CS 472 — Project 3

GAs & DEs — A Land of Lisp Approach

Steven White
swhite24@mix.wvu.edu
8258

1. ABSTRACT

1.1 Introduction

Introduces problem description and intent of study.

1.2 Models

Describes Conrad Barski's *Land of Lisp* model in detail. Outlines modifications made to model to suit purposes of this study.

1.3 Algorithms

Compares and constrasts differential evolution with genetic evolution with pseudo-code.

1.4 Expectation

Lays out four hypothesis indicating what is expected of study.

1.5 Results

Displays results in tables and figures and also elaborates on the data.

1.6 Validity

Comments on the degree which the results are valid.

1.7 Conclusion

Evaluates each hypothesis based on results.

1.8 Further Work

Lays out further steps to take to improve study.

2. INTRODUCTION

The purpose of this project is to objectively compare two subsets of evolutionary algorithms, differential evolution (DE) and genetic algorithms (GA), using the model of evolving rats found in Conrad Barski's *Land of Lisp.* The two sets of algorithms will be tested for success in several criteria — run-time, diversity of population, scoring, and time taken to achieve a stable population.

The algorithms will be tested with both a single-objective and multiple-objective version of the evolving rats problem. The simplest and most obvious objectives to use in the context of this problem are the amount of energy a candidate has in comparison to the maximum potential energy and the number of children a candidate has produced relative to its age. Due to the nature of these objectives and the model itself, a slight diversion was made from the traditional versions of DEs and GAs which immediately compare the scores of candidates with parents. Instead, we will allow the candidates free roam for a period of generations, in order to be able to accurately assess their success. Once a candidate is of correct of age, it will be compared with its parent. Each objective will be scored independently and will be unweighted. If a candidate dominates the parent, the parent will be removed from future generations. Conversely, if a candidate is dominated by the parent, the candidate will be removed. If neither the parent nor the candidate dominate the other, both will remain for future generations.

In addition to comparing one algorithm set with the other, each set will be compared to itself by changing their corresponding choice options, such as scaling factor when mutating and the number of objectives to be scored. Changing these options will affect each of the previously listed comparative criteria and may also arrive at a better solution.

3. MODEL

3.1 World Description

The model to be used for both differential evolution and genetic algorithms will be the rat evolution model of *Land of Lisp.* This model evolves a randomly generated initial population of a single rat. The landscape of the "world" is thirty units tall by one hundred units wide. The rats are herbivores, and the landscape is populated with two additional plants for each generation. Rats have a set amount of energy, and must find plants to regain lost energy in order to remain alive. Should a rat's energy reach zero, the rat will die and be removed from further generations. A central "oasis" is present in the landscape. Of the two plants placed in the landscape for each generation, one will be placed randomly inside the oasis, and the other placed randomly somewhere outside the oasis. Figure 1 depicts a visual representation of the landscape with a central oasis. [1]

Each generation of this model involves a specific life-cycle of the rats. The first thing rats do is turn. The direction of the turn is determined by the rat's genes. Each rat has a set of genes corresponding to how likely they are to turn

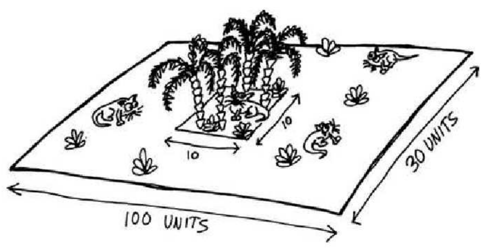

Figure 1: Visual representation of rat "world".

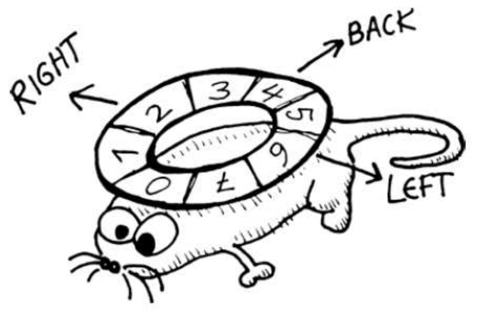

Figure 2: Gene "wheel" depicting directions of rat travel.

a certain direction. A visual depiction of this these genes and their corresponding directions is shown in Figure 2. [1] For example, a rat with higher values in the lower-indexed positions of the gene will make less dramatic turns.

Once turned, the next step in a rat's life-cycle is to move in the specified direction. Due to the centrally located oasis, two distinct types of genes are expected to evolve. The first will have higher valued genes with lower indexes, causing them to run at a more constant direction and often returning to the oasis. The second will have higher valued genes with higher indexes, causing them to wander around a fairly consistent area, i.e. the oasis.

The final step in the life-cycle of a rat is to produce offspring. Offspring is produced froma single parent, similar to DE. This implies that GAs will have no cross-over, only mutation. In order for a rat to be eligible to reproduce, a minimum energy requirement must be met. Should that requirement be met, the total energy of the parent is split equally between itself and the offspring. The genes of the offpring are uniquely generated by each algorithm, though a basic premise remains that offpsring are similar, but typically not identical, to their parent.

3.2 Modifications for our Purposes

Unlike the original version, the model used for all experiments described later had an initial population of four rats. The primary reason for this difference is that differential evolution requires the parent and three other randomly generated rats for the parent to produce offspring. An initial population of just a single rat would prevent any diversity in the population of rats, merely a horde of clones. The same setup was used for genetic algorithms simply for consistency.

Staying true to traditional evolutionary algorithms, offspring will compete with parents for a position in the population for future generations. Once a child is of sufficient age, it will be scored and compared to its parent, with the individual with a better score living on. This presents two distinct possibilities of rats being terminated, natural dying by running out of energy as well as being "killed off" by either a parent or child. This is different from the original model in which rats only died if their energy reached zero.

3.3 Decision Space

The decision space of this model consists entirely of the genes of each rat. No other controllable factor has so large an effect on the success of an individual. The genes are what determine direction of travel, and ultimately what areas of the landscape are covered by each rat. This facilitates the natural killing off of animals with bad genes, since they will travel in patterns not ideal for this model, as well as allowing the rats with better genes to flourish, since they will frequently encounter plants in the landscape to devour.

