271P Project Report

THREE KILO BYTES

Dhritish Karanth | dkaranth | 35083247
Ammunje Karthik Nayak | nayakak | 16047415
Kumar Manjunatha | kmanjuna | 75485333

2020-12-17

Traveling Salesman Problem (TSP) using Branch and Bound DFS

A traveler/salesman needs to visit all the cities from a given list, where distances between all the cities are known and each city should be visited only once. The problem is to find the shortest possible route in terms of cost that he visits each city exactly once and returns to the origin city.

For example: Consider the below graph of cities and their corresponding costs.
The shortest and optimal solution is 0 → 1 → 3 → 2 → 0 and the cost is 10 + 25 + 30 + 15 = 80.

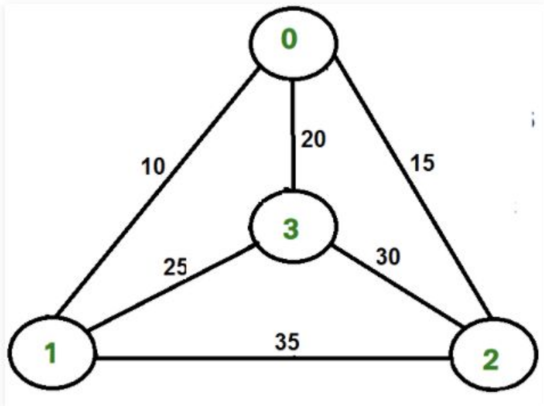

Image credit: geeksforgeeks.org

In this design document we will discuss Branch and Bound with DFS to provide a solution to the traveling salesman problem.

Branch and Bound

Branch and Bound is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search.

Branch and Bound with DFS Traversal

Traveling Salesman Problem can be formulated as a search in a state space. A state space consists of a set of states and collection of operators in a graph-like structure. The states include the problem to be solved and all partial problems that can be generated. For TSP, the graph contains vertices as states and edges (a direct path between 2 vertices) as operators. It is a systematic approach of exploration to find one or more goal nodes. In TSP, different paths from initial node to goal node will be explored to find the shortest and optimal solution path in terms of cost.

The state space tree will be explored using Depth First Search in our proposed algorithm.

Data Structures

Cost Matrix is defined as C[i][j], where C[i][j] is the distance between nodes i and j, if there is a direct path between them, and ∞ otherwise.
Node/State representation - Object to represent a node in the state space tree.
Data Members:

1. Vertex - An Integer representing the city
2. Cost - The cost for a given a node after matrix reduction
3. 2D Dimensional Reduced Matrix - The reduced matrix after computing the heuristic
4. Level - Number of cities visited so far

Priority Queue - Implements a custom comparator to compare "reduced cost" and place the minimum at the front. This queue essentially represents the "frontier", and will supply the node selected for exploration at each iteration.

Algorithm

Input: *cost_matrix*, *N (number of cities)*
Output: *Minimum Cost for the traveling salesman problem*

```
best_cost ← ∞
root ← Initialize the root node
root.cost ← calculateCost(cost_matrix,0,0,0)
priority_queue ← Initialize an empty priority queue to hold the unexpanded nodes according to their cost
priority_queue ← add root to the priority queue
while priority_queue is not empty:
        node ← get the least cost node from priority_queue
        if node.level is equal to N-1 and node.cost is lesser than best_cost then:
            best_cost ← node.cost
        if node.cost is greater than lowest_cost then:
            continue
        else
            start_vertex ← node.vertex
            for end_vertex in 0 to N-1 do:
                if node.reduced_matrix[start_vertex][end_vertex] is not ∞ then:
                    child ← Initialize a child node from its parent and calculate its cost
                    child.cost ← node.cost +
                                 node.reduced_matrix[start_vertex][end_vertex] +
                                 calculateCost(node.reduced_matrix)
                    priority_queue ← add child to the priority queue
        return best_cost

function calculateCost(parent_matrix)
        cost ← Initialize reduced cost of a given node to zero
        Row ← Initialize a row matrix to ∞
        rowReduction(parent_matrix, row)
        Column ← Initialize a column matrix to ∞
        columnReduction(parent_matrix, column)

        for i in 0 to N do:
          if Row[i]!=∞
              cost ← cost + Column[i]
```

```
        if Column[i]!=∞
            cost ← cost + Column[i]
    return cost

function rowReduction(matrix, row)
    for i in 0 to N do:
        for j in 0 to N do:
            if matrix[i][j] is less than row[j]
                row[j] ← matrix[i][j]

    for i in 0 to N do:
        for j in 0 to N do:
            if matrix[i][j] is not ∞ and row[j] is not ∞ then:
                matrix[i][j] ← matrix[i][j] - row[j]

function columnReduction(matrix, column)
    for i in 0 to N do:
        for j in 0 to N do:
            if matrix[i][j] is less than column[j]
                row[j] ← matrix[i][j]

    for i in 0 to N do:
        for j in 0 to N do:
            if matrix[i][j] is not ∞ and column[j] is not ∞ then:
                matrix[i][j] ← matrix[i][j] - column[j]
```

