271P final project

[**Team Name**](https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK)

skynetUCI

[**Team Members**](https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK)

[Wu, Yu-Jie, yujiew19, 12646408](https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK)

[Parvez, Mohammed Farhaan, parvezm, 48768989](https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK)

[Bharot, Vineet, vbharot, 88649968](https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK)

(Stochastic) Local Search (SLS) - Topic: MAX-SAT

**Maximum Satisfiability Problem**

**Definition:**

Consider an n-variable [CNF(Conjunctive Normal Form)](https://en.wikipedia.org/wiki/Conjunctive_normal_form) formula F with m clauses and a truth assignment state σ ∈ {0, 1}n and i ∈ {1, 2, . . . , m}.

[Maximum Satisfiability Problem(maxSAT)](https://en.wikipedia.org/wiki/Maximum_satisfiability_problem) for F can be written as can be written as the following optimization problem over σ ∈ {0, 1}n:

minimize N(σ) = , where Ui(σ) is 0 if clause i is satisfied by σ, else 1. ∀i ∈ {1, 2, . . . , m}

N(σ) ≥ 0 and only equals 0 if and only if all clauses of F are satisfied. N(σ) is the **objective function**.

**State Space**: assignment of true/false(1/0) to the variables(literals) involved in the CNF. Here σ defines a particular state. We explore the state space for a state which results in the minimum value for the Objective function.

**Approach Description:**

To explain our approach we first need to define the flipping operation of a variable and the definition of the break-count of a variable (v) with respect to CNF and state σ.

**Flipping a variable(v): flip() =**

If the current value of the variable is True(1), then flipping it makes it False(0) and if the current value is False(0) the flipping it makes it True(1).

**Break Count**: The Number of currently satisfied Clauses in the CNF which will become unsatisfied by σ if we flipped the value of variable v.

**Approach:**

To begin our search we start with a Initial Truth Assignment and as this is an Optimization Problem then a **Greedy** Local Search move of flipping a variable in which the **maximum possible number of clauses, or with the least Break Count** is a good approach, but this suffers from Problems of Local Maxima/Minima and large plateaus.

To counter these we introduce a **Stochastic element** into our solution by selecting a random variable to flip with a probability called noise (**Pnoise**).

To widen our search of the state space we set an upper threshold limit on the maximum number of flips allowed in a search iteration(**max\_flips**)**,** after which we **reinitiate the initial assignment,** and start all over again. For all tries we track the **best truth assignment** with the minimum value of the Objective function by maintaining a **global variable (α).** If we find an assignment with 0 value of the objective function, then it means it’s the solution of the CNF and we return it as the best solution.

**Algorithm Explanation:**

**Pseudo-Code**

**function** : maxWalkSAT

**Input** : A CNF formula F

**Parameters** : max\_duration in seconds, max-flips; noise parameter pnoise ∈ [0, 1]

**Output** : A maximum satisfying assignment α for F

**begin**

**for** time ← current\_time to current\_time + max\_duration **do**

σ ← a randomly generated truth assignment for F // start fresh attempt

**for** j ← 1 to max-flips **do**

**if** N(σ) == 0 **then** //σ satisfies F then

**return** α // success

C ← an unsatisfied clause of F chosen at random

**if** ∃ variable x ∈ C with breakCount(σ, x) = 0 **then**

v ← x // freebie move

**else if** random(0, 1) < pnoise **then** // random walk move

v ← a variable in C chosen at random

**else** // greedy move

v ← a variable in C with the smallest breakCount

flip(v, σ) // Flip v in σ

**If** N(σ) < N(α) **then** // N(x) is the objective function

α ← σ

**end**

**end**

**return** α

**end**

**function**  : breakCount

**Input**  : A CNF formula F

**Parameters** : a truth assignment σ, a variable x to flip to calculate break count

**Output**  : number of satisfied clauses of σ which becomes unsatisfied on flipping x

**begin**

before\_flip ← σ

flip(x, σ)

after\_flip ← σ

break\_count ← 0

**for each** clause(c) in CNF(F) **do**

**If** U(c, before\_flip) == 1 and U(c, after\_flip) == 0 **do**

break\_count += 1

**end**

**return** break\_count

**end**

**NOTE: Ui** and **flip** are defined in the previous sections

**Pseudo-code Explanation:**

Try to find an assignment α that minimizes Objective function N(x) or makes the Objective function N(x) absolute minimum i.e. 0, till the time runs out or we find a solution. For each attempt with a maximum of max-flips flips allowed, we start with a randomly generated truth assignment for F so that we explore wider state space.

At each iteration we check if the current assignment σ satisfies the CNF if it does then we return it. Else we pick one of the unsatisfied clauses C of F at random. Now from this clause try finding a variable with break-count 0 ie. flipping it does not make any of the previously satisfied clauses unsatisfied. If found then flip it and move to the next iteration.

If not found then, from selected clause C with probability Pnoise perform a **random move** and select a random variable to flip and move to next iteration and with 1-Pnoise perform a **greedy move** which selects a variable from C with minimum break-count, flip it and move to next iteration.

After flipping a variable check if there is any improvement in the Objective function by comparing N(σ) < N(α). if it’s true then we’ve found a better assignment σ better than previous α, so we make σ(current assignment) the new α(best assignment).

If we run out of time then we return the α which is the best assignment we have found so far with the minimum Objective function value and maximum number of clauses satisfied.

**Properties Analysis/Time and Space Complexity:**

As for n variables we have two values, true and false. And as maxSAT is a variation of k-SAT problem which has a complexity of O(2\*(k-1)/k)n [[1](https://homepages.cwi.nl/~rdewolf/schoning99.pdf), [2](https://www.researchgate.net/publication/228780993_A_Probabilistic_Algorithm_for_k_-SAT_Based_on_Limited_Local_Search_and_Restart)], hence this algorithm has **Exponential time complexity.**

As for the space complexity as this algorithm explores the state space in depth first search manner without any other data structure hence this has **linear space complexity.**

**Attributes and Experiment Observations:**

**Attribute : max-flips**

Maximum number of flips allowed in an iteration with respect to an initial truth assignment (**Max-Flips)** is inversely proportional to the number of times we start fresh with a new initial assignment, which results in smaller state space explored.

But keeping max-flips extremely small to allow large state space search will also keep us from finding the best local solution. So we need an optimum value for the max-flips, hence we performed the following experiments to get to an optimum value.

