Will you live see your program end?

Chuck Cartledge

February 8, 2017

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

In the world of Computer Science, there are all sorts of problems. Problems that can be easily stated, but not so easily answered. Problems for which there is a single, optimal answer that can be arrived at in a reasonable length of time. Problems for which an optimal answer can not be arrived at using the world's fastest computers before the universe comes to an end. Some of these problems are simple, and some are complex. We will be taking a look at some of these problems in general, and then look how some can be attacked using R.

2 Discussion

A simple place to start is talking about how long programs take to run is *sorting*.

2.1 Bubble sort

A bubble sort is a type of exchange sort, where key K_n in record R_n is compared to key K_{n+1} in record R_{n+1}. If the keys are in the wrong order, then the records are swapped[4]. Pseudo code for a bubble sort is provided (see Algorithm 1). Examining the algorithm, we can see that:

1. The outer loop sets the limits for the inner loop, and
2. The inner loop is executed Σ_2^n times.

Meaning that the comparison is executed Σ_2^n times. To quantify how often the comparison is made, we:

$$\begin{aligned} Comparisons &= \Sigma_2^n \\ &= \frac{n*(n+1)}{2} - 1 \\ &= \frac{n^2+n}{2} - 1 \end{aligned} \tag{1}$$

Based on this analysis (see Equation 1), the exact, and approximate number of comparisons can be computed based on selected values of n (see Table 1). The approximate values are: $\frac{n^2}{2}$, usually the approximate

Table 1: Bubble sort comparisons for selected input sizes. The approximate values are: $\frac{n^2}{2}$, usually the approximate value is reduced further to n^2 (this is called *big O*, written as $O(n^2)$)when comparing different algorithms. When comparing different algorithms, the performance of the algorithm is what is important with large data sets, vice the relatively small scaling factors of the implementation.

n	**Exact value**	**Approximate**
10	45	50
100	4,950	5,000
1,000	499,500	500,000
10,000	49,995,000	50,000,000
100,000	4,999,950,000	5,000,000,000
1,000,000	49,999,500,000	50,000,000,000

value is reduced further to n^2 (this is called *big O*, written as $O(n^2)$)when comparing different algorithms. When comparing different algorithms, the performance of the algorithm is what is important with large data sets, vice the relatively small scaling factors of the implementation.

Data: Collection of records (R), each with a key value(K)
Result: R a list of records sorted by key value, from lowest key to highest
$n \leftarrow$ number of records ;
for $i \leftarrow 1$ **to** $n - 1$ **do**
  for $j \leftarrow i + 1$ **to** n **do**
    if $K_i < K_j$ **then**
      Swap R_i and R_j
    end
  end
end

Algorithm 1: Bubble sort.

2.2 Counting sort

"Counting sort assumes that each of the n input elements is an integer in the range 1 to k.

Cormen, Leiserson, and Rivest [1]

The basic idea is to determine for an input element x, how many input elements are less than element x. Pseudo code for a counting sort is provided (see Algorithm 2). Examining the algorithm, we can see that:

1. There are 3 loops that are executed n times,
2. There are no comparisons,
3. The keys are used as indices.

Meaning that the length of time to sort the input data is $3n$.

Based on this analysis, the exact and approximate number of operations can be computed for selected values of n (see Table 2). The approximate values are: $3n$. The approximate value (called *big O*), written as $O(kn)$ when comparing different algorithms. When comparing different algorithms, the performance

Table 2: Counting sort comparisons for selected input sizes. The approximate values are: $3n$. The approximate value (called *big O*), written as $O(kn)$ when comparing different algorithms. When comparing different algorithms, the performance of the algorithm is what is important with large data sets, vice the relatively small scaling factors of the implementation.

n	Exact value	Approximate
10	30	30
100	300	300
1,000	3,000	3,000
10,000	30,000	30,000
100,000	300,000	300,000
1,000,000	3,000,000	3,000,000

of the algorithm is what is important with large data sets, vice the relatively small scaling factors of the implementation.

Data: Collection of records(R), each with a key value(K)
Result: B a list of records sorted by key value, from lowest key to highest
$n \leftarrow$ number of records ;
for $i \leftarrow 1$ **to** n **do**
  $C[i] \leftarrow 0$
end
for $i \leftarrow 1$ **to** n **do**
  $C[R[i]] \leftarrow C[R[i]] + 1$
end
$C[R[i]]$ *now contains the number of elements equal* $R[i]$;
for $i \leftarrow 2$ **to** n **do**
  $C[i] \leftarrow C[i] + C[i-1]$
end
$C[R[i]]$ *now contains the number of elements less than* $R[i]$;
for $i \leftarrow 1$ **to** n **do**
  $B[C[R[i]]] \leftarrow R[i]$;
  $C[R[i]] \leftarrow C[R[i]] - 1$
end

Algorithm 2: Counting sort.

3 Complexity

When talking about sorting (see Section 2), the idea of *big O* was introduced as a general way to talk about the number of operations different types of sorts took. By inference, the number of operations points to the length of time it would take for the algorithm to complete. While not a direct way to compute time, if one algorithm was $O(n^2)$ and another was $O(nk)$, the second would complete much, much faster than the first when n was large. O is not the only notation that is available when talking about the complexity of an algorithm (see Table 3). For our purposes, we will be interested in O versus the other possible notations.

One way to understand computational complexity is to compare the $O()$ of different sorts (see Figure 1).

Table 3: Asymptotic notations. Definitions taken from [3, 1].

