Advanced Programming and Data Structure
Course 2022-2023

Project 2 of the first semestre – Combinatorial optimization

CatTheHobie - The Sequel

G16

Students	Login	Name
	adrianjorge.sanchez	Adrián Sánchez López

Date	19/03/2023

Table of Contents

Programming language choice

I could have chosen between Java and C, since those 2 are the languages that I know the most. And I have chosen java. I know C is a compiled language, so in contrast with Java it can be faster as it doesn't need to be interpreted and then translated to machine code after compiling it. However, as in this project we are focusing on analyzing different algorithms and its efficiency. So it doesn't matter if I use an interpreted language or compiled one, as the algorithm's execution time will be the same. In addition, with C you can have more control over memory usage. But, as before, it doesn't affect the algorithm's cost.

In contrast, with Java I have prebuilt methods and data structures which I can use without having to implement them. Which saves me a lot of time. As it is easier to work with a list of objects or priority queues which will be used during this project. As I have this data structures already implemented with methods that are intuitive to easily control them.

Algorithms:

High Speed Sailing

The objective of the algorithms implemented in this problem is to assign each different sailors the program has to a boat until that boat reaches the maximum capacity. Moreover, they have to be assigned in a way the total sum of speeds of all boats is the maximum possible. And to calculate the speed of a boat based on the sailors it has are the following ones:

$$speed_{real} = speed_{maximum} \cdot \prod_{i} impact_{sailor}$$

$$impact_{sailor} = \frac{impact_{weight} + impact_{skill}}{2}$$

$$impact_{weight} = \frac{100 - \ weight_{participant}}{weight_{boat}} \qquad impact_{skill} = \frac{ability[type] + win_{rate}}{2}$$

So the real speed of a boat is calculated by multiplying its speed by the impact sailor of each of the sailors a boat has. And the impact sailor is the average of two other values: impact weight which has to do with the weight of a sailor and the boats he is in, and impact skill which is another average between the skill a sailor has with the type of boat he is in and its overall win rate.

Having that in mind now a configuration must be designed. And the best way of representing the data for this problem in my opinion, would be to define an array of integers having as length the same value as the number of sailors. So that each position of the array represents the sailor stored in the same position. And inside each position an integer value ranging from 0 to the number of boats. In order to assign one value or another depending on the boat this sailor is in, and assign 0 if it is not assigned to any boat. As there can be cases where there are more sailors than what the boats can have. Therefore, if for example i have the following configuration: [1, 2, 0], that would mean that from the list of sailors, the one in position 0 is assigned to the boat stored in the position 1 in the array of boats, the sailor in position 1 is assigned to the boat in position 2, and the sailor stored in last position is stored in the boat stored in position 0. Which is a boat without data, just to have there all the sailors that are not assigned to any boat.

Backtracking

After having the config defined, to implement the backtracking algorithm. I started implementing a brute force approach. To later on add different strategies to transform that brute force into backtracking.

The implementation of brute force is pretty easy. It is just a while-loop that loops until there are no more options in the current level. Which in this case will be while it is lower than the number of boats the system has. And inside this while-loop there are 2 possible cases: if the config is at the last level or not. If it is at the last level it will check if it is feasible. A solution is feasible if it meets the requirement specified in the statement. In this case, all boats have the maximum capacity of sailors inside. This functionality is implemented in my code through a method called correct, which checks that no boat has more sailors than possible. To do so, marking is applied. Apart from the capacity each boat has currCapacity, which stores the maximum capacity as well. But each time a sailor is assigned to a boat 1 is subtracted from currCapacity. And at the end of the loop, before increasing the value of the level the function is by 1, one is added to the currCapacity , as the configuration where a sailor x was assigned to boat 2 is now gonna be assigned to boat 3. So we have recovered the capacity boat 2 had before adding the sailor x. Because now sailor x is gonna be assigned to the 3 instead of the 2. And at the start of the loop after sailor x was added to boat 3. The marking where 1 is subtracted from the currCpacity of the boat is gonna happen again, but for the new boat. Therefore the function correct, apart from checking that no more sailors than allowed are assigned to a boat. It also checks that the maximum capacity minus the current capacity of boat 0(where the sailors are assigned to say that this sailor won't be assigned to any boat) is not bigger than the number of sailors minus the sum of the total capacity of all boats. As if it is bigger, it will mean that there are more sailors unassigned than the amount of sailors that can be unassigned. Meaning that one or more boats will not be full.

