Universiteit Leiden
The Netherlands

Bachelor Computer Science

Optimization algorithms for solving

incremental box placement problems.

Dennis Moser

Supervisors:
Micheal Emmerich
Yali Wang

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

13/07/2021

Abstract

In this paper the Incremental Box Placement (IBP) problem is defined, which is similar to an incremental version of the two-dimensional Bin Packing (2BP) problem. The primary goal of the thesis was to find the best algorithm for the IBP problem. To this end, common metaheuristics, Simulated Annealing and Particle Swarm Optimization, were implemented, a tailor-made algorithm was designed and created and a web dashboard application with testing and visualization features was built. All the algorithms were also specifically tuned for the IBP problem. It was concluded that Simulated Annealing performed the best in both the score and time metrics, notwithstanding some incidental unintended behavior by the algorithm. This behavior can be ascribed to the design of the performance metrics, which influences algorithm behavior. The custom-made algorithm called Adaptive Structure Optimization used targeting and propagation to improve on the base (1 + 1) algorithm, but did not manage to best Simulated Annealing in the defined performance metrics.

Contents

1 Introduction

1.1 Problem Definition

In this paper we will use common metaheuristics and create tailor-made algorithms to solve incremental box placement problems. We define the Incremental Box Placement (IBP) problem to be a geometric-optimization problem tasked with optimizing for minimum overlap and minimum displacement as boxes are incrementally placed on a board.

- *box*: In this context we define a box to be a two-dimensional rectangular shape. You could also define other variations of the problem that use any other geometric shapes.
- *incremental placement*: The set of boxes that are to be placed on the board are not placed all at once. Instead, they are placed one by one and the positions of the boxes are to be optimized for minimum overlap and minimum displacement during each iteration.
- *overlap*: Overlap occurs when one or more boxes occupy the same space. This is something that typically happens when the target area of a new box placement is already occupied by a previously placed box. Later in this paper we will explain how overlap and displacement are calculated, scored and weighted.
- *displacement*: Displacement is the total distance moved by boxes on the board as a result of making room for a new box. This is the sum of the distances between the original box positions and those after the new box was placed. The goal of minimizing this value is to avoid unnecessary movement of boxes.
- *board*: The board is the two-dimensional rectangular container wherein the boxes are placed.

Additionally, this version of the IBP problem has the concept of a *target location* for the box to be placed. After the box is placed the distance from its target location is calculated and added to the displacement value of the configuration. This is only done for the box that is currently being placed.

The placement of a box can only be an *axis aligned placement without rotation*. This means that each box must be parallel to the sides of the board and is not allowed to be rotated.

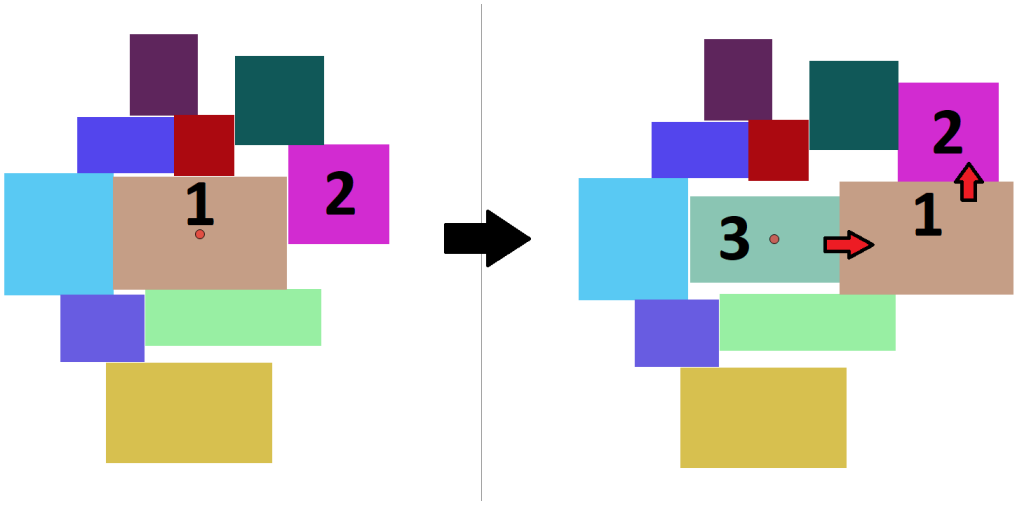

Figure 1.1: IBP problem: The placement of box 3 causes box 1 and box 2 to be moved out of the way.

In figure 1.1 a typical scenario in an IBP problem is visualized. The left side of the picture is the initial configuration of boxes. The target location of the new box to be added to the system is marked by the red circle, which can be found roughly in the center of box 1. On the right side you can see that box 3 has been added on the target location circle, but in order to avoid overlap boxes 1 and 2 were moved out of the way to create space.

The incremental box placement problem bears many similarities to the two-dimensional Bin Packing (2BP) problem. Bin packing is a well known optimization problem that has been widely researched since being listed by Garey and Johnson in Computers and Intractability [3]. Bin packing is NP-hard and therefore difficult to solve.

In a typical bin packing problem a finite set of objects of different sizes must be packed within a defined container. For the IBP problem the objects are boxes, the container is the board and instead of fitting all the objects at once they are placed one by one and the algorithm optimizes for minimum overlap and minimum displacement in each iteration.

Bin packing also has no concept of displacement, since objects are never moved. The purpose of a displacement score is to penalize unnecessary movement and thereby preserve stability. This is critical in certain applications of the algorithms, as will be discussed later. A good example for showcasing the value of stability and penalizing disruptions is the Airline Schedule Disturbances Problem (ASDP) in airline schedule optimization [1].

During the normal operation of an airport disruptions can occur that affect an airline's flight schedules. These disruptions can be very costly for an airline and models have been built to provide new flight schedule proposals that minimize the negative effects of such disruptions [9]. Similar to how these solutions penalize disruptions and assigns costs to its effects to optimize for airline profit, the IBP problem assigns a cost to moving boxes. These costs can be translated to real-world scenarios. For example, in a three-dimensional version of the IBP problem a practical costs of displacement could be a person having to physically lift and

move packages.

A more applicable two-dimensional application is a graphical user interface (GUI). The exploration of new user interface techniques by placing and moving geometric shapes on a screen is the main inspiration for this paper. Specifically the web browser has improved significantly in its capabilities over the last decade, but most sites still use it as a document viewer with static HTML pages. Far more interactive experiences are now possible in the web browser by employing client side scripting and new rendering techniques. Algorithms that solve the IBP problem can be used to built content windowing systems that self-organize their placements to improve the UI of content suggestion, browsing and file management systems. In this case the cost of displacement of content on the screen would be a negative User Experience (UX).

1.2 Research Objective

- Research question 1 (**general goal of thesis**):

  What is the best algorithm for the problem of incremental box placement?

- Research question 2 (performance metrics):

  How do we measure performance of the incremental box placement algorithms?

  How do the performance measures used in the algorithm influence the behavior of the algorithms?

- Research question 3 (algorithm comparison):

  How do common metaheuristics perform compared to tailor-made algorithms?

  How do parameter settings influence performance?

1.3 Thesis Structure

In chapter 2 Methods we will discuss the different algorithms that were used, describe their implementations, explain the performance metrics and define the parameters. Then in chapter 3 Experimental Setup we will showcase the application and dashboard interface that was built to run the algorithms and visualize the results. We will also go over the parameter tuning efforts and list the final parameter settings that were used. The next chapter 4 Results contains the results of our experiment. In this chapter we go over each algorithm's performance and characteristics, and use this to work towards answering the research questions. Finally, in chapter 5 Conclusions we answer the research questions, discuss any other observations and limitations and discuss future work.

2 Methods

2.1 Optimization

Mathematical optimization is finding the best configuration from a set, or finding the optimum of a problem with respect to defined criterion. In a basic optimization problem there is an objective function, $f(x)$, which is being maximized or minimized [10]. The variables x_1, x_2, x_3 and so forth are the inputs to the objective function, these are typically the parameters of the algorithm. In this paper we will build an objective function using a scoring system for *overlap* and *displacement* from 0 to 1 that we will then attempt to maximize using the optimization algorithms that will be described next. A list of parameter definitions can be found in section 2.4 Parameter Values.

2.2 Optimization algorithms

2.2.1 (1+1)-Evolutionary Algorithm

Background

The (1 + 1) Evolutionary Algorithm is a very simple variant of an Evolutionary Algorithm (EA). It will be used as the base case to which we can compare our implementations of the other algorithms. Evolutionary algorithm are inspired by evolution in nature [4]. As such, concepts such as *generations*, $fitness$, *crossover* and *mutation* are used to find the optimal solution. In our discussions of EAs a possible solution is referred to as an *individual* (as some representation of a configuration) and the set is called the *population*. For the (1 + 1) algorithm, for each generation, there will be one parent and one child [11]. Mutation happens by randomly selecting one element from the configuration and changing it with a mutation function. The best configuration between the parent and the child will then be chosen for the next generation.

Implementation

The (1 + 1) algorithm that was created for this paper is purposefully very basic. The initial configuration is the current location of all the boxes on the board, together with the dimensions of the board and each box. The box that is being placed is part of the initial configuration and is located on its target location, any movement away from the target location is counted as displacement. This is the initial configuration for every algorithm in this paper.

Every iteration, the mutation function is executed once. The algorithm for the mutation function is described below. This mutation function is used in every algorithm implementation, except for Particle Swarm Optimization.

