An Efficient Implicit OBDD-Based Algorithm for Maximal Matchings*

Beate Bollig[1] and Tobias Pröger[2]

[1] TU Dortmund, LS2 Informatik, Germany
[2] ETH Zürich, Institut für Theoretische Informatik, Switzerland

Abstract. The maximal matching problem, i.e., the computation of a matching that is not a proper subset of another matching, is a fundamental optimization problem and algorithms for maximal matchings have been used as submodules for problems like maximal node-disjoint paths or maximum flow. Since in some applications graphs become larger and larger, a research branch has emerged which is concerned with the design and analysis of implicit algorithms for classical graph problems. Inputs are given as characteristic Boolean functions of the edge sets and problems have to be solved by functional operations efficiently supported by the chosen data structure. An OBDD is a well-known data structure for Boolean functions and sometimes capable to take advantage over the presence of regular substructures which may lead to sublinear graph representations. As a result OBDD-based algorithms are used as a heuristic approach to handle very large graphs. Here, an implicit OBDD-based maximal matching algorithm is presented that uses only a polylogarithmic number of functional operations with respect to the number of vertices of the input graph. In order to investigate the algorithm's behavior on large and structured networks, it has been analyzed on grid graphs and it has been shown that the overall running time and the space requirement is also polylogarithmic.

1 Introduction

Since some modern applications require huge graphs, explicit representations by adjacency matrices or adjacency lists may cause conflicts with memory limitations and even polynomial time algorithms are sometimes not fast enough. As time and space resources do not suffice to consider individual vertices and edges, one way out seems to be to deal with sets of vertices and edges represented by their characteristic functions. Ordered binary decision diagrams, denoted OBDDs, introduced by [6], are well suited for the representation and manipulation of Boolean functions and sometimes capable to take advantage over the presence of regular substructures which leads to sublinear graph representations. Therefore, a research branch has emerged which is concerned with the design and analysis of so-called implicit or symbolic algorithms for classical graph problems on OBDD-represented graph instances (see, e.g., [8, 9], [11], [15], [16, 17], and [21]). Implicit algorithms have to solve problems on a given graph instance by efficient functional operations offered by the OBDD data structure. Here, a functional operation is an operation that works on Boolean functions. Representing graphs with regularities by means of data structures smaller than adjacency matrices or adjacency lists seems to be a natural idea but for several graph problems an exponential blow-up from input to output size is possible in the implicit setting ([1–3] and [16]). However, OBDD-based algorithms are successful in many applications and it has been pointed out that worst-case hardness results do not adequately capture the complexity of the problems on real-world

* The first author is supported by DFG project BO 2755/1-1.

instances ([7]). Moreover, in practical applications, typically, a careful analysis of the problem at hand reveals sufficient structure to limit the graphs under investigation to a restricted class.

The design of efficient implicit algorithms requires new paradigms and techniques but it has turned out that some methods known from the design of parallel algorithms are useful, e.g., the technique of iterative squaring is similar to the path-doubling strategy. For instance, it is well known that the transitive closure of a graph, i.e., all pairs of vertices (u, v) in an input graph G for which there exists a path from u to v, can be computed similarly as in parallel algorithm by $\mathcal{O}(\log^2 |V|)$ functional operations using iterative squaring. One may remark that one has to be careful whether to use iterative squaring because despite the improvement in the number of functional operations there can be intermediate results of exponential size (with respect to the input length). Nevertheless, Sawitzki has demonstrated that iterative squaring can also be useful in applications. The maximum flow problem in 0-1 networks has been one of the first classical graph problems for which an implicit OBDD-based algorithm has been presented and Hachtel and Somenzi were able to compute a maximum flow for a graph with more than 10^{27} vertices and 10^{36} edges in less than one CPU minute ([11]). To improve this algorithm Sawitzki has used iterative squaring for the computation of augmenting paths by $\mathcal{O}(\log^2 |V|)$ functional operations ([15]). If the maximum flow value is constant with respect to the network size, the algorithm performs altogether a polylogarithmic number of operations. Both max flow algorithms belong to the class of so-called layered-network methods but Sawitzki's algorithm prevents breadth-first searches by using iterative squaring and as a result overcomes the dependence on the depths of the layered networks. In order to confirm the practical relevance of his algorithm he has implemented both maximum flow algorithms and has shown that his algorithm outperforms the algorithm of Hachtel and Somenzi for very structured graphs.

The maximal matching problem, i.e., the computation of a matching that is not a proper subset of another matching, is a fundamental optimization problem and algorithms for maximal matchings have been used in some maximal node-disjoint paths or maximum flow algorithms (see, e.g., [10]). Using an efficient degree reduction procedure, the first optimal parallel algorithm for maximal matchings has been presented by [13]. It runs in time $\mathcal{O}(\log^3 |V|)$ using $\mathcal{O}((|E| + |V|)/\log^3 |V|)$ processors on bipartite graphs $G = (V, E)$ and is optimal in the sense that the time processor product is equal to that of the best sequential algorithm. The main result of our paper is an efficient implicit algorithm for the computation of maximal matchings. Here, we make use of the algorithm presented by [13] but for the implicit setting also new ideas are necessary.

Theorem 1. *A maximal bipartite matching in an implicitly defined graph $G = (V, E)$ can be implicitly computed by $\mathcal{O}(\log^4 |V|)$ functional operations on Boolean functions over a logarithmic number of Boolean variables. For general graphs $\mathcal{O}(\log^5 |V|)$ functional operations are sufficient.*

Note, that our aim is not to achieve new algorithmic techniques for explicit graph representations but to demonstrate the similarity of paradigms in the design of parallel and implicit algorithms that can also be used as building blocks for the solution of other combinatorial problems on one hand and on the other hand to develop efficient algorithms for large structured graphs.

The similarity between implicit and parallel algorithms has also been demonstrated by the following result proved by [17, 18]. A problem can be solved in the implicit setting with a polylogarithmic number of functional operations on a logarithmic number of Boolean variables (with respect to the number of vertices of the input graph) if and only if the problem

is in NC, the complexity class that contains all problems computable in polylogarithmic time with polynomially many processors. Nevertheless, this structural result does not lead directly to efficient implicit algorithms. Since a problem in NC can be characterized by so-called logarithmic space-uniform circuits of polynomial size and polylogarithmic depth, an implicit algorithm has been generated by simulating the logarithmic space-uniform Turing machine which computes the corresponding circuit followed by a simulation of this circuit. During the simulation Boolean functions are used whose OBDD size can almost be cubic with respect to the circuit size. Therefore, the resulting implicit algorithms are presumably not the most efficient ones to solve the corresponding problems in NC.

The rest of the paper is organized as follows. In Section 2 we define some notation and review some basics concerning OBDDs and functional operations, implicit graph representations and matchings, as well as important Boolean functions in the design and analysis of implicit graph algorithms. Section 3 contains the main result, an implicit algorithm for the maximal matching problem that uses only a polylogarithmic number of functional operations with respect to the number of vertices of the input graph. Since most operations on OBDDs require time and space proportional to the sizes of the operands, the number of functional operations is only a rough measure for the efficiency of an implicit algorithm. Therefore, we demonstrate in Section 4 that there exists graphs for which our implicit algorithm is enormously more efficient than the traditional algorithms. Finally, we finish the paper with some concluding remarks.

2 Preliminaries

In order to make the paper self-contained we briefly recall the main notions we are dealing with in this paper.