Symbol	Name	Meaning/usage
Θ	big theta	denotes the set of all $g(n)$ such that there exist positive constants C, C' , and n_O with $Cf(n) < g(n) < C'f(n)$ for all $n \geq n_0$, this is the **average** case expected performance.
O	big micron	denotes the set of all constants C and n_O with $\|g(n)\| \leq Cf(n)$ for all $n \geq n_0$, this is the **worst** case expected performance.
Ω	big omega	denotes the set of all $g(n)$ such that there exist positive constants C and n_O with $g(n) \geq Cf(n)$ for all $n \geq n_0$, this is the **best** case expected performance.
o	small micron	for any positive constant C, there exists a constant $n > 0$ that $0 \leq f(n) < Cg(n)$ for all $n \geq n_0$
ω	small omega	for any positive constant C, there exists a constant $n > 0$ that $0 \leq Cg(n) < f(n)$ for all $n \geq n_0$

The O values in the image compare well with the values we derived (see Section 2). The effects of different $O()$ functions are enlightening (see Figure 2).

One way to look at the $O()$ evaluations of the sorting algorithms is that they are all of the form:
$O(fn^k)$
where:

- f is some function, that may be a constant, or even a function of n. f may have many terms, although most disappear as n becomes larger and larger.
- k is a constant positive exponent. It may be 0, or larger.

Given this perspective, we can say the sorting algorithms are *deterministic* because, given some input, if they complete then they will complete with the same output every time. And, that they will arrive at this output after sometime that we can estimate based on the O *polynomial*. Combining these two ideas, we can characterize the algorithms as *deterministic polynomials*. Deterministic polynomial programs will:

1. Always return the same output for the same input, and
2. If they complete, will complete with a time constrained by a polynomial based on the size of the input.

In the world of computer program/algorithm complexity and the universe of possible questions to be answered, there are a few levels of complexity (see Figure 3):

P: Polynomial time complexity – Whenever an algorithm, on an n sized input, takes at most n^c operations to solve the problem (for some fixed constant $c > 0$), it is called a polynomial time algorithm or is said to have polynomial time complexity[2].

NP: Non-deterministic polynomial time complexity – A set or property of computational decision problems solvable by a non-deterministic Turing Machine in a number of steps that is a polynomial function of the size of the input c^n[7]. A solution (also known as a certificate) to an NP problem can be verified in P time.

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Figure 1: Complexity comparisons for various sorting algorithms. Image taken from [5].

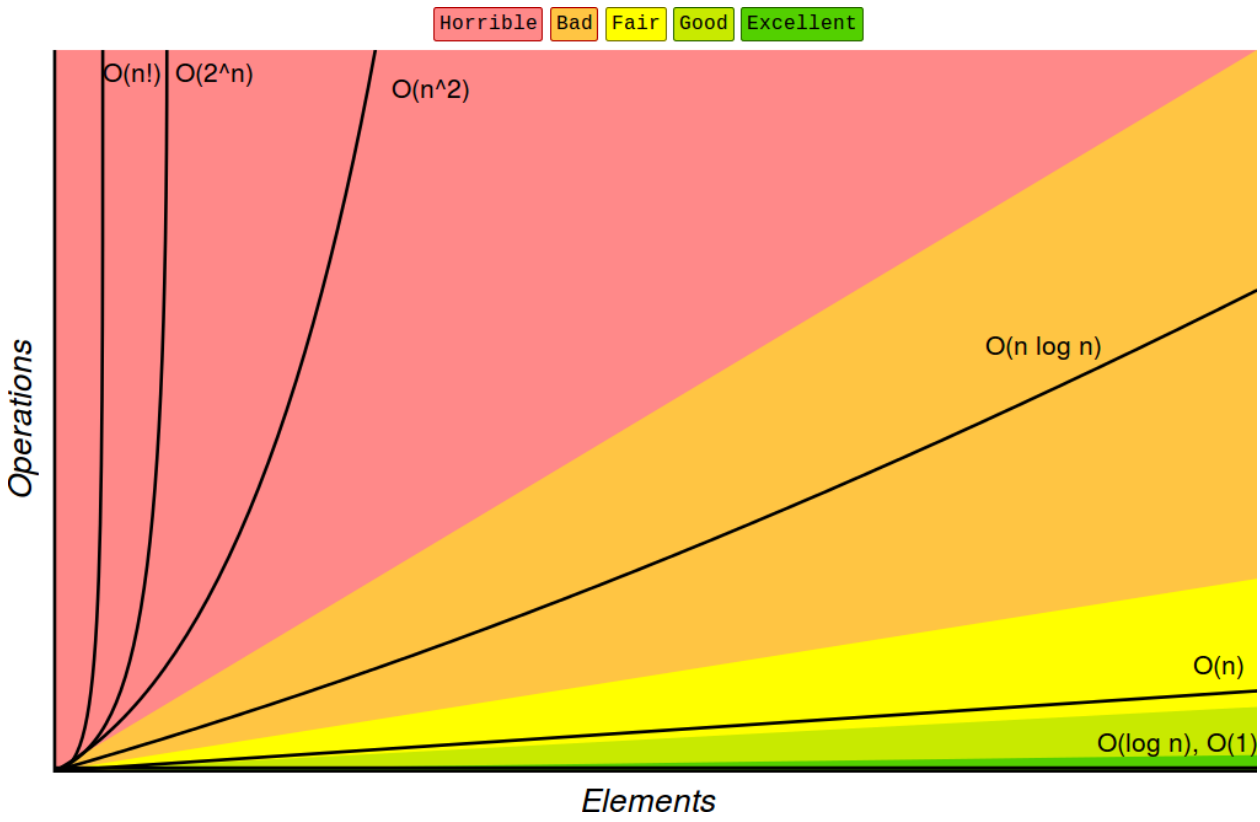

Figure 2: Complexity comparisons for various O functions and problem size. Image taken from [5].

NPH: Non-deterministic hard polynomial time complexity – Any input (language) can be reduced to or changed to another input (language) in NP in polynomial time, then the input (language) is NPH[1].

NPC: Non-deterministic complete polynomial time complexity – Any input (language) is in NP[1].