Algorithm 1 Mutation Function

1: **procedure** MUTATIONFUNCTION
2: Choose one box b out of the configuration at random.
3: Generate two random floating-point numbers $d1$ and $d2$ in the range from -($MutationStrength$ / 2) to ($MutationStrength$ / 2).
4: Update the x-axis location of b by adding $d1$ and update the y-axis location b by adding $d2$.
5: Confirm that b still resides within the boundaries of the board. If not, move the box against the boundary of the relevant axis.
6: **end procedure**

After mutation, the best configuration is chosen for the next generation. The (1 + 1) algorithm stops executing when the $MaxConfigurations$ has been reached or the score of the current configuration satisfies the $AcceptCriteria$ value. These are stop conditions, where $MaxConfigurations$ is the maximum number of score evaluations and $AcceptCriteria$ is a score value that will allow the algorithm to stop early because the current configuration is considered "good enough".

2.2.2 Simulated Annealing (SA)

Background

The SA algorithm was introduced by Kirkpatrick et al. and was inspired by the heat treatment process called annealing in metal working [5]. Annealing consists of different stages of controlled heating and cooling, which are meant to reduce hardness and increase ductility of the metal. The SA algorithm adopts this principle through a variable temperature parameter that imitates the annealing of metals [2]. The temperature variable will dictate the likelihood of a configuration getting *accepted* even if its score is less than the current best configuration. Getting *accepted* means that it will be the new current configuration in the next generation from which new mutations are made. The (1 + 1) algorithm never accepts new configurations that have a score less than that of the current best, but that also causes it to get stuck in local optima. This means that it

never explores and evaluates the best possible configurations because all the neighboring configurations of the current configuration are worse. Through the annealing mechanism the SA algorithm should be able to do a broader search when temperature is high and then also hone in on the optimal configuration when the temperature is low.

Implementation

The SA algorithms starts with the same initial configuration as (1 + 1), uses the same mutation function and has the same stop conditions. Where it differs is in how it uses the temperature value to decide if a configuration should be accepted rather than just checking which score is greater. The current temperature T starts at the initial temperate T^0 and is then cooled by a factor alpha a after every n iterations. The variable n is obtained by dividing the $MaxConfigurations$ value by 400. Since $MaxConfigurations$=40,000 in this paper, the cooling rate was only applied after the SA algorithm tried 100 mutations at the current temperature. The formula for cooling is shown in 2.1 below.

$$T = T * \alpha \tag{2.1}$$

During each iteration the $AcceptanceProbabilty$ is calculated as follows in 2.2 and 2.3.

$$ScoreDifference = NewScore - CurrentScore \tag{2.2}$$

$$AcceptanceProbability = \frac{1}{1 + \exp(\frac{ScoreDifference}{T})} \tag{2.3}$$

This is then compared to a randomly generated number to decide if the configuration should be accepted.

2.2.3 Particle Swarm Optimization (PSO)

Background

Whereas each generation of the (1 + 1) algorithm has a population size of 1, the PSO has a population size of pop and each member of the population is called a particle. PSO was originally meant to imitate social behavior and is often represented as a flock of birds [7]. The particles influence each other as they are moving across the search-space. Each has a position and gets updated by the algorithm based on their current velocity (v_i), individual best location (b_i) and the global best location (g).

Implementation

During the initialization of a typical PSO algorithm each particle in the population has a randomized configuration. In the IBP problem this would mean that all the boxes on the board have a random location. This would immediately introduce huge displacement. PSO would then have to iteratively move all the boxes back to near their original positions to find the optimal solution. That's why we opted to keep the initial configuration the same for every particle in the population without randomizing the locations. We rely, instead, solely on randomized initial velocities v to explore the search-space.

2.2.4 Adaptive Structure Optimization (ASO)

The ASO algorithm is a new solution that has been specifically created by the author of this paper to tackle the IBP problem. The starting point of the algorithm is the (1 + 1) algorithm, to which new features are added in an attempt to improve its performance. The ASO is called *adaptive* because it evaluates the full structure of a given configuration and adapts to its weaknesses and strengths to take a more *targeted* approach to the mutation function [8].

This *targeting* mechanism works as shown in the following algorithm:

Algorithm 2 Adaptive Mutation Function

```
1: procedure AdaptiveMutationFunction
2:     Evaluate each individual box in the configuration and assign overlap and displacement scores.
3:     Pick a random box b, where worse scores give a box a higher chance to get picked.
4:     Execute the original mutation function only on box b.
5: end procedure
```

When the boxes are evaluated their scores are flipped by doing $x = 1 - x$, such that a score of 0.3 becomes a 0.7. This is done because we want the boxes with a low score to have a higher likelihood of getting picked. The code of the random box picking function is shown in algorithm 3.

Algorithm 3 Pick random box based on scores

```
1: procedure pickRandomBoxBasedOnScores(boxes, totalScoreSum, highestScore)
2:     totalScore ← totalScore + highestScore
3:     bonusScore ← highestScore / numberOfBoxes
4:     pickerTotal ← 0
5:     randomNumber ← random number between 0 and totalScore
6:     for each box ∈ boxes do
7:         pickerTotal ← pickerTotal + boxScore + bonusScore
8:         if randomNumber ≤ pickerTotal then
9:             return box
10:        end if
11:    end for
12: end procedure
```

You can think of this code as a picker wheel where the size of a boxes' section on the wheel is based on its individual overlap and displacement scores. As such, the chance of getting picked is inversely correlated to the quality of that boxes' scores. A bonus score is also given to every box so that even with zero overlap and zero displacement the box still has a chance to get picked.

2.2.5 Adaptive Structure Optimization with Propagation (ASOP)

Building on the ASO algorithm, we will add another feature. Propagation means that every time that a box is mutated it has a chance to spread the mutation to boxes that it interacts with [6]. Interaction is defined by any two boxes overlapping each other.

Interaction is decided using the new position of a box after it has been mutated. Whether the mutation is propagated to other boxes depends on a propagation rate variable p that is simply compared to a randomly generated number between 0 and 1 (1 not including) every time that an interaction is determined. The algorithm also keeps track of a list of mutated boxes so that a mutated box cannot get mutated multiple times in a single iteration of the algorithm.

2.3 Scoring

As part of the initialization of every algorithm the *MaxOverlap* and *MaxDisplacement* are calculated so that they can be used in the scoring systems to calculate a score between 0 and 1 for a given amount of overlap or displacement. The *MaxOverlap* is the sum of the sizes of all the boxes and the *MaxDisplacement* is calculated as follows in 2.4:

$$MaxDisplacement = \sqrt{BoardWidth * BoardHeight} * NumberOfBoxes \tag{2.4}$$

In the code implementation every box has a $x1$, $x2$, $y1$ and $y2$ value that represent the position of the four corners of the box within the board. These values are used to calculate an *OverlapScore* as shown in algorithm 4.

Algorithm 4 Overlap score calculation

```
1: procedure calculateOverlapScore(boxes, maximumOverlap)
2:     totalOverlap ← 0
3:     for each boxOne ∈ boxes do
4:         totalOverlapForBox ← 0
5:         for each remaining boxTwo ∈ boxes do
6:             overlap ← calculate overlap between boxOne and boxTwo
7:             totalOverlapForBox ← totalOverlapForBox + overlap
8:         end for
9:         totalOverlap ← totalOverlap + totalOverlapForBox
10:    end for
11:    overlapScore ← 1 − (totalOverlap / maximumOverlap)
12: end procedure
```

A *DisplacementScore* is calculated as shown in the code in algorithm 5.

Algorithm 5 Displacement score calculation

```
1: procedure calculateDisplacementScore(currentBoxes, newBoxes, maximumDisplacement)
2:     totalDisplacement ← 0
3:     for each currentBox ∈ currentBoxes and newBox ∈ newBoxes do
4:         displacement ← distance between currentBox and newBox
5:         totalDisplacement ← totalDisplacement + displacement
6:     end for
7:     displacementScore ← 1 − (totalDisplacement / maximumDisplacement)
8: end procedure
```

The application uses a linear weighting scheme to calculate an *OverallScore* using the *OverlapScore*, *DisplacementScore* and their weighting values *OverlapWeight* and *DisplacementWeight*. The following formula 2.5 is used to calculate the *OverallScore*:

$$OverallScore = \frac{((DisplacementWeight * DisplacementScore) + (OverlapWeight * OverlapScore))}{(DisplacementWeight + OverlapWeight)} \tag{2.5}$$

In this paper *OverlapWeight*=10 and *DisplacementWeight*=1. The *OverlapWeight* needs to be considerably higher than the *DisplacementWeight* to get the desired behavior of the algorithms where they actually try to reduce overlap. If the weights are set to similar values, the algorithms are less likely to move stacked boxes because while it might decrease overlap it also always increases displacement. After experimenting with many weight values, a 10-to-1 ratio was found to encourage the best result. Reducing overlap is the clear priority, but reducing displacement is also not just an afterthought for the algorithms.

2.4 Parameter values

2.4.1 Global Parameters

- $MaxConfigurations$: The maximum number of score evaluations the algorithm is allowed to run before execution is stopped.
- $AcceptCriteria$: The score value that must be reached for the algorithm to be allowed to stop before the $MaxConfigurations$ value is reached.
- $TargetLocation$: A position within the board expressed by a left and top pixel value that represents the center of the latest placed box.
- $NumberOfShapes$: The number of boxes that will be placed on the board in successive runs of the algorithm during an algorithm's evaluation.