2.1 OBDDs and functional operations

OBDDs are a very popular dynamic data structure in areas working with Boolean functions, like circuit verification or model checking. (For a history of results on binary decision diagrams see, e.g., [20]).

Definition 1. *Let $X_n = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. A variable ordering π on X_n is a permutation on $\{1, \ldots, n\}$ leading to the ordered list $x_{\pi(1)}, \ldots, x_{\pi(n)}$ of the variables. A π-*OBDD *on X_n is a directed acyclic graph $G = (V, E)$ whose sinks are labeled by the Boolean constants 0 and 1 and whose non-sink (or decision) nodes are labeled by Boolean variables from X_n. Each decision node has two outgoing edges, one labeled by 0 and the other by 1. The edges between decision nodes have to respect the variable ordering π, i.e., if an edge leads from an x_i-node to an x_j-node, then $\pi^{-1}(i) < \pi^{-1}(j)$ (x_i precedes x_j in $x_{\pi(1)}, \ldots, x_{\pi(n)}$). Each node v represents a Boolean function $f_v \in B_n$, i.e., $f_v : \{0,1\}^n \to \{0,1\}$, defined in the following way. In order to evaluate $f_v(b)$, $b \in \{0,1\}^n$, start at v. After reaching an x_i-node choose the outgoing edge with label b_i until a sink is reached. The label of this sink defines $f_v(b)$. The size of a π-*OBDD *G, denoted by $|G|$, is equal to the number of its nodes. A π-*OBDD *of minimal size for a given function f and a fixed variable ordering π is unique up to isomorphism. A π-*OBDD *for a function f is called reduced if it is the minimal π-*OBDD *for f. The π-*OBDD *size of a function f, denoted by π-*OBDD*(f), is the size of the reduced π-*OBDD *representing f. The *OBDD *size of f is the minimum of all π-*OBDD*(f).*

Sometimes it is useful to have the notion of OBDDs where there are only edges between nodes labeled by neighboring variables, i.e., if an edge leads from an x_i-node to an x_j-node, then $\pi^{-1}(i) = \pi^{-1}(j) - 1$.

Definition 2. *An* OBDD *on X_n is complete if all paths from the source to one of the sinks have length n. The width of a complete* OBDD *is the maximal number of nodes labeled by the same variable.*

Let f be a Boolean function on the variables $x_1, \ldots, x_n$. The subfunction $f_{|x_i=c}$, $1 \leq i \leq n$ and $c \in \{0, 1\}$, is defined as $f(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n)$. It is well known that the size of an OBDD representing a function f that depends essentially on n Boolean variables (a function f depends essentially on a Boolean variable z if $f_{|z=0} \neq f_{|z=1}$) may be different for different variable orderings and may vary between linear and exponential size with respect to n. In the following a variable ordering π is sometimes identified with the corresponding ordering $x_{\pi(1)}, \ldots, x_{\pi(n)}$ of the variables if the meaning is clear from the context. The size of the reduced π-OBDD representing f is described by the following structure theorem proved by [19].

Theorem 2. *The number of $x_{\pi(i)}$-nodes of the minimal π-*OBDD *for f is the number of different subfunctions $f_{|x_{\pi(1)}=a_1,\ldots,x_{\pi(i-1)}=a_{i-1}}$, $a_1, \ldots, a_{i-1} \in \{0, 1\}$, that essentially depend on $x_{\pi(i)}$. The number of $x_{\pi(i)}$-nodes of the minimal complete π-*OBDD *for f is the number of different subfunctions $f_{|x_{\pi(1)}=a_1,\ldots,x_{\pi(i-1)}=a_{i-1}}$, $a_1, \ldots, a_{i-1} \in \{0, 1\}$.*

Now, we briefly describe a list of important operations on data structures for Boolean functions and the corresponding time and additional space requirements for OBDDs (for a detailed discussion see, e.g., [20]). In the following let f and g be Boolean functions in B_n on the variable set $X_n = \{x_1, \ldots, x_n\}$ and G_f and G_g be π-OBDDs for the representations of f and g, respectively.

- Negation: Given G_f, compute a π-OBDD for the function $\overline{f} \in B_n$. This can be done in time $\mathcal{O}(1)$.
- Replacement by constant: Given G_f, an index $i \in \{1, \ldots, n\}$, and a Boolean constant $c_i \in \{0, 1\}$, compute a π-OBDD for the subfunction $f_{|x_i=c_i}$. This can be done in time $\mathcal{O}(|G_f|)$ and the π-OBDD for $f_{|x_i=c_i}$ is not larger than G_f.
- Minimization: Given a π OBDD G_f, compute the reduced π-OBDD for f. This can be done in time and space $\mathcal{O}(|G_f|)$.
- Equality test: Given G_f and G_g, decide, whether f and g are equal. This can be done in time $\mathcal{O}(|G_f| + |G_g|)$.
- Satisfiability count: Given G_f, compute $|f^{-1}(1)|$. This can be done in time $O(|G_f|)$.
- Synthesis: Given G_f and G_g and a binary Boolean operation $\otimes \in B_2$, compute a π-OBDD G_h for the function $h \in B_n$ defined as $h := f \otimes g$. This can be done in time and space $\mathcal{O}(|G_f| \cdot |G_g|)$ and the size of G_h is bounded above by $\mathcal{O}(|G_f| \cdot |G_g|)$.
  Let G_h^* be the graph that consists of the nodes in the product graph of G_f and G_g reachable from the node representing the function h. The computation of G_h can be done in time $\mathcal{O}(|G_h^*| \log |G_h^*|)$ and space $\mathcal{O}(|G_h^*|)$.
- Quantification: Given G_f, an index $i \in \{1, \ldots, n\}$, and a quantifier $Q \in \{\exists, \forall\}$, compute a π-OBDD G_h for the function $h \in B_n$ defined as $h := (Qx_i)f$, where $(\exists x_i)f := f_{|x_i=0} \vee f_{|x_i=1}$ and $(\forall x_i)f := f_{|x_i=0} \wedge f_{|x_i=1}$. The computation of G_h can be realized by two replacements of constants and a synthesis operation. This can be done in time and space $\mathcal{O}(|G_f|^2)$.

In the rest of the paper quantifications over k Boolean variables $(Qx_1, \dots, x_k)f$ are denoted by $(Qx)f$, where $x = (x_1, \dots, x_k)$.

The following result proved by [17] has turned out to be helpful in the analysis of implicit graph algorithms.

Lemma 1. *Let $f \in B_n$ be a function on the variable set $X_n = \{x_1, \dots, x_n\}$, $X \subseteq X_n$, $Q \in \{\exists, \forall\}$, and G_f be a complete* OBDD *representing f with respect to a variable ordering π. The function h is defined by $(QX)f$, which means the function obtained from f by quantifying all variables in X. The width of the minimal complete π*-OBDD *G_h for h is bounded by 2^w, where w is the width of G_f. G_h can be computed in time $\mathcal{O}(|X| n 2^{2w} \log(n 2^{2w}))$ and space $\mathcal{O}(n 2^{2w})$.*

Sometimes it is useful to reverse the edges of a given graph. Therefore, we define the following operation (see, e.g., [16]).

