Optimizing Open Space for Furniture Layout using Quantized Firefly Algorithm

Randall Fowler and Conor King

ABSTRACT

This project explores the application of the Firefly Algorithm (FA), a meta-heuristic inspired by the behavior of tropical fireflies, to optimize the layout of a room. The problem is defined discretely using rectangular objects within a grid. Each object will have a width, depth, rotation, reserved space, and a unique identifier. While optimizing a room may be subjective, the objective function will be defined to promote roundness in open areas. Overlapping objects will be resolved iteratively using a noise separation technique. Results provided demonstrate the proof of concept for small cases with availability for improvements.

KEYWORDS

Firefly Algorithm, Discrete Optimization, Contour Mapping

ACM Reference Format:

Randall Fowler and Conor King . 2024. Optimizing Open Space for Furniture Layout using Quantized Firefly Algorithm. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designing a room layout typically consists of optimizing configuration of a room with the bias of the designer. A desirable space should be efficient and aesthetically pleasing for the users. Traditional methods of designing a room could be considered time-consuming and may not yield the best results. This paper delves into the complexity of optimizing a room layout by arranging objects to maximize the amount of open space.

To reduce the complexity of the optimization problem, the space is discretized into a grid with rectangular. This reduces the amount of search space and relates to the shaping of common furniture. Similar problems would relate to the knapsack problem or optimally designing integrated chips.

Defining the problem as discrete does not reduce the complexity enough to exhaustively search for the best solution. The employed heuristic for this project is the Firefly Algorithm (FA), inspired by the natural behavior of fireflies. Fireflies will represent potential solutions in the solution space. Similar to a normal firefly, these bugs will be attracted to one another and drift toward the brightest firefly. This brightest firefly will be the current approximated best solution that all fireflies will attempt to get close to. If the brightest firefly is far away, the lesser flies will consider local bright flies more than the global brightest.

Firefly movements assist in searching the complex solution space by distributing the flies and allow time for finding better solutions. The novel approach proposed in this paper is to use the FA for spatially reassigning object location and rotations in a room. Due to quantizing the locations of objects, the algorithm will propose challenges relating to the conflicts between objects. New methods were developed for considering the desired location of objects and how to resolve any overlap between objects that fight over the same space.

2 METHODOLOGY

2.1 Firefly Algorithm

The original FA mimics the behavior of tropical fireflies. Each firefly will represent a solution, and a uniform distribution will be used to initialize the location of each firefly throughout the solution space. This follows a similar method to the particle swarm optimization (PSO) algorithm, but there is a major difference between FA and PSO [1]. While particles in PSO will move in the direction of a local better solution, fireflies expand upon this by moving toward the current global best solution. Movement toward the global best solution is handled by shifting fireflies toward the brightest firefly with respect to distance.

The FA is designed for continuous optimization with a maximization objective defined as

$$Maximize \;\; f(x), x \in \mathbb{R}^D. \tag{1}$$

It is an unconstrained problem that requires constraints to be added during movement of the fireflies. For a discrete optimization problem, such as the problem described in this paper, new solutions formed after movement can be quantized to integer values.

Movement of these fireflies is defined by an equation that weights distance similar to absorption of light intensity. Calculation of movement is defined as

$$x_i^{t+1} = x_i^t + \beta_0 e^{-\gamma r_{i,j}^2}(x_j^t - x_i^t) + \alpha \epsilon_i^t, \tag{2}$$

where x_i^t represents the location of the ith firefly in the tth iteration. While the distance between fireflies is a main component in the equation, the weighting is known as the attractiveness. Hyperparameters for this attractiveness term would be β_0, the attractiveness at zero distance, and γ, the absorption coefficient. Distance between solutions is calculated using Cartesian distances, and $r_{i,j}$ would be

$$r_{i,j} = \sqrt{\sum_{k=1}^{D}(x_{i,k} - x_{j,k})^2}, \tag{3}$$

an $l2$ norm of the two fireflies. This attractiveness term will reduce movement between fireflies if the distance is relatively large, and this will enable movement toward the global best solution while encouraging more movement toward local bests.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

https://doi.org/10.1145/nnnnnnn.nnnnnnn

The final term in the movement equation 2 represents the exploration or noisy movement of the fireflies. While the vector ϵ_i^t will contain random numbers following a normal distribution with zero mean and unity variance, the hyperparameter α will be a scaling factor. Both β_0 and α will depend greatly on the scaling of the solution vector.

