Programmation Project - Ant Simulation

Malte Lichtenberg, Lucas Tittmann

17th May 2012

Contents

1 Introduction **2**

2 Choosing the language **2**

3 Definitions and limitations **3**
3.1 Constructing the simulation . . . 3
3.2 Make it a game . . . 4
3.3 The "traditional" Ant Colony Optimisation Algorithm (ACO) . . . 4
3.4 Aims and limitations . . . 4

4 Algorithm(s) **5**
4.1 Basic movement . . . 5
4.1.1 Defining an utility function . . . 5
4.1.2 Basic micro-optimisation . . . 6
4.2 Stable non-optimal Forms . . . 6
4.3 Optimisation by Randomisation . . . 7
4.3.1 All ants randomise . . . 7
4.3.2 Proportion of ants randomise . . . 7

5 Optimisation of parameters **7**
5.1 General genetic algorithm . . . 7
5.2 Implementation of the genetic algorithm . . . 8
5.3 Visualising different optimisation runs . . . 9
5.3.1 Optimisation without obstacle . . . 9
5.3.2 Optimisation with obstacle . . . 10

6 Gaining speed: Optimisation of the programming environment **10**
6.1 Algorithmic optimisation . . . 10
6.2 Technical optimisation . . . 11

7 Conclusion **12**

8 Appendix **13**

1 Introduction

Ant colony optimisation algorithms are inspired by the behaviour of colony ants whilst searching for food. The goal is to find numerical solutions for problems which are hard to solve algebraically. One observes that ants find quite optimal paths from their colony to a food source which is astonishing regarding their size and perception radius. They achieve this by using their collective intelligence. Every ant is like a micro optimiser which interacts with the other micro optimisers to find optimal paths on the macro level. Furthermore, the ants are not totally deterministic. Hence, if a solution is found, the ants are still looking for an even better solution.

Our goal is to imitate this behaviour and display it graphically. The behaviour of ants and therefore their solving speed depend on different parameters. In order to improve the results of our ant colony optimsation, we want to determine our parameters via a genetic algorithm optimisation. Results could be used to solve e.g. algebraic problems.

2 Choosing the language

At first, we had to choose in which programming language and in which environment we realise our project. We chose Python[1] and PyGame[2] for this purpose. Though most of modern games are written in C++, C#, Objective-C or Flash, we decided in favour of Python for two reasons.

Firstly, though Python is considerably slow in comparison to C++, it has a very easy-to-read syntax and a foolproofed way of coding, so we had not to deal with variable typing or null pointer handling. We wanted to spend as little time as possible in the technical parts in favour of developping our actual model. In keeping with the motto "Premature optimisation is the root of all evil" (Donald Knuth[3]), we added speed only at the critical parts at the end of our developing process (see chapter 6).

Secondly, in contrast to the propretary Flash which is also widely used because of its intuitivity, PyGame is published under the GNU Lesser General Public License (LGPL). That means that the use of its libraries is free and also the source code is freely available. Therefore we are in complete control of our project, meaning that we own full rights to distribute or copy or sell our code as we want to. Furthermore and of greater importance, there is a big and active community on the internet which provides a lot of examples and tutorials, so we had not to spend too much time in coding.

There are also other game developing frameworks for Python. We refer to Unity[4] which also supports a Python dialect called Boo. We chose PyGame as it is free (LGPL) and platform independent whereas Boo uses the .NET Libraries.

[1] http://python.org/
[2] http://pygame.org/news.html
[3] http://en.wikiquote.org/w/index.php?title=Donald_Knuth&oldid=1417149
[4] http://unity3d.com/

3 Definitions and limitations

3.1 Constructing the simulation

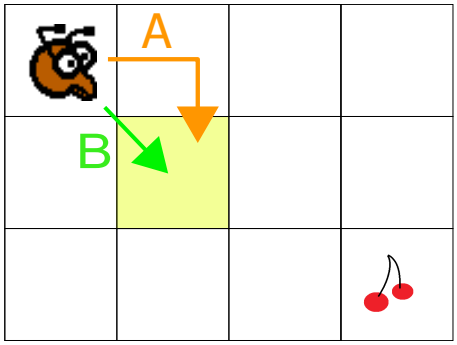
The goal of the project was to do an analysis of the path optimisation behaviour of ants. In order to get a fast understanding of the underlying problem, we first created a framework to visualise the optimisation behaviour. We started with the implementation of the necessary classes. So we built a `Board` where all the action should take place. As the object printed on the board could not levitate in midair, we chose to pin them on the board making `Board` a matrix of `Field` objects. The fields can hold all the other different kinds of items. There are 5 different kinds of items:

