# KnapsackWriteup

*Michelle King, Maggie Casale*

*February 20, 2016*

*You should at a minimum describe what search methods you used and why, as well as analyzing and discussing your results. You don't need to "win" here, so it's totally OK if your analysis ends up being something of the form "Well, that didn't work very well", but I do really want you to think about what's happening in your data and make some effort to explain and understand what you're seeing.*

# 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).

*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.

Half of these runs use 10,000 `max-tries`, while the other half use 100,000. I'm pretty sure even more tries would help, especially on the larger problems, but my goal here was more to assess the impact relative performance of the three search techniques rather than try to optimally solve the problems, so I capped it at 100,000 so I wouldn't spend forever generating data.

# 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.

# 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_4`, in diagrams below.) Half of these are 20 item problems, and half are 200 item problems. Ultimately we'll probably want to apply our techniques to larger problems, but again my goal here was to try to understand the differences between my three search techniques, so I really just wanted hard enough problems to expose those differences.

I did 30 indepedent runs of each treatment on each problem, for a total of

$$3 \times 2 \times 6 \times 30 = 1080 \textrm{ runs}$$

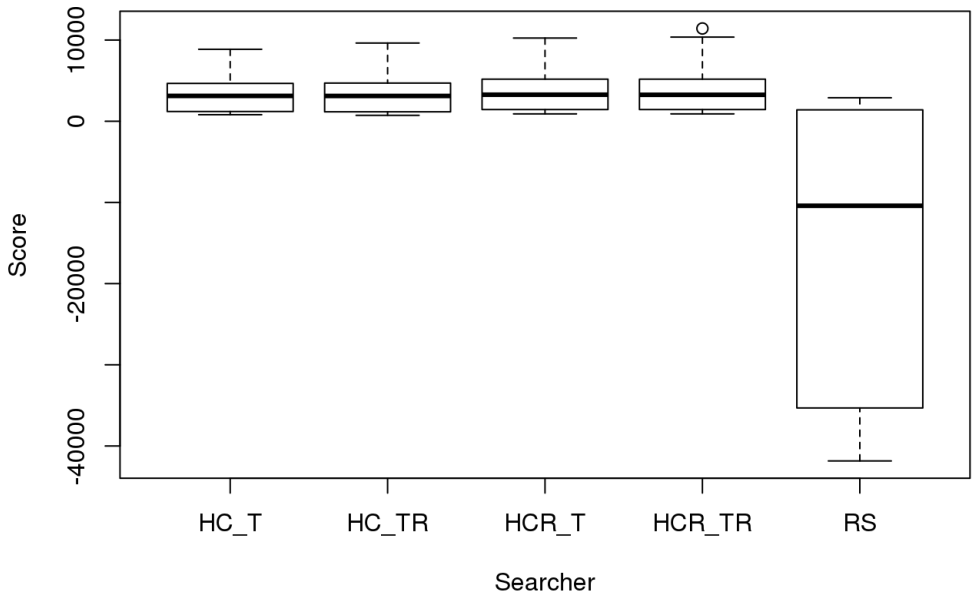
# Results

## A basic comparison of the searchers

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

```
data_30_runs <- read.csv("/home/casal033/ECAI/simple-search/data/both30runs.txt", sep="",header=TRUE)

plot(data_30_runs$Score ~ data_30_runs$Search_method,
     xlab="Searcher", ylab="Score")
```




A quick check confirms that (a) there are a fair amount such runs (180 in total) and (b) that all of them are on the 200 item problems.

```
negs <- subset(data_30_runs, Score<0)
nrow(negs)
```

```
## [1] 180
```

