Project 2: the Cookie Cutter (Group 6)

Behrooz Badii (bb2122@columbia.edu)
Hanhua Feng (hanhua@cs.columbia.edu)
Edan Harel (edan@columbia.edu)

October 14, 2003

1 Introduction

In this project, groups were assigned to create players that could place as many cookies on a board, variable in the x-axis, while minimizing the amount of dough used in the direction of the x-axis. There are many solutions to this problem, with different solutions working better on specific cookies types. Group 6 used many algorithms and considered many others to use in creating an ultimate cookie cutter. The introduction of the importance of time before the project due date spurred the group to reconsider some approaches as well.

This report is organized as follows. In section 2 we discuss our ideas and methods we have investigated and implemented. In section 3 we present the tournament results. In section 4 we discuss the performance and weakness of our methods and programs. In appendix A, we shall diverse to a topic of resolutionality.

2 Methods

2.1 Dilating the shapes

Before doing any operations, we expand the polygon by dilating their edges outwards by a small value δ. The vertex then moves to the cross points of two dilated edges. For the vertices that have very sharp angles, the situation is tricky – we do not want the far-away crosspoint of two dilated edges, so we actually replace the old vertex with a pair of two vertices, each of which is on one of the dilated edges, and the distance between the old vertex and the new edge incident to the two new vertices is at least δ. The following figure illustrates the dilation operation (the red circle indicates the new vertices).

After dilation, we can implement either gap-aware or gap-unaware algorithms without difficulties.

2.2 Rectangular bounding box: the first approach

The first idea that the group implemented was a "dumb" player that would place cookies right next to each other. Using the concept of a bounding box, which, in this case, was a rectangle, the dumb player just placed a cookie; the next cookie's bounding box was placed below the previous cookie's bounding box, if possible. If not, it was placed next to the last cookie's box. All cookies are placed on rectangular grid points with the same orientation. We need to find an optimal angle to place the cookies so that a minimum bounding box can be obtained. Intuitively, a rectangle that has a minimum area should be better. However, since the number of cookies and the dimensions of the dough are finite, this solution is definitely not optimal – there

might be much space wasted at one side of the dough. The actual problem is to find an optimal angle of placement of the cookies, namely ϕ, such that a minimum objective function

$$w(\phi) \left\lceil \frac{n}{\lfloor \frac{1.0}{h(\phi)} \rfloor} \right\rceil$$

can be obtained, where n is the number of cookies and $h(\phi)$ and $w(\phi)$ are the height and width of the corresponding bounding box when the cookie is rotated by an angle of ϕ.

The running time of finding a sub-optimal angle is independent of the number of cookies. As a result, the overall running time of this algorithm is almost constant, except for the very small fragment of time used to prepare the actual positions of cookies. The whole algorithm is super-fast, and it also leads to the following fast algorithms.

This algorithm is implemented in our first player, the first version of "Dumb and Dumber". It actually showed up in our final player for the case that the number of cookies equals one, since other algorithms are not necessary for this case.

2.3 Improving the bounding box method

Clearly, the previous algorithm is not a good one – a neighboring cookie can certainly etch into the bounding box of another without any overlaps of cookies. The enhanced algorithm tries to push two vertically neighboring cookies together and finds the minimum distance, namely $d_y(\phi)$, of two cookies, then pushes a cookie to the left and finds a minimum distance, namely, $d_x(\phi)$, when its left-side neighbor and two cookies vertically adjacent to this neighbor are present. Then our task is to find the value of ϕ to minimize this objective function

$$d_x(\phi) \left\lceil \left\lceil \frac{n}{\lfloor \frac{1.0-h(\phi)}{d_y(\phi)} \rfloor + 1} \right\rceil - 1 \right\rceil + w(\phi)$$

This algorithm may lead to a bug. In the case that $d_y(\phi) < h(\phi)/2$ the cookie may overlap with some cookies of the previous column other than its left, upper-left, and lower-left neighbors. Since we do not want to make the algorithm more complicated, we fixed it by simply limiting $d_y(\phi)$ to be no less than $h(\phi)/2$, and lost optimality for some cases as a trade-off of getting an almost constant running time. The second "Dumb and Dumber" player was actually using this algorithm.