Explanation

This algorithm is similar to DFS except that it does not stop at the first solution, it continues after recording it as the best solution and continues to explore the state space tree and find better solutions.

Heuristic Formula:
Cost(Child) ← Cost(Parent) + Cost(Parent → Child) + ReducedMatrixCost(Child)

Note: The values of Cost(Parent) + Cost(Parent → Child) will be 0 for the root node.

We initialize a variable *best_solution* to ∞ and create a priority queue (*priority_queue*) which contains unexplored nodes according to their cost with the lowest at the front of the queue.

The algorithm starts with the root node and calculates its cost from the given input cost matrix by reducing each row and each column to have at least one zero. In order to obtain this matrix representation we reduce each element of a row with the minimum element in that row and similarly each element of a column is reduced by the minimum element in that column.

The cost of the root node will be the sum of all minimum elements of each row and each column. This root node will be added into a priority queue and we enter the loop to explore the state space tree. All the children of the minimum cost node (root in the first iteration) will be enumerated in the inner loop. Each child's *cost* will be calculated using the calculateCost method and they are added into the priority queue
The above steps will continue and the state space graph will expand in a DFS like manner. Once the leaf node is reached, it means all the cities are visited and thereby the *best_solution* will be recorded.
In the following iterations, each node's cost will be checked with the best solution, if it is greater then the node is considered as a dead end and the branch will not be explored. In other words the branch is pruned.

Otherwise the branch will be explored to find a better solution. Additionally, when the child node is added to the queue, its cost is checked against the best_solution so far, if greater the child is not added to the queue thereby pruning beforehand.

If there is a better solution obtained in one of the iterations, the *best_solution* will be updated and further pruning may be achieved. This loop continues until all the nodes (excluding the pruned branches) are explored and the minimum cost will be returned in the end.

The heuristic used to calculate the cost of a given node is admissible as the value acts as a lower bound to each unexplored subproblem and therefore never overestimates the best solution in a subproblem.

Execution Analysis

Given a TSP problem for N cities, in a brute force solution there are (N-1)! tours. Only for small size problems, a brute force evaluation of all permutations will be possible. The algorithm that we have implemented, Branch and Bound, has a bounding functionality which prunes unwanted or non-optimal sub branches.

After analyzing the results for the given sample problems and benchmark problems, below are our observations.

For a given problem set `tsp-problem-15-15-100-5-1.txt`, the code generates 10907 nodes in the state space tree and prunes 9552 nodes and finds the optimal cost 1416.70 with the default JVM configuration. The same configuration cannot be achieved using brute force approach because there will be 14! tours that have to be explored.

The code works upto N = 23 well within the time limit of 15 minutes, the minimal cost obtained is the optimal cost and some of the results are attached in the Appendix.
But beyond N = 24, the code fails due to out of memory or heap space is completely used and does not complete within the time limit with increased heap memory. As each node needs a reduced matrix representation that is obtained from the parent node and matrix reduction process requires row and column matrix. So the memory is completely utilized before the leaf node is reached because the branching factor is too high. Beyond that, the code times out and the best possible solution found so far is returned.

