The Discrete Voronoi Game

Fritz Reese[1]
George Mason University
M.S. Computer Science '18
fritzoreese at gmail dot com

Abstract— **The discrete Voronoi game is an abstraction of the *competitive facility location* problem, wherein two entities compete to place facilities among a region of customers with the goal of maximizing service area. In this paper, we offer a simple implementation of the n-th round discrete Voronoi game based on the L$_1$ `MaxCov` algorithm.**

I. INTRODUCTION

In the competitive facility location problem [4], [5], [3], two competing entities place facilities alternately in some demand region described by a set of discrete locations of customers (or "users"), denoted by $\mathcal{U}$. There are many variants of the problem involving different demand regions and rules for how many facilities may be placed and when. The primary variant we consider is the n-th round Voronoi game in $\mathbb{R}^2$, wherein both players have already placed n facilities in $\mathbb{R}^2$, and they each have a chance to place one more. This problem is interesting because it generalizes to the entire game, where each round is the i-th round game from 1 to n.

Regardless of the variation of the game, The ultimate goal of each competitor (player) is to end the game with more customers closer to its facilities than to its competitor's facilities. The region within which users are closer to one competitor's facilities than the other is known as the competitor's *service area*. It is easy to see that the when the player facilities are considered as sites in a Voronoi diagram, each player's service area is the union of regions covered by the Voronoi cells containing the player's sites. Such a Voronoi diagram is known as bichromatic [2], where each player's sites (facilities) and their corresponding cells are granted a different color. In a continuous version of the game, the player with maximal service area is the victor.

However, in the discrete game, each player is scored by a metric P_k known as payoff. Let the set of users u_k which lie in a Voronoi cell containing a facility belonging to player k be known as $\mathcal{U}_k$. The payoff P_k for player k is naturally the number of users served by the player divided by the total number of users, as described in equation 1.

$$P_k = \frac{|\mathcal{U}_k|}{|\mathcal{U}|} \tag{1}$$

Each player's goal in the n-th round discrete Voronoi game is to maximize their payoff after playing the $n+1$-th round of the game given the existing user points $\mathcal{U}$ and facility locations $\mathcal{F}$.

In section II, I discuss the optimal solutions of each player according to *The Discrete Voronoi Game in* $\mathbb{R}^2$ by Banik et al. [1]. In section III I discuss the algorithm for computing the optimal player two placement as described by Cabello et al. [2] and Imai & Asaon [6]. In section IV I describe my project implementation and how it differs from the algorithms in section III. In section V I discuss how I might continue with my work in the short- and long-term.

II. OPTIMAL STRATEGIES

A. Player two solution

Banik et al. [1] describes the optimal strategies for both players in the n-th round discrete Voronoi game to achieve the greatest payoff. According to Banik, player two's optimal facility location given n facilities already placed by both players as the easier of the two choices. In the L$_2$ norm (using Euclidean distance) simply draw a circle centered on each user in $\mathcal{U}_1$ with its boundary touching its nearest facility. The cell formed by the greatest number of intersecting circles is the ideal placement for a new facility for player two. Since each of the circles touch the site of a Voronoi diagram, we know the circles are all empty of other sites (facilities) by definition. Therefore, inside the bounds of every circle is a region wherein player two's facility would be placed closer to that circle (a user) than any other site. This region is known as the cell of maximal depth, or equivalently the cell with maximal coverage.

Instead of using circles and L$_2$ distance, an approximation of the solution can be found in more favorable runtime using the L$_1$ and L$_\infty$ norms (Manhattan distance). Banik et al. [1] cites Cabello et al. [2] for solving the maximal cell coverage problem for the L$_1$ and L$_2$ norms. Indeed Cabello describes an algorithm `MaxCov` for computing the exact solution using L$_2$ in O(n^2), and in O($n \log n$) using L$_1$, as well as faster methods for approximating the L$_2$ solution. These algorithms will be discussed in more detail in section III.

B. Player one solution

The optimal solution for player one is less intuitive. Player one's general strategy is to place its facilities such that the maximal payoff of player two is minimized. A simple example of this is when no facilities have yet been placed; then optimal placement for the first facility lies in the center region of $\mathcal{U}$ - the so-called "Tukey half-space median" [7].

[1] As part of CS633 – Computational Geometry – Fall 2017 with prof. Jyh-Ming Lien.

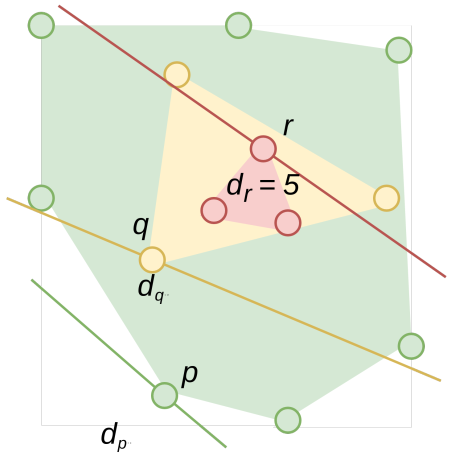


Fig. 1. Tukey depth.

Let $\mathcal{H}_u$ be the set of half-planes through point $u \in \mathcal{U}$. Let the coverage of $h \in \mathcal{H}_u$ be the number of points $p \in \mathcal{U}$ on the closed side of h, denoted by $p \in \mathcal{C}_h$. The Tukey depth d_u, or just depth, of u is thus that of the half-plane $h \in \mathcal{H}_u$ with minimum coverage, as described in equation 2:

$$d_u = \min_{h \in \mathcal{H}_u} |\{p \in \mathcal{C}_h\}| \tag{2}$$

An example is also shown in Figure 1. You can see regions of equal depth form convex shapes within which each point has equal depth. The idea is that the points u with the greatest depth are "embedded deeply" in $\mathcal{U}$, such that they are nearer to more other points than those with a lower depth. At least at a high level, it is intuitive that this concept could be extended to find the goal region for player one, which should be closest to the most number of points, given a set of existing facilities.