A partial list of NPC problems (some of these problems may be solvable for small n, but are intractable for large n)[1]:

1. 1-planarity – 1-planar graph is a graph that can be drawn in the Euclidean plane in such a way that each edge has at most one crossing point, where it crosses a single additional edge.
2. 3-dimensional matching – is a generalization of bipartite matching (also known as 2-dimensional matching) to 3-uniform hypergraphs.
3. Battleship – The Battleship puzzle (sometimes called Bimaru, Yubotu, Solitaire Battleships or Battleship Solitaire) is a logic puzzle based on the Battleship guessing game.
4. Bejeweled – Bejeweled is a tile-matching puzzle video game by PopCap Games, first developed for browsers in 2001.
5. Bin packing problem – objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used.
6. Bipartite dimension – or bi-clique cover number of a graph G = (V, E) is the minimum number of bicliques (that is complete bipartite subgraphs), needed to cover all edges in E.
7. Bulls and Cows (Master Mind) – is an old code-breaking mind or paper and pencil game for two or more players, predating the similar commercially marketed board game Mastermind. It is a game with numbers or words that may date back a century or more. It is played by two opponents.
8. Capacitated minimum spanning tree – is a minimal cost spanning tree of a graph that has a designated root node r and satisfies the capacity constraint c. The capacity constraint ensures that all subtrees (maximal subgraphs connected to the root by a single edge) incident on the root node r have no more than cc nodes.
9. Clique problem – is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph. It has several different formulations depending on which cliques, and what information about the cliques, should be found.
10. Knapsack problem, quadratic knapsack problem, and several variants – given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
11. Route inspection problem – is to find a shortest closed path or circuit that visits every edge of a (connected) undirected graph.
12. Traveling salesman problem – Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

NPC problems have the complexity form $O(k^n)$, making it virtually impossible to conduct exhaustive search for a solution through all possible solutions. Therefore optimization approaches are taken to arrive at solutions that are "good enough."

[1] `https://en.wikipedia.org/wiki/List_of_NP-complete_problems`

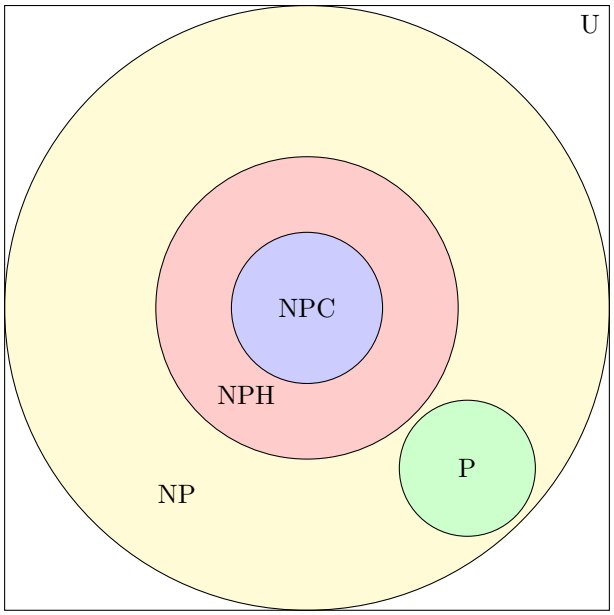

Figure 3: Venn complexity classification diagram.

4 Simple problems

> *"Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we dont know how to find an optimal solution in polynomial time. Even if a problem is NP- complete, there may be hope. We have at least three ways to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that we can solve in polynomial time. Third, we might come up with approaches to find near-optimal solutions in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an approximation algorithm."*
>
> Cormen, Leiserson, and Rivest [1]

For this investigation, we will be exploring genetic algorithms (GAs) using various R packages. GA are sometimes called "genetic programming."

A genetic algorithm (GA) is a non-deterministic approach evolutionary computational technology based on ideas from biology. GA is a systematic, domain-independent approach for having computers solve problems based on a high order description of what needs to be done. At a macro level (see Algorithm 3):

1. GA initially creates a random population of possible solutions to the abstract problem,
2. Each population generation is evaluated to see if the acceptance criterion is met, and if the search for a solution should continue. If works need to be done:
   (a) Parents are selected from the current population based on their individual closeness to the acceptance criteria,
   (b) The parents "breed" and create the next generation by having parts of their individual solution combined (a part from one parent is exchanged with a similarly located part in the other parent)

(c) Some parts of the new population are changed randomly (like biology spontaneous mutations).

Because GA is a randomly created and modified approach; it may never find a solution that meets the acceptance criteria, and it may get "stuck" at a local minima and not find the global minima.

Data: An evaluation function, acceptance criterion, a set of operators
Result: The best solution found within execution constraints
$Population \leftarrow$ operators randomly combined ;
while $Evaluate(Population) > acceptancecontinue = TRUE$ **do**
  $Parents \leftarrow Best(Population)$; $Population \leftarrow Parents(operators)$ Random selection ;
  $Population \leftarrow Mutate(Population)$ Random selection ;
end

Algorithm 3: Genetic algorithm.

4.1 Traveling salesman problem (TSP)

The traveling salesman problem (TSP) asks the question: given a set of cities, and a distance between connected cities, what is the shortest possible route that will visit all the cities exactly once? In a general form, given a graph G with vertices V and valued edges E, what is the shortest (least cost) path that visits each vertex exactly once. Using this approach, many different problems can be characterized as instances of a TSP.

Our TSP exploration is based on code from `http://rpubs.com/somasdhavala/GAeg`[2]. The modified code is included in this report (see Section 7). The program uses data from the "datasets" package, which is part of the default R installation. Specifically, two different datasets within the package can be processed by the code, based on the TRUE or FALSE assignment to the variable europeanFlag.

By default, the europeanFlag flag is set to TRUE, and a TSP is solved for 21 European cities. Three different plots are produced. They are:

1. The last and best solution (see Figure 4).
2. How the GA "closed" in on the locally best solution (see Figure 5).
3. How often a link between each city was considered (see Figure 6).

