

Implicit Flow Maximization by Iterative Squaring^{*}

Daniel Sawitzki^{**}

University of Dortmund, Computer Science 2
Baroper Str. 301, D-44221 Dortmund, Germany
`daniel.sawitzki@cs.uni-dortmund.de`

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 [1], [2], [3]. 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 $|V| \geq 2^{26}$.

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,

^{*} An extended version of this paper can be obtained via
<http://ls2-www.cs.uni-dortmund.de/~sawitzki/IFMBIS.pdf>.

^{**} Supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Research Cluster "Algorithms on Large and Complex Networks" (1126).

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 regular structures rather than to be randomly originated. If we consider graphs as Boolean functions, we can represent them by *Ordered Binary Decision Diagrams* (OBDDs) [1], [2], [3], which reward regularities with good compression. 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*, and is not essentially larger than explicit ones. Nevertheless, we hope that advantageous properties of G lead to “small,” that is sublinear OBDD-sizes. Examples are grid graphs [4], [5], which have OBDDs of size $O(\log |V|)$, and cographs [6], 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. Although each single operation is efficient, a sequence of $O(\log |V|)$ operations may generate OBDDs of exponential size. Thus, the over-all runtime of an implicit 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 [7], [8], 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 adjacency lists, they are considered as heuristics to save time and/or space when the input graphs are heavily structured and possibly too large for an explicit representation. Hopefully, each OBDD-operation processes many edges in parallel.

Implicit OBDD-algorithms for some particular graph problems like reachability analysis have been well studied in the context of logic design [9]. Newer research tries to attack more general graph problems. The algorithm of Hachtel and Somenzi [10] represents one of the first steps in this direction. It computes a maximum s - t -flow in an implicitly given 0–1 network $N = (V, E, s, t)$. In experiments, the algorithm was able to handle very large state-transition graphs as well as dense random graphs. Anyhow, the individual processing of network layers may enforce a superlinear amount of $\Omega(|V| \log |V|)$ iterations, and destroys the possibility of sublinear runtime. This paper's algorithm circumvents this problem by using the technique of *iterative squaring* (called *IS-algorithm* in the following). Its essential idea is to construct graph paths by iteratively doubling their lengths. In this way, $O(\log^2 |V|)$ OBDD-operations suffice to compute at least one augmenting path. This enables exponential less OBDD-operations than Hachtel and Somenzi's algorithm. We pursue this approach with the focus on heavily structured networks with high diameter, on which we expect efficient over-all runtimes.

The paper is organized as follows: Section 2 introduces the principles of graph representation by OBDDs. Section 3 gives an overview of the IS-algorithm, while separate sections describe its submodules: The layered-network module in Sect. 4, the augmenting path construction in Sect. 5. In Sect. 6, we present analytical and experimental results for the case of grid networks. Finally, Sect. 7 gives conclusions, and hints to possible future research in the area of implicit graph algorithms.

2 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.

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: $\chi_S(x, y) = 1 \Leftrightarrow (x, y) \in S$. The Boolean functions χ_S in turn are represented by OBDDs G_{χ_S} [1], [2], [3], which are graph-based data structures considered as black boxes in this paper.

Because an OBDD G_f for a function $f \in B_n$ is also a graph, its *size* $\text{size}(G_f)$ is measured by the number of its nodes. 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. Then, it is $\text{size}(G_f) = O(2^n/n)$ for every $f \in B_n$, and a graph's edge set $E \subseteq V \times V$ has an OBDD of worst-case size $O(V^2/\log |V|)$.