Potential Improvements

Since a lot of memory is required to represent the reduced cost matrix at each node, we need to optimize the memory consumption as this will be very high for higher values (N > 23).

Complexity Analysis

Space complexity: O(bm)
Time complexity: $O(b^m)$
Where ‘b’ is the branching factor and ‘m’ is the number of cities.

MaxSAT using Stochastic Local Search

Introduction

The objective of the MaxSAT problem (Maximum Satisfiability) is to find values for certain boolean variables such that the number of clauses that are true in a given boolean expression is maximum. The boolean expression is provided as a Conjunctive Normal Form (CNF), which consists of several clauses joined by AND operator. Each clause consists of several variables joined by an OR operator.

Approach

The general idea behind local search is to start off with a randomly generated solution, and progressively search its neighborhood for better solutions. To avoid getting stuck in local maxima, we also incorporate the idea of random restarts, so that the solution space is somewhat evenly explored. After a predefined number of cycles or time elapsed, search will be restarted with a new random solution.

In this application, we randomly generate an assignment of values to each variable as a starting point for the local search, and explore its neighborhood for better solutions. The neighborhood in this case is the set of assignments that differ by one (i.e. only one variable with a different value).

Additionally, we use multi-threading to make better use of multiple CPU cores, taking advantage of the fact that each instance of a local search operation is essentially independent, and does not directly depend on other instances.

Data structures

clauses: 2D array that stores the input CNF clauses.
We associate each clause with an index number.

occurrences: A lookup table, mapping symbols a list of clauses where they are found. In Java, this would be a HashMap<Integer, List<Integer>>.
We use this structure to update the current state after a symbol is modified.

symbols: Boolean array that stores value given to each literal.

Unsatisfied: Integer array that lists tracks clauses that are newly unsatisfied if a literal is changed.
Satisfied: Integer array that tracks clauses that are newly satisfied if a literal is changed.

flip_history: Array that records the iteration number when the given symbol was flipped.

Algorithm

Input: *clauses*, *p*, *dp*, *max_flips*, *max_tries*
Note: *p* and *dp* are values in range [0, 1] to be determined experimentally. By default they are set to 0.5 and 0.05 respectively.

Output: the values for symbols that result in the greatest number of satisfied clauses.

```
for i from 1 to max_tries do:
        current_state ← a randomly generated assignment of values
        best_state ← current_state

        for j from 1 to max_flips do:
                if best_state satisfies all clauses then:
                        return best_state
                Update Satisfied[], Unsatisfied[]
```

selected_clause ← a randomly selected clause from *clauses* that is unsatisfied with *current_state*
flip_symbol ← the symbol to be flipped, as determined by Heuristic(*selected_clause*, *p*, *dp*)
current_state ← *current_state* after flipping *flip_symbol*
if *current_state* satisfies more clauses than *best_state* then:
best_state ← *current_state*
return *best_state*

function Heuristic(*clause, p, dp*):
with probability *dp* do:
return the least recently flipped symbol in *clause*
else:
sorted_symbols ← the symbols in *clause*, with each symbol *x*'s position decided by Score(*x*). The symbol with the highest Score will be the first element in the list. In case of ties, the least recently flipped variable goes first.
first_best, *second_best* ← first and second element in *sorted_symbols*
if *first_best* is not the most recently flipped symbol then:
return *first_best*
else:
with probability *p* do:
return *second_best*
else:
return *first_best*

function Score(*symbol*):
return Satisfied[*symbol*] - Unsatisfied[*symbol*]

Explanation

Our algorithm is a variant of the popular WalkSAT method used to solve maxSAT problems. However, instead of a heuristic that selects the next move based on a score (as is done in plain versions of WalkSAT), we use a heuristic called Novelty++ that aims to avoid some of the issues that affect the simpler heuristic.

