LECTURE 12

Ishaan Lal

April 15, 2025

1 Introduction

Last week, we introduced the Traveling Salesperson Problem and looked at some *constructive heuristics* to obtain an approximate solution for the problem, including Nearest Neighbors, Nearest Insertion, Farthest Insertion, and the Savings Heuristic. Constructive heuristics start with nothing and iteratively build up a partial solution.

This week, we'll turn our attention to *improvement heuristics* where we will start with *any* solution and try to find a better solution.

2 Local Search

The first technique we will look at is called **Local Search**, and is the most straightforward implementation of an improvement heuristic.
The idea is to construct a graph G, called a **neighborhood graph** where:

- Each vertex will represent a unique solution to TSP
- An edge will exist between two vertices (two solutions to TSP) if we can transform one into the other by a simple operation.

Local Search will construct this graph, then start at any solution node v and attempt to reach a better one by exploring the neighborhood of v (the vertices adjacent to it). And then iterate on this procedure, continuously finding "better" neighbors.

2.1 Terminating Local Search

In its current formulation, Local Search will only terminate once we find the "best" solution (as there would be no "better" neighbor). Note that $|V| = n!$, as there are $n!$ feasible solutions to TSP, so the runtime of Local Search is currently $\Omega(n!)$.

As a result, in practice, it is often helpful to introduce a bound or stopping condition. Some options include:

- **Time Bound**: Stop the search if it is taking too long
- **Step Bound**: Stop the search after some number of steps. Problem-specific knowledge can be useful here.
- **Improvement Bound**: Stop the search if we have not improved our solution enough (marginal improvement is very small).

2.2 Back to TSP

While construction of a neighborhood graph is visually helpful in understanding Local Search, it may be over-formalized, and we can instead simulate a graph search.

With TSP, Local Search is very natural. We will start with any tour, and continuously try to improve it into a cheaper tour. For now, the biggest question is: *"What operations are reasonable to improve our tour"* or equivalently, *"What is a reasonable 'neighbor relation' on all tours"*.
We'll keep things simple by utilizing a simple operation to transform one tour into another tour...

2.3 2-Opt Heuristic

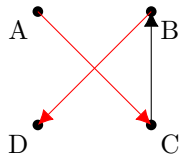
We will say that TSP tours T and T' are **2-adjacent** if we can transform one into the other by deleting two edges and adding two edges.

And we will say that TSP tour T is **2-optimal** if there is no cheaper tour adjacent to T

We formalize the process of "transformation" by the following algorithm:

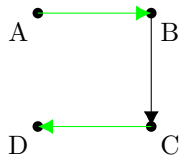
```
swap(T, i, j) = T[1:i-1] + rev(T[i:j]) + T[i+1:j]
```

where T is a tour, i and j are indices, and `rev` is a reversal function.
Consider the following examples.



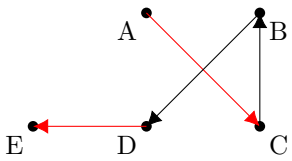
Suppose we started with the tour $A \to C \to B \to D$, and we called `swap([A, C, B, D], 2, 3)`. This would yield the following tour:

```
swap([A, C, B, D], 2, 3) = [A] + rev([C, B]) + [D] = [A, B, C, D]
```

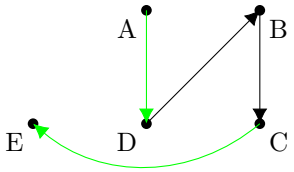


With this operation, we have removed two edges ($A \to C$ and $B \to D$), and introduced two edges ($A \to B$ and $C \to D$), and all other edges have stayed intact, though we may traverse them in a different direction.

Notice that the indices i and j **do not** need to be consecutive. The following example demonstrates this:




```
swap([A, C, B, D, E], 2, 4) = [A] + rev([C, B, D]), [E] = [A, D, B, C, E]
```

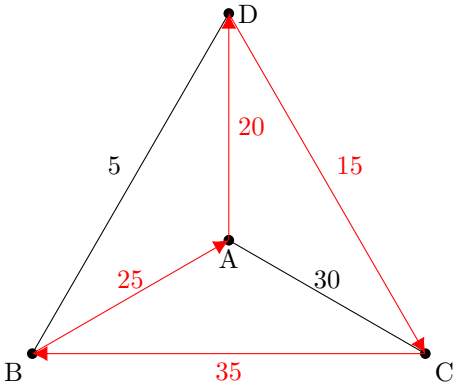


Again, only two edges are removed, only two edges are added, and the remaining remain but reverse direction.

