Mini Extended Essay

Research Question

How can different algorithms to create superpermutations be used and how do they compare in effectiveness?

Subject: Mathematics

Abstract word count: 94

Page count: 22

Author: Lorenzo Catani

School: International School in Genoa

Contents

Abstract

In mathematics, a superpermutation on n symbols is defined as a string which contains all the $n!$ permutations of n symbols as substrings of itself. For instance, on $n = 2$, the optimal superpermutation would be 121, since it contains both 12 and 21, all the permutations on 2 symbols, as substrings. The length of the optimal superpermutation on any n, checked for any $n \leq 5$, was conjectured to be $\sum_{i=0}^{n} i!$. In this essay, we explore different algorithms used to produce and conceptualise superpermutations, namely the recursive algorithm and the graphical one. Specifically, a python implementation of the latter approach will be used in order to create a graph for an optimal superpermutation on $n = 4$. Finally, we will explore Robin Houston's formalised proof which sets the lower bound for all superpermutations to $n! + (n-1)! + (n-2)! + (n-3)$.

1 Introdcution

In mathematics, the set of all permutations of a string of elements $(a_1, a_2, a_3, \ldots a_n)$ amounts to the totality of unique strings of length n in which each of these elements appears exactly once. Less formally, this set comprises of all the ways which one could, given a group of objects and an indefinite amount of time, arrange them using every object precisely once for each composition.

Using the last definition, it is evident that the number of possible permutations of a set of n objects amounts exactly to $n!$, the product of $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. This is because, while one is constructing a legal permutation (one in which each element appears once), one has a choice between n elements for the first digit of the permutation, a choice between $n-1$ for the second digit, a choice of $n-2$ for the third and, in general, a choice of $n-k$ for the digit in position $k+1$. Following this construction, the combined number of choices for all the digits, and therefore the number of possible permutations, is equivalent to the product of the number of choices for the singular digits, or $n \cdot (n-1) \cdot (n-2) \ldots \cdot 2 \cdot 1$, which is in fact equal to $n!$.

A superpermutation, on the other hand, is defined as a string formed by n symbols which contains as substrings all the $n!$ permutations of those symbols (for the sake of simplicity, this essay will only use integers as symbols, however, theoretically, they can be anything at all). The simplest possible superpermutation can be created by placing all the permutations of n symbols one after the other. The purpose of this essay is, however, to try to create optimal superpermutations, or superpermutations which include each permutation exactly once inside themselves. An example of an optimal superpermutation with $n = 3$ is shown below.

$$\underline{1\overline{2\mathit{3}}}\overline{\mathit{1}}\boldsymbol{\mathit{2}}\underline{\mathbf{1}\overline{\mathbf{3}2}}\overline{1}$$

Since, for $n = 3$, there are only 6 permutations, it isn't hard to separate the previous string of integers into all the single permutations in the order in which they appear:

$$\underline{123}, \overline{231}, \mathit{312}, \mathbf{213}, \underline{132}, \overline{321}$$

In this essay, the effectiveness of two main algorithms used to produce short superpermutations, namely the recursive algorithm and the graphical one, will be discussed and, in the end, the latter approach will be used to prove the conjectured lower bound for the length of superpermutations on any n of $n! + (n-1)! + (n-2)! + (n-3)$.

2 The Recursive Algorithm

The length, denoted $L(n)$, of superpermutations with $n < 5$ symbols can be relatively easily found using brute force computational methods which check if each combination (starting with length 1 then increasing) of the n symbols is a legal superpermutation and returns the smallest of these strings. This length amounts to 1, 3, 9 and 33 for respective values of n. Since modern computers can't possibly produce a similar string for $n = 5$ by brute force in reasonable time, it is necessary to find an algorithm to create them which can function given an arbitrary number of symbols.

A plausible candidate for such an algorithm is the *recursive algorithm*, which produces a superpermutation of $n + 1$ symbols given any superpermutation of length n after performing the steps below:

1. Separately writing out all the permutations in the original string of digits, in the order in which they appear.
2. Duplicating each of them and placing the desired symbol n in the middle.
3. Compressing everything together by removing overlaps of up to $n - 1$ digits between the single blocks.