2.4.2 Algorithm Parameters

- $MutationStrength$: A parameter used in the mutation function to calculate the maximum distance that a box can be displaced in the x and y directions.
- T^0: The initial temperature used in the SA algorithm.
- α: The alpha factor in the SA algorithm, which determines the cooling rate of the system.
- pop: Population size of the PSO algorithm.
- v: Initial velocity of the PSO algorithm.
- $r1$: Inertia coefficient of the PSO algorithm.
- $r2$: Cognitive coefficient of the PSO algorithm.
- $r3$: Social coefficient of the PSO algorithm.
- p: The propagation rate of the ASOP algorithm.

3 Experimental Setup

3.1 Software

A single-page web application was built to test the algorithms and visualize the results. This website is publicly accessible and can be found at https://sysmo.com/Thesis and the GitHub repository can be found at https://github.com/dmoserreal/ibp. It allows for changing the global parameters and running experiments with a sequence of algorithm-specific settings. Finally, it calculates averages, standard deviations and stores the results.

The client-side application is written in JavaScript, which then communicates with a server that stores the results in an SQL database. All the optimization algorithms are implemented on the client-side, because we are interested in the prospect of using the algorithms for user interface solutions. The data calculations and data storage management is done on the server-side.

No special third-party libraries were used to create any part of the application other than what a developer might typically use to build any generic web application that has a component of data management. As it is a single-page web application, all content and data is loaded dynamically without page refreshes. This allows us to run a sequence of experiments and browse the results without having to reload the website.

3.2 UI Components

A screenshot of the full interface of the dashboard is displayed in figure 3.1. The front-end of the application consists of two pages that can be accessed through the header navigation bar. The first is the Experiment Dashboard where the optimization algorithms can be ran and tested. The second is the Data Viewer page, where results from past experiments can be explored and exported.

In the left sidebar it is possible to adjust the $MaxConfigurations$ parameter and the $Overlap$ and $Displacement$ weightings. The "Current Config" section displays scoring details on the box configuration that is currently displayed on the screen, such as displacement score, overlap score and overall score. It is also possible to choose a particular algorithm in the "Algorithms" section if the user is going to manually add boxes using the controls in the right sidebar.

Figure 3.1: User interface of the dashboard.

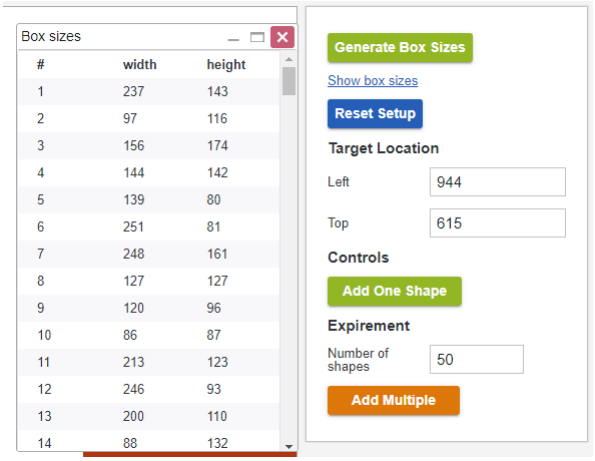

Figure 3.2: Box sizes interface.

The right sidebar has a button for generating new box sizes (figure 3.2), but for this paper the box sizes were kept static and can be found in appendix table 6.1. There are two input boxes for setting the target location of the next box to be placed, from which its displacement value will be calculated. This location can also be adjusted by clicking anywhere on the board. For this paper the target location was also kept static at the center of the screen. The *Numberof Shapes* parameter can be adjusted in the right sidebar and there are controls for manually incrementally adding one or multiple boxes to the board.

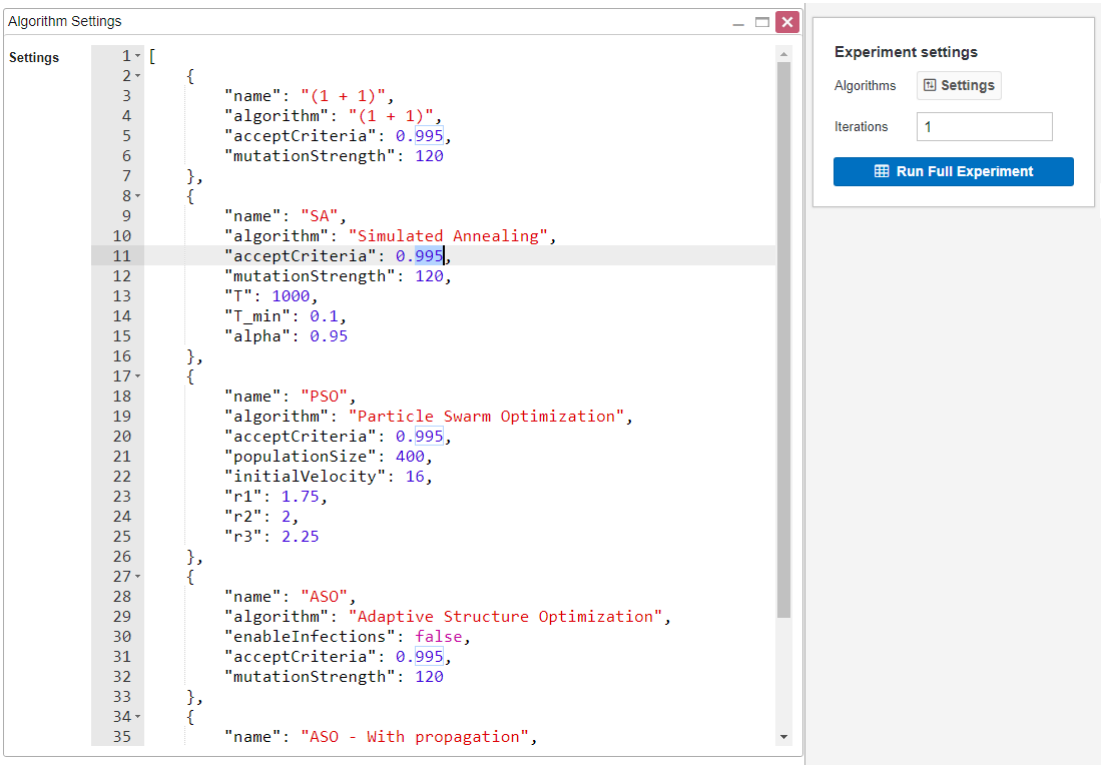

Figure 3.3: Experiment settings section.

The most critical section in the right sidebar is "Experiment settings", displayed in figure 3.3. The settings button opens up a window called "Algorithm Settings". In this window the user can set up a series of algorithms to be ran sequentially by the software. The user will have to enter these in a JSON format. For example, in figure 3.3 the algorithm settings in JSON that are displayed are those that were used in this paper to compare the algorithm performances. This feature was also used to run the same algorithm repeatedly with slightly different parameters for the purpose of tuning the algorithm parameters. The "Experiment settings" section also has an input box for changing the Iterations parameter, which controls how often each algorithm is ran and finally a button to start the experiment.

3.3 Workflow

First, the interface is used to set up all the global parameter settings, which are the same for all experiments and are listed in the next section 3.4 Parameter Settings. For the purpose of parameter tuning, a series of JSON files are created for each algorithm with a sequence of different parameter settings, which are loaded one by one in the algorithm settings window. Finally, a JSON file is created containing the tuned settings for each algorithm so that the performance can be compared.

Each time an experiment is ran, the behavior of the algorithms can be monitored visually through the display of the box configurations after each box is placed. A window also pops up that shows the score and elapsed time of each box placement. There is also a window titled "Experiment: *ExperimentId*" that displays average

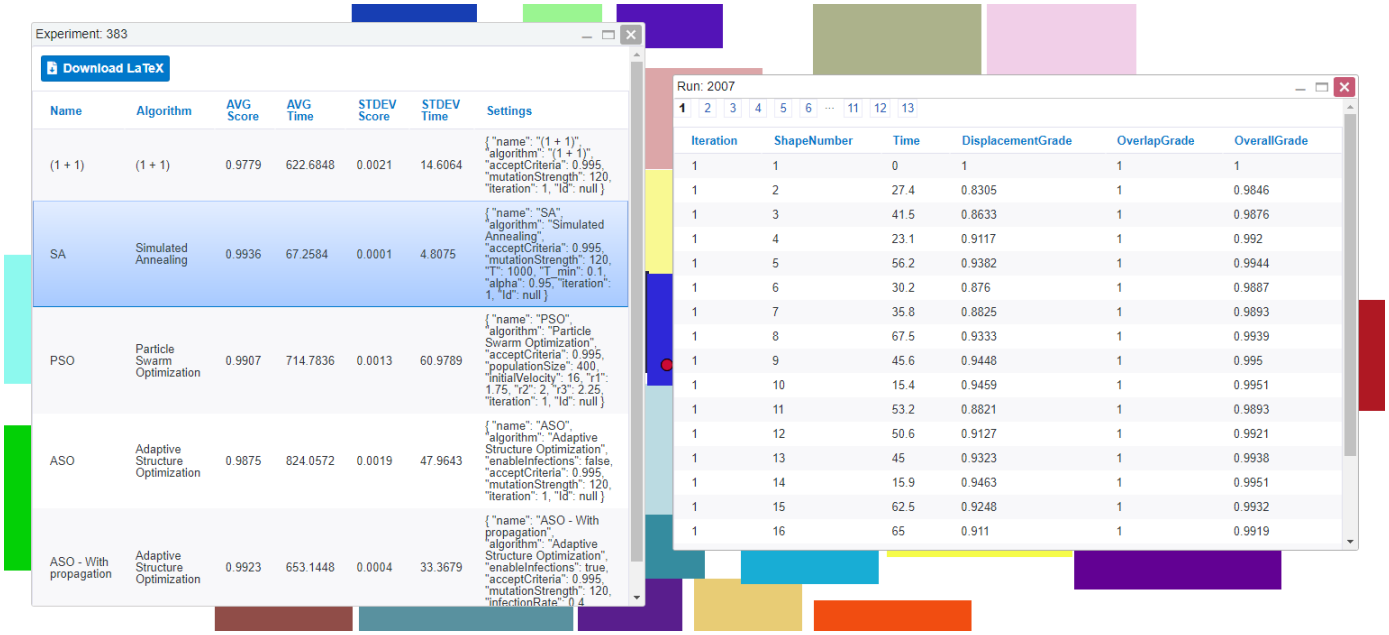

Figure 3.4: Browsing and exporting data.

scores, average elapsed times in milliseconds and standard deviations for each algorithm. This window is shown in figure 3.4. All the data is stored in the SQL database, but there is also a convenient download button that converts the data straight to LaTeX tables.