Whether the function f represented by G_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 $O(\text{size}(G_f))$. Whether two functions f and g are equivalent (i. e., $f = g$) can be decided in time $O(\text{size}(G_f) + \text{size}(G_g))$. These operations are considered as *cheap*. Further essential operations offered by OBDDs are the *binary synthesis* $f \otimes g$ for $f, g \in B_n$, $\otimes \in B_2$ (e. g., “ \wedge ” and “ \vee ”) and the *quantification* $(Qx_i)f$ for a quantifier $Q \in \{\exists, \forall\}$. The computation of $G_{f \otimes g}$ takes time $O(\text{size}(G_f) \cdot \text{size}(G_g))$, and is considered as an *expensive* operation. The computation of $G_{(Qx_i)f}$ can be realized as two cheap operations and one binary synthesis.

Besides implicit node sets and edge sets, there may occur characteristic functions $\chi_S \in B_{kn}$ representing sets $S \in V^k$ during an algorithm, which consequently receive kn Boolean variables encoding k node numbers. We will denote implicit sets $S \in V^k$ by $S(x^1, \dots, x^k)$. In this context, the *swapping* of node arguments is an important OBDD-operation possible in linear time $\text{size}(G)$ (e. g., $F(x, y) := G(y, x)$).

For example, the OBDD G_{χ_E} representing a graph's edge set E will be just denoted by $E(x, y)$; a swap of x and y yields $E(y, x)$, and reverses E 's edges.

Singletons $\{s\} \subseteq V$ are represented by OBDDs $s(x)$ with $s(x) = 1 \Leftrightarrow x = s$. 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-operations, this corresponds to $O(n)$ operations.

3 The IS-Algorithm

This section introduces the IS-algorithm, which computes a maximum flow from a source node s to a terminal node t in a 0–1 network $N = (V, E, s, t)$, where (V, E) is an asymmetric and directed graph. “0–1” means that all edges have capacity 1. The input N is passed to the IS-algorithm as three OBDDs $E(x, y)$, $s(x)$, and $t(x)$ corresponding to the sets E , $\{s\}$, and $\{t\}$. The number n of variables encoding one node number implies $|V| = 2^n$. The algorithm outputs the implicit edges $F(x, y)$ of a maximum set of edge-disjoint s – t -paths, which particularly is a maximum s – t -flow.

Algorithm 1 shows the outline of the IS-algorithm. As Hachtel and Somenzi’s method, it pursues the layered-network approach [11] (assumed to be known by the reader). That is, the algorithm improves an actual flow $F(x, y)$ (which is initially empty) during a sequence of *phases*. In each phase, the *residual network* $A(x, y)$ is obtained by reversing the actual flow edges $F(x, y)$ in $E(x, y)$. Then, the *layered-network* $U(x, y)$ is computed, which contains the edges of all shortest s – t -paths in $A(x, y)$ (called *augmenting paths*). The main loop in lines 3–12 performs phases as long as $A(x, y)$ contains such an augmenting s – t -path (i. e., $F(x, y)$ is not maximum). The layered-networks are computed by the module `layeredNetwork` (lines 2 and 11), described in Sect. 4. It returns the characteristic function $U(x, y)$ respectively the zero-function if no augmenting path exists.

Algorithm 1 The IS-algorithm.

```

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:      $B^*(x, y) := \text{compPaths}(U(x, y))$ ;
7:      $B(x, y) := B(x, y) \vee B^*(x, y)$ ;
8:      $U(x, y) := U(x, y) \wedge B^*(x, y)$ ;
9:   until  $B(x, y)$  is maximal
10:   $F(x, y) := [E(x, y) \wedge B(x, y)] \vee [F(x, y) \wedge \overline{B(y, x)}]$ 
11:   $U(x, y) := \text{layeredNetwork}(E(x, y), s(x), t(x), F(x, y))$ ;
12: end while

```

At next, a maximal (not necessarily maximum) set $B(x, y)$ of edge-disjoint s - t -paths (called *blocking-flow*) is constructed in $U(x, y)$. This construction is divided into iterations called *sweeps* (lines 5–9), each containing one call to the module `compPaths` (see Sect. 5). The latter results in a nonempty set $B^*(x, y)$ of edge-disjoint s - t -paths, that is added to the “blocking-flow candidate” $B(x, y)$ (line 7), while it is removed from $U(x, y)$ (line 8). Starting with $B(x, y) := 0$, this is iterated until $U(x, y)$ contains no further s - t -paths because $B(x, y)$ is maximal. In order to decide this in line 9, we compute the transitive closure [12] of $U(x, y)$ using $O(n^2)$ OBDD-operations.

