Generic Distributed Exact Cover Solver

Jan Magne Tjensvold

December 6, 2007

Contents

Chapter 1

Introduction

This report details the design and implementation of a distributed computing system to solve exact cover problems. Donald Knuth's Dancing Links (DLX) algorithm [7] is used to solve the exact cover problem. Distributed computing with this algorithm is accomplished by making use of the recursive nature of DLX to split the problem into smaller pieces. The Generic Distributed Exact Cover Solver (DECS) then takes advantage of a distributed computing middleware called BOINC [1] to handle the work distribution and result collection process. The report explains in detail how the DLX algorithm works and some concrete types of problems it can be applied to. It also explains how DLX is used together with BOINC to construct a complete distributed computing system.

1.1 Related work

There are numerous implementations of the Dancing Links algorithm available with and without source code. In addition to Knuth's own implementation in CWEB [8] a quick search found source code for Java, Python, C, C++, Ruby, Lisp, MATLAB and Mathematica. Some of the implementations were generic while others aimed for a specific application (mostly Sudoku). Common for all these implementations is that none of them were made for parallel processing.

Alfred Wassermann developed a parallel version of Knuth's algorithm in [10] by using PVM [4] to solve a problem presented in Knuth's original paper. However, he only published the solutions to the problem and not the actual implementation. The only available open source parallel version is written by Owen O'Malley for the Apache Hadoop project [2] in May 2007. O'Malley's implementation uses the MapReduce framework [3] provided by Hadoop to do the computations in parallel. The details of this implementation and how it divides the problem into smaller pieces has not been documented.

1.2 Report organization

This report is organized into several chapters. Chapter 2 describes the Dancing Links algorithm. Chapter **??** explains the distributed computing concept. Chapter **??** discusses different aspects of the implementation. Chapter **??** describes the test results with the system and Chapter **??** concludes this report.

Chapter 2

Dancing Links

Dancing Links (DLX) is an algorithm invented by Donald Knuth to solve any exact cover problem. It was first described in [7] where he looks at the details of the algorithm and uses it to solve some practical problems. Before we look at the DLX algorithm in more detail we need to explain what an exact cover problem is.

2.1 Exact cover

Exact cover is a general type of problem which hcan be applied to a wide area of problems. It can be used to solve problems like N queens, polyomino tiling, Latin square puzzles, Sudoku, set packing and set partitioning. For more detailed information about how exact cover can be applied to N queens, polyomino tiling, Latin square and Sudoku, see Section **??**, **??**, **??** and **??** respectively.

To represent an exact cover problem we use a matrix in which each element is either zero or one (non-zero). This type of matrix is called a boolean, binary, logical or {0,1}-matrix. We use "matrix" in the rest of this report to mean a boolean matrix and "non-zero" to mean one since a boolean matrix can only have elements zero and one. The idea is that each column in the matrix represents a specific constraint and each row is a way to fill some of the constraints.

For the general matrix

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n} \end{bmatrix}$$

$a_{i,j}$ is an element in the matrix at row i, column j where $a_{i,j} \in \{0, 1\}$. The number of rows is m and the number of columns is n.

A subset of rows from a matrix is an exact cover iff (if and only if) each column has exactly one non-zero (one) element. Let R_A and R_B be the set of rows in matrix A and B respectively. If $R_B \subseteq R_A$ then B forms the following

matrix

$$B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & b_{2,3} & \cdots & b_{2,n} \\ b_{3,1} & b_{3,2} & b_{3,3} & \cdots & b_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & b_{k,3} & \cdots & b_{k,n} \end{bmatrix}$$

B is a reduced matrix of A where the number of rows $k \leq m$. The number of columns in A and B is always the same. The subset R_B is an exact cover iff the following equation is satisfied

$$\sum_{i=1}^{k} b_{i,j} = 1 \quad \forall j \in \{1, 2, \ldots, n-1, n\}$$

In practical applications we are usually given an initial matrix A and tasked with finding all the subsets of rows which are exact covers. For example the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.1}$$

represents a specific exact cover problem. In this matrix $\{2, 3\}$ is a valid solution (exact cover) because the subset consisting of rows 2 and 3, and thus the reduced matrix

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

has exactly one non-zero element in each column. By adopting a trial and error approach one can find that the full set of solutions for the matrix in (2.1) is $\{\{1, 4, 5\}, \{2, 3\}, \{3, 5, 6\}\}$.