The actual proof of the player one strategy presented by Banik is fairly dense. In the end, suffice it to summarize that, for $|\mathcal{U}| = n$ using L_1 or L_∞ distance, player one's optimal strategy is to compute an arrangement of $O(n^2)$ rectangles in $O(n^2)$ time to find $O(n^4)$ depth cells, each of whose depth can be calculated in $O(n \log n)$ time. The result is a fairly hefty $O(n^5 \log n)$ algorithm for computing the player one solution. On the bright side, the runtime is still polynomial. It has been shown that some decisions involving variations of the Voronoi game on graphs are NP-hard [11], so a polynomial-time algorithm (even of an unfavorable order) is appreciated.

III. L_1 `MaxCov` FOR THE PLAYER TWO SOLUTION

Due to the complexity of the algorithm for the player one solution and the time constraints of the project, I decided to investigate the player two solution first. Recall from section II-A that this involves computing the cell of maximal depth using the `MaxCov` algorithm described by Cabello et al. [2].

First, let's talk about the characteristics of the input arrangement. The game is played by players consecutively placing sites. In our case we randomly generate customer points. For every customer point we draw a rectangle with its border on its nearest facility, discarding those which are already within player two's service area. (In fact, all the rectangles are squares, since we simply compute the Manhattan distance from the customer to its nearest facility and draw the square with that radius.) To use L_∞ distance, simply rotate the axes by 45 degrees. Either way, the rectangles can be processed axis-up so that the bottoms/tops of each rectangle are parallel to the line $y = 0$.

It turns out that Cabello does not describe the solution in very much detail. Instead most of the paper is dedicated to gaining speedup by using faster algorithms to obtain approximate solutions for the L_2 `MaxCov`. This is useful for us because its time complexity is a bottleneck in the L_2 Voronoi game. However, for simplicity and the sake of time I chose to study the L_1 `MaxCov` problem. Fortunately, Cabello references a paper by Imai and Asano [6] which describes the algorithm precisely. (An interesting side note: Imai points out that aside from competitive facility location, the problem of finding a maximum clique in an intersection graph also has applications in integrated circuit layout techniques, image processing, and numerical analysis [9], [6].)

Imai breaks the `MaxCov` problem into two parts: the first is the *connected connected components* problem, and the second is the *maximum clique* problem. These are discussed below in sections III-A and III-B.

A. *Connected Components*

The goal in the connected components problem is to compute a structure called an *intersection graph*. To understand this, it is useful to consider an example. Figure 2 shows the example input adapted from Imai. There are eight input rectangles labeled R_1 through R_8. Together these comprise the input set $\mathcal{R}$. The output should be a graph with one vertex per input rectangle, where the vertexes are connected by an edge if and only if the two corresponding rectangles intersect. Once this data structure is acquired, the clique of maximal size represents the "depth cell" with maximal depth - the region representing an intersection of the most number of rectangles. Recall from section II-A that the region within this maximal depth cell represents the optimal solution area for Player 2 in the n-th round discrete Voronoi game. We will expand on how to compute the max-depth clique in Section III-B.

After the n-th round of the discrete Voronoi game, let $\mathcal{R}$ be set of rectangles R_u centered on points $u \in \mathcal{U}$ in general position. The algorithm for computing the intersection graph $\mathcal{G}_\mathcal{R}$ from a set of axis-up input rectangles described by Imai follows the template of a classical line sweep algorithm. The sweep line at step k is a line parallel to $y = 0$ which coincides with a horizontal edge of the input rectangle $\mathcal{R}_k$. Let $\mathcal{Y}$ be the partitioning of the space induced by such lines coinciding with *bottom* edges of rectangles in $\mathcal{R}$. We visit the regions $Y_k \in \mathcal{Y}$ sorted in ascending y coordinate. Within each region Y_k, we define V_k as the set of rectangles which have vertical edges that intersect Y_k.

As we visit Y_k via the bottom edge of a new rectangle R_i, we also maintain a balanced search tree sorted by x

