B

# Backtracking Based *k*-SAT Algorithms

**2005; Paturi, Pudlák, Saks, Zane**

RAMAMOHAN PATURI[1], PAVEL PUDLÁK[2], MICHAEL SAKS[3], FRANCIS ZANE[4]
[1] Department of Computer Science and Engineering, University of California at San Diego, San Diego, CA, USA
[2] Mathematical Institute, Academy of Science of the Czech Republic, Prague, Czech Republic
[3] Department of Mathematics, Rutgers, State University of New Jersey, Piscataway, NJ, USA
[4] Bell Laboratories, Lucent Technologies, Murray Hill, NJ, USA

## Problem Definition

Determination of the complexity of *k*-CNF satisfiability is a celebrated open problem: given a Boolean formula in conjunctive normal form with at most *k* literals per clause, find an assignment to the variables that satisfies each of the clauses or declare none exists. It is well-known that the decision problem of *k*–CNF satisfiability is NP-complete for $k \geq 3$. This entry is concerned with algorithms that significantly improve the worst case running time of the naive exhaustive search algorithm, which is $\text{poly}(n)2^n$ for a formula on $n$ variables. Monien and Speckenmeyer [8] gave the first real improvement by giving a simple algorithm whose running time is $O(2^{(1-\varepsilon_k)n})$, with $\varepsilon_k > 0$ for all $k$. In a sequence of results [1,3,5,6,7,9,10,11,12], algorithms with increasingly better running times (larger values of $\varepsilon_k$) have been proposed and analyzed.

These algorithms usually follow one of two lines of attack to find a satisfying solution. Backtrack search algorithms make up one class of algorithms. These algorithms were originally proposed by Davis, Logemann and Loveland [4] and are sometimes called Davis–Putnam procedures. Such algorithms search for a satisfying assignment by assigning values to variables one by one (in some order), backtracking if a clause is made false. The other class of algorithms is based on local searches (the first guaranteed performance results were obtained by Schöning [12]). One starts with a randomly (or strategically) selected assignment, and searches locally for a satisfying assignment guided by the unsatisfied clauses.

This entry presents **ResolveSat**, a randomized algorithm for *k*-CNF satisfiability which achieves some of the best known upper bounds. **ResolveSat** is based on an earlier algorithm of Paturi, Pudlák and Zane [10], which is essentially a backtrack search algorithm where the variables are examined in a randomly chosen order. An analysis of the algorithm is based on the observation that as long as the formula has a satisfying assignment which is isolated from other satisfying assignments, a third of the variables are expected to occur as unit clauses as the variables are assigned in a random order. Thus, the algorithm needs to correctly guess the values of at most 2/3 of the variables. This analysis is extended to the general case by observing that there either exists an isolated satisfying assignment, or there are many solutions so the probability of guessing one correctly is sufficiently high.

**ResolveSat** combines these ideas with resolution to obtain significantly improved bounds [9]. In fact, **ResolveSat** obtains the best known upper bounds for *k*-CNF satisfiability for all $k \geq 5$. For $k = 3$ and 4, Iwama and Takami [6] obtained the best known upper bound with their randomized algorithm which combines the ideas from Schöning's local search algorithm and **ResolveSat**. Furthermore, for the promise problem of unique *k*-CNF satisfiability whose instances are conjectured to be among the hardest instances of *k*-CNF satisfiability [2], **ResolveSat** holds the best record for all $k \geq 3$. Bounds obtained by **ResolveSat** for unique *k*-SAT and *k*-SAT, for $k = 3, 4, 5, 6$ are shown in Table 1. Here, these bounds are compared with those of of Schöning [12], subsequently improved results based on local search [1,5,11], and the most recent improvements due to Iwama and Takami [6]. The upper bounds obtained by these algorithms are expressed in the form $2^{cn-o(n)}$ and the numbers in the table represent the exponent $c$. This comparison focuses only on the best bounds irrespective of the type of the algorithm (randomized versus deterministic).

**Notation** In this entry, a CNF boolean formula $F(x_1, x_2, \ldots, x_n)$ is viewed as both a boolean function and a set of clauses. A boolean formula $F$ is a $k$-CNF if all the clauses have size at most $k$. For a clause $C$, write $var(C)$ for the set of variables appearing in $C$. If $v \in var(C)$, the *orientation* of $v$ is positive if the literal $v$ is in $C$ and is negative if $\bar{v}$ is in $C$. Recall that if $F$ is a CNF boolean formula on variables $(x_1, x_2, \ldots, x_n)$ and $a$ is a partial assignment of the variables, the *restriction* of $F$ by $a$ is defined to be the formula $F' = F\lceil_a$ on the set of variables that are not set by $a$, obtained by treating each clause $C$ of $F$ as follows: if $C$ is set to 1 by $a$ then delete $C$, and otherwise replace $C$ by the clause $C'$ obtained by deleting any literals of $C$ that are set to 0 by $a$. Finally, a *unit clause* is a clause that contains exactly one literal.

## Key Results

### ResolveSat Algorithm

The **ResolveSat** algorithm is very simple. Given a $k$-CNF formula, it first generates clauses that can be obtained by resolution without exceeding a certain clause length. Then it takes a random order of variables and gradually assigns values to them in this order. If the currently considered variable occurs in a unit clause, it is assigned the only value that satisfies the clause. If it occurs in contradictory unit clauses, the algorithm starts over. At each step, the algorithm also checks if the formula is satisfied. If the formula is satisfied, then the input is accepted. This subroutine is repeated until either a satisfying assignment is found or a given time limit is exceeded.

The **ResolveSat** algorithm uses the following subroutine, which takes an arbitrary assignment $y$, a CNF formula $F$, and a permutation $\pi$ as input, and produces an assignment $u$. The assignment $u$ is obtained by considering the variables of $y$ in the order given by $\pi$ and modifying their values in an attempt to satisfy $F$.

```
Function Modify(CNF formula $G(x_1, x_2, \ldots, x_n)$, permutation $\pi$ of $\{1, 2, \ldots, n\}$, assignment $y$) $\longrightarrow$ (assignment $u$)
    $G_0 = G$.
    for $i = 1$ to $n$
        if $G_{i-1}$ contains the unit clause $x_{\pi(i)}$
            then $u_{\pi(i)} = 1$
        else if $G_{i-1}$ contains the unit clause $\bar{x}_{\pi(i)}$
            then $u_{\pi(i)} = 0$
        else $u_{\pi(i)} = y_{\pi(i)}$
        $G_i = G_{i-1}\lceil_{x_{\pi(i)}=u_{\pi(i)}}$
    end /* end for loop */
    return $u$;
```

The algorithm **Search** is obtained by running **Modify**$(G, \pi, y)$ on many pairs $(\pi, y)$, where $\pi$ is a random permutation and $y$ is a random assignment.

```
Search(CNF-formula $F$, integer $I$)
    repeat $I$ times
        $\pi$ = uniformly random permutation of $1, \ldots, n$
        $y$ = uniformly random vector $\in \{0, 1\}^n$
        $u$ = Modify($F, \pi, y$);
        if $u$ satisfies $F$
            then output($u$); exit;
    end/* end repeat loop */
    output('Unsatisfiable');
```

The **ResolveSat** algorithm is obtained by combining **Search** with a preprocessing step consisting of *bounded resolution*. For the clauses $C_1$ and $C_2$, $C_1$ and $C_2$ *conflict* on variable $v$ if one of them contains $v$ and the other contains $\bar{v}$. $C_1$ and $C_2$ is a *resolvable pair* if they conflict on exactly one variable $v$. For such a pair, their *resolvent*, denoted $R(C_1, C_2)$, is the clause $C = D_1 \vee D_2$ where $D_1$ and $D_2$ are obtained by deleting $v$ and $\bar{v}$ from $C_1$ and $C_2$. It is easy to see that any assignment satisfying $C_1$ and $C_2$ also satisfies $C$. Hence, if $F$ is a satisfiable CNF formula containing the resolvable pair $C_1, C_2$ then the formula $F' = F \wedge R(C_1, C_2)$ has the same satisfying assignments as $F$. The resolvable pair $C_1, C_2$ is *s-bounded* if $|C_1|, |C_2| \le s$ and $|R(C_1, C_2)| \le s$. The following subroutine extends a formula $F$ to a formula $F_s$ by applying as many steps of $s$-bounded resolution as possible.

```
Resolve(CNF Formula $F$, integer $s$)
    $F_s = F$.
    while $F_s$ has an $s$-bounded resolvable pair $C_1, C_2$
            with $R(C_1, C_2) \notin F_s$
        $F_s = F_s \wedge R(C_1, C_2)$.
    return ($F_s$).
```

The algorithm for $k$-SAT is the following simple combination of **Resolve** and **Search**:

```
ResolveSat(CNF-formula $F$, integer $s$, positive integer $I$)
    $F_s$ = Resolve($F, s$).
    Search($F_s, I$).
```

**Backtracking Based *k*-SAT Algorithms, Table 1**
**This table shows the exponent $c$ in the bound $2^{cn-o(n)}$ for the unique $k$-SAT and $k$-SAT from the ResolveSat algorithm, the bounds for $k$-SAT from Schöning's algorithm [12], its improved versions for 3-SAT [1,5,11], and the hybrid version of [6]**

| $k$ | unique $k$-SAT [9] | $k$-SAT [9] | $k$-SAT [12] | $k$-SAT [1,5,11] | $k$-SAT [6] |
|---|---|---|---|---|---|
| 3 | 0.386 … | 0.521 … | 0.415 … | 0.409 … | 0.404 … |
| 4 | 0.554 … | 0.562 … | 0.584 … | | 0.559 … |
| 5 | 0.650 … | | 0.678 … | | |
| 6 | 0.711 … | | 0.736 … | | |

**Analysis of ResolveSat**

The running time of **ResolveSat**$(F, s, I)$ can be bounded as follows. **Resolve**$(F, s)$ adds at most $O(n^s)$ clauses to $F$ by comparing pairs of clauses, so a naive implementation runs in time $n^{3s}\text{poly}(n)$ (this time bound can be improved, but this will not affect the asymptotics of the main results). **Search**$(F_s, I)$ runs in time $I(|F| + n^s)\text{poly}(n)$. Hence the overall running time of **ResolveSat**$(F, s, I)$ is crudely bounded from above by $(n^{3s} + I(|F| + n^s))\text{poly}(n)$. If $s = O(n/\log n)$, the overall running time can be bounded by $I|F|2^{O(n)}$ since $n^s = 2^{O(n)}$. It will be sufficient to choose $s$ either to be some large constant or to be a *slowly growing* function of $n$. That is, $s(n)$ tends to infinity with $n$ but is $O(\log n)$.

The algorithm **Search**$(F, I)$ always answers "unsatisfiable" if $F$ is unsatisfiable. Thus the only problem is to place an upper bound on the error probability in the case that $F$ is satisfiable. Define $\tau(F)$ to be the probability that **Modify**$(F, \pi, y)$ finds some satisfying assignment. Then for a satisfiable $F$ the error probability of **Search**$(F, I)$ is equal to $(1 - \tau(F))^I \leq e^{-I\tau(F)}$, which is at most $e^{-n}$ provided that $I \geq n/\tau(F)$. Hence, it suffices to give good upper bounds on $\tau(F)$.

Complexity analysis of **ResolveSat** requires certain constants $\mu_k$ for $k \geq 2$:

$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j(j + \frac{1}{k-1})} \,.$$

It is straightforward to show that $\mu_3 = 4 - 4\ln 2 > 1.226$ using Taylor's series expansion of $\ln 2$. Using standard facts, it is easy to show that $\mu_k$ is an increasing function of $k$ with the limit $\sum_{j=1}^{\infty}(1/j^2) = (\pi^2/6) = 1.644\ldots$

The results on the algorithm **ResolveSat** are summarized in the following three theorems.

**Theorem 1** ***(i)*** *Let $k \geq 5$, and let $s(n)$ be a function going to infinity. Then for any satisfiable $k$-CNF formula $F$ on $n$ variables,*

$$\tau(F_s) \geq 2^{-(1-\frac{\mu_k}{k-1})n-o(n)} \,.$$

*Hence,* ***ResolveSat****$(F, s, I)$ with $I = 2^{(1-\mu_k/(k-1))n+O(n)}$ has error probability $O(1)$ and running time $2^{(1-\mu_k/(k-1))n+O(n)}$ on any satisfiable $k$-CNF formula, provided that $s(n)$ goes to infinity sufficiently slowly.*

***(ii)*** *For $k \geq 3$, the same bounds are obtained provided that $F$ is uniquely satisfiable.*

Theorem 1 is proved by first considering the uniquely satisfiable case and then relating the general case to the uniquely satisfiable case. When $k \geq 5$, the analysis reveals that the asymptotics of the general case is no worse than that of the uniquely satisfiable case. When $k = 3$ or $k = 4$, it gives somewhat worse bounds for the general case than for the uniquely satisfiable case.

**Theorem 2** *Let $s = s(n)$ be a slowly growing function. For any satisfiable $n$-variable 3-CNF formula, $\tau(F_s) \geq 2^{-0.521n}$ and so* ***ResolveSat****$(F, s, I)$ with $I = n2^{0.521n}$ has error probability $O(1)$ and running time $2^{0.521n+O(n)}$.*

**Theorem 3** *Let $s = s(n)$ be a slowly growing function. For any satisfiable $n$-variable 4-CNF formula, $\tau(F_s) \geq 2^{-0.5625n}$, and so* ***ResolveSat****$(F, s, I)$ with $I = n2^{0.5625n}$ has error probability $O(1)$ and running time $2^{0.5625n+O(n)}$.*

## Applications

Various heuristics have been employed to produce implementations of 3-CNF satisfiability algorithms which are considerably more efficient than exhaustive search algorithms. The **ResolveSat** algorithm and its analysis provide a rigorous explanation for this efficiency and identify the structural parameters (for example, the width of clauses and the number of solutions), influencing the complexity.

## Open Problems

The gap between the bounds for the general case and the uniquely satisfiable case when $k \in \{3, 4\}$ is due to a weakness in analysis, and it is conjectured that the asymptotic bounds for the uniquely satisfiable case hold in general for all $k$. If true, the conjecture would imply that **ResolveSat** is also faster than any other known algorithm in the $k = 3$ case.

Another interesting problem is to better understand the connection between the number of satisfying assignments and the complexity of finding a satisfying assignment [2]. A strong conjecture is that satisfiability for formulas with many satisfying assignments is strictly easier than for formulas with fewer solutions.

Finally, an important open problem is to design an improved $k$-SAT algorithm which runs faster than the bounds presented in here for the unique $k$-SAT case.

## Cross References

► Local Search Algorithms for $k$SAT
► Maximum Two-Satisfiability
► Parameterized SAT
► Thresholds of Random $k$-SAT

## Recommended Reading

1. Baumer, S., Schuler, R.: Improving a Probabilistic 3-SAT Algorithm by Dynamic Search and Independent Clause Pairs. In: SAT 2003, pp. 150–161
2. Calabro, C., Impagliazzo, R., Kabanets, V., Paturi, R.: The Complexity of Unique $k$-SAT: An Isolation Lemma for $k$-CNFs. In: Proceedings of the Eighteenth IEEE Conference on Computational Complexity, 2003
3. Dantsin, E., Goerdt, A., Hirsch, E.A., Kannan, R., Kleinberg, J., Papadimitriou, C., Raghavan, P., Schöning, U.: A deterministic $(2-\frac{2}{k+1})^n$ algorithm for $k$-SAT based on local search. Theor. Comp. Sci. **289**(1), 69–83 (2002)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Commun. ACM **5**, 394–397 (1962)
5. Hofmeister, T., Schöning, U., Schuler, R., Watanabe, O.: A probabilistic 3–SAT algorithm further improved. In: STACS 2002. LNCS, vol. 2285, pp. 192–202. Springer, Berlin (2002)
6. Iwama, K., Tamaki, S.: Improved upper bounds for 3-SAT. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, 2004, pp. 328–329
7. Kullmann, O.: New methods for 3-SAT decision and worst-case analysis. Theor. Comp. Sci. **223**(1–2), 1–72 (1999)
8. Monien, B., Speckenmeyer, E.: Solving Satisfiability In Less Than $2^n$ Steps. Discret. Appl. Math. **10**, 287–295 (1985)
9. Paturi, R., Pudlák, P., Saks, M., Zane, F.: An Improved Exponential-time Algorithm for $k$-SAT. J. ACM **52**(3), 337–364 (2005) (An earlier version presented in Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, 1998, pp. 628–637)
10. Paturi, R., Pudlák, P., Zane, F.: Satisfiability Coding Lemma. In: Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, 1997, pp. 566–574. Chicago J. Theor. Comput. Sci. (1999), http://cjtcs.cs.uchicago.edu/
11. Rolf, D.: 3-SAT $\in RTIME(1.32971^n)$. In: ECCC TR03-054, 2003
12. Schöning, U.: A probabilistic algorithm for $k$-SAT based on limited local search and restart. Algorithmica **32**, 615–623 (2002) (An earlier version appeared in 40th Annual Symposium on Foundations of Computer Science (FOCS '99), pp. 410–414)

# Best Response Algorithms for Selfish Routing

## 2005; Fotakis, Kontogiannis, Spirakis

PAUL SPIRAKIS
Computer Engineering and Informatics, Research and Academic Computer Technology Institute, Patras University, Patras, Greece

## Keywords and Synonyms

Atomic selfish flows

## Problem Definition

A setting is assumed in which $n$ selfish users compete for routing their loads in a network. The network is an $s-t$ directed graph with a single source vertex $s$ and a single destination vertex $t$. The users are ordered sequentially. It is assumed that each user plays after the user before her in the ordering, and the desired end result is a Pure Nash Equilibrium (PNE for short). It is assumed that, when a user plays (i. e. when she selects an $s-t$ path to route her load), the play is a best response (i. e. minimum delay), given the paths and loads of users currently in the net. The problem then is to find the class of directed graphs for which such an ordering exists so that the implied sequence of best responses leads indeed to a Pure Nash Equilibrium.

### The Model

A *network congestion game* is a tuple $((w_i)_{i\in N}, G, (d_e)_{e\in E})$ where $N=\{1,\dots,n\}$ is the set of users where user $i$ controls $w_i$ units of traffic demand. In *unweighted* congestion games $w_i=1$ for $i=1,\dots,n$. $G(V,E)$ is a directed graph representing the communications network and $d_e$ is the latency function associated with edge $e\in E$. It is assumed that the $d_e$'s are non-negative and non-decreasing functions of the edge loads. The edges are called *identical* if $d_e(x)=x,\ \forall e\in E$. The model is further restricted to single-commodity network congestion games, where $G$ has a single source $s$ and destination $t$ and the set of users' strategies is the set of $s-t$ paths, denoted $P$. Without loss of generality it is assumed that $G$ is connected and that every vertex of $G$ lies on a directed $s-t$ path.

A vector $P=(p_1,\dots,p_n)$ consisting of an $s-t$ path $p_i$ for each user $i$ is a *pure strategies profile*. Let $l_e(P)=\sum_{i:e\in p_i} w_i$ be the load of edge $e$ in $P$. The authors define *the cost* $\lambda_p^i(P)$ for user $i$ routing her demand on path $p$ in the profile $P$ to be

$$\lambda_p^i(P) = \sum_{e \in p \cap p_i} d_e\left(l_e(P)\right) + \sum_{e \in p \smallsetminus p_i} d_e\left(l_e(P) + w_i\right) .$$

The cost $\lambda^i(P)$ of user $i$ in $P$ is just $\lambda_{p_i}^i(P)$, i. e. the total delay along her path.

A pure strategies profile $P$ is a Pure Nash Equilibrium (PNE) iff no user can reduce her total delay by *unilaterally deviating* i. e. by selecting another $s-t$ path for her load, while all other users keep their paths.

### Best Response

Let $p_i$ be the path of user $i$ and $P^i = (p_1, \ldots, p_i)$ be the pure strategies profile for users $1, \ldots, i$. Then the *best response* of user $i+1$ is a path $p_{i+1}$ so that

$$p_{i+1} = avg \min_{p \in P^i} \left\{ \sum_{e \in p} \left( d_e \left( l_e \left( P^i \right) + w_{i+1} \right) \right) \right\} .$$

### Flows and Common Best Response

A (feasible) flow on the set $P$ of $s-t$ paths of $G$ is a function $f \colon P \to \Re_{\geq 0}$ so that

$$\sum_{p \in P} f_p = \sum_{i=1}^{n} w_i .$$

The single-commodity network congestion game $((w_i)_{i \in N}, G, (d_e)_{e \in E})$ has the Common Best Response property if for every initial flow $f$ (not necessarily feasible), all users have the same set of best responses with respect to $f$. That is, if a path $p$ is a best response with respect to $f$ for some user, then for all users $j$ and all paths $p'$

$$\sum_{e \in p'} d_e\left(f_e + w_j\right) \geq \sum_{e \in p} d_e\left(f_e + w_j\right) .$$

Furthermore, every segment $\pi$ of a best response path $p$ is a best response for routing the demand of any user between $\pi$'s endpoints. It is allowed here that some users may already have contributed to the initial flow $f$.

### Layered and Series-Parallel Graphs

A directed (multi)graph $G(V, E)$ with a distinguished source $s$ and destination $t$ is *layered* iff all directed $s-t$ paths have exactly the same length and each vertex lies on some directed $s-t$ path.

A multigraph is *series-parallel* with *terminals* $(s, t)$ if

1. it is a single edge $(s, t)$ or
2. it is obtained from two series-parallel graphs $G_1, G_2$ with terminals $(s_1, t_1)$ and $(s_2, t_2)$ by connecting them either in *series* or in *parallel*. In a series connection, $t_1$ is identified with $s_2$ and $s_1$ becomes $s$ and $t_2$ becomes $t$. In a parallel connection, $s_1 = s_2 = s$ and $t_1 = t_2 = t$.