One step remains to finish the actual phase: The paths of $B(x, y)$ are used to improve (*augment*) the actual flow $F(x, y)$ (line 10). Edges (u, v) with the same direction in $E(x, y)$ and $B(x, y)$ (called *forward-edges*) have not been reversed, and are added to $F(x, y)$. Reversed edges (u, v) with $(u, v) \in B$ and $(v, u) \in E$ (called *backward-edges*) are subtracted from $F(x, y)$. So the augmented flow corresponds to $[E(x, y) \wedge B(x, y)] \vee [F(x, y) \wedge \overline{B(y, x)}]$.

This outline is similar to Hachtel and Somenzi’s algorithm. In Sects. 4 and 5, we will incorporate the concept of iterative squaring in order to reduce the number of operations. The following theorem on the number of OBDD-operations is proven at the end of Sect. 5.

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

4 Layered-Network Construction

Nodes x whose shortest s - x -paths have length ν in $U(x, y)$ are said to be in *node layer* ν . Accordingly, edges from node layer ν to $\nu + 1$ are said to be in *edge layer* ν . Hachtel and Somenzi’s algorithm builds $U(x, y)$ layer by layer. In contrast, the IS-algorithm works 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. Their common node layer will be called *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 $Z_r(x) := t(x)$.

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

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

Then, we determine functions $\text{PATH}_k^{\leq}(x, y)$ and $\text{SPATH}_k(x, y)$ by iterative squaring. $\text{PATH}_k^{\leq}(x, y)$ contains node pairs (x, y) such that there is an x - y -path not longer than 2^k in the residual network $A(x, y)$. $\text{SPATH}_k(x, y)$ contains node

pairs (x, y) with a shortest x - y -path of length exactly 2^k . Both functions are initialized for $k = 0$; then, the path lengths are doubled in each iteration from k to $k + 1$:

$$\begin{aligned} \text{PATH}_0^{\leq}(x, y) &:= (x = y) \vee A(x, y) \text{ ,} \\ \text{PATH}_{k+1}^{\leq}(x, y) &:= (\exists z) [\text{PATH}_k^{\leq}(x, z) \wedge \text{PATH}_k^{\leq}(z, y)] \text{ .} \end{aligned}$$

A similar recursion is used to compute $\text{SPATH}_{k+1}(x, y)$ from $\text{SPATH}_k(x, y)$ starting with $\text{SPATH}_0(x, y) = A(x, y)$. Both PATH_k^{\leq} and SPATH_k are now computed iteratively with increasing k until for some k^* it is either $\text{PATH}_{k^*}^{\leq}(s, t) = 1$ (the terminal has been reached) or $\text{PATH}_{k^*+1}^{\leq}(x, y) = \text{PATH}_{k^*}^{\leq}(x, y)$ (no augmenting path exists; $U(x, y) = 0$ is returned). Due to the exponential growth of the path lengths, it is $k^* = \lceil \log |V| \rceil = O(n)$.

Let us consider the layered-network depth ℓ 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 as ℓ is composed of addends 2^{k_i} . We will use $\text{PATH}_k^{\leq}(x, y)$ and $\text{SPATH}_k(x, y)$ to perform a sort of recursive binary search called `findTerminal` (see Algorithm 2). This subalgorithm decides for every bit ℓ_j of ℓ whether it is set, i. e., whether a partition-component with 2^j edge-layers has to be generated.

