A Tangram Puzzle Solver in Common Lisp

Michael Wessel
lambdamikel@gmail.com
Lambda @ Home
Palo Alto, California, USA

ABSTRACT

In this demo, we present a Common Lisp program capable of solving arbitrary geometric tiling problems. Well-known examples of such problems are Tangram puzzles. The program is implemented in Common Lisp, using the Common Lisp Interface Manager (CLIM) for its GUI. It consists of an editor and solver. We describe and demo the program and comment on its implementation as well as employed techniques to make the approach feasible.

CCS CONCEPTS

• **Computing methodologies** → **Discrete space search**; *Continuous space search*; **Spatial and physical reasoning**; • **Theory of computation** → **Computational geometry**; • **Mathematics of computing** → **Combinatoric problems**; **Solvers**; • **Human-centered computing** → **Graphical user interfaces**; **User interface programming**.

KEYWORDS

Tiling problems, Tangram, Search, Common Lisp, Heuristics, Common Lisp Interface Manager (CLIM), Geometrical Reasoning, Computational Geometry, NP-Hard Problems

ACM Reference Format:
Michael Wessel. 2021. A Tangram Puzzle Solver in Common Lisp. In *Proceedings of the 14th European Lisp Symposium (ELS'21).* 5 pages. https://doi.org/10.5281/zenodo.4711456

1 INTRODUCTION

Geometric tiling problems have always fascinated humanity. Tiling problems occur naturally in many domains, from bathroom floor designs, over new forms of matter organized in quasicrystals via Penrose tilings [13, 19], to Turing-complete Domino systems used for establishing undecidability of computational problems. Tiling problems also provide for entertaining and challenging leisure time activities in the form of dissection puzzles, most prominently, Tangram puzzles [20].

A program capable of solving arbitrary finite – and hence decidable – tiling problems could be of great practical utility. The long term goal of this endeavor is to create a program capable of solving Penrose tilings, as shown in Fig.2. However, the more modest goal of solving Tangram puzzles was selected as a first milestone on this journey. A first version of this program was running in 2003 [17]. Given the tremendous advances in computer performance since 2003, the author was curious whether the program would be more successful in finding solutions, and hence resurrected, refursbished, and reworked the heuristics of the original program.

ELS'21, May 03–04 2021, Online, Everywhere

https://doi.org/10.5281/zenodo.4711456

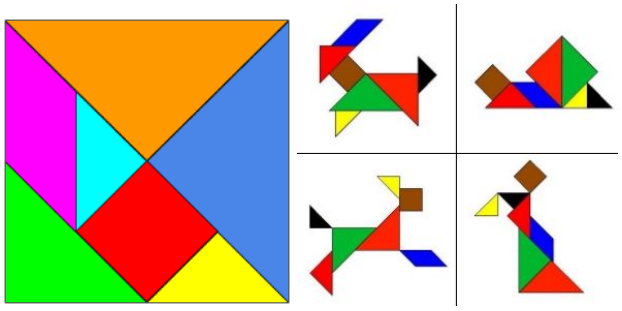

Figure 1: Tangram Tiles & Tangram Problems with Solutions

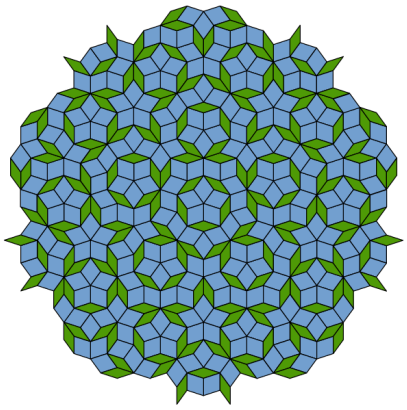

Figure 2: Penrose Tiling (Source: Wikipedia, CC)

The set of standard Tangram tiles and some problems and their solutions are shown in Fig.1. Only the outline (silhouette) of the problem polygon is given, and the goal is to cover (or partition) the silhouette polygon completely with the available tile polygons, such that tiles do not overlap. Naturally, tiles can can be translated, rotated, and reflected. In a standard Tangram problem, each tile must be used exactly once.

Unlike many other classic search problems (8-Queens, Othello, Checkers, …), which are standard AI textbook examples these days [9], the geometric tiling problem addressed here is considerably more complex from an implementation point of view, given its geometric nature.

