THE RANKABILITY OF DATA: AN EVOLUTIONARY OPTIMIZATION APPROACH

An essay submitted in partial fulfillment of

the requirements for graduation from the

Honors College at the College of Charleston

with a Bachelor of Science in

Mathematics

Isabel K. Johnston

May 2018

Advisor: Paul Anderson

The Rankability of Data: An Evolutionary Optimization Approach

Isabel Johnston

I. Introduction

A. What is Rankability?

We see applications of the science of ranking every day in the world and technology around us. From your Netflix recommendations, to the websites suggested to meet your googling needs, ranking tactics and algorithms are used to improve user experiences in a multitude of varying domains. However, while new and exciting approaches to ranking are being studied, little work is being done to research and ensure the validity and credibility of such ranking methods. This paper will explore the measure of the rankability of data, asking: "Does it make sense to rank this data set?", and "What does this ranking mean?" [1].

This paper will explore the ways in which rankability can be quantified as a function of a data set alone, without reliance on a specific ranking method. This paper is largely based on work done by Dr. Amy Langville and Dr. Paul Anderson from the College of Charleston, as well as Dr. Timothy Chartier from Davidson College in their unpublished paper *The Rankability of Data* [2].

B. Creating a New Pipeline

The strategy that is currently used in ranking problems is displayed by the solid lines in Figure 1. This shows a procedure without any consideration for the rankability of a data set, before implementing an actual ranking method. This demonstrates the problems that arise when no examination of rankability is put into place. After data collection, data is ranked and evaluated, potentially resulting in a ranking that is not trustworthy. In order to solve this issue, [2] has included a step which tests for rankability before selecting a ranking method. Therefore, with this new pipeline, only data sets that are deemed rankable will actually proceed to undergo a ranking method. However, if data is found to have low rankability, the pipeline will suggest data refinements–collecting more data and removing noise–leading again to the data processing stage. This results in a process that will continue to loop until the data is found to be adequately rankable. [2].

II. Quantifying Rankability

A. What do we want out of a Rankability Measure?

As outlined by [2], a valuable rankability measure must satisfy these three conditions: "(1) it must be effective, i.e., it must sync with our intuitive binary classification of structured datasets as rankable or unrankable;

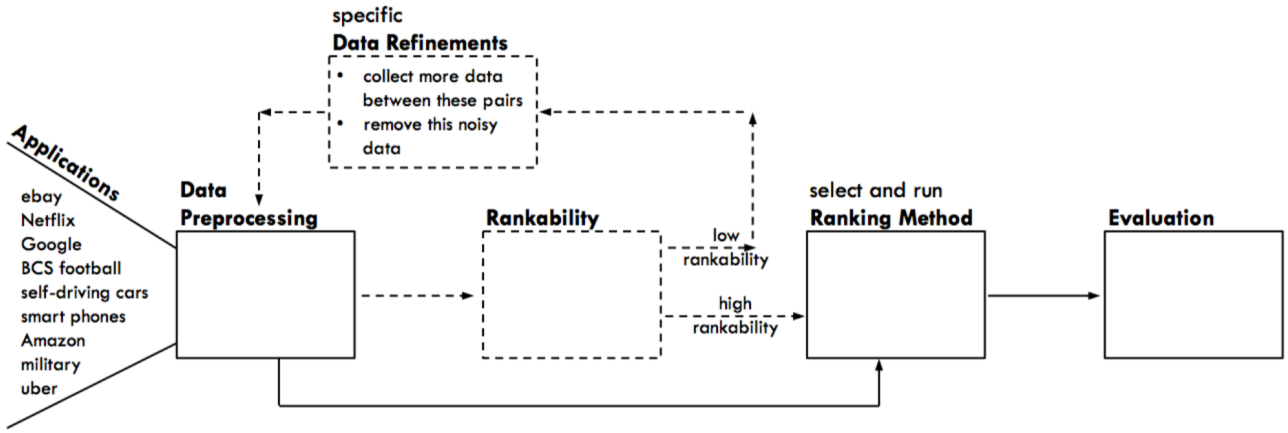


Fig. 1: **Improved Pipeline for Ranking.** The Ranking pipeline with added rankability check. This ensures that a data set is adequately rankable before implementing a ranking method. If data is found to have a low rankability score, then data refinements will be made, looping through the pipeline until it is found to have high rankability [2].

