Flows

Michel Koskas, Cecile Murat

Flows !

Michel Koskas*, Cécile Murat*

Abstract

Some tools used in Combinatorics of Words allow the profiling of "divide and conquer" algorithms in a number of Operational Research fields, like database management, automatic translation, image pattern recognition, flowing or shortest path problems.... This paper details one of them, the maximization of a flow over a network.

1 Introduction

Some problems make easy a "divide and conquer" approach, which usually leads to efficient algorithms (even often optimal). For instance for sorting an array, we may split it in two parts, sort the left part, then the right part and merge the two sorted parts. Some other problems do not seem to naturally allow such an approach: for instance when searching the shortest path between two vertices of a graph, it is not clear at all that it may be obtained in such a way. Neither the image pattern recognition nor problems like automatic translation, database management or flow problems seem to allow a natural "divide and conquer" approach.

Actually these problems can be solved thanks to a "divide and conquer" approach. The goal of this paper is to fully detail one of them, namely the flowing problem.

For solving a problem thanks to a "divide and conquer" algorithm, the data is usually split in smaller parts (usually halved) whose solutions are merged together.

Let us take back the example of sorting an array. It is possible to use a quicksort, a mergesort, a heapsort ... The heapsort is not a "divide and conquer" algorithm, is optimal in average and worst case but is averagely less efficient than the quicksort. The quicksort

*Work supported by Lamsade, Université Paris IX Dauphine
{koskas,murat}@lamsade.dauphine.fr

proceeds by choosing an element of the array (say the middle one called "pivot"), puts at its right elements that are greater than or equal to the pivot, at its left elements that are lower than or equal to the pivot, and sorts the left and right parts. The step of dealing with the two parts is performed at first and then the independent computations are performed. The mergesort consists in sorting the left and right halves of he array and then deals with the two parts by merging two sorted sub arrays. A dichotomic search has at each step only to deal with one half of the array and therefore does not necessitates a "deal with two parts" step. The algorithm presented in this paper uses an algorithm to find the shortest paths in an unweighed graph that performs several times the "deal with two parts" step and the "merge" (or "check solutions") part.

We use an extra data structure where the method applies. This means that the first step of these algorithms consists in building the extra data structure and, fortunately, this step is performed only once and may be considered as part of the data representation. For instance when dealing with graphs for, say, the shortest path problem, the extra data structure is computed only once and is used for any couple of vertices.

All these algorithms lie on the use of radix trees, an efficient structure for storing data and for performing hierarchical computations on it.

The paper is organized as follows. After giving the basic definitions and notation, we present the common knowledge about radix trees followed by some applications in various domains. Section 4 is devoted to the study of the shortest path problem and its refinements, which in turn leads to an efficient improvement of the well known Ford-Fulkerson Algorithm for flows.

2 Definitions and Notation

A graph G is a triple (V, A, C) where V is a finite set of *vertices*, A is a subset of $V \times V$ (the *arcs* of the graph) and C a function from A to $\mathbb{R}^+$ (the *capacities*).

The sets A and C may be considered as an adjacency matrix $M = (m_{i,j})_{1 \leq i,j \leq n}$ where $v = \#(V)$ and

$$\forall i, j,\ m_{i,j} = \left\{ \begin{array}{ll} 0 & \text{if } (i,j) \notin A \\ C((i,j)) & \text{else.} \end{array} \right\}.$$

For any vertices i and j, the arc (i, j) is an outgoing arc of i and an incoming arc of j. For any vertex i, we denote $\mathrm{Out}(i)$ the set of its outgoing arcs and $\mathrm{Inc}(i)$ its incoming vertices.

The flowing problem is the following. Among the vertices of G, one chooses a vertex called Source (denoted by S from now on), and another one called Well (denoted by W from now on). The goal is to find a function φ over A such that $\forall a \in A\, \varphi(a) \leq C(a)$,

and for all vertices (except S and W) of G, the Kirchhoff's laws apply, which means that

$$\sum_{a \in Inc(v)} \varphi(a) = \sum_{a \in Out(v)} \varphi(a).$$

The value of the flow φ is

$$\sum_{a \in Out(S)} \varphi(a) = \sum_{a \in Inc(W)} \varphi(a).$$

The problem is to find a flow maximizing this value.

The algorithm usually used to solve this problem is the well known Ford-Fulkerson Algorithm, enhanced in the section 5.

3 Presentation of Radix Trees

Radix trees are trees allow to store data in a hierarchical way. Let us suppose for instance that we want to store a set of words over an alphabet $\{a, b, c\}$. Then the edges are labeled with the letters of the alphabet (or an empty letter) and the words are obtained by reading the path between the root of the tree and any of its leaves. For instance the set $\{a, ab, aba, abc, bab, bac\}$ may be stored as (see Figure 1) :

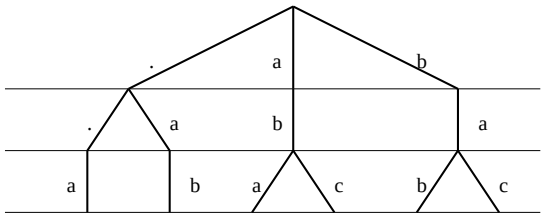

Figure 1: The set $\{a, ab, aba, abc, bab, bac\}$ stored in a radix tree

The efficiency of the radix tree structure is revealed by the computations it allows. For instance, a set of integers may be stored as a set of words by writing them in a given basis (and adding 0 at the left of shorter integers). Then a set of couples of integers may also be stored the same way. Let us take an example : the set $S = \{(0, 1), (0, 2), (1, 3), (0, 4),$-$(3, 1), (2, 2)\}$ may be written in basis 2 as