The program run time is a function of the number of generations the GA will create, and the size of the population of each generation. The multiplicative result of these two factors determines the number of individual solutions that must be evaluated. The more candidate solutions, the longer the run time. ga() attempts to maximize the value returned by the fitness function vice minimize, so the fitness function returns the inverse of the distance along the candidate route. A longer route will return a small value, a shorter route a larger value.

4.2 Curve fitting

Curve fitting is another application where GA can be used. Time series data can come from a variety of sources, and may represent natural or man-made processes. The curve fitting problem is to arrive at a set of operators that "fit" the data as closely as possible given the constraints of the available operators, and time to explore the solution space.

In this exploration, we will be using the R library rgp to compute a curve given data points. The source code for this exploration is embedded in this report (see Section 7). The program is configured to support

[2]The page does not give the author's name or contact information, so full attribution is not possible.

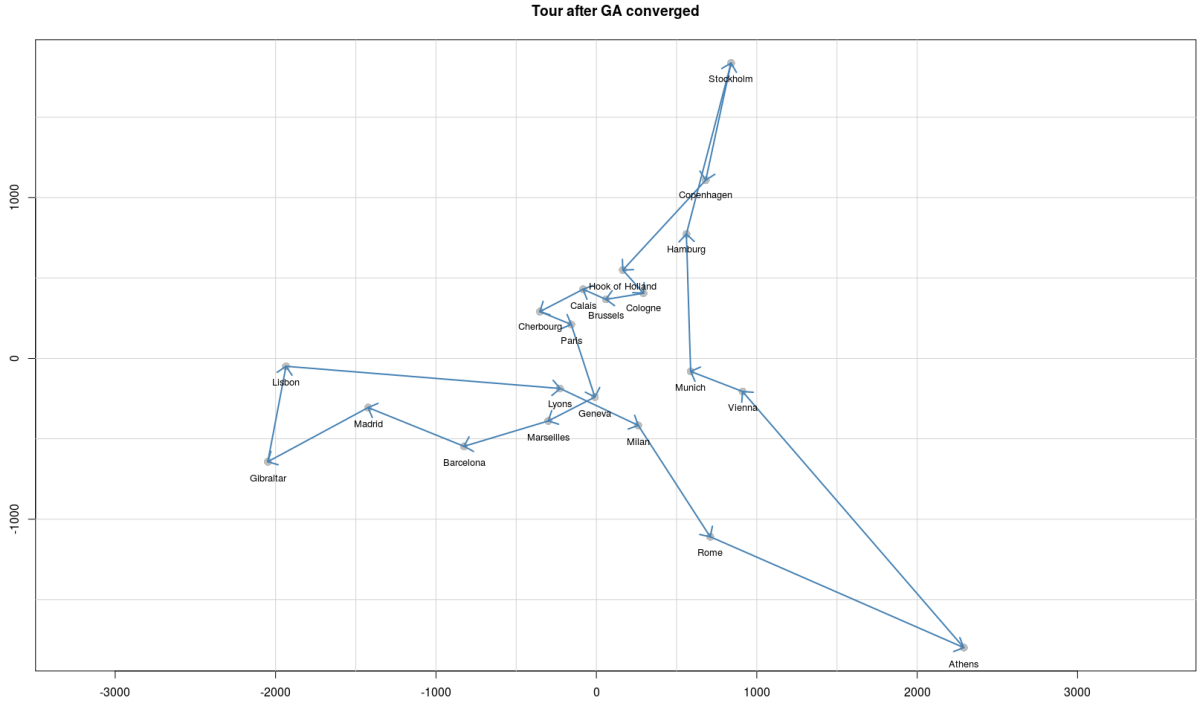

Figure 4: TSP solution for European cities.

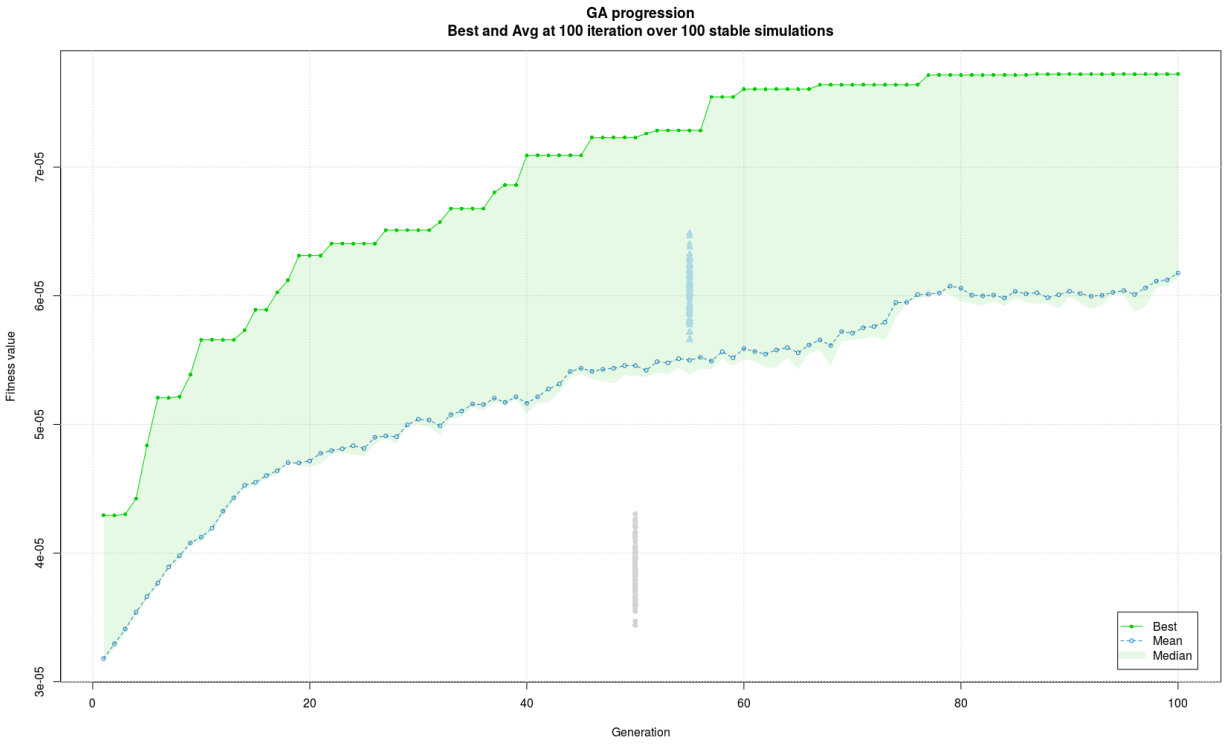

Figure 5: Final TSP solution search for European cities.

