AGAR Report

Clément ANTHEAUME, Camille-Amaury JUGE

November 2020

Contents

1 Introduction **3**
1.1 The subject . . . 3
1.2 A deeper dive on the problem . . . 3
1.3 Organization through the problem . . . 5

2 Naive Approach **5**
2.1 Approximating ratio of reward over distance . . . 5
2.2 Data visualization for a better understanding . . . 11
2.3 Improving naive approach through data visualization . . . 13

3 Structured Approaches **15**
3.1 Linear Programming and Solvers . . . 15
3.2 Reasonable Linear Programming . . . 17

4 Reproducibility **18**
4.1 Languages, Frameworks, Tools . . . 19
4.2 Where to get the sources . . . 19

1 Introduction

In this section, you will find everything about the project and its details.

1.1 The subject

The AGAR project was introduced by Hoël Le Capitaine, a french professor of the Data Science Master Degree, during the Classification And Representation Learning course. Thus, this work can't be used without having our permission first.

We will briefly present the project : Let's consider an environment where an agent has to move through a grid to find and eat food. This grid has a size $l * L$. Let's now focus on the food, for this purpose we will label as n the number of food bubbles. Each food bubble is located in a cell of the grid. We also know the fact that in a cell, it can't have two or more food bubbles in it, meaning that $n \leq l * L$. In order for the agent to eat a bubble of food, it has to travel from his current cell to the cell which contains this bubble. But this movement costs time, at least there is a distance between the two coordinates which can be computed with the euclidean distance (the time and distance unit are the same).

$$EuclideanDistance([x_{source}, y_{source}], [x_{destination}, y_{destination}]) = \\ \sqrt{(x_{source} - x_{destination})^2 + (y_{source} - y_{destination})^2}$$

Moreover, each bubble has a random size affected which represents the reward of eating it. Thus, the following problem raised is : Considering a maximum time T, the agent has to maximize the sum of rewards of the eating cells without travelling much than T.

Our aim is to find different approaches from the most intuitive ones to the more complex and structured ones to optimize the problem.

1.2 A deeper dive on the problem

Now that the problem is defined, we can discuss in details about the complexity and findings of the subject. Firstly, the example delivered to us contains the following parameters :

- $l \simeq 20000$, $L \simeq 20000$, then the grid has a size of 400000000 possible coordinates. Notice that, the coordinate range are $[-100.00, 100.00]$ for both axis.
- the number of cell $n = 10.000$. Meaning that the grid will be highly sparse.

With this medium values, we already see that we will run into a problem of complexity. So the first step is to find an appropriate data-structure to represent the problem. One solution, which will to reduce the initial grid is to see the problem as an undirected graph structure :

- Note G, the graph which has n vertices, and m edges.
- Note V the set of vertices in G. Each vertex v represents a food bubble, then $v \in V, |V| = n$. v has the following attributes :
  - v_{reward}, the size/reward given by the eating action
  - v_x, the coordinate on the first axis
  - v_y, the coordinate on the second axis
- Note E the set of edges in G. Each edge $e_{v_i,v_j} \in E$ represents the distance or time consumed to go from an vertex v_i to a vertex v_j, with $v_i, v_j \in V$. Since our problem doesn't state a limitation of movement, it exists $\frac{(n-1)*(n)}{2}$ edges, meaning that we have a complete graph.

Hence, with a little bit of optimization, the initialization algorithm is :

Algorithm 1 Initialize Data Structure

```
Require: $n$ : the number of vertices.
Require: $Data$ : the vector of containing each food bubble object containing
  its x coordinate, y coordinate, reward.
  $G \leftarrow Matrix[n, n]$ fill with 0.
  $R \leftarrow Vector[n]$ fill with 0.
  for $i$ from 1 to $n$ step 1 do
    $R[i] \leftarrow Data[i].reward$
    for $j$ from 1 to $n$ step 1 do
      $G[i, j] \leftarrow \sqrt{(Data[i].x - Data[j].x)^2 + (Data[i].y - Data[j].y)^2}$
    end for
  end for
```

In our example, this gives us a 10 000 vertices graph and a 100000000 edges graph. So from a 400000000^2 Graph we obtained a 10000^2 Graph, which considerably reduces the complexity. Since our data structure is now defined, we will try to take a look at the problem complexity. Since our graph is complete, it exists a number of path which is not linear in the size of our data :

- Considering the first vertex, in our example, there are 9999 edges possible.
- Next step, on the chosen second vertex, there are 9998 (we will study the case where we don't want to visit twice a vertex) ... and so on, until you reach the last vertex.
- So, the number of path in our graph G is $(n - 1)!$, which is totally unreasonable.
- Notice that, this is the maximum path allowed if we had a maximum Time $T = \infty$, Thus with $T = 10000$. this probably reduces a lot of the possible paths. But it is still unreasonable to explore each solutions.