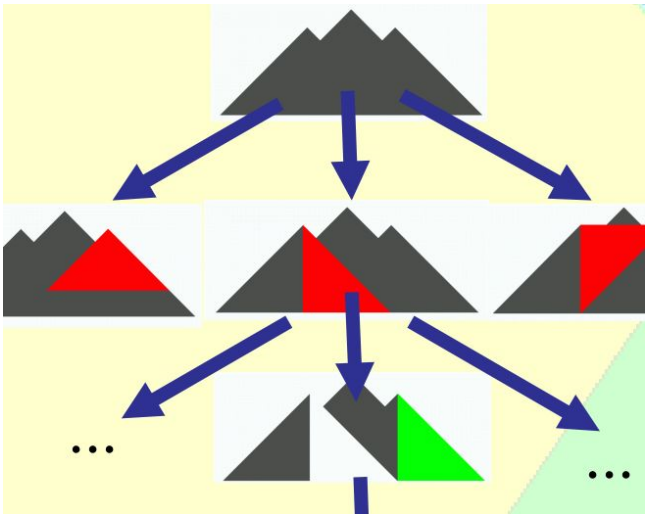

Figure 3: Solving Tiling Problems by Geometric Search

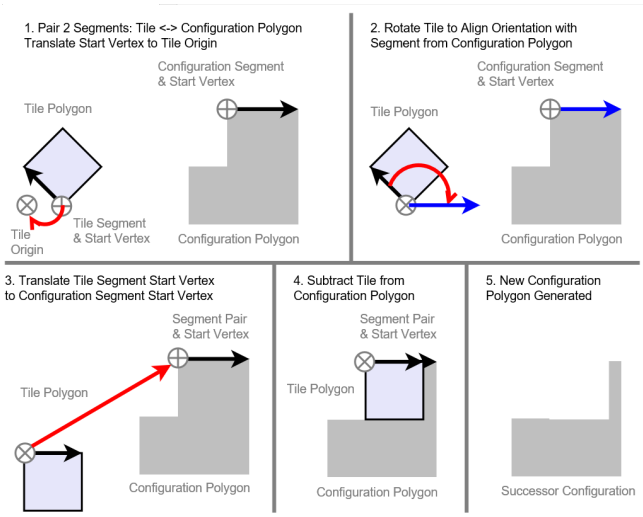

Figure 4: Generating Successor Configurations

2 SOLVING GEOMETRIC TILING PROBLEMS

Given a set of tiles $\mathscr{T}$ such as the standard Tangram tiles shown in Fig.1, and the outline (silhouette) of a tiling problem (note that the concrete arrangements of tiles that constitute its outline is unknown), one way of solving a tiling problem is by geometric search.

The search space and process is illustrated in Fig.3. Starting with the initial problem polygon as the current configuration polygon (grey root at the top in Fig.3), possible successor configuration polygons are generated by finding possible alignments of all the remaining tiles $\mathscr{T}$ with the problem polygon $\mathscr{C}$. Each aligned polygon tile is then subtracted from the current configuration polygon. The search continues with the remainder of the configuration polygon, until it is empty, and has hence been completely covered – a solution to the tiling problem has been found. Depending on the tiling problem, for Tangram puzzles it is required that each tile must be used exactly once, but the solver can also search for solutions that admit less tiles or allow multiple copies of the same tile to be used (e.g., for Penrose tilings).

Note that the current configuration polygon is potentially split into multiple components, as illustrated with the bottom configuration in Fig.3. In our solver, a configuration $\mathscr{C}$ hence consists of a set of (not necessarily convex) polygons without holes, and their union represents the remaining area that still needs to be partitioned: $\mathscr{C} = \{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$.

A central function in every search algorithm is the successor state generator. Here, given a selected tile and the current configuration, the function `find-possible-alignments` finds reasonable ways of aligning a tile with a configuration, and returns a set of successor configurations: $\mathscr{SC} = \{\mathscr{C}_1, \ldots, \mathscr{C}_n\}$). The search is performed either depth-first or breadth-first. As a central heuristics, the successor generator will always try to align a tile with the outer edges of the current configuration – this is a strategy that humans use when solving Jigsaw puzzles, but it would probably fail for Penrose tilings.