(2) it must be efficient, i.e., the time to compute the measure should be reasonable; and (3) it must be algorithm-agnostic, i.e., independent of a ranking or ranking method"[2]. This final quality is what sets this pursuit of rankability apart from most research. There exists methods for determining the value and effectiveness of certain ranking methods, however, these are dependent on the type of ranking used. Instead of getting a quality of ranking: $q = f(D, r)$ as a function of both the data set (D) and the ranking (r), [2] aims to develop a measure $r = f(D)$. Thus, uniquely retrieving a rankability measure which is dependant only on a data set.

B. Rankable vs. Unrankable Graphs

The graphs in Figure 2 represent a spectrum of data sets and relationships that vary from rankable to completely unrankable. The dominance graph is intuitively the ideal and most unquestionably rankable, with one evident ranking. On the other end, with the completely connected graph, it is difficult to discern a clear ranking. In fact, there are many rankings that would have equivalent logical values, thus leading us to believe that there is no ranking better than the others. Therefore, We can understand that this type of graph would represent the other side of the spectrum, as completely unrankable. Thus, in order to transition from a completely connected graph to a dominance graph, perturbations will be required. The next section will explain the variables used to represent these changes and the actual measure of rankability that has been developed.[2]

C. Rankability Measure

In this rankability measure, the variable k will be used to represent the minimum number of changes–meaning the minimal addition

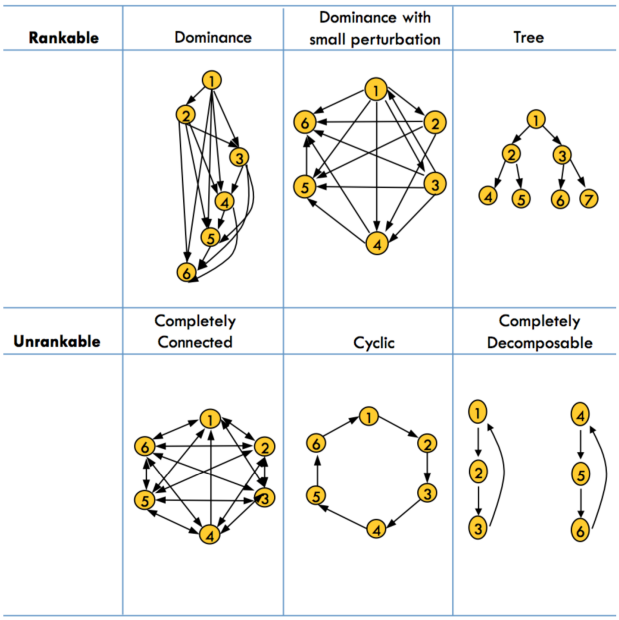


Fig. 2: **Rankable vs. Unrankable Graphs.** Various directed graphs categorized as rankable or unrankable [2].

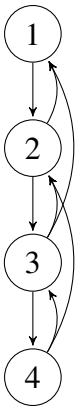
or removal of edges–necessary to transform any graph into a dominance graph. This will measure 'how far' a graph is from being a completely rankable dominance graph. We will also use variable p to represent the number of differing dominance graphs that can be made with only k changes. Finally, we introduce the set P of all of the rankings made with exactly k changes, such that $p = |P|$. Using these variables, [2] has created a rankability measure

$$(r = \frac{kp}{k_{max}p_{max}})$$

with k_{max} as the maximum number of changes for an n-node graph and p_{max} as the maximum number of rankings of an n-node graph. [2]

D. Example of the Rankability Measure

On the next page, we see an example of a directed graph expressed with nodes and edges. Then we have the adjacency matrix representing the directed graph to the right, with elements in row i, column j, reflecting the existence (1) or absence (0) of an edge. Thus, a dominance graph in matrix form would have an entire upper(or lower) triangular of ones and entire lower(or upper) triangular of zeroes. With this form, k can be found by counting the number of elements that deviate from this dominance form for every permutation of the matrix. In this example we see $k = 4$, as 4 changes are needed to transform this matrix into a dominance matrix. In order to do this, the three 1's in the upper triangle must become 0's–subsequently removing those edges–and 0 at $(4, 1)$ must become a 1–subsequently adding an edge. We see the representation of this new graph to the right. Thus, for this example we end up with $k = 4$ and $p = 6$.

