# CSE 417
# Branch & Bound (pt 4)
## Advanced Examples (out of scope)

UNIVERSITY of WASHINGTON

# Reminders

- **HW9 due on Friday**
  - start early
  - program will be slow, so debugging will be slow...
  - should run in 2-4 minutes
- **Please fill out course evaluations**

# Review of previous lectures

- Complexity theory: P & NP
  - answer can be found vs checked in polynomial time
- NP-completeness
  - hardest problems in NP
- Reductions
  - reducing from Y to X proves $Y \leq X$
    - if you can solve X, then you can solve Y
  - X is NP-hard if every Y in NP is $Y \leq X$

# Review of previous lectures

Coping with NP-completeness:

1. Your problem could lie in a special case that is easy
   - example: small vertex covers (or large independent sets)
   - example: independent set on trees

2. Look for approximate solutions
   - example: Knapsack with rounding
     > switch to n x V (from n x W) table: store min weight, not max value
     > round V's (up) even if they don't have a common multiple
       - want approximate values **not** approximate weights

# Review of previous lectures

3. Look for “fast enough” exponential time algorithms
   - example: faster exponential time for 3-SAT
     - > 10k+ variables and 1m+ clauses solvable in practice
     - > (versus <100 variables with brute force solution)
   - example: Knapsack + Vertex Cover
     - > only pay exponential time in the difficulty of the vertex cover constraints
     - > will be fast if vertex covers are small
   - branch & bound…

# Review of previous lectures

3. Look for “fast enough” exponential time algorithms
   - branch & bound
     - branch: recursive on pieces of the search space
     - bound: return immediately if global upper bound < lower bound on piece
     - global upper bound: best
     - local lower bound: remove some hard constraints
   - example: flow-shop scheduling
     - bound: let elements run on one of the machines simultaneously
   - example: TSP
     - bound 1: remove deg(u) = 2 constraint… MST
     - bound 2: remove connected constraint… 2-factor
       - model min-cost 2-factor as a min cost flow problem

# Today

- More advanced examples of all three
  - easy cases
  - approximation
  - branch & bound

- Specific examples are all (a bit) out of scope

# Outline for Today

- Tree Width ⇐
- TSP Approximation
- Integer Linear Programming

# Tree Width Motivation

> Recall: problems on trees are easy with dynamic programming

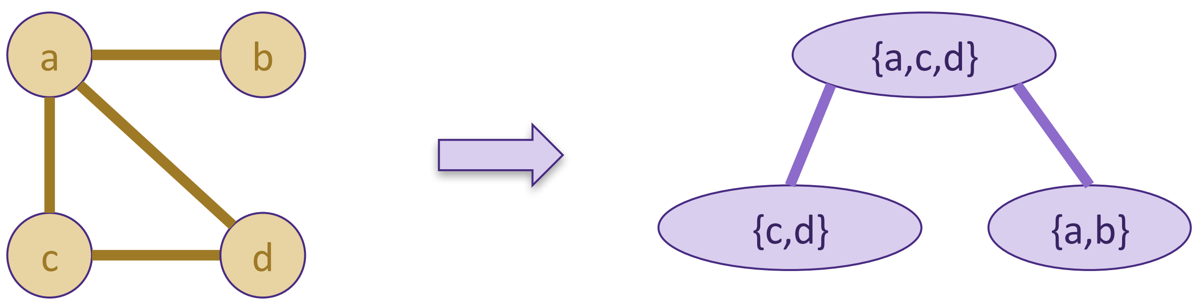
> For problems that are NP-complete on general graphs, we must accept exponential time for exact algorithms

> BUT we would like exponential in “distance from trees”

> Tree width will measure that...

# Tree Decomposition

> **Definition**: A tree decomposition of a connected graph G is a tree T whose nodes $S_1, \ldots, S_M$ are subsets of nodes of G such that
  - for every edge (u,v) of G, we have $\{u, v\} \subseteq S_i$ for some i
  - for every node u of G, the nodes $S_i$ with u in $S_i$ form a sub-tree of T

