KTH Royal Institute of Technology

DD2380

Artificial Intelligence

Year 2019 - 2020

Rescuing Matt Damon from Mars

Group Project

Authors, Group 12 :

Prashant Maheshwari	pramah@kth.se
Magnus Pierrau	mpierrau@kth.se
Anirudh Seth	aniset@kth.se
Matej Buljan	buljan@kth.se
Jackson Waschura	waschura@kth.se

Professors :

Iolanda Dos Santos Carvalho Leite

Jana Tumová

GROUP 12

Matej Buljan

Prashant Maheshwari

Magnus Pierrau

Anirudh Seth

Jackson Waschura

October 25, 2019

1 Abstract

In this project we create a simulated environment within which we test three planning algorithms: one using a genetic algorithm (GA), one using ant colony optimization (ACO), and one using a greedy approach. We attempt to measure the sensitivity of these methods to uncertainty as well as outline when each algorithm may outperform the others.

Our simulated environment and the task we evaluate these planning algorithms on are both inspired by the 2015 film, The Martian. The environment is a grid world representing the Martian surface and containing objects which represent Matt Damon and valuable scientific equipment. The task is to plan a path for a rover to traverse the grid world to collect Matt Damon and as many pieces of scientific equipment as possible before returning to the starting location. The challenge of this task is the high degree of uncertainty present in the problem as well as the constraint of limited battery usage. Instead of providing the rover with knowledge of where Damon and the equipment are in the world, only a PDF showing the likelihood of finding each object of interest is provided. Additionally, instead of providing the rover with knowledge of the terrain in advance, the rover is provided with a noisy approximation of the terrain.

Our analysis ends with a discussion of the ways to improve our analysis and possible directions for future work.

Contents

1 Abstract 4

2 Introduction 6

3 Related work 7

3.1 Greedy Method . 7

3.2 Genetic algorithm 7

3.3 Ant Colony Optimisation 7

4 Methods and Implementation 8

4.1 Implementation . 8

4.2 Algorithms 8

4.2.1 Genetic Algorithm 8

4.2.2 Greedy Search 10

4.2.3 Ant Colony Optimisation 10

5 Findings and Analysis 11

5.1 Results 11

5.1.1 Effect of Board Size 11

5.1.2 Effect of Estimation Noise 12

5.1.3 Effect of Battery Constraint 12

5.2 Discussion 12

5.2.1 Comparison of Approaches 12

5.2.2 Extensions and Improvements 13

6 Summary 14

6.1 Conclusion 14

6.2 Future Work 14

References 15

Appendix A 16

Appendix B 19

2 Introduction

In the 2015 cinematic masterpiece, *The Martian*, Matt Damon finds himself stranded alone in a research station on Mars. While Damon works tirelessly to keep himself alive on Mars, the rest of humanity combines their greatest thinkers to work out a plan to save him. In this project, we tackled the problem of saving Damon by sending an autonomous rover to collect him and other valuable scientific equipment from Mars and returning to the location where the rover landed.

The problem of collecting Damon and the scientific equipment can be seen as an extension of the traveling salesperson problem embedded in a grid world where much of the information provided in the classical problem is obscured by uncertainty. The Martian environment is represented as a graph where the cost of traversing an edge represents the battery consumed. At the beginning of the simulation, the rover will be provided a noisy approximation of these costs, and as the rover traverses the terrain, it will measure nearby edge costs more accurately. The exact locations of Damon and the scientific equipment to be collected are unknown, but the rover is provided a probability distribution over the nodes of the graph which describes the likelihood of finding a certain object at that location. During the simulation, the rover searches the environment, moving one edge at a time, and replanning as necessary when new information is discovered. The rover’s goal is to collect Damon and as many pieces of scientific equipment as possible then to return to the landing spot, all within the battery constraint.

The case studies we investigate are problem instantiations using different initializations. We investigate the impact of varying the level of uncertainty, board size, number of objects to collect, and strictness of the battery constraint and draw conclusions about the tradeoffs between our considered solutions along all of these dimensions.

3 Related work

In this section we briefly describe the algorithms used in our project and give references to related work on those algorithms.

3.1 Greedy Search:

The greedy approach focuses on finding the locally optimal solution. A good approximation of a heuristic that accounts for all the degrees of uncertainty in the current problem ie. the terrain, the pdfs, the battery etc could get good results and retrieve damon and most of the scientific equipment. SInce the implemented greedy algorithm only focuses on immediate neighbours and the true terrain is available. In a noisy terrain the performance of this approach should not be affected at all.

3.2 Genetic algorithm:

There are many different approaches to the following moments in the genetic algorithm. We have followed a basic approach did not explore any optional methods, as can be found in papers such as [1] and [2]. Below follow some discussion on the chosen methods.