- `Food`
- `Barrier`
- `Colony`
- `Ant`
- `Information`

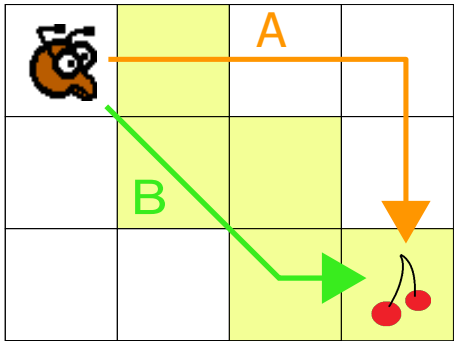
`Food` has only a quantity attribute. A `Barrier` contains nothing and just flags a `Field` where an ant cannot go. Colonies spawn a given number of ants. An Ant has either the objective "food" or "colony" when it already found food. There are several parameters influencing the behaviour of the ant. The ant can be a scouter, than it is influenced by the parameters `stampede` and `straight_forward`. `Stampede` is the probability that the ant makes a random step (uniform distribution over the fields it can see); `straight_forward` is the likelihood of going in the same direction as last round if the ant does a random step. `Velocity` limits the number of steps an ant can make per round (in our cases always set at two) while `range_of_perception` restricts the number of fields an ant can perceive in one direction (in our cases always set at three). An `Information` stores the time when it was created, the objective of the ant creating the information and the distance the ant already walked since its last find.

Due to our decision that every item of pinning the objects to a matrix, we had to deal with a discrete metric (see Fig. 1). If the ant wants to go to the food, it has a range of paths to select from. However, if each step either in x- or y-direction is exactly 1 distance unit, a diagonal step will also need 2 steps. Therefore, all the direct paths to food in the rectangle have exactly the same distance.

That is why we chose to implement another metric were a diagonal step is worth exactly $\sqrt{2}$. We call it square-root-two-metric. As for the limitation of speed to 2 fields per round, an ant can make either 2 steps in horizontal resp. vertical direction or one diagonal step per round. As a result, an ant which goes along just in x- or y-direction is as fast as an ant head along the diagonal way. But the distance is smaller for the ant using the diagonal.



(a) In discrete metric, only path A is possible (taking 1 round, length of 2). In square-root-two-metric, also B is possible (1 round, length of $\sqrt{2}$).



(b) In discrete metric, path A has length 5 as do all the other direct discrete paths in the rectangle. In square-root-two-metric, only the direct diagonal paths over the yellow fields have the minimum length of $2\sqrt{2}+1$.

Figure 1: Comparing the metrics

3.2 Make it a game

To make it a game and to interact with the simulation in realtime, we added a graphical user interface. We built a class `Menu` and `Button`. At the start of the programme, an instance of `Menu` is created which holds the buttons and an instance of `Board`. There are options to place objects, to start a new game in different sizes and to show some information about the game. Furthermore, we constructed a `read_map` method to feed in coordinates for colonies and food as well as the parameters which we wanted to optimise. Accordingly, everything was ready to launch the optimisation via genetic algorithms.

3.3 The "traditional" Ant Colony Optimisation Algorithm (ACO)

The traditional Ant Colony Optimisation Algorithm was proposed by Dorigo (1992)[5]. Its optimisation relies on a preferation of high pheromone densities. The pheromones evaporate over time and thus shorter paths get higher pheromone densities over time. Their algorithms are based on a fixed number of different paths in a complete graph where one path does not affect the other (i.e. an ant on one path can not smell or see any other path). See also Dorigo & Socha (2007)[6].

3.4 Aims and limitations

In contrast to the existing algorithms which apply to problems with a fixed number of paths, we wanted to develop an algorithm which applies to a infinity of paths which may possibly

[5] M. Dorigo, Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italie, 1992.
[6] M. Dorigo & K. Socha, An Introduction to Ant Colony Optimization. T. F. Gonzalez (Ed.), Approximation Algorithms and Metaheuristics, CRC Press, 2007.http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2006-010r003.pdf

affect each other. This would allow us to let the ants walk freely on the board. We decided to try another approach by implementing two pheromones (see above) with quality information rather than quantity information. Our algorithm is based on micro-optimisation within every ant's range of perception as described in the algorithm-section.

Another question is whether or not the ants should have a memory. A memory prevents the ant from visiting fields it already explored. After looking at real ants[7], we dismissed the idea of a memory and just kept `straight_forward` so that an ant runs out of physical reasons roughly in the same direction(for details see chapter 4).

4 Algorithm(s)

4.1 Basic movement

4.1.1 Defining an utility function

In order to find the shortest way on the board, the ant has to consider the information it can percept and optimise its way on a micro level. The ant should not go on fields it cannot go, but always go on fields where it finds its target (i.e. food or colony). It should like fields with pheromones which indicate its target and prefer a pheromone closer to the target over one which is afar. To prevent the ants from clustering together they should have a slight dislike of ants searching for the same target, so they can explore a bigger area in the same time. Furthermore, the ant should prefer a field in front of it over one behind it, so it avoids to run in circles. At last, if an ant has the same preferation for two fields it should pick one randomly. Hence, we modelled our ant as follows:

Let P be the perception of the ant and f be a field in perception range. Thus the optimisation problem of the ant can be described as

$$\max_{f \in P} U(f)$$

Let d be the distance an ant walked before creating the pheromone. Read $\mathbb{1}_x$ like "there is x on the field". The utility of a field is therefore

$$\begin{aligned} U(f) &= u_{\text{aim}} \mathbb{1}_{\text{aim}} + u_{\text{good smell}}(d) \mathbb{1}_{\text{good smell}} + u_{\text{forward}} \mathbb{1}_{\text{forward}} \\ &\quad + u_{\text{barrier}} \mathbb{1}_{\text{barrier}} + u_{\text{bad smell}} \mathbb{1}_{\text{bad smell}} + u_{\text{random}} \end{aligned}$$

Let $u_{\text{aim}}, u_{\text{forward}} \in \mathbb{R}^+, u_{\text{bad smell}}, u_{\text{barrier}} \in \mathbb{R}^-, u_{\text{ random}} \sim \mathcal{U}(0,1)$. $u_{\text{good smell}}$ is a positive, linear function with negative slope.