Another parameter in FA would be the time allowed for movement. With FA being a search algorithm to reduce the need for exhaustive searching. Algorithm 1 describes the pseudocode for FA. For the specified amount of time, the algorithm will iterate over all fireflies, check if the value of other fireflies is greater, and move if so. After every firefly has been visited, the known best will be saved.

Algorithm 1 Firefly Algorithm

Input: Number of fireflies N, number of iterations T
Output: Best solution found

```
Initialize fireflies x_i, i = 1, 2, ..., N.
Evaluate the objective function f(x_i) for each firefly.
Set time t = 0.
while t < T do
  for i = 1 to N do
    for j = 1 to N do
      if f(x_j) > f(x_i) then
        Move firefly i toward j using equation 2.
        Evaluate the objective function f(x_i).
      end if
    end for
  end for
  Sort fireflies by objective function and update the best solution.

  t = t + 1.
end while
```

2.2 Defining Objects in the Room

To reduce the complexity of objects shape, location, and orientation, the optimization of the room layout is defined as a discrete problem with rectangular objects. Each object will have a width, depth, rotation, reserved space, and a unique identification (uid). Rotation is defined by an enumeration of up, right, down, and left with values of 0, 1, 2, and 3 respectively. For values larger or smaller than the enumerated values, the value will be shifted to the respective values, 4 would be equivalent to 0, and -1 would be equivalent to 3. The names of the enumeration can be interchangeable with north, east, south, and west. Reserve space will be considered to be the space in front of the object from wherever it is facing. When something is facing up, the reserve space of the object will be above the object.

Width and depth are the size parameters defining the size by orientation. If an object is facing up or down, the width will represent the distance in the x direction while depth would be in the y direction. This will be the opposite case for facing left and right as depth will be in the x direction and width will be in the y direction. Reserve space will be the distance in front of the object that is reserved open space, and the width of the reserved space will be the same as the width of the object. The purpose of the reserved space is to allow access to the object; most objects, such as a desk, door, or bookshelf, would require space in front of it for someone to stand or sit. Objects that need reserved space in more than one direction are not considered in this paper, but it wouldn't be a large change to add it.

For the uid, objects will have an identification equal to a prime number larger than 1. The reason for this is due to the nature of solving conflicts when moving objects. An open space would be given a value of 1 and adding an object would be to multiply the rectangular area by its uid. If a point in the graph has a non-prime number, this will be an indication of an overlap between objects.

2.3 Defining the Room

A room will be defined simply as its width, height, and graph. The graph will be defined as an array with relative units shared between the sizing of the objects and room. This graph will hold the value 1 for being an open space and a value larger than one is an object taking up that space. For the uid graph, the reserved space will be included in the graph, but for open space graph, the reserved space will not be included. Since reserved space will be open, it will be considered for the evaluation of the layout.

2.4 Solution Vector

Initialization of fireflies requires a uniform distribution amongst the solution space. Each firefly will have an associated solution vector holding the location and rotation of every object in the room.

The dimensions of the solution vector will be defined by the number of objects in the room, and the order of these objects will be by shape and distance from a reference point. This reference point is chosen to be the origin, bottom left corner of the room. One issue that arises from having a set index for the solution vector is that multiple solutions could be the same due to objects having the same shape. This shape is defined by its width, depth, and reserved space. Indexing of the solution vector will be iterated by the different shapes, and for each shape, there will be an indexing in the order of distance to the origin.

Some objects, such as doors and windows, are not moveable or rotatable in the room. For these objects and others that may not want to be moved, these will be left out of the solution vector. However, if a shape is moveable, but not rotatable, it will be included in the solution vector without a rotation variable and as its own shape. Any additional objects with the same shape and without rotation will be grouped together in the solution vector, similar to shapes with rotation.

Random placement of objects within the room will be bound to the size of the room. Coordinates and rotation will be gathered from a uniform distribution, and if the object does not fit in the room, a new coordinate and rotation will be gathered. This technique for generating initial solutions may require more time for more objects as objects will often fail when fitting.