We easily conclude that we will probably have to use a heuristic or a method which will try to approximate a good solution.

1.3 Organization through the problem

In the first place, we are going to give a try to what we will call "naive approaches". This leads us to a starting point, which will serve as a reference score for the future improvements. Nextly, we propose a improved naive approach following visualization of our data. Finally, we discuss about some of the different structured approaches that we use to try to solve the subject.

2 Naive Approach

In this section, we will introduce basic approaches and improved naive approaches in order to solve the problem.

2.1 Approximating ratio of reward over distance

One of the most generic way to deal with such an issue, in the first time, is to think of really intuitive and easy to implement approaches. It enables us to have a first statement of the problem and a base score to overcome with more structured approaches. So firstly, we have to plot the data in order to see how the points are distributed.

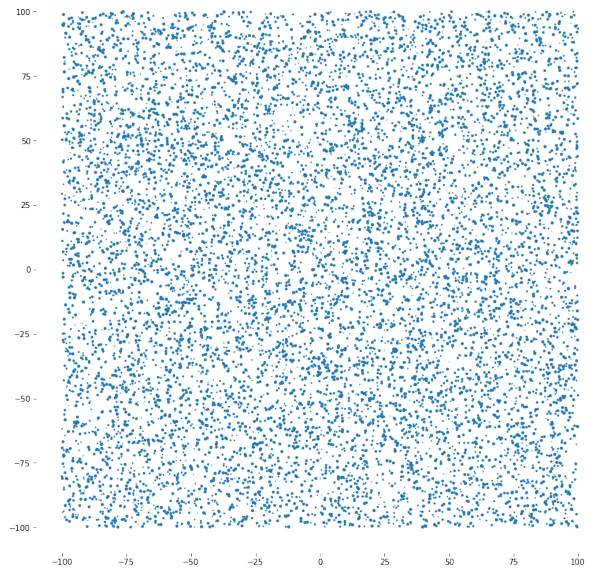

Figure 1: Initial data plot

The figure 1 represents the food bubble distributed over the two dimensional space. Each cell has a size which scales with the reward it offers. As we can easily see, even if there is a little bit of bias/randomness, it seems that the cells are normally distributed. Thus, distances are quite short between each food bubble (they are well distributed over the space, there are no huge area without cells) and we can expect our naive algorithm to eat at least some thousand of points. In order to explore our intuition, we propose a first approach that we call "Nearest Non-Null Neighbor".

Algorithm 2 Nearest Non-Null Neighbor

```
Require: $G$ : Edges distance Matrix.
Require: $R$ : Reward vector, $T$ : The maximum time given.
Require: $coord$ : the initial coordinate.
Require: $Data$ : the vector containing each food bubble object containing its
  x coordinate, y coordinate, reward.
Require: $n$ : the number of vertices.
Require: $ArgMin(x)$ : function which returns the position of the minimum in
  a vector.
  $Path \leftarrow$ empty list
  $time \leftarrow 0.$
  $reward \leftarrow 0.$
  $initDistances \leftarrow$ vector of size $n$ fill with 0.
  for $i$ from 1 to $n$ step 1 do
    $initDistances[i] \leftarrow \sqrt{(coord.x - Data[i].x)^2 + (coord.y - Data[i].y)^2}$
  end for
  $choice \leftarrow ArgMin(initDistances)$
  if $time + initDistances[choice] < T$ then
    add $choice$ in $Path$
    while $time + initDistances[choice] < T$ do
      $time \leftarrow time + initDistances[choice]$
      $reward \leftarrow R[choice]$
      $R[choice] \leftarrow 0.$
      $distances \leftarrow G[choice]$
      for $i$ from 1 to $n$ step 1 do
        if $R[i] = 0.$ then
          $distances[i] \leftarrow 0.$
        end if
      end for
      $choice \leftarrow ArgMin(distances)$
    end while
  end if
```

Applying this algorithm to our instance of the problem gives us a final score of 327584 with 6538 cells eaten. Which is a really good score for a first try. Refer yourself to the Reproducibility part in order to get the proof of work.