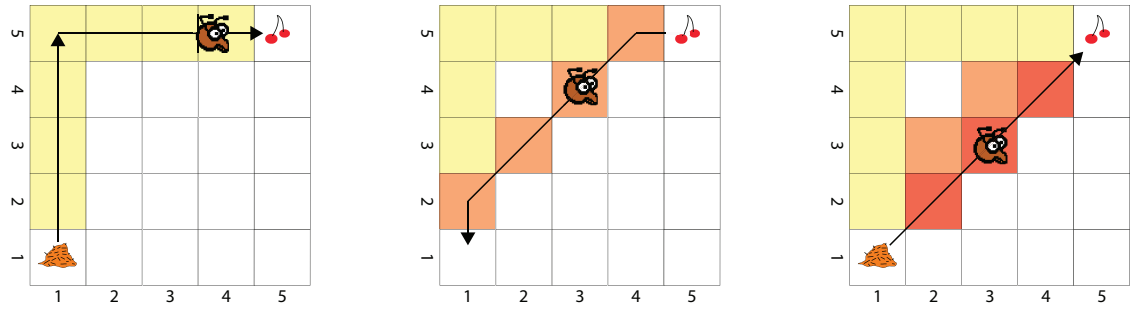
The ants mark each field they visit with a pheromone. It contains information about the number of rounds the ant has been walking since its last departure. This capability of "counting"

[7] e.g. https://www.youtube.com/watch?v=UJo2FbGUwYc&feature=player_embedded

the walked distance is a strong assumption, which is essential for our algorithm. One could interpret this as increasing exhaustion which leads to a decreasing strength of the pheromone.

4.1.2 Basic micro-optimisation

The named implementations already lead to converging results, because the ant finds optimal paths inside its range of perception. In the square-root-two-metric the hypothenus is shorter than the catheti. Hence, if the target is not on the horizontal or vertical line, the ant makes at first a diagonal step towards it. Consequently, the ant straightens out many corners within their range of perception as shown in Fig. 2.



(a) First path (b) Second path (c) Third path

Figure 2: Let the ant find a non-optimal first path (`range_of_perception` = 3) (a). On the way back, when it stands on (4,5) it sees (1,2) and thus approaches it via (3,4) and (2,3) (b). On the way back to the food, the ant sees field (3,4) and approaches it diagonally. Standing on (2,2) the ant already sees the food and follows the optimal path (c).

Consequently, a reduction of the `range_of_perception` to 1 leads to non-convergence for this particular algorithm, because the ants are not capable of straighten out corners and walk the first path over and over again.

4.2 Stable non-optimal Forms

Although the previous algorithm can lead to convergence, there are stable forms which are never optimised. Mark that the existence of non-optimal stable forms does not depend on the range of perception as long as the range of perception is small in comparison to the optimal path. A stable form which corresponds to our algorithm is displayed in Fig. 3

We tried to fix this problem by introducing random behaviour (see Section. 4.3). We observed that these stable forms do not appear if the colony and the food source are on the same diagonal on the board. In contrast, those stable forms most often appear when the food source is close to either the x or y line of the colony. We assume that this is due to the "diagonal first"-approach method (see above).

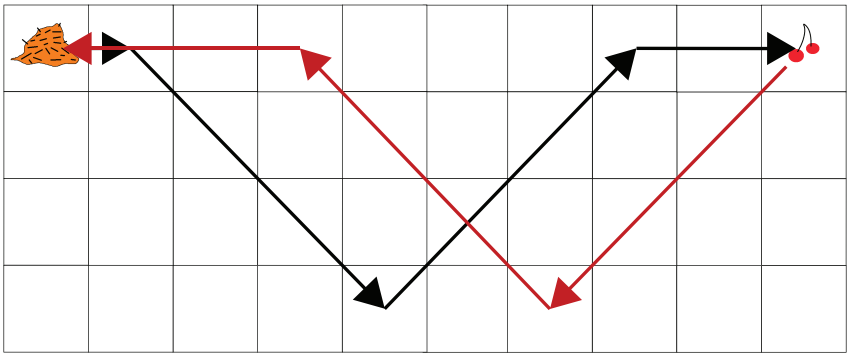


Figure 3: "Stable form" for `range_of_perception` = 3. Black being the initial first path, the ant always walks back the red path and never find the optimal straight path.

4.3 Optimisation by Randomisation

4.3.1 All ants randomise

The imperfection as well as the limited capabilities of ants may actually help to reach a higher rate of convergence for the swarm. Besides, randomizing ants are capable of breaking up stable forms mentioned above. We modeled this imperfection by giving ants a certain probability (`stampede`) of moving randomly. We started with all ants having this particular "ability". This resulted in non-convergence, because every ant deviates several times from the potentially optimal path and counts these deviations in their distance count. Consequently, no ant comes near the optimal path.