2.1.1 Generalized exact cover

A generalized form of the exact cover problem is sometimes better suited to solve certain types of problems. The generalized cover problem can be translated to an exact cover problem by adding additional rows, but translating in the opposite direction is not always possible. The generalized cover problem divides the matrix into primary and secondary columns. Each primary column in the solution must have exactly one non-zero element, like before. However, each secondary column in the solution can have either zero or one non-zero element, instead of exactly one.

Let C_P be the set of primary columns and C_S the set of secondary columns in matrix A and B. The subset of rows R_B is an exact cover iff both of the following equations are satisfied

$$\sum_{i=1}^{k} b_{i,j} = 1 \quad \forall j \in C_P \quad \wedge \quad \sum_{i=1}^{k} b_{i,j} \leq 1 \quad \forall j \in C_S$$

N queens (see Section **??**) is one type of problem the generalized cover problem can be applied to. Creating a secondary column for each diagonal on the chess board will reduce the number of rows in the final matrix. Given a smaller matrix the DLX algorithm will have to do less processing to find the solutions which results in better performance. The DLX algorithm itself does not require any modifications to solve generalized cover problems, but the matrix construction procedure requires some minor adjustments (see Section **??**).

2.2 Algorithm X

Exact cover is a type of problem known to be NP-complete. To find all the solutions to an exact cover problem the most straight forward algorithm is to check all possible sets of rows. Given a set we then check to see if there is exactly one non-zero element in each column. However, as the size of the matrix increases we will experience a combinatorial explosion on the number of possible sets to test. The exact number of sets is $2^m - 1$ given a matrix with m rows. The eight queens problem (see Section **??**) which has a matrix consisting of 63 rows, gives an immense 9 223 372 036 854 775 808 sets. Given that only 92 of these are valid solutions this simple algorithm can hardly be recommended.

Another approach, which is presented in Knuth's paper, is Algorithm X (for the lack of a better name). This backtrack algorithm uses a more intelligent elimination method to "wriggle" its way through the matrix and find all the solutions. Looking at matrix (2.1) we can easily determine that row 1 and 3 can never be in the same set. They are in conflict with each other because both of them have a non-zero element in the first column. Since there must be exactly one non-zero element in each column we can rule out any set containing both row 1 and 3.

Algorithm X uses similar logic to recursively traverse the search tree by backtracking. A modified version of Knuth's original algorithm is presented in Algorithm 1. Changes are made to improve the readability, logical consistency and to make it easier to compare with the Dancing Links algorithm. Algorithm X is initially called with the matrix A and the column header list H. H is initialized with the numbers $1, 2, \ldots, n-1, n$, where n is the number of columns in A.

If H is empty the partial solution is an exact cover and the algorithm returns. Otherwise, the algorithm chooses a column c and loops through each row r which has a non-zero element in column c. Any conflict between row r and the remaining rows are resolved at line 8 to 12. The algorithm then calls itself recursively with the reduced matrix and column list. This continues until all the rows with non-zero elements in column c have been tested, in which case all the branches in the search tree have been traversed.

Any rule for choosing column c will produce all the solutions, but there are some rules that work better than others. Knuth uses what he refers to as the S heuristic, which is to always choose the column with the least amount of non-zero elements. This approach has proved to work well in a large number of cases so it is a reasonable rule to make use of in practice.

