

Exact and Approximate Techniques for Solving General Tangrams

Mark Sabini – `msabini` & Brian Zhang – `bhz`

December 16, 2016

1 Introduction

Tangrams - just seven simple pieces. For centuries, humans have applied their spatial and analytical skills to crack configurations ranging from the majestic horse to the graceful swan. Yet, despite all the advances in computer technology, computerized tangrams solving has remained a relatively untouched problem. Among the few papers that address the puzzle, one paper by Oflazer [1] has implemented a tangram solver, but it makes the extremely strong assumption that the vertices of all tangram pieces are lattice points. Another paper by Deutsch [2] gives various heuristics for solving tangrams, but only addresses the classic 7-piece case. Thus, we aim to develop techniques to solve general tangrams, i.e. tangrams with as few underlying assumptions as possible.

2 Background

In its most basic form, a tangrams problem $T = (P, t)$ consists of a set of polygonal pieces $P = \{p_1, p_2, \dots, p_n\}$ and a polygonal target t . The high-level objective is then to rotate, translate, and/or flip the pieces to find an arrangement where $\bigcup_i p_i = t$. We refer to this problem as **general tangrams**, as the pieces and target can be highly irregular, and even non-convex. For ease of implementation, we make the additional assumption that pieces cannot be flipped.

Explicitly, we utilize the Polygon type from the Python library Shapely in order to represent the pieces and target. Under our scheme, the set of pieces P becomes a list of Polygons, and t becomes a single Polygon. In addition, given a known initial arrangement of p_1, \dots, p_n relative to t , we can represent a solution to $T = (P, t)$ as $S = (x_1, y_1, \theta_1, x_2, y_2, \theta_2, \dots, x_n, y_n, \theta_n)$, where x_i , y_i , and θ_i are the x -displacement, y -displacement, and rotation of p_i from its initial configuration, respectively.

Given this representation, we can then define the following general tangrams problems:

- T_{trap} : 4-piece tangrams consisting of identical 45-45-90 triangles and a trapezoidal target. While this is small enough to be solved by brute force, it is complex enough to check basic correctness.

- T_{sq} : Classic 7-piece square tangrams, challenging as the square target lacks much edge information. This problem is depicted in Figure 1.
- T_{sq}^2 : Classic 14-piece rectangular tangrams, consisting of two instances of T_{sq} side-by-side. This is computationally challenging due to the great number of pieces.
- $T_r(k)$: Randomized k -piece tangrams, where the piece set P consists of k arbitrary pieces.

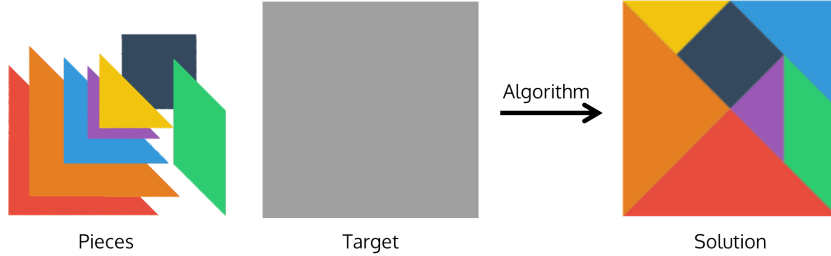


Figure 1: **Visual Problem Formulation for T_{sq}**

3 Task Definition

Our objective is to devise both exact and approximate methods to solve general tangrams problems. Mathematically, given a solution S , we can define our loss with respect to a given problem $T = (\{p_1, \dots, p_n\}, t)$ as:

$$\mathcal{L}_T(S) = \text{Area} \left(\text{Diff} \left(t, \bigcup_{i=1}^n p'_i \right) \right) \quad (1)$$

Above, Diff refers to the difference of two polygons, and p'_i refers to p_i after applying the respective translations and rotation prescribed by S . As a result, solving general tangrams is equivalent to finding the optimal solution S_{opt} such that:

$$S_{\text{opt}} = \arg \min_{S=(x_1, y_1, \theta_1, \dots, x_n, y_n, \theta_n)} \mathcal{L}_T(S) \quad (2)$$