4.3.2 Proportion of ants randomise

As a solution, we split the one randomising parameter into two independent ones. `Stampede` is the probability of an ant to choose a field randomly. This ability applies only if the ant is a scout. The proportion of scouts is given by `scouter` If those scouts randomly find better paths, non-scouting ants follow those better marks, thus profiting from these improvements. The problem remains that if the ant deviates too often from the actual way before it finds an optimising solution, its distance count may be already too high to set a competitive pheromone.

5 Optimisation of parameters

5.1 General genetic algorithm

In order to find optimal parameters, we programmed a genetic algorithm. This class of heuristic optimisation algorithms mimics the process of natural evolution. In every generation there are n parameter vectors each forming a basis for one simulation. The program returns the value of the fitness function $F_i(a, r, q)$ where a is the number of ants, r the number of rounds until convergence and q the quality of convergence for a simulation i. While the first generation of parameter vectors is drawn randomly, the following generations result from the respective

preceding generation. For this purpose, the preceding generation is sorted ascendingly by $F(a, r, q)$ and the new parameter vectors for the next generation are created by the following operators:

- **elitism**: choosing the best e solutions of the preceding generation
- **mutation**:slightly changing the best m solutions by adding a small random vector to the initial vector
- **convex combination**:building the mean of the values of the best solution with the co next best solutions
- **crossover combination**: taking several values from the best solution-vector and filling up the missing values from the following cr solution vectors

5.2 Implementation of the genetic algorithm

The main goal being to maximise the rate of convergence, we minimised the number of calculated ants (i.e. `round_count` $*$ `seed_ant_number`) given a fixed distance to the food source. For each simulation the perfect distance from the colony to the food source *opt* is calculated in advance. Let *act* be the distance of the shortest solution found of any ant. Then the game stops either after a maximum number of rounds (`break_cond`) without convergence or when the ants found the optimal way (i.e. when $act - opt < \epsilon$). In real world problems the total time is the crucial factor. That is why we decided to take the total round count as determining criteria and not for example the difference between the round when the first path was found and the round of convergence. This decision, however, brought along a huge random factor in the determination of the fitness value as the ants walk mainly randomly until they find the food source. And so the same combination of parameters result in different values of the fitness function. That is why we let each combination of parameters run m times and calculated the minimum of the m values of the fitness function afterwards.

But before we could start our analysis we had to decide which parameters we wanted to study. The range of perception is pretty small. Making the range of perception bigger automatically leads to better results as an ant optimizes its path independently on its micro level. If now the micro level (perception of an ant) is bigger in relation to the macro level (the board), the path will automatically become more direct . For the same reason, we did not change the velocity of an ant.

We choose to optimise our algorithm over the four previously mentioned parameters: `stampede`, `straight_forward`, `scouter` and `number_of_ants`. The decision to optimise also over the number of ants implicated to let the fitness function depend on the number of ants as well. This seems reasonable as one could interpret `round_count` $*$ `seed_ant_number` as the actual processing time. If calculated in real time, the solution would depend on the speed of the respective computers.

5.3 Visualising different optimisation runs

In attempt to finding a pattern of how the different parameters interact and whether different runs of the genetic algorithm converge to the same solution, we used **R**[8] to visualise different runs of the genetic algorithm on the same plot [9]. With the help of the **R**-Package FactoMineR[10] we effected a Principal Component Analysis (PCA[11]) over the best results of each generation. Thus, we could display the evolution of the 4 parameters on a 2-axis plot as seen in Fig. 4

5.3.1 Optimisation without obstacle

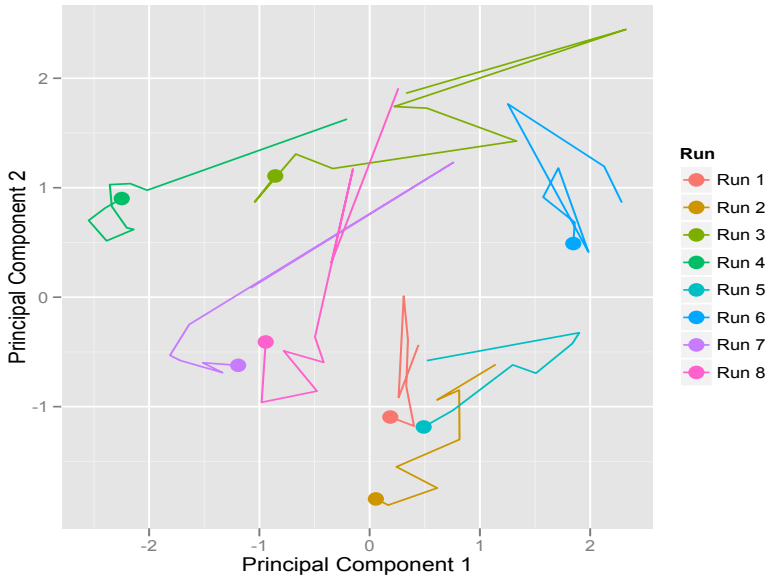


Figure 4: The paths of eight runs of the genetic algorithm without obstacles. The dots mark the respective end points

Obviously, the different runs do not converge to the same local optima of parameter combinations. This may be due to a too little number of generations per run. Another problem is the small consistency of a parameter set. One parameter set often leads to very different fitness values because of the the great impact of randomness in the ant algorithm. However, we noticed that all runs evolute negatively on the second axis. The parameter `straight_forward` has the highest (negative) correlation with the second axis and consequently seems to have a throughout positive impact on the fitness function (see appendix Fig. 7). As our algorithm mainly works through micro-optimisation and the straightening of curves, a high value for `straight_forward` results rapidly in already micro-optimised (straight) paths.