$$S = \{(000, 001), (000, 010), (001, 011), (000, 100), (011, 001), (010, 010)\}$$

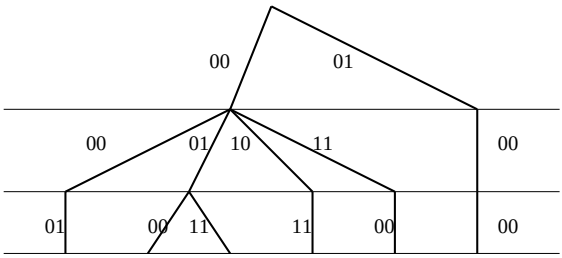

Figure 2: The set $S = \{(0,1),(0,2),(1,3),(0,4),(3,1),(2,2)\}$ stored in a radix tree

and, using the alphabet $\{00, 01, 10, 11\}$ may be rewritten as $S = \{00.00.01, 00.01.00, 00.01.11, 01.00.00, 00.10.11, 00.11.00\}$ (the i-th digit of an element is made of the catenation of the i-th digits of the corresponding couple). This store may finally be stored in the set (see Figure 2).

These sets allow efficient operations such as intersection, union, complementation (for set operations) and when the elements stored are integers or couples of integers one may also perform a translation of the whole set asymptotically faster than by adding the constant to all the elements of the set. Let us furthermore notice that the elements are stored sorted.

It is also possible to advantageously store a graph in a radix tree, by thickening the graph.

Definition 3.1 *Let $G = (V, A)$ be a graph where $V = \{v_0, \ldots, v_{v-1}\}$. Let $G' = (V', A')$ be an unweighed graph where $V' = \{v'_0, \ldots, v'_{v'-1}\}$ is a set of $v' = \lceil \frac{v}{2} \rceil$ vertices and such that*

$$\forall i, j \leq v', (v'_i, v'_j) \in A' \Leftrightarrow \exists a \in \{2i, 2i+1\}, b \in \{2j, 2j+1\},\ (v_a, v_b) \in A.$$

Such a graph G' is said to be a thickening of G or equivalently G is a refinement of G'.

Remark 3.2 Any equivalence relation may be used to define the thickening of a graph (in the definition two vertices are in the same equivalence class if and only if they have the same quotient by 2).

In other words, the vertices of V are grouped by two and there exists an arc between any two couples if and only if there exists an arc between an element of the first couple and an element of the second couple. A graph may be recursively thickened as long as the obtained graph has a number of arcs a and of vertices v verifying $P(\frac{a}{s^2}) \geq \frac{1}{2}$ where

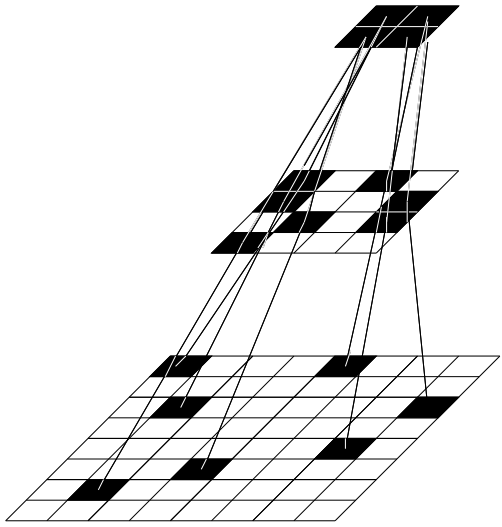

Figure 3: a Graph thickened

$P(X) = 1 - \frac{3}{2}X + X^2 - \frac{1}{4}X^3$. One may finally store a graph in a radix tree. Let us consider for instance an unweighed graph by its adjacency matrix. The adjacency matrix is then "summarized" as in Figure 3.

In the next sections we shall explain how this data structure may be used to profile "divide and conquer" algorithms on a number of problems.

3.1 Database Management

A relational database is made of several relations linked by sets of primary keys and foreign keys. A primary key of a relation T is an attribute or a set of attributes such that T may not contain two lines whose values are equal on these attributes. Another way to say it is that the value of a primary key identifies a single line of T. Any relation has at least a primary key.

A foreign key of a relation T is an attribute or a set of attributes targeting a primary key of a relation T'. In other words a foreign key is a recall of a primary key. A foreign key replaces all the attributes of T' for the line identified by its value. It is a cheap storage of all the values of T' (only the values of the primary key of T' are stored instead of all its attributes). For instance if a relation contains the data of a set of clients, each command should not contain all the data of the client but only the value of the primary key of the relation storing the clients data.

Given a relation T, one builds the matching "expansion relation" , builds the thesauruses and radix trees and answer SQL queries (see below). For a full description of this algorithm one may refer to [13]

Expansion Relations

We give a recursive definition of a expansion relation. It is a denormalized version of the database. The foreign keys are replaced by the attributes of the targeted relations.

Definition 3.3 *The expansion of a foreign key belonging to a relation is the replacement of the attributes of this foreign key by the attributes of the targeted relation. Let T be a relation. The related expansion relation $E(T)$ is the relation obtained by expanding recursively all its foreign keys.*

Building Indexes

Once the expansion tables are built, the database is made of relations no more linked because the jointures have been formally expanded in each relation. Then one builds the thesaurus of each attribute and store for each element of the thesaurus the set of the line indexes it appears at. For example, let us consider the attribute in Table 1:

0	1	2	3	4	5	6	7	8	9	10
Male	Female	Female	Male	Female	Male	Male	Female	Female	Male	Male

Table 1: An example of simple relation

Then its thesaurus is {Female, Male} and the indexes at which Female appear (resp. Male) is $\{1, 2, 4, 7, 8\}$ (resp. $\{0, 3, 5, 6, 9, 10\}$). The radix trees associated to the words of the thesaurus are shown in Figure 4).

Solving SQL Requests

We give briefly indications to solve SQL requests. For more details one may refer to [13].