Moreover, marking will also be used to keep track of the current speed of each boat. So apart from the speed a boat has it will also have a real speed. Which will be initialized at the speed of the boat. But, as with the capacity, each time a sailor is assigned to a boat. The impact of the sailor will be calculated and multiplied to the current real speed of the boat. And before changing the assignment of that sailor to another boat, the real speed of that boat will be divided by the impact of the sailor(unmarked). Therefore, with that marking applied, PBCBS can be applied after checking that one solution is correct. If the sum of the real speeds of each boat is bigger than the best solution, this solution configuration and solution will be stored as best configuration and best solution respectively.

On the other hand, if it is not at the last level it will check that the solution is correct. As if there are more sailors unassigned than the amount it could be or if there is a boat with more sailors than capacity. No matter what it does in the next levels, the solution will be wrong. And after checking that it will also be checked that the current sum of speeds is bigger than the best solution. Because as you keep assigning sailors the sum of speeds can only keep decreasing. So if the sum of speeds is already equal or smaller than the best solution. You do not want to keep exploring its possibilities, you want to prune them. Therefore, it will only call the function recursively, increasing the level + 1 when the config is correct and the sum of speeds is bigger than the best solution.

Branch and bound

After backtracking, as it was taking some time to get the solution. I decided that the next approach will be branch and bound. Because branch and bound and backtracking are essentially the same. But backtracking explores the solutions of each space in the same order. While branch and bound explore directly an optimal solution. So it can prune most of the solutions after finding 1 solution. In contrast, backtracking will depend on luck. As it would depend on where the optimals solutions are. If they are close to where backtracking starts finding solutions, then it will prune a lot. But if they are found at the end, then it wont prune that much. So branch and bound is always faster than backtracking in terms of finding a solution(except when backtracking finds an optimal solution at the start). But, to do so, branch and bound wastes much more memory and exploring a configuration takes longer than backtracking. So it must be used when you don’t have problems with memory storage and when you think the extra pruning will compensate for the extra time it takes in exploring configurations.

To develop the branch and bound algorithm, a new class BnBConfig has been created. Which stores a config(which is the same as the config used before), an array of integers storing the currCapacity of each boat, an array of floats storing the current real speed of each boat. And an integer storing the level this configuration is in.

The algorithm will start generating one configuration by default and adding it to a priority queue. Then it will enter in a while-loop until there are no more configurations in the priority queue.

To be able to explore the solution space in a way that it finds an optimal solution as fast as possible it needs to compare the different configurations before exploring one. But the priority queue only has 1 configuration. So the different variations of this configuration must be generated. To do so, it first needs to generate those configurations without exploring them. Which is done by using the expand() method, which is called through a configuration and generates all the variants of that config and adds them to a list that will be returned. The way of generating the variants is by a for-loop that lasts while it is lower than the number of boats. And in each iteration it clones the config from which the method was called, changes the value of the level in which the config is to i(number of the iteration we are in), marks currCapacity of the boat by subtracting 1 to the position i of that array and marks speed of the boat by multiplying the speed of position i in the array of speeds by the impact of that sailor. Then adds 1 to the level and adds that config to the list. So if a config [0,0,0] calls expand and the level of this config is 1. Expand will generate the following configs: [0,0,0], [0,1,0], [0,2,0], [0,3,0]... while that value is smaller than the number of boats.