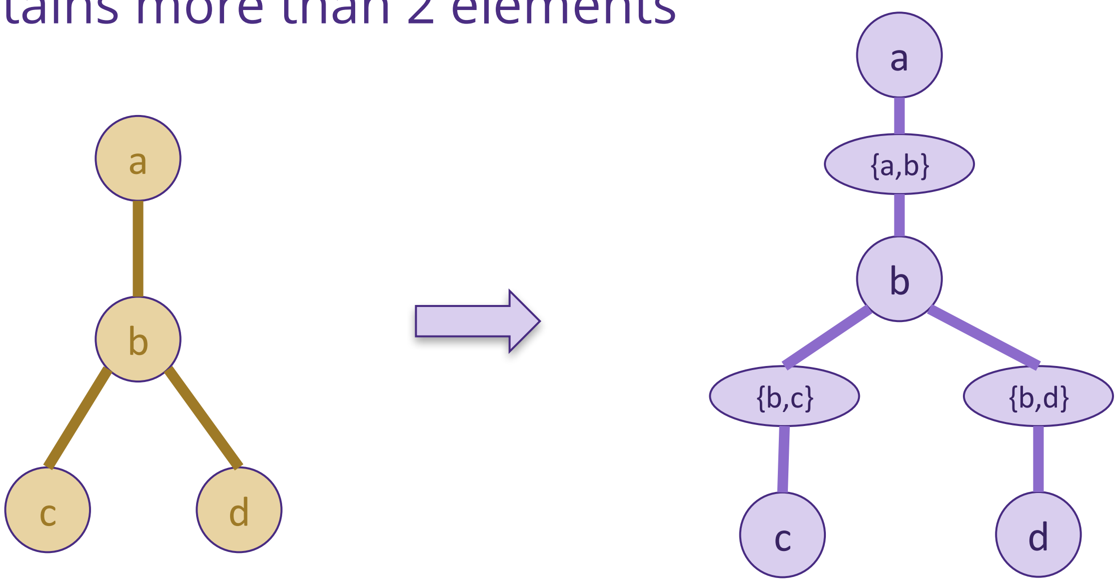


# Tree Decomposition

- **Definition**: A tree decomposition of a connected graph G is a tree T whose nodes $S_1$, ..., $S_M$ are subsets of nodes of G such that
  - for every edge (u,v) of G, we have $\{u, v\} \subseteq S_i$ for some i
  - for every node u of G, the nodes $S_i$ with $\{u\} \subseteq S_i$ form a sub-tree of T
- Example: for every G, there is a tree decomposition with a single tree node $S_1$ = N
  - every edge is a subset of $S_1$
  - for every u in G, the only node containing u is $S_1$, which is the entire tree

# Tree Decomposition

- Example: if G is a tree, it has a tree decomposition where no $S_i$ contains more than 2 elements

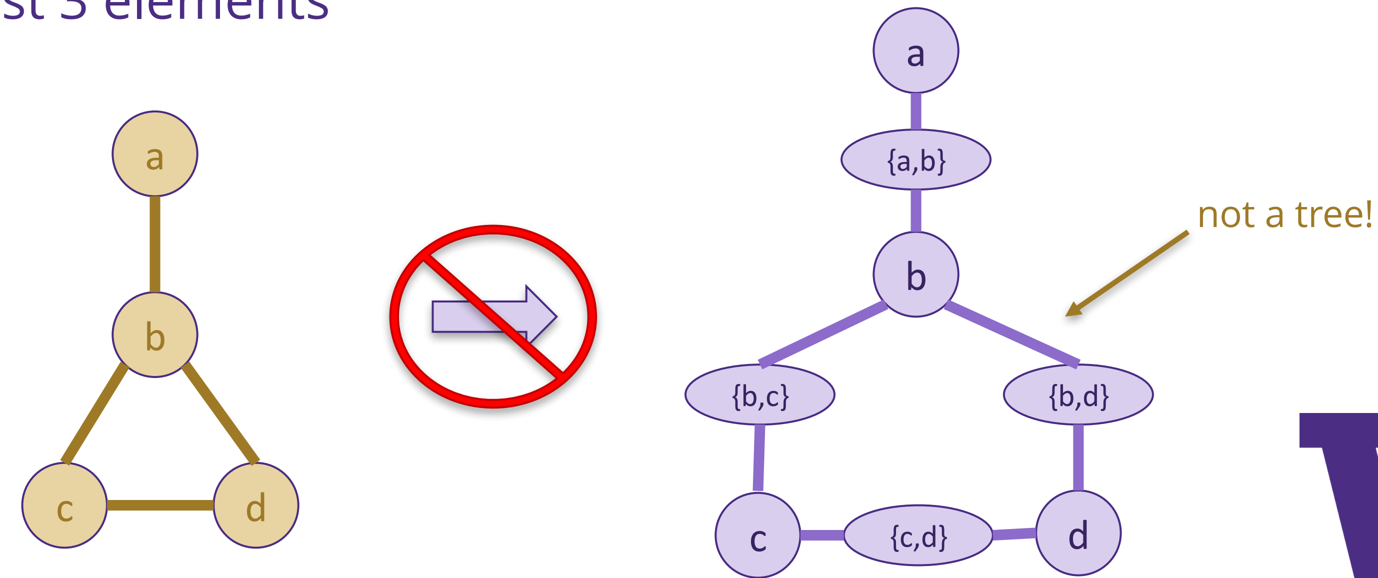



u appears in {u} and {u,v} for each (u,v) of G

they are a subtree of T (a “star” graph)

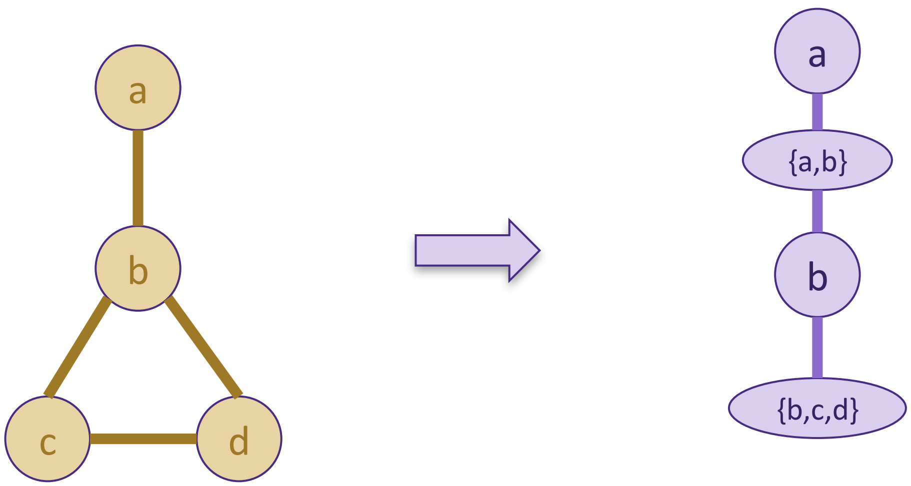
# Tree Decomposition

> **Fact**: if G is a cycle, *any* tree decomposition has some $S_i$ with at least 3 elements