1. **Satisfiable CNF verification**
   1. **Custom All true Solution assignment:**

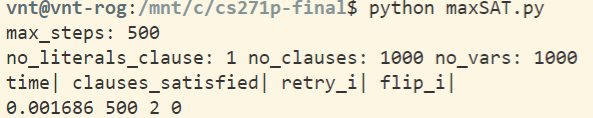
an all true n variable, n clause cnf with 1 variable in each clause like [(1,), (2,) …. (n, )]

* 1. **Custom Alternate true, false solution assignment:**

alternate true, false n variable, n clause with 1 variable in each clause assignment like: cnf = [(1,), (-2,), (3,).....(-(n-1), ) (n,)]

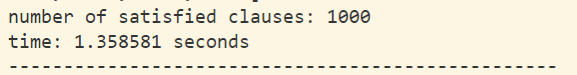
**Observations:**

**Example Run #1:**

****

**………..**

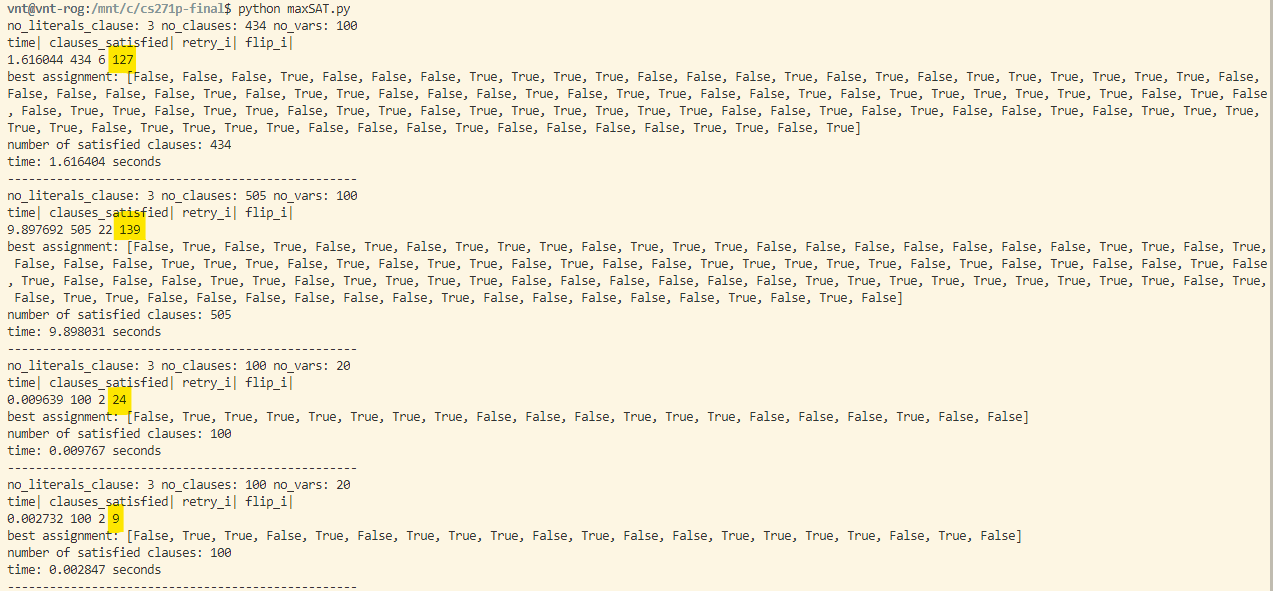
**………..**

****

Here our experiments observed that our initial assignment always results in around half of the clauses satisfied, which always results in a solution being n/2 steps away from the solution, where n is the number of clauses as there is always a free move available.

* 1. **Randomly generated satisfiable cnf:**

We have verified the validity of our solution by running on two 20 variables, 100 clauses 3-MAXSAT problem, 3rd **benchmark** of 100 variables, 505 clauses 3-MAXSAT problem and an 100 clauses 434 clauses Problem.

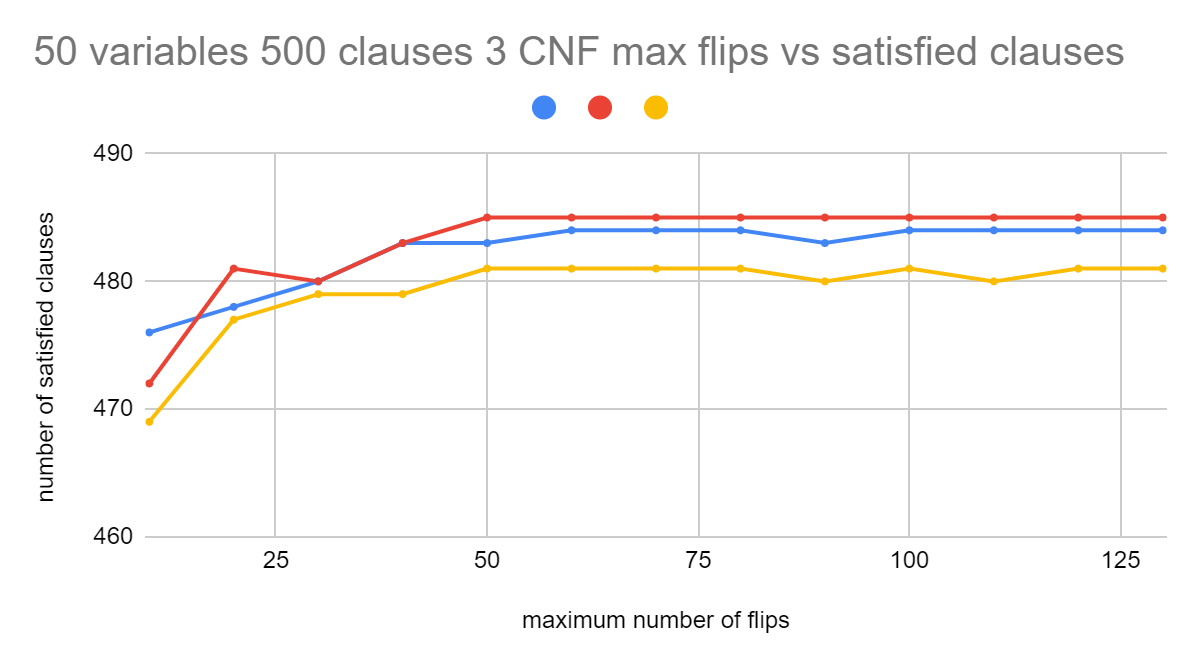


As shown above, our solution is able to reach the answer pretty fast with less that half the number of clauses flips performed (highlighted in yellow the value of flip\_i)

