Comparative Empirical Analysis of Dancing Links Implementations to Solve the Exact Cover Problem

Andrija Sevaljevic
Department of Computer Science
Idaho State University
Pocatello, Idaho, USA
andrijasevaljevic@isu.edu

Paul Bodily
Department of Computer Science
Idaho State University
Pocatello, Idaho
bodipaul@isu.edu

***Abstract*—Understanding and efficiently solving NP-complete problems remains a fundamental challenge in computational theory (CT). This paper presents XD, a modified Dancing Links algorithm (DLX) that solves the NP-complete exact cover problem. We implement the algorithm using dictionaries and compare this implementation against a more traditional implementation that uses matrices to assess cases in which each implementation performs best. One of the key strengths of the modified algorithm is its simplicity and adaptability. Our implementation is included in Redux, an online, interactive, dynamic knowledgebase of NP-complete problems, reductions, and solution algorithms.**

***Index Terms*—NP-complete, Computational Theory, Time Complexity**

I. Introduction

Across disciplines as varied as biology, economics, English, technology, finance, and business, many of the most interesting and urgent problems remain a challenge because the fundamental nature of these problems is that they require non-trivial and sometimes non-tractable algorithmic solutions. A subset of these problems—known in the field of computational theory as NP-complete (short for nondeterministic polynomial-time complete) problems— is particularly troublesome because efficient algorithms for solving non-trivial instances of these problems do not exist. Such problems include task scheduling, social network analysis, routing packets in a network, and GPS routing.

To address the inefficiency of solving these problems, computer scientists typically take one of two algorithmic approaches. The first approach is to design algorithms that use heuristics that deliberately forfeit an attempt at guaranteeing exact or optimal solutions in order to meet certain time constraints. A second approach is to design algorithms that guarantee exactness and optimality but use clever tricks to reduce exponential runtime time by significant margins compared to more naïve or brute-force approaches.

In this paper, we present our findings in relation to the implementation of an efficient algorithm for finding exact solutions to a prominent NP-complete problem known as the exact cover (EC) problem. The EC problem is defined as follows: given a set of elements U (often called the universal set) and a family of subsets S of U, determine whether or not there exists a subfamily T of S such that all the subsets in T are disjoint and their union is equal to U [1]. This problem shows up in many real-world applications, including job scheduling, class scheduling (Figure 1), electronic music [2], cryptography [3], component-based automation systems [4], and more. A more complex task would be related to genome sequencing, which often involves breaking down the genome into smaller fragments, called reads. The goal of assembling these reads into the original sequence such that each position in the genome is covered exactly once can be formulated as an EC problem.

The preeminent algorithm for solving EC problems is the Dancing Links (DLX) algorithm, developed by Donald Knuth [5], [6]. Knuth's algorithm was implemented in the year 2000 and has few variations on its fundamental approach since its creation. This solver employs a recursive approach to explore a doubly linked list structure to identify an exact solution. It has been further extended to also solve Sudoku [7], N-Queens [8], and Polyomino puzzles [6]. This method has been implemented in DlxLib (https://github.com/taylorjg/DlxLib), a C# library. DlxLib leverages a binary matrix representation, using ones and zeros to construct a custom data structure. It is an open-source project on GitHub and solves Free Flow, EC, and Sudoku.

We present a comparative analysis of the DlxLib implementation against an alternative implementation called XD, that uses dictionaries rather than the doubly linked list found in many EC solvers. To analyze algorithmic performance, we created an EC problem generator and ran trials on randomized instances. The algorithm was inspired by Ali Assaf's Python implementation of the algorithm (Algorithm 1), found at https://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html. We describe how users can access our implementation which is available online via Redux, an interactive, dynamic knowledgebase of NP-complete problems, reductions, and solutions [9]. Although comparative studies have been done on solving exact cover using dancing links versus satisfiability [10], there has been no empirical analysis of the benefits of using dictionaries.

II. Methods

Whereas our original goal had been to implement a solver for Sudoku, it was soon discovered that Sudoku is a variation of the EC problem, and thus our focus shifted towards more