2.4 Two as a pair: the second approach

The bounding box algorithm is good for shapes similar to a rectangle, but performs badly for some other shapes, such as a triangle, in which case there is always at least one-half of the space wasted. If we can somehow bind two triangles to a parallelogram, it can be well solved by the bounding box algorithm. This idea leads to our second approach – the pairing algorithm, although the pairing algorithm is not limited to forming a parallelogram with two triangles.

The pairing algorithm first finds a way to bind two cookies together (each has its own orientation), such that the bounding box is minimum in the same sense as in the enhanced bounding box algorithm, then we can do exactly the same thing as in the previous enhanced "Dumb and Dumber" player. The only difference is now we use two paired cookies as a single object.

We also made one more improvement. Suppose we have n cookies and, by the pairing algorithm, m cookies are filled in the same column. If we found in the last column, there are not enough cookies to fill up a column, i.e., $k = n - m\lfloor n/m \rfloor \neq 0$, we would instead call the "Dumb and Dumber" player for the last k cookies, and check if it can get a better solution. If so, we shall combine two results, one for $m\lfloor n/m \rfloor$ cookies, one for k cookies.

The pairing algorithm is used in our player "Cookie Monster".

2.4.1 Subproblem: finding a best combination of two cookies

The only problem left in the pairing algorithm is to find the best combination of two cookies. The *placement space* of one cookie is 3-dimensional (which is constituted by the x,y-displacements and the angle of rotation). For two cookies, the *placement space* is the Cartesian product of two 3-dimensional *placement spaces* for each cookie, which is 6-dimensional. Since we are going to translate and/or rotate two cookies as a whole, so we can remove three dimensions from our analysis. Then the *relative placement space* of two cookies are 3-dimensional. We used two representations to coordinate this space:

1. by (d, ϕ_1, ϕ_2) where d is the distance between two cookies, and ϕ_1 and ϕ_2 are the corresponding rotation angles of two cookies, with respect to the line across their centers (we used centroids of vertices as their centers).

2. by (x_2, y_2, ϕ_2) where x_2, y_2, and ϕ_2 are the coordinates and the rotation angle of the second cookie in the first cookie's coordinate system.

Now this problem is converted to an optimization problem with three free variables. Intuitively, in the first representation, the bounding box is decreasing as d decreases if ϕ_1 and ϕ_2 are fixed. Similarly in the second representation, the bounding box is decreasing as x_2 decreases, if y_2 and ϕ_2 are fixed. Then, this problem is divided into a 1-dimensional optimization problem and a 2-dimensional optimization problem.

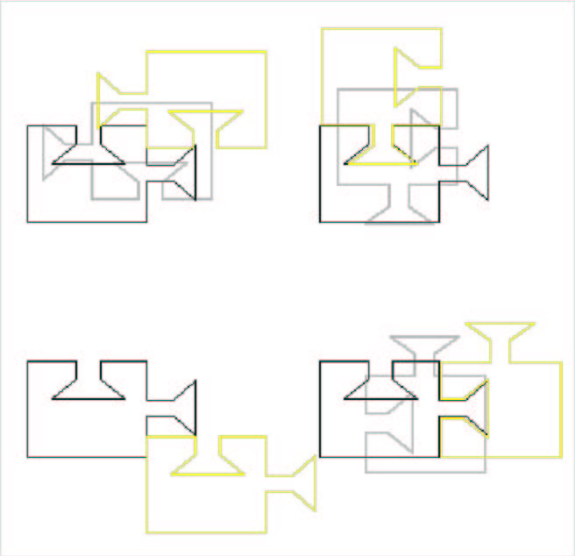
Let us use the first case as the example. If ϕ_1 and ϕ_2 are both given, the optimal combination happens if d is minimum. Therefore the optimal d, namely $d(\phi_1, \phi_2)$ is a function of ϕ_1 and ϕ_2. The objective function, which is the original function of all three variables, is now a function of ϕ_1 and ϕ_2. In our implementation, we just discretize ϕ_1 and ϕ_2, enumerate them, and then compute and compare the objective function. To improve the performance, it is also possible to use other optimization methods.