Also, it is important to know that as all configurations of a level are generated and then for each configuration generated for that level if it is ok the configuration for the next level will be generated and so on. Therefore, once the value of a level of a configuration has been set, no modification will be done to that value. So it only needs to be marked once it adds a value to a level. But it does not need to be unmarked as no modification will be done.

Then, after generating all those configs in a list, a for-loop will check each of those configs generated. If the config is full(meaning that its level is the last), it will check if the sum of the speeds is bigger than the best solution and if all boats are full. If both conditions are met, the current config and its solution will be stored as the best config and solution.

In case those configs generated are not at the last level, the program will check if its sum of speeds is bigger than the best solution. If it is, it will add the config to the priority queue.

In addition, the compare to method of the priority queue class has been overridden to make it so that it compares how promising a config is over another. To do so the method getQuality has been created. This method, given a config, calculates the impact sailor average from the start of the config until the level from bigger to smaller. That way it will sort the configs being the one with the biggest average(meaning that is the one that has all the sailors assigned in their appropriate boat), the first in the queue and the one with the worst the last.

Therefore, going back to how the algorithm works. While the queue is not empty, it will take out of the queue the most promising config check. So it explores the solutions in order(from most promising to least). Making it easier to find an optimal solution as early as possible. Note that the evaluation of how promising a config is it’s not exact. As it will have to take into account far more parameters and calculate a lot more different things. But overall prioritizes the ones that will be better. Even though not in the same exact order.

Full fleet

The objective of the algorithms implemented in this problem is to organize the regatta. To do so: there must be at least one boat available of each type. But, as boats are stored in different centers, it might be difficult for a center to have at least 1 boat of each type. So, to organize the regatta the minimum number of centers that together satisfy this condition must be found. Moreover, it is possible that no solution is found. As one boat type might not be in any center.

In addition, the config used for both algorithms was an array of integers representing the different centers(which will be further explained below). Therefore, I created a new class called Center. Which stores the name of the center, the number of different types it has and the different types stored. As it will be very difficult and troublesome to create the algorithms without one. Not to talk about the increase in the cost it will cause. Instead of having each center order in a list of center objects, with their types of boat and the number of different types. I would have had to look for the type and the center of each boat in the system. As well as checking if that center and type already exist or if they are new ones.

Backtracking

The first approach I had facing this problem was backtracking. But before being able to implement it. I need a way of representing the data(my config). After thinking about a good way of representing the data and considering different opinions. I decided to represent the data as an array with the same length as the number of centers that exist in the system. All the information about the centers is stored in a list of center objects(previously explained before). Each position representing the center in the same position in the list of centers. And each position can take a value between 0 and 1. 0 representing that this center is not used and 1 representing that this center is used. So having a config like this: [1, 1, 0, 0]. Means that the centers in the list stored in the positions 0 and 1 are used. And the centers stored in the last two positions are not used.

After having the config defined, to implement the backtracking algorithm. I started implementing a brute force approach. To later on add different strategies to transform that brute force into backtracking.

The implementation of brute force is pretty easy. It is just a while-loop that loops until there are no more options in the current level. In this case while it is lower or equal to 1, as each center will have only two options: 0 unused, 1 used. And inside this while-loop there are 2 possible cases: if the config is at the last level or not. If it is at the last level it will check if it is feasible. A solution is feasible if it meets the requirement specified in the statement. In this case, between all the centers used, all the types of boat appear at least once. This functionality is implemented in my code through a method called isCorrect, which gets all the types stored in all the centers used in an array that has 1 position for each type. And adds 1 to the position of the array that represents a type each time that this type is found. After, it goes through that array and looks that each position is bigger than 0.

On the other hand, if it is not at the last level it will call the function recursively increasing the level + 1. Finally, even if it is at the last level or not, it increases the value of the current level plus 1.

