CIS 189

Lecture 12:
Local Search

Jediah Katz jediahk@seas.upenn.edu

Logistics

- **Project check-in due before next class!**
- Project presentations in class the week after
  - Please email me if you get COVID

Recap: Heuristics

- Last week: *construction heuristics*
  - Start with nothing and build up a partial solution
  - Nearest neighbor, nearest/farthest insertion, savings
- This week: *improvement heuristics*
  - Start with any solution and try to find a better one
  - In particular: **local search**

Local Search

- Out of all possible solutions, consider some of them as “neighbors” in (undirected) **neighborhood graph**
  - Typically, two solutions are neighbors if we can transform one into the other by a simple operation
- Start with any solution node, and attempt to reach a better one by exploring its neighborhood
- Limit which moves are acceptable to make the graph directed

Terminating Local Search

- When should we give up exploring?
- **Time bound:** give up if it's taking too long
- **Step bound:** give up after some number of steps
    - Problem-specific knowledge will help here
- **Improvement bound:** give up if we have not improved our solution (enough)
    - Can combine with time/step bounds

Back to TSP

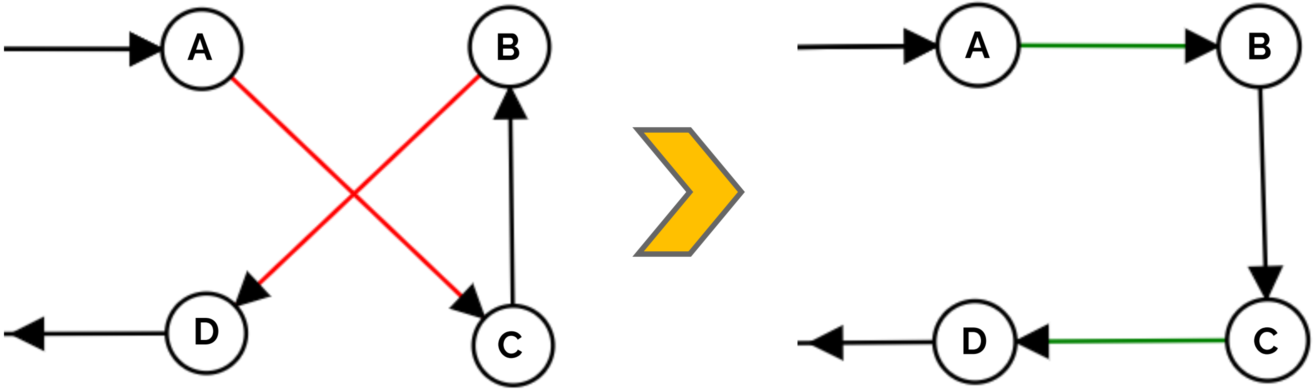
- Local search is natural for TSP
- Start with any tour, and try to improve it into a cheaper tour
- What's a reasonable "neighbor relation" on all tours?
    - What's a simple operation to transform one tour into another tour?

2-Adjacency and 2-Optimality

- We say TSP tours T and T' are **2-adjacent** if we can transform one into the other by deleting two edges and adding two edges
- We say TSP tour T is **2-optimal** if there is no cheaper tour adjacent to T

The 2-opt Swap

- **Idea:** “uncross” the tour where it crosses over itself

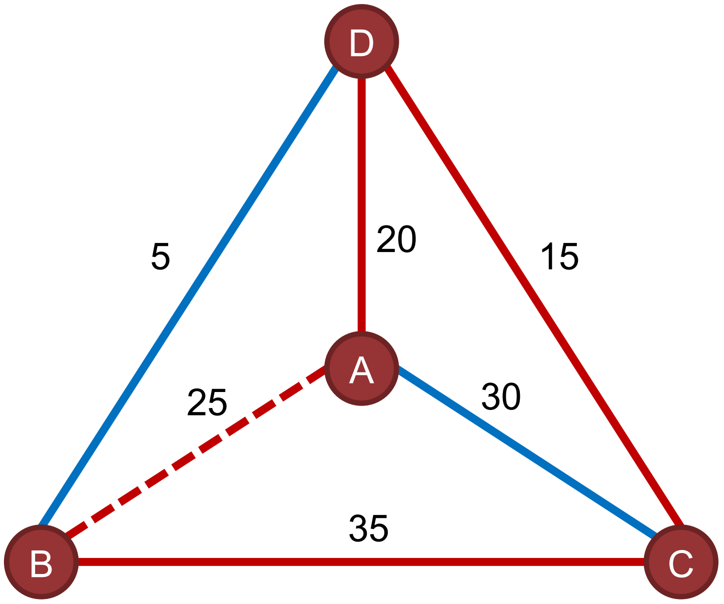


- $\text{swap}(T, i, j) = T[1 : i-1] + T[i\!: j]^R + T[j+1 : n]$
    - $\text{swap}([A, C, B, D], 2, 3) = [A] + [C, B]^R + [D] = [A, B, C, D]$