## Key Results

### The Greedy Best Response Algorithm (GBR)

GBR considers the users one-by-one in *non-increasing* order of weight (i. e. $w_1 \geq w_2 \geq \cdots \geq w_n$). Each user adopts her best response strategy on the set of (already adopted in the net) best responses of previous users. No user can change her strategy in the future. Formally, GBR *succeeds* if the eventual profile $P$ is a Pure Nash Equilibrium (PNE).

### The Characterization

In [3] it is shown:

**Theorem 1** *If G is an $(s-t)$ series-parallel graph and the game $((w_i)_{i \in N}, G, (d_e)_{e \in E})$ has the common best response property, then GBR succeeds.*

**Theorem 2** *A weighted single-commodity network congestion game in a layered network with identical edges has the common best response property for any set of user weights.*

**Theorem 3** *For any single-commodity network congestion game in series-parallel networks, GBR succeeds if*
1. *The users are identical (if $w_i = 1$ for all $i$) and the edge-delays are arbitrary but non-decreasing or*
2. *The graph is layered and the edges are identical (for arbitrary user weights)*

**Theorem 4** *If the network consists of bunches of parallel-links connected in series, then a PNE is obtained by applying GBR to each bunch.*

**Theorem 5**
1. *If the network is not series-parallel then there exist games where GBR fails, even for 2 identical users and identical edges.*
2. *If the network does not have the common best response property (and is not a sequence of parallel links graphs connected in series) then there exist games where GBR fails, even for 2-layered series-parallel graphs.*

Examples of such games are provided in [3].

## Applications

GBR has a natural distributed implementation based on a leader election algorithm. Each player is now represented by a process. It is assumed that processes know the network and the edge latency functions. The existence of a message passing subsystem and an underlying synchronization mechanism (e. g. logical timestamps) is assumed, that allows a distributed protocol to proceed in logical rounds.

Initially all processes are active. In each round they run a leader election algorithm and determine the process of largest weight (among the active ones). This process routes its demand on its best response path, announces its strategy to all active processes, and becomes passive. Notice that each process can compute its best response locally.

## Open Problems

What is the class of networks where (identical) users can achieve a PNE by a $k$-round repetition of a best responses sequence? What happens to weighted users? In general, how the network topology affects best response sequences? Such open problems are a subject of current research.

## Cross References

► General Equilibrium

## Recommended Reading

1. Awerbuch, B., Azar, Y., Epstein, A.: The price of Routing Unsplittable Flows. In: Proc. ACM Symposium on Theory of Computing (STOC) 2005, pp. 57-66. ACM, New York (2005)
2. Duffin, R.J.: Topology of Series-Parallel Networks. J. Math. Anal. Appl. **10**, 303–318 (1965)
3. Fotakis, D., Kontogiannis, S., Spirakis, P.: Symmetry in Network Congestion Games: Pure Equilibria and Anarchy Cost. In: Proc. of the 3rd Workshop of Approximate and On-line Algorithms (WAOA 2005). Lecture Notes in Computer Science (LNCS), vol. 3879, pp. 161–175. Springer, Berlin Heidelberg (2006)
4. Fotakis, D., Kontogiannis, S., Spirakis, P.: Selfish Unsplittable Flows. J. Theor. Comput. Sci. **348**, 226–239 (2005)
5. Libman, L., Orda, A.: Atomic Resource Sharing in Noncooperative Networks. Telecommun. Syst. **17**(4), 385-409 (2001)

# Bidimensionality

## 2004; Demaine, Fomin, Hajiaghayi, Thilikos

ERIK D. DEMAINE[1], MOHAMMADTAGHI HAJIAGHAYI[2]

[1] Computer Science and Artifical Intelligence Laboratory, MIT, Cambridge, MA, USA

[2] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA

## Problem Definition

The theory of bidimensionality provides general techniques for designing efficient fixed-parameter algorithms and approximation algorithms for a broad range of NP-hard graph problems in a broad range of graphs. This theory applies to graph problems that are "bidimensional" in the sense that (1) the solution value for the $k \times k$ grid graph and similar graphs grows with $k$, typically as $\Omega(k^2)$, and (2) the solution value goes down when contracting edges and optionally when deleting edges in the graph. Many problems are bidimensional; a few classic examples are vertex cover, dominating set, and feedback vertex set.

### Graph Classes

Results about bidimensional problems have been developed for increasingly general families of graphs, all generalizing planar graphs.

The first two classes of graphs relate to embeddings on surfaces. A graph is *planar* if it can be drawn in the plane (or the sphere) without crossings. A graph has *(Euler) genus* at most $g$ if it can be drawn in a surface of Euler characteristic $g$. A class of graphs has *bounded genus* if every graph in the class has genus at most $g$ for a fixed $g$.

The next three classes of graphs relate to excluding minors. Given an edge $e = \{v, w\}$ in a graph $G$, the *contraction* of $e$ in $G$ is the result of identifying vertices $v$ and $w$ in $G$ and removing all loops and duplicate edges. A graph $H$ obtained by a sequence of such edge contractions starting from $G$ is said to be a *contraction* of $G$. A graph $H$ is a *minor* of $G$ if $H$ is a subgraph of some contraction of $G$. A graph class $C$ is *minor-closed* if any minor of any graph in $C$ is also a member of $C$. A minor-closed graph class $C$ is *$H$-minor-free* if $H \notin C$. More generally, the term "$H$-minor-free" refers to any minor-closed graph class that excludes some fixed graph $H$. A *single-crossing graph* is a minor of a graph that can be drawn in the plane with at most one pair of edges crossing. A minor-closed graph class is *single-crossing-minor-free* if it excludes a fixed single-crossing graph. An *apex graph* is a graph in which the removal of some vertex leaves a planar graph. A graph class is *apex-minor-free* if it excludes some fixed apex graph.

### Bidimensional Parameters

Although implicitly hinted at in [2,5,10,11], the first use of the term "bidimensional" was in [3].

First, "parameters" are an alternative view on optimization problems. A *parameter* $P$ is a function mapping graphs to nonnegative integers. The *decision problem associated with $P$* asks, for a given graph $G$ and nonnegative integer $k$, whether $P(G) \le k$. Many optimization problems can be phrased as such a decision problem about a graph parameter $P$.

Now, a parameter is *$g(r)$-bidimensional* (or just *bidimensional*) if it is at least $g(r)$ in an $r \times r$ "grid-like graph" and if the parameter does not increase when taking either minors $g(r)$(*-minor-bidimensional*) or contractions ($g(r)$*-contraction-bidimensional*). The exact definition of "grid-like graph" depends on the class of graphs allowed and whether one considers minor- or contraction-bidimensionality. For minor-bidimensionality and for any $H$-minor-free graph class, the notion of a "grid-like graph" is defined to be the $r \times r$ *grid*, i. e., the planar graph with $r^2$ vertices arranged on a square grid and with edges connecting horizontally and vertically adjacent vertices. For contraction-bidimensionality, the notion of a "grid-like graph" is as follows:

1. For planar graphs and single-crossing-minor-free graphs, a "grid-like graph" is an $r \times r$ grid partially triangulated by additional edges that preserve planarity.
2. For bounded-genus graphs, a "grid-like graph" is such a partially triangulated $r \times r$ grid with up to genus($G$) additional edges ("handles").
3. For apex-minor-free graphs, a "grid-like graph" is an $r \times r$ grid augmented with additional edges such that each vertex is incident to $O(1)$ edges to nonboundary vertices of the grid. (Here $O(1)$ depends on the excluded apex graph.)

Contraction-bidimensionality is so far undefined for $H$-minor-free graphs (or general graphs).

Examples of bidimensional parameters include the number of vertices, the diameter, and the size of various structures such as feedback vertex set, vertex cover, minimum maximal matching, face cover, a series of vertex-removal parameters, dominating set, edge dominating set, $R$-dominating set, connected dominating set, connected edge dominating set, connected $R$-dominating set, unweighted TSP tour (a walk in the graph visiting all vertices), and chordal completion (fill-in). For example, feedback vertex set is $\Omega(r^2)$-minor-bidimensional (and thus also contraction-bidimensional) because (1) deleting or contracting an edge preserves existing feedback vertex sets, and (2) any vertex in the feedback vertex set destroys at most four squares in the $r \times r$ grid, and there are $(r-1)^2$ squares, so any feedback vertex set must have $\Omega(r^2)$ vertices. See [1,3] for arguments of either contraction- or minor-bidimensionality for the other parameters.

## Key Results

Bidimensionality builds on the seminal Graph Minor Theory of Robertson and Seymour, by extending some mathematical results and building new algorithmic tools. The foundation for several results in bidimensionality are the following two combinatorial results. The first relates any bidimensional parameter to treewidth, while the second relates treewidth to grid minors.

**Theorem 1** ([1,8]) *If the parameter $P$ is $g(r)$-bidimensional, then for every graph $G$ in the family associated with the parameter $P$, $tw(G) = O(g^{-1}(P(G)))$. In particular, if $g(r) = \Theta(r^2)$, then the bound becomes $tw(G) = O(\sqrt{P(G)})$.*

**Theorem 2** ([8]) *For any fixed graph $H$, every $H$-minor-free graph of treewidth $w$ has an $\Omega(w) \times \Omega(w)$ grid as a minor.*

The two major algorithmic results in bidimensionality are general subexponential fixed-parameter algorithm, and general polynomial-time approximation scheme (PTASs):

**Theorem 3** ([1,8]) *Consider a $g(r)$-bidimensional parameter $P$ that can be computed on a graph $G$ in $h(w)n^{O(1)}$ time given a tree decomposition of $G$ of width at most $w$. Then there is an algorithm computing $P$ on any graph $G$ in $P$'s corresponding graph class, with running time $[h(O(g^{-1}(k))) + 2^{O(g^{-1}(k))}]n^{O(1)}$. In particular, if $g(r) = \Theta(r^2)$ and $h(w) = 2^{o(w^2)}$, then this running time is subexponential in $k$.*

**Theorem 4** ([7]) *Consider a bidimensional problem satisfying the "separation property" defined in [4,7]. Suppose that the problem can be solved on a graph $G$ with $n$ vertices in $f(n, tw(G))$ time. Suppose also that the problem can be approximated within a factor of $\alpha$ in $g(n)$ time. For contraction-bidimensional problems, suppose further that both of these algorithms also apply to the "generalized form" of the problem defined in [4,7]. Then there is a $(1+\epsilon)$-approximation algorithm whose running time is $O(nf(n, O(\alpha^2/\epsilon)) + n^3 g(n))$ for the corresponding graph class of the bidimensional problem.*

## Applications

The theorems above have many combinatorial and algorithmic applications.

Applying the parameter-treewidth bound of Theorem 1 to the parameter of the number of vertices in the graph proves that every $H$-minor-free graph on $n$ vertices has treewidth $O(\sqrt{n})$, thus (re)proving the separator theorem for $H$-minor-free graphs. Applying the parameter-treewidth bound of Theorem 1 to the parameter of the diameter of the graph proves a stronger form of Eppstein's diameter-treewidth relation for apex-minor-free graphs. (Further work shows how to further strengthen the diameter-treewidth relation to linear [6].) The treewidth-grid relation of Theorem 2 can be used to bound the gap between half-integral multicommodity flow and fractional multicommodity flow in $H$-minor-free graphs. It also yields an $O(1)$-approximation for treewidth in $H$-minor-free graphs. The subexponential fixed-parameter algorithms of Theorem 3 subsume or strengthen all previous such results. These results can also be generalized to obtain fixed-parameter algorithms in arbitrary graphs. The PTASs of Theorem 4 in particular establish the first PTASs for connected dominating set and feedback vertex set even for planar graphs. For details of all of these results, see [4].

## Open Problems

Several combinatorial and algorithmic open problems remain in the theory of bidimensionality and related concepts.

Can the grid-minor theorem for $H$-minor-free graphs, Theorem 2, be generalized to arbitrary graphs with a polynomial relation between treewidth and the largest grid minor? (The best relation so far is exponential.) Such polynomial generalizations have been obtained for the cases of "map graphs" and "power graphs" [9]. Good grid-treewidth bounds have applications to minor-bidimensional problems.

Can the algorithmic results (Theorem 3 and Theorem 4) be generalized to solve contraction-bidimensional problems beyond apex-minor-free graphs? It is known that the basis for these results, Theorem 1, does not generalize [1]. Nonetheless, Theorem 3 has been generalized for one specific contraction-bidimensional problem, dominating set [3].

Can the polynomial-time approximation schemes of Theorem 4 be generalized to more general algorithmic problems that do not correspond directly to bidimensional parameters? One general family of such problems arises when adding weights to vertices and/or edges, and the goal is e. g. to find the minimum-weight dominating set. Another family of such problems arises when placing constraints (e. g., on coverage or domination) only on subsets of vertices and/or edges. Examples of such problems include Steiner tree and subset feedback vertex set.

For additional open problems and details about the problems above, see [4].

## Cross References

► Approximation Schemes for Planar Graph Problems
► Branchwidth of Graphs
► Treewidth of Graphs

## Recommended Reading

1. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Bidimensional parameters and local treewidth. SIAM J. Discret. Math. **18**(3), 501–511 (2004)
2. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Fixed-parameter algorithms for $(k, r)$-center in planar graphs and map graphs. ACM Trans. Algorithms **1**(1), 33–47 (2005)
3. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Subexponential parametrized algorithms on graphs of bounded genus and $H$-minor-free graphs. J. ACM **52**(6), 866–893 (2005)
4. Demaine, E.D., Hajiaghayi, M.: The bidimensionality theory and its algorithmic applications. Comput. J. To appear
5. Demaine, E.D., Hajiaghayi, M.: Diameter and treewidth in minor-closed graph families, revisited. Algorithmica **40**(3), 211–215 (2004)
6. Demaine, E.D., Hajiaghayi, M.: Equivalence of local treewidth and linear local treewidth and its algorithmic applications. In: Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA'04), January 2004, pp. 833–842 (2004)
7. Demaine, E.D., Hajiaghayi, M.: Bidimensionality: New connections between FPT algorithms and PTASs. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005), pp. 590–601. Vancouver, January (2005)
8. Demaine, E.D., Hajiaghayi, M.: Graphs excluding a fixed minor have grids as large as treewidth, with combinatorial and algorithmic applications through bidimensionality. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005), pp. 682–689. Vancouver, January (2005)
9. Demaine, E.D., Hajiaghayi, M., Kawarabayashi, K.: Algorithmic graph minor theory: Improved grid minor bounds and Wagner's contraction. In: Proceedings of the 17th Annual International Symposium on Algorithms and Computation, Calcutta, India, December 2006. Lecture Notes in Computer Science, vol. 4288, pp. 3–15 (2006)
10. Demaine, E.D., Hajiaghayi, M., Nishimura, N., Ragde, P., Thilikos, D.M.: Approximation algorithms for classes of graphs excluding single-crossing graphs as minors. J. Comput. Syst. Sci. **69**(2), 166–195 (2004)
11. Demaine, E.D., Hajiaghayi, M., Thilikos, D.M.: Exponential speedup of fixed-parameter algorithms for classes of graphs excluding single-crossing graphs as minors. Algorithmica **41**(4), 245–267 (2005)
12. Demaine, E.D., Hajiaghayi, M., Thilikos, D.M.: The bidimensional theory of bounded-genus graphs. SIAM J. Discret. Math. **20**(2), 357–371 (2006)

# Binary Decision Graph

## 1986; Bryant

AMIT PRAKASH[1], ADNAN AZIZ[2]
[1] Microsoft, MSN, Redmond, WA, USA
[2] Department of Electrical and Computer Engineering, University of Texas, Austin, TX, USA

## Keywords and Synonyms

BDDs; Binary decision diagrams

## Problem Definition

### Boolean Functions

The concept of a *Boolean function* – a function whose domain is $\{0,1\}^n$ and range is $\{0,1\}$ – is central to computing. Boolean functions are used in foundational studies of complexity [7,9], as well as the design and analysis of logic circuits [4,13]. A Boolean function can be represented using a *truth table* – an enumeration of the values taken by the function on each element of $\{0,1\}^n$. Since the truth table representation requires memory exponential in $n$, it is impractical for most applications. Consequently, there is a need for data structures and associated algorithms for efficiently representing and manipulating Boolean functions.

### Boolean Circuits

Boolean functions can be represented in many ways. One natural representation is a *Boolean combinational circuit*, or circuit for short [6, Chapter 34]. A circuit consists of *Boolean combinational elements* connected by *wires*. The Boolean combinational elements are *gates* and *primary inputs*. Gates come in three types: NOT, AND, and OR. The NOT gate functions as follows: it takes a single Boolean-valued *input*, and produces a single Boolean-valued *output* which takes value 0 if the input is 1, and 1 if the input is 0. The AND gate takes two Boolean-valued inputs and produce a single output; the output is 1 if both inputs are 1, and 0 otherwise. The OR gate is similar to AND, except that its output is 1 if one or both inputs are 1, and 0 otherwise.

Circuits are required to be acyclic. The absence of cycles implies that a Boolean-assignment to the primary inputs can be unambiguously propagated through the gates in topological order. It follows that a circuit on $n$ ordered primary inputs with a designated gate called the *primary output* corresponds to a Boolean function on $\{0,1\}^n$. Every Boolean function can be represented by a circuit, e. g., by building a circuit that mimics the truth table.

The circuit representation is very general – any decision problem that is computable in polynomial-time on a Turing machine can be computed by circuits polynomial in the instance size, and the circuits can be constructed efficiently from the Turing machine program [15]. However, the key analysis problems on circuits, namely satisfiability and equivalence, are NP-hard [7].

### Boolean Formulas

A *Boolean formula* is defined recursively: a *Boolean variable* $x_i$ is a *Boolean formula*, and if $\varphi$ and $\psi$ are Boolean formulas, then so are $(\neg\phi)$, $(\phi\wedge\psi)$, $(\phi\vee\psi)$, $(\phi\rightarrow\psi)$, and $(\phi\leftrightarrow\psi)$. The operators $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ are referred to as *connectives*; parentheses are often dropped for notational convenience. Boolean formulas also can be used to represent arbitrary Boolean functions; however, formula satisfiability and equivalence are also NP-hard. Boolean formulas are not as succinct as Boolean circuits: for example, the parity function has linear sized circuits, but formula representations of parity are super-polynomial. More precisely, $XOR_n\colon \{0,1\}^n \mapsto \{0,1\}$ is defined to take the value 1 on exactly those elements of $\{0,1\}^n$ which contain an odd number of 1s. Define the *size* of a formula to be the number of connectives appearing in it. Then for any sequence of formulas $\theta_1, \theta_2, \ldots$ such that $\theta_k$ represents $XOR_k$, the size of $\theta_k$ is $\Omega(k^c)$ for all $c \in Z^+$ [14, Chapters 11,12].

A *disjunct* is a Boolean formula in which $\wedge$ and $\neg$ are the only connectives, and $\neg$ is applied only to variables; for example, $x_1 \wedge \neg x_3 \wedge \neg x_5$ is a disjunct. A Boolean formula is said to be in *Disjunctive Normal Form* (DNF) if it is of the form $D_0 \vee D_1 \vee \cdots \vee D_{k-1}$, where each $D_i$ is a disjunct. DNF formulas can represent arbitrary Boolean functions, e. g., by identifying each input on which the formula takes the value 1 with a disjunct. DNF formulas are useful in logic design, because it can be translated directly into a PLA implementation [4]. While satisfiability of DNF formulas is trivial, equivalence is NP-hard. In addition, given DNF formulas $\varphi$ and $\psi$, the formulas $\neg\phi$ and $\phi\wedge\psi$ are not DNF formulas, and the translation of these formulas to DNF formulas representing the same function can lead to exponential growth in the size of the formula.

### Shannon Trees

Let $f$ be a Boolean function on domain $\{0,1\}^n$. Associate the $n$ dimensions with variables $x_0, \ldots, x_{n-1}$. Then the *positive cofactor* of $f$ with respect to $x_i$, denoted by $f_{x_i}$, is the function on domain $\{0,1\}^n$, which is defined by

$$\begin{aligned} & f_{x_i}(\alpha_0, \ldots, \alpha_{i-1}, a_i, \alpha_{i+1}, \ldots, \alpha_{n-1}) \\ &= f(\alpha_0, \ldots, \alpha_{i-1}, 1, \alpha_{i+1}, \ldots, \alpha_{n-1})\,. \end{aligned}$$

The *negative cofactor* of $f$ with respect to $x_i$, denoted by $f_{x_i'}$ is defined similarly, with 0 taking the place of 1 in the right-hand side.

Every Boolean function can be decomposed using Shannon's expansion theorem:

$$f(x_1, \ldots, x_n) = x_i \cdot f_{x_i} + x_i' \cdot f_{x_i'}\,.$$

This observation can be used to represent $f$ by a *Shannon tree* – a full binary tree [6, Appendix B.5] of height $n$, where each path to a leaf node defines a complete assignment to the $n$ variables that $f$ is defined over, and the leaf node holds a 0 or a 1, based on the value $f$ takes for the assignment.

The Shannon tree is not a particularly useful representation, since the height of the tree representing every Boolean function on $\{0,1\}^n$ is $n$, and the tree has $2^n$ leaves. The Shannon tree can be made smaller by merging isomorphic subtrees, and bypassing nodes which have identical children. At first glance the reduced Shannon tree representation is not particularly useful, since it entails creating the full binary tree in the first place. Furthermore, it is not clear how to efficiently perform computations on the reduced Shannon tree representation, such as equivalence checking or computing the conjunction of functions presented as reduced Shannon trees.

Bryant [5] recognized that adding the restriction that variables appear in fixed order from root to leaves greatly reduced the complexity of manipulating reduced Shannon trees. He referred to this representation as a Binary Decision Diagram (BDD).

## Key Results

### Definitions

Technically, a BDD is a directed acyclic graph (DAG), with a designated root, and at most two sinks – one labeled 0, the other labeled 1. Nonsink nodes are labeled with a variable. Each nonsink node has two outgoing edges – one labeled with a 1 leading to the *1-child*, the other is a 0, leading to the *0-child*. Variables must be ordered – that is if the variable label $x_i$ appears before the label $x_j$ on some path from the root to a sink, then the label $x_j$ is precluded from appearing before $x_i$ on any path from the root to a sink. Two nodes are *isomorphic* if both are equi-labeled sinks, or they are both nonsink nodes, with the same variable label, and their 0- and 1-children are isomorphic. For the DAG to be a valid BDD, it is required that there are no isomorphic nodes, and for no nodes are its 0- and 1-children the same.