Now, once all the correct solutions are generated. Pruning based on the current best solution will be implemented. To do so, three new variables have been created: best solution which stores the best solution found, solution which stores the solution of the current configuration being analyzed and bestConfig which stores the best configuration found. As I am looking for the solution with the smallest number of centers, the variable best solution is initialized with the maximum possible value. So once found the first solution no matter what it will be better than the maximum possible value. Finally, to implement the pruning, after checking that a possible solution is correct, I will also check that the solution of the current config is smaller than the best solution. If that is the case, the current config will be stored as the best config and the current solution will be stored as the best solution. Moreover, I will also check before calling the function recursively that the current solution is smaller than the best solution. So in case that the current number of centers is equal or bigger to the best solution. The function will not be called recursively. As after calling a function recursively, the number of centers will only get bigger. That way I will stop generating all the children of those configs whose solutions are already worse than the best solution.

Lastly, in case there is no possible solution(worst case). The program will go through all the configs that can be generated. So once it finishes, to know if a solution has been found or not. The best solution is checked: if the best solution is different from the maximum possible value, then the solution has been found. If the best solution is the maximum possible value, then there is no solution to the given data.

Marking could have been added also to keep track inside the loop of the centers being used. But as the algorithm already finds the solution almost instantly as will be seen in the case analysis. I decided to leave it that way.

Greedy

Apart from backtracking this problem has also been solved with greedy. In contrast with backtracking, greedy is a non-exhaustive algorithm. Which means that it does not necessarily find the best possible solution. However, it does find a solution close to the best solution in most of the cases and in polynomial time, ideally in linear. To sum up, it greatly reduces the time the algorithm takes at the cost of precision. So, even though this algorithm does not find the best solution, it is an interesting approach to take if you do not strictly need to find the best solution or if you want to prioritize computational cost at the expense of quality in the solution. Moreover, this approach can be used together with PBCBS. By running greedy first and then initializing the PBCBS with the solution obtained with greedy. Which will be a pretty good solution. Pruning most of the branches.

To implement this algorithm I decided to continue using the same way of representing data as in the backtracking approach. To be able to tell a possible solution quickly, the config needed to be arranged in a way that it uses the centers with the highest possibility of having all types. The best solution possible to this problem is that it finds a center with all the types. Therefore, the config will be arranged from bigger to smaller. So it starts using the center with more types in the system, and if that center does not have all types it will also use the second one, and so on… That way even if the first center has not all the types, it will have most of them. And by taking the next one that has more types the possibility of having between those types the few types we need is much bigger than if a center with 1 type is used. As the possibility of finding the type we need there would be of types needed/ 6.

To sort all the centers in descending order I used the prebuilt method sort(). I would have used one of the sorting methods implemented in the previous project. But I developed that project in c. Therefore, to not waste time implementing a sorting method in java, I used the prebuilt one.

Now, after having the list of centers sorted, I created a for-loop that goes from 0 to the total number of centers - 1(as I start with 0) and in each iteration it adds 1 to the value in that position. That by default in java an integer array is initialized with 0s. So this for loop will transform the array full of 0s I start having to an array full of 1s at the end of the for. But as we want the smallest number of centers possible, before finishing an iteration. It will check if the config is correct(with the function isCorrect previously explained in backtracking). And if it is correct it will break the loop.

Finally, once again, after finishing the algorithm it will check if the config is correct or not with the function isCorrect. To see if the config is correct or not. As in the worst case scenario, where there is no possible solution, the program will get out of the for-loop. But it wont know if it got out due to finding a correct solution or after iterating through all the for-loop.

Result analysis:

High Speed Sailing

During the testing of algorithms it could be found how with a sailors file with more than 14 sailors the branch and bound get out of memory. That's why I decided to edit the xs file and test it with 14 sailors, 13, 12, 11 and 10. The boat xs has also been edited to match the number of sailors. But as the boat just adds 1 number more or less to the value of each position it almost does not affect the time so it doesn't have been added.