The most basic, non-trivial example, namely that of going from $n = 2$ to $n = 3$ is shown below.

$$\begin{tikzcd}
 & 121 \arrow[dl] \arrow[dr] & \\
12 \arrow[d] & & 21 \arrow[d] \\
12312 \arrow[dr] & & 21321 \arrow[dl] \\
 & 1231\mathbf{22}1321 \arrow[d] & \\
 & 123121321 &
\end{tikzcd}$$

Following this line of reasoning, it is relatively easy to gain an intuitive understanding for why the strings produced by this algorithm effectively contain each of the $n!$ permutations of n digits. In fact, given a permutation of length n, we can produce a permutation of length $n-1$ simply by reading all the digits after the digit with value n. For example, given the permutation for $n = 5$ of 12534, we can produce a permutation of 4 digits by reading all the digits after the five in order. In this case, this would yield 3412. Furthermore, it is important to note that, since shifting each of the digits in the permutation with n symbols won't change the digits' arrangement, exactly n different permutations of n symbols, corresponding to different rotations of the original one, will produce equivalent permutations of length $n-1$. Applying this concept to the previous example, we notice that, like expected, the permutations **41253**, **34125**, **53412** and **25341** all produce the string 3412 when read starting from the digit of value 5. Therefore, conversely, if we take two copies of the permutation of length $n-1$ and place an n in the middle, then there will be n consecutive 'frames' each of which produces rotation of the original length n permutation (**3412** 5 **3412** produces **34125**, **41253**, **12534**, etc.). Since we are performing this task for each permutation of length $n-1$, we are necessarily producing every permutation of length n as well.

It is also possible to show, with a straightforward inductive argument, that this algorithm always produces superpermutations of length

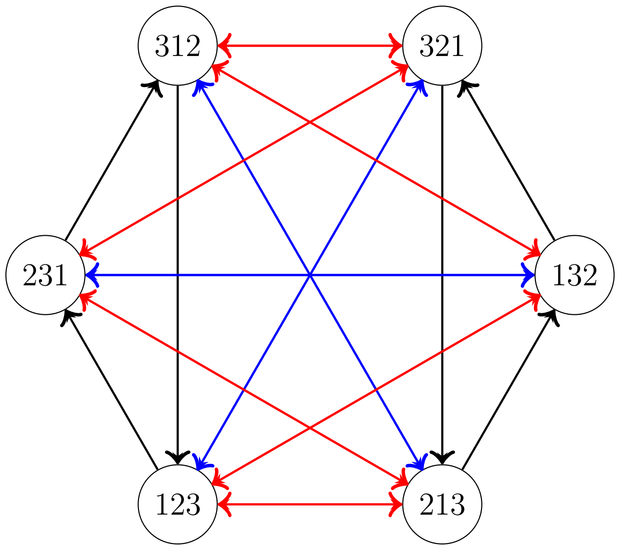
$$L(n) = \sum_{i=1}^{n} i! \tag{1}$$

for any n. First of all, it is easy to see that, applying this algorithm for the trivial case of $n = 1$ produces the length-1 string 1, whose length is obviously equivalent to $1! = 1$. For the inductive step, assuming this formula holds for any $k = n$, one can prove it also holds for any $k+1$ by noticing that, each time the recursive algorithm is applied to a superpermutation of length k, one is essentially adding strings of length $k+1$ (the repeated initial permutation of length k and a digit of value $k+1$) to each of the $k!$ permutations. Therefore, since overlaps from the original length k superpermutation are maintained in that of length $k+1$, the length of the final string will be exactly $(k+1)\cdot k! = (k+1)!$ digits longer than the first one. Therefore, since $L(n)$ holds for $n = 1$ and, if $L(n)$ holds then $L(n+1)$ holds as well, because of the principle of mathematical induction, $L(n)$ must hold for any $n \in \mathbb{Z}^{+}$.

Even if it was conjectured that this algorithm would produce minimal length superpermutations for any n, unfortunately, such a conjecture was proven false by the finding of a permutation of length 872 for $n = 6$, one shorter than the hypothesized lower bound, by Robin Houston. This superstring was found using the graphical approach, which is outlined below.