The method in which genes are tweaked is unique to each algorithm. Differential evolution uses a single parent and three additional randomly generated members of the population to create a candidate using cross-over and mutation. Our genetic algorithm uses only mutation and not crossover, despite traditional GAs performing crossover, due to crossover requiring two parents and this model reproduces asexually.

3.4 Objective Space

Two objectives were decided upon for these experiments. The first is an energy-score. This may initially seem like a peculiar choice since parents and their offspring start with the same amount of energy. However, since children are given a specific period of time to live before being compared to their parent, often the child or the parent will be able to further consume plants or reproduce, resulting in different amounts of energy. The energy-score for each rat is to maximized, up to a value of 1. The calculation of the energy objective is as follows.

$$energyScore = currentEnergy/maxEnergy$$

The second objective for these experiments involves the number of times a rat is able to produce offspring. This objective may seem to favor the parents, due to them being alive longer, however the score is not determined solely on the number of offspring. In order to put offspring on a level ground with parents, the number of offspring is divided by the age in terms of generations alive. This result yields a maximum value of one, the best score, and is calculated as follows.

$$reproductionScore = numberOfChildren/ageOfRat$$

Since each of these two objectives are to be maximized, an ideal rat will always have the maximum amount of energy and reproduce in every generation. This, however, is impossible due to the nature of this model. While maximizing each objective would contribute to the overall success of an individual rat, the objectives compete with one another. Hoarding energy means the rat is not reproducing regularly and, conversely, reproducing at a high rate will result in a lower amount of energy.

4. ALGORITHMS

4.1 Genetic Algorithm and Differential Evolution Similarities

When comparing GAs with DEs there are multiple apparent similarities. Firstly, each algorithm requires a randomly generated initial population. The size of this initial population sets a limit on the maximum population size for all subsequent generations. Each member of the population will produce a candidate, which is where the primary difference between the two lies. Ordinarily, this candidate is then compared with its "parent" based on some pre-defined objective criteria, and the winner is then reinserted back into the population with the loser begin discarded. However, as described previously, for our purposes the child rat will be given a set period of time to live before being compared with its parent. A modified version of the basic outline for DE can adequately describe the similarities of each algorithm and is shown below.[4]

Algorithm 4.1: DEandGA()

```
$\mathcal{P} \leftarrow$ randomly generated initial population
while stopping criterion not met
  do for each $member \in \mathcal{P}$
       do if member is old enough
            then compare to parent and keep better
          generate candidate from member, add to $\mathcal{P}$
```

Another commonly used algorithm for each algorithm is the method which shows a summary of the living population of rats. This is used to look for distinct populations of rats in the population. Algorithm 4.2 describes the procedure and is shown below.

Algorithm 4.2: PopulationSummary()

```
$x \leftarrow$ array of 8
for $i \leftarrow 1$ to $length(all\_rats)$
  do $x \leftarrow x + all\_rats[i]$
for $i \leftarrow 1$ to 8
  do $x[i] \leftarrow x[i]/length(all\_rats)$
return $(x)$
```

The generation of candidate from each member of the population is what distinguishes the two algorithms from each other. The following two sections describe how each algorithm produces a candidate.

4.2 Genetic Algorithms

All genetic algorithms use two operators to generate new candidates for the population, crossover and mutation.[3] Crossover involves combining two "parents" to produce an offspring. There are several different varieties of crossover, such as single-point crossover and multi-point crossover. Mutation randomly changes the offspring produced from crossover, typically by a small amount. The main purpose of mutation is to introduce a larger amount of diversity in the population. The GA procedure used to generate offspring in our experiments is shown below.[3]

Algorithm 4.3: GA_Candidate(Parent p)

```
$n \leftarrow$ random integer from 1 to 8
$candidate \leftarrow p$
$candidate \leftarrow candidate$ (with gene $n$ modified by f)
return $(candidate)$
```

Once an offspring rat is created, it is allowed to live freely for a set period of time before being scored and compared with its parent. Once this free period expires, the offspring is compared with its parent and if it scores better, it remains in the population and replaces the parent. If it does not score better, it is discarded.

4.3 Differential Evolution

Differential evolution uses crossover, similar to genetic algorithms. However, DEs do not use mutation, and instead generate an offspring from three random individuals from the population other than the parent. The randomly generated offspring then undergoes crossover with parent using a predefined crossover probability. The outline of generating a candidate with DE is shown below.[4]

Algorithm 4.4: DE_Candidate(Parent p)

```
procedure Candidate1$(x, y, z)$
  $candidate \leftarrow x + f * (y - z)$
  return $(candidate)$

main
  $x, y, z \leftarrow$ random members of $\mathcal{P}$
  $candidate \leftarrow$ Candidate1$(x, y, z)$
  Modify $candidate$ by binary crossover with $p$
  return $(candidate)$
```

Once a candidate offspring is produced it is given a period of free life, similarly to how GAs are treated, instead of being immediately compared with the parent. If the candidate is better than the parent, the parent is replaced, otherwise the candidate is discarded.

5. EXPECTATIONS

Hypothesis 1. *Due to the location of newly created plants in the rat landscape, the rats who frequently visit the center "oasis" will have a higher chance of survival. This will lead to two groups of rats: those who roam around the oasis, and those who run loops around the landscape.*

Hypothesis 1 predicts that two distinct groups of rats should arise after a sufficient number of generations. This prediction was originally presented by Conrad Barski in the description of the rat model.[1] His reasoning for the model forming two groups of rats is that the most successful rats will frequently visit the central oasis of landscape, which gets significantly more plants to consume than any other portion of the landscape. The first anticipated group will have higher values in the latter portion of their genes, resulting in them frequently doing an about face and remaining in the landscape for the majority of their lives. The second group will have higher values in the earlier portion of their genes, resulting in very little turning doing complete revolutions around the landscape and traveling through the oasis on each revolution.

Hypothesis 2. *Genetic Algorithms will execute faster than Differential Evolution for all combinations of the number of generations and the number of objectives.*

Hypothesis 2 predicts that genetic algorithms will execute faster than differential evolution no matter the number of objectives use to score the rats or the number of generations conducted. This prediction is based on the fact that differential evolution requires a total of four members of the population in order to produce a new candidate[2], whereas genetic algorithms require only a single parent. The additional members of the population require more processing time to obtain.

