Ant Colony Optimization: Finding the Longest Path in a Maze

Armand Maree
120 178 00
Department of Computer Science
University of Pretoria

***Abstract*—Ant Colony Optimization (ACO) is a meta heuristic inspired by ant colonies. It is an algorithmic approach to solve combinatorial optimization problems such as the Traveling Salesman Problem (TSP). This paper compares the performance of an ACO to a Beam Search (BS) algorithm to find the longest path from the entrance to the exit of a maze.**

I. Introduction

ACO is a meta heuristic inspired by ant colonies [1]. The sought after characteristic of the ant colony is the pheromone trailing used between ants to communicate. Another characteristic inherited from the ant colony is the extremely simplistic nature of a single individual, but due to collaboration between these simple individuals an intelligent "hive mind" can emerge.

Pheromone trails are laid down by the ants to indicate the path that they have traveled on. These pheromone trails are then used by other ants to guide them to food sources found by other ants. The idea is that the more pheromone is present on a specific path the more likely an ant will be to follow the path. Eventually most ants will be following one path. This path will be very close to a direct path to the food source (in other words, the shortest/most optimal path).

This paper will use an ACO to find the longest acceptable path through a maze stored in an image file. The results of this will be compared to the more classical BS algorithm.

II. Background

ACO is usually used to solve problems like the TSP which fall in the field of the combinatorial optimization problems. The solutions to these problems are usually found within a finite (although sometimes countably infinite) set of possible solutions [2]. Since finding the most optimal solution can often be extremely hard, an acceptable solution is usually used as the solution.

One aspect where ACO does not completely model the behavior of the actual ant is when pheromones are dropped. The ACO only drops pheromone as they return to the start. This means that only the paths that actually lead to a solution will have a higher concentration of pheromone on it.

III. Implementation

A. ACO Implementation

1) Maze initialization: Once the maze image (a *.bmp* image) is loaded into the program a matrix (M) is constructed where -1 in position $M_{i,j}$ means that the pixel in row i and column j was black and is thus a wall. All other positions (white pixels) are initialized to $(numRows + numCols) \div 2$. The purpose of M is to firstly keep track of the maze layout but also to keep track of the pheromone concentrations all over the map.

The reason non-wall positions in M is initialized to $(numRows + numCols) \div 2$ is to allow every path to always have a chance (although small) to be chosen as a next move. This value is thus the minimum pheromone concentration a non-wall position in M can have.

2) Choosing a move: The ACO was implemented in such a way that moves leading in the general direction of the exit are less likely to be chosen as the next position to visit in the maze. In other words, if an ant is in the center of a map, and the solution is in the top right corner, then the ant will favor moves that go down or left by multiplying the pheromone concentration of those positions by 1.5. This will cause the ant to be repelled by the exit position and will thus help find the longest path.

When an ant has to make a move, the ant uses equation 1 to calculate the probability of choosing a specific move. $\tau_{i,j}$ refers to the pheromone concentration of a specific position, i, coming from position j and $\mathcal{N}_i^k$ is the set of possibles moves ant k can make from node i. After this it choses a pseudo random number, $r \in [0, 1]$, and based on this choses which move to make, by following algorithm 1.

$$P_{i,j} = \begin{cases} \frac{\tau_{i,j}^{\alpha}(t)}{\sum_{e \in \mathcal{N}_i^k} \tau_{i,e}^{\alpha}}, & \text{if } j \in \mathcal{N}_i^k \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$\text{where, } \tau_{i,j}^{\alpha}(t+1) = \tau_{i,j}^{\alpha}(t) + \Delta\tau_{i,j}^{\alpha}(t) \tag{2}$$

$$\text{where, } \Delta\tau_{i,j}^{\alpha}(t) = \sum_{k=1}^{n_k} \Delta\tau_{i,j}^{k}(t), \alpha = 1.0 \tag{3}$$

Algorithm 1: Move selection algorithm

```
1  possibleMoves = posssible moves ant can make
2  randomNumber = random number between 0 and 1
3  sum = sum of all pheromone of moves in possibleMoves
4  sumCounter = 0
5  targetCell = null
6  do
7      sumCounter += possibleMoves(index) / sum
8      if randomNumber <= sumCounter then
9          targetCell = possibleMoves(index)
10     else
11         index++
12     end
13 while targetCell == null and index < possibleMoves.length;
```

