COMPUTATIONAL FRAMEWORK FOR THE GENERALIZED BERGE SORTING CONJECTURE

COMPUTATIONAL FRAMEWORK FOR THE GENERALIZED BERGE SORTING CONJECTURE

By

ZHUOYU SUN, B.Eng.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Master of Applied Science

Master of Applied Science (2017) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE: Computational framework for the generalized Berge Sorting conjecture

AUTHOR: Zhuoyu Sun

B.Eng.

McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Antoine Deza

Abstract

In 1966, Claude Berge proposed the following sorting problem. Given a string of n alternating white and black pegs, rearrange the pegs into a string consisting of all white pegs followed immediately by all black pegs (or vice versa) using only moves which take 2 adjacent pegs to 2 vacant adjacent holes. Berge's original question was generalized by considering the same sorting problem using only Berge k-moves, i.e., moves which take k adjacent pegs to k vacant adjacent holes. Let $h(n, k)$ denote the minimum number of Berge k-moves to sort a string of n alternating white and black pegs.The generalized Berge sorting conjecture states that $h(n, k) = \lceil \frac{n}{2} \rceil$ for any k and large enough n. We develop a computational framework to determine $h(n, k)$ for small instances with a focus on the most computationally challenging instances; that is, the determination of $h(k + 2, k)$.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Antoine Deza for his invaluable guidance, generous support and continuous encouragement to my research studies and my life.

My special thanks go to the members of the supervisory and defence committees: Dr. Antoine Deza, Dr. Frantisek Franek, Dr. Reza Samavi.

I appreciate the help and moral support from all my colleagues including the members of Advanced Optimization Laboratory.

Furthermore, I am grateful for the financial aid provided by the department of Computing and Software, McMaster School of Graduate Studies and Dr. Antoine Deza.

Finally, I would like to thank my family and friends from the bottom of my heart for always believing in me, and for their constant love, support and encouragement.

Contents

List of Figures

Chapter 1

Introduction

In a column that appeared in the Revue Française de Recherche Opérationnelle in 1966, entitled *Problèmes plaisans et délectables* in homage to the 17th century work of Bachet [2], Claude Berge [3] proposed the following sorting problem:

> For $n \geq 5$, given a string of n alternating white and black pegs on a one-dimensional board consisting of an unlimited number of empty holes, we are required to rearrange the pegs into a string consisting of $\lceil \frac{n}{2} \rceil$ white pegs followed immediately by $\lfloor \frac{n}{2} \rfloor$ black pegs (or vice versa) using only moves which take 2 adjacent pegs to 2 vacant adjacent holes. Berge noted that the minimum number of moves required is 3 for $n = 5$ and 6, and 4 for $n = 7$. See Figure 1.1 for a sorting of 5 pegs in 3 moves.

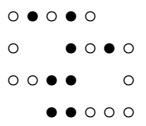

Figure 1.1: Sorting 5 pegs in 3 moves

Let $h(n, k)$ denote the minimum number of Berge k-moves to sort a string of n

alternating white and black pegs. We believe that this problem was looked at within the last 40 years. For example, Shin-ichi Minato [8] found out that $h(n,2) \leq \lceil \frac{n}{2} \rceil$ when he was a high-school student in 1981. However, the first published answer to Berge's question might have been given by Avis and Deza [1] who showed that $h(n,2) = \lceil \frac{n}{2} \rceil$ for $n \geq 5$. The Berge sorting question appeared in the *12th Prolog Programming Contest* [4] held in Seattle in 2006. In the statement of the problem, it is noted that this result is surprising given that initially half of the white pegs and half of the black pegs are incorrectly positioned. The following generalization displays an equally surprising pattern. Consider the same sorting problem using only *Berge k-moves*, i.e., moves which take k adjacent pegs to k vacant adjacent holes. Avis and Deza [1] proved that $h(n,k) \geq \lceil \frac{n}{2} \rceil$. After generating minimal solutions for a large number of k and n, it turned out that $h(n,k) = \lceil \frac{n}{2} \rceil$ except for the few first small values of n, Deza and Hua [5] conjectured that, for n large enough, the minimum number of Berge k-moves to sort the alternating n-string is independent of k and $h(n,k) = \lceil \frac{n}{2} \rceil$. As the case $k = 1$ is trivial and the case $k = 2$ corresponds to the original Berge's question and proven by Avis and Deza [1]. As for the case $k = 3$, $h(n,3) = \lceil \frac{n}{2} \rceil$ was proven for $n \geq 5$, and $n \not\equiv 0 \pmod 4$ by Deza and Hua [5] and the case was closed for $n \geq 20$ and $n \equiv 0 \pmod 4$ by Deza and Xie [6].