3 DESIGN DECISIONS

3.1 Objective Function

The purpose of this project is to design an optimal layout for a room with known objects. However, this is a challenging problem to address as optimizing the layout of a room is biased toward the

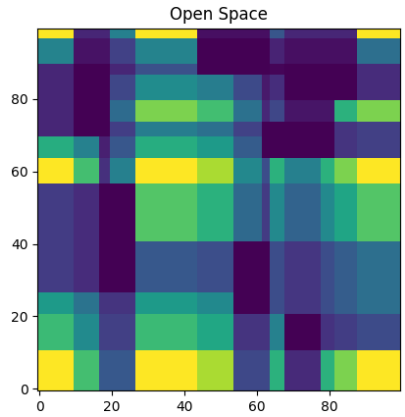


Figure 1: Open Distance Contour Map

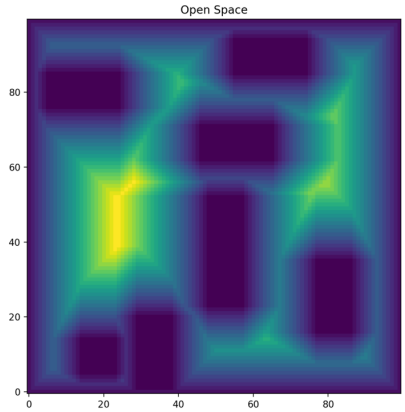


Figure 2: Taxicab Distance Contour Map

opinion of the user. A common preference for individuals when designing their room is to have the largest amount of open space possible. With a set number of objects, the amount of open space would be the same for every possible room arrangement, but maximizing open space for this project holds a different definition. The objective of maximizing open space consists of maximizing the roundness of open areas. This means that larger open spaces would be preferred, but stretched open space would be less preferable than more circular open spaces; A large rectangular open space would be preferred over a long rectangle with the same area.

At first, this objective function was thought to be its own optimization problem where largest ratio of area to perimeter would be useful for estimating value of the room. However, another method emerged to try and satisfy the objective function. A pseudo contour map of the room could be created to find the distance between objects and maximize on spreading objects around the room. For less objects, the optimal solution could be to push everything against the walls, but for a larger number of objects, the layout would need to be creative about placement.

A contour map for the room would be an estimate of distance between objects at every point in the room. Objects themselves would have a value of 0 since that is the location of an object, and the walls could be either seen as an object or not. For this project, it was decided to treat the walls as an object, but exploring the other option could be insightful.

The first idea of a contour map thought of would be known as open distance, and this represents the product of width and height of the open space at a given position of the graph. This algorithm is simply created by iterating over all positions of the graph and getting the width by moving up and down till hitting a wall or object, and the height similarly with left and right. Results for this method are shown in figure 1, and it is difficult to spot where the objects actually are.

The second idea of a contour map was to use Euclidean distance to the nearest object or wall. This algorithm would require a good search method that could keep its searching distance to under the distance to the nearest wall. However, Taxicab geometric distance would provide a similar contour map, but the algorithm could be greatly improved.

Algorithm 2 shows the method for calculating distances. All points would be initialized as a number larger than possible, objects would be added as 0 values, and the search would begin by adding all nodes with 0 value to a list of nodes. While there are nodes in the list, each node will be iterated through, and if a neighbor has a larger value than the current node, the neighbor will be given a value of one larger than its own. After iterating through all nodes, it will then add all nodes with a value of one larger than the nodes it just visited. Eventually, all points in the map will have a distance. Results of this algorithm can be seen in figure 2.

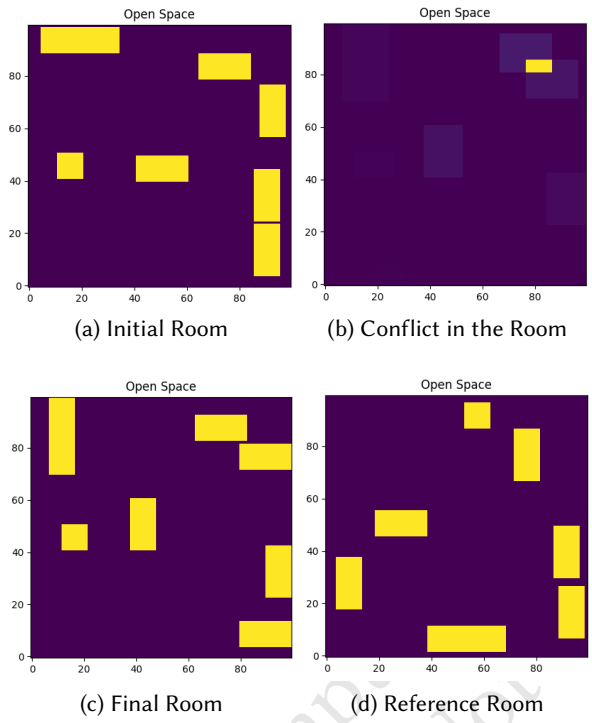
Algorithm 2 Taxicab Distance Contour Map