3) Pheromone dropping: Once an ant has found an exit to the maze the ant is "teleported" back to the entrance to the maze and its memory is cleared (i.e. the path it followed and its avoid list, see section III-A4, is reset). Each position the ant visited gets a pheromone increase by following equation 4. In oder to prevent over saturation of a particular path, the maximum amount of pheromone, τ_{max}, that a specific position can have is limited to $numRows * numCols$.

$$\begin{gathered}\Delta\tau_{i,j}^{k}(t) = L_k \\ \text{where, } L_k \text{ is the total length} \\ \text{of the path that ant } k \text{ found.}\end{gathered} \tag{4}$$

Some paths found early on might not be very good paths and it would be more advantages if these paths play less of a roll the longer the algorithm runs for. This was accomplished by using pheromone evaporation. After each iteration the pheromone of each position of the maze is updated using equation 5 where $\rho = 0.01$.

$$\tau_{i,j} = \begin{cases} (1-\rho)\times\tau_{i,j}, & (1-\rho)\times\tau_{i,j} \geq \tau_{min} \\ \tau_{min}, & \text{otherwise} \end{cases} \tag{5}$$

4) Loop detection: Detecting loops can be done in two ways, either by removing them right at the end after an ant has found a path or on each iteration when the ant choses a move. The latter option was implemented for this assignment.

Initially the algorithm was implemented to simply remove loops from the ant's memory as soon as it detects a loop and pretends like the loop never occurred. This option proved to be somewhat successful for small mazes, but inefficient for larger mazes. The reason for this is that the ant can keep moving in circles and possibly take a very long time to escape the loop, hindering progress significantly.

The second approach was to rather have the ant back track until it reaches a point where it can chose a move which it has not visited previously. As the ant back tracks it removes the move from its memory and adds it to an "avoidance matrix" (a matrix that indicates what positions should never be visited again). This approach requires more memory than the first approach but it makes up for this with the reduction in computational time.

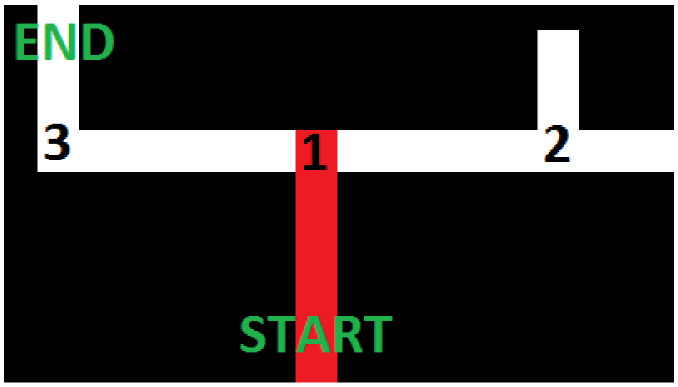

Figure 1. Possible move choices

Originally this approach iterated over the path of the ant in order to determine whether the position has been visited before. This, again, proved useful for smaller maps, but if the path became long this process become exponentially expensive. In order to reduce this constant iteration, a matrix system was implemented. Using this matrix it became possible to mark all the positions an ant has visited and thus make use of the random access nature of arrays in order to speed up the loop detection process. This matrix and the avoidance matrix was combined to reduce memory.

5) Stopping conditions: For this assignment, the ACO will run until the first path to the exit is found. Once this first path has been found the algorithm will be allowed to run for another $MAX(2 \times previousSolutionIteration, 10000)$ iterations.

6) Optimizations : In order to reduce the usage of computer memory, ants move in ways that allow them to skip positions that will not require any decision making. For instance, if an ant is located at position 1 in figure 1 then the ant will chose its next move as position 2 or 3. This is because any position between any of these points only has one possible move that can follow. Since position 3 requires that an ant changes direction, the ant is required to stop here even though there is only one possible move that can follow.

When an ant reaches a point where it has no further possible moves, either due to a dead end or due to a loop that might occur, the ant has to back track in order to find another route that it skipped previously. Since there is no real reason why the ant has to physically traverse there, the ant can simply teleport to the last position it decided not to chose. This reduces computational time by avoiding the whole process of moving back on a path that has been previously discovered contains no solution.