Backtracking:

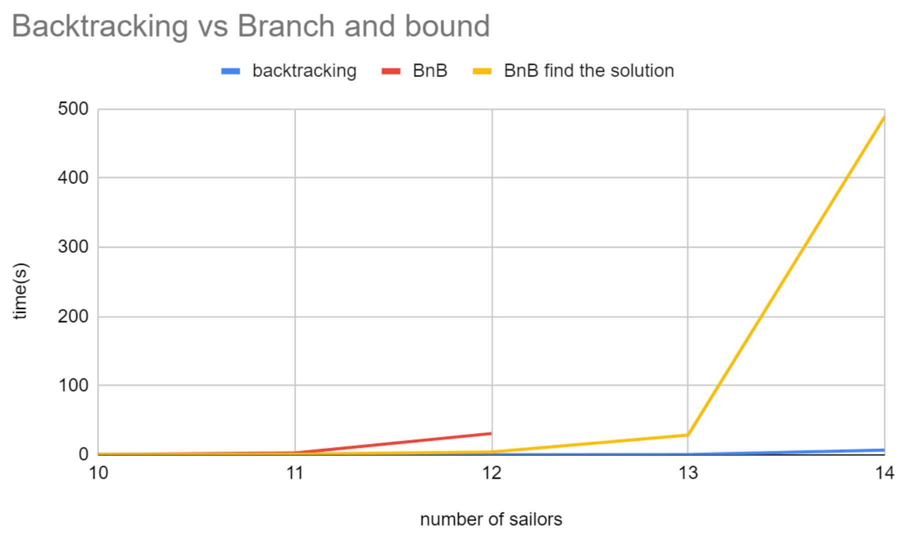
Number of sailors	Average time(s)
10	0,006
11	0,017
12	0,111
13	0,53
14	7

Branch and bound:

Number of sailors	Average time(s)	find the solution
10	0,31	0,31
11	2,736	1,1
12	30,838	4
13	301.83 seconds	28,51
14	heap space error(after around 13 mins)	488,75

As it can bee seen the branch and bound table adds 1 more column which finds the solution. Branch and bound seems to work well with a range of sailors from 10 to 12. But with 13 and 14 sailors it finds the solution but after finding the solution. It keeps executing something, though not printing a solution. When there are 13 sailors it takes 28,51 seconds to find the solution. But 30,8s to finish the programm. And with 14 sailors is even larger: to find the solution it takes: 488,75s to find the solution and after, more or less, 780 s it threw a heap space exception and finished the program, which makes no sense. So for values between 10-12 it seems to be working well. But for 13 and 14 sailors something wrong must be happening as the time it takes makes no sense.

To better analyze those tables let's take a look at the graph:



As it can be seen, branch and bound takes longer than backtracking, either taking the time it takes to finish the program or finding the solution. Even though branch and bound is known to be better than backtracking, as I said before it would depend on the case. Because, as explained before, branch and bound uses a lot of memory and takes way more time than backtracking to explore a solution space(as it needs to generate the solutions and sort them in the queue). But the thing is that it gets an optimal solution as the first solution. It will prune almost all the solutions, so even if it takes longer to explore, it will have to explore far less. Which works very well with large amounts of data. But as I am working with small amounts of data, backtracking is better. If no error was thrown due to memory space error and I could check the times it takes to branch and bound in the large dataset compared to the time it takes to backtracking. It will be seen how branch and bound is faster.

