Universiteit Leiden
The Netherlands

Opleiding Informatica

The Greeting Problem

How to have people greet one another in a rectangular room

Branco Adelaar

Supervisors:
Walter Kosters & Hendrik Jan Hoogeboom

BACHELOR THESIS

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

29/7/2024

Abstract

We have researched the so-called Greeting Problem, that is concerned with a rectangular room filled with agents (one per square), where we want all agents to move past all others. Using a brute force search algorithm and manual experimentation, we have found efficient, reliably scaleable, understandable new solutions for several room sizes. These are the Expanding Loop Algorithm for rooms of height 2, which relies on wallflowers and growing and shrinking loops, the Butterfly Loop Algorithm for rooms of height 3, which relies on loops covering the full upper rows of the room followed by loops covering the full lower rows of the room, and the Stacking Loop Algorithm for rooms of height 4, which relies on a specific sequence of loop sets to move the various rows past each other.

Contents

1 Introduction

Consider a busy room after a party that is just about winding down, filled with people who really do not want to get in anyone's personal space while at the same time really wanting to say goodbye to everyone at the party. Since the room is only so large, and as such there is limited space between people to move, the process of having everyone greet each other becomes tricky. If the party wants to get home in a timely manner, they cannot just wander around what little walking space available to them without direction.

What the people need, is a coordinated plan, a strategy for the group to move through the room in a way that minimises time spent moving around each other pointlessly.

This is, in essence, what the *Greeting Problem* is: a pathing problem concerned with moving all agents in a space to be adjacent to each other at least once. It has as its ultimate goal finding the quickest and most efficient movement algorithms possible, reducing the number of steps required to perform all greetings.

1	2	3	4	5
	6	7	8	9
10	11	12	13	14

Figure 1: Example of a room of size 3 by 5, with a single empty space.

In previous research [BBCM12] into the Greeting Problem, a number of algorithms were found. These algorithms, generic algorithms primarily dependant on the number of empty spaces in the room, all rely on a single track that all agents in the room follow.

Our goal is to find more efficient algorithms to solve the Greeting Problem. For the scope of this paper, we are limiting our research to crowded rooms, rooms with only a single empty space (see Figure 1.

We want the algorithms we find to be scaleable for any width of the room, and to be relatively straightforward for a human to understand. In Figure 2 a potential solution is shown, the arrows representing the path the empty space follows moving through the room, in this case traversing through a loop through the upper two rows, followed by a loop through the lower two rows.

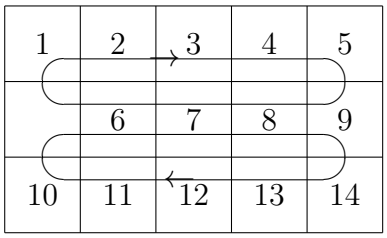

Figure 2: The same 3 by 5 room, overlaid with a template for a potential solution.

In Chapter 2 we will establish the mechanics of the Greeting Problem, and explain the terminology to be used throughout the paper. In Chapter 3 we will examine the existing research, and establish

the upper and lower bounds of our experiments. In Chapter 4 we will propose the methods used to perform our experiments. In Chapter 5 we will examine the results of the experiments, explaining and proving the algorithms we have found. In Chapter 6 we will conclude our paper, discussing our results and proposing potential future research to be done on the Greeting Problem.

This research is a bachelor project at Leiden University (LIACS) and is supervised by Walter Kosters and Hendrik Jan Hoogeboom.

2 Greeting Problem

The Greeting Problem involves a rectangular room of M by N discrete spaces, filled with a number of agents, with all other spaces empty. An agent occupies exactly one space and can only move into an adjacent empty space. The goal is to have all agents greet all other agents. A greeting is made when two agents are vertically or horizontally adjacent to each other.
In this paper, we are using a number of terms to refer to aspects of the Greeting Problem. The following definitions are used:

- A *room* is a two dimensional array, of size M by N, where each array element can hold either an agent or an empty space.
- An *agent* is an element in a room, indicated by a unique number.
- An *empty space* is an element in a room, indicated by XX.
- An agent can *move* through the room by swapping places with an empty space that is vertically or horizontally adjacent to that agent.
- An agent can *greet* another agent whenever the two agents are vertically or horizontally adjacent to each other. This is a symmetrical action, whenever agent A greets agent B, agent B also greets agent A.

In finding algorithms to solve the Greeting Problem, several recurring methods occur. A *wallflower* is an agent that does not move for the entirety of the algorithm, staying in its starting position the entire time. All greetings it performs occur when other agents move next to it.
A *loop* of length K is a pattern of movement the empty space makes through the room, defined as a vertical move, followed by K horizontal moves in a given direction, followed by a vertical move in the opposite direction to the first vertical move, followed by K horizontal moves in the opposite direction to the first horizontal moves. Of note is that the elements inside of the loop move in the opposite direction that the empty space does, shifting a single space in this opposite direction.

2.1 Movement

Movement of a single agent through the room is accomplished by continuously positioning the empty space around the given agent to the required side, depending on the direction that needs to be moved in. Movement in a straight line, either vertically or horizontally, requires five steps per single step of the agent, as the empty space requires 4 steps to move around the agent, followed by the movement of the agent itself (see Figures 3 to 8). However, when switching directions, from vertical to horizontal or vice versa, only three steps are needed to move the agent one step, as it only requires two steps to position the empty space above or below the agent from directly besides the agent, or to position it directly besides the agent from above or below the agent. Following from this, the most efficient way to move an agent through the room, is to move diagonally as much as is possible, alternating between horizontal and vertical movements. Moving an agent in location (i, j) in the room to a secondary position (k, ℓ), in the most efficient way possible, assuming the empty space starts off on the required side of the agent, requires the following number of steps:

$$1 + 3n + 5(m - 1)$$

where $n = 2 \cdot \min(|i - k|, |j - \ell|)$ and $m = \max(|i - k|, |j - \ell|) - \min(|i - k|, |j - \ell|)$.