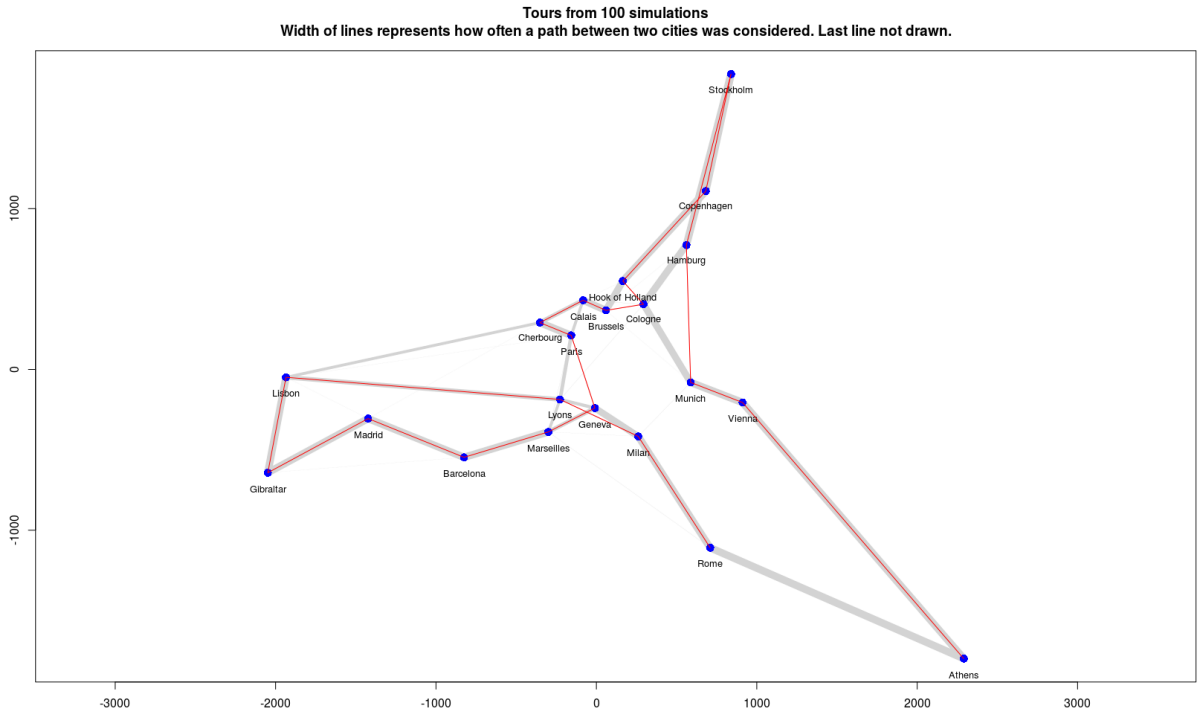

Figure 6: How often links between European cities were considered.

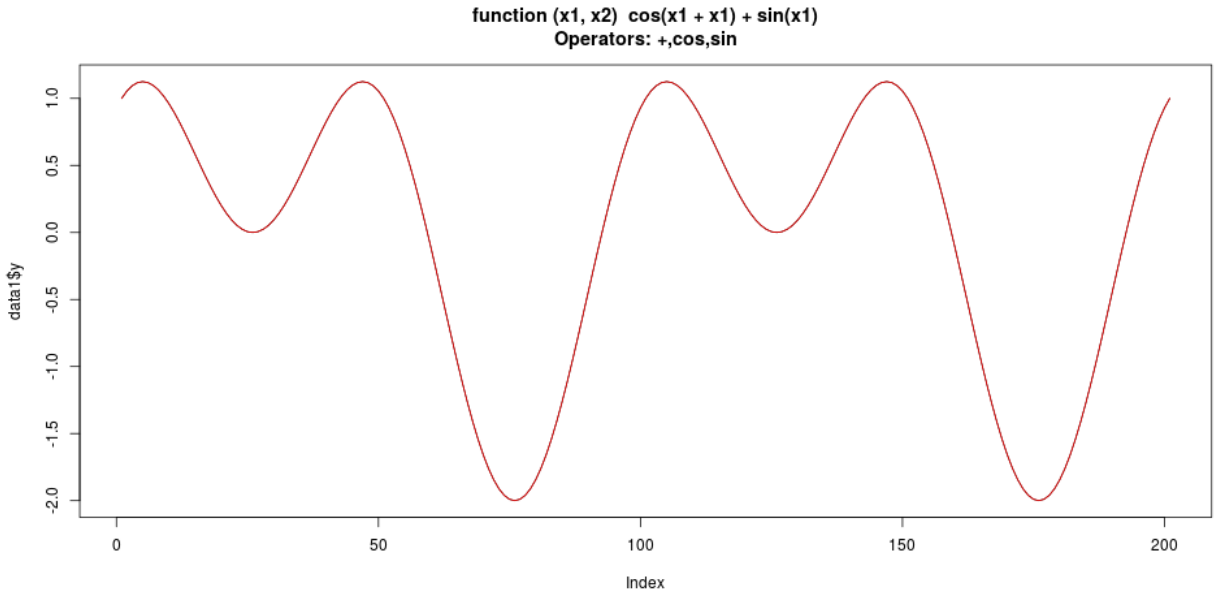

Figure 7: Test case 001, result 54. Generating equation:$y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$. Operators are:$+, cos, sin$

a wide variety of test cases, where each test case generate a set of values using different functions, and a different set of operators can be used in an attempt to arrive at a "good" solution. Here a solution means, a subset of the operators allowed and a set of constants used by those operators. At the end of each test case, the best solution is returned as a string, and the best solution is plotted as a series of points on the same plot as the original data. An ideal solution is one where the solution string is short and simple (vice long and complex), and the points are indistinguishable from the original data. Because GA is a random process, all combinations of operators are used. A long list of operators can lead to many explorations.