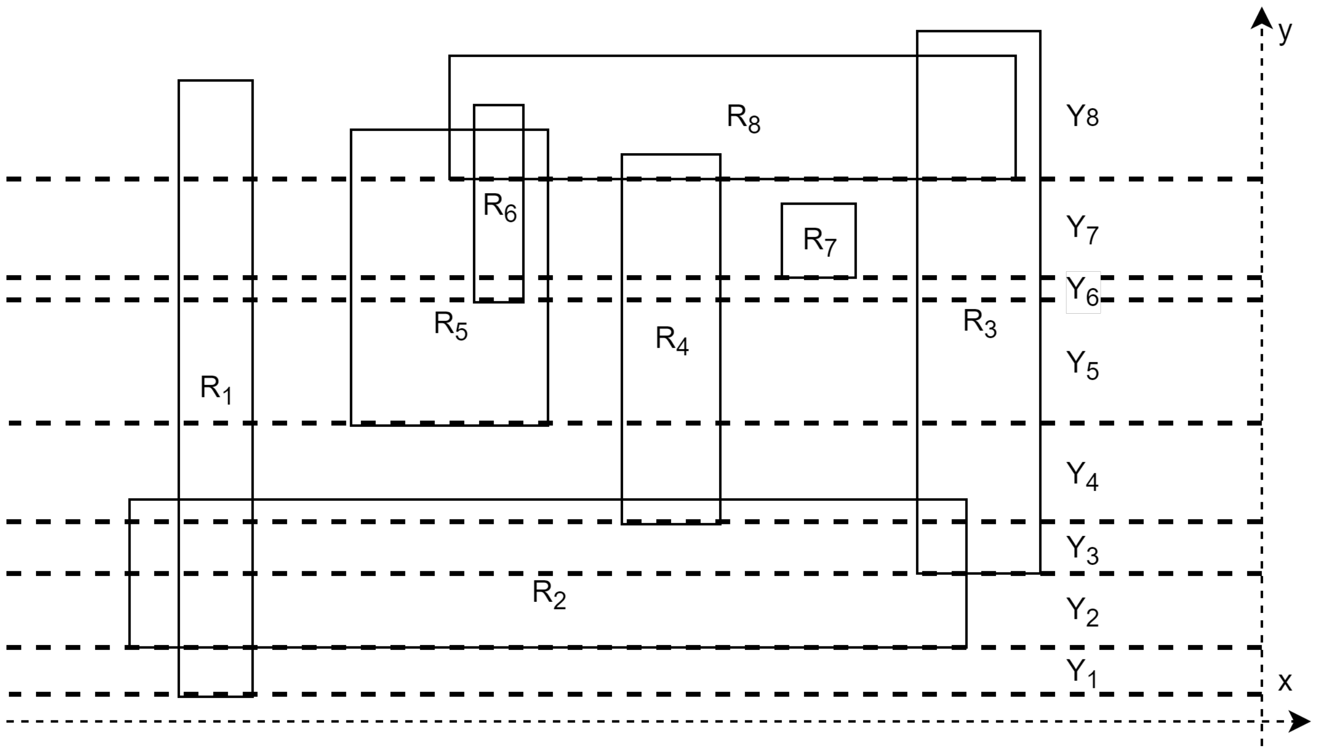


Fig. 2. Connected cells example.

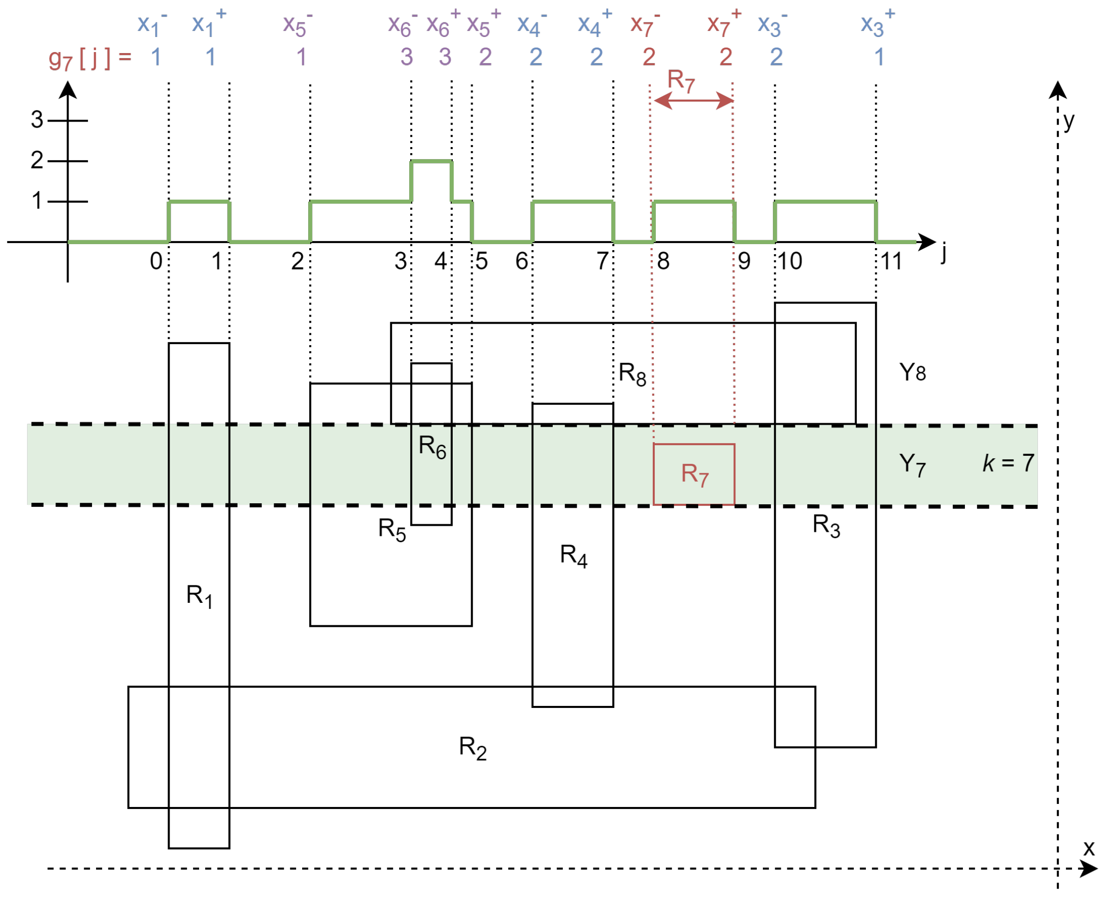
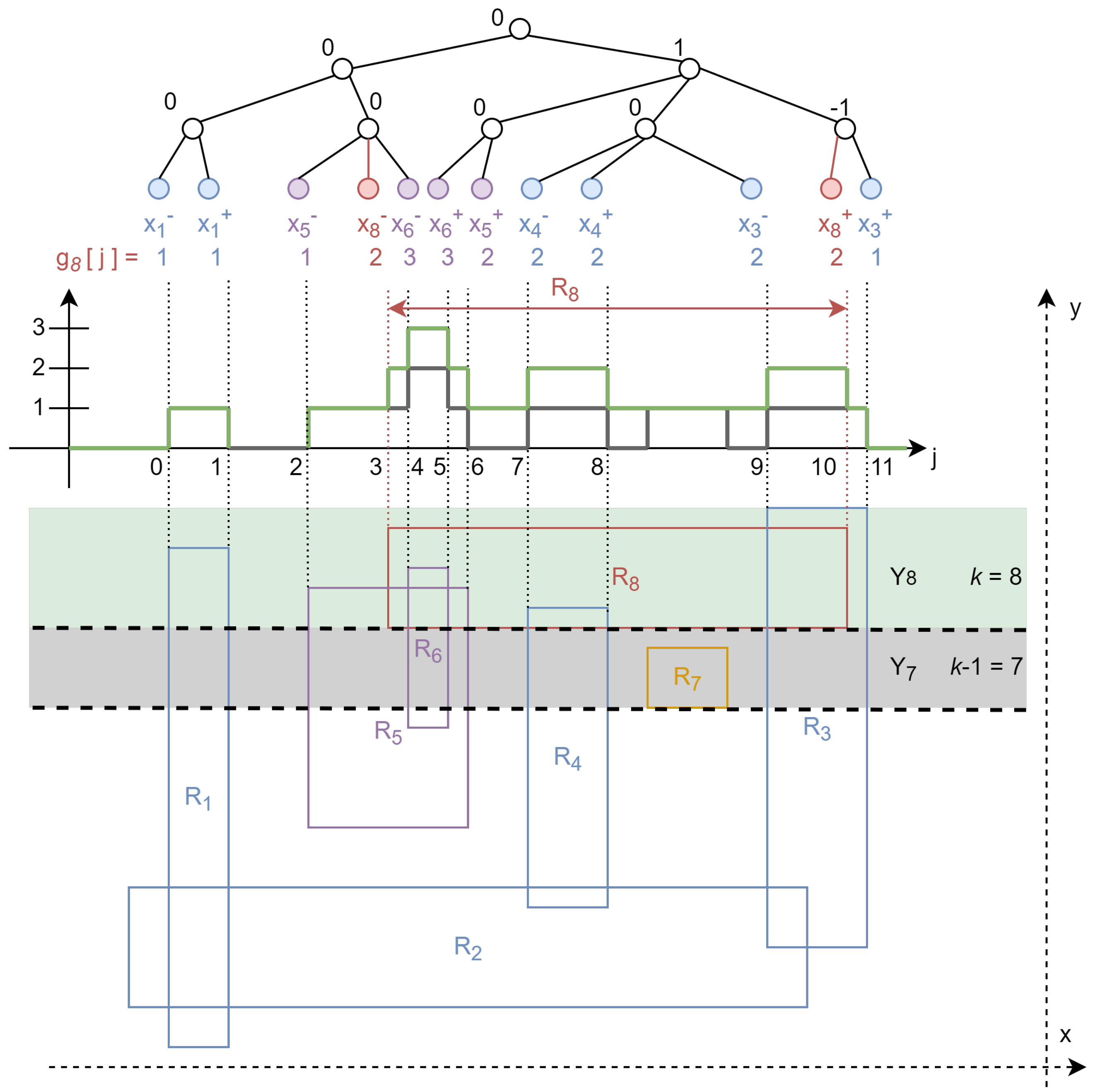