Selection
Some predetermined proportion of the top performing plans survive the natural selection step and get the chance to reproduce. Out of these chosen plans, a proportion of the top performing are selected as “elite” plans. These are guaranteed to be chosen for reproduction. The rest of the offspring are generated by randomly picking two parents from the surviving population. In order to more closely emulate true natural selection one can for example assign each plan a selection probability weighted by its relative fitness [1][2]. By now choosing some top percentage of the plans, we deny all below-par plans to even have a shot at making it to the next generation. This is likely to increase the convergence rate and increase the risk of local maxima, as it decreases the genetic variation within the population.

Breeding
The process of breeding has some pitfalls and is highly dependant on the plan representation. The plan returned to the rover has the format of coordinates, but we cannot stitch together the coordinates of two different plans, unless they have an intersection, since this would mean that the rover takes leaps and not steps. We therefore chose to combine the movements of the two plans, rather than their coordinates. To create a new child, we use one point crossover, as in [2]. Here an optional approach would have been to extract an arbitrary section from the first parent and replace it with the corresponding section of the second parent (two point crossover).

Mutation
As with the previous moments there are multiple ways to choose how to mutate a chosen gene. This also depends on the domain as well as the representation of the gene. In our implementation a mutation is manifested as a number of moves in the plan switching places with its right neighbor, while under the restriction that the result is still a valid plan. Alternatively this could be chosen as swapping two random moves without the restriction of them being neighbors.

3.3 Ant colony optimization algorithm

The ant colony optimisation algorithm (ACOA) is inspired by how a colony of ants goes about finding the best path to a food source in nature. Imagine that a group of ants goes out searching for food in different directions. While walking, ants produce pheromones which can be detected by other ants and works as an indication of a path that an ant took. Pheromones evaporate with time, but their concentration at a specific point increases when the number of ants passing there increases. An ant that found a short path to the food will pass the points on that path with a higher frequency compared to those of ants that take long paths before reaching the food source. This will consequently increase the concentration of pheromones on that path. High concentrations of pheromones attract more ants to follow that path as they trust that high pheromone concentrations indicate paths that “the crowd” takes, and those are considered the best (on average). More ants following a certain path will yet again

increase the pheromone concentration on that path and attract even more ants until eventually all ants take the same path.
A thorough review of the Ant colony optimisation algorithm along with applications can be found in [4]. The initial inspiration for exploring and implementing an ant colony optimisation approach to solving the project's main problem came from [5].

4 Methods and Implementations

4.1 Implementation

The architecture and implementation of this project is designed to be modular so that we encounter as few merge conflicts as possible. Thus, implementing a new algorithm does not require changing any of the base simulation code. Instead, new algorithms belong inside their own class which extends Rover and implements the two abstract methods it declares:

shouldUpdate() is called at the beginning of every step of the simulation and is meant to return whether the updatePlan() method should be called during this step.

updatePlan() is called if shouldUpdate() returns true. It uses the information known to the rover (PDFs, terrain, currentLocation, startingLocation, currentBattery) to produce an ArrayList that represents the path to travel. If we only want to plan one step ahead, this ArrayList should just contain a single point with the world coordinates of the next move. If we want to plan the entire path out, we can return an ArrayList where each node along the path is specified in order from the immediate next step until the final planned step. Even if this function returns a long plan, *shouldUpdate()* will still be called during every step of execution in case the rover decides to change its plan.

The simulation code is broken up into the following files:

Debug.java: This class implements some useful debugging tools.

Environment.java: This class is a Tensor. It contains the 2D double arrays which represent the terrain as well as PDFs for Matt Damon and the objects. This class also provides a getCost(Point p1, Point p2) method which determines the cost of traveling from a point to its provided neighbor.

EnvironmentBuilder.java: This class contains several static methods and helper functions used to manipulate the terrain and PDFs of an environment.

Renderer.java: This class extends JFrame and is the class which renders simulations to the screen.This is useful for debugging and will produce visuals for our presentation.

Rover.java: This abstract class defines the two methods that any planning algorithm must implement: *shouldUpdate()* and *updatePlan()*. In order to implement a new algorithm, we should extend this class and implement those methods.

Results.java: This class is a custom data structure meant to store and format information regarding a complete run of a simulation.

Main.java: This class contains the main method and the *evaluateAlgorithms()* method which was used to drive the gridsearch over simulation parameters.

4.2 Algorithms

4.2.1 Genetic Algorithm

The genetic algorithm (GA) is inspired by the robustness and adaptivity of the evolutionary process seen in nature. The algorithm lets a population of random genoms (in our case plans) iteratively evolve by (randomly) selecting the genomes with the best performance according to some heuristic and letting these reproduce to generate new offspring, which are then mutated with some probability. This process repeats until reaching some predetermined number of generations or until some other convergence criteria is fulfilled.