Hypothesis 3. *Differential Evolution will approach the two population hypothesis in fewer generations than Genetic Algorithms.*

Hypothesis 3 predicts that differential evolution will arrive at the two dominant populations of rats described in Hypothesis 1 in fewer generations than genetic algorithms. The method in which differential evolution produces new candidates, by using three members of the population to produce an initial candidate and then a binary crossover with a parent, allows for more diversity in the population. The specific genetic algorithm used in this experiment uses only a single parent's genes with mutation, resulting in less diversity.

Hypothesis 4. *Both algorithms will maintain a similar number of living rats after a sufficient amount of generations due to the natural killing off rats in the model.*

Hypothesis 4 predicts that each algorithm will result in a fairly constant number of living rats after a sufficient number of generations. The natural starving off of rats in this model, occuring when a rat reaches an energy level of 0 by not consuming enough plants, puts an implicit population cap on the rats. The cap is also affected by the size of the landscape, a mere 100 units by 30 units, as well as the number of new plants introduced to the landscape, two new plants for each generation.[1] The small size and number of plants is not compatible with a large population, and as such the number of living rats will be limited.

6. RESULTS

Performance, effectiveness, better on one or multiple objectives. Graphs, etc.

6.1 Test Problem

The problem tested in our experiments is that of evolving rats from Conrad Barski's *Land of Lisp* described in detail in section 3 on page 1. The constraints on world size, oasis size, and number of plants introduced to the landscape per generation are followed exactly. Each algorithm, when compared with the other, also uses the same value of scaling factor and mutation rate.

6.2 Performance Measures

The performance measures used in our experiments are quite simple in nature. The first is the *Average gene metric*. This metric calculates the average gene of all living rats in the landscape, then determines the percentage of total genes found in each specific gene. These percentages allow an observer to look for patterns in the types of rats still living. For example, an *Average gene metric* with larger percentages in lower-indexed portion of the gene would indicate a rat population with the majority favoring a behavior involving little to no change of direction, resulting in frequent revolutions of the landscape.[1]

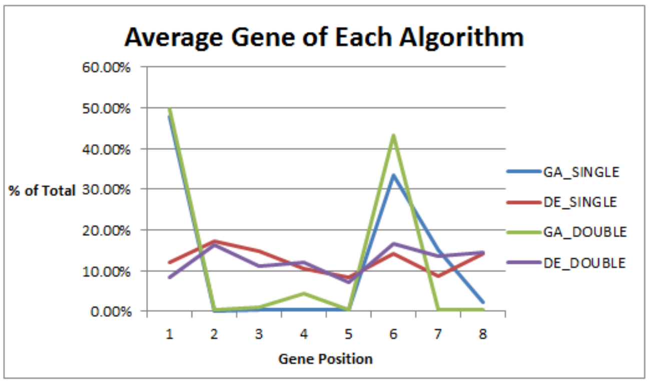

Figure 3: Average gene for each algorithm, indicating two populations.

The second performance metric used is the *Number of living rats*. This metric determines if the algorithm has any affect on the number of living rats outside of the already existing constraints put in place by the problem itself, such as the limited size of the landscape and limited amount of plants.

The third performance metric used is the *Time of execution*. This metric simply measures the run-time of each algorithm for different numbers of generations.

6.3 Results of Experiments

Table 1 on page 5 illustrates the number of living rats for each algorithm for different numbers of conducted generations. It adequately demonstrates how each the initial population of rats, 10,000, quickly decreases to the natural population limit put in place by the model. It shows that neither algorithm, using single or multiple objectives, puts any additional constraint on the population.

Figure 3 on page 4 shows the average gene, in percentages of the total gene, for each algorithm using both single and multiple objectives. This metric shows all four variants having two "humps" in the genes of the population, with genetic algorithms and differential evolution having stark differences in degrees. The two "humps" associated with genetic algorithms contain a much larger portion of the genes in the humps, indicating two very different types of rats in the population, whereas the two "humps" associated with differential evolution make up only a mild increase of the total genes, indicating a more uniform population.

Figure 4 on page 6 shows four graphs depicting the runtimes of single-objective GA, multi-objective GA, single-objective DE, and multi-objective DE. The first two graphs show GA against DE on both a single objective and multiple objectives, and they depict a clear difference in the runtimes, with genetic algorithms performing significantly better than differential evolution for both versions. This is to be expected as differential evolution has more processing involved with the creation of new candidates. The second two graphs compare the runtime difference of single objective and multiple objectives for each algorithm. These graphs

Table 1: Average number of living rats by generation

Number of Generations	Number of Living Rats-DE	Number of Living Rats-GA
0	10,000	10000
100	40,013	40,012
200	210	215
300	95	85
400	70	71
500	84	79
600	79	78
700	85	81
800	91	95
900	86	91
1,000	83	84

show a consistent increase in the runtime of multiple objectives, though perhaps not to the degree one would expect.

The final set of results obtained in this study is the number of generations required for each algorithm to portray a consistent grouping pattern in the average genes of the rat population. These results show that genetic algorithms took on average 20,000 generations whereas differential evolution took on average 15,000 generations to produce a pattern.

7. VALIDITY

Comment on limitations of study.

7.1 Conclusion Validity

The degree to which conclusions made about relationships between variables in this experiment are justified is mostly sound. Concluding that genetic algorithms execute faster than differential evolution can be done simply by observing the statistic directly involved, time of execution. The same can be said for concluding that neither algorithm will have enough of an affect on the population size to overcome the already in place constraints by the problem itself. This can be determined simply by observing the number of living rats.

The third conclusion made on the two groups of rats forming in the population, however, is not as direct. The basis for this conclusion is the *Average gene metric* described in section 6.2 on page 4 and calculated via algorithm 4.2 on page 3. This procedure returns a list of the percentages each individual gene makes up of the total. These percentages are then used to infer the movement tendencies of the living rats in the populations and group them into "sub-species". The use of this metric comes from Conrad Barski's *Land of Lisp* where he describes how the genes of the rats indicate distinct patterns.[1]

7.2 Internal Validity

The internal validity of this study is quite good. The different experiments use the same variables for each algorithm and then make conclusions based on the results. Each algorithm is also compared to itself while changing only one parameter and keeping all others constant, allowing for a higher degree of internal validity.