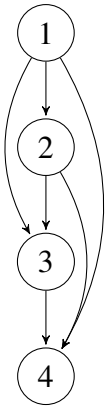


$$\longrightarrow \quad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Then find the permutations of the adjacency matrix with lowest k value. We find 6 such rankings...

$$P = \left\{ \begin{matrix} 3 & 3 & 3 & 4 & 4 & 4 \\ 1 & 4 & 4 & 2 & 3 & 3 \\ 4 & 1 & 2 & 3 & 1 & 2 \\ 2 & 2 & 1 & 1 & 2 & 1 \end{matrix} \right\}$$

$$\begin{array}{c} \\ 4 \\ 3 \\ 2 \\ 1 \end{array} \begin{array}{c} \begin{array}{cccc} 4 & 3 & 2 & 1 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & \mathbf{0} \\ \mathbf{1} & 0 & 1 & 1 \\ 0 & \mathbf{1} & 0 & 1 \\ 0 & 0 & \mathbf{1} & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \\ 4 \\ 3 \\ 2 \\ 1 \end{array} \begin{array}{c} \begin{array}{cccc} 4 & 3 & 2 & 1 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & \mathbf{1} \\ \mathbf{0} & 0 & 1 & 1 \\ 0 & \mathbf{0} & 0 & 1 \\ 0 & 0 & \mathbf{0} & 0 \end{bmatrix} \end{array}$$



E. Computing r

Through the process described above, we can find the entire P set, from which we get all of the variables needed for the calculation of rankability

$$r = \frac{kp}{k_{max}p_{max}}$$

While this process returns an exact result, in application this brute force method is far from time efficient. The computation of all the rankings made with exactly k changes, the P set, requires finding all possible permutations of the dominance graph. While this is manageable for smaller graphs, as soon as we get to graphs with 10 nodes, the brute force program written by [2] can take well over half an hour to get the minimum k value. With a time complexity of $n!$, brute force is impractical and burdensome to use, especially when it requires unreasonable computing time for such small graphs. In order to develop a more functional approach to computing these values for larger scale conditions, it was important to pursue more efficient methods. The main method that we focused on, Evolutionary Optimization, is outlined in the next section.

III. Evolutionary Optimization

A. In General

Evolutionary Algorithms can be utilized in a large variation of applications, however, the underlying idea behind each of these are the same. As you could suspect, these algorithms work similarly to natural selection: provided a population of individuals, under the pressures of their environmentl, the most fit individuals survive, thus producing a