# Tree Decomposition

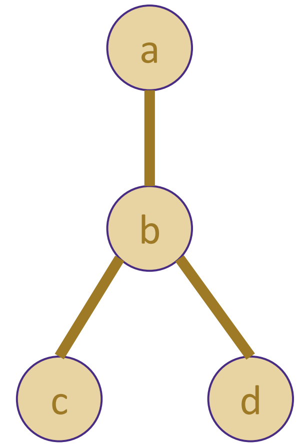
> **Fact**: if G is a cycle, *any* tree decomposition has some $S_i$ with at least 3 elements



# Tree Width

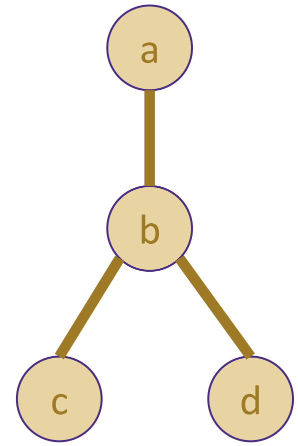
- **Definition**: Let T be a tree decomposition of G with nodes $S_1$, ..., $S_M$. The width of T is max $|S_i|$ – 1.
- **Definition**: The tree width of G is the minimum width of any tree decomposition of G.
- **Proposition**: The tree width of G is 1 iff G is a tree.

# Separators



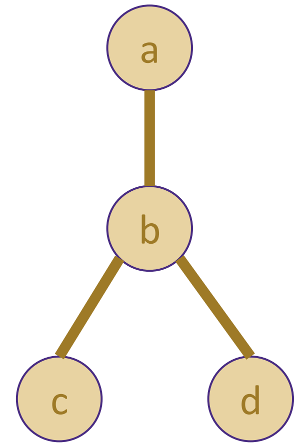

- Recall: removing any non-leaf node from a tree disconnects the graph…
- **Proposition**: Let T be a tree decomposition of G with nodes $S_1$, …, $S_M$. For any non-leaf $S_i$ in T, removing every u in $S_i$ from G disconnects the graph.
  - let $S_j$ and $S_k$ be two other nodes of T
  - if u in $S_j$ and u in $S_k$, then we must have u in $S_i$
    - tree nodes containing u form a (connected) sub-tree
    - only path from $S_j$ to $S_k$ in T goes through $S_i$

# Separators




- Recall: removing any non-leaf node from a tree disconnects the graph…
- **Proposition**: Let T be a tree decomposition of G with nodes $S_1$, …, $S_M$. For any non-leaf $S_i$ in T, removing every u in $S_i$ from G disconnects the graph.
  - let $S_j$ and $S_k$ be two other nodes of T
  - if u in $S_j$ and u in $S_k$, then we must have u in $S_i$
  - so each disconnected piece of T contains disjoint nodes of N – $S_i$
    - (removing $S_i$ from T disconnects T since it is a tree)
    - only nodes in common are those of $S_i$ that we removed

# Separators

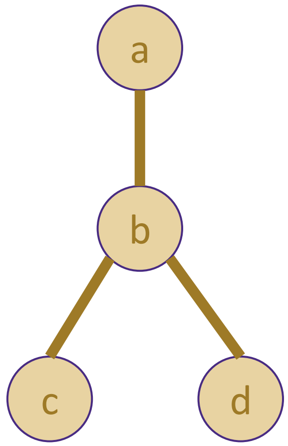



> Recall: removing any non-leaf node from a tree disconnects the graph…

> **Proposition**: Let T be a tree decomposition of G with nodes $S_1$, …, $S_M$. For any non-leaf $S_i$ in T, removing every u in $S_i$ from G disconnects the graph.

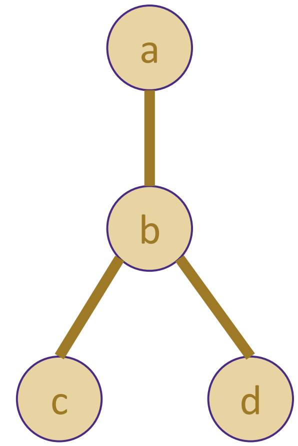
- let $S_j$ and $S_k$ be two other nodes of T
- each disconnected piece of T contains disjoint nodes from $N - S_i$
- every edge (u,v) of G appears as $\{u,v\} \subseteq S_t$ for some t
- so any such edge with $\{u,v\} \subseteq N - S_i$ is between appearing in the same disconnected piece of T

# Separators




- Recall: removing any non-leaf node from a tree disconnects the graph…
- **Proposition**: Let T be a tree decomposition of G with nodes $S_1$, …, $S_M$. For any non-leaf $S_i$ in T, removing every u in $S_i$ from G disconnects the graph.
  - let $S_j$ and $S_k$ be two other nodes of T
  - each disconnected piece of T contains disjoint nodes from $N - S_i$
  - any edge (u,v) of G with $\{u,v\} \subseteq N - S_i$ is between appearing in the same disconnected piece of T
  - so subgraphs on nodes of each piece of T are disconnected