To compute $d(\phi_1, \phi_2)$, the previous mentioned algorithm of finding a minimum distance is used. Because of this algorithm, it is no longer an optimization problem – we do not need to discretize and enumerate d, or use some other 1-dimensional optimization methods (which has a problem of local maxima) to find an approximation – actually this algorithm can find an exact optimal solution, as explained in the appendix.

2.5 Solving the puzzle

We implemented two puzzle solving methods, a corner-puzzle solving method and a side-puzzle solving method. The corner-puzzle solving method is one that works with a pairing of the polygon shapes, and attempts to find a good pairing that takes the minimum area when conjoined. This is done by placing one of the corners of the first polygon on a corner of the second polygon. Then, the first polygon is rotated around that corner, continually checking that the positioning is valid and if so, find out its minimal bounding box. This is repeated for every unique pairing of corners, and using the minimal pairing, tessellating the bounding box, with some checking to try to fit individual shapes into gaps where pairs couldn't fit. This algorithm does a reasonable job of fitting many of the puzzle shapes and is reasonably quick, with $O(n^2)$ where n is the number of vertices on the polygon. After this is completed, the compressor function is used to try to minimize the gaps between the shapes further.

Also, the previous algorithm of finding a best combination can also actually solve the puzzle-like shapes. We used the second representation of the *relative placement space* for two cookies to solve the puzzle, because we guess that we can just use a small number of discretized values of ϕ_2 (e.g. some standard directions or the directions of all edges of two polygons) while using a very dense discretization of y_2. We actually find this works. In the final version we only used these rotation angles: 0, $\pi/2$, π and $3\pi/2$. This method, namely the side-puzzle solving method, can dealing with Group 1's shape the "Martini glass". The following figure shows the minimum distances with $\phi_2 = 0, \pi/2, \pi, 3\pi/2$. The black polygons are the position of the first cookie, and the gray polygons indicate the original place of the second. Yellow polygons indicate the place of the second cookie for different ϕ_2s so that a minimum distance can be obtained.



2.6 A greedy algorithm: the third approach

The third method is a brute-force greedy algorithm. This greedy algorithm would look for a local minimum between cookies. After bringing up the consideration of several initial placements (differentiated only by rotation around the centroid of the cookie), that idea was implemented into the greedy algorithm. No randomness other than the random rotation for initial placement is apparent in the greedy algorithm. Subsequent cookies are rotated in many different ways (currently 12) and fitted up against the previous cookie in many different directions (currently also 12) in order to minimize this evaluation function

$$x_b + 0.9 y_b \frac{n-j}{n},$$

where x_b and y_b are the x- and y-bounds of the subsequent cookie, j is the index of the subsequent cookie (one-based), and n is the total number of cookies. The evaluation function takes both x and y bounds into consideration, but the importance of y bound is diminishing while the index increases. The greedy algorithm has super-linear time, so it takes a lot of time to compute all the cookie placements using the greedy algorithm. Due to the importance of time, the greedy algorithm would only run by our final player when there are fewer than 15 cookies to be placed. This is apparent in the amount of time it takes for 13 cookies when compared to the time it takes to place 53 cookies.

The algorithm for the minimum distance between two cookies is the same as that in the pairing algorithm. The program pre-computes and stores the distances for all necessary combinations of angles and directions (so in the program there are currently 144 combinations), in order to speed up the process.