In general, we have already computed a sequence $s(x) = Z_0(x), \dots, Z_i(x)$ of implicit separation-layers, the component length exponents k_0, \dots, k_{i-1} , and the set $R(x)$ of nodes covered by the part of U considered so far. Subalgorithm `findTerminal` is called to compute the remaining $(Z_{i+1}(x), \dots, Z_r(x)) =: \mathcal{Z}$ as well as $(k_i, \dots, k_{r+1}) =: \mathcal{K}$. It receives the arguments $Z(x) := Z_i(x)$, $R(x)$, and the exponent a , which determines the search interval $[0, 2^a]$. We start the binary search with `findTerminal`($s(x), s(x), k^*$) from source node $s(x)$ with $a = k^*$, because 2^{k^*} is an upper bound for the depth ℓ of U . The result is the lists $(Z_1(x), \dots, Z_r(x))$ and (k_0, \dots, k_{r-1}) . Figure 1 shows the partitioning of an example layered-network of depth $\ell = 13$.

Algorithm 2 The recursive algorithm `findTerminal`($R(x), Z(x), a$).

- 1: **if** $[\text{SPATH}_a(x, t) \wedge Z(x)] \neq 0$ **then** {Case 1}
 - 2: return $t(x), a$;
 - 3: **else if** $[\text{PATH}_{a-1}^{\leq}(x, t) \wedge Z(x)] \neq 0$ **then** {Case 2}
 - 4: return `findTerminal`($R(x), Z(x), a - 1$);
 - 5: **else** {Case 3}
 - 6: $Z'(x) := (\exists y) [\text{SPATH}_{a-1}(y, x) \wedge Z(y)] \wedge \overline{R(x)}$;
 - 7: $R'(x) := R(x) \vee (\exists y) [\text{PATH}_{a-1}^{\leq}(y, x) \wedge Z(y)]$;
 - 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**
-

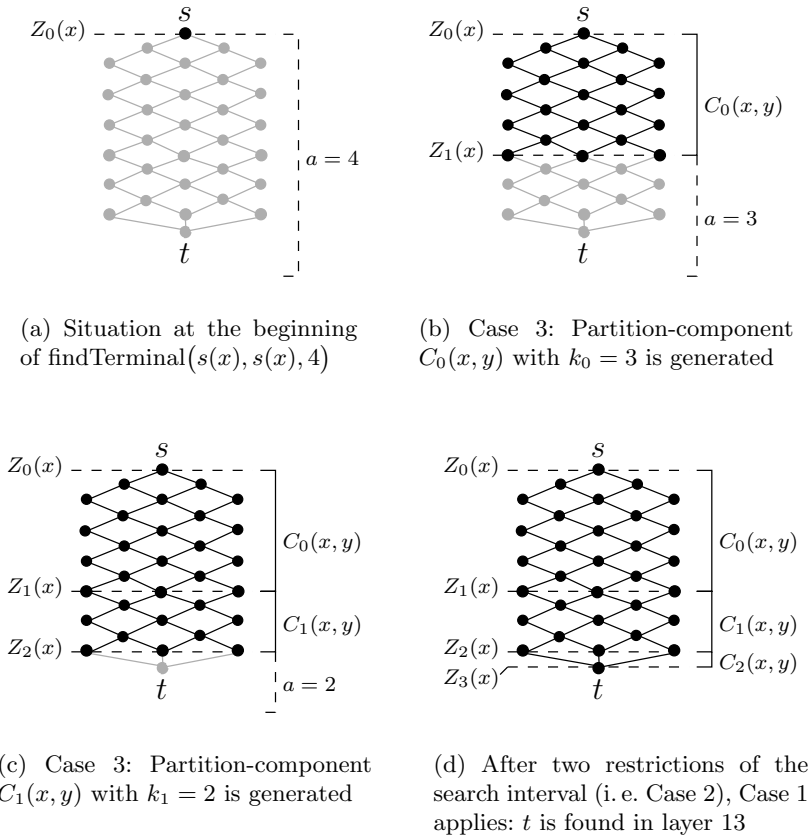


Fig. 1. Partitioning an example layered-network of depth 13. Black nodes/edges have already been partitioned; grey parts are unknown yet. Dashed lines indicate separation-layers. Solid brackets span partition-components, dashed brackets span search intervals

We now consider the three cases which are distinguished within Algorithm 2.