The GA does not necessarily produce an optimal plan for traversing the grid, but has the advantage of producing multiple feasible plans, as opposed to other search algorithms (such as DFS, BFS and A*), which only search for the optimal plan. This property allows for flexibility and can be advantageous, for example if a human agent is supposed to choose one of the generated plans and the optimal plan cannot be carried out due to some additional information unknown to the algorithm. The random element in the algorithm also allows us to find new plans that might not have been discovered by a more conservative algorithm.

When observing the behaviour of the implemented GA after many generations, we found that the population converges into a single plan, often before reaching the specified number of max generations. This is known as premature convergence and results in suboptimal plans. Increasing the probability of a plan mutating does increase the genetic variability within the population, however we have not pursued the effect that this parameter has on the resulting final plan.

The genetic algorithm has a large number of hyperparameters: the population size, number of generations, the proportion of the population selected for breeding and selected as elite plans, mutation probability and mutation proportions. This allows the algorithm to produce many interesting and varying results, but can also be hard to tune correctly. We have chosen to not spend significant time optimizing the GA's hyperparameters and instead compare our reasonable settings with the other algorithms. The results presented below are produced from using the settings presented in the table below.

Hyperparameter	**Value**	**Hyperparameter**	**Value**
Plan length	15	Proportion of elite plans	0.1
Population size	100	Mutation probability	0.05
Selection proportion	0.25	Mutation proportion	0.25
Number of generations	50	Battery margin	1.5

Another strength of the GA is the ability to adapt different selection, breeding and mutation techniques. There are multiple choices for all of the just mentioned moments, as discussed in section 3.1.

Comments on implementation
The rover that uses the Genetic Algorithm for planning first creates a plan to locate and acquire Matt Damon before searching for any other objects on the map. However, if it does encounter an object as one of it neighbours, it will move to the location of the object and then replan its' path towards Damon. Once Damon is collected the algorithm considers the probabilities of all remaining items when planning. Depending on the plan length, the algorithm is therefore drawn either to the area with the most objects, or to the closest object.

At all times the rover keeps track of the estimated cost of returning home by calculating the mean cost of all past movements as well as the distance to home. If the estimated cost of returning home surpasses the remaining battery by some margin, the rover goes into retreat mode. When retreating the rover will use the genetic algorithm to try to create a path of the same length as the distance to home. Since unobserved terrain updates as we approach it, the estimated return plan might have a higher cost than expected, which might result in a failure. In retreat mode the fitness of the plans is set to *1e7 - estimatedPowerUsed* if the plan reaches home and to *-(100 ** (*dist(homeCoord, lastCoord)) + estimatedPowerUsed)*, where *lastCoord* is the last coordinate in the current plan, if it does not reach home. When retreating, the rover calculates a new plan only if the current plan is not set to succeed. See the Appendix for a pseudo code of our implementation of the Genetic algorithm.

4.2.2 Greedy Search

This greedy algorithm follows the problem-solving heuristic of making the locally optimal choice at each step with the intent of finding the global optimum.

In this approach, the rover uses the information about the immediate neighbours that includes the probability distribution of Damon, the scientific equipment to be collected and the terrain cost(battery consumed) to create an initial plan under the constraint of battery. Once an object is collected, the rover re-plans and starts looking for the next object.

The rover starts from the initial location and starts looking for Damon first. The starting location is used to create a probability distribution along the map which is used on its way back home. At each step, it only continues if there is sufficient battery left. The rover creates a local memory to store all the states it has visited so far when looking for an object to avoid moving in loops. Once this object is collected, this information is wiped. Once damon is collected the next scientific equipment to be collected is chosen at random. An alternative approach - to overlay PDF for all the scientific objects still to be collected was also investigated but it resulted in poor performance due to the shortcomings of the greedy approach (resulted in looping between same states) and converged to local maxima quite frequently.

The heuristic chosen played a critical role in finding a ‘good solution’. The initial approach chose the state that maximised the PDF of the object but this could result in a path that drained the battery if the terrain/cost of making the transition was very high. To account for this issue, the heuristic was changed to PDF(state,object)/cost(state).

See Appendix A for a pseudo code of our implementation of the Greedy search algorithm.

4.2.2 Ant Colony Optimisation

When implementing ACOA, we consider a population of *m* ants that initially traverse nodes according to some a-priori probabilities of going to node *j* from node *i*, η_{ij}. Later on, the probabilities for moving around will depend on both, the pheromone concentrations and the η_{ij} values. The pheromone values are initialized to be equal across the entire environment. After having walked the first set of paths, we calculate the concentrations of pheromones, which are specific for the transition from node *i* to node *j* according to the formula $\tau_{ij} = (1-p)*(\tau_{ij} + \Sigma_k^m(\Delta\tau_{ij})^{(k)})$. In the given formula, $(\Delta\tau_{ij})^{(k)}$ represents the “pheromone gain” that is contributed by each ant (indexed by *k*) and is an indication of how optimal that transition is and p is the evaporation factor.
In the formulation of the travelling salesman problem (TSP), for which ACOA is commonly used, the $(\Delta\tau_{ij})^{(k)}$ values are calculated as the inverse of the distance between the nodes *i* and *j.* This will result in transitions that are shorter producing higher pheromone values and hence increase the chance of making that transition when planning the salesman’s traversal.