A key result in the theory of BDDs is that given a fixed variable ordering, the representation is unique upto isomorphism, i. e., if $F$ and $G$ are both BDDs representing $f: \{0,1\}^n \mapsto \{0,1\}$ under the variable ordering $x_1 \prec x_2 \prec \cdots x_n$, then $F$ and $G$ are isomorphic.

The definition of isomorphism directly yields a recursive algorithm for checking isomorphism. However, the resulting complexity is exponential in the number of nodes – this is illustrated for example by checking the isomorphism of the BDD for the parity function against itself. On inspection, the exponential complexity arises from repeated checking of isomorphism between pairs of nodes – this naturally suggest dynamic programming. Caching isomorphism checks reduces the complexity of isomorphism checking to $O(|F| \cdot |G|)$, where $|B|$ denotes the number of nodes in the BDD $B$.

### BDD Operations

Many logical operations can be implemented in polynomial time using BDDs: *bdd_and* which computes a BDD representing the logical AND of the functions represented by two BDDs, *bdd_or* and *bdd_not* which are defined similarly, and *bdd_compose* which takes a BDD representing a function $f$, a variable $v$, and a BDD representing a function $g$ and returns the BDD for $f$ where $v$ is substituted by $g$ are examples.

The example of *bdd_and* is instructive – it is based on the identity $f \cdot g = x \cdot (f_x \cdot g_x) + x\prime \cdot (f_{x\prime} \cdot g_{x\prime})$. The recursion can be implemented directly: the base cases are when either $f$ or $g$ are 0, and when one or both are 1. The recursion chooses the variable $v$ labeling either the root of the BDD for $f$ or $g$, depending on which is earlier in the variable ordering, and recursively computes BDDs for $f_v \cdot g_v$ and $f_{v'} \cdot g_{v'}$; these are merged if isomorphic. Given a BDD $F$ for $f$, if $v$ is the variable labeling the root of $F$, the BDDs for $f_{v'}$ and $f_v$ respectively are simply the 0-child and 1-child of $F$'s root.

The implementation of *bdd_and* as described has exponential complexity because of repeated subproblems arising. Dynamic programming again provides a solution – caching the intermediate results of *bdd_and* reduced the complexity to $O(|F| \cdot |G|)$.

### Variable Ordering

All symmetric functions on $\{0,1\}^n$, have a BDD that is polynomial in $n$, independent of the variable ordering. Other useful functions such as comparators, multiplexers, adders, and subtracters can also be efficiently represented, if the variable ordering is selected correctly. Heuristics for ordering selection are presented in [1,2,11]. There are functions which do not have a polynomial-sized BDD under any variable ordering – the function representing the $n$-th bit of the output of a multiplier taking two $n$-bit unsigned integer inputs is an example [5]. Wegener [17] presents many more examples of the impact of variable ordering.

## Applications

BDDs have been most commonly applied in the context of formal verification of digital hardware [8]. Digital hardware extends the notion of circuit described above by adding *state elements* which hold a Boolean value between updates, and are updated on a *clock* signal.

The gates comprising a design are often updated based on performance requirements; these changes typically are not supposed to change the logical functionality of the design. BDD-based approaches have been used for checking the equivalence of digital hardware designs [10].

BDDs have also been used for checking properties of digital hardware. A typical formulation is that a set of "good" states and a set of "initial" states are specified using Boolean formulas over the state elements; the property holds iff there is no sequence of inputs which leads a state in the initial state to a state not in the set of good states. Given a design with $n$ registers, a set of states $A$ in the design can be characterized by a formula $\varphi_A$ over $n$ Boolean variables: $\varphi_A$ evaluates to true on an assignment to the variables iff the corresponding state is in $A$. The formula $\varphi_A$ represents a Boolean function, and so BDDs can be used to represent sets of states. The key operation of computing the *image* of a set of states $A$, i. e., the set of states that can be reached on application of a single input from states in $A$, can also be implemented using BDDs [12].

BDDs have been used for *test generation*. One approach to test generation is to specify legal inputs using constraints, in essence Boolean formulas over the the primary input and state variables. Yuan et al. [18] have demonstrated that BDDs can be used to solve these constraints very efficiently.

Logic synthesis is the discipline of realizing hardware designs specified as logic equations using gates. Mapping equations to gates is straightforward; however, in practice a direct mapping leads to implementations that are not acceptable from a performance perspective, where performance is measured by gate area or timing delay. Manipulating logic equations in order to reduce area (e. g., through constant propagation, identifying common subexpressions, etc.), and delay (e. g., through propagating late arriving signals closer to the outputs), is conveniently done using BDDs.

## Experimental Results

Bryant reported results on verifying two qualitatively distinct circuits for addition. He was able to verify on a VAX 11/780 (a 1 MIP machine) that two 64-bit adders were equivalent in 95.8 minutes. He used an ordering that he derived manually.

Normalizing for technology, modern BDD packages are two orders of magnitude faster than Bryant's original implementation. A large source the improvement comes from the use of the *strong canonical form*, wherein a global database of BDD nodes is maintained, and no new node is added without checking to see if a node with the same label and 0- and 1-children exists in the database [3]. (For this approach to work, it is also required that the children of any node being added be in strong canonical form.) Other improvements stem from the use of complement pointers (if a pointer has its least-significant bit set, it refers to the complement of the function), better memory management (garbage collection based on reference counts, keeping nodes that are commonly accessed together close in memory), better hash functions, and better organization of the computed table (which keeps track of sub-problems that have already been encountered) [16].

## Data Sets

The SIS (http://embedded.eecs.berkeley.edu/pubs/downloads/sis/) system from UC Berkeley is used for logic synthesis. It comes with a number of combinational and sequential circuits that have been used for benchmarking BDD packages.

The VIS (http://embedded.eecs.berkeley.edu/pubs/downloads/vis) system from UC Berkeley and UC Boulder is used for design verification; it uses BDDs to perform checks. The distribution includes a large collection of verification problems, ranging from simple hardware circuits, to complex multiprocessor cache systems.

## URL to Code

A number of BDD packages exist today, but the package of choice is CUDD (http://vlsi.colorado.edu/~fabio/CUDD/). CUDD implements all the core features for manipulating BDDs, as well as variants. It is written in C++, and has extensive user and programmer documentation.

## Cross References

► Symbolic Model Checking

## Recommended Reading

1. Aziz, A., Tasiran, S., Brayton, R.: BDD Variable Ordering for Interacting Finite State Machines. In: ACM Design Automation Conference, pp. 283–288. (1994)
2. Berman, C.L.: Ordered Binary Decision Diagrams and Circuit Structure. In: IEEE International Conference on Computer Design. (1989)
3. Brace, K., Rudell, R., Bryant, R.: Efficient Implementation of a BDD Package. In: ACM Design Automation Conference. (1990)
4. Brayton, R., Hachtel, G., McMullen, C., Sangiovanni-Vincentelli, A.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers (1984)

5. Bryant, R.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Transac. Comp. **C-35**, 677–691 (1986)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.H., Stein, C.: Introduction to Algorithms. MIT Press (2001)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman and Co. (1979)
8. Gupta, A.: Formal Hardware Verification Methods: A Survey. Formal Method Syst. Des. **1**, 151–238 (1993)
9. Karchmer, M.: Communication Complexity: A New Approach to Circuit Depth. MIT Press (1989)
10. Kuehlmann, A., Krohm, F.: Equivalence Checking Using Cuts and Heaps. In: ACM Design Automation Conference (1997)
11. Malik, S., Wang, A.R., Brayton, R.K., Sangiovanni-Vincentelli, A.: Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In: IEEE International Conference on Computer-Aided Design, pp. 6–9. (1988)
12. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
13. De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw Hill (1994)
14. Schoning, U., Pruim, R.: Gems of Theoretical Computer Science. Springer (1998)
15. Sipser, M.: Introduction to the Theory of Computation, 2nd edn. Course Technology (2005)
16. Somenzi, F.: Colorado University Decision Diagram Package. http://vlsi.colorado.edu/~fabio/
17. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM (2000)
18. Yuan, J., Pixley, C., Aziz, A.: Constraint-Based Verfication. Springer (2006)

# Bin Packing

## 1997; Coffman, Garay, Johnson

DAVID S. JOHNSON
Algorithms and Optimization Research Department, AT&T Labs, Florham Park, NJ, USA

## Keywords and Synonyms

Cutting stock problem

## Problem Definition

In the one-dimensional bin packing problem, one is given a list $L = (a_1, a_2, \dots, a_n)$ of items, each item $a_i$ having a size $s(a_i) \in (0, 1]$. The goal is to pack the items into a minimum number of unit-capacity bins, that is, to partition the items into a minimum number of sets, each having total size of at most 1. This problem is NP-hard, and so much of the research on it has concerned the design and analysis of approximation algorithms, which will be the subject of this article.

Although bin packing has many applications, it is perhaps most important for the role it has played as a proving ground for new algorithmic and analytical techniques. Some of the first worst- and average-case results for approximation algorithms were proved in this domain, as well as the first lower bounds on the competitive ratios of online algorithms. Readers interested in a more detailed coverage than is possible here are directed to two relatively recent surveys [4,11].

## Key Results

### Worst-Case Behavior

**Asymptotic Worst-Case Ratios** For most minimization problems, the standard worst-case metric for an approximation algorithm $A$ is the maximum, over all instances $I$, of the ratio $A(I)/OPT(I)$, where $A(I)$ is the value of the solution generated by $A$ and $OPT(I)$ is the optimal solution value. In the case of bin packing, however, there are limitations to this "absolute worst-case ratio" metric. Here it is already NP-hard to determine whether $OPT(I) = 2$, and hence no polynomial-time approximation algorithm can have an absolute worst-case ratio better than 1.5 unless P = NP. To better understand the behavior of bin packing algorithms in the typical situation where the given list $L$ requires a large number of bins, researchers thus use a more refined metric for bin packing, the *asymptotic worst-case ratio* $R_A^\infty$. This is defined in two steps as follows.

$$R_A^n = \max\{A(L)/OPT(L) : L \text{ is a list with } OPT(L) = n\}$$
$$R_A^\infty = \limsup_{n\to\infty} R_A^n$$

The first algorithm whose behavior was analyzed in these terms was *First Fit* (FF). This algorithm envisions an infinite sequence of empty bins $B_1, B_2, \dots$ and, starting with the first item in the input list $L$, places each item in turn into the first bin which still has room for it. In a technical report from 1971 which was one of the very first papers in which worst-case performance ratios were studied, Ullman [22] proved the following.

**Theorem 1** ([22]) $R_{FF}^\infty = 17/10$.

In addition to FF, five other simple heuristics received early study and have served as the inspiration for later research. *Best Fit* (BF) is the variant of FF in which each item is placed in the bin into which it will fit with the least space left over, with ties broken in favor of the earliest such bin. Both FF and BF can be implemented to run in time $O(n \log n)$ [12]. *Next Fit* (NF) is a still simpler and linear-time algorithm in which the first item is placed in the first bin, and thereafter each item is placed in the last nonempty bin if it will fit, otherwise a new bin is started. *First Fit Decreasing* (FFD) and *Best Fit Decreasing* (BFD) are the variants of those algorithms in which the input list is first sorted into nonincreasing order by size and then the corresponding packing rule is applied. The results for these algorithms are as follows.

**Theorem 2** ([12]) $R^{\infty}_{NF} = 2$.

**Theorem 3** ([13]) $R^{\infty}_{BF} = 17/10$.

**Theorem 4** ([12,13]) $R^{\infty}_{FFD} = R^{\infty}_{BFD} = 11/9 = 1.222\ldots$

The abovementioned algorithms are relatively simple and intuitive. If one is willing to consider more complicated algorithms, one can do substantially better. The current best polynomial-time bin packing algorithm is very good indeed. This is the 1982 algorithm of Karmarkar and Karp [15], denoted here as "KK." It exploits the ellipsoid algorithm, approximation algorithms for the knapsack problem, and a clever rounding scheme to obtain the following guarantees.

**Theorem 5** ([15]) $R^{\infty}_{KK} = 1$ *and there is a constant c such that for all lists L,*

$$KK(L) \le OPT(L) + c\log^2(OPT(L))\ .$$

Unfortunately, the running time for KK appears to be worse than $O(n^8)$, and BFD and FFD remain much more practical alternatives.

**Online Algorithms** Three of the abovementioned algorithms (FF, BF, and NF) are *online* algorithms, in that they pack items in the order given, without reference to the sizes or number of later items. As was subsequently observed in many contexts, the online restriction can seriously limit the ability of an algorithm to produce good solutions. Perhaps the first limitation of this type to be proved was Yao's theorem [24] that no online algorithm $A$ for bin packing can have $R^{\infty}_{A} < 1.5$. The bound has since been improved to the following.

**Theorem 6** ([23]) *If A is an online algorithm for bin packing, then* $R^{\infty}_{A} \ge 1.540\ldots$

Here the exact value of the lower bound is the solution to a complicated linear program.

Yao's paper also presented an online algorithm *Revised First Fit* (RFF) that had $R^{\infty}_{RFF} = 5/3 = 1.666\ldots$ and hence got closer to this lower bound than FF and BF. This algorithm worked by dividing the items into four classes based on size and index, and then using different packing rules (and packings) for each class. Subsequent algorithms improved on this by going to more and more classes. The current champion is the online *Harmonic++* algorithm (H++) of [21]:

**Theorem 7** ([21]) $R^{\infty}_{H++} \le 1.58889$.

**Bounded-Space Algorithms** The NF algorithm, in addition to being online, has another property worth noting: no more than a constant number of partially filled bins remain open to receive additional items at any given time. In the case of NF, the constant is 1 – only the last partially filled bin can receive additional items. Bounding the number of open bins may be necessary in some applications, such as packing trucks on loading docks. The bounded-space constraint imposes additional limits on algorithmic behavior however.

**Theorem 8** ([17]) *For any online bounded-space algorithm A,* $R^{\infty}_{A} \ge 1.691\ldots$.

The constant $1.691\ldots$ arises in many other bin packing contexts. It is commonly denoted by $h_\infty$ and equals $\sum_{i=1}^{\infty}(1/t_i)$, where $t_1 = 1$ and, for $i > 1$, $t_i = t_{i-1}(t_{i-1}+1)$.

The lower bound in Theorem 8 is tight, owing to the existence of the *Harmonic*$_k$ algorithms ($\mathrm{H}_k$) of [17]. $\mathrm{H}_k$ is a class-based algorithm in which the items are divided into classes $C_h$, $1 \le h \le k$, with $C_k$ consisting of all items with size $1/k$ or smaller, and $C_h$, $1 \le h < k$, consisting of all $a_i$ with $1/(h+1) < s(a_i) \le 1/h$. The items in each class are then packed by NF into a separate packing devoted just to that class. Thus, at most $k$ bins are open at any time. In [17] it was shown that $\lim_{k\to\infty} R^{\infty}_{H_k} = h_\infty = 1.691\ldots$. This is even better than the asymptotic worst-case ratio of 1.7 for the unbounded-space algorithms FF and BF, although it should be noted that the bounded-space variant of BF in which all but the two most-full bins are closed also has $R^{\infty}_{A} = 1.7$ [8].

### Average-Case Behavior

**Continuous Distributions** Bin packing also served as an early test bed for studying the average-case behavior of approximation algorithms. Suppose $F$ is a distribution on $(0, 1]$ and $L_n$ is a list of $n$ items with item sizes chosen independently according to $F$. For any list $L$, let $s(L)$ denote the lower bound on $OPT(L)$ obtained by summing the sizes of all the items in $L$. Then define

$$ER^n_A(F) = E\left[A(L_n)/OPT(L_n)\right]\ ,$$
$$ER^{\infty}_A(F) = \limsup_{n\to\infty} ER^n_A$$
$$EW^n_A(F) = E\left[A(L_n) - s(L_n)\right]$$

The last definition is included since $ER^{\infty}_A(F) = 1$ occurs frequently enough that finer distinctions are meaningful. For example, in the early 1980s, it was observed that for the distribution $F = U(0, 1]$ in which item sizes are uniformly distributed on the interval $(0, 1]$, $ER^{\infty}_{FFD}(F) = ER^{\infty}_{BFD}(F) = 1$, as a consequence of the following more-detailed results.

**Theorem 9** ([16,20]) *For $A \in \{FFD, BFD, OPT\}$, $EW_A^n(U(0,1]) = \Theta(\sqrt{n})$.*

Somewhat surprisingly, it was later discovered that the online FF and BF algorithms also had sublinear expected waste, and hence $ER_A^\infty(U(0,1]) = 1$.

**Theorem 10** ([5,19])

$$EW_{FF}^n(U(0,1]) = \Theta(n^{2/3})$$
$$EW_{BF}^n(U(0,1]) = \Theta(n^{1/2}\log^{3/4} n)$$

This good behavior does not, however, extend to the bounded-space algorithms NF and H$_k$:

**Theorem 11** ([6,18])

$$ER_{NF}^\infty(U(0,1]) = 4/3 = 1.333\ldots$$
$$\lim_{k\to\infty} ER_{H_k}(U(0,1]) = \pi^2/3 - 2 = 1.2899\ldots$$

All the above results except the last two exploit the fact that the distribution $U(0,1]$ is symmetric about 1/2, and hence an optimal packing consists primarily of two-item bins, with items of size $s > 1/2$ matched with smaller items of size very close to $1 - s$. The proofs essentially show that the algorithms in question do good jobs of constructing such matchings. In practice, however, there will clearly be situations where more than matching is required. To model such situations, researchers first turned to the distributions $U(0,b]$, $0 < b < 1$, where item sizes are chosen uniformly from the interval $(0,b]$. Simulations suggest that such distributions make things worse for the online algorithms FF and BF, which appear to have $ER_A^\infty(U(0,b]) > 1$ for all $b \in (0,1)$. Surprisingly, they make things better for FFD and BFD (and the optimal packing).

**Theorem 12** ([2,14])
1. *For $0 < b \le 1/2$ and $A \in \{FFD, BFD\}$, $EW_A^n(U(0,b]) = O(1)$.*
2. *For $1/2 < b < 1$ and $A \in \{FFD, BFD\}$, $EW_A^n(U(0,b]) = \Theta(n^{1/3})$.*
3. *For $0 < b < 1$, $EW_{OPT}^n(U(0,b]) = O(1)$.*

**Discrete Distributions** In many applications, the item sizes come from a finite set, rather than a continuous distribution like those discussed above. Thus, recently the study of average-case behavior for bin packing has turned to *discrete distributions*. Such a distribution is specified by a finite list $s_1, s_2, \ldots, s_d$ of rational sizes and for each $s_i$ a corresponding rational probability $p_i$. A remarkable result of Courcoubetis and Weber [7] says the following.

**Theorem 13** ([7]) *For any discrete distribution F, $EW_{OPT}^n(F)$ is either $\Theta(n)$, $\Theta(\sqrt{n})$, or O(1).*

The discrete analogue of the continuous distribution $U(0,b]$ is the distribution $U\{j,k\}$, where the sizes are $1/k, 2/k, \ldots, j/k$ and all the probabilities equal $1/j$. Simulations suggest that the behavior of FF and BF in the discrete case are qualitatively similar to the behavior in the continuous case, whereas the behavior of FFD and BFD is considerably more bizarre [3]. Of particular note is the distribution $F = U\{6,13\}$, for which $ER_{FFD}^\infty(F)$ is strictly greater than $ER_{FF}^\infty(F)$, in contrast to all the previously implied comparisons between the two algorithms.

For discrete distributions, however, the standard algorithms are all dominated by a new online algorithm called the *Sum-of-Squares* (SS) algorithm. Note that since the item sizes are all rational, they can be scaled so that they (and the bin size $B$) are all integral. Then at any given point in the operation of an online algorithm, the current packing can be summarized by giving, for each $h$, $1 \le h \le B$, the number $n_h$ of bins containing items of total size $h$. In SS, one packs each item so as to minimize $\sum_{h=1}^{B-1} n_h^2$.

**Theorem 14** ([9]) *For any discrete distribution F, the following hold.*
1. *If $EW_{OPT}^n(F) = \Theta(\sqrt{n})$, then $EW_{SS}^n(F) = \Theta(\sqrt{n})$.*
2. *If $EW_{OPT}^n(F) = O(1)$, then $EW_{SS}^n(F) \in \{O(1), \Theta(\log n)\}$.*

*In addition, a simple modification to SS can eliminate the $\Theta(\log n)$ case of condition 2.*

## Applications

There are many potential applications of one-dimensional bin packing, from packing bandwidth requests into fixed-capacity channels to packing commercials into station breaks. In practice, simple heuristics like FFD and BFD are commonly used.

## Open Problems

Perhaps the most fundamental open problem related to bin packing is the following. As observed above, there is a polynomial-time algorithm (KK) whose packings are within $O(\log^2(OPT))$ bins of optimal. Is it possible to do better? As far as is currently known, there could still be a polynomial-time algorithm that always gets within one bin of optimal, even if P $\neq$ NP.

## Experimental Results

Bin packing has been a fertile ground for experimental analysis, and many of the theorems mentioned above were first conjectured on the basis of experimental results. For example, the experiments reported in [1] inspired Theorems 10 and 12, and the experiments in [10] inspired Theorem 14.