7.3 Intentional Validity

The chosen model, described in section 3 on page 1, is sufficient in assessing the effectiveness of both differential evolution and genetic algorithms, due to it having multiple potential objectives to be optimized. The only drawback of this model is the natural decreasing of population size.

Since the purpose of this study is to compare genetic algorithms with differential evolution in terms of runtimes and effectiveness arriving at an optimal solution, the metrics described in section 6.2 on page 4 adequately assess each of those goals.

7.4 External Validity

The degree of external validity of this study is quite high since the model used behaves the same for each algorithm. Performing this same study on a different model would yield similar results.

7.5 Ecological Validity

The nature of this study, comparing and constrasting genetic algorithms with differential evolution, allows for a high degree of ecological validity since there are no ethical concerns involved with the assessment of algorithms. The rat evolution model used in this study, however, may present some concerns if offspring are expected to compete with parents for a spot in future generations.

8. CONCLUSION

Based on the results described in section 6 on page 4, Hypothesis 1 can be confirmed. This hypothesis predicted that two groups of rats would arise as the dominant groups: those who roam around the central oasis, and those who run circles around the landscape and frequently visit the oasis. Figure 3 shows that two "humps" are present in the average genes of the rat population for both single and multiple objective versions of each algorithm, indicating the presence of two majority groups. The hump with in earlier portion of genes represents the rats who rarely turn from their present direction of travel, and consequently run circles around the landscape. The hump in the latter portion of genes represents the rats who frequently turn back on their current direction of travel, resulting in them staying in a very specific section of the landscape, the oasis.

Hypothesis 2, predicting that GAs will execute faster than DEs, can also be confirmed based on the results displayed in Figure 4. This figure shows a graphical representation of the runtimes for single and multiple objective versions of both genetic algorithms and differential evolution, with GAs performing better in both after a sufficient number of generations. GAs and DEs appear to perform similarly for smaller numbers of conducted generations, but as the number of generations grows the difference in performance begins

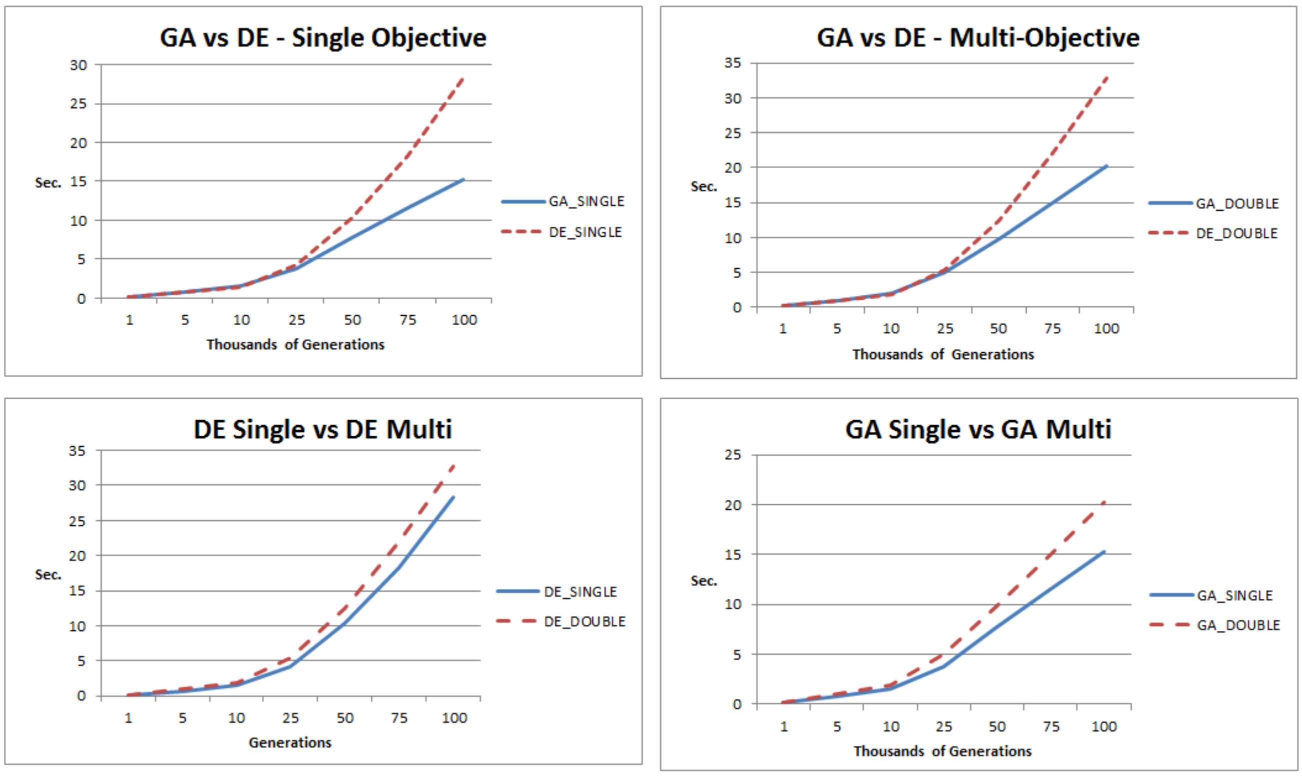

Figure 4: Comparison of runtimes for each algorithm and objective type.

to swing more and more into the favor of GAs. A second inference can also be made based on this figure — that the difference between GAs and DEs is greater than that of single objective and multiple objective.

Hypothesis 3 can be partially confirmed based on the results. This hypothesis predicted that differential evolution will arrive at two distinct populations in fewer generations than genetic algorithms. The two population grouping of the rats is in fact observable at an earlier generation in differential evolution, around generation 15,000, than that of genetic algorithms, around generation 20,000. However, the difference is not nearly as great. As shown in Figure 3 on page 4, the degree to which the genes representing the two populations stand out from the remainder of the genes is much greater in GAs than in DEs. This implies that GAs result in populations that are much more diverse.

Finally, Hypothesis 4, predicting that both genetic algorithms and differential evolution will have similar numbers of living rats after a sufficient number of generations, can be confirmed as well. Table 1 on page 5 shows the number of living rats present when conducting either genetic algorithms or differential evolution for generations ranging from 0 to 1,000. The initial population of rats is constant for both algorithms, and after approximately generation 200 the number of living rats sharply declines. This decline is a result of rats having their energy values decreased to 0, a constraint put in place by the model itself, and does not relate to either algorithm.