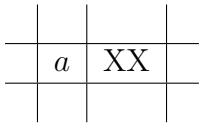

Figure 3: Starting position for the movement of agent a.

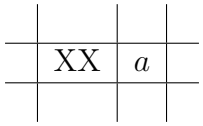

Figure 4: Position after agent a has been moved.

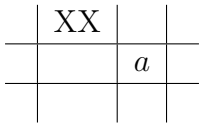

Figure 5: First step in moving the empty space around agent a.

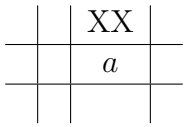

Figure 6: Second step in moving the empty space around agent a.

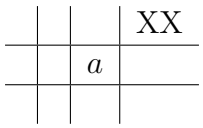

Figure 7: Third step in moving the empty space around agent a.

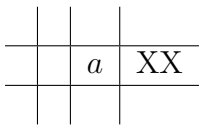

Figure 8: Final step in moving the empty space around agent a.

3 Related Work

In *The Kissing Problem: How to End a Gathering When Everyone Kisses Everyone Else Goodbye* by Bender et al. [BBCM12], the primary research on the Greeting Problem (previously called the Kissing Problem), the authors propose the *Boustrophedon Algorithm* as a solution for the Greeting Problem in crowded rooms. This algorithm moves the empty space through the entire room in a single linear path, going back and forth through the room alternating directions. This path does not overlap with itself, and the exact form of the path depends on the size of the room. If both the height and width of the room are even, the path simply goes back and forth going left and right as it goes down the room. If one of the dimensions is odd, the main straight path is parallel with the odd sided dimension (see Figure 9, part a). If both sides are odd, the algorithm employs a number of wallflowers to aid in the creation of the path, performing the path of the half-even case on a section of the room before involving the rest of the room (see Figure 9, part b). The number of steps this algorithm takes to complete in a room of size M by N is $M^2N^2 + 7MN - 7$.

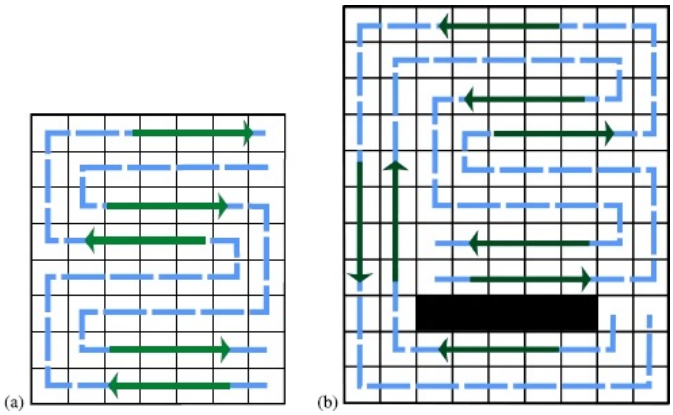

Figure 9: Examples of the Boustrophedon Algorithm in a) a 7 by 8 room and b) a 9 by 11 room with wallflowers marked black. Image taken from [BBCM12].

This number serves as the upper bound of our solutions, and will be our reference point when judging our acquired solutions.
Furthermore, the article provides a lower bound for any potential solutions, based on three factors: the total required greetings, the number of greetings made in the initiation of the room (this being the agents that start of adjacent to each other before movement), and the maximum number of greetings that can be made with each movement — at most 2 after the first movement, since after an agent k has moved, it will have three new neighbours, one of which was previously occupying the empty space agent k moved into, which as a result has already greeted agent k. This lower bound is $\frac{1}{4}(M^2N^2 - 7MN + 12 - 2M - 2N)$.
The Greeting Problem is similar in shape to sliding block puzzles ([RAH09]), like the 15-puzzle ([Wil74]) or the game Rush Hour ([FB02]), relying on similar mechanics to affect the state of the room. However, where for most of these sliding block puzzles the aim is to move the board state into a specific configuration, either the entirety of the board or a specific element, the Greeting Problem's goal involves relations between the various agents of the room, and the end configuration of the board is only relevant insofar as it might imply something about the method used to achieve it.

Most sliding block puzzles are generally NP-hard. Based on this, we could infer something about the question for the Greeting Problem, whether it can be solved for a given room in at most t steps. Since there is already a proven upper bound to this question, in the form of the Boustrophedon Algorithm, the value of t could be expressed as some fraction of the number of steps of the Boustrophedon Algorithm.

4 Methods

For the purpose of finding solutions for the Greeting Problem for a given sized room, we have made a C++ program that, given a room size and starting location of the empty space, finds the solutions with the smallest number of steps, printing all solutions with this number of steps, which includes the final state of the room, the number of steps required, and the path the solution takes.