The algorithm operates within two nested loops. The outer one, controls random restarts, by initializing a random starting point as the current state, which consists of each variable/symbol in the CNF being assigned a random boolean value. This approach of periodically starting afresh gets the algorithm out of dead-ends / local maxima.

In the inner loop, we first check if the current state is the solution we are looking for (i.e. it satisfies all clauses in the CNF). If yes, then it is returned. Else, we search the neighborhood of the current configuration for better solutions. We will randomly select a clause that is currently unsatisfied by the current state. The heuristic function will select a variable/symbol for flipping. If the new configuration (after flipping) has more number of satisfied clauses, then it is saved.

The heuristic function's job is to decide which symbol is to be flipped, given a clause. With a probability *dp*, it selects the least recently flipped symbol in the clause. With a probability 1-*dp*, it selects the symbol in a slightly longer way. First, it orders the symbols in the clause according to their score metric, which is a measure of how 'good' the flipping of the symbol would be for our goal. In case of a tie, the least recently flipped symbol is given precedence.

Once we have an ordering of the symbols, the best and the second-best symbols are selected. If the best symbol happened to be also the most recently flipped one, then it is returned with a probability of 1-*p*. The second best symbol is returned with a probability of p. If the best symbol was not the most recently flipped variable, then it is returned immediately.

The score of a symbol is an approximation of the amount of useful "change" flipping a symbol would cause. It is calculated as the difference between the number of clauses that would be newly satisfied, and the number of clauses that would be newly unsatisfied, if the symbol were to be flipped.

The heuristic used to select the symbol to be flipped deserves additional explanation. In traditional WalkSAT, the symbol selected for the greedy move is only dependent on the score. This makes it possible for the same symbol to get repeatedly flipped, reducing the diversity of symbols explored. Also, it is likely that the most recently flipped symbol was useful (otherwise it wouldn't have been flipped). Flipping it again potentially cancels out the gains made previously. To counter this, the improved version of the heuristic (called Novelty) selects the two best candidates, and chooses between them depending on p, ensuring that the same symbol is not repeatedly flipped.

While Novelty represents an improvement, it is still possible for the algorithm to get stuck in a loop, which prevents potential solutions from being explored. Although the situation is remedied by the random restarts, this depends on the choice of *max_tries*, which cannot be determined before running the search.

Novelty++ tries to address this scenario (of local maxima). In such situations, it is possible that there is a clause c that is repeatedly made unsatisfiable. The addition of diversity probability (dp) in Novelty++ allows it to flip all symbols in c one-by-one (since it selects symbols based on recency of flips). This enables Novelty++ to flip symbols that would have been left untouched in previous heuristics.

Execution Analysis

The first key observation is that by its very nature, local search is well suited to parallelization, much more so than other search strategies. Due to the relative independence of each search instance, each instance can be potentially executed by other threads or processors. Such parallelism is more difficult to achieve in other algorithms because the result computed by one search instance is needed for future steps, effectively eliminating any possibility of parallelism. We make use of this fact by implementing local search for MaxSAT as a multi-threaded program, allowing it to take full advantage of multi-core systems.

The second key observation is that local search arrives at a 'good-enough' solution very rapidly, and any progress henceforth is incremental. This is understandable, since the random starting locations (in the solution space) ensure that at least one starting location for the search is somewhat close to the optimal solution.

For example, consider the solution computed for the problem `max-sat-problem-25000-3-108749-1` in Appendix. The program arrived at the best value of 106223 (out of 108749) in 61.16s, even though the time limit was much higher. This shows that even in scenarios where computing power is not available, or if exact solutions are not required, local search can be of use.

Potential Improvements

The quality of results is highly dependent on the randomness of the initial starting state. The more random it is, more the likelihood that the solution space will be searched evenly. For this implementation, we have used Java's default random number generation library. More advanced random number generators could be used, which will likely result in improved results.

The idea implemented in this project looks at the contribution made towards the end goal that is possible by flipping a given variable. This is used to compute the heuristic, i.e. selection of a variable to flip. More advanced algorithms have been published, which seek to transform the problem into a simpler, but equivalent one, using inference rules. One such paper is [3] in the Reference section. An implementation of such an algorithm would likely lead to further improvements in the run time or the quality of results.