The number of potential successor configurations β (branching factor) can be very large, and for given configuration $\mathscr{C}$ with a set of (remaining) tiles $\mathscr{T}$ given by the formula

$$\beta = \sum_{c \in \mathscr{C}} \sum_{t \in \mathscr{T}} \#\,\text{find-possible-alignments}(t, c)$$

Heuristics are essential in order to make the search succeed. Three types of selection heuristics are used to advise the search as follows:

Component selection heuristic: which of the component polygons is tackled next?

Tile selection heuristic: which of the remaining tiles will be tried next for alignment with the selected component polygon?

Successor configuration selection heuristic: for the selected component polygon and tile, which of the generated potential successor configurations (see Fig.3) will be tried next?

These heuristics are implemented as scoring functions, calculating geometric characteristic of polygons such as polygon area, number of polygon components, and compactness, which is defined as the ratio of the polygon area to the area of a circle with the same perimeter. These are passed as `:key` arguments to the `sort` function for prioritization of the generated successor configurations, i.e., the *ordered* set $\mathscr{SC} = \{\mathscr{C}_1, \ldots, \mathscr{C}_n\}$). Configurations $\mathscr{C}_i$ that yield higher scores are prioritized in the search. Two configurations are considered equal in this set if they are *congruent*.

We have implemented and experimented with an array of different geometric functions for these three types of heuristics and found a configuration that ensures that many Tangram problems can be solved in a reasonable amount of time.

The computation of successor configurations / possible alignments is illustrated in Fig.4. For a given tile $t \in \mathscr{T}$ and configuration polygon $c \in \mathscr{C}$, all possible pairings of segments from the configuration polygon and the tile polygon are considered. Each such segment pair can result in a potential successor configuration. For a chosen pair of segments, the tile polygon is first translated and