Case 1. It is $[SPATH_a(x, t) \wedge Z(x)] \neq 0$: The depth 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).

Case 2. It is $[PATH_{a-1}^{\leq}(x, t) \wedge Z(x)] \neq 0$: 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.

Case 3. None of the two cases above applies: On the one hand, 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 return $Z'(x)$ and $a - 1$ (line 9) prepended to the result lists \mathcal{Z}' resp. \mathcal{K}' of the remaining partitioning (done by `findTerminal($R'(x), Z'(x), a - 1$)` in line 8). The binary search continues in the right half of the search space.

With $SPATH_k(x, y)$, $k \in \{k_0, \dots, k_{r-1}\}$, and $Z_i(x)$, $i \in \{0, \dots, r\}$, we have all necessary information to compute the layered-network edges $U \subseteq \bigcup_{i=0}^{r-1} C_i$. For this easy step we refer to the extended version of this paper. Finally, $U(x, y)$ contains exactly those edges belonging to a shortest augmenting s - t -path in the residual network $A(x, y)$.

5 Blocking-Flow Construction

The module `compPaths` computes a non-empty set $B^*(x, y)$ of edge-disjoint s - t -paths in $U(x, y)$. At first, it tries to construct a possibly large set of paths by composing longer paths from shorter ones, starting with simple edges as paths of length 1. This method (called *multi-path method*) may fail and return an empty set of paths. In this case, the *single-path method* is applied, which computes exactly one s - t -path p in $U(x, y)$. Therefore, `compPaths` returns at least one path. In the following, we will present only the single-path method, while the multi-path method can be found in this paper's extended version. Both methods use $O(n^2)$ OBDD-operations.

5.1 Single-Path Method

All s - t -paths in $U(x, y)$ have length $\ell \leq |V| - 1$ (the layered-network depth). Although ℓ has not to be a power of two, we assume $\ell = 2^h$ for $h \in \mathbb{N}$, in order to simplify the method's description. For general depths $\ell \notin \{2^h \mid h \in \mathbb{N}\}$, the algorithm can be easily adapted.

In a preprocessing step, we compute functions $P_k(x, y, z)$, $k \in \{0, \dots, h\}$, representing triples (x, y, z) such that an x - y -path as well as a y - z -path both of length 2^{k-1} exist in $U(x, y)$. This is done by iterative squaring similar to $PATH_k^{\leq}(x, y)$ in Sect. 4. Then, the edges $B^*(x, y)$ of an augmenting s - t -path p are obtained by computing functions $D_k(x, z)$ for $k \in \{0, \dots, h\}$. We initialize $D_h(x, z) := s(x) \wedge t(z)$ with the pair (s, t) of p 's start- and end-node. In general, $D_k(x, z)$ contains pairs (x, z) connected by paths of length 2^k in $U(x, y)$. This means that p visits x and z , while the path between these nodes is not fixed yet. Therefore, for every pair $(x, z) \in D_k$ one node y with $(x, y, z) \in P_k$ is fixed to be part of p . These resulting pairs (x, y) and (y, z) , which represent subpaths of length 2^{k-1} , are united in $D_{k-1}(x, z)$.