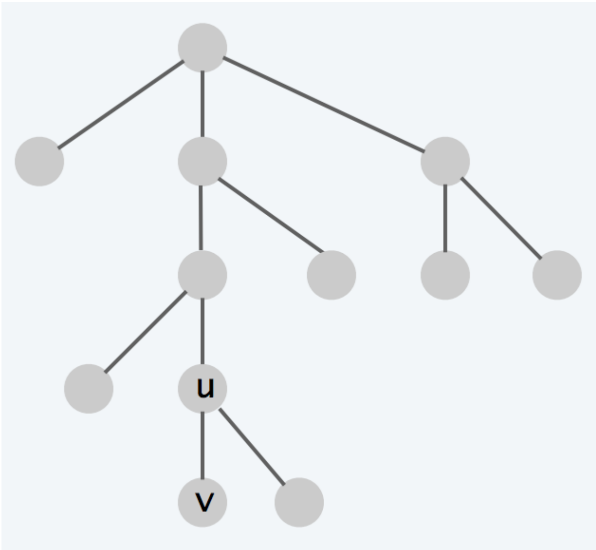
# Separators



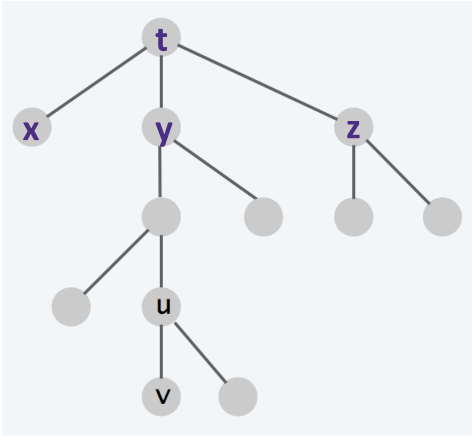

- Intuition: problems are easy on trees because the children of a node are independent given what is happening in the parent
  - DP only needs to consider the cases of what the parent might have
- For general graphs, we get the same property, but we may need to consider what is happening in multiple nodes
  - that number is the tree width

# Recall: Independent Set on a Tree

> **Independent Set**: Given graph G and number k, find a subset of k nodes such that no two are connected by an edge



# Independent Set on a Tree




- Apply dynamic programming…
  - optimal solution on tree rooted at t = larger of
    optimal solution with t excluded
    (optimal solution to which t can be legally added) + 1
  - optimal solution with t excluded =
    (opt solution on x) + (opt solution on y) + (opt solution on z)
    - no problem from edges (t,x), (t,y), (t,z) since t is not included
  - optimal solution with t included =
    (opt solution on x with x excluded) +
    (opt solution on y with y excluded) +
    (opt solution on z with z excluded)
    - no problem from edges (t,x), (t,y), (t,z) since x, y, z not included

# Independent Set and Tree Width

- Only consider two possibilities for each subtree: whether or not the root node is included in the solution
- Given node $S_i$ from a tree decomposition, we will consider every option for which subset of those are included
  - $2^k$ total, where $k = |S_i|$
  - k is bounded by the 1 + tree width of the graph
  - this is O(1) if tree width is O(1)

# Independent Set and Tree Width

- Apply dynamic programming…
  - optimal solution on subtree rooted at $S_i$ = largest of
    optimal solution at $S_i$ with X included and ($S_i$ – X) excluded
    (over each subset X of $S_i$ that are independent in G)
  - optimal solution at $S_i$ with X included and Y excluded = |X| +
    sum over the children $S_j$ of $S_i$ of
    optimal solution on subtree rooted at $S_j$
    with X' included and Y' excluded
    where X' and Y' are consistent with X and Y, respectively
    - all choices for subtree rooted at $S_j$ can be made independently except for choices on nodes also appearing in $S_i$
      - recall: removing $S_i$ disconnects the graph into independent pieces
    - (consistent meaning, e.g., $X \cap S_i \cap S_j = X' \cap S_i \cap S_j$)

# Independent Set and Tree Width

- Running time in terms of tree width, k
  - table size is $M\,2^k$
  - time per node is #children · $k\,2^k$
  - total time is $O(k\,M\,(2^k)^2)$
    - this can be easily optimized down to $O(k\,M\,2^k)$

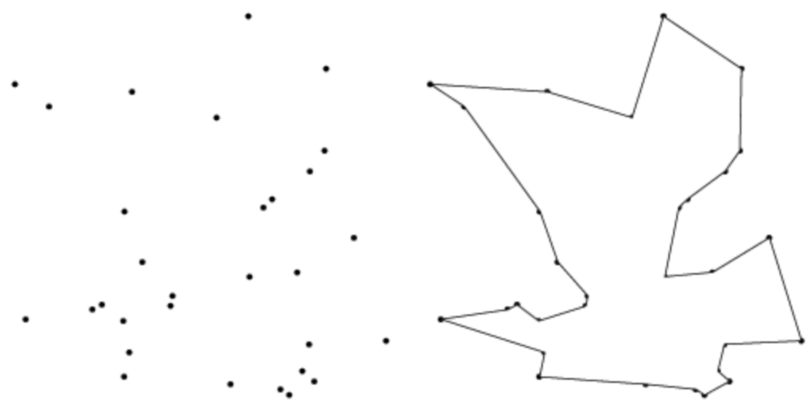
- This is linear time for fixed k

# Outline for Today

> Tree Width
> TSP Approximation
> Integer Linear Programming

# Traveling Salesperson Problem

- **Traveling Salesperson Problem (TSP)**: Given weighted graph G and number v, find a Hamiltonian cycle of minimum length
  - cycle is Hamiltonian if it goes through each node exactly once



from http://mathworld.wolfram.com/TravelingSalesmanProblem.html

# TSP Special Cases

- (symmetric) TSP

- Metric TSP: distances form a metric space
  - satisfy the triangle inequality: $d(a,c) \leq d(a,b) + d(b,c)$
  - (direct path a → c cannot be longer than indirect path a → b → c)