The 2-opt Heuristic

```
# attempt to improve tour $T$
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($T, i, j$)
```

The 2-opt Heuristic



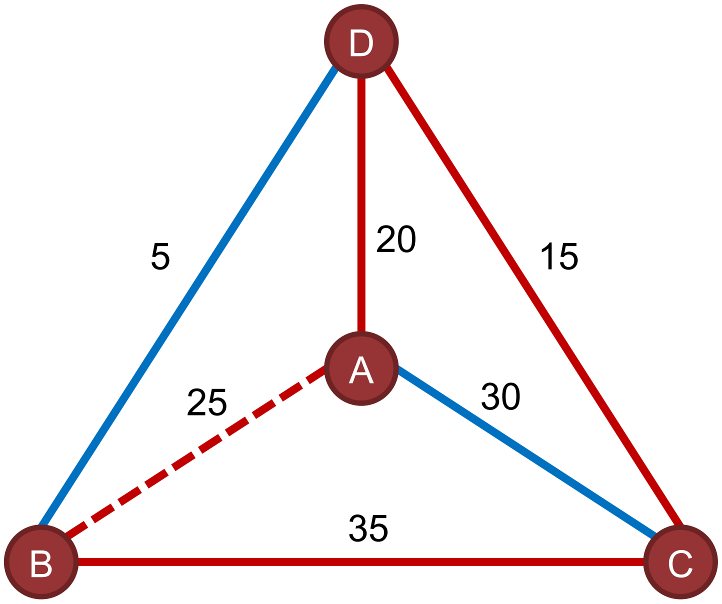
- Current tour:

  A, D, C, B

- Current cost:

  20 + 10 + 35 + 30 = 95

The 2-opt Heuristic



- Current tour:

  A, D, C, B

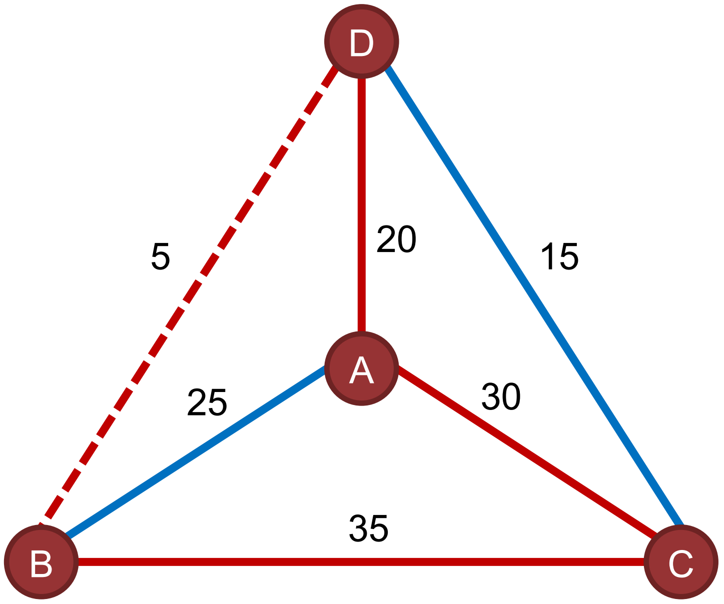
- Current cost:

  20 + 10 + 35 + 30 = 95

- $\mathrm{cost}\big(\mathrm{swap}(T,1,2)\big) = \mathrm{cost}([D,A,C,B])$:

  20 + 30 + 35 + 5 = 90

The 2-opt Heuristic



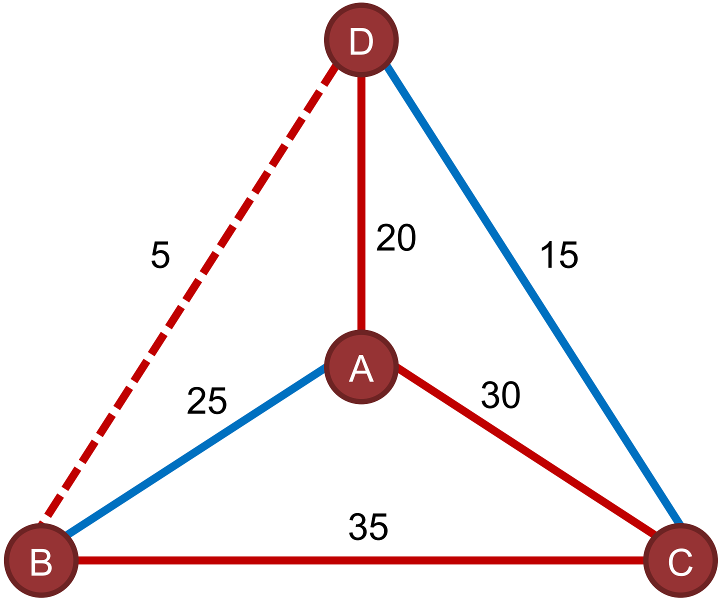
- Current tour:

  D, A, C, B

- Current cost:

  20 + 30 + 35 + 5 = 90

The 2-opt Heuristic



- Current tour:

  D, A, C, B

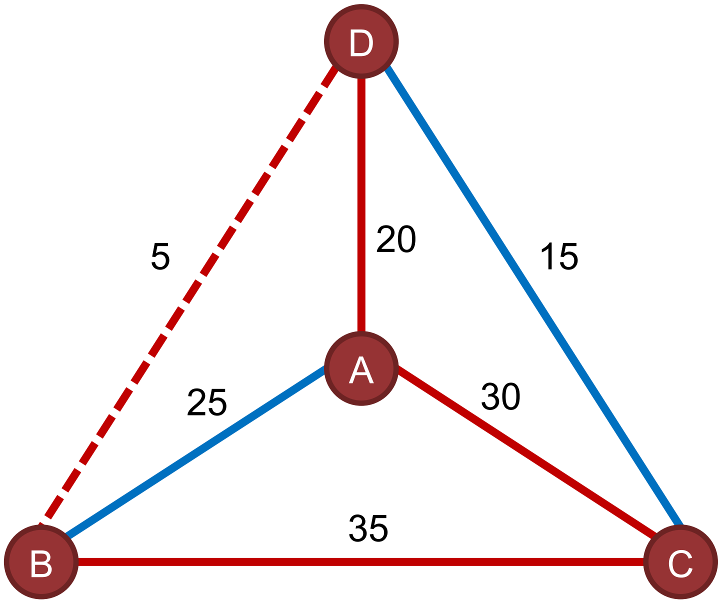
- Current cost:

  20 + 30 + 35 + 5 = 90

- $\mathrm{cost}\left(\mathrm{swap}(T, 1, 2)\right) = \mathrm{cost}([A, D, C, B])$

  20 + 10 + 35 + 30 = 95

The 2-opt Heuristic



- Current tour:

  D, A, C, B

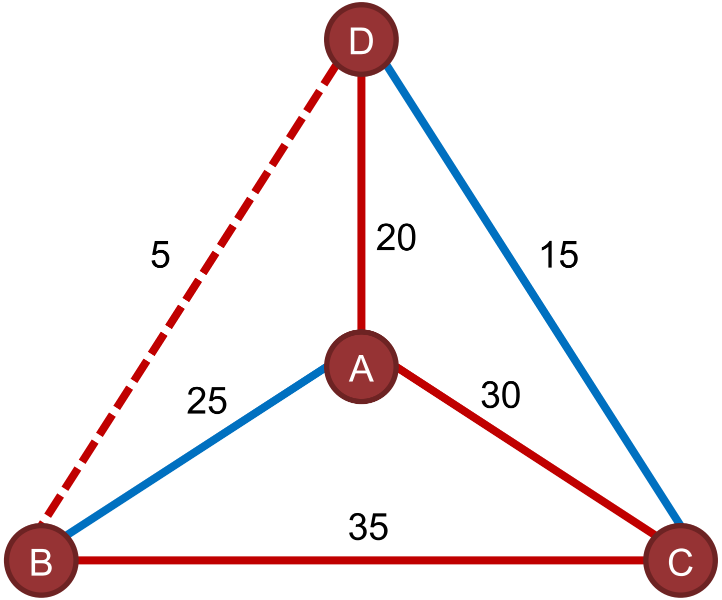
- Current cost:

  20 + 30 + 35 + 5 = 90

- $\mathrm{cost}\big(\mathrm{swap}(T,1,3)\big) = \mathrm{cost}([C,A,D,B])$:

  30 + 20 + 5 + 35 = 90

The 2-opt Heuristic



- Current tour:

  D, A, C, B

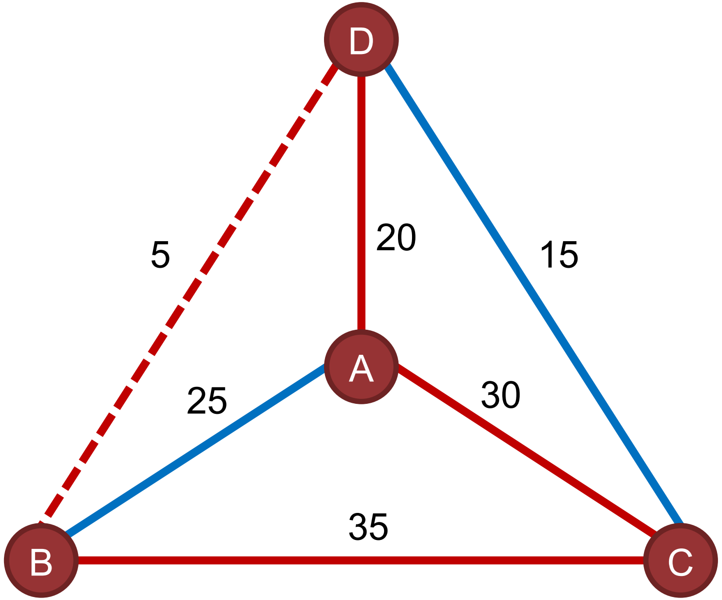
- Current cost:

  20 + 30 + 35 + 5 = 90

- $\mathrm{cost}\big(\mathrm{swap}(T,1,4)\big) = \mathrm{cost}([B,C,A,D])$:

  35 + 30 + 20 + 5 = 90

The 2-opt Heuristic



- Current tour:

  D, A, C, B

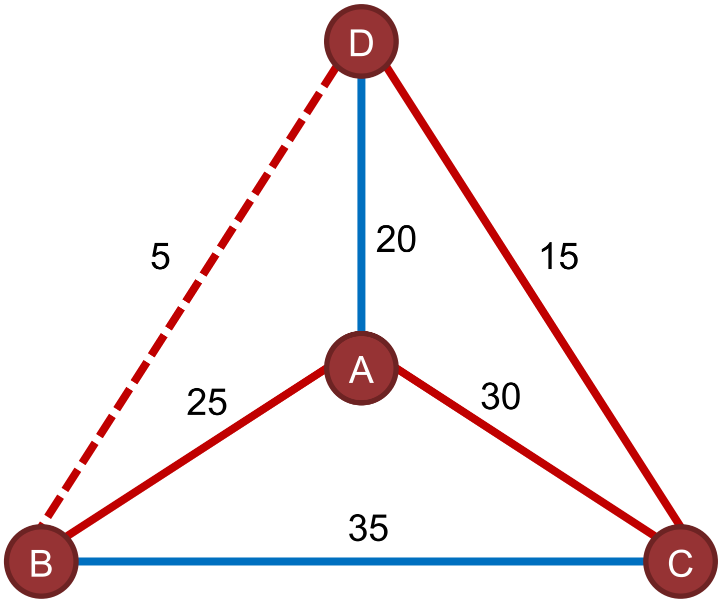
- Current cost:

  20 + 30 + 35 + 5 = 90

- $\mathrm{cost}\left(\mathrm{swap}(T,2,3)\right) = \mathrm{cost}([D,C,A,B])$:

  15 + 30 + 25 + 5 = 75

The 2-opt Heuristic



- Current tour:

  D, C, A, B

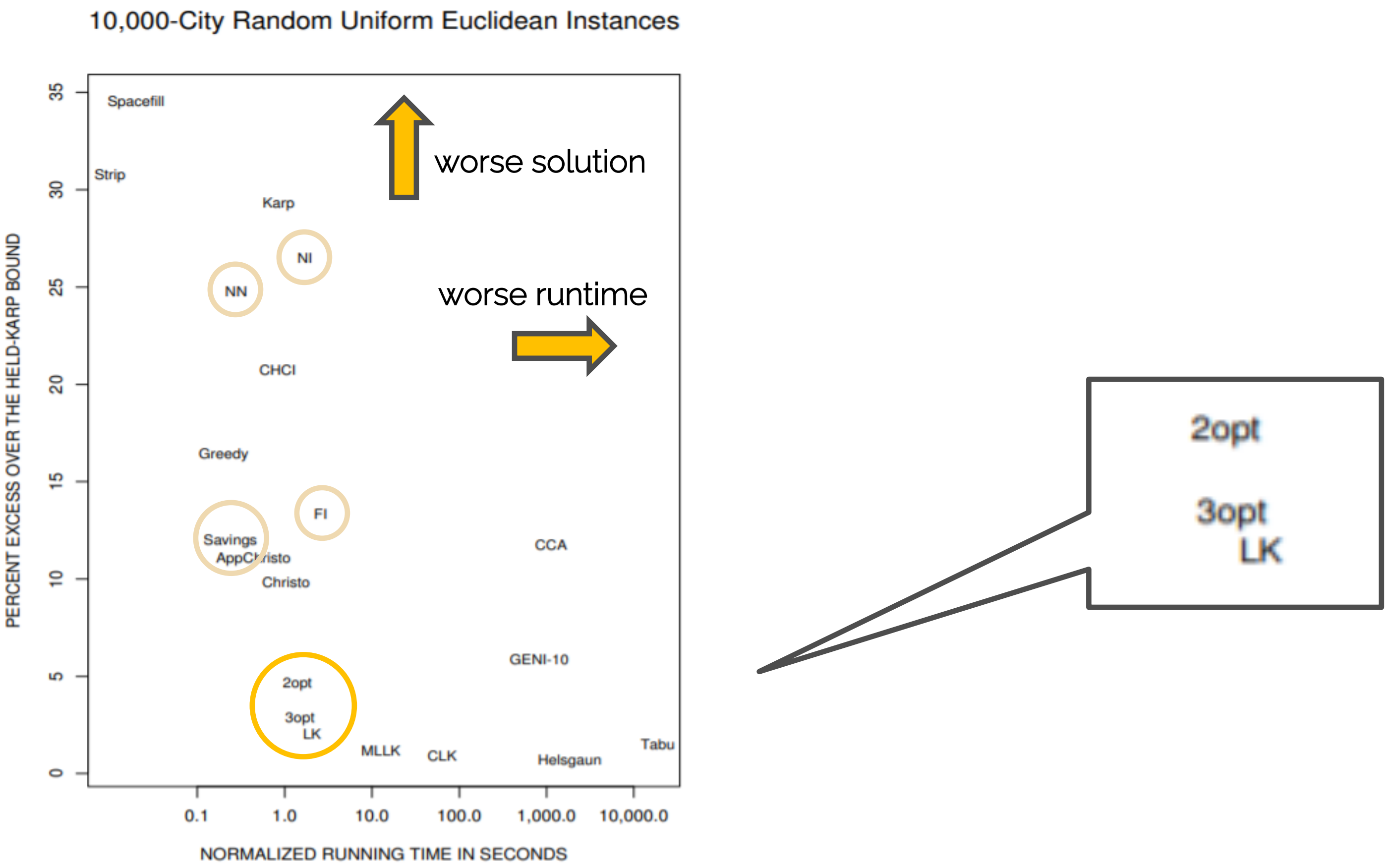
- Current cost:

  15 + 30 + 25 + 5 = 75

- Etc...

Generalizing 2-opt

- Can easily generalize 2-opt to **3-opt, 4-opt, k-opt**...
- **Lin-Kernighan heuristic:** start with k-opt for $k = 2$, then dynamically increase/decrease k over time based on several criteria
  - One of the most effective TSP heuristics!



10,000-City Random Uniform Euclidean Instances

worse solution

worse runtime

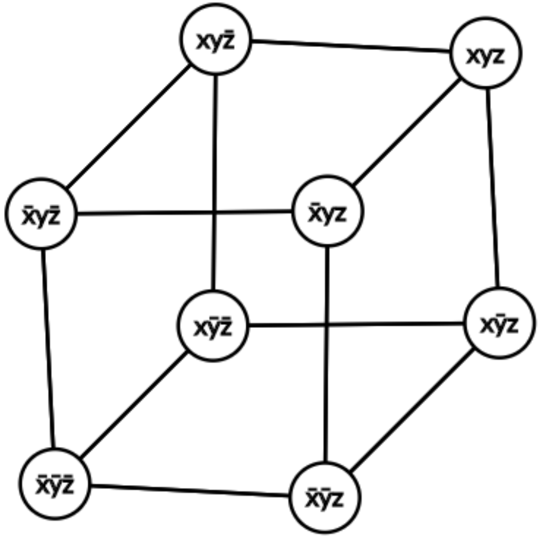
Local Search for SAT

- Even though SAT isn’t an optimization problem, we can still try to solve it with local search
- A “solution” will be any truth assignment, even if it isn’t satisfying
- What is a reasonable “neighbor relation” on all assignments?

