KnapsackWriteup

Michelle King, Maggie Casale

February 22, 2016

Introduction

In this set of experiments we explored the performance of three different search techniques with two mutation techniques on a set of six knapsack problems, and two different values for the maximum number of tries (i.e., the maximum number of evaluated answers). A third of these runs use 1,000 `max-tries`, while another third uses 10,000, and the final third uses 100,000.

Our Functions

`Hill-Search`: Takes a mutator function (like tweaker, tweaker-with-rates), an instance, and the number of tries it attempts. Our start instance is a random answer generated by random-answer. From here, we create a loop that initially assigns our "current" as the start instance and assigns the variable "tries" as 0. We do this to print our initial instance so we can see what the initial randomly generated score was. In the loop we check if "tries" is greater than the max number of tries. If so, the function stops and returns the "current" variable. Else, we check if the score of a tweaked instance is better than the score of "current". If it is, the hill search recurs with the tweaked instance and an increased number of tries. Else, the hill search recurs with the "current" instance and an increased number of tries.

`Hill-Search-with-Random-Restart`: Takes a mutator function (like tweaker, tweaker-with-rates), an instance, and the number of tries it attempts. Our start instance is a random answer generated by random-answer. We do this to print our initial instance so we can see what the initial randomly generated score was. Our restart num is the number of items in a knapsack and will be used as the variable to track when to cause a restart. We set this so that the frequency of restarts is connected to the number of items in the knapsack. From here, we create a loop that initially assigns our "current" as the start instance, assigns "last-best" to track the best instance that has occurred so far, assigns the variable "tries" as 0, and sets the variable "counter" as 0. In the loop we check if "tries" is greater than the max number of tries. If so, the function stops and returns the "last-best" variable. Else, we compare our restart-num and counter. If counter is greater than restart-num, we cause a restart to occur. This happens by checking the score of the last-best instance with that of the current instance, the better score of the two is assigned 'last-best' and recurs with a new instance from random-answer and increases the number of tries. Else, we check if the score of a tweaked instance is better than the score of "current". If it is, the hill search recurs with the tweaked instance and an increased number of tries. Else, the hill search recurs with the "current" instance and an increased number of tries.

`Tweaker`: Tweaker is a mutating function. It takes in an instance and creates a new instance of either adding (if the knapsack is underweight) or removing (if the knapsack is overweight) an item from ':choices'. This item is chosen randomly.

`Tweaker-with-Rates`: Tweaker-with-Rates is another mutating function, it takes an instance. We assign initial as a tweaked instance, this allows us to assign it to current. Here we create a loop for performing tweaks. We use initial to determine the number of tweaks to perform by handing initial and instance (we were handed) to rate-to-tweak. Here we determine which mutation to perform by checking if the instance (we were handed) is over or underweight. If overweight, we remove items tweak-num times. Else, its underweight and we add items tweak-num times. We always make an instance of the results from these to update the knapsack.

`Rate-to-Tweak`: This takes a tweaked-instance and a current instance. We get the rate or effective slope of search by handing these instances to Rate-It. If the rate is less than .25, we return 1. If the rate is more than .25 but less than 1, we return 2. Else we return 3.

`Rate-It`: This takes a tweaked-instance and a current instance. We find the difference between the two total weights and divide this by the capacity of the knapsack. This gives and effective slope of search, showing us how fast the knapsack is filling/emptying.

`Remove Item`: In 'remove-item' we map choices and multiply each element by it's index (plus one for zero case, and we subtract one from each element after) to create a new vector. That allows us to randomly chose one of these elements without causing a stackoverflow error, and flipping the 1 to a 0 in :choices.

`Add Item`: In 'add-item' we use the result of 'remove-item' and map it with a vector of each element is its index (plus one for zero case, we subtract one from each element after). In this map we remove items from the index vector, which results in only the zero indices from :choices. Like 'remove-item', this allows us to randomly chose one of these elements without causing a stackoverflow error, and flipping the 0 to a 1 in :choices.

`Make Instance` : In 'make-instance' the process from creating an instance in 'random-answer' is repeated. Now, we give it our new choices and the instance. This is handed to 'included-items', which grabs the appropriate elements from the knapsack; allowing us to recreate elements in the new instance.

Overarching Comments: Our Tweaker-with-Rates finds a rate of change for the weight of the knapsack. This results in the tweaker tweaking (hill-climber climbing) at varying rates. This is helpful because as the function approaches a maximum, the slope should be 0 and the rate of change around it will typically be small. In our hill-search-with-random-restarts we have a counter for the hill-search that counts how many times that the last-best didn't change. Theoretically, as the function reaches the local maximum, the function should stop changing. Once it stops changing more, it is more likely to have reached a local maximum. At this point, the hill-climber will begin again at a random location. This is to increase the probability that the function will have found the global maxima and solved the knapsack problem to the best of its ability.

Experimental setup

I applied each combination of these 3 searchers and two values of `max-tries` to fairly randomly chosen knapsack problems:

- `knapPI_11_20_1000_4`
- `knapPI_13_20_1000_4`
- `knapPI_16_20_1000_4`
- `knapPI_11_200_1000_4`
- `knapPI_13_200_1000_4`
- `knapPI_16_200_1000_4`

(These names are abbreviated to, e.g., `K_11_20`, in diagrams below.) Half of these are 20 item problems, and half are 200 item problems.