This ensures us to have a reasonable score since we are greedily choosing the less distant vertex.

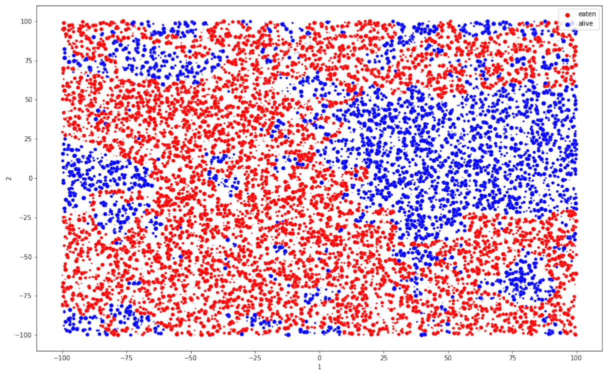

Figure 2: Eaten cell of NNN metric (red = eaten)

Nextly, we have to generalize the metric to give a better score on other instances of the problem, since the previous algorithm was not taking into account the reward of the cells. Then, we propose several Metrics which tends to approximate the optimization function. We use the following template and detail the best metric we find.

Algorithm 3 Template Naive Metric

```
Require: $G$ : Edges distance Matrix.
Require: $R$ : Reward vector, $T$ : The maximum time given.
Require: $coord$ : the initial coordinate.
Require: $Data$ : the vector containing each food bubble object containing its
  x coordinate, y coordinate, reward.
Require: $n$ : the number of vertices.
Require: $ArgMin(x)$ : function which returns the position of the minimum in
  a vector.
Require: $ApplyMetric(x, v, r)$ : function which applies a metric over a vector
  and returns it.
Require: $METRIC$ : The metric used
  $Path \leftarrow$ empty list
  $time \leftarrow 0.$
  $reward \leftarrow 0.$
  $initDistances \leftarrow$ vector of size $n$ fill with 0.
  for $i$ from 1 to $n$ step 1 do
    $initDistances[i] \leftarrow \sqrt{(coord.x - Data[i].x)^2 + (coord.y - Data[i].y)^2}$
  end for
  $choice \leftarrow ArgMin(initDistances)$
  if $time + initDistances[choice] < T$ then
    add $choice$ in $Path$
    while $time + initDistances[choice] < T$ do
      $time \leftarrow time + initDistances[choice]$
      $reward \leftarrow R[choice]$
      $R[choice] \leftarrow 0.$
      $distances \leftarrow G[choice]$
      $distances \leftarrow ApplyMetric(METRIC, distances, R)$
      $choice \leftarrow ArgMin(distances)$
    end while
  end if
```

Algorithm 4 Root Square Difference Ratio Metric

```
Require: $distances$ : a vector of distances.
Require: $R$ : a vector of rewards.
Require: $n$ : the number of vertices.
  for $i$ from 1 to $n$ step 1 do
    if $R[i] = 0.$ then
      $distances[i] \leftarrow 0.$
    else
      $distances[i] \leftarrow \frac{\sqrt{R[i]} - \sqrt{distances[i]}}{distances[i]}$
    end if
  end for
```

This leads to a score of 335057 with 5744 eaten cells.

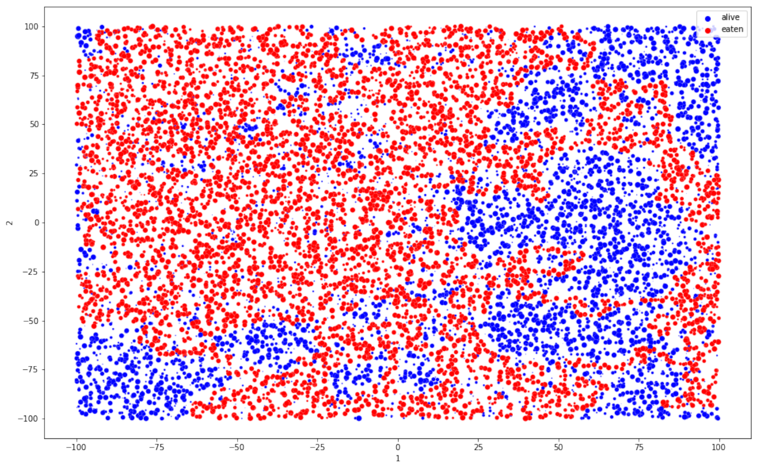

Figure 3: Eaten cell of RSDR metric (red = eaten)