## Cross References

▶ Approximation Schemes for Bin Packing

## Recommended Reading

1. Bentley, J.L., Johnson, D.S., Leighton, F.T., McGeoch, C.C.: An experimental study of bin packing. In: Proc. of the 21st Annual Allerton Conference on Communication, Control, and Computing, Urbana, University of Illinois, 1983 pp. 51–60
2. Bentley, J.L., Johnson, D.S., Leighton, F.T., McGeoch, C.C., McGeoch, L.A.: Some unexpected expected behavior results for bin packing. In: Proc. of the 16th Annual ACM Symposium on Theory of Computing, pp. 279–288. ACM, New York (1984)
3. Coffman Jr, E.G., Courcoubetis, C., Garey, M.R., Johnson, D.S., McGeoch, L.A., Shor, P.W., Weber, R.R., Yannakakis, M.: Fundamental discrepancies between average-case analyses under discrete and continuous distributions. In: Proc. of the 23rd Annual ACM Symposium on Theory of Computing, New York, 1991, pp. 230–240. ACM Press, New York (1991)
4. Coffman Jr., E.G., Garey, M.R., Johnson, D.S.: Approximation algorithms for bin-packing: A survey. In: Hochbaum, D. (ed.) Approximation Algorithms for NP-Hard Problems, pp. 46–93. PWS Publishing, Boston (1997)
5. Coffman Jr., E.G., Johnson, D.S., Shor, P.W., Weber, R.R.: Bin packing with discrete item sizes, part II: Tight bounds on first fit. Random Struct. Algorithms **10**, 69–101 (1997)
6. Coffman Jr., E.G., So, K., Hofri, M., Yao, A.C.: A stochastic model of bin-packing. Inf. Cont. **44**, 105–115 (1980)
7. Courcoubetis, C., Weber, R.R.: Necessary and sufficient conditions for stability of a bin packing system. J. Appl. Prob. **23**, 989–999 (1986)
8. Csirik, J., Johnson, D.S.: Bounded space on-line bin packing: Best is better than first. Algorithmica **31**, 115–138 (2001)
9. Csirik, J., Johnson, D.S., Kenyon, C., Orlin, J.B., Shor, P.W., Weber, R.R.: On the sum-of-squares algorithm for bin packing. J. ACM **53**, 1–65 (2006)
10. Csirik, J., Johnson, D.S., Kenyon, C., Shor, P.W., Weber, R.R.: A self organizing bin packing heuristic. In: Proc. of the 1999 Workshop on Algorithm Engineering and Experimentation. LNCS, vol. 1619, pp. 246–265. Springer, Berlin (1999)
11. Galambos, G., Woeginger, G.J.: Online bin packing – a restricted survey. ZOR Math. Methods Oper. Res. **42**, 25–45 (1995)
12. Johnson, D.S.: Near-Optimal Bin Packing Algorithms. Ph. D. thesis, Massachusetts Institute of Technology, Department of Mathematics, Cambridge (1973)
13. Johnson, D.S., Demers, A., Ullman, J.D., Garey, M.R., Graham, R.L.: Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM J. Comput. **3**, 299–325 (1974)
14. Johnson, D.S., Leighton, F.T., Shor, P.W., Weber, R.R.: The expected behavior of FFD, BFD, and optimal bin packing under $U(0, \alpha])$ distributions (in preparation)
15. Karmarkar, N., Karp, R.M.: An efficient approximation scheme for the one-dimensional bin packing problem. In: Proc. of the 23rd Annual Symposium on Foundations of Computer Science, pp. 312–320. IEEE Computer Soc, Los Alamitos, CA (1982)
16. Knödel, W.: A bin packing algorithm with complexity $O(n \log n)$ in the stochastic limit. In: Proc. 10th Symp. on Mathematical Foundations of Computer Science. LNCS, vol. 118, pp. 369–378. Springer, Berlin (1981)
17. Lee, C.C., Lee, D.T.: A simple on-line packing algorithm. J. ACM **32**, 562–572 (1985)
18. Lee, C.C., Lee, D.T.: Robust on-line bin packing algorithms. Tech. Rep. Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL (1987)
19. Leighton, T., Shor, P.: Tight bounds for minimax grid matching with applications to the average case analysis of algorithms. Combinatorica **9** 161–187 (1989)
20. Lueker, G.S.: An average-case analysis of bin packing with uniformly distributed item sizes. Tech. Rep. Report No 181, Dept. of Information and Computer Science, University of California, Irvine, CA (1982)
21. Seiden, S.S.: On the online bin packing problem. J. ACM **49**, 640–671 (2002)
22. Ullman, J.D.: The performance of a memory allocation algorithm. Tech. Rep. 100, Princeton University, Princeton, NJ (1971)
23. van Vliet, A.: An improved lower bound for on-line bin packing algorithms. Inf. Proc. Lett. **43**, 277–284 (1992)
24. Yao, A.C.: New algorithms for bin packing. J. ACM **27**, 207–227 (1980)

# Block Edit Distance

▶ Edit Distance Under Block Operations

# Block-Sorting Data Compression

▶ Burrows–Wheeler Transform

# Boolean Formulas

▶ Learning Automata

# Boolean Satisfiability

▶ Exact Algorithms for General CNF SAT

# Boosting Textual Compression

## 2005; Ferragina, Giancarlo, Manzini, Sciortino

PAOLO FERRAGINA[1], GIOVANNI MANZINI[2]

[1] Department of Computer Science, University of Pisa, Pisa, Italy

[2] Department of Computer Science, University of Eastern Piedmont, Alessandria, Italy

## Keywords and Synonyms

High-order compression models; Context-aware compression

## Problem Definition

Informally, a boosting technique is a method that, when applied to a particular class of algorithms, yields improved algorithms. The improvement must be provable and well defined in terms of one or more of the parameters characterizing the algorithmic performance. Examples of boosters can be found in the context of Randomized Algorithms (here, a booster allows one to turn a BPP algorithm into an RP one [6]) and Computational Learning Theory (here, a booster allows one to improve the prediction accuracy of a weak learning algorithm [10]). The problem of Compression Boosting consists of designing a technique that improves the compression performance of a wide class of algorithms. In particular, the results of Ferragina et al. provide a general technique for turning a compressor that uses no context information into one that always uses the best possible context.

The classic Huffman and Arithmetic coding algorithms [1] are examples of *statistical* compressors which typically encode an input symbol according to its *overall* frequency in the data to be compressed.[1] This approach is efficient and easy to implement but achieves poor compression. The compression performance of statistical compressors can be improved by adopting *higher*-order models that obtain better estimates for the frequencies of the input symbols. The PPM compressor [9] implements this idea by collecting (the frequency of) all symbols which follow *any* $k$-long context, and by compressing them via Arithmetic coding. The length $k$ of the context is a parameter of the algorithm that depends on the data to be compressed: it is different if one is compressing English text, a DNA sequence, or an XML document. There exist other examples of sophisticated compressors that use context information in an *implicit* way, such as Lempel–Ziv and Burrows–Wheeler compressors [9]. All these context-aware algorithms are effective in terms of compression performance, but are usually rather complex to implement and difficult to analyze.

Applying the boosting technique of Ferragina et al. to Huffman or Arithmetic Coding yields a new compression algorithm with the following features: (*i*) the new algorithm uses the boosted compressor as a black box, (*ii*) the new algorithm compresses in a PPM-like style, automatically choosing the *optimal* value of $k$, (*iii*) the new algorithm has essentially the same time/space asymptotic performance of the boosted compressor. The following sections give a precise and formal treatment of the three properties (*i*)–(*iii*) outlined above.

[1]In their dynamic versions these algorithms consider the frequency of a symbol in the already scanned portion of the input.

## Key Results

### Notation: The Empirical Entropy

Let $s$ be a string over the alphabet $\Sigma = \{a_1, \ldots, a_h\}$ and, for each $a_i \in \Sigma$, let $n_i$ be the number of occurrences of $a_i$ in $s$. The *0th order empirical entropy* of the string $s$ is defined as $H_0(s) = -\sum_{i=1}^{h}(n_i/|s|)\log(n_i/|s|)$, where it is assumed that all logarithms are taken to the base 2 and $0 \log 0 = 0$. It is well known that $H_0$ is the maximum compression one can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol. Greater compression is achievable if the codeword of a symbol depends on the $k$ symbols following it (namely, its *context*).[2] Let us define $w_s$ as the string of single symbols immediately preceding the occurrences of $w$ in $s$. For example, for $s = \texttt{bcabcabdca}$ it is $\texttt{ca}_s = \texttt{bbd}$. The value

$$H_k(s) = \frac{1}{|s|} \sum_{w \in \Sigma^k} |w_s|\, H_0(w_s) \tag{1}$$

is the $k$-th order empirical entropy of $s$ and is a lower bound to the compression one can achieve using codewords which only depend on the $k$ symbols immediately following the one to be encoded.

*Example 1* Let $s = \texttt{mississippi}$. For $k = 1$ it is $\texttt{i}_s = \texttt{mssp}$, $\texttt{s}_s = \texttt{isis}$, $\texttt{p}_s = \texttt{ip}$. Hence,

$$\begin{aligned} H_1(s) &= \frac{4}{11} H_0(\texttt{mssp}) + \frac{4}{11} H_0(\texttt{isis}) + \frac{2}{11} H_0(\texttt{ip}) \\ &= \frac{6}{11} + \frac{4}{11} + \frac{2}{11} = \frac{12}{11}\,. \end{aligned}$$

Note that the empirical entropy is defined for any string and can be used to measure the performance of compression algorithms without any assumption on the input source. Unfortunately, for some (highly compressible) strings, the empirical entropy provides a lower bound that is too conservative. For example, for $s = \texttt{a}^n$ it is $|s|\, H_k(s) = 0$ for any $k \geq 0$. To better deal with highly

[2]In data compression it is customary to define the context looking at the symbols *preceding* the one to be encoded. The present entry uses the non-standard "forward" contexts to simplify the notation of the following sections. Note that working with "forward" contexts is equivalent to working with the traditional "backward" contexts on the string $s$ reversed (see [3] for details).

compressible strings [7] introduced the notion of 0*th order modified empirical* entropy $H_0^*(s)$ whose property is that $|s|H_0^*(s)$ is at least equal to the number of bits needed to write down the length of $s$ in binary. The *kth order modified empirical entropy* $H_k^*$ is then defined in terms of $H_0^*$ as the maximum compression one can achieve by looking at *no more than* $k$ symbols following the one to be encoded.

## The Burrows–Wheeler Transform

Given a string $s$, the Burrows–Wheeler transform [2] (bwt) consists of three basic steps: (1) append to the end of $s$ a special symbol \$ smaller than any other symbol in $\Sigma$; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $s\$$, sorted in lexicographic order; (3) construct the transformed text $\hat{s} = \mathtt{bwt}(s)$ by taking the last column of $\mathcal{M}$ (see Fig. 1). In [2] Burrows and Wheeler proved that $\hat{s}$ is a permutation of $s$, and that from $\hat{s}$ it is possible to recover $s$ in $O(|s|)$ time.

To see the power of the bwt the reader should reason in terms of empirical entropy. Fix a positive integer $k$. The first $k$ columns of the bwt matrix contain, lexicographically ordered, all length-$k$ substrings of $s$ (and $k$ substrings containing the symbol \$). For any length-$k$ substring $w$ of $s$, the symbols immediately preceding every occurrence of $w$ in $s$ are grouped together in a set of consecutive positions of $\hat{s}$ since they are the last symbols of the rows of $\mathcal{M}$ prefixed by $w$. Using the notation introduced for defining $H_k$, it is possible to rephrase this property by saying that the symbols of $w_s$ are consecutive within $\hat{s}$, or equivalently, that $\hat{s}$ contains, as a substring, a permutation $\pi_w(w_s)$ of the string $w_s$.

*Example 2* Let $s = \mathtt{mississippi}$ and $k = 1$. Figure 1 shows that $\hat{s}[1,4] = \mathtt{pssm}$ is a permutation of $\mathtt{i}_s = \mathtt{mssp}$. In addition, $\hat{s}[6,7] = \mathtt{pi}$ is a permutation of $\mathtt{p}_s = \mathtt{ip}$, and $\hat{s}[8,11] = \mathtt{ssii}$ is a permutation of $\mathtt{s}_s = \mathtt{isis}$.

Since permuting a string does not change its (modified) 0th order empirical entropy (that is, $H_0(\pi_w(w_s)) = H_0(w_s)$), the Burrows–Wheeler transform can be seen as a tool for reducing the problem of compressing $s$ up to its $k$th order entropy to the problem of compressing *distinct portions* of $\hat{s}$ up to their 0*th order* entropy. To see this, assume partitioning of $\hat{s}$ into the substrings $\pi_w(w_s)$ by varying $w$ over $\Sigma^k$. It follows that $\hat{s} = \bigsqcup_{w\in\Sigma^k} \pi_w(w_s)$ where $\bigsqcup$ denotes the concatenation operator among strings.[3]

[3] In addition to $\sqcup_{w\in\Sigma^k} \pi_w(w_s)$, the string $\hat{s}$ also contains the last $k$ symbols of $s$ (which do not belong to any $w_s$) and the special symbol \$. For simplicity these symbols will be ignored in the following part of the entry.

By (1) it follows that

$$\sum_{w\in\Sigma^k} |\pi_w(w_s)|H_0(\pi_w(w_s)) = \sum_{w\in\Sigma^k} |w_s|H_0(w_s) = |s|H_k(s).$$

Hence, to compress $s$ up to $|s|\ H_k(s)$ it suffices to compress each substring $\pi_w(w_s)$ up to its 0th order empirical entropy. Note, however, that in the above scheme the parameter $k$ must be chosen in advance. Moreover, a similar scheme cannot be applied to $H_k^*$ which is defined in terms of contexts of length *at most* $k$. As a result, no efficient procedure is known for computing the partition of $\hat{s}$ corresponding to $H_k^*(s)$. The compression booster [3] is a natural complement to the bwt and allows one to compress any string $s$ up to $H_k(s)$ (or $H_k^*(s)$) simultaneously for all $k \geq 0$.

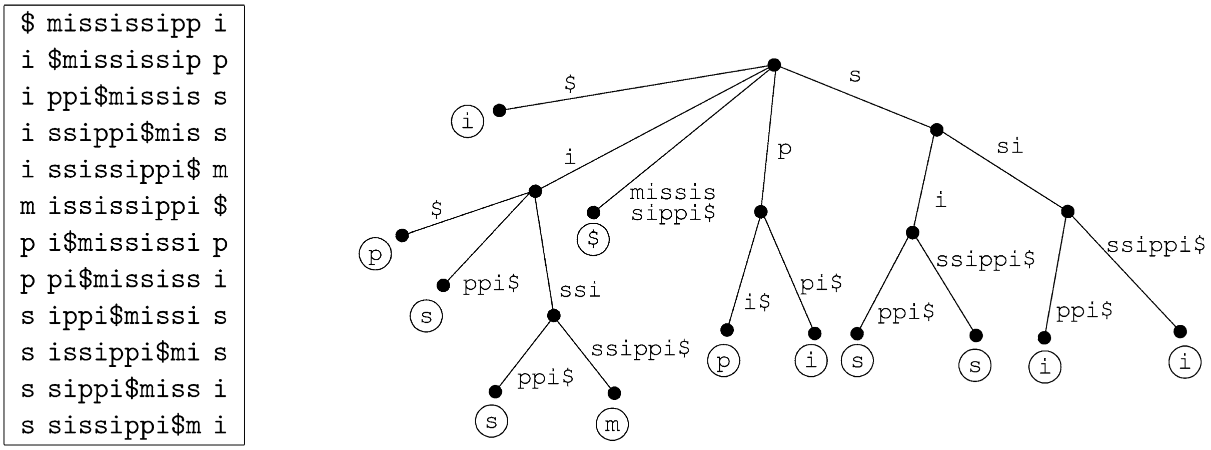
## The Compression Boosting Algorithm

A crucial ingredient of compression boosting is the relationship between the bwt matrix and the suffix tree data structure. Let $\mathcal{T}$ denote the suffix tree of the string $s\$$. $\mathcal{T}$ has $|s| + 1$ leaves, one per suffix of $s\$$, and edges labeled with substrings of $s\$$ (see Fig. 1). Any node $u$ of $\mathcal{T}$ has *implicitly associated* a substring of $s\$$, given by the concatenation of the edge labels on the downward path from the root of $\mathcal{T}$ to $u$. In that implicit association, the leaves of $\mathcal{T}$ correspond to the suffixes of $s\$$. Assume that the suffix tree edges are sorted lexicographically. Since each row of the bwt matrix is prefixed by one suffix of $s\$$ and rows are lexicographically sorted, the $i$th leaf (counting from the left) of the suffix tree corresponds to the $i$th row of the bwt matrix. Associate to the $i$th leaf of $\mathcal{T}$ the $i$th symbol of $\hat{s} = \mathtt{bwt}(s)$. In Fig. 1 these symbols are represented inside circles.

For any suffix tree node $u$, let $\hat{s}\langle u\rangle$ denote the substring of $\hat{s}$ obtained by concatenating, from left to right, the symbols associated to the leaves descending from node $u$. Of course $\hat{s}\langle root(\mathcal{T})\rangle = \hat{s}$. A subset $\mathcal{L}$ of $\mathcal{T}$'s nodes is called a *leaf cover* if every leaf of the suffix tree has a *unique* ancestor in $\mathcal{L}$. Any leaf cover $\mathcal{L} = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of $\mathcal{T}$. Because of the relationship between $\mathcal{T}$ and the bwt matrix this is also a partition of $\hat{s}$, namely $\{\hat{s}\langle u_1\rangle, \ldots, \hat{s}\langle u_p\rangle\}$.

*Example 3* Consider the suffix tree in Fig. 1. A leaf cover consists of all nodes of depth one. The partition of $\hat{s}$ induced by this leaf cover is $\{\mathtt{i}, \mathtt{pssm}, \$, \mathtt{pi}, \mathtt{ssii}\}$.

Let $C$ denote a function that associates to every string $x$ over $\Sigma \cup \{\$\}$ a positive real value $C(x)$. For any leaf cover $\mathcal{L}$, define its cost as $C(\mathcal{L}) = \sum_{u\in\mathcal{L}} C(\hat{s}\langle u\rangle)$. In other words, the cost of the leaf cover $\mathcal{L}$ is equal to the sum of the costs of the strings in the partition induced by $\mathcal{L}$. A leaf cover



```
$ mississipp i
i $mississip p
i ppi$missis s
i ssippi$mis s
i ssissippi$ m
m ississippi $
p i$mississi p
p pi$mississ i
s ippi$missi s
s issippi$mi s
s sippi$miss i
s sissippi$m i
```

**Boosting Textual Compression, Figure 1**
**The bwt matrix (*left*) and the suffix tree (*right*) for the string $s$ = `mississippi$`. The output of the bwt is the last column of the bwt matrix, i. e., $\hat{s}$ = bwt($s$) = `ipssm$pissii`**

$\mathcal{L}_{\min}$ is called *optimal* with respect to $C$ if $C(\mathcal{L}_{\min}) \le C(\mathcal{L})$, for any leaf cover $\mathcal{L}$.

Let $\mathbb{A}$ be a compressor such that, for any string $x$, its output size is bounded by $|x|H_0(x) + \eta|x| + \mu$ bits, where $\eta$ and $\mu$ are constants. Define the cost function $C_{\mathbb{A}}(x) = |x|H_0(x) + \eta|x| + \mu$. In [3] Ferragina et al. exhibit a linear-time greedy algorithm that computes the optimal leaf cover $\mathcal{L}_{\min}$ with respect to $C_{\mathbb{A}}$. The authors of [3] also show that, for any $k \ge 0$, there exists a leaf cover $\mathcal{L}_k$ of cost $C_{\mathbb{A}}(\mathcal{L}_k) = |s|\,H_k(s) + \eta|s| + O(|\Sigma|^k)$. These two crucial observations show that, if one uses $\mathbb{A}$ to compress each substring in the partition induced by the optimal leaf cover $\mathcal{L}_{\min}$, the total output size is bounded in terms of $|s|\,H_k(s)$, for any $k \ge 0$. In fact,

$$\begin{aligned}\sum_{u\in\mathcal{L}_{\min}} C_{\mathbb{A}}(\hat{s}\langle u\rangle) &= C_{\mathbb{A}}(\mathcal{L}_{\min}) \le C_{\mathbb{A}}(\mathcal{L}_k)\\ &= |s|\,H_k(s) + \eta|s| + O(|\Sigma|^k)\end{aligned}$$

In summary, boosting the compressor $\mathbb{A}$ over the string $s$ consists of three main steps:

1. Compute $\hat{s}$ = bwt($s$);
2. Compute the optimal leaf cover $\mathcal{L}_{\min}$ with respect to $C_{\mathbb{A}}$, and partition $\hat{s}$ according to $\mathcal{L}_{\min}$;
3. Compress each substring of the partition using the algorithm $\mathbb{A}$.

So the boosting paradigm reduces the design of effective compressors that use context information, to the (usually easier) design of 0th order compressors. The performance of this paradigm is summarized by the following theorem.

**Theorem 1 (Ferragina et al. 2005)** *Let $\mathbb{A}$ be a compressor that squeezes any string $x$ in at most $|x|H_0(x)+\eta|x|+\mu$ bits. The compression booster applied to $\mathbb{A}$ produces an output whose size is bounded by $|s|\,H_k(s) + \log|s| + \eta|s| + O(|\Sigma|^k)$ bits simultaneously for all $k \ge 0$. With respect to $\mathbb{A}$, the booster introduces a space overhead of $O(|s|\log|s|)$ bits and no asymptotic time overhead in the compression process.* □

A similar result holds for the modified entropy $H_k^*$ as well (but it is much harder to prove): Given a compressor $\mathbb{A}$ that squeezes any string $x$ in at most $\lambda|x|\,H_0^*(x) + \mu$ bits, the compression booster produces an output whose size is bounded by $\lambda|s|\,H_k^*(s) + \log|s| + O(|\Sigma|^k)$ bits, simultaneously for all $k \ge 0$. In [3] the authors also show that no compression algorithm, satisfying some mild assumptions on its inner working, can achieve a similar bound in which both the multiplicative factor $\lambda$ and the additive logarithmic term are dropped simultaneously. Furthermore [3] proposes an instantiation of the booster which compresses any string $s$ in at most $2.5|s|H_k^*(s) + \log|s| + O(|\Sigma|^k)$ bits. This bound is analytically superior to the bounds proven for the best existing compressors including Lempel–Ziv, Burrows–Wheeler, and PPM compressors.