Neighborhood of Assignments

- What's a simple operation to transform one assignment into another?

Flip the truth value of a single variable



GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let’s just be **greedy**
- Flip the variable that **maximizes** the number of clauses that **become satisfied**
    - “**Hill-climbing** step”
- What termination criterion makes sense?
    - Steps!

GSAT (Greedy SAT)

- Slight improvement to objective:
- **Makecount:** number of clauses that become satisfied if we flip a variable
- **Breakcount:** number of clauses that become unsatisfied if we flip a variable
- Instead of maximizing makecount, maximize diffscore = makecount – breakcount
    - Corresponds to maximizing total number of satisfied clauses

GSAT Data Structures

- How do we efficiently calculate which flip is best?
- **Unsat list:** all currently unsatisfied clauses
- **Occurrence lists:** clauses containing each literal
- **Makecount and breakcount lists:** for each variable, store the number of clauses that become satisfied/unsatisfied if we flip
    - When we flip x, update counts for all other variables in clauses containing x
- Store number of true literals in each clause

GSAT Flip Pseudocode

```
# for simplicity assume $v$ = T and we set $v$ = F afterwards
pre_flip($v$):
  for clause $C$ containing $v$:
    if n_true_lits[$C$] = 1: # case 1 -> 0
      add $C$ to unsat_list
      for literal $l$ in $C$: make_count[var($l$)] += 1
      break_count[$v$] -= 1
    else if n_true_lits[$C$] = 2: # case 2 -> 1
      let $l$ = the other true literal in $C$
      break_count[var($l$)] += 1
  for clause $C$ containing $\overline{v}$:
    # false -> true case is essentially symmetric
```

GSAT (Greedy SAT)

$(\bar{2})$

$(\bar{3})$

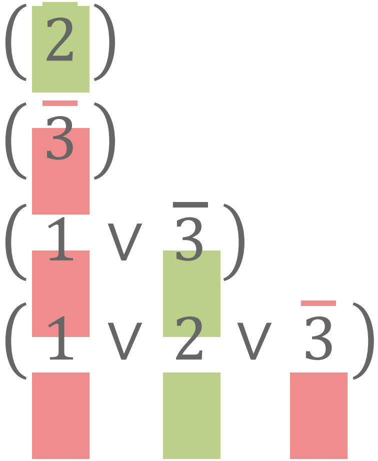
$(1 \vee \bar{3})$

$(1 \vee 2 \vee \bar{3})$

	1	2	3
Value	F	F	F
Makecount	1	2	2
Breakcount	0	0	1

We started with a “random” assignment. It just happened to be (F, F, F).

GSAT (Greedy SAT)



(2)

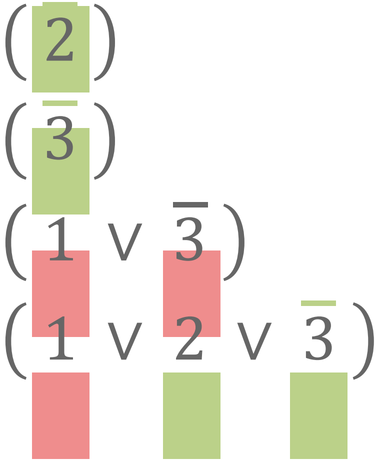
$(\overline{3})$

$(1 \vee \overline{3})$

$(1 \vee 2 \vee \overline{3})$

	1	2	3
Value	F	T	F
Makecount	0	0	1
Breakcount	0	2	1

GSAT (Greedy SAT)



(2)