Algorithm 5 Logarithm Difference Ratio Metric

```
Require: $distances$ : a vector of distances.
Require: $R$ : a vector of rewards.
Require: $n$ : the number of vertices.
  for $i$ from 1 to $n$ step 1 do
    if $R[i] = 0.$ then
      $distances[i] \leftarrow 0.$
    else
      $distances[i] \leftarrow \frac{\log R[i] - \log distances[i]}{distances[i]}$
    end if
  end for
```

This leads to a score of 336191 with 6221 eaten cells.

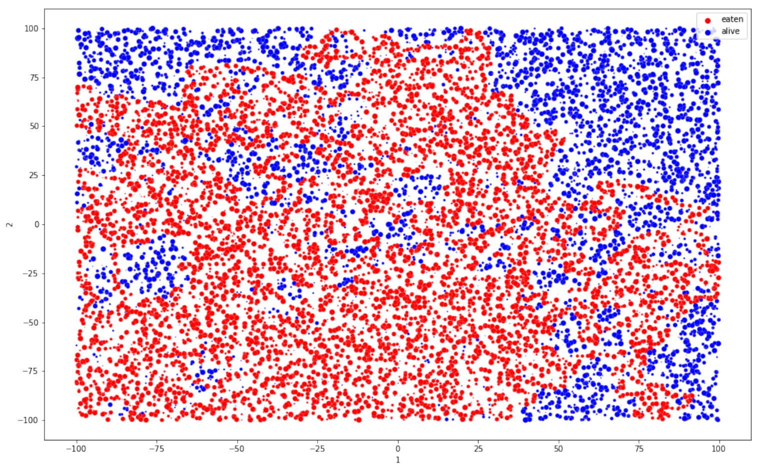

Figure 4: Eaten cell of LDR metric (red = eaten)

Algorithm 6 Reward over Distance^3 Ratio Metric

```
Require: $distances$ : a vector of distances.
Require: $R$ : a vector of rewards.
Require: $n$ : the number of vertices.
  for $i$ from 1 to $n$ step 1 do
    if $R[i] = 0.$ then
      $distances[i] \leftarrow 0.$
    else
      $distances[i] \leftarrow \frac{R[i]}{distances[i]^3}$
    end if
  end for
```

This leads to a score of 340915 with 6206 eaten cells.

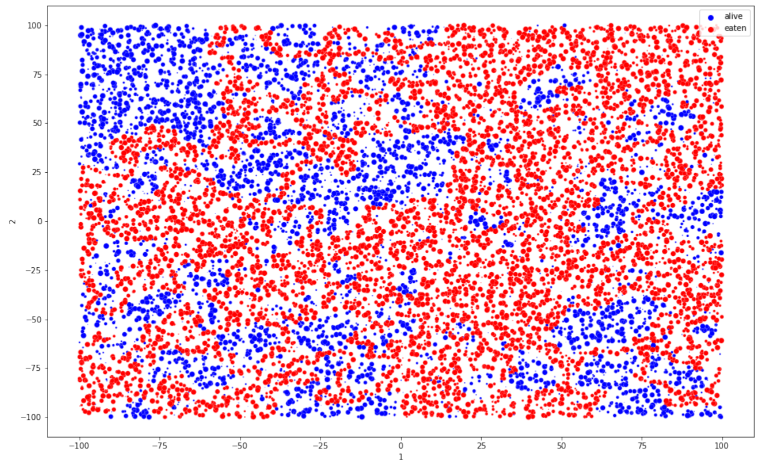

Figure 5: Eaten cell of RD3 metric (red = eaten)

The results are a little bit better but we find ourselves stopped around those scores. There are no major improvements by only changing the metric. Regarding the plots, those metrics are really dependent on the neighborhood and take really different paths while we are waiting for quite stable choices.