9. FURTHER WORK

One beneficial next step to take following this study is to observe how changing some of the specifics of the model used affects the final population of living rats. One of these variables to change would be the size of the landscape and/or oasis. Changing this would affect the density of plants in the landscape, and perhaps alter the types of rats that would be successful. A second variable to change would be the rate which new plants are introduced to the landscape. Changing this variable would also affect plant density.

A second step to take would be to alter the internal variables that affect offspring generation for each algorithm, such as mutation rate, scaling factor, and/or crossover frequency. Altering these values would affect the type of offspring produced and ultimately the population as a whole, although the effect will likely be profound in DE due to it using both scaling factor and crossover frequency.

10. REFERENCES

[1] Conrad Barski. *Land of Lisp : learn to program in Lisp, one game at a time!* No Starch Press, Inc., 2011.

[2] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Evolutionary Computation*, 15(1), February 2011.

[3] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91, January 2006.

[4] Tea Robic and Bogdan Filipic. Demo: Differential evolution for multiobjective optimization. In *In*

Proceedings of the 3rd International Conference on Evolutionary MultiCriterion Optimization (EMO 2005), pages 520–533. Springer. LNCS, 2005.

11. LISTINGS

All source code and example output is provided below.

```
who= cs472 3 White,Steven
here= /home/swhite24/github/cs472/project3/472.3
total 60
 4 drwxr-xr-x 4 swhite24 swhite24  4096 2012-04-01 17:03 .
 4 drwxr-xr-x 4 swhite24 swhite24  4096 2012-03-12 17:29 ..
 4 -rw-r--r-- 1 swhite24 swhite24  1537 2012-03-31 02:51 de.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   698 2012-03-17 04:03 gade_cf_genes.lisp
 4 -rw-r--r-- 1 swhite24 swhite24  2576 2012-03-17 03:58 gade_genes.txt
12 -rw-r--r-- 1 swhite24 swhite24 10594 2012-03-31 17:48 gade.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   424 2012-03-17 03:56 gade_runtimes.txt
 4 -rw-r--r-- 1 swhite24 swhite24  1616 2012-03-31 03:03 ga.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   659 2012-03-17 03:13 get_genes.lisp
 4 -rw-r--r-- 1 swhite24 swhite24   988 2012-03-17 03:49 get_runtimes.lisp
 4 drwxr-xr-x 2 swhite24 swhite24  4096 2012-04-01 16:57 latex
 4 -rw-r--r-- 1 swhite24 swhite24   232 2012-04-01 17:02 main.lisp
 4 drwxr-xr-x 2 swhite24 swhite24  4096 2012-04-01 17:03 unused


--------------------------------------
running ...

;testing  !RANDS
;testing  !TIME-IT
;testing  !SHELL->OUTPUT
; fail : expected (boot.lisp city.dot city.dot.png go go_bash go_lisp
                   graph-util.lisp known-city.dot known-city.dot.png
                   tricks.lisp wumpus.lisp)
; pass : 2 =  66.7%
; fail : 1 =  33.3%
NIL





=====|  de.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; DE specific ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun run_de (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
               (n 10) (summarize_freq 10) (np 10000)
               (obj_func #'single_obj)
               (compare_func #'get_parent)
               (mem (make-rat_mem :c_freq c_freq
                                  :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n np
           summarize_freq obj_func compare_func mem))

(defun run_de_dec (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                   (n 10) (summarize_freq 10)
                   (obj_func #'single_obj)
                   (mem (make-rat_mem :c_freq c_freq
                                      :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_dec mem))

(defun run_de_obj (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
               (n 10) (summarize_freq 10)
               (obj_func #'single_obj)
               (mem (make-rat_mem :c_freq c_freq
                                  :scale_fact scale_fact)))
  (run_alg #'de_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_obj mem))

(defun de_candidate (mem parent)
  "produces a child gene based on parent and 3 others"
  (labels ((candidate1 (x y z)
             (min 10
                  (max 1 (round (+ x (* (rat_mem-scale_fact mem)
                                        (- y z)))))))
           (cross-over (parent child)
             (if (<= (randf 1.0) (rat_mem-c_freq mem))
                 parent child)))
```

```
    (mapcar #'cross-over (rat-genes parent)
            (mapcar #'candidate1
                    (rat-genes (any_rat mem))
                    (rat-genes (any_rat mem))
                    (rat-genes (any_rat mem))))))





=====|  gade_cf_genes.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Used to generate average gene with constant number ;;
;; of generations and varying scale factor ;;;;;;;;;;;;

(load "main.lisp")

(let ((pop_size 100000))
  (labels ((header (name size)
             (format t "~11:@<~a~>~8:@<~a~>" name size)))
    (dolist (sf '(0.1 0.3 0.5 0.7 0.9))
      (header 'ga_single sf)
      (run_ga :gens pop_size :n 1 :scale_fact sf)
      (header 'de_single sf)
      (run_de :gens pop_size :n 1 :scale_fact sf)
      (header 'ga_double sf)
      (run_ga :gens pop_size :n 1 :scale_fact sf :obj_func #'two_obj)
      (header 'de_double sf)
      (run_de :gens pop_size :n 1 :scale_fact sf :obj_func #'two_obj))))



=====|  gade_genes.txt |===========================
 GA_SINGLE   1000    6.01     7.21     4.93     6.68     2.59     6.31    12.12
    13.40
 DE_SINGLE   1000   18.96     5.35     2.94     5.53     7.58     5.63     7.55
    17.62
 GA_DOUBLE   1000    2.94     5.64     3.53     7.28     4.93     5.49     2.71
    6.53
 DE_DOUBLE   1000   10.70     4.35     6.36     8.88    11.84     6.14    12.17
    6.00
 GA_SINGLE   5000   54.55     4.77    75.93    16.95     1.48     1.54     2.64
    2.92
 DE_SINGLE   5000   13.75    78.92    34.59    26.51     5.33    33.78    10.01
    34.05
 GA_DOUBLE   5000    8.81     6.81     2.46     1.65     2.16    63.39    13.29
    2.91
 DE_DOUBLE   5000   61.89     8.92     6.91    19.92     6.66    22.96    23.76
    28.37
 GA_SINGLE  10000    2.98    14.45     1.51     1.62     1.96   100.23     7.63
