CIS 1921

# Lecture 11: Local Search

Ishaan Lal ilal@seas.upenn.edu

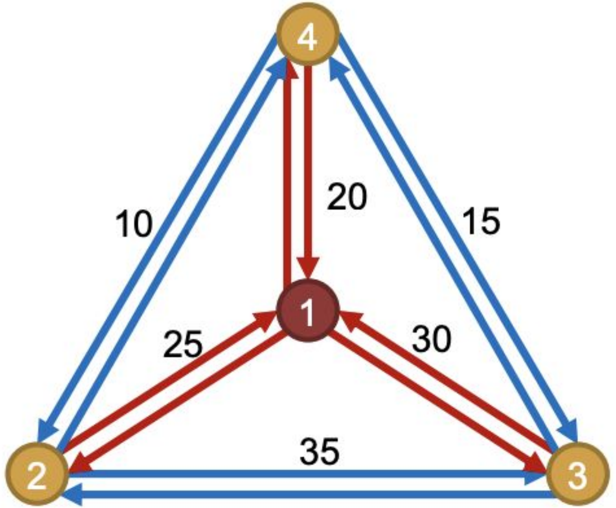
# Logistics

- **Project check-in now due Monday!**
- Next week last Ishaan lecture
- Final class on 4/24
  - Project presentations (7-9 minutes, hard stop at 9)
  - All details regarding project presentations and final submission are on the master doc on the website

# Savings Heuristic

- Pick any vertex $x$ to be the "central vertex"
- Start with $n-1$ subtours: $x \to v \to x$ for all $v \in V - x$
- For each edge $(i,j)$, where $i,j \in V - x$, compute its **savings** $s(i,j)$
  - $s(i,j) = w(i,x) + w(x,j) - w(i,j)$
- Sort edges in decreasing order of savings
- Repeat until only one tour remains:
- Let $(i,j)$ be the next edge in sorted order
- If edges $(i,x)$ and $(x,j)$ are in our subtours, and $i,j$ are not already in the same tour: replace $(i,x)$ and $(x,j)$ by $(i,j)$

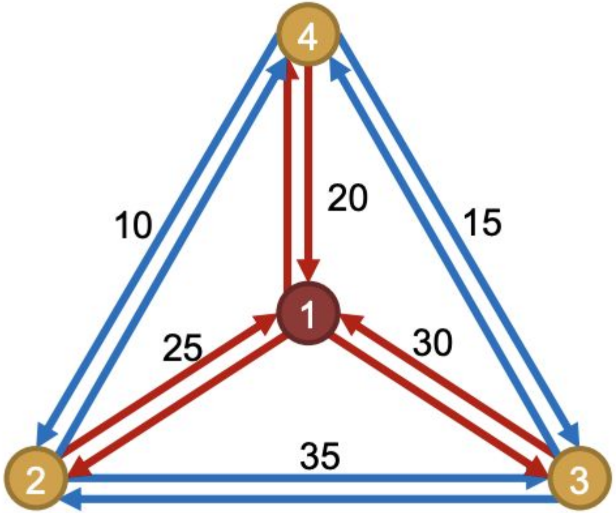
# Savings Heuristic




- Current cost:

25 + 25 + 30 + 30 + 20 + 20 = 150

# Savings Heuristic

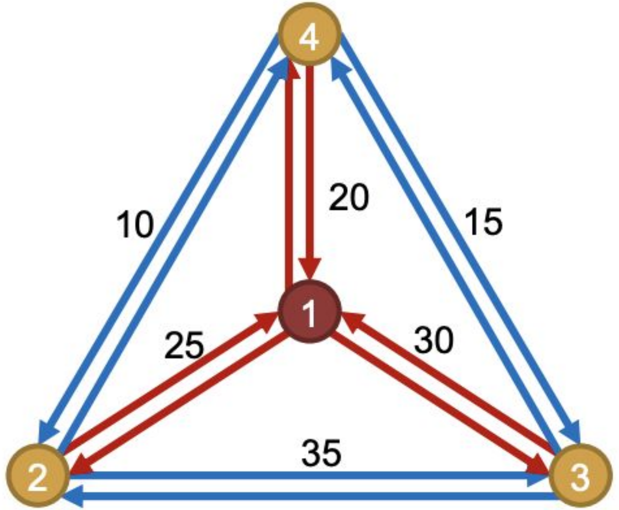



- Current cost:

25 + 25 + 30 + 30 + 20 + 20 = 150

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1)+w(1,3)-w(2,3)$ $=25+30-35=20$ |
| $(3,2)$ | $w(3,1)+w(1,2)-w(3,2)$ $=30+25-35=20$ |
| $(2,4)$ | $w(2,1)+w(1,4)-w(2,4)$ $=25+20-10=35$ |
| $(4,2)$ | $w(4,1)+w(1,2)-w(4,2)$ $=20+25-10=35$ |
| $(3,4)$ | $w(3,1)+w(1,4)-w(3,4)$ $=30+20-15=35$ |
| $(4,3)$ | $w(4,1)+w(1,3)-w(4,3)$ $=20+30-15=35$ |

# Savings Heuristic

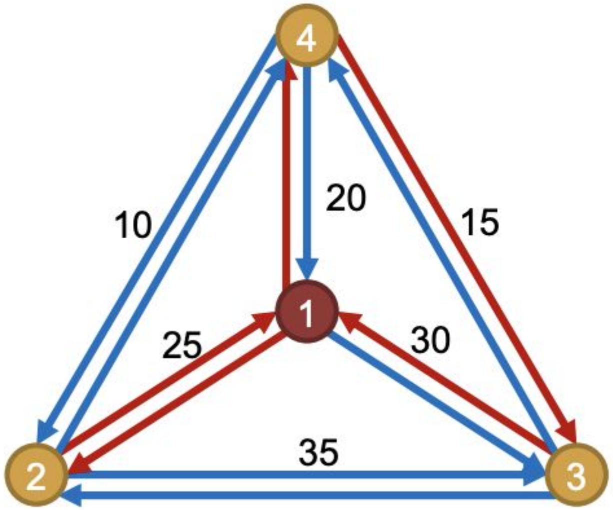



- Current cost:

25 + 25 + 30 + 30 + 20 + 20 = 150

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1)+w(1,3)-w(2,3)$ $=25+30-35=20$ |
| $(3,2)$ | $w(3,1)+w(1,2)-w(3,2)$ $=30+25-35=20$ |
| $(2,4)$ | $w(2,1)+w(1,4)-w(2,4)$ $=25+20-10=35$ |
| $(4,2)$ | $w(4,1)+w(1,2)-w(4,2)$ $=20+25-10=35$ |
| $(3,4)$ | $w(3,1)+w(1,4)-w(3,4)$ $=30+20-15=35$ |
| $(4,3)$ | $w(4,1)+w(1,3)-w(4,3)$ $=20+30-15=35$ |

# Savings Heuristic

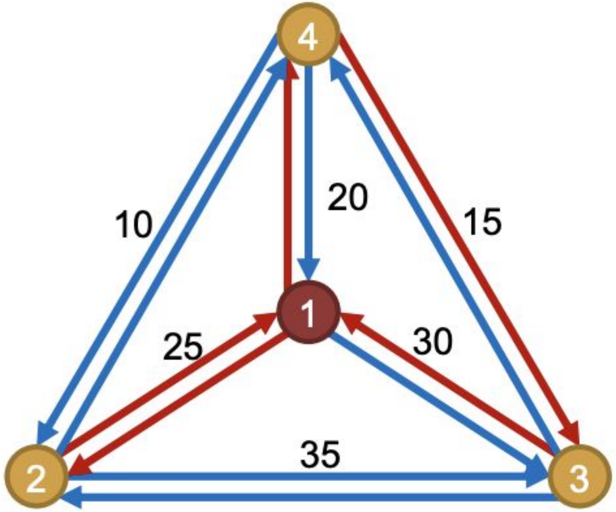



- Current cost:

25 + 25 + 20 + 15 + 30 = 115

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1)+w(1,3)-w(2,3)$ $=25+30-35=20$ |
| $(3,2)$ | $w(3,1)+w(1,2)-w(3,2)$ $=30+25-35=20$ |
| $(2,4)$ | $w(2,1)+w(1,4)-w(2,4)$ $=25+20-10=35$ |
| $(4,2)$ | $w(4,1)+w(1,2)-w(4,2)$ $=20+25-10=35$ |
| $(3,4)$ | $w(3,1)+w(1,4)-w(3,4)$ $=30+20-15=35$ |
| $(4,3)$ | $w(4,1)+w(1,3)-w(4,3)$ $=20+30-15=35$ |

# Savings Heuristic

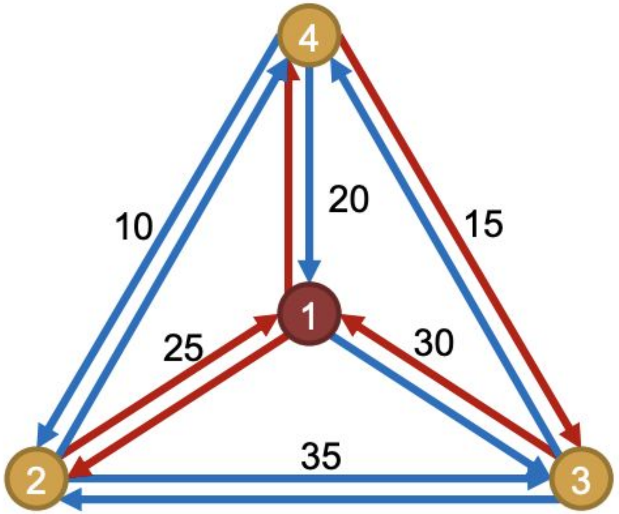



- Current cost:

25 + 25 + 20 + 15 + 30 = 115

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1)+w(1,3)-w(2,3)$ $=25+30-35=20$ |
| $(3,2)$ | $w(3,1)+w(1,2)-w(3,2)$ $=30+25-35=20$ |
| $(2,4)$ | $w(2,1)+w(1,4)-w(2,4)$ $=25+20-10=35$ |
| $(4,2)$ | $w(4,1)+w(1,2)-w(4,2)$ $=20+25-10=35$ |
| $(3,4)$ | $w(3,1)+w(1,4)-w(3,4)$ $=30+20-15=35$ |
| $(4,3)$ | $w(4,1)+w(1,3)-w(4,3)$ $=20+30-15=35$ |

# Savings Heuristic

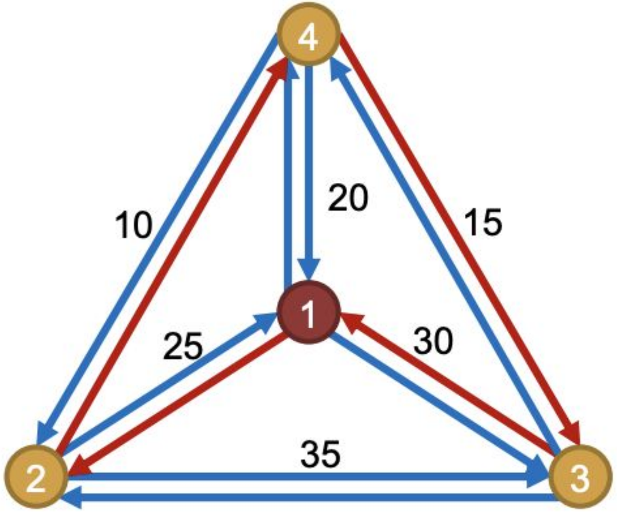



- Current cost:

25 + 25 + 20 + 15 + 30 = 115

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1)+w(1,3)-w(2,3)$<br>$=25+30-35=20$ |
| $(3,2)$ | $w(3,1)+w(1,2)-w(3,2)$<br>$=30+25-35=20$ |
| $(2,4)$ | $w(2,1)+w(1,4)-w(2,4)$<br>$=25+20-10=35$ |
| $(4,2)$ | $w(4,1)+w(1,2)-w(4,2)$<br>$=20+25-10=35$ |
| $(3,4)$ | $w(3,1)+w(1,4)-w(3,4)$<br>$=30+20-15=35$ |
| $(4,3)$ | $w(4,1)+w(1,3)-w(4,3)$<br>$=20+30-15=35$ |

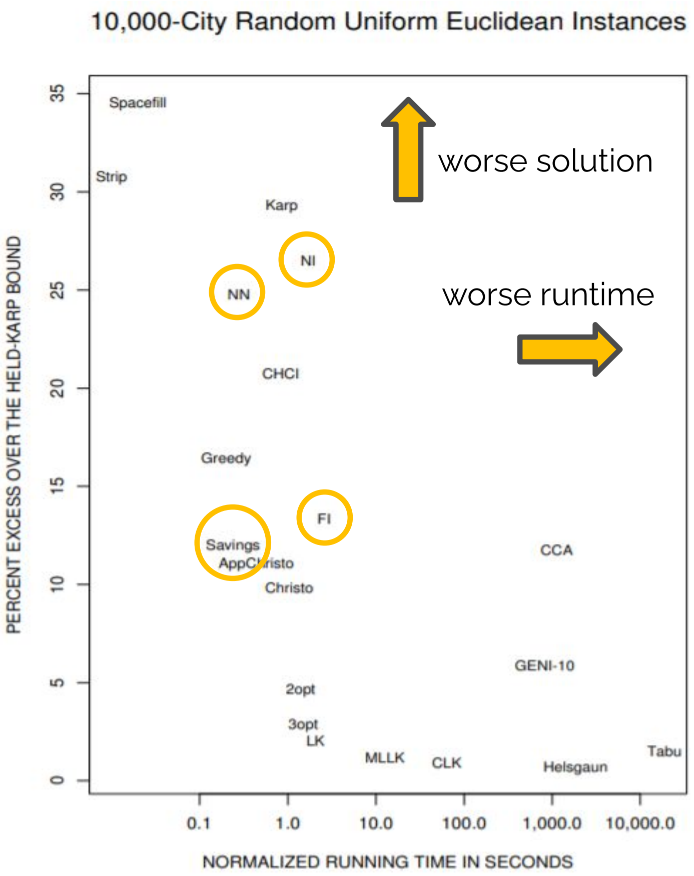
# Savings Heuristic

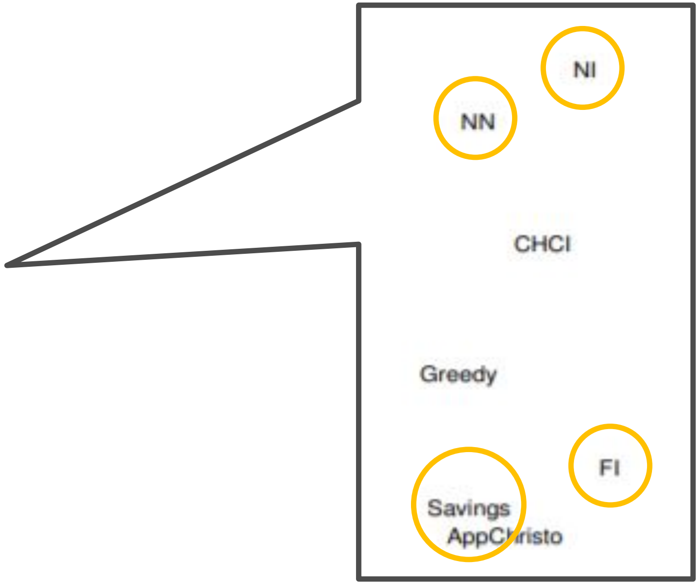



- Current cost:

25 + 10 + 15 + 30 = 80

| $(i,j)$ | Savings $s(i,j)$ |
|---|---|
| $(2,3)$ | $w(2,1) + w(1,3) - w(2,3)$ $= 25 + 30 - 35 = 20$ |
| $(3,2)$ | $w(3,1) + w(1,2) - w(3,2)$ $= 30 + 25 - 35 = 20$ |
| $(2,4)$ | $w(2,1) + w(1,4) - w(2,4)$ $= 25 + 20 - 10 = 35$ |
| $(4,2)$ | $w(4,1) + w(1,2) - w(4,2)$ $= 20 + 25 - 10 = 35$ |
| $(3,4)$ | $w(3,1) + w(1,4) - w(3,4)$ $= 30 + 20 - 15 = 35$ |
| $(4,3)$ | $w(4,1) + w(1,3) - w(4,3)$ $= 20 + 30 - 15 = 35$ |




https://pubsonline.informs.org/doi/abs/10.1287/ijoc.4.4.387
http://www.atgc-montpellier.fr/permutmatrix/manual/SeriationPPI.htm

# Vehicle Routing Problem

- Actually, the Savings heuristic was created to solve a generalization of the TSP:
- The **Vehicle Routing Problem (VRP)** also takes place in a weighted, complete graph
- Instead of one salesman, we have a fleet of vehicles which are all parked at a central vertex (the **depot**)
  - May or may not be a limit on the number of vehicles
- **Goal**: find routes starting and ending at the depot for each vehicle with minimum total weight so that each vertex is visited once by some vehicle

# Constrained VRP

- In real life: why use a fleet of vehicles when you could have one vehicle that travels all the routes?
- There may be additional constraints for vehicles, e.g.:
  - Maximum distance a vehicle can travel
  - Carrying capacity of a vehicle, where each node has some volume to be delivered

# Savings Heuristic for VRP

- Let $x$ denote the depot
- Start with $n-1$ subtours: $x \rightarrow v \rightarrow x$ for all $v \in V - x$
- For each edge $(i, j)$, where $i, j \in V - x$, compute its **savings** $s(i, j)$
  - $s(i,j) = w(i,x) + w(x,j) - w(i,j)$
- Sort edges in decreasing order of savings
- Repeat until ~~only one~~ k tour remains or we reach negative savings:
- Let $(i, j)$ be the next edge in sorted order
- If edges $(i, x)$ and $(x, j)$ are in our subtours, and $i, j$ are not already in the same tour: replace $(i, x)$ and $(x, j)$ by $(i, j)$...
  - ...unless it would violate our constraints

# Solving TSP with OR-Tools

- OR-Tools comes with a **routing solver** that can solve the TSP and VRP with much more complex constraints!
    - Pickups and drop-offs, time windows, penalties…
- The guide is pretty good: [https://developers.google.com/optimization/routing](https://developers.google.com/optimization/routing)
- Comes with many heuristics including NN, Savings, etc…
    - By default, solver automatically chooses a heuristic to use based on the problem at hand
- Note: the routing solver is optimized for getting a "good enough" solution to constrained problems, not exact solving huge TSPs

# 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
- In other words, start with any solution, and continuously tweak it to a better solution.

# 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

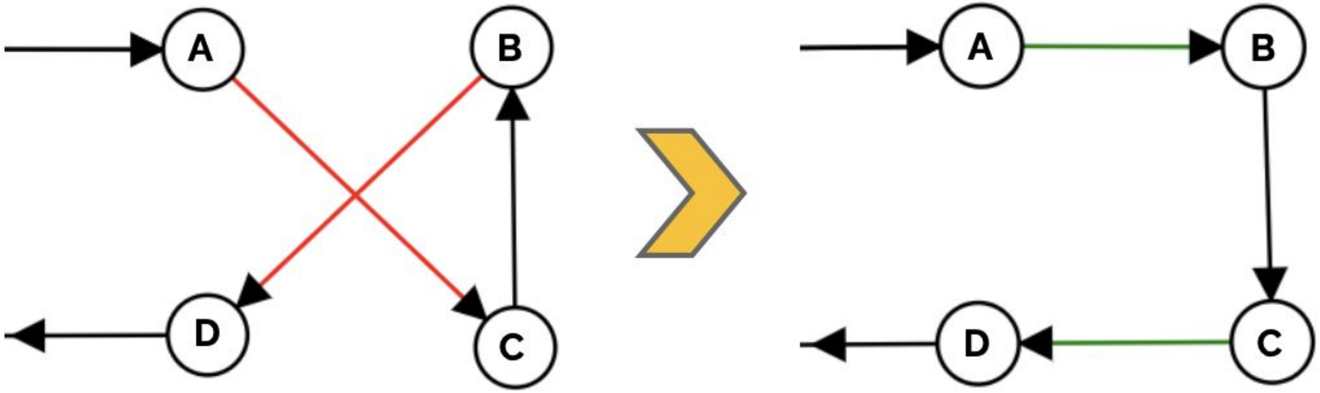
- Local search is natural for TSP
- Start with any tour, and try to improve it into a cheaper tour
- What's a reasonable "neighbor relation" on all tours?
    - What's a simple operation to transform one tour into another tour?

# 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 $T$

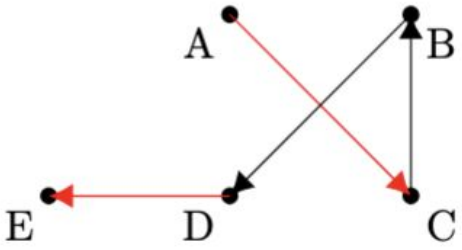
# The 2-opt Swap

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

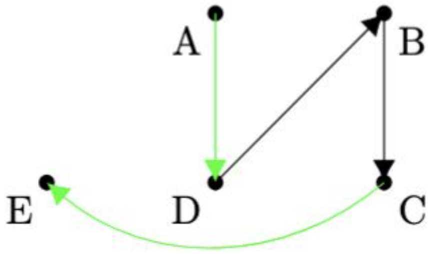



- $\text{swap}(T, i, j) = T[1 : i - 1] + T[i{:}\, j]^R + T[j + 1 : n]$
  - $\text{swap}([A, C, B, D], 2, 3) = [A] + [C, B]^R + [D] = [A, B, C, D]$

# The 2-opt Swap




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



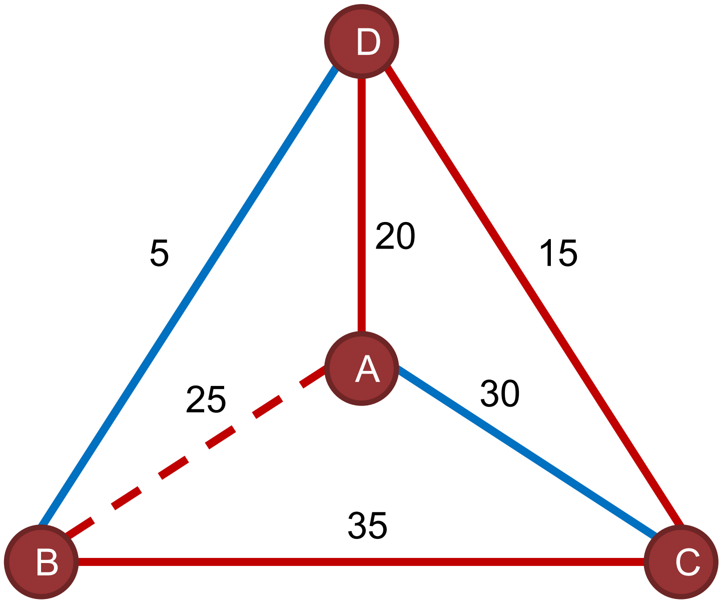
# The 2-opt Heuristic

```
# 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)
```

This heuristic *does not guarantee* you will find the optimal solution.