I did 50 indepedent runs of each treatment on each problem, for a total of

$$5 \text{ searcher and mutation combinations} \times 3 \text{ different max-tries} \times 6 \text{ problems} \times 50 \text{ times} = 4500 \text{ runs}$$

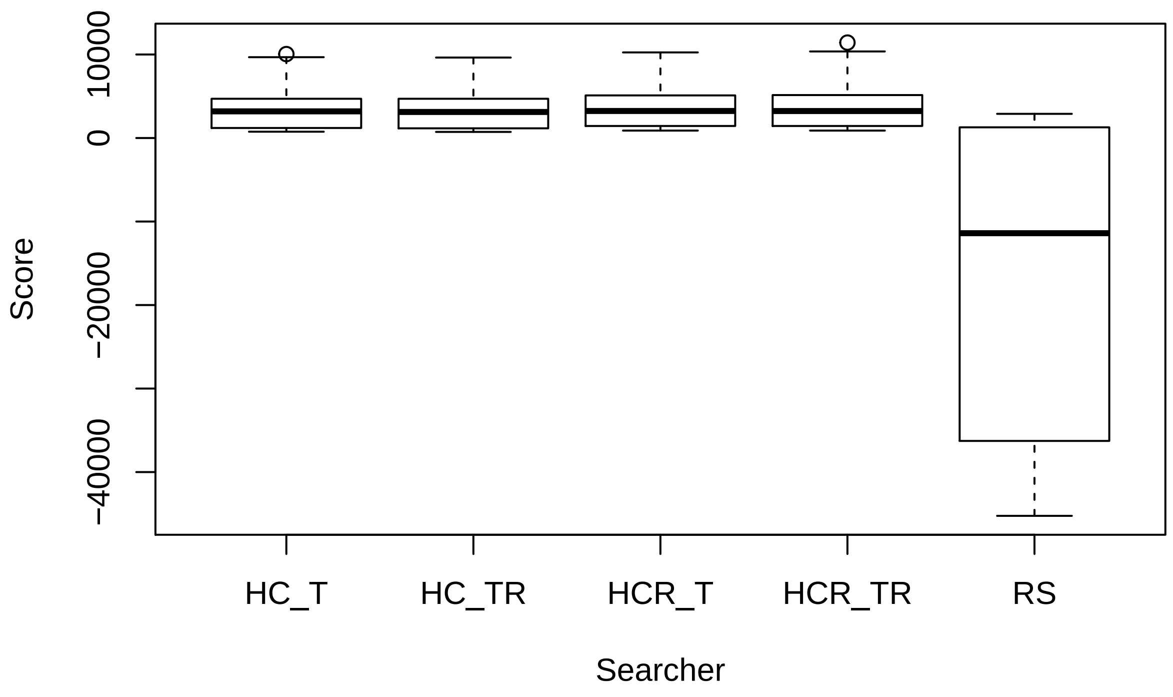
Results

A basic comparison of the searchers

As we can see in the plot below, the random search often returns negative scores. This means it doesn't always manage to generate an answer out of the realm of illegal solutions with the given maximum number of tries.

```
data_50_runs <- read.csv("/home/casal033/ECAI/simple-search/data/allThree50runs.txt", sep="",header=TRUI

plot(data_50_runs$Score ~ data_50_runs$Search_method,
     xlab="Searcher", ylab="Score")
```



A quick check confirms that (a) there are a fair amount such runs (511 in total) and (b) that all problems produce negative scores except `K_11_20`.

```
negs <- subset(data_50_runs, Score<0)
nrow(negs)
```

```
## [1] 511
```

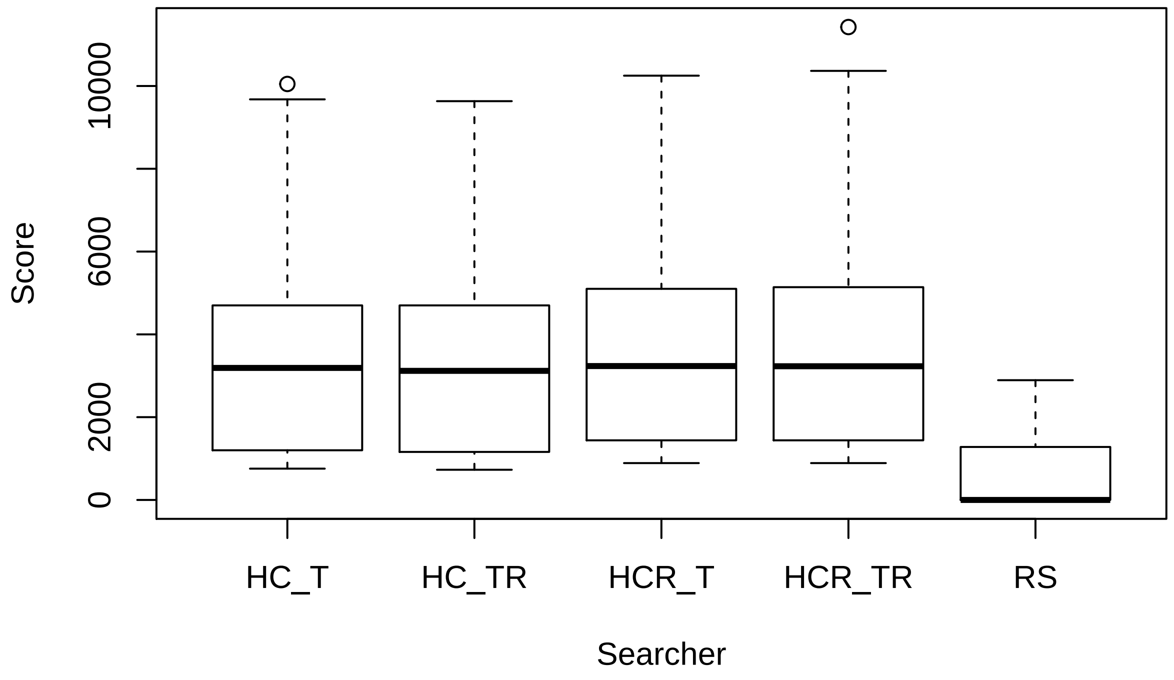
```
unique(negs$Problem)
```

```
## [1] K_11_200 K_13_200 K_16_200 K_13_20  K_16_20
## Levels: K_11_20 K_11_200 K_13_20 K_13_200 K_16_20 K_16_200
```