3.4 Parameter settings

The *MaxConfigurations* parameter is set to 40,000 and the *AcceptCriteria* parameter is set to 0.995 for all algorithms. This is a matter of preference for the user of the application, if these parameters are set to lower values it will speed up the algorithms, but they will produce worse scores. In particular, the *AcceptCriteria* cut-off also introduces a lot of variance in the time measures, because the time measured for an algorithm that reaches the accept criteria threshold can be a fraction of one that doesn't. If a user would like to speed up the algorithms they could lower the max configurations and accept criteria parameters and the effect would be quite significant on average time spent computing with limited deterioration in the quality of the results. This would, however, introduce an increased sensitivity to randomness, which we would like to avoid while we are evaluating the algorithms for their performance in this paper.

The size of the board that the boxes were placed on was 1657x1106 pixels. This size is typical when loading the application on a 2560x1440 pixels monitor. The *NumberOfShapes* parameter is set to 50. This value was chosen because it tests the algorithms in scenarios where it is difficult to make room for the newest boxes without causing unavoidable overlaps on the board.

3.4.1 (1 + 1)-Evolutionary Algorithm

The (1 + 1) algorithm only has one parameter that was tuned, the *MutationStrength*. The captured data is displayed in table 3.1. When the *MutationStrength* is set to a low value, 100 or less, the (1 + 1) algorithm seems to be more likely to get stuck in local optimums as smaller movements in boxes often don't go far enough to cause a reduction in overlap. A higher *MutationStrength*, which gives the boxes a chance to be moved further in single iterations, gives the (1 + 1) algorithm a chance to find an open spots for boxes at a greater distance. After the *MutationStrength* reaches about 120 the benefits level off, likely because this is about the size of a typical box and this value allows two stacked boxes to clear their overlap in a single or only a few iterations, which is important for the (1 + 1) algorithm to not get stuck in a local optimum.

The *MutationStrength* setting did not have a significant impact on the performance of the other algorithms as long as it wasn't set very low. For example, in table 3.2 it is shown that the *MutationStrength* doesn't affect the average scores of the SA algorithm significantly. This is happening because SA and ASO have methods of getting out of local optimums that (1 + 1) doesn't possess and don't rely as much on the *MutationStrength* to get boxes out of overlap situations in a single iteration.

The *MutationStrength* was set to 120 for all algorithms because (1 + 1) performs well with this value and tests for other algorithms indicated that this value was nearly optimal with negligible score differences when compared to other settings.

Mutation Strength	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
2	(1 + 1)	0.8953	825.0890	0.0101	15.0308
6	(1 + 1)	0.9100	804.7290	0.0110	7.8236
10	(1 + 1)	0.9253	778.6195	0.0058	5.1772
20	(1 + 1)	0.9417	758.8895	0.0031	19.1529
40	(1 + 1)	0.9652	746.6085	0.0022	7.5953
60	(1 + 1)	0.9682	710.8520	0.0042	15.9976
80	(1 + 1)	0.9733	673.1465	0.0020	6.1218
100	(1 + 1)	0.9730	656.0125	0.0042	42.7201
120	(1 + 1)	**0.9779**	654.5110	0.0017	16.0614
140	(1 + 1)	0.9767	627.5795	0.0013	19.8004
160	(1 + 1)	0.9774	625.1855	0.0038	23.6833
180	(1 + 1)	0.9763	617.2555	0.0031	30.2608
200	(1 + 1)	0.9753	**609.1350**	0.0030	17.4814

Table 3.1: (1 + 1) *MutationStrength* tuning results.

Mutation Strength	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
60	SA	**0.9936**	161.7775	0.0003	10.5311
80	SA	0.9929	171.2265	0.0005	4.0495
100	SA	0.9927	170.4780	0.0004	3.4135
120	SA	0.9926	165.9720	0.0010	15.2911
140	SA	0.9925	164.6915	0.0002	11.2986
160	SA	0.9934	154.0270	0.0003	5.7332
180	SA	0.9914	170.5020	0.0008	3.0724
200	SA	0.9922	**152.7825**	0.0009	9.1852
220	SA	0.9922	160.1535	0.0004	7.0765

Table 3.2: SA Mutation Strength results.

3.4.2 Simulated Annealing

The Simulated Annealing algorithm used the same mutation function and mutation setting as the (1 + 1) algorithm, but had two remaining parameters to be tuned. A limitation of the software setup for this paper is that there is no built-in features for tuning parameters, such as a grid search. Instead, the JSON Settings feature was used to load a set of 28 possible parameter settings which would then be tested and evaluated. For these tests a grid was set up with the initial temperature T^0 ranging from 10 to 1000 and the alpha α from 0.95 to 0.9999. The results can be found in table 3.3.

Settings	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
T^0=10, α=0.95	SA	0.9775	256.3965	0.0025	10.9300
T^0=10, α=0.98	SA	0.9783	616.1120	0.0010	21.6589
T^0=10, α=0.99	SA	0.9803	572.0150	0.0016	21.7774
T^0=10, α=0.995	SA	0.9796	543.2370	0.0017	11.8089
T^0=10, α=0.999	SA	0.9846	477.0230	0.0007	12.0303
T^0=10, α=0.9995	SA	0.9835	470.4265	0.0022	3.3288
T^0=10, α=0.9999	SA	0.9855	449.5495	0.0011	14.2582
T^0=50, α=0.95	SA	0.9816	263.9080	0.0012	3.4096
T^0=50, α=0.98	SA	0.9845	465.8840	0.0021	33.5981
T^0=50, α=0.99	SA	0.9877	417.9565	0.0011	12.4291
T^0=50, α=0.995	SA	0.9840	487.3200	0.0025	16.8099
T^0=50, α=0.999	SA	0.9922	229.7510	0.0003	17.5061
T^0=50, α=0.9995	SA	0.9924	206.5340	0.0001	15.1319
T^0=50, α=0.9999	SA	0.9923	205.0670	0.0003	11.7781
T^0=100, α=0.95	SA	0.9867	242.1730	0.0015	19.2769
T^0=100, α=0.98	SA	0.9908	368.6445	0.0003	12.9012
T^0=100, α=0.99	SA	0.9917	300.5980	0.0004	21.6665
T^0=100, α=0.995	SA	0.9929	214.6365	0.0001	24.5172
T^0=100, α=0.999	SA	0.9924	180.5900	0.0001	9.5942
T^0=100, α=0.9995	SA	0.9919	173.7220	0.0001	6.6194
T^0=1000, α=0.95	SA	**0.9930**	**114.6395**	0.0001	3.7577
T^0=1000, α=0.98	SA	0.9927	192.8320	0.0001	10.7297
T^0=1000, α=0.99	SA	0.9917	184.8940	0.0001	1.6709
T^0=1000, α=0.995	SA	0.9883	168.3985	0.0005	2.9719
T^0=1000, α=0.999	SA	0.9692	177.3190	0.0004	1.9492
T^0=1000, α=0.9995	SA	0.9660	171.2790	0.0011	8.6786
T^0=1000, α=0.9999	SA	0.9605	184.1825	0.0017	6.9552

Table 3.3: SA parameter tuning results.

For both score and time performance, the standout parameter combination was T^0=1000, α=0.95. The final set of parameter setting for the SA algorithm can be seen in table 3.4.

Parameter	Setting
$MutationStrength$	120
T^0	1000
α	0.95

Table 3.4: SA parameter settings.

3.4.3 Particle Swarm Optimization

The PSO algorithm had five parameters that were to be tuned. Given the particular set of features in the dashboard application, it was decided that it was best to do this in multiple steps. First, tests were done with typical PSO settings to find an adequate initial velocity (v) setting, which turned out to be v=12. Then a parameter grid search was created using v=12, pop ranging from 50 to 1000 and all three r parameters ranging from 1.5 to 4 (appendix table 6.3). Then with pop=400 and r=2 the initial velocity was re-evaluated (appendix table 6.4) and was changed from v=12 to v=16. Finally, the $r1$, $r2$ and $r3$ parameters were tuned individually (appendix table 6.5). The final set of parameter setting for the PSO algorithm can be seen in table 3.5.

Parameter	Setting
pop	400
v	12
$r1$	1.75
$r2$	2
$r3$	2.25

Table 3.5: PSO parameter settings.