Fig. 3. Example where $k = 7$, showing $g_7[j]$.



0
0
1
0
0
0
0
-1
x1- x1+ x5- x8- x6- x6+ x5+ x4- x4+ x3- x8+ x3+
g8[j] = 1 1 1 2 3 3 2 2 2 2 2 1
R8
3
2
1
0 1 2 3 4 5 6 7 8 9 10 11
j
y
R8
Y8 k = 8
R6
R7
Y7 k-1 = 7
R5
R3
R4
R1
R2
x

Fig. 4. Comparing $k = 8$ to $k = 7$, showing $\mathcal{T}(V_8)$ and $g_8[j]$.

coordinate whose leaves are the vertical edges of rectangles in V_k, denoted by $\mathcal{T}(V_k)$. This allows us to quickly find edges of other rectangles which are covered by the newest rectangle R_i, as we will see later. Additionally, we maintain corresponding to the leaves of $\mathcal{T}(V_k)$ an array denoted g_k, where $g_k[j]$ is the current depth of rectangle R_j for $j \in V_k$.

The invariant of the sweep is that we already have an accurate representation of the intersection graph $\mathcal{G}_\mathcal{R}$ for all rectangles formed by the half-plane below the sweep line. We shall also maintain another balanced search tree known as L_k which represents at each step G_k, a partial solution for $\mathcal{G}_\mathcal{R}$. After the algorithm completes, $G_{|\mathcal{R}|} = \mathcal{G}_\mathcal{R}$. L_k will be described in more detail later.

Figure 3 shows the example from Figure 2 when $k = 7$. In that step, we are adding the new rectangle R_7, whose vertical edges are to be placed at indexes 8 and 9 in V_k. Shown at the top of the figure are the leaves of $\mathcal{T}(V_7)$, where x_i^- and x_i^+ refer to the left and right edges respectively of R_i. The depth values maintained in g_7 are also shown for each vertical edge $j \in V_7$. In this case, $V_7 = \{1, 5, 6, 4, 3, 7\}$, since those are the rectangles which intersect Y_7.

After we update the data structures appropriately for $k =$

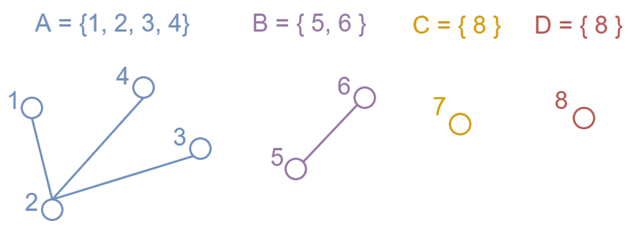


Fig. 5. Partial intersection graph G_8 when $k = 8$.

7, the next event we reach is the top of rectangle R_7. Whenever we reach the top of rectangle R_k, the rectangle's vertical edges are removed from the search tree in $\mathcal{T}(V_{k+1})$. Figure 4 shows the transition from $k = 7$ to $k = 8$ in much more detail. First, $\mathcal{T}(V_8)$ has been expanded at the top. You can see that the edges $x_7^{\pm}$ from R_7 have been removed as leaves since the rectangle has been passed and could no longer possibly intersect Y_8. You can see also that the depths $g_8[j]$ are incremented throughout the interval that R_8 spans. Remembering the depths through g_k is critical for ultimately determining the depth of the maximal cell. It is easy to see at a glance that $R_5 \cap R_6 \cap R_8$ is actually the solution, and you can see that the maximal depth of 3 in g_8 hints at this.