- Euclidian TSP: Euclidian distance between points
  - special case of Metric TSP

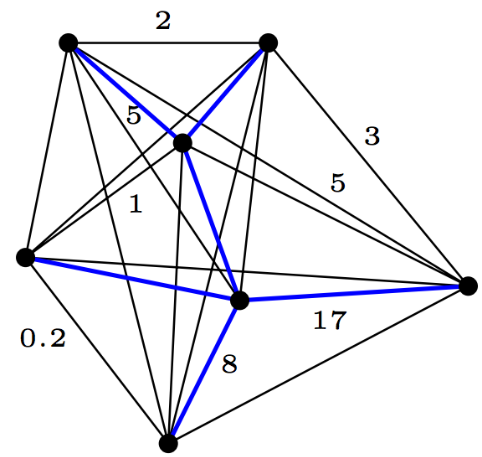
# TSP Approximations

- **Definition**: c-approximation algorithm returns a solution that is guaranteed to be within a factor c of optimal

- TSP cannot be efficiently approximated at all
  - no efficient f(n)-approximation algorithm for any computable function f

- Sanjeev Arora (more later) helped prove the PCP theorem
- It implies that some NP-complete problems cannot be efficiently approximated to any constant factor

# Metric TSP 2-Approximation

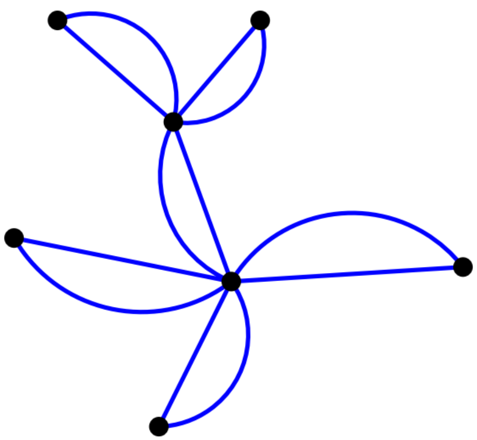



- This is not true of Metric TSP

- Simple 2-approximation
  - compute an MST in the graph
    - cost is a lower bound on the shortest Hamiltonian cycle (since a Hamiltonian cycle is also a ST + an extra edge)

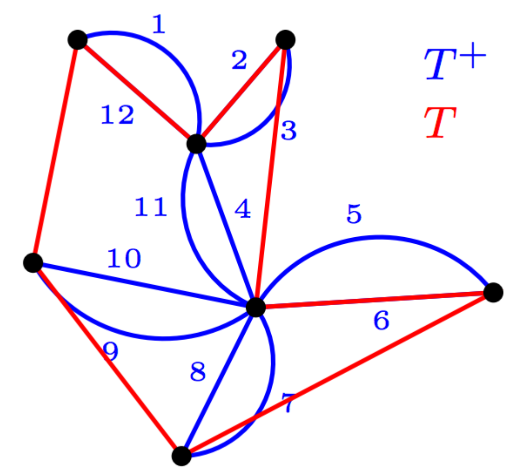
pictures from
https://graphics.stanford.edu/courses/cs468-06-winter/Slides/steve_tsp_ptas_winter.pdf

# Metric TSP 2-Approximation



- This is not true of Metric TSP
- Simple 2-approximation
  - compute an MST in the graph
    - cost is a lower bound on the shortest Hamiltonian cycle (since a Hamiltonian cycle is also a ST + an extra edge)
  - tour around the tree uses every edge twice
    - cost is twice MST cost, so at most twice optimal value
  - short-cutting to avoid re-visiting nodes cannot increase cost

# Metric TSP 2-Approximation



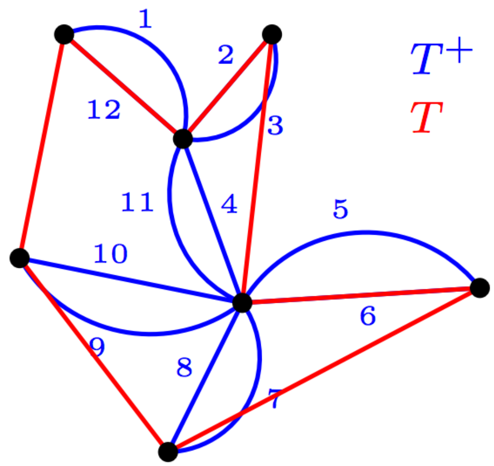

> This is not true of Metric TSP

> Simple 2-approximation
  - compute an MST in the graph
    > cost is a lower bound on the shortest Hamiltonian cycle (since a Hamiltonian cycle is also a ST + an extra edge)
  - tour around the tree uses every edge twice
    > cost is twice MST cost, so at most twice optimal value
  - short-cutting to avoid re-visiting nodes cannot increase cost
    > triangle inequality: direct path cannot be longer