4.1 Initial algorithm

Initially, our method of finding new solutions involved brute-forcing all possible step from a given room state. This method is supplemented with a number of limits placed on the potential moves, to reduce redundant movement and speed up processing time. The main limiting factor prohibits backtracking, as this will rarely add to successful solutions, while potentially causing the program to include a lot of meaningless loops of moving back and forth, which would impact execution times. The two other limiting factors are less stringent, and can be adjusted or removed to suit the need. The first of these is a hard limit to the number of consecutive horizontal or vertical moves that can be made before a vertical or horizontal movement must be made. However, this limit does not provide much in the way of a noticeable improvement in performance, as it turns out it is usually the most efficient to have the maximum number of horizontal moves available.
The second adjustable limit is symmetry in the movements. When this limiting factor is in play, all movement in the second half of the solution (based on the current depth of search that the program is running) is based on the movement in the first half. We have different levels of symmetry implemented, with varying degrees of strictness in the requirements for symmetry. At the most strict, every move to the left must be mirrored by a move to the right, and vice versa, and every move up must be mirrored by a move down, and vice versa. Less strict versions drop the hard symmetry on one or both of the directions (vertical and horizontal), only requiring that a vertical move is mirrored by a vertical move, and that a horizontal move is mirrored by a horizontal move. As a note, in solutions with an odd number of moves, the centre move will not have a symmetrical mirror move.
For our experiments, we have decided to limit our parameters to just symmetry. Our experiments were run using four differing levels of symmetry:

- Asymmetry, with no enforced mirroring of moves.
- Vertical-Horizontal Symmetry, where a vertical move must be mirrored by a vertical move, in either direction, and likewise for a horizontal move.
- Vertical-Left-Right Symmetry, where a vertical move must by mirrored by a vertical move, and a move to the right must be mirrored by a move to the left, and a move to the left must be mirrored by a move to the right.
- Hard Symmetry, where all moves must by mirrored by their opposite direction.

These levels of symmetry limit the potential direction the empty space can move in. For asymmetry, there is no real limit to the direction, excepting the rule that prohibits backtracking. As seen in Figure 10, the path of the empty space is not mirrored on itself. For any solutions that do actually employ some form of symmetry, this means that a found solution will have the moves in the first

half of the solution be mirrored in the second half of the solution. If a solution has an odd number of moves, the middlemost move will not have a mirrored move.

R R D R U U L L L

Figure 10: Example path using asymmetry.

For Vertical-Horizontal Symmetry, it is, for example, possible for a series of horizontal moves at the start of the path to be mirrored by the same set of horizontal moves at the other end of the path. Figure 11 shows an example path, with the string of horizontal right movements at the start of the path being mirrored by a string of right movements, and the vertical movement up at the start of the path being mirrored by a vertical movement up at the end of the path.

U R R D L L U R R U

Figure 11: Example path using Vertical-Horizontal symmetry.

For Vertical-Left-Right Symmetry, the same degree of vertical symmetry applies, meaning a vertical move at the start of the solution can be mirrored by either a move up or down, but any horizontal move must be mirrored by the opposite direction. Figure 12 shows an example path, with the string of horizontal right movements at the start of the path being mirrored by a string of left movements at the end, while the vertical movement up at the start is mirrored by another up movement. Of note is that the center movement down is never mirrored, as the path has an odd number of moves.

U R R R D L L L U

Figure 12: Example path using Vertical-Left-Right symmetry.

For Hard Symmetry, all movements are fully mirrored between the two halves of the path, the potential center move in odd length paths excepted. Figure 13 shows an example path, where the string of vertical moves up at the start of the path is mirrored by a string of moves down at the end of the path, and the string of of horizontal moves right is mirrored by a string of moves left at the end of the path. Of note is that, due to the harder limits of symmetry, valid paths applying hard symmetry always have an odd number of moves, as it is otherwise not possible for the two halves to fully mirror each other, since the limit on backtracking mean that the potential center two moves cannot be full mirrors of each other, as that would lead to a sequence of up-down, down-up, left-right, or right-left, all of which are forbidden.

U U R R R D L L L D D

Figure 13: Example path using hard symmetry.

4.2 Loops

While the automated program was sufficient to find solutions in rooms of a limited size, and provided templates for solutions of scaled up rooms, each increase in size in either width or height

drastically increases the execution time. Although the increase in time with a greater width is of limited importance, the slowdown for greater heights is sufficient that it becomes practically impossible to find consistent solutions for any rooms of a height greater than 3 with the time available to us. Any options for limiting the search space of the program, for example by making use of symmetry or other limiting factors in finding solutions, also increases the depth that the search needs to go to find solutions, which in turn increases run time.
As a result, using the automated program quickly becomes impractical, and while it aided in finding solutions for rooms of height 2 and 3, the results for all rooms of a greater height than three could not practically be found with this automated process.
Since the algorithms for the height 2 and 3 rooms both involved loops of some sort, we have decided to focus our attention on solutions that primarily incorporate loops. As such, we have implemented a user interface that allows the user to input parameters for loops for the program to run. This way, while the task of finding a solution does lie mostly with the user, the testing is done automatically and quickly, and it is still possible to run automated tests by setting the input parameters separately. The parameters for the loops are:

- Height h of the loop, where $2 \leq h \leq M$.
- Width w of the loop, where $2 \leq w \leq N$.
- Direction *dir* that the loop moves in.
- The first vertical movement *ud* the loop goes through, up or down.
- The vertical position *pos* of the empty space after the batch of loops is complete, with position 0 being the top position and position $M - 1$ being the bottom position.
- The number of loops *nr* to be run with these parameters.

We define the function loop as $loop(h, w, dir, ud, pos, nr)$ using the aforementioned parameters. Taking *start* as the y value of the starting position of the empty space, the total of number of moves for *loop* is $nr \cdot (2 \cdot w + 2 \cdot (h - 1)) + |pos - start|$,

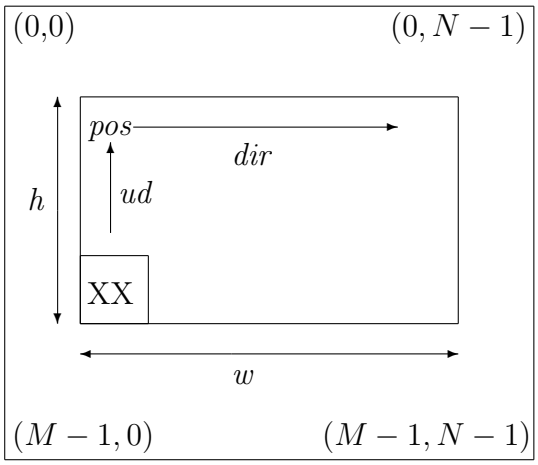

Figure 14: A visualisation of the parameters of the loop function.