Input: Room with objects
Output: Contour map of room
  Initialize map to height or width of room.
  Add all objects as 0 values.
  Add all 0 value nodes to a list of nodes.
  while There are nodes in the list **do**
    for Each node in the list **do**
      for Each neighbor of the node **do**
        if Neighbor has a larger value than the current node **then**
          Neighbor is given a value of one larger than the current node.
        end if
      end for
    end for
    Add all nodes with a value of one larger than the nodes it just visited to the list.
  end while

Once the contour map has been found, evaluation of the map will provide the objective function. This value can be found by summing the entire contour map. Larger values will be calculated if there is more space between objects, and the calculated values could be normalized using the size of the room.

3.2 Types of Conflicts

As objects begin to move in the room, there will be conflicts emerging from objects trying to move past the boundaries of the room and overlapping objects. For locations that conflict with the boundaries of the room, the object coordinates will be truncated and set to the position closest to the wall. However, overlapping object is a larger consider with a need for resolve to have a valid solution. Figure 3 displays an example of the algorithm starting with one room in 3a, moving towards another room in 3d, conflicting with overlapping objects in 3b, and ending in a new position 3c.



(a) Initial Room (b) Conflict in the Room

(c) Final Room (d) Reference Room

Figure 3: Moving Objects in a Room

There are 3 known types of conflicts that relate to overlapping objects:

(1) One object didn't move or rotate.
(2) One object didn't move but rotated.
(3) All objects moved.

The first type of conflict is the easiest to resolve as the object that didn't move will be left in its original position. However, it's debatable how to handle the other two conflicts. Some ideas would relate to filling the overlapped position using a random distribution with more weighting given to the objects that didn't move as far. This would require a bit more computation and could provide new conflicts with other objects along the movement path. Another idea would be to apply noise to the objects that have conflicts. The next section discusses this idea where noise is applied to all objects simultaneously.

3.3 Noise Separation

To resolve conflicts, one technique would be to iteratively increase noise applied to an object's location till there is no longer a conflict. Algorithm 3 demonstrates the proposed noise separation algorithm to follow this procedure. Figure 4 contains the results of each iteration for the same example shown in figure 3. The algorithm begins by creating a list of conflicts and removing all of them as shown in 4a. Objects that didn't move are not considered as conflicting objects and remain in the plot. All conflicts in the list will be added back to the plot with noisy new positions. If the new positions fail, the amount of noise will increase and try again, and this can be seen through three iterations between figures 4b to 4d.

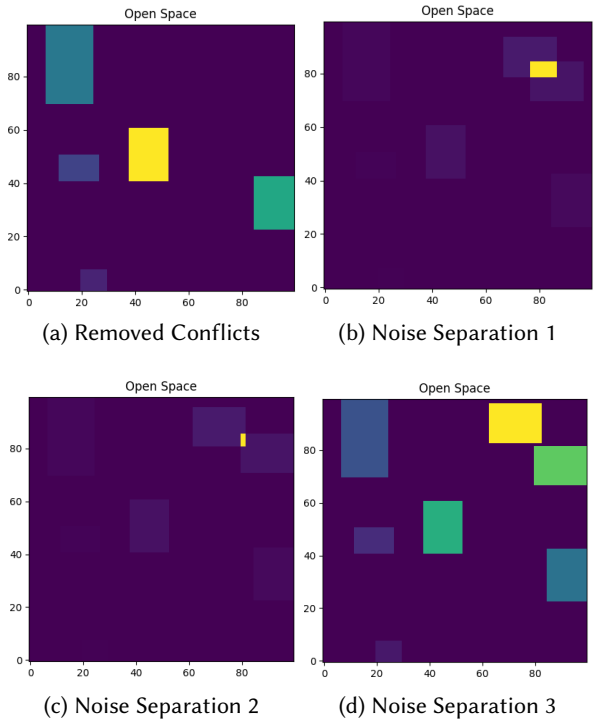
Once an object resolves its position, it will be given that position and the algorithm with check for remaining conflicts for other objects. It will no longer be receiving noise for a new position once a conflict has been resolved. Figure 4d shows the result of the noise separation with no overlapping objects.