1. **Unsatisfiable CNF maximum satisfiable value convergence:**

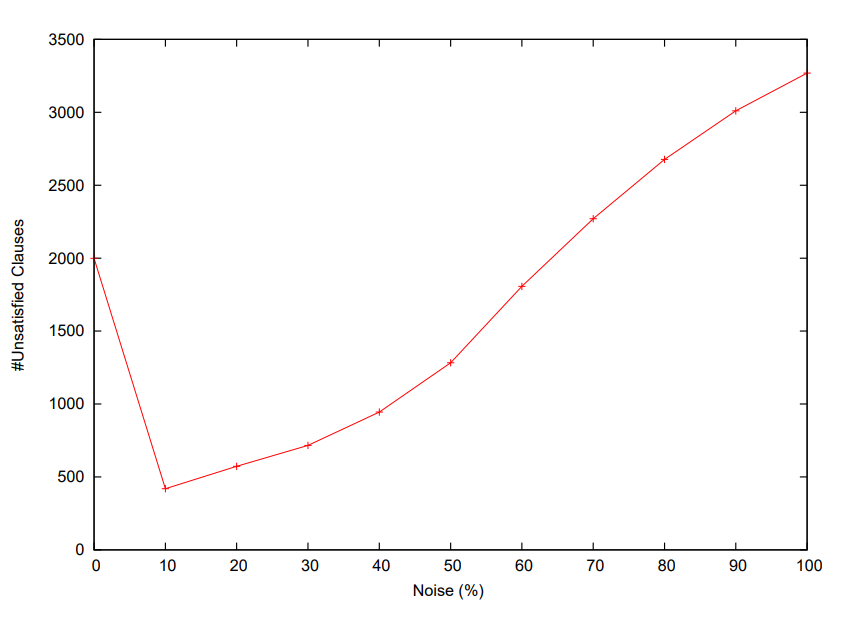
We have measured the number of flips required to reach the maximum value against different values of the max\_flips and charted the following graph. We cross verified the max number of clauses satisfiable by using [pysat RC2Stratified solver](https://pysathq.github.io/docs/html/api/examples/rc2.html).

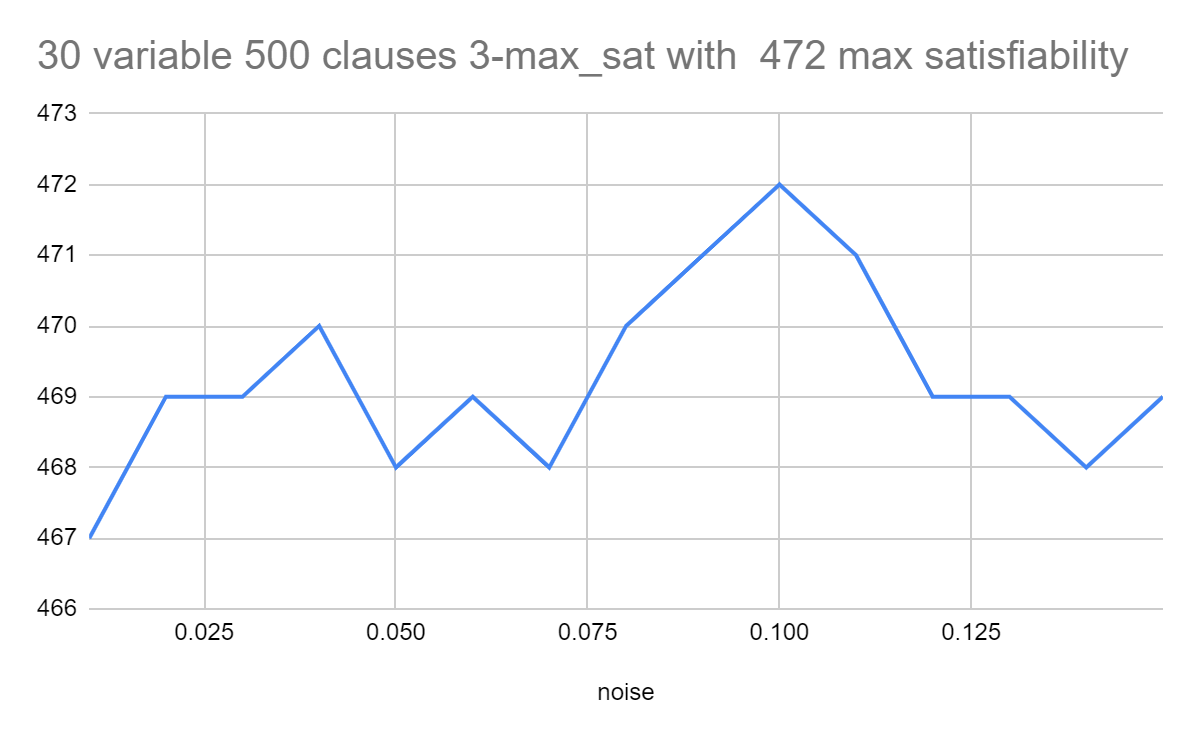
**Observations:** Here we observed that the maximum value of satisfied clauses is achieved around the 50 max flips, which is no\_clauses/10. Similar pattern is observed in other tests.

****

**Conclusion:** from the above two observations we’ve concluded that we need to set the max\_flips to at least half the number of clauses in the CNF.

**Attribute : Pnoise**

We then focused on **Pnoise**, the random move probability. After researching we’ve found a [Research Paper](https://1library.net/document/q05xp6gy-enhanced-walksat-finite-learning-automata-max-sat.html) which stated that “*Peak performance with respect to the lowest number of unsatisfied clauses is achieved when the walking probability was set to* ***10****.*” in the sections 6.1 on page 27.



Noise probability vs Number of Unsatisfied Clauses Noise probability vs Satisfied Clauses

dividers6-hack.dimacs.filtered: Variables = 35376,clauses = 132699 Variables = 30, Clauses = 500 3-CNF max Satisfiable: 472

From [Enhanced Walksat with Finite Learning Automata For MAX SAT](https://1library.net/document/q05xp6gy-enhanced-walksat-finite-learning-automata-max-sat.html) finer resolution experiment conducted by our team

We’ve also conducted our experiment on a much smaller data set and a similar trend is observed.

**Conclusion:** we’ve found that the best value of noise is 10% or 0.1

**Attribute:** **initial random assignment probability**

After that we focused on the **initial random assignment.**

Here we had 3 choices while setting the boolean value for a variable.

1. Equiprobabilityassignment of True and False each with 0.5 probability for each variable
2. Higher probability of True.
3. Higher Probability of False.

Logically thinking the number of flips required will be higher if the solutions has the higher concentration of True and initial assignment sets False with higher probability, and vice versa. So logically the Equiprobabilityassignment of True and False is Optimum.

The same inference is also verified experimentally.

**Conclusion:** we have found that equilikey assignment of true and false value for each variable in initial assignment is the best optimal way.

**Improvements:**

During our experiments we’ve found following improvements that boosted the performance of our solution.