Another area for improving the implementation is to enable data sharing between threads. In the current implementation, each thread has a full copy of the input data, due to reduced code complexity. This results in high memory usage, especially for large problem sizes. For example, for a problem with 170,000 clauses, memory usage is about 1250MB. This can be reduced by deduplicating the data and ensuring efficient access to it by each thread.

Complexity analysis

Space: O(number_of_clauses * number_of_symbols)

Time: O(max_tries * max_flips * max(number_of_clauses, number_of_symbols))

Since the algorithm runs for the full extent of the two nested loops in the worst case scenario, max_tries * max_flips term is expected.

The cost of an iteration of the innermost loop is determined by two operations: the symbol selection by the heuristic (with time complexity O(number_of_symbols)), and updating the state after the flip has been applied (O(number_of_clauses)).

Appendix

Data for BnB-DFS

TSP Problem	*Tour cost*
tsp-problem-100-100-100-25-1.txt	8820.03076
tsp-problem-100-100-100-5-1.txt	3664.608276
tsp-problem-100-1000-100-25-1.txt	8761.595526
tsp-problem-100-1000-100-5-1.txt	4087.183055
tsp-problem-100-2000-100-25-1.txt	8779.386128
tsp-problem-100-2000-100-5-1.txt	3849.399564
tsp-problem-100-4000-100-25-1.txt	8765.42494
tsp-problem-100-4000-100-5-1.txt	4102.115996
tsp-problem-100-500-100-25-1.txt	8910.272167
tsp-problem-100-500-100-5-1.txt	21720.44991
tsp-problem-1000-10000-100-25-1.txt	84195.90758
tsp-problem-1000-10000-100-5-1.txt	20528.66029
tsp-problem-1000-100000-100-25-1.txt	84022.2536
tsp-problem-1000-100000-100-5-1.txt	20693.06267
tsp-problem-1000-200000-100-25-1.txt	84128.62189
tsp-problem-1000-200000-100-5-1.txt	21151.12393
tsp-problem-1000-400000-100-25-1.txt	83964.96698
tsp-problem-1000-400000-100-5-1.txt	22629.84069

tsp-problem-1000-50000-100-25-1.txt	84301.61378
tsp-problem-1000-50000-100-5-1.txt	6564.362909
tsp-problem-200-16000-100-25-1.txt	17380.83732
tsp-problem-200-16000-100-5-1.txt	7727.188367
tsp-problem-200-2000-100-25-1.txt	17377.91423
tsp-problem-200-2000-100-5-1.txt	5982.798245
tsp-problem-200-400-100-25-1.txt	17570.76007
tsp-problem-200-400-100-5-1.txt	6623.27383
tsp-problem-200-4000-100-25-1.txt	17363.13532
tsp-problem-200-4000-100-5-1.txt	6748.020114
tsp-problem-200-8000-100-25-1.txt	17368.53219
tsp-problem-200-8000-100-5-1.txt	1373.273428
tsp-problem-25-125-100-25-1.txt	2277.155208
tsp-problem-25-125-100-5-1.txt	1315.387936
tsp-problem-25-250-100-25-1.txt	2285.362022
tsp-problem-25-250-100-5-1.txt	1331.700655
tsp-problem-25-31-100-25-1.txt	2275.200681
tsp-problem-25-31-100-5-1.txt	1383.407745
tsp-problem-25-6-100-25-1.txt	2447.578941
tsp-problem-25-6-100-5-1.txt	1490.940664
tsp-problem-25-62-100-25-1.txt	2312.674547
tsp-problem-25-62-100-5-1.txt	9471.203164
tsp-problem-300-18000-100-25-1.txt	25837.57096
tsp-problem-300-18000-100-5-1.txt	9024.45646
tsp-problem-300-36000-100-25-1.txt	25855.77269
tsp-problem-300-36000-100-5-1.txt	9221.323097
tsp-problem-300-4500-100-25-1.txt	25894.49355
tsp-problem-300-4500-100-5-1.txt	9681.815484
tsp-problem-300-900-100-25-1.txt	25644.43765
tsp-problem-300-900-100-5-1.txt	9279.436185
tsp-problem-300-9000-100-25-1.txt	25881.02169
tsp-problem-300-9000-100-5-1.txt	11341.12414
tsp-problem-400-1600-100-25-1.txt	34312.29891
tsp-problem-400-1600-100-5-1.txt	10723.04152
tsp-problem-400-16000-100-25-1.txt	34288.99433
tsp-problem-400-16000-100-5-1.txt	11371.11216
tsp-problem-400-32000-100-25-1.txt	34152.65026
tsp-problem-400-32000-100-5-1.txt	10681.65066
tsp-problem-400-64000-100-25-1.txt	34141.77114
tsp-problem-400-64000-100-5-1.txt	11555.38193
tsp-problem-400-8000-100-25-1.txt	33909.56948
tsp-problem-400-8000-100-5-1.txt	2237.082572
tsp-problem-50-1000-100-25-1.txt	4477.035551
tsp-problem-50-1000-100-5-1.txt	2884.204403
tsp-problem-50-125-100-25-1.txt	4415.069846
tsp-problem-50-125-100-5-1.txt	1728.669572
tsp-problem-50-25-100-25-1.txt	4624.908669
tsp-problem-50-25-100-5-1.txt	2271.28388