3.1 Generating Random Tangrams Problems

In order to check robustness and correctness of our methods, it is important to generate problems such as $T_r(k)$, which inherently require a general tangrams solver. To do this, we first generate k polygons p_1, p_2, \dots, p_k , where each polygon is the result of selecting a number of vertices $n \sim (3 + \text{Poi}(1))$, generating n points, and taking their convex hull. From here, we build the target t by iteratively joining the side of a random unused piece to a side of the target, and ensuring that the resulting union remains a polygon.

3.2 Correctness/Profiling Utilities

Given the relative difficulty of visualizing a solution from just the final coordinates of the pieces, we developed a JavaScript tangrams visualizer¹ to allow us to step through and see a solution piece-by-piece. By outputting solutions as JSON feeding this raw data into the visualizer, we were able to quickly assess the correctness of our approaches. In addition, we wrote a simple Timer class to help us profile and optimize our code. The combination of these two tools proved instrumental to the development of our general tangrams approaches.

4 Exact Approaches

For our exact approaches, we cast tangrams solving as a search problem. If we are provided a problem $T = (P_0 = \{p_1, \dots, p_n\}, t_0)$, we can represent states as $(P = \{p_{i_1}, \dots, p_{i_k}\}, t)$, where $P \subseteq P_0$ represents the set of pieces yet to be placed, and t represents the current target silhouette. The start state is then $s_{\text{start}} = (P_0, t_0)$, and a state (P, t) is an ending state iff $P = \emptyset$ or t has no exterior. From any given state (P, t) , a valid action consists of selecting a piece $p \in P$, placing it on the target, and taking the difference. Thus, the resulting successor state is $(P - \{p\}, \text{Diff}(t, p))$, with cost $\text{Area}(\text{Diff}(t, p)) + \text{Area}(p) - \text{Area}(t)$.²

There remains one issue with this formulation of the tangrams search problem, in that given a selected piece, there are infinitely many positions in which it can be placed, and thus infinitely many potential actions. To address this, we use the **corner heuristic**, which enforces that a piece p is placed on a target t subject to the following conditions:

- A corner c_p of p must coincide with a corner c_t of t . (Corner-match)
- One of the edges surrounding c_p must coincide with one of the edges surrounding c_t . (Edge-match)

Adopting the corner heuristic allows for the search space to be finite without jeopardizing the correctness of the tangrams search problem.

4.1 Recursive Backtracking

One natural approach to solving the tangrams search problem is to apply recursive backtracking, pruning when the cost of a given action exceeds $\epsilon = 1 \times 10^{-8}$. To attempt to speed up the runtime of recursive backtracking, we applied the **angle heuristic**, which when used with the corner heuristic, forces an entire corner of the target to be filled at once with pieces whose coinciding corner angles sum to the target's corner angle.

4.2 Uniform Cost Search

As the costs of all actions are non-negative, we can also apply Uniform Cost Search to solve the tangrams search problem. Unfortunately, it is not possible to apply A^* to optimize runtime.

¹The visualizer, preloaded with results from select runs, is available at marksabini.com/cs221-tangrams

²This is exactly the area of the piece that lies outside the target, so a valid tangrams solution has cost 0.

This is because the optimal solution to tangrams has 0 cost, so no non-trivial consistent heuristic will exist for use in A^* .

5 Approximate Approaches

While the exact approaches mentioned above will always perfectly solve tangrams, the high branching factor in the search problem makes them unwieldy in practice without the aid of special move heuristics. Thus, we also introduce various approximate methods to attempt to solve T_{sq} in reasonable time.