With this helpful operation, we formalize the 2-Opt Heuristic:

```
2-opt(T):
    until cost(T) does not decrease:
        for each pair of indices i < j:
            if cost(swap(T, i, j)) < cost(T):
                let T = swap(T, i, j)
```

Now, for a simulation of 2-opt. We will begin with the following graph and tour $[A, D, C, B]$:

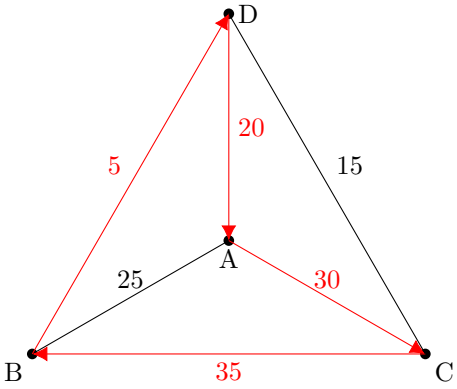


The initial tour has cost 95. Following the procedure, we would next consider every pair of indices $i < j$ until we find one where the `swap` will improve our cost. It turns out that $i = 1, j = 2$ will do just that:

$$\texttt{swap([A, D, C, B], 1, 2)} = \texttt{rev([A, D]) + [C, B]} = \texttt{[D, A, C, B]}$$

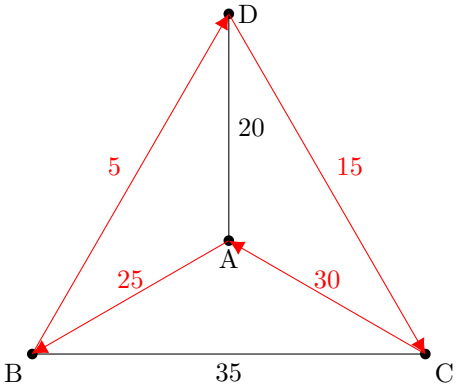
$$\texttt{cost([D, A, C, B]) = 90 < 95}$$

Since the cost after the swap operation is less than the current cost, we will perform the swap operation and obtain the new tour:



Our current tour cost is 90. We would now again consider every pair of indices i, j until we find one that reduces cost after a swap. This occurs when $i = 2, j = 3$:

```
swap([D, A, C, B], 2, 3) = [D] + rev([A, C]) + [B] = [D, C, A, B]
cost([D, C, A, B]) = 15 + 30 + 25 + 5 = 75 < 90
```



And the process would continue.

2.3.1 Generalizing 2-Opt and Analysis

It turns out that we can easily generalize 2-opt to 3-opt, 4-opt, or even k-opt for $k \in \mathbb{Z}^+$. These involve more indices and more swapping/rearranging.

The **Lin-Kernighan (LK) Heuristic** suggests to start with k-opt for $k = 2$, then dynamically increase or decrease k over time based on several criteria. This ends up being one of the most effective TSP heuristics.

When we compare 2-opt, 3-opt and LK to the constructive heuristics from last class (NN, NI, FI, Savings), in practice, 2-opt, 3-opt and LK tend to produce better solutions. Their runtime is perhaps comparable to NI and FI, and is a bit worse than NN and Savings.

3 Visiting SAT One More Time

Even though SAT is not an optimization problem, we can still try to solve it with local search. We will say that a "solution" is any truth assignment, even if it isn't satisfying. We will start with a solution, and continuously tweak/improve it to try to find a satisfying assignment. To use Local Search, we need a reasonable "neighbor relation". What could this be? What is a simple operation to transform one assignment into another? Simply just **flip** the truth value/assignment of a single variable! But naturally, the follow up question is: *"Which variable should we flip?"*

3.1 Greedy SAT

As a first attempt, let us just be **greedy**. Flip the variable that **maximizes** the number of clauses that **become satisfied**. If a termination criterion is desired, it makes most sense to use a **step bound**.

For some vocabulary, we say that the **makecount** of a flipping operation is the number of clauses that become satisfied upon flipping the variable. The **breakcount** is the number of clauses that become *un*satisfied if we flip a variable.