Definition 3. *Let ρ be a permutation on $\{1, \dots, k\}$ and $f \in B_{kn}$ be defined on Boolean variable vectors $x^{(1)}, \dots, x^{(k)}$ of length n. The argument reordering $\mathcal{R}_\rho(f) \in B_{kn}$ with respect to ρ is $\mathcal{R}_\rho(f)(x^{(1)}, \dots, x^{(k)}) = f(x^{(\rho(1))}, \dots, x^{(\rho(k))})$.*

Given a π-OBDD G_f representing f a π-OBDD for the function $\mathcal{R}_\rho(f)$ can be computed by renaming the variables followed by at most $(k-1)n$ so-called jump-up operations, where a variable jumps to another position in the variable ordering. A jump-up operation can be realized by two replacements of constants followed by a synthesis step (see, e.g., [4]). A k-interleaved variable ordering on the k variable vectors $x^i = (x^i_n, \dots, x^i_1)$, $1 \le i \le k$, denoted by $\pi^\tau_{k,n}$ is defined in the following way:

$$\pi^\tau_{k,n} = \left(x^{(1)}_{\tau(1)}, \dots, x^{(k)}_{\tau(1)}, x^{(1)}_{\tau(2)}, \dots, x^{(k)}_{\tau(2)}, \dots, x^{(k)}_{\tau(n)} \right),$$

where $\tau \in \Sigma_n$, the set of all permutations on $\{1, \dots, n\}$. In the design and analysis of implicit graph algorithms it is very common to use interleaved variable orderings mainly because of two reasons. The first one is that some important Boolean function can be represented in small size according to such an ordering and the second one is that an argument reordering operation does not lead to a blow-up in the representation size of the function if k is a constant which means independent of n. To be more precise, in this case it is not difficult to show that the size for the result of the reordering procedure is only a factor of 2^k larger than the size of the input OBDD. If we consider complete OBDDs for f and $\mathcal{R}_\rho(f)$ and interleaved variable orderings, the width of the OBDD for $\mathcal{R}_\rho(f)$ obtained by the reordering procedure is at most by a factor of 2^{k-1} larger than the width of the OBDD for f. (These results can be shown by the proof of Theorem 2 presented by [4].)

Altogether, the negation, replacement by constant, minimization, equality test, satisfiability count, and synthesis are functional operations. The quantification over k variables can be realized by $3k$ functional operations and the argument reordering over k vectors of length n by $3(k-1)n$ functional operations.

The test whether a Boolean function f represented by an OBDD G_f is not the constant function 0, in other words the satisfiability test, can be done in different ways. One possibility is to make an equality test between G_f and an OBDD for the constant function 0 that consists only of the 0-sink. Another one is to perform the operation satisfiability count. Both can be done in time and space $\mathcal{O}(|G_f|)$.

2.2 OBDD-based graph representations and matching problems

Let $G = (V, E)$ be a graph with N vertices $v_0, \ldots v_{N-1}$ and $|z|_2 := \sum_{i=0}^{n-1} z_i 2^i$, where $z = (z_{n-1}, \ldots, z_0) \in \{0,1\}^n$ and $n = \lceil \log N \rceil$. Now, E can be represented by an OBDD for its characteristic function, where $x, y \in \{0,1\}^n$ and

$$\chi_E(x, y) = 1 \Leftrightarrow (|x|_2, |y|_2 < N) \wedge (v_{|x|_2}, v_{|y|_2}) \in E.$$

(For the ease of notation we omit the index 2 in the rest of the paper.) Undirected edges are represented by symmetric directed ones. In the rest of the paper we assume that N is a power of 2 since it has no bearing on the essence of our results. Furthermore, we do not distinguish between vertices of the input graph and their Boolean encoding since the meaning is clear from the context. It is well known that for every variable ordering π the size of the reduced π-OBDD for a given function $f \in B_n$ is upper bounded by $(2 + o(1))2^n/n$ (see, e.g., [5]). Moreover, it is not difficult to prove that the size is also upper bounded by $O(n \cdot \min(|f^{-1}(1)|, |f^{-1}(0)|))$, where $|S|$ denotes the cardinality of a set S and $f^{-1}(1)$ ($f^{-1}(0)$) is the set of the 1-inputs (0-inputs) for f. The idea is to show that there cannot be more than $|f^{-1}(1)|$ or $|f^{-1}(0)|$ nodes labeled by the same variable in an OBDD for f since there are at most $\min(|f^{-1}(1)|, |f^{-1}(0)|)$ different subfunctions which depend on the same variable. Therefore, the characteristic function $\mathcal{X}_E$ of an edge set $E \subseteq V \times V$ can be represented by OBDDs of size $\mathcal{O}(\min(|V|^2/\log|V|, |E|\log|V|))$. To be more precise, a reduced OBDD representing $\mathcal{X}_E$ with respect to an arbitrary variable ordering has size $\mathcal{O}(\min(|V|^2/\log|V|, |E|\log|V|))$.

A graph $G = (V, E)$ is bipartite, if V can be partitioned into two disjoint nonempty sets U and W, such that for all edges $(u, w) \in E$ it holds $u \in U$ and $w \in W$ or vice versa. The distance between two edges on a (directed) path is the number of edges between them. The distance between two vertices on a (directed) path is the number of vertices between them plus 1. The degree of a vertex v in G is the number of edges in E incident to v. A matching in an undirected graph $G = (V, E)$ is a subset $M \subseteq E$ such that no two edges of M are adjacent. M is a maximum matching if there exists no matching $M' \subseteq E$ such that $|M'| > |M|$. A matching M is maximal if M is not a proper subset of another matching. Given a matching M a vertex v is matched if $(v, w) \in M$ for some $w \in V$ and free otherwise.

In the implicit setting the maximum (maximal) matching problem is the following one. Given an OBDD for the characteristic function of the edge set of an undirected input graph G, the output is an OBDD that represents the characteristic function of a maximum (maximal) matching in G.

2.3 Important Boolean functions for the design and analysis of OBDD-based graph algorithms

For implicit computations some Boolean functions are helpful. The equality function EQ_n is defined on two n-bit inputs x and y and computes 1 iff $|x| = |y|$. NEQ is the negated equality function. It is easy to see that both functions can be represented in linear size with respect to the number of Boolean variables if the variables with the same significance are tested one after another.

Since sometimes a vertex (or an edge) has to be chosen out of a given set of vertices (or edges), several priority functions $\Pi_{\prec}$ have been defined in the implicit setting (see, e.g., [11, 15]). We define $\Pi_{\prec}(x, y, z) = 1$ iff $y \prec_x z$, where $\prec_x$ is a total order on the vertex set V and x, y, z are vertices in V. In the following we only use a very simple priority function independent of the choice of x, where $\Pi_{\prec}(x, y, z) = 1$ iff $|y| < |z|$. It is easy to show that

$\Pi_{\prec}$ can be represented by OBDDs of linear size with respect to variable orderings, where y_i and z_i are tested one after another.

The equality, nonequality, and priority function are special cases of a class of functions called multivariate threshold and modulo functions introduced by [21].

Definition 4. *Let $X_{k,n}$ be the set of variables x_j^i, where $1 \le i \le k$ and $0 \le j \le n-1$, and x^i the vector of the n variables $(x_{n-1}^i, \dots, x_0^i)$. A Boolean function $f \in B_{kn}$ defined on the variable set $X_{k,n}$ is called k-variate threshold function, if there exist a threshold $T \in \mathbb{Z}$ and weights $w_1, \dots, w_k \in \mathbb{Z}$ such that*

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

Let $w := \max(|w_1|, \dots, |w_k|)$. The set of k-variate threshold functions on $X_{k,n}$ is denoted by $\mathbb{T}_{k,n}^w$.