5.1 Loss Minimization

We note that instead of denoting a state by the pieces/target tuple (P, t) , we can represent a state by a tentative solution $S = (x_1, y_1, \theta_1, \dots, x_n, y_n, \theta_n)$. By doing this, we effectively target the tangrams objective itself by finding the optimal solution S_{opt} that minimizes $\mathcal{L}_T(S)$. Now, to derive our initial state S_0 , we simulate a single branch of Uniform Cost Search by taking random actions until we reach an end state. This produces a configuration adhering to the corner heuristic, C' . As solutions are expressed relative to the starting configuration C_0 ,³ we calculate the translation and rotation necessary to transform each piece from its position in C_0 to its new position in C' , and set S_0 accordingly.

Now that we have our initial state S_0 and objective \mathcal{L}_T , we can now apply the various optimization methods from SciPy. The specific techniques used for the loss minimization technique were Nelder-Mead, modified Powell (Powell), Constrained Optimization By Linear Approximation (COBYLA), and Sequential Least Squares Programming (SLSQP).

5.2 Genetic Algorithm

In the interest of avoiding local optima traps, we also solve tangrams using a genetic algorithm. In this technique inspired by natural selection, we begin with a population $A = \{S_1, S_2, \dots, S_N\}$ of N tentative solutions resulting from exploring random branches of UCS. Each member $S \in A$ has a cost $\mathcal{L}_T(S)$, with a lower cost indicating higher fitness, and we refer to the elements of a solution S as its **genes**.

In iteration i of the genetic algorithm, we assign weights w_1, w_2, \dots, w_N to the members of the population, where $w_k \propto \exp\left(-K \cdot \frac{i \cdot \mathcal{L}_T(S_i)}{M}\right)$. We thereafter normalize the weights to sum to 1. Here, K functions as a tunable learning rate, and M is the maximum number of iterations for which to run the genetic algorithm. To generate a member of the next generation, we sample two parents S_{p_1}, S_{p_2} with replacement according to the distribution of weights, which favors members that are fitter. We subsequently breed S_{p_1} and S_{p_2} to produce a child S' by randomly choosing a parent from which to inherit each gene. S' then becomes the result of a crossover, or mixing, between its two parents. We generate a total of $N - R$ new members using this method, and introduce R members derived from random branches of UCS; together, these

³This is the configuration that the pieces are given in, as a Shapely Polygon is defined by a set of coordinates.

N members form the next generation of the population to be used in iteration $i + 1$. At the end of iteration M , we output the fittest member of the population as our tentative solution.

5.3 Simulated Annealing

The Tangrams problem can also be viewed as finding a minimum-cost node by searching through some graph by transitioning from node to node. In particular, starting from a random initial tentative solution S , we repeat the following algorithm:

- Compute the difference polygon D between the target and the current solution.
- Consider the tentative solution S' formed by taking a random polygon P and placing it on D using the corner heuristic.
- With some probability $p(S, S', i)$ (where i is the number of iterations completed so far), and their costs, we move from S to S' .

The question then becomes what to use for the transition probability function p . The standard simulated annealing transition is given by

$$p(S, S', i) = \min \left(1, \exp \left(\frac{\mathcal{L}_T(S) - \mathcal{L}_T(S')}{\text{Temp}(i)} \right) \right)$$

where Temp is the **temperature function**, usually a decreasing function on i [3]. While setting, for example, $\text{Temp}(i) = 1/i$ performed decently, we found that a slightly different transition probability gave even better results:

$$p(S, S', i) = \min \left(1, \exp \left(\frac{\mathcal{L}_T(S) - \mathcal{L}_T(S')}{k(i - i_0)\text{Area}(P)} \right) \right)$$

where k is a tunable constant (which we have set to 0.01), i_0 is the iteration number of the last successful transition to a lower-cost state, and P is the polygon that was moved to get from S to S' . Intuitively, we add an area factor in the denominator to encourage moving large pieces around (without this factor, large pieces often got stuck because moving them resulted in large loss) and a factor of $i - i_0$ so that, the longer we get stuck in a local minimum, the higher the probability that we will try to jump out of it. This algorithm can be run for as long as necessary to converge to the zero-cost state.