To ultimately solve the connected components problem, we also need to keep track of the *closure* of each rectangle; that is, the union of rectangles which intersect each other in a single chain, called a *component*. Figure 5 shows the partial intersection graph G_8 which contains connected sub-graphs of vertexes v_k that share an edge corresponding to where rectangles R_k intersect. For example, vertex 2 is connected to vertexes 1, 3, and 4 because R_2 intersects R_1, R_3 and R_4. In the figure, the closure for R_8 has not yet been computed. You can see the connected components A, B, C and D are drawn with matching colors in Figure 4 for clarity.

The search tree L_k is not shown explicitly for this example, but is related to both G_k and $\mathcal{T}(V_k)$. It is a balanced search tree whose leaves represent connected components, in the same order as the vertical edges from $\mathcal{T}(V_k)$. The difference here is that there is one leaf in L_k for each run of adjacent edges belonging to the same component. In the example from Figure 4, the leaves of L_k would be $\langle A, B, D, B, A, D, A \rangle$. When $k = 8$ as the edges of R_8 are inserted into L_k, component D must be merged with every component which lies between x_8^- and x_8^+. In the example above, this is every component, so that the final intersection graph after R_8 passes is as shown in Figure 6.

The key to the algorithm lies in the calculations of L_k and g_k. A naiive approach results in $O\left(|\mathcal{R}|^2\right)$ runtime, as any given Y_k may contain $O(|\mathcal{R}|)$ other rectangles for which g_k and L_k must be calculated. In order to achieve optimal log-linear runtime, g_k is actually never calculated directly. Instead, the tree $\mathcal{T}(V_k)$ is carefully maintained upon the insertion and removal of each rectangle's edges $x_k^{\pm}$. The value $g_k[j^{\pm}]$ is equal to the sum of the values of all nodes in $\mathcal{T}(V_k)$ along the path from node $x_j^{\pm}$ to the root. As nodes are inserted (removed), internal nodes which are *direct right (left) children* of every node along the path to the new (old) node have their values incremented (decremented). This results in only $O(\log |V_k|)$ operations, rather than $O(|V_k|)$, to update $g_k[j] \, \forall j \in V_k$.

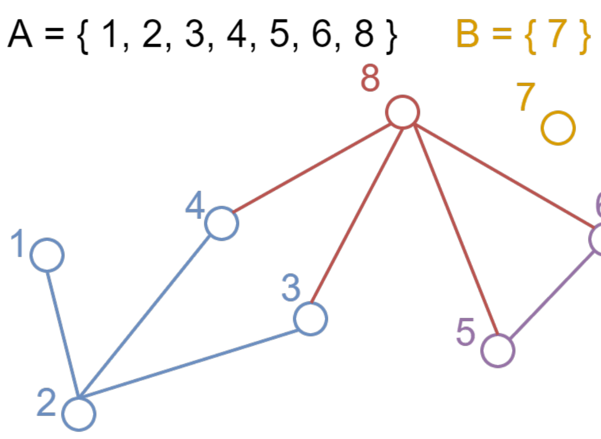


Fig. 6. Final intersection graph $\mathcal{G}_{\mathcal{R}}$.

We also maintain the balanced search tree L_k which represents G_k in log-linear time. As described above, whenever the edges of R_k are inserted/removed into L_k, the component of R_k is updated to be the union of R_k with all rectangles R_j such that $x_k^- < x_j^{\pm} < x_k^+$. Additionally, the component of R_k is unioned with the nearest surrounding rectangles $R_{j^{\pm}\pm 1}$ if the values of $g_k[j^{\pm}]$ are greater than 1. Imai and Asano prove that $\sum_{k=1}^{|\mathcal{R}|} |V_k| \leq |\mathcal{R}| - 1$, and therefore the total number of such unions is bounded linear and the algorithm runs in $O(n \log n)$ time total (where $n = |\mathcal{R}|$). □

B. Maximum Clique

Once the intersection graph is obtained per Section III-A, the max-depth cell can be found from the so-called *maximum clique* of $\mathcal{G}_{\mathcal{R}}$. Recall that a clique is a set of vertexes within which every pair of vertexes is adjacent. Interestingly, the general problem of finding a maximum clique is NP-complete [8]. However, due to the characteristics of the graphs generated by the connected components algorithm, the problem can be solved in polynomial time.

To find the maximum clique, we must first observe $d^* = \max_{R_k \in \mathcal{R}} g_k$. This identifies the maximum number of rectangles which intersect one another. To do so efficiently we simply record into each internal node $n_p \in \mathcal{T}(V_k)$ the maximal value of g_k in its subtree as we perform the insertions and deletions of nodes. Then the root of each tree has value $d_k = \max_{j \in V_k} g_k[j]$. For some tree $\mathcal{T}(V_m)$, the root node has $d_m = d^*$. Then the maximum clique is formed by the edges $\left\{x_j^{\pm} | g_m[j] = d^* : j \in V_m\right\}$. One can find these edges by scanning the $O(n \log n)$ vertexes of the tree $\mathcal{T}(V_m)$. In our example above, $d^* = 3$ and $g_8[4] = g_8[5] = d^* = 3$ in Y_8 until we remove R_5, therefore $x_6^{\pm} \cap R_8 \cap R_5$ form a maximal clique. □