A Boolean function $g \in B_{kn}$ defined on the variable set $X_{k,n}$ is called k-variate modulo M function, if there exist a constant $C \in \mathbb{Z}$ and weights $w_1, \dots, w_k \in \mathbb{Z}$ such that

$$g(x^1, \dots, x^k) = 1 \Leftrightarrow \sum_{i=1}^{k} w_i \cdot |x^i| \equiv C \mod M.$$

The set of k-variate modulo M functions on $X_{k,n}$ is denoted by $\mathbb{M}_{k,n}^M$.

A function $f \in B_n$ can be decomposed into m functions in a class $\mathcal{C}$ of functions on n Boolean variables, if there exist a formula F on m variables and $f_1, \dots, f_m \in \mathcal{C}$ such that $f = F(f_1, \dots, f_m)$. Any function decomposable into a constant number of threshold and modulo functions with respect to constant weights has a small OBDD size. More precisely, the following lemma has been shown by [21].

Lemma 2. *Let $f = F(f_1, \dots, f_m)$ for a formula F of size s and $f_1, \dots, f_m \in \mathbb{T}_{k,n}^w \cup \mathbb{M}_{k,n}^M$. The $\pi_{k,n}$-OBDD size of f is $\mathcal{O}(L^s kn)$, where $\pi_{k,n} = (x_0^1, x_0^2, \dots, x_0^k, x_1^1, \dots, x_1^k, \dots, x_{n-1}^k)$ and $L = \max(4kw + 5, M)$.*

Finally, the following result proved by [17] shows that a constant number of operations on functions representable by OBDDs of small width can be done efficiently.

Lemma 3. *Let $f \in B_{kn}$ be defined on variable vectors $x^1, \dots, x^k \in \{0,1\}^n$ and G_f be a complete* OBDD *representing f with respect to the variable ordering*

$$\pi_{k,n} = (x_0^1, x_0^2, \dots, x_0^k, x_1^1, \dots, x_1^k, \dots, x_{n-1}^k).$$

*Furthermore, let S be a sequence of a constant number of functional operations and quantifications over variable vectors x^i, $1 \le i \le k$, applied on f, functions in $\mathbb{T}_{k,n}^w$, where w is independent of k and n, and intermediate functions generated by the current prefix of S. The width of the minimal complete $\pi_{k,n}$-*OBDD *for each function generated by S only depends on the width w_f of G_f.*

3 The Maximal Matching Algorithm

In this section we present an implicit algorithm for the maximal bipartite matching problem and prove Theorem 1. The algorithm can easily be extended for general graphs. The idea is

to use the parallel algorithm for the computation of maximal matchings on explicitly defined input graphs presented by [13] and to adapt this algorithm to the implicit setting. In the parallel setting more or less only a high-level description of the algorithm is given. Moreover, we have to add more ideas because we cannot access efficiently single vertices or edges in the implicit setting.

The algorithm `findMaximalBipartiteMatching` is simple. Step-by-step a current matching is enlarged by computing a matching in the subgraph of $G = (V, E)$ that consists only of the edges that are not incident to the current matching. The key idea is an algorithm `match` that computes a matching M', $M' \subseteq E$, adjacent to at least a fraction of $1/6$ of the edges in the input graph for `match`. After removing these edges from the input graph the procedure is repeated. Therefore, after $\mathcal{O}(\log |V|)$ iterations the considered subgraph is empty and the current matching is obviously a maximal matching in G.

Algorithm 1 `findMaximalBipartiteMatching`

Input: $\chi_E(x, y)$

(1) ▷ `Initialize. Start with the empty matching.`
$M(x, y) \leftarrow 0$
(2) **while** $\chi_E(x, y) \neq 0$ **do**
(3) ▷ `Compute a matching` M'.
$M'(x, y) \leftarrow \texttt{match}(\chi_E(x, y))$
(4) ▷ `Delete the edges incident to a matched vertex with respect to` M'.
$\texttt{INCNODE}(x) \leftarrow (\exists y)(M'(x, y))$
$\chi_E(x, y) \leftarrow \chi_E(x, y) \wedge \overline{\texttt{INCNODE}(x)} \wedge \overline{\texttt{INCNODE}(y)}$
(5) ▷ `Add the edges from` M' `to` M.
$M(x, y) \leftarrow M(x, y) \vee M'(x, y)$
(6) **return** $M(x, y)$

The algorithm `match` makes use of another algorithm `halve` that halves approximately the degree of each vertex in a bipartite graph. The idea is to compute an Euler partition of the input graph such that the graph is decomposed into edge-disjoint paths. Each vertex of odd (even) degree is the endpoint of exactly 1 (0) open path. By two-coloring the edges on each path in the Euler partition and deleting all edges of one color, the degree of each vertex in the input graph is approximately halved. Here, we use the fact that bipartite graphs have no cycles of odd length. Therefore, for each path, where a vertex v is not an endpoint, and for each cycle, the number of edges incident to v colored by one of the two colors is equal to the number of edges colored by the other one. In fact in the algorithm `halve` we only use the color red and delete all red edges after the coloring. A precondition of the parallel algorithm `halve` is that for each vertex its incident edges have been paired ([13]). Here, we present a new algorithm called `calculatePairing` which computes implicitly a pairing of the edges with $\mathcal{O}(\log^2 |V|)$ functional operations. This algorithm together with the degree reduction procedure in `halve` can possibly be used as building blocks for the solution of other combinatorial problems in the implicit setting. We assume that there is for each vertex an ordering on its incident edges given by a priority function (see Section 2). In the first step of the algorithm `calculatePairing` for each vertex the neighborhood of its incident edges is determined. Almost all edges have two neighbors with respect to one of its endpoints (all but the first and the last one). In order to compute a symmetric pairing every other edge

incident to the same vertex is colored red. This is realized by an indicator function called RED. Afterwards, two neighboring edges $(x, y), (x, z)$ are paired iff (x, y) is red, i.e., $\texttt{RED}(x, y) = 1$, and (x, y) has a higher priority than (x, z), i.e., $\Pi_{\prec}(x, y, z) = 1$. Therefore, each edge has at most one chosen neighbor with respect to one of its endpoints and at most one of the edges incident to the same vertex is not paired. The output of `calculatePairing` is a function which depends on three vertex arguments x, y, and z and whose function value is 1 iff y and z are two vertices adjacent to x which have been paired. Therefore, the function is symmetric in the second and third argument. For the computation of the pairing we determine for each edge its position with respect to all edges incident to the same vertex according to a priority function. This procedure is similar to the well-known list ranking algorithm: given a linked list for each member of the list the number of its position in the list has to be calculated (for a nice introduction into design and analysis of parallel algorithms see, e.g., [12]). In our case we are only interested in whether the number of the position of an edge according to a priority function is odd or even.