[8] http://www.r-project.org/
[9] Inspired by 'simonraper' http://drunks-and-lampposts.com/2012/04/23/visualising-the-path-of-a-genetic-algorithm/
[10] http://factominer.free.fr/classical-methods/principal-components-analysis.html
[11] http://en.wikipedia.org/wiki/Principal_component_analysis

5.3.2 Optimisation with obstacle

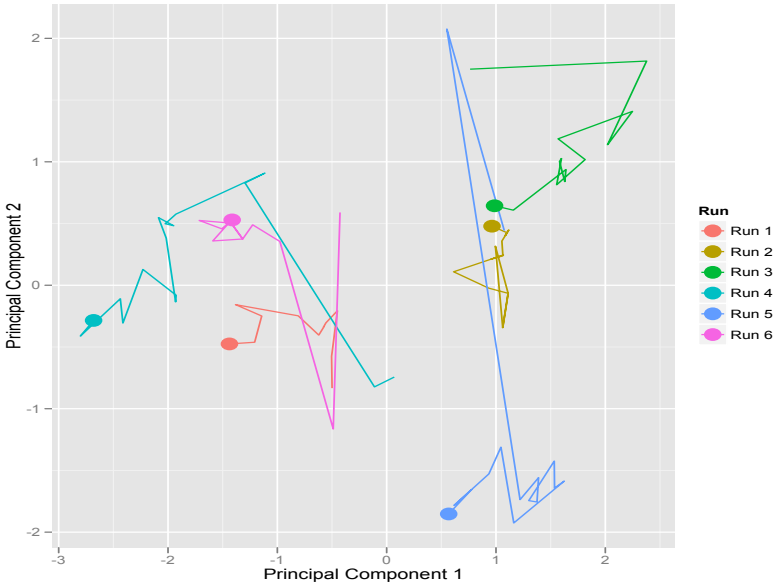


Figure 5: The paths of six runs of the genetic algorithm with an obstacle. The dots mark the respective end points

After we found the intuitive solution that going very straight forward on a plane is quite optimale, we wanted to study the relevance of the parameters on a map with an obstacle in the way (see appendix Fig. 8). To avoid the problem of a stable form, we placed the barrier in a way that both the colony to the edge of the barrier as well as the edge of the corner to the food were on diagonals. As it can be seen in Fig. 5 the test runs did not find a optimal solution. As can be seen in appendix Fig. 7, in this scenario `straight_forward` loses its importance over the other parameters. We guess that either the random part in the path finding in our model is too strong to get results from a genetic algorithm or that we need more generations to see a tendency. However, the big variance of the fitting value seems to indicate that even with a bigger number of generations the parameter optimisation would not converge.

6 Gaining speed: Optimisation of the programming environment

6.1 Algorithmic optimisation

After having a fully working environment for our test series, it was time to speed up our game. At the beginning, the game ran with 30 ants at 25 frames per second (FPS). This was far too slow for our planned test series which would have taken weeks by then.

In order to speed up the simulation we made the ants blinder: after choosing their `target_field` inside their `range_of_perception`, the ants go directly to the `target_field` without

noting better choices during their journey. As the journey to the `target_field` takes up to three rounds, the change reduced the required processing time for an ant by around 50% because the ant does not have to evaluate its surrounding each round. However, this process of evaluating only every 3rd round resulted into even more stable forms as shown in Fig. 6.

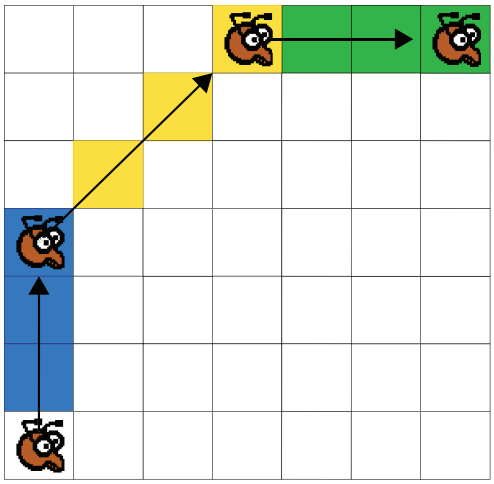


Figure 6: "Stable form" for `range_of_perception` = 3 and evaluating of the best field only every 3rd round.

The form of Fig. 6 is no stable form in the basic algorithm. That is why we dismissed this idea of optimisation.

