

Implicit Flow Maximization by Iterative Squaring*

Daniel Sawitzki[†]

University of Dortmund, Computer Science 2

D-44221 Dortmund, Germany

`daniel.sawitzki@cs.uni-dortmund.de`

January 7, 2004

Abstract

Application areas like logic design and network analysis produce large graphs $G = (V, E)$ on which traditional algorithms, which work on adjacency list representations, are not practicable anymore. These large graphs often contain regular structures that enable compact implicit representations by decision diagrams like OBDDs [2, 3, 17]. To solve problems on such implicitly given graphs, specialized algorithms are needed. These are considered as heuristics with typically higher worst-case runtimes than traditional methods. In this paper, an implicit algorithm for flow maximization in 0–1 networks is presented, which works on OBDD-representations of node and edge sets. Because it belongs to the class of layered-network methods, it has to construct blocking-flows. In contrast to previous implicit methods, it avoids breadth-first searches and layer-wise proceeding, and uses iterative squaring instead. In this way, the algorithm needs to execute only $O(\log^2 |V|)$ operations on the OBDDs to obtain a layered-network or at least one augmenting path, respectively. Moreover, each OBDD-operation is efficient if the node and edge sets are represented by compact OBDDs during the flow computation. In order to investigate the algorithm's behavior on large and structured networks, it has been analyzed on grid networks, on which a maximum flow is computed in polylogarithmic time $O(\log^3 |V|)$ and space $O(\log^2 |V|)$. In contrast, previous methods need time and space $\Omega(|V|^{1/2} \log |V|)$ on grids, and are beaten also in experiments for $|V| \geq 2^{26}$.

*Extended version of a paper appeared in SOFSEM 2004, LNCS 2932, pp. 301–313.

[†]Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research Cluster “Algorithms on Large and Complex Networks” (1126).

1 Introduction

Algorithms on graphs $G = (V, E)$ typically work on adjacency lists, which contain for every node $v \in V$ the set of its adjacent nodes $\text{adj}(v) = \{w \mid (v, w) \in E\}$. This kind of representation has size $\Theta(|V| + |E|)$, and is called *explicit*.

However, there are application areas in which problems on graphs of such large size have to be solved that an explicit representation on today's computers is not possible. In the verification and synthesis of sequential circuits, state-transition graphs with for example 10^{27} nodes and 10^{36} edges occur. Other applications produce graphs which are representable in explicit form, but for which even runtimes of efficient polynomial algorithms are not practicable anymore. Modeling of the WWW, street, or social networks are examples of this problem scenario.

Yet we expect the large graphs occurring in application areas to contain structures and regularities rather than to be randomly originated. Therefore, our motivation is to use data structures that reward these graph properties with good compression. *Ordered Binary Decision Diagrams* (OBDDs) [2, 3, 17] are graph-based data structures for Boolean functions which meet this requirement. In order to represent a graph $G = (V, E)$ by an OBDD, its edge set E is considered as a Boolean *characteristic function* χ_E , which maps binary encodings of E 's elements to 1 and all others to 0. This representation is called *implicit*.

The size of an OBDD for E is bounded both by $O(|V|^2)$ and $O(|E| \log^2 |V|)$ for every graph G . Hence, it is not essentially larger than the corresponding adjacency matrix resp. the adjacency lists. However, we hope that advantageous structural properties of G lead to "small," that is sublinear OBDD-sizes. Examples are grid graphs [19, 15], which have OBDDs of logarithmic size, and cographs [18], which have OBDDs of size $O(|V| \log |V|)$.

OBDDs offer a set of functional operations which are efficient w. r. t. the sizes of the participating OBDDs. These operations like disjunction and quantification have to suffice for the design of algorithms working on implicitly defined data (called *implicit algorithms*). Although each single operations is efficient, a sequence of $O(\log |V|)$ operations may generate OBDDs of exponential size. Thus, the over-all runtime of an algorithm is essentially influenced by the size of both the input OBDDs and of OBDDs generated during the algorithm execution. In general, it is difficult to analyze these sizes for an interesting input subset. Sometimes, the number of OBDD-operations is bounded as a hint on the real over-all runtime [1, 14], while most papers on OBDD-algorithms contain only experimental results.

Because implicit algorithms typically have a higher worst-case runtime than corresponding algorithms which work on traditional adjacency lists (called *explicit algorithms*), they can be considered as heuristics to save time and/or space when the input graphs are heavily structured and possibly too large for an explicit representation (e. g., by adjacency lists). Then, we hope that each OBDD-operation implicitly processes as many nodes and edges as possible in parallel. Node map-

pings for example can be computed by $O(\log |V|)$ OBDD-operations using so called priority functions (see Section 4).

Implicit OBDD-algorithms for some particular graph problems like reachability analysis have been well studied in the context of logic design and sequential circuits [10]. Newer research tries to attack more general graph problems by implicit methods. The algorithm of Hachtel and Somenzi [11] represents one of the first steps in this direction. It solves the maximum flow problem on implicitly given 0–1 networks, i. e., it computes an edge-disjoint path set of maximum cardinality between a source node $s \in V$ and a terminal node $t \in V$ in a network $N = (V, E, s, t)$.

Although Hachtel and Somenzi obtained good experimental results, they mention an inherent sequential property: The individual processing of network layers, whose number may be $\Omega(|V|)$, may enforce super-linear runtime $\Omega(|V| \log |V|)$ even if all participating OBDDs are small. The algorithm presented in this paper circumvents this problem by using the technique of *iterative squaring* (therefore called *IS-algorithm* in the following). Its essential idea is to construct graph paths by iteratively doubling their lengths. Simple problems like reachability analysis can be easily performed that way. The advancement of this work is to utilize iterative squaring also for the more sophisticated problem of implicit flow maximization. The IS-algorithm computes at least one augmenting path by $O(\log^2 |V|)$ OBDD-operations, while this may suffice for the whole flow maximization in the best case. The latter applies especially if the maximum flow value is constant, and enables exponential less OBDD-operations than Hachtel and Somenzi’s algorithm on graphs with long augmenting paths.

Although reducing the number of operations, iterative squaring is known to result in very large intermediate OBDDs within implicit algorithms in the area of logic design (e. g., see [12]). Nevertheless, we pursue this approach with the focus on heavily structured networks with high diameter, on which we expect efficient over-all runtimes. This is confirmed both by analytical and experimental results for grid networks [15] (see Section 9), on which the IS-algorithm takes over-all runtime $O(\log^3 |V|)$, while Hachtel and Somenzi’s method needs $\Omega(|V|^{1/2} \log |V|)$ operations. Moreover, Woelfel [19] obtained the same polylogarithmical time and space results on grid graphs for his topological sorting algorithm, which also employs iterative squaring.

A further candidate for the successful employment of the iterative squaring approach is the maximization of flows over time [8, 9, 7] on a network N with transit-times $\tau: E \rightarrow \mathbb{N}$ on the edges. There, where we want to send as much flow as possible from s to t within a time horizon \mathcal{T} . This problem can be solved by computing a maximum flow on N ’s *time-expanded network* N^{te} , which contains a copy of N for every discrete time step $\{0, \dots, \mathcal{T}\}$. N^{te} ’s diameter grows at least linear w. r. t. \mathcal{T} , i. e., it has pseudopolynomial size. If τ has certain properties, the size of N^{te} ’s implicit OBDD-representation depends only polylogarithmically on \mathcal{T} [16]. Though, Hachtel and Somenzi’s algorithm always needs time $\Omega(\mathcal{T} \log |V|)$

on such networks. In contrast, there exist even simple examples of time-expanded networks on which the IS-algorithm gets along with $O(\log^k |V|)$ operations for some $k > 0$.

The paper is organized as follows: Section 2 gives foundations on flow maximization using layered-networks. In Section 3, we consider the principles of implicit graph representation by OBDDs. In Section 4, priority functions are introduced, which play a central role in the computation of augmenting paths. Section 5 gives a brief overview of the IS-algorithm’s modules, while separate sections are dedicated to their detailed discussion: Section 6 describes the layered-network module, which computes a layered-network according to an implicitly given input network N and an actual flow F . Sections 7 and 8 discuss two sub-algorithms used to construct a blocking-flow on a layered-network. Both have different advantages and disadvantages, why the IS-algorithm uses them in combination. In Section 9, we present analytical and experimental results for the case of grid networks, and sketch a proof on the runtime. Finally, Section 10 gives conclusions, and hints to possible future research in the area of implicit graph algorithms.

2 The Layered-Network Approach

The algorithm of Hachtel and Somenzi [11] as well as this paper’s method pursue the layered-network approach [4, 5, 6, 13] in order to compute a maximum flow in a 0–1 network. As input a network $N = (V, E, s, t)$ is given, consisting of the asymmetric directed graph (V, E) and two special nodes s , the source, and t , the terminal. Since we consider 0–1 networks, all edges have capacity 1. A map $f: E \rightarrow [0, 1]$ is called *flow* if and only if the following *flow conservation constraints* hold:

$$\forall v \in V \setminus \{s, t\}: \sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w).$$

The *flow value* is defined as $\text{val}(f) := \sum_{v|(s,v) \in E} f(s, v)$. Our aim is to compute a flow f_{\max} with maximum flow value $\text{val}(f_{\max})$. Because all edges of N have capacity 1, there is an f_{\max} with $\forall e \in E: f_{\max}(e) \in \{0, 1\}$. Therefore, we consider a flow f as the set $F := f^{-1}(1)$ of edges carrying flow.

Layered-network algorithms start with an empty flow $F := \emptyset$, and improve it during so-called *phases*. For a flow F in a network $N = (V, E, s, t)$, the *residual network* $N^F = (V, A^F, s, t)$ is obtained by reversing the edges of F in N , i. e.,

$$A^F := \{(u, v) \mid (u, v) \in (E \setminus F)\} \cup \{(u, v) \mid (v, u) \in F\}. \quad (1)$$

Paths in N^F are called *residual paths*. *Augmenting paths* are residual s – t -paths.

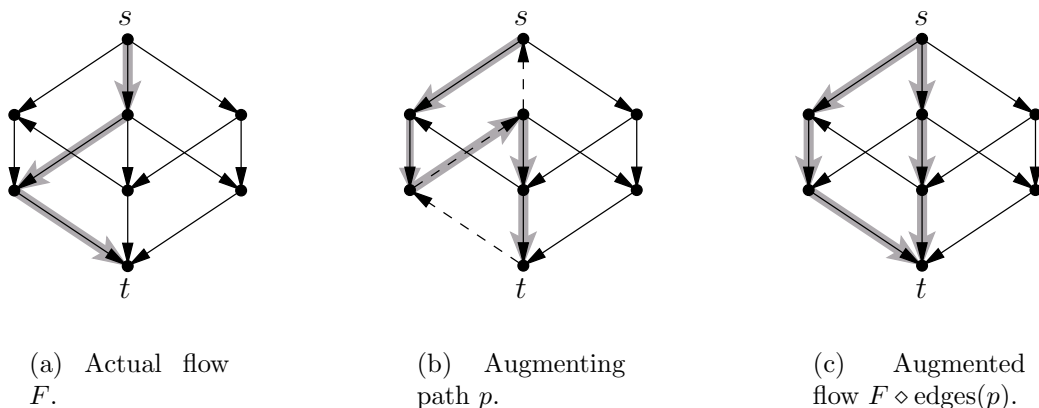


Figure 1: Example of an augmenting path p and its use to augment the actual flow F . The subfigure captions refer to the particular grey edges. Dashed edges indicate backward-edges.