Since our implementation is considering a grid of cells that can only be traversed by making a step to a node that shares a side with the current cell, our η_{ij} and τ_{ij} parameters are limited to only being applicable for neighboring cells *j* of a cell *i.* The a-priori probabilities of making a certain transition, η_{ij} are inversely proportional to the cost of making that transition and scaled to sum to one, when considering the transitions from a fixed point to its neighbors. When calculating the pheromone gain that each ant that makes the transition from *i* to *j* contributes to the overall pheromone concentration for that transition, i.e. the $(\Delta\tau_{ij})^{(k)}$, we consider the average of the sum of probabilities to find our current target for that path. To say that more mathematically, if an ant *a* takes the path P of length *l,* traversing cells $c_i \in P$, where the probability of finding a target at each cell c_i is expressed as pdfTarget(c_i), then we will have that $(\Delta\tau_{ij})^{(a)} = (\Sigma_i^l \text{pdfTarget}(c_i))/l$. Therefore the optimality of each transition is dependent on the entire paths that the ants took.

After each set of ants’ walks (what we simply called an *iteration* here), the probabilities of transitioning from cell *i* to cell *j,* p_{ij} are updated according to the formula: $p_{ij} = [\ (\tau_{ij})^{\alpha}\ *\ (\eta_{ij})^{\beta}\ ]/[\Sigma(\tau_{ij})^{\alpha}\ *\ (\eta_{ij})^{\beta}\ ]$. The pheromone values τ_{ij}, and consequently the p_{ij} values across the map are updated for a fixed number of iterations to simulate the effect of ants changing their “usual” paths to those that are more popular, i.e. indicated by higher pheromone concentrations.

In our problem where a rover navigates a partially-known terrain, looking for Matt Damon and objects to collect, it will simulate a large number of plans that an ant colony would produce, and then take a number of steps based on the same probability pairs p_{ij} that the ants use while searching before sending out a new ant colony to make another set of plans.
See the Appendix for a pseudo code of our implementation of the Ant colony optimization algorithm.

5 Findings and Analysis

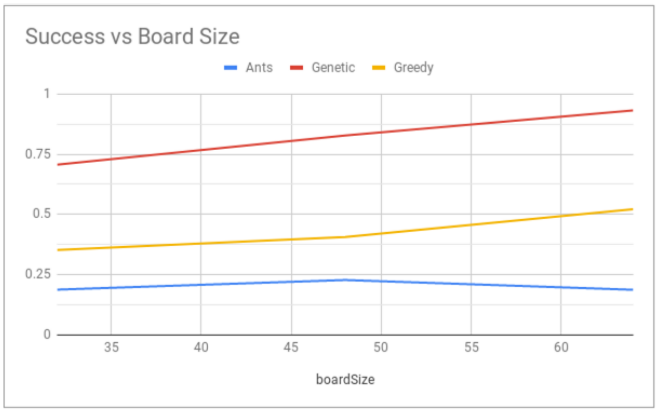
5.1 Results

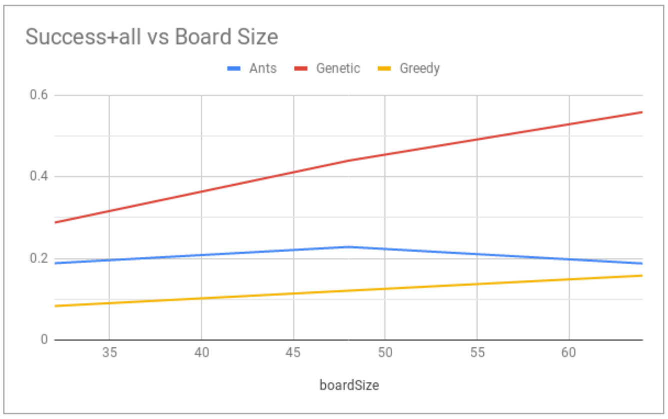
In order to compare the three algorithms and investigate the effect of various simulation hyperparameters, we elected to run a grid search over a region of the simulation hyperparameter space which we suspected would yield the most insightful results. Every combination of hyperparameter settings was run multiple times with different random seeds so that different pdfs for Damon and the objects were also factored into the analysis. The hyperparameter values we searched over are presented in the table below:

Hyperparameter	Range of values	Hyperparameter	Range of values
board size (side length of square grid world)	{32, 48, 64}	estimation noise (maximum deviation from true terrain)	{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6}
number of objects	{3, 5}	battery constraint (percentage of battery provided to rover)	{0.125, 0.25, 0.5, 1.0}

After running approximately 9000 simulations for 48.86 computer hours, we compiled a large dataset with the hopes of investigating the effects of some of the simulation hyperparameters on the algorithms. Below, we plot the effect of each simulation hyperparameter on the success rate of the rover, where success is defined as retrieving Damon and returning to the starting point. We also consider the rate of retrieving all objects as well as Damon.