# Metric TSP 3/2-Approximation

- Improved algorithm due to Christofides
- **Idea**: try to find the optimal way to turn this MST into a tour
  - in principle, that could be efficiently computable
    - no reason to think the optimal solution can be found that way
    - i.e., we haven't proved P = NP
  - actual algorithm will not quite do that

# Metric TSP 3/2-Approximation



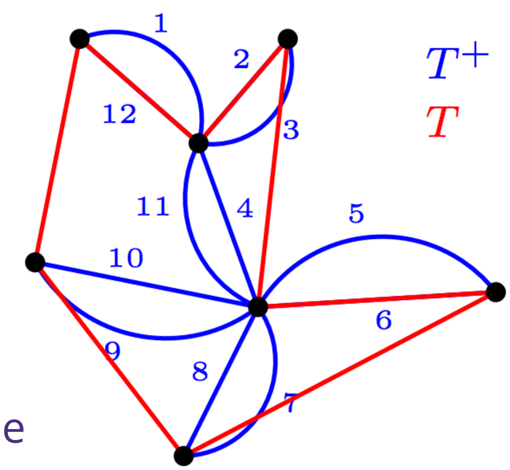

- Difficulty with the tree is odd-degree nodes
  - e.g., leaf nodes
  - they require us to return to the parent again

- **Fact**: if every node has even degree, then there is a cycle that uses every edge exactly once
  - called an “Euler tour”
  - can build a path that never reuses an edge… must return to start
    - becomes odd degree after using incoming edge, so 1+ edges left
  - avoid leaving out any nodes by never using an edge whose removal would disconnect the graph

# Metric TSP 3/2-Approximation

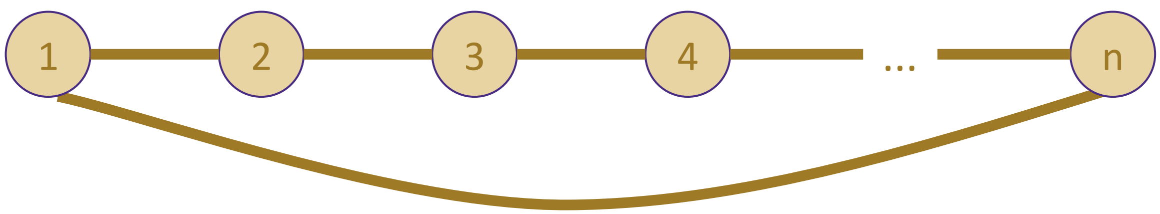
> Difficulty with the tree is odd-degree nodes
  – e.g., leaf nodes
  – they require us to return to the parent again

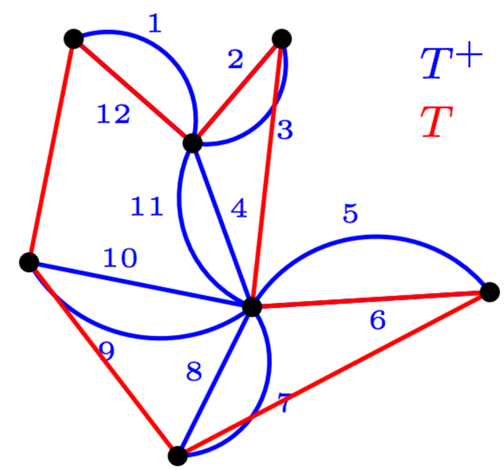
> **Idea**: find a matching of the odd-degree nodes
  – adding these to the graph makes every edge even degree
  – hence, there is an Euler tour
  – if any nodes are visited more than once, short-cutting out the re-visits can only decrease the cost



# Metric TSP 3/2-Approximation

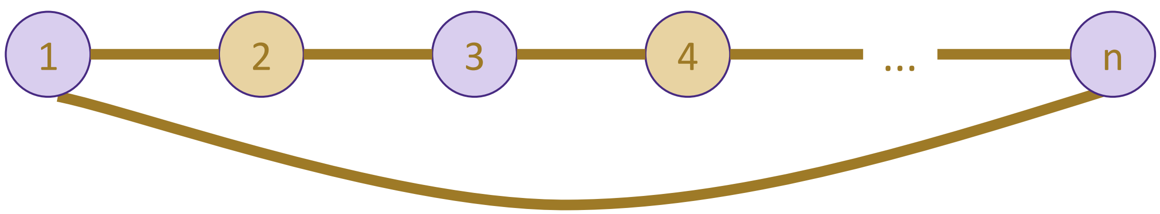
- **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  - consider the min cost tour…

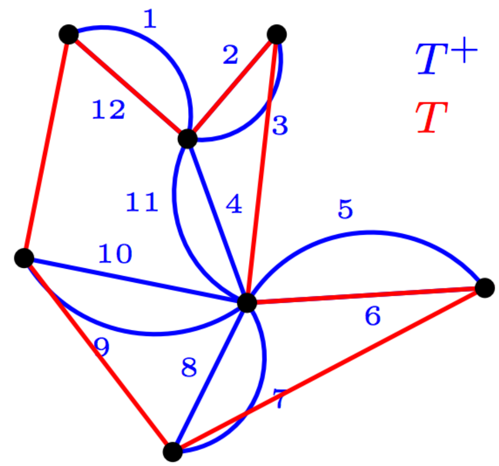


# Metric TSP 3/2-Approximation

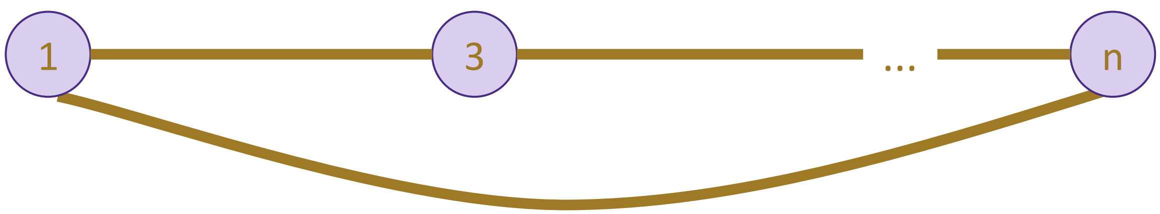
> **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  - consider the min cost tour
  - some of these are our odd degree nodes