5 Results

Through experimentation using the program we created, a number of viable solutions have been found for various rooms. Many of the most efficient solutions are asymmetrical paths with no clear pattern to them, which makes them hard to clearly describe, and makes scaling them up to larger rooms impossible.
However, a number of the solutions found, especially those with a greater degree of symmetry, follow patterns that are easier for a human to discern, and that can scale up for larger rooms, allowing for reliable algorithms to be followed to create solutions for rooms of any width for a given height. While these algorithms are not necessarily the most efficient paths, these paths are all shorter than the solution provided by the Boustrophedon algorithm [BBCM12].
In addition to the base results of our experimentation, the shortest path found by our program using the differing levels of symmetry, we will be providing a number of algorithms to be used to create efficient solutions for rooms of height 2, 3, and 4, as well as proofs for the validity of these algorithms.

5.1 Brute force results for height 2 and 3

Using the automated algorithm we have created, we have acquired data for the fastest found solutions for rooms of various sizes, when using our differing levels of symmetry. In Figure 15 and Figure 16 the results of our experiments for rooms of height 2 and 3 respectively can be found. In these figures, *Size* indicates the size of the room, primarily the width of the room; *Pos* indicates the starting position of the empty space in the room; and the other four column names indicate the versions of symmetry used:

- Asym: Asymmetry.
- Sym L/R UD/UD: Vertical-Left-Right Symmetry.
- Sym L/R U/D: Hard Symmetry.
- Sym LR/LR UD/UD: Vertical-Horizontal Symmetry.

These same column names are used in Figures 17 and 18, which show the runtime for the program for both room sizes. For both room heights our experimentation increased the room width up until the point that the runtime required to complete the algorithm exceeded the time we had reasonably available to us. In both cases, this meant we cut off our experimentation whenever run times started exceeding two hours. Notably, while the run time for the program finding solutions for rooms of height 2 steadily increased the larger the room got (though the jump in run time from width 6 to width 7 is still quite large), there was a more sudden jump in run time for rooms of height 3, where the execution time jumps for a few seconds for width 4 to not having completed within the allotted time for width 5.

Size	Pos	Asym	Sym L/R UD/UD	Sym L/R U/D	Sym LR/LR UD/UD
2×2	0,0	1	1	1	1
2×3	0,0	7	7	9	7
2×3	0,1	9	9	9	9
2×4	0,0	21	21	25	21
2×4	0,1	21	21	25	21
2×5	0,0	37	37	49	37
2×5	0,1	37	37	49	37
2×5	0,2	38	39	49	39
2×6	0,0	57	57	73	57
2×6	0,1	57	57	73	57
2×6	0,2	58	59	73	59

Figure 15: Results in number of steps of our experiments for rooms of height 2.

Size	Pos	Asym	Sym L/R UD/UD	Sym L/R U/D	Sym LR/LR UD/UD
3×3	0,0	19	21	25	19
3×3	0,1	19	23	23	19
3×3	1,0	19	19	23	19
3×3	1,1	21	23	27	21
3×4	0,0	40	45	49	41
3×4	0,1	41	43	51	41
3×4	1,0	41	45	49	43
3×4	1,1	42	45	51	43

Figure 16: Results in number of steps of our experiments for rooms of height 3.

Size	Pos	Asym	Sym L/R UD/UD	Sym L/R U/D	Sym LR/LR UD/UD
2×2	0,0	0.002s	0.002s	0.002s	0.002s
3×2	0,0	0.002s	0.002s	0.002s	0.002s
3×2	0,1	0.002s	0.002s	0.002s	0.002s
4×2	0,0	0.003s	0.002s	0.002s	0.002s
4×2	0,1	0.003s	0.002s	0.003s	0.002s
5×2	0,0	0.374s	0.024s	0.319s	0.040s
5×2	0,1	0.443s	0.023s	0.396s	0.048s
5×2	0,2	0.984s	0.047s	0.424s	0.110s
6×2	0,0	6m21.335s	0m7.061s	11m34.396s	0m17.931s
6×2	0,1	7m47.234s	9.113s	15m4.313s	22.706s
6×2	0,2	20m46.048s	21.356s	16m54.302s	1m5.346s

Figure 17: Runtime of the program for finding solutions for rooms of height 2.

Size	Pos	Asym	Sym L/R UD/UD	Sym L/R U/D	Sym LR/LR UD/UD
3×3	0,0	0.003s	0.003s	0.005s	0.003s
3×3	0,1	0.003s	0.005s	0.004s	0.003s
3×3	1,0	0.003s	0.003s	0.004s	0.003s
3×3	1,1	0.006s	0.005s	0.008s	0.004s
3×4	0,0	0.269s	2.477s	9.127s	0.218s
3×4	0,1	1.380s	0.810s	25.363s	0.221s
3×4	1,0	1.205s	2.803s	10.879s	1.688s
3×4	1,1	5.641s	3.350s	29.618s	1.779s

Figure 18: Runtime of the program for finding solutions for rooms of height 3.

5.2 Expanding Loop Algorithm

The Expanding Loop Algorithm is an efficient solution of the greeting problem for rooms of size 2 by N, with $N \geq 2$, that start in the configuration shown in Figure 19 :

A_1	A_2	...	A_N
B_1	XX	...	B_N

Figure 19: Required starting position for the Expanding Loop Algorithm.