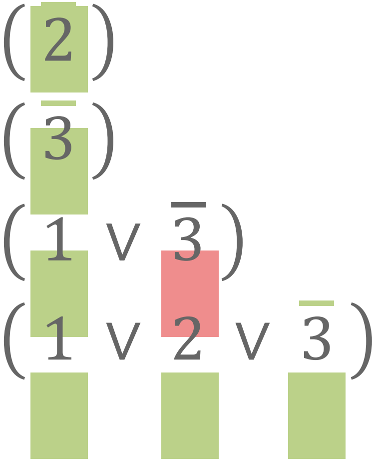
$(\overline{3})$

$(1 \vee \overline{3})$

$(1 \vee 2 \vee \overline{3})$

	1	2	3
Value	F	T	T
Makecount	1	0	1
Breakcount	0	1	1

GSAT (Greedy SAT)



(2)

$(\bar{3})$

$(1 \vee \bar{3})$

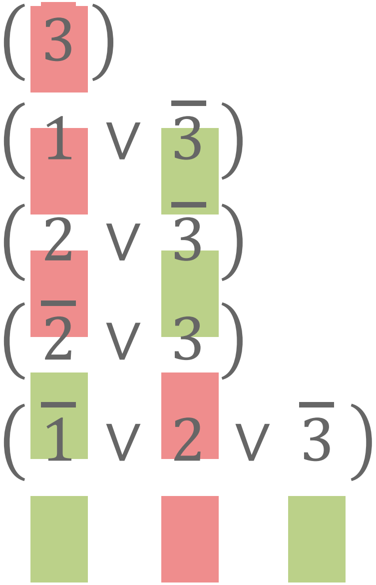
$(1 \vee 2 \vee \bar{3})$

	1	2	3
Value	T	T	T
Makecount	0	0	0
Breakcount	1	1	1

Incompleteness

- Unlike DPLL, GSAT (and many local search algorithms in general) is **incomplete**
    - May not necessarily find an optimal/feasible solution even given unlimited time
- May start at node that can't reach any feasible/optimal node or get stuck in a cycle/local optimum

A bad GSAT example



(3)

$(1 \vee \overline{3})$

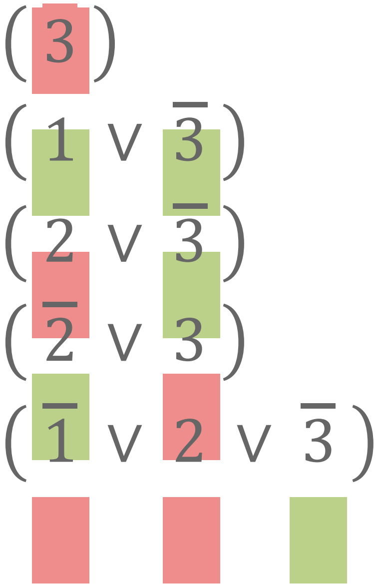
$(2 \vee \overline{3})$

$(\overline{2} \vee 3)$

$(\overline{1} \vee 2 \vee \overline{3})$

	1	2	3
Value	F	F	F
Makecount	0	0	1
Breakcount	0	1	2

A bad GSAT example



$(\overline{3})$

$(1 \vee \overline{3})$

$(2 \vee \overline{3})$

$(\overline{2} \vee 3)$

$(\overline{1} \vee 2 \vee \overline{3})$

	1	2	3
Value	T	F	F
Makecount	0	0	1
Breakcount	0	1	2

Avoiding local optima

- Can use a technique we've seen before...
- Aggressive **restarts**: whenever we can't greedily increase number of satisfied clauses, restart with a new random assignment

Towards a better algorithm

- Might still just repeatedly get stuck in local maxima
- How can we explore the search space more loosely to escape?
- Also, our greedy heuristic is slow: requires checking all variables at each step

Simplified WalkSAT

- For now, let's just consider 2-SAT
- **Simplified WalkSAT algorithm:**
    - 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

- "Random walk" might never finish!

Simplified WalkSAT

$(3 \lor 2)$

$(1 \lor \overline{3})$

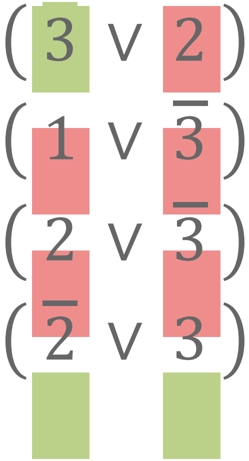
$(2 \lor \overline{3})$

$(\overline{2} \lor 3)$

Flip 3!