The main part of this work is to solve the `where` clause, composed of equalities or inequalities separated by logical operators (and, or) and joint clauses etc. This where clause returns a radix tree, which contains the line indexes answering the clause. The joint clause are irrelevant here because of the use of expansion relations.

An equality between an attribute and a constant is the simplest case because one has simply to read the corresponding radix tree. An inequality may be computed by several

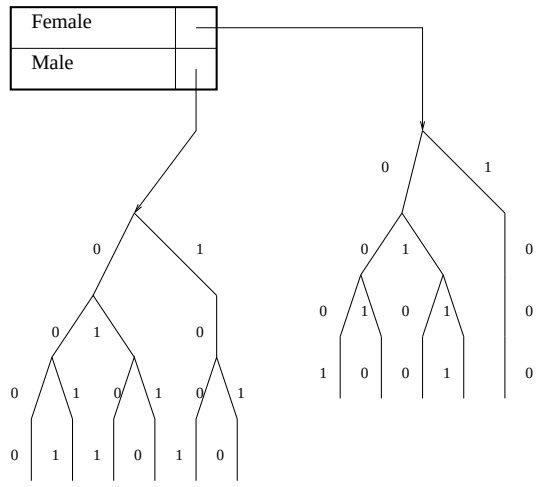

Figure 4: The indexes of an attribute

readings of radix trees and the computation of an "or". A "and" (resp. "or") clause is computed by performing the "and" (resp. "or") of the corresponding radix trees.

All the other operations may be performed in a similar way.

3.2 Image Pattern Recognition

An image may be considered as a mapping from $[\![0, L-1]\!] \times [\![0, H-1]\!]$ to a finite set (of colors), l being the width and L the height of the image. Each color appears at a set of coordinates that one may store in a radix tree. For instance the picture (see figure 5) contains three colors, black, grey and white. The black pixels are at indexes $B = \{(3,0),(4,0),$-$(2,1),(3,1),(5,1),(6,1),(2,2),(6,2),(0,3),(3,3),(4,3),(7,3),(0,4),(3,4),(4,4),(7,4),$-$(2,5),(6,5),(2,6),(3,6),(5,6),(6,6),(3,7),(4,7)\}$, the white ones at indexes $W = \{(1,0),$-$(2,0),(5,0),(6,0),(0,1),(7,1),(0,2),(3,2),(4,2),(7,2),(2,3),(5,3),(2,4),(5,4),(0,5),$-$(3,5),(4,5),(7,5),(0,6),(7,6),(1,7),(2,7),(5,7),(6,7)\}$ and the grey ones at $G = \{(0,0),$-$(7,0),(3,1),(4,1),(2,2),(5,2),(2,3),(6,3),(2,4),(6,4),(2,5),(5,5),(3,6),(4,6),(0,7),$-$(7,7)\}$.

The set of appearance of the black pixels may be rewritten, in basis 2, as $B = \{(011,000),(100,000),(010,001),(011,001),(101,001),(110,001),(010,010),(110,010),$-$(000,011),(011,011),(100,011),(111,011),(000,100),(011,100),(100,100),(111,100),$-$(010,101),(110,101),(010,110),(011,110),(101,110),(110,110),(011,111),(100,111)\}$ and the three sets may be stored in radix trees as (see figure 6):

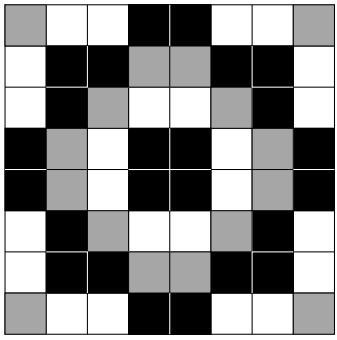

Figure 5: The pixels of an image

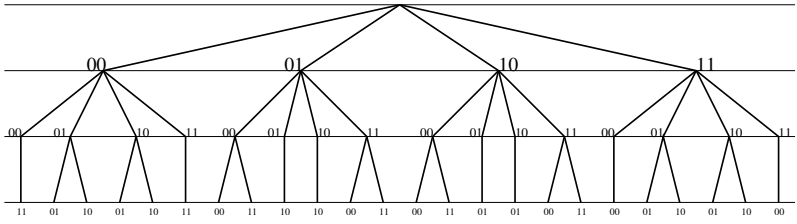

Figure 6: The radix tree of the black pixels

Such a radix tree may be translated by any vector (the computation is easy and asymptotically faster than the translation of a whole set stored in an array for instance). Then by computing $(B + (a, b)) \cap W$ one obtains the set of coordinates of the white pixels having a black pixel at relative coordinates $(-a, -b)$.

By iterating this process one finds any pattern in an image. (for more details on this algorithm one may refer to [14]).

3.3 Automatic Translation

This subsection describes a work in progress. One of the problems arising to an automated translation process comes from the fact that a word has usually several senses and each of these senses has a different translation in the target language. The Prophyre tree is a tree in which all the meanings of all the words (this tree is actually unachieved whatever the language) in a way allowing one to compute the semantic distance between two meanings of two words. Let a sentence be made of n words, m_1, m_2, ..., m_n and let $m_{i,j}$ be vertices of a graph representing the j-th meaning of m_i. Then one may build the arcs of this graph by linking $m_{i,j}$ and $m_{i+1,k}$ for any i, j, k with an arc whose weight is the semantic distance between the two involved meanings. Then by adding two vertices, e_b linked to any meaning of m_1 (with a weight 0) and e_e linked to any meaning of m_n (with a weight 0).

Then the problem of finding a pertinent sense to the sentence is to find a shortest path between e_b and e_e. Furthermore, if there exists several shortest paths between these two vertices, then there exists several meaningful choices for the senses of the words of the sentence (which allows to find automatically ambiguous sentences).