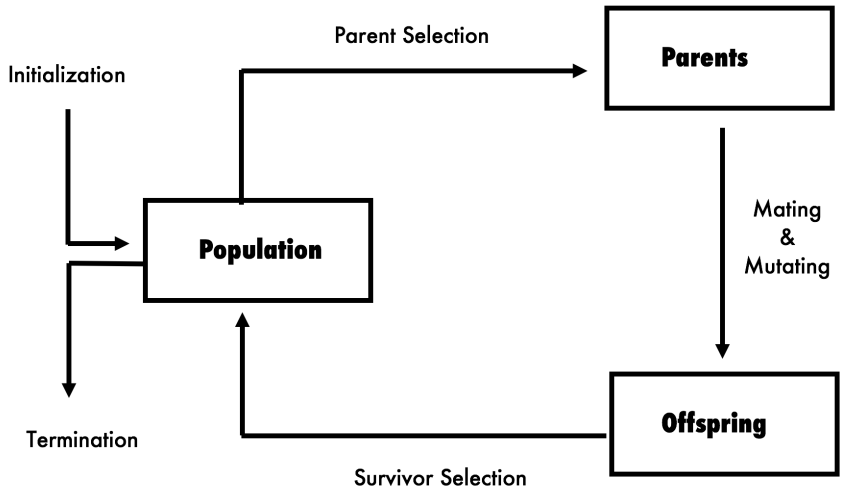


Fig. 3: **Evolutionary Optimization Flowchart** Evolutionary algorithms typically begin with a population which the try to improve through creating offspring by mating and mutating. The population then updates to be the fittest overall members of both the population and the offspring, and the process repeats until termination.

increasingly fit next generation–resulting in an overall increase in fitness of the entire population.

The general process for Evolutionary Algorithms is illustrated in figure [3]. Thus, based on the definition of fitness–dependant on the application–a fitness measurement is imposed on the population. Then, some of the most fit individuals (parents) are mated or combined together, ideally making more fit individuals(offspring). Additionally, in order to introduce variety and avoid local minimums, mutations are applied to the population. Then the best offspring compete with the parents to become part of the next population and this process iterates through until a sufficiently fit population is reached.

The general process described above is programmed in the following fashion. Evolutionary Optimization (EO), first generates an initial population P by randomly generating n solutions. Then it evaluates the solutions in P and updates the best solutions found (if necessary.) As the survival of the fittest practice goes, those with the best survival calculation are selected and randomly paired for combination. Each pairing generates one or more offspring to be added to P through mating and mutating. This process continues, selecting a new population based off of offspring, until a specified stopping condition is met and the best solution in the population P is given as an output.

B. Evolutionary Optimization Applied to Rankability

The use of Evolutionary Optimization (EO) to compute a rankability score was a promising solution to our time efficiency problems with the Brute Force method. The

extremely time consuming step of the Brute Force method is the calculation of the P set, where it is necessary to find all of the permutations of our dominance matrix. EO provided a way to find the best k values without having to calculate all of the possible rankings. Instead, we could find the best rankings and use the 'natural selection' of EO to produce the very best possible rankings.

In order to pursue a ranking value, the general process of EO described above is applied to our rankability problem. The initial population P has become a random yet minimal set of rankings for some $n \times n$ matrix. Then, for rankability, our 'survival calculation' is the calculation of the fitness (k) for each ranking in the population. We then take the top rankings and mate them with the mating method that is described below. Additionally, in order to avoid getting stuck in a local minimum–as a form of mutation–we throw random rankings into the population to mate with the rest. The results of these matings and mutations are the offspring that then make up the new population. This process is continued until the entire population has the same k value, or a certain number of iterations is reached. We will now discuss several options for the various steps within this process: selection, mating, and mutating. We will then present our Evolutionary Algorithm pseudocode.

C. Selection Methods

The selection step in the Evolutionary Optimization (EO) method is essential to the success of the entire algorithm, for it selects those that will go on to make up the new population. If this is done poorly, it can effect the entire process. Below is the first selection method that we came up with and the selection method that was actually implemented in the final code.

1) Probability Calculation for Pulling Rankings: For the purpose of the EO algorithm, the probability(p_k) that we pull a ranking should be based on how good it's k value is. The way in which we chose to do this is by making each ranking's probability calculated based on how much better it's k is than the very worst k value in the population (k_{max}.) Thus we have a value b where $b = \frac{k_{max}}{k}$, representing how much better k is than k_{max}.