Algorithm 3 Noise Separation

Input: New uid space with conflicts
Output: New uid space without conflicts
  Initialize list of conflicts
  Remove conflicts from new uid space.
  Initialize noise parameter σ
  while There are conflicts **do**
    Initialize resolve space as the new uid space..
    for Each object with a conflict **do**
      Scale noise using σ.
      Apply noise to the objects position.
      Add object to the resolve space.
      if There is a conflict **then**
        Add conflict to new list of conflicts.
      end if
    end for
    Get resolved conflicts from difference between new conflicts and old conflicts.
    Add resolved conflicts to new uid space.
    Set new conflicts as the conflicts.
    Increment σ.
  end while

3.4 Hyperparameters

The new solution vector, calculated from fireflies moving toward each other, will contain the locations for objects to move to and updated rotations. Since the rotation has a much smaller set of possible numbers than the coordinates, the hyperparameters α and β_0 may need to be defined differently. These parameters can be treated as a vector with a value relating to position and a value relating to rotation. To get started with testing, α and β_0 were defined as 0.2 and 0.5 for rotation respectively, and these parameters for coordinates were 2 and 5 respectively. Ideally, the movement equation should be adjusted for rotation as it may be preferable for the object to rotate more as it gets closer to the other fireflies but rotate less the further away. For now, the firefly movement remains



(a) Removed Conflicts (b) Noise Separation 1

(c) Noise Separation 2 (d) Noise Separation 3

Figure 4: Moving Objects in a Room

unchanged for rotation, but there is a large exploration that can be done on the hyperparameter space.

4 EXPERIMENTAL TESTS

A shared laboratory space was chosen to optimize for a standard desk configuration. Table 1 represents the inventory of the defined space, including the information pertaining to the sizing of the room and objects. For objects listed, only the doors are considered as unmovable due to the nature of this object. For future consideration, white boards mounted in the room could also be listed as unmovable objects that require reserved space to access.

Table 1: Object Inventory

Object	Width	Depth	Reserved Space	Quantity
Room	40	20	-	-
Desk	5	2	3	16
Shelf	3	1	4	2
Cabinet	3	2	2	3
Couch	6	2	3	1
Table	4	2	3	3
Door	4	0	4	2

When evaluating the room for the objective function, an empty room with the taxicab distance metric would have a value of 0.02155625. This establishes an upper bound on the estimated value of any solution the algorithm might find. For this reason, all evaluations will be normalized with respect to an empty room.

Table 2: Evaluation of Firefly Algorithm with 10 Desks and 10 Iterations

Fireflies	Runtime (s)	Objective Function
10	2	0.180052189
20	10	0.187663091
40	64	0.193461873
80	315	0.19614381
160	2915	0.20585677

Table 3: Evaluation of Firefly Algorithm with 10 Desks and 10 Fireflies

Iterations	Runtime (s)	Objective Function
10	2	0.180052189
20	9	0.184908669
40	17	0.195636416
80	35	0.199115686
320	77	0.203247318
640	157	0.204117135
3000	1627	0.201870107

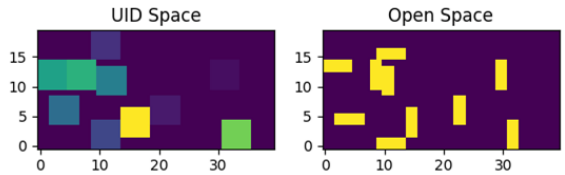


Figure 5: Initial Room Configuration

The algorithm's operation hinges on two key hyperparameters: the initial number of fireflies and the number of iterations. In a basic setup, featuring 10 desks and devoid of additional furniture or doors, both these parameters were systematically altered while keeping the other constant at 10. Both α and β_0 were defined the same as what was specified in section 3.4 and $\gamma = 0.1$. Each test involved recording the final room layout, the corresponding objective function value, and the runtime. Detailed numerical results can be found in tables 2 and 3, with figures 6 and 7 showcasing the optimal room configurations for the best-performing setups from the initial setup shown in figure 5.