Since we don't really care *how* illegal our final solutions are, I made a new column (`Non_negative_score`) that has negative scores converted to zeros. This gives us a plot that is arguably a more "sensible" comparison than the one with the negative values.

```
data_50_runs$Non_negative_score = ifelse(data_50_runs$Score<0, 0, data_50_runs$Score)

plot(data_50_runs$Non_negative_score ~ data_50_runs$Search_method,
     xlab="Searcher", ylab="Score")
```



Here it looks like the right-middle boxplot, `Hill_Search_and_Tweaker_with_Rates`, may have a slight advantage over our other hill climbing methods; while the rightmost boxplot, `Random_Search`, is the worst of the search methods. However, all of the hill climber methods have similar medians, with the `Hill_Climber_Random_Restart` methods having marginally better score medians than the `Hill_Climber` methods.

Its interesting to notice there are two outliers, they are both high. Since the average score is around 3000, it's often the case we get solutions around this score or higher. The outliers show cases that are scored much higher than the typical data.

So let's run a pairwise Wilcoxon test:

```
pairwise.wilcox.test(data_50_runs$Non_negative_score, data_50_runs$Search_method)
```

```
##
##  Pairwise comparisons using Wilcoxon rank sum test
##
## data:  data_50_runs$Non_negative_score and data_50_runs$Search_method
##
##        HC_T    HC_TR   HCR_T   HCR_TR
## HC_TR  1       -       -       -
## HCR_T  2.2e-07 2.2e-07 -       -
## HCR_TR 2.2e-07 2.2e-07 1       -
## RS     < 2e-16 < 2e-16 < 2e-16 < 2e-16
##
## P value adjustment method: holm
```

Now all the differences with Random_Search are strongly significant, with $p < 2^{-16}$ in each case.

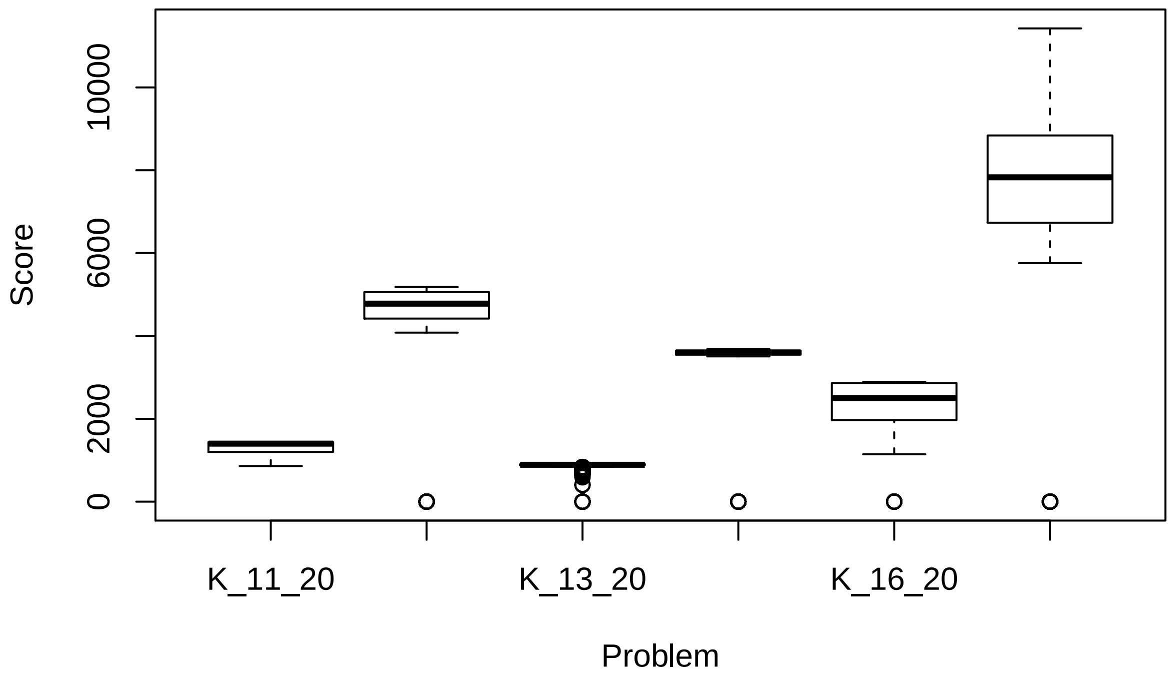
Comparing Hill_Climber_and_Tweaker with Hill_Climber_and_Tweaker_with_Rates, we get a p-value of 1. This tells us we're very sure of having the same outcome using either function combinations. We also see this with Hill_Climber_Random_Restarts_and_Tweaker and Hill_Climber_Random_Restarts_and_Tweaker_with_Rates. The main difference between tweaker and tweaker with rates is that the function with rates should find the peaks faster than that of just tweaker (either adding or removing a single item). The results here are similar because in these cases, max-tries is large enough to compensate for the inefficiency of the tweaker function compared to the tweaker with rates function. However, if we ran data sets with small values for max-tries, we would expect to see a greater difference in tweaker and tweaker with rates.