3.4.4 Adaptive Swarm Optimization with Propagation

The ASO algorithm without propagation only uses the Mutation Strength parameter, so we can move on straight to ASOP tuning. The ASOP algorithm was tested with a variety of propagation rate (p) settings, ranging from 0.01 to 1 (table 3.6). It turns out that both low and high values don't perform as well, but any propagation rate value performs better than ASO without propagation when the results from table 3.6 are compared to the ASO score averages in the upcoming results section (4.1). The final set of parameter setting for the ASOP algorithm can be seen in table 3.7.

Settings	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
p=0.01	ASOP	0.9881	337.0665	0.0015	24.1271
p=0.05	ASOP	0.9893	320.7745	0.0009	18.9414
p=0.1	ASOP	0.9896	306.4795	0.0011	25.3127
p=0.2	ASOP	0.9900	297.7325	0.0004	17.5444
p=0.3	ASOP	0.9904	289.0285	0.0005	9.1029
p=0.4	ASOP	**0.9911**	**271.8485**	0.0004	24.9294
p=0.5	ASOP	0.9901	281.3920	0.0015	37.8935
p=0.6	ASOP	0.9907	278.4790	0.0003	11.8073
p=0.7	ASOP	0.9887	308.2250	0.0014	18.2850
p=0.8	ASOP	0.9906	281.8345	0.0004	16.9461
p=0.9	ASOP	0.9875	318.7885	0.0027	31.5536
p=1	ASOP	0.9851	328.5670	0.0043	37.8053

Table 3.6: ASOP parameter tuning results.

Parameter	Setting
$MutationStrength$	120
p	0.4

Table 3.7: ASOP parameter settings.

4 Results

4.1 Algorithm performance evaluation

Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
(1 + 1)	0.9747	623.8025	0.0035	29.1732
SA	**0.9929**	**75.8795**	0.0001	3.1454
PSO	0.9904	421.9971	0.0013	51.6700
ASO	0.9858	348.7890	0.0016	28.1880
ASOP	0.9906	275.3725	0.0006	14.1137

Table 4.1: Final score and time averages after 30 iterations of 50 incremental box placements.

Table 4.1 displays the final score and time averages after running each algorithm 30 times using the settings discussed in section 3.4 Parameter settings. As expected, the (1 + 1) performed the worst in both score and time averages. SA and ASOP both performed very well with high average score and SA is outstanding when it comes to time performance. Using these metrics the SA algorithm definitely comes out as the best choice to tackle the Incremental Box Placement problem, but it does have a downside that is not directly represented in this data. We will discuss these kinds of characteristics and behaviors of each algorithm in the next sections.

4.1.1 (1+1)-Evolutionary Algorithm

Given the simplistic nature of the (1 + 1) algorithm, it actually performed surprisingly well when there is a low to medium number of boxes on the board. This can be seen in figure 4.1. Each box was incrementally placed on the target marker represented by the red circle and the other boxes were neatly moved out of the way without many unnecessary gaps and with zero overlap. In rare cases it even achieves higher scores than the SA and PSO algorithms when there are only a few boxes on the board, because it is less likely to make moves that don't result in a direct score increase and is therefore unlikely to introduce unnecessary displacement.

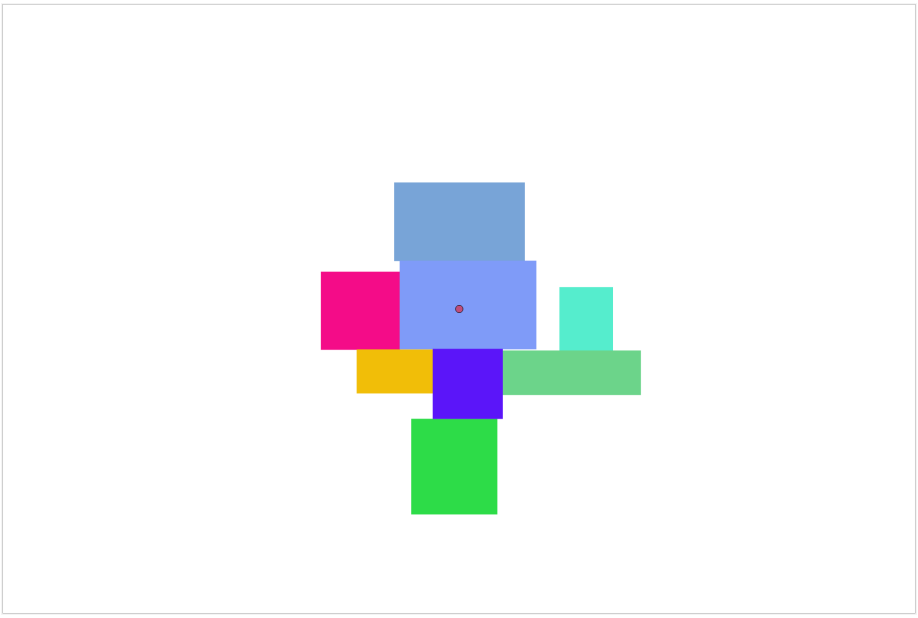

Figure 4.1: 8th box added with (1 + 1)

Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
(1 + 1)	0.9898	143.6617	0.0023	26.3612
SA	**0.9920**	**37.6550**	0.0002	2.2527
PSO	0.9908	152.8856	0.0013	18.3380
ASO	0.9913	157.3378	0.0006	16.4174
ASOP	**0.9920**	147.0128	0.0002	10.0474

Table 4.2: Only 18 incremental box placements.

As soon as there are around 20 boxes on the board the performance of the (1 + 1) algorithm usually starts deteriorating rapidly. Table 4.2 shows that score-wise the (1 + 1) algorithm can actually keep up quite well up to the 18 boxes mark. The average score is moderately lower than other algorithms, and the standard deviation is significantly higher. The standard deviation is high because in some iterations it already gets into situations where it can't solve the box overlap problem properly. This only gets worse from here on out for the (1 + 1) algorithm as it incrementally adds additional boxes. This is also why the standard deviation is high in table 4.1, it can get completely stuck in a local optimum, but this doesn't always happen.

A typical box overlap problem caused by a local optimum is shown in figure 4.2. As additional boxes are placed in the center of the screen the (1 + 1) algorithm doesn't have a mechanism to create sufficient space to avoid overlap. The cause of this problem can be explained by its mutation function. During each mutation iteration only a single box is moved. If this box happens to be one of the stacked boxes in the center you'd think that moving it would likely decrease overall overlap, but this isn't always the case. Since the (1 + 1) algorithm tends to place all boxes so tightly next to each other it gets into a situation where moving a stacked box in the center in any direction just creates new overlap and therefore doesn't improve the scoring.

If one of the boxes on the outside is moved instead it won't help with reducing overlap, since that mostly occurs in the center, so it will only increase displacement, which in turn decreases the score. Moving the boxes on the outside is actually necessary to create space for the stacked boxes in the center, but since only one box is moved in each iteration there is no reward mechanism for creating space. What ends up happening is that more and more boxes are stacked in the center with no way out. This problem can be delayed with a higher *MutationStrength* setting, as it allows the center boxes to simply jump over the outside boxes in a single iteration and thereby reduce overall overlap. This is why (1 + 1) responds so well to higher *MutationStrength* in section 3.4 Parameter settings, but this only works up to a certain point, as high *MutationStrength* starts negatively affecting displacement scores.

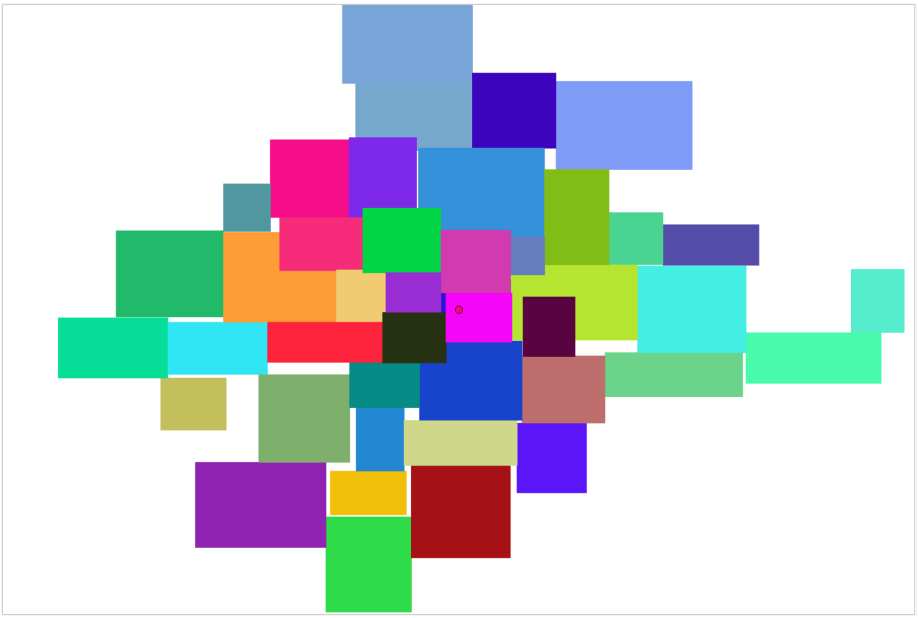

Figure 4.2: 43rd box added with (1 + 1)

4.1.2 Simulated Annealing

The SA algorithm performs the best out of all algorithms based on both the average score and average time metrics. The standard deviations are also very low. The algorithm is simply consistently good at keeping the overlap at zero and at the same time moving the boxes very efficiently to keep displacement low. Its ability to solve the overlap problems so well can be explained by contrasting it to the mutation issues of the (1 + 1) algorithm. The (1 + 1) algorithm would work itself into configurations where any movement of the stacked center boxes would not result in overlap improvements. The SA algorithm, however, tends to make small adjustments to surrounding boxes even when they don't immediately cause a score improvement and those tiny gaps then create score improvements when the stacked boxes are moved, which then cascades outwards and creates space for stacked boxes.