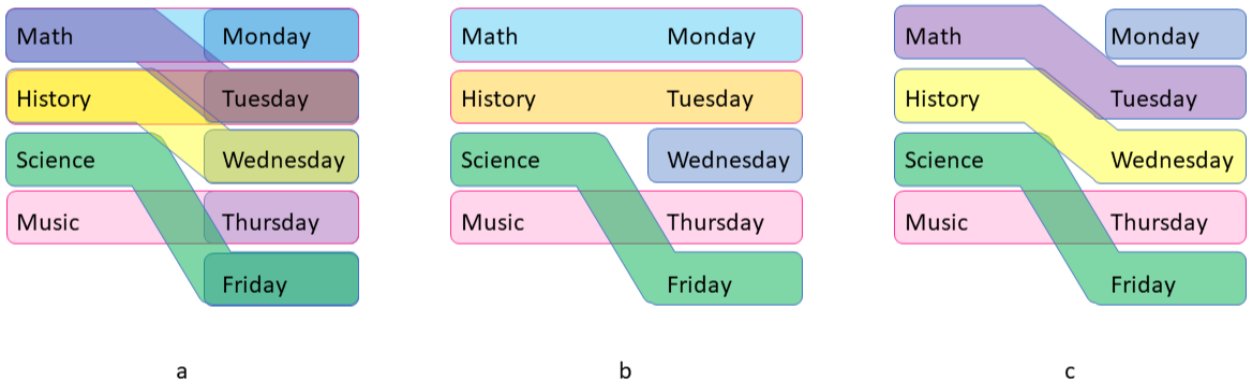

Fig. 1: In this example, the universe U contains 9 elements (4 subjects and 5 days). Each set of circled elements in Figure a represents a subset in the family (defined in the EC problem as S). Each subset consists either of a class and the days of the week on which the class has lecture or just a single day of the week. In this representation, a solution to the EC problem represents a possible schedule that includes all four classes and no class conflicts (allowing for some days on which there are no class lectures). Figures b and c show two possible solutions for (U, S).

general solutions to the EC problem. The DLX algorithm is a well-known, efficient algorithm for solving the EC problem, and the implementation of DLX in DlxLib fulfills the same purpose inside C#. However, it is possible that we can solve EC instances with distinctive traits more efficiently. In this paper, we present results from an empirical analysis to precisely characterize the conditions under which one implementation outperforms the other.

A. *Implementation*

One of the primary constraints for this research was that the algorithm needed to be implemented in C#. One key factor influencing this choice was the existence of DlxLib, a well-documented open-source library in C# that utilizes matrices. Given its prevalence, implementing our variant using dictionaries in the same language was needed for direct comparison. However, the decision to implement the algorithm in C# went beyond mere interest in comparing algorithms in a common language. The most significant advantage emerged from its seamless integration with Redux, our chosen application framework [9]. This integration facilitated extensive testing of numerous instances and enabled straightforward comparisons. An additional benefit was a feature within Redux that allows users to input their instance of EC using standardized mathematical notation (Figure 2), allowing users to interact with the tool without needing the knowledge of transforming their data into an appropriate data structure.

III. Results

To evaluate the effectiveness of each algorithm and compare them, we analyzed two different components of each algorithm. The first was the space complexity of the algorithms, which is related to the data structure used to represent the problem instance. The second metric analyzed was their respective time complexities which were compared by testing their average speeds on instances of different sizes.

Algorithm 1 The pseudocode for an implementation of DLX

```
1:  iterate(Y,X,solution):
2:  if IsEmpty(X) then
3:      return
4:  else
5:      minCol = leastCommonElement(X)
6:      for row in Y[minCol] do
7:          solution.Push(row)
8:          columns = select(Y, X, row)
9:          iterate(Y, X, solution)
10:         deselect(Y, X, row, columns)
11:         solution.pop()
12:
13: deselect(Y,X,row):
14: columns = []
15: for j in Y[row] do
16:     for i in X[j] do
17:         for k in Y[i] do
18:             if k != j then
19:                 X[k].remove[i]
20:     columns.append(X.pop(j))
21: return columns
22:
23: select(Y,X,row,columns):
24: for j in reversed(Y[row] do
25:     X[j] = cols.pop()
26:     for i in X[j] do
27:         for k in Y[i] do
28:             if k != j then
29:                 X[k].add[i]
```