When comparing Hill_Climber_and_Tweaker OR Hill_Climber_and_Tweaker_with_Rates with Hill_Climber_Random_Restarts_and_Tweaker OR Hill_Climber_Random_Restarts_and_Tweaker_with_Rates we get a p-value of 2.2e-07 which suggests that either of the first two options or second two options will have the higher score outcome. We can see from the non-negative plot that the medians of the second two options have a slightly higher score than the first two options, so they are more likely to have a higher score on these search problems. The p values that compare the hill-climbers result in statistically significant results, however they are not quite as statistically significant as the hill climbers compared to the random search.

How do things change by problem? Max evals?

We saw earlier that there was some difference between the 20 and 200 item problems, because all the negative final results from `Random_Search` were on 200 item problems. This plot shows the performance on all six problems, regardless of search method used:

```r
plot(data_50_runs$Non_negative_score ~ data_50_runs$Problem,
     xlab="Problem", ylab="Score")
```

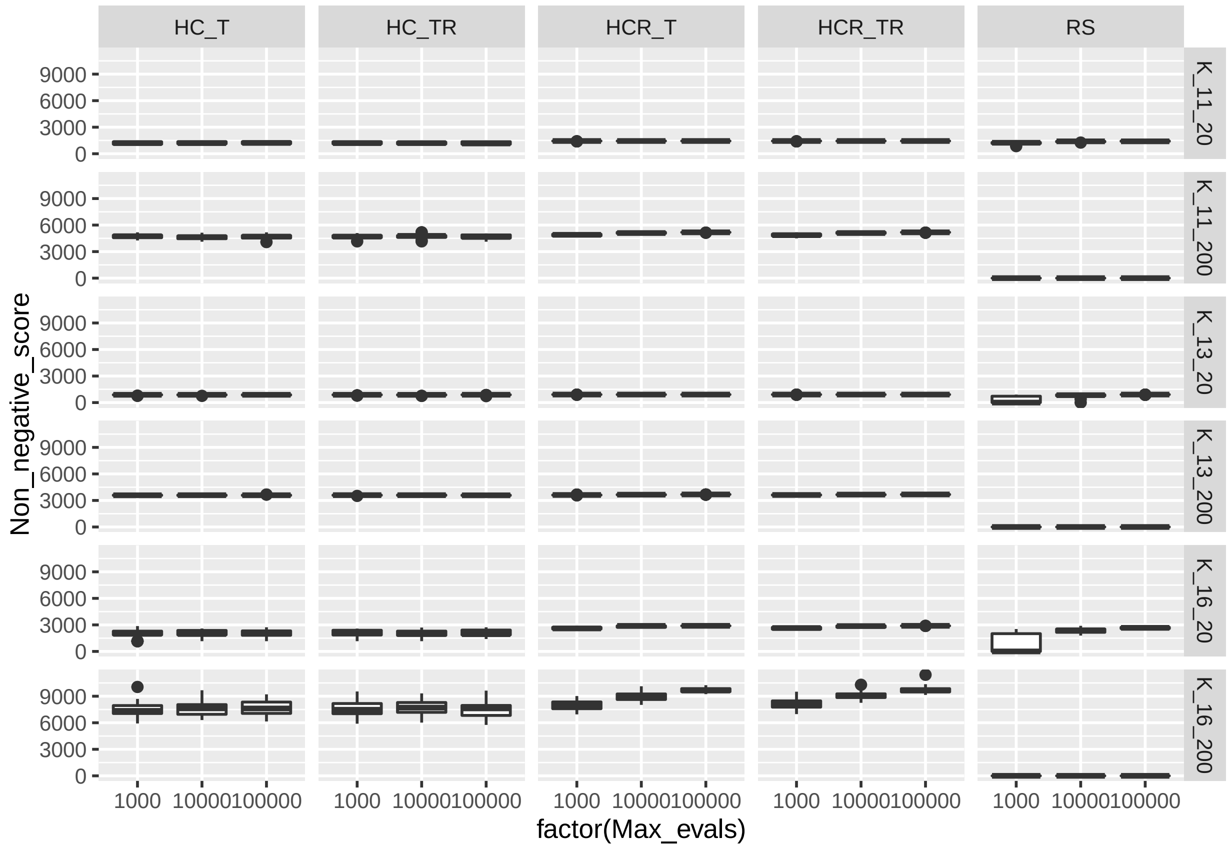


There are clearly differences. Some, such as the much higher values on the rightmost boxplot, are likely at least partly because of differences in the maximum possible values of the problems. Others seem to be more about the difficulty of the problems.

The following plot shows the performance broken out by essentially *all* our independent variables: Searcher, problem, and `max-tries`.