This thesis entails our investigation of the most computationally challenging instances; that is, the determination of $h(k+2,k)$. We developed a computational framework to determine $h(k+2,k)$ for various values of k. Deza and Hua [5] found the $h(k+2,k)$ for up to $k = 14$, our goal is to find values for $k > 14$ and to extrapolate the behaviour of $h(k+2,k)$ to large k.

1.1 Thesis Outline

In order to find the determination of $h(k+2,k)$, we needed solutions for more k values to draw any conclusion. We build a computational frame to find $h(k+2,k)$ for different k values and explored different methods to optimize the code and help generate more data points to drew conclusion from. In chapter 2, we describe the computational frame we have build to find $h(k+2,k)$ and different approaches we took and conditions we set to help minimize the computational time and narrow down the search space. In chapter 3, we discuss some of the results we get from our computational approach and discuss the interpretation of it and future work to be done in chapter 4.

1.2 Notations

We follow and adapt the notation used in [1, 3, 5]. The starting game board consists of n alternating white and black pegs sitting in the positions 1 through n. A single Berge k-move will be denoted as $\{\ j\ i\ \}$, in which case, the pegs in the positions $i, i+1, \ldots, i+k-1$ are moved to the vacant holes $j, j+1, \ldots, j+k-1$. Successive moves are concatenated as $\{\ j\ i\ \} \cup \{\ l\ k\ \}$, which means perform $\{\ j\ i\ \}$ followed by $\{\ l\ k\ \}$. Often, a move fills an empty hole created as an effect of the previous move, and the resulting notation $\{\ j\ k\ \} \cup \{\ k\ i\ \}$ is abbreviated as $\{\ j\ k\ i\ \}$. This can be extended to more than two such moves as well. Let $h(n,k)$ denote the minimum number of required k-moves, i.e., the length of a shortest solution, and $O_{n,k}$ denote an optimal solution for n pegs, i.e., a solution using $h(n,k)$ Berge k-moves. For example, we have $h(5,2) = 3$ and the optimal solution given in Figure 1.1 is $O_{5,2} = \{\ 6\ 2\ 5\ 1\ \}$. Up to symmetry, we can assume that the first

move is to the right. Let $\mathcal{O}_{n,k}$ be the set of all optimal solutions starting with a move to the right. For example, there are 7 such optimal solutions in 10 Berge 3-moves to sort the alternating 20-string; that is, we have $h(20,3) = 10$ and $\mathcal{O}_{20,3} =$ { { 21 2 7 12 17 } ∪ { 24 13 22 6 1 } ∪ { 17 8 24 }, { 21 2 13 6 17 } ∪ { 24 7 22 12 1 } ∪ { 17 8 24 }, { 21 2 13 8 17 } ∪ { 24 6 22 12 1 } ∪ { 17 7 24 }, { 21 6 13 2 17 } ∪ { 24 7 22 12 1 } ∪ { 17 8 24 }, { 21 8 13 2 17 } ∪ { 24 6 22 12 1 } ∪ { 17 7 24 }, { 21 12 7 2 17 } ∪ { 24 13 22 6 1 } ∪ { 17 8 24 }, { 21 16 3 10 } ∪ { 24 17 22 5 1 } ∪ { 10 15 6 24 } }. See Figure 1.2 for an illustration of the optimal solution $O_{20,3} =$ { 21 2 7 12 17 } ∪ { 24 13 22 6 1 } ∪ { 17 8 24 }.