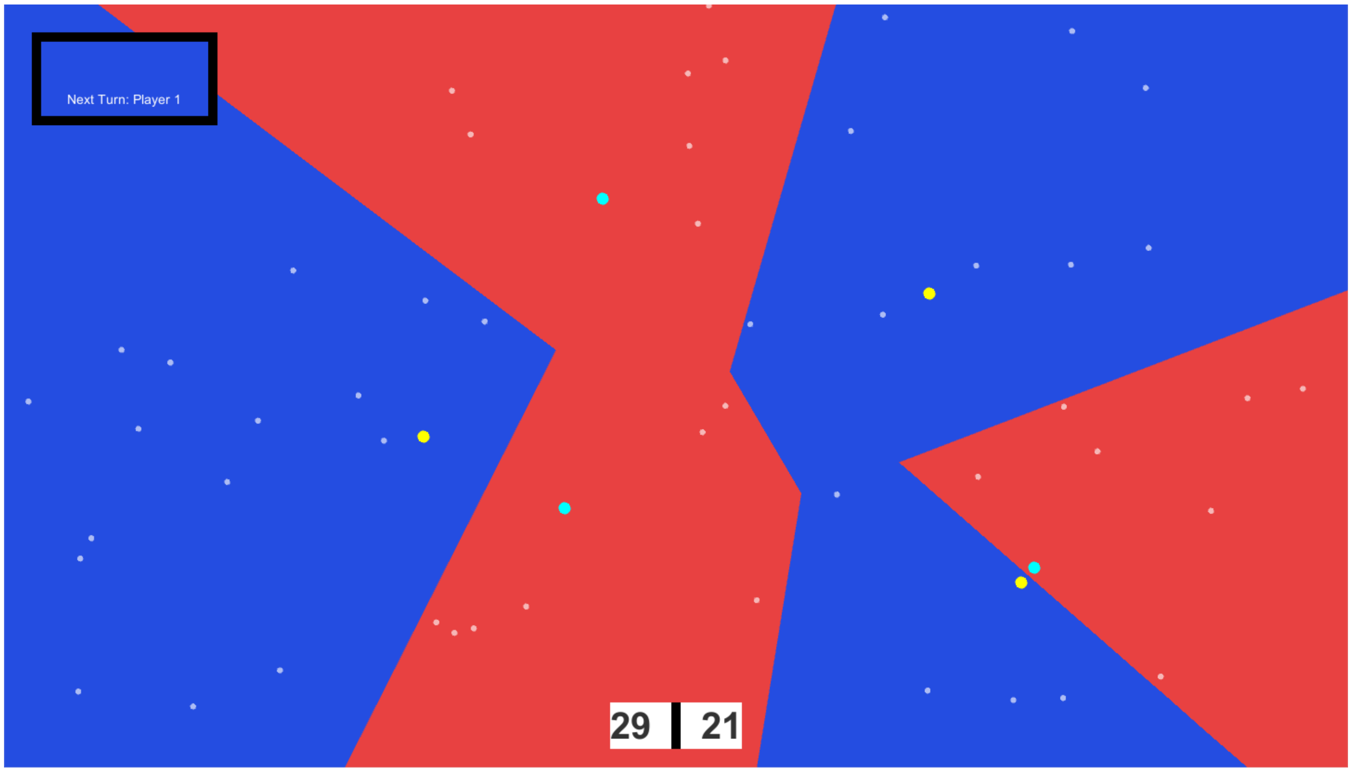


Fig. 7. The discrete Voronoi game after $n = 3$ rounds.

IV. IMPLEMENTATION & RESULTS

A. *Voronoi game and diagram implementation*

The actual game is implemented as a Unity3D [10] project. The user interface and associated code is written in C#, as per usual in a Unity3D project. The human player who interacts with the game represents player one in the discrete Voronoi game. A simple interface is presented where the player is shown all the customer points as white dots. Then the program acts as player two, placing its facility at the optimal player two location given the existing player one and two facilities using the L_1 `MaxCov` algorithm.

As the players play the game, the bichromatic Voronoi diagram is drawn with each player's facilities as the sites. This way the human player can easily see which customer points are within his service area. The human player's areas are drawn blue, and the computer player's areas are drawn red. Figure 7 shows the game after 3 rounds.

The Voronoi diagram itself is constructed using a simple trick. Since we are using Unity3D, we render a right 3-D cone mesh with its base in the x-y plane and its tip out on the negative z axis, towards the viewport camera. The camera faces the cone so that the viewport plane is along x-y as well. The result, when several cones are drawn, is that the graphics hardware naturally renders the cone intersections as the edges of a Voronoi diagram with the sites at the tips of the cone using the depth buffer. Using the Unity3D engine, we then cast a ray from the viewport camera through each customer point on the x-y plane. The first cone the ray intersects describes the customer's service region. Using pointers we can know exactly which player to score. 8 shows the diagram with a light source and the viewport camera tilted slightly so that the characteristics of the cones are clear.

C# doesn't offer very many useful data structures for complex algorithms, so I decided to implement the `MaxCov` algorithm itself in a separate library using C++.

B. `MaxCov` *implementation*

When I began attempting to implement the `MaxCov` algorithm, I initially was unaware of much research into the field. I attempted to follow the algorithm outlined in Cabello et al. [2]. Unfortunately, they do not describe the algorithm very well. In the end, I came up with an algorithm that uses some of the techniques from the true algorithm in Imai [6]. I had attempted to research the problem by various names, and found that it was not well studied under the name "`MaxCov`". Thus, unfortunately, the implementation of my project is not as complete as I would like it to be.

The algorithm I follow is listed in 1. I define $y_k^{\pm}$ and $x_k^{\pm}$ to be the horizontal (top and bottom) and vertical (left and right) edges of input rectangle R_k, respectively. Q is a priority queue initialized with the vertical edges $y_k^{\pm}$ from the input, sorted by descending y coordinate. We will denote $T, B, L,$ and R as the set of top, bottom, left, and right edges, respectively. S is a BST with vertical edges as the leaves, sorted by ascending x coordinate. C is a set of *depth cells*, each of which are the intersection of two rectangles from $\mathcal{R}$ or the intersection of a rectangle R_k and another cell c_k.