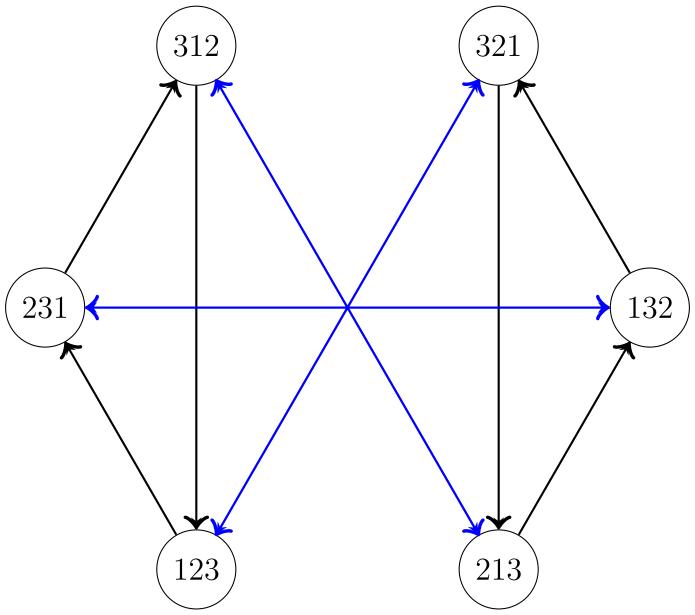
3 The graphical approach

In order to understand Robin Houston's method and construction, it is necessary to think of each of the $n!$ permutations of n digits as separate nodes on a directed weighted graph and a superpermutation as a minimum weight path which visits each of the nodes exactly once. In this graph, to start with, there would be a directed edge connecting each node, containing a permutation, to every other node of the graph. Each of these edges would also have a weight, denoted $w(p_1, p_2)$, where p_1 and p_2 are two permutations of n symbols, and therefore two nodes of the graph. This would represent the minimum number of digits, d, one would need to add to the end of p_1 such that the final string would also contain p_2 as a substring of itself. For instance, the edge connecting the permutation 123 to 231 would have $w = 1$ since adding a 1 at the end of the first string would produce 1231, which contains the second one as a substring of itself. The raw graph, with no optimisations, on $n = 3$, is shown below, where black arrows represent edges of weight 1, blue those of weight 2 and red those of weight 3.



As you can probably notice, this construction would produce a spectacular quantity of edges as n increases ($n!$ nodes multiplied by $n! - 1$ connections for each node) and therefore needs some refinement. The first optimisation one can make to this graph is to remove *improper edges*, or edges of weight $w > 1$ between two nodes which bypass another permutation when adding d characters. An example of these type of edges is the one connecting 123 to 312, with $w = 2$. In fact, when we add the digits 12 to end of 123 in order to produce 312 as a substring, we are also producing the permutation 231 in the process. One such edge would obviously be useless since it would have the same weight as the sum of the weight-one edge going from 123 to 231 and that going from 231 to 312, but would travel through one less node in the process. A consequence of the removal of improper edges is the removal of all edges with $w = n$. In fact, with these type of edges, by definition, the leading character of p_2, in the form $(b_1, b_2, \ldots, b_n)$ must be equal to one of the one of the characters, excluding the last one (because then the edge would have weight $n - 1$) of p_1, in the form $(a_1, a_2, \ldots, a_n)$. However, if we denote the index of the repeated character of p_1 as r, the concatenation of p_1 and p_2 would allow for another permutation, specifically that in the form $(a_{r+1}, a_{r+2}, \ldots, a_n, b_1, b_2 \ldots, b_{r-1})$, which would contain each of the n symbols, proving that all such edges are improper.

Returning to the previous graph on $n = 3$ and applying these optimisations in order to remove improper edges would result in the following representation:

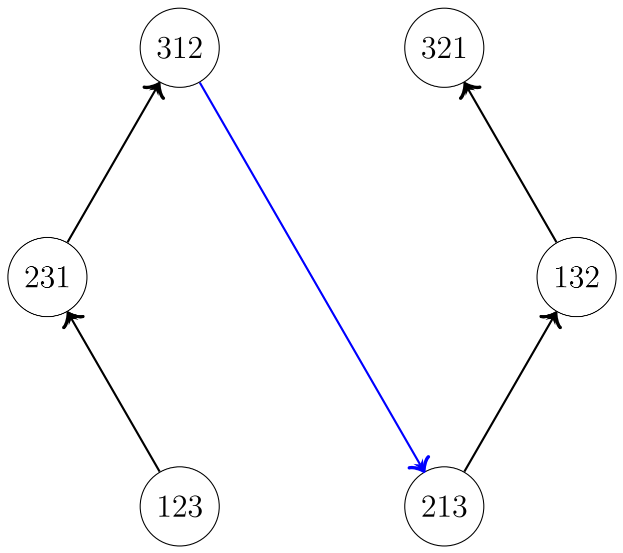


Now that we have a method for constructing graphs with all the permutations of n symbols which are relatively feasible to work with as n grows, it is necessary to find an efficient way to locate paths through this graph which include each node where the sum of all the weights of the edges is as low as possible. In the *Recursive solution* subsection, a highly symmetric recursive construction is explored while in the *The Travelling Salesman Problem* subsection, this problem will be looked at from the perspective of one of the most studied NP-hard problems, namely the Travelling Salesman problem, or TSP.