6.2 Technical optimisation

The first major breakthrough was the use of `DirtySprite` and `RenderUpdates`. Before we started to use these, we updated the whole screen regulary which runs maximum at 28 FPS. With `DirtySprite` it is possible to just update the changing parts so that at the end the screen printing just makes about 15% of the overall time consumed per cycle.

With 50 FPS, it was still too slow to do a parameter optimisation. We now knew that the slow part had to be in the calculations of our programme. We used the profiler Python Call Graph[12] to find the bottleneck (Appendix Fig. 9). The more bright violet a bubble is, the more time the method takes. With that help, we restructed the AI of the ants, unifying the former seperated food and colony search function into one utility function, saving 3 nested for loops. The result was the use of an universal `Ant.preferation_matrix` which holds all the bonuses and penalties for the different fields to which the ant could go. In addition, we deleted the ant's memory. An implemented memory was not only very time consuming but also unrealistic as discussed in chapter 3.4. At the beginning, we had an area effect of bad smells. That means that an ant not only avoided the field which a bad smell but also fields nearby, thus driving the ant forward, away from home and its relatives. Replacing this effect with a the general tendency to run forward and the usage of more ants, we increased the FPS by 80 to 140.

[12] http://pycallgraph.slowchop.com/

In the last part of the optimisation, after we were done with algorithm improvement, we substituted parts of our pure Python code with Numpy[13] calculations and embedded it in the Cython[14] environment thus gaining another 50% acceleration.

7 Conclusion

We implemented an ant colony optimisation on the basis of combined micro-optimisation within a little range of perception. Our algorithm converges when both the colony and the food are on the same diagonal. If not, many stable non-optimal forms emerge which we tried to break up by randomising behaviour of a proportion of ants. At the same time, the randomness of every single simulation made the usage of a genetic algorithm to optimise parameters much harder as expected, as results were not always consistent. Nonetheless, we observed the tendency that walking straightly forward leads to better results on an empty plan. If there is an obstacle on the way, the influence of this paramters is not as clear. A greater number of generations and repetitions per parameter vector would probably improve the results of the genetic algorithm.

[13] http://numpy.scipy.org/

[14] Cython is a tool to translate Python code in C code before runtime. Moreover, it provides the option to declare variables at critical points and switch off the Python wrapper to get the speed of C. A useful tool is an HTML print of the code which is produced while compiling (Appendix Fig. 11). It shows how Pythonic the compilation still is (from dark yellow meaning pure Python to white meaning pure C). For further details we refer to http://cython.org/

8 Appendix

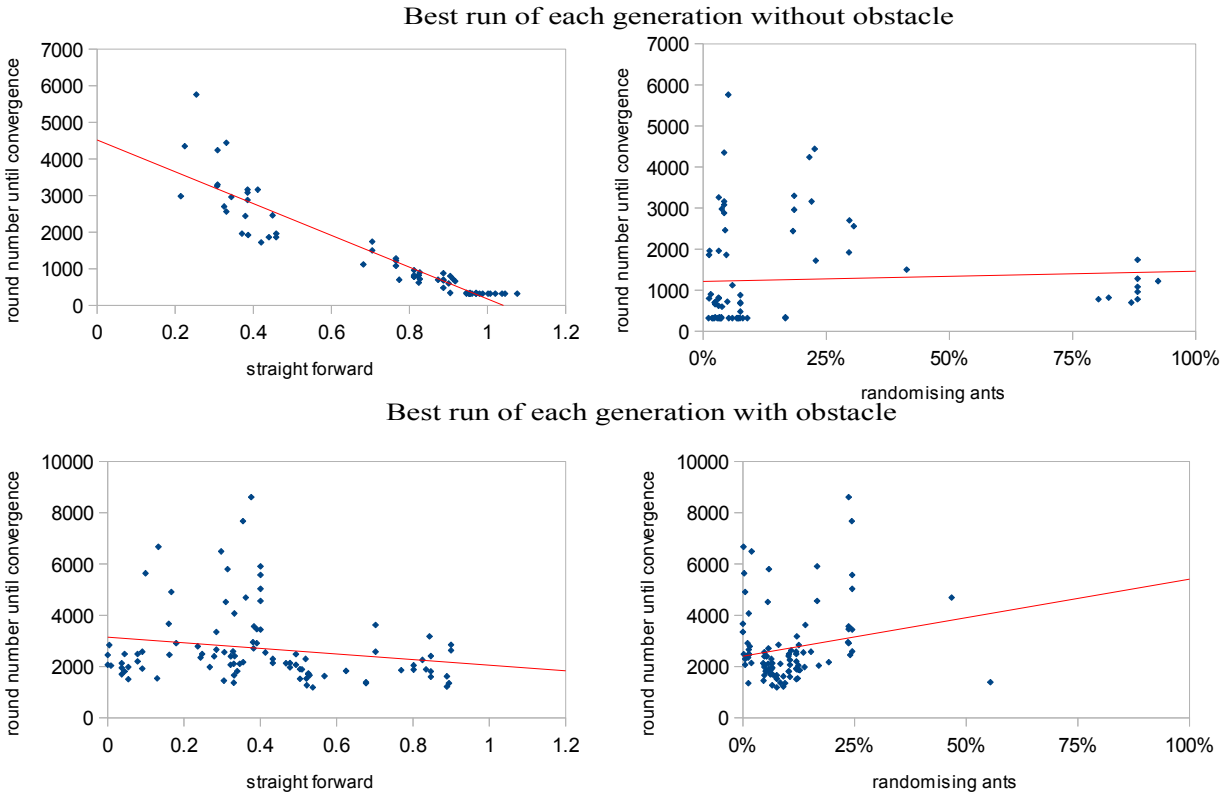


Figure 7: Fitness value as function of `straight_forward` and percentage of randomising ants for the genetic algorithm