4 Shortest Path Problem

The shortest path problem is one of the oldest and more studied of the graph area. A lot of papers have been written on this subject (see [2], [7], [9], [21] or [20] for instance). Some algorithms span graphs with trees or use spanning trees (see [10] for instance), and the multicast problem is also very studied (see for instance [3], [4], [12], [17], [18], [19], [22]).

The best algorithm known up to now to solve the shortest path problem is the Dijkstra's algorithm. It runs in time $O(V \ln E)$ where V is the number of vertices of the graph and E the number of edges (see [5], [11] or [23]). This algorithm has been improved by many authors since 1959.

Definition 4.1 *Let $G = (V, A)$ be an unweighted graph. Let R be an equivalence relation*

among the vertices of G. We shall call thickening of G the graph $G'(V', A')$ for the relation R where V' is the set of equivalence relations for R and $\forall V_1, V_2 \in V'$, $(V_1, V_2) \in A' \Leftrightarrow \exists v_1 \in V_1, v_2 \in V_2$ such that $(v_1, v_2) \in A$. In such a case we shall say that G' is a thickening of G or equivalently that G is a refinement of G'.

Let us denote by π the application which maps a vertex v of V onto the vertex v' of G' such that $v \in v'$. Let $v_1, v_2 \in V$ and let us suppose that there exists an integer k and a path from v_1 to v_2 of length k. Then there exists a path of length k between $\pi(v_1)$ and $\pi(v_2)$. Indeed if $p = (v_0, v_1, \ldots, v_k)$ is a path in G of length k between v_0 and v_k then $p' = (\pi(v_0), \pi(v_1), \ldots, \pi(v_k))$ is a path of length k in G' between $\pi(v_0)$ and $\pi(v_k)$. Then we shall call p a refinement of p' or, equivalently, p' a thickening of p. The problem is then to find a path of minimal length in G' which may be refined in G.

Let $G_0 = G, G_1, \ldots, G_t$ a sequence of graphs such that $\forall i \leq t-1$, G_{i+1} is a thickening of G_i. If we call v_i the number of vertices of G_i and a_i its number of arcs, we suppose that $\forall 0 \leq i \leq t$, $P(\frac{a_i}{v_i^2}) \geq \frac{1}{2}$. An algorithm to find the shortest path between two vertices of G, say d and a is the following.

```
ShortestPath(G[])
Input :  a sequence G[] of thickenings of G_0,
         two vertices of G_0, d and a
Output :  the shortest paths between d and a.
Variables :  l, th :  integer
l ← 0
While no path is found between d and a Do
      P[t] ← all the paths of length l in G[t] between d[t]
            and a[t]
      th ← t
      While (th > 0 And P[th] is not empty) Do
            Foreach path p in P[th] Do
                  refine p and if it may be refined
                        store its refinement in P[th − 1]
            EndForEach
            th ← th - 1
      EndWhile
      if P[0] is not empty
         Then the shortest paths are found:  output them
              and exit
      Else l ← l + 1
      EndIf
```

```
EndWhile
End
```

In $G[t]$ the shortest paths are computed thanks to a refinement of the BFS algorithm. Let us detail this part.

4.1 Finding paths in $G[t]$

Let $G = (V, A)$ be an unweighed graph such that $P(\frac{a}{v^2}) < \frac{1}{2}$ with $P(X) = 1 - \frac{3}{2}X + X^2 - \frac{1}{4}X^3$. Let d and a be two of its vertices and let us compute the shortest paths between d and a. We suppose furthermore that the adjacency vertices of any given vertex are given through a sorted list of vertices.

Let $S \subset V$ be a nonempty subset of V. The cost of the computation of the set $S' = \{v' \in V, \exists v \in S, (v, v') \in A\}$ is averagely majored by the cardinality of S'. Indeed, the set S' may be stored in a bit vectors in which one adds the vertices linked to the elements of S by a bit-or. In a similar way, the set $S'' = \{v'' \in V, \exists v \in S, (v'', v) \in A\}$ is averagely majored by the cardinality of S'' (same reason). The operation consisting in computing S' (resp. S'') knowing S will be called an increment (resp. decrement). Furthermore, the cost of the intersection of two sorted sets is majored by the sum of their cardinals.

Let us consider the following algorithm, consisting in incrementing $\{d\}$, decrementing $\{a\}$ and intersect them until there exists a nonempty intersection. Then one may read the shortest paths between d and a.

This algorithm may be written the following way:

```
ShortestPath(G)
Input :  a graph G, two vertices of G, d and a
Output :  the shortest paths between d and a.
Variables Increments[], Decrements[], inc=0,
      dec=0, Parity = 0, Paths[]
Increments[Inc] = {d}
Decrements[Dec] = {a}
While (Increments[Inc] ∩ Decrements[Dec] = ∅)
      if (Parity = 0)
         Inc = Inc + 1
         Increments[Inc] = Increment(Increments[Inc - 1])
      Else
         Dec = Dec + 1
```

```
         Decrements[Dec] = Decrement(Decrements[Dec - 1])
      EndIf
EndWhile
Paths[Inc] = Increments[Inc] ∩ Decrements[Dec]
For I = Inc - 1 Downto 0 Do
   Paths[I] = Decrement(Paths[I + 1]) ∩ Increments[I]
EndFor
For I = Dec - 1 Downto 0 Do
   Paths[Inc + Dec - I] = Increment(Paths[Inc + Dec - I - 1])
        ∩ Decrements[I]
EndFor
ReadPaths(Paths)
End
```

After this treatment, the variable Paths contains sets of vertices and a vertex v lies on a shortest path between d and a, i arcs away from d (and $l - i$ arcs away from d where l is the length of the path) if and only if v belongs to $Paths[i]$.