For an augmenting path p with edges $\text{edges}(p)$, the *augmented flow* $F \diamond \text{edges}(p)$ is obtained by adding the *forward-edges* $\text{edges}(p) \cap E$ to F and by removing the *backward-edges* $\text{edges}(p) \setminus E$ from F (see Figure 1):

$$F \diamond \text{edges}(p) := \{(u, v) \mid (u, v) \in E, (u, v) \in \text{edges}(p)\} \cup \{(u, v) \mid (u, v) \in F, (v, u) \notin \text{edges}(p)\}. \quad (2)$$

The result is still a valid flow, while its value has increased by 1.

This idea is extended to the so-called *layered-network* U^F , which consists of all shortest augmenting paths according to the actual flow F . U^F can be separated into node and edge layers. Node layer m consists of all nodes v which are visited by a shortest augmenting path and whose shortest residual s - v -path has length m . These nodes are called V_m^F . Edge layer m contains all edges of A^F directed from node layer m to $m + 1$; it is called U_m^F :

$$U_m^F := \{(u, v) \in V_m^F \times V_{m+1}^F \mid (u, v) \in A^F\}.$$

The shortest augmenting path length is denoted by ℓ , and corresponds to the layered-network depth.

In each phase, the algorithm determines U^F , and computes a maximal (i.e., not enlargeable) set of edge-disjoint s - t -paths on it, called *blocking-flow* B^F (see Figure 2).

The actual flow F is subsequently augmented to $F^{\text{new}} := F \diamond B^F$. Thereby, every path of B^F increases F 's value by 1. It is a known maximality criterion [4] that F^{new} is a maximum flow if and only if there does not exist a further augmenting path according to F^{new} . If so, the algorithm stops; otherwise, another

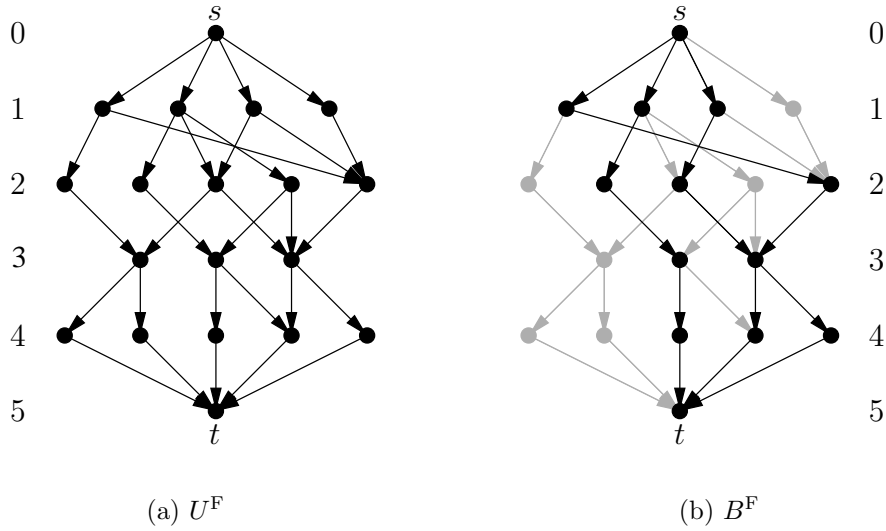


Figure 2: Example of a layered network U^F of depth 5 and an according blocking-flow B^F , each indicated by black edges.

Algorithm 1 Outline of layered-network algorithms.

```

 $F := \emptyset$ ;
while (augmenting path exists) do {Phase}
    Build layered-network  $U^F$ ;
    Construct blocking-flow  $B^F$  on  $U^F$ ;
     $F := F \diamond B^F$ ;
end while

```

phase is started. Algorithm 1 shows the outline of layered-network methods. It is known that this procedure guarantees an upper bound of $|V| - 1$ on the number of phases, due to the strictly growing depths of the layered-networks. Explicit algorithms working with adjacency lists may use a breadth-first search to build the layered-network U^F as well as a depth-first search to compute a blocking-flow B^F . In the case of 0–1 networks, both is possible in time $O(|E|)$, which leads to an over-all runtime of $O(|V| \cdot |E|) = O(|V|^3)$.

Two properties of the input network may force Hachtel and Somenzi’s algorithm to perform $\Omega(|V| \log |V|)$ OBDD-operations:

1. $\Theta(|V|)$ phases are needed for the flow maximization. It is known that networks exist which demand this amount of phases.
2. Some layered-network has depth $\Theta(|V|)$. Because the network is constructed layer-wise, this directly influences the runtime.

Independent from the OBDD-sizes, these properties destroy the possibility of

sublinear runtime, which is the hope when working with implicit algorithms, besides of saving space. While we cannot overcome the dependence on phase numbers when working with layered-network approaches, the IS-algorithm avoids the linear dependence on the layered-network depth ℓ through the use of iterative squaring.

3 Implicit Graph Representation by OBDDs

In the following, the class of Boolean functions $f: \{0, 1\}^n \rightarrow \{0, 1\}$ is denoted by B_n . The i th character of a binary string x is denoted by x_i , while $|x| := \sum_{i=0}^{n-1} x_i 2^i$ identifies its value. If not stated otherwise, x has length n , and encodes the graph node number $|x|$.

Node resp. edge sets S handled by implicit algorithms are represented by their characteristic Boolean function χ_S , which maps binary encodings of elements $e \in S$ to 1 and all others to 0. Therefore, the nodes get numbers $|x| \in \{0, \dots, |V| - 1\}$, whose binary encoding x is passed to characteristic functions as $n := \lceil \log |V| \rceil$ Boolean variables. Edges are passed as pairs of node numbers by $2n$ variables.

The Boolean functions in turn are represented by OBDDs [2, 3, 17]. In particular, the π -OBDD G_f for the function $f \in B_n$ defined on the variable set $X := \{x_0, \dots, x_{n-1}\}$ is an acyclic directed graph containing two kinds of nodes: *inner nodes* and *sinks*. Inner nodes are labeled with variables, and have two outgoing edges: the *0-edge* and the *1-edge*. Sinks are labeled with one of the function values 0 and 1 and have no outgoing edges. Finally, the *function pointer* points to the node that represents f . The variable labels at inner nodes have to respect the permutation $\pi \in \Sigma_n$ (called *variable ordering*) on every path from the function pointer to a sink. This implies that every variable occurs at most once on these paths.

In order to evaluate the function according to a given variable assignment, we start at the function pointer. If we reach an inner node v labeled with variable x_i , we follow v 's 0-edge for $x_i = 0$ respectively its 1-edge for $x_i = 1$. If we reach a sink, its label gives us the desired function value. f can be evaluated in time $O(n)$ because each variable is tested once at most.

The *size* $|G_f|$ of an OBDD G_f is measured by the number of its nodes. In general, the OBDD-representation of a function $f \in B_n$ is not canonical. Though, for a fixed variable ordering π , we can minimize a π -OBDD F for f in linear time $|F|$. In this paper, we adopt the usual assumption that all occurring OBDDs are minimized. This is reasonable since all mentioned OBDD-operations produce only minimized diagrams, which are known to be canonical.

Whether a function f is satisfiable (i. e., $f \neq 0$) can be decided in time $O(1)$. The negation \bar{f} as well as the replacement of a function variable x_i by a constant c (i. e., $f_{|x_i=c}$) is computable in time $O(|G_f|)$. Whether two functions f and g are

equivalent (i. e., $f = g$) can be decided in time $O(|G_f| + |G_g|)$. These operations are considered as *cheap*.

Further essential operations offered by OBDDs are the *binary synthesis* $f \otimes g$ for $f, g \in B_n$, $\otimes \in B_2$ and the *quantification* $(Qx_i)f$ for $Q \in \{\exists, \forall\}$. The computation of $G_{f \otimes g}$ takes time $O(|G_f| \cdot |G_g|)$, and, therefore, is considered as an *expensive* operation. The quantification $(Qx_i)f$ can be realized as two cheap operations and one binary synthesis.

Besides implicit node sets $S \subseteq V$ and edge sets $T \subseteq V^2$, there may be functions created within an algorithm which receive more than two node arguments. Let k be the maximum number of node encodings passed to a characteristic function along the algorithm execution. The OBDDs are then defined on the variable set $X_{k,n} := \{x_j^i \mid 1 \leq i \leq k, 0 \leq j \leq n-1\}$, which contains k subsets $x^i \in \{0,1\}^n$ corresponding to node numbers $|x^i|$, $1 \leq i \leq k$. We will denote implicit sets $S \in V^k$ by $S(x^1, \dots, x^k)$. Singletons $\{s\} \subseteq V$ are represented by OBDDs $s(x)$ with $s(x) = 1 \Leftrightarrow x = s$.

The *interleaved variable ordering* $\pi_{k,n} \in \Sigma_{kn}$ defined on $X_{k,n}$ maps variable x_j^i to position $jk + i$. Accordingly, $\pi_{k,n}$ tests the node numbers' bits with increasing significance, and has the appearance

$$(x_0^1, \dots, x_0^k, x_1^1, \dots, x_1^k, \dots, x_{n-1}^1, \dots, x_{n-1}^k).$$

The use of $\pi_{k,n}$ as the variable ordering for an OBDD G_f allows to swap node arguments in time and space $O(|G_f|)$ (e. g., $F(x, y) := G(y, x)$). Furthermore, it leads to small OBDDs of size $O(n)$ for important functions like value comparisons. Due to these convenient properties, we assume the use of $\pi_{k,n}$ in this paper.

The IS-algorithm will be described by functional assignments like “ $F(x) := G(x) \wedge H(x)$.” This example corresponds to the computation of a new OBDD called $F(x)$ representing the node set $F = G \cap H$. Analogously, the disjunction “ \vee ” corresponds to the set union “ \cup .” The quantification $(\exists y_{n-1} \dots \exists y_0) F(x, y)$ over the n bits of a node number y will be denoted by $(\exists y) F(x, y)$. Although this seems to be one OBDD-operation, this corresponds to $O(n)$ operations.

4 Priority Functions

A priority function $\Pi(x, y, z)$ is the implicit representation of total orders $<_x$ on $|V|$ that depend on $x \in V$, i. e., $\Pi(x, y, z) = 1 \Leftrightarrow y <_x z$. Hachtel and Somenzi's implicit max-flow algorithm [11] as well as both blocking-flow construction methods in this paper (see Sections 7 and 8) use priority functions to compute mappings of node and edge sets.