6 Results & Discussion

6.1 Method Performance on T_{sq}

Compared to other tangram problems, T_{sq} is arguably one of the most difficult, as it provides nearly no edge information. As a result of this, exact methods such as stock backtracking and UCS could not feasibly solve the 7-piece problem. However, backtracking coupled with the angle heuristic was able to solve T_{sq} in approximately 4s by avoiding moves that would leave an angle unable to later be filled.








Alg.	Nelder-Mead	Powell	COBYLA	SLSQP	Genetic Alg.	Simul. Anneal.	BT + Angle
Soln.							
Cost	0.1646	0.1232	0.1244	0.0732	0.0846	0.0000	0.0000
Time	~40s	~40s	~6s	~14s	~80s	~20s	~4s

Figure 2: **Performance of Exact and Approximate Algorithms on T_{sq}**

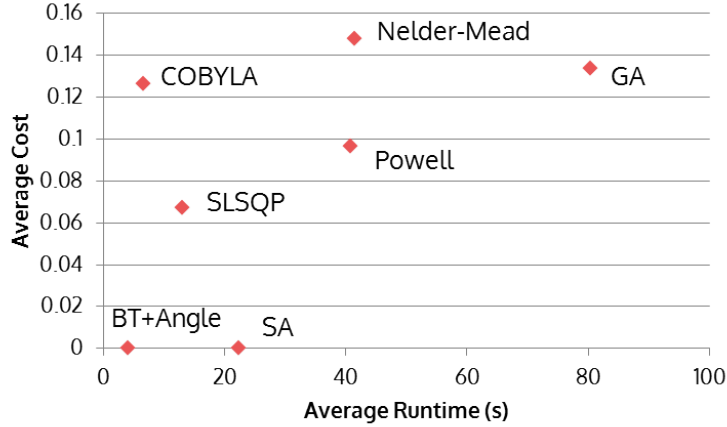


Figure 3: **Average Cost vs. Time for T_{sq} .** The average cost and time for 3 trials of T_{sq} were plotted for each method. Points closer to the origin show better performance.

When run on T_{sq} , the various loss minimization techniques finished within reasonable time (Figure 2), with Nelder-Mead and Powell running in approximately 40s, COBYLA in approximately 6s, and SLSQP in approximately 14s. Out of these four techniques, SLSQP consistently gave the lowest-cost solutions. Regardless, the solutions output by loss minimization are not aesthetically pleasing as they do not preserve the corner heuristic, and the final cost still remains high. The high cost is likely due to the abundance of local optima present when solving the tangrams problem. The genetic algorithm took longest out of all methods, and returned solutions comparable in loss to loss minimization techniques; this approach was able to escape local optima due to the introduction of random members, but it often oscillated between local optima. However, as the genetic algorithm implicitly preserves the corner heuristic through the breed operation, the results look visually better. The final approximate method, simulated annealing, was able to perfectly solve T_{sq} , doing so in a time of approximately 20s.

6.2 Method Performance on $T_r(k)$

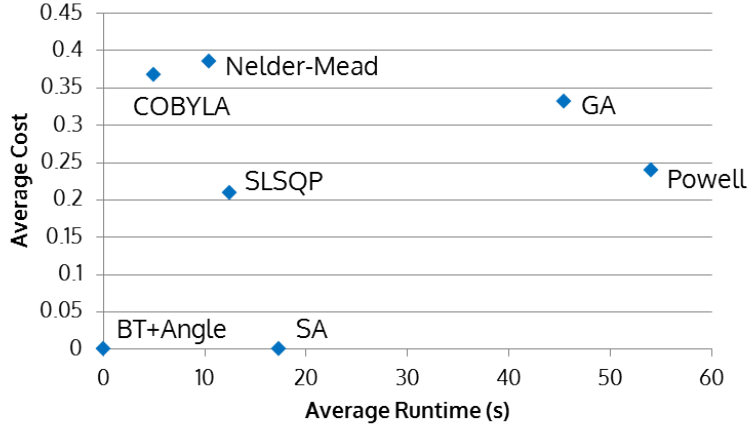


Figure 4: **Average Cost vs. Time for $T_r(6)$.** The average cost and time for 3 trials of $T_r(6)$ were plotted for each method. Points closer to the origin show better performance.