Algorithm 2 `calculatePairing`

Input: $\chi_E(x, y)$

(1) ▷ Determine the neighborhood of the edges.
$\texttt{ORDER}(x, y, z) \leftarrow \chi_E(x, y) \wedge \chi_E(x, z) \wedge \Pi_{\prec}(x, y, z) \wedge \overline{(\exists \xi)(\chi_E(x, \xi) \wedge \Pi_{\prec}(x, y, \xi) \wedge \Pi_{\prec}(x, \xi, z))}$

(2) ▷ Compute the distance between edges incident to the same vertex using iterative squaring.
$\texttt{DIST}_0(x, y, z) \leftarrow \texttt{ORDER}(x, y, z)$
for $i = 1, 2, ..., \log |V|$ **do**
  $\texttt{DIST}_i(x, y, z) \leftarrow (\exists \xi)(\texttt{DIST}_{i-1}(x, y, \xi) \wedge \texttt{DIST}_{i-1}(x, \xi, z))$

(3) ▷ Color for each vertex its incident edges alternately.
$\texttt{RED}(x, y) \leftarrow \chi_E(x, y) \wedge \overline{(\exists \xi)(\chi_E(x, \xi) \wedge \Pi_{\prec}(x, \xi, y))}$
for $i = 1, 2, ..., \log |V|$ **do**
  $\texttt{RED}(x, y) \leftarrow \texttt{RED}(x, y) \vee (\exists \xi)(\texttt{RED}(x, \xi) \wedge \texttt{DIST}_i(x, \xi, y))$

(4) ▷ Select only edge pairs $((x, y), (x, z))$, where the first edge is red.
return $(\texttt{ORDER}(x, y, z) \wedge \texttt{RED}(x, y)) \vee (\texttt{ORDER}(x, z, y) \wedge \texttt{RED}(x, z))$

Lemma 4. *The algorithm* `calculatePairing` *computes for all vertices a pairing of its incident edges respectively with* $\mathcal{O}(\log^2 |V|)$ *functional operations.*

Proof. The correctness of `calculatePairing` follows from the following observations: the function $\texttt{ORDER}(x, y, z)$ computes the output 1 iff y and z are adjacent to the vertex x, the edge (x, y) is smaller than the edge (x, z) according to the chosen priority function, and there is no edge between (x, y) and (x, z) (in the order defined by the priority function). In step 2 the function $\texttt{DIST}_i(x, y, z)$ computes 1 iff the distance, i.e., the number of edges between the edge (x, y) and (x, z) with respect to the priority function, is $2^i - 1$. Afterwards for each vertex the first of its incident edges according to the priority function is colored red and then all edges which have an odd distance to the first one are also colored red. Now, the output of `calculatePairing` is a function on three vertex arguments x, y and z, where the value is 1 iff the edges (x, y) and (x, z) are neighbored and the first one with respect to the priority function is red.

The most time consuming steps during `calculatePairing` are (2) and (3), where the position of the edges and the coloring of the incident edges are calculated. Traversing the

incident edges of a vertex needs $\mathcal{O}(\log |V|)$ iterations each using $\mathcal{O}(\log |V|)$ operations for the quantification of the $\mathcal{O}(\log |V|)$ variables. □

Algorithm 3 `halve`

Input: $\chi_E(x, y)$

(1) ▷ `Compute the successor relation.`
  $\mathtt{PAIRING}(x, y, z) \leftarrow$ `calculatePairing` $\chi_E(x, y)$
  $\mathtt{SUCC}(x, y, z) \leftarrow \mathtt{PAIRING}(y, x, z)$

(2) ▷ `Compute distance and reachability relations on the directed edges defined by the successor relation.`
  $i \leftarrow 0$
  $\mathtt{DIST}_0(v, w, x, y) \leftarrow EQ(w, x) \wedge \mathtt{SUCC}(v, w, y)$
  $\mathtt{REACHABLE}(v, w, x, y) \leftarrow (EQ(v, x) \wedge EQ(w, y)) \vee (EQ(w, x) \wedge \mathtt{SUCC}(v, w, y))$
  repeat
    $i \leftarrow i + 1$
    $\mathtt{REACHABLE}'(v, w, x, y) \leftarrow \mathtt{REACHABLE}(v, w, x, y)$
    $\mathtt{REACHABLE}(v, w, x, y) \leftarrow \mathtt{REACHABLE}(v, w, x, y) \vee$
      $(\exists \xi, \theta)(\mathtt{REACHABLE}(v, w, \xi, \theta) \wedge \mathtt{REACHABLE}(\xi, \theta, x, y))$
    $\mathtt{DIST}_i(v, w, x, y) \leftarrow (\exists \xi, \theta)(\mathtt{DIST}_{i-1}(v, w, \xi, \theta) \wedge \mathtt{DIST}_{i-1}(\xi, \theta, x, y))$
  until $\mathtt{REACHABLE}'(v, w, x, y) = \mathtt{REACHABLE}(v, w, x, y)$

(3) ▷ `On each path, color an appropriate edge red.`
  $\mathtt{RED}(x, y) \leftarrow \chi_E(x, y) \wedge (\forall \xi, \theta)(\overline{\mathtt{REACHABLE}(\xi, \theta, x, y)} \vee EQ(\xi, x) \vee \Pi_{\prec}(x, x, \xi)) \wedge$
   $(\forall \theta, \xi)(\overline{\mathtt{REACHABLE}(\theta, \xi, y, x)} \vee EQ(\theta, x) \vee \Pi_{\prec}(x, x, \theta)) \wedge$
   $(\forall \xi)((\overline{\mathtt{REACHABLE}(x, y, \xi, x)} \wedge \overline{\mathtt{REACHABLE}(\xi, x, x, y)}) \vee \Pi_{\prec}(x, y, \xi))$
  $\mathtt{RED}(x, y) \leftarrow \mathtt{RED}(x, y) \vee \mathtt{RED}(y, x)$

(4) ▷ `Color the edges alternately.`
  for $j = 1, 2, ..., i$ **do**
    $\mathtt{RED}(x, y) \leftarrow \mathtt{RED}(x, y) \vee (\exists \xi, \theta)(\mathtt{RED}(\xi, \theta) \wedge \mathtt{DIST}_j(\xi, \theta, x, y))$

(5) ▷ `Delete the red edges.`
  return $\chi_E(x, y) \wedge \overline{\mathtt{RED}(x, y)} \wedge \overline{\mathtt{RED}(y, x)}$

The pairing computed by `calculatePairing` is symmetric and it is used by the algorithm `halve` to define (directed) paths in the (undirected) input graph. An edge (y, z) is a successor edge of an edge (x, y) and $\mathtt{SUCC}(x, y, z) = 1$ iff the edges (y, z) and (y, x) are paired according to `calculatePairing` (see Figure 1 for an illustration of the calculated pairing and the successor relation SUCC).

Using this successor relation `SUCC` the undirected input graph is decomposed into directed edge-disjoint paths. Since the pairing is symmetric, (y, x) is also a successor of (z, y). Therefore, for each directed path from a vertex u' to a vertex u'' defined by the successor relation `SUCC`, there exists also a directed path from u'' to u'. This property is important in order to guarantee that a coloring of the directed edges can be used for an appropriate coloring of the undirected edges in the input graph. For each directed path in the decomposition every other edge is colored red and a directed edge (u, v) is red iff the directed edge (v, u) is red. Therefore, for each pair of (undirected) edges computed by `calculatePairing` exactly one edge is red and by deleting the red edges the degree of each vertex is approximately halved. A crucial step, which is new in the implicit setting, is the choice of the first edges that are colored red on a directed path, because all directed paths are investigated simultaneously,

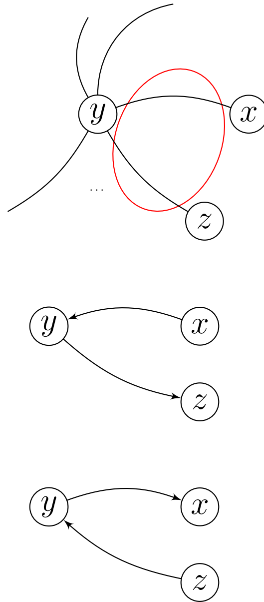

Fig. 1. *The relation between the calculated pairing and the successor relation SUCC: the vertices x and z are adjacent to y and paired according to* `calculate pairing`*, therefore, the directed edge (y, z) is the successor edge of the directed edge (x, y) and (y, x) the successor edge of (z, y).*

i.e., for each directed path from a vertex s to a vertex t the directed path from t to s is considered at the same time. We have to avoid the situation that two edges (x, y) and (x, z), where (x, y) is a directed edge on a directed path from a vertex s to a vertex t and (x, z) a directed edge on the path from t to s, are colored red at the same time, because otherwise all edges from s to t and from t to s would be red after the coloring procedure. (See Figure 2 for a situation that has to be avoided.) For this reason we ensure that in the beginning we color either directed edges on the path from s to t or on the path from t to s.