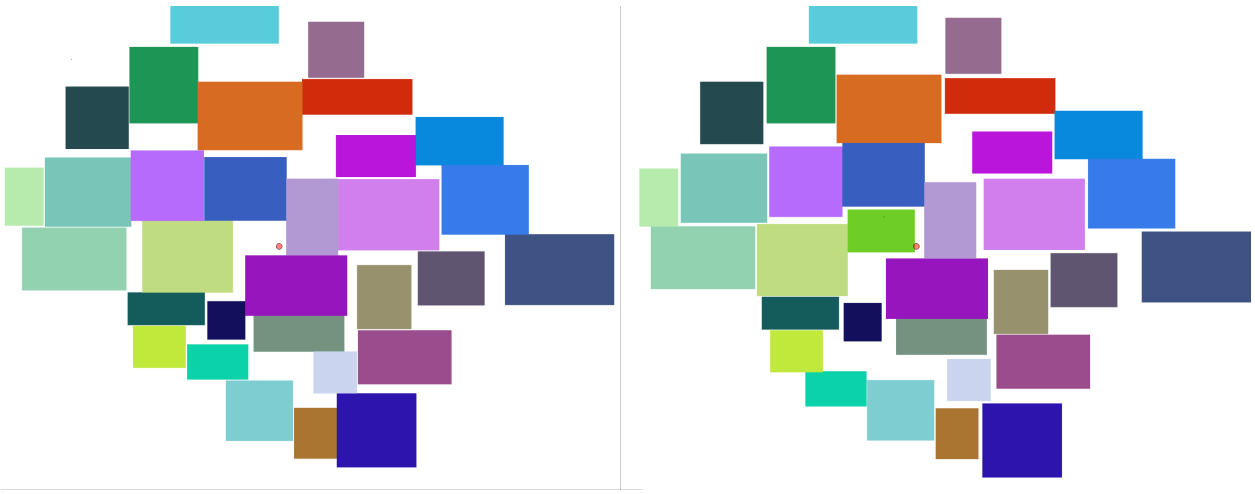

Figure 4.3: 31 boxes on the left and then 32 boxes on the right.

While this mechanism usually works fantastically, it does sometimes create a new problem due to unnecessary displacement. This scenario is illustrated in figure 4.3. When the 32nd box was placed many of the boxes were moved in an outward direction further than was necessary to make space in the center for the newest box.

This only seems to happen when the algorithm and struggles with finding a solution for the overlap for longer than usual. During this time additional outward movement of the boxes occurs and then when the overlap is solved not enough iterations remain to bring back the boxes to more optimal displacement locations.

It is reasonable to assume that this problem would have a noticeably negative affect on average scoring for SA when compared to other algorithms, but this is not the case. The unnecessary displacement tends to only be a few pixels for each box and therefore doesn't impact the displacement score significantly. This characteristic of SA can be best described as "incidental noise". There is currently no scoring metric used in this paper that specifically penalizes noise.

For certain use cases such as user interfaces, this potentially creates a disconnect between the jarring visual effect of seeing many boxes move slightly in random directions and the scoring methods used in this paper. This is the only reason that was found that could justify choosing another algorithm such as ASOP over SA in certain use cases, since ASOP doesn't suffer from this problem to the same extent.

The main proponent for using SA is then still the incredibly low average time spent when compared to the other algorithms. There are two main reasons for this. Firstly, the mutation and evaluation components of SA are computationally cheap when compared to PSO and ASOP. Where PSO needs to calculate and execute velocities and ASOP evaluates and ranks every single box, SA just needs to keep track of the temperature and uses the most basic mutation function that is also used in (1 + 1). Secondly, because of the efficiency of SA in solving overlap issues it is much more likely to reach the Accept Criteria score of 0.995 that was used in this paper.

The time efficiency of SA opens the door to solving the "incidental noise" problem with an adjustment of parameters instead of using a different algorithm to avoid it. As explained earlier, the noise seems to occur

when it expended most of its computational allowance finding a solution for overlap and not having enough iterations remaining for optimizing the displacement thereafter. This means that simply increasing the Max Configuration setting specifically for SA might resolve this issue. And this would be fine from a time spent perspective, because SA would probably still be the fastest algorithm even if you significantly increase the number of Max Configurations for the algorithm.

4.1.3 Particle Swarm Optimization

PSO is the most distinctive algorithm in this list, since it doesn't use the (1 + 1) mutation function. Still, it performs quite similarly score-wise. The average time spent computing is far worse than both SA (456.14% slower) and ASOP (53.25% slower), but after SA it is the next best algorithm for average score together with ASOP.

When you take a closer look at the box configurations after each incremental addition more contrasts begin to appear, however. The most stark one being the way that PSO displaces boxes to solve overlap problems. With SA we saw a mechanism of creating space for new boxes by moving them away from the center of the board, but PSO tends to handle this differently. In figure 4.4 you can see how instead of incrementally creating space, PSO decides to move the box marked as 1 all the way to the outside to make room for the new box marked as 2.

The new box marked as 2 leaves a lot of empty space surrounding it, so slightly moving the surrounding boxes to make room instead could have resulted in a better displacement score in this scenario. This seems to be the main reason why PSO lags behind SA in overall scoring. There are, however, also instances where the PSO method of creating space for new boxes wins out.

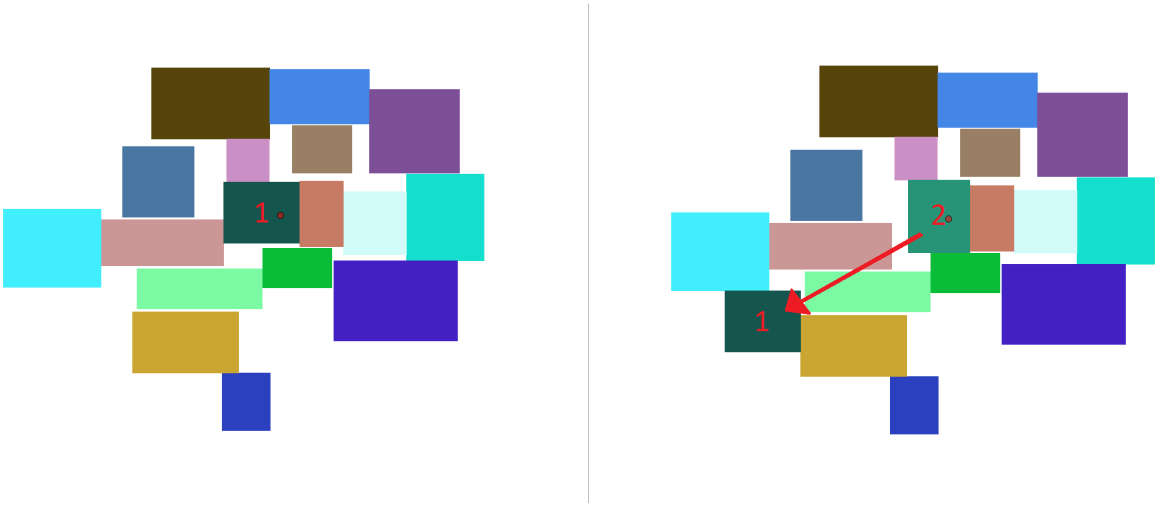

Figure 4.4: PSO: 17 boxes on the left and then 18 boxes on the right.

Box Number	Displacement Score	Overlap Score	Overall Score
1	1	1	1
2	0.8091	1	0.9826
3	0.8283	1	0.9844
4	0.8825	1	0.9893
5	0.9118	1	0.9920
6	0.8949	1	0.9904
7	0.8711	1	0.9883
8	0.8474	1	0.9861
9	0.8796	1	0.9891
10	0.9474	1	0.9952
11	**0.01**	1	**0.91**
12	**1**	1	**1**
13	0.9770	1	0.9979
14	0.9358	1	0.9942

Table 4.3: PSO scoring details for the extreme displacement scenario.

There is a more significant consistency issue with the PSO algorithm that needs to be discussed, which is displayed in figure 4.5. Sometimes the PSO implementation suddenly creates an extreme amount of displacement for most or all of the boxes, which can be seen in the transition from 10 boxes on the left to 11 boxes on the right in the image. Table 4.3 shows the scoring details for each incrementation. In the table you can see that when box 11 was placed the resulting displacement score was 0.01, which is the minimum score, but the overlap score was 1. Because of the 10 to 1 weighting of overlap to displacement this results in an overall score of 0.91 for the addition of box 11. This is a very bad score, so one could wonder why this configuration was chosen and how PSO still manages to get high overall averages.