**Implemented:**

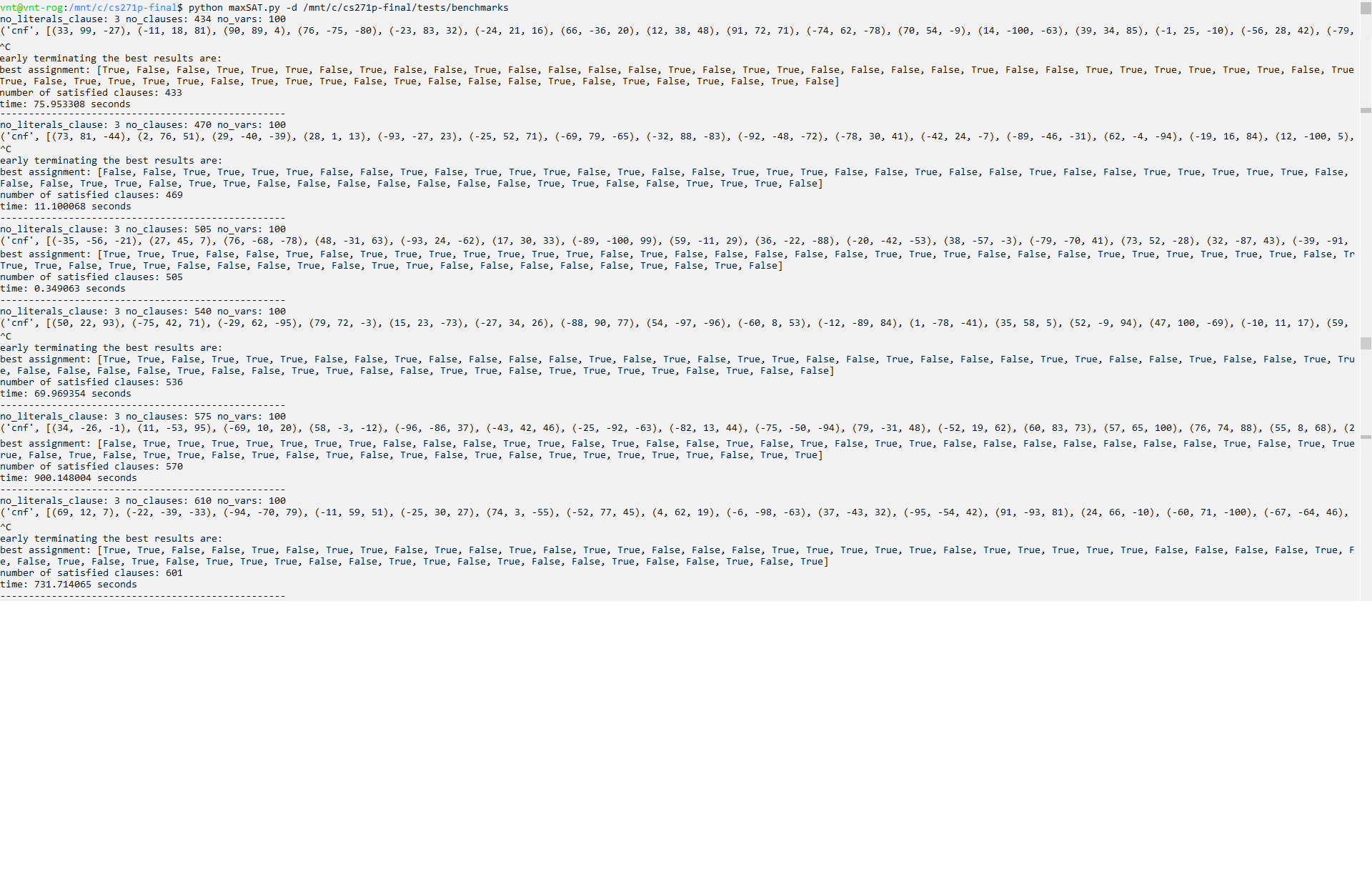
1. We found that checking the first initial unaltered(unflipped)assignment for Optimality gave better results than not doing so, as not doing so will skip on the truth assignment always.
2. As the performance of the solutions depends largely on the initial assignment, hence we decided to hash each initial assignment and during re-run we would pick the already occurring assignment with lower probability.

This optimization worked best when the number of variables were lower as the probability of initializing with the same initial assignment is inversely proportional to the number of variables.

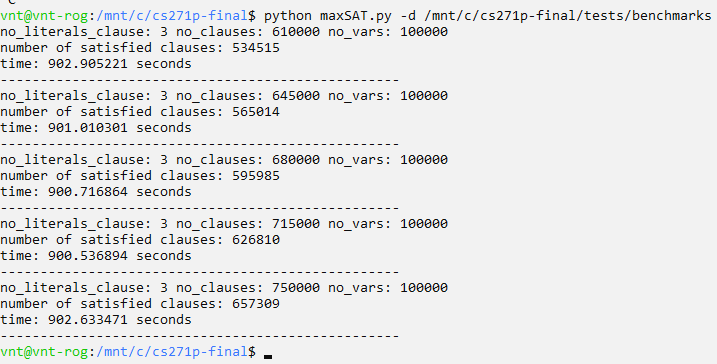
**Potential:**

1. Various Research papers have stated that using 2 or 3 flips instead 1 flip has given better results.
2. If the problem is 3-CNF then we can use a ⅞ approximation algorithm like Johnson's algorithm. More information can be found in the Research by [Howard Karloff and Uri Zwick](http://www.cs.umd.edu/~gasarch/BLOGPAPERS/max3satu.pdf) .
3. Improve the neighbourhood exploration algorithm as stated in the Research Article: [A Variable Neighborhood Walksat-Based Algorithm for MAX-SAT Problems](https://www.hindawi.com/journals/tswj/2014/798323/)

**First Few Benchmarks Results:**



**Last Few Benchmarks Results:**

****

Branch-and-Bound Depth-First-Search (BnB) - Topic: TSP

**Problem Definition:**

Given a list of cities and the distances between any two cities, find the shortest route that visits each city exactly once and returns to the origin city.

*State Space:*

State space (partial assignment of values to variables) is basically all the possible routes that have visited some cities (not visiting every city and returning to the original city) just once.

(For example, assume there are 5 cities, state space can be a route that has visited city A, city B, city C, which has a current path cost of 75, or a route that has visited city B, city D, which has a current path cost of 10, and so on.)

**Approach Description:**

Before describing the procedure of Branch-and-Bound Depth-First-Search algorithm, we first define the definition of the upper bound, current cost function, heuristic function, and evaluation function.

* **Upper Bound U:** a estimate of the maximum value for the optimal solution
* **Current Cost Function g(n):** the current cost from starting state to the current state
* **Heuristic Function h(n):** the estimated cost from the current state to the goal
* **Evaluation Function f(n):** the estimated optimal path cost for the possible solution of the current state

*Approach Procedure:*

For Branch-and-Bound Depth-First-Search algorithm, we first randomly select one city as the starting node and expand the state space tree as deep as possible.