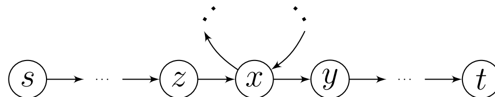

Fig. 2. *The following situation has to be avoided: if (x, z) and (x, y) are both colored red in the beginning, the directed edges (z, x) and (x, y) which are on the same directed path are both colored red in the end of step (3) in* `halve`*. The coloring procedure in step (4) would lead to a coloring of all edges on the directed path from s to t (and vice versa), because the distance between an incoming and an outgoing edge in a bipartite graph is odd. (Note, that the edge (z, x) can also be behind the edge (x, y) on the directed path from s to t.)*

For this reason an edge relation called `Reachable` is used, where `REACHABLE`(v, w, x, y) is 1 iff there exists a directed path from the edge (v, w) to the edge (x, y) defined by the successor

relation SUCC. Due to the symmetry of SUCC the relation REACHABLE is also symmetric in the following way: iff $\texttt{REACHABLE}(v, w, x, y) = 1$ then $\texttt{REACHABLE}(y, x, w, v) = 1$. Therefore, using REACHABLE it is also possible to determine the predecessors of a directed edge. Now, the first red edges on a directed path are the edges with the highest priority: for each directed path the smallest vertex v on the path according to a priority function together with a successor u is chosen if there exists no predecessor of v which has a higher priority than u according to the chosen priority function. This procedure ensures that either for a directed path starting from a vertex s and ending in a vertex t an edge (v, u) is chosen or for the directed path from t to s. Afterwards an edge (u, v) is colored red iff the edge (v, u) is red. Next, each edge on a directed path for which the distance to one of the first red edges on the path is odd is also colored red and all red edges are deleted from the input graph. Note, that it is possible for a directed path that more than one edge is chosen in the beginning but these edges have the same starting point, therefore the distance between these edges is even because the input graph is bipartite such that no problem occurs.

Lemma 5. *Let $d(v)$ and $d'(v)$ denote the degree of a vertex v in the graph given by $\chi_E(x, y)$ before and after running procedure* `halve` *on $\chi_E(x, y)$. Then $d'(v) \in \{\lfloor d(v)/2 \rfloor, \lceil d(v)/2 \rceil\}$. The algorithm* `halve` *uses $\mathcal{O}(\log^2 |V|)$ functional operations.*

Proof. For the number of functional operations step (2) and step (4) are the most expensive ones. The graph is traversed via iterative squaring in the second step. Since the length of a path is $\mathcal{O}(|E|)$, the number of iterations is $\mathcal{O}(\log |E|) = \mathcal{O}(\log |V|)$, and the quantification over the Boolean variables that encode an edge can also be done using $\mathcal{O}(\log |V|)$ operations ($\mathcal{O}(1)$ functional operations for the quantification of each variable). The number of functional operations in step (4) can be calculated in a similar way.

For the correctness of the algorithm `halve` step (3) is the most interesting one. The directed paths according to the successor relation SUCC are edge-disjoint but not vertex disjoint. In step (3) for each directed path according to the successor relation at least one edge is carefully chosen and colored red. The first condition ensures that only edges that belong to the input graph can be chosen. The second and third requirements guarantee that a first red edge is incident to the vertex x with the highest priority on the path. The fourth condition ensures that two arbitrary edges incident to x that are on the same directed path and have an even distance are not both colored red. This condition is sufficient, we do not have to choose only the neighbor of x which has the highest priority because we color the edges afterwards alternately and edges whose distance is odd get the same color anyway. Together with our considerations above we are done. ◻

The idea for the correctness of `match` is that a (directed) subgraph $P(x, y)$ of the input $\chi_E(x, y)$ is computed for which each vertex has indegree and outdegree at most 1. It consists only of vertex-disjoint open simple paths and trivial cycles. Therefore, the subgraph can be seen as the union of two matchings. By choosing the matching which is the larger one we are done. For this reason we color each path in $P(x, y)$ alternately and we remove the edges that are not red. Since the paths are vertex-disjoint the coloring is easier than the coloring of the edges in the algorithm `halve`. In fact we color the vertices and not the edges. We choose for each directed simple path the first vertex and color afterwards all vertices for which the distance to the first one is even. Then we choose the (directed) edges (x, y) iff the vertex x is red. Finally, we traverse the computed directed subgraph into the corresponding undirected one.

The directed subgraph P is computed in the following way. In each iteration the vertices with degree 1 are determined. For each vertex x adjacent to vertices with degree 1 in the

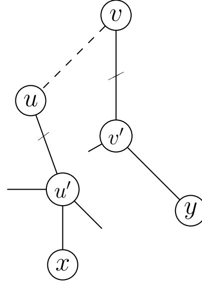

Fig. 3. *A simplified situation for an edge (u, v) not incident to the computed matching in* `match`*: the edge (u, v) is canceled during* `halve` *and the last edges incident to u and v are deleted in step* (5) *of* `match`*.*

remaining graph one of these vertices y is chosen according to a priority function and (x, y) is added to P. Afterwards all edges incident to vertices with degree 1 are deleted and the degree of all vertices is (approximately) halved. Note, that during the computation an edge (x, y) in P can be eliminated later on if x gets another partner that has a higher priority than y. At any time each vertex x has at most one partner.

It can be shown that at the beginning of step (8) in `match` at least $1/3$ of the input edges are incident to edges defined via $P(x, y)$. The intuition is the following one. An edge (u, v) is not incident to the computed matching iff the edge (u, v) is deleted during the algorithm `halve` and the last edges (u, u') and (v, v') incident to u and v during the **while** loop are eliminated in step (5) of `match` because u' and v' are at the same time adjacent to vertices x and y which have degree 1 and are chosen as partners in the respective iteration of the **while** loop. (See Figure 3 for a simplified illustration of the situation.) As a consequence we can conclude that the degree of u' and v' is (approximately) at least twice the degree of u and v in the input graph because Lemma 5 ensures that the degree of each node is (almost) regularly halved in each iteration. Therefore the output of the algorithm `match` is a matching incident to at least $1/6$ of the input edges.

Lemma 6. *The algorithm* `match` *implicitly computes a matching in an implicitly defined input graph $G = (V, E)$ incident to at least $1/6$ of the edges in E. It needs $\mathcal{O}(\log^3 |V|)$ functional operations.*

Proof. There are $\mathcal{O}(\log |V|)$ iterations of the **while** loop, each of them costs $\mathcal{O}(\log^2 |V|)$ functional operations. The algorithm `halve` is the dominating step during the **while** loop of `match`. Therefore, $\mathcal{O}(\log^3 |V|)$ functional operations are sufficient. The correctness follows from our considerations above. (See also [13].) □

Altogether, we have proved that the algorithm `findMaximalBipartiteMatching` uses $\mathcal{O}(\log^4 |V|)$ functional operations for the computation of a maximal matching in an implicitly defined input graph $G = (V, E)$. Adapting the ideas presented for the decomposition of general graphs into a logarithmic number of bipartite subgraphs presented by [13], our algorithm can be similarly generalized with an additional factor of a logarithmic number of functional operations.