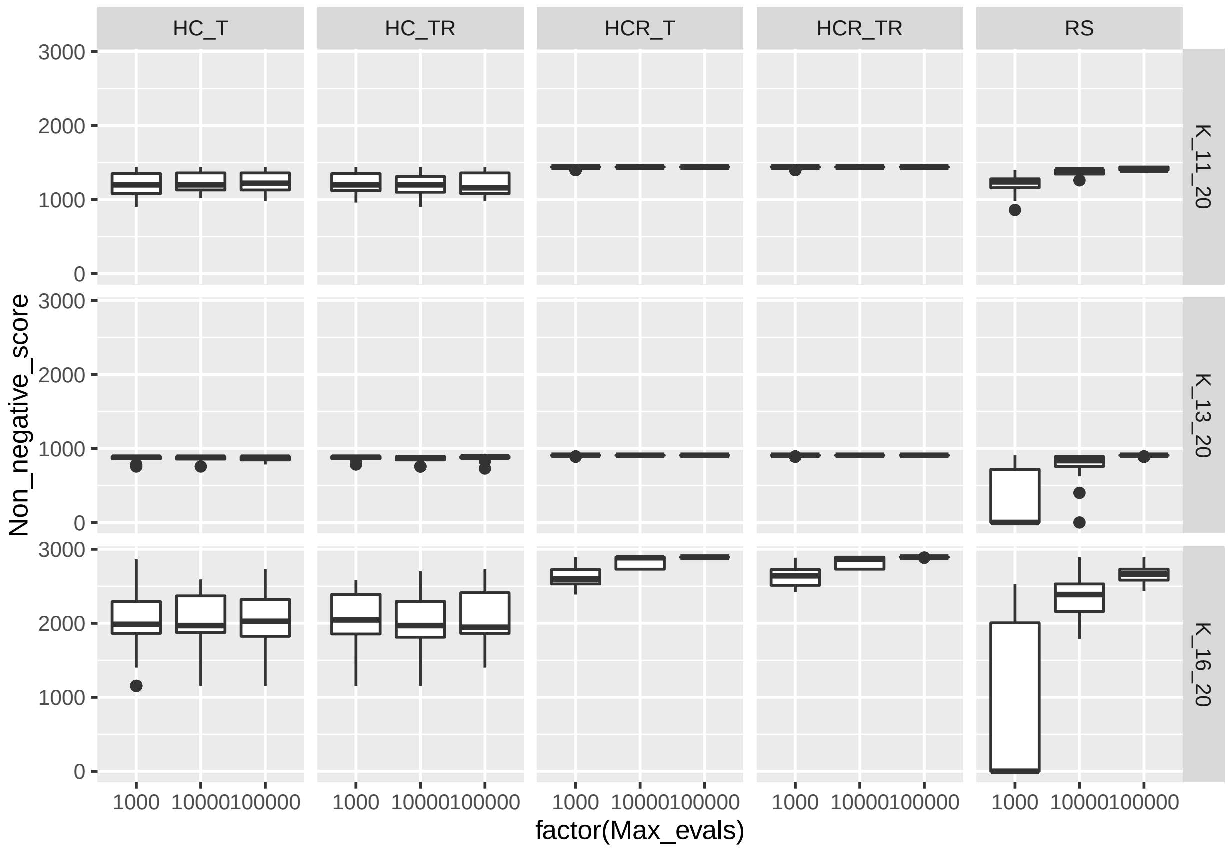
```r
library("ggplot2")

ggplot(data_50_runs,
      aes(x=factor(Max_evals), y=Non_negative_score, group=Max_evals)) +
 geom_boxplot() + facet_grid(Problem ~ Search_method)+ theme_grey(base_size = 10)
```



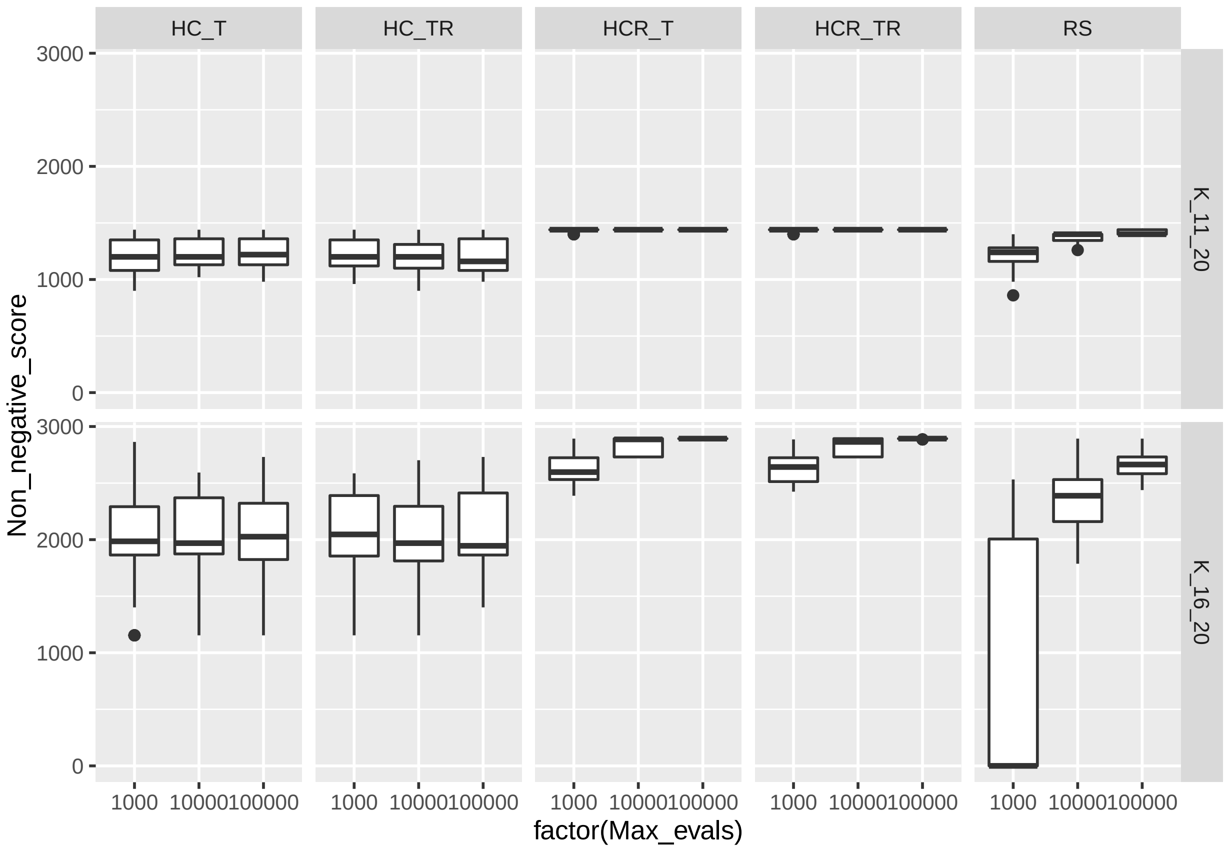
However this is difficult to interpret as a whole, so let's break it down a bit more.

```r
twenty_item_problems = subset(data_50_runs, Problem=="K_11_20" | Problem=="K_13_20" | Problem=="K_16_20
ggplot(twenty_item_problems, aes(factor(Max_evals), Non_negative_score)) + geom_boxplot() + facet_grid(
```