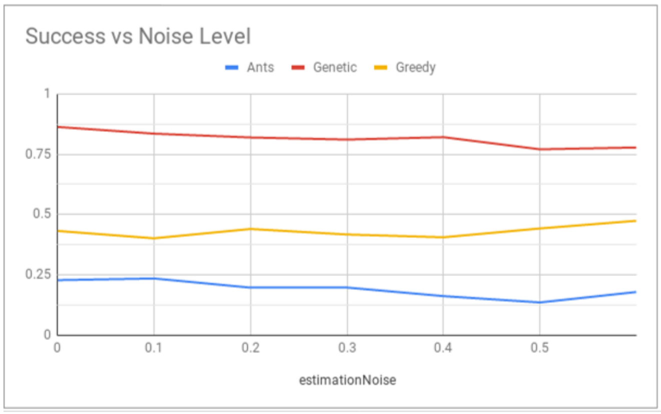
5.1.1 Effect of Board Size

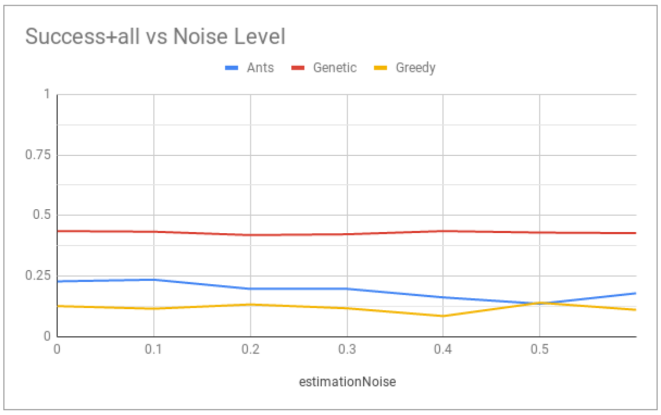




In the above plots we can see that the Genetic and Greedy algorithms perform better on larger board sizes, but that ACO seems to perform nearly equivalently on all board sizes. One potential explanation for the greater success rates for larger boards could be that since the range of possible variances for the gaussian distributions used to approximate the position of Damon and the objects did not scale with board size, so the ratio of variance to board size became smaller and the larger boards contained less uncertainty about the position of Damon and the objects. Another plausible explanation is that the amount of battery provided to the rover for larger boards grew too quickly, making the task easier.

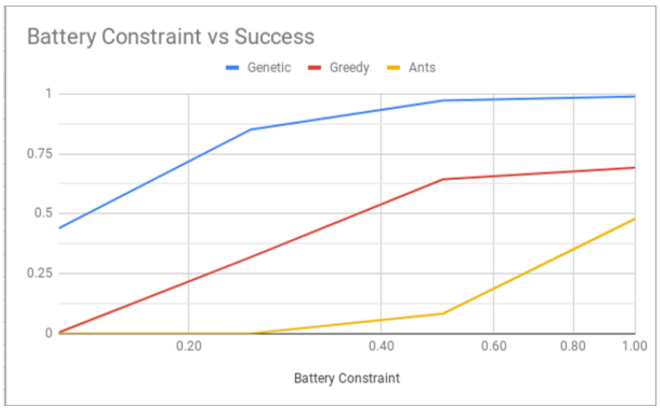
5.1.2 Effect of Estimation Noise

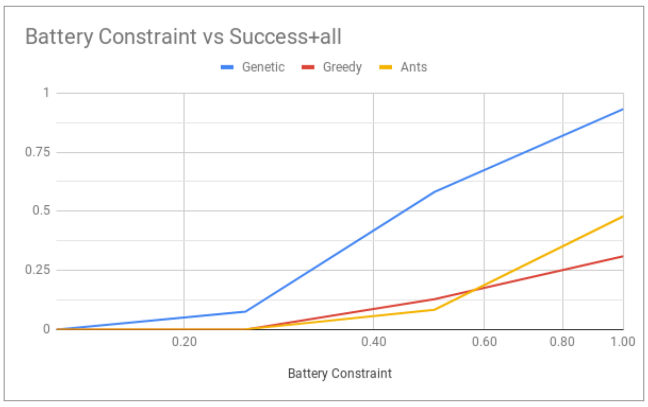




In the figures above we consider the effect of estimation noise in the terrain. Contrary to our original expectations, increasing estimation noise does not seem to significantly impact any of the algorithms. Greedy, in particular, is not affected at all by high levels of noise, while both GA and ACO suffer only slightly.