3.1 Recursive solution

One possible way to find low-weight paths through the graph described above is to use a a highly symmetric, recursive construction. In order to understand this algorithm, one must first introduce some notation. First of all, a w-cycle will be defined as a path which starts at the permutation p and travels through edges of weight w until it gets back to p. For example, a 1-cycle starting at the node containing 123 would go through 231, 312 before to the original node. Specifically, it is important to note that each of the 1-cycles of the graph will contain exactly n nodes and therefore $n-1$ edges (we are omitting the edge connecting the last node of the cycle back to the start). This is because each of the permutations in such a cycle correspond to a shift of the previous where each element in position k moves to position $k-1$ and the first element moves to the last place.

From here, we can find a short path through the graph by starting at the permutation $1, 2, \ldots, n-1, n$ (we could start at any node but this one was chosen for the sake of simplicity), following its 1-cycle then connecting the last element to another permutation via an edge of weight 2, following the 1-cycle yet again and repeating these steps until $n-1$ 1-cycles have been followed. At this point, it is necessary to travel through an edge of weight 3 before repeating the previous steps. When it is not possible to connect the remaining element of the last 1-cycle with an edge of $w \leq 3$, one must then follow an edge of weight 4. This process is repeated until every node has been visited exactly once. A diagram outlining the usage of this construction for $n = 3$, where weight-1 edges are the black lines and weight-2 ones are blue is shown below:



This construction, given an arbitrary n as the number of digits for each permutation, will use exactly 1 edge of weight $n-1$, 4 edges of weight $n-2$ and, in general,

$$k^2 \cdot (k-1)! \tag{2}$$

edges of weight $(n-k)$, for $1 \leq k \leq n-1$. The proof of this statement, which is similar in style to that outlined in section 1, is shown below.

First of all, it is easy to check that for the most basic, non-trivial case, namely that of $n = 2$, this formula holds. In fact, this construction will produce only 1 edge, which connects the string 12 to 21, with $w = 1$ and $k = 1$, and it is obvious that $1! \cdot 1$ amounts to 1, giving the desired result. From here, it is necessary to notice that, each time n increases by 1, one is essentially changing each weight-w edge to one of weight $w+1$ and connecting each of the length-n permutations ascended from those of length $n-1$ with weight-1 edges. For instance, to construct a similar diagram for $n = 3$ starting from the one for $n = 2$, one can increase the original weight-1 edge to a weight-2 one before replacing each length-2 permutation with 3 length-3 new ones and connecting them all with weight-1 edges. It is easy to see that, each time this method is applied to go from a construction consisting of permutations of length $n-1$ to that with strings of length n, the number of weight-1 edges of the latter graph, with $k = n-1$, will amount to the number of permutations of of length $n-1$ multiplied by $n-1$ edges connecting each of the n new permutations (coming from the original ones of length $n-1$). This is therefore equivalent to $(n-1)! \cdot (n-1)$ or, more simply, $k! \cdot k = k^2 \cdot (k-1)!$, thus confirming the correctness of formula (1) when calculating the number of weight-1 edges in such a graph for any n. Following this line of reasoning, it is also possible to see why this formula applies to any k in range $1 \leq k \leq n-1$. To do this, it is sufficient to notice that, each time n is incremented by 1, all weight-w edges in the original graph will have their weight increased by 1 too and will therefore appear with the same frequency as those of weight $w-1$ in the construction with permutations of length $n-1$. Specifically, this means that any edge of weight $n-k$ in such a graph with permutations of length-n will have been a weight-1 edge in the same style construction with strings of length $n-k$ and therefore will appear with the same frequency as weight-1 edges in the latter graph. Consequently, since we have proved that the frequency of weight-1 edges for any n can be found using formula (1), in order to find that of edges of weight $n-k$, it will be sufficient to apply the same formula with the corresponding value of k since it will return the number of weight-1 edges k graphs before the one in question, proving the correctness of formula (1) for any k in the specified range.