Using matrix (2.1) as an example, we wish to demonstrate how Algorithm X works. The columns and rows have been numbered to make it easier to keep

Algorithm 1 Algorithm X recursive search procedure.

```
1:  procedure search(A, H)
2:      if H is empty then
3:          Print solution and return.                {Base case for the recursion}
4:      Choose a column c.
5:      foreach row r such that a_{r,c} = 1 do
6:          Add r to partial solution.
7:          Save state of matrix A and list H.
8:          foreach column j such that a_{r,j} = 1 do
9:              foreach row i such that a_{i,j} = 1, except i = r do
10:                 Delete row i from matrix A.
11:             Delete column j from matrix A and list H.
12:         Delete row r from matrix A.
13:         search(A, H)
14:         Restore state of matrix A and list H.
15:         Remove r from the partial solution.
```

track of them when the matrix is modified.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

We begin by choosing column 1. Looking at this column we choose row 1 where there is a non-zero element. Our partial solution is now $\{1\}$. Row 1 only has one conflicting row which is row 3, which has a conflict in column 1. We remove column 1, row 1 and row 3 which results in the following matrix

$$\begin{array}{c} \\ 2 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array} \tag{2.2}$$

This time we choose column 2 and then row 2 so that the partial solution becomes $\{1, 2\}$. Row 2 conflicts with all the remaining rows (row 5 in column 2 and row 4 and 6 in column 3). After all the conflicts have been resolved the matrix itself is empty, but the column list H is not. Because there are no non-zero elements left in the matrix the recursive call will return immediately (the loop condition at line 5 is not satisfied) and matrix (2.2) is restored. This time we choose row 5 which results in the partial solution $\{1, 5\}$. Row 5 conflicts with row 2 in column 2 and by eliminating the conflicts we get the following matrix

$$\begin{array}{c} \\ 4 \\ 6 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \end{array}$$

We choose column 3 and then row 4 which gives us the partial solution $\{1, 5, 4\}$. After all the conflicts are resolved the matrix is completely empty along with the column header list. This tells us that $\{1, 5, 4\}$ is one of the solutions to this problem.

The search tree in Figure 2.1 emerges as we continue in the same manner, until all the solutions have been found. Each node i, j, for row i and column j, indicates the choices made by the algorithm. The rectangular nodes is where each of the solutions were found. The search tree is a binary tree as a result of the small matrix used in this example so this behavior cannot be generalized.

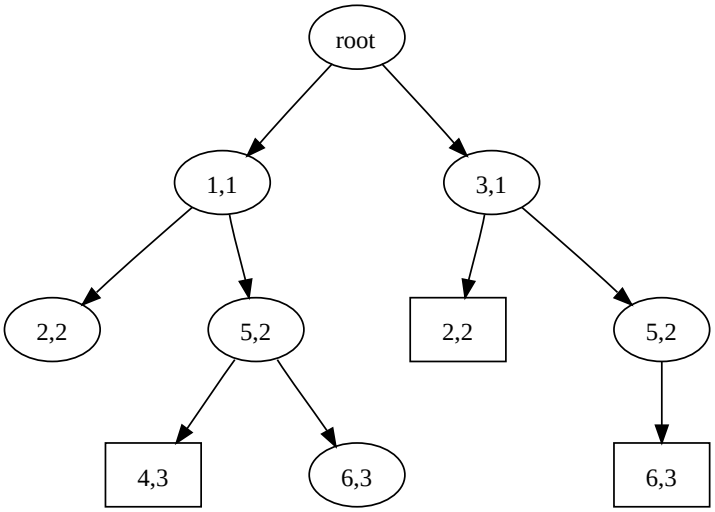

Figure 2.1: Algorithm X search tree for the example matrix

One issue when trying to implement Algorithm X is that the state of the matrix needs to be saved and restored multiple times during the backtrack process. Each time the algorithm returns the old state must be restored before another path can be explored. Searching through the matrix to find the non-zero elements is also very time consuming if the matrix is stored as a two dimensional array. To solve these problems Knuth introduced the Dancing Links algorithm.

2.3 Dancing Links

The Dancing Links (DLX) algorithm is based on Algorithm X, but it contains some significant modifications which makes it more suitable for practical applications. DLX is based on a simple, yet powerful, technique which allows one to reverse any operation made to a doubly-linked list. If x represents an element in such a list then $x.left$ and $x.right$ points to the previous and next element respectively. To remove element x from the list the following two operations are applied:

$$\begin{aligned} x.right.left &\leftarrow x.left \\ x.left.right &\leftarrow x.right \end{aligned} \tag{2.3}$$

Applying these two operations to the linked list in Figure 2.2 results in the list in Figure 2.3. These operations modify the links pointing to element x so

that an iteration through the list will no longer traverse through this element, but instead skip right past it.