In tables 2 and 3, it's evident that for 10 desks and 10 iterations, the algorithm's runtime escalates exponentially as the number of fireflies increases. Doubling the fireflies augments runtime by a factor ranging between 5 and 10, while concurrently enhancing the final solution by approximately 3%. This trend persists until practical testing becomes unfeasible due to excessively long runtimes. Increasing fireflies corresponds to augmenting the algorithm's exploration capability.

Alternatively, for 10 desks and 10 fireflies, the algorithm's runtime demonstrates a linear growth concerning the number of iterations. Initially, there's a substantial improvement compared to

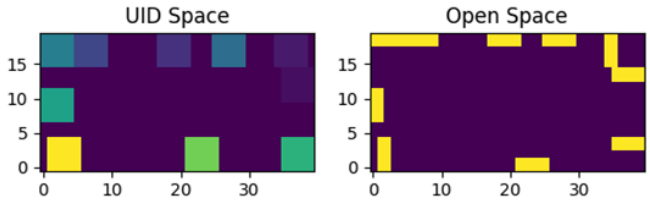


Figure 6: Optimal Room Configuration for 10 Desks and 10 Iterations

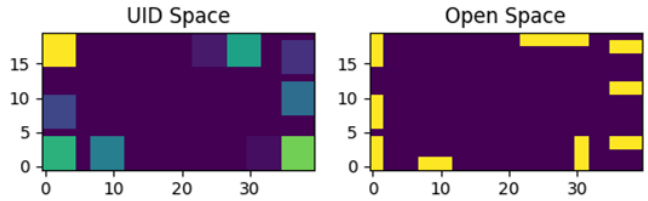


Figure 7: Optimal Room Configuration for 10 Desks and 10 Fireflies

doubling the number of fireflies, approximately 6% versus 3%. However, increasing iterations enhance the exploitative aspect of the algorithm, with improvements tapering off as iterations increase and local minima are encountered. Notably, the performance of the last row in the tables is worse than the preceding row. This indicates that increasing the number of iterations to a larger number than 640 may not be beneficial to the performance.

Although the difference in objective function between the first and last rows in table 2 and 3 may seem negligible, a visual inspection of figures 5 through 7 reveals a significant variance in the optimality of room layouts. There is a 67% increase in the value of the room between these figures with figure 5 having a score of 0.122934184. The algorithm tends to prioritize moving objects towards the walls, albeit with notable spatial issues evident in the final solutions depicted in the figures.

The next phase involves assessing the algorithm's performance as the room's complexity or "busyness" escalates. This proves more challenging to objectively test compared to simply increasing parameters. Table 4 outlines the strategy employed for this evaluation, where each entry denotes the presence of furnishings from its column in the respective complexity test's row. For each test, 20 fireflies were initialized, and the algorithm ran for 30 iterations. Figures 8 to 10 depict the final room configurations for each complexity test. For Complexity 3, the number of furnishings was substantial enough to prompt the algorithm to deviate from the strategy of solely aligning objects against walls. The absence of cabinets and shelves across all rows in table 4, with the number of desks never exceeding 16, indicates that the algorithm tends to stall out during initialization as the complexity increases. This is due to the escalating challenges of finding conflict-free random configurations, and the objective function over iterations can be seen in figure 11.