2.7 The compressor: the last enhancement

After a solution is found, we introduced a post-enhancement method. This algorithm came from the idea we mentioned in the class: if we want to make a bottle of stones more compact, we can shake the bottle and let the stones move. This idea seems to be related to the simulation of physics, but the actual implementation is very different from real physics. This is the pseudo code of this algorithm:

```
repeat for several times
    for each cookie in the solution
        for each angle between 90 and 270 degrees (w.r.t. x-axis)
            compute how far the cookie can move
            compute the evaluation function of this move
        find an optimal angle such that the evaluation function is optimal
        move towards this direction for a maximum distance
```

The angles are within 90 degrees or 270 degrees, inclusive, which means cookies can move toward any direction as long as its x-coordinate does not increase. Even moving vertically is helpful, since it may leave some space so other cookies can fit in. However, a move toward the left is preferred, so we have used this evaluation function,

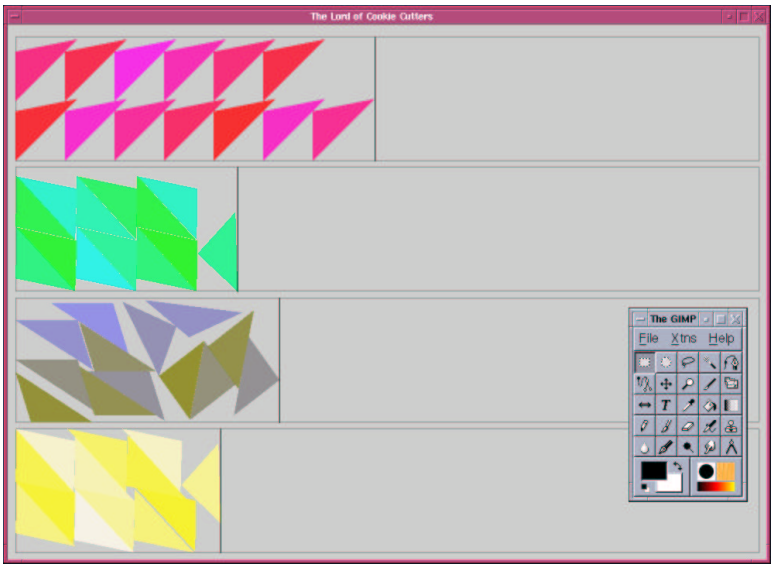
$$x^2(\alpha) + 0.01 * y^2(\alpha),$$

where $(x(\alpha), y(\alpha))$ is the relative coordinates of the farthest place the cookie can go along the direction determined by angle α.

2.8 Final combination of all available methods

By now, we have three algorithms: the enhanced bounding box algorithm ("Dumb and Dumber"), the cookie-pairing algorithm with puzzle solver enabled ("Cookie Monster"), and the greedy algorithm ("Greedy Dumber"). We also have one post-enhancement algorithm: the compressor. In the final version of our player, "The Lord of the Cookie Cutters", we try to run all algorithms and compress their results, then find the best solution for this problem.

The following figure shows a comparison of our algorithms. The first solution was made by the enhanced bounding box algorithm. The second solution was made by the pairing algorithm. The third algorithm was made by the greedy algorithm. The last one is the solution that the compressor made after the pairing algorithm.



3 Results

This monster of a player, dubbed the Lord of the Cookie Cutters, combines several strategies: the greedy algorithm, the enhanced bounding box algorithm, and the puzzle-solving algorithm. Each algorithm is run, their respective solutions undergo compression, and whichever algorithm gives the best result (the smallest amount of length used), is chosen. Thus, with many choices, we have many possibilities of coming out on top. Due to the constraints on the greedy algorithm, the time it takes to place higher numbers of cookies is quite low as compared to other brute force or A* players. This is definitely apparent in the results of the tournament.