That is, we want each node $v \in V$ to select another node $w \in V$, whereby the condition $(v, w) \in P$ for $P \subseteq V^2$ has to be satisfied. Having the functions $P(x, z)$ and $\Pi(x, y, z)$, a valid selection map $D(x, z)$ can be computed through

$$D(x, z) := P(x, z) \wedge \overline{(\exists y) (P(x, y) \wedge \Pi(x, y, z))}.$$

The interpretation is that every node x selects that node z with $(x, z) \in P$ which is minimal according to $<_x$. Because of the quantification over the node y , this step takes $O(n) = O(\log |V|)$ OBDD-operations.

Both the *datum proximity* Π^{dp} and the *relative proximity* Π^{rp} [11] are priority functions with OBDD-size $O(n)$. Π^{dp} depends only formally on x , because it just compares the binary values $|y|$ and $|z|$ independent from x :

$$\Pi^{\text{dp}}(x, y, z) = 1 :\Leftrightarrow |y| < |z|.$$

The *relative proximity* Π^{rp} is essentially influenced by x . It compares the binary values of $x \oplus y$ and $x \oplus z$, where “ \oplus ” denotes the bit-wise exclusive or:

$$\Pi^{\text{rp}}(x, y, z) = 1 :\Leftrightarrow |y \oplus x| < |z \oplus x|.$$

In general, the dependence on x leads to different selection behaviors for different nodes x , and is expected to give better results when applied in implicit graph algorithms.

5 The Algorithm Using Iterative Squaring

This section introduces the IS-Algorithm, which solves the maximum flow problem on implicitly represented 0–1 networks by using the technique of iterative squaring (see Algorithm 2). As in the work of Hachtel and Somenzi, the input network $N = (V, E, s, t)$ is represented by characteristic functions $E(x, y)$, $s(x)$, and $t(x)$.

Algorithm 2 The algorithm using iterative squaring.

```

1:  $F(x, y) := 0$ ;
2:  $U(x, y) := \text{layeredNetwork}(E(x, y), s(x), t(x), F(x, y))$ ;
3: while ( $U(x, y) \neq 0$ ) do {Phase}
4:    $B(x, y) := 0$ ;
5:   repeat {Sweep}
6:      $B^*(x, y) := \text{multiPath}(U(x, y))$ ;
7:     if ( $B^*(x, y) = 0$ ) then
8:        $B^*(x, y) := \text{singlePath}(U(x, y))$ ;
9:     end if
10:     $B(x, y) := B(x, y) \vee \overline{B^*(x, y)}$ ;
11:     $U(x, y) := U(x, y) \wedge \overline{B^*(x, y)}$ ;
12:   until ( $B(x, y)$  is maximal)
13:    $F(x, y) := (E(x, y) \wedge B(x, y)) \vee (F(x, y) \wedge \overline{B(y, x)})$ ;
14:    $U(x, y) := \text{layeredNetwork}(E(x, y), s(x), t(x), F(x, y))$ ;
15: end while

```

Like every layered-network approach it contains a main loop which performs phases as long as there is an augmenting path (lines 3–15). In each phase a layered-network U is computed by the module `layeredNetwork` (lines 2 and 14), described in Section 6. It returns the characteristic function $U(x, y)$ of U 's edges, respectively the zero-function if no augmenting path exists. At next, a blocking-flow $B(x, y)$ is constructed on $U(x, y)$. As in Hachtel and Somenzi's algorithm, this construction is divided into sweeps (lines 5–12), each resulting in a nonempty set $B^*(x, y)$ of s - t -paths in $U(x, y)$. This is achieved by the combination of two path construction methods:

1. The so-called *single-path construction* finds exactly one path per call; Section 7 introduces it in detail.
2. The *multi-path construction* hopefully yields many paths, while it possibly does not find even one; Section 8 is dedicated to its detailed description.

In each sweep the multi-path method is applied at first (line 6). If this approach does not succeed in constructing any path, the single-path procedure is started (line 8). So it is guaranteed that $B^*(x, y)$ contains at least one s - t -path. This sweep result is added to the blocking-flow candidate $B(x, y)$ (line 10), while it is removed from $U(x, y)$ (line 11).

If t is still reachable from s via the remaining edges of $U(x, y)$, the paths $B(x, y)$ are not a blocking-flow yet and another sweep is started. Else $B(x, y)$ is maximal and the sweep loop is left. In order to decide this in line 12, a reachability analysis on $U(x, y)$ has to be performed. This is done through $O(n^2)$ OBDD-operations and works in the same way as in the module `layeredNetwork`, where the functions $PATh_k^<(x, y)$ are used to decide whether an augmenting path exists (see Section 6.1).

One step remains to finish the actual phase: The augmentation of $F(x, y)$ by $B(x, y)$ in line 13 corresponding to (2). The augmented flow contains edges of two kinds:

1. Edges (x, y) contained in $B(x, y)$ and $E(x, y)$. Because their direction is the same in E and U , they are forward-edges used by the blocking-flow.
2. Edges (x, y) contained in the old flow $F(x, y)$ whose reversed version (y, x) is not contained in $B(x, y)$. If it would be, the blocking-flow would use them backwards, and, therefore, take the flow off it.

The following Theorem 1 represents the central statement on the runtime by means of executed OBDD-operations. Its proof is placed at the end of Section 8 because it refers to results of Sections 6, 7, and 8.

Theorem 1 *The IS-algorithm computes a maximum flow f_{\max} on a network $N = (V, E, s, t)$ through $O(n^2\mathcal{S}) = O(n^2\text{val}(f_{\max})) = O(n^2|V|)$ OBDD-operations, whereby \mathcal{S} is the number of sweep iterations.*

6 Layered-Network Construction

At the beginning of each phase, we have to build a layered-network $U(x, y)$, on which a blocking-flow $B(x, y)$ will be constructed. This is done by the module `layeredNetwork`, which receives four arguments: The input network N in form of $E(x, y)$, $s(x)$, and $t(x)$, as well as the set $F(x, y)$ of actual flow edges. The module returns the layered-network's edge set $U(x, y)$. If no augmenting path according to $F(x, y)$ exists, it returns $U(x, y) := 0$.

In contrast to Hachtel and Somenzi's algorithm, we do not build $U(x, y)$ layer-wise (by breadth-first search) but component-wise. That is, we compute a partition $C_0(x, y), \dots, C_{r-1}(x, y)$, whereby each *partition-component* $C_i(x, y)$ contains not only one but a whole sequence of 2^{k_i} successive edge layers. The exponents k_i are strictly decreasing w. r. t. i .

The edge layers of C_{i-1} and C_i are neighbored in U ; C_{i-1} 's last one and C_i 's first one are adjacent: They share a common node layer, which we call the *separation-layer* Z_i . All shortest residual s - Z_i -paths have length $\sum_{j=0}^{i-1} 2^{k_j}$. Therefore, the depth of U is $\ell = \sum_{j=0}^{r-1} 2^{k_j}$. Separation-layer 0 just contains the source s ($Z_0(x) := s(x)$). In addition, we define $Z_r(x) := t(x)$.

6.1 Preprocessing

At first, the edges $A(x, y)$ of the residual network have to be computed corresponding to (1):

$$A(x, y) := (E(x, y) \wedge \overline{F(x, y)}) \vee F(y, x).$$

Then, some supporting functions are determined by iterative squaring before the partition-components and separation-layers can be considered. They represent different kinds of paths in A , whose lengths will be doubled iteratively. Because these lengths are limited by $|V| - 1$, a number of $O(\log |V|) = O(n)$ iterations suffices.

We now consider the functions $PATH_k^{\leq}(x, y)$ which represent pairs (x, y) of nodes connected by a residual path not longer than 2^k . Path lengths 0 and 1 correspond to pairs (x, x) resp. $(x, y) \in A$. When $PATH_k^{\leq}(x, y)$ is known, $PATH_{k+1}^{\leq}(x, y)$ can be computed by recursion. Node pair (x, y) is connected by a residual path not longer than 2^{k+1} if and only if there is an intermediate node z such that there are x - z - and z - y -paths each not longer than 2^k :

$$\begin{aligned} PATH_0^{\leq}(x, y) &:= (x = y) \vee A(x, y), \\ PATH_{k+1}^{\leq}(x, y) &:= (\exists z) (PATH_k^{\leq}(x, z) \wedge PATH_k^{\leq}(z, y)). \end{aligned}$$

Analogously, the functions $PATH_k^{\bar{=}}(x, y)$ of node pairs (x, y) with a residual x - y -path of length exactly 2^k are obtained. Only the initialization has to be

modified, which now allows only paths of length 1 (edges):

$$\begin{aligned} PATH_0^=(x, y) &:= A(x, y), \\ PATH_{k+1}^=(x, y) &:= (\exists z) (PATH_k^=(x, z) \wedge PATH_k^=(z, y)). \end{aligned}$$

One further tool is needed: The functions $PATH_k^<(x, y)$ of residual x - y -paths shorter than 2^k . For $k = 0$ only the path length 0 is allowed ($x = y$). Considering the recursion, $PATH_{k+1}^<(x, y)$ holds if and only if for some intermediate node z there is an x - z -path not longer 2^k and a z - y -path shorter than 2^k :

$$\begin{aligned} PATH_0^<(x, y) &:= (x = y), \\ PATH_{k+1}^<(x, y) &:= (\exists z) (PATH_k^<(x, z) \wedge PATH_k^<(z, y)). \end{aligned}$$

Finally, let $SPATH_k(x, y)$ (standing for ‘‘shortest path’’) be the characteristic function of all node pairs (x, y) whose shortest residual x - y -path (if existing) has length exactly 2^k . For $k = 0$ this corresponds to the edge relation $A(x, y)$. Otherwise, $SPATH_k(x, y)$ holds if and only if there is an x - y -path not longer than 2^k , but no one which is shorter:

$$\begin{aligned} SPATH_0(x, y) &:= A(x, y), \\ SPATH_k(x, y) &:= PATH_k^=(x, y) \wedge \overline{PATH_k^<(x, y)}. \end{aligned}$$

Now, the node pairs connected by shortest residual paths of lengths 2^k are known and we also need to know their edges. These become represented by $A_k(v, w, x, y)$, which is satisfied for edges $(x, y) \in A$ on a shortest v - w -path with 2^k edges.

All edges $(x, y) \in A$ can be considered as a member of a residual x - y -path of length 1. So $A_0(v, w, x, y)$ is true for $(x, y) \in A$ with $v = x$ and $w = y$. $A_{k+1}(v, w, x, y)$ holds if and only if we can split the v - w -path into parts (v, \dots, u) and (u, \dots, w) , each consisting of 2^k edges, whereby (x, y) is contained in one of both:

$$\begin{aligned} A_0(v, w, x, y) &:= A(x, y) \wedge (v = x) \wedge (w = y), \\ A_{k+1}(v, w, x, y) &:= (\exists u) \left((A_k(v, u, x, y) \wedge SPATH_k(u, w)) \right. \\ &\quad \left. \vee (SPATH_k(v, u) \wedge A_k(u, w, x, y)) \right). \end{aligned}$$

The functions $PATH_k^<(x, y)$, $PATH_k^=(x, y)$, $PATH_k^<(x, y)$, $SPATH_k(x, y)$, and $A_k(v, w, x, y)$ are now computed iteratively with increasing k until for some k^* one of the following conditions holds:

- $PATH_{k^*}^<(s, t) = 1$: Then, $k^* = \lceil \log \ell \rceil$ and $\ell \in]2^{k^*-1}, 2^{k^*}]$. We proceed with partitioning the layered-network U by a kind of binary search (see Section 6.2).