Algorithm 4 `match`

Input: $\chi_E(x,y)$

(1) ▷ `Initialize.`
$\chi_{E'}(x,y) \leftarrow \chi_E(x,y)$; $P(x,y) \leftarrow 0$
(2) **while** $\chi_{E'}(x,y) \neq 0$ **do**
(3) ▷ `Determine the vertices of degree at least` 2.
$\text{TwoOrMoreNeighbors}(x) \leftarrow (\exists y,z)(NEQ(y,z) \wedge \chi_{E'}(x,y) \wedge \chi_{E'}(x,z))$
(4) ▷ `Set` $P(x,y)=1$ `iff` y `has only one neighbor and is the partner of` x.
$Q(x,y) \leftarrow \chi_{E'}(x,y) \wedge \overline{\text{TwoOrMoreNeighbors}(y)}$
$Q'(x,y) \leftarrow Q(x,y) \wedge \overline{(\exists z)(Q(x,z) \wedge \Pi_{\prec}(x,z,y))}$
$P(x,y) \leftarrow (P(x,y) \wedge \overline{(\exists z)(Q'(x,z))}) \vee Q'(x,y)$
(5) ▷ `Delete edges incident to vertices of degree 1.`
$\chi_{E'}(x,y) \leftarrow \chi_{E'}(x,y) \wedge \text{TwoOrMoreNeighbors}(x) \wedge \text{TwoOrMoreNeighbors}(y)$
(6) ▷ `Halve (approximately) the degree of each vertex.`
$\chi_{E'}(x,y) \leftarrow$ `halve`$(\chi_{E'}(x,y))$
(7) ▷ `Add trivial cycles to the computed matching.`
$M_1(x,y) \leftarrow P(x,y) \wedge P(y,x)$
(8) ▷ `Color the vertices in the graph given by` $P(x,y)$ `alternately and choose an edge` (x,y) `iff` x `is red.`
$\mathtt{RED}(x) \leftarrow (\forall \xi)\overline{(P(\xi,x))}$; $\mathtt{DIST}_0(x,y) \leftarrow P(x,y)$
for $i = 1,2,...,\log|V|$ **do**
$\mathtt{DIST}_i(x,y) \leftarrow (\exists \xi)(\mathtt{DIST}_{i-1}(x,\xi) \wedge \mathtt{DIST}_{i-1}(\xi,y))$
$\mathtt{RED}(x) \leftarrow \mathtt{RED}(x) \vee (\exists \xi)(\mathtt{RED}(\xi) \wedge \mathtt{DIST}_i(\xi,x))$
$M_2(x,y) \leftarrow P(x,y) \wedge \mathtt{RED}(x)$
(9) **return** $M_1(x,y) \vee M_2(x,y) \vee M_2(y,x)$

4 Analytical Evaluation of the Maximal Matching Algorithm

Most operations on OBDDs require time and space proportional to the sizes of the operands if the corresponding OBDDs are ordered with respect to the same variable ordering. As a result each single operation is efficient but a sequence of $\mathcal{O}(\log|V|)$ functional operations may lead to OBDDs of exponential size (with respect to the number of Boolean variables). Therefore, the number of functional operations is only a rough measure for the complexity of an OBDD-based algorithm. In the rest of this section we present analyses for the overall running time and the space usage of general graphs and of a very structured graph class.

Assuming that all participating OBDDs are reduced and using the fact that the number of 1-inputs of a Boolean function f can be approximately used as an upper bound for the size of a reduced OBDD representing f (see Section 2), a careful worst-case analysis gets the result that maximal matchings in arbitrary graphs can be computed in time and space $\tilde{\mathcal{O}}(|E|^6)$, where an algorithm uses time (space) $\tilde{\mathcal{O}}(f(n))$ if it needs time (space) $\mathcal{O}(f(n)\log^k n)$ for some constant k. Here, the overall worst-case running time and space requirement has been determined, i.e., it has been assumed that all OBDDs during the computation are as large as possible. The most dominating step is the computation of the function `REACHABLE` in the algorithm `halve` for which the function value for two directed edges (v,w) and (x,y) is 1 iff there exists a directed path from (v,w) to (x,y). The number of 1-inputs for `REACHABLE` is $\mathcal{O}(|E|^2)$, therefore a synthesis on `REACHABLE` functions can be done in time and space $\tilde{\mathcal{O}}(|E|^4)$. The result is a function that depends on three edge arguments, therefore the size of an OBDD is $\tilde{\mathcal{O}}(|E|^3)$. Now, a quantification over a variable of a function f cannot lead to a

function with more 1-inputs than f. Therefore, the operation can be done in time and space $\tilde{\mathcal{O}}(|E|^6)$ and the resulting OBDD has size $\tilde{\mathcal{O}}(|E|^3)$. As a consequence all quantification steps together during one iteration of the `repeat` loop can be done in time and space $\tilde{\mathcal{O}}(|E|^6)$.

The worst-case bound is pretty bad in comparison to a simple greedy strategy that leads to a linear time algorithm in the explicit setting. One reason may be that the known methods for the analysis are rough. Another one, that our worst-case analysis is even independent of the chosen variable ordering. Nevertheless, implicit algorithms are a heuristic method in case that explicit algorithms could not be applied because the input graphs are too large but very structured. In order to demonstrate that there exist graphs for which our implicit algorithm is more efficient than explicit algorithms, grid graphs have been investigated. The intention has been to analyze our heuristic method working on OBDDs for a very structured graph class and to prove that our maximal matching algorithm is efficient although it does not use any explicit information about the structure of the inputs. The undirected grid graph on $N = 2^n$ vertices consists of the vertex set $V = \{0,1\}^{n/2} \times \{0,1\}^{n/2}$ and the edge set E, where $((x^1, y^1), (x^2, y^2)) \in E$ iff $|x^1| = |x^2|$ and $||y^1| - |y^2|| = 1$ or $||x^1| - |x^2|| = 1$ and $|y^1| = |y^2|$. We assume that $N^{1/2}$ is a power of 2 since it has no bearing on the essence of our results. (See Figure 4 for an example of a grid graph.) Directed grid graphs have already been chosen as graph class in the investigation of the behavior of maximum flow algorithms in 0-1 networks and of a topological sorting algorithm in the implicit setting ([15] and [21]). In the following we assume that all participating OBDDs are complete reduced OBDDs and the input is represented with respect to an interleaved variable ordering where the variables are ordered with increasing significance.