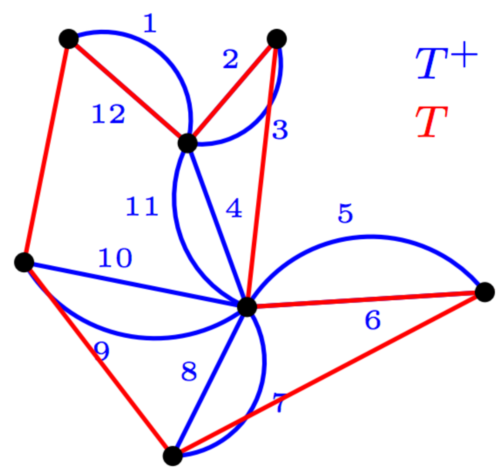


# Metric TSP 3/2-Approximation

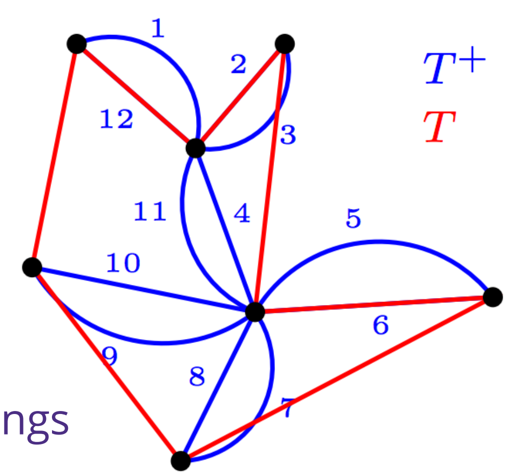
> **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  - consider the min cost tour
  - some of these are our odd degree nodes
  - short-cutting out the others can only decrease cost

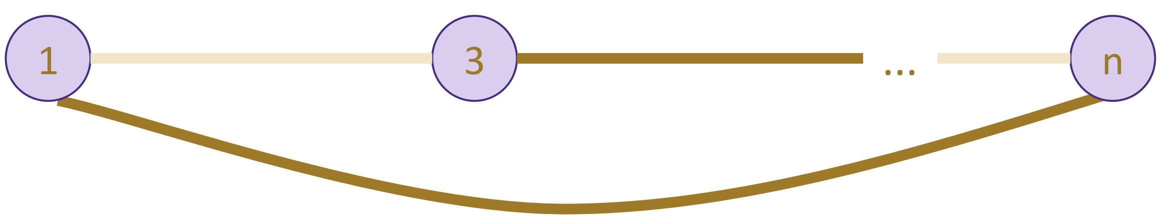


# Metric TSP 3/2-Approximation

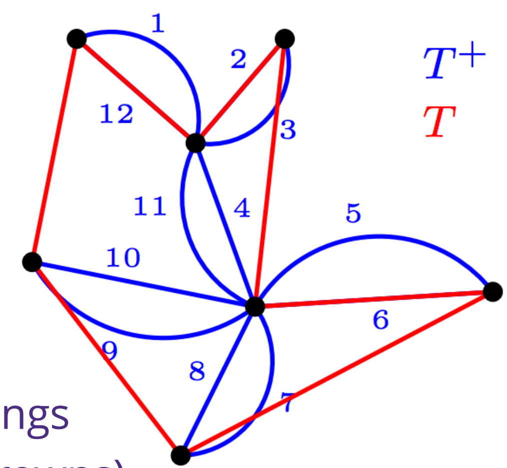
> **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  - consider the min cost tour
  - some of these are our odd degree nodes
  - short-cutting out the others can only decrease cost
  - splitting every other edge into two sets gives two matchings