It can be intuitively understood that we would then want a ranking k to have a probability of being selected that is b times more than the probability of selecting the worst k. Thus, we want p_k to be b times better than p_{kmax}. This gives us

$$(p_{kmax})(b) = p_k \quad \rightarrow (p_{kmax})(\frac{k_{max}}{k}) = p_k$$

Finally, we know that all of the probabilities of selecting each ranking must add up to one,

$$1 = p_1 + p_2 + ... + p_n$$

We the insert the equations above to get

$$1 = (p_{kmax})(\frac{k_{max}}{k_1} + \frac{k_{max}}{k_2} + ... + \frac{k_{max}}{k_n})$$

As an example of this selection, suppose we have a population P of 3 rankings, with the corresponding k values $\rightarrow [2, 3, 5]$. Following the selection algorithm, we have our worst k as $k_{max} = 5$. Thus, for $k = 2$, we want the ranking represented as the first element, to be selected $\frac{5}{2} = 2.5$ times more than $k_{max} \rightarrow (2)(2.5) = 5$. Therefore we want $(p_5)(2.5) = (p_2)$, and when we plug

these into our equation

$$1 = (p_5)(\frac{5}{2}+\frac{5}{3}+\frac{5}{5}) \rightarrow 1 = (p_5)(2.5+1.66+1)$$
$$\rightarrow 1 = (p_5)(5.16) \rightarrow p_5 \approx 0.194$$

Then based on the probability of selection the ranking with the worst k, $p_{kmax} = p_5$, we can get the probabilities of selecting the other elements.

$$(p_5)(\frac{5}{2}) = p_2 \rightarrow (0.194)(2.5) \approx 0.485 = p_2$$

$$(p_5)(\frac{5}{3}) = p_3 \rightarrow (0.194)(1.66) \approx 0.322 = p_3$$

Thus with this method, we will be selecting the ranking with k value $2 \approx 49\%$ of the time, the ranking with k value $3 \approx 32\%$ of the time, and our ranking with the worst k value of $5 \approx 19\%$ of the time.

2) Final Selection Method: While the selection method above accurately represented the variability of Evolution and natural selection, it actually was working 'too well.' When using the selection method above, the population became populated with extremely similar rankings. Thus, the mating only created more of the same ordered rankings. This resulted in convergences at local minimums, the program would produce k values that were far from the actual smallest possible k.

As a result, we transitioned to a more simple, yet effective new selection method. We directly select the rankings with the top k values, and mate them together. We then select a set of random rankings–chosen with no bearing on their k values, and mate them with the population. This new selection method served the code well, as it efficiently shocked the population with diversity and kept it from converging at a local minimum.

D. Mating and Mutating Methods

In our evolution optimization, we employed two different mating methods in order to combine fit members of our population to create equally fit offspring.

1) Borda Count: Borda Count is a ranking technique that scores each element in a ranked list based on the number of elements that said element outranks. These scores from multiple lists are then summed to create a final score, a Borda Count, for each element. The new list is then made by listing the elements in descending order. [1] While this ranking technique has promising applications, it is easily manipulated. We have used a similar technique, however, our strategy provides a more accurate ranking. Our modification of Borda Count scores each element in a list based on it's position in the list. However, before totaling the element scores, our method checks if an element has a consistent score. If an element is consistently at a certain score, our method will maintain that elements score in the resulting rank.

For example, in the rankings below, 2 should intuitively be ranked second, however, Borda Count would put 2 in first place. Our adaptation of the Borda Count method would maintain 2 in second place and use a variation of the Borda Count scoring method to rank the rest of the elements.

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} \begin{bmatrix}1\\2\\4\\3\end{bmatrix} \begin{bmatrix}4\\2\\1\\3\end{bmatrix} \begin{bmatrix}3\\2\\4\\1\end{bmatrix}$$

1: 3+3+1+0=7

2: 2+2+2+2=8

3: 1+0+0+3=4

```
population = [10 random permutations]
while population hasn't converged or iterations < 20 do
    for i = 1 : sizePopulation do
        for j = 1 : sizePopulation do
            offspring = mate(mate i, mate j)
            population.append(offspring)
    for i = sizePopulation do
        perm = random permutation
        population.append(perm)
    selectTopTen(population)
    k = minFit(matingArray)
```

Fig. 4: Above is the Pseudocode we used to calculate k.

4: 0+1+3+1=5

2) Left Tie Break Mate: Given two members of the population we would like to mate, if they have any element sharing the same rank in both members, then the offspring will also have that element in that position. For the rest of the elements, the numbers associated with the positions in each member are added, divided by two, and then the element is placed in the nearest open spot to that number. When doing this calculation for all members, if there is a tie, break that tie using the member on the left.