Mutli-threading has also been used to speed up the searching process. Since the ACO was run on a 6-CPU system, 6 threads were used. Each thread was responsible for 4 ants to make a total of 24 ants in the colony.

B. Beam Search Implementation

1) Maze Initialization: When the program starts up the maze image is read into a matrix. If a pixel is black, then the cell in the matrix is initialized to -1. A white pixel, a.k.a. a path, is initialized to 0 in the matrix.

Each cell in the matrix can contain either the value -1, 0, 1 or 2. The two remaining values (1 and 2), indicates the cell is part of the tree already and the cell is not part of the tree but has been visited, respectively.

2) Choosing a move: Similar to the way the ACO was implemented, the BS was also implemented in such a way to always favor moves that are further away from the exit. In the BS the euclidean distance (equation 6) was used to determine the distance between any specific cell and the exit to the maze.

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{6}$$

Each time the algorithm has to choose which positions it will explore it gathers a list of all possible moves. From these possible moves it choses two and marks their corresponding fields in the matrix (from section III-B1) as 1. The remaining fields that is not used will be given the value 2 in the matrix. The purpose of this is further discussed in section III-B4.

3) Tree details: A beam size of 2 was used to construct the a N-ary tree. As the algorithm visits new positions in the maze, a node representing the position is then added as a child to the previously visited node.

4) Loop avoidance: ACO will, if no counter strategy is implemented, get stuck moving in circles when a loop is encountered. BS is more likely to rather terminate without a solution if a loop is encountered. In order to avoid this occurrence the BS will always save any node it did not explore earlier in a temporary list. Should a situation occur where the BS has no more possible moves it simply pops the last two unexplored nodes from the saved list and it carries on like normal. This mechanism ensures that if a path exist BS will find it. BS is not guaranteed to find the most optimal path since it is possible that the solution is in a subtree that that was previously discarded.

5) Optimization: Similar to what was done with the ACO, BS also has a jumping behavior when selecting new positions. This saves computational time by avoiding to physically traverse a path that cannot provide ant change in direction. The algorithm simply jumps to the next position that is either an intersection, a dead end, or a turn (or corner).

Maze Name	Time to Complete (min)	Length of path (num dots in path)
Small 1	<1	83
Small 2	<1	29
Small-Medium 1	<1	1 581
Small-Medium 2	<1	1 507
Small-Medium 3	<1	716
Small-Medium 4	<1	85
Medium 1	377	103 419
Medium 2	605	75 610
Medium 3	33	5 791
Large 1	>3205	N/A

Table I
TIME/STEPS TO SOLVE MAZES FOR ACO

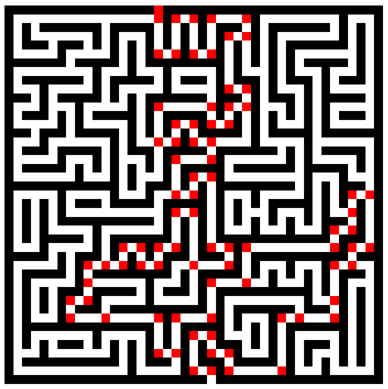
Figure 2. ACO solution for maze Small1

IV. Research Results

A. ACO results

Table I shows the times and number of iterations it took to solve each of the mazes. Figures 2, 3 and 4 shows the paths that were found for mazes Small1, Small2 and Medium1 respectively. As you can note in figure 2 and 3 the path that is shown clearly indicates the manner in which the ant jumped to next positions as described in section III-A6.