3.1 Tournament performance

In a few words, we did very well in the tournament. We consistently placed in the top third, and we still placed first very frequently. The lowest ranking we had in any for any of the cookies for any of the number of

cookies was sixth, and that was only once. Due to the max cookie cap that was put on the greedy algorithm, our time was decreased without giving up the player's efficiency too much. The following is a set of tables describing our performance numerically:

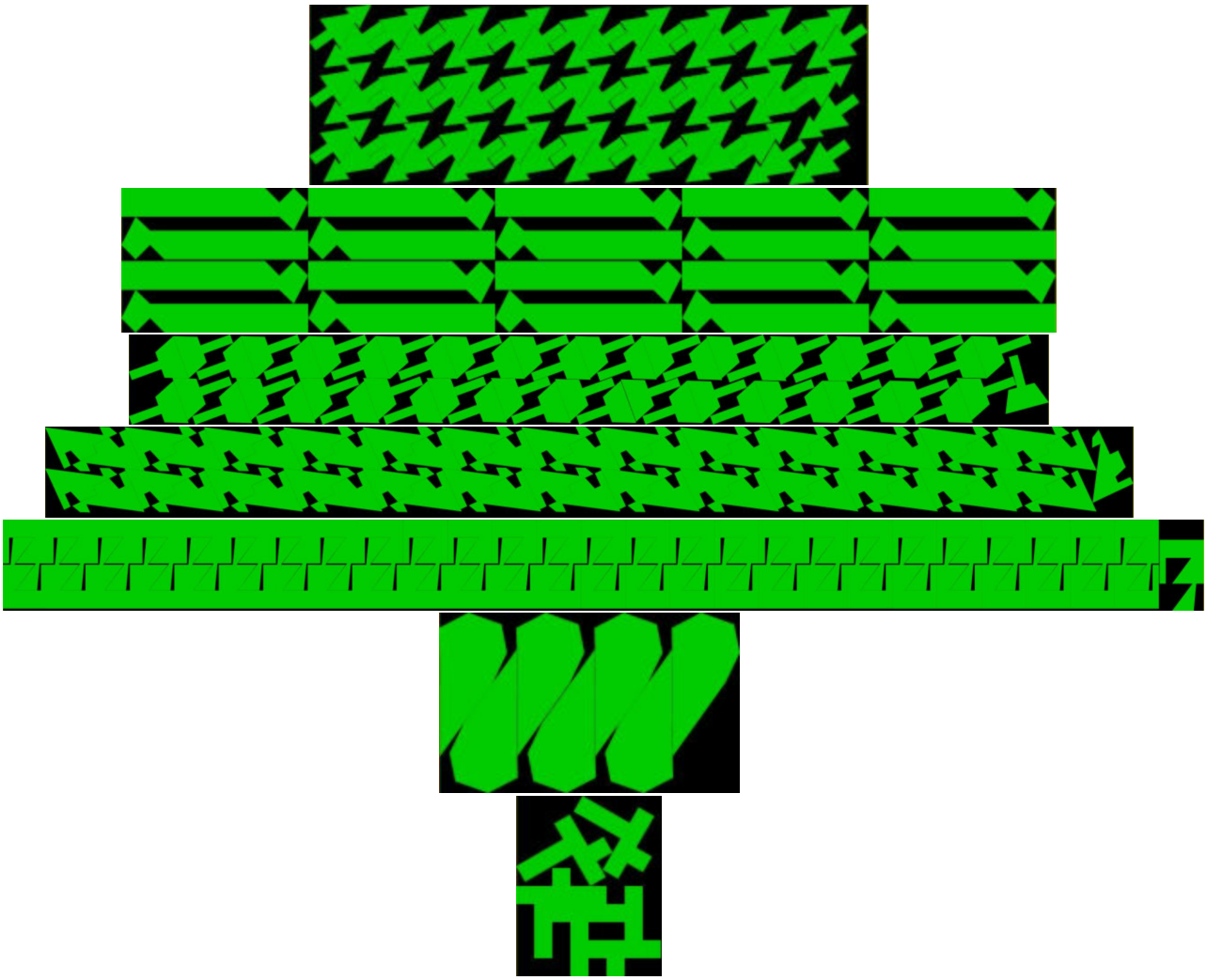
3.2 Tournament ranking results

Number of times 1st	47
Number of times 2nd	32
Number of times 3rd	12
Number of times 4th	5
Number of times 5th	2
Number of times 6th	1
Number of times 7th	0