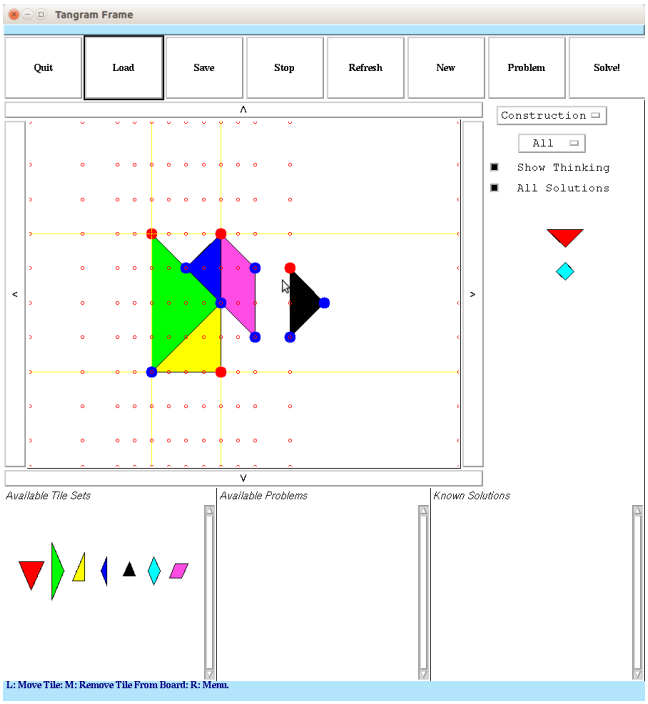

Figure 5: Tangram Problem Editor

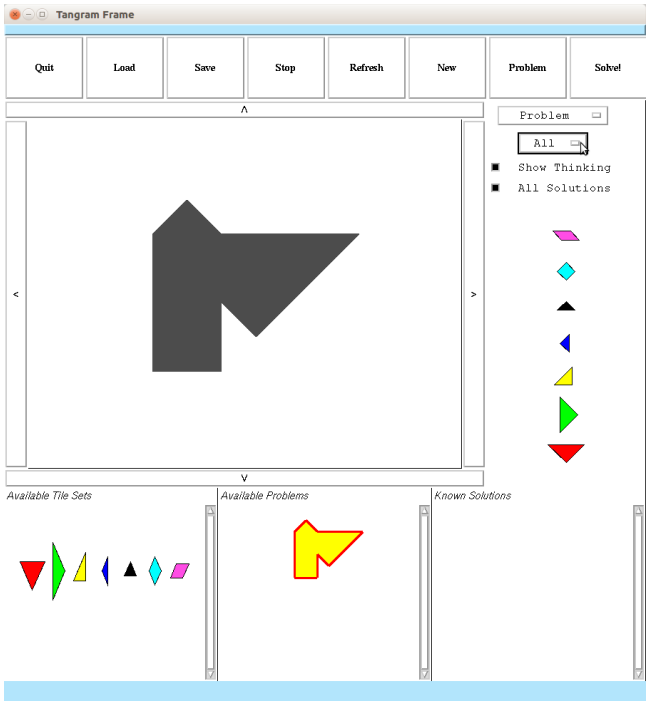

Figure 6: Tangram Problem Library

rotated to match the orientation of the configuration segment, and then translated onto the configuration polygon. If the transformed tile polygon then happens to lie completely on the inside of the configuration polygon (note that it may also overlap or lie fully on the outside as a result of the transformation), then the transformed tile polygon is subtracted from the configuration polygon, yielding a new successor configuration polygon. Polygon subtraction might result in several (disconnected or only connected over a shared single vertex, i.e., touching in a point) component polygons.

As a further complication, note that in order to accommodate for *reflections*, the individual tiles in the tile set $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ are actually sets of polygons themselves – each tile $\mathcal{T}_i$ consist of a set of (simple) non-congruent polygons which are its *reflected variants*. Hence, $\mathcal{T}_i = \{\mathcal{TV}_{i,1} \ldots, \mathcal{TV}_{i,m}\}$, and the solver ensures that only one variant is used of each tile. Note that $\#\ \mathcal{T}_i = 1$ for all but the parallelogram tile.

3 THE TANGRAM SOLVER APPLICATION

The GUI of the solver is written using the Common Lisp Interface Manager, CLIM [5, 18]. The standard workflow for using the program is the following:

(1) The graphical editor, see Fig.5, offers rapid construction of Tangram problems. First, an active set of (construction) tiles is determined from the "Available Tile Sets" pane. The selected tile set then appears on the right side pane of the editor, from where they can be pulled into the editor area. Once in the editor area, tiles can be moved around, rotated (by 45 degrees), reflected, etc. CLIM gestures and context menus are available on the tiles. An active adaptive grid for proper alignment makes precise problem construction a breeze.
(2) Once the arrangement is complete, the "Problem" button is pushed to record a Tangram problem. Only the polygon outline of the arrangement is recorded, not the tiles and their actual arrangements. The constructed problem appears in the "Available Problems" pane, see Fig.6. From there, context menus for selecting and deleting the problem are available.
(3) To start the problem solver, first the desired options are determined (whether to compute all solutions or only one, and if all tiles, fewer or more tiles are admissible). Then, the "Solve!" button starts the engine. If "Show Thinking" is checked, the tiling process is visualized and the program's progress can be witnessed, see Fig.7. Visualizing the program at work comes with some performance penalties, but is very helpful for judging the effectiveness of the solver's heuristics. The solver can be stopped at any time. Computed solutions show up in the "Known Solutions" pane visible in Fig.7 as soon as they are found.

The usual convenience functions are offered as well; e.g., problems and solutions can be saved to (and loaded from) disk, and so forth.

3.1 Performance

We haven't conducted extensive performance evaluations yet, but let us consider the Tangram problem in Fig.6.

- With the heuristics enabled, an older 2012 iCore 7-2670QM 2.2 Ghz Xenial Ubuntu PC with 8 GBs of RAM finds the first (dozen) solutions after 10 seconds, see Fig.8.