# The 2-opt Heuristic



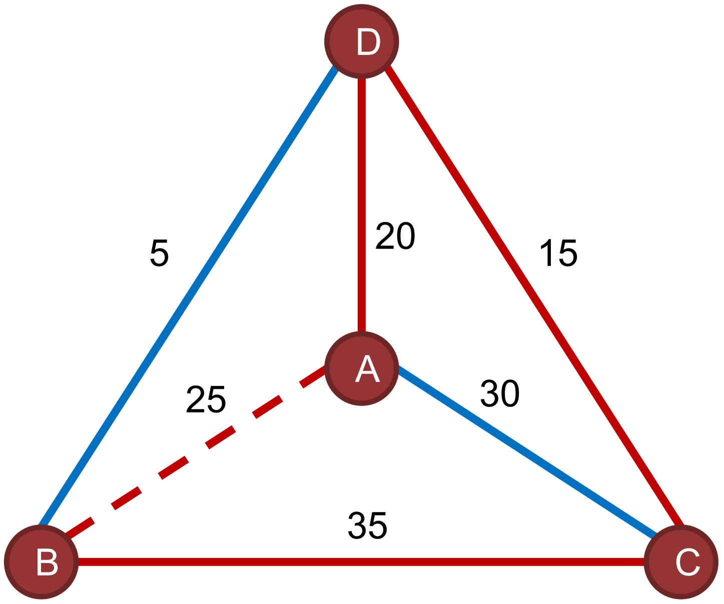

- Current tour:

  A, D, C, B

- Current cost:

  20 + 10 + 35 + 30 = 95

# The 2-opt Heuristic




- **Current tour:**

  A, D, C, B

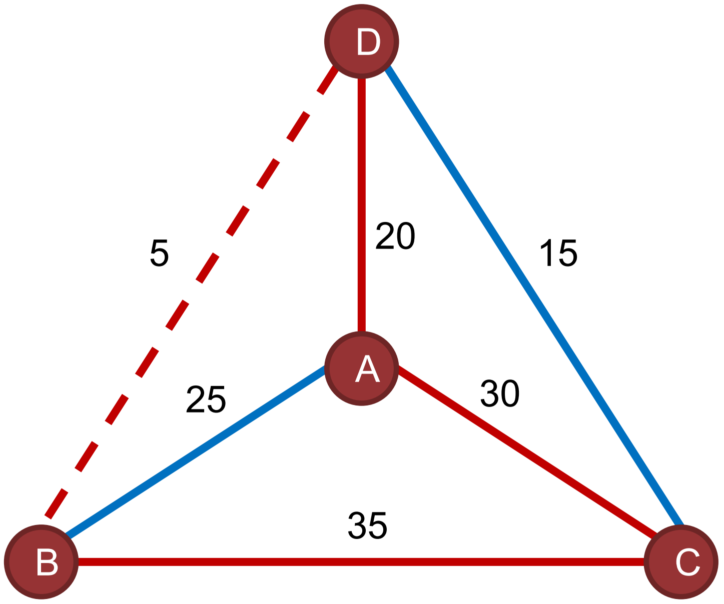
- **Current cost:**

  20 + 10 + 35 + 30 = 95

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

  20 + 30 + 35 + 5 = 90

# The 2-opt Heuristic



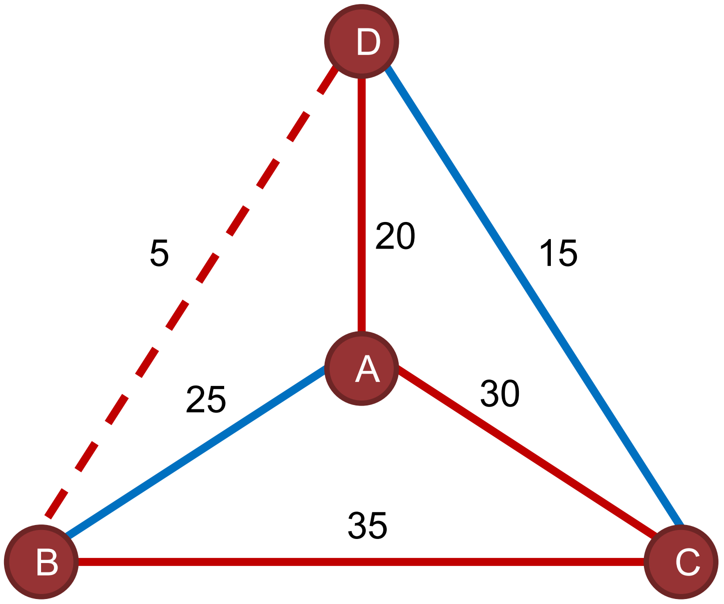

- Current tour:

  D, A, C, B

- Current cost:

  20 + 30 + 35 + 5 = 90

# The 2-opt Heuristic




- **Current tour:**

  D, A, C, B

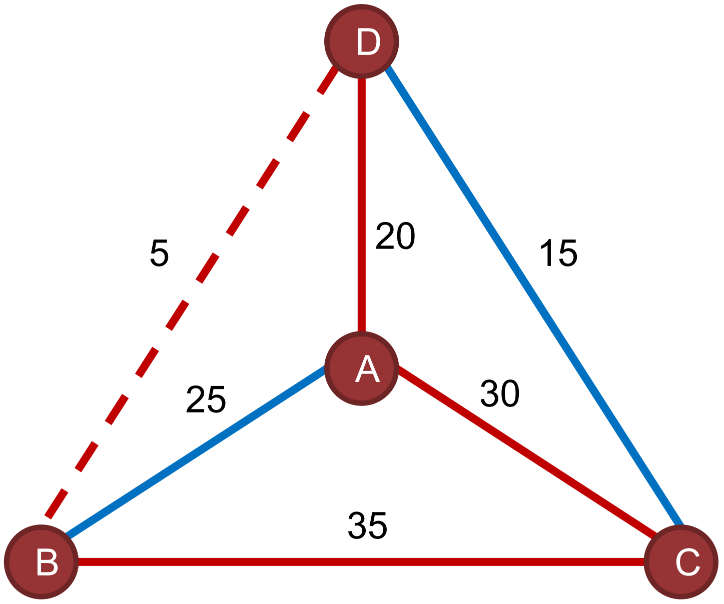
- **Current cost:**

  20 + 30 + 35 + 5 = 90

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

  20 + 10 + 35 + 30 = 95

# The 2-opt Heuristic




- **Current tour:**

  D, A, C, B

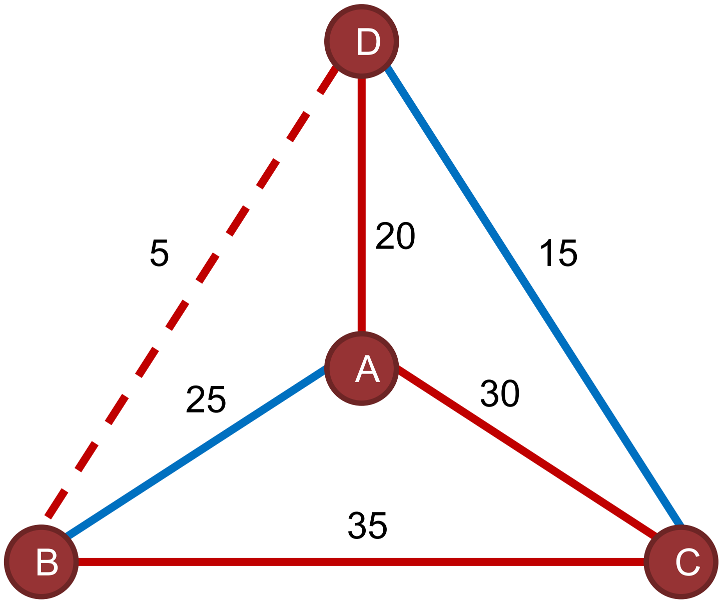
- **Current cost:**

  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

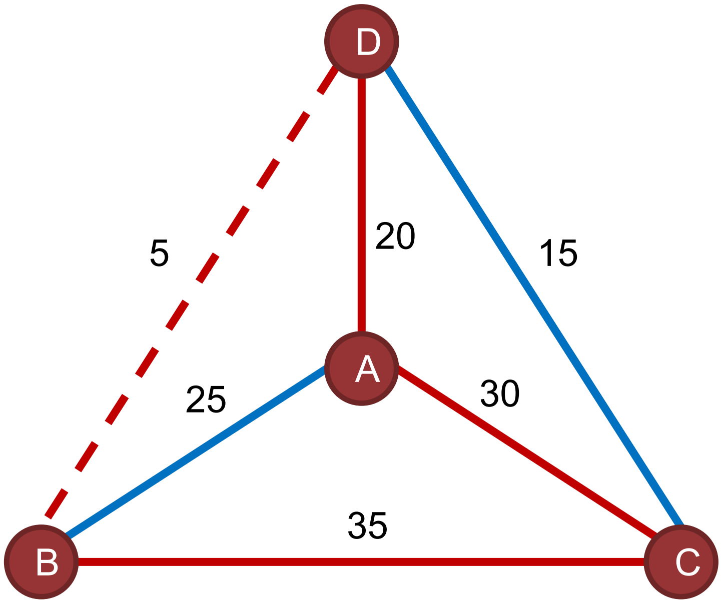
- **Current cost:**

  20 + 30 + 35 + 5 = 90

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

  35 + 30 + 20 + 5 = 90

# The 2-opt Heuristic




- **Current tour:**

  D, A, C, B

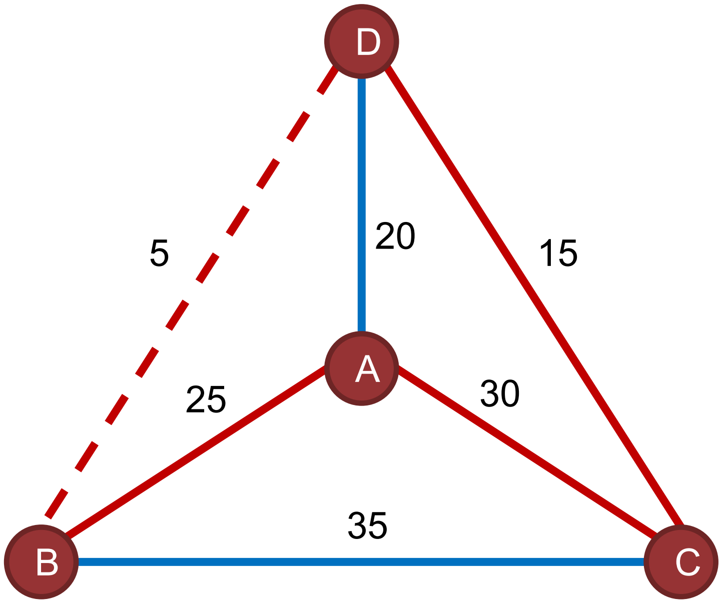
- **Current cost:**

  20 + 30 + 35 + 5 = 90

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

  15 + 30 + 25 + 5 = 75

# The 2-opt Heuristic




- Current tour:

  D, C, A, B

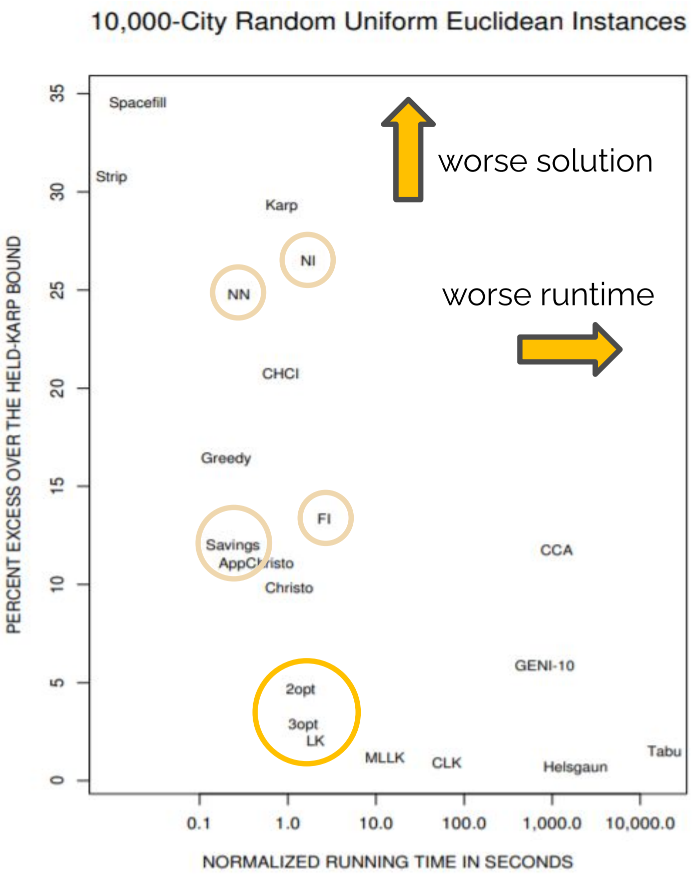
- Current cost:

  15 + 30 + 25 + 5 = 75

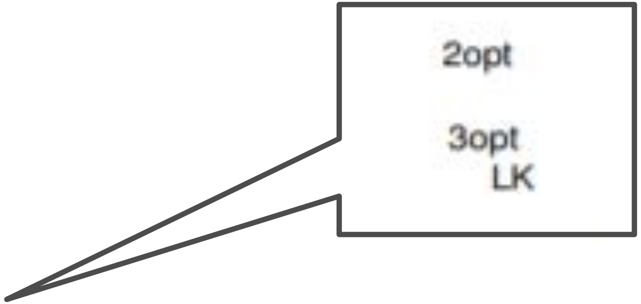
- Etc...