This algorithm goes as follows: Starting at $p = 2$, the empty space moves in loops of width p, moving to the right. For the first $N - 2$ loops, p increments by 1 after each loop, increasing the coverage of the loop after each loop. Loop $N - 1$ has the same size as loop $N - 2$, and after this loop, the loop decrements for the next $N - 3$ loops, until the length of the loop is back down to $p = 2$. The algorithm ends with a single vertical move. The leftmost agents in the room, agent 1 and 2, are wallflowers.
In terms of loops, the algorithm can be expressed as seen in Figure 20:

Expanding Loop Algorithm

for$(i = 2; i \leq N - 1; i^{++})$
  $loop(2, i, c, u, 1, 1)$
for$(i = N - 1; i > 2; i^{--})$
  $loop(2, i, c, u, 1, 1)$
$loop(2, 2, c, d, 0, 1)$

Figure 20: The Expanding Loop Algorithm expressed in the *loop* function.

The number of moves needed to complete the Expanding Loop Algorithm is the sum of all loops involved, which is itself twice the sum of the first half of the set of loops, plus an additional single move. The total number of steps is $2N^2 - 2N - 3$. See below for the calculation.

$$1 + 2 \cdot \sum_{i=1}^{N-2} 2i + 2 = 1 + 2(N^2 - N - 2) = 1 + 2N^2 - 2N - 4 = 2N^2 - 2N - 3$$

Below (see Figure 21) is a graph visualising the greetings made in a 2 by 6 room over the course of the Expanding Loop Algorithm. The nodes are the agents, marked as they are in Figure 22, and the edges are greetings between two agents, with the colour signifying the loop that the greeting was made. Here a black edge represents a greeting made initially, before any movement is made, red represents a greeting made in the first loop, orange a greeting made in the second loop, yellow a greeting made in the third loop, light green a greeting made in the fourth loop, dark green a greeting made in the fifth loop, blue a greeting made in the sixth loop, purple a greeting made in the seventh loop, and magenta a greeting made in the final loop and the last vertical movement.

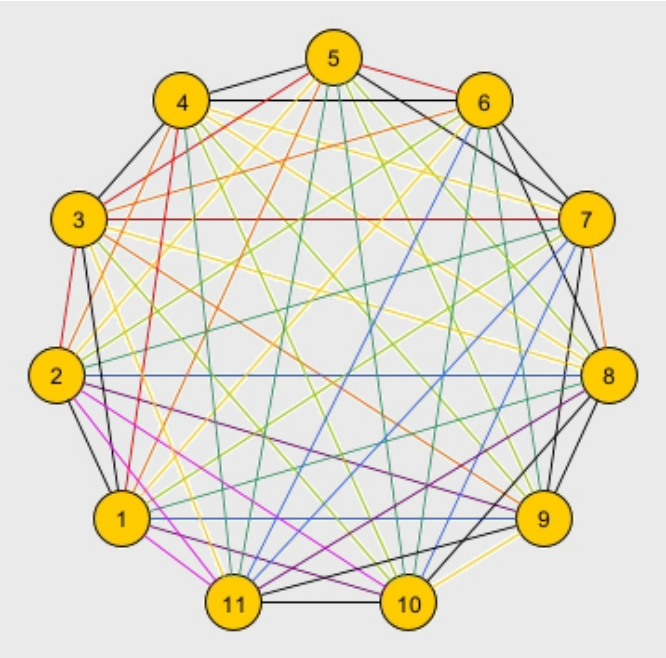

Figure 21: Meeting Graph for the Expanding Loop Algorithm in a 2 by 6 room.

5.2.1 Correctness proof

Proof for the correctness of the Expanding Loop Algorithm for $N = 3$ is provided below. For all cases where $N \geq 4$, the following applies: For a given k, where k is even, and $4 < k \leq 2N - 2$, k will have met all agents of value smaller than k after $k - 2$ loops, and $k + 1$ will have met all agents of value smaller than $k + 1$ after $k - 2$ loops and 1 step.
In the initial state (see Figure 22), k meets $k + 1$ and $k - 2$ and $k + 1$ meets $k - 1$. Every loop, all agents in the loop move one step counterclockwise, with the top row moving first. Every agent ℓ in the top row of the loop, that is not in the leftmost position, can make two new greetings over the course of a loop, discounting the continued meeting with the agent moving directly in front of ℓ in the loop; one meeting when ℓ moves horizontally, where it meets the agent to its diagonally lower left, and another when the agent left to that agent moves horizontally. In the first $\frac{k}{2} - 3$ loops, no

relevant meetings are made regarding k and $k+1$. In loop $\frac{k}{2}-2$, $k+1$ meets 3 and k meets $k-1$ (see Figure 23). In loops $\frac{k}{2}-1$ and $\frac{k}{2}$, agents k and $k+1$ both get moved into the upper row (see Figure 24). In the process, k meets 3 and 4, and $k+1$ meets 4 and 5. The last new agent k meets before the wallflowers 1 and 2 is $k-3$, and the last new agent $k+1$ meets before the wallflowers 1 and 2 is $k-2$.

1	3	4	...	k	...	$2N-2$
2	XX	5	...	$k+1$	...	$2N-1$

Figure 22: Initial state.

1	$\frac{k}{2}+1$	$\frac{k}{2}+2$	...	$k-2$	$k-1$	k	...	$2N-2$
2	XX	$\frac{k}{2}$	...	4	3	$k+1$	...	$2N-1$

Figure 23: Board after $\frac{k}{2}-2$ rounds. Agents 3 to $k-1$ have been ordered clockwise starting in the lower right.