In this way, the set $D_h(x, z)$ (which initially corresponds to all possible s - t -paths) shrinks to the set $D_0(x, z)$ whose pairs (x, z) are just edges of one

augmenting path p . We now describe how $D_{k-1}(x, z)$ is computed from $D_k(x, z)$. At first, we determine all triples $(x, y, z) \in P_k$ whose start- and end-nodes x resp. z are fixed to be part of p :

$$Q_k(x, y, z) := P_k(x, y, z) \wedge D_k(x, z) .$$

$Q_k(x, y, z)$ now represents all possible subpaths (x, \dots, y, \dots, z) of length 2^k . The middle-nodes y are selected using a *priority function* $\Pi_{x,z}$, which represents a total order $<_{x,z}$ on V depending on x and z . Hachtel and Somenzi [10] suggested two priority functions whose OBDDs have size $O(n)$.

$$\Pi_{x,z}(y', y) = 1 \Leftrightarrow y' <_{x,z} y$$

$$T_k(x, y, z) := Q_k(x, y, z) \wedge \overline{(\exists y') [Q_k(x, y', z) \wedge \Pi_{x,z}(y', y)]}$$

A triple (x, y, z) is in T_k if and only if (x, \dots, y, \dots, z) is a possible subpath of p that respects D_k (guaranteed by Q_k) and there is no alternative $(x, y', z) \in Q_k$ whose middle-node y' is lower than y according to $\Pi_{x,z}$. That is, we choose the node y that is minimal w. r. t. $<_{x,z}$. $D_{k-1}(x, z)$ simply consists of all “upper” and “lower” parts (x, y) and (y, z) of the triples $(x, y, z) \in T_k$.

Finally, we return the edges of p as $B^*(x, y) := D_0(x, y)$. Figure 2 shows an example of a single-path construction in a layered-network of depth $\ell = 8$.

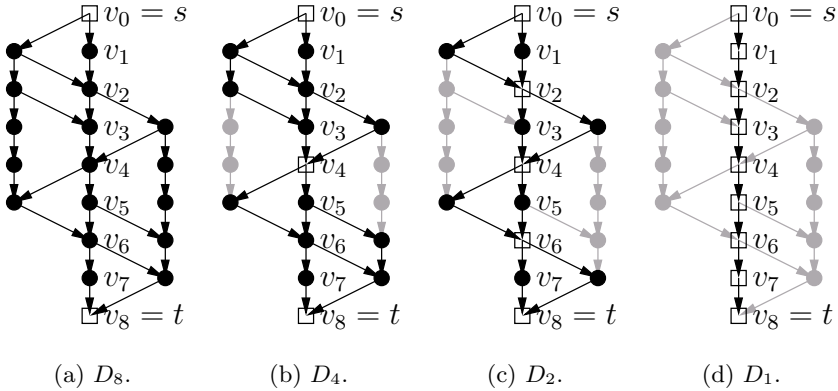


Fig. 2. Single-path construction of a path $(s = v_0, \dots, v_8 = t)$. Boxes symbolize nodes that are already fixed in D_k . Grey nodes/edges symbolize parts of $U(x, y)$ which are already excluded from the path construction. Initially, D_8 contains the start-/end-node pair (v_0, v_8) . Finally, D_1 contains p 's edges and the path construction is finished

Proof (Theorem 1). Without loss of generality, we assume that a new layered-network is computed after each sweep. Each of these $O(S)$ layered-network computations contains $O(k^* + r) = O(\log |V|)$ iterations. Each iteration performs $O(n) = O(\log |V|)$ OBDD-operations (caused by quantifications over node

number of length n). Each sweep calls module `compPaths`, which consists of $O(h) = O(\log |V|)$ iterations with $O(n)$ operations each. The maximality test of $B(x, y)$ incorporates a transitive closure computation with $O(\log^2 |V|)$ operations. The remaining steps of removing $B^*(x, y)$ from $U(x, y)$ and augmenting $F(x, y)$ by $B(x, y)$ can be done by a constant number of operations. Altogether, each sweep executes $O(\log |V|^2)$ OBDD-operations, and increases F 's value. \square

6 Grid Networks

In this section, both an analytical and an experimental result for the case of quadratic $(2^k + 1) \times (2^k + 1)$ -grid networks $N = (V, E, s, t)$ is presented. (For a detailed analysis the reader is referred to [5].) These grids are $(2^k + 1) \times (2^k + 1)$ -matrices of nodes $V = \{0, \dots, 2^k\}^2$ 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 IS-algorithm executes two sweeps at most. In fact, it can be shown that only one sweep is necessary. Figure 3 shows such a network for $k = 2$ as well as the maximum flow constructed by the IS-algorithm.