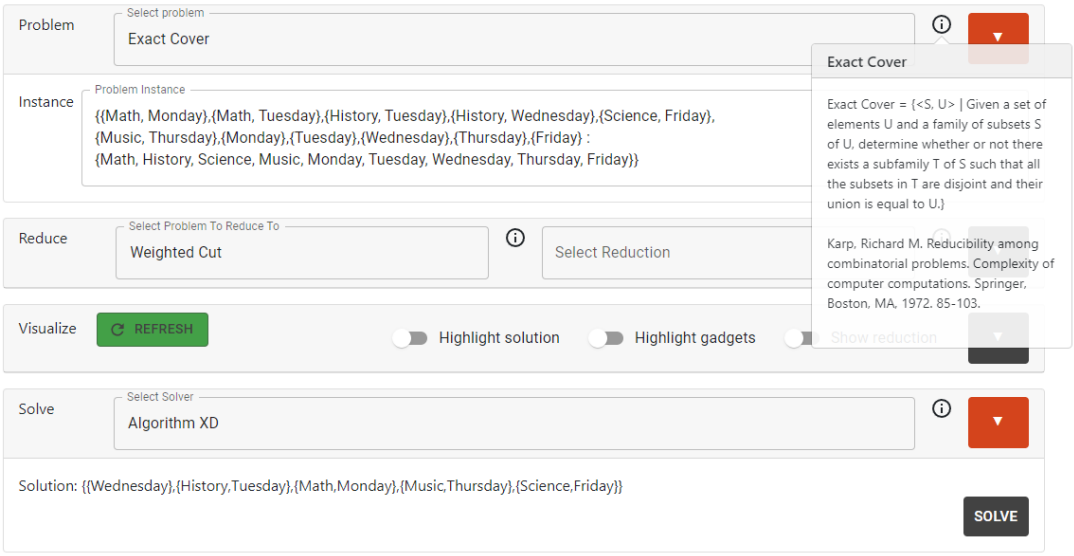

Fig. 2: Within Redux, users have the option to select the EC problem and input their problem instance, which can be solved using Algorithm XD. The example instance solves a class schedule for which there will be no conflicts (Figure 1).

A. Space Complexity

The space complexity of the algorithms was split up into two components: the required memory to store the sets and elements of our EC instance; and the additional storage of recursive functions. The DLX implementation reduces the size of the binary matrix from O(U * S) down to the doubly linked list complexity of O($\bar{x}$ * S + S), where $\bar{x}$ represents the average number of elements in a list. The dictionary data structure in algorithm XD takes a key-value pair where the key represents the set index and the value is a list of elements belonging to that set. The complexity of the dictionary implementation is O($\bar{x}$ * S + S) making it identical in terms of space to the doubly linked list.

Regarding storage of additional stack frames for recursive function calls, the two algorithms share a similar heuristic, so the number of function calls is also similar. In the worst case, the depth of the recursion will be equal to the total number of subsets. This means that the deciding factor in performance will be the method of utilizing these functions.

B. Time complexity

Computing a practical big O complexity for dancing links is non-trivial as conditions vary significantly based on the specifics of the input instance. Furthermore, due to the recursive nature of backtracking algorithms, it further complicates the process of estimating its speed. Efforts have been made to evaluate the complexity, such as David Epstein's blog 11011110 (https://11011110.github.io/blog/2008/01/10/analyzing-algorithm-x.html). where he attempts to approximate dancing links with a doubly linked list implementation. In his article "Analyzing Algorithm X", Epstein estimates a complexity of $O\left(4^{n/5}\right) \approx O\left(1.31951^{n}\right)$ in the worst case scenario, where n is the total number of subsets. In practice, this is not the case and the average time is much lower, making it impossible to have a predictable time evaluation. This is why we must do experimental trials to conclude the efficiency of our algorithms.

To fairly and conveniently compare the speeds of the algorithms, an EC problem generator was implemented within the program. The generator had four different input variables to generate the instances: universe size, size of each set, deviation from set size, and number of subsets. During testing the deviation from set size was 10%. This allowed us to accordingly change the needed parameters and quickly create dozens of instances to test on the two algorithms. The parameters used to test the algorithm speed were the number of elements in the universe set (U) and the number of elements in the subset set (S). For each set of parameters, a total of eleven runs were done, and in each trial, the first run was discarded as the algorithm performance needed to normalize. In each test run, both algorithms were tested using the same instances of EC. It is important to note that the EC generator only generates instances with at least one solution.

C. Empirical Analysis

The results of the controlled trials are visualized on the heat map (Figure 3), using information gathered from each algorithm's performance (Figure 4 and Figure 5), which provide evidence that under certain circumstances, an implementation of DLX using dictionaries is more effective than the current DlxLib algorithm. There is a common pattern that can be noticed for the dictionary variation. Unlike Knuth's implementation, which always grows in time complexity as both variables increase in size, the variant using dictionaries, it is not the case. Instead, when the universe size is increased, the time needed to find a solution goes down until it reaches its lowest peek and then goes up again. However, the graphs do

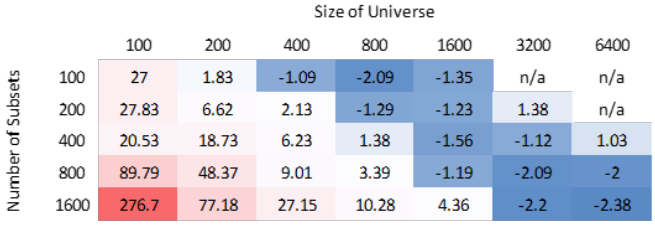

Fig. 3: Heatmap of performance comparison of Algorithm XD versus DlxLib. The number in each cell represents the ratio of the algorithms' speeds. The color blue represents variable entries for which Algorithm XD performed better, meanwhile red indicates DlxLib performing better.