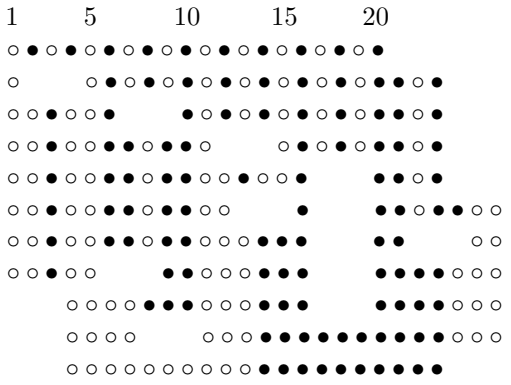

Figure 1.2: Sorting the alternating 20-string in 10 Berge 3-moves

Chapter 2

Computational Approach

In this chapter, we discuss our approach to solving the problem. We felt we needed more data so we could analyze it and try to find a solution to the problem. In order to get more data points, we needed to write a program that takes $k+2$, the number of pegs, as an input and sorts the pegs then returns $h(k+2, k)$ and the actual moves made to sort the pegs. Since every possible move will generate a new path in which the pegs can be solved, this will ultimately become a huge growing tree with each node being one configurations of the pegs. We adapted Depth-First Search to traverse through this tree to find the solution. We will start by discussing some of the restrictions and conditions we placed on our computational frame to make it run faster and how we used it to help find the solutions quicker.

2.1 Search Space

The search space is essentially a tree. The pegs can be in different configurations throughout the program run, they could be in different positions and have different sizes of gap between different pegs. Each of these configurations is a node and at each

node, every possible move we could make to move the pegs will generate a new path and a new node. The tree will grow exponentially. The root is unsorted alternating black and white pegs one after another as shown in Figure 2.1 and that is where we start and the tree starts. Sometimes, there could be quite a few children to generate from a node depending on the choices of available k pegs to pick and each of those might have a few spots to move into but we can discard many of them by different techniques explained in later sections. We keep track of the disorder at each node and check to see if the pegs are sorted. If they are sorted at a node, that node will not generate any new nodes and we will return necessary information for the sorted node and explore other nodes. For this tree traversal, we decided to use Depth-First Search to explore the nodes because it makes coding a lot easier and it is faster to get the first solution.

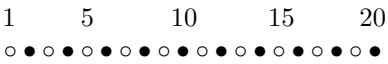

Figure 2.1: Alternating 20-string

2.2 Possible Forms of Peg Placement

At each node, there will be two or more sets of consecutive k pegs that we can choose to move around, each set of pegs can be moved to create a new node with new configuration but we need to keep the total gap between pegs within $\lceil \frac{k}{2} \rceil * k$. The pegs can be in the following 3 forms.

2.2.1 Type 1

The first form of peg placement is shown in Figure 2.2. There are no gaps between the pegs. The restriction here is that we don't make a move that will create a gap

that's greater than $\lceil \frac{k}{2} \rceil * k$. We could take pegs from position 0 to $k-1$, 1 to k or 2 to $k+1$ and move them around to make new nodes.

Figure 2.2: $k+2$ pegs, Type1

2.2.2 Type 2

The second form is shown in Figure 2.3. It has a gap of size $x-1$ between the first and second pegs. We have a choice to take the pegs from x to $x+k-1$ or $x+1$ to $x+k$. We would have different moves to make for each set.

Figure 2.3: $k+2$ pegs, type2

2.2.3 Type 3

The third type of placement of pegs if shown in Figure 2.4. In this form, there is one gap between the pegs at k and $x+k$ with $x-1$ being the gap size. We can take either the pegs from 0 to $k-1$ or 1 to k. This is very similar to Type 2 2.2.2, we just have $k+1$ pegs on the left hand side. All the moves we can make in this case will be symmetrical to the moves of 2.2.2.

Figure 2.4: $k+2$ pegs, type3

2.3 Actual Implementation

For the actual implementation, I created 3 functions, one for each type, to call upon recursively. As we can see from 2.2, we have 3 possible forms of the peg configurations. Since moves available for each type are essentially the same, we could have one function for each type. We start off with root, which is Type 1, make moves and call appropriate functions with information needed. At any node, we make moves and we know the types of peg configurations as a result of these moves so we can call the appropriate functions and pass them the necessary information as well. We do this recursively and thus creating the tree and traversing through it using Depth-First Search. In this section, we discuss the actual implementation in more detail.