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} \begin{bmatrix}1\\2\\4\\3\end{bmatrix} \longrightarrow \begin{bmatrix}1\\2\\3\\4\end{bmatrix}$$

While in many instances, left tie break and Borda count have the same result, our results showed that in many cases left tie break converged more quickly to the optimal value, so we decided to use it for most experiments.

This method has been extremely successful, we can see this in Figure 5 below, where the pink line represents the minimum k value. Before the first iteration, the k values vary greatly with none of them at the minimum k value. However, after only one iteration, this method produces an offspring with the minimum k value and then quickly converges to have the entire population at the minimum k.

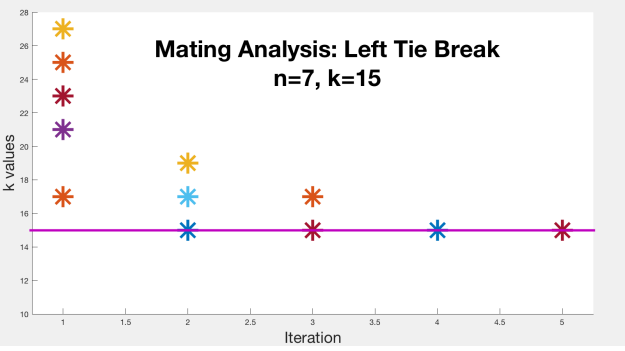


Fig. 5: The Left Tie Break mating method converges to the global minimum for k within only one iteration in this instance.

3) Mutating: We tried many mutating methods, but ultimately we wanted mutations

to add the most diverse members possible to our population to help break out of local minimums, so we decided our "mutations" should be the inclusion of completely random permutations. However, one mutating method, other than creating random permutations that arises later, is what we will call “single swap." Single swap merely selects two elements of our permutation to interchange at random. We don’t use this to create more diverse members of the population, but instead we use it to create similar members.

E. EO Pseudocode

As presented in figure 4, we begin with a population of random permutations, and then we mate all of them using a mating method of choice, and then, to add some diversity into the population, we add some random permutations, the equivalent of mutating. Finally, we calculate the fitness of all offspring, and select the best to continue. This process continues until we reach some predetermined number of iterations, 20 in this case, or if our entire population has converged to some k value. You can see the actual code from our program shown at the end of this paper.

F. Results of EO Algorithm

We compared the performance of our evolutionary optimization code with brute force, with very positive results. Our EO computed exact k most of the time for matrices with size 10 and under, and ran in significantly less time. The results can be seen graphically in figures [6] and [7] on the next page.

G. EO for P

When we run EO on a given matrix to calculate k, it turns out that we incidentally find several members of the set P, and once the algorithm terminates, we find than our final population consists mostly of members of P. Using the mutating method, single swap, described above, we were able to find many more members of P. For one sparse matrix, using this trick alone, helped us find 1000 members of P on average per minute, for a set P with $|P| = 1.2$ million. This method effectively improves our EO code, thus making EO more than comparable to Brute Force.

REFERENCES

[1] Langville,Amy N., and Carl D. Meyer. *Who’s #1? : The Science of Rating and Ranking.* Princeton: Princeton University Press, 2012. *eBook Academic Collection (EBSCOhost)*, EBSCOhost (accessed September 10, 2017).

[2] Anderson, P.E., Chartier, T.P., & Langville, A.N. (2018). *The Rankability of Data.* Unpublished paper.