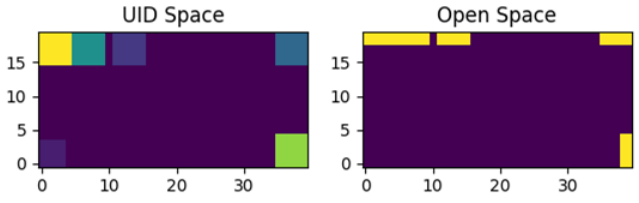


Figure 8: Optimal Room Configuration for Complexity 1

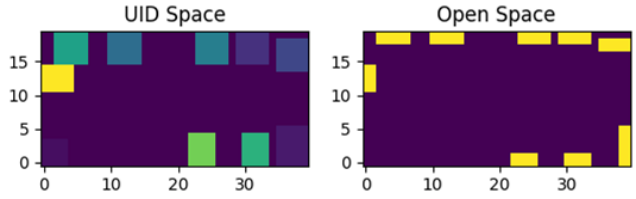


Figure 9: Optimal Room Configuration for Complexity 2

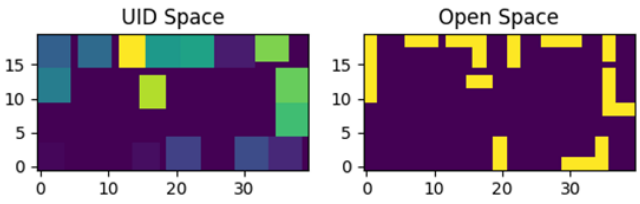


Figure 10: Optimal Room Configuration for Complexity 3

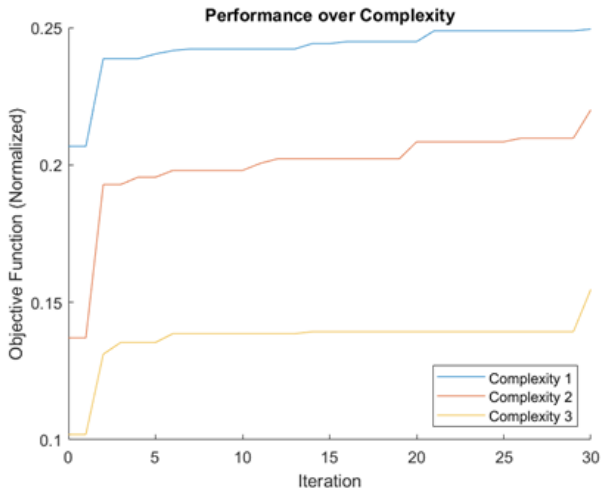


Figure 11: Objective Function Over Iterations for Complexity Tests with 20 Fireflies

Table 4: Room Complexity Tests

Complexity	Doors	Desks	Couches	Tables
1	1	5	0	0
2	1	5	1	3
3	2	10	1	3

Table 5: Evaluation of Firefly Algorithm with 10 Desks, a Couch, and Two Doors

Fireflies	Iterations	Runtime (s)	Objective Function
10	10	3	0.168382140
40	10	64	0.179689765
10	100	69	0.183603943
40	50	594	0.186213395
30	100	793	0.186720789
50	200	3064	0.192012177

For a final test of 10 desks, a couch, and two doors, the performance over various fireflies and iterations are provided in table 5. The table is sorted by the normalized score to visualize the required number of fireflies and iterations to improve the room's value. This may vary greatly from other tables due to the number of objects in the room.

5 CONCLUSIONS

The problems of optimally designing furniture layout were explored through an algorithmic approach. With the firefly algorithm (FA), the search space was significantly reduced to a manageable runtime. Results proved to be promising as the estimated optimal solutions were reasonably close to what would be expected. With less objects in a room, objects were moved to the walls to provide large amounts of open space. Challenges arouse when object could no longer be placed along the wall.

When objects were placed into the room, they were randomly placed one at a time. This approach did work for this project, but the runtime would greatly increase when the room was densely populated with objects. An alternative approach to attempt in the future would be to place all the objects at once or distribute positions uniformly rather than in random locations.

The evaluation strategy of using contour maps was satisfactory as it did provide information for maximizing on the open space. One issue that wasn't resolved was contiguous open space. When objects cut off areas of space from one another, the layout would no longer be valid. Ideas to resolve this would relate to finding the shortest distance between the divided open spaced. However, the connection between open spaces would need to be redefined to allow someone to walk from one space to the next.

Noise separation did resolve conflicts rather well, but there may be plenty of edge cases that it would not work well for. If objects are getting close, they may not be able to resolve by wiggling the objects around with noise. Further testing on this method could provide more clarity into the usefulness of it.

In conclusion, the method of designing the room proved to be adequate by getting close to solutions that would be expected. There are plenty of improvements that could be made to this project, but the proof of concept appears to be valid.

ACKNOWLEDGMENT

The authors would like to thank Dr. Soheil Ghiasi for his lectures relating to the optimization aspects of this project.

CONTRIBUTIONS

Randall Fowler: Algorithm development, algorithm implementation, and experimental testing. Conor King: Algorithm development and experimental testing.

REFERENCES

[1] Slowik, A. (Ed.). (2020). Swarm Intelligence Algorithms: A Tutorial (1st ed.). CRC Press. https://doi.org/10.1201/9780429422614