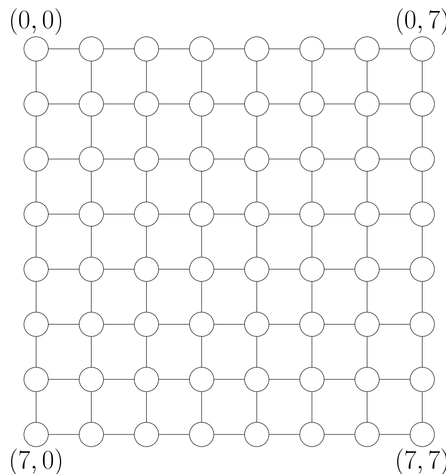

Fig. 4. *The undirected* 8×8 *grid graph.*

The encoding of a vertex in the grid graph consists of two parts: the coordinate of the corresponding row and the coordinate of the corresponding column in the grid graph. It is not difficult to see that the characteristic function of the grid graph is decomposable into a constant number of 4-variate threshold functions, where the maximal absolute weight is independent of n. Therefore, using the variable ordering $\pi_{4,n/2} = (x_0^1, x_0^2, y_0^1, y_0^2, x_1^1, \ldots, y_{n/2-1}^2)$ and applying Lemma 2 it can be shown that the $\pi_{4,n/2}$-OBDD size for the characteristic

function of the $2^{n/2} \times 2^{n/2}$ grid graph is linear with respect to the number of Boolean variables which means logarithmic in the number of vertices of the input graph. Alternatively we can use Theorem 2 in order to prove that the width of the $\pi_{4,n/2}$-OBDD for the grid graph is 6.

The analysis of the maximal matching algorithm on grid graphs is not very complicated but kind of tedious. In summary it has been shown that a maximal matching for grid graphs can be computed in $\mathcal{O}(\log^3 |V| \log\log |V|)$ time and $\mathcal{O}(\log^2 |V|)$ space (a detailed analysis has been done by [14] (Section 8.3.3)). The ideas are the following ones. Obviously, the undirected grid graph is bipartite. Since the degree of a vertex in the grid graph is at most 4 and the input graph is very structured, only a constant number of runs of the algorithm `match` in `findMaximalBipartiteMatching` is necessary. Moreover, a constant number of iterations of the `while` loop in `match` is sufficient. As a consequence there is only a constant number of pass of the algorithm `halve` and therefore of the algorithm `calculatePairing`. The most important observation is that all occuring OBDDs for intermediate results have constant width and therefore size $\mathcal{O}(\log |V|)$. Using Lemma 3 this can easily be shown for all steps without loops assuming that the corresponding input OBDDs are of constant width. Now, for step (2) and step (3) of the algorithm `calculatePairing` we observe that the constant degree of the input graph implies only a constant number of runs which lead to different resulting OBDDs. Hence, all intermediate OBDDs have constant width. In step (8) in `match` all OBDDs for the functions DIST_i are small because the function $P(x, y)$ has a very simple structure. Finally, for step (2) and (4) in `halve` we have to ensure that the OBDDs for the functions $\text{REACHABLE}(v, w, x, y)$ and $\text{DIST}_i(v, w, x, y)$ which compute the output 1 if the edge (x, y) is reachable via a directed path from the edge (v, w) or the edge (x, y) is reachable via a directed path of length 2^i from the edge (v, w) respectively have small OBDD width. This can be guaranteed because of the simple structure of the grid graph and Lemma 3. As all intermediate OBDDs have constant width and there are at most $\mathcal{O}(\log |V|)$ different functions which have to be stored at the same time, the space requirement is $\mathcal{O}(\log^2 |V|)$. The number of pass of each loop is constant and each loop can be done by $\mathcal{O}(\log^2 |V|)$ functional operations. Since the OBDDs involved are of constant width the running time altogether is $\mathcal{O}(\log^2 |V| \cdot \log |V| \log\log |V|) = \mathcal{O}(\log^3 |V| \log\log |V|)$.

Concluding Remarks

We have shown that maximal matchings can be computed with a polylogarithmic number of functional operations in the implicit setting and that there exists a graph class for which even the overall running time is $\mathcal{O}(\log^3 |V| \log\log |V|)$ and the space usage is $\mathcal{O}(\log^2 |V|)$, where V is the set of vertices of the input graph. Moreover, our maximal matching algorithm seems to be simple enough to be useful in practical applications. One direction for future work is to implement the algorithm and to perform empirical experiments to determine its practical value. It would be interesting to investigate how the performance of the maximal matching algorithm depends on the chosen priority function. Here, we have used a very simple one. The maximal number of Boolean variables on which a function in the maximal matching algorithm depends dominates the overall worst-case bounds for the running time and the space usage. Therefore, another open question is whether we can reduce this number without increasing significantly the number of functional operations. Experimental evaluation of different maximal matching algorithms might be revealing.

References

1. Bollig, B. (2010), Exponential space complexity for OBDD-based reachability analysis, Information Processing Letters **110**, 924-927.
2. Bollig, B. (2010), Exponential space complexity for symbolic maximum flow algorithms in 0-1 networks, *in* Proc. of MFCS, LNCS 6281, pp. 186–197.
3. Bollig, B. (2010) , On symbolic representations of maximum matchings and (un)directed graphs, *in* Proc. of TCS IFIP AICT 323, pp. 263–300.
4. Bollig, B., Löbbing, M. and Wegener, I. (1996), On the effect of local changes in the variable ordering of ordered decision diagrams, Information Processing Letters **59** , 233–239.
5. Breitbart, Y. Hunt III, H. B., and Rosenkrantz, D.J. (1995), On the size of binary decision diagrams representing Boolean functions, Theoretical Computer Science **145**, 45–69.
6. Bryant, R. E. (1986) , Graph-based algorithms for Boolean function manipulation, IEEE Trans. on Computers **35**, 677–691.
7. Feigenbaum, J., Kannan, S., Vardi, M.V. Viswanathan, M. (1998), Complexity of problems on graphs represented as OBDDs, *in* Proc. of STACS, LNCS 1373, pp. 216–226.
8. Gentilini, R., Piazza, C. Policriti, A. (2003), Computing strongly connected components in a linear number of symbolic steps, *in* Proc. of SODA, ACM Press, pp. 573–582.
9. Gentilini, R., Piazza, C. Policriti, A. (2008), Symbolic graphs: linear solutions to connectivity related problems, Algorithmica **50**, 120–158.
10. Goldberg, A.V., Plotkin, S.K. Vaidya, P.M. (1993), Sublinear time parallel algorithms for matching and related problems, Journal of Algorithms **14**(2), 180–213.
11. Hachtel, G.D. and Somenzi, F. (1997), A symbolic algorithm for maximum flow in $0-1$ networks, Formal Methods in System Design **10**: 207–219.
12. Jájá, J. (1992), *An introduction to parallel algorithms*, Addison-Wesley Publishing Company.
13. Kelsen, P. (1994), An optimal parallel algorithm for maximal matching, Information Processing Letters **52**(4): 223–228.
14. Pröger, T. (2010), Implizite Graphprobleme für Matchingprobleme, TU Dortmund, Fakultät für Informatik, Diploma thesis, in German.
15. Sawitzki, D. (2004), Implicit flow maximization by iterative squaring, *in* Proc. of SOFSEM, LNCS 2932, pp. 301–313.
16. Sawitzki, D. (2006), Exponential lower bounds on the space complexity of OBDD-based graph algorithms, *in* Proc. of LATIN, LNCS 3887, pp. 781–792.
17. Sawitzki, D. (2006), The complexity of problems on implicitly represented inputs, *in* Proc. of SOFSEM, LNCS 3831, pp. 471–482.
18. Sawitzki, D. (2007), Implicit simulation of FNC algorithms, ECCC Report TR07-028.
19. Sieling, D. and Wegener, I. (1993), NC-algorithms for operations on binary decision diagrams, Parallel Processing Letters **48**, 139–144.
20. Wegener, I. (2000), *Branching Programs and Binary Decision Diagrams - Theory and Applications*, SIAM Monographs on Discrete Mathematics and Applications.
21. Woelfel, P. (2006), Symbolic topological sorting with OBDDs, Journal of Discrete Algorithms **4**(1), 51-71.