How to store the string

Initially, I used HashMap to store this information. I would use key to store its current position and value to denote the color of the peg with 1 being a white peg and 0 being a black peg. Every time we make a move, we simply update their keys to its current position. Since each type will have at least $k + 1$ pegs that are consecutive we only need to know the position of the isolated peg if it's not Type 1 and that of the first peg of the consecutive $k+1$ pile. Also, we use ArrayList to keep track of all the moves made to get to the current node. However, this method would require a for loop to update the peg positions so that was very inefficient and also redundant to keep track of the peg positions. What we did was to use the Deque data type, which is a double queue where we can take out the first element or last in constant time. This takes full advantage of the fact that the order of at least k pegs will not change at each move. So, in fact, we only have to move one peg from the $k + 1$ pile which is stored in a Deque and add it to the other side and update the gap size.

How to count the disorder

It is important to count the *disorder*, i.e., the number of pegs whose right neighbour is not a peg of the same colour because we want to keep track of the disorders so we know when to stop. As mentioned in How to store the string 2.3, I didn't go with array to store the peg position because the pegs will be moved around and going by their concurrent positions would require sorting the array each time as the array values denote their position, so HashMap was used instead. Since at least k pegs will remain the same in their orders at each move, we only really needed to check a few pegs who have lost their right neighbour to update their disorder. After we changed our approach to string storage, the concept still remains the same and we only need to check a few pegs to determine the disorder as most still don't change.

Termination

It is also important that the program terminate. At each execution, I give a limit to the number of moves the program can make. Thus, when a node reaches the limit given, it will stop making more moves and calling other functions and the tree stop growing from there, also if the pegs are confirmed to be sorted by checking disorder then the node will stop making new moves and calling other functions too and it will print out the results. Eventually all children either reach the limit or get sorted, the program terminates then. Otherwise the program will never end.

2.4 Redundant Nodes

There are cases that we should always consider so that the program does not process redundant nodes and sub trees that stem from them so the execution time can be shortened. There are there such cases, relabelling, translation and symmetry, to look

at and we discuss them in this section.

2.4.1 Translation

The first case is to reach a previously achieved node because this will create the exact same sub tree as the one created by the previously achieved node and with more moves as this new node would have the same configuration but with more pegs to get there. Thus, we can definitely delete this repeated node. For example, in Figure 2.5, both have the exact same configuration but different number of moves to get where they are. The bottom one has every peg in the same order but shifted by a certain amount so this is essentially the same node and will produce the same sub tree as well just with everything shifted by the same amount so if this one has more moves it needs to be deleted. The most trivial and common case of this is going back to the same state after one move. Here, you make a move then on the next turn, you make a move to place these pegs back where they come from.

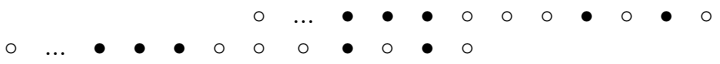

Figure 2.5: Pruning by translation

2.4.2 Relabelling

Second case is the relabelling of white and black pegs as they are interchangeable. If we flip the white pegs and black pegs, they are still the same thing and the moves to be made to sort them will be exactly the same just with white and black pegs swapped. White pegs followed by black pegs are the same as black pegs followed by white pegs. As shown in Figure 2.6, those two are the same and the would produce

the same results, just black and white swapped. These can be considered repeated nodes as well so we would delete the one with more moves.

Figure 2.6: Pruning by relabelling

2.4.3 Symmetry

The final case is when 2 sets of pegs are symmetrical about the half point as shown in Figure 2.7. These two nodes are symmetric in that everything is a mirror image of one other about the middle point and the moves to be made from these will be symmetric as well so essentially the same thing. Thus, these could be considered redundant as well.

Figure 2.7: Pruning by Symmetry

It is important to keep track of all the configurations so we know when these three cases occur and we can eliminate the redundant nodes. However, identifying all these with different positions with relabelling and translation can be troublesome, so we will reset the positions at each node so the position of the first peg will always be relabelled 0 and the rest follow, this takes care of translation as all pegs will occupy positions 0 to $k+1$ now and we will keep track of the type and gap size as well. This way, we can detect all repeated nodes in translation. To deal with relabelling(white and black swapped), we could simply set the first peg at each node to be a white peg then reset the rest accordingly. If the first peg is white, we do not change anything in

terms of the peg colour. As for symmetry, we simply create a reversed version of the pegs at each node and with the information of gap size and type while applying the relabelling technique, we can determine if this reversed configuration has already been accomplished before. Another trick is using the fact that symmetry only occurs for type 1 and type 2 so we could simply take out type 2 so symmetry would never happen to begin with because they are the reverse image of each other for every configuration of type 1 there's a symmetry case of it in type2. Using these techniques, we can track down all the redundant nodes and delete them if need be. However, in order to show the solutions, we need to convert the positions back at the end. This approach gave us a huge speed up.