The results from test case 1 ($y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$) are provided:

Result 54. The program returned exact results using the operator set: $+, cos, sin$ (see Figure 7) (see Table 4).

Result 159. The program returned exact results using the operator set: $+, cos, sin, *$ (see Figure 8) (see Table 4).

Result 151. The program returned exact results using the operator set: $+, cos, tan, sin$ (see Figure 9) (see Table 4).

Result 64. The program returned long results within limits using the operator set:$+, cos, exp$ (see Figure 10) (see Table 4).

The selection of operators is crucial achieving good results. Operator selection appears to be more of an art than a science.

4.3 Knapsack problem

The knapsack problem is an combinatorial resource allocation optimization problem. Typically it is formulated a hiker must fill a knapsack with some collection of differently valued items, where each item has an associated weight, and the knapsack can only carry so much weight. In most cases the items going into the knapsack must fir entirely, or not at all. If partial items were permitted, then the problem could be solved

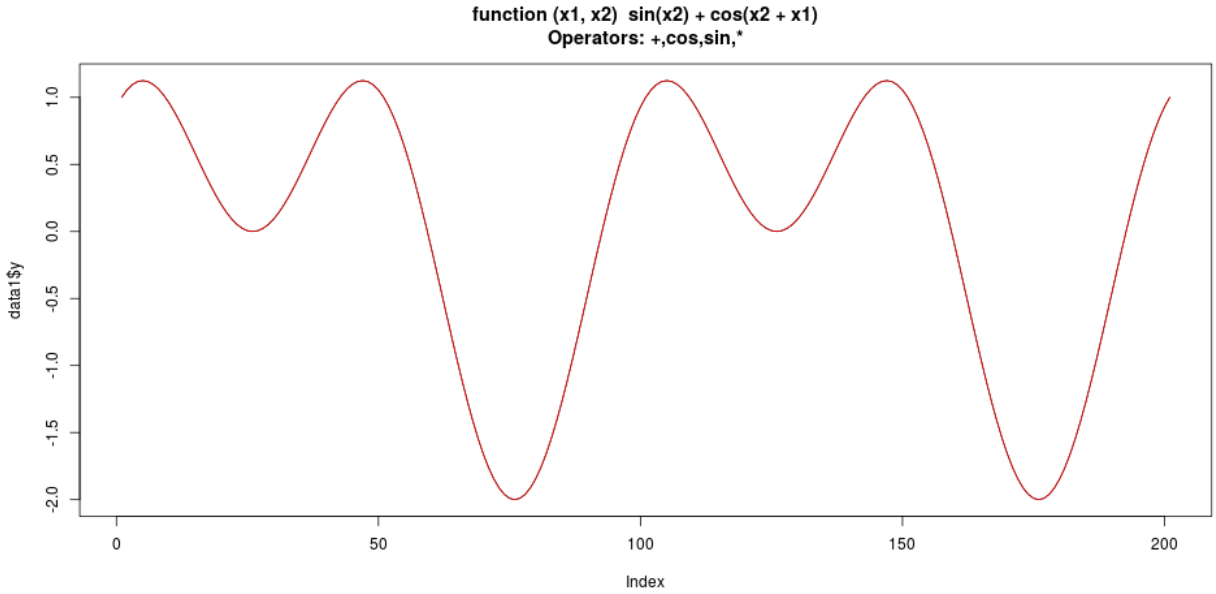

Figure 8: Test case 001, result 159. Generating equation:$y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$. Operators are:$+, cos, sin, *$

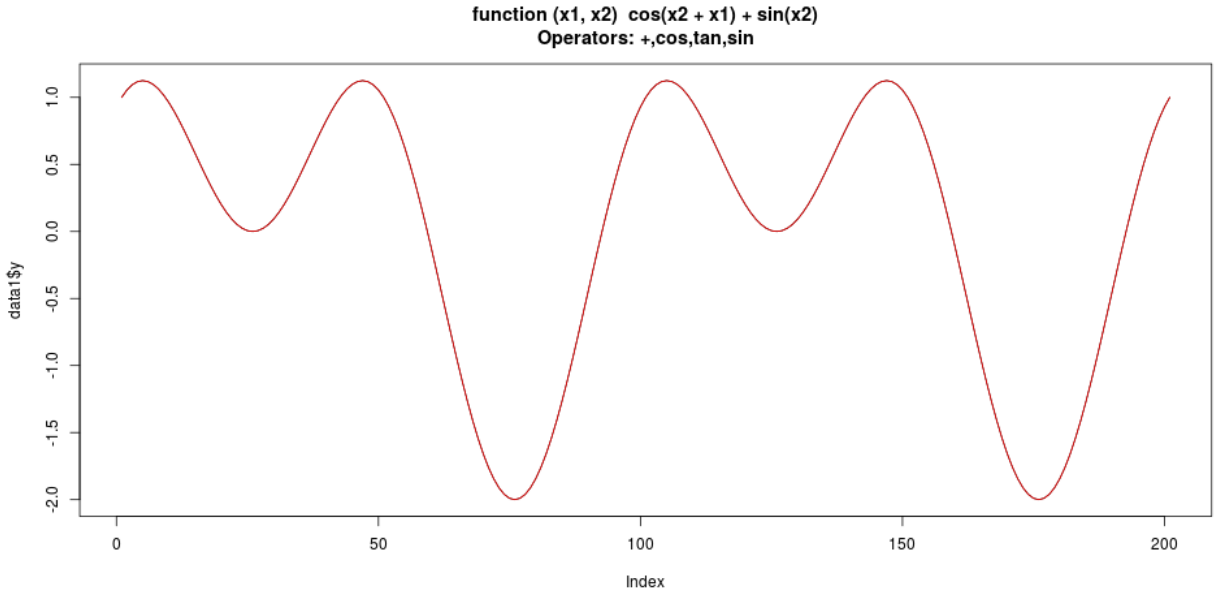

Figure 9: Test case 001, result 151. Generating equation:$y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$. Operators are:$+, cos, tan, sin$