# 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
Spacefill
Strip
Karp
NI
NN
CHCI
Greedy
FI
Savings
AppChristo
Christo
CCA
GENI-10
2opt
3opt
LK
MLLK
CLK
Helsgaun
Tabu
PERCENT EXCESS OVER THE HELD-KARP BOUND
0
5
10
15
20
25
30
35
0.1
1.0
10.0
100.0
1,000.0
10,000.0
NORMALIZED RUNNING TIME IN SECONDS

2opt
3opt
LK

**Are you sure 2-OPT doesn't always eventually return optimal?**

YES

# 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?

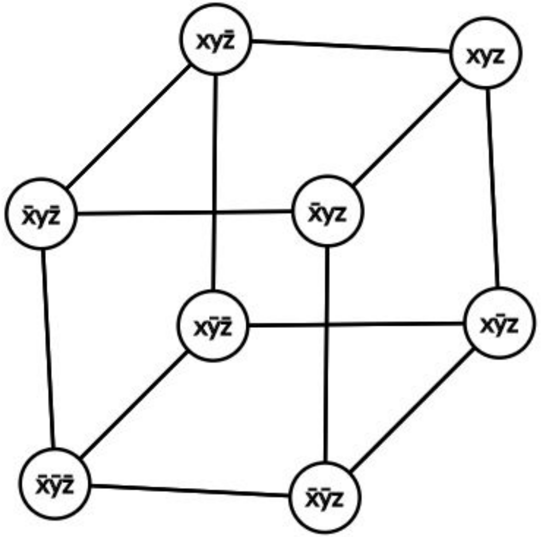
# Neighborhood of Assignments

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

# 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?

# GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let's just be **greedy**

# GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let's just be **greedy**
- Idea: flip the variable that will make the most unsatisfied clauses become satisfied.

# GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let's just be **greedy**
- Idea: flip the variable that will make the most unsatisfied clauses become satisfied.
- **Issue:** if flipping variable x changes 100 clauses from unsat → sat, **but at the same time** changes 200 clauses from sat → unsat, we aren't making progress in the right direction

# GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let's just be **greedy**
- **Idea 2:** Flip the variable that **maximizes** the number of clauses that **become satisfied**
    - The **net change** in satisfied clauses

# GSAT (Greedy SAT)

- Which variable to flip?
- First attempt: let's just be **greedy**
- **Idea 2:** Flip the variable that **maximizes** the number of clauses that **become satisfied**
  - The **net change** in satisfied clauses
- 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 $x$