The ReadPaths algorithm consists in reading the shortest paths through a deep first journey of paths. This last algorithm may be written as:

```
ReadPaths(Paths[], l, Current = 0, Res[] = 0)
Input :  The shortest paths given in an array of sets of
         vertices, l the paths lengths
Output :  the shortest paths between Paths[0] and Paths[l].
If  (Current = l)
   Output (Res[])
Else
   Foreach vertex v in Paths[Current] Do
      Res[Current] = v
      Memory[Current + 1] = Paths[Current + 1]
      Paths[Current + 1] = Paths[Current + 1] ∩ Increment({v})
      ReadPaths(Paths, l, Current + 1, Res[])
      Paths[Current + 1] = Memory[Current + 1]
   EndForeach
End
```

The complexity if the H algorithm is majored by $O(v)$ where v is the number of vertices of the graph, because the first steps, consisting in computing $Paths[]$ is majored

by $O(v)$ and the ReadPaths algorithm is also majored by the number of vertices involved in the shortest paths, v.

Example

Let us compute all the shortest paths between S and W in the following graph (see figure 7).

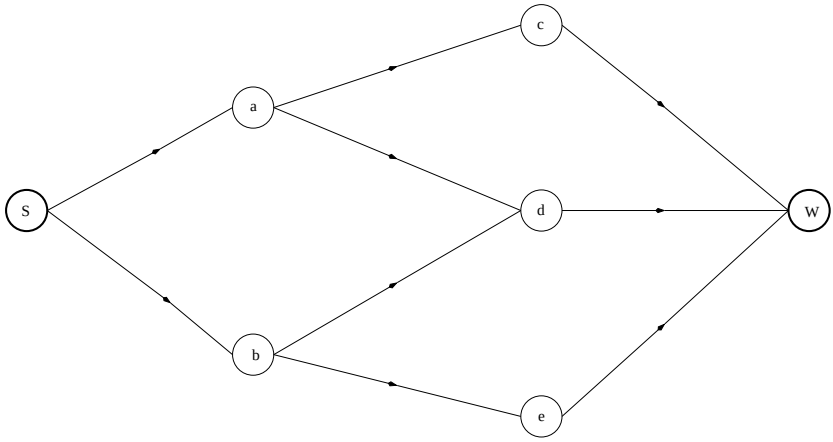

Figure 7: The graph G_0 itself

Then the vertices are gathered for instance two by two and one obtains the graph (see figure 8):

By grouping again the vertices two by two one obtains (see figure 9):

and stops here because $\frac{a_2}{a_1} \leq \frac{1}{2}$.

- The first step is to look for a path of length 0, which does not exist in G_0 because it does not exist in G_2.
- Now there exists a path of length 1 between $\overline{S}$ and $\overline{W}$, which is $\overline{S} \rightarrow \overline{W}$. This path may not be refined in G_1 because there is no path in G_1 between $\overline{S}$ and $\overline{W}$.
- Let us look for paths of length 2 between $\overline{S}$ and $\overline{W}$ in G_2. There exists two such paths, $P_1 = \overline{S} \rightarrow \overline{S} \rightarrow \overline{W}$ and $P_2 = \overline{S} \rightarrow \overline{W} \rightarrow \overline{W}$.

  The first of these two paths may be refined in G_1 in $P_1' = \overline{S} \rightarrow \overline{b} \rightarrow \overline{W}$ and P_2 may be refined in G_1 in $\overline{S} \rightarrow \overline{d} \rightarrow \overline{W}$.

  Now neither the path P_1' nor P_2' may be refined in G_0.

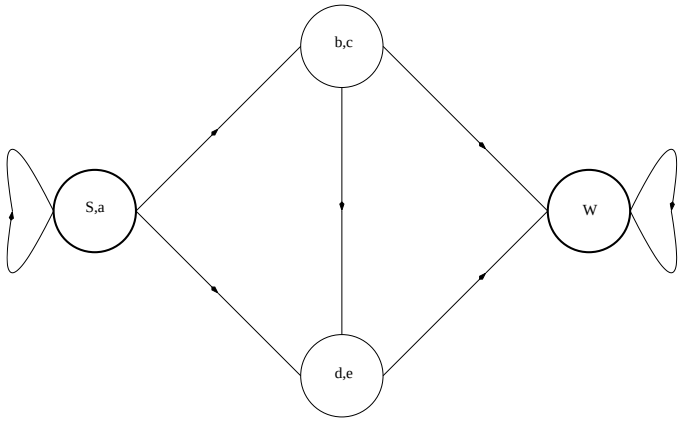

Figure 8: G_1, a thickening of G_0

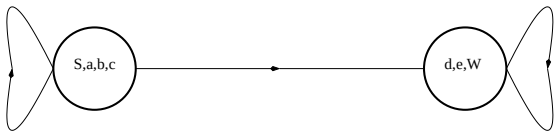

Figure 9: G_2, a thickening of G_1

- Let us look for paths of length 3. There exists three paths of length 3 in G_2 between $\overline{S}$ and $\overline{W}$: $P_1 = \overline{S} \to \overline{S} \to \overline{S} \to \overline{W}$, $P_2 = \overline{S} \to \overline{S} \to \overline{W} \to \overline{W}$ and $P_3 = \overline{S} \to \overline{W} \to \overline{W} \to \overline{W}$.

  The path P_1 may be refined in $P_1' = \overline{S} \to \overline{S} \to \overline{b} \to \overline{W}$.

  The path P_2 may be refined in three different paths, $P_2^1 = \overline{S} \to \overline{b} \to \overline{W} \to \overline{W}$, $P_2^2 = \overline{S} \to \overline{b} \to \overline{d} \to \overline{W}$, and $P_2^3 = \overline{S} \to \overline{S} \to \overline{d} \to \overline{W}$. The path P_3 may be refined in G_1 in $P_3' = \overline{S} \to \overline{d} \to \overline{W} \to \overline{W}$.

  Now let us refine these paths in G_0.

