MIE-PAA
Paul Rondeau
rondepau
December 15, 2014

MAX-SAT Solution Using an Advanced Iterative Method

Algorithm and Implementation:

The algorithm used to solve the MAX-SAT problem was an advanced iterative algorithm called Simulated Annealing, specifically one tailored to the MAX-3SAT problem. Simulated Annealing is a good approximation to the global optimum of a given problem in a large search space. No advanced data structures were used in this algorithm, given the simplicity of it. Only arrays used to track the current state of the algorithm as well as input clauses and variables' weights. These proved to be adequate for the problem as will be shown in the experimental results below.

A few sophisticated techniques were used that included 'repairing' the state of the algorithm, if it reached a non-feasible state. This was done by selecting a neighbour at random, computing feasibility and repeating if not feasible. Another technique used to improve results, was to store the highest cost computed during the algorithm, this would ensure if the global minimum was found, it would not be lost by the heuristic.

No large changes to the concept of simulated annealing were used. However, minor implementation were adjusted to increase performance. These included, the changes to the feasibility checking outlined below:

```
def feasible(state):
    for index, clause in enumerate(maxSAT):
        tmpArr = [0 for x in xrange(n)]
        for index, value in enumerate(clause):
            tmpArr[abs(value) - 1] = 1 if value > 0 else 0

        if not ((state[0] == tmpArr[0] ) or (state[1] == tmpArr[1] ) or (state[2] == tmpArr[2] )):
            return False
    return True
```

Here, each clause is checked, and converted into a new representation used to check equivalence with the current state. So, if a variable was set to 1 ($x_1 = 1$), the clause would be satisfied if there is an x_1 that is not negated in the clause. The *if* statement checks all of the variables with the converted clause for equivalence, thus identifying if a clause is satisfied. This algorithm assuming there are only 3 variables, but could be extending to n variables.

Working with the Heuristics:

There were several parameters that were used as input to find the optimal algorithm performance. These were:

#Set initial params
INITIAL_TEMP = 100 * input[1]
FINAL_TEMP = 100
ALPHA = input[2]
INNER_LOOP_STEPS = input[3]

By setting the initial temperature to a multiple of the final temperature, we use the ratio to calculate it easily. Thus, the three inputs temperature ratio, alpha and inner loop steps were used and could be changed easily to find optimal results. Adjusting these parameters to find optimal solutions proved difficult. Since there was no efficient way of calculating correct results for large data sets, many possibilities were tested blindly and compared by hand. However, it was found that checking the results against each other proved successful in finding which parameters performed better relatively. The experiments were planned before execution by creating *default* values for each parameter and fluctuating one input at a time to find better results. This was done programmatically with a many loops testing different values for each. This is shown here:

```
#!/bin/bash
sizeArr=(1 2 4)
tempRatio=(2 4 5 7 10)
alpha=(80 85 90 95 99)
innerLoop=(1 5 10 50 100)

defaultSize=4
defaultTempRatio=5
defaultAlpha=80
defaultInnerLoop=1

for i in "${sizeArr[@]}"
do
    echo Running $i Knapsack

    START=$(gdate +%s%3N)
    python maxSAT.py $i $defaultTempRatio $defaultAlpha $defaultInnerLoop > results/simAnnResults_$i.txt
    FINISH=$(gdate +%s%3N)
    echo $i $defaultTempRatio $defaultAlpha $defaultInnerLoop '|' $((FINISH - START)) > results/simAnnResults_timing.txt

done
```

Each input was given a default value shown here and then this for loop was repeated for each variable. These default values were chosen given past experience with this algorithm on the 0/1 knapsack problem as well as tweaks performed to check validity with this algorithm. Given the rigorousness of the inputs tested the space tested is satisfactory, especially when comparing computation time and accuracy of results against each other. This algorithm was able to efficiently compute solutions for instances with clause-to-variables ratio 1,2, and 4, but for 8 or more the algorithm took far too long to confirm any result. When the 8 clause-to-variable ratio was tested, the algorithm was

analyzed to find the contributing factor, and it ended up being in the repairing of the state. Since the repairing would flip a random variable and then test for feasibility, it took an exponential amount of time to continually call this, resulting in no results being found efficiently. The function that was causing the trouble is here:

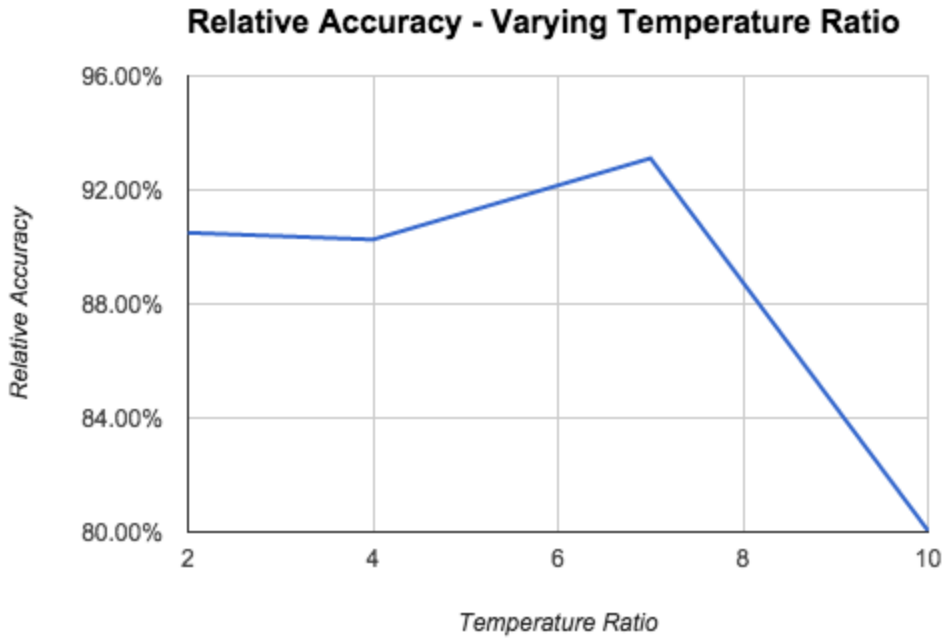
```
def repair(state):
    while not (feasible(state)):
        itemIndex = randint(0, n-1)
        state[itemIndex] = state[itemIndex] ^ 1
    return state
```

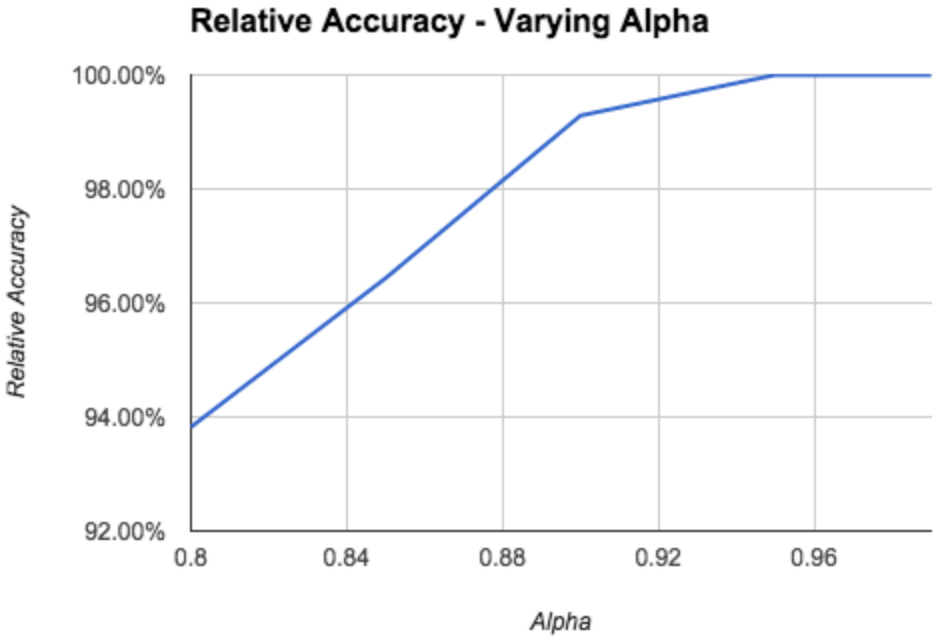
Experimental Evaluation:

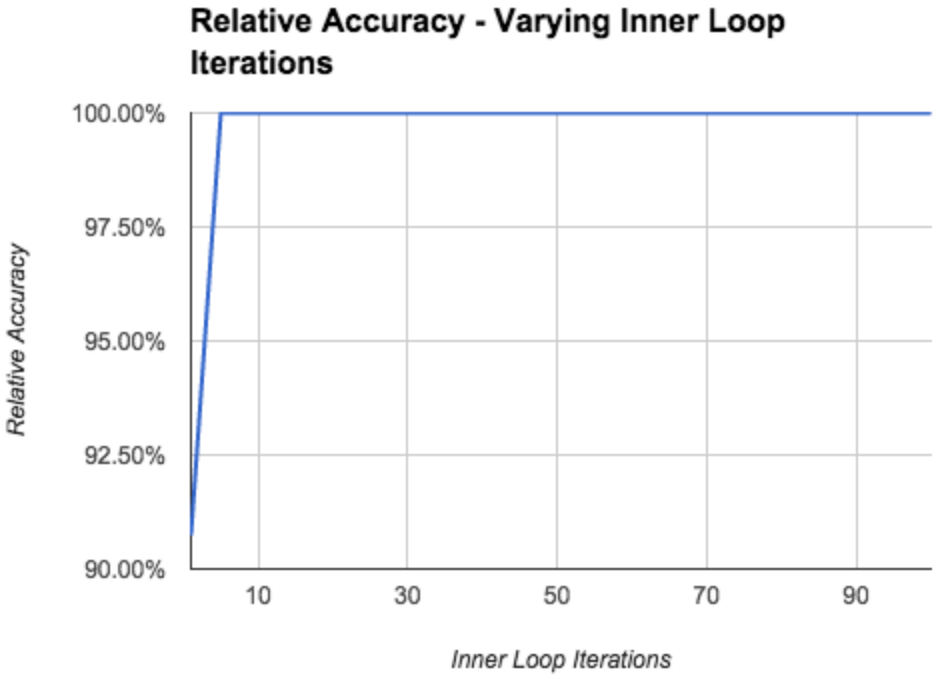
Since there was not an optimal solution for the instances given. A relative check was used to test performance for each variable. A probable, best case solution was found by increasing the variables to a maximum value that kept computation time manageable. These values were:

tempRatio=10
alpha=0.99
innerLoop=100

Thus, for the 4 clause/variable ratio test, the maximum weight found was 421 (for 50 instances of randomly assigned weights). If that is considered the maximum value, it is possible to compare all test cases with this value and measure accuracy that way. All of the following charts show the relative accuracy changing as a parameter changes.







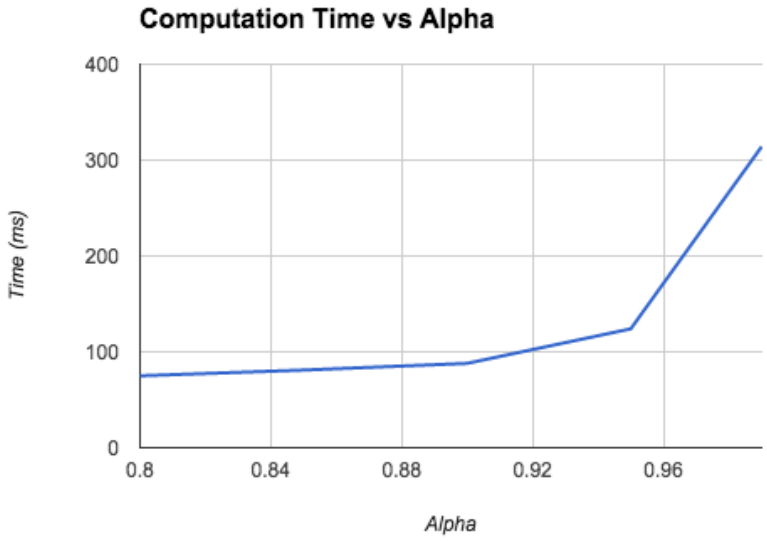
These experiments show a lot about the algorithm's performance, specifically, that the temperature does not play a significant role in the quality of results, the alpha is one of the largest factors for good results. Specifically when alpha is greater than 0.95, it yielded perfect

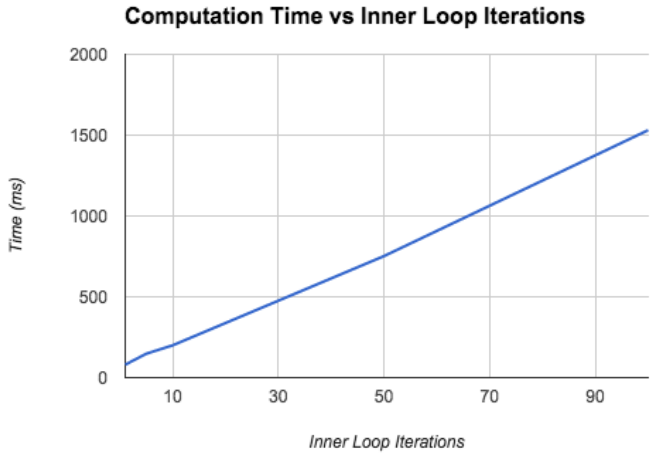
results. As well, the inner loop iterations increased accuracy dramatically, when raised to 5 iterations, the algorithm performed perfectly as well.

These results were run on 50 instances of various weights and checked for outliers, thus eliminating any randomness effects. This is an extremely powerful algorithm and can be push far, but on a non-supercomputer, anything higher than 4 clause/variable ratio proved impossible. So any real application would be tough with this algorithm in the given language at least.

From the data collected it can be said that pushing the alpha above 0.95 and the inner loop iterations above 5 is unnecessary and only increases computation time. The time for completion is outlined below.

Temperature Ratio	**2**	**4**	**5**	**7**	**10**
Time (ms)	80	83	76	78	74





As expected, the temperature ratio increase does not impact the computation time just as it didn't affect the accuracy. The change in alpha exponentially increases the computation time and the inner loop iteration only affected it linearly.

Conclusions:

From these results it seems as though the inner loop iteration count is the best variable in terms of performance gain. Alpha is a good to increase, provided the computation time is within reason and the temperature ratio has no effect on overall computation time.

SOURCE CODE:
https://edux.fit.cvut.cz/courses/MI-PAA/_media/en/student/rondepau/assignment5.zip