Number of Cookies	Average Ranking	Average Time Consumption	Standard Deviation of Time Consumption
1	2.9	0.026777778	0.035860068
2	2.444444444	1.948222222	0.573533081
3	1.666666667	2.863888889	1.250830069
4	1.555555556	2.237111111	0.632905096
5	2	4.589777778	1.377301417
6	1.444444444	3.532222222	1.334250893
7	1.333333333	5.886666667	2.095636598
12	1.555555556	5.194222222	1.53638177
13	1.777777778	8.514444444	2.585946979
20	1.555555556	8.663555556	2.98880633
53	1.777777778	45.61744444	14.98612789

When there is only one cookie to place, one can see that our avg. ranking was not absolutely spectacular. However, one must note that, in general, there was only a .01 or less difference for the right boundary of the top five cookie placers. By cutting down on the number of comparisons and taking out the greedy algorithm for 15 or more cookies, the time it took to create our solution was fantastic. When there were 20 or fewer cookies to be placed, our player never took more than 12.356 seconds. Compared to the other algorithms, we were at least average (in some cases we were the fastest algorithm) in terms of the amount of time it took to create a solution. But this average amount of time produced very high rankings in the tournaments. In general, the A* or greedy algorithms implemented by other groups, such as Group 1, took around 1.5 to 4 times as much time as our cookie cutter (in extreme cases, Group 4 would take around 1000 seconds where our group would take only around 60 seconds), whereas the other cookie cutters took less time (when the number of cookies was at 53, the Group 3 player would spend as little as a ninth of what our player spent in time, but their performance was much lower than ours in that situation). One number that stands out is the standard deviation of 53 cookies. This is due to our algorithms dealing mostly with corners or sides. Cookies, obviously, were allowed to have a defined number of vertices from 3 to 12. The different kinds of cookies created a different number of vertices, and our algorithms' vertex or side related strategies become apparent with this standard deviation. A variable number of vertices means a variable number of comparisons between cookies to be placed and placed cookies. Hence, the standard deviation is much higher at 53 cookies.

The following are pictures of some of the winning placements of our player. The first six are "Cookie Monster" and "Dumb and Dumber" placements, and the last is obviously a greedy algorithm placement.



4 Discussion

4.1 Puzzle-solving methods versus greedy algorithms

Both the corner puzzle solving method and the side puzzle solving method are roughly comparable algorithms, both tending to produce similar pairings of shapes - given that, only the one that was more finely tuned was used in the final submission to try to minimize the time.

The puzzle solving methods, not surprisingly, tends to do well with puzzle shapes, while the Dumb and Dumber methods often tends to produce a good solution with random shapes. The greedy algorithm rarely produced the best solution, only about 10% of the time, usually with an odd number of cookies or with random shapes.

As seen with some of the other players in the tournament, a greedy, solution using random sampling – sort of I Ching for problem solving – can produce very good results, but the time of execution can quickly become a problem as the number of cookies increases, unless the binning system by Group 2 is used, or the algorithm looks at fewer possibilities, and even then time will continue to be a significant problem.

4.2 The running time

The performance of our combined algorithm is superior, as it was in the tournament. Also, the running time is good in contrast to other competitive adversaries, since the running time is actually one of our objectives.

Both "Dumb and dumber" and "Cookie Monster" run in almost constant time, and the resolution of "Greedy Dumber" has been cut off and it is even disabled when the number of cookies is greater than 15. Another time-consuming algorithm present in our final player is the compressor, which also takes super-linear time. Despite of this, our player still outperformed other competitive players in the measure of the time consumed whenever we won or lost.

It's not necessary to discuss the good performance any more – the result of the tournament proved that. In the following discussion, we shall talk about the problems either discovered during our development or concluded from the tournament.