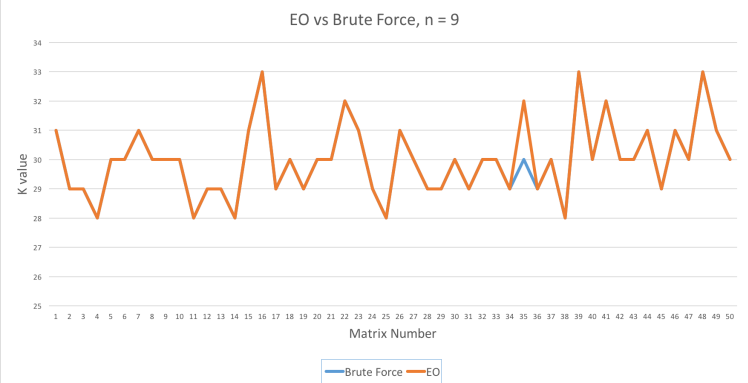


Fig. 6: **EO vs. Brute Force k values for size n=9**

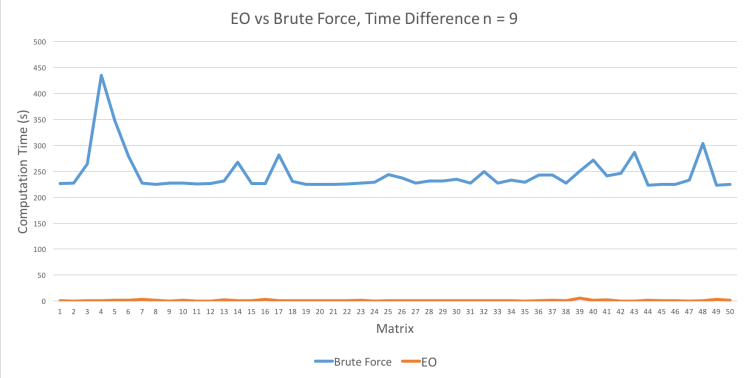


Fig. 7: **EO vs. Brute Force time for size n=9**

Evolutionary Optimization Matlab Functions

1 EO Function

```
tic;

%Collect information about data

[numRows, numCols] = size(D);
n = numRows;
D(1:n+1:end) = zeros(1,n); %make diagonals all 0

perm = [1:numRows];
maxP = factorial(numRows);
populationSize = 15;
maxk = numRows*numRows - numRows;

%First: Generate a population (i.e. generate rankings)
%       There are numRows! possible rankings

population = [];
for i = 1:populationSize
    perm = randperm(numRows);
    population = [population perm];
end

%Second: Calculate fitness
%As of now, fitness = k
%Need to find a way to keep track of permuatations and their fitness

fitArray = [];
 for i = 1:length(population)
     perm = population(:,i);
     fitness=nnz(tril(D(perm,perm)))+(n*(n-1)/2 -
        nnz(triu(D(perm,perm))));
     fitArray = [fitArray; fitness];
 end
[fitArray, index] = sort(fitArray);
```

```
population = population(:,index);
offspringArray =[]
w=2;
fitArrayOff = zeros(1,10);

%Mating and Mutating
while w<19 & populationSize~=nnz(min(fitArrayOff)==fitArrayOff)

    for i=1:length(population)
        for j=1:length(population)
            mate1 = population(:,i);
            mate2 = population(:,j);
            offspring = leftTieBreakMate(mate1,mate2);
            offspringArray = [offspringArray,offspring];
        end
    end

fitArrayOff= [];
    for i = 1:length(offspringArray);
        perm = offspringArray(:,i);
        fitness=nnz(tril(D(perm,perm)))+(n*(n-1)/2 -
           nnz(triu(D(perm,perm))));
        fitArrayOff = [fitArrayOff, fitness];
    end
[fitArrayOff, I] = sort(fitArrayOff);
offspringArray = offspringArray(:,I);
offspringArray = unique(offspringArray,rows, stable);
offspringArray = offspringArray;
population = offspringArray(:,1:populationSize);
fitArrayOff = fitArrayOff(:,1:populationSize);


w=w+1;
end
k=min(fitArrayOff)
offspringArray=[];
oldPopulation = zeros(1);
z=0;

toc;
```