- $PATH_{k^*+1}^{\leq}(x, y) = PATH_{k^*}^{\leq}(x, y)$: No further residual paths have been added, in particular no augmenting path. The actual $F(x, y)$ is already a maximum flow, so `layeredNetwork` returns $U(x, y) := 0$.

The considered path lengths 2^k are doubled in each iteration, and do not exceed $|V|$, so it follows $k^* \leq \lceil \log |V| \rceil = n$.

Lemma 1 *The computation of the functions $PATH_{k^*}^{\leq}(x, y)$, $PATH_{k^*}^{\bar{=}}(x, y)$, $PATH_{k^*}^{<}(x, y)$, $SPATH_{k^*}(x, y)$, and $A_{k^*}(v, w, x, y)$ takes $O(n^2)$ OBDD-operations.*

Proof. Both the initialization and the recursion equation of all of the five functions contain a constant number of binary syntheses and quantifications. Each of the latter is accomplished through $O(n)$ syntheses itself. Due to $k^* \leq n$, no more than $O(n)$ functions are computed. Altogether, $O(n^2)$ OBDD-operations are executed. \square

6.2 Partitioning the layered-network

The layered-network depth ℓ is temporarily considered as a binary number $\ell_{k^*} \dots \ell_0$. Each of the r set bits $\ell_j = 1$ corresponds to one partition-component i with $k_i = j$. That is, $U(x, y)$ is composed of partition-components $C_i(x, y)$ in the same way ℓ is composed of addends 2^{k_i} .

A sort of recursive binary search for the terminal t decides for every bit ℓ_j of ℓ whether it is set, i. e., whether a partition-component with 2^j layers has to be generated. The corresponding Algorithm 3, `findTerminal`, is supplied with the node sets $R(x)$ and $Z(x) \subseteq R(x)$ as well as with the exponent a . It returns a pair $(\mathcal{Z}, \mathcal{K})$ consisting of an OBDD list \mathcal{Z} and an integer list \mathcal{K} .

The *concatenation* of two ordered lists $\mathcal{L}^1 := (e_1^1, \dots, e_n^1)$ and $\mathcal{L}^2 := (e_1^2, \dots, e_m^2)$ will be denoted by $\mathcal{L}^1 : \mathcal{L}^2 := (e_1^1, \dots, e_n^1, e_1^2, \dots, e_m^2)$. Lists (e) containing only one element are just denoted by e .

When calling `findTerminal`($R(x), Z(x), a$) in general, the following information is already known:

- the bits $k^*, \dots, a + 1$ of ℓ (let it contain i set bits),
- the separation-layers $Z_0(x), \dots, Z_i(x) = Z(x)$,
- the exponents k_0, \dots, k_{i-1} , and
- the set $R(x)$ of nodes covered by C_0, \dots, C_{i-1} .

Then, the actual instance of `findTerminal` computes the following information:

- the bits $a, \dots, 0$ of ℓ ,

Algorithm 3 The recursive algorithm $\text{findTerminal}(R(x), Z(x), a)$.

```

1: if ( $SPATH_a(x, t) \wedge Z(x) \neq 0$ ) then {Case 1}
2:   return ( $t(x), a$ );
3: else if ( $PATH_{a-1}^{\leq}(x, t) \wedge Z(x) \neq 0$ ) then {Case 2}
4:   return  $\text{findTerminal}(R(x), Z(x), a - 1)$ ;
5: else {Case 3}
6:    $Z'(x) := (\exists y) (SPATH_{a-1}(y, x) \wedge Z(y)) \wedge \overline{R(x)}$ ;
7:    $R'(x) := R(x) \vee (\exists y) (PATH_{a-1}^{\leq}(y, x) \wedge Z(y))$ ;
8:    $(Z', \mathcal{K}') := \text{findTerminal}(R'(x), Z'(x), a - 1)$ ;
9:   return ( $Z'(x) : \mathcal{Z}', a - 1 : \mathcal{K}'$ );
10: end if

```

- the separation-layers $Z_{i+1}(x), \dots, Z_r(x)$, and
- the exponents k_i, \dots, k_{r-1} .

The resulting list \mathcal{Z} contains the remaining separation-layers $Z_{i+1}(x), \dots, Z_r(x) = t(x)$. Analogously, \mathcal{K} contains the exponents k_i, \dots, k_{r-1} of the remaining partition-components $i, \dots, r - 1$. They correspond to the indices of set bits of ℓ .

For $\ell' := \ell_a \dots \ell_0$ it is $|\ell'| \leq 2^a$, so the part of U that was not yet partitioned is not deeper than 2^a . We now describe the three cases which are distinguished within Algorithm 3.

1. ($SPATH_a(x, t) \wedge Z(x) \neq 0$): The depth ℓ' of the remaining layered-network equals 2^a .

One further partition-component $i = r - 1$ with $k_i = a$ is needed. This exponent together with the final separation-layer $Z_r(x) = t(x)$ is returned (line 2).

Speaking in terms of a binary search, ℓ' has been found on the right border of the search space $]0, 2^a]$. It holds $\ell_a = 1$ and $\ell_j = 0$ for $j < a$.

2. ($PATH_{a-1}^{\leq}(x, t) \wedge Z(x) \neq 0$): There is a residual Z - t -path not longer than 2^{a-1} .

Speaking in terms of a binary search, we continue in the left half $]0, 2^{a-1}]$ of the search space $]0, 2^a]$. This is achieved by the recursive call $\text{findTerminal}(R(x), Z(x), a - 1)$ in line 4.

Viewed as the computation of the bits of ℓ , we know that no partition-component of length 2^a is needed to reach t . Therefore, it is $\ell_a = 0$; the recursion investigates ℓ_{a-1} .

3. None of the two cases above applies: $\ell' \in]2^{a-1}, 2^a[$.

No partition-component of length 2^a is needed to reach t ; on the other hand, none of length 2^{a-1} suffices. Thus, a further partition-component i with $k_i = a - 1$ is necessary. The nodes $Z_{i+1}(x) =: Z'(x)$ (line 6), which are reached by C_i 's last edge layer, represent the starting point for the next search in $]2^{a-1}, 2^a[$. The nodes visited by paths in $C_i(x, y)$ are added to $R(x)$; the result is called $R'(x)$ (line 7).

We know that $\ell_a = 0$ and $\ell_{a-1} = 1$. That is, the resulting lists \mathcal{Z} and \mathcal{K} consist of the result of the remaining partitioning (done by $\text{findTerminal}(R'(x), Z'(x), a - 1)$ in line 8) appended to $Z'(x)$ resp. $a - 1$ (line 9). The binary search continues in the right half of the search space.

At the beginning of the partitioning, only $Z_0(x) = R(x) = s(x)$ as well as $k^* + 1$ are known. The binary search starts at source node s , which represents both the first separation-layer $Z_0(x)$ and the set of nodes reached so far. The initial search interval is $]0, 2^{k^*}]$. Hence, Algorithm 3 is called through $\text{findTerminal}(s(x), s(x), k^*)$. The result is the lists $\mathcal{Z} = (Z_1(x), \dots, Z_r(x))$ and $\mathcal{K} = (k_0, \dots, k_{r-1})$. Figure 3 shows the partitioning of an example layered-network of depth $\ell = 13$.

Lemma 2 *The call to $\text{findTerminal}(s(x), s(x), k^*)$ takes $O(n^2)$ OBDD-operations.*

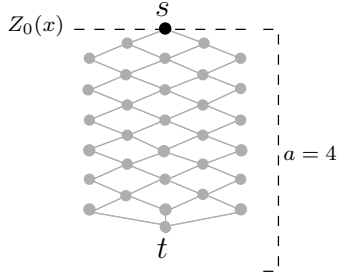
Proof. Besides the recursive calls, Algorithm 3 executes $O(n)$ OBDD-operations. In each recursive call to findTerminal , the parameter a decreases by 1. For $a = 0$ Case 1 applies and no further recursion is entered. Therefore, findTerminal is called no more than $k^* + 1 \leq n + 1$ times. Hence, all calls together take $O(n^2)$ OBDD-operations. \square

6.3 Computing the layered-network edges

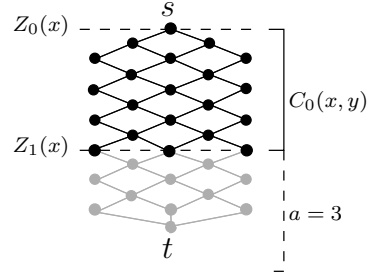
At last, we have to determine the edge sets $C_0(x, y), \dots, C_{r-1}(x, y)$ of all partition components. Remember that $C_i(x, y)$ should consist of shortest residual paths connecting Z_i and Z_{i+1} . An edge (x, y) belongs to such a path if and only if it is part of a shortest residual path (v, \dots, w) with $v \in Z_i$ and $w \in Z_{i+1}$, expressed by the following assignment:

$$C_i(x, y) := (\exists v, w) (A_{k_i}(v, w, x, y) \wedge Z_i(v) \wedge Z_{i+1}(w)).$$

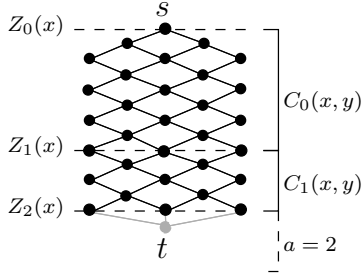
Their union gives us the function $C(x, y) := \bigvee_{i=0}^{r-1} C_i(x, y)$, which includes all layered-network edges, but may also contain other ones. Note that Z_i , $i \in \{1, \dots, r\}$, contains all nodes v whose shortest residual s - v -path has length $\sum_{j=0}^{i-1} 2^{k_j}$. In particular, $Z_r(x)$ may contain nodes besides t , which are also reachable from s via ℓ edges. It follows that $C(x, y)$ may contain edges not belonging to any shortest s - t -path.



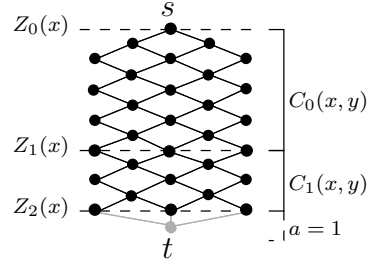
(a) Situation at the beginning of $\text{findTerminal}(s(x), s(x), 4)$.