## Applications

Apart from the natural application in data compression, compressor boosting has been used also to design Compressed Full-text Indexes [8].

## Open Problems

The boosting paradigm may be generalized as follows: Given a compressor $\mathbb{A}$, find a permutation $\mathcal{P}$ for the symbols of the string $s$ *and* a partitioning strategy such that the boosting approach, applied to them, minimizes the output size. These pages have provided convincing evidence that the Burrows–Wheeler Transform is an elegant and efficient permutation $\mathcal{P}$. Surprisingly enough, other classic Data Compression problems fall into this framework: Shortest Common Superstring (which is MAX-SNP hard), Run Length Encoding for a Set of Strings (which is polynomially solvable), LZ77 and minimum number of phrases (which is MAX-SNP-Hard). Therefore, the boosting approach is general enough to deserve further theoretical and practical attention [5].

## Experimental Results

An investigation of several compression algorithms based on boosting, and a comparison with other state-of-the-art compressors is presented in [4]. The experiments show that the boosting technique is more robust than other bwt-based approaches, and works well even with less effective 0th order compressors. However, these positive features are achieved using more (time and space) resources.

## Data Sets

The data sets used in [4] are available from http://www.mfn.unipmn.it/~manzini/boosting. Other data sets for compression and indexing are available at the Pizza&Chili site http://pizzachili.di.unipi.it/.

## URL to Code

The Compression Boosting page (http://www.mfn.unipmn.it/~manzini/boosting) contains the source code of all the algorithms tested in [4]. The code is organized in a highly modular library that can be used to boost any compressor even without knowing the bwt or the boosting procedure.

## Cross References

- ▶ Arithmetic Coding for Data Compression
- ▶ Burrows–Wheeler Transform
- ▶ Compressed Text Indexing
- ▶ Table Compression
- ▶ Tree Compression and Indexing

## Recommended Reading

1. Bell, T.C., Cleary, J.G., Witten, I.H.: Text compression. Prentice Hall, NJ (1990)
2. Burrows, M. Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Report 124, Digital Equipment Corporation (1994)
3. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM **52**, 688–713 (2005)
4. Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: Theory vs practice in bwt compression. In: Proc. 14th European Symposium on Algorithms (ESA). LNCS, vol. 4168, pp. 756–767. Springer, Berlin (2006)
5. Giancarlo, R., Restivo, A., Sciortino, M.: From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. Theor. Comput. Sci. **387**(3):236-248 (2007)
6. Karp, R., Pippenger, N., Sipser, M.: A Time-Randomness tradeoff. In: Proc. Conference on Probabilistic Computational Complexity, AMS, 1985, pp. 150–159
7. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**, 407–430 (2001)
8. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Comput. Surv. **39**(1) (2007)
9. Salomon, D.: Data Compression: the Complete Reference, 4th edn. Springer, London (2004)
10. Schapire, R.E.: The strength of weak learnability. Mach. Learn. **2**, 197–227 (1990)

# Branchwidth of Graphs

## 2003; Fomin, Thilikos

FEDOR FOMIN[1], DIMITRIOS THILIKOS[2]
[1] Department of Informatics, University of Bergen, Bergen, Norway
[2] Department of Mathematics, National and Kapodistrian University of Athens, Athens, Greece

## Keywords and Synonyms

Tangle Number

## Problem Definition

Branchwidth, along with its better-known counterpart, treewidth, are measures of the "global connectivity" of a graph.

### Definition

Let $G$ be a graph on $n$ vertices. A *branch decomposition* of $G$ is a pair $(T, \tau)$, where $T$ is a tree with vertices of degree 1 or 3 and $\tau$ is a bijection from the set of leaves of $T$ to the edges of $G$. The *order*, we denote it as $\alpha(e)$, of an edge $e$ in $T$ is the number of vertices $v$ of $G$ such that there are leaves $t_1, t_2$ in $T$ in different components of $T(V(T), E(T) - e)$ with $\tau(t_1)$ and $\tau(t_2)$ both containing $v$ as an endpoint.

The *width* of $(T, \tau)$ is equal to $\max_{e \in E(T)}\{\alpha(e)\}$, i. e. is the maximum order over all edges of $T$. The *branchwidth* of $G$ is the minimum width over all the branch decompositions of $G$ (in the case where $|E(G)| \leq 1$, then we define the branchwidth to be 0; if $|E(G)| = 0$, then $G$ has no branch decomposition; if $|E(G)| = 1$, then $G$ has a branch decomposition consisting of a tree with one vertex – the width of this branch decomposition is considered to be 0).

The above definition can be directly extended to hypergraphs where $\tau$ is a bijection from the leaves of $T$ to the hyperedges of $G$. The same definition can easily be extended to matroids.

Branchwidth was first defined by Robertson and Seymour in [25] and served as a main tool for their proof of Wagner's Conjecture in their Graph Minors series of papers. There, branchwidth was used as an alternative to the parameter of treewidth as it appeared easier to handle for the purposes of the proof. The relation between branchwidth and treewidth is given by the following result.

**Theorem 1 ([25])** *If $G$ is a graph, then* $\text{branchwidth}(G) \leq \text{treewidth}(G) + 1 \leq \lfloor 3/2\ \text{branchwidth}(G) \rfloor$.

The algorithmic problems related to branchwidth are of two kinds: first find fast algorithms computing its value and, second, use it in order to design fast dynamic programming algorithms for other problems.

## Key Results

### Algorithms for Branchwidth

Computing branchwidth is an NP-hard problem ([29]). Moreover, the problem remains NP-hard even if we restrict its input graphs to the class of split graphs or bipartite graphs [20].

On the positive side, branchwidth is computable in polynomial time on interval graphs [20,24], and circular arc graphs [21]. Perhaps the most celebrated positive result on branchwidth is an $O(n^2)$ algorithm for the branchwidth of planar graphs, given by Seymour and Thomas in [29]. In the same paper they also give an $O(n^4)$ algorithm to compute an optimal branch decomposition. (The running time of this algorithm has been improved to $O(n^3)$ in [18].) The algorithm in [29] is basically an algorithm for a parameter called carving width, related to telephone routing and the result for branchwidth follows from the fact that the branch width of a planar graph is half of the carving-width of its medial graph.

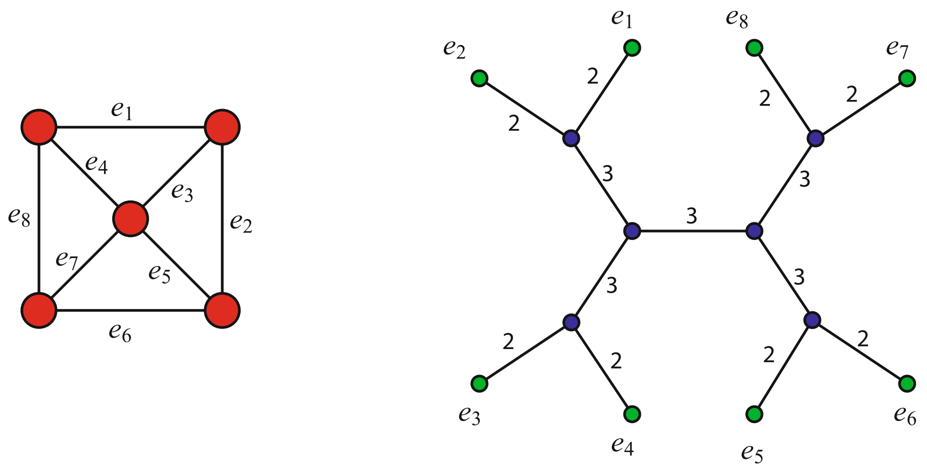
The algorithm for planar graphs [29] can be used to construct an approximation algorithm for branchwidth of some non-planar graphs. On graph classes excluding a single crossing graph as a minor branchwidth can be approximated within a factor of 2.25 [7] (a graph $H$ is a *minor* of a graph $G$ if $H$ can be obtained by a subgraph of $G$ after applying edge contractions). Finally, it follows from [13] that for every minor closed graph class, branchwidth can be approximated by a constant factor.

Branchwidth cannot increase when applying edge contractions or removals. According to the Graph Minors theory, this implies that, for any fixed $k$, there is a finite number of minor minimal graphs of branchwidth more than $k$ and we denote this set of graphs by $\mathcal{B}_k$. Checking whether a graph $G$ contains a fixed graph as a minor can be done in polynomial time [27]. Therefore, the knowledge of $\mathcal{B}_k$ implies the construction of a polynomial time algorithm for deciding whether $\text{branchwidth}(G) \leq k$, for any fixed $k$. Unfortunately $\mathcal{B}_k$ is known only for small values of $k$. In particular, $\mathcal{B}_0 = \{P_2\}, B_1 = \{P_4, K_3\}, \mathcal{B}_2 = \{K_4\}$ and $\mathcal{B}_3 = \{K_5, V_8, M_6, Q_3\}$ (here $K_r$ is a clique on $r$ vertices, $P_r$ is a path on $r$ edges, $V_8$ is the graph obtained by a cycle on 8 vertices if we connect all pairs of vertices with cyclic distance 4, $M_6$ is the octahedron, and $Q_3$ is the 3-dimensional cube). However, for any fixed $k$, one can construct a linear, on $n = |V(G)|$, algorithm that decides whether an input graph $G$ has branchwidth $\leq k$ and, if so, outputs the corresponding branch decomposition (see [3]). In technical terms, this implies that the problem of asking, for a given graph $G$, whether $\text{branchwidth}(G) \leq k$, parameterized by $k$ is fixed parameter tractable (i. e. belongs in the parameterized complexity class FPT). (See [12] for further references on parameterized algorithms and complexity.) The algorithm in [3] is complicated and uses the technique of characteristic sequences, which was also used in order to prove the analogous result for treewidth. For the particular cases where $k \leq 3$, simpler algorithms exist that use the "reduction rule" technique (see [4]). We stress that $\mathcal{B}_4$ remains unknown while several elements of it have been detected so far (including the dodecahedron and the icosahedron graphs). There is a number of algorithms that for a given $k$ in time $2^{O(k)} \cdot n^{O(1)}$ either decide that the branchwidth of a given graph is at least $k$, or construct a branch decomposition of width $O(k)$ (see [26]). These results can be generalized to compute the branchwidth of matroids and even more general parameters.

An exact algorithm for branchwidth appeared in [14]. Its complexity is $O((2 \cdot \sqrt{3})^n \cdot n^{O(1)})$. The algorithm exploits special properties of branchwidth (see also [24]).

In contrast to treewidth, edge maximal graphs of given branchwidth are not so easy to characterize (for treewidth there are just $k$-trees, i. e. chordal graphs with all maximal cliques of size $k + 1$). An algorithm for generating such graphs has been given in [23] and reveals several structural issues on this parameter.

It is known that a large number of graph theoretical problems can be solved in linear time when their inputs are restricted to graphs of small (i. e. fixed) treewidth or branchwidth (see [2]).



**Branchwidth of Graphs, Figure 1**
**Example of a graph and its branch decomposition of width 3**

Branchwidth appeared to be a useful tool in the design of exact subexponential algorithms on planar graphs and their generalizations. The basic idea behind this approach is very simple: Let $\mathcal{P}$ be a problem on graphs and $\mathcal{G}$ be a class of graphs such that

- For every graph $G \in \mathcal{G}$ of branchwidth at most $\ell$, the problem $\mathcal{P}$ can be solved in time $2^{c \cdot \ell} \cdot n^{\mathcal{O}(1)}$, where $c$ is a constant, and;
- For every graph $G \in \mathcal{G}$ on $n$ vertices a branch decomposition (not necessarily optimal) of $G$ of width at most $h(n)$ can be constructed in polynomial time, where $h(n)$ is a function.

Then for every graph $G \in \mathcal{G}$, the problem $\mathcal{P}$ can be solved in time $2^{c \cdot h(n)} \cdot n^{\mathcal{O}(1)}$. Thus, everything boils down to computations of constants $c$ and functions $h(n)$. These computations can be quite involved. For example, as was shown in [17], for every planar graph $G$ on $n$ vertices, the branchwidth of $G$ is at most $\sqrt{4.5n} < 2.1214\sqrt{n}$. For extensions of this bound to graphs embeddable on a surface of genus $g$, see [15].

Dorn [9] used fast matrix multiplication in dynamic programming to estimate the constants $c$ for a number of problems. For example, for the MAXIMUM INDEPENDENT SET problem, $c \leq \omega/2$, where $\omega < 2.376$ is the matrix product exponent over a ring, which implies that the INDEPENDENT SET problem on planar graphs is solvable in time $O(2^{2.52\sqrt{n}})$. For the MINIMUM DOMINATING SET problem, $c \leq 4$, thus implying that the branch decomposition method runs in time $O(2^{3.99\sqrt{n}})$. It appears that algorithms of running time $2^{O(\sqrt{n})}$ can be designed even for some of the "non-local" problems, such as the HAMILTONIAN CYCLE, CONNECTED DOMINATING SET, and STEINER TREE, for which no time $2^{O(\ell)} \cdot n^{\mathcal{O}(1)}$ algorithm on general graphs of branchwidth $\ell$ is known [11]. Here one needs special properties of some optimal planar branch decompositions, roughly speaking that every edge of $T$ corresponds to a disk on a plane such that all edges of $G$ corresponding to one component of $T - e$ are inside the disk and all other edges are outside. Some of the subexponential algorithms on planar graphs can be generalized for graphs embedded on surfaces [10] and, more generally, to graph classes that are closed under taking of minors [8].

A similar approach can be used for parameterized problems on planar graphs. For example, a parameterized algorithm that finds a dominating set of size $\leq k$ (or reports that no such set exists) in time $2^{\mathcal{O}(\sqrt{k})} n^{\mathcal{O}(1)}$ can be obtained based on the following observations: there is a constant $c$ such that every planar graph of branchwidth at least $c\sqrt{k}$ does not contain a dominating set of size at most $k$. Then for a given $k$ the algorithm computes an optimal branch decomposition of a palanar graph $G$ and if its width is more than $c\sqrt{k}$ concludes that $G$ has no dominating set of size $k$. Otherwise, find an optimal dominating set by performing dynamic programming in time $2^{\mathcal{O}(\sqrt{k})} n^{\mathcal{O}(1)}$. There are several ways of bounding a parameter of a planar graph in terms of its branchwidth or treewidth including techniques similar to Baker's approach from approximation algorithms [1], the use of separators, or by some combinatorial arguments, as shown in [16]. Another general approach of bounding the branchwidth of a planar graph by parameters, is based on the results of Robertson et al. [28] regarding quickly excluding a planar graph. This brings us to the notion of *bidimensionality* [6]. Parameterized algorithms based on branch decompositions can be generalized from planar graphs to graphs embedded on surfaces and to graphs excluding a fixed graph as a minor.

## Applications

See [5] for using branchwidth for solving TSP.

## Open Problems

1. It is known that any planar graph $G$ has branchwidth at most $\sqrt{4.5} \cdot \sqrt{|V(G)|}$ (or at most $\frac{3}{2} \cdot \sqrt{|E(G)| + 2}$) [17]. Is it possible to improver this upper bound? Any possible improvement would accelerate many of the known exact or parameterized algorithms on planar graphs that use dynamic programming on branch decompositions.
2. In contrast to treewidth, very few graph classes are known where branchwidth is computable in polynomial time. Find graphs classes where branchwidth can be computed or approximated in polynomial time.
3. Find $\mathcal{B}_k$ for values of $k$ bigger than 3. The only structural result on $\mathcal{B}_k$ is that its planar elements will be either self-dual or pairwise-dual. This follows from the fact that dual planar graphs have the same branchwidth [29,16].
4. Find an exact algorithm for branchwidth of complexity $O^*(2^n)$ (the notation $O^*()$ assumes that we drop the non-exponential terms in the classic $O()$ notation).
5. The dependence on $k$ of the linear time algorithm for branchwidth in [3] is huge. Find an $2^{O(k)} \cdot n^{O(1)}$ step algorithm, deciding whether the branchwidth of an $n$-vertex input graph is at most $k$.

## Cross References

► Bidimensionality
► Treewidth of Graphs

## Recommended Reading

1. Alber, J., Bodlaender, H.L., Fernau, H., Kloks, T., Niedermeier, R.: Fixed parameter algorithms for dominating set and related problems on planar graphs. Algorithmica **33**, 461–493 (2002)
2. Arnborg, S.: Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey. BIT **25**, 2–23 (1985)
3. Bodlaender, H.L., Thilikos, D.M.: Constructive linear time algorithms for branchwidth. In: Automata, languages and programming (Bologna, 1997). Lecture Notes in Computer Science, vol. 1256, pp. 627–637. Springer, Berlin (1997)
4. Bodlaender, H.L., Thilikos, D.M.: Graphs with branchwidth at most three. J. Algorithms **32**, 167–194 (1999)
5. Cook, W., Seymour, P.D.: Tour merging via branch-decomposition. Inf. J. Comput. **15**, 233–248 (2003)
6. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Bidimensional parameters and local treewidth. SIAM J. Discret. Math. **18**, 501–511 (2004)
7. Demaine, E.D., Hajiaghayi, M.T., Nishimura, N., Ragde, P., Thilikos, D. M.: Approximation algorithms for classes of graphs excluding single-crossing graphs as minors. J. Comput. Syst. Sci. **69**, 166–195 (2004)
8. Dorn, F., Fomin, F.V., Thilikos, D.M.: Subexponential algorithms for non-local problems on $H$-minor-free graphs. In: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2008). pp. 631–640. Society for Industrial and Applied Mathematics, Philadelphia (2006)
9. Dorn, F.: Dynamic programming and fast matrix multiplication. In: Proceedings of the 14th Annual European Symposium on Algorithms (ESA 2006). Lecture Notes in Computer Science, vol. 4168 , pp. 280–291. Springer, Berlin (2006)
10. Dorn, F., Fomin, F.V., Thilikos, D.M.: Fast subexponential algorithm for non-local problems on graphs of bounded genus. In: Proceedings of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006). Lecture Notes in Computer Science. Springer, Berlin (2005)
11. Dorn, F., Penninkx, E., Bodlaender, H., Fomin, F.V.: Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions. In: Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005). Lecture Notes in Computer Science, vol. 3669, pp. 95–106. Springer, Berlin (2005)
12. Downey, R.G., Fellows, M.R.: Parameterized complexity. In: Monographs in Computer Science. Springer, New York (1999)
13. Feige, U., Hajiaghayi, M., Lee, J.R.: Improved approximation algorithms for minimum-weight vertex separators. In: Proceedings of the 37th annual ACM Symposium on Theory of computing (STOC 2005), pp. 563–572. ACM Press, New York (2005)
14. Fomin, F.V., Mazoit, F., Todinca, I.: Computing branchwidth via efficient triangulations and blocks. In: Proceedings of the 31st Workshop on Graph Theoretic Concepts in Computer Science (WG 2005). Lecture Notes Computer Science, vol. 3787, pp. 374–384. Springer, Berlin (2005)
15. Fomin, F.V., Thilikos, D.M.: Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speed-up. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004). Lecture Notes Computer Science, vol. 3142, pp. 581–592. Springer, Berlin (2004)
16. Fomin, F.V., Thilikos, D.M.: Dominating sets in planar graphs: Branch-width and exponential speed-up. SIAM J. Comput. **36**, 281–309 (2006)
17. Fomin, F.V., Thilikos, D.M.: New upper bounds on the decomposability of planar graphs. J. Graph Theor. **51**, 53–81 (2006)
18. Gu, Q.P., Tamaki, H.: Optimal branch-decomposition of planar graphs in O($n^3$) time. In: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005). Lecture Notes Computer Science, vol. 3580, pp. 373–384. Springer, Berlin (2005)
19. Gu, Q.P., Tamaki, H.: Branch-width, parse trees, and monadic second-order logic for matroids. J. Combin. Theor. Ser. B **96**, 325–351 (2006)
20. Kloks, T., Kratochvíl, J., Müller, H.: Computing the branchwidth of interval graphs. Discret. Appl. Math. **145**, 266–275 (2005)
21. Mazoit, F.: The branch-width of circular-arc graphs. In: 7th Latin American Symposium on Theoretical Informatics (LATIN 2006), 2006, pp. 727–736

22. Oum, S.I., Seymour, P.: Approximating clique-width and branch-width. J. Combin. Theor. Ser. B **96**, 514–528 (2006)
23. Paul, C., Proskurowski, A., Telle, J.A.: Generating graphs of bounded branchwidth. In: Proceedings of the 32nd Workshop on Graph Theoretic Concepts in Computer Science (WG 2006). Lecture Notes Computer Science, vol. 4271, pp. 205–216. Springer, Berlin (2006)
24. Paul, C., Telle, J.A.: New tools and simpler algorithms for branchwidth. In: Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), 2005 pp. 379–390
25. Robertson, N. Seymour, P.D.: Graph minors. X. Obstructions to tree-decomposition J. Combin. Theor. Ser. B **52**, 153–190 (1991)
26. Robertson, N. Seymour, P.D.: Graph minors. XII. Distance on a surface. J. Combin. Theor. Ser. B **64**, 240–272 (1995)
27. Robertson, N. Seymour, P.D.: Graph minors. XIII. The disjoint paths problem. J. Combin. Theor. Ser. B **63**, 65–110 (1995)
28. Robertson, N., Seymour, P.D., Thomas, R.: Quickly excluding a planar graph. J. Combin. Theor. Ser. B **62**, 323–348 (1994)
29. Seymour, P.D., Thomas, R.: Call routing and the ratcatcher. Combinatorica **14**, 217–241 (1994)

# Broadcasting in Geometric Radio Networks

**2001; Dessmark, Pelc**

Andrzej Pelc
Department of Computer Science,
University of Québec-Ottawa, Gatineau, QC, Canada

## Keywords and Synonyms

Wireless information dissemination in geometric networks

## Problem Definition

### The Model Overview

Consider a set of stations (nodes) modeled as points in the plane, labeled by natural numbers, and equipped with transmitting and receiving capabilities. Every node $u$ has a *range* $r_u$ depending on the power of its transmitter, and it can reach all nodes at distance at most $r_u$ from it. The collection of nodes equipped with ranges determines a directed graph on the set of nodes, called a *geometric radio network* (GRN), in which a directed edge $(uv)$ exists if node $v$ can be reached from $u$. In this case $u$ is called a *neighbor* of $v$. If the power of all transmitters is the same then all ranges are equal and the corresponding GRN is symmetric.