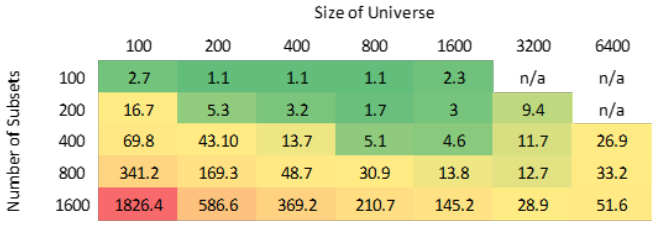

Fig. 4: Heatmap of Algorithm XD performance. The number in each cell represents the average time it took to solve the instances.

not provide an indication on when or if the time needed to solve an instance will be slower than DlxLib again, mainly because such cases would be trivial. The reason why, is because there would not be enough elements in the subsets to cover the entire universe. This means if we know enough details about the instances, such as a low number of expected solutions, alongside a lower number of subsets compared to the universe.

IV. Conclusion

In today's world NP-hard problems play an important role as they have many applications whether it be organizing the most efficient route or assembling DNA sequences. This is why being able to understand and solve these problems efficiently is important for researchers and computer scientists. One such problem is EC which has applications for crew scheduling, benchmarking and across other disciplines and industries. The most famous solver for EC is Donald Knuth's DLX which uses a binary matrix data structure to solve instances of the EC problem. We provide an empirical analysis of an existing implementation of DLX with an alternative implementation of the algorithm using dictionaries, which provides faster solutions depending on the circumstance of the problem instance. The different implementations are compared to each other by manipulating the values of different variables of the problem. Our future work will include implementing new heuristics for Algorithm XD and comparing the two algorithms with high speed computers.

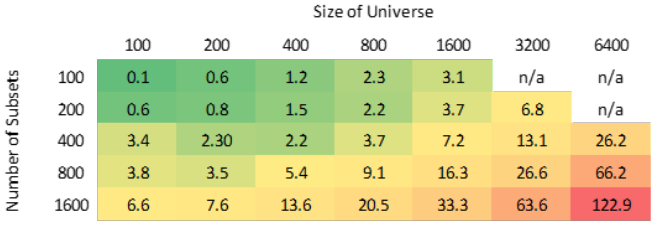

Fig. 5: Heatmap of DlxLib performance. The number in each cell represents the average time it took to solve the instances.

References

[1] R. M. Karp, "Reducibility among combinatorial problems," pp. 85–103, 1972.

[2] B. Bemman and D. Meredith, "Exact cover problem in milton babbitt's all-partition array," in *International Conference on Mathematics and Computation in Music*. Springer, 2015, pp. 237–242.

[3] S. K. Bisoyi and H. Reza, "Toward securing cyber-physical systems using exact cover set," *J Inform Tech Softw Eng*, vol. 7, no. 198, p. 2, 2017.

[4] J. M. Tjensvold, "Generic distributed exact cover solver," *URL http://decs. googlecode. com*, 2007.

[5] D. E. Knuth, "Dancing links," 2000.

[6] M. Nishino, N. Yasuda, S. ichi Minato, and M. Nagata, "Efficient algorithm for enumerating all solutions to an exact cover problem," vol. 15, pp. 19–23, 2017.

[7] M. Hunt, C. Pong, and G. Tucker, "Difficulty-driven sudoku puzzle generation," *The UMAP Journal*, vol. 29, no. 3, pp. 343–362, 2007.

[8] R. D. Chatham, M. Doyle, J. J. Miller, A. M. Rogers, R. D. Skaggs, and J. A. Ward, "Algorithm performance for chessboard separation problems," *J. Combin. Math. Combin. Comput*, vol. 70, pp. 127–142, 2009.

[9] K. Marchetti, A. Sevaljevic, A. Diviney, R. Phillips, C. Eardley, R. Khadka, D. Igbokwe, and P. Bodily, "Redux: An interactive, dynamic knowledge base for teaching NP-completeness," in *Proceedings of the 29th annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2024, accepted for publication.

[10] T. Junttila and P. Kaski, "Exact cover via satisfiability: An empirical study," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2010, pp. 297–304.