# 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 v̄:
    # false -> true case is essentially symmetric
```

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | F | F |
| **Makecount** | | | |
| **Breakcount** | | | |
| **Difference** | | | |

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

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | F | F |
| **Makecount** | 1 | 2 | 2 |
| **Breakcount** | 0 | 0 | 1 |
| **Difference** | 1 | 2 | 1 |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | F | F |
| **Makecount** | 1 | 2 | 2 |
| **Breakcount** | 0 | 0 | 1 |
| **Difference** | 1 | 2 | 1 |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | **T** | F |
| **Makecount** | | | |
| **Breakcount** | | | |
| **Difference** | | | |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | T | F |
| **Makecount** | 0 | 0 | 1 |
| **Breakcount** | 0 | 2 | 1 |
| **Difference** | 0 | -2 | 0 |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | T | F |
| **Makecount** | 0 | 0 | 1 |
| **Breakcount** | 0 | 2 | 1 |
| **Difference** | 0 | -2 | 0 |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | T | **T** |
| **Makecount** | | | |
| **Breakcount** | | | |
| **Difference** | | | |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 1 | 2 | 3 |
|---|---|---|---|
| **Value** | F | T | T |
| **Makecount** | 1 | 0 | 1 |
| **Breakcount** | 0 | 1 | 1 |
| **Difference** | 1 | -1 | 0 |

# GSAT (Greedy SAT)

$(2)$

$(3)$

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

$(1 \vee 2 \vee 3)$

| | 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 |
| **Difference** | 0 | -1 | -1 |

# 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** | **T** | F | F |
| **Makecount** | | | |
| **Breakcount** | | | |

# 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** | **T** | F | F |
| **Makecount** | 0 | 0 | 1 |
| **Breakcount** | 0 | 1 | 2 |

# 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** | **T** | F | F |
| **Makecount** | 0 | 0 | 1 |
| **Breakcount** | 0 | 1 | 2 |
| **Difference** | 0 | -1 | -1 |

# 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 $\varphi$
  - 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

$(3 \vee 2)$

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

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

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

🎲 Flip 3!

| 1 | 2 | 3 |
|---|---|---|
| F | F | F |

# Simplified WalkSAT

$(3 \vee 2)$

$(1 \vee \overline{3})$ 🎲 **Flip 1!**

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

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

| 1 | 2 | 3 |
|---|---|---|
| F | F | T |

# Simplified WalkSAT

$(3 \lor 2)$

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

$(2 \lor \overline{3})$ 🎲 **Flip 3!** (oops…!)

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

| 1 | 2 | 3 |
|---|---|---|
| T | F | T |

# Simplified WalkSAT

$(3 \vee 2)$ 🎲 **Flip 2!**

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

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

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

| 1 | 2 | 3 |
|---|---|---|
| T | F | F |

# Simplified WalkSAT

$(3 \vee 2)$

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

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

$(\overline{2} \vee 3)$ 🎲 **Flip 3!**

| 1 | 2 | 3 |
|---|---|---|
| T | T | T |

# Simplified WalkSAT

$(3 \vee 2)$

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

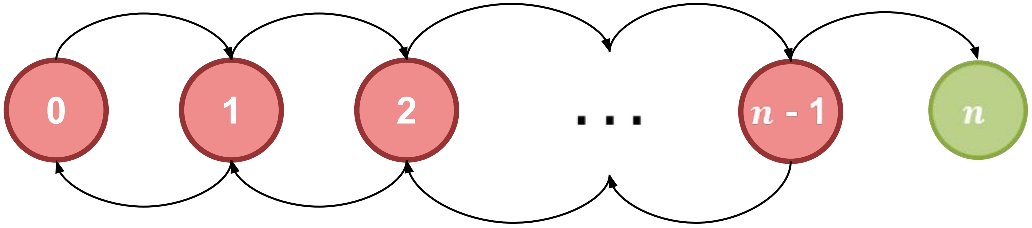
$(2 \vee \overline{3})$

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

| 1 | 2 | 3 |
|---|---|---|
| T | T | T |

# Analyzing Simplified WalkSAT

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



Let our "current" assignment be $\mathbf{S_t}$ and let the true assignment be $\mathbf{S^*}$

$\mathbf{S_t}$

| | |
|---|---|
| $x_1$ | F |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | F |
| $x_5$ | T |

State = 3

$\mathbf{S^*}$

| | |
|---|---|
| $x_1$ | T |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | T |
| $x_5$ | T |

State = 5

Case 1

$$[x_1 \vee x_4]$$

Both variables in the unsatisfied clause we chose happen to differ from **S***

**$S_t$**

| | |
|---|---|
| $x_1$ | F |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | F |
| $x_5$ | T |

State = 3

**$S^*$**

| | |
|---|---|
| $x_1$ | T |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | T |
| $x_5$ | T |

State = 5

Case 1

$$[x_1 \vee x_4]$$

Both variables in the unsatisfied clause we chose happen to differ from S*

**$S_t$**

| | |
|---|---|
| $x_1$ | F |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | F |
| $x_5$ | T |

State = 3

**S***

| | |
|---|---|