When expanding one node, we consider *the current path cost* and a *predefined heuristic function* together as an *evaluation function*, which is used to compare with a *predefined upper bound* so that we could determine whether we should visit deeper states or not. If we decide not to expand deeper nodes, then the state space tree is *pruned*. Otherwise, we will keep on expanding this possible route.

During the BnB-DFS, when one possible route is found, we compare the cost of this route with the predefined upper bound, if the cost of the current solution is lower than the upper bound, we update the upper bound with the cost of the current solution; otherwise, we discard the current solution.

Once the BnB-DFS algorithm is done, we are guaranteed to get an optimal solution, which has the minimal path cost.

*Pseudo-code for BnB DFS algorithm*

1. Predefine the upper bound, heuristic function.
2. Using a heuristic, find a solution *xh* to the optimization problem. Store its value, *B* = *f*(*xh*). (Initially, set *B* to infinity.) *B* will denote the best solution found so far, and will be used as an upper bound for possible solutions.
3. Initialize a queue to hold a partial assignment solution with none of the variables of the problem assigned.
4. Loop until the queue is empty:
5. Take a node *N* off the queue.
6. If *N* represents one possible solution *x* and *f*(*x*) < *B*, then *x* is the best solution so far. Record it and set *B* ← *f*(*x*).
7. Else, *branch* on *N* to produce new nodes *Ni*. For each of these:
   1. If bound(*Ni*) > *B*, immediately stop expanding this node; since the lower bound on this node is greater than the upper bound of the problem, it will never lead to the optimal solution, and can be pruned.
   2. Else, store *Ni* on the queue.

**Algorithm Explanation:**

In our experiments, we tested 5 approaches for the TSP problem, which are the *brute-force DFS*, *naive BnB DFS*, *BnB DFS with Greedy*, *BnB DFS with an optimized lower bound*, and *BnB DFS with an optimized lower bound and Greedy method*.

***Algorithm 1* : Brute Force DFS Approach**

* explanation: the normal Depth-First search algorithm, which explores the whole state space search tree without pruning.

***Algorithm 2* : Naive Branch-and-Bound DFS:**

* upper bound: the current optimal path cost we have found so far
* current cost function g(n): the path cost from the start city to the current city
* evaluation function f(n): the estimated optimal path cost from the current city to the goal city
* When f(n)>=upper bound, we prune this path.
* When a solution (cost) is found such that cost < upper bound, we update upper\_bound = cost

***Algorithm 3* : Branch-and-Bound DFS with Greedy**

* greedy definition: when choosing which city to be expanded, choose the one with least cost. The next nodes are chosen in increasing order of edge costs.
* We modify Naive BnB (Algorithm 2) by adding this greedy strategy for choosing the next node to expand. Pruning condition remains the same

***Algorithm 4* : Optimized Branch-and-Bound DFS with A Lower Bound:**

* upper bound: the current optimal path cost we have found so far
* Our heuristic function h(n) is the minimum cost required to travel the remaining nodes in the graph, that are not in the current path. It is calculated by adding the two least cost edges of every node and halving the total sum.

h(n) = 1/2 \* (sum of the cost of two minimum cost adjacent paths of each unvisited city)

* current cost function g(n): the path cost from the start city to the current city
* evaluation function f(n) = g(n) + h(n): the estimated optimal path cost from the current city to the goal city

***Algorithm 5* : Optimized Branch-and-Bound DFS with A Lower Bound And Greedy Approach**

* This algorithm is a combination of Algorithm 3 and Algorithm 4. It uses both lower bound of remaining cities approach and greedy approach.
* greedy definition: when choosing which city to be expanded, choose the one with least cost.
* upper bound: the current optimal path cost we have found so far
* h(n) = 1/2 \* (sum of the cost of two minimum cost adjacent paths to each unvisited city)
* current cost function g(n): the path cost from the start city to the current city
* evaluation function f(n) = g(n) + h(n): the estimated optimal path cost from the current city to the goal city

**Properties Analysis:**

* time complexity: O(N!), where N is the number of cities. For this method, at worst case, the pruning condition will not be triggered and all possible branches are traversed.
* space complexity: To store the path that has been traversed is O(N), and the space for storing the graph as an adjacency matrix is O(N^2). Hence, total space complexity is O(N^2).

*Branch Pruning Analysis :*

1. *Algorithm 1* (Force DFS Approach):

In this approach, paths are not pruned during search, the whole state space tree is traversed.

1. *Algorithm 2* (Naive Branch-and-Bound DFS):

In this approach, the pruning condition will be triggered when current path cost is larger than upper bound.

1. *Algorithm 3* (Branch-and-Bound DFS with Greedy):

In this approach, since we choose the lower cost edges first, more optimal solutions are found earlier with higher probability and the pruning condition is expected to be triggered more often than in Algorithm 2.

1. *Algorithm 4* (Optimized Branch-and-Bound DFS with a Lower Bound):

In this approach, the pruning condition will be triggered much more frequently since a stronger heuristic function is used and the branch is bounded more tightly

1. ***Algorithm 5* (Optimized BnB DFS with a Lower Bound and Greedy Approach):**

In this approach, the pruning condition will be triggered the most as both lower bound and greedy approaches are used. This algorithm is expected to be the **most optimal** (verified from experiments shown in the following section).

**Experiments And Observations:**

In this project, we conducted 3 experiments, each of which considers different parameters.

The following is the description of experimental parameters:

* N: # of cities
* K: # of distinct path cost values
* U: mean of normal distribution for path costs
* V: variance of normal distribution for path costs

**Experiments Descriptions:**

* Experiment #1: our algorithms are tested with different number of N: # of cities
* Experiment #2: our algorithms are tested with different number of K: # of distinct path cost values
* Experiment #3: our algorithms are tested with different number of V: variance of normal distribution for path costs
* Performance Measurement: we consider *time elapsed (in milliseconds)* and *the number of cities expanded* to evaluate our performance.

