figure 4: Array of hexades

We also have to define a specific order inside the hexades of each row in the array. Namely we define the order of the six links of a vertex to be as in Figure 5.

In this way, as we will show in the following, there is no need to store the endpoint fields, pointers to the inverse arc, or pointers for linking the "neighbor list".

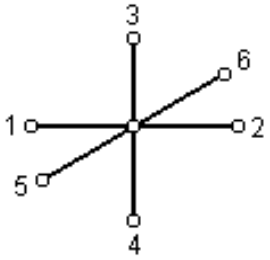


Figure 5: Order inside the hexades

In fact, by taking advantage of the specific order of the array, the endpoint of an arc whose capacity and residual capacity are stored in the array in the (u, k) place and the pointer to its inverse arc can be found in $O(1)$ time by the following routines:

```
END-POINT(u, k)
INDEX(u,i,j,d)
switch(k)
   case 1:
      return INVERSE-INDEX(i-1,j,d)
   case 2:
      return INVERSE-INDEX(i+1,j,d)
   case 3:
      return INVERSE-INDEX(i,j-1,d)
   case 4:
      return INVERSE-INDEX(i,j+1,d)
   case 5:
      return INVERSE-INDEX(i,j,d-1)
   case 6:
      return INVERSE-INDEX(i,j,d+1)

INVERSE(u, k)
v = END-POINT(u,k);
switch(k){
   case 1:
      return adress(graph[v,2])
   case 2:
      return adress(graph[v,1])
   case 3:
      return adress(graph[v,4])
   case 4:
      return adress(graph[v,5])
   case 5:
      return adress(graph[v,6])
   case 6:
      return adress(graph[v,5])
```

Therefore $6 * (4 + 4 + 4) = 72$ bytes less per vertex are used than in the classical implementations of lift-to-front algorithm. In other words for each vertex there are needed twenty eight bytes, that is the algorithm will totally use:

Method	**Correct** (%)
Roy S. and Cox J. I. [5]	22.26 %
Cox J. I. [2]	36.19 %
Tzovaras D. [1]	44.21 %
Intille S. S. and Bobick F. A. [3]	45.12 %
Proposed method	46.74 %

Table 2: Percentages of the well estimated disparity values for each method.

$$28 * x_{max} * y_{max} * d_{max}$$

bytes only. This is a drastic reduction of memory resources compared with the classical implementations. For the previous example of the 256×256 images with disparity resolution 30 the memory used is 55Mb.

7 Experiments and Results

A numerical comparison of the efficiency of various disparity methods is made. For this the synthetic "Corridor" stereo pair (courtesy of Thorsten Froehlinghaus, University of Bonn) with its ground truth disparity map was selected. After estimating the disparity with the estimators proposed in [1, 2, 3, 5] the percentages of the correctly estimated values for each method were compared (Table 2). The increase in the percentage of the correct matches is due to the cost function used, if we compare with [2]. Regarding the other methods, the increase in the percentage of the correct matches is due to the new method of casting the stereo matching problem to the maximum flow problem.

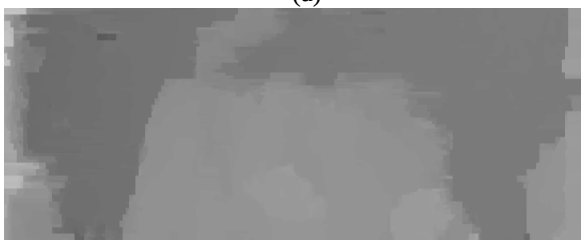
Results obtained in real stereo image pair "Aqua" are shown in Figure 6. If we compare visually the results obtained with the dynamic programming method as for example in the case of [1] with the results obtained with the proposed method we can see that the artifacts at the vertical edges are reduced significantly using the present method. As can be seen, the disparity edges in the right side of the rock and at the top of image are sharper and more accurate.

It should be noted that the disparity map images are equalized in order to improve their contrast.

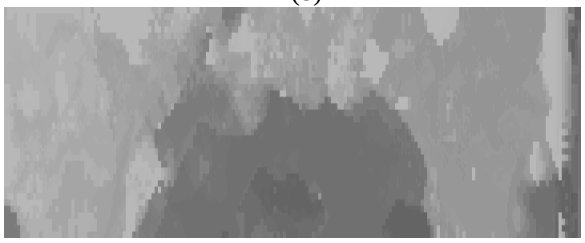
8 Conclusions

A novel method for disparity estimation in stereo images was presented. The method is based on concepts similar to those in [5]. However it performs considerably better than [5], improving the disparity estimation accuracy in [5] by at least 100%. Combining the method of maximum flow in a graph for solving the optimization prob-

(a)



(b)



(c)

Figure 6: “Aqua”, disparity estimation (a) original image (b) method in [1], (c) proposed method,

lem of disparity estimation with the power of a sound optimization cost function we achieve better results than the well known methods in the literature. Further we reduce the prohibitive memory cost of the classical implementation of the maximum flow method using an efficient data structure that takes advantage of the special topology of the graph reducing the memory resources required about four times.

References

[1] D. Tzovaras, N. Grammalidis, and M. G. Strintzis, “Object-Based Coding of Stereo Image Sequences Using Joint 3D mo tion/disparity Compensation,” *IEEE Trans. on Ciscuits and Systems for Video Technology*, vol. 7, pp. 312–328, April 1997.

[2] I. Cox, “A Maximum Likehood N-camera Stereo Algorithm,” in *Proceedings, IEEE Conference on Computer Vision and Pattern Recognition*, (Seattle, WA), pp. 733–739, 1994.

[3] S. S. Intille and A. F. Bobick, “Disparity-Space Images and Large Occlusion Stereo,” tech. rep., M.I.T. Media Lab Perceptual Computing Group, No. 220, 1994.

[4] O. Faugeras, *Three Dimensional Computer Vision*. Cambridge, MA: MIT Press, 1993.

[5] S. Roy and I. J. Cox, “A maximum flow formulation of the n-camera stereo correspondence problem,” in *Proc. IEEE Int. Conf. Computer Vision, ICCV’ 98*, (Bombay, India), 1998.

[6] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[7] O. Egger, W. Li, and M. Kunt, “High compression image coding using an adaptive morphological subband decomposition,” *Proc. IEEE*, vol. 83, pp. 272–287, February 1995.