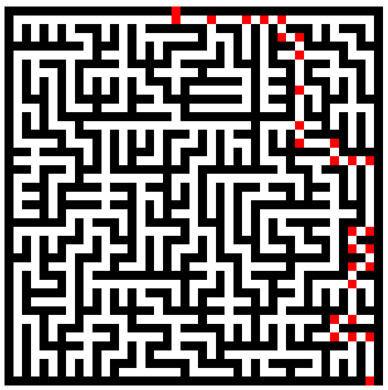
Figure 3. ACO solution for maze Small2

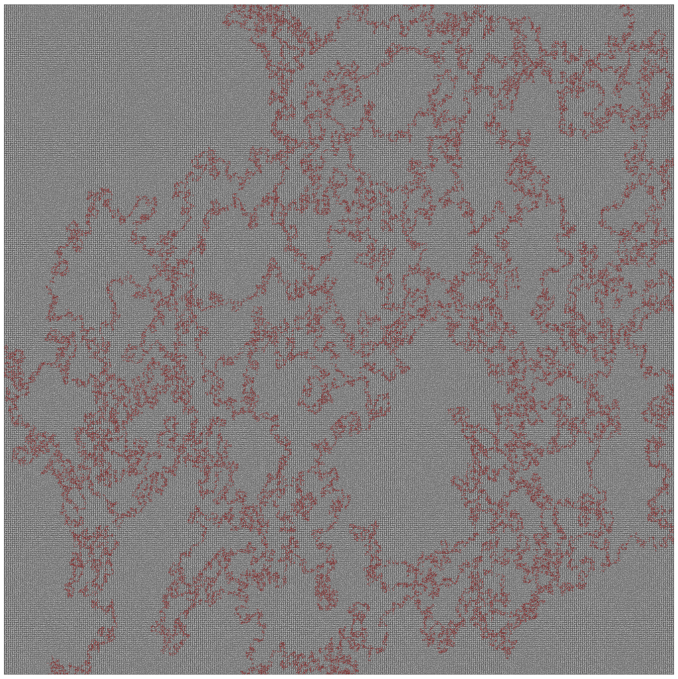

Figure 4. ACO solution for maze Medium1

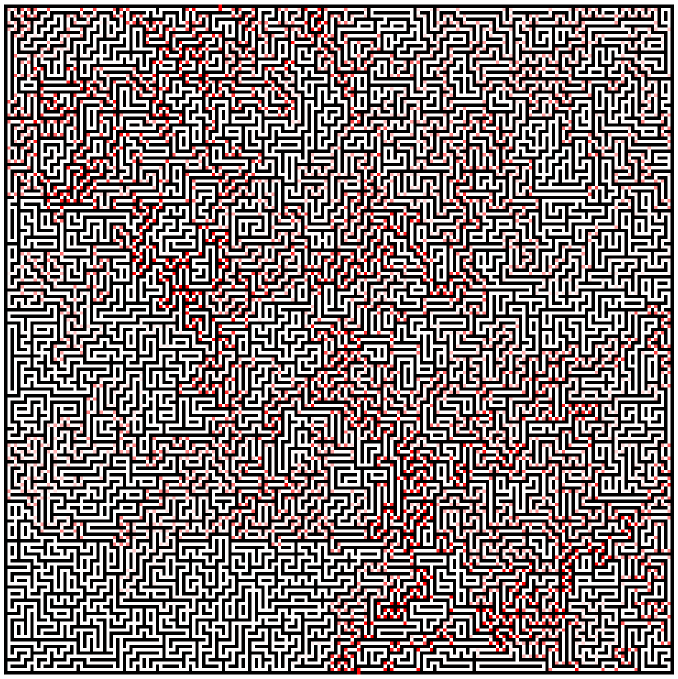

Figure 5. ACO pheromone concentrations for maze SmallMedium2

Figure 5 faintly shows the different pheromone concentrations that were present during some stage of the ACO life cycle. More opaque red dots indicate a higher concentration of pheromone where a more transparent dot indicates a lower concentration of pheromone.

Maze Name	Time to Complete (min)	Length of path (num dots in path)
Small 1	<1	71
Small 2	<1	29
Small-Medium 1	<1	1 385
Small-Medium 2	<1	1 530
Small-Medium 3	<1	716
Small-Medium 4	<1	85
Medium 1	3	123 568
Medium 2	2	118 059
Medium 3	<1	5 791
Large 1	795	3 012 138

Table II
TIME/STEPS TO SOLVE MAZES FOR BS

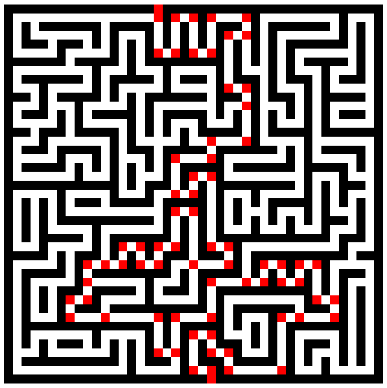

Figure 6. BS solution for maze Small1

B. Beam search results

Table II shows the results obtained from the BS for each of the mazes provided. Figures 6, 7 and 8 shows the paths that were found for mazes Small1, Small2 and Medium1 respectively. Once again the "jumping" behavior of the BS as described in section III-B5 can be seen in figure 6 and 7.