tsp-problem-50-250-100-25-1.txt	4496.226455
tsp-problem-50-250-100-5-1.txt	2546.955961
tsp-problem-50-500-100-25-1.txt	4504.837897
tsp-problem-50-500-100-5-1.txt	14737.84574
tsp-problem-600-144000-100-25-1.txt	51027.93035
tsp-problem-600-144000-100-5-1.txt	14307.34936
tsp-problem-600-18000-100-25-1.txt	50900.3716
tsp-problem-600-18000-100-5-1.txt	19389.36055
tsp-problem-600-3600-100-25-1.txt	50173.86874
tsp-problem-600-3600-100-5-1.txt	14686.38964
tsp-problem-600-36000-100-25-1.txt	50884.45973
tsp-problem-600-36000-100-5-1.txt	15234.96176
tsp-problem-600-72000-100-25-1.txt	50834.99287
tsp-problem-600-72000-100-5-1.txt	3307.748302
tsp-problem-75-1125-100-25-1.txt	6672.072696
tsp-problem-75-1125-100-5-1.txt	3319.386763
tsp-problem-75-2250-100-25-1.txt	6661.618363
tsp-problem-75-2250-100-5-1.txt	3276.837783
tsp-problem-75-281-100-25-1.txt	6681.909302
tsp-problem-75-281-100-5-1.txt	3610.81197
tsp-problem-75-56-100-25-1.txt	6841.136349
tsp-problem-75-56-100-5-1.txt	2927.142824
tsp-problem-75-562-100-25-1.txt	6658.014915
tsp-problem-75-562-100-5-1.txt	17900.02192
tsp-problem-800-128000-100-25-1.txt	67576.87937
tsp-problem-800-128000-100-5-1.txt	18507.74926
tsp-problem-800-256000-100-25-1.txt	67509.23995
tsp-problem-800-256000-100-5-1.txt	18187.55541
tsp-problem-800-32000-100-25-1.txt	67732.30381
tsp-problem-800-32000-100-5-1.txt	14669.85255
tsp-problem-800-6400-100-25-1.txt	67369.89355
tsp-problem-800-6400-100-5-1.txt	18482.09138
tsp-problem-800-64000-100-25-1.txt	67586.11672