1	$\frac{k}{2}+2$	$\frac{k}{2}+3$	...	$k-1$	k	$k+1$	...
2	XX	$\frac{k}{2}+1$	...	5	4	3	...

Figure 24: Board after $\frac{k}{2}-1$ rounds. Both agents k and $k+1$ have been moved into the upper row of the next loop.

5.2.2 Proof $k = 4$

As the starting column of 4 is directly next to the column where the empty space starts, the proof for this section is much more condensed, as it only consists of the free floating 3 and the two agents in column 3, agents 4 and 5. All steps needed for agents 4 and 5 to make all relevant meetings are depicted in Figures 25 to 29. These steps also qualify as the solution for the 2 by 3 board.

1	3	4
2	XX	5

Figure 25: Initial State: Greetings 1/2 4/5 1/3 3/4.

1	XX	4
2	3	5

Figure 26: Step 1: Greetings 2/3 3/5.

1	4	5
2	XX	3

Figure 27: First Loop: Greetings 1/4.

1	5	L
2	XX	4

Figure 28: Second Loop: Greetings 2/4 1/5.

1	L	K
2	XX	5

Figure 29: Third Loop: Greetings 2/5.

5.3 Butterfly Loop Algorithm

The Butterfly Loop Algorithm is an efficient solution of the Greeting Problem for rooms of size 3 by N, with $N \geq 3$ that start in the configuration shown in Figure 30:

A_1	A_2	...	A_N
XX	B_2	...	B_N
C_1	C_2	...	C_N

Figure 30: Starting configuration for the Butterfly Loop Algorithm.

The algorithm consists of $N-1$ loops in the upper section of the room followed by $N-1$ loops in the lower section of the room. These loops have a width of N, thus covering the full width of the room. The algorithm ends with a single vertical move.
In terms of the *loop* function, the algorithm can be expressed as seen in Figure 31:

Butterfly Loop Algorithm

$loop(2, N, c, u, 1, N-1)$
$loop(2, N, a, d, 2, N-1)$

Figure 31: The Butterfly Loop Algorithm expressed in the *loop* function

For the Butterfly Loop Algorithm, the number of steps it takes to complete the algorithm is the sum of the two sets of loops in the algorithm, which are mostly of equal length, with a single step added to the second set of loops. The number of moves is $4N^2 - 4N + 1$, the calculation for which is below.

$$2(N-1)(2N) + 1 = 2N(2N-2) + 1 = 4N^2 - 4N + 1$$

5.3.1 Correctness proof

In this subsection we prove the following claim: For all rooms of height 3 and a width of $N > 1$, starting in the position seen in Figure 32, the following applies: the agent starting in position (1,1) (here $N + 1$), marked in Figure 33, will have greeted all agents except the ones in the first column (here 1 and $2N$) after $N - 2$ loops have elapsed. Notice that here loops can be upper or lower, and in any order.
We use the this claim to prove correctness of the Butterfly Loop Algorithm.

A_1	A_2	...	A_N
XX	B_2	...	B_N
C_1	C_2	...	C_N

Figure 32: Begin state for the Butterfly Loop Algorithm.

A_1	A_2	...	A_N
XX	B_2	...	B_N
C_1	C_2	...	C_N

Figure 33: Marked begin location at the start of the loop.

Proof of claim: Over the course of a loop, all agents inside of the loop will move one space counter-clockwise, with the outer row involved in the movement moving in the first half of the loop, and the centre row moving in the second half of the loop. We will refer to the outer row as the *rising row*, and the centre row as the *falling row*. Meanwhile, the third row, not involved in the loop, stays the same between loops. We will refer to this row as the *stable row*.
Every loop, the tracked agent will greet two agents in the rising row. The first when the rising row moves, moving a new agent adjacent to the tracked agent, and the second when the falling row moves, moving the agent adjacent to a new agent. Every loop, one agent from the rising row will be moved to the falling row and one agent from the falling row will be moved to the rising row. Through this movement, the agents that started on the falling row on positions $x = 3$ and onward will greet the tracked agent as described above.
After $N - 2$ loops, the tracked agent will have moved from its starting position of (1,1), to position $(N - 1, 1)$. Over the course of this movement, it will have moved adjacent to every agent in the stable row from position $(1, x)$ onward.

Proof of algorithm correctness Using this claim, the validity of the algorithm can be proven. After every loop, a new agent will arrive in position (1,1), this being the leftmost agent in the rising loop. This movement also provides the previously tracked agent with one of the greetings it still needed to make.
During the second half of the algorithm, the leftmost agent on the secondary stable row will greet each of the tracked agents *before* they arrive at position (1,1), on the first step of the loop. Since the leftmost agent on the secondary stable row will not move any longer, and never left its starting row, this is the only moment to make this greeting. This is also how agent C_N will perform the last greeting of the algorithm.

The last $N-1$ agents to pass through the starting position of the empty space, (0,1), do no get sufficient loops to fully complete the above described process. However, they will have already performed most of their greetings in the previous loops, and the remaining agents don't need the full $N-2$ loops to perform their remaining greetings.
These last agents all started in the initial stable row, in the positions $x = 1$ and onward. Based on their starting x positions (x_i), the x position they need to greet all remaining agents in the secondary stable row and the secondary rising row can be determined as $n-1-x_i$. In the loops required to get to this position, the agent will greet all agents that were initially in a higher x position on the first falling row, and the rightmost agent on the first rising row, which never leaves its row, and all, if any, agents on the secondary rising row that started in a x position $x_i + 2$ or higher.