Notice that maximizing **makecount** is not exactly what we desire, as it is very possible for an operation with high makecount has an even higher breakcount, which would be a disimprovement towards the objective. Thus, we will aim to maximize the difference:

```
diffscore = makecount - breakcount
```

Consider the following example of GreedySAT in action.

$$\varphi = (2) \wedge (3) \wedge (1 \vee \overline{3}) \wedge (1 \vee 2 \vee 3)$$

We start with any assignment, so suppose we start with $1 =$ `FALSE`, $2 =$ `FALSE`, $3 =$ `FALSE`. We compute the makecount and breakcount for each of the variables. For a single example, the makecount of variable 3 is equal to 2, because if we flip $3 =$ `FALSE` to become $3 =$ `TRUE`, the second and fourth clauses are now satisfied. The breakcount of 3 is equal to 1, because when we flip $3 =$ `FALSE` to become $3 =$ `TRUE`, the third clause becomes unsatisfied. The full statistics are described below:

Variables →	1	2	3
Value	F	F	F
Makecount	1	2	2
Breakcount	0	0	1
Difference	1	2	1

Since variable 2 has the highest difference, that will be the variable we choose to flip. So the new assignment is $1 =$ `FALSE`, $2 =$ `TRUE`, $3 =$ `FALSE`. Now, three of the clauses are satisfied. We compute the makecount and breakcount again under the new assignment:

Variables →	1	2	3
Value	F	T	F
Makecount	0	0	1
Breakcount	0	2	1
Difference	0	-2	0

Here, there is a tie of variables with maximum difference. We break the tie arbitrarily, an choose to flip 3 from $3 =$ `FALSE` to become $3 =$ `TRUE`. Our formula is still not satisfied, because clause 3 is not satisfied. Again, compute the makecount/breakcount:

Variables →	1	2	3
Value	F	T	T
Makecount	1	0	1
Breakcount	0	1	1
Difference	1	-1	0

Here, we choose variable 1 to flip to become $1 =$ `TRUE`. With the assignment of $1 =$ `TRUE`, $2 =$ `TRUE`, $3 =$ `TRUE`, the formula is satisfied.

Students should refer to the lecture slides for a better visual walkthrough of the above example.

3.2 Incompleteness

It is important to note that, unlike DPLL, GreedySAT (and many local search algorithms in general) is **incomplete**. This means that GSAT may not necessarily find an optimal/feasible solution even given unlimited time. This may be due to starting at a node (initial solution) that cannot reach any feasible/optimal node via our operation, or gets stuck in a cycle/local optimum.

For an example of this, consider the following formula:

$$\varphi = (3) \wedge (1 \vee \overline{3}) \wedge (2 \vee \overline{3}) \wedge (\overline{2} \vee 3) \wedge (\overline{1} \vee 2 \vee \overline{3})$$

Suppose our starting assignment was $1 = \texttt{FALSE}, 2 = \texttt{FALSE}, 3 = \texttt{FALSE}$. The makecount and breakcount are described as:

Variables →	1	2	3
Value	F	F	F
Makecount	0	0	1
Breakcount	0	1	2
Difference	0	-1	-1

We would flip variable 1 to become $1 = \texttt{TRUE}$. Upon doing this, the formula remains unsatisfied (due to clause 1). The new makecounts and breakcounts are:

Variables →	1	2	3
Value	T	F	F
Makecount	0	0	1
Breakcount	0	1	2
Difference	0	-1	-1

the same as before! We again choose to flip 1 to be $1 = \texttt{FALSE}$, but now we start to notice that we are back to where we started from, and are in a cycle.

We can avoid issues like this by utilizing aggressive restarts: whenever we can't greedily increase the number of satisfied clauses, restart with a new random assignment. But the new random assignment may also lead us to a maxima. How can we explore the search space more loosely to escape? How can we speed up our algorithm?

3.3 Simplified WalkSAT

For now, let us just consider 2-SAT – the SAT problem where each clause has 2 literals. The Simplified WalkSAT algorithm is described as follows:

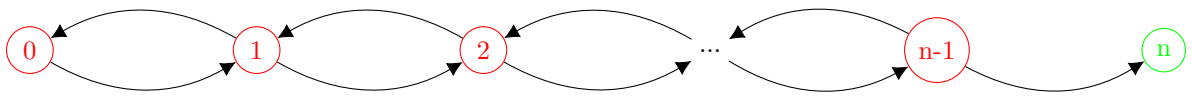
- Start with any assignment of φ
- Arbitrarily pick a clause C that is not satisfied.
- Randomly flip the value of one of C's literals.