5.1.3 Effect of Battery Constraint





In the above figures, we can see that providing the rover with less battery led to lower success rates across all algorithms, however this drop was not the same across algorithms. The Genetic Algorithm's battery requirement for successful runs appears to be notably lower than Greedy or ACO, achieving success just shy of half the time constrained to 12.5% battery when Greedy and ACO both fail to achieve a single success with that constraint.

5.2 Discussion

5.2.1 Comparison of Approaches

Between the three algorithm implementations we tested, our GA implementation is by far the most successful. Not only does it have the greatest success rate over the 9000+ simulations we ran (82.9% overall), it also seems to be surprisingly resilient, performing well under tight battery constraints and under high levels of uncertainty. The only case where another algorithm may be preferred would be if execution time is an additional constraint, in which case Greedy was by far the fastest algorithm tested. Greedy had an average runtime under 1ms while ACO and GA both had average runtimes over 10 seconds. Our implementation of ACO did not seem to be an advantageous choice in any dimension, taking longer and performing worse than the other two algorithms. See appendix B for a plot comparing execution times for each of the algorithms in more detail. In nearly all situations, it seems reasonable to prefer our GA implementation over the others.

5.2.2 Extensions and Improvement

It is important to note here that our analysis only pertains to our implementations of these algorithms. While we attempted to create the most powerful implementations of these algorithms we could, we have identified many ways to improve our implementations. It is impossible to say now whether this comparison would hold true after accounting for these improvements.

Suggested improvements for Genetic Algorithm

One could introduce a convergence criteria that checks if all genes in the population are equal, and if so we stop the iterations. This would decrease the execution time but can lead to premature convergence to local maxima, which we want to avoid. We therefore choose not to implement this criteria. Another criteria that we could implement is to regard the ratio of increase in fitness between generations, and to stop when this falls below some threshold.

In the implementation we allow the same gene to reproduce multiple times, and do not keep track of which genes have previously created a child. If the population is large and genetic variation is high within the subpopulation selected for breeding, this ought to not be an issue. If we also consider that the probability of two genes creating two equal children is 1/planLength (due to randomly selecting incision point, and not considering possible symmetries), then the probability of having multiple equal children decreases quickly in a large and varied population. But for small populations this can cause the algorithm to converge to a suboptimal solution.

Another improvement would be to allow the length of the genes to change through mutation and breeding. This could also be used in determining a fitting value for the hyperparameter *planLength* for a particular environment. Of course this implementation would require an updated breeding function that can create a new child from two parents of differing lengths.

Change so that it does not automatically jump at damon or objects, but rather replans. The algorithm SHOULD then choose to pick up these items based on the PDF. Makes it more “naturally genetic” and less hard coded.

Suggested improvements for Greedy Algorithm

The greedy algorithm implemented for the project looks at the PDF of the immediate neighbours. A possible improvement could be to look further than just one neighbour. The heuristics play an important role for giving a score to the next state. The initial algorithm looks for damon first and doesn’t take into account the PDF values for the other scientific objects. An alternative approach could be to create a path that collects objects on the way to find damon.

Suggested improvements to the Ant colony optimisation algorithm:

The first point of improvement with the ACOA in our implementation has to do with testing its performance depending on the varying hyperparameters. Due to our limited time, we decided to try a more “powerful” approach with a large number of ants and iterations. However, that might not be necessary and could lead to drastic runtime improvements. Secondly, we could test different heuristics that value different transitions differently. In the current version, the heuristic is not only dependent on the number of ants that make a transition from one particular cell to another, but it also depends on the entire path that each of the ants took. The problem with this approach is that in that case we are rewarding all the transitions that a certain ant took in its path equally, which means that if the ant first went into a region of high pdf gain and then into a very low pdf gain, its later choices will be rewarded along with the earlier ones and so that might make the rover not search a desired area thoroughly enough. A possible alternative heuristic that could fix this issue is to use the cubes of the consecutive differences in the pdf values of finding Damon or the objects for the entire length of the path.

6 Summary

6.1 Conclusion

Based on our analysis we determine uncertainty of terrain does not significantly impact the performance of any of the tested algorithms, larger boards tend to be easier, and our Genetic Algorithm implementation is much more capable of adjusting to strict battery constraints. Finally, we conclude that our Genetic Algorithm implementation is the preferred approach in every case except when runtime is a critical factor, in which case our Greedy implementation is preferred.

6.2 Future Work

Range of Visibility

An interesting factor to explore would be the range of visibility of the true terrain and PDF values for the rover. The expected behaviour would be an improvement in the performance for the GA and ACO as this range increases, but the more interesting analysis would be to compare the performance gain across algorithms.

