271P Project Draft Design

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

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

[**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

**DESCRIBING PROBLEM:**

Consider an n-variable CNF formula F with clauses C1, C2,...,Cm. For a truth assignment

σ ∈ {0, 1}n and i ∈ {1, 2, . . . , m}, let Ui(σ) be a function that is 0 if Ci is satisfied by σ, and 1 otherwise.

**STATE SPACE**: assignment of true/false(1/0) to the variables(literals) involved in the CNF. Here σ defines a particular state.

**OBJECTIVE FUNCTION:**

the maxSAT problem for F can be written as the following optimization problem over σ ∈ {0, 1}n:

minimize N(σ) =

subject to Ui(σ) = 0 ∀i ∈ {1, 2, . . . , m}

N(σ) ≥ 0 and equals 0 if and only if all clauses of F are satisfied.

Hence the objective functions is **N(σ) =**

**NEIGHBORHOOD RELATION:**

To explain it we first need to define the break-count.

Break-count: the number of currently satisfied clauses that become unsatisfied by flipping a variable value.

We improve the states by following steps

1. While choosing the variable to flip from an unsatisfied CNF, if there exists a variable which when flipped does not make any more CNF false. Then we choose this variable
2. If no such variable exists then we pick a random variable with probability Pnoise and flip it, this step helps us move out of the local minimas.
3. With 1-Pnoise we pick a variable such that flipping it causes least break-count.
4. When we exhaust the max-flips we start again with a fresh initial randomly generated truth assignment for F, this helps us explore larger state space.

1. Design the algorithm(s) and data structures

To Solve this problem , we maintain a variable holding the best assignment encountered yet in all the attempts during the WalkSAT iteration that satisfied the most clauses as of yet.

When we exhaust all the flips without finding a solution then we continue with a fresh initial assignment till the time runs out, after which we report the best solution encountered yet.

We do not need any complex data structures to implement this algorithm.

2. Present the algorithms using pseudo-code

**PSEUDO-CODE:**

**function :** maxWalkSAT

**Input** : A CNF formula F

**Parameters** : Integers max\_duration in seconds, max-tries; 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 //σ satisfies F **then**

α ← σ

**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 in σ

number\_clauses\_satisfied ← count number of clauses satisfied (σ)

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

α ← σ

**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**

Satisfied\_count ← 0

**for** clause in CNF **do**:

**If** clause is satisfied by σ **then**:

Satisfied\_count ← Satisfied\_count + 1

Flip x in σ

**for** clause in CNF **do**:

**If** clause is satisfied by σ **then**:

Satisfied\_count ← Satisfied\_count - 1

**return** max(Satisfied\_count, 0)

**end**

3. Explain/describe the (pseudo-code of the) algorithms

**EXPLANATION PSEUDO-CODE:**

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 yes 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’ve found so far with the minimum Objective function value and maximum number of clauses satisfied.

4. The assessment of the time/space complexity of our algorithms

**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.**

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

1. Design the algorithm(s) and data structures

**PROBLEM DESCRIPTION:**

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

**STATE SPACE:**

(a partial assignment of values to variables) basically is a route that has visited some cities (not visiting every city) just once.

**INPUT DATA STRUCTURE:**

A complete weighted undirected graph representing the cities and path cost between them (if it is a non-complete graph, assigning an infinity large number to the non-existent path)

**EXPLANATION of INPUT DATA STRUCTURE:**

Use an adjacency matrix (or an adjacency list) to represent the graph since it is a complete graph. (E.g. if there exists five cities, there will be a 5x5 matrix(or vector<vector<int>> graph in C++) representing the graph. For each i, j, graph[i][j] means the path cost from i to j).

**OUTPUT:**

The minimum cost of a possible route that visits each city exactly once and returns to the origin city

2. Present the algorithms using pseudo-code

**PSEUDO-CODE (Naive algorithm with weak heuristic)**

1. The current path cost of visited cities is the 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. (bound(*Ni*): the current path cost of visited cities)
   2. Else, store *Ni* on the queue.

**PSEUDO-CODE (optimized heuristic)**

1. Calculate MST (compute Minimum Spanning Tree using Kruskal’s or Prim’s algorithm) as our heuristic function.
2. Using a heuristic, find a solution *xh* to the optimization problem. Store its value, *B* = *f*(*xh*). (Initially, set *B* to 2\*MST.) *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:

1. Take a node *N* off the queue.

2. 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*).

3. 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. (bound(*Ni*): MST of unvisited cities)
2. Else, store *Ni* on the queue.

3. Explain/describe the (pseudo-code of the) algorithms

**CODE EXPLANATION (Naive algorithm with weak heuristic):**

Initially, we assign an infinity value to upper bound (B) denoting the best solution we have found so far.

Then start doing depth-first search, when reaching a goal state, we compare the current solution with B, the best solution found so far and update B if the current solution is better than B. (Since B is infinite at the beginning, B will be the path cost of the first solution we have found during searching.)

When expanding the vertices (cities), if the current path cost (partial solution) is already larger than B, the best solution found so far, we prune the vertex and immediately stop expanding it. For this naive algorithm, the heuristic function is basically the current path cost of visited vertices.

When the process is finished, B, the best solution so far will be the optimal solution.

**CODE EXPLANATION (Optimized algorithm using Minimum Spanning Tree):**

Initially, we assign 2\*MST to upper bound (B) denoting the best solution we have found so far.

Then start doing depth-first search, when reaching a goal state, we compare the current solution with B, the best solution found so far and update B if the current solution is better than B.

When expanding the vertices (cities), if the heuristic function of the vertex (partial solution) is already larger than B, the best solution found so far, we prune the vertex and immediately stop expanding it. For this optimized algorithm, the heuristic function is to compute the MST (Minimum Spanning Tree) of the nodes that have not been visited as the lower bound.

When the process is finished, B, the best solution so far will be the optimal solution.

4. The assessment of the time/space complexity of our algorithms

**ANALYSIS (Naive algorithm with weak heuristic)**

Time complexity: O(b^m), where b is the number of cities and m is the maximum depth of the tree. Since, at worst case, the pruning condition will not be triggered during searching .

Space complexity: to store the current path has been traversed is O(N), and space complexity for storing the graph as an adjacency list is O(E). Hence, total space complexity is O(N+E) , where N is the number of cities and E is the number of edges.

**ANALYSIS (Optimized algorithm using Minimum Spanning Tree):**

Time complexity: O(b^m), where b is the number of cities and m is the maximum depth of the tree. Since, at worst case, the pruning condition will not be triggered during searching.

Space complexity: to store the current path has been traversed is O(N), and space complexity for storing the graph as an adjacency list is O(E). Hence, total space complexity is O(N+E) , where N is the number of cities and E is the number of edges.

Reference:

---DFS B&B 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/>

--- 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/>