    15.27
 DE_SINGLE  10000  327.53   113.31   411.14   102.55    68.50   223.75   135.91
   413.56
 GA_DOUBLE  10000    1.63    14.98     2.72     2.54     2.09     1.67   173.95
    22.47
 DE_DOUBLE  10000  183.98   219.06   165.08    86.48    59.39   249.06   337.27
   571.37
 GA_SINGLE  25000  257.71    32.27   194.02     1.93     3.63     4.26     1.66
    2.30
 DE_SINGLE  25000  481.62   332.41   507.76   327.48   304.84   526.40   359.08
   520.66
 GA_DOUBLE  25000  278.79    40.45     3.35     1.77     2.68   368.54     5.25
    1.37
 DE_DOUBLE  25000  488.83   449.00   524.23   358.87   265.82   484.74   547.44
   633.82
 GA_SINGLE  50000   71.76     2.23     3.00   215.00     1.82     1.61     2.15
    5.36
 DE_SINGLE  50000  435.55   576.70   446.32   385.15   277.90   523.55   473.81
   623.64
 GA_DOUBLE  50000   58.05     1.68     1.72     2.58     1.40   236.88     2.13
    8.28
 DE_DOUBLE  50000  618.03   392.88   507.87   362.57   335.74   530.75   395.61
   421.83
 GA_SINGLE  75000   82.35     8.11     9.67   235.22     1.88     1.34     5.20
```

```
      3.14
   DE_SINGLE  75000   563.64   524.74   573.30   387.86   320.18   530.18   389.94
     424.19
   GA_DOUBLE  75000   368.72    90.42     3.32     2.32     8.11   204.46     6.79
     2.12
   DE_DOUBLE  75000   571.30   572.15   350.84   391.42   429.05   367.91   418.42
     532.48
   GA_SINGLE  100000   40.48    17.32     1.37   328.92     1.51    10.71     3.67
     5.85
   DE_SINGLE  100000  625.13   493.91   442.01   499.01   314.43   368.95   593.81
     467.59
   GA_DOUBLE  100000   73.54     8.84     4.44   331.15     2.63     1.82     5.63
     5.06
   DE_DOUBLE  100000  348.16   510.24   471.66   469.08   137.23   507.52   566.67
     491.24


=====|  gade.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; GA & DE shared constants & functions ;;;;;;;;;;

;; size of landscape
(defparameter *width* 100)
(defparameter *height* 30)

;; oasis in landscape
(defparameter *jungle* '(45 10 10 10))

;; constants
(defparameter *plant_energy* 80)
(defparameter *reproduction-energy* 200)

;; counters
(defparameter *dead_rats* 0)
(defparameter *plants_eaten* 0)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Memory of all rats & plants ;;;;;;;;;;;;;;;;;;;
(defstruct rat_mem
  (c_freq 0.5)
  (scale_fact 0.3)
  ;; unique id for future rats
  (current 0)
  ;; generation counter, used for aging rats
  (gen 0)
  (np 50)
  (all_rats (make-hash-table :test #'equal))
  (all_plants (make-hash-table :test #'equal)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Struct for rat ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defstruct rat
  (x (randi *width*))
  (y (randi *height*))
  (energy 1000)
  (dir 0)
  ;; unique id of parent in all_rats
  (parent most-positive-fixnum)
  ;; generation when rat was created
  (creation -25)
  (id 0)
  (genes (loop for x from 1 to 8
               collect (1+ (randi 10)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rat_mem methods ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defmethod update_mem ((mem rat_mem) candidate_func freq
                       obj_func compare_func)
  "update each rat, add plants, and remove dead rats"
  (with-slots (all_plants all_rats gen) mem
    (maphash #'(lambda (key r)
```

```
                 (update_rat r mem candidate_func
                             obj_func compare_func)) all_rats)
    (add_plants mem)
    (kill_rats mem)
    (summarize mem freq)
    (incf gen)))

(defmethod add_rat ((mem rat_mem) new_rat)
  "add rat to list of living rats..if no rat is passed
   a default will be created"
  (with-slots (all_rats current) mem
    (setf (gethash current all_rats)
          (if (null new_rat)
              (make-rat :id current)
              new_rat))
    (incf current)))

(defmethod any_rat ((mem rat_mem))
  "select any currently living rat"
  (with-slots (all_rats current) mem
    (let ((rand (randi current)))
      (or (gethash rand all_rats)
          (any_rat mem)))))

(defmethod random_plant ((mem rat_mem) left top width height)
  "create plant within given constraints"
  (with-slots (all_plants) mem
    (let ((pos (cons (+ left (random width)) (+ top (random height)))))
      (setf (gethash pos all_plants) t))))

(defmethod add_plants ((mem rat_mem))
  "add plant in jungle and somewhere else"
  (apply #'random_plant (cons mem *jungle*))
  (random_plant mem 0 0 *width* *height*))

(defmethod kill_rats ((mem rat_mem))
  "kills off rats with energy <= 0"
  (with-slots (all_rats) mem
    (let ((dead_rats '()))
      (maphash #'(lambda (key r)
                   (if (<= (rat-energy r) 0)
                       (push key dead_rats)))
               all_rats)
      (dolist (key dead_rats)
        (incf *dead_rats*)
        (remhash key all_rats)))))

(defmethod summarize ((mem rat_mem) n)
  "prints summary of population every n generations"
  (with-slots (gen all_rats) mem
    (if (= (mod (rat_mem-gen mem) n) 0)
        (progn (format t "~%gen: ~a living rats: ~a "
                       gen (hash-table-count all_rats))
               (format t "dead rats: ~a plants eaten: ~a~%"
                       *dead_rats* *plants_eaten*)
               (average_gene mem)))))

(defmethod average_gene ((mem rat_mem))
  "print average gene for all living rats"
  (with-slots (all_rats) mem
    ;(format t "Average gene: ")
    (let ((total_gene (make-list 8 :initial-element 0)))
      (maphash #'(lambda (key r)
                   (setf total_gene (mapcar #'+ total_gene (rat-genes r))))
               all_rats)
      (mapc #'(lambda (x) (format t "~9:@<~,2F~>" x))
            (mapcar #'(lambda (x) (/ x (hash-table-count all_rats)))
                    total_gene))
      (format t "~%"))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Rat methods ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```lisp
(defmethod update_rat ((r rat) (mem rat_mem) candidate_func
                       obj_func compare_func)
  "a day in the life of a rat"
  (turn_rat r)
  (move_rat r)
  (eat_rat r mem)
  (reproduce_rat r mem candidate_func)
  (funcall obj_func r mem #'score_rat_energy
           (funcall compare_func r mem)))

(defmethod get_parent ((r rat) (mem rat_mem))
  (rat-parent r))