2 Left Tie Brake Mating Method

```
function y = leftTieBreakMate(leftMate, rightMate)

%Information needed to loop through mates.
```

```
mateDimension = length(leftMate);
usedLeftIndices = zeros(mateDimension, 1);
usedRightIndices = zeros(mateDimension, 1);

%Allocate space forr offspring
offspring = zeros(mateDimension, 1);


%If both mates agree on a ranking, the the offspring will inherit
   the ranking.
for i = 1:mateDimension
  if leftMate(i) == rightMate(i)
    offspring(i) = leftMate(i);
    usedLeftIndices(i) = 1;
    usedRightIndices(i) = 1;
  end
end

%when the mates do not agree on a ranking, average the positions and
   use the left
%mates ranking as a tie breaker.
for i =1:mateDimension
  if usedLeftIndices(i)
    continue
  end
  for j = 1:mateDimension
    if usedRightIndices(j) || leftMate(i) ~= rightMate(j)
      continue
    end
      %when the average of left and right rankings is an integer,
         and that
      %position is not filled in offspring, then place in desired
         place.
      if mod((i+j)/2,1) == 0 && offspring((i+j)/2) == 0
        offspring((i+j)/2) = leftMate(i);
        usedLeftIndices(i) = 1;
        usedRightIndices(j) = 1;

      %when the average of left and right rankings is an integer,
         and that
      %position is filled in offspring, fill in next closest.
      elseif mod((i+j)/2, 1) == 0
        openPositions = find(~offspring);
        closestPosition = openPositions(1);
        for k = 1:length(openPositions)
          if abs(openPositions(k) - ((i+j)/2)) < abs(closestPosition
             - ((i+j)/2))
            closestPosition = openPositions(k);
          end
        end
```

```
        offspring(closestPosition) = leftMate(i);
        usedLeftIndices(i) = 1;
        usedRightIndices(j) = 1;

      %when the average of left and right ranking is not an integer,
         place in
      %nearest position.
      elseif mod((i+j)/2, 1) ~= 0
        %Go closer to left ranking
        if i > ((i+j)/2)
          desiredPosition = ceil((i+j)/2);
        else
          desiredPosition = floor((i+j)/2);
        end

        if offspring(desiredPosition) == 0
          offspring(desiredPosition) = leftMate(i);
          usedLeftIndices(i) = 1;
          usedRightIndices(j) = 1;
        else
          openPositions = find(~offspring);
          closestPosition = openPositions(1);
          for k = 1:length(openPositions)
            if abs(openPositions(k) - desiredPosition) <
               abs(closestPosition - desiredPosition)
              closestPosition = openPositions(k);
            end
          end
          offspring(closestPosition) = leftMate(i);
          usedLeftIndices(i) = 1;
          usedRightIndices(j) = 1;
        end
      end
    end
  end
y = offspring;
end
```

3 Borda Count Mating Method

```
function y = BordaMethod(mate1, mate2)

   n = length(mate1)
   B = [];
   C = [];
   F = [];
```

```
  for i=1:n
      if mate1(i,1)==mate2(i,1)
          offspring(i,1)=mate1(i,1);
          B=[B,i];
      end
      if mate1(i)~=mate2(i)
          j=find(mate2==mate1(i));
          C=[C; mate1(i),i+j];
      end

  end
  F=sortrows(C,2);
  E=setdiff([1:n],B);
  j=0;

  for i=E
        offspring(i,1)=F(j+1,1);
        j=j+1;
  end

y = offspring;
end
```

4 Selection Probability Method

```
function z =probabilitySelect(firstGenFit)

%Probability of choosing a member is proportional to its fitness

distinctMeme = [];
postnMult = [];

firstGenFit
distinctMeme = unique(firstGenFit)
counts = histc(firstGenFit(:),distinctMeme)


totper = 0
maxk = max(distinctMeme)

for i = 1:length(distinctMeme);
    maxovk = maxk/(distinctMeme(i,1));
    totper = totper + maxovk;

end
```

```
pmax = 1/totper
pk=[]
for i = 1:length(distinctMeme);
    maxovk = maxk/(distinctMeme(i,1));
    pmulk= pmax * maxovk ; %pmulk total probability for all
       permutations with fitness k
    pk = [pk repelem(pmulk/ counts(i),counts(i))];


T = [[1:10], firstGenFit]
U = sortrows(T,2)
V = [U , pk]
```