Here we can see only the 20 item problems. Again, it's a bit difficult to look at the `K_13_20` problem, so let's only look at 11 and 16 for these.

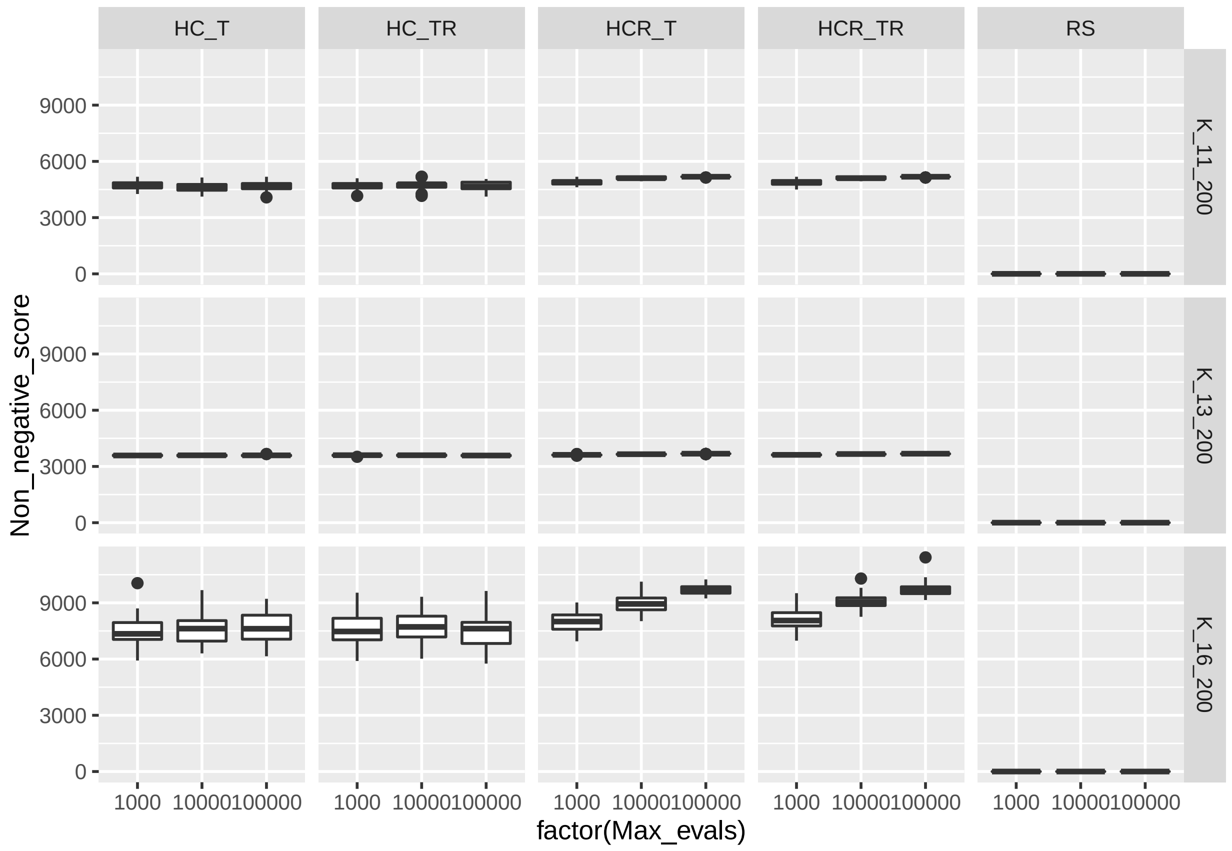
```
twenty_item_problems_11_16 = subset(data_50_runs, Problem=="K_11_20" | Problem=="K_16_20")
ggplot(twenty_item_problems_11_16, aes(factor(Max_evals), Non_negative_score)) + geom_boxplot() + facet_
```



Now it's easier to see that both of our `Hill_Climber_Random_Restarts` have the highest scores amoungst the rest, while our basic `Hill_Climbers` seem to have a wider and lower range of scores. `Random_Search` seems to be slightly better than our `Hill_Climbers`, with a higher median. This could be due to the fact that `Random_Search` generates a new knapsack each time, allowing it to find answers on other hills. `Hill_Climbers` have a chance to climb a hill given a random started knapsack, but what if they don't start on the "tallest" hill somewhere? This limits the `Hill_Climbers` range of options, while `Random_Search` can be at any point on any hill. It's not surprising that our `Hill_Climber_Random_Restarts` have the highest scores since they're allowed to generate a new random answer after attempting to "tweak" x many times with no improvment. The number x is the number of items in the bag (so x is 20 for the 20 item problems, and 200 for the 200 item problems). By generating a new random answer, we're more likely to climb all of the possible hills, especially with higher "max-tries".