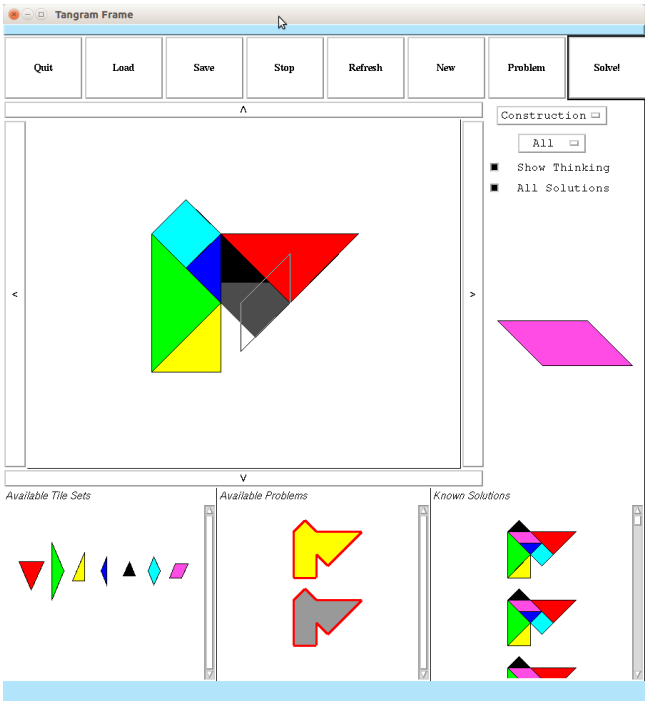

Figure 7: Watching the Solver at Work

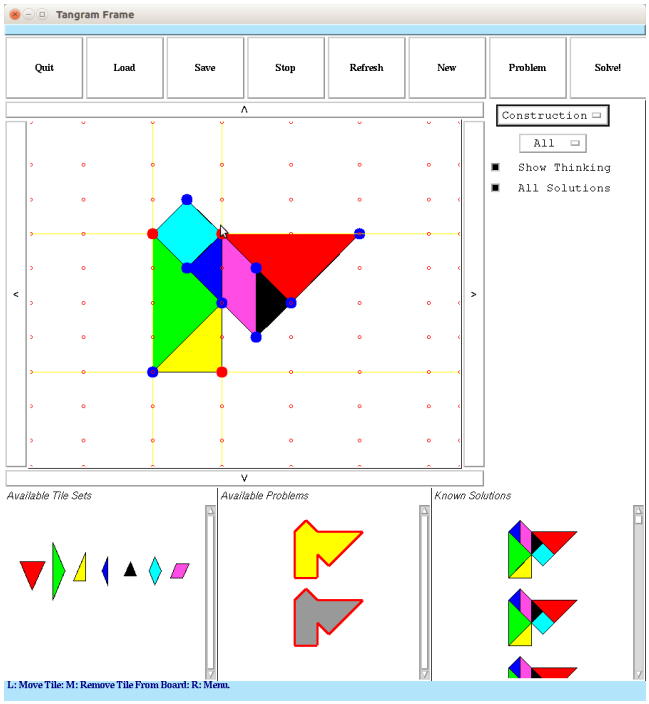

Figure 8: Unexpected Solutions found by the Solver

- With the heuristics disabled, the author stopped the search after 1.25 hours with no solution so far, nicely demonstrating the necessity of intelligent geometric heuristics for this approach, even if these heuristics are expensive to compute due to their geometric nature.

It is interesting to note that the solver frequently finds creative, new solutions to problems, i.e. solutions that are different from the original tile arrangement used to construct the problem. Search sometimes leads to surprising and unexpected solutions. Novel solutions to standard Tangram problems from the Internet, e.g. the problems shown in Fig.1, are frequently found by the solver. A concrete example of this is shown in Fig.8, where the first solutions (in the "Known Solutions" pane) are clearly different from the tile arrangement that was used for constructing the problem silhouette shown in the editor in the first place.

4 IMPLEMENTATION DETAILS

The author has implemented the geometric substrate of the solver over a number of years as part of his CS master and PhD thesis' work, from 1996 to 2003 [15, 16]. This base layer encompasses CLOS classes for geometric object representation (polygons, multiple-component polygons, lines, chains, points, segments), and metric, topological and geometric basic functions and methods (computation of angle, length, orientation, area, and compactness computation; line intersection, point-in-polygon test, methods for polygon union, intersection and difference, etc.)

Importantly, qualitative spatial relationships (Cohn's well-known *RCC8 relationships* [2, 11, 12]) for polygons are used for determining if a pair of polygons are congruent (RCC8 relation *Equal*), and if a candidate tile alignment is valid. Valid aligned tiles must be in a *Tangential Proper Part Inverse* relation like in Fig.3, unlike the invalid *Proper Overlap* relation from Fig.7.