From here, it is possible to create a Python program (for the sake of this essay, Python 2.7 was used together with the *Networkx*[1] and *Matplotlib*[2] modules) which creates a complete graph of all permutations of n symbols and then finds a close-to-optimal path using this construction before removing overlaps and writing out the corresponding superpermutation. You can download the code for both of these files and see an example output on $n = 5$ from here.

Even if this method is highly regular and symmetric, it unfortunately produces optimal superpermuations only up to $n = 5$, after which the only way of finding such strings is probably by using the ”brute force” method described below.

3.2 The Travelling Salesman Problem

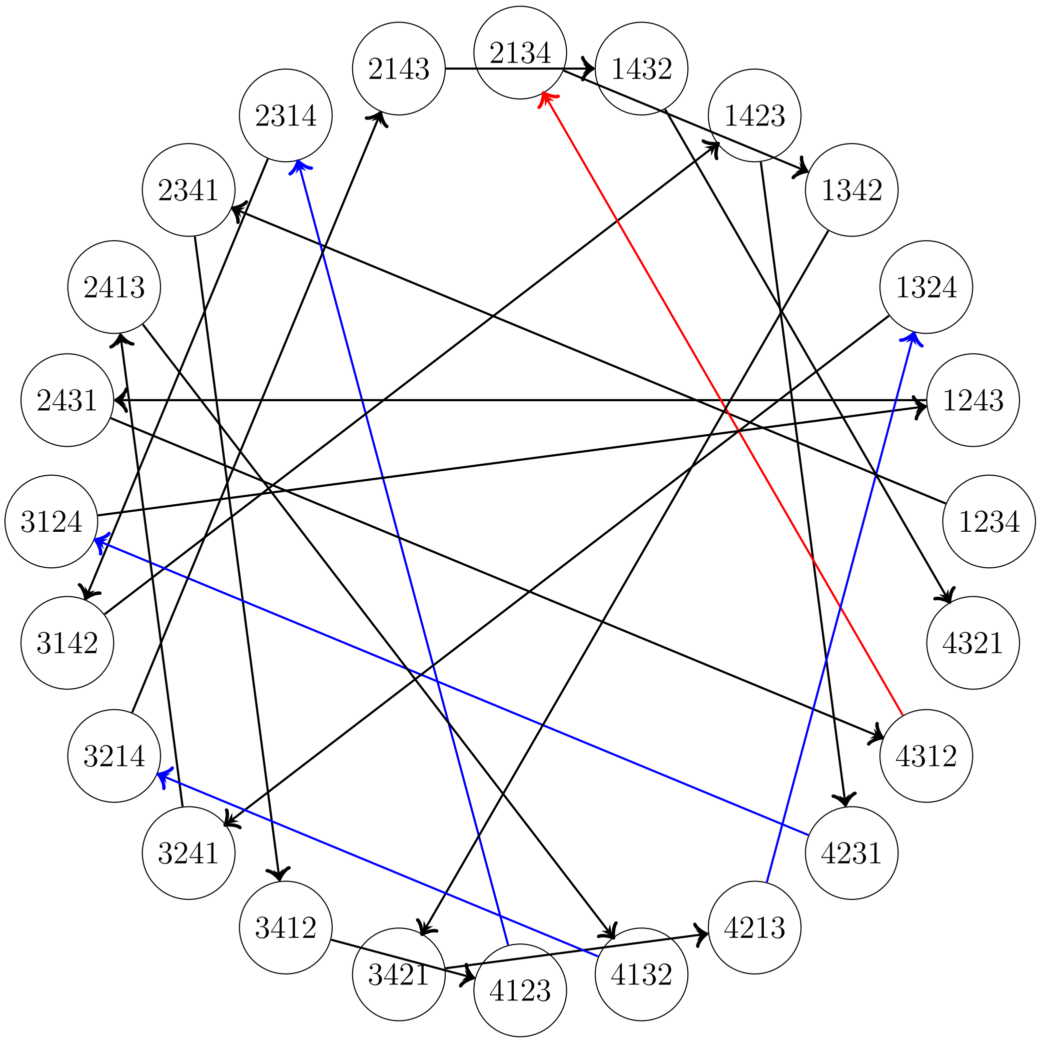
Once one has produced a graph like one described in the section *The graphical approach*, one way to find optimal superpermutations is to use a computer to manually check through all the possible paths while searching for the one which, passing through all nodes exactly once, accumulates less weight (this problem is commonly referred to as the *Travelling Salesman Problem*, or TSP for short, and is still an unsolved problem of NP-complexity). Even if, theoretically, this method is bound to work given any n, in practice, it encounters some complications. Firstly, the most obvious complication is the fact that, for larger and larger values of n, the amount of nodes in the graph will increase drastically (even after the improvements described in section 2), making it impossible for a computer to search through all of them in reasonable time (or even in the span of a few million years). Furthermore, the path we are looking for is asymmetric (doesn't get back to the starting node) and, for most well-know TSP solvers to work, it needs to