| $x_1$ | T |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | T |
| $x_5$ | T |

State = 5

If we randomly choose $x_1$ to flip, we will move to state 4.

If we randomly choose $x_4$ to flip, we will move to state 4.

Case 2

$$[x_2 \vee x_4]$$

$S_t$

| | |
|---|---|
| $x_1$ | F |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | F |
| $x_5$ | T |

State = 3

$S^*$

| | |
|---|---|
| $x_1$ | T |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | T |
| $x_5$ | T |

State = 5

Case 2

$$[x_2 \vee x_4]$$

**$S_t$**

| | |
|---|---|
| $x_1$ | F |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | F |
| $x_5$ | T |

State = 3

**$S^*$**

| | |
|---|---|
| $x_1$ | T |
| $x_2$ | F |
| $x_3$ | F |
| $x_4$ | T |
| $x_5$ | T |

State = 5

If we randomly choose $x_2$ to flip, we will move to state 2.

If we randomly choose $x_4$ to flip, we will move to state 4.

Probability of “making progress”

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

Probability of “going backwards”

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

# Analyzing Simplified WalkSAT

$[\ x \lor y\ ]$

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 $\varphi$
  - 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):

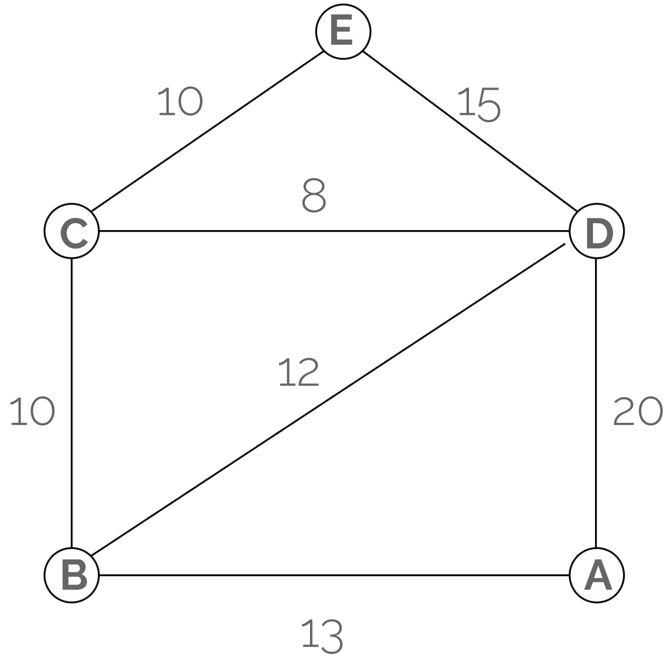
> For random hard 3SAT problems (those with the ratio of clauses to variables around 4.25) $p$ = 0.5 works well. For 3SAT formulas with more structure, as generated in many applications, slightly more greediness, i.e. $p$ < 0.5, is often better.

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

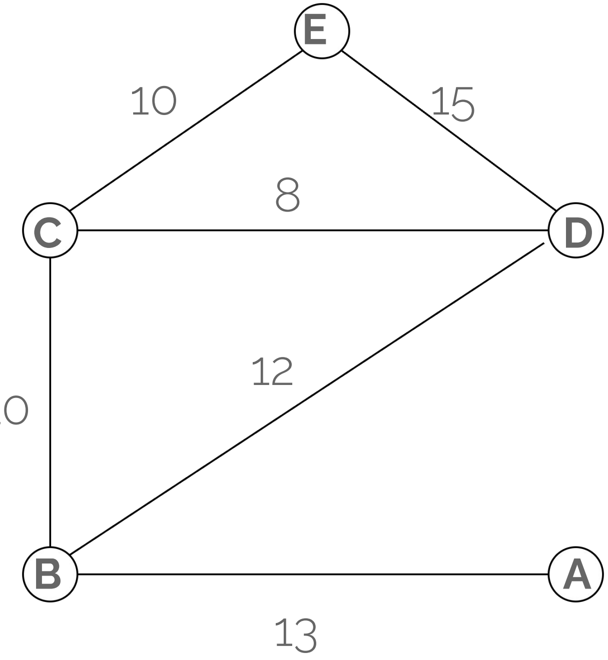
# Chinese Postman Problem (CPP)

- Studied by Chinese mathematician Kwan Mei-Ko in 1960
- Given an undirected weighted graph G, what is the **least weight** traversal of the graph that visits every **edge** at least one time?
- Example: A postman delivering letters wants to know the optimal route that traverses every street in a given area.

# Chinese Postman Problem (CPP)



# Chinese Postman Problem (CPP)

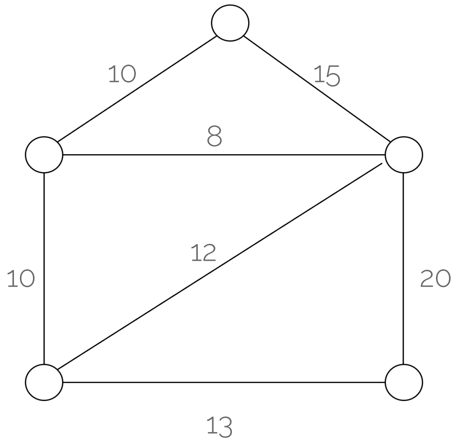



OPT: A → B → C →E → D → C → D → B

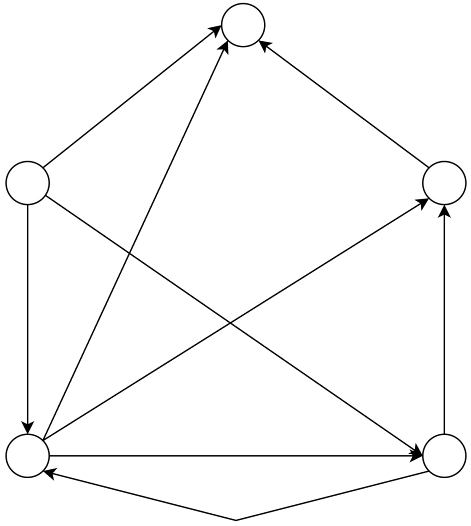
# Chinese Postman Problem (CPP)

- CPP can be solved in Polynomial Time.
  - $O(n^3)$ solution using T-joins
  - Directed CPP is also Poly-time solvable $O(V^2E)$
- CPP can be solved in Polynomial Time.

# Variations of CPP

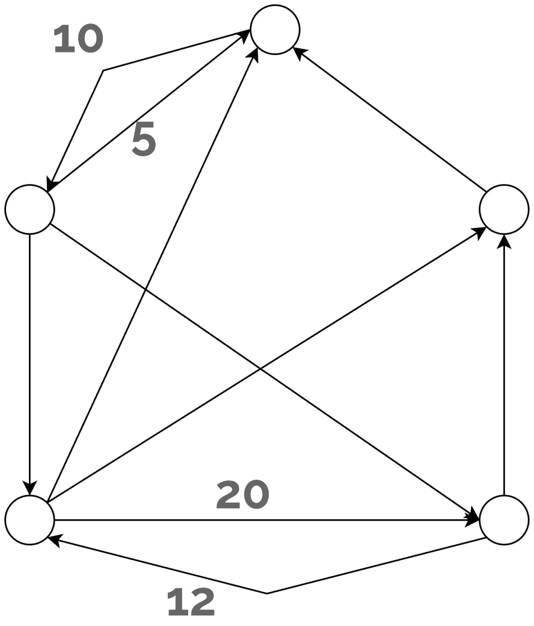



Undirected CPP
(UCPP)



Directed CPP
(DCPP)
NY Street Sweeper

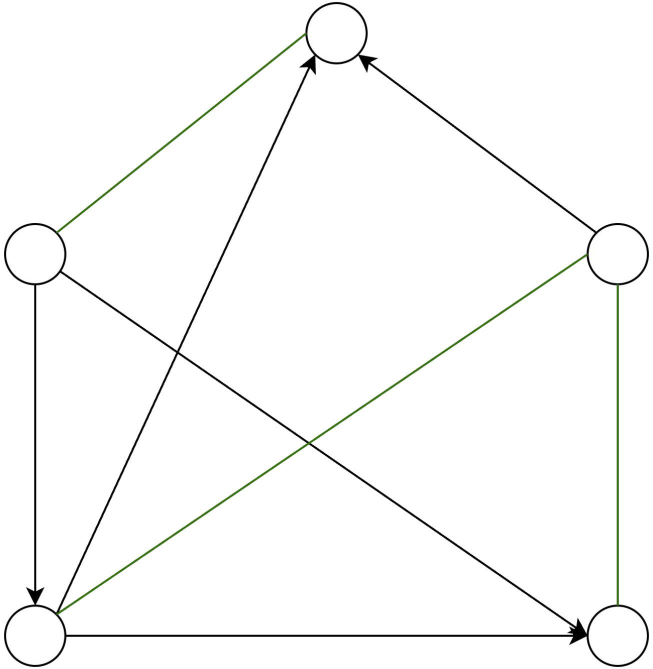
# Variations of CPP




Windy CPP

- For some pairs of vertices, edges exist in both directions, but they have different weight.
- WCPP is NP-Hard

# Variations of CPP



Mixed CPP

- Graph has a mixture of directed edges and undirected edges
- Mix of one-way and two-way streets
- Undirected edges only need to be traversed in one direction
- MCPP is NP-Hard

# From MCPP to TSP

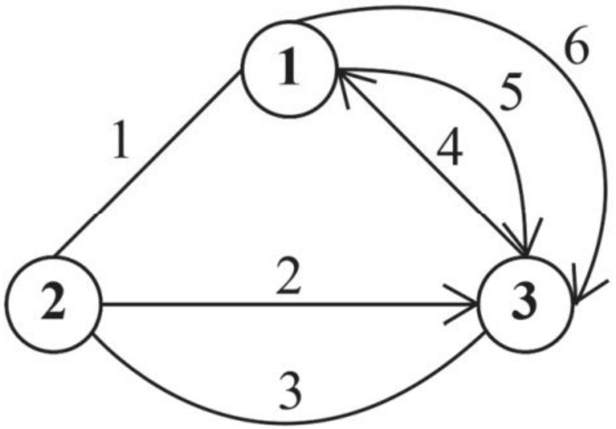



**Fig. 1.** The original MCPP problem in multigraph

Step 1:

- Replace undirected edges with parallel edges

# From MCPP to TSP

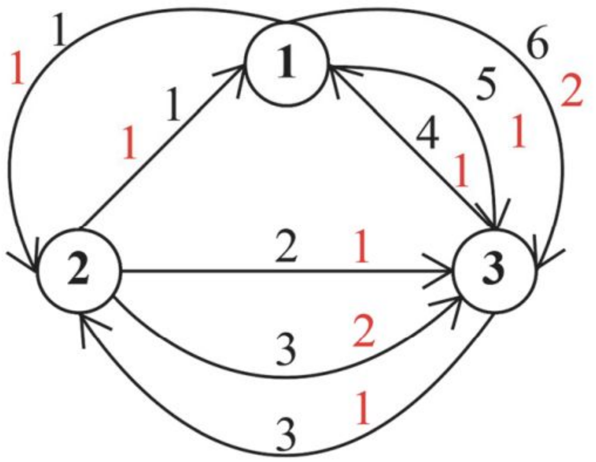



Fig. 2. The results of numbering each parallel arc

Step 1:

- Replace undirected edges with parallel edges
- Red numbers are indices for edges pointing between the same pair of vertices.

# From MCPP to TSP

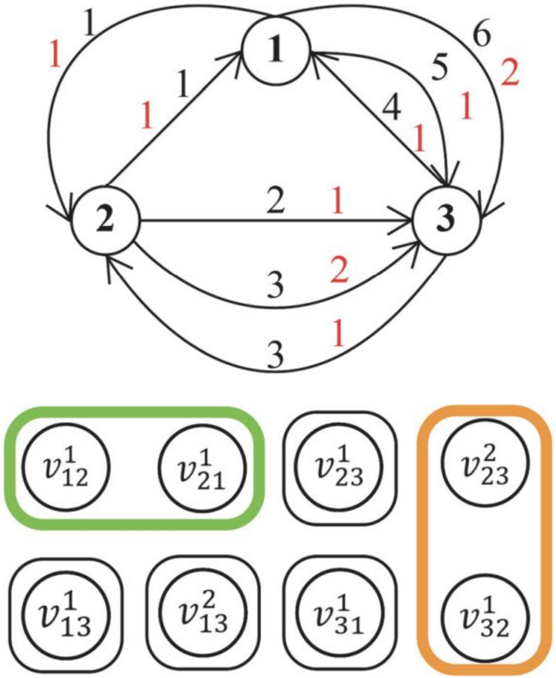



**Fig. 3.** The vertices and clusters of transformed problems

Step 2:

- Create a vertex for each edge.
- Vertices in the same colored box represent the edges that create an undirected edge.

# From MCPP to TSP

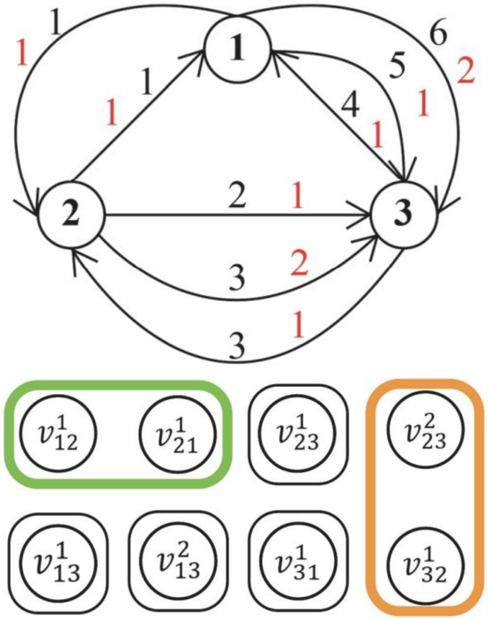



**Fig. 3.** The vertices and clusters of transformed problems

Step 3:

- Determine the weight on the edge between all pairs of vertices

$$c(v_{ab}^{k_1}, v_{cd}^{k_2}) = d_{bc} + c_{cd}^{k_2}$$

$d_{bc}$ is the shortest distance between vertices b and c in the original graph.

$$c(v_{21}^{1}, v_{31}^{1}) = d_{13} + c_{31}^{1}$$
$$= (1 + 2) + 4$$

# From MCPP to TSP

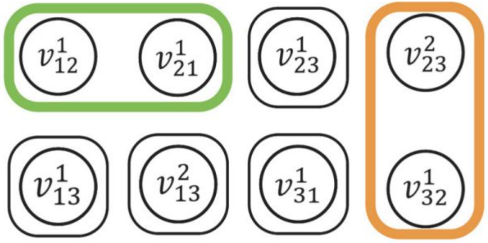



**Fig. 3.** The vertices and clusters of transformed problems

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | | $v^1_{12}$ | $v^1_{13}$ | $v^2_{13}$ | $v^1_{21}$ | $v^1_{23}$ | $v^2_{23}$ | $v^1_{31}$ | $v^1_{32}$ |
| 1 | $v^1_{12}$ | - | 6 | 7 | 1 | 2 | 3 | 6 | 5 |
| 2 | $v^1_{13}$ | 5 | - | 10 | 4 | 5 | 6 | 4 | 3 |
| 3 | $v^2_{13}$ | 5 | 9 | - | 4 | 5 | 6 | 4 | 3 |
| 4 | $v^1_{21}$ | 1 | 5 | 6 | - | 3 | 4 | 7 | 6 |
| 5 | $v^1_{23}$ | 5 | 9 | 10 | 4 | - | 6 | 4 | 3 |
| 6 | $v^2_{23}$ | 5 | 9 | 10 | 4 | 5 | - | 4 | 3 |
| 7 | $v^1_{31}$ | 1 | 5 | 6 | 2 | 3 | 4 | - | 6 |
| 8 | $v^1_{32}$ | 2 | 6 | 7 | 1 | 2 | 3 | 6 | - |

Step 4:

- Choose your favorite TSP algorithm!