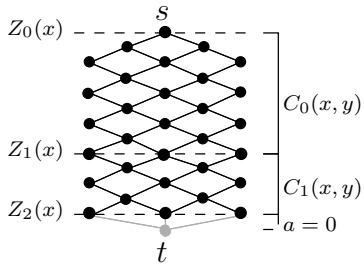
(b) Case 3: Because of $\ell \in]2^3, 2^4[$, partition-component $C_0(x, y)$ with $k_0 = 3$ is generated. The search continues from $Z'(x) = Z_1(x)$ with $a = 3$.



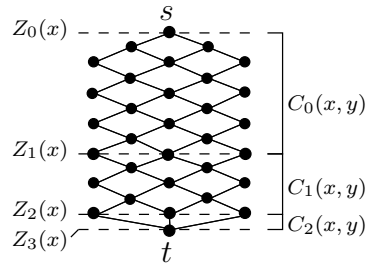
(c) Case 3: Because of $\ell \in]2^3 + 2^2, 2^4[$, partition-component $C_1(x, y)$ with $k_1 = 2$ is generated. The search continues from $Z'(x) = Z_2(x)$ with $a = 2$.



(d) Case 2: The search interval is restricted to $]2^3 + 2^2, 2^3 + 2^2 + 2^1]$.



(e) Case 2: The search interval is restricted to $]2^3 + 2^2, 2^3 + 2^2 + 2^0]$.



(f) Case 1: t has been found in layer $2^3 + 2^2 + 2^0 = 13$.

Figure 3: Partitioning an example layered-network of depth 13. Black nodes/edges have already been partitioned; grey parts are unknown yet. Dashed lines indicate separation-layers. Solid brackets span partition-components, dashed brackets span search intervals.

In order to discard these edges, we determine all nodes $R(x)$ that are covered by $C(x, y)$ and that lie on an s - t -path. It is $v \in R$ if and only if v is reachable from t in the reversed network $C^R(x, y) := C(y, x)$. In order to perform a reachability analysis on $C^R(x, y)$, we proceed like in the preprocessing at the beginning of this section: Functions $PATH_k^{\leq}(x, y)$ initialized by $PATH_0^{\leq}(x, y) := (x = y) \vee C^R(x, y)$ are computed yielding $R(x) := PATH_{k^*}^{\leq}(t, x)$. Only edges covering reachable nodes are taken over in $U(x, y)$:

$$U(x, y) := C(x, y) \wedge R(x).$$

Finally, $U(x, y)$ contains exactly those edges belonging to a shortest augmenting path. We now have all implicitly represented sets we need to compute a blocking-flow $B(x, y)$ on $U(x, y)$. In contrast to Hachtel and Somenzi's algorithm, this has to be done without knowing the assignment of each node and edge to its particular layer.

Lemma 3 *The computation of $U(x, y)$ takes $O(n^2)$ OBDD-operations.*

Proof. Each of the $r \leq n$ functions $C_i(x, y)$, $i \in \{0, \dots, r-1\}$, is computed by $O(n)$ OBDD-operations. The disjunction of all $C_i(x, y)$ corresponds to $r-1$ binary syntheses. Due to Lemma 1, the reachability analysis also takes $O(n^2)$ operations. Altogether, the layered-network edges $U(x, y)$ are computed through $O(n^2)$ OBDD-operations. \square

Corollary 1 *Due to Lemmas 1, 2, and 3, each call to $\text{layeredNetwork}(U(x, y))$ executes $O(n^2)$ OBDD-operations.* \square

7 Single-Path Construction

The module `singlePath` constructs one s - t -path p of length ℓ in $U(x, y)$, represented by $B^*(x, y)$.

The depth ℓ is not necessarily a power of 2. In order to use iterative squaring, ℓ has to be partitioned into a sequence $\ell_0^1 := \ell, \ell_0^2 := \ell - 1, \dots, \ell_q^1 := 1, \ell_q^2 := 0$. Each ℓ_m^a , $m \in \{0, \dots, q\}$, $a \in \{1, 2\}$, is considered as a path length. Moreover, paths of length ℓ_m^1 and ℓ_m^2 should be composable by two subpaths of length ℓ_{m+1}^1 and/or ℓ_{m+1}^2 . This is achieved by the subdivision rules $\ell_{m+1}^1 := \lceil \ell_m^1 / 2 \rceil$ and $\ell_{m+1}^2 := \lfloor \ell_m^2 / 2 \rfloor = \ell_{m+1}^1 - 1$. For $e, o \in \{1, 2\}$ such that ℓ_m^e is even and ℓ_m^o is odd, it is $\ell_m^e = 2\ell_{m+1}^e$ and $\ell_m^o = \ell_{m+1}^e + \ell_{m+1}^o$. We stop the subdivision for q with $\ell_q^1 = 1$, $\ell_q^2 = 0$. From $\ell_{m+1}^a \leq (\ell_m^a + 1) / 2$, $a \in \{1, 2\}$, it follows that $q = O(\log \ell) = O(n)$.

As preparation of the single-path construction, the functions $P_{\ell_m^a}(x, y, z)$ are computed for $m \in \{q-1, \dots, 0\}$, $a \in \{1, 2\}$. They represent paths (x, \dots, y, \dots, z) of length ℓ_m^a such that the subdivision into (x, \dots, y) and

(y, \dots, z) happens according to the appropriate subdivision of ℓ_m^a . The composition is supported by functions $P_{\ell_m^a}^*(x, z)$ representing layered-network paths (x, \dots, z) of length ℓ_m^a .

Again, let $e, o \in \{1, 2\}$ be chosen such that ℓ_m^e is even and ℓ_m^o is odd. We iterate the computation for $m \in \{q-1, \dots, 0\}$, $a \in \{1, 2\}$, based on an initialization for $\ell_q^1 = 1$ and $\ell_q^2 = 0$:

$$\begin{aligned} P_{\ell_q^1}^*(x, z) &:= U(x, z), \\ P_{\ell_q^2}^*(x, z) &:= (x = z), \\ P_{\ell_m^e}^*(x, y, z) &:= P_{\ell_{m+1}^e}^*(x, y) \wedge P_{\ell_{m+1}^e}^*(y, z), \\ P_{\ell_m^o}^*(x, y, z) &:= P_{\ell_{m+1}^e}^*(x, y) \wedge P_{\ell_{m+1}^o}^*(y, z), \\ P_{\ell_m^a}^*(x, z) &:= (\exists y) P_{\ell_m^a}^*(x, y, z). \end{aligned}$$

For paths (x, \dots, y, \dots, z) of even length ℓ_m^e the node y lies exactly in the middle; for paths with odd length ℓ_m^o the subpath (x, \dots, y) has even length ℓ_{m+1}^e , while (y, \dots, z) has odd length ℓ_{m+1}^o . Due to $\ell_0^1 = \ell$, we represent the pair (s, t) of p 's start- and end-node by $D_{\ell_0^1}(x, z) := s(x) \wedge t(z)$. The function $D_{\ell_0^2}(x, z)$ is not needed for the following subdivision of p , why it is defined as the zero-function.

The path construction itself works in a top-down manner by concretizing longer paths by shorter ones. Through choosing an intermediate node y which lies on an s - t -path, we divide p into two parts (s, \dots, y) and (y, \dots, t) , represented by $D_{\ell_1^1}(x, z)$ and/or $D_{\ell_1^2}(x, z)$. This subdivision is iterated for the resulting paths until we know the edges D_1 of p . Each iteration performs the subdivision step in parallel for subpaths of same length.

The general situation is that we know start- and end-nodes of subpaths of length ℓ_m^1 and ℓ_m^2 , represented by $D_{\ell_m^1}(x, z)$ and $D_{\ell_m^2}(x, z)$. We obtain $Q_{\ell_m^a}(x, y, z)$ by deleting all paths (x, \dots, y, \dots, z) from $P_{\ell_m^a}^*$, $a \in \{1, 2\}$, whose start-end-node pairs are not included in $D_{\ell_m^a}$:

$$Q_{\ell_m^a}(x, y, z) := P_{\ell_m^a}^*(x, y, z) \wedge D_{\ell_m^a}(x, z).$$

$Q_{\ell_m^a}(x, y, z)$ now represents all possible subpaths of length ℓ_m^a which visit the nodes already fixed by $D_{\ell_m^a}(x, z)$. One start-/end-node pair (x, z) may occur in several triples $(x, y, z) \in Q_{\ell_m^a}$, each representing a possible subpath of p . Through selecting one of the intermediate nodes y , one of these subpaths gets fixed to be part of p . The selection uses a priority function Π as introduced in Section 3.

However, these priority functions work with just one selecting node x that determines the order $<_x$. We now want both the start-node x as well as the end-node z of each path of $Q_{\ell_m^a}$ to influence the selection of the middle-node y . Therefore, as first argument of Π the *blending* $x \odot z$ of x and z is used, which we define by

$$(x \odot z)_i := \begin{cases} x_i, & i \text{ even} \\ z_i, & i \text{ odd} \end{cases}.$$

The function $T_{\ell_m^a}(x, y, z)$ is the subset of $Q_{\ell_m^a}(x, y, z)$ obtained by selecting an intermediate node y for each subpath (x, \dots, z) :

$$T_{\ell_m^a}(x, y, z) := Q_{\ell_m^a}(x, y, z) \wedge (\exists y') \overline{(Q_{\ell_m^a}(x, y', z) \wedge \Pi(x \circledast z, y', y))}.$$

Before entering the next iteration for path lengths ℓ_{m+1}^a we compute $D_{\ell_{m+1}^a}(x, z)$, $a \in \{1, 2\}$, which contains the upper parts (x, \dots, y) as well as the lower parts (y, \dots, z) obtained by the subdivision at y . Again, let ℓ_m^e be even and ℓ_m^o odd:

$$\begin{aligned} D_{\ell_{m+1}^e}(x, z) &:= (\exists y) (T_{\ell_m^e}(x, z, y) \vee T_{\ell_m^e}(y, x, z) \vee T_{\ell_m^e}(x, z, y)), \\ D_{\ell_{m+1}^o}(x, z) &:= (\exists y) T_{\ell_m^o}(y, x, z). \end{aligned}$$

The subpaths (x, \dots, z) of length ℓ_{m+1}^e , represented by $D_{\ell_{m+1}^e}(x, z)$, contribute as upper parts to paths (x, \dots, z, \dots, y) both of even length ℓ_m^e ($T_{\ell_m^e}(x, z, y)$) and of odd length ℓ_{m+1}^o ($T_{\ell_m^o}(x, z, y)$). As lower parts, they contribute only to paths (y, \dots, x, \dots, z) of even length ℓ_m^e ($T_{\ell_m^e}(y, x, z)$). The subpaths (x, \dots, z) of length ℓ_{m+1}^o , represented by $D_{\ell_{m+1}^o}(x, z)$, contribute as lower parts to paths (y, \dots, x, \dots, z) of odd length ℓ_m^o ($T_{\ell_m^o}(y, x, z)$).

Due to $\ell_q^1 = 1$, the subpaths of $D_{\ell_q^1}(x, z)$ represent p 's edges, and make the construction result $B^*(x, y) := D_{\ell_q^1}(x, z)$. Figure 4 shows an example of a single-path construction in a layered-network of depth $\ell = 8$.