All of the geometric objects can be dynamically affected by nested transformation matrices, similar to the ones found in CLIM (`with-translation, with-rotation`). Rotating tiles causes floating-point precision challenges, e.g., congruency tests might fail because of inaccuracies from cos, sin multiplications. Hence, floating point rounding and an epsilon equality predicate (`=-eps`) is employed to compensate for these inaccuracies – note that some rotations correspond to irrational numbers, which cannot be represented precisely by Common Lisp's rational numbers.

Due to high-level support functions, the main covering algorithm and its heuristics are only ≈ 500 Lines of Code (LOC). However, the underlying geometric substrate implementation contributes ≈ 3400 LOC (not all of its functionality is being used though), plus another ≈ 1800 LOC for the CLIM GUI. There is also a library of general purpose helper functions over ≈ 1000 LOC (only very few are being used); as a rough overall estimate, the application consists of ≈ 5000 LOC.

The program is OpenSource under GPL3 [7], and a YouTube video of the program in action is available as well [8]. The current version requires LispWorks (6 or 7) with CLIM, and runs on Windows, Linux, and Mac. Windows and Linux executables can be found in the GitHub repository as well [7].

5 RELATED & FUTURE WORK

This is a demo paper, so we do not attempt to survey the scientific literature on solving dissection or assembly puzzles in depth.

However, we would like to present some pointers for the interested reader for further study, and put our work into context and perspective.

According to [22, 23], Tangram puzzles are an example of irregular shape packing problems, which are known to be NP-hard. Analyzing the computational complexity of special classes of (dissection) puzzles is an active field of computational geometry research; for example, it was shown only recently that symmetric assembly puzzles are NP-complete [3].

A plethora of Tangram puzzle game programs and apps is available on the Internet; however, with the notable exceptions of [1, 6, 21], these programs are typically incapable of solving new Tangram problems (without resorting to pre-recorded solutions).

The earliest approach for solving Tangram puzzles was given by Deutsch & Hayes [4] in 1972. Similar to the author's program, their algorithm is geometric in nature, and geometric heuristics are proposed. An early neural network-based approach is presented in [10], and a solver utilizing raster-based mathematical morphology in [23], working on bitmap raster representations. Interestingly, geometric search is not always the preferred line of attack for Tangram solving. In our opinion, it is the most straightforward and most intuitive method though.

Our solver [17] was developed without knowledge of previous work in the area, and precedes the papers [22, 23] as well as the programs [1, 6, 21]. It presents a somewhat naïve approach, mimicking some aspects of human Tangram solving. As such, the performance is probably not as good as some of the algorithms presented in the literature, but an in-depth comparison has not been performed yet. Our program performs surprsingly well on many standard Tangram problems, but can certainly benefit from refined heuristics and future research. A more thorough comparison of our approach with algorithms and heuristics from the scientific literature should hence be undertaken in the near future. Moreover, a set of Tangram benchmark problems is available [22] and could be used for comparative performance and completeness evaluations in the future.

One limitation of our current algorithm is that it does not accommodate for Tile scaling during the alignment process. The scale of the tiles could probably be computed as the square root of the quotient of the area of the silhouette polygon and the sum of the Tangram tile areas, assuming the presented problem has a solution. Another shortcoming of our current algorithm is that it cannot deal with polygons with holes; standard Tangram problems don't require polygons with holes, but Tangrams problems with holes exist as a variant of the game.

The author intends to develop this further, and will eventually also consider more complex tiling problems than Tangrams, especially the aforementioned Penrose tilings. More complex matching rules will have to be implemented then. A thorough effectiveness evaluation of the already implemented geometric heuristics on different problems should be conducted before proceeding with more sophisticated heuristics.

Another action item is to port the application to SBCL and McCLIM [14]; currently, it requires LispWorks CLIM 2.

ACKNOWLEDGMENTS

The author would like to thank Bernd Neumann for providing the initial motivation and funding for this work [17], the anonymous reviewers for thoughtful comments, and Rainer Joswig (aka Lispm) for compiling the application with LispWorks 7.1 on a Silicon M1 Mac and streamlining the sourcecode for LispWorks application delivery.