**Experiment #1 (with different N: # of cities)**

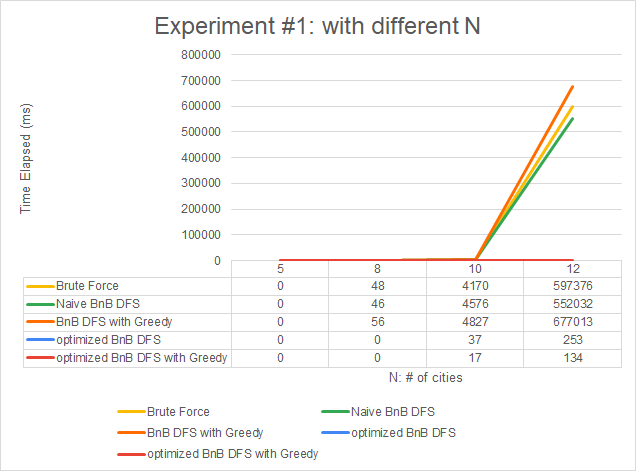
*Test Case #1*: N=5, K=0.4\*5\*5=10, U=100, V=5 Minimum cost: 482.889

*Test Case #2*: N=8, K=0.4\*8\*8=25.6**≈**25, U=100, V=5 Minimum cost: 777.943

*Test Case #3:* N=10, K=0.4\*10\*10=40, U=100, V=5 Minimum cost: 954.516

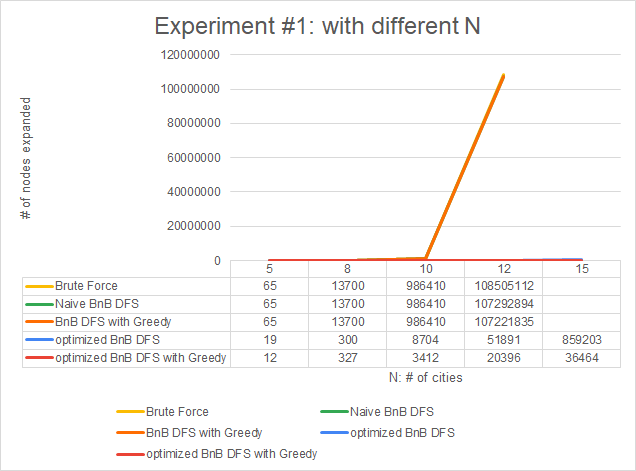
*Test Case #4*: N=12, K=0.4\*12\*12=57.6≈57, U=100, V=5 Minimum cost: 1129.63

*Test Case #5:* N=15, K=0.4\*15\*15=90, U=100, V=5 Minimum cost: 1408.85

For experiment 1, we vary the number of cities N from 5, 8, 10, 12 to 15.

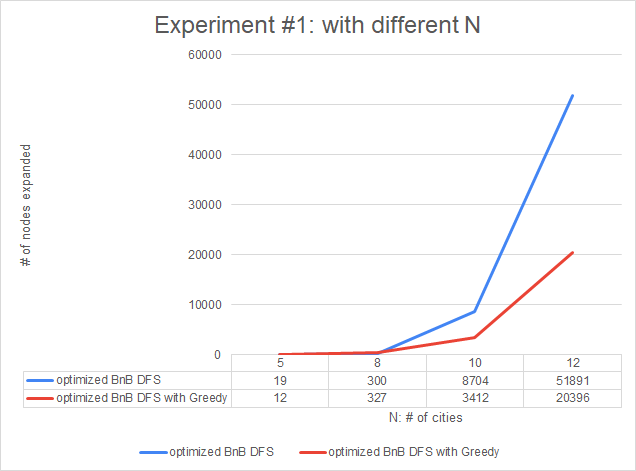
*Plot 1 :*

From the first left graph, we can observe that as N increases, the time elapsed increases for each algorithm. For the 4th and 5th algorithm, the time elapsed does not increase as much as the first 3 algorithms. The reason for this is that both algorithms were provided with a tighter bound so that pruning will be triggered more frequently during search. In this case, many possible routes are not considered, and execution time is reduced a lot.

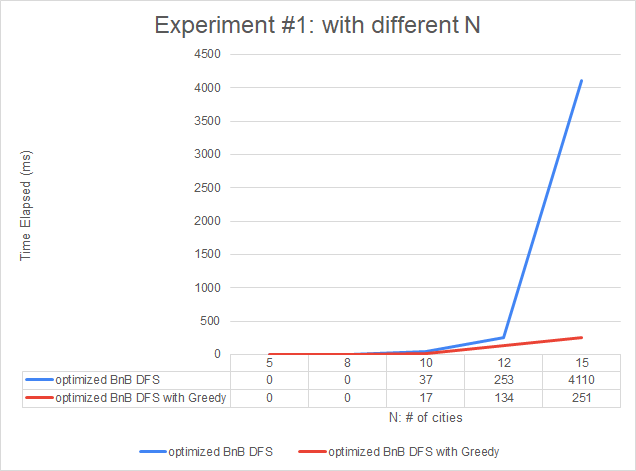


*Plot 2 :*

The same trend can also be observed from the second left plot graph, in which the number of nodes expanded was used as our performance measurement. As N increases, the number of nodes expanded increases a lot for the first 3 algorithms much less for the 4th and 5th algorithms.



*Comparison of Algorithms 4 and 5 :*

*Plot 3 :* From the third left plot graph, only the time elapsed for 4th and 5th algorithms are shown. We can see that, as N increases, the time elapsed increases a lot for each algorithm. However, for optimized BnB DFS with greedy, the performance was much better than the one without greedy. The reason for this is that, since when we decide which city to be explored, we first select the path with the lowest cost. Thus, the greedy technique led us to a solution with a lower cost, which updated the upper bound with a lower and stricter value; and therefore the pruning condition would be triggered more frequently than the 4th method. Also, with the greedy approach, we quickly found a possible route that was very likely to be the optimal solution during the search.

*Plot 4 :*

The same plot trend can also be seen in the forth left graph. As N increased, the number of nodes needed to be expanded also increased for the 4th and 5th algorithms.

**Experiment #2:** (with different **K: # of distinct distance values**)

Test Case #1: N=10, K=0.01\*10\*10=1, U=100, V=5, Minimum cost: 908.398

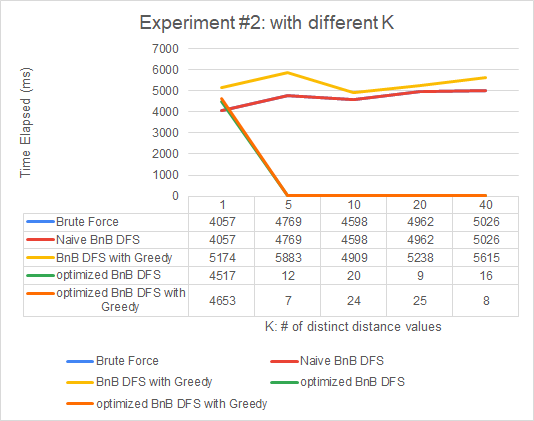
Test Case #2: N=10, K=0.05\*10\*10=5, U=100, V=5, Minimum cost: 1005.3

Test Case #3: N=10, K=0.1\*10\*10=10, U=100, V=5, Minimum cost: 977.367

Test Case #4: N=10, K=0.2\*10\*10=20, U=100, V=5, Minimum cost: 960.012

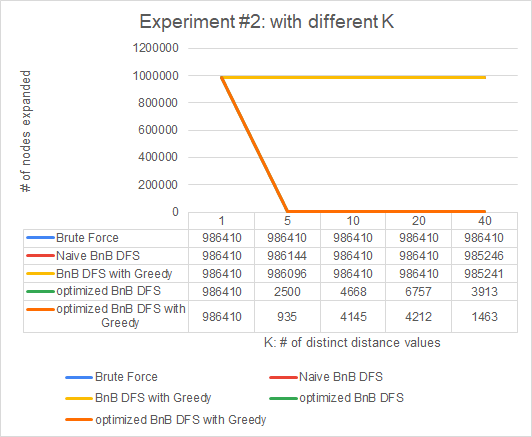
Test Case #5: N=10, K=0.4\*10\*10=40, U=100, V=5, Minimum cost: 939.733

For experiment 2, we vary the number of distinct distance values K from 1, 5, 10, 20 to 40.