1	2	3
F	F	F

Simplified WalkSAT



$(3 \lor 2)$

$(1 \lor \overline{3})$

$(2 \lor \overline{3})$

$(\overline{2} \lor 3)$

🎲 **Flip 1!**

1	2	3
F	F	T

Simplified WalkSAT

$(3 \vee 2)$ 🟩 🟥

$(1 \vee \overline{3})$ 🟩 🟥

$(2 \vee \overline{3})$ 🟥 🟥

$(\overline{2} \vee 3)$ 🟩 🟩

1	2	3
T	F	T

🎲 **Flip 3!** (oops…!)

Simplified WalkSAT

$(3 \lor 2)$ 🟥 🟥

$(1 \lor \overline{3})$ 🟩 🟩

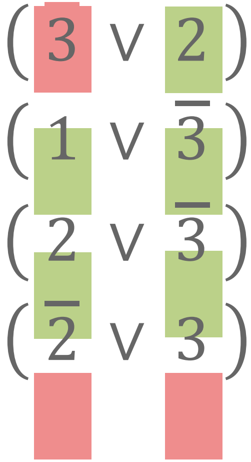
$(2 \lor \overline{3})$ 🟥 🟩

$(\overline{2} \lor 3)$ 🟩 🟥

🎲 **Flip 2!**

1	2	3
T	F	F

Simplified WalkSAT



$(3 \vee 2)$

$(1 \vee \overline{3})$

$(2 \vee \overline{3})$

$(\overline{2} \vee 3)$

1	2	3
T	T	T

Flip 3!

Simplified WalkSAT

$(3 \lor 2)$

$(1 \lor \overline{3})$

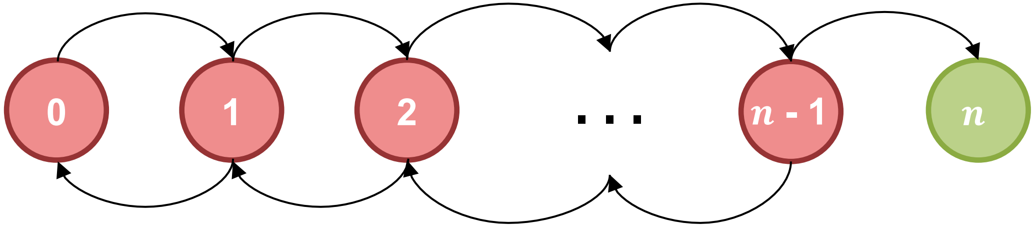
$(2 \lor \overline{3})$

$(\overline{2} \lor 3)$

1	2	3
T	T	T

Analyzing Simplified WalkSAT

- For now, let's just consider 2-SAT
- Simplified WalkSAT is mathematically "nice"
- Suppose φ has a satisfying assignment α
- *State* of WalkSAT: how many variables in the current assignment agree with α?



Analyzing Simplified WalkSAT

at least one variable in the clause must disagree

$$[\ x \lor y\ ]$$

worst case: accidentally flip a correct variable

unsatisfied clause

- At least ½ probability of advancing to next state
    - If we reach state n, done
- In expectation, satisfying assignment will be found in $O(n^2)$ steps

From 2-SAT to 3-SAT

- Intuition behind simplified WalkSAT running time: we're at least as likely to move forward as backwards, so given enough time we'll get lucky
- Who cares about 2-SAT? Not NP-complete.
- OK, so let's just do the same procedure for 3-SAT

The Problem with 3-SAT

- Probability of moving to next state is at least 1/3
- Probability of moving backwards to previous state can be as bad as 2/3!
- **Intuition:** we're "pulled" backwards, and the more steps we take the farther we are from our goal
- Expected runtime: $O(2^n)$

A Smarter 3-CNF WalkSAT

- **Idea:** since we move farther “backwards” the longer we run, we should not run for long
- Can utilize aggressive **restarts**
  - If we don’t find a satisfying assignment in $3n$ steps, restart
- Expected runtime: $O\left(\left(\frac{4}{3}\right)^n\right)$
  - Assuming we start from a random assignment

WalkSAT in Practice

- In practice, rather than just rely on randomness, we'll **mix random walks and greediness**
- **WalkSAT algorithm:**
    - 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
    - Else with probability $1 - p$:
        - Flip literal in C to maximize number of clauses that become satisfied

Choosing a Mixing Probability

- What to choose for the mixing probability p?
- Prof. Charles Elkan (UCSD):

> 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.

- Best to determine experimentally for your problem
  - For industrial (non-random) and unsatisfiable SAT instances, WalkSAT is probably much worse than CDCL