2.2 Data visualization for a better understanding

Following the last statement, we deeper explore the data and get a new intuition. Firstly, we have the idea to try k-means clustering on the data in order to group them by proximity. We won't show the result in this report, but you can refer yourself to the source and to the Reproducibility part to see more about it.

Instead, we will focus on the distribution of rewards. We show that rewards are equally ranged over some levels.

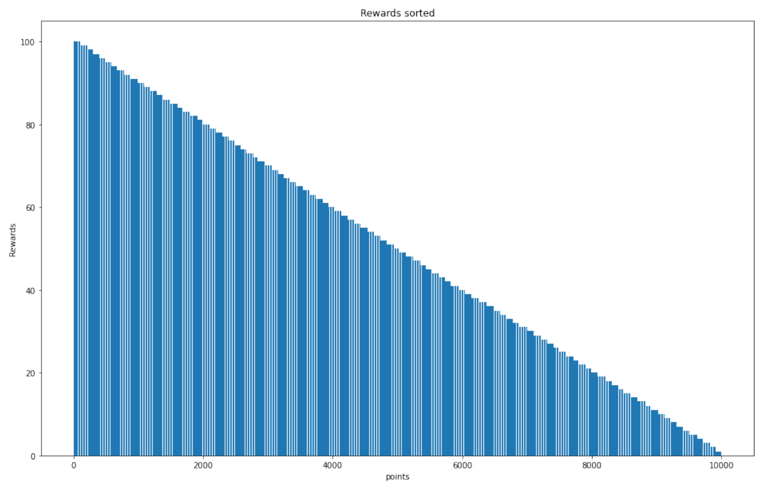

Figure 6: Rewards Decrease Sorting

Then, our first intuition seeing this plot was that since there are as many "10" valued rewards than "100" valued, then our agent has to focus on the higher rewards. Moreover, since the neighborhood was really important in the last metrics, loosing time on low rewards is not a good strategy. This second figures confirms us what we have stated previously.

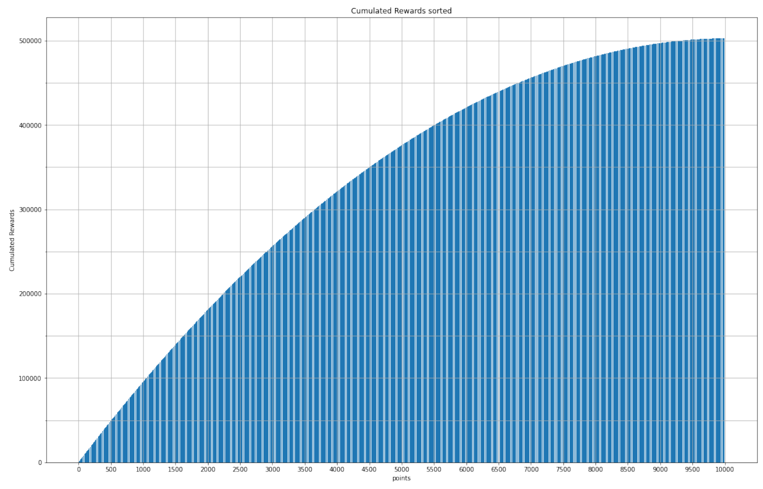

Figure 7: Cumulative Sum of Rewards Decrease Sorting

As we can observe, the cumulative decreasing sum of rewards acts as Root Squared Weighted Function. By this, we mean that the more we focus on higher rewards the higher the score will be. It means that the lower rewards can be considered as a lost of time. In order for the function to be linear, the distribution should have contained more lower rewards than higher rewards (equity distribution), but here they are equally distributed.

A last observation, which could help further improvements is to consider low reward areas.

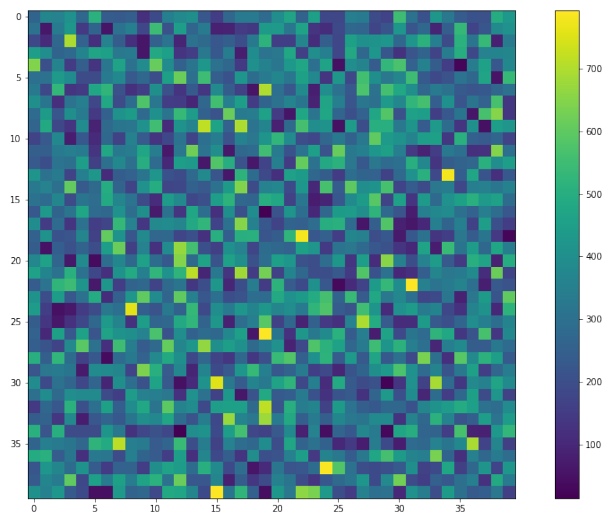

Figure 8: rewards sum Area Heat map

Sometimes it is better to eliminate low-rewards area with metrics based on neighborhood. We don't implement it with our metrics but it could be a future work to do.