The output of my algorithm, rather than a connected components list, is a list of all cells which are formed from

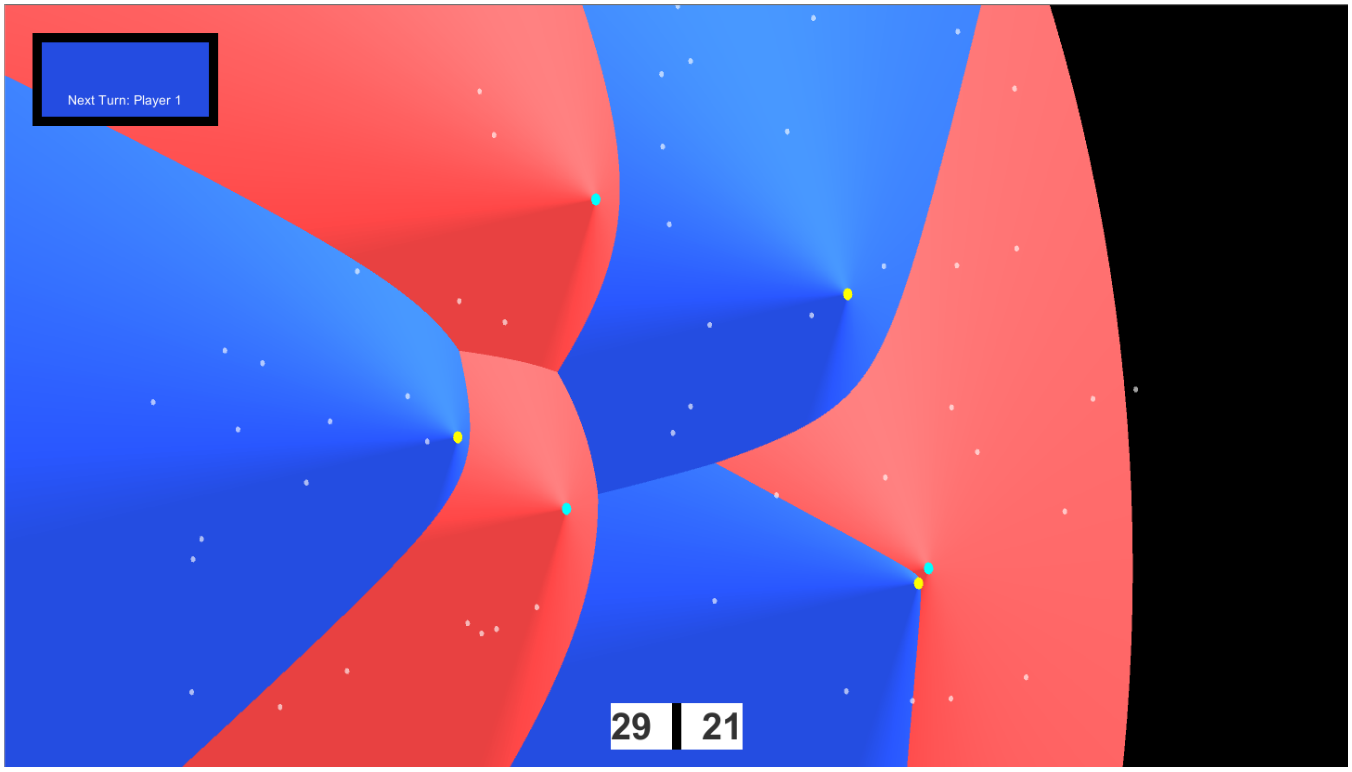


Fig. 8. Voronoi cones for figure 7.

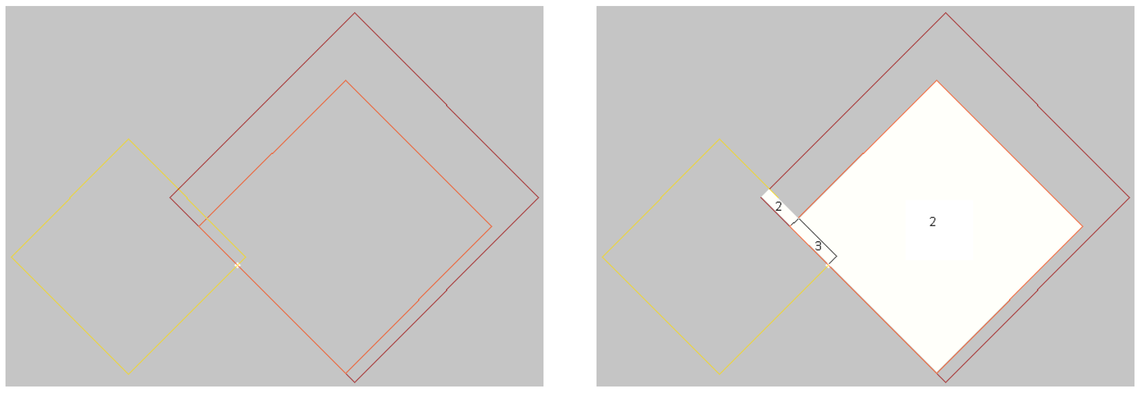
Algorithm 1 L1 MaxCov

```
1:  procedure MAXCOV
2:      $S \leftarrow \emptyset$
3:      $Q \leftarrow \langle y_k^{\pm} \forall R_k \in \mathcal{R} \rangle$
4:      while $Q \neq \emptyset$ do
5:          $e_k \leftarrow pop(Q)$
6:          if $e_k \in T$ then                    ▷ $e_k$ is a top edge
7:              CHECKINTERSECT($e_k$)
8:              INSERT($S, left(R_k)$)
9:              INSERT($S, right(R_k)$)
10:         else                                  ▷ $e_k$ is a bottom edge
11:             REMOVE($S, left(R_k)$)
12:             REMOVE($S, right(R_k)$)
13: function CHECKINTERSECT($y_k^+$)
14:     for $x_j^+ : x_j^+ \geq x_k^+$ do
15:         if $y_k^+ \notin T \cap C$ & INTERSECTS($R_j, R_k$) then
16:             $c \leftarrow R_j \cap R_k$
17:             $depth(c) \leftarrow depth(R_j) + depth(R_k)$
18:             APPEND($C, c$)
19:             PUSH($Q, top(c)$)
20:             PUSH($Q, bottom(c)$)
```

the intersection of rectangles or rectangles and other cells. After some careful thought, it is clear that the output space of algorithm 1 algorithm is exponential. That is, the number of intersections of rectangles with each other and other cells is $O\left(2^n\right)$ for n rectangles. While developing and implementing the algorithm, I did not realize this at the time.