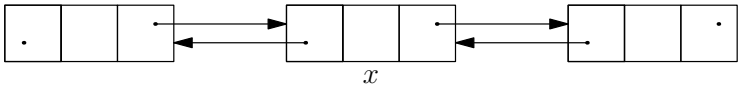

Figure 2.2: Doubly-linked list

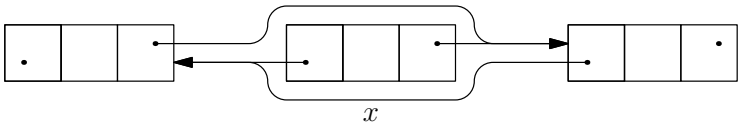

Figure 2.3: Doubly-linked list with element x removed

When programming one might be tempted to set $x.left$ and $x.right$ to a null value and delete the x object or let the garbage collector do its thing. However, smart as that might seem it would prevent one from applying a second set of operations. In [5] Hitotumatu and Noshita introduced a pair of operations which allows one to insert an element back into the list in exactly the same place it was removed from. The following two operations work as the inverse of the operations in (2.3) by adding x back into the list.

$$\begin{array}{rcl} x.right.left & \leftarrow & x \\ x.left.right & \leftarrow & x \end{array} \tag{2.4}$$

Knuth makes use of the operations in (2.3) and (2.4) to maintain the state information for the matrix. The x element is preserved so that the algorithm can reverse the remove operations, which are used to reduce the matrix.

2.3.1 Data structure

DLX stores the matrix as a collection of several circular doubly-linked lists where each non-zero value in the matrix is an element in the lists. Using this sparse matrix representation saves a lot of memory because the number of zero elements usually outnumber the non-zero elements. This advantage will normally grow when the size of the matrix increases. As an example the N queens problem for $N = 10$ has 396 non-zero elements, but they only account for 7.33% of the total number of elements.

Each row and column in the matrix is represented by a separate list. In addition the set of column headers is also stored in a list. Each element x in the linked lists have six attributes: $x.left$, $x.right$, $x.up$, $x.down$, $x.column$ and $x.row$. The $x.row$ attribute is an addition to Knuth's original algorithm to enable detection of the row number. The first four attributes contains a pointer to an element in the respective list. $x.left$ and $x.right$ belongs to a row list and $x.up$ and $x.down$ belongs to a column list. $x.column$ is a pointer to the column header and $x.row$ is a non-negative integer storing the row number of the element. A column header c has the additional $c.name$ (column name/number) and $c.size$ (number of elements in column) attributes. Secondary column headers used by the generalized cover problem have their $c.left$ and $c.right$ attributes

pointing to c (itself). The special column header element h acts as a root element for the rest of the data structure.

Figure 2.4 shows shows how the matrix in (2.1) can be represented using this data structure. To avoid clutter the $x.column$ links pointing to the column headers are not displayed in the figure.

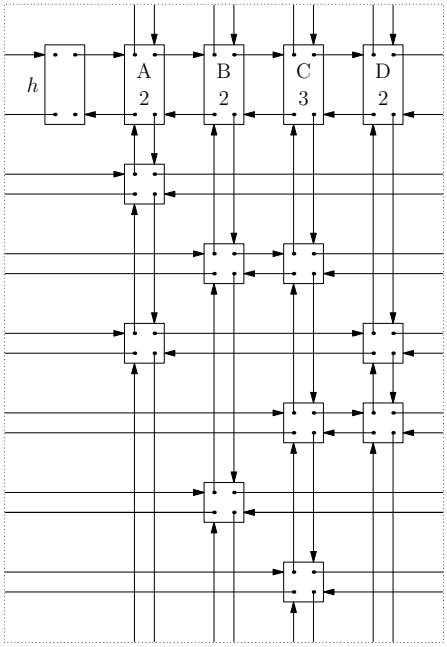

Figure 2.4: Sparse boolean matrix circular quad-linked list representation of the example matrix

2.3.2 Algorithm

The DLX algorithm is very similar in nature to Algorithm X and in essence the two algorithms work exactly in the same way. Given the example matrix (2.1) the DLX algorithm will follow the same path as shown in Figure 2.1. The difference is that DLX uses a specialized data structure and the operations in (2.3) and (2.4) to save and restore the state of the matrix. Comparing the pseudo code in Algorithm 1 and 2 reveals that they are very similar in nature.