2.5 Branch And Bound Like Pruning

The concept of this type of pruning is that once we have an upper bound to the number of moves to make, we can use that as a limit to eliminate all nodes that will require more moves than the upper bound to sort the pegs, thus pruning a huge chunk of the tree. Let $D_{k+2,k}(i)$ denote the *disorder* after the i-th Berge k-move. Initially, $D_{k+2,k}(0) = k + 2$, and when the pegs are sorted $D_{k+2,k}(i) = 2$. We will try to find solutions with number of moves strictly less than the upper bound. $i + \lceil \frac{D_{k+2,k}(i)-2}{2} \rceil \geq$ *upperbound* will be added as the stopping criteria, nodes that satisfy this criteria will stop further executing from there. $\lceil \frac{D_{k+2,k}(i)-2}{2} \rceil$ simply tells us how many moves are needed at least to sort the remaining pegs as only optimal moves that decrease the disorder by 2 can sort the remaining pegs in $\lceil \frac{D_{k+2,k}(i)-2}{2} \rceil$ moves. This will be used heavily to prevent unnecessary computation.

2.6 Heuristic Bound On The Range

Initially, we decided to limit gap between pegs to $2k$ at most since having a gap greater than $2k$ is sub-optimal. With this restriction, the length from the first peg to the last peg would be $3k + 2$ since we have $k + 2$ elements and the largest gap is $2k$. This length can range from $k + 2$, when there is no gap in between, to $3k + 2$. This significantly limited the range of moves we could make and therefore reducing a significant amount of computing due to the reduced tree size. However, this did not give us the minimal solutions and judging from the previous solutions and it seemed like increasing the gap limit to $\lceil \frac{k}{2} \rceil * k$ was a better idea so we went with that instead.

2.7 Heuristic approach: Stage1

With all the ground work done, we can finally run the program and use it to find the solution to our problem. With the brute force approach, the computation was still too much so we needed to take a heuristic approach to further reduce the computation time. We took a look at the past solutions and tried to find the patterns that we could use to narrow down the search. We have two stages to this process and Stage 1 will be discussed in detail in this section. The purpose of this stage is to get the best estimate for how many moves are needed to sort n pegs using different techniques to shorten the computation time. We run a few rounds with each round having more relaxed constraints from previous rounds in a hope to improve the estimate after each round. After examining the solutions computed by Deza and Hua [5], we found out that, for the most part, the first two moves reduce the disorder by one and a lot of them have their disorder decrease by 2 after 3 moves. It is optimal not to to have any bad moves that increase the disorder but examining the past solutions revealed that

a lot of them do have one or two bad moves that increase the disorder by 1. Further investigation of the past solutions has shown that there are no bad moves that occur within the first $\lfloor \frac{k}{2} \rfloor$ moves and the first bad move occurs by the kth move. Some of the solutions do have the second bad move after the first one and disorder decreasing by 2 after 5 moves instead of 3. We will use all of these rules to prune for round 1 to get the initial estimate. Our goal is to have our code get as close to the generalized algorithm as possible so that we do not need the brute force algorithm to find the minimum solution. Thus, for round 1 of this stage, we want to create a fast pruning heuristic such that with all the rules, this formulation will return the optimal solution for all values up until $k = 14$, which we could confirm with past solutions. These will be used to find a good estimate for $k > 14$ and the estimate we get in round 1 will be used in round 2 as an upper bound to prune more, this technique is discussed in 2.6.

1. Allow no bad moves in the first $\lfloor \frac{k}{2} \rfloor$ moves one bad move by kth move and another one after that, disorder to decrease by 1 in the first 2 moves and 2 in first 5 moves

2. Everything from round 1, and prune using the estimate from round 1 as upper bound

Redundant nodes, as discussed in section 2.4, will be checked against all nodes throughout the execution and they will be eliminated to shorten the execution time.