Data for MaxSAT

Problem name	***Number of satisfied clauses***	***Time to obtain best solution (* indicates complete solution)***
max-sat-problem-100-3-434-1.txt	433	0.081s
max-sat-problem-100-3-470-1.txt	469	0.079s
max-sat-problem-100-3-505-1.txt	505	0.013s*
max-sat-problem-100-3-540-1.txt	536	0.053s

max-sat-problem-100-3-575-1.txt	570	0.117s
max-sat-problem-100-3-610-1.txt	601	0.057s
max-sat-problem-100-3-645-1.txt	634	0.09s
max-sat-problem-100-3-680-1.txt	667	0.076s
max-sat-problem-100-3-715-1.txt	702	0.217s
max-sat-problem-100-3-750-1.txt	733	0.072s
max-sat-problem-1000-3-4350-1.txt	4344	26.039s
max-sat-problem-1000-3-4700-1.txt	4688	33.494s
max-sat-problem-1000-3-5050-1.txt	5022	44.482s
max-sat-problem-1000-3-5400-1.txt	5354	92.254s
max-sat-problem-1000-3-5750-1.txt	5684	10.428s
max-sat-problem-1000-3-6100-1.txt	6021	56.472s
max-sat-problem-1000-3-6450-1.txt	6337	10.144s
max-sat-problem-1000-3-6800-1.txt	6668	1.343s
max-sat-problem-1000-3-7150-1.txt	6993	7.339s
max-sat-problem-1000-3-7500-1.txt	7311	5.972s
max-sat-problem-10000-3-43500-1.txt	43145	117.099s
max-sat-problem-10000-3-47000-1.txt	46448	92.477s
max-sat-problem-10000-3-50500-1.txt	49710	59.729s
max-sat-problem-10000-3-54000-1.txt	52895	29.734s
max-sat-problem-10000-3-57500-1.txt	56205	96.462s
max-sat-problem-10000-3-61000-1.txt	59413	99.998s
max-sat-problem-10000-3-64500-1.txt	62573	38.398s
max-sat-problem-10000-3-68000-1.txt	65842	37.786s
max-sat-problem-10000-3-71500-1.txt	69007	94.085s
max-sat-problem-10000-3-75000-1.txt	72165	94.199s
max-sat-problem-100000-3-434999-1.txt	400493	264.73s
max-sat-problem-100000-3-470000-1.txt	431726	291.277s
max-sat-problem-100000-3-505000-1.txt	462420	301.035s
max-sat-problem-100000-3-540000-1.txt	492846	302.349s
max-sat-problem-100000-3-575000-1.txt	522202	301.731s
max-sat-problem-100000-3-610000-1.txt	552623	302.917s
max-sat-problem-100000-3-645000-1.txt	583941	300.82s
max-sat-problem-100000-3-680000-1.txt	612686	304.511s
max-sat-problem-100000-3-715000-1.txt	644555	300.456s
max-sat-problem-100000-3-750000-1.txt	673423	301.645s
max-sat-problem-200-3-1010-1.txt	1004	0.169s
max-sat-problem-200-3-1080-1.txt	1073	0.093s
max-sat-problem-200-3-1150-1.txt	1141	0.099s
max-sat-problem-200-3-1220-1.txt	1206	0.187s
max-sat-problem-200-3-1290-1.txt	1273	0.367s
max-sat-problem-200-3-1360-1.txt	1335	2.635s
max-sat-problem-200-3-1430-1.txt	1406	2.753s
max-sat-problem-200-3-1500-1.txt	1469	21.912s