The algorithm is initially called with $k = 0$ (recursion level 0) and h is initialized with the contents of the matrix. See Section **??** for details on how the contents of h is initialized. Printing a solution using the $x.row$ attribute is done by printing $O_i.row$ for all $i \in \{1, 2, \ldots, k\}$.

Column selection is done using the S heuristic, which picks the column with the lowest value for $c.size$, as described in Knuth's paper. Algorithm 3 simply steps through each column header looking for the lowest size.

The cover(c) and uncover(c) algorithms are the main differences between DLX and Algorithm X. The purpose of cover(c) is to remove column c from the column header list and to resolve any conflicts in the column. It uses the operations in (2.3) to remove the conflicting elements from the column lists.

The cover(c) algorithm also increments the value of *updates* which is used to measure how many operations the search algorithm requires to complete. One update equals four link modifications or one application of both operation (2.3) and (2.4). The *size* column header attribute is maintained by both the cover(c) and uncover(c) algorithms so that the S heuristic works properly. Algorithm 4 contains the pseudo code for the cover procedure.

The uncover(c) algorithm restores the state of the matrix using the operations in (2.4). Notice that Algorithm 5 walks up and left in the lists while Algorithm 4 walks down and right. This ensures that the elements are put back in the reverse order in which they were removed. This is the only way to make sure that all the links are restored to their original state.

2.4 Parallel Dancing Links

To be able to solve more complex exact cover problems the solution process must be distributed to a larger number of computers. To accomplish this we must first break the problem into smaller pieces. In [9] Maus and Aas investigates recursive procedures as the unit for parallelizing. They introduce some techniques on how to split the recursion tree which can be directly applied to the search tree in DLX. The main idea is that the algorithm is first run in a breath-first mode so that the search tree is explored one level at a time. When a certain number of nodes has been discovered the search stops and the nodes are distributed to a set computers and solved in parallel.

The search procedure of DLX is a backtrack algorithm which explores the search tree depth-first. Making this a breath-first algorithm can be achieved by not allowing it to proceed deeper than a certain level in the search tree. When the given level is reached the partial solution O is printed. O can then be used to initialize a separate process by running DLX on a different computer (or another processor on the same computer). As long as the predefined depth d is not too deep or shallow the partial solutions can be used to efficiently solve the exact cover problem in a distributed manner. If d is too deep (high) the algorithm will find all solutions before the splitting happens, and if it is too shallow (low) the number of partial solutions might be too low to be of any use. Adding line 10 to 12 in Algorithm 6 is the only changes required to make the original Algorithm 2 support this scheme.

Each partial solution produced by psearch(k, d) can be used as an initialization vector for the modified search procedure search_init(O) in Algorithm 7. O is the initialization vector and $O.size$ is the length of the vector (number of rows in the partial solution). The initialization vectors and the matrix can be distributed and the modified search procedure can be run in parallel on each computer. If required each computer can do further splitting locally to take advantage of multiple processors.

This approach does not guarantee that each initialization vector provides the same amount of work. Unfortunately there is no straight forward method to estimate the complexity of the subtree given by a specific initialization vector. In [6] Knuth uses a Monte Carlo approach to estimate the running time of a backtrack algorithm. By doing random walks in the subtree he is able to estimate the cost of backtracking. This approach would be worth investigating for a future version of DECS.