[1] “Overview of NetworkX” *Overview of NetworkX - NetworkX 2.2 Documentation*, `https://networkx.github.io/documentation/networkx-2.2/index.html`.

[2] “Matplotlib: Python Plotting.” *Matplotlib*, `https://matplotlib.org/`.

be symmetric. To solve the latter problem, it is sufficient to add weight-0 edges which go from each node to the initial permutation ($123 \ldots n$, in our case).

From here, it is possible to use D-wave[3]'s quantum computing Python library, *Dwave_networkx*, which uses *NetworkX* to solve symmetric TSP for a large enough graph in a reasonable amount of time using the built-in *traveling_salesman* method. Running this function on the complete graph produced with the previously described Python program on $n = 4$ (which it completes in a few seconds) yields the following graph:



[3] "D-Wave Quantum Systems." *Home*, `https://www.dwavesys.com/home`.

with black edges being those of weight 1, blue being those of weight 2 and red those of weight 3. Concatenating all the nodes in order and removing overlaps would in fact produce an optimal, length-33 superpermutation which contains all permutations of 4 digits as substrings of itself. If one is interested in seeing the whole superpermutation, it is written out in the *Addendum* section at the end of the paper.

Using the same method and the randomised TSP heuristic LKH[2] Robin Houston managed to find a length-872 superpermutation for $n = 6$ (one shorter than the one found using the recursive algorithm), which disproves the conjectured lower bound for superpermutations described in section 1.

4 An improved lower bound on the length of the shortest superpermutation

In this section, we will use the same notation used in *the graphical approach* chapter. Specifically, the nodes of our graph are all the $n!$ permutations of length n, referred to as $p_1, p_2, p_3, \ldots, p_n$, and there is a directed edge joining each permutation to every other permutation with weight equal to the least number of symbols one would need to add to p_k such that the result would include p_{k+1} as a substring of itself, denoted $w(p_k, p_{k+1})$.

A superpermutation corresponds to a walk $p_1, p_2, \ldots, p_m$ which visits all of the graph's nodes at least once. If we use the previous definition for the $w(p_k, p_{k+1})$,

[2] *Keld Helsgaun Traveling Salesman heuristic*, `http://akira.ruc.dk/~keld/research/LKH/`

then it follows that

$$w(p_1, p_2, \ldots, p_m) = \sum_{i=1}^{m-1} w(p_i, p_{i+1}). \tag{3}$$

The length of the superpermutation created by the path in question will be equal to the result of equation (3) plus the n symbols of the first permutation p_1 of the path.

4.1 1-cycles

Every node in our graph will have exactly one edge of weight 1 leading out of it. This edge will lead from the permutation p, in the form $p(1)p(2)\ldots p(n)$, to its cyclic rotation $p(2)\ldots p(n)p(1)$ obtained by shifting all the $p(k)$ to the left by one position. For instance, in the graph on $n = 3$, there will a weight-1 edge leading from the permutation 123 to the permutation obtained by moving each of the three digits one position to the left, namely 231. It's easy to see that, if one were to add $p(1)$ to the end of p, the result would contain the second permutation as well. Furthermore, since this shift can be performed exactly n times before one returns to the original permutation, if we follow $n-1$ consecutive weight-1 edges, each going from a rotation of p to the corresponding permutation shifted to the left, we will visit the *1-loop* or the *1-cycle* of p.

Obviously, since the complete graph consists of $n!$ nodes and each node is a part of 1-cycle comprising of n permutations, the number of 1-cycles in the whole construction will amount to $\frac{n!}{n} = (n-1)!$. For now, it is sufficient to note that, in a walk that visits every permutation, we will obviously need to visit each of these cyclic-classes and their members as well.