Lemma 4 *Each call to $\text{singlePath}(U(x, y))$ executes $O(n^2)$ OBDD-operations.*

Proof. The functions $P_{\ell_m^a}(x, y, z)$, $P_{\ell_m^a}^*(x, z)$, $Q_{\ell_m^a}(x, y, z)$, $T_{\ell_m^a}(x, y, z)$, and $D_{\ell_m^a}(x, z)$ are computed for $m \in \{0, \dots, q\}$ and $a \in \{1, 2\}$. Each function computation takes $O(n)$ OBDD-operations. Due to $q = O(\log \ell) = O(n)$, an over-all number of $O(n^2)$ operations is executed. \square

8 Multi-Path Construction

The module multiPath constructs a set of edge-disjoint s - t -paths in $U(x, y)$ represented by $B^*(x, y)$. It composes longer paths of length ℓ_m^a from shorter ones of length ℓ_{m+1}^1 and/or ℓ_{m+1}^2 for $m \in \{q-1, \dots, 0\}$ and $a \in \{1, 2\}$. This construction is performed for all $\ell - \ell_m^a + 1$ layer sequences of the currently considered path length ℓ_m^a in parallel, but independent from each other.

In general, we know constructed paths of length ℓ_{m+1}^a , $a \in \{1, 2\}$. The pairs $((v, w), (y, z))$ of their start- and end-edges are represented by $D_{\ell_{m+1}^a}(v, w, y, z)$. By $S_{\ell_{m+1}^a}(u, v, w, x, y, z)$ we will represent the edges (u, v) lying on these paths (w, x, \dots, y, z) . Initially, all edges of $U(x, y)$ are considered as paths of length $1 = \ell_q^1 = \ell_{q-1}^2$:

$$\begin{aligned} D_1(v, w, y, z) &:= U(v, w) \wedge (v = y) \wedge (w = z), \\ S_1(u, v, w, x, y, z) &:= D_1(w, x, y, z) \wedge (u = w) \wedge (v = x). \end{aligned}$$

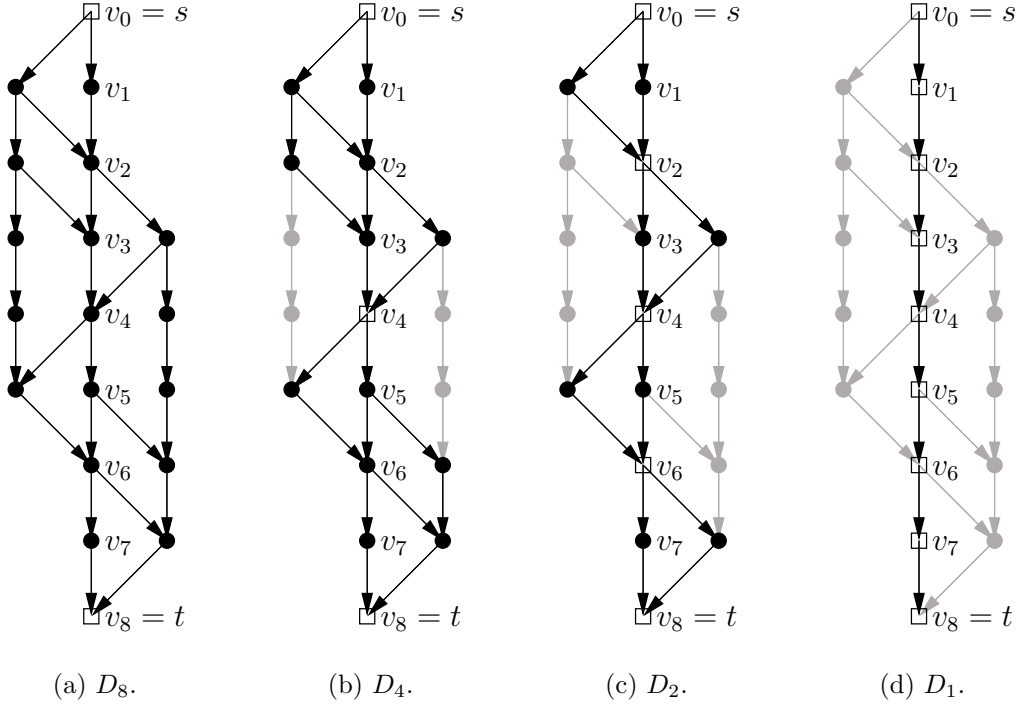


Figure 4: Example of a single-path construction of $p = (s = v_0, \dots, v_8 = t)$. Boxes symbolize fixed path-nodes contained in a function $D_{\ell_n^a}$. Grey nodes/edges symbolize parts of U which are already excluded from the path construction. D_8 contains the start-/end-node pair (v_0, v_8) . All nodes/edges may still become part of p . D_4 contains the pairs (v_0, v_4) and (v_4, v_8) ; the middle-node v_4 has been chosen. D_2 contains (v_0, v_2) , (v_2, v_4) , (v_4, v_6) , and (v_6, v_8) ; the middle-nodes v_2 and v_6 have been chosen. After the last path subdivision of D_1 , all nodes of p are selected and the path construction is finished.

Paths of length $\ell_q^2 = 0$ are not needed in the following, why D_0 and S_0 stay undefined.

We use $D_{\ell_{m+1}^a}$ to compute possible start-parts (v, w, \dots, x) (represented by $P_{\ell_m^a}^S(v, w, x)$) and end-parts (x, \dots, y, z) (represented by $P_{\ell_m^a}^E(x, y, z)$) of paths of length ℓ_m^a , $a \in \{1, 2\}$. For even lengths ℓ_m^e we allow start- and end-parts both of length ℓ_{m+1}^e :

$$\begin{aligned}
 P_{\ell_m^e}^S(v, w, x) &:= (\exists y) D_{\ell_{m+1}^e}(v, w, y, x), \\
 P_{\ell_m^e}^E(x, y, z) &:= (\exists w) D_{\ell_{m+1}^e}(x, w, y, z).
 \end{aligned}$$

For odd lengths ℓ_m^o we allow start-parts with ℓ_{m+1}^e edges and end-parts with ℓ_{m+1}^o edges:

$$\begin{aligned} P_{\ell_m^o}^S(v, w, x) &:= (\exists y) D_{\ell_{m+1}^e}(v, w, y, x), \\ P_{\ell_m^o}^E(x, y, z) &:= (\exists w) D_{\ell_{m+1}^o}(x, w, y, z). \end{aligned}$$

By disjunction and quantification we obtain candidates for start–end-edges pairs of paths of length ℓ_m^a :

$$\begin{aligned} P_{\ell_m^a}(v, w, x, y, z) &:= P_{\ell_m^a}^S(v, w, x) \wedge P_{\ell_m^a}^E(x, y, z), \\ P_{\ell_m^a}^*(v, w, y, z) &:= (\exists x) P_{\ell_m^a}(v, w, x, y, z). \end{aligned}$$

In order to construct the set $D_{\ell_m^a}(v, w, y, z)$ of edge-disjoint paths of length ℓ_m^a , we uniquely assign start-edges (v, w) and end-edges (y, z) with an existing path (v, w, \dots, y, z) to each other. We want edges to choose other ones, so priority functions $\Pi(u, v, w, x, y, z)$ (see Section 4) are used which receive node pairs (u, v) , (w, x) and (y, z) instead of single nodes as arguments:

$$\Pi(u, v, w, x, y, z) \Leftrightarrow (x, y) <_{(u,v)} (y, z).$$

In a first step, we let possible start-edges choose according end-edges, and denote the achieved subset of $P_{\ell_m^a}^*(v, w, y, z)$ by $Q_{\ell_m^a}(v, w, y, z)$:

$$\begin{aligned} Q_{\ell_m^a}(v, w, y, z) &:= P_{\ell_m^a}^*(v, w, y, z) \\ &\quad \wedge \overline{(\exists y', z') (P_{\ell_m^a}^*(v, w, y', z') \wedge \Pi(v, w, y', z', y, z))}. \end{aligned}$$

In a second step, every end-edge that has been chosen by a start-edge in the first step selects one of these, and completes the unique assignment $D_{\ell_m^a}(v, w, y, z)$:

$$\begin{aligned} D_{\ell_m^a}(v, w, y, z) &:= Q_{\ell_m^a}(v, w, y, z) \\ &\quad \wedge \overline{(\exists v', w') (Q_{\ell_m^a}(v', w', y, z) \wedge \Pi(y, z, v', w', v, w))}. \end{aligned}$$

The constructed path set has not to be maximal, so the assignment procedure may be iterated until no more paths of length ℓ_m^a can be constructed. In the following runtime considerations, a constant number of iterations is assumed.

Finally, we have to compute the edge set $S_{\ell_m^a}(u, v, w, x, y, z)$ of the newly composed paths:

$$\begin{aligned} S_{\ell_m^e}(u, v, w, x, y, z) &:= D_{\ell_m^e}(w, x, y, z) \\ &\quad \wedge (\exists w', x') (S_{\ell_{m+1}^e}(u, v, w, x, w', x') \\ &\quad \quad \vee S_{\ell_{m+1}^e}(u, v, w', x', y, z)), \\ S_{\ell_m^o}(u, v, w, x, y, z) &:= D_{\ell_m^o}(w, x, y, z) \\ &\quad \wedge (\exists w', x') (S_{\ell_{m+1}^o}(u, v, w, x, w', x') \\ &\quad \quad \vee S_{\ell_{m+1}^o}(u, v, w', x', y, z)). \end{aligned}$$

An edge (u, v) lies on a path (w, x, \dots, y, z) of length ℓ_m^a if and only if such a path exists (expressed by $D_{\ell_{m+1}^a}(w, x, y, z)$, $a \in \{e, o\}$) and the edge lies on one of the parts this path was composed of. Again, the lengths of the composing parts depend on the evenness or oddness of ℓ_m^a , for even ℓ_m^e and odd ℓ_m^o .

When for $\ell_0^1 = \ell$ the function $S_{\ell_0^1}(u, v, w, x, y, z)$ is known, we get the path edges $B^*(x, y)$ by quantification over possible start- and end-edges:

$$B^*(x, y) := (\exists u, v, w, z) S_{\ell_0^1}(x, y, u, v, w, z).$$

Because all subpaths of a certain length are constructed independent from each other, this procedure may lead into a dead end. Then, Algorithm 2 applies the single-path construction (line 8) to obtain at least one augmenting path. Figure 5 shows an example of an unsuccessful multi-path construction in a layered-network of depth $\ell = 8$.