4.3 For the one-cookie case

The tournament results show that, for the one-cookie case, we are always led by other players by a small difference of scores. We guess the reasons are that

- We used a large gap. The gaps are at least 2×10^{-5} on each side. Originally we think the value of the gap is of no importance, we rather make it large to make our problem more robust to some singular shapes.
- We did not write a program to find the exact minimum diameter. Although we know how to do it, but we did not actually implement it due to time limit.

4.4 Long and thin shapes

As mentioned previously we found a bug in the enhanced bounding box algorithm. We fixed this bug with a compromise to simplicity and efficiency. For this reason the enhanced bounding box algorithm and the pairing algorithm cannot generate an oblique arrangement for a long and thin shapes (e.g. the Group 7's tournament shape).

4.5 Weak greedy algorithm

Our greedy algorithm is on average weaker than those of other groups, although we observed it sometimes won for us. In most cases our greedy algorithm loses to our own pairing algorithm. We guess the possible reasons are

- Although we tried to pre-compute and stored possible actions to save time, because our greedy algorithm used a data structure supporting backtracking, which is kind of slow for a pure greedy algorithm.
- The evaluation function need to be improved.
- A well-structured system frequently introduces slow-down.

We did not spend much time to find a better evaluation function, which is very important to the performance of a greedy algorithm. Originally we were going to implement a one-level backtracking algorithm, or a finite-backtracking branch-pruning $A*$ algorithm. If these are done, a bad evaluation function might easily beat a very powerful greedy algorithm. Later we tended to use faster algorithms and greedy algorithm seems already too slow, and these search algorithms are then put aside.

4.6 Non-rotatable compressor

In many cases the compressor can rotate a cookie to get better performance. However we have not programmed this because of the time limit, and also because there are resolution problems in dealing with rotations.

5 Conclusion

We were very happy with our tournament results. Hours of slaving away at code really paid off. The many strategies that were implemented gave us great flexibility; therefore, we obtained near optimal solutions for almost every cookie. The competition between the groups gave reason to go above and beyond regular solutions and attack and solve stronger, more complex solutions. The sheer amount of coding was difficult to overcome, but now that the framework is set, subsequent classes can utilize the foundations this class has set for the Cookie Cutter problem.

6 Acknowledgments

We'd like to acknowledge the class discussions for bringing up the concepts of a brute-force greedy algorithm and a bounding box idea. We'd also like to acknowledge the class for considering puzzle shapes, leading to the idea of a puzzle solving algorithm.

A A journey to break the curse of resolutionality

A.1 The curse of resolutionality

When groups began to study this project, a common problem is raised – How do we know the best place for the second cookie if a cookie is already placed? More exactly, the problem is, what is the minimum distance so that the two cookie can be placed without overlapping?

An intuitive algorithm is to check all possible distances and to find the minimum. However, since the space is continuous but computers are discrete, one can only find its approximation. If one wants to halve the approximation error, one has to double the sample resolution. This situation becomes worse if there are more dimensions – to place a single cookie, we have to specify three free variables, which constitute a 3-dimensional *placement space*. In the case of best-fit for puzzle-like cookies, if the tolerable gap between cookies is very much possible below the sample resolution, this algorithm cannot even guarantee the accuracy of approximation. For example, suppose the maximum gap is 10^{-4} for the best-fit of a pair of unit-sized cookies, we have to sample at least 10^4 points on one dimension, which means the total number of sampled points will be at the order of 10^{12} in order to ensure that the best-fit can be discovered. This is almost impossible for this project!

This computational complexity is induced neither by the number of polygons nor by the number of vertices of each polygon, but the resolution of discretization in the algorithm. We call it the curse of resolutionality.