Now let's look at the 200 item problems.

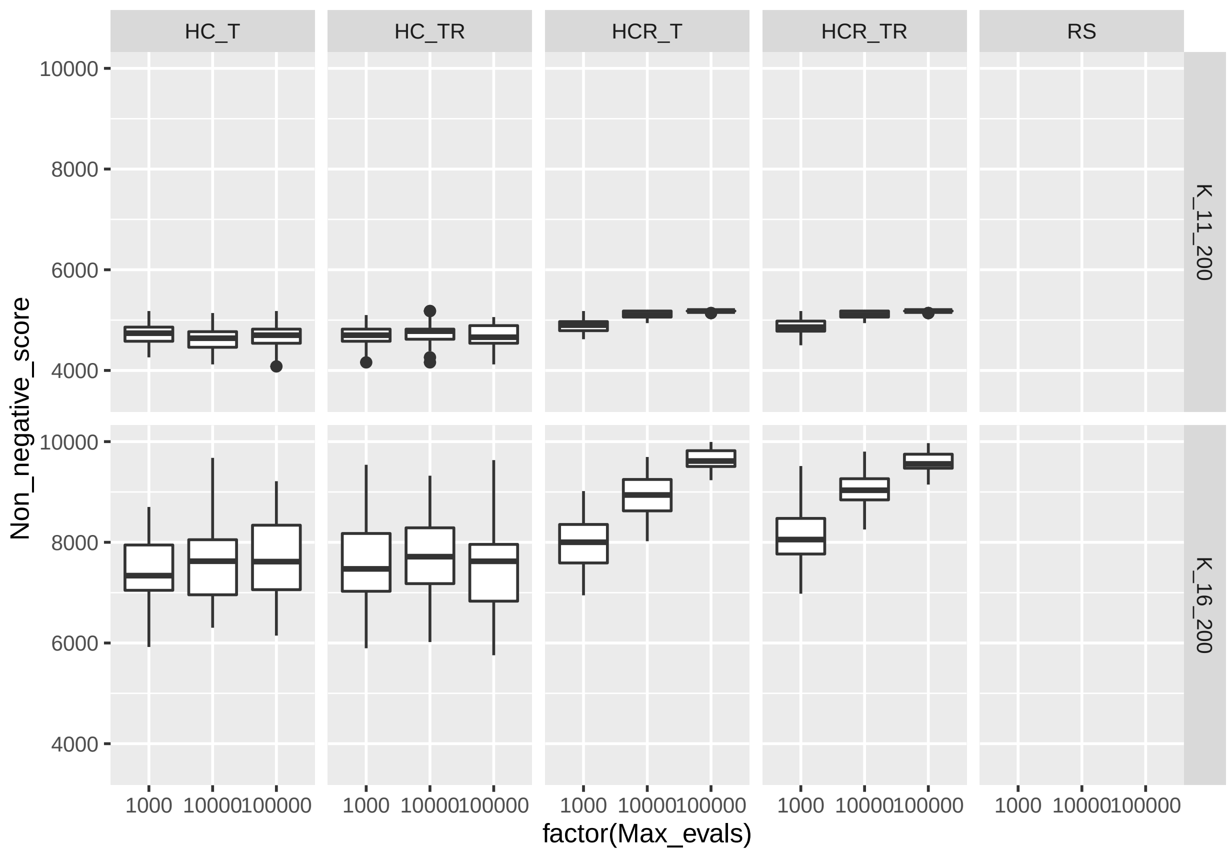
```
two_hundren_item_problems = subset(data_50_runs, Problem=="K_11_200" | Problem=="K_13_200" | Problem=="
ggplot(two_hundren_item_problems, aes(factor(Max_evals), Non_negative_score)) + geom_boxplot() + facet_
```



Again, this is difficult to read and the 13 problem seems to have consistent scores amoung all searchers, except `Random_Search`. Let's look at the 11 problem and 16 problem individualy.

```
two_hundren_item_problems_11_16 = subset(data_50_runs, Problem=="K_11_200" | Problem=="K_16_200")
ggplot(two_hundren_item_problems_11_16, aes(factor(Max_evals), Non_negative_score)) + ylim(3500, 10000)
```

```
## Warning: Removed 316 rows containing non-finite values (stat_boxplot).
```



This is much easier to interpret. It's important to note that all scores from `Random_Search` were at 0, so our y range was limited between 3,500 and 10,000. These scores were all negative to begin with, meaning all of the "best" randomly generated knapsacks were overweight given "max-tries" (10,000 or 100,000).

Looking at our `Hill_Climber_and_Tweaker` and `Hill_Climber_and_Tweaker_with_Rates`, these seem to fair similarily for both the 11 and 16 problem. All scores range between 7K and ~8.5K for the 16 problem with a median around 7750. For the 11 problem all scores range between 4.5K and 5K with a median around 4750.

Now the data for `Hill_Climber_Random_Restarts_and_Tweaker` and `Hill_Climber_Random_Restarts_and_Tweaker_with_` is a bit more interesting. Here we can see a 500 to 1000 score point improvement with the higher "max-tries" of 100,000 than 10,000. These also seem to have a narrower range of scores than those with 10,000 "max-tries", through those with 10,000 "max-tries" have scores in the range of those with 100,000 "max-tries".