It is important to note that this procedure might never finish, for a similar idea as before – getting stuck in a loop. Students should refer to the lecture slides for a walkthrough of WalkSAT.

Remember, for now we are assuming that φ is an instance of 2-SAT. Suppose that φ has a satisfying assignment α. We say that the **state** of WalkSAT is the amount of variables in the current assignment that agree with α. Clearly, $0 \leq \texttt{state} \leq n$, where n is the amount of variables. And if $\texttt{state} = n$, then we have found the satisfying assignment.

Let $\texttt{state}_t$ denote the state at time t. Then, it must be the case that $\texttt{state}_{i+1} = \texttt{state}_i \pm 1$. This holds because in between timestamps, we are always changing exactly one variable according to the procedure.

We can visualize the states as a Markov Chain:



This is to say that as we iterate through WalkSAT, we can mark our state on the Markov Chain by following edges. Notice that for $1 \leq i \leq n-1$, and $\forall t$ we have:

$$\Pr[\texttt{state}_{t+1} = i+1 \mid \texttt{state}_t = i] \geq 1/2 \qquad \text{and} \qquad \Pr[\texttt{state}_{t+1} = i-1 \mid \texttt{state}_t = i] \leq 1/2$$

Notice that the probability is *bounded* at $1/2$ and is not exactly equal to $1/2$. This is because our current assignment, when compared to the true satisfying assignment, differs at one **or both** of the two variables in our clause.
We also have the edge case of the transition from 0 to 1 having probability 1. The question becomes: *"Starting anywhere on the Markov Chain, what is the expected number of steps it takes to reach n?"* The mathematics for this are omitted for the sake of brevity, but the answer is $\mathcal{O}(n^2)$

3.4 Towards 3-SAT

But who cares about 2-SAT? It isn't even an NP-Complete problem! What happens if we do the same procedure for 3-SAT? The states and shape of our Markov Chain remain the same, but the probabilities now change. Let $S^\star$ be a satisfying assignment for φ, and let S_t be the satisfying assignment we have at time t. At time t in our algorithm, we choose a clause c and flip one of its variables. Notice that c IS NOT satisfied under S_t, but IS satisfied under $S^\star$. Thus, between S_t and $S^\star$ *at least one of the three variables* in c differs. This gives us:

$$\Pr[\texttt{state}_{t+1} = i+1 \mid \texttt{state}_t = i] \geq 1/3$$

and as a consequence:

$$\Pr[\texttt{state}_{t+1} = i-1 \mid \texttt{state}_t = i] \leq 2/3$$

These transition probabilities greatly affect the expected runtime, as it is now $\mathcal{O}(2^n)$. The intuition behind this is that there is a larger force "pulling us backwards" along the Markov chain, and the more steps we take, the farther we are from our goal.

Thus, to reconcile this issue, since we move farther "backwards" the longer we run, we should not run for long. This can be implemented by utilizing aggressive restarts. In practice, if you don't find a satisfying assignment in $3n$ steps, restart. This idea improves the expected runtime to $\mathcal{O}((4/3)^n)$.

3.5 WalkSAT in Practice

In practice, rather than just relying on randomness, we mix random walks with greediness:

- Start with any assignment of φ
- Arbitrarily pick a clause c that is not satisfied.
- With fixed probability p: randomly flip the value of one of c's literals (like WalkSAT)
- Else, with probability $1-p$: flip the literal in c to maximize the number of clauses that become satisfied (like greedy SAT)

A natural question is: *"What do we choose for the mixing probability p"*. For this, we cite Professor Charles Elkan from UCSD who writes:

> For random hard 3SAT problems (those with the ratio of clauses to variables around 4.25), $p = 0.5$ works well. For 3SAT formulas with more structure, as generated in many applications, slightly more greediness, i.e. $p < 0.5$ is often better.

In general, it is best to determine the value of p experimentally for your problem. However, for industrial, non-random, and unsatisfiable SAT instances, WalkSAT tends to be much worse than CDCL.