Nodes send messages in synchronous *rounds*. In every round every node acts either as a *transmitter* or as a *receiver*. A node gets a message in a given round, if and only if, it acts as a receiver and exactly one of its neighbors transmits in this round. The message received in this case is the one that was transmitted. If at least two neighbors of a receiving node $u$ transmit simultaneously in a given round, none of the messages is received by $u$ in this round. In this case it is said that a *collision* occurred at $u$.

### The Problem

Broadcasting is one of the fundamental network communication primitives. One node of the network, called the *source*, has to transmit a message to all other nodes. Remote nodes are informed via intermediate nodes, along directed paths in the network. One of the basic performance measures of a broadcasting scheme is the total time, i. e., the number of rounds it uses to inform all the nodes of the network.

For a fixed real $s \geq 0$, called the *knowledge radius*, it is assumed that each node knows the part of the network within the circle of radius $s$ centered at it, i. e., it knows the positions, labels and ranges of all nodes at distance at most $s$. The following problem is considered:

How the size of the knowledge radius influences deterministic broadcasting time in GRN?

### Terminology and Notation

Fix a finite set $R = \{r_1, \dots, r_\rho\}$ of positive reals such that $r_1 < \dots < r_\rho$. Reals $r_i$ are called *ranges*. A *node* $v$ is a triple $[l, (x, y), r_i]$, where $l$ is a binary sequence called the *label* of $v$, $(x, y)$ are coordinates of a point in the plane, called the *position* of $v$, and $r_i \in R$ is called the *range* of $v$. It is assumed that labels are consecutive integers 1 to $n$, where $n$ is the number of nodes, but all the results hold if labels are integers in the set $\{1, \dots, M\}$, where $M \in O(n)$. Moreover, it is assumed that all nodes know an upper bound $\Gamma$ on $n$, where $\Gamma$ is polynomial in $n$. One of the nodes is distinguished and called the *source*. Any set of nodes $C$ with a distinguished source, such that positions and labels of distinct nodes are different is called a *configuration*.

With any configuration $C$ the following directed graph $\mathcal{G}(C)$ is associated. Nodes of the graph are nodes of the configuration and a directed edge $(uv)$ exists in the graph, if and only if the distance between $u$ and $v$ does not exceed the range of $u$. (The word "distance" always means the geometric distance in the plane and not the distance in a graph.) In this case $u$ is called a neighbor of $v$. Graphs of the form $\mathcal{G}(C)$ for some configuration $C$ are called *geometric radio networks* (GRN). In what follows, only configurations $C$ such that in $\mathcal{G}(C)$ there exists a directed path from the source to any other node, are considered. If the size of the set $R$ of ranges is $\rho$, a resulting configuration and the corresponding GRN are called a $\rho$-configuration and

$\rho$-GRN, respectively. Clearly, all 1-GRN are symmetric graphs. $D$ denotes the *eccentricity* of the source in a GRN, i. e., the maximum length of all shortest paths in this graph from the source to all other nodes. $D$ is of order of the diameter if the graph is symmetric but may be much smaller in general. $\Omega(D)$ is an obvious lower bound on broadcasting time.

Given any configuration, fix a non-negative real $s$, called the *knowledge radius*, and assume that every node of $C$ has initial input consisting of all nodes whose positions are at distance at most $s$ from its own. Thus it is assumed that every node knows *a priori* labels, positions and ranges of all nodes within a circle of radius $s$ centered at it. All nodes also know the set $R$ of available ranges.

It is not assumed that nodes know any global parameters of the network, such as its size or diameter. The only global information that nodes have about the network is a polynomial upper bound on its size. Consequently, the broadcast process may be finished but no node needs to be aware of this fact. Hence the adopted definition of broadcasting time is the same as in [3]. An algorithm accomplishes broadcasting in $t$ rounds, if all nodes know the source message after round $t$, and no messages are sent after round $t$.

Only deterministic algorithms are considered. Nodes can transmit messages even before getting the source message, which enables preprocessing in some cases. The algorithms are *adaptive*, i. e., nodes can schedule their actions based on their local history. A node can obviously gain knowledge from previously obtained messages. There is, however, another potential way of acquiring information during the communication process. The availability of this method depends on what happens during a collision, i. e., when $u$ acts as a receiver and two or more neighbors of $u$ transmit simultaneously. As mentioned above, $u$ does not get any of the messages in this case. However, two scenarios are possible. Node $u$ may either hear nothing (except for the background noise), or it may receive *interference noise* different from any message received properly but also different from background noise. In the first case it is said that there is no *collision detection*, and in the second case – that collision detection is available (cf., e. g., [1]). A discussion justifying both scenarios can be found in [1,7].

## Related Work

Broadcasting in geometric radio networks and some of their variations was considered, e. g., in [6,8,10,11]. In [11] the authors proved that scheduling optimal broadcasting is NP-hard even when restricted to such graphs, and gave an $O(n \log n)$ algorithm to schedule an optimal broadcast when nodes are situated on a line. In [10] broadcasting was considered in networks with nodes randomly placed on a line. In [8] the authors discussed fault-tolerant broadcasting in radio networks arising from regular locations of nodes on the line and in the plane, with reachability regions being squares and hexagons, rather than circles. Finally, in [6] broadcasting with restricted knowledge was considered but the authors studied only the special case of nodes situated on the line.

## Key Results

The results summarized below are based on the paper [5] of which [4] is a preliminary version.

### Arbitrary GRN in the Model Without Collision Detection

Clearly all upper bounds and algorithms are valid in the model with collision detection as well.

#### Large Knowledge Radius

**Theorem 1** *The minimum time to perform broadcasting in an arbitrary GRN with source eccentricity D and knowledge radius $s > r_\rho$ (or with global knowledge of the network) is $\Theta(D)$.*

This result yields a centralized $O(D)$ broadcasting algorithm when global knowledge of the GRN is available. This is in sharp contrast with broadcasting in arbitrary graphs, as witnessed by the graph from [9] which has bounded diameter but requires time $\Omega(\log n)$ for broadcasting.

**Knowledge Radius Zero** Next consider the case when knowledge radius $s = 0$, i. e., when every node knows only its own label, position and range. In this case it is possible to broadcast in time $O(n)$ for arbitrary GRN. It should be stressed that this upper bound is valid for arbitrary GRN, not only symmetric, unlike the algorithm from [3] designed for arbitrary symmetric graphs.

**Theorem 2** *It is possible to broadcast in arbitrary n-node GRN with knowledge radius zero in time $O(n)$.*

The above upper bound for GRN should be contrasted with the lower bound from [2,3] showing that some graphs require broadcasting time $\Omega(n \log n)$. Indeed, the graphs constructed in [2,3] and witnessing to this lower bound are not GRN. Surprisingly, this sharper lower bound does not require very unusual graphs. While counterexamples from [2,3] are not GRN, it turns out that the reason for a longer broadcasting time is really not the topology of the graph but the difference in knowledge available to nodes.

Recall that in GRN with knowledge radius 0 it is assumed that each node knows its own position (apart from its label and range): the upper bound $O(n)$ uses this geometric information extensively.

If this knowledge is not available to nodes (i. e., each node knows only its label and range) then there exists a family of GRN requiring broadcasting time $\Omega(n \log n)$. Moreover it is possible to show such GRN resulting from configurations with only 2 distinct ranges. (Obviously for 1-configurations this lower bound does not hold, as these configurations yield symmetric GRN and in [3] the authors showed an $O(n)$ algorithm working for arbitrary symmetric graphs).

**Theorem 3** *If every node knows only its own label and range (and does not know its position) then there exist $n$-node GRN requiring broadcasting time $\Omega(n \log n)$.*

### Symmetric GRN

**The Model with Collision Detection** In the model with collision detection and knowledge radius zero optimal broadcast time is established by the following pair of results.

**Theorem 4** *In the model with collision detection and knowledge radius zero it is possible to broadcast in any $n$-node symmetric GRN of diameter $D$ in time $O(D + \log n)$.*

The next result is the lower bound $\Omega(\log n)$ for broadcasting time, holding for some GRN of diameter 2. Together with the obvious bound $\Omega(D)$ this matches the upper bound from Theorem 4.

**Theorem 5** *For any broadcasting algorithm with collision detection and knowledge radius zero, there exist $n$-node symmetric GRN of diameter 2 for which this algorithm requires time $\Omega(\log n)$.*

**The Model Without Collision Detection** For the model without collision detection. it is possible to maintain complexity $O(D + \log n)$ of broadcasting. However, a stronger assumption concerning knowledge radius is needed: it is no longer 0, but positive, although arbitrarily small.

**Theorem 6** *In the model without collision detection, it is possible to broadcast in any $n$-node symmetric GRN of diameter $D$ in time $O(D + \log n)$, for any positive knowledge radius.*

## Applications

The radio network model is applicable to wireless networks using a single frequency. The specific model of geometric radio networks described in Sect. "Problem Definition" is applicable to wireless networks where stations are located in a relatively flat region without large obstacles (natural or human made), e. g., in the sea or a desert, as opposed to a large city or a mountain region. In such a terrain, the signal of a transmitter reaches receivers at the same distance in all directions, i. e., the set of potential receivers of a transmitter is a disc.

## Open Problems

1. Is it possible to broadcast in time $o(n)$ in arbitrary $n$-node GRN with eccentricity $D$ sublinear in $n$, for knowledge radius zero?
   Note: in view of Theorem 2 it is possible to broadcast in time $O(n)$.
2. Is it possible to broadcast in time $O(D + \log n)$ in all symmetric $n$-node GRN with eccentricity $D$, without collision detection, when knowledge radius is zero?
   Note: in view of Theorems 4 and 6, the answer is positive if either collision detection or a positive (even arbitrarily small) knowledge radius is assumed.

## Cross References

► Deterministic Broadcasting in Radio Networks
► Randomized Broadcasting in Radio Networks
► Randomized Gossiping in Radio Networks
► Routing in Geometric Networks

### Acknowledgments

Research partially supported by NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

## Recommended Reading

1. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time complexity of broadcast in radio networks: an exponential gap between determinism and randomization. J. Comput. Syst. Sci. **45**, 104–126 (1992)
2. Bruschi, D., Del Pinto, M.: Lower bounds for the broadcast problem in mobile radio networks. Distrib. Comput. **10**, 129–135 (1997)
3. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in ad hoc radio networks. Distrib. Comput. **15**, 27–38 (2002)
4. Dessmark, A., Pelc, A.: Tradeoffs between knowledge and time of communication in geometric radio networks. Proc. 13th Ann. ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 59–66, Crete Greece, July 3–6, 2001
5. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. J. Discret. Algorithms **5**, 187–201 (2007)

6. Diks, K., Kranakis, E., Krizanc, D., Pelc, A.: The impact of knowledge on broadcasting time in linear radio networks. Theor. Comput. Sci. **287**, 449–471 (2002)
7. Gallager, R.: A Perspective on Multiaccess Channels. IEEE Trans. Inf. Theory **31**, 124–142 (1985)
8. Kranakis, E., Krizanc, D., Pelc, A.: Fault-tolerant broadcasting in radio networks. J. Algorithms **39**, 47–67 (2001)
9. Pelc, A., Peleg, D.: Feasibility and complexity of broadcasting with random transmission failures. Proc. 24th Ann. ACM Symposium on Principles of Distributed Computing (PODC), pp. 334–341, Las Vegas, July 17–20 2005
10. Ravishankar, K., Singh, S.: Broadcasting on [0, *L*]. Discret. Appl. Math. **53**, 299–319 (1994)
11. Sen, A., Huson, M. L.: A New Model for Scheduling Packet Radio Networks. Proc. 15th Annual Joint Conference of the IEEE Computer and Communication Societies (IEEE INFOCOM'96), pp. 1116–1124, San Francisco, 24–28 March, 1996

# B-trees

## 1972; Bayer, McCreight

JAN VAHRENHOLD
Faculty of Computer Science,
Dortmund University of Technology,
Dortmund, Germany

## Keywords and Synonyms

Multiway Search Trees

## Problem Definition

This problem is concerned with storing a linearly ordered set of elements such that the DICTIONARY operations FIND, INSERT, and DELETE can be performed efficiently.

In 1972, Bayer and McCreight introduced the class of B-trees as a possible way of implementing an "index for a dynamically changing random access file" [6, p. 173]. B-trees have received considerable attention both in the database and in the algorithms community ever since; a prominent witness to their immediate and widespread acceptance is the fact that the authoritative survey on B-trees authored by Comer [9] appeared as soon as 1979 and, already at that time, referred to the B-tree data structure as the "ubiquitous B-tree".

### Notations

A *B-tree* is a multiway search tree defined as follows (the definition of Bayer and McCreight [6] is restated according to Knuth [16, Sect. 6.2.4] and Cormen et al. [10, Chap. 18.1]):

**Definition 1** Let $m \geq 3$ be a positive integer. A tree $T$ is a *B-tree* of degree $m$ if it is either empty or fulfills the following properties:

1. All leaves of $T$ appear on the same level of $T$.
2. Every node of $T$ has at most $m$ children.
3. Every node of $T$, except for the root and the leaves, has at least $m/2$ children.
4. The root of $T$ is either a leaf or has at least two children.
5. An internal node with $k$ children $c_1[\nu], \dots, c_k[\nu]$ stores $k-1$ keys, and a leaf stores between $m/2-1$ and $m-1$ keys. The keys $\text{key}_i[\nu]$, $1 \leq i \leq k-1$, of a node $\nu \in T$ are maintained in sorted order, i. e. $\text{key}_1[\nu] \leq \dots \leq \text{key}_{k-1}[\nu]$.
6. If $\nu$ is an internal node of $T$ with $k$ children $c_1[\nu], \dots, c_k[\nu]$, the $k-1$ keys $\text{key}_1[\nu], \dots, \text{key}_{k-1}[\nu]$ of $\nu$ separate the range of keys stored in the subtrees rooted at the children of $\nu$. If $x_i$ is any key stored in the subtree rooted at $c_i[\nu]$, the following holds:

$$x_1 \leq \text{key}_1[\nu] \leq x_2 \leq \text{key}_2[\nu] \leq \dots \leq x_{k-1} \leq \text{key}_{k-1}[\nu] \leq x_k\,.$$

To search a B-tree for a given key $x$, the algorithm starts with the root of the tree being the current node. If $x$ matches one of the current node's keys, the search terminates successfully. Otherwise, if the current node is a leaf, the search terminates unsuccessfully. If the current node's keys do not contain $x$ and if the current node is not a leaf, the algorithm identifies the unique subtree rooted at child of the current node that may contain $x$ and recurses on this subtree. Since the keys of a node guide the search process, they are also referred to as *routing elements*.

### Variants and Extensions

Knuth [16] defines a $B^*$*-tree* to be a B-tree where Property 3 in Definition 1 is modified such that every node (except for the root) contains at least $2m/3$ keys.

A $B^+$*-tree* is a leaf-oriented B-tree, i. e. a B-tree that stores the keys in the leaves only. Additionally, the leaves are linked in left-to-right order to allow for fast sequential traversal of the keys stored in the tree. In a leaf-oriented tree, the routing elements usually are copies of certain keys stored in the leaves ($\text{key}_i[\nu]$ can be set to be the largest key stored in the subtree rooted at $c_i[\nu]$), but any set of routing elements that fulfills Properties 5 and 6 of Definition 1 can do as well.

Huddleston and Mehlhorn [13] extended Definition 1 to describe a more general class of multiway search trees that includes the class of B-trees as a special case. Their class of so-called $(a,b)$*-trees* is parametrized by two integers $a$ and $b$ with $a \geq 2$ and $2a-1 \leq b$. Property 2 of

Definition 1 is modified to allow each node to have up to $b$ children and Property 3 is modified to require that, except for the root and the leaves, every node of an $(a, b)$-tree has at least $a$ children. All other properties of Definition 1 remain unchanged for $(a, b)$-trees. Usually, $(a, b)$-trees are implemented as leaf-oriented trees.

By the above definitions, a B-tree is a $(b/2, b)$-tree (if $b$ is even) or an $(a, 2a - 1)$-tree (if $b$ is odd). The subtle difference between even and odd maximum degree becomes relevant in an important amortization argument of Huddleston and Mehlhorn (see below) where the inequality $b \geq 2a$ is required to hold. This amortization argument actually caused $(a, b)$-trees with $b \geq 2a$ to be given a special name: *weak* B-trees [13].

## Update Operations

An INSERT operation on an $(a, b)$-tree first tries to locate the key $x$ to be inserted. After an unsuccessful search that stops at some leaf $\ell$, $x$ is inserted into $\ell$'s set of keys. If $\ell$ becomes too full, i. e. contains more than $b$ elements, two approaches are possible to resolve this *overflow* situation: (1) the node $\ell$ can be split around its median key into two nodes with at least $a$ keys each or (2) the node $\ell$ can have some of its keys be distributed to its left or right siblings (if this sibling has enough space to accommodate the new keys). In the first case, a new routing element separating the keys in the two new subtrees of $\ell$'s parent $\mu$ has to be inserted into the key set of $\mu$, and in the second case, the routing element in $\mu$ separating the keys in the subtree rooted at $\ell$ from the keys rooted at $\ell$'s relevant sibling needs to be updated. If $\ell$ was split, the node $\mu$ needs to be checked for a potential overflow due to the insertion of a new routing element, and the split may propagate all the way up to the root.

A DELETE operation also first locates the key $x$ to be deleted. If (in a non-leaf-oriented tree) $x$ resides in an internal node, $x$ is replaced by the largest key in the left subtree of $x$ (or the smallest key in the right subtree of $x$) which resides in a leaf and is deleted from there. In a leaf-oriented tree, keys are deleted from leaves only (the correctness of a routing element on a higher levels is not affected by this deletion). In any case, a DELETE operation may result in a leaf node $\ell$ containing less than $a$ elements. Again, there are two approaches to resolve this *underflow* situation: (1) the node $\ell$ is merged with its left or right sibling node or (2) keys from $\ell$'s left or right sibling node are moved to $\ell$ (unless the sibling node would underflow as a result of this). Both underflow handling strategies require updating the routing information stored in the parent of $\ell$ which (in the case of merging) may underflow itself. As with overflow handling, this process may propagate up to the root of the tree.

Note that the root of the tree can be split as a result of an INSERT operation and that it may disappear if the only two children of the root are merged to form the new root. This implies that B-trees grow and shrink at the top, and thus all leaves a guaranteed to appear on the same level of the tree (Property 1 of Definition 1).

## Key Results

Since B-trees are the premier index structure for external storage, the results given in this section are stated not only in the RAM-model of computation but also in the I/O-model of computation introduced by Aggarwal and Vitter [1]. In the I/O-model, not only the number $N$ of elements in the problem instance, but also the number $M$ of elements that simultaneously can be kept in main memory and the number $B$ of elements that fit into one disk block are (non-constant) parameters, and the complexity measure is the number of I/O-operations needed to solve a given problem instance. If B-trees are used in an external-memory setting, the degree $m$ of the B-tree is usually chosen such that one node fits into one disk block, i. e., $m \in \Theta(B)$, and this is assumed implicitly whenever the I/O-complexity of B-trees is discussed.

**Theorem 1** *The height of an N-key B-tree of degree* $m \geq 3$ *is bounded by* $\log_{\lceil m/2 \rceil}((N + 1)/2)$.

**Theorem 2 ([18])** *The storage utilization for large B-trees of high order under random insertions and deletions is approximately* $\ln 2 \approx 69\%$.

**Theorem 3** *A B-tree may be used to implement the abstract data type* Dictionary *such that the operations* Find, Insert, *and* Delete *on a set of N elements from a linearly ordered domain can be performed in* $\mathcal{O}(\log N)$ *time (with* $\mathcal{O}(\log_B N)$ *I/O-operations) in the worst case.*

*Remark 1* By threading the nodes of a B-tree, i. e. by linking the nodes according to their in-order traversal number, the operations PREV and NEXT can be performed in constant time (with a constant number of I/O-operations).

A (one-dimensional) *range query* asks for all keys that fall within a given query range (interval).

**Lemma 1** *A B-tree supports (one-dimensional) range queries with* $\mathcal{O}(\log N + K)$ *time complexity (*$\mathcal{O}(\log_B N + K/B)$ *I/O-complexity) in the worst case where K is the number of keys reported.*

Under the convention that each update to a B-tree results in a new "version" of the B-tree, a *multiversion* B-tree is a B-tree that allows for updates of the current version but also supports queries in earlier versions.

**Theorem 4 ([8])** *A multiversion B-tree can be constructed from a B-tree such that it is optimal with respect to the worst-case complexity of the* Find, Insert, *and* Delete *operations as well as to the worst-case complexity of answering range queries.*

## Applications

### Databases

One of the main reasons for the success of the B-tree lies in its close connection to databases: any implementation of Codd's relational data model (introduced incidentally in the same year as B-trees were invented) requires an efficient indexing mechanism to search and traverse relations that are kept on secondary storage. If this index is realized as a $B^+$-tree, all keys are stored in a linked list of leaves which is indexed by the top levels of the $B^+$-tree, and thus both efficient logarithmic searching and sequential scanning of the set of keys is possible.

Due to the importance of this indexing mechanism, a wide number of results on how to incorporate B-trees and their variants into database systems and how to formulate algorithms using these structures have be published in the database community. Comer [9] and Graefe [12] summarize early and recent results but due to the bulk of results even these summaries cannot be fully comprehensive. Also, B-trees have been shown to work well in the presence of concurrent operations [7], and Mehlhorn [17, p. 212] notes that they perform especially well if a top-down splitting approach is used. The details of this splitting approach may be found, e. g., in the textbook of Cormen et al. [10, Chap. 18.2].