C. Observations

Considering the results in section IV it is clear that BS was vastly more superior during the tests. The algorithm is a lot simpler to implement and also takes less time to implement, and it can also find solutions rapidly even though

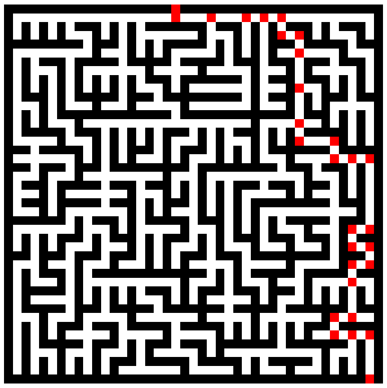

Figure 7. BS solution for maze Small2

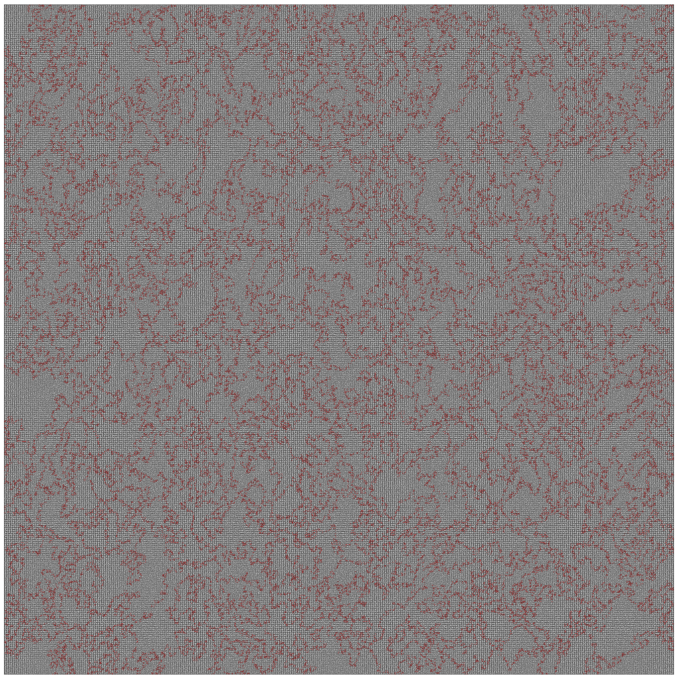

Figure 8. BS solution for maze Medium1

the probability of it discarding a better solution is higher than with the ACO. Despite this possibility the BS found better solutions than ACO more often.

Another factor to consider is the stochastic nature of the ACO, during the testing of this algorithm it occasionally found solutions very rapidly (although not as rapidly as BS). The results shown in table I are the closer to the average time it took to solve the mazes. On the other hand, the BS will always produce the same result for a given maze since it does not contain any stochastic component. This can be seen as a negative aspect if the algorithm finds a solution that is very far from the optimal, as it will always in subsequent runs produce this path.

Another interesting event to note is that ACO spends less time to find a solution than it spends on optimizing that solution. The outcome of these results (w.r.t. the lengths of the paths) might have been different if the heuristic for the ACO was adapted in such a way that ants **always** chose moves that lead further away from the end instead of **preferring** moves that lead further from the exit. This might be tested in a subsequent project. Another change that might improve the times of these solutions would be to use a programming language that allows primitive types, instead of wrapper classes, as generic types in standard list classes.

V. Conclusion

As the results in this report indicate, for the application of finding the longest path in a maze, BS is a better option to use if the ACO and BS setup is done as it is in this paper. It was interesting to be able to visualize the effect of the pheromones as figure 5 indicated with its pheromone concentration map. Another interesting aspect was the reduction in memory usage that BS has over ACO, as this is what is intended for.

References

[1] ANT COLONY. Guest editorial special section on ant colony optimization. *IEEE Transactions on evolutionary computation*, 6(4):317, 2002.

[2] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.