  The path P_1' may be refined in $P_1'' = S \to a \to c \to W$. The path P_2^1 may not be refined in G_0. The path P_2^2 may be refined in $S \to b \to d \to W$ and in $S \to b \to e \to W$. The path P_2^3 may be refined in $S \to a \to d \to W$. The path P_3' may not be refined in G_0.

So one finds the shortest paths between S and W, which are: $S \to a \to c \to W$, $S \to b \to d \to W$, $S \to b \to e \to W$ and $S \to a \to d \to W$.

It is clear that one may use this algorithm to find the paths of a given length between two vertices of a graph.

4.2 Complexity

In [15] the author computes the complexity of the algorithm presented roughly in the preceding section. This complexity is $O(v)$ where v is the number of vertices of the graph. Since this paper is not yet published we give here the proof of this complexity in the particular case useful in the context of flows computations, the unweighted graphs.

Let $G_0 = (V, A)$ be an unweighted graph and let us denote v its number of vertices and a its number of arcs. We suppose in this section that the equivalence classes are sets of two vertices (and at most a class made of a single vertex).

Lemma 4.2 *Let G_1 be a thickening of G_0. Then the number of vertices of G_1 is $\lceil \frac{v}{2} \rceil$ and its number of arcs is averagely $aP(\frac{a}{v^2})$ with $P(X) = 1 - \frac{3}{2}X + X^2 - \frac{1}{4}X^3$.*

Proof: The number of vertices of a thickening of G_0 in which the vertices are grouped by two is obviously $v' = \lceil \frac{v}{2} \rceil$. There is no arc between two vertices v' and w' of G_1 if and only if there is no arc between four pairs of vertices (if $v = (v_1, v_2)$ and $w = (w_1, w_2)$, the four pairs are $(v_i, w_j)_{(1 \le i,j \le 2)}$, and we may suppose that the probability of any of these pairs to be an arc does not depend on the probability of the others). The number of arcs is hence $a' = \lceil \frac{v}{2} \rceil^2 (1 - (1 - \frac{a}{v^2})^4)$. So $a' = aP(\frac{a}{v^2})$ with $P(X) = 1 - \frac{3}{2}X + X^2 - \frac{1}{4}X^3$.

Remark 4.3 The ratio $\frac{a}{v^2}$ increases in refinements.

Let us prove this remark. Let G be a graph and G' be one of its refinements in which the vertices have been gathered by two. Let us denote by a, v, a' and v' the number of arcs and vertices of G and G'.

One has $\frac{a'}{v'^2} = 4P(\frac{a}{s^2})\frac{a}{v^2}$. Since $P(x)$ decreases from 1 towards $\frac{1}{4}$ when x raises from 0 to 1, the remark is proved.

Lemma 4.4 *The average number of paths of length $l \in \mathbb{N}$ between two vertices in G_0 is $N(G_0, l) = \frac{1}{v}(\frac{a}{v})^l$.*

Proof: by induction on the length of the paths. The graph contains a arcs and v vertices. The formula is correct for paths of length 0 (the average number of paths of length 0 between two vertices is $\frac{1}{v}$). The average number of paths of length 1 between two vertices is hence $\frac{a}{v^2}$. If we suppose that the average number of paths of length l between two vertices is $\frac{1}{v}(\frac{a}{v})^l$ then each of these paths may averagely be prolonged in $\frac{a}{v}$ different arcs and the number of paths of length $l+1$ between two vertices is averagely $\frac{1}{v}(\frac{a}{v})^{l+1}$.

Refinements

An algorithm to refine a path from G_{k+1} to G_k is the following:

```
RefinePath(Graph G_{k+1}, Graph G_k, Path P)
Input:  a graph G_k, a refinement of G_k, G_{k+1}, a path P in G_k
Output:  All the refinements of P in G_{k+1}
If the first arc of P is refinable
       Truncate P by its first arc (a,b)
       Foreach refinement (c,d) of (a,b)
               Answer.FirstElement =(c,d)
               Answer.Catenate(Refine(G_{k+1}, G_k, P)
       EndForEach
       Put back its first arc to P
EndIf
End
```

Lemma 4.5 *Let G be a graph containing v vertices and a arcs, and G' be one of its thickenings in which the vertices are gathered two by two. Let v_1 and v_2 be two vertices of*

G and let us suppose that there exists a path $w = (w'_0 = v'_1, w'_1, \ldots, w'_l = v'_2)$ of length l (we denote as usually v'_1, resp. v'_2, the equivalence class of v_1, resp. v_2 in G'). Then there exists averagely $(\frac{2}{v})^{l-1}(\frac{a}{v})^l$ paths of length l in G between v_1 and v_2 which are refinements of the path w.

Proof: The number of possible refinements of w is 2^{l-1}. Each of these possible paths has a probability $\frac{1}{v^{l-1}}(\frac{a}{v})^l$ to exist. So the number of refinements of w is averagely $(\frac{2}{v})^{l-1}(\frac{a}{v})^l = \frac{v}{2}(\frac{2a}{v^2})^l$.

The most expansive operation while refining a path from G' to G is to check wether a given arc exists in G. The complexity of the refinement is hence majored by this number of checks.

Lemma 4.6 *The number of checks while refining a path of length l from G to G' is $\sum_{i=0}^{l} \frac{v}{2}(\frac{2a}{v^2})^i = \frac{v}{2}\frac{1-(\frac{2a}{v^2})^{l+1}}{1-\frac{2a}{v^2}}$.*

Proof: when one has refined paths of length $i-1$, one has only to extend the found paths by one arc. The number of checks is hence the sum of the number of paths of all the lengths from 0 to l.