5.3.2 Order of loops

For the Butterfly Loop Algorithm, the order that the loops move up or down does not matter, as long as the number of loops satisfies the requirements of the algorithm.
Over the course of the $N-2$ loops required for the tracked agent to move from its starting position to the rightmost position in the centre row, its own movement will have moved it past all agents on the outer two rows, regardless of any moves on these rows.
The agents on the centre row past the position of the tracked agent will be moved onto either the upper or lower row, depending on the direction of the loop that they are moved onto the outer row in. In this movement, all agents on the outer row get moved one to the left, granting the tracked agent an extra opportunity to greet another agent, effectively compressing the affected row so it can fit one more agent.
As in the original butterfly loop algorithm, the last $N-1$ agents that pass through the starting empty position do not get sufficient loops to fully complete the required $N-2$ loops. Rather, most of their greetings will take place as they move through their starting outer row and as agents move past them through the centre row. The last agent to make a greeting and enter the starting empty position does not leave their initial starting row until this greeting is made. This agent starts off in the rightmost position on their row, and it's pattern of movement means it is always positioned such that all elements on the centre row to the right of it, have already greeted it. This is because it starts to the far right, and if the loop moves in its vertical direction, it will always move first on the rising edge of the loop, meaning it will meet the agent previously to the left of it on the centre row before this agent can move.
For the last $N-1$ agents it applies that the only agents that they do not meet over the course of moving through their outer row, are the agents initially on the centre row in a position to the right of the agent, and the agents on the same row as the agent that are more than one to the right of the agent; see Figure 34 for example.

A_1	A_2	A_3	A_4
XX	B_2	B_3	B_4
C_1	C_2	C_3	C_4

Figure 34: Example room. Given that C_2 is one of the $N-1$ last agents, the only agents it still needs to meet in the centre row excepting A_4 are in red.

The agents initially on the centre row will all end up in the outer two rows, with their specific position dependant on the order of directions of the first $N-1$ loops. The rightmost agent on the centre row will always end up in column 1 of one of the outer rows. The second most right agent will be in column 1 or 2. This pattern continues up to and including the agent starting in column 1 on the centre row, which can end up in any position on the outer rows from column 1 and up.
Agents on the outer rows require a number of loops in the direction of the row that they start off in to reach the centre row equal to i, where i is the starting column of the agent. After this their eventual position depends on the number of loops still left in the algorithm
The agents that finally end up on the centre row are dependant on the direction of the last $N-1$ loops. From right to left, the agents will originate from the upper or lower row based on the direction of the loop, the upper row for an up movement, the lower row for a down movement. From this it follows that the number of agents from each row that can not complete the full $N-2$ loops equals the number of movements in the corresponding direction in the last $N-1$ loops of the algorithm. From this it follows that the starting position of the last agents can be inferred from the order of direction of the last $N-1$ loops.
The last agents going through the tracked position end up in positions dependant on their positions relevant to each other. The leftmost agent ends up in the second most right column, the next agent in the column to the left of that, and so forth.

5.4 Stacking Loop Algorithm

The Stacking Loop Algorithm is an efficient solution of the greeting problem for rooms of size 4 by N, with $N \geq 4$ that start in the configuration shown in Figure 35:

A_1	A_2	...	A_N
XX	B_2	...	B_N
C_1	C_2	...	C_N
D_1	D_2	...	D_N

Figure 35: Starting configuration for the Stacking Loop Algorithm.

The algorithm consists of a sequence of loop configurations, each spanning different sections of the room. These configurations are as follows:

1. $N-1$ loops of height 2 on the upper two rows.
2. $2N-1$ loops of height 2 on the center two rows.
3. N loops of height 3 spanning the lower three rows.
4. N loops of height 2 on the upper two rows.
5. $N-3$ loop of height 2 and width $N-2$ spanning the upper two rows, ending with a single vertical movement up to y position 0.

As expressed in the *loop* function, the algorithm reads as follows:

For the Stacking Loop Algorithm, the number of moves is a summation of the number of moves of each step in the algorithm. The number of moves in each step is as follows:

Stacking Loop Algorithm

$loop(2, N, c, u, 1, N)$
$loop(2, N, c, d, 1, 2N-1)$
$loop(3, N, a, d, 1, N)$
$loop(2, N, c, u, 1, N)$
$loop(2, N-2, c, u, 1, N-3)$

Figure 36: The Stacking Loop Algorithm expressed in the *loop* function.

1. $(N-1)(2(N-1)+2) = 2N(N-1) = 2N^2 - 2N$
2. $(2N-1)(2(N-1)+2) = 2N(2N-1) = 4N^2 - 2N$
3. $N(2(N-1)+2\cdot 2) = N(2N+2) = 2N^2 + 2N$
4. $N(2(N-1)+2) = N \cdot 2N = 2N^2$
5. $(N-3)(2(N-2-1)+2)+1 = (N-3)(2N-6+2)+1 =$
   $(N-3)(2N-4)+1 = 2N^2 - 10N + 13$

And following from this, the total number of moves to complete the algorithm is $12N^2 - 12N + 13$ (see the following calculation):

$$2N^2 - 2N + 4N^2 - 2N + 2N^2 + 2N + 2N^2 + 2N^2 - 10N + 13 = 12N^2 - 12N + 13$$

5.4.1 Correctness proof

For the sake of this proof, we will track each row by number, identifying each row by its starting position (see Figure 37). Whenever the agents of one row (mostly) swap positions with agents of another row, we will indicate this by swapping their row identifiers.

1	A_1	A_2	...	A_N
2	XX	B_2	...	B_N
3	C_1	C_2	...	C_N
4	D_1	D_2	...	D_N

Figure 37: Starting configuration for the Stacking Loop Algorithm, with each row marked.