2.3 Improving naive approach through data visualization

Regarding the last observations, we decided to put a threshold on the initial data which will only keep the vertex with rewards higher than the threshold. In order to keep ensuring that we won't do too much travel per iteration, we will keep the Nearest Non-Null Neighbor Metrics. One thing to mention is that, since this metric is really dependent of the neighborhood, we decided to iterate over a range of values for the threshold from 0 to 49. $Threshold = 0$ corresponds to the NNN metric without any change in the data. Here is a result of those iterations.

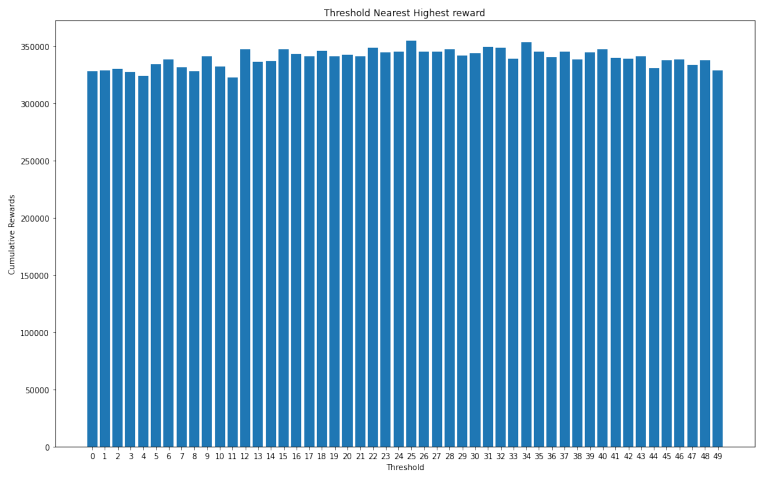

Figure 9: Threshold and NNN metric

The previous RD3 metric has been beaten several times with different thresholds. The better one is $Threshold = 25$ with 354453 with 5646 eaten cells. By eating 600 cells less than RD3, we obtained a score 15000 higher. It confirms our intuition.

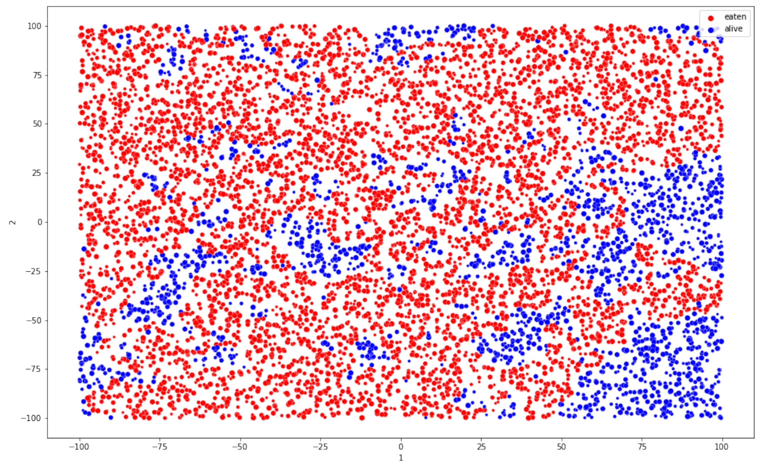

Figure 10: Eaten cell of T-NNN metric (red = eaten)

Last but not least, we still want to put the light on the fact that those metrics will work well in this kind of distribution. If the distribution totally changes, we would have to visualize again to adapt the strategy. Therefore, those strategies are not aimed to generalize on all instances of the problem and this raises a new pitfall. Then, we propose a new approach which can be generalized to all instances and is proven to give good or optimal result.

3 Structured Approaches

In this section, we will introduce one of the numerous structured approaches we thought of. Notice this is just a little part of our work, since a lot of our tries didn't succeed of didn't let us the time to implement them all.