Algorithm 2 Dancing Links recursive search.

```
1:  procedure search(k)
2:    if h.right = h then
3:      Print solution and return.                    {Base case for the recursion}
4:    c ← choose_column()
5:    cover(c)
6:    foreach r ← c.down, c.down.down, ..., while r ≠ c do
7:      O_k ← r                                       {Add r to partial solution}
8:      foreach j ← r.right, r.right.right, ..., while j ≠ r do
9:        cover(j.column)
10:     search(k + 1)
11:     foreach j ← r.left, r.left.left, ..., while j ≠ r do
12:       uncover(j.column)
13:   uncover(c)
```

Algorithm 3 Column selection using the S heuristic.

```
1:  function choose_column()
2:    s ← ∞
3:    foreach j ← h.right, h.right.right, ..., while j ≠ h do
4:      if j.size < s then
5:        c ← j
6:        s ← j.size
7:    return column c
```

Algorithm 4 Cover column c.

```
1:  procedure cover(c)
2:    c.right.left ← c.left
3:    c.left.right ← c.right
4:    foreach i ← c.down, c.down.down, ..., while i ≠ c do
5:      foreach j ← i.right, i.right.right, ..., while j ≠ i do
6:        j.down.up ← j.up
7:        j.up.down ← j.down
8:        j.column.size ← j.column.size − 1
9:        updates ← updates + 1
```

Algorithm 5 Uncover column c.

```
1:  procedure uncover(c)
2:    foreach i ← c.up, c.up.up ..., while i ≠ c do
3:      foreach j ← i.left, i.left.left, ..., while j ≠ i do
4:        j.column.size ← j.column.size + 1
5:        j.down.up ← j
6:        j.up.down ← j
7:    c.right.left ← c
8:    c.left.right ← c
```

Algorithm 6 Dancing Links parallel recursive splitter.

```
1:  procedure psearch(k, d)
2:    if h.right = h then
3:      Print solution and return.                    {Base case for the recursion}
4:    c ← choose_column()
5:    cover(c)
6:    foreach r ← c.down, c.down.down, ..., while r ≠ c do
7:      O_k ← r                                       {Add r to partial solution}
8:      foreach j ← r.right, r.right.right, ..., while j ≠ r do
9:        cover(j.column)
10:     if k ≥ d and h.right ≠ h then
11:       Print partial solution.                     {Prevent further recursion}
12:     else
13:       psearch(k + 1)
14:     foreach j ← r.left, r.left.left, ..., while j ≠ r do
15:       uncover(j.column)
16:   uncover(c)
```

Algorithm 7 Dancing Links search initialization.

```
1:  procedure search_init(O)
2:    for k ← 0 to O.size − 1 do
3:      c ← choose_column()
4:      cover(c)
5:      r ← O_k
6:      foreach j ← r.right, r.right.right, ..., while j ≠ r do
7:        cover(j.column)
8:    search(O.size)                                  {Do actual search}
```

Bibliography

[1] BOINC - Berkeley Open Infrastructure for Network Computing. A software platform for volunteer computing and desktop Grid computing used by projects such as SETI@home., URL `http://boinc.berkeley.edu/`.

[2] Apache. Hadoop. Hadoop is an open source Java software framework for running parallel computation on large clusters of commodity computers., URL `http://lucene.apache.org/hadoop/`.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS*, pages 137–150. 2004. URL `http://labs.google.com/papers/mapreduce.html`.

[4] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Scientific and Engineering Computation, 1994. URL `http://www.csm.ornl.gov/pvm/`.

[5] Hirosi Hitotumatu and Kohei Noshita. A Technique for Implementing Backtrack Algorithms and its Application. *Information Processing Letters*, 8(4):174–175, April 1979.

[6] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathemathics of Computation*, 29(129):121–136, January 1975.

[7] Donald E. Knuth. Dancing Links. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 187–214. Palgrave, Houndmills, Basingstoke, Hampshire, 2000.

[8] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation*. Addison Wesley, 1993. ISBN 0-201-57569-8. CWEB is a software system that facilitates the creation of readable programs in C, C++, and Java., URL `http://www-cs-faculty.stanford.edu/~knuth/cweb.html`.

[9] Arne Maus and Torfinn Aas. PRP - Parallel Recursive Procedures, October 1995. URL `http://heim.ifi.uio.no/~arnem/PRP/`.

[10] Alfred Wassermann. Covering the Aztec Diamond with One-sided Tetrasticks – Extended Version, December 1999. URL `http://did.mat.uni-bayreuth.de/wassermann/`.