Bounding the Terrain Uncertainty

In a noisy terrain, a good way to deal with the uncertainty would be to use an estimate for the noise which is adapted from experience as rover explores unknown terrain. Some estimates that could be used to achieve this have been mentioned below:

1. Highest noise experienced so far : This is a strict bound that expects the highest noise for every unvisited state. This approach would only affect GA and ACO.
2. Mean of noise experienced so far : The average of the noise experience so far is added to the unvisited states to deal with uncertainty.
3. Fitting a parametric distribution to the noise seen, and unvisited states are augmented by the (1-alpha) percentile of the distribution, to parameterize the rover's degree of caution.

Simulated Annealing Optimisation

To solve the issues/shortcomings of the greedy approach, simulated annealing was also implemented. The initial algorithm created a candidate plan that maximized the same heuristics as the greedy approach. With high initial temperature , the probability of choosing a random/less optimal plan was higher resulting in more exploration of the map. With each step the temperature was reduced with a cooling rate to reduce the exploration. The expected results of the algorithm was to make some bad initial moves but converge to a greedy solution with increasing steps. The algorithm did not perform as expected within the constraints of time for the project. This algorithm could be improved further and provide interesting and better results.

References

[1] INAGAKI, Jun & Shirakawa, Tomoaki & Shimono, Tetsuo & Haseyama, Miki. (2009). A Genetic Algorithm for Path Generation and its Applications.

[2] David E. Goldberg. 1989. Genetic Algorithms in Search, Optimization and Machine Learning (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Chen, Mao. (2008). A Greedy Algorithm with Forward-Looking Strategy. 10.5772/6351.

[4] Denis Darquennes (2005), Implementation and Applications of Ant Colony Algorithms

[5] Dorigo M., Birattari M. (2011) Ant Colony Optimization. In: Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA

[6] A comparative study between a simulated annealing and a genetic algorithm for solving a university timetabling problem BY JONAS DAHL RASMUS FREDRIKSON

[7] Holger H. Hoos,Thomas Stutzle,(2002) Stochastic Search Algorithms

Appendix A: Pseudocode

Genetic Algorithm

```
function generatePlan(populationSize, numberOfGenerations)

    population = create population of size populationSize of random valid plans

    while (counter < numberOfGenerations) then:

      evaluate population with fitness heuristic
      fitnessHeuristic(plan) =
            if (Rover.batteryLow == true) then:
              if (plan.lastCoordinate == homeCoord) then:
                plan.fitness = 1e7 - estimatedPowerUsed
              else
                plan.fitness = -(100 * (dist(homeCoord, last coordinate in plan)) + estimatedPowerUsed)
              end if
            else if (foundMattDamon == false) then:
              plan.fitness = sum accumulated probability of finding Matt Damon along planned path
            else
              plan.fitness = sum accumulated probability of finding any object along planned path
            end if

  -   natural selection step
       sort plans after fitness
       selectedPopulation = select top selectionProp performers from population
       mark top eliteProp performers from population as elitePlan

  -   breeding step
       for each elitePlan in selectedPopulation:
            plan = random plan from selectedPopulation
            child = breedPlan(elitePlan, plan)
            add child to newPopulation
       end for

       while (size(newPopulation) < populationSize):
            planA = random plan from selectedPopulation
            planB = random plan from selectedPopulation \ planA
            child = breedPlan(planA, planB)
            if (planA != planB) then:
                  add child to newPopulation
            end if
       end while
      for each plan in newPopulation:
            with some probability p mutate(plan)
            add plan to mutatedPopulation (even if mutated or not)
      end for

      evaluate mutatedPopulation with fitnessHeuristic
      counter = counter + 1
    end while

    return plan in mutatedPopulation with highest fitness
```

Function for breeding two plans

```
function breed(planA, planB):
 newPlan.isValid = false
    while (newPlan.isValid == false) then:
        cutSize = random.integer(0,planA.length)
        newPlan = take moves up until cutSize from planA and rest
             of moves from cutSize to end of planB
        newPlan.isValid = newPlan coordinates not outside grid?
    end while
 return child
```

Function for mutating plan

```
function mutate(plannedMoves,mutateSize):
 mutatedIndices = 0
 while (mutatedIndices < mutateSize) do:
    swap two moves in plannedMoves
    if (plannedMoves still valid) do:
        mutatedIndices = mutatedIndices + 1
    end if
 end while
```

Greedy Algorithm

Function for nextMove (rover,environment)

```
        n ← neighbours(currentLocation)
    roverMemory {} ← CurrentState,Terrain,PDF
    if(lowBattery) then
        nextState←max{heuristics(Home,n)}
    else if(!damonCollected) then
            nextState←max{heuristics(Damon,n)}
    else if(!allObjectsCollected) then
            o← pick(object)
            nextState←max{heuristics(Object.n)}
   else
        nextState←max{heuristicsHome(n)}
            endif
    return nextState
```