- Genetic Algorithms provide good approximation of those kind of problems. The closest related one that we try to implements was Ant Colony to solve the Traveler Salesman Problem which is quite close to this one in some ways : By using a threshold we could only keep the high rewards and find a path that goes to every vertex. But there were two problems to it :
  - We use ACO-Pant library for python but the customisation doesn't offer what we want to customize the problem.
  - The major issue of the TSP problem is that the path is optimized to return to the starting point. Meaning that, we have a cycle which is not what we are looking for (rather a path which does not come back to the initial vertex).
- Monte Carlo Simulation and Randomness weighting on distance. This doesn't lead to good solutions and thus we abandoned this alternative.
- Deep Reinforcement Learning was also cast aside due to the fact that there are too much states and actions possible. Leading to an increasingly connected network which takes too much time to learn.

We will now focus on Linear Integer Programming.

3.1 Linear Programming and Solvers

Linear Programming Solvers offer the opportunity to find the optimal solution quicker than if we had to implement it and moreover, they are really understandable without having to dive into the programming field. Thus, we propose the theoretical problem translated to ILP.

- Note T the maximum time allowed.
- Note G the initial graph obtained by the preprocessing step. E the set of edges of G and V the set of vertices.
- Note $x_{a \to b}$ the variables with $a, b \in V$ and $(a, b) \in E$. These variables represent the activation or not of the edge m. Thus, there are binary variable $\in 0, 1$.
- Note $x_{init \to n}$ the variables which represent edges from the initial coordinate to all the possible vertices.

- Note $d_{a \to b}$ the distance associated to an edge $(a, b) \in E$.
- Note $d_{init \to n}$ the distances associated to edges from the initial coordinate to all vertices.
- Note r_n the reward associated to an edge $n \in V$.
- Note $f(x)$ the optimization function which takes the variables as argument.
- Note i the vertex which represents the end of the path.
- Note $S \in V$ all the partition of vertices of V at least of size 2 from which we obtain the following edges $E_S \in E$ which has an origin and a destination $\in S$.

Therefore we obtain the following ILP :

$$\begin{aligned}
OptimizationFunction &= max.f(x) \\
&= max. \sum_{1}^{n} \Big(\sum^{(j,n) \in E | (n,n)} x_{j \to n} \Big) * r_n \qquad (1)
\end{aligned}$$

$$\begin{aligned}
&u.c: \\
&x_{a \to b} \in 0, 1 && (2) \\
&\sum_{1}^{n} init \to n = 1 && (3) \\
&\sum^{j \in V | i} x_{j \to i} = 1 && (4) \\
&\sum_{1}^{n} \Big(\sum^{(n,j) \in E | (n,i)} x_{n \to j} - \sum^{(j,n) \in E | (i,n)} x_{j \to n} \Big) = 0 && (5) \\
&\sum_{1}^{n} \Big(\sum^{(n,j) \in E | (n,i)} x_{n \to j} + \sum^{(j,n) \in E | (i,n)} x_{j \to n} \Big) \leq 2 && (6) \\
&foreach(S \in V): \sum^{(j,j') \in E_S} x_{j \to j'} \leq |S| - 1 && (7) \\
&\sum^{(j,j') \in E} x_{j \to j'} * d_{j \to j'} + \sum^{j \in V} x_{init \to j} * d_{init \to j} \leq T && (8)
\end{aligned}$$

Let's now explain it :

(1) The optimization function is a sum of all activation variables which goes to the vertex n times the reward of going to n. This formulation supposes that we can't visit two times the same vertex.

(2) Boolean variables, 0 means not-activated, 1 activated.

(3) The sum of all edges coming from the initial node has to be 1. It means that there is only one edge activated which represents the beginning of our path.

(4) The sum of all edges going to the final node i has to be 1. It means that there is only one edge activated which represents the end of our path.

(5) For each vertex different from initial or final ones, the difference between the number of edges coming from and going to the vertex should be equal to 0. It means that when we go to a vertex we have to go away nextly (impossible to finish or begin on those nodes). But we can still visit a vertex more than once.

(6) For each vertex different from initial or final ones, the addition between the number of edges coming from and going to the vertex should be lower or equal to 2. It means that we can't visit more than once a vertex.

(7) Each partition of edges which forms a cycle is prohibited. Then, this ensures having a unique path.