Over the course of the first set of loops (see Figure 38, as well as the first $N-1$ loops of the second set of loops (see Figure 39, the algorithm follows the Butterfly Loop Algorithm that we have proved previously.
After the first two sets of loops, all agents in row 1, 2 and 3 will have met each other.

2	A_N	B_N	...	B_1
1	XX	A_{N-1}	...	A_1
3	C_1	C_2	...	C_N
4	D_1	D_2	...	D_N

Figure 38: Configuration after the first set of loops.

Additionally in the second set of loops (again, see Figure 39), over the full $2N-1$ loops, all agents of row 1 and 3 meet all agents in row 4, since the second set is a full rotation of the the rows, moving all agents across every position in the two rows, meaning all agents move past all agents in row 4.
After the second set of loops, all agents in row 1 and 3 will have met all agents in row 4.

2	A_N	B_N	...	B_1
1	XX	A_{N-1}	...	A_1
3	C_1	C_2	...	C_N
4	D_1	D_2	...	D_N

Figure 39: Configuration after the second set of loops.

In the third set of loops (see Figure 40), the number of new greetings is limited, and this set mostly serves to set up row 4 next to row 2 to facilitate the last greetings that need to be performed. During this set, what is effectively the first half of a full rotation between row 2 and row 4 is performed, which, combined with the movement in the fourth set of loops performing the latter half of the pseudo full rotation (see Figure 41), makes all agents in row 4 meet all agents in row 2.

2	A_N	B_N	...	B_1
4	XX	D_{N-1}	...	D_1
3	D_N	C_2	...	C_1
1	C_N	A_1	...	A_{N-1}

Figure 40: Configuration after the third set of loops.

4	D_1	D_2	...	A_N
2	XX	B_1	...	B_N
3	D_N	C_2	...	C_1
1	C_N	A_1	...	A_{N-1}

Figure 41: Configuration after the fourth set of loops.

After the third and fourth set of loops, all agents in row 2 have met all agents in row 4, and all agents in row 4 have met all agents in row 4 except agent D_N, which was left out of the loop on the first and second row.

Finally, the last set of loops (see Figure 42) are purely to facilitate meetings with the remaining agents on row 4 with agent D_N, which started off on row 4 but does not move fully with it into the loops covering the first and second row of the room. In this set of loops, the remaining $N-2$ agents that have not met agent D_N yet are moved past it.

4	XX	D_{N-1}	...	A_N
2	D_{N-2}	D_{N-3}	...	B_N
3	D_N	C_2	...	C_1
1	C_N	A_1	...	A_{N-1}

Figure 42: Configuration after the fifth set of loops.

After the fifth set of loops, all agents in row 4 have met all other agents in row 4, in particular, all agents in row 4 have met agent D_N.

6 Conclusion

Over the course of this paper, we have discussed the Greeting Problem, looking into previous research done on the topic and seeking to improve on the results of this research. We have gone in depth on crowded rooms, developing methods of finding fast, efficient solutions to the problem for rooms of varying sizes. We have formatted several clear, relatively efficient algorithms to solve the Greeting Problem for rooms of height 2, 3, and 4, which can scale out freely to any width.
Comparing the number of steps of our algorithms with the results from the Boustrophedon algorithm (see Figure 43), we can see clear improvements between our algorithms and the slower Boustrophedon. While the difference is relatively small for rooms of height 2, the Expanding Loop Algorithm only linearly improving on the basic loops, both the Butterfly Loop and Stacking Loop Algorithms have drastically shorter paths to complete the problem, a difference of $5N^2 + 25N$ for the rooms of height 3, and a difference of $4N^2 + 40N$ for rooms of height 4.

Height	2	3	4
Lower Bound	$2N^2 - 6N + 4$	$(9N^2 - 23N + 6)/4$	$(16N^2 - 30N + 4)/4$
Found	$2N^2 - 2N - 3$	$4N^2 - 4N + 1$	$12N^2 - 12N + 13$
Boustrophedon	$2N^2 - N$	$9N^2 + 21N - 7$	$16N^2 + 28N - 7$

Figure 43: Number of moves for various algorithms, measured by room height.

6.1 Future research

The most obvious line of further research into the Greeting Problem, regarding the crowded rooms, is finding workable algorithms for larger room sizes. Optimally, an algorithm could be found that could be scaled to both the room's height, as well as the width.
Furthermore, there is matter of the potential of using wallflowers in solutions. Only the Expanding Loop Algorithm currently makes use of wallflowers, and none of our methods of finding potential solutions have an option to enforce wallflowers. Implementing this for the automatic methods brings certain problems; since, in many cases, moving the wallflowers might be beneficial, or even required, to finding an optimal solution, which means the program needs to be able swap the wallflowers from being stationary to being able to move.
Finally, looking further into the influence of symmetry in solutions could provide insights in the potential for algorithms for larger rooms. From our current results, both the Expanding Loop and Butterfly Loop Algorithm are symmetrical, keeping to hard symmetry in their horizontal movements (which turns to soft symmetry in the Butterfly Loop Algorithm if the standard order is not followed), and soft symmetry in their vertical movements. Both rely on loops of similar sizes being mirrored on either end of the solution, with a singular vertical move mirroring the opening vertical move.

References

[BBCM12] Michael A. Bender, Ritwik Bose, Rezaul Chowdhury, and Samuel McCauley. The kissing problem: How to end a gathering when everyone kisses everyone else goodbye. In *Fun with Algorithms (FUN2012)*, pages 28–39. Springer LNCS 7288, 2012.

[FB02] Gary William Flake and Eric B. Baum. Rush hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1):895–911, 2002.

[RAH09] Erik D. Demaine Robert A. Hearn. *Games, Puzzles & Computation*. A K Peters Ltd., 2009.

[Wil74] Richard M. Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, 16(1):86–96, 1974.