2.8 Heuristic approach: Stage2

At Stage 2, the program input are $k + 2$, the number of pegs and the estimate from Stage 1. Our job here is the verify that the estimate we acquire from Stage 1 is indeed

the best we can do or the one last try to improve the estimate as at this stage, all bets are off and we allow all kinds of move to go but redundant nodes will be eliminated as described in section 2.4. The estimate from stage 1 comes in handy because this could act as a limit and we use it to prune all nodes that would exceed the estimate or equal. $i + \lceil \frac{D_{k+2,k}(i)-2}{2} \rceil \geq estimate$ will be used as the stopping criteria. If no result comes out of this, it verifies the estimate from Stage 1 as the best solution. If estimate does get improved here, that is the best solution as we look at all the possibilities at this stage. We do not impose the disorder to decrease in the first few moves or limit the gap to $\lceil \frac{k}{2} \rceil * k$ here. Result from here can verify whether the heuristic approach we took in stage 1 is correct.

Chapter 3

Results and Analysis

In this chapter, we will look at the results we get from Stage 1 and Stage 2 as explained in chapter 2. We will talk about what they represent and analyze them here.

3.1 Stage 1 Result

Stage 1 Result(moves)		
Input k	Round1	Round2
$k = 3$	3	3
$k = 4$	6	6
$k = 5$	6	6
$k = 6$	7	7
$k = 7$	8	8
$k = 8$	11	11
$k = 9$	12	12
$k = 10$	16	16
$k = 11$	18	18
$k = 12$	22	22
$k = 13$	22	22
$k = 14$	29	29
$k = 15$	30	30

As we can see, Round 1 gives the minimum solution for all values $k \leq 14$. For example, for $k = 15$, round 1 gave 30 and round 2 did not improve that number.

3.2 Stage 2 Result

At Stage 2, we take string size $k + 2$ and the estimate from Stage 1 as the program inputs and we verify that those estimates are the best we can do. This is where any

kind of moves are allowed except the ones that are prohibited by the elementary conditions. The results are as follows:

Stage 2 Result(moves)	
Input k	Round1
$k = 3$	3
$k = 4$	6
$k = 5$	6
$k = 6$	7
$k = 7$	8
$k = 8$	11
$k = 9$	12
$k = 10$	16
$k = 11$	18
$k = 12$	22
$k = 13$	22
$k = 14$	29
$k = 15$	30

Chapter 4

Conclusion

Even though we were not able to find the determination of $h(k+2, k)$ due to increased execution time when k gets larger because more moves are required and that means longer path and more nodes to compute and since the tree grows exponentially, there really are a lot of nodes to compute. Unfortunately for me, I did not have enough data points to evaluate from and from the ones I already have, it was not possible to draw any conclusion on the general case for $h(k+2, k)$. However, we do have a working code and we made it a lot faster with different pruning techniques and with more time and data to work with, it should be easier to find the determination of $h(k+2, k)$. Also, it is interesting to notice that moves that are considered sub optimal do indeed help sort the pegs quicker in most cases.

Bibliography

[1] David Avis and Antoine Deza: Un des "problèmes plaisans et délectables" de Claude Berge. *Discrete Mathematics* 306 (2006) 2299–2302.

[2] Claude Gaspard Bachet de Méziriac, Problèmes plaisans et délectables qui se font par les nombres. Partie recueillie de divers autheurs, et inventez de nouveau avec leur démonstration, par Claude Gaspard Bachet, Sr de Méziriac. Très utile pour toutes sortes de personnes curieuses, qui se servent d'arithmétique, A Lyon, chez Pierre Rigaud (1612), seconde édition (1624).

[3] Claude Berge: Problèmes plaisans et délectables, le problème des jetons noirs et des jetons blancs. *Revue Française de Recherche Opérationnelle* 41 (1966) 388.

[4] Bart Demoen: *12th Prolog Programming Contest, Seattle 2006* HP.
`http://www.cs.kuleuven.be/~bmd/PrologProgrammingContests/Contest2006.html`

[5] Antoine Deza and William Hua: Berge sorting. *Pacific Journal of Optimization* 3-1 (2007) 27–35.

[6] Antoine Deza and Feng Xie: On the Generalized Berge Sorting Conjecture. *Journal of Discrete Algorithms* 8 (2010) 1–7.

[7] Antoine Deza: *Berge Sorting* HP.
`http://www.cas.mcmaster.ca/~deza/berge_sorting.html`

[8] Kouichi Taji: *personal communication.*