Recursive partitioning

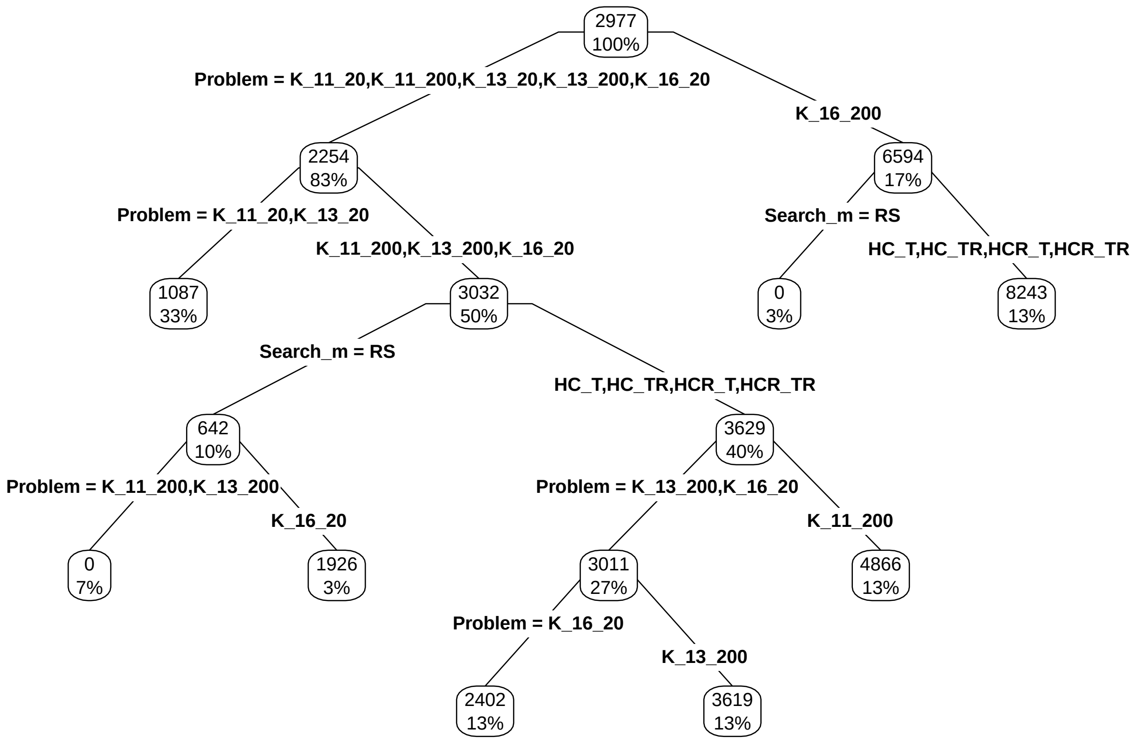
The results in the previous plot separating things by problem, searcher, and `max-tries` suggests that the interactions of the independent variables is somewhat complex, so I used `rpart` to try to understand the relative importance of the many differences.

```r
# https://cran.r-project.org/web/packages/rpart.plot/rpart.plot.pdf
library("rpart")
library("rpart.plot")

rp <- rpart(Non_negative_score ~ Search_method + Problem + Max_evals, data=data_50_runs)
rp
```

```
## n= 4500
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 4500 29834290000 2977.3580
##    2) Problem=K_11_20,K_11_200,K_13_20,K_13_200,K_16_20 3750  9272187000 2253.9740
##      4) Problem=K_11_20,K_13_20 1500   126708200 1087.4530 *
##      5) Problem=K_11_200,K_13_200,K_16_20 2250  5743550000 3031.6550
##       10) Search_method=RS 450   529144300  641.9467
##         20) Problem=K_11_200,K_13_200 300          0    0.0000 *
##         21) Problem=K_16_20 150   158258300 1925.8400 *
##       11) Search_method=HC_T,HC_TR,HCR_T,HCR_TR 1800  2002134000 3629.0820
##         22) Problem=K_13_200,K_16_20 1200   581416800 3010.6890
##           44) Problem=K_16_20 600   136125000 2402.2830 *
##           45) Problem=K_13_200 600     1102554 3619.0950 *
##         23) Problem=K_11_200 600    44044150 4865.8670 *
##    3) Problem=K_16_200 750  8788189000 6594.2800
##      6) Search_method=RS 150          0    0.0000 *
##      7) Search_method=HC_T,HC_TR,HCR_T,HCR_TR 600   634840200 8242.8500 *
```

```r
rpart.plot(rp, type=4, extra=100, Margin=0.0001)
```



The first split is between `K_16_200` and all other problems. This is because the score results for this problem are much higher than the others. From this, random search results in 0, and the hill climbers result in an average of 8243. Other interesting aspects of this graph include that random searches results in 0 for all 200 item searches. There is clear evidence from this graph that our hillclimbers are significantly better than random search. The hillclimbers also end up having similar results in terms of max score.

Conclusions

Based on the results, it is clear that overall, `HCR_TR` and `HCR_T` are the best, followed closely by `HC_TR` and `HC_T`. The results are statistically significant from random restarts and improve upon it by a large margin. To make a fair for comparison between tweaking with and without rates, tweaker with rates could incorporate rates into max tries. This would make processing equal. However, this somewhat defeats the purpose of using rates in the first place. Rates would allow the climber to climb the hill faster, while without rates will have the tweaker only change the knapsack by one item each step. Ideally, we would find an efficient way to make n tweaks to choices - this function would be more optimized and allow us to process faster and still reach the top most peaks.

Some improvements we could make would be to make the rates function change with number of total items. Right now we have it change by either 1, 2, or 3 items depending on the rate of change in the weight of the function. By having a dynamic rate based on the total number of items, we could more finely tune the changes in order to quickly reach the local maxima of the problems.