### Priority Queues

A B-tree may be used to serve as an implementation of the abstract data type PRIORITYQUEUE since the smallest key always resides in the first slot of the leftmost leaf.

**Lemma 2** *An implementation of a priority queue that uses a B-tree supports the* Min *operation in $\mathcal{O}(1)$ time (with $\mathcal{O}(1)$ I/O-operations). All other operations (including* DecreaseKey*) have a time complexity of $\mathcal{O}(\log N)$ (an I/O-complexity of $\mathcal{O}(\log_B N)$) in the worst case.*

Mehlhorn [17, Sect. III, 5.3.1] examined B-trees (and, more general, $(a, b)$-trees with $a \geq 2$ and $b \geq 2a - 1$) in the context of *mergeable* priority queues. *Mergeable priority queues* are priority queues that additionally allow for concatenating and splitting priority queues. Concatenating priority queues for a set $S_1 \neq \emptyset$ and a set $S_2 \neq \emptyset$ is only defined if $\max\{x \mid x \in S_1\} < \min\{x \mid x \in S_2\}$ and results in a single priority queue for $S_1 \cup S_2$. Splitting a priority queue for a set $S_3 \neq \emptyset$ according to some $y \in \mathrm{dom}(S_3)$ results in a priority queue for the set $S_4 := \{x \in S_3 \mid x \leq y\}$ and a priority queue for the set $S_5 := \{x \in S_3 \mid x > y\}$ (one of these sets may be empty). Mehlhorn's result restated in the context of B-trees is as follows:

**Theorem 5 (Theorem 6, Sect. III, 5.3.1 in [7])** *If sets $S_1 \neq \emptyset$ and $S_2 \neq \emptyset$ are represented by a B-tree each then operation* Concatenate$(S_1, S_2)$ *takes time $\mathcal{O}(\log \max\{|S_1|, |S_2|\})$ (has an I/O-complexity of $\mathcal{O}(\log_B \max\{|S_1|, |S_2|\})$) and operation* Split$(S_1, y)$ *takes time $\mathcal{O}(\log |S_1|)$ (has an I/O-complexity of $\mathcal{O}(\log_B |S_1|)$). All bounds hold in the worst case.*

### Buffered Data Structures

Many applications (including sorting) that involve massive data sets allow for batched data processing. A variant of B-trees that exploits this relaxed problem setting is the so-called *buffer tree* proposed by Arge [3]. A *buffer tree* is a B-trees of degree $m \in \Theta(M/B)$ (instead of $m \in \Theta(B)$) where each node is assigned a buffer of size $\Theta(M)$. These buffers are used to collect updates and query requests that are passed further down the tree only if the buffer gets full enough to allow for cost amortization.

**Theorem 6 (Theorem 1 in [3])** *The total cost of an arbitrary sequence of $N$ intermixed* Insert *and* Delete *operations on an initially empty buffer tree is $\mathcal{O}(N/B \log_{M/B} N/B)$ I/O operations, that is the amortized I/O-cost of an operation is $\mathcal{O}(1/B \log_{M/B} N/B)$.*

As a consequence, $N$ elements can be sorted spending an optimal number of $\mathcal{O}(N/B \log_{M/B} N/B)$ I/O-operations by inserting them into a (leaf-oriented) buffer tree in a batched manner and then traversing the leaves. By the preceding discussion, buffer trees can also be used to implement (batched) priority queues in the external memory setting. Arge [3] extended his analysis of buffer trees to show that they also support DELETEMIN operations with an amortized I/O-cost of $\mathcal{O}(1/B \log_{M/B} N/B)$.

Since the degree of a buffer tree is too large to allow for efficient single-shot, i. e. non-batched operations, Arge et al. [4] discussed how buffers can be attached to (and later detached from) a multiway tree while at the same time keeping the degree of the base structure in $\Theta(B)$. Their discussion uses the R-tree index structure as a running example, the techniques presented, however, carry over to the B-tree. The resulting data structure is accessed through standard methods and additionally allows for batched update operations, e. g. *bulk loading*, and queries. The amortized I/O-complexity of all operations is analogous to the complexity of the buffer tree operations.

### B-trees as Base Structures

Several external memory data structures are derived from B-trees or use a B-tree as their base structure—see the survey by Arge [2] for a detailed discussion. One of these structures, the so-called *weight-balanced* B-tree is particularly useful as a base tree for building dynamic external data structures that have secondary structures attached to all (or some) of their nodes. The weight-balanced B-tree, developed by Arge and Vitter [5], is a variant of the B-tree that requires all subtrees of a node to have approximately, i. e., up to a small constant factor, the same number of leaves. Weight-balanced B-trees can be shown to have the following property:

**Theorem 7 ( [5])** *In a weight-balanced B-tree, rebalancing after an update operation is performed by splitting or merging nodes. When a rebalancing operation involves a node $v$ that is the root of a subtree with $w(v)$ leaves, at least $\Theta(w(v))$ update operations involving leaves below $v$ have to be performed before $v$ itself has to be rebalanced again.*

Using the above theorem, amortized bounds for maintaining secondary data structures attached to nodes of the base tree can be obtained—as long as each such structure can be updated with an I/O-complexity linear in the number of elements stored below the node it is attached to [2,5].

### Amortized Analysis

Most of the amortization arguments used for $(a, b)$-trees, buffer trees, and their relatives are based upon a theorem due to Huddleston and Mehlhorn [13, Theorem 3]. This theorem states that the total number of rebalancing operations in any sequence of $N$ intermixed insert and delete operations performed on an initially empty *weak* B-tree, i. e. an $(a, b)$-tree with $b \geq 2a$, is at most linear in $N$. This result carries over to buffer trees since they are $(M/4B, M/B)$-trees. Since B-trees are $(a, b)$-trees with $b = 2a - 1$ (if $b$ is odd), the result in its full generality is not valid for B-trees, and Huddleston and Mehlhorn present a simple counterexample for $(2, 3)$-trees.

A crucial fact used in the proof of the above amortization argument is that the sequence of operations to be analyzed is performed on an initially *empty* data structure. Jacobsen et al. [14] proved the existence of *non-extreme* $(a, b)$-trees, i. e. $(a, b)$-trees where only few nodes have a degree of $a$ or $b$. Based upon this, they re-established the above result that the rebalancing cost in a sequence of operations is amortized constant (and thus the related result for buffer trees) also for operations on initially non-empty data structures.

In connection with concurrent operations in database systems, it should be noted that the analysis of Huddleston and Mehlhorn actually requires $b \geq 2a + 2$ if a top-down splitting approach is used. In can be shown, though, that even in the general case, few node splits (in an amortized sense) happen close to the root.

## URL to Code

There is a variety of (commercial and free) implementations of B-trees and $(a, b)$-trees available for download. Representatives are the C++-based implementations that are part of the LEDA-library (http://www.algorithmic-solutions.com), the STXXL-library (http://stxxl.sourceforge.net), and the TPIE-library (http://www.cs.duke.edu/TPIE) as well as the `Java`-based implementation that is part of the `javaxxl`-library (http://www.xxl-library.de). Furthermore, (pseudo-code) implementations can be found in almost every textbook on database systems or on algorithms and data structures—see, e. g., [10,11]. Since textbooks almost always leave developing the implementation details of the DELETE operation as an exercise to the reader, the discussion by Jannink [15] is especially helpful.

## Cross References

► Cache-Oblivious B-Tree
► I/O-model
► R-Trees

## Recommended Reading

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31**, 1116–1127 (1988)
2. Arge, L.A.: External memory data structures. In: Abello, J., Pardalos, P.M., Resende, M.G.C. (eds.) Handbook of Massive Data Sets, pp. 313–357. Kluwer, Dordrecht (2002)
3. Arge, L.A.: The Buffer Tree: A technique for designing batched external data structures. Algorithmica **37**, 1–24 (2003)
4. Arge, L.A., Hinrichs, K.H., Vahrenhold, J., Vitter, J.S.: Efficient bulk operations on dynamic R-trees. Algorithmica **33**, 104–128 (2002)
5. Arge, L.A., Vitter, J.S.: Optimal external interval management. SIAM J. Comput. **32**, 1488–1508 (2003)
6. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. Acta Inform. **1**, 173–189 (1972)
7. Bayer, R., Schkolnick, M.: Concurrency of operations on B-trees. Acta Inform. **9**, 1–21 (1977)

8. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion B-tree. VLDB J. **5**, 264–275 (1996)
9. Comer, D.E.: The ubiquitous B-tree. ACM Comput. Surv. **11**, 121–137 (1979)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series, 2nd edn. MIT Press, Cambridge (2001)
11. Elmasri, R., Navanthe, S.B.: Fundamentals of Database Systems, 5th edn. Addison-Wesley, Boston (2007)
12. Graefe, G.: B-tree indexes for high update rates. SIGMOD RECORD **35**, 39–44 (2006)
13. Huddleston, S., Mehlhorn, K.: A new data structure for representing sorted lists. Acta Inform. **17**, 157–184 (1982)
14. Jacobsen, L., Larsen, K.S., Nielsen, M.N.: On the existence of non-extreme $(a,b)$-trees. Inform. Process. Lett. **84**, 69–73 (2002)
15. Jannink, J.: Implementing deletions in $B^+$-trees. SIGMOD RECORD **24**, 33–38 (1995)
16. Knuth, D.E.: Sorting and Searching. The Art of Computer Programming, vol. 3, 2nd edn. Addison-Wesley, Reading (1998)
17. Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. EATCS Monographs on Theoretical Computer Science, vol. 1. Springer, Berlin (1984)
18. Yao, A.C.-C.: On random 2–3 trees. Acta Inform. **9**, 159–170 (1978)

## Buffering

► Direct Routing Algorithms
► Packet Switching in Single Buffer

## Burrows–Wheeler Transform

### 1994; Burrows, Wheeler

PAOLO FERRAGINA[1], GIOVANNI MANZINI[2]
[1] Department of Computer Science, University of Pisa, Pisa, Italy
[2] Department of Computer Science, University of Eastern Piedmont, Alessandria, Italy

### Keywords and Synonyms

Block-sorting data compression

### Problem Definition

The Burrows–Wheeler transform is a technique used for the lossless compression of data. It is the algorithmic core of the tool bzip2 which has become a standard for the creation and distribution of compressed archives.

Before the introduction of the Burrows–Wheeler transform, the field of lossless data compression was dominated by two approaches (see [1,15] for comprehensive surveys). The first approach dates back to the pioneering works of Shannon and Huffman, and it is based on the idea of using shorter codewords for the more frequent symbols. This idea has originated the techniques of Huffman and Arithmetic Coding, and, more recently, the PPM (Prediction by Partial Matching) family of compression algorithms. The second approach originated from the works of Lempel and Ziv and is based on the idea of adaptively building a dictionary and representing the input string as a concatenation of dictionary words. The best-known compressors based on this approach form the so-called ZIP-family; they have been the standard for several years and are available on essentially any computing platform (e. g. gzip, zip, winzip, just to cite a few).

The Burrows–Wheeler transform introduced a completely new approach to lossless data compression based on the idea of *transforming* the input to make it easier to compress. In the authors' words: "(this) technique [. . .] works by applying a reversible transformation to a block of text to make redundancy in the input more accessible to simple coding schemes" [3, Sect. 7]. Not only has this technique produced some state-of-the-art compressors, but it also originated the field of compressed indexes [14] and it has been successfully extended to compress (and index) structured data such as XML files [7] and tables [16].

### Key Results

#### Notation

Let $s$ be a string of length $n$ drawn from an alphabet $\Sigma$. For $i = 0, \ldots, n-1$, $s[i]$ denotes the $i$-th character of $s$, and $s[i, n-1]$ denotes the suffix of $s$ starting at position $i$ (that is, starting with the character $s[i]$). Given two strings $s$ and $t$, the notation $s \prec t$ is used to denote that $s$ lexicographically precedes $t$.

#### The Burrows–Wheeler Transform

In [3] Burrows and Wheeler introduced a new compression algorithm based on a reversible transformation, now called the *Burrows–Wheeler Transform* (bwt). Given a string $s$, the computation of $\mathrm{bwt}(s)$ consists of three basic steps (see Fig. 1):

1. Append to the end of $s$ a special symbol \$ smaller than any other symbol in $\Sigma$;
2. Form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $s\$$ sorted in lexicographic order;
3. Construct the transformed text $\hat{s} = \mathrm{bwt}(s)$ by taking the last column of $\mathcal{M}$.

Notice that every column of $\mathcal{M}$, hence also the transformed text $\hat{s}$, is a permutation of $s\$$. As an example

```
mississippi$         $ mississipp i
ississippi$m         i $mississip p
ssissippi$mi         i ppi$missis s
sissippi$mis         i ssippi$mis s
issippi$miss         i ssissippi$ m
ssippi$missi         m ississippi $
sippi$missis    ⟹    p i$mississi p
ippi$mississ         p pi$mississ i
ppi$mississi         s ippi$missi s
pi$mississip         s issippi$mi s
i$mississipp         s sippi$miss i
$mississippi         s sissippi$m i
```

**Burrows–Wheeler Transform, Figure 1**
**Example of Burrows–Wheeler transform for the string $s$=`mississippi`. The matrix on the right has the rows sorted in lexicographic order. The output of the bwt is the last column of the sorted matrix; in this example the output is $\hat{s}$ = bwt($s$) = `ipssm$pissii`**

F, the first column of the bwt matrix $\mathcal{M}$, consists of all characters of $s$ alphabetically sorted. In Fig. 1 it is F = `$iiiimppssss`.

Although it is not obvious from its definition, the bwt is an invertible transformation and both the bwt and its inverse can be computed in $O(n)$ optimal time. To be consistent with the more recent literature, the following notation and proof techniques will be slightly different from the ones in [3].

**Definition 1** For $1 \le i \le n$, let $s[k_i, n-1]$ denote the suffix of $s$ prefixing row $i$ of $\mathcal{M}$, and define $\Psi(i)$ as the index of the row prefixed by $s[k_i + 1, n-1]$.

For example, in Fig. 1 it is $\Psi(2) = 7$ since row 2 of $\mathcal{M}$ is prefixed by `ippi` and row 7 is prefixed by `ppi`. Note that $\Psi(i)$ is not defined for $i = 0$ since row 0 is not prefixed by a proper suffix of $s$.[1]

**Lemma 1** *For* $i = 1, \ldots, n$, *it is* $\mathsf{F}[i] = \hat{s}[\Psi(i)]$.

*Proof* Since each row contains a cyclic shift of $s\$$, the last character of the row prefixed by $s[k_i + 1, n-1]$ is $s[k_i]$. Definition 1 then implies $\hat{s}[\Psi(i)] = s[k_i] = \mathsf{F}[i]$ as claimed. □

**Lemma 2** *If* $1 \le i < j \le n$ *and* $\mathsf{F}[i] = \mathsf{F}[j]$ *then* $\Psi(i) < \Psi(j)$.

*Proof* Let $s[k_i, n-1]$ (resp. $s[k_j, n-1]$) denote the suffix of $s$ prefixing row $i$ (resp. row $j$). The hypothesis $i < j$ implies that $s[k_i, n-1] \prec s[k_j, n-1]$. The hypothesis $\mathsf{F}[i] = \mathsf{F}[j]$ implies $s[k_i] = s[k_j]$ hence it must be $s[k_i + 1, n-1] \prec s[k_j + 1, n-1]$. The thesis follows since by construction $\Psi(i)$ (resp. $\Psi(j)$) is the lexicographic position of the row prefixed by $s[k_i + 1, n-1]$ (resp. $s[k_j + 1, n-1]$). □

**Lemma 3** *For any character* $c \in \Sigma$, *if* $\mathsf{F}[j]$ *is the* $\ell$*-th occurrence of* $c$ *in* F, *then* $\hat{s}[\Psi(j)]$ *is the* $\ell$*-th occurrence of* $c$ *in* $\hat{s}$.

*Proof* Take an index $h$ such that $h < j$ and $\mathsf{F}[h] = \mathsf{F}[j] = c$ (the case $h > j$ is symmetric). Lemma 2 implies $\Psi(h) < \Psi(j)$ and Lemma 1 implies $\hat{s}[\Psi(h)] = \hat{s}[\Psi(j)] = c$. Consequently, the number of $c$'s preceding (resp. following) $\mathsf{F}[j]$ in F coincides with the number of $c$'s preceding (resp. following) $\hat{s}[\Psi(j)]$ in $\hat{s}$ and the lemma follows. □

In Fig. 1 it is $\Psi(2) = 7$ and both $\mathsf{F}[2]$ and $\hat{s}[7]$ are the second `i` in their respective strings. This property is usually expressed by saying that corresponding characters maintain the *same relative order* in both strings F and $\hat{s}$.

**Lemma 4** *For any* $i$, $\Psi(i)$ *can be computed from* $\hat{s}$ = bwt($s$).

*Proof* Retrieve F simply by sorting alphabetically the symbols of $\hat{s}$. Then compute $\Psi(i)$ as follows: (1) set $c = \mathsf{F}[i]$, (2) compute $\ell$ such that $\mathsf{F}[i]$ is the $\ell$-th occurrence of $c$ in F, (3) return the index of the $\ell$-th occurrence of $c$ in $\hat{s}$. □

Referring again to Fig. 1, to compute $\Psi(10)$ it suffices to set $c = \mathsf{F}[10] =$ `s` and observe that $\mathsf{F}[10]$ is the second `s` in F. Then it suffices to locate the index $j$ of the second `s` in $\hat{s}$, namely $j = 4$. Hence $\Psi(10) = 4$, and in fact row 10 is prefixed by `sissippi` and row 4 is prefixed by `issippi`.

**Theorem 5** *The original string* $s$ *can be recovered from* bwt($s$).

*Proof* Lemma 4 implies that the column F and the map $\Psi$ can be retrieved from bwt($s$). Let $j_0$ denote the index of the special character \$ in $\hat{s}$. By construction, the row $j_0$ of the bwt matrix is prefixed by $s[0, n-1]$, hence $s[0] = \mathsf{F}[j_0]$. Let $j_1 = \Psi(j_0)$. By Definition 1 row $j_1$ is prefixed by $s[1, n-1]$ hence $s[1] = \mathsf{F}[j_1]$. Continuing in this way it is straightforward to prove by induction that $s[i] = \mathsf{F}[\Psi^i(j_0)]$, for $i = 1, \ldots, n-1$. □

## Algorithmic Issues

A remarkable property of the bwt is that both the direct and the inverse transform admit efficient algorithms that are extremely simple and elegant.

[1]In [3] instead of $\Psi$ the authors make use of a map which is essentially the inverse of $\Psi$. The use of $\Psi$ has been introduced in the literature of compressed indexes where $\Psi$ and its inverse play an important role (see [14]).

Procedure sa2bwt
```
1. bwt[0]=s[n-1];
2. for(i=1;i<=n;i++)
3. if(sa[i] == 1)
4. bwt[i]='$';
5. else
6. bwt[i]=s[sa[i]-1];
```

Procedure bwt2psi
```
71. for(i=0;i<=n;i++)
2. c = bwt[i];
3. if(c == '$')
4. j0 = i;
5. else
6. h = count[c]++;
7. psi[h]=i;
```

Procedure psi2text
```
891. k = j0; i=0;
2. do
3. k = psi[k];
4. s[i++] = bwt[k];
   while(i<n);
```

**Burrows–Wheeler Transform, Figure 2**
**Algorithms for computing and inverting the Burrows–Wheeler Transform. Procedure sa2bwt computes bwt($s$) given $s$ and its suffix array sa. Procedure bwt2psi takes bwt($s$) as input and computes the $\Psi$ map storing it in the array `psi`. bwt2psi also stores in `j0` the index of the row prefixed by $s[0, n-1]$. bwt2psi uses the auxiliary array `count`$[1, |\Sigma|]$ which initially contains in `count[i]` the number of occurrences in bwt($s$) of the symbols $1, \dots, i-1$. Finally, procedure psi2text recovers the string $s$ given bwt($s$), the array `psi`, and the value $j_0$**

**Theorem 6** *Let $s[1, n]$ be a string over a constant size alphabet $\Sigma$. String $\hat{s} = \mathsf{bwt}(s)$ can be computed in $O(n)$ time using $O(n \log n)$ bits of working space.*

*Proof* The Suffix Array of $s$ can be computed in $O(n)$ time and $O(n \log n)$ bits of working space by using, for example, the algorithm in [11]. The Suffix Array is an array of integers $\mathsf{sa}[1, n]$ such that for $i = 1, \dots, n$, $s[\mathsf{sa}[i], n-1]$ is the $i$-th suffix of $s$ in the lexicographic order. Since each row of $\mathcal{M}$ is prefixed by a unique suffix of $s$ followed by the special symbol \$, the Suffix Array provides the ordering of the rows in $\mathcal{M}$. Consequently, bwt($s$) can be computed from sa in linear time using the procedure sa2bwt of Fig. 2. □

**Theorem 7** *Let $s[1, n]$ be a string over a constant size alphabet $\Sigma$. Given* bwt($s$)*, the string $s$ can be retrieved in $O(n)$ time using $O(n \log n)$ bits of working space.*

*Proof* The algorithm for retrieving $s$ follows almost verbatim the procedure outlined in the proof of Theorem 5. The only difference is that, for efficiency reasons, all the values of the map $\Psi$ are computed in one shot. This is done by the procedure bwt2psi in Fig. 2. In bwt2psi instead of working with the column F, it uses the array count which is a "compact" representation of F. At the beginning of the procedure, for any character $c \in \Sigma$, count[$c$] provides the index of the first row of $\mathcal{M}$ prefixed by $c$. For example, in Fig. 1 count[`i`] = 1, count[`m`] = 5, and so on. In the main `for` loop of bwt2psi the array bwt is scanned and count[$c$] is increased every time an occurrence of character $c$ is encountered (line 6). Line 6 also assigns to `h` the index of the $\ell$-th occurrence of $c$ in F. By Lemma 3, line 7 stores correctly in psi[$h$] the value $i = \Psi(h)$. After the computation of array psi, $s$ is retrieved by using the procedure psi2text of Fig. 2, whose correctness immediately follows from Theorem 5.