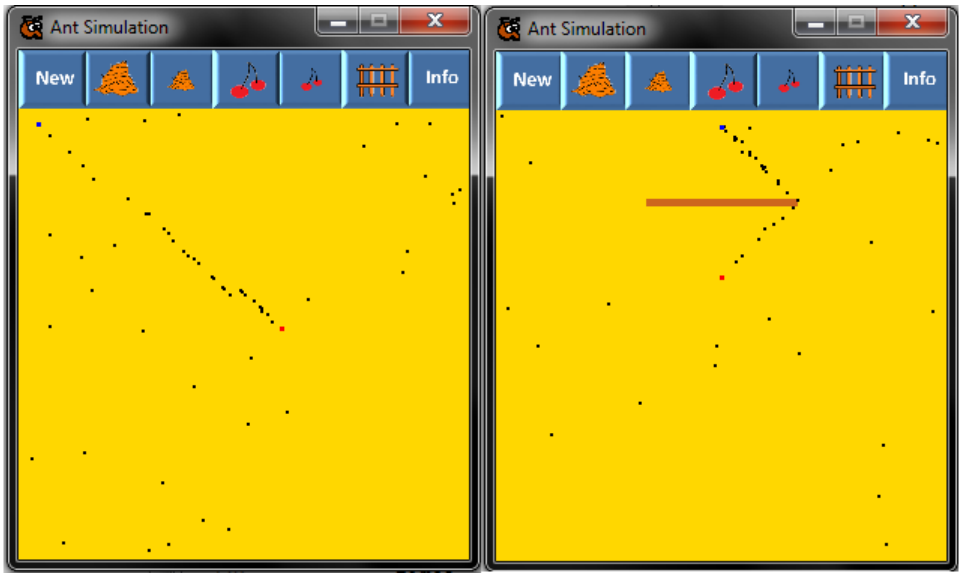


Figure 8: Screenshots of simulations with and without obstacle

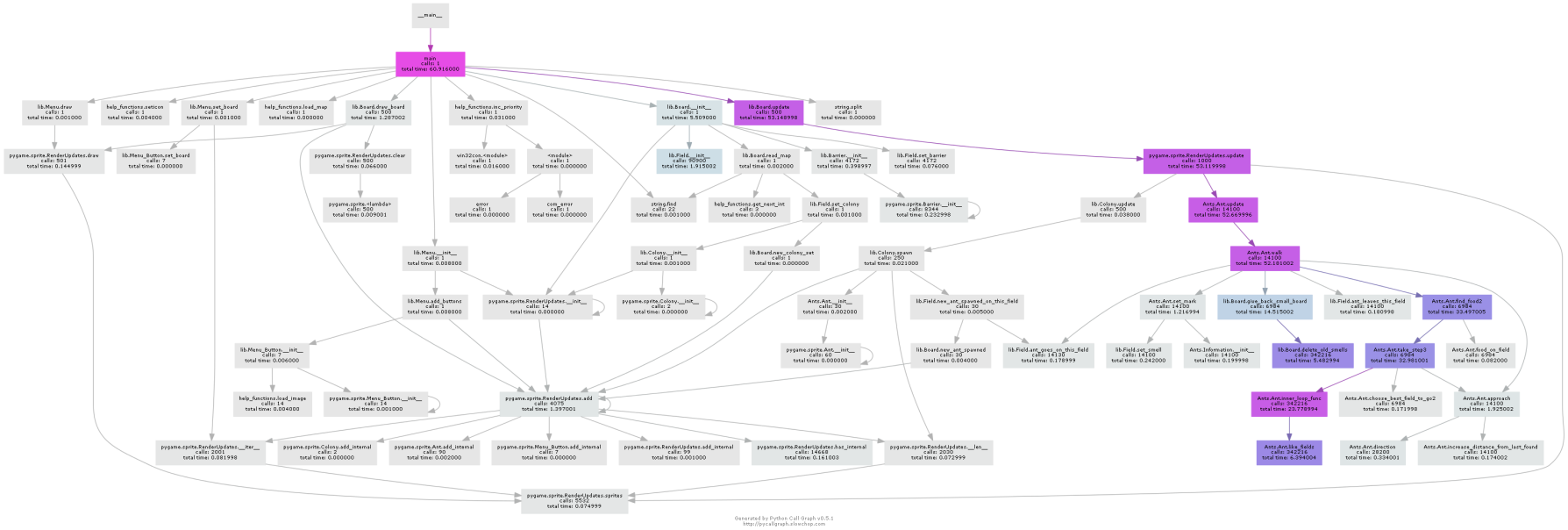


Figure 9: Organigram of the programme without an area effect of bad smells. The purple and violet methods consume the most time. overall time: 61 s, time in `Ant.like_fields`: 6 s