When I was developing the algorithm, I tried to take what I thought might be shortcuts. Cabello describes computing the arrangement of rectangle intersections, into what may be assumed to be a doubly-connected half-edge list (DCHEL). When thinking about the input, I observed that every intersection of axis-up rectangles will produce another axis-up rectangle. From this observation I thought, why generate the entire arrangement? I wondered if it was possible to avoid representing the intermediate cells using a DCHEL or some such complicated data structure, and instead compute only these cells formed by neat rectangle intersections. It is clear that the max depth cell will always be such a rectangle. To explain this further, consider **Fig. 10**.

In the arrangement of such intersecting rectangles, you can see that the white regions are more complex than the red regions in the sense that they each have six edges, whereas the red region is another simple rectangle with only four edges. As a result, I noted that the white regions could grow up to $O\left(n^2\right)$ in space complexity given enough rectangles intersecting appropriately. Yet only regions such as the one shown in red will ever have the greatest depth. As a result of this, I thought I would avoid trying to view the problem as an arrangement of an arbitrary line segments.



(a) Input

(b) Output

Fig. 9. Input and output for a simple L_∞ MaxCov problem.

In the end, of course, my decision to push each new cell back into the queue from lines 19-20 was my downfall. This causes the intersections of newly created cells to compound exponentially, as mentioned before. I added the recursion because intersections of multiple rectangles could not be detected properly without considering intersections between cells and other rectangles.

The algorithm I implemented still works sometimes. However, the exponential problem space means that it is intractible for an interactive game like I intended it to. I wrote a test program to help me understand and debug the various stages of my output, and I can show that the solution is obtained (at needless cost, of course) for some simple cases. As a first example, I was able to generate the rectangles centered on the user points and touching player one's Voronoi site facilities. Figure 11 shows the output of the test program on the same set of sites and user points as in Figure 7. The three sites are shown as bright yellow dots. The number written on each solution cell is the depth of the cell.

Even though the output appears to have three cells, my program generates four, but they are drawn over-top of each other in order of depth, so the lowest depth cells are squashed. In another simple case with two and three facility sites, already my program outputs up to 100 cells.

A much simpler case is shown in Figures 9(a) and 9(b) for clarity.

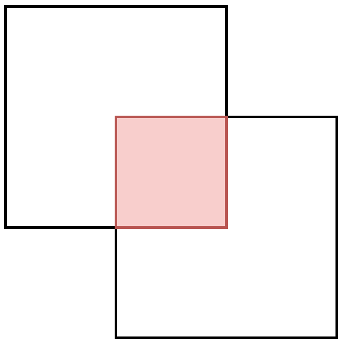


Fig. 10. Intersecting rectangles.

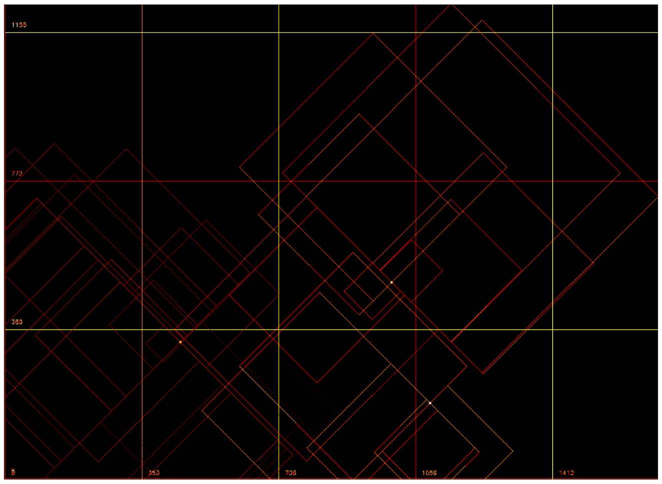


Fig. 11. L_∞ input rectangles after the 3-rd round Voronoi game.

V. IMPROVEMENTS

In order to improve my discrete Voronoi game program, I would clearly start by rewriting my `MaxCov` solution from the shoddy version in Listing 1 to match the algorithm described in great detail by Imai [6]. Of course, after that, I could extend the program to offer the L_2 distance metric for finding the player two solution. Since the problem is fundamentally $\Omega\left(n^2\right)$, there are many more papers describing approximation algorithms for speedups than there were for the "boring" L_1 solution, which has had an $O\left(n \log n\right)$ solution for years. Cabello et al. [2] discusses two good methods for the L_2 problem, and cites many thorough papers on it.

An interesting topic in the future would be applying distance data from traffic flow analysis or geographical maps to actually weight the distance from customer sites. I have seen many papers on the discrete Voronoi game, but none that offer real results or implementations. I would like to see a solution which actually unifies these fun geometric algorithms with real data to be useful for practical competitive facility location.

References

[1] Aritra Banik, Bhaswar B. Bhattacharya, Sandip Das, and Satyaki Mukherjee. The discrete voronoi game in r2. *Computational Geometry*, 63(Supplement C):53 – 62, 2017.

[2] S. Cabello, J.M. Daz-Bez, S. Langerman, C. Seara, and I. Ventura. Facility location problems in the plane based on reverse nearest neighbor queries. *European Journal of Operational Research*, 202(1):99 – 106, 2010.

[3] Tammy Drezner. Competitive facility location in the plane. 7, 12 2014.

[4] H. A. Eiselt, Gilbert Laporte, and Jacques-Franois Thisse. Competitive location models: A framework and bibliography. *Transportation Science*, 27(1):44–54, 1993.

[5] H.A. Eiselt and G. Laporte. Competitive spatial models. *European Journal of Operational Research*, 39(3):231 – 242, 1989.

[6] Hiroshi Imai and Takao Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of Algorithms*, 4(4):310 – 323, 1983.

[7] Jiri Matousek. Computing the center of planar point sets. *Papers from the DIMACS Special Year, Comput. Geom.*, pages 221 – 230, 2000.

[8] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3(1):23–28, Mar 1965.

[9] Hanan Samet. Connected component labeling using quadtrees. *J. ACM*, 28(3):487–501, July 1981.

[10] Unity Technologies. https://unity3d.com, accessed 12-2017.

[11] Sachio Teramoto, Erik D. Demaine, and Ryuhei Uehara. The voronoi game on graphs and its complexity. *Journal of Graph Algorithms and Applications*, 15(4):485–501, 2011.