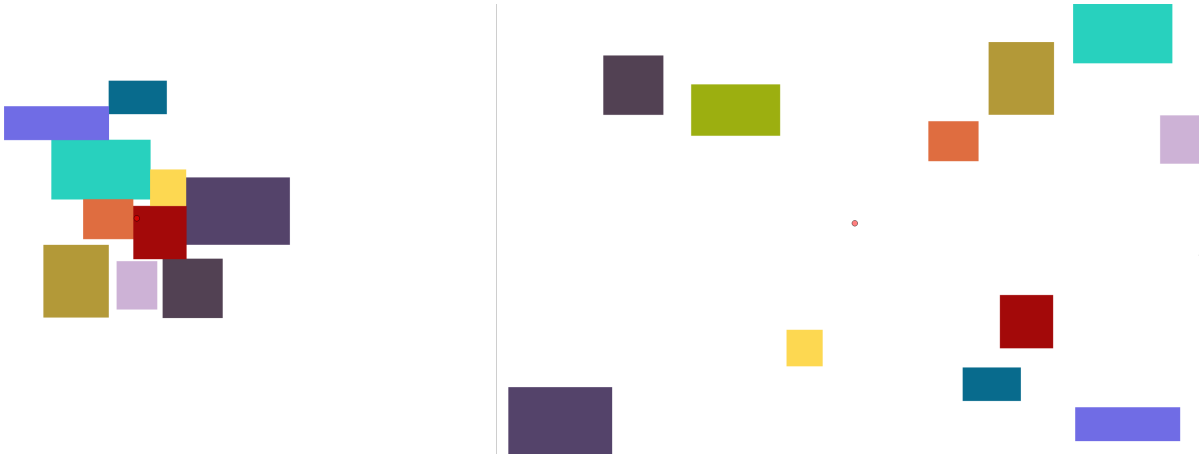

Figure 4.5: PSO: 10 boxes on the left and then 11 boxes on the right.

What is likely happening is that sometimes due to pure randomness none of the particles in the PSO's

population were assigned velocities that were able to properly solve the overlap problem in the center. There is a limited population and a limited number of maximum configurations. In fact, where (1 + 1) has 40,000 maximum configurations, for PSO that number is divided by its population to find the number of maximum mutations for each particle, which in this case would be only 100 mutations. For example, in the referenced image the next box would create overlap with 5 different boxes and each particle in the PSO's population only has 100 velocity-based mutations to create a solution. Since overlap is highly weighted compared to displacement the PSO seems to choose a particle in the population that still gets a 1 as its overlap score even though it obtained that by exploding all the boxes outward way too far. One could suggest increasing the number of maximum configurations per particle to solve this problem, but that would take the PSO out of line with the other algorithms and the PSO is already comparatively slow as is.

The next part is interesting, because it shows how the PSO "cheats" the scoring system to still obtain a good overall score. Table 4.3 shows that right after getting a displacement score of 0.01 for box 11 it receives a perfect score of 1 when box 12 is placed. This is because displacement is always calculated only compared to the previous configuration of boxes. When box 12 was placed none of the boxes in configuration 11 had to move to make room, so it received a perfect score for both displacement and overlap. This "benefit" from the one rogue configuration with a score of 0.01 then continues as many of the boxes in the corners are "out of the game" for the remainder of the incremental box placements. Figure 4.6 shows that after 38 boxes are placed those boxes that were pushed all the way into the corner during the placement of box 11 still remain in the exact same positions in the corners and make it easier for PSO to score well on overlap and displacement since they don't interact with any other boxes.

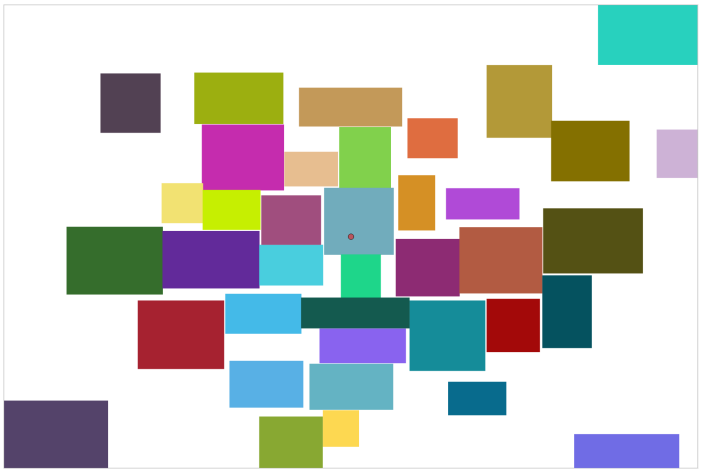

Figure 4.6: PSO: 38 boxes.

In conclusion, even though the PSO algorithm scores quite well overall, one could reasonably decide not to use it because of this inconsistency issue and the fact that this issue actually helps the PSO algorithm "cheat" the scoring system to get better scores thereafter without penalty.

4.1.4 Adaptive Structure Optimization

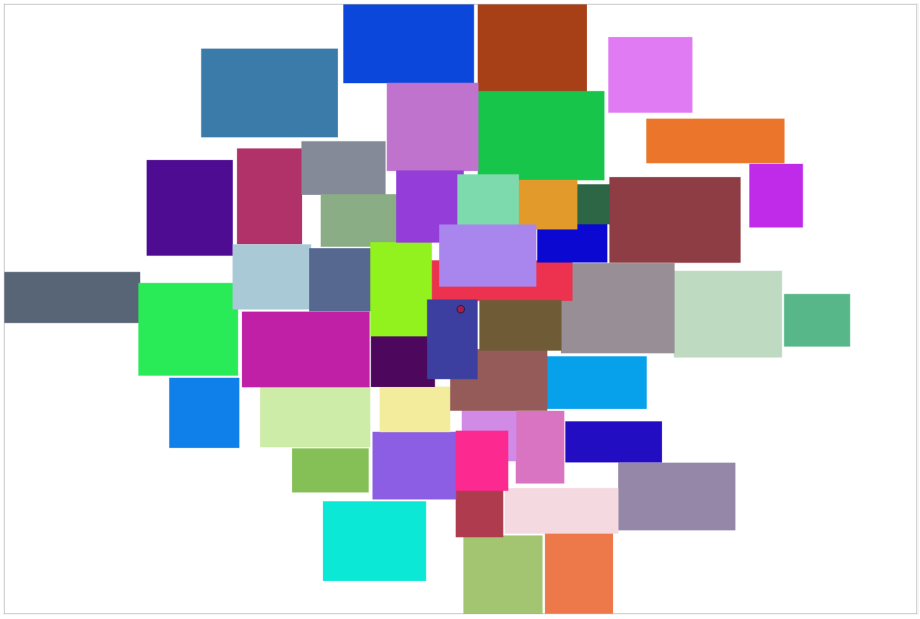

Figure 4.7: 31st box added with SA

The ASO algorithm runs into the same problems as the (1 + 1) algorithm. Like (1 + 1) it does not have a mechanism to create space for stacking boxes in the center of the board, and therefore it also gets stuck in similar local optimums. This is why the average score of 0.9858 is low when compared to SA, PSO and ASOP. It is, however, faster than (1 + 1) at getting to those local optimums and makes better use of its computational allowance. This is because of its targeting system, which does work well, as can be seen by the significant score and time improvements. From 0.9747 to 0.9858 in score and from 623.8 ms to 348.8 ms in time when compared to (1 + 1). We will now evaluate the ASOP's performance to see if adding a propagation mechanism can improve the performance further.

4.1.5 Adaptive Structure Optimization with Propagation

The ASOP algorithm is the second-best performer in both the average score (0.9906) and average time (275.3725 ms) categories. The standard deviations are also comparatively low as ASOP is providing good results consistently. Adding the propagation mechanism to ASO improved the average score by 0.0048 and closed the gap to SA's performance by 67.61%. On top of that, it made the ASO algorithm 21.05% faster. Compared to (1 + 1) the ASOP algorithm is 55.86% faster and produces significantly better results as it isn't running into the same local optimum problems. The propagation mechanism provides ASO with a way to efficiently move boxes away from the target location.

While it is difficult to visually notice the difference in average score from SA's 0.9929 to ASOP's 0.9906 in practice, the difference in average time spent is still considerable. With an average of 75.8795 ms the SA algorithm is 72.44% faster than the OSAP algorithm. A redeeming factor of the ASOP algorithm is that it

doesn't suffer from the same "incidental noise" issues as SA or inconsistency issues as PSO. So it can still be an attractive option for certain applications when minimal computational time spent isn't as important.

Figure 4.8 shows how ASOP was able to place 31 boxes on the board with minimal overlap using the parameter settings that were listed in this paper.

Figure 4.8: 31st box added with SA

5 Conclusions

In this paper the IBP problem was defined, common metaheuristics were implemented, a tailor-made algorithm was developed and a web application for testing and visualization was designed.

Each algorithm also had its parameters tuned, through which its performances dramatically improved. The (1 + 1)-Evolutionary algorithm was initially only tested with a low *MutationStrength* settings as it was built, so when it was properly tuned, the performance increased significantly because of how it helped it escape the stacking of boxes on the target location in a single iteration. As such, tuning the algorithms also gave a lot of insight into their behavior, which in turn was instrumental for evaluating the quality of the performance metrics.

The best algorithm for the IBP problem when judging it purely by the performance metrics is undoubtably the SA algorithm by a wide margin. It is incredibly fast and effective. The only downside is the "incidental noise" phenomenon, but this did not significantly affect its average score. Which brings up a good point about the nature of our performance metrics.

The scoring systems that were designed did not account properly for some potentially unwanted effects, so the algorithms were not incentivized to avoid these issues. Especially the behavior displayed by SA and PSO was sometimes unexpected, but could be explained by the way the scores are calculated. In future research, it might be beneficial to enhance the scoring system if potential applications of the algorithms call for stricter evaluation of the algorithms' behaviors.

The custom-built ASO algorithm shows that performance is improved when the boxes that negatively impact the overall score the most are more likely to be mutated. However, compared to common metaheuristics like our implementations of SA and PSO it still couldn't compete. The addition of the propagation mechanism in ASOP again significantly improved the performance in both time and score. The fact that each feature that was added to the algorithm led to increased performance does inspire confidence that in future work additional mechanisms could be added to the ASOP algorithm, such a context-based guideline for steering mutation strength and direction in the mutation function. This could improve performance further and maybe even beat our SA implementation eventually.