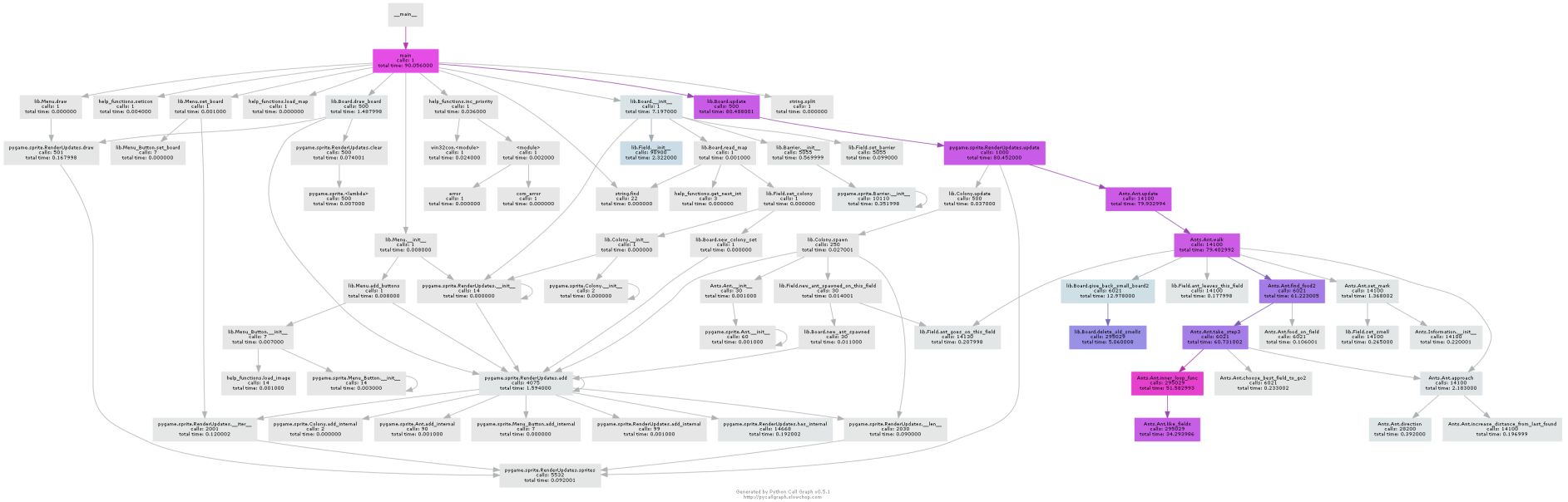


Figure 10: Organigram of the programme with an area effect of bad smells. In comparison to Fig. 9, the programme is 50% slower, all the time consumed in the method `Ant.like_fields`.
overall time: 90 s, time in `Ant.like_fields`: 34 s

```
8: import numpy as np
9: cimport numpy as np
10: FLOAT = np.float
11: ctypedef np.float_t FLOAT_t
12: import cython
13:
14: cdef inline int max_int(int a, int b): return a if a > b else b
15:
16: class Ant_mod():
17:
18:     @cython.wraparound(False)
19:     def ctake_step (self):
20:         cdef np.ndarray[FLOAT_t, ndim=2] preferation_matrix = self.preferation_matrix
21:         cdef list perception = self.perception
22:         cdef list col, list_of_good_smells
23:         cdef str aim = self.aim
24:         cdef int index_smells_good = self.index_smells_good
25:         cdef int index_smells_bad = self.index_smells_bad
26:         cdef int p_range = len(perception)
27:         cdef int x, y, obj_looking_for
28:         cdef int i, dist, minimum
29:
30:         for x in xrange(p_range):
31:             col = perception[x]
32:             for y in xrange(p_range):
33:
34:                 # if ant cannot go on the field
35:                 if col[y].barrier:
36:                     preferation_matrix[x,y] += -1000000
37:                     continue
38:
39:                 # is aim near
40:                 obj_looking_for = 0
41:                 if aim == "food" and col[y].food is not None:
42:                     obj_looking_for = 1
43:                 elif aim == "home" and col[y].colony is not None:
44:                     obj_looking_for = 1
45:
46: ################################################################################
47: ###################### EDITING FROM THIS POINT ON ##############################
48: ################################################################################
49:
50:                 # if object the ant is looking for is in sight
51:                 if obj_looking_for == 1:
52:                     preferation_matrix[x,y] += 10000
53:                     continue
54:
55:                 # if food mark is near
56:                 list_of_good_smells = col[y].smell[index_smells_good]
57:                 if len(list_of_good_smells)>0:
58:
59:                     minimum = list_of_good_smells[0].distance
60:
61:                 # Possibility: if following lines are outcommented, the ant
62:                 # just takes the first smell to decide about the field
63:                 # if they are executed, the ant looks for the best smell on the field
64:                     for i in xrange(len(list_of_good_smells)):
65:                         dist = list_of_good_smells[i].distance
66:                         #print dist,
67:                         if dist < minimum:
68:                             minimum = dist
69:                     #print "minimum:",minimum, x, y
```

Figure 11: Output of Cython while compiling the *.pyx file.