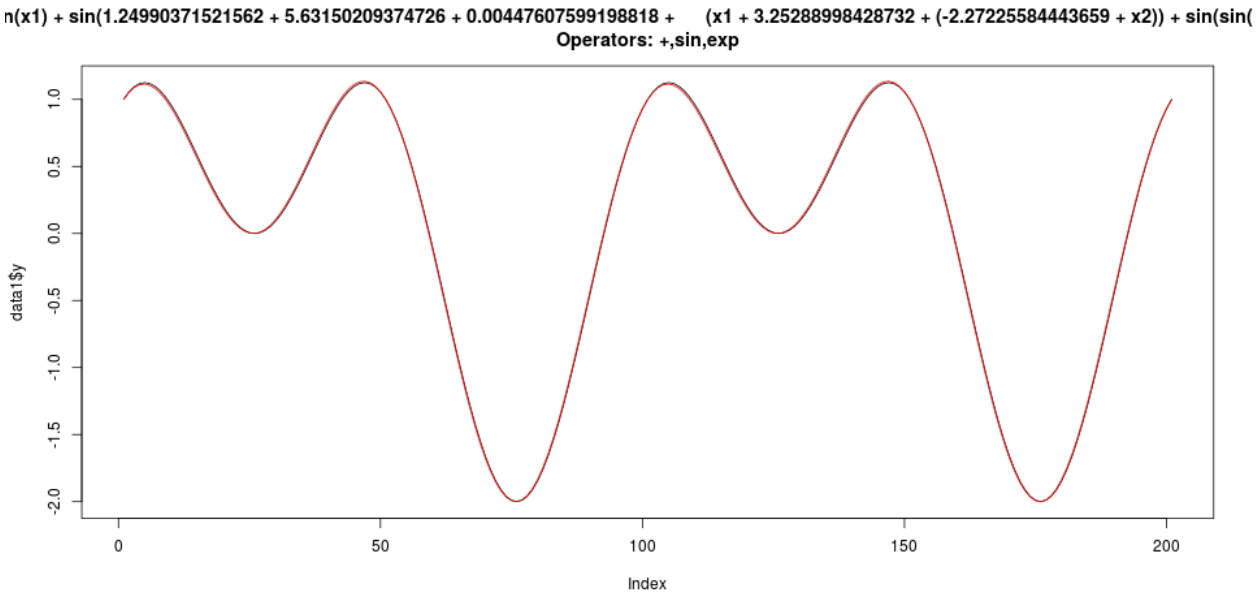

Figure 10: Test case 001, result 64. Generating equation:$y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$. Operators are:$+, cos, exp$

Table 4: GA results for test case 1. Generating equation:$y = \sin(x1) + \cos(2 * x2)$ where $x1 = x2$. A fitness value closer to 0.0 is better than one further greater than 0.

Figure	**Fitness**	**Function (x1, x2)**
Figure 7	0.00000000	cos(x1 + x1) + sin(x1)
Figure 8	0.00000000	cos(x2 + x1) + sin(x2)
Figure 9	0.00000000	sin(x2) + cos(x2 + x1)
Figure 10	0.01539919	sin(x1) + sin(1.24990371521562 + 5.63150209374726 + 0.00447607599198818 + (x1 + 3.25288998428732 + (-2.27225584443659 + x2)) + sin(sin(0.00929841306060553)))
	(simplified)	sin(x1) + sin(6.885882 + (x1 + 3.25288998428732 + (-2.27225584443659 + x2)) + 0.009298145)
	(simplified)	sin(x1) + sin(6.885882 + x1 + 3.25288998428732 + -2.27225584443659 + x2 + 0.009298145)
	(simplified)	sin(x1) + sin(x1 + x2 + 6.885882 + 3.25288998428732 + -2.27225584443659 + 0.009298145)
	(simplified)	sin(x1) + sin(x1 + x2 + 7.875814)

Table 5: Knapsack R script output

Item	Pts.	Weight	Binary inc.	Binary Wght	Floating inc.	Floating Wght
pocketknife	10	1	1	1	4.9263	4.9263
beans	20	5	1	5	0.4141	2.0706
potatoes	15	10	0	0	0.2482	2.4826
onions	2	1	1	1	0.3264	0.3264
sleeping bag	30	7	1	7	0.4676	3.2735
rope	10	5	1	5	0.4008	2.0042
compass	30	1	1	1	4.9063	4.9063

Table 6: Knapsack results for binary and floating point cases.

Result name	Binary result	Floating result
weight limit	20.000000	20.000000
final weight	20.000000	19.990266
survival pts	102.000000	227.152107

exactly via linear programming, but the restriction of all or nothing (1 or 0) for each item makes problem NPH[6].

Embedded in this document is a GA solution to the knapsack problem (see Section 7) based on the R library genalg. genalg() attempts to minimize a fitness value, so the evaluation function returns either:

0 – because the proposed solution exceeds the knapsack weight limit, or

- survival points – a more negative value is a better value.