*Plot 1 :* 

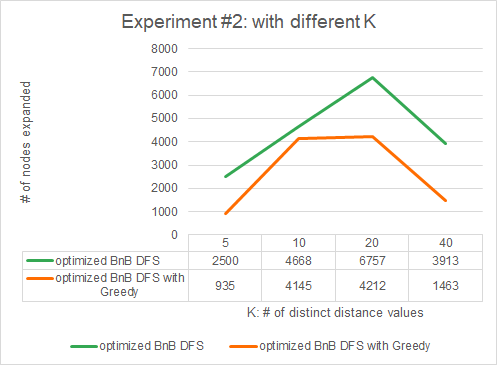
From the first graph, we can indicate that, for K=1, all the algorithms required approximate large execution time; while as K increased from 5 to 40, the time elapsed decreased a lot for all the algorithms except the first 2.

The reason for that is when we have only fewer distinct path costs, all the possible routes would cost pretty much the same so that the pruning condition would not be triggered that frequently. Thus the whole space state tree would still be traversed for all the algorithms.



*Plot 2 :*

We can observe this pattern much easilier from the second graph, for K=1, all the methods expanded the whole state space search tree. As K increased from 5 to 40, the number of nodes expanded decreased greatly for all the algorithms except the first 2.



*Plot 3 :*

From the third plot graph, we can observe that, if we applied the greedy approach, the number of nodes expanded decreased earlier as K increased.

**Experiment #3:** (with different **V: variance of normal distribution**)

*Test Case #1*: N=10, K=40, U=100, V=5%\*100=5, Minimum cost = 956.743

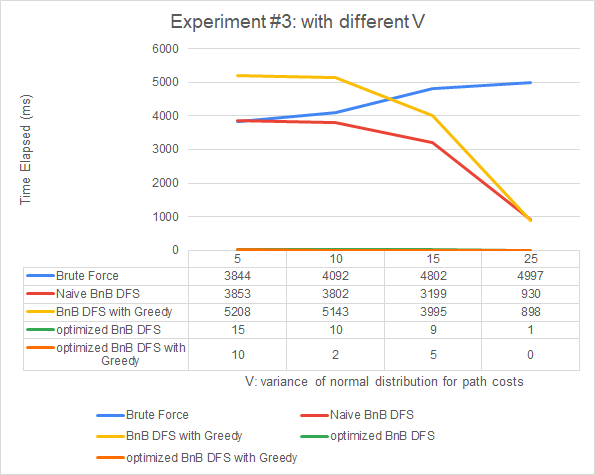
*Test Case #2:* N=10, K=40, U=100, V=10%\*100=10, Minimum cost = 922.05

*Test Case #3*: N=10, K=40, U=100, V=15%\*100=15, Minimum cost = 831.049

*Test Case #4:* N=10, K=40, U=100, V=25%\*100=25, Minimum cost = 677.714

For experiment 3, we vary the variance of normal distribution for path cost from 5, 10, 15, to 25.

For normal distribution, the larger the variance value is, the wider the range of path cost value is. In other words, the larger the variance is, the bigger the difference between path costs.