Let us detail an hierarchical algorithm (H-algorithm) :

```
ShortestPath(Graph G[0..M], int o, int e)
Input:  a sequence of graphs, one being a thickening of the
          preceding, all verifying P(a/v²) ≥ 1/2,
        an origi vertex index o
        an extremity vertex index e
Output:  the shortest paths between v_o and v_e.
Variables :
            l :  Path's length
l ← 0
While the shortest paths qre not found Do
   Find the shortest paths P_i in G_M thanks to the refined BFS
   Foreach shortest path P
      For k = M - 1 Downto 0
         P[i, k-1] = Refine(G[k+1], G[K], P[i, k])
      EndFor
      If there exists a path in G[0] output the paths and exit
      EndIf
   EndForEach
```

```
EndWhile
```

Theorem 4.7 *The mean complexity of the computation of the shortest path between two vertices of a graph by using the H-algorithm is $C(G) \leq v$.*

Proof: The mean complexity of the refinements of paths of length l from a graph made of a single vertex to the initial graph G_0 is

$$\begin{array}{rl} C(l) & = \sum_{i=0}^{\ln v} \frac{v_i}{2} \left(\frac{2a_i}{v_i^2}\right)^l \\ & \leq \sum_{i=0}^{\ln v} \frac{v_i}{2} \\ & \leq \sum_{i=0}^{\ln v} \frac{v_0}{2^{i+1}} \\ & < v_0 \end{array} .$$

The complexity of the shortest path computation is majored by the complexity of the computations of the refinements of paths of length l

5 The Flow Problem

We now present a new algorithm, which will be fully detailed in [16], for maximizing the flow over a network.

5.1 The Ford-Fulkerson Algorithm

```
MaximumFlow(G)
input:  a graph G, two vertices s and t
output:  a flow Φ = (φ_{a_1}, ..., φ_{a_|A|})
(0) Initialisation :  φ_(a) = 0,  ∀a ∈ A
                      v(Φ) = 0
                      i=0
                      G_R^0 = G
(1) Find a shortest path from s to t in the residual graph G_R^i
(2) If there is no such path then END
(3)          Else μ ←a shortest path
                     (consisting in the fewest number of
                          arcs)
                     from s to t in G_R^i
```

(4) cap(μ)= the minimum capacity of residual capacities of μ in G_R^i
(5) $v(\Phi) \leftarrow v(\Phi)+$ cap(μ)
(6) $\forall a \in \mu \cap A$, $\varphi_{(a)} \leftarrow \varphi_{(a)} + cap(\mu)$
(7) $\forall a \in (\mu \notin \mu \cap A)$, $\varphi_{(a)} \leftarrow \varphi_{(a)} - cap(\mu)$
(8) $i \leftarrow i + 1$
(9) Build the new residual graph G_R^i

The initial algorithm of Ford and Fulkerson, in [8], runs in pseudopolynomial time. The implementation proposed here is due to Edmonds and Karp, [6], and runs in $O(|V|^2|A|)$ time. There exist others implementations of the algorithm of Ford and Fulkerson, which are detailed in the reference book [1].

Radix Trees and Shortest paths

The main step of the preceding algorithm is the step 1, which may be performed thanks to the H-algorithm. The complexity of the Ford-Fulkerson Algorithm is $O(n^3)$ when this step is done by the use of the Dijkstra's algorithm, whose complexity is $O(n^2)$. the use of the H-algorithm allows a mean complexity in $O(n^2)$.

An Example

Let us consider the following graph (see figure 10) in which we wish to optimize the flow between the source S and the well W.

The shortest paths in terms of number of arcs are $P_1 = S \rightarrow a \rightarrow c \rightarrow W$, $P_2 = S \rightarrow a \rightarrow d \rightarrow W$, $P_3 = S \rightarrow b \rightarrow d \rightarrow W$ and $P_4 = S \rightarrow b \rightarrow e \rightarrow W$.

After the path P_1 (of flow 6) is considered as a part of the flow, the residual graph is (see figure fig:flows2):

After that paths P_2 (capacity 1), P_3 (capacity 5) and P_4 (capacity 3) are included in the flow the residual graph is (see figure fig:flows3):

Since in this last residual graph the source and the well are no more connected, the maximal flow is reached, with a full capacity of $6 + 1 + 5 + 4 = 16$.