Regarding the time it takes to finish the program. A possibility could be that this problem is not the best to solve by using PBCBS, as for how it works how the real speed is generated. You can not prune most of the solutions until you have already explored 50 % or more of the solution space. Even though that shouldn’t in any case make the branch and bound find the solution with 13 sailors in 28,51 seconds and finish the program in 300 seconds. So something else must be the cause of the problem. The most probable cause is an error in my implementation. Even though I looked at it many times and I can’t figure out what's wrong with it. As what I did makes a lot of sense

Full fleet

As said before, the full fleet problem has been solved by using 2 different approaches: backtracking and greedy. By analyzing both algorithms with the different datasets I can get the following tables:

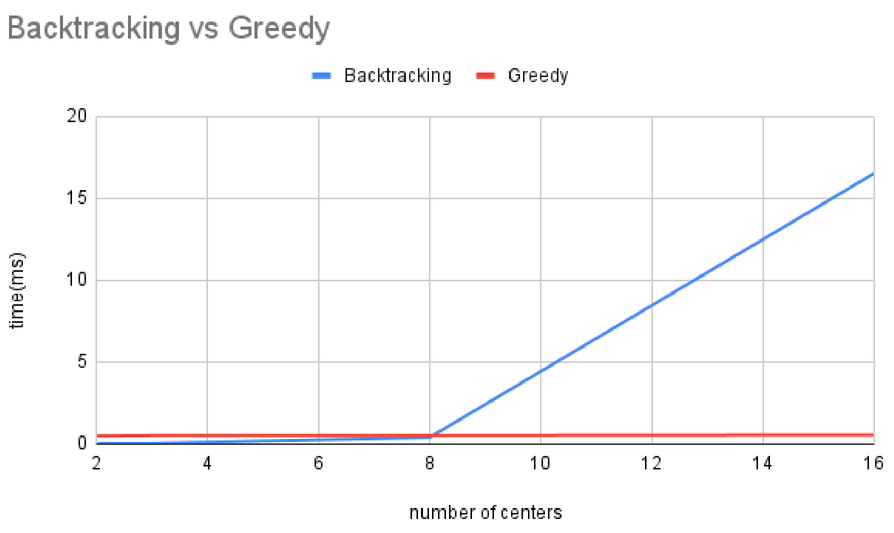
Backtracking:

Dataset	Average time(ms)	Solution	Number of centers
XS	0,0356	2	2
S	0,0473	No solution	3
M	0,4076	5	8
L	16,525	6	16

Greedy:

Dataset	Average time(ms)	Solution	Number of centers
XS	0,4972	2	2
S	0,5204	No solution	3
M	0,5226	5	8
L	0,5535	6	16

To better analyze those tables let's take a look at the graph:



As it can be seen, greedy has a linear cost which is what we ideally look for when developing a non-exhaustive algorithm. Whereas backtracking has a linear cost similar to greedy but a little less until the 8 centers. But on the 16th it rises exponentially. Even though the change is between 8 centers and 16 seems the start of an exponential function. It is not exponential. If we had far more centers it would be seen how backtracking will not rise exponentially. This abrupt change in the values between the 8 and 16 is probably due to the pruning. As it must have found a good solution a little bit later. Whereas greedy found the best solution, which means that greedy had the best possible case for that set of data, as it couldn't have been faster. If you take that into account the time backtracking takes is not that much. Even less, if we see that the time is in ms. I did not develop a brute force approach separately to see how it is compared to backtracking. But if I did, it could be seen how backtracking is at least a hundred times faster than brute force. Because brute will go through all the possible options. While backtracking is probably pruning between 80% to 90% of them.

Coming back to greedy and backtracking again. It can be seen how from the start to the middle backtracking is faster than greedy. That's because: even though the greedy algorithm is faster than the backtracking one(Except for the xs dataset because both will go through the 2 and only centers). The greedy sorts the centers from bigger to smaller in terms of the number of differ types of boats they have before executing its algorithm. So for really really small amounts of data, greedy is not worth it compared to backtracking. Because the time it takes sorting the data plus the time generating a solution is bigger than the time backtracking takes to just generate the best solution.