max-sat-problem-200-3-869-1.txt	869	0.128s*
max-sat-problem-200-3-940-1.txt	938	0.67s
max-sat-problem-2500-3-10875-1.txt	10847	25.137s
max-sat-problem-2500-3-11750-1.txt	11703	22.061s
max-sat-problem-2500-3-12625-1.txt	12528	48.121s
max-sat-problem-2500-3-13500-1.txt	13343	25.927s
max-sat-problem-2500-3-14375-1.txt	14169	108.755s
max-sat-problem-2500-3-15250-1.txt	14993	49.179s
max-sat-problem-2500-3-16125-1.txt	15805	55.606s
max-sat-problem-2500-3-17000-1.txt	16599	24.236s
max-sat-problem-2500-3-17875-1.txt	17395	63.533s
max-sat-problem-2500-3-18750-1.txt	18218	32.177s
max-sat-problem-25000-3-108749-1.txt	106223	61.159s
max-sat-problem-25000-3-117500-1.txt	114112	100.82s
max-sat-problem-25000-3-126250-1.txt	122101	70.144s
max-sat-problem-25000-3-135000-1.txt	130035	73.297s
max-sat-problem-25000-3-143750-1.txt	138041	81.204s
max-sat-problem-25000-3-152500-1.txt	145922	64.652s
max-sat-problem-25000-3-161250-1.txt	153926	66.056s
max-sat-problem-25000-3-170000-1.txt	161802	87.099s
max-sat-problem-25000-3-178750-1.txt	169697	93.581s
max-sat-problem-25000-3-187500-1.txt	177683	90.319s
max-sat-problem-500-3-2175-1.txt	2174	13.901s
max-sat-problem-500-3-2350-1.txt	2344	2.047s
max-sat-problem-500-3-2525-1.txt	2514	1.561s
max-sat-problem-500-3-2700-1.txt	2683	19.672s
max-sat-problem-500-3-2875-1.txt	2849	12.144s
max-sat-problem-500-3-3050-1.txt	3014	5.885s
max-sat-problem-500-3-3225-1.txt	3178	16.906s
max-sat-problem-500-3-3400-1.txt	3339	22.447s
max-sat-problem-500-3-3575-1.txt	3507	3.429s
max-sat-problem-500-3-3750-1.txt	3661	25.296s
max-sat-problem-5000-3-21750-1.txt	21647	88.743s
max-sat-problem-5000-3-23500-1.txt	23325	31.559s
max-sat-problem-5000-3-25250-1.txt	24980	97.450s
max-sat-problem-5000-3-27000-1.txt	26583	34.149s
max-sat-problem-5000-3-28750-1.txt	28252	33.725s
max-sat-problem-5000-3-30500-1.txt	29871	93.504s
max-sat-problem-5000-3-32250-1.txt	31473	79.053s
max-sat-problem-5000-3-34000-1.txt	33075	39.933s
max-sat-problem-5000-3-35750-1.txt	34678	100.645s
max-sat-problem-5000-3-37500-1.txt	36282	73.715s
max-sat-problem-50000-3-217499-1.txt	206612	141.58s
max-sat-problem-50000-3-235000-1.txt	222249	221.488s

max-sat-problem-50000-3-252500-1.txt	238015	94.559s
max-sat-problem-50000-3-270000-1.txt	253777	229.152s
max-sat-problem-50000-3-287500-1.txt	269209	236.575s
max-sat-problem-50000-3-305000-1.txt	285149	147.958s
max-sat-problem-50000-3-322500-1.txt	300787	171.915s
max-sat-problem-50000-3-340000-1.txt	316327	157.579s
max-sat-problem-50000-3-357500-1.txt	331997	163.135s
max-sat-problem-50000-3-375000-1.txt	347683	168.937s

Reference

maxSAT using SLS

[1] *Noise Strategies for Improving Local Search* by Bart Selman, Henry A. Kautz, and Bram Cohen (1994): https://www.aaai.org/Papers/AAAI/1994/AAAI94-051.pdf

[2] *Diversification and determinism in local search for satisfiability* by Chu Min Li and Wen Qi Huang (2014): https://www.researchgate.net/publication/220944441_Diversification_and_Determinism_in_Local_Search_for_Satisfiability

[3] *Inference Rules in Local Search for Max-SAT* by Andre Abrame and Djamal Habet (2012): https://ieeexplore.ieee.org/document/6495048

TSP using BnB DFS

https://www.techiedelight.com/travelling-salesman-problem-using-branch-and-bound/

https://www2.seas.gwu.edu/~bell/csci212/Branch_and_Bound.pdf

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

https://www.cs.ubc.ca/~mack/CS322/lectures/3-CSP1.pdf

Branch-and-Bound Search Algorithms and their Computational Complexity by Weixiong Zhang (1996): https://apps.dtic.mil/dtic/tr/fulltext/u2/a314598.pdf