(defmethod move_rat ((r rat))
  "move rat in direction indicated by dir, then decrement energy"
  (with-slots (dir x y energy) r
    (setf x (mod (+ x (cond ((and (>= dir 2) (< dir 5)) 1)
                            ((or (= dir 1) (= dir 5)) 0)
                            (t -1))
                    *width*) *width*)
          y (mod (+ y (cond ((and (>= dir 0) (< dir 3)) -1)
                            ((and (>= dir 4) (< dir 7)) 1)
                            (t 0))
                    *height*) *height*))
    (decf energy)))

(defmethod turn_rat ((r rat))
  "turn rat based on genes"
  (with-slots (dir genes) r
    (let ((x (randi (apply #'+ genes))))
      (labels ((angle (genes x)
                 (let ((xnu (- x (car genes))))
                   (if (< xnu 0)
                       0
                       (1+ (angle (cdr genes) xnu))))))
        (setf dir
              (mod (+ dir (angle genes x)) 8))))))

(defmethod eat_rat ((r rat) (mem rat_mem))
  "check if rat is near plant, if so consume it and add
   plant_energy to rats curren energy"
  (with-slots (x y energy) r
    (with-slots (all_plants) mem
      (let ((pos (cons x y)))
        (when (gethash pos all_plants)
          (incf *plants_eaten*)
          (incf energy *plant_energy*)
          (remhash pos all_plants))))))

(defmethod reproduce_rat ((r rat) (mem rat_mem) candidate_func)
  "if a rat has enough energy, create a child with genes
   from de_candidate, then check who survives"
  (with-slots (energy id creation) r
    (when (and (>= energy *reproduction-energy*)
               (> (- (rat_mem-gen mem) creation) 25))
      (setf energy (ash energy -1))
      (let ((child (copy-structure r)))
        (setf (rat-genes child) (funcall candidate_func mem r)
              (rat-creation child) (rat_mem-gen mem)
              (rat-parent child) id
              (rat-id child) (rat_mem-current mem))
        (add_rat mem child)))))

(defmethod closest_dec ((r rat) (mem rat_mem))
  "return rat with most similar gene"
  (with-slots (genes) r
    (let ((distance most-positive-fixnum)
          best)
    (maphash #'(lambda (key val)
                 (let ((diff 0)
                       (genes1 (rat-genes val)))
                   (loop for i from 0 to 7 do
```

```lisp
                        (incf diff (abs (- (nth i genes)
                                           (nth i genes1)))))
                   (if (and (< diff distance)
                            (not (equal val r)))
                       (setf distance diff
                             best key))))
             (rat_mem-all_rats mem))
    best)))

(defmethod closest_obj ((r rat) (mem rat_mem))
  "return closest rat in objective space, i.e. the one
   with the most similar energy value"
  (let ((distance most-positive-fixnum)
        best)
    (maphash #'(lambda (key val)
                 (let ((new_dist (abs (- (score_rat_energy r mem)
                                         (score_rat_energy val mem)))))
                   (if (and (< new_dist distance)
                            (not (equal val r)))
                       (setf distance new_dist
                             best key))))
             (rat_mem-all_rats mem))
    best))

(defmethod single_obj ((r rat) (mem rat_mem) obj_func compare_to)
  "check if rat is old enough to be killed, then compare
   its score with parent score.  loser dies."
  (with-slots (energy creation) r
    (with-slots (all_rats gen) mem
      (when (and (= (- gen creation) 25)
                 (gethash compare_to all_rats))
        (let* ((p (gethash compare_to all_rats))
               (p_age (- gen (rat-creation p)))
               (c_score (funcall obj_func r mem))
               (p_score (funcall obj_func p mem)))
          (when (> p_age 25)
            (if (> c_score p_score)
                (and (remhash (rat-id p) all_rats)
                     (incf *dead_rats*)))
                ;(setf (rat-energy (gethash compare_to all_rats)) 0))
            (if (< c_score p_score)
                ;(setf energy 0)
                (and (remhash (rat-id r) all_rats)
                     (incf *dead_rats*))))))))))

(defmethod two_obj ((r rat) (mem rat_mem) obj_func compare_to)
  "Check if rat is old enough to be killed, then compare
   energy and children score with parent.  If one dominates
   the other, the other dies. If neither dominates, both live."
  (with-slots (creation energy) r
    (with-slots (all_rats gen) mem
      (when (and (= (- gen creation) 25)
                 (gethash compare_to all_rats))
        (let* ((p (gethash compare_to all_rats))
               (p_age (- gen (rat-creation p)))
               (child_score_en (score_rat_energy r mem))
               (child_score_ch (score_rat_children r mem))
               (p_score_en (score_rat_energy p mem))
               (p_score_ch (score_rat_children p mem)))
          (when (>= p_age 25)
            (if (< child_score_en p_score_en)
                (if (<= child_score_ch p_score_ch)
                    (setf energy 0)))
            (if (< child_score_ch p_score_ch)
                (if (<= child_score_en p_score_en)
                    (setf energy 0)))
            (if (< p_score_en child_score_en)
                (if (<= p_score_ch child_score_ch)
                    (setf (rat-energy (gethash compare_to all_rats))
                          0)))
            (if (< p_score_ch child_score_ch)
                (if (<= p_score_en child_score_en)
                    (setf (rat-energy (gethash compare_to all_rats))
                          0)))))))))

(defmethod kill_rat ((r rat))
  "true if energy <= 0"
  (with-slots (energy) r
    (if (<= energy 0)
        (progn (incf *dead_rats*) t))))

(defmethod score_rat_genes ((r rat))
  "scores rat based on percentage of values in either
   the front or back of gene, max of 1"
  (with-slots (genes) r
    (let ((genes_sum (apply #'+ genes))
          (early_sum (apply #'+ (butlast genes 5)))
          (late_sum (apply #'+ (last genes 3))))
      (max (/ early_sum genes_sum)
           (/ late_sum genes_sum)))))

(defmethod score_rat_children ((r rat) (mem rat_mem))
  "score rat based on the number times a child was
   successfully created, max of 1"
  (with-slots (creation id) r
    (with-slots (all_rats gen) mem