4.2 2-cycles

Since we have removed all improper edges from our graph, each node will have exactly one weight-2 edge leading out of it. This edge leads from the permutation p to the permutation in the form $p(3) \dots p(n)p(2)p(1)$, or, more simply, it leads to the permutation obtained by shifting all the digits $p(k)$ of the first permutation to the left by 2 positions before swapping the last two. The last swap is done in order to avoid the creation of an improper edge which connects p to another permutation p_1 in the form $p(3) \dots p(n)p(1)p(2)$, passing through the permutation $p(2)p(3) \dots p(n)p_1(n-1)$ in the meantime (remember that, since the edge is of weight 2, the first $n-2$ digits of p_1 are removed when p and p_1 are merged together). By introducing the final swap, we remove the equality between $p(1)$ and $p_1(n-1)$, making the edge proper.

A 2-cycle will be defined as the walk which starts at the node p, follows $n-1$ edges of weight 1 (its 1-cycle), then follows an edge of weight 2 and repeats these steps $n-2$ more times (after we apply the above construction to find edges of weight-2 $n-2$ times, we will just return to the original permutation p). It is important to note that, if a 2-cycle is generated by a permutation p, then it is also generated by the permutations obtained by fixing the last digit of p and shifting each digit in position $p(k)$ to the spot $p(k-1)$. This is because this construction is essentially equivalent to following the 1-cycle of p until the last node, which will be in the form $p(n)p(1) \dots p(n-1)$ before applying the construction described above for the creation of weight-2 edges between nodes. It easy easy to see that this is equivalent to starting the 2-cycle generated by p from the permutation obtained by the weight-2 edge leading out of the last element in the first 1-cycle of p, which will obviously generate the same 2-cycle as p, just shifted by 1.

From here, we see that every 2-loop contains exactly $n(n-1)$ permutations (n elements in each 1-cycle multiplied by $n-1$ elements connected by weight-2 edges) and it is clear from the previous paragraph that there will be $\frac{n!}{n-1} = n(n-2)!$ distinct 2-cycles in the whole graph (every permutation generates a 2-cycle and each 2-cycle is generated by $n-1$ permutations). Finally, because each 2-cycle contains $n(n-1)$ permutations, a walk that visits each permutation will have to enter at least $\frac{n!}{n(n-1)} = (n-2)!$ 2-cycles.

4.3 The proof

Theorem 1. Every superpermutation on n symbols has length at least

$$n! + (n-1)! + (n-2)! + n - 3$$

proof. Given a walk $p_1, \dots, p_m$ in the previously described graph, we will introduce the following definitions:

$s(p_1, \dots, p_m) =$ the total number of permutations visited,

$c(p_1, \dots, p_m) =$ the number of $1-$cycles completed throughout the walk $\mathrm{p_1, \dots, p_{m-1}}$, and

$v(p_1, \dots, p_m) =$ the number of $2-$cycles entered throughout the walk.

Given these defiinitions, the aim of this proof is to show that

$$w(p_1, \dots, p_m) \geq s(p_1, \dots, p_m) + c(p_1, \dots, p_m) + v(p_1, \dots, p_m) - 2 \qquad (4)$$

where the final -2 represents the last weight-2 edge of the final 2-cycle which we are not using in the path (the last edge will always be a weight-1 one, since another weight-2 edge would return back to the start).

Since it is trivial to check the correctness of formula (4) with the basic case of $m = 1$, by induction, if we assume that it holds for all walks of length m, our proof will depend on $w(p_m, p_{m+1})$ in a walk in the form $p_1, \ldots, p_m, p_{m+1}$. We will consider now consider all the possibilities for the weight of this final edge and prove that formula (4) will hold in every case.

1. Firstly, if $w(p_m, p_{m+1}) = 1$, then the two permutations lie in the same 1-cycle, and therefore the value of v does not increase. Furthermore, if we have visited p_{m+1} already, the value of s does not increase. On the other hand, if we have not visited p_{m+1} before, then p_m has not completed its 1-cycle yet, and therefore c does not increase. In either case, both sides of inequality (4) increase by 1 (in the first case, the increase of the right-hand side is given by c while in the second it is given by s), and equation (4) continues to hold.

2. Secondly, if $w(p_m, p_{m+1}) = 2$ then, by the construction described in the previous subsection,

$$p_{m+1} = p_m(3) \ldots p_m(n) p_m(2) p_m(1).$$

   Now, our intent is to show that, if c increases, and therefore p_m has already completed its 1-cycle and

$$c(p_1, \ldots, p_m, p_{m+1}) = c(p_1, \ldots, p_m) + 1$$

then the value of v will not change.