Clearly, the procedures bwt2psi and psi2text in Fig. 2 run in $O(n)$ time. Their working space is dominated by the cost of storing the array psi which takes $O(n \log n)$ bits. □

## The Burrows–Wheeler Compression Algorithm

The rationale for using the bwt for data compression is the following. Consider a string $w$ that appears $k$ times within $s$. In the bwt matrix of $s$ there will be $k$ consecutive rows prefixed by $w$, say rows $r_w + 1, r_w + 2, \dots, r_w + k$. Hence, the positions $r_w + 1, \dots, r_w + k$ of $\hat{s} = \mathsf{bwt}(s)$ will contain precisely the symbols that immediately precede $w$ in $s$. If in $s$ certain patterns are more frequent than others, then for many substrings $w$ the corresponding positions $r_w + 1, \dots, r_w + k$ of $\hat{s}$ will contain only a few distinct symbols. For example, if $s$ is an English text and $w$ is the string `his`, the corresponding portion of $\hat{s}$ will likely contain many `t`'s and blanks and only a few other symbols. Hence $\hat{s}$ is a permutation of $s$ that is usually *locally homogeneous*, in that its "short" substrings usually contain only a few distinct symbols.[2]

To take advantage of this property, Burrows and Wheeler proposed to process the string $\hat{s}$ using move-to-front encoding [2] (mtf). mtf encodes each symbol with the number of distinct symbols encountered since its previous occurrence. To this end, mtf maintains a list of the symbols ordered by recency of occurrence; when the next symbol arrives the encoder outputs its current rank and moves it to the front of the list. Note that mtf produces

[2]Obviously this is true only if $s$ has some regularity: if $s$ is a random string $\hat{s}$ will be random as well!

a string which has the same length as $\hat{s}$ and, if $\hat{s}$ is locally homogeneous, the string mtf($\hat{s}$) will mainly consist of small integers.[3] Given this skewed distribution, mtf($\hat{s}$) can be easily compressed: Burrows and Wheeler proposed to compress it using Huffman or Arithmetic coding, possibly preceded by the run-length encoding of runs of equal integers.

Burrows and Wheeler were mainly interested in proposing an algorithm with good practical performance. Indeed their simple implementation outperformed, in terms of compression ratio, the tool gzip that was the current standard for lossless compression. A few years after the introduction of the bwt, [9,12] have shown that the compression ratio of the Burrows–Wheeler compression algorithm can be bounded in terms of the $k$-th order empirical entropy of the input string for any $k \geq 0$. For example, Kaplan et al. [9] showed that for any input string $s$ and real $\mu > 1$, the length of the compressed string is bounded by $\mu n H_k(s) + n \log(\zeta(\mu)) + \mu g_k + O(\log n)$ bits, where $\zeta(\mu)$ is the standard Zeta function and $g_k$ is a function depending only on $k$ and the size of $\Sigma$. This bound holds *pointwise* for *any* string $s$, *simultaneously* for any $k \geq 0$ and $\mu > 1$, and it is remarkable since similar bounds have not been proven for any other known compressor. The theoretical study on the performance of bwt-based compressors is currently a very active area of research. The reader is referred to the recommended readings for further information.

## Applications

After the seminal paper of Burrows and Wheeler, many researchers have proposed compression algorithms based on the bwt (see [4,5] and references therein). Of particular theoretical interest are the results in [6] showing that the bwt can be used to design a "compression booster", that is, a tool for improving the performance of other compressors in a well-defined and measurable way.

Another important area of application of the bwt is the design of Compressed Full-text Indexes [14]. These indexes take advantage of the relationship between the bwt and the Suffix Array to provide a compressed representation of a string supporting the efficient search and retrieval of the occurrences of an arbitrary pattern.

## Open Problems

In addition to the investigation on the performance of bwt-based compressors, an open problem of great practical significance is the space efficient computation of the bwt. Given a string $s$ of length $n$ over an alphabet $\Sigma$, both $s$ and $\hat{s}$ = bwt($s$) take $O(n \log |\Sigma|)$ bits. Unfortunately, the linear time algorithms shown in Fig. 2 make use of auxiliary arrays (i. e. sa and psi) whose storage takes $\Theta(n \log n)$ bits. This poses a serious limitation to the size of the largest bwt that can be computed in main memory. Space efficient algorithms for inverting the bwt have been obtained in the compressed indexing literature [14], while the problem of space- and time-efficient computation of the bwt is still open even if interesting preliminary results are reported in [8,10,13].

[3]If $s$ is an English text, mtf($\hat{s}$) usually contains more that 50% zeroes.

## Experimental Results

An experimental study of the performance of several compression algorithms based on the bwt and a comparison with other state-of-the-art compressors is presented in [4].

## Data Sets

The data sets used in [4] are available from http://www.mfn.unipmn.it/~manzini/boosting. Other data sets relevant for compression and compressed indexing are available at the Pizza&Chili site http://pizzachili.di.unipi.it/.

## URL to Code

The Compression Boosting page (http://www.mfn.unipmn.it/~manzini/boosting) contains the source code of the algorithms tested in [4]. A more "lightweight" code for the computation of the bwt and its inverse (without compression) is available at http://www.mfn.unipmn.it/~manzini/lightweight. The code of bzip2 is available at http://www.bzip.org.

## Cross References

- ► Arithmetic Coding for Data Compression
- ► Boosting Textual Compression
- ► Compressed Suffix Array
- ► Compressed Text Indexing
- ► Suffix Array Construction
- ► Table Compression
- ► Tree Compression and Indexing

## Recommended Reading

1. Bell, T.C., Cleary, J.G., Witten, I.H.: Text compression. Prentice Hall, NJ (1990)
2. Bentley, J., Sleator, D., Tarjan, R., Wei, V.: A locally adaptive compression scheme. Commun. ACM **29**, 320–330 (1986)

3. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Tech. Report 124, Digital Equipment Corporation (1994)
4. Ferragina, P., Giancarlo, R., Manzini, G.: The engineering of a compression boosting library: Theory vs practice in bwt compression. In: Proc. 14th European Symposium on Algorithms (ESA). LNCS, vol. 4168, pp. 756–767. Springer, Berlin (2006)
5. Ferragina, P., Giancarlo, R., Manzini, G.: The myriad virtues of wavelet trees. In: Proc. 33th International Colloquium on Automata and Languages (ICALP), pp. 561–572. LNCS n. 4051. Springer, Berlin, Heidelberg (2006)
6. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. J. ACM **52**, 688–713 (2005)
7. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS), 184–193, Pittsburgh, PA (2005)
8. Hon, W., Sadakane, K., Sung, W.: Breaking a time-and-space barrier in constructing full-text indices. In: Proc. of the 44th IEEE Symposium on Foundations of Computer Science (FOCS), 251–260, Cambridge, MA (2003)
9. Kaplan, H., Landau, S., Verbin, E.: A simpler analysis of Burrows-Wheeler-based compression. Theoretical Computer Science **387**(3): 220–235 (2007)
10. Kärkkäinen, J.: Fast BWT in small space by blockwise suffix sorting. Theoretical Computer Science **387**(3): 249–257 (2007)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53**(6), 918–936 (2006)
12. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**, 407–430 (2001)
13. Na, J.: Linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space for large alphabets. In: Proc. 16th Symposium on Combinatorial Pattern Matching (CPM). LNCS, vol. 3537, pp. 57–67. Springer, Berlin (2005)
14. Navarro, G., Mäkinen, V.: Compressed full text indexes. ACM Comput. Surv. **39**(1) (2007)
15. Salomon, D.: Data Compression: the Complete Reference, 3rd edn. Springer, New York (2004)
16. Vo, B.D., Vo, K.P.: Using column dependency to compress tables. In: Proc. of IEEE Data Compression Conference (DCC), pp. 92–101, IEEE Computer Society Press (2004)

# Byzantine Agreement

## 1980; Pease, Shostak, Lamport

Michael Okun
Weizmann Institute of Science, Rehovot, Israel

## Keywords and Synonyms

Consensus; Byzantine generals; Interactive consistency

## Problem Definition

The study of Pease, Shostak and Lamport was among the first to consider the problem of achieving a coordinated behavior between processors of a distributed system in the presence of failures [21]. Since the paper was published, this subject has grown into an extensive research area. Below is a presentation of the main findings regarding the specific questions addressed in their paper. In some cases this entry uses the currently accepted terminology in this subject, rather than the original terminology used by the authors.

### System Model

A distributed system is considered to have $n$ independent processors, $p_1, \dots, p_n$, each modeled as a (possibly infinite) state machine. The processors are linked by a communication network that supports direct communication between every pair of processors. The processors can communicate only by exchanging messages, where the sender of every message can be identified by the receiver. While the processors may fail, it is assumed that the communication subsystem is fail-safe. It is not known in advance which processors will not fail (remain correct) and which ones will fail. The types of processor failures are classified according to the following hierarchy.

**Crash failure** A crash failure means that the processor no longer operates (ad infinitum, starting from the failure point). In particular, other processors will not receive messages from a faulty processor after it crashes.

**Omission failure** A processor fails to send and receive an arbitrary subset of its messages.

**Byzantine failure** A faulty processor behaves arbitrarily.

The Byzantine failure is further subdivided into two cases, according to the ability of the processors to create unfalsifiable signatures for their messages. In the *authenticated Byzantine failure* model it is assumed that each message is *signed* by its sender and that no other processor can fake a signature of a correct processor. Thus, even if such a message is forwarded by other processors, its authenticity can be verified. If the processors represent malevolent (human) users of a distributed system, a Public Key Infrastructure (PKI) is typically used to sign the messages (which involves cryptography related issues [17], not discussed here). Practically, in systems where processors are just "processors", a simple signature, such as CRC (cyclic redundancy check), might be sufficient [13]. In the *unauthenticated Byzantine failure* model there are no message signatures.

### Definition of the Byzantine Agreement Problem

In the beginning, each processor $p_i$ has an externally provided input value $v_i$, from some set $V$ (of at least size 2). In the *Byzantine Agreement* (BA) problem, every correct processor $p_i$ is required to decide on an output value $d_i \in V$ such that the following conditions hold:

**Termination** Eventually, $p_i$ decides, i. e., the algorithm cannot run indefinitely.
**Validity** If the input value of all the processors is $v$, then the correct processors decide $v$.
**Agreement** All the correct processors decide on the same value.

For crash failures and omission failures there exists a stronger agreement condition:

**Uniform Agreement** No two processors (either correct or faulty) decide differently.

The termination condition has the following stronger version.

**Simultaneous Termination** All the correct processors decide in the *same round* (see definition below).

### Timing Model

The BA problem was originally defined for *synchronous* distributed systems [18,21]. In this timing model the processors are assumed to operate in lockstep, which allows to partition the execution of a protocol to rounds. Each round consists of a send phase, during which a processor can send a (different) message to each processor directly connected to it, followed by a receive phase, in which it receives messages sent by these processors in the current round. Unlimited local computations (state transitions) are allowed in both phases, which models the typical situation in real distributed systems, where computation steps are faster than the communication steps by several orders of magnitude.

### Overview

This entry deals only with *deterministic* algorithms for the BA problem in the synchronous model. For algorithms involving randomization see the ► Probabilistic Synchronous Byzantine Agreement entry in this volume. For results on BA in other models of synchrony, see ► Asynchronous Consensus Impossibility, ► Failure Detectors, ► Consensus with Partial Synchrony entries in this volume.

## Key Results

The maximum possible number of faulty processors is assumed to be bounded by an a priori specified number $t$ (e. g., estimated from the failure probability of individual processor and the requirements on the failure probability of the system as a whole). The number of processors that actually become faulty in a given execution is denoted by $f$, where $f \leq$ t.

The complexity of synchronous distributed algorithms is measured by three complementary parameters. The first is the *round complexity*, which measures the number of rounds required by the algorithm. The second is the *message complexity*, i. e., the total number of messages (and sometimes also their size in bits) sent by all the processors (in case of Byzantine failures, only messages sent by correct processors are counted). The third complexity parameter measures the number of local operations, as in sequential algorithms.

*All* the algorithms presented bellow are efficient, i. e., the number of rounds, the number of messages and their size, and the local operations performed by each processor are polynomial in $n$. In most of the algorithms, both the exchanged messages and the local computations involve only the basic data structures (e. g., arrays, lists, queues). Thus, the discussion is restricted only to the round and the message complexities of the algorithms.

The network is assumed to be fully connected, unless explicitly stated otherwise.

### Crash Failures

A simple BA algorithm which runs in $t + 1$ rounds and sends $O(n^2)$ messages, together with a proof that this number of rounds is optimal, can be found in textbooks on distributed computing [19]. Algorithms for deciding in $f + 1$ rounds, which is the best possible, are presented in [7,23] (one additional round is necessary before the processors can stop [11]). Simultaneous termination requires $t + 1$ rounds, even if no failures actually occur [11], however there exists an algorithm that in any given execution stops in the earliest possible round [14]. For uniform agreement, decision can be made in $\min(f + 2, t + 1)$ rounds, which is tight [7].

In case of crash failures it is possible to solve the BA problem with $O(n)$ messages, which is also the lower bound. However, all known message-optimal BA algorithms require a superlinear time. An algorithm that runs in $O(f + 1)$ rounds and uses only $O(n \text{ polylog } n)$ messages, is presented in [8], along with an overview of other results on BA message complexity.

### Omission Failures

The basic algorithm used to solve the crash failure BA problem works for omission failures as well, which allows to solve the problem in $t + 1$ rounds [23]. An algorithm which terminates in $\min(f + 2, t + 1)$ rounds was presented in [22]. Uniform agreement is impossible for $t \geq n/2$ [23]. For $t < n/2$, there is an algorithm that achieves uniform agreement in $\min(f + 2, t + 1)$ rounds (and $O(n^2 f)$ message complexity) [20].

### Byzantine Failures with Authentication

A $(t + 1)$-round BA algorithm is presented in [12]. An algorithm which terminates in $\min(f + 2, t + 1)$ rounds can be found in [24]. The message complexity of the problem is analyzed in [10], where it is shown that the number of signatures and the number of messages in any authenticated BA algorithm are $\Omega(nt)$ and $\Omega(n + t^2)$, respectively. In addition, it is shown that $\Omega(nt)$ is the bound on the number of messages for the unauthenticated BA.

### Byzantine Failures Without Authentication

In the unauthenticated case, the BA problem can be solved if and only if $n > 3t$. The proof can be found in [1,19]. An algorithm that decides in $\min(f + 3, t + 1)$ rounds (it might require two additional rounds to stop) is presented in [16]. Unfortunately, this algorithm is complicated. Simpler algorithms, that run in $\min(2f + 4, 2t + 1)$ and $3\min(f + 2, t + 1)$ rounds, are presented in [24] and [5], respectively. In these algorithms the number of sent messages is $O(n^3)$, moreover, in the latter algorithm the messages are of constant size (2 bits). Both algorithms assume $V = \{0, 1\}$. To solve the BA problem for a larger $V$, several instances of a binary algorithm can be run in parallel. Alternatively, there exists a simple 2-round protocol that reduces a BA problem with arbitrary initial values to the binary case, e. g., see Sect. 6.3.3 in [19]. For algorithms with optimal $O(nt)$ message complexity and $t + o(t)$ round complexity see [4,9].

### Arbitrary Network Topologies

When the network is not fully connected, BA can be solved for crash, omission and authenticated Byzantine failures if and only if it is $(t + 1)$-connected [12]. In case of Byzantine failures without authentication, BA has a solution if and only if the network is $(2t + 1)$-connected and $n > 3t$ [19]. In both cases the BA problem can be solved by simulating the algorithms for the fully connected network, using the fact that the number of disjoint communication paths between any pair of non-adjacent processors exceeds the number of faulty nodes by an amount that is sufficient for reliable communication.

### Interactive Consistency and Byzantine Generals

The BA (consensus) problem can be stated in several similar ways. Two widely used variants are the *Byzantine Generals* (BG) problem and the *Interactive Consistency* (IC) problem. In the BG case there is a designated processor, say $p_1$, which is the only one to have an input value. The termination and agreement requirements of the BG problem are exactly as in BA, while the validity condition requires that if the input value of $p_1$ is $v$ and $p_1$ is correct, then the correct processors decide $v$. The IC problem is an extension of BG, where every processor is "designated", so that each processor has to decide on a vector of $n$ values, where the conditions for the $i$-th entry are as in BG, with $p_i$ as the designated processor. For deterministic synchronous algorithms BA, BG and IC problems are essentially equivalent, e. g., see the discussion in [15].

### Firing Squad

The above algorithms assume that the processors share a "global time", i. e., all the processors start in the same (first) round, so that their round counters are equal throughout the execution of the algorithm. However, there are cases in which the processors run in a synchronous network, yet each processor has its own notion of time (e. g., when each processor starts on its own, the round counter values are distinct among the processors). In these cases, it is desirable to have a protocol that allows the processors to agree on some specific round, thus creating a common round which synchronizes all the correct processors. This synchronization task, known as the *Byzantine firing squad* problem [6], is tightly realted to BA.

### General Translation Techniques

One particular direction that was pursued as part of the research on the BA problem is the development of methods that automatically translate any protocol that tolerates a more benign failure type into one which tolerates more severe failures [24]. Efficient translations spanning the entire failure hierarchy, starting from crash failures all the way to unauthenticated Byzantine failures, can be found in [3] and in Ch. 12 of [1].

## Applications

Due to the very tight synchronization assumptions made in the algorithms presented above, they are used mainly in real-time, safety-critical systems, e. g., aircraft control [13]. In fact, the original interest of Pease, Shostak and Lamport in this problem was raised by such an application [21]. In addition, BA protocols for the Byzantine failure case serve as a basic building block in many cryptographic protocols, e. g., secure multi-party computation [17], by providing a broadcast channel on top of pairwise communication channels.

## Cross References

▶ Asynchronous Consensus Impossibility
▶ Atomic Broadcast
▶ Consensus with Partial Synchrony
▶ Failure Detectors
▶ Optimal Probabilistic Synchronous Byzantine Agreement
▶ Randomization in Distributed Computing
▶ Renaming
▶ Set Agreement

## Recommended Reading

1. Attiya, H., Welch, J.L.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. McGraw-Hill, UK (1998)
2. Barborak, M., Dahbura, A., Malek, M.: The Consensus Problem in Fault-Tolerant Computing. ACM Comput. Surv. **25**(2), 171–220 (1993)
3. Bazzi, R.A., Neiger, G.: Simplifying Fault-tolerance: Providing the Abstraction of Crash Failures. J. ACM **48**(3), 499–554 (2001)
4. Berman, P., Garay, J.A., Perry, K.J.: Bit Optimal Distributed Consensus. In: Yaeza-Bates, R., Manber, U. (eds.) Computer Science Research, pp. 313–322. Plenum Publishing Corporation, New York (1992)
5. Berman, P., Garay, J.A., Perry, K.J.: Optimal Early Stopping in Distributed Consensus. In: Proc. 6th International Workshop on Distributed Algorithms (WDAG), pp. 221–237, Israel, November 1992
6. Burns, J.E., Lynch, N.A.: The Byzantine Firing Squad problem. Adv. Comput. Res. **4**, 147–161 (1987)
7. Charron-Bost, B., Schiper, A.: Uniform Consensus is Harder than Consensus. J. Algorithms **51**(1), 15–37 (2004)
8. Chlebus, B.S., Kowalski, D.R.: Time and Communication Efficient Consensus for Crash Failures. In: Proc. 20th International Symposium on Distributed Computing (DISC), pp. 314–328, Sweden, September 2006
9. Coan, B.A., Welch, J.L.: Modular construction of a Byzantine agreement protocol with optimal message bit complexity. Inf. Comput. **97**(1), 61–85 (1992)
10. Dolev, D., Reischuk, R.: Bounds on Information Exchange for Byzantine Agreement. J. ACM **32**(1), 191–204 (1985)
11. Dolev, D., Reischuk, R., Strong, H.R.: Early Stopping in Byzantine Agreement. J. ACM **37**(4), 720–741 (1990)
12. Dolev, D., Strong, H.R.: Authenticated Algorithms for Byzantine Agreement. SIAM J. Comput. **12**(4), 656–666 (1983)
13. Driscoll, K., Hall, B., Sivencrona, H., Zumsteg, P.: Byzantine Fault Tolerance, from Theory to Reality. In: Proc. 22nd International Conference on Computer Safety, Reliability, and Security (SAFECOMP), pp. 235–248, UK, September 2003
14. Dwork, C., Moses, Y.: Knowledge and Common Knowledge in a Byzantine Environment: Crash Failures. Inf. Comput. **88**(2), 156–186 (1990)
15. Fischer, M.J.: The Consensus Problem in Unreliable Distributed Systems (A Brief Survey). Research Report, YALEU/DCS/RR-273, Yale University, New Heaven (1983)
16. Garay, J.A., Moses, Y.: Fully Polynomial Byzantine Agreement for $n > 3t$ Processors in $t + 1$ Rounds. SIAM J. Comput. **27**(1), 247–290 (1998)
17. Goldreich, O.: Foundations of Cryptography, vol. 1-2. Cambridge University Press, UK (2001) (2004)
18. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine Generals Problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
19. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, CA (1996)
20. Parvédy, P.R., Raynal, M.: Optimal Early Stopping Uniform Consensus in Synchronous Systems with Process Omission Failures. In: Proc. 16th Annual ACM Symposium on Parallel Algorithms (SPAA), pp. 302–310, Spain, June 2004
21. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching Agreement in the Presence of Faults. J. ACM **27**(2), 228–234 (1980)
22. Perry, K.J., Toueg, S.: Distributed Agreement in the Presence of Processor and Communication Faults. IEEE Trans. Softw. Eng. **12**(3), 477–482 (1986)
23. Raynal, M.: Consensus in Synchronous Systems: A Concise Guided Tour. In: Proc. 9th Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 221–228, Japan, December 2002
24. Toueg, S., Perry, K.J., Srikanth, T.K.: Fast Distributed Agreement. SIAM J. Comput. **16**(3), 445–457 (1987)