      (let ((child_count 0))
        (maphash #'(lambda (key r)
                     (when (= (rat-parent r) id)
                       (incf child_count)))
                 all_rats)
        (/ child_count
           (- gen creation))))))

(defmethod score_rat_energy ((r rat) (mem rat_mem))
  "scores rat based on energy, max of 1"
  (with-slots (energy) r
    (/ energy 1000)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Util Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun run_alg (alg c_freq scale_fact gens n init summarize_freq
                obj_func compare_func mem)
  (when (> n 0)
    (setf *dead_rats* 0)
    (setf *plants_eaten* 0)
    ;; initial population - large enough
    (dotimes (i init)
      (add_rat mem nil))
    ;; conduct generations
    (dotimes (i gens)
      (update_mem mem alg summarize_freq obj_func compare_func))
    (summarize mem summarize_freq)
    ;(average_gene mem)
    (run_alg alg c_freq scale_fact gens (1- n) init
             summarize_freq obj_func compare_func
             (make-rat_mem :c_freq c_freq
                           :scale_fact scale_fact))))

(defun print_rats (mem)
  (maphash #'(lambda (key r)
               (print r))
           (rat_mem-all_rats mem)))



=====|  gade_runtimes.txt |============================
   N     GA_SINGLE  DE_SINGLE  GA_DOUBLE  DE_DOUBLE
  1000     0.136      0.120      0.184      0.168
  5000     0.752      0.684      0.980      0.920
 10000     1.548      1.448      1.961      1.904
 25000     3.820      4.228      5.029      5.340
 50000     7.724     10.253      9.885     12.516
```

```
 75000    11.529     18.205     14.997     22.010
100000    15.225     28.289     20.206     32.770



=====|  ga.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; GA specific ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun run_ga (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
               (n 10) (summarize_freq 10) (np 10000)
               (obj_func #'single_obj)
               (compare_func #'get_parent)
               (mem (make-rat_mem :c_freq c_freq
                                  :scale_fact scale_fact)))
  (run_alg #'ga_candidate c_freq scale_fact gens n np
           summarize_freq obj_func compare_func mem))

(defun run_ga_dec (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                   (n 10) (summarize_freq 10)
                   (obj_func #'single_obj)
                   (compare_func #'get_parent)
                   (mem (make-rat_mem :c_freq c_freq
                                      :scale_fact scale_fact)))
  (run_alg #'ga_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_dec mem))

(defun run_ga_obj (&key (c_freq 0.5) (scale_fact 0.7) (gens 100)
                   (n 10) (summarize_freq 10)
                   (obj_func #'single_obj)
                   (compare_func #'get_parent)
                   (mem (make-rat_mem :c_freq c_freq
                                      :scale_fact scale_fact)))
  (run_alg #'ga_candidate c_freq scale_fact gens n 10
           summarize_freq obj_func #'closest_obj mem))

(defun ga_candidate (mem parent)
  "produce a child gene with mutation"
  (let* ((which (randi 8))
         (plus_minus (randi 2))
         (temp_genes (copy-list (rat-genes parent)))
         (gene_to_fiddle (elt temp_genes which)))
    (setf (elt temp_genes which)
          (min 10 (max 1
                       (if (= plus_minus 0)
                           (- gene_to_fiddle
                              (* gene_to_fiddle
                                 (rat_mem-scale_fact mem)))
                           (+ gene_to_fiddle
                              (* gene_to_fiddle
                                 (rat_mem-scale_fact mem)))))))
    temp_genes))



=====|  get_genes.lisp |===========================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Used to generate average gene for single and ;;;;;;;
;; multi-objective runs. ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "main.lisp")

(labels ((header (name size)
           (format t "~11:@<~a~>~8:@<~a~>" name size)))
    (dolist (pop_size '(1000 5000 10000 25000 50000
                        75000 100000))
      (header 'ga_single pop_size)
      (run_ga :gens pop_size :n 1)
      (header 'de_single pop_size)
      (run_de :gens pop_size :n 1)
      (header 'ga_double pop_size)
      (run_ga :gens pop_size :n 1 :obj_func #'two_obj)
      (header 'de_double pop_size)
      (run_de :gens pop_size :n 1 :obj_func #'two_obj)))


=====|  get_runtimes.lisp |============================
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Used to generate runtimes for each algorithm with ;;;
;; a varying number of generations ;;;;;;;;;;;;;;;;;;;;;

(load "main")

(format t "~8:@<~a~>~11:@<~a~>~11:@<~a~>~11:@<~a~>~11:@<~a~>~%"
        'n 'ga_single 'de_single 'ga_double 'de_double)

(let ((start_time (get-internal-run-time))
      (n          1))
  (dolist (pop_size '(1000 5000 10000 25000 50000 75000 100000))
    (format t "~8:@<~a~>" pop_size)
    (format t "~11:@<~,3f~>" (time-it n (run_ga :gens pop_size
                                               :n 1)))
    (format t "~11:@<~,3f~>" (time-it n (run_de :gens pop_size
                                               :n 1)))
    (format t "~11:@<~,3f~>" (time-it n (run_ga :gens pop_size
                                               :n 1
                                               :obj_func #'two_obj)))
    (format t "~11:@<~,3f~>~%" (time-it n (run_de :gens pop_size
                                                 :n 1
                                                 :obj_func #'two_obj))))
  (format t "~%~%Total time for all executions: ~f~%"
          (/ (- (get-internal-run-time) start_time)
             internal-time-units-per-second)))


=====|  latex |============================



=====|  main.lisp |============================
(handler-bind ((style-warning #'muffle-warning))
  (mapc 'load '(
                "../tricks.lisp"
                "unused/system.lisp"
                "unused/pick.lisp"
                "gade.lisp"
                "de.lisp"
                "ga.lisp"
                )))

(defun ! () (load "main.lisp"))

(defun main () (tests))



=====|  unused |============================
```