The embedded program has two modes of operation based on whether the binaryCaseFlag is TRUE or FALSE. Sample run from the R script is provided (see Table 5). Different results from the binary and floating point algorithms are insightful (see Table 6) because the floating point solution appears to be counter intuitive. The compass has the highest survival points to weight ratio. So a maximal solution to the floating point option would be $30 * 20 = 600$ points, far from the 227 that the program stabilized at. It is possible that if the program were to run long enough, it would arrive at the 600 point solution, but it is unlikely. Depending on the arguments given to the rbga function, the number of items to include will be a random number based on the population size, and "seed" values given as pass parameters. In the case of the knapsack program, the random number will be between 0 and 5.

Both the binary and floating point best solutions converge to their final solution quickly (see Figure 13), and then the average of all solutions levels out close to the best.

5 Conclusion

At a 50,000 foot view, it is possible to divide the world of computer programs and algorithms into to two camps. One camp where a single optimal solution can be obtained in a reasonable amount of time based on the complexity of the algorithm. These algorithms have a complexity characterized by $O(fn^k)$ where k is normally a reasonably sized number. The other camp has complexity characterized by $O(fk^n)$ where the size of the solution space is an exponent of some small k.

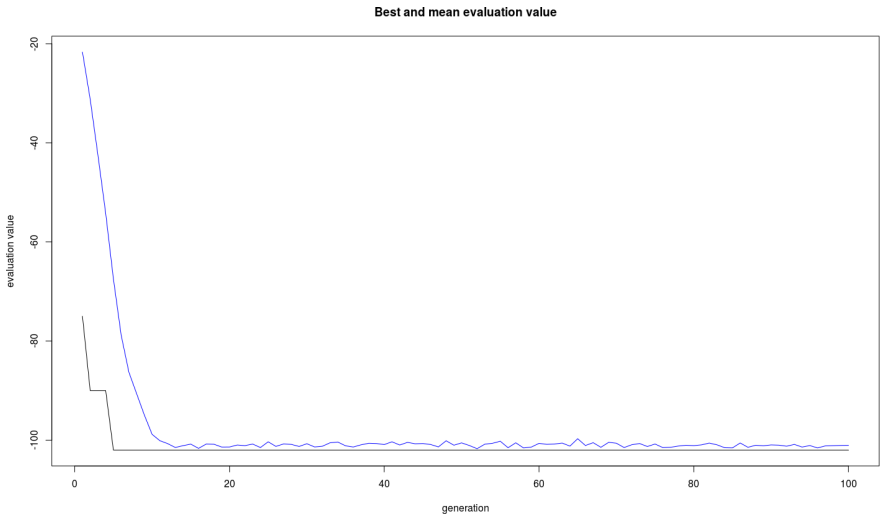

Figure 11: Knapsack problem binary solution.

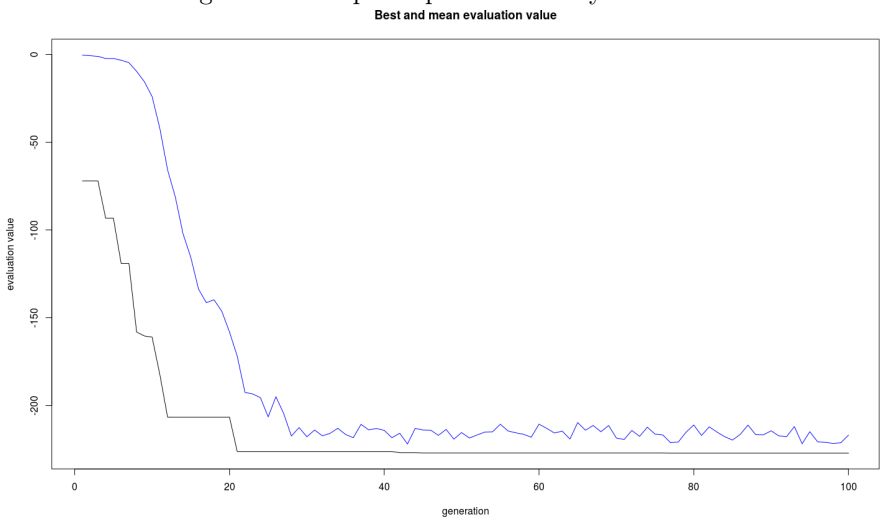

Figure 12: Knapsack floating point solution.

Figure 13: Comparison of binary and floating point knapsack solutions. The binary approach reaches its best solution faster and the mean stays closer to the best solution when compared to the behavior of the floating point solution because the binary has fewer "degrees of freedom."

The first camp is called *polynomial time*. The second is called *non-deterministic polynomial time*. Solutions in the first camp are optimal and exact. Because the solution space for the second camp can be too large for reasonably sized values of n, heuristic approaches are used to get an answer that is "close enough." Many of these heuristic approaches are called genetic algorithms (GAs). GAs are not guaranteed to arrive at the optimal solution, and may never terminate. But, they often work well enough to be useful and practical. The report includes a sample GS programs written in R that look at traditional knapsack, symbolic regression, and traveling salesman problems.

Genetic algorithms are interesting and useful approaches whose application can be more of an art then a science.

6 References

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The Massachusetts Institute of Technology, 1990.

[2] Purushottam Kar, *What is polynomial time complexity?*, `https://www.quora.com/What-is-polynomial-time-complexity`, 2014.

[3] Donald E Knuth, *BIG OMICRON AND BIG OMEGA AND BIG THETA*, ACM Sigact News **8** (1976), no. 2, 18–24.

[4] Donald Ervin Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3, Pearson Education, 1998.

[5] Eric Rowell, *Know thy complexities!*, `http://bigocheatsheet.com/`, 2016.

[6] Steven S Skiena, *The Algorithm Design Manual*, Springer-Verlag, 2008.

[7] Dictionary Staff, *nondeterministic polynomial time*, `http://www.dictionary.com/browse/nondeterministic-polynomial-time`, 2017.

7 Files

A collection of miscellaneous files mentioned in the report.

- knapsack.R – The classical knapsack problem, and it also allows partial items to be loaded.
- symbolicRegression.R – The test and investigation R script used during this investigation.
- tsp02.R – A traveling salesman program (TSP) using data available in the prepackaged datasets R package.