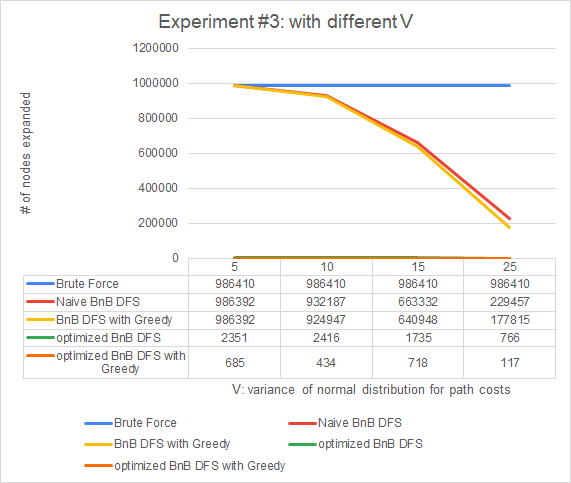


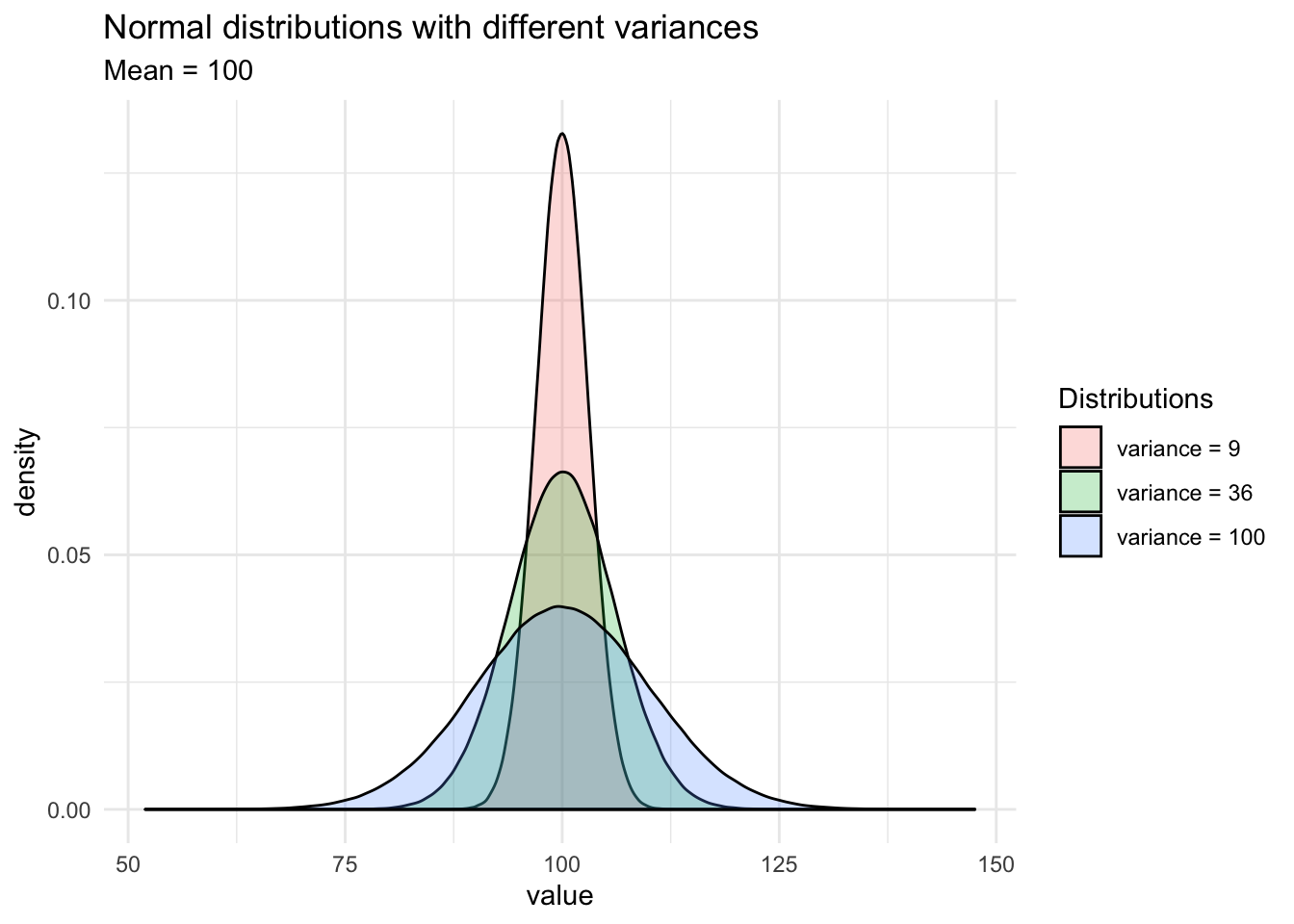
*Plot 1 & 2:*

From the two graphs shown here, we can see that, as V increases, the time elapsed and the number of nodes expanded for theFrom the third left plot graph, only the time elapsed for 4th and 5th algorithms are shown. We can see that, as N increases, the time elapsed increases a lot for each algorithm.

2nd and 3rd algorithms decreased a lot.

On the other hand, the 4th and 5th algorithms stayed as a steady line in both graphs and we could say that in this experiment, the value of variance did not affect them so much.

The reason why larger V led to a smaller time elapsed and fewer number of nodes expanded is because when V was larger, the likelihood of larger path costs increased. 

For example, from the graph below, U: mean of normal distribution = 100, the value of path costs concentrated within the range around [90, 110] when variance = 9, whereas, when variance = 100, the value of path costs ranges from 75 to 125, which means that the difference between path costs becomes bigger. 

With a larger variance, it is more likely that the pruning condition would be triggered more frequently for branch and bound algorithms. Since the higher possibility we encounter a large path cost, the more often we prune, and the more likely we visit a small path cost state, the more frequently we update the upper bound.

**Benchmark Problem Experiments Observation:**

For the 100 benchmark problems, we used the 5th algorithm: *Branch-and-Bound DFS with An Optimized Lower Bound and Greedy* to run those test files.

We tested N: the number of cities which ranges from 25, 50, 75, 100, 200, 300, 400, 600, 800, and 1000, and we found that for almost all test cases, our algorithm exceeded the time limit = 15 minutes, except for some cases, where N=25.

From the test results of benchmark problems and 3 experiments, we can conclude that for small cities number, our algorithm worked reasonably to find the optimal solution and greatly improved the performance compared to brute force DFS algorithm. However, for large cities number N = 25 or greater, our solution found so far within the time limit would not be an optimal solution, and could be far from the optimal solution as N increases.

*Analysis & Improvements:*

The reason for this problem is that our heuristic function is still considered weak for large N. To address this issue, what we can do is to define a better and stronger heuristic in hope of more frequent pruning conditions triggered during search. From the reference [here](https://mathcourses.nfshost.com/archived-courses/mat-375-001-2015-fall/lectures/lec-28-travelling-salesman-problem.pdf), it indicates that using the minimum spanning tree (MST), or combining with MST and our heuristic, sum of lengths of two shortest edges from city v, could possibly be a better lower bound choice.

References

--- B&B DFS on TSP---

<https://www.coursera.org/learn/advanced-algorithms-and-complexity>

<https://www.coursera.org/lecture/advanced-algorithms-and-complexity/tsp-branch-and-bound-RkoEK>

<https://en.wikipedia.org/wiki/Branch_and_bound#Pseudocode>

<https://artint.info/html/ArtInt_63.html>

<https://apps.dtic.mil/dtic/tr/fulltext/u2/a314598.pdf>

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

<https://artint.info/html/ArtInt_63.html>

<https://mathcourses.nfshost.com/archived-courses/mat-375-001-2015-fall/lectures/lec-28-travelling-salesman-problem.pdf>

<https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008/>

<https://towardsdatascience.com/do-my-data-follow-a-normal-distribution-fb411ae7d832>

--- SLS for MAX-SAT ---

[[2003.02981] Learning Complexity of Simulated Annealing (arxiv.org)](https://arxiv.org/abs/2003.02981)

[IncompleteAlg-SAT-Handbook-prelim.pdf (cornell.edu)](http://www.cs.cornell.edu/~sabhar/chapters/IncompleteAlg-SAT-Handbook-prelim.pdf)

[Local Search for Hard SAT Formulas: The Strength of the Polynomial Law (princeton.edu)](https://www.cs.princeton.edu/~sixuel/others/aaai16_liu.pdf)

[MAX-SAT Project, Additional Experiments (uchicago.edu)](http://people.cs.uchicago.edu/~pankratov/addl_exp)

[A Variable Neighborhood Walksat-Based Algorithm for MAX-SAT Problems (nih.gov)](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4142167/)

[MAX-SAT Project, Additional Experiments (uchicago.edu)](http://people.cs.uchicago.edu/~pankratov/addl_exp)

[A Variable Neighborhood Walksat-Based Algorithm for MAX-SAT Problems (nih.gov)](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4142167/)

[Algorithms for Maximum Satisfiability - with Applications to AI (helsinki.fi)](https://www.cs.helsinki.fi/group/coreo/aaai16-tutorial/aaai16-maxsat-tutorial.pdf)

<https://www.hindawi.com/journals/tswj/2014/798323/>

[Enhanced Walksat with Finite Learning Automata For MAX SAT (1library.net)](https://1library.net/document/q05xp6gy-enhanced-walksat-finite-learning-automata-max-sat.html)

<http://www.cs.umd.edu/~gasarch/BLOGPAPERS/max3satu.pdf>

http://web.cs.iastate.edu/~cs511/handout08/Approx\_Max3SAT.pdf