In addition to testing our methods on T_{sq} , we also tested performance on $T_r(k)$ for $k = 6$ to ensure that our algorithms remained general enough to solve highly irregular tangram problems. While the same general trends for runtime and performance were observed as in T_{sq} , Nelder-Mead finished quicker due to failure to converge. In addition, simulated annealing was still able to exactly solve $T_r(6)$, but the fastest method now was actually backtracking with the angle heuristic. This is because during random tangram generation, polygon angles are very likely to be unique. This property significantly reduces the search space for backtracking, as piece corners can now often be mapped to their corresponding target corners with relative certainty.

6.3 Challenges of Solving T_{sq}^2




Soln.			
Pieces	6	7	8
Time	3.1367s	30.8532s	417.2470s

Figure 5: **Performance of Simulated Annealing on $T_r(k)$.** SA solved instances of the randomized $T_r(k)$ (for $k = 6, 7, 8$), but each added piece grew the solving time approximately 10-fold.

As simulated annealing and backtracking with the angle heuristic were able to exactly solve T_{sq} , we also decided to examine the growth of runtime with respect to piece count to determine

if it was possible to solve T_{sq}^2 . Using $T_r(k)$ as a proxy, we tested $k = 6, 7, 8$, but unfortunately observed approximately a 10-fold increase in runtime of simulated annealing for every piece added (as shown in Figure 5). A similar trend was observed for backtracking with the angle heuristic, with a 2-fold increase in runtime for every additional four pieces.⁴ As T_{sq}^2 has a whopping 14 pieces, coupled with as little edge information as T_{sq} , our algorithms are likely still unable to solve T_{sq}^2 in their current state.

7 Future Work

As we have developed correct algorithms that can exactly solve T_{sq} , the next step is to optimize these existing approaches or develop new ones to solve T_{sq}^2 . For the exact methods, we would like to explore other placement heuristics in addition to the angle heuristic. Deutsch & Hayes [2] provide a myriad of heuristics in their work; although most of these only work on T_{sq} , some can potentially be generalized for to work for general tangrams. For the approximate methods, a possible extension is to implement parallel tempering, which simulates a system at a variety of temperatures.

8 Conclusion

In this project, we aimed to solve the little-studied problem of general tangrams. To this end, we developed an efficient representation of tangrams problems and devised exact and approximate techniques accordingly. Exact techniques using only the corner heuristic were unable to solve T_{sq} in a reasonable amount of time, but coupling backtracking search with the angle heuristic allowed this exact method to efficiently solve said problem. For approximate methods, the different loss minimization approaches and genetic algorithm fell into local optima traps; only simulated annealing was able to achieve the global optimum, although the runtime for solving T_{sq} was slower than that of backtracking with the angle heuristic by a factor of nearly 6. We have thus shown that both exact and approximate techniques, given sufficient sophistication, can exactly solve general tangrams problems.

Acknowledgements

We would like to thank Irving Hsu and Percy Liang for their continued guidance throughout this project and course.

References

- [1] K. Oflazer, “Solving tangram puzzles: A connectionist approach,” *International journal of intelligent systems*, vol. 8, no. 5, pp. 603–616, 1993.

⁴ $T_r(k)$ is not a perfect proxy, as the unique angles will make backtracking with the angle heuristic extremely efficient.

- [2] E. Deutsch and K. Hayes Jr, “A heuristic solution to the tangram puzzle,” *Machine Intelligence*, vol. 7, p. 205, 1972.
- [3] D. Bertsimas, J. Tsitsiklis, *et al.*, “Simulated annealing,” *Statistical science*, vol. 8, no. 1, pp. 10–15, 1993.