Lemma 5 *One call to $\text{multiPath}(U(x, y))$ executes $O(n^2)$ OBDD-operations.*

Proof. Due to the assumption of a constant number of selection iterations for each path length ℓ_m^a , $m \in \{0, \dots, q\}$, $a \in \{1, 2\}$, one can argue analogously to Lemma 4. \square

Corollary 1 as well as Lemmas 4 and 5 are now used to prove Theorem 1:

Proof of Theorem 1. Without loss of generality, we assume that a new layered-network is computed after each sweep. Due to Corollary 1, each of these $O(\mathcal{S})$ layered-network computations consumes $O(n^2)$ operations. Due to Lemmas 4 and 5, both the calls to $\text{multiPath}(U(x, y))$ and $\text{singlePath}(U(x, y))$ take $O(n^2)$ operations each. The remaining steps of removing $B^*(x, y)$ from $U(x, y)$ and of augmenting $F(x, y)$ by $B(x, y)$ can be done by a constant number of operations. Altogether, each sweep executes $O(n^2)$ OBDD-operations, and increases F 's value at least by 1. \square

9 Grid Networks

In the following, we will consider the the IS-algorithm on quadratic $(2^k + 1) \times (2^k + 1)$ -grid networks $N = (V, E, s, t)$. These are $(2^k + 1) \times (2^k + 1)$ -matrices of nodes which contain edges between horizontally as well as between vertically adjacent nodes. Edges are directed towards the higher row resp. column index. $(0, 0) =: s$ serves as the source, while $(2^k, 2^k) =: t$ serves as the terminal. Independent from k , the maximum flow value of these grids is 2 and the algorithm executes two sweeps at most. In fact, it can be shown that only one sweep is necessary. Figure 6 shows such a network for $k = 2$.

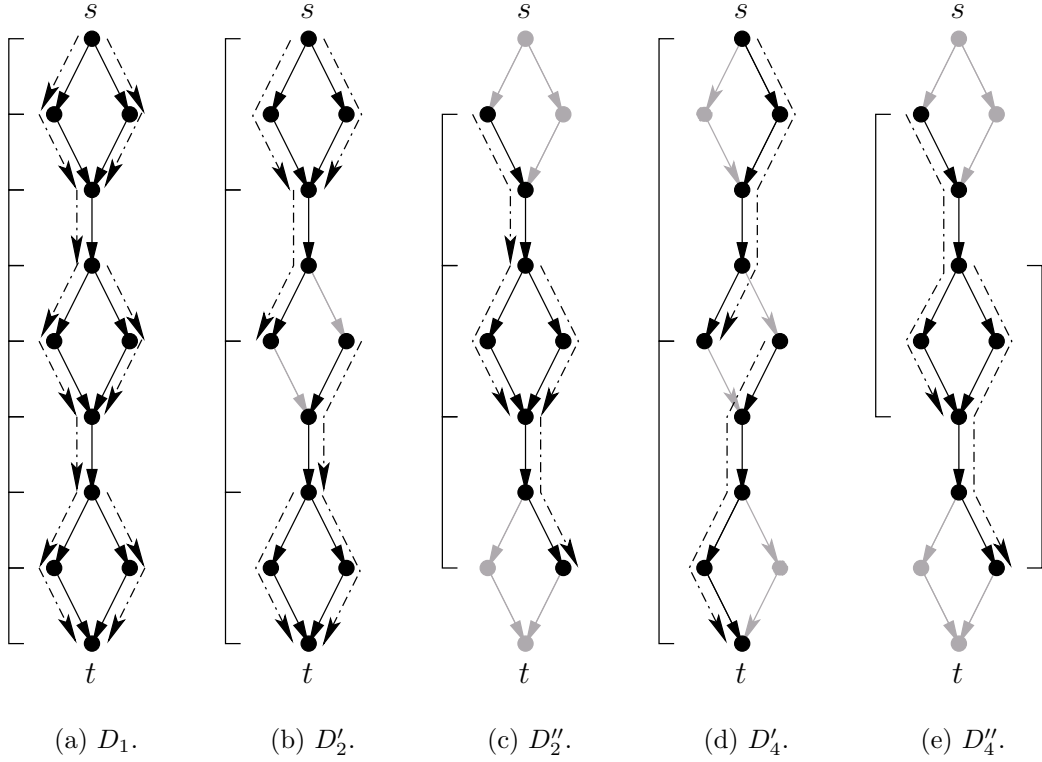


Figure 5: Example of a multi-path construction. The brackets indicate the considered path lengths. Dashed arrows symbolize constructed paths. Grey nodes/edges are not used by any of the particular subfigure's paths. D_1 contains just the edges of U . D_2 consists of the paths D_2' and D_2'' of Subfigures (b) and (c). D_4 consists of the paths D_4' and D_4'' of Subfigures (d) and (e). Only those of Subfigure (d) cover s and t and could be composed to paths of length 8. Anyhow, this is not possible, because they do not share any connecting node. Hence, no s - t -path is constructed.

9.1 Analysis

Theorem 2 represents the main result, whose proof will only be sketched in this paper. For a detailed analysis, we refer the reader to [15].

Theorem 2 *Using the datum proximity Π^{dp} as priority function, the IS-algorithm computes a maximum flow on $(2^k + 1) \times (2^k + 1)$ -grid networks in time $O(k^3)$ and space $O(k^2)$.*

Note that k is logarithmic in the number of nodes $|V| = 2^{2k} + 2^{k+1} + 1$. We assume that the binary number x of a grid node $v_{|x|} \in V$ consists of a row part x^r and a column part x^c of $k + 1$ Boolean variables each. Therefore, $n := 2(k + 1)$ bits are used to encode one node. Due to Theorem 1, $O(\log^2 |V|) = O(k^2)$ OBDD-

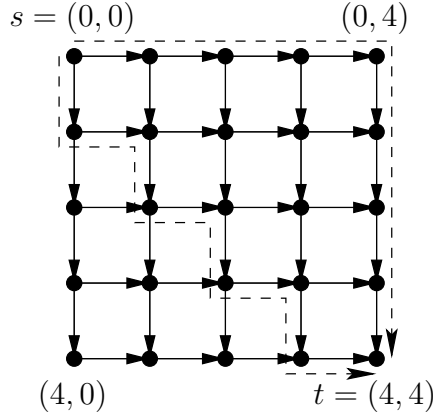


Figure 6: The 5×5 -grid network. Dashed arrows indicate the constructed maximum flow.

operations are executed. The time consumed besides these operations is of the same magnitude and therefore negligible.

Definition 1 *Let F be an OBDD defined on n variables.*

1. *F is called complete if every path from F 's function pointer to a sink has length n . Therefore, F may be not minimal.*
2. *The width of F refers to the maximum numbers of nodes labeled with the same variable.*

The idea of the analysis is to prove for each function $f \in B_{cn}$ (defined on $c = O(1)$ node encodings) occurring during the flow maximization that it is representable by a complete OBDD \mathcal{F} of constant width $k = O(1)$. That is, every variable labels k nodes at most. Because \mathcal{F} depends on $O(k)$ Boolean variables, it is $|\mathcal{F}| = O(k)$. The algorithm represents f by a minimized (possibly non-complete) OBDD F with $|F| \leq |\mathcal{F}|$. It follows directly that each OBDD-operation (and, in particular, each binary synthesis) takes time $O(k^2)$. Moreover, it can be shown that each operation on such OBDDs needs only linear time and space $O(k)$ [15].

Hence, the OBDD-operations take time $O(k^3)$. Because no more than $O(k)$ functions are represented at the same time during the algorithm, the space usage is $O(k^2)$. It remains to show the constant width property for all participating OBDDs.

Woelfel [19] has introduced the class of *k -variate threshold functions*.

Definition 2 *A function $f \in B_{kn}$, $k, n \in \mathbb{N}$, defined on the variable set $X_{k,n} := \{x_j^i \mid 1 \leq i \leq k, 0 \leq j < n\}$ is called k -variate threshold function if there are*

weights $w_1, \dots, w_k \in \mathbb{Z}$ and a threshold $T \in \mathbb{Z}$ such that

$$f(x^1, \dots, x^k) = \left(\sum_{i=1}^k w_i \cdot |x^i| \geq T \right).$$

The maximum absolute weight of f is defined as $w(f) := \max\{|w_1|, \dots, |w_k|\}$.

k -variate threshold functions f with constant $k = O(1)$ and $w(f) = O(1)$ have complete $\pi_{k,n}$ -OBDDs \mathcal{F} of constant width, where $\pi_{k,n}$ is the interleaved variable ordering (see Section 3). The same holds for formulas over a constant number of such threshold functions. It is easy to see that the comparisons $=$, $>$, \leq , and $<$ can be expressed by such a formula. For example, $x = y$ corresponds to

$$[1 \cdot x + (-1) \cdot y \geq 0] \wedge [(-1) \cdot x + 1 \cdot y \geq 0].$$

Due to some convenient properties of complete OBDDs of constant width, it suffices to consider only some selected OBDDs occurring during the flow maximization and to show that they represent constant size formulas over such threshold functions. We will show this property exemplary for the input edge set $E(x, y)$ as well as for one function in each part of the algorithm.

Input edge set: $E(x, y)$ can be written as composition of seven threshold functions:

$$\begin{aligned} E(x, y) &= (y^r - x^r + y^c - x^c = 1) \\ &\quad \wedge (y^r - x^r \geq 0) \wedge (y^c - x^c \geq 0) \\ &\quad \wedge (x^r \leq 2^k) \wedge (x^c \leq 2^k) \wedge (y^r \leq 2^k) \wedge (y^c \leq 2^k). \end{aligned}$$

Only those nodes lying in the $(2^k + 1) \times (2^k + 1)$ -square are touched by edges. Moreover, there is an edge between node x and y if and only if they are either horizontal adjacent or vertical adjacent.

Layered-network construction: It is easy to see that the layered-network U of the first and only phase corresponds to its residual network A , and consists simply of all edges of E . Furthermore, the length of an x - y -path equals $(y^r - x^r) + (y^c - x^c)$. Nodes $x \in V$ whose s - x -paths have length ν lie in node layer ν . These properties lead to simple representations of the different $PATH$ -functions and A_k . For example, $PATH_k^<(x, y)$ can be written as

$$PATH_k^<(x, y) = (y^r - x^r \geq 0) \wedge (y^c - x^c \geq 0) \wedge (y^r - x^r + y^c - x^c < 2^k).$$

Analogously, it can be shown that the functions $Z'(x)$ and $R'(x)$ computed in Case 3 of subalgorithm `findTerminal` as well as the edge layer functions $C_i(x, y)$ can be composed of simple threshold functions.

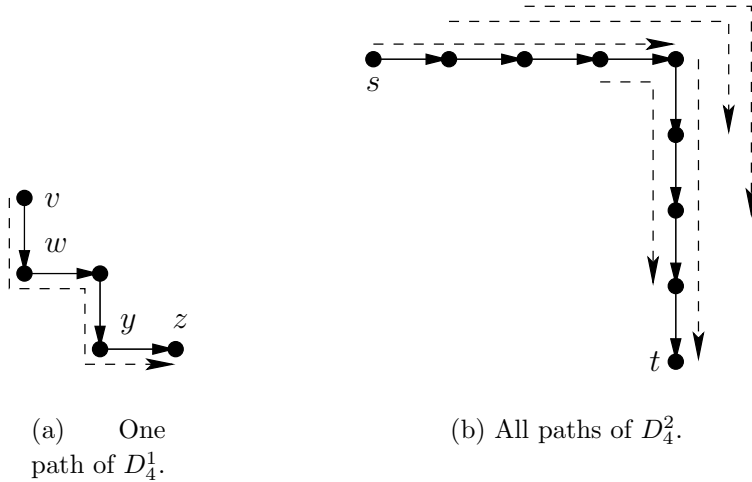


Figure 7: The different path structures of D_4^1 and D_4^2 . Dashed arrows indicate constructed paths.