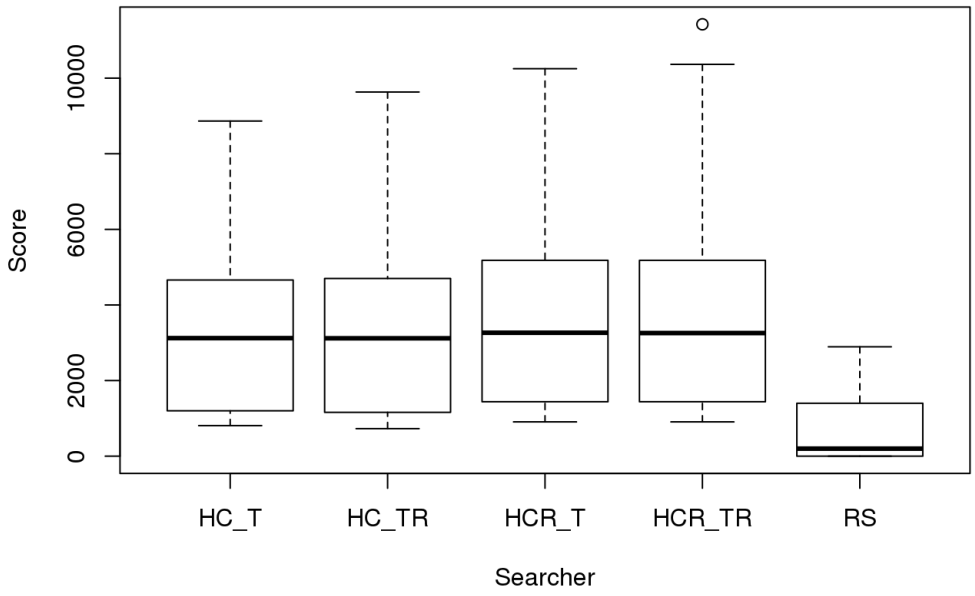
```
unique(negs$Problem)
```

```
## [1] K_11_200_4 K_13_200_4 K_16_200_4
## 6 Levels: K_11_20_4 K_11_200_4 K_13_20_4 K_13_200_4 ... K_16_200_4
```

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_30_runs$Non_negative_score = ifelse(data_30_runs$Score<0, 0, data_30_runs$Score)

plot(data_30_runs$Non_negative_score ~ data_30_runs$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.

So let's run a pairwise Wilcoxon test:

```
pairwise.wilcox.test(data_30_runs$Non_negative_score, data_30_runs$Search_method)
```

```
##
##  Pairwise comparisons using Wilcoxon rank sum test
##
## data:  data_30_runs$Non_negative_score and data_30_runs$Search_method
##
##        HC_T    HC_TR   HCR_T   HCR_TR
## HC_TR  1.00000 -       -       -
## HCR_T  0.00092 0.00092 -       -
## HCR_TR 0.00092 0.00092 1.00000 -
## 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.

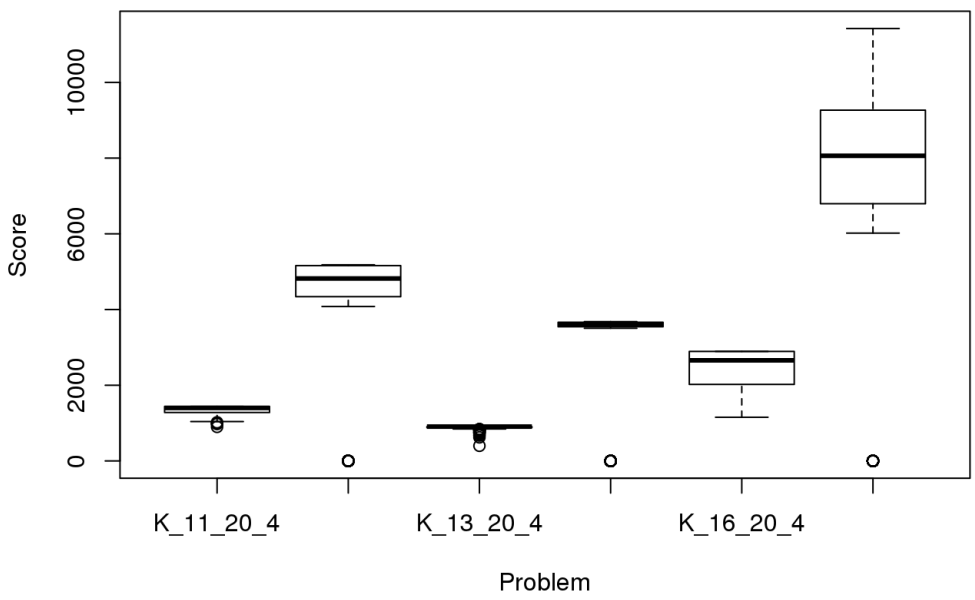
When comparing Hill_Climber_and_Tweaker OR Hill_Climber_and_Tweaker_with_Rates with Hill_Climber_Random_Restarts_and_Tweaker OR Hill_Climber_Random_Restarts_and_Tweaker_with_Rates we get a p-value of 0.00092 which suggests that either of the first two options or second two options will have the higher 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 a tad more likely to have a higher score.

# 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:

```
plot(data_30_runs$Non_negative_score ~ data_30_runs$Problem,
```

```
      xlab="Problem", ylab="Score")
```

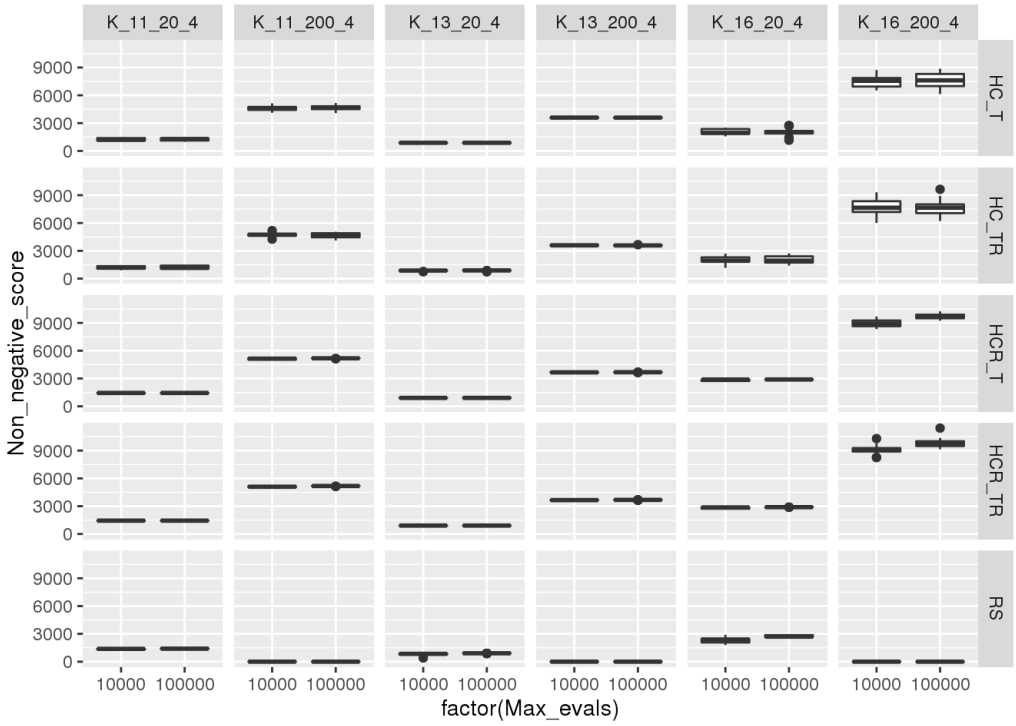



There are clearly differences. Some, such as the much higher values on the rightmost boxplot, are likely at least partly because of differences in the maximum possible values of the problems. Others seem to be more about the difficulty of the problems.

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

```
library("ggplot2")

ggplot(data_30_runs,
       aes(x=factor(Max_evals), y=Non_negative_score, group=Max_evals)) +
  geom_boxplot() + facet_grid(Search_method ~ Problem)
```




Reading this horizontally shows differences in the problems with, for example, `knapPI_16_200_1000_4` clearly having much higher values (at least for `HC_penalty`) than any of the other problems. Reading the columns vertically shows differences across searchers for a specific problem; it's clear for example that whatever advantage `HC_penalty` has is *much* stronger on the 200 item problems, where the other two searchers never get above zero.

This also suggests that using 10,000 tries instead of 1,000 often didn't change things much. There are exceptions (e.g., `knapPI_16_2000_1000_4` on `HC_penalty` again), but typically the medians are quite close. This suggests that we might stick to 1,000 tries in future *initial* explorations, and only switch to larger number of tries when we've identified which searchers, etc., we're especially interested in.

# Recursive partitioning

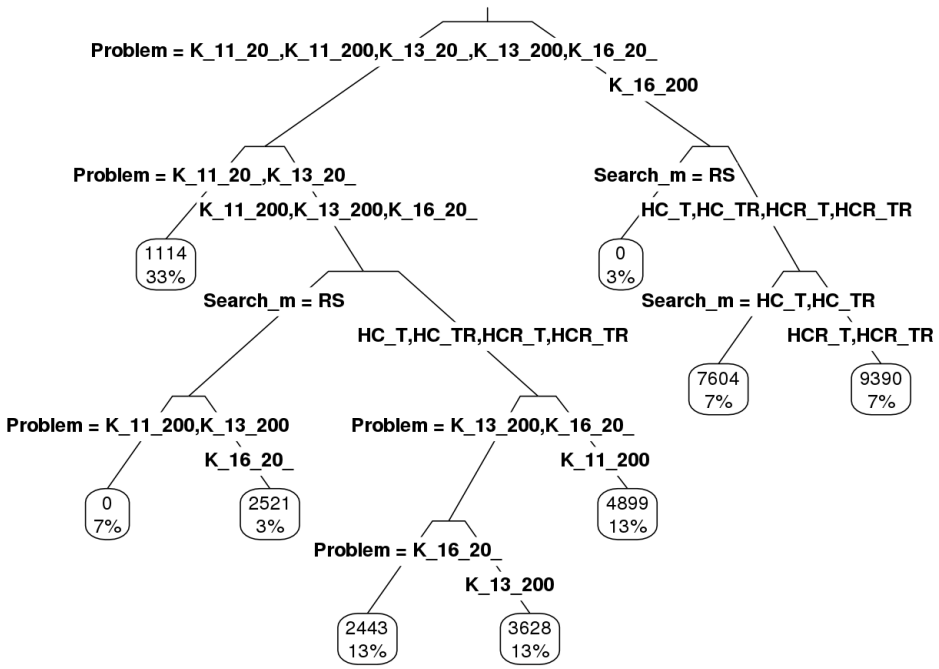
The results in the previous plot separating things by problem, searcher, and `max-tries` suggests that the interactions of the independent variables is somewhat complex, so I used `rpart` to try to understand the relative importance of the many differences.