In many optimization problems, this can be solved by using some *global information* of the problems, or in other words, some additional properties of the evaluation function, such as continuity, monotony or quadraticity. For non-puzzle shapes, this works. Let one cookie move to the other, initially in large stride, until the two intersect. Then we halve the stride and move forward and backward depending on whether they intersect or not, then we can make the error of approximation below some ϵ in $O(\log(1/\epsilon))$ turns. Unfortunately, this only works in 1-dimensional case, and it would almost surely fail to find a puzzle solution. This algorithm was used in our early program to examine the minimum distance between two cookies.

A.2 Solving the puzzle without resolutionality

During the early stage of this project, we were trying to solve the puzzles without sampling. We assume there are only two cookies, and they can only be translated but cannot be rotated. We consider the *placement space* of the second polygon (since no rotation is allowed, the *placement space* of the second polygon contains only two dimensions: x displacements and y displacements. We let the *placement set* of the second polygon with respect to the first polygon stand for the set of points on the *placement space* of the second polygon such that two polygons do not intersect. Now what is the *placement set* of the second polygon? Instead of two polygons, we first consider two line segments. Clearly, the *placement set* of the second line segment with respect to the first line segment is the complement of a parallelogram on the *placement space*. Then

returning to the polygons, the *placement set* of the second polygon is the intersections of the *placement sets* of all pairs of lines segments, each of which consists of one edge in the first polygon and one in the second.

How can we compute the *placement set*? We observed that a parallelogram is a polygon. Because the unions, intersections, differences, and complements of polygons are still surrounded by polygons. Some polygons might be in other polygons as holes. If we can get the polygon representation of the *placement set* of the second polygon, we can easily find a minimum distance on this 2-dimensional space.

Initially, we were trying to represent polygon sets, and to implement these set operations. However, the problem seems to be too hard – which was referred as "first non-trivial problem in computer graphics". In computer graphics, there are some packages, but some of them use integer coordinates, and probably use a method of horizontal scanning. (Other methods are also available, such as triangulation algorithms.)

After analysis of the projects, we think only the set difference is needed for this project. Then we finally fully implemented the difference operations of polygon sets, and found it works for this project – it is so exciting to see the second polygon is placed at the right place we wanted.

The discouraging news is that the implementation is not stable, and we do not know why. The possible reasons are:

- program bugs. The program is too long to be well tested.
- computational errors. We were trying to make the program be robust all the time, but since too many polygons are cutting at the same line, with inexact math computation, we can hardly avoid problems.
- special cases. There are many singular cases: all vertices of some polygons may be in the same line, there are very small angles or line segments, and etc.

A.3 Returning to the resolution: a compromise

Having realized the 2-d algorithm is too huge for this quarter-semester project, we have to return to use the methods of "resolution". However, we are glad to find that it is fairly easy to implement and test a 1-d version, while using the methods of "resolution" for other dimensions. The 1-d algorithm is given as follows.

Suppose we have two polygons, namely P_A and P_B, both on its original places. The centers of the polygons can move along a line, namely L, and we want to find the minimum distance of two polygons. Two polygons are not necessarily the same. Then

- for each vertex, namely V, of polygon B, draw a line parallel to line L, namely L', and V is on L'. Then do the following:
  - Find all intersection points of line L' against all edges of polygon A.
  - Compute coordinates of these points with the coordinate system of line L', on which vertex V is the origin.
  - Sort these intersection points. Then we can get a set of sorted intervals containing points on line L which are at the outside of polygon A.
- Find the intersection set, namely S_1, for all previous sets of sorted intervals. Computing the intersection set of two sets of sorted intervals can be done in linear time, very much like the merge algorithm of sorted arrays.
- Swap polygon A and polygon B, do all the previous steps again to get another intersection set, namely S_2.
- Find the intersection set, namely S, of S_1 and negated S_2.
- Now, S contains all possible distances. Check all bounds of intervals in set S, and find the minimum distance.

This algorithm will get the exact minimum distances between two cookies, if one is moving toward the other along a fixed line. It can even solve the puzzle problem, if there actually is a solution on the line. This is the one of the most heavily used algorithms in our final player, and also one of the most interesting algorithms in our whole project.