In fact, because p_m has completed its 1-cycle, and therefore had not been previously visited, we must also have visited the permutation we would get to by following an edge of weight 1 from p_m (this permutation, σ would correspond to the initial permutation of the 1-cycle that p_m completes). However, since we hadn't visited p_m before, we must have got to σ through an edge of weight ≥ 2, which implies that we have already entered the 2-cycle generated by σ: Furthermore, like we have shown in the previous subsection, σ and p_m must generate and be part of the same 2-cycle, showing that, in this case, the value of v cannot increase. Finally, since either v or c (but never both of them) will increase and s will increase, equation (4) is verified also in this case.

3. Finally, if $w(p_k, p_{k+1}) \geq 3$, then, since the right-hand side of equation (4) can increase by at most 3 when traversing an edge (one cannot increase the amount of any of the parameters by 2 with just one edge), the inequality continues to hold.

From here, the final step of the proof of Theorem 1 follows easily. In fact, if such a walk $p_1, \ldots, p_m$ visits every permutation, then $s(p_1, \ldots, p_m)$ is obviously $\geq n!$. Consequently, the path must visit each of the $(n-1)!$ 1-cycles at least once so $c(p_1, \ldots, p_m) \geq (n-1)! - 1$ and each of the $(n-2)!$ 2-cycles, from which we derive $v(p_1, \ldots, p_m) \geq (n-2)!$. Therefore,

$$w(p_1, \ldots, p_m) \geq n! + (n-1)! + (n-2)! - 3.$$

Finally, the length of the corresponding superpermutation will be exactly n digits more than the weight of its path (accounting for the length of p_1) and will therefore be at least $n!+(n-1)!+(n-2)!+n-3$ digits long, giving the desired result. $\square$

5 conclusion

When looking at the algorithms and ideas described in this essay, it is clear that research regarding superpermutations has returned some very interesting results, both analytically and theoretically, but it is also evident that many more discoveries are reserved for the future.

Furthermore, when referring back to this paper's original question, it is very likely that, of all the algorithms discussed, the graphical one, being the most versatile and mathematically rigorous, will be what will drive further research on this topic after fully refined and optimised. For example, with a few optimisations to the graph and, specifically, to the total number of edges, it may become possible to prove the length-872 string found by Robin Houston as the optimal superpermutation on $n = 6$ digits in reasonable time using computational methods. The same approach might also help find a more efficient, graphical algorithm which can return optimal superpermutations for larger n values or even help find some more optimised lower bounds for superstrings on any n.

References

[1] Egan, Greg. "Superpermutations." *Superpermutations - Greg Egan,* https://www.gregegan.net/SCIENCE/Superpermutations/Superpermutations.html.

[2] Houston, Robin. *"Tackling the Minimal Superpermutation Problem." Arxiv,* https://arxiv.org/pdf/1408.5108.pdf.

[3] Johnston, Nathaniel. *Non-Uniqueness of Minimal Superpermutations,* https://arxiv.org/pdf/1303.4150.pdf.

[4] *The on-Line Encyclopedia of Integer Sequences,* https://oeis.org/A180632.

[5] Wilshaw, Oliver. *The Minimal Superpermutation Problem: Move by Move,*https://docs.google.com/viewer?a=v&pid=forums&srcid=
MTUwMTUxMjExNDk4NTk5NjY5OTkBMDkzMTY4NzkwMTkyMDgwNjc0MjABNDk2Y19JemVHQUFKATAuMQEBc
authuser=0.

[6] Barnett, Jeffrey A. *Permutation Strings,* https://www.notatt.com/permutations.pdf.

[7] "The Minimal Superpermutation Problem." *Nathaniel Johnston,* 22 Aug. 2014, https://www.njohnston.ca/2013/04/the-minimal-superpermutation-problem/.

[8] Houston, Robin. *A Lower Bound on the Length of the Shortest Superpattern,* oeis.org/A180632/a180632.pdf.

[9] Keld Helsgaun. LKH results for Soler's ATSP instances. http://www.akira.ruc.dk/~keld/research/LKH/Soler_results.html.

Addendum

The length-33 permutation produced in *The Traveling Salesman* section using D-wave's quantum computing TSP algorithm and the NetworkX library is fully shown below:

$$123412314231243121342132413214321$$