```
library("rpart")
library("rpart.plot")

rp <- rpart(Non_negative_score ~ Search_method + Problem + Max_evals, data=data_30_runs)
rp
```

```
## n= 1800
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 1800 1.246238e+10 3050.8480
##    2) Problem=K_11_20_4,K_11_200_4,K_13_20_4,K_13_200_4,K_16_20_4 1500 3.655516e+09 2301.5010
##      4) Problem=K_11_20_4,K_13_20_4 600 3.815107e+07 1113.5420 *
##      5) Problem=K_11_200_4,K_13_200_4,K_16_20_4 900 2.206118e+09 3093.4730
##       10) Search_method=RS 180 2.593329e+08  840.2778
##         20) Problem=K_11_200_4,K_13_200_4 120 0.000000e+00    0.0000 *
##         21) Problem=K_16_20_4 60 5.148892e+06 2520.8330 *
##       11) Search_method=HC_T,HC_TR,HCR_T,HCR_TR 720 8.044852e+08 3656.7720
##         22) Problem=K_13_200_4,K_16_20_4 480 2.271930e+08 3035.5330
##           44) Problem=K_16_20_4 240 5.819842e+07 2443.1330 *
##           45) Problem=K_13_200_4 240 5.444269e+05 3627.9330 *
##         23) Problem=K_11_200_4 240 2.154186e+07 4899.2500 *
##    3) Problem=K_16_200_4 300 3.753166e+09 6797.5870
##      6) Search_method=RS 60 0.000000e+00    0.0000 *
##      7) Search_method=HC_T,HC_TR,HCR_T,HCR_TR 240 2.876269e+08 8496.9830
##       14) Search_method=HC_T,HC_TR 120 6.395869e+07 7603.8920 *
##       15) Search_method=HCR_T,HCR_TR 120 3.224118e+07 9390.0750 *
```

```
rpart.plot(rp, type=3, extra=100)
```




This indicates that despite the various differences between problems and different values of maximum evaluations, the choice of search searcher is the most important first-order difference, splitting on `HC_penalty` (on the right) vs. the other two searchers. After that split, though, the problems were the next most important factor along both branches. Focussing on the more interesting searcher ( `HC_penalty` ), `knapPI_16_200_1000_4` was “different” than the others, which isn’t surprising given the substantially higher maximum values found on that problem than on the other problems. Once `rpart` is focusing on that particular problem, it also highlights the substantial difference between the 1,000 and 10,000 maximum evaluation runs.

# Conclusions

Based on these runs, it's clear that at least for these six problems `HC_penalty` is consistently as good or, in some cases, substantially better than the other two searchers tried here. This suggests that having a gradient to act on in the "illegal" part of the search space is a significant advantage on these problems.

Have more evaluations does sometimes help, and occassionally quite a bit, but it often doesn't make a substantial difference, especially on the easier problems. So I might consider starting with just 1,000 evaluations in future explorations, saving the higher number of evaluations for when I've narrowed down the pool of search tools I really want to explore more deeply. (That would also be a good time to include some test problems with more items.)

Laslty, the facetted plot and the `rpart` analysis make it clear that I *really* should normalize my data by dividing all my scores by the highest score found for a given problem. That would reduce effects caused by disproportionate maximum values for problems like `knapPI_16_200_1000_4`, and allow tools like `rpart` to focus on differences caused by the choice of searchers or maximum evaluations.