(8) This ensures that all the edges activated take less or equal time to T.

This ILP is proven to give the optimal solution if it exists. But one thing to mention is that we have to do this n times corresponding to the number of $i =$ final nodes we can have in our graph. Thus, we just have to compute the ILP for each final possible vertices and taking the max score of all of these.

This leads us to two major issues :

- In practice, in a large graph like the one of our instance, the solvers existing are really struggling to find a solution in a reasonable time.
- It is impossible to compute all the partitions $S \in V$ since there is an exponential number of cycles in a complete graph. Thus, it leads to an exponential number of constraints and then ILP won't work correctly.

We tried to implement it with success, but due to runtime necessities, we can't provide the results. The only difference is that we don't give any constraint of cycles in the first place. Then we solve an unknown times the ILP problem. At each step, we have cycles, that we add as constraint to the last ILP and solve again until there are no cycle left in the solution given. This is far more efficient but still impossible to correctly compute (the first iteration on the instance takes on average 2 minutes to find a first solution, then when adding the constraint of cycles, the second iteration doesn't find a solution within 1 hour, we didn't try more).

3.2 Reasonable Linear Programming

Following the constraints raised above, we propose a simplified solution which will use ILP with greedy behavior. We expect to have a correct solution.

One thing to mention, is that we firstly decide to remove most of the edges in the graph to have a really less complicated structure and optimize time. We decide that for each vertex, we will choose the 3 nearest vertices and keep the edge between those (notice that, these are undirected edge, leading to 6 directed edges). And this is quite a good approximation since we see above that the best behavior we could have lastly is to only consider nearest neighbors.

Nextly, we define subsections of the plots. Meaning that for each $5 * 5$ area in our plan, we create a ILP problem with only the food cells that are in this area. Then, we fix an amount $T' = T/(200 * 200/5 * 5)$ which will be the maximum time spent in a subarea. We greedily choose the next subarea to visit with a metric of cumulated sum of rewards over distance and finally choose three of the closest vertices in the next subarea to apply the ILP (they represent the end of path). We take the max of these three solutions and keep iterating until the time is over.

As we expected, the iterations are really quick (a little less than naive metrics) and provide an acceptable score, which in fact is not better than the previous ones : 308048. We try several improvements in our decision (increasing the number of vertices chosen for each next subarea, changing the ratio) but this was the better solution.

So why is it not working as well as we thought ?

– The subareas acting as clusters don't take into account the correct distribution and clusters of cells. In fact, doing a grid cuts some clusters and then doesn't enable to visit correctly those clusters.

– One improvement is to consider already visited areas in order not to go back there. When we reach the end of our time, the agent is always struggling in low reward areas.

– It's a greedy approach, so in all the cases this doesn't ensure us to have a great solution.

How would we improve if we had more time ? Let's consider again the full ILP problem, if we are to do only a few iterations for cycle constraints and obtain a number of cycles inferior to 10, we can concatenate the path with those cycles by removing some edges and adding others. We will probably have to delete some other edges due to distance overflow. But in the same time, this solution will be close the optimal one. For example, with the first iteration we obtained a score of $\simeq 410000$ but with $\simeq 1300$ cycles. Then, if we obtain a solution which has such a score but with only a number of cycles that is really low, we will obtain a score really near to the initial score with cycles.

4 Reproducibility

This section is dedicated to the specificity of our implementation.

4.1 Languages, Frameworks, Tools

Languages :

- Python 3.7
- Julia (only for ILP optimization)

Frameworks-Libraries :

- Python :
  - ACO-Pants
  - Numpy, Pandas
  - Matplotlib, Seaborn
  - Pulp
- Julia :
  - JUMP

Tools :

- Google Collaboratory/Anaconda/Jupyter Notebook

4.2 Where to get the sources

Go to the following link : https://github.com/camilleAmaury/ReportAGAR

Files to execute related to the different parts :

- Naive Metrics : Python/NaiveApproach.ipynb
- Data Visualization : Python/DataVisualization.ipynb
- Improved Naive Metrics : Python/NaiveApproachImprovement.ipynb
- Improved Naive Metrics : Python/NaiveApproachImprovement.ipynb
- ILP : Python/ILP.ipynb
- ILP-Greedy : Python/ILP-Section.ipynb

CSV results are stored in the CSV folder.