6 Appendix

6.1 Box sizes

#	width (px)	height (px)
1	237	143
2	97	116
3	156	174
4	144	142
5	139	80
6	251	81
7	248	161
8	127	127
9	120	96
10	86	87
11	213	123
12	246	93
13	200	110
14	88	132
15	181	168
16	196	157
17	152	123
18	124	146
19	206	83
20	175	75
21	99	96
22	153	137

Table 6.1: Part 1/2 - Box sizes in pixels.

#	width (px)	height (px)
23	198	158
24	182	96
25	238	156
26	230	162
27	206	164
28	166	160
29	187	145
30	232	137
31	118	174
32	153	97
33	138	96
34	128	83
35	143	118
36	95	109
37	176	112
38	259	74
39	127	70
40	116	93
41	127	115
42	101	73
43	121	90
44	112	171
45	123	131
46	112	94
47	149	92
48	92	145
49	176	113
50	141	124

Table 6.2: Part 2/2 - Box sizes in pixels.

6.2 PSO tuning

Settings	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
Pop=50, r=1.5	PSO	0.9259	374.1180	0.0104	24.7914
Pop=100, r=1.5	PSO	0.9122	404.8900	0.0024	7.1792
Pop=200, r=1.5	PSO	0.9200	402.0240	0.0015	2.1447
Pop=400, r=1.5	PSO	0.9222	409.8765	0.0066	11.4537
Pop=600, r=1.5	PSO	0.9488	363.2960	0.0213	42.0034
Pop=800, r=1.5	PSO	0.9527	**363.2955**	0.0203	45.3755
Pop=1000, r=1.5	PSO	0.9682	339.7120	0.0171	33.2257
Pop=50, r=2	PSO	0.9756	450.2240	0.0047	57.8919
Pop=100, r=2	PSO	0.9800	490.7600	0.0028	40.2912
Pop=200, r=2	PSO	0.9848	438.2315	0.0030	71.4828
Pop=400, r=2	PSO	**0.9876**	455.8860	0.0019	43.4731
Pop=600, r=2	PSO	0.9864	477.8880	0.0017	12.7644
Pop=800, r=2	PSO	0.9865	465.5995	0.0028	50.2823
Pop=1000, r=2	PSO	0.9861	461.7365	0.0013	30.7302
Pop=50, r=3	PSO	0.9558	570.2100	0.0087	109.2857
Pop=100, r=3	PSO	0.9631	587.9085	0.0056	60.4825
Pop=200, r=3	PSO	0.9557	723.8565	0.0039	58.3310
Pop=400, r=3	PSO	0.9668	660.0765	0.0079	65.7644
Pop=600, r=3	PSO	0.9596	744.8215	0.0077	92.1406
Pop=800, r=3	PSO	0.9658	715.6025	0.0101	102.3794
Pop=1000, r=3	PSO	0.9695	694.0035	0.0037	24.0244
Pop=50, r=4	PSO	0.9212	655.3785	0.0062	23.1158
Pop=100, r=4	PSO	0.9344	670.7375	0.0041	11.3436
Pop=200, r=4	PSO	0.9421	688.6540	0.0062	29.7865
Pop=400, r=4	PSO	0.9464	720.7975	0.0018	33.5237
Pop=600, r=4	PSO	0.9510	703.5680	0.0039	14.7853
Pop=800, r=4	PSO	0.9449	740.6355	0.0032	14.2323
Pop=1000, r=4	PSO	0.9496	727.6440	0.0057	23.8037

Table 6.3: PSO parameter tuning results - Pop, r.

Settings	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
v=2	PSO	0.9851	475.5360	0.0014	51.7717
v=4	PSO	0.9854	471.8390	0.0020	41.3134
v=8	PSO	0.9850	479.4810	0.0028	53.7598
v=12	PSO	0.9851	489.8440	0.0018	43.0161
v=16	PSO	**0.9867**	**437.1235**	0.0012	30.2762
v=20	PSO	0.9847	492.8725	0.0028	63.2469
v=24	PSO	0.9838	508.1420	0.0028	56.9650
v=28	PSO	0.9846	489.0735	0.0024	41.9456
v=32	PSO	0.9847	522.2940	0.0019	21.6718

Table 6.4: PSO parameter tuning results - Initial velocity

Settings	Algorithm	AVG Score	AVG Time	STDEV Score	STDEV Time
r1=2, r2=2, r3=2	PSO	0.9872	431.3730	0.0022	47.7166
r1=1.75, r2=2, r3=2.25	PSO	**0.9891**	450.2295	0.0012	39.9887
r1=2, r2=1.75, r3=2.25	PSO	0.9862	457.2400	0.0029	65.5590
r1=2, r2=2.25, r3=1.75	PSO	0.9888	**386.8060**	0.0015	41.7530
r1=2.25, r2=2, r3=1.75	PSO	0.9851	413.8900	0.0028	50.4158
r1=2.25, r2=1.75, r3=2	PSO	0.9825	473.2810	0.0032	59.3896
r1=1.75, r2=2.25, r3=2	PSO	0.9889	433.5640	0.0010	42.0128

Table 6.5: PSO parameter tuning results - r1, r2, r3

6.3 Source Code Fragments

```
1   function pickRandomBoxBasedOnScores (boxes, totalScoreSum, highestScore) {
2       var totalScoreWithBonusses = totalScoreSum + highestScore;
3       var bonusScore = highestScore / boxes.length;
4
5       var randomNumber = this.randomNumberBetween(0, totalScoreWithBonusses);
6
7       var box, currentTotal = 0;
8
9       for (var a = 0; a < boxes.length; a++) {
10          box = boxes[a];
11
12          currentTotal += box.grade + bonusScore;
13
14          if (randomNumber <= currentTotal) {
15              return a;
16          }
```

```
            }
        }
```

Listing 6.1: Pick random box based on scores

```
        function calculateOverlapScore (boxes, maxOverlap) {
            var totalOverlap = 0, totalForBox, box1, box2;

            for (var a = 0; a < boxes.length; a++) {
                box1 = boxes[a];
                totalForBox = 0;

                for (var b = a + 1; b < boxes.length; b++) {
                    box2 = boxes[b];

                    if (box1.x2 >= box2.x1 && box1.x1 <= box2.x2 && box1.y2 >= box2.y1 && box1
                        .y1 <= box2.y2) {
                        xOverlap = Math.max(0, Math.min(box1.x2, box2.x2) - Math.max(box1.x1,
                            box2.x1));
                        yOverlap = Math.max(0, Math.min(box1.y2, box2.y2) - Math.max(box1.y1,
                            box2.y1));
                        totalForBox += xOverlap * yOverlap;
                    }
                }

                totalOverlap += totalForBox;
            }

            return 1 - (totalOverlap / maxOverlap);
        }
```

Listing 6.2: Overlap score calculation

```
        function calculateDisplacementScore (currentBoxes, newBoxes, maxDisplacement) {
            var totalDisplacement = 0;

            for (var i = 0; i < currentBoxes.length; i++) {
                var currentBox = currentBoxes[i],
                    newBox = newBoxes[i],
                    displX = currentBox.x1 - newBox.x1,
                    displY = currentBox.y1 - newBox.y1;
                totalDisplacement += Math.sqrt(displX * displX + displY * displY);
            }

            return 1 - (totalDisplacement / (maxDisplacement));
        }
```

Listing 6.3: Displacement score calculation

Bibliography

[1] Obrad Babica, Milica Kalica, Danica Babica and Slavica Dozica. "The airline schedule optimization model: validation and sensitivity analysis". In: *Procedia - Social and Behavioral Sciences* volume 20 (2011), pp. 1029–2040.

[2] Ozan Erdinc. *Optimization in Renewable Energy Systems*. Elsevier Science, 2017.

[3] Michael R. Garey and David S. Johnson. *Computers and Interactibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman Company, 1979.

[4] Stefan Droste, Thomas Jansen and Ingo Wegener. "On the analysis of the (1 + 1) evolutionary algorithm". In: *Theoretical Computer Science* 267 (2002), pp. 51–81.

[5] S. Kirkpatrick, C. D. Gelatt Jr.1 and M. P. Vecchi2. "Optimization by Simulated Annealing". In: *Science* 220 (1983), pp. 671–680.

[6] Asep Maulana, Marios Kefalas and Michael T. M. Emmerich. "Immunization of networks using genetic algorithms and multiobjective metaheuristics". In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)* (2017), pp. 1–8.

[7] J. Kennedy. "The particle swarm: social adaptation of knowledge". In: *Proceedings of IEEE International Conference on Evolutionary Computation 1997.* (1997), pp. 303–308.

[8] Kay Thielemann, Martin Maier and Philipp Steibler. "Adaptive Structural Optimization of Technical Fibre Composites Under Consideration of Bionic Aspects - Industrial Applications". In: *11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* (2006), p. 7043.

[9] Danica Pavlovic. "Sensitivity analysis of airline schedule optimization (ASO) advanced model". In: *Journal of Air Transport Studies* (2010), pp. 1791–6771.

[10] J. Sauppe, ed. *Mathematical Programming Glossary*. Originally authored by Harvey J. Greenberg, 1999-2006. `http://glossary.computing.society.informs.org`: INFORMS Computing Society, 2006–18.

[11] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. Wiley, 1995.