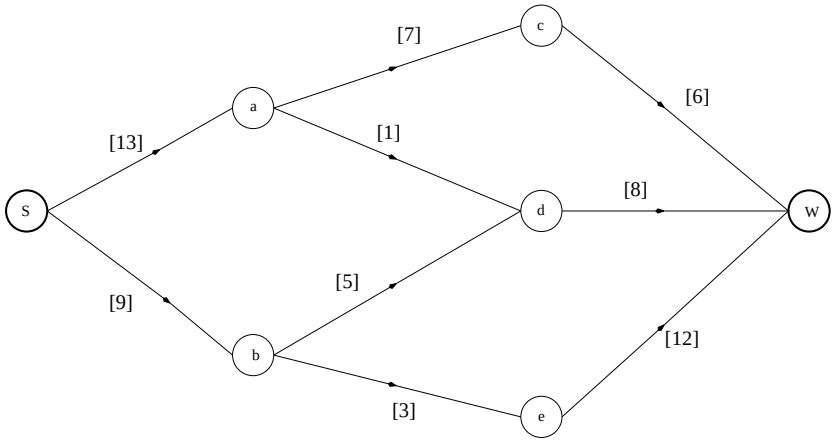

Figure 10: An optimization Problem

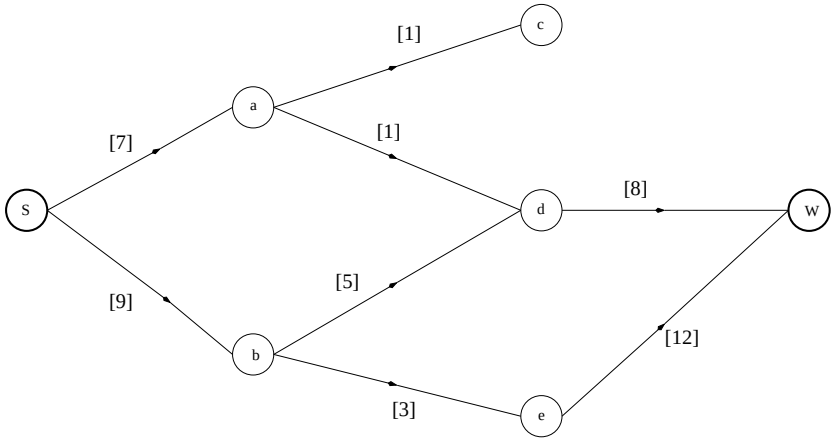

Figure 11: After a first path part of the flow

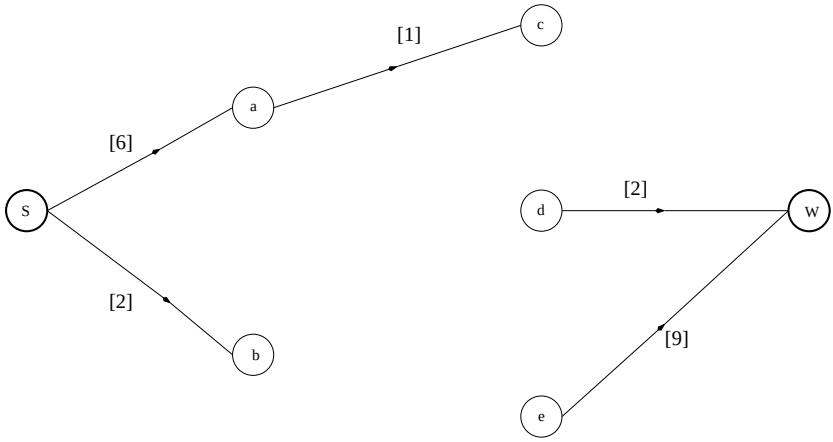

Figure 12: The residual graph after the shortest paths are included in the flow

6 Conclusion

The flowing problem has a important applications: transportation of energy or water, schedule of network tasks etc.. . . The algorithm presented in this paper allows one to compute the optimal flow over a network in a mean time of $O(n^2)$ while the Ford-Fulkerson algorithm has a complexity in $O(n^2)$ where n is the number of vertices of the graph.

References

[1] Ahuja R.K, Magnanti T.L and Orlin J.B., *Network flows: theory, algorithms and applications*, Prentice Hall, New Jersey, 1993.

[2] Bertsekas, D. and Gallager, R. *Data Networks*, Prentice-Hall, Inc., 1987.

[3] Choplin, S., *Dimensionnements de Résaux Virtuels de Télécommunications*, Université de Sophia-Antipolis, Thèse, November 2002.

[4] Crawford, J.S., Waters, A.G., *Low Cost Quality of Service Multicast Routing in High Speed NEtworks.*, Technical Reports 13-97, University of Kentat Canterbury, December 1997.

[5] DIJKSTRA, E W. *A note on two problems in connexion with graphs.*, Numer. Math. 1 (1959), 269-271.

[6] Edmonds J. and Karp R.M.,*Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of ACM, 19, 1972, pp 248–264,

[7] Floyd, R. W., *Algorithm 97, Shortest Path*, Communications of the ACM, 5 (6):345, 1962.

[8] Ford L.R., Fulkerson D.R., *Maximum flow through a network*, Canadian Journal of Mathematics, 8,1956, pp 399-404,

[9] Gibbons, A., *Algorithmic Graph Theory*, Cambridge University Press, 1989.

[10] Gilbert, E. N., Pollack, H., O., *Steiner Minimal Trees*, SIAM Journal on Applied Mathematics, 16, 1968.

[11] Jhonson, D. B., *Efficient Algorithms for Shortest Paths in Sparse Networks*, Journal of the ACM (JACM), v.24 n.1, p.1-13, Jan. 1977

[12] Kompella, P. *Multicast Routing Algorithms for Multimedia Traffic*, PhD Thesis, University of California, San Diego, USA, 1993.

[13] Koskas, M., *A Hierarchical Database Manager*, Annales du Lamsade, **2**, 2004, pp 277-317.

[14] Koskas, M., *A Divide and Conquer Algorithm of Image Pattern Finding*, submitted to publication

[15] Koskas, M., *A Divide And Conquer Algorithm To Solve The Shortest Path Problem*, submitted to publication.

[16] Koskas, M., Murat, C., *Flows !*, in preparation.

[17] Moy, J., *Multicast Extensions to OSPF*, RCF 1584, march 1994.

[18] Salama, H.F., Reeves, D.S., Vinitos, I., Tsang-Lin, S., *Evaluation of Multicast Routing Algorithms for Real-Time Communications on High Speed Networks*, Proceedings of the 6-th IFIP Conference on High Performance Networks (HPN'95), 1995.

[19] Salama, H.F., Reeves, D.S., Vinitos, I., *Evaluation of Multicast Routing Algorithms for Real-Time Communication on High Speed Networks*, IEEE Journal on Selected Area in Communications, 15(3):332-345, April 1997.

[20] Widyono, R., *The Design and Evaluation of Routing Algorithms for Real-Time Channels*, Tr-94-024, University of California at Berkeley and International Computer Science Institute, September 1994.

[21] Waxman, B.M., *Routinf of Multipoint Connection*, IEEE journal on selected areas in communications, 6(9):1617-1622, 1988.

[22] Waters, A. G., Crawforf, J.S., *Low-Cost ATM Multimedia Routing with Constrained Delays.*, Multimedia Telecommunications and Applications (3rd COST 237 Workshop, Barcelona, Spain), 23-40. Springer, November, 1996.

[23] Weide, B., *A Survey of Analysis Techniques for Discrete Algorithms*, ACM Computing Surveys (CSUR), v.9 n.4, p.291-313, Dec. 1977