REFERENCES

[1] axel7083. Tangram solver in python 3, 2021. URL https://github.com/axel7083/tangram-solver.
[2] A. G. Cohn, B. Bennett, J. M. Gooday, and N. Gotts. RCC: a calculus for region based qualitative spatial reasoning. *GeoInformatica*, 1:275–316, 1997.
[3] Erik D. Demaine, Matias Korman, Jason S. Ku, Joseph S.B. Mitchell, Yota Otachi, AndrÃ© van Renssen, Marcel Roeloffzen, Ryuhei Uehara, and Yushi Uno. Symmetric assembly puzzles are hard, beyond a few pieces. *Computational Geometry*, 90: 101648, 2020. ISSN 0925-7721. doi: https://doi.org/10.1016/j.comgeo.2020.101648. URL https://www.sciencedirect.com/science/article/pii/S0925772120300420.
[4] E.S. Deutsch and K.C. Hayes. A Heuristic Solution to the Tangram Puzzle. *Machine Intelligence*, 7:205–240, 1972.
[5] Franz Inc. *Common Lisp Interface Manager (CLIM) 2.0: User Guide*, 1994.
[6] Rachel Eaton (Invisibility17). Tangram solver written in python 3 using opencv, 2019. URL https://github.com/Invisibility17/tangram_solver.
[7] Michael Wessel (LambdaMikel). Tangram github repository, mar 2021. URL https://github.com/lambdamikel/Common-Lisp-Tangram-Solver.
[8] Michael Wessel (LambdaMikel). Tangram youtube demo video, mar 2021. URL https://www.youtube.com/watch?v=UUn_np8I3zg.
[9] P. Norvig. *Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.
[10] K. Oflazer. Solving Tangram Puzzles: A Connectionist Approach. *Journal of Intelligen Systems*, 8(5):603–616, 1993.
[11] D. A. Randell, Z. Cui, and A. G. Cohn. A Spatial Logic based on Regions and Connections. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning*, pages 165–176, 1992.
[12] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. pages 165–176, 1992.
[13] P. Steinhardt. *The Second Kind of Impossible: The Extraordinary Quest for a New Form of Matter*. Simon & Schuster, 2019.
[14] Robert Strandh and Timothy Moorey. A Free Implementation of CLIM. 10 2002. URL https://common-lisp.net/project/mcclim/static/documents/clim-paper.pdf.
[15] Michael Wessel. *Flexible und konfigurierbare Software-Architekturen für datenintensive ontologiebasierte Informationssysteme*. doctoralthesis, Technische Universität Hamburg, 2008. URL http://tubdok.tub.tuhh.de/handle/11420/834.
[16] Michael Wessel and Ralf Möller. Flexible software architectures for ontology-based information systems. *Journal of Applied Logic*, 7(1):75–99, 2009. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2007.07.006. URL https://www.sciencedirect.com/science/article/pii/S157086830700064X. Special Issue: Empirically Successful Computerized Reasoning.
[17] Michael Wessel and Bernd Neumann. Ein Tangram-spielendes LISP-Programm, 2003. URL https://www.michael-wessel.info/papers/tangram-poster.pdf. Exponat auf der "Informatik-Expo 2004", Fachbereich Informatik, Univeristät Hamburg.
[18] Wikipedia. Common lisp interface manager (clim), mar 2021. URL https://en.wikipedia.org/wiki/Common_Lisp_Interface_Manager.
[19] Wikipedia. Quasicrystal, mar 2021. URL https://en.wikipedia.org/wiki/Quasicrystal.
[20] Wikipedia. Tangram, mar 2021. URL https://en.wikipedia.org/wiki/Tangram.
[21] Ziyi Wu (Wuziyi616). Tangram solver written in python 3 using opencv, 2019. URL https://github.com/Wuziyi616/Artificial_Intelligence_Project1.
[22] Fernanda Yamada and Harlen Batagelo. A comparative study on computational methods to solve tangram puzzles. In *30th Conference on Graphics, Patterns and Images (SIBGRAPI)*, 10 2017.
[23] Fernanda Yamada, Joao Gois, and Harlen Batagelo. Solving tangram puzzles using raster-based mathematical morphology. In *32th Conference on Graphics, Patterns and Images (SIBGRAPI)*, pages 116–123, 10 2019.