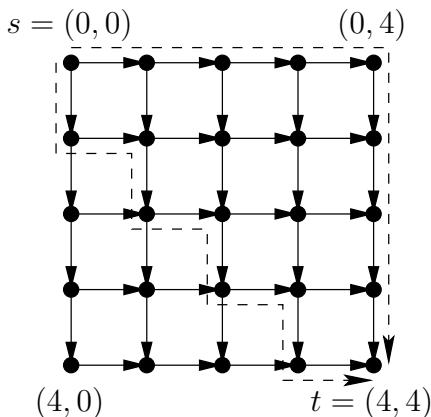


Fig. 3. The 5×5 -grid network and a maximum flow. Dashed arrows indicate the constructed maximum flow

6.1 Analysis

Theorem 2. *The IS-algorithm computes a maximum flow on $(2^k + 1) \times (2^k + 1)$ -grid networks in time $O(k^3)$ and space $O(k^2)$.*

It is $|V| = 2^{2k} + 2^{k+1} + 1$ and $k = \Theta(\log |V|)$. We assume that grid nodes $x \in V$ are encoded by $n := 2(k + 1)$ Boolean variables $x_{n-1} \dots x_0$, which represent the row

index x^r and column index x^c of x each by $k + 1$ bits. Consider a function $F \in B_{d(k+1)}$ defined on $d = O(1)$ row or column arguments $y^1, \dots, y^d \in \{0, 1\}^{k+1}$. If there are weights $w_1, \dots, w_d \in \{-1, 0, 1\}$ and a threshold value $T \in \mathbb{Z}$ such that

$$F(y^1, \dots, y^d) = \left(\sum_{i=1}^k w_i \cdot |y^i| \geq T \right),$$

then F is called a *multivariate threshold function* [4]. Any constant-size formula over such multivariate threshold functions can be represented by an OBDD of size $O(k)$. Due to Woelfel [13], this still holds after the application of an arbitrary sequence $\exists y^{i_1} \dots \exists y^{i_m}$ of m existential quantifiers, $i_1, \dots, i_m \in \{1, \dots, d\}$. Moreover, operations on such OBDDs take only time $O(k)$ [5].

The idea of the analysis is to prove for functions $F \in B_{dn}$ occurring during the flow maximization that they realize constant-size formulas over multivariate threshold functions. We exemplarily consider the implicit edge set $E(x, y)$:

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

Only nodes within the $(2^k + 1) \times (2^k + 1)$ -square are touched by edges. There is an edge between node x and y if and only if either their horizontal or vertical distance is 1.

In the same way, it can be shown that all OBDDs during the flow maximization on $(2^k + 1) \times (2^k + 1)$ -grid networks have size $O(k)$ and can also be processed in time $O(k)$. This implies a runtime $O(k)$ for each of the $O(k^2)$ operations and, therefore, the over-all runtime result. Because no more than $O(k)$ OBDDs are present at the same time during the algorithm, the space usage is $O(k^2)$. In contrast, Hachtel and Somenzi's method [10] needs $\Omega(2^k \cdot k)$ operations. These results hold if the square between s and t is embedded in a larger non-quadratic grid.

6.2 Experimental Results

Both the IS-algorithm and Hachtel and Somenzi's maximum flow method [10] 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 4(a) and 4(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 $k \geq 16$.