Function for *heuristics (pdfEstimates, neighbours)*

```
  for state=0 to neighbours
        candidate ← max{(pdfEstimates(state))/cost(currentState,state)}
   return candidate
```

Ant colony optimisation algorithm

local variables used:
list of all paths for each ant: antsPath
global pheromones values
local plan suggestion: antPlan

```
function updatePlan()
    if (reached home)
        return null
    end if
    if (collected Damon & not registered it locally)
        resetPheromones()
    end if
    for (every object o in environment)
        if (collected o & not registered collecting o locally)
            resetPheromones()
    end for
    for (integer i < # of iterations)
        clear antsPath variable
        for (ant a in list of ants)
            batteryLeft = Rover.currentBattery()
            while (batteryLeft > 0.5*Rover.currentBattery() )
                # calculate probabilities $p_{ij}$ of going to neighbor point $j$ from point $i$:
                getNextPositionProb(current ant location)
                pick next point from pdf given by the $p_{ij}$ values
                if (next point is in antsPath for ant “a”)
                    pick next point from remaining neighbors according to their $p_{ij}$ values
                end if
                if (already visited all neighbors)
                    break
                end if
                batteryLeft = batteryLeft  - getCost(current ant location, next point)
            end while
            add current ant path to antsPath
        end for
        updatePheromones()
    end for
    update antPlan variable
return antPlan
```

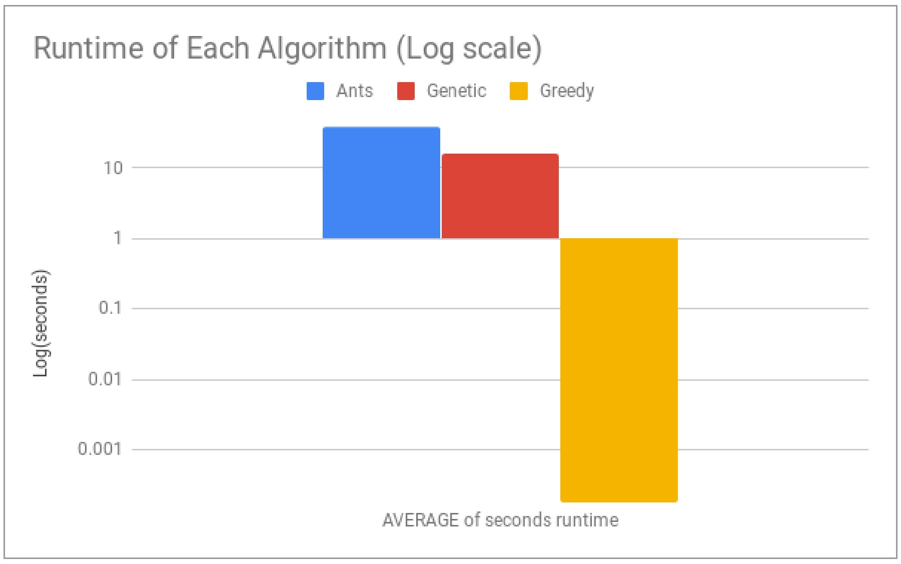
```
function getNextPositionProb(point p)
    for (neighbor n of point p)
        calculate probability of going from $p$ to $n$, $\eta_{pn}$, as $\eta_{pn}$ = 1/(getCost($p$,$n$))
    end
    scale all $\eta_{pn}$ to sum up to 1
    for (neighbor n of point p)
        calculate the probability of going to $n$ from $p$, $p_{pn}$, depending on the pheromones and $\eta_{pn}$:
        $p_{pn} = (\tau_{pn})^{\alpha} * (\eta_{pn})^{\beta}$
    end for
    scale all $p_{pn}$ to sum up to 1
return probabilities $p_{pn}$
```

function updatePheromones()

```
    for (each pair of neighbors i and j)
        # update pheromones from i to j, $\tau_{ij}$ according to the evaporation rate p:
            $\tau_{ij} = (1-p)*\tau_{ij}$
    end for
    for (ant a in list of ants)
        if (not everything is collected)
            for (each point p that a visited)
                if (not found Damon)
                    # update pheromone gain by adding the likelihood of finding Damon at p:
                      pheromone gain += (P(damonAt(p)))/(length of ant a's path)
                else
                      for (each object o that isn't collected)
                          pheromone gain += (P(object_o_At(p)))/(length of ant a's path)
                end if
            end for
        else
            # update pheromones on ant's path according to how close the path takes the ant to origin:
              pheromone gain = 1/(taxicab distance from origin)
        end if
    # update pheromones on the path that ant a took by adding the average pheromone gains to each step:
        $\tau_{ij}$ += pheromone gain
end function
```

Appendix B: Additional Graphs and Data

Runtime Of Each Algorithm (log(seconds))


Runtime of Each Algorithm (Log scale)
Ants
Genetic
Greedy
Log(seconds)
AVERAGE of seconds runtime