Blocking flow construction: The layered-network U has depth 2^{k+1} . We assume that the implementation of the flow algorithm considers only path lengths $\ell_m^1 = 2^{k-m+1}$ for $m \in \{0, \dots, k+1\}$, while omitting the unnecessary lengths $\ell_m^2 = \ell_m^1 - 1$. Two augmenting paths are constructed using the multi-path method with one selection iteration per path length 2^m .

In order to prove the constant width property, we basically have to represent the functions D_{2^m} and S_{2^m} , $m \in \{1, \dots, k+1\}$, as formulas over threshold functions. It follows from a technical case distinction for D_2 that D_{2^m} for $m \geq 2$ consists of two subsets of simple paths, which we call $D_{2^m}^1$ and $D_{2^m}^2$. Paths of $D_{2^m}^1$ with start-edge (v, w) and end-edge (y, z) are generated for any pair of a vertical edge (v, w) and a horizontal edge (y, z) , such that v and z lie on the same diagonal and have a distance of 2^m (see Figure 7(a)). $D_{2^m}^2$ simply contains all paths of lengths 2^m that lie on the upper right boundary of the grid (see Figure 7(b)). $D_{2^m}^1$, $D_{2^m}^2$, and, therefore, $D_{2^m} = D_{2^m}^1 \vee D_{2^m}^2$ can be composed of threshold functions. We exemplary show this for $D_{2^m}^1$:

$$\begin{aligned}
 D_{2^m}^1(v, w, y, z) &= E(v, w) \wedge E(y, z) \\
 &\quad \wedge ((v^c = w^c) \wedge (y^r = z^r)) \\
 &\quad \wedge (z^r - v^r = 2^{m-1}) \wedge (z^c - v^c = 2^{m-1}).
 \end{aligned}$$

Analogously, we can represent the corresponding edge sets S_{2^m} by complete OBDDs of constant width. Finally, $S_{2^{k+1}}$ contains both augmenting paths, which make up the maximum flow illustrated in Figure 6.

9.2 Experimental results

Both the IS-algorithm and Hachtel and Somenzi’s maximum flow method [11] have been implemented¹ in C++. In order to confirm the practical relevance of Theorem 2, both algorithms have been applied on $(2^k + 1) \times (2^k + 1)$ -grids for $0 \leq k \leq 16$ on a PC with Pentium 4 2GHz processor and 512 MB of main memory. For $k > 16$, the system’s memory did not suffice to apply Hachtel and Somenzi’s method. In contrast, the IS-algorithm was executed up to $k = 19$, and did not reach the memory limit in the experiments. Figures 8(a) and 8(b) show the experimental results by means of runtime and space usage. Space is measured in the maximum number of nodes contained in all OBDDs at any time. It can be seen that the IS-algorithm beats the space usage of Hachtel and Somenzi’s method for $k \geq 13$, while the runtime is beaten for $k \geq 16$.

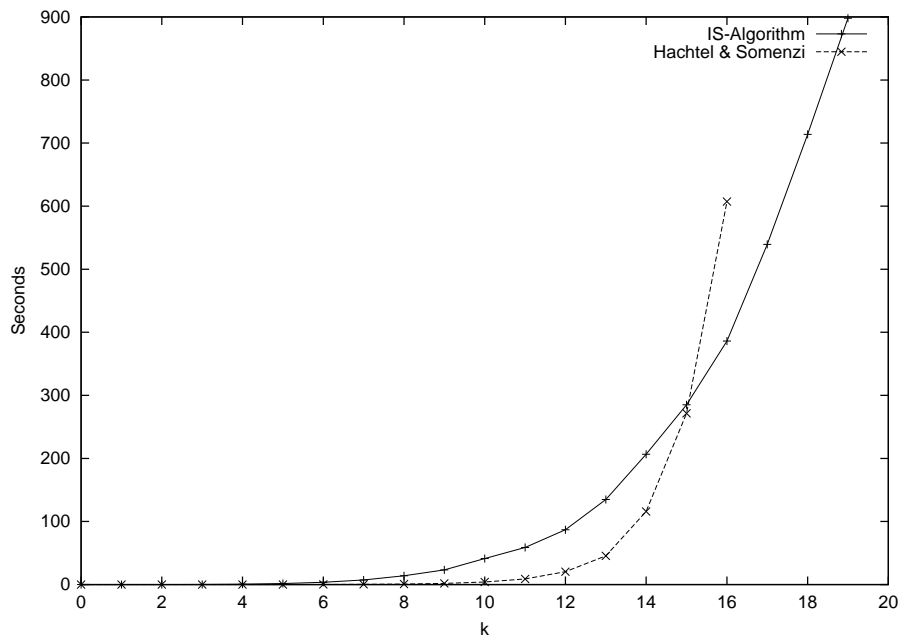
10 Conclusions

An implicit algorithm for flow maximization in 0–1 networks has been presented which uses the technique of iterative squaring. It has been proven that it executes $O(n^2\mathcal{S})$ OBDD-operations, whereby $\mathcal{S} \leq \text{val}(f_{max}) \leq |V|$ is the number of performed sweep iterations and $n := \lceil \log |V| \rceil$. Hence, this runtime result is independent from the maximum depth ℓ of layered-networks. This is the advantage compared to Hachtel and Somenzi’s max-flow algorithm, which executes $\Omega(\ell n)$ OBDD-operations for every input. Advantageously network properties may lead to a constant number of sweeps. This applies especially if the maximum flow value is independent from the network size. Then, only $O(n^2)$ OBDD-operations are necessary to compute a maximum flow.

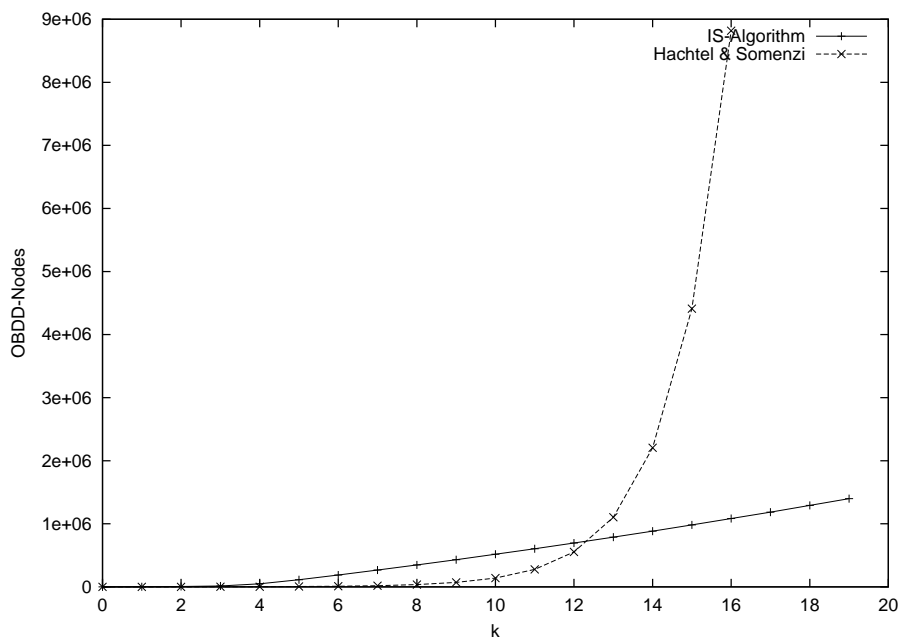
If we focus on special input networks, we are able to analyze also the OBDD-sizes occurring during the IS-algorithm and, thereby, to obtain a bound on the over-all runtime. This has been done for the case of grid networks [15], where all OBDDs have size $O(n)$ and each OBDD-operation takes time $O(n)$. Due to the constant flow value, a number of $O(n^2)$ operations is executed leading to the over-all runtime of $O(n^3)$ and space usage $O(n^2)$. In contrast, Hachtel and Somenzi’s algorithm performs $\Omega(|V|^{1/2} \log |V|)$ operations on grid networks, which is an exponential larger amount of time. In experiments, Hachtel and Somenzi’s method is beaten w. r. t. time and space for $k \geq 16$. These results hint on the potential of this paper’s algorithm to achieve better over-all runtimes on structured networks through using less OBDD-operations.

At the moment, there exist only implicit flow maximization methods for the special case of 0–1 networks. The extension to networks with arbitrary edge capacities could be an area of future research. Moreover, it is desirable to analyze the runtime behavior of implicit max-flow algorithms on more complex

¹Implementation available at <http://thefigaro.sourceforge.net/>.



(a) Runtime comparison.



(b) Space usage comparison.

Figure 8: Experimental results on $(2^k + 1) \times (2^k + 1)$ -grid networks.

and praxis-relevant network classes. Promising candidates are slightly irregular grids, decomposable graphs, and graphs with locality properties. Furthermore, flow problems like flow minimization or multicommodity flows as well as other important graph problems like clustering may be attacked by implicit methods.

Acknowledgments

Thanks to Thomas Hofmeister and Ingo Wegener for proofreading and for helpful discussions.

References

- [1] R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2000.
- [2] R.E. Bryant. Symbolic manipulation of Boolean functions using a graphical representation. In *Design Automation Conference*, pages 688–694. ACM Press, 1985.
- [3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [4] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.
- [5] E.A. Dinits. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
- [6] S. Even. *Graph Algorithms*. Computer Science Press, Rockville, 1979.
- [7] L. Fleischer and M. Skutella. The quickest multicommodity flow problem. In *Integer Programming and Combinatorial Optimization*, volume 2337 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2002.
- [8] L.R. Ford and D.R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.
- [9] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1962.
- [10] G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, 1996.

- [11] G.D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0–1 networks. *Formal Methods in System Design*, 10:207–219, 1997.
- [12] R. Hojati, H. Touati, R.P. Kurshan, and R.K. Brayton. Efficient ω -regular language containment. In *Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 396–409. Springer, 1993.
- [13] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7: 277–278, 1978.
- [14] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2000.
- [15] D. Sawitzki. Implicit flow maximization on grid networks. Technical report, Universität Dortmund, 2004. URL <http://ls2-www.cs.uni-dortmund.de/~sawitzki/IFMoGN.pdf>.
- [16] D. Sawitzki. Implicit maximization of flows over time. Technical report, Universität Dortmund, 2004. URL <http://ls2-www.cs.uni-dortmund.de/~sawitzki/IMoFoT.pdf>.
- [17] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, Philadelphia, 2000.
- [18] P. Woelfel. The OBDD-size of cographs. Internal report, Universität Dortmund, 2003.
- [19] P. Woelfel. Symbolic topological sorting with OBDDs. In *Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 671–680. Springer, 2003.