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

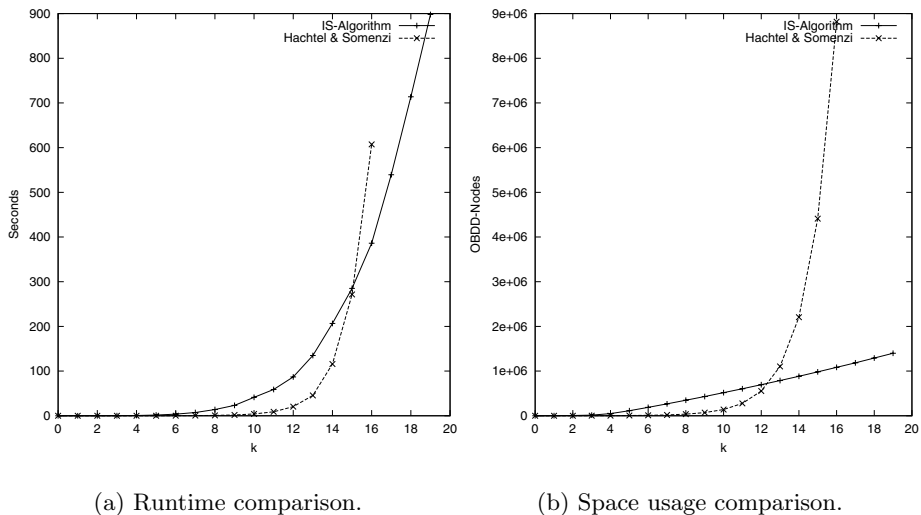


Fig. 4. Experimental results on $(2^k + 1) \times (2^k + 1)$ -grid networks

7 Conclusions

An implicit algorithm for flow maximization in 0–1 networks has been presented which uses the technique of iterative squaring. In this way, only $O(\log^2 |V|)$ OBDD-operations are needed to compute a layered-network or at least one augmenting path. At best, this amount suffices for the whole flow computation (in particular for constant maximum flow values). The over-all runtime is influenced also by the occurring OBDD-sizes. These have been analyzed exemplarily for grid networks, on which the IS-algorithm beats previous methods both asymptotically and in experiments.

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 sophisticated 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 may be attacked by implicit methods.

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

References

1. Bryant, R.: Symbolic Manipulation of Boolean Functions Using a Graphical Representation. In: Design Automation Conference, ACM Press (1985) 688–694
2. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* **35** (1986) 677–691
3. Wegener, I.: *Branching Programs and Binary Decision Diagrams*. SIAM, Philadelphia (2000)
4. Woelfel, P.: Symbolic Topological Sorting with OBDDs. Volume 2747. (2003) 671–680
5. Sawitzki, D.: Implicit Flow Maximization on Grid Networks. Technical Report, Universität Dortmund (2003)
6. Woelfel, P.: The OBDD-Size of Cographs. Internal Report, Universität Dortmund (2003)
7. Bloem, R., Gabow, H., Somenzi, F.: An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. In: Formal Methods in Computer-Aided Design. LNCS, Vol. 1954. Springer (2000) 37–54
8. Ravi, K., Bloem, R., Somenzi, F.: A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In: Formal Methods in Computer-Aided Design. LNCS, Vol. 1954. Springer (2000) 143–160
9. Hachtel, G., Somenzi, F.: *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston (1996)
10. Hachtel, G., Somenzi, F.: A Symbolic Algorithm for Maximum Flow in 0–1 Networks. *Formal Methods in System Design* **10** (1997) 207–219
11. Even, S.: *Graph Algorithms*. Computer Science Press, Rockville (1979)
12. Hojati, R., Touati, H., Kurshan, R., Brayton, R.: Efficient ω -Regular Language Containment. In: Computer-Aided Verification. LNCS, Vol. 663. Springer (1993) 396–409
13. Woelfel, P.: Private Communication (2003)