In conclusion, greedy is much faster than backtracking(except for really small amounts of data in which it is not). But it's a non-exhaustive algorithm so the answer it gets might not be the best one. Even though, taking out the first dataset which best answer was 2(out of 2 centers it had) and the second dataset which had no solution. Greedy has found the best solution for either the L dataset and the M dataset. Even though it depends on probability. Therefore it might have the worst possible case where he got the worst answer. That is very unlikely for a problem like that, where you have to find 6 types of boats between different centers having them ordered in terms of the number of different types they have. The probability that out of N centers the only boat type you missed was in the last center which only stored 1 boat type. So, overall, for problems where you can find a good way of finding a first good solution, it is pretty recommendable to use greedy if you don't have to strictly find the best solution. As you will find the best or a very close one.

Observed Problems

I could observe some memory issues with the branch and bound approach. As it generated far more configurations to add into the queue than the pc could handle. To try to solve it, a better marking could be done to improve the pruning and store less. But a more complex marking will affect the computational cost a little bit. Plus it would not solve the memory problems as the cost is exponential.

Regarding branch and bound another problem that I found was the incredibly high amount it takes to finish the program compared to the time it takes to find the solution with datasets with 13 or more sailors. Which is very probable to be some error in my implementation. But even wasting a lot of time trying to solve it, I was not able to figure out what was happening. As my branch and bound makes sense and I can’t see anything strange that might be causing the error.

Conclusion

This project has helped me settle the knowledge about combinatorial optimization and its different algorithms and types of approach. It also made more evident the difference between one algorithm or another, its pros and its cons. With this I believe I have got an important knowledge and skill to face any kind of similar problems I might face in the future. As I will be able to tell the best way of solving a problem of this type.

Moreover, I go home knowing the extreme importance of designing a good way of representing the data(configuration). As not choosing an appropriate design might raise the computational cost even exponentially, greatly increase the difficulty of implementing the algorithm or both at the same time. So one must take the appropriate time to design a good configuration and make sure it will work correctly and will not cause any type of issue in the future.

In addition, the thing that catched my attention the most was the greedy algorithm. As with that you can get one the best possible solution in no time. Which makes me believe greedy is the best possible algorithm to use. As for all the problems that do not strictly require the best possible solution, greedy is the way to tackle them. And if you do need the best possible solution I would use backtracking or Branch and bound, but initializing the PBCBS to the solution obtained with greedy. Therefore, it does not matter which type of problem, I do believe greedy is an approach that would be needed. Either to solve the problem or to be part of the solution.

After explaining that, I would say that to solve the full fleet problem. Geedy would be the proper algorithm. As in this specific type of problem, greedy will in most of the cases give you the best solution. Because in this type of problem the solution is 3 centers, there probably are different combinations of 3 centers that can solve the problem. Which means that there are different best solutions. So greedy, except for some cases where something strange happens, it will find one of the variations of the best possible solution.

Regarding the first problem, high speed sailing, it would depend on the size of the competition. Even though I could examine the time it would take to branch and bound to find the solution for a large amount of data. I am more than sure that it would be faster than backtracking. But the thing is that probably the competition Cat the hobby wants to organize is probably a normal to small one. Which means that the best way to approach it would be backtracking. As for small/medium-small amounts of data is faster. Because for this amount of data the amount that branch and bound extra prunes compared to backtracking does not surpass the extra time it takes for BnB to sort the queue and generate the configurations.

Bibliography

Source I consulted to develop the project:

Pol Muñoz: *S1_Notes*. September 2022

- S1 Notes.pdf

Geeksforgeeks: Difference between Backtracking and Branch-N-bound. 30th January 2023

- https://www.geeksforgeeks.org/difference-between-backtracking-and-branch-n-bound-technique/

Geeksforgeeks: N Queen Problem using Branch and bound. 16th January 2023

- https://www.geeksforgeeks.org/n-queen-problem-using-branch-and-bound/

Geeksforgeeks: Traveling salesman using Branch and bound. 22th February 2023

- https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/