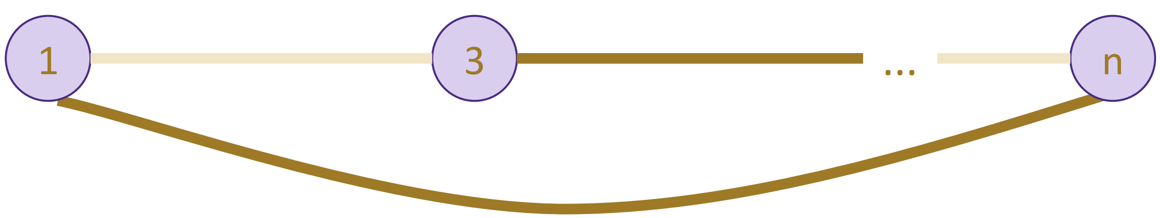


# Metric TSP 3/2-Approximation

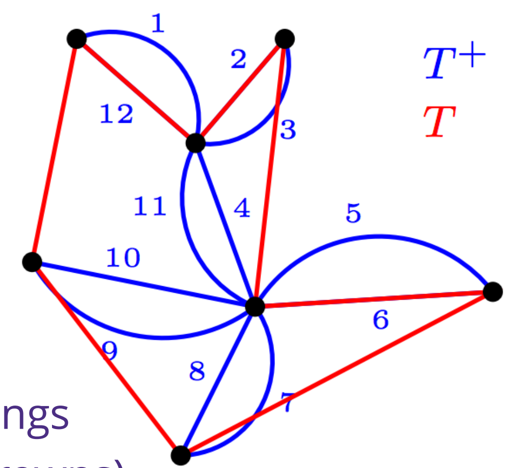
> **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  - consider the min cost tour
  - some of these are our odd degree nodes
  - short-cutting out the others can only decrease cost
  - splitting every other edge into two sets gives two matchings
  - total length = (length of dark browns) + (length of light browns)



# Metric TSP 3/2-Approximation

> **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour
  – consider the min cost tour
  – some of these are our odd degree nodes
  – short-cutting out the others can only decrease cost
  – splitting every other edge into two sets gives two matchings
  – total length = (length of dark browns) + (length of light browns)
  – min(length of dark browns, length of light browns)
    ≤ (1/2) total length
    ≤ (1/2) length of min cost Hamiltonian cycle
      > due to shortcutting out the even degree nodes



# Metric TSP 3/2-Approximation

- **Theorem**: min cost matching of odd-degree nodes has cost at most 1/2 of min cost tour

- Algorithm returns shortcutting of (MST + matching of odd degrees)
  - total cost is at most that of MST + min cost matching of odd degree nodes
  - cost of MST ≤ min cost tour
  - min cost matching of odd degrees ≤ (1/2) min cost tour
  - cost of result ≤ (3/2) min cost tour

- (In practice, results are typically off by 10-15%)

# Metric TSP 3/2-Approximation

- How do we actually compute a min-cost matching of the odd degree nodes?

- **Q**: Is this a network flow problem?
- **A**: No
  - that is only true of bipartite graphs

- Nonetheless, efficient algorithms exist
  - can be solved in $O(n^3)$ time

# Euclidian TSP

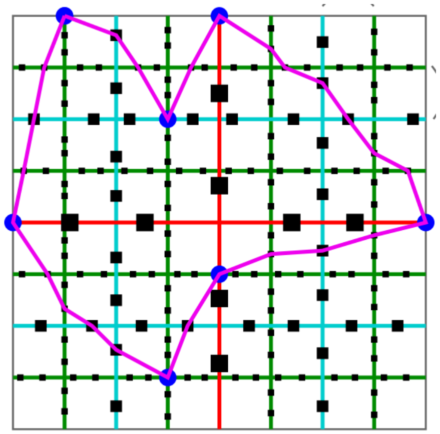
- TSP cannot be approximated at all

- Metric TSP has 3/2-approximation
- BUT there is no $(1 + \varepsilon)$-approximation for some $\varepsilon > 0$
  - best bound is $\varepsilon > 0.008$

- Arora: Euclidian TSP has $(1 + \varepsilon)$-approximation for any $\varepsilon > 0$
  - depends *exponentially* on $1/\varepsilon$ (PTAS not FPTAS)

# Euclidian TSP

- Arora: Euclidian TSP has (1 + ε)-approximation for any ε > 0
  - depends *exponentially* on 1/ε (PTAS not FPTAS)



- Construction:
  - moves cities to points on a grid
  - considers only paths going through mid-points of gridlines

- Algorithm is complicated **dynamic program**
- Correctness proof is also complicated
  - must use facts about Euclidian distance beyond triangle-inequality

# Outline for Today

- > Tree Width
- > TSP Approximation
- > Integer Linear Programming

# Linear Programs

- A linear programming problem asks you to minimize a linear function subject to linear equality and inequality constraints
- Example (from "Network Flows"):

$$
\begin{array}{ll}
\text{minimize} & x_1 + 2\,x_2 - x_3 + x_4 + 3\,x_5 \\
\text{subj. to} & x_2 + x_4 + x_5 \geq 5 \\
 & x_1 + x_2 + x_5 \geq 12 \\
 & x_1 + x_2 + x_3 \geq 10 \\
 & x_1 + x_2 + x_3 \geq 6 \\
\text{and} & x_1, x_2, x_3, x_4, x_5 \geq 0
\end{array}
$$

# Integer Linear Programs

- An integer linear programming problem asks you to find the best solution to a linear program where all $x_i$'s are integers

- Problem is NP-complete even if each is $x_i$ is restricted to {0,1}
  - potentially still $2^n$ solutions to consider

# 0/1 Integer Linear Programs

- Branch:
  - nodes set {0,1} values for a subset of the $x_i$'s
    - e.g, [$x_1$=0, $x_5$=1, $x_2$=1]
  - branching factor will be 2
    - previous + [$x_i$=0] and previous + [$x_i$=1]

- **Q**: How do we get a lower bound on a 0/1 Integer LP?
- **A**: Drop the integrality constraints
  - rest is a normal LP that can be solved in polynomial time
  - if answer is not integral, branch on a fractional $x_i$

# 0/1 Integer Linear Programs

- Child nodes add additional $x_i = b_i$ constraints to the LP
- This approach works very well in practice
  - (note: need to use an LP solver that returns integer solutions when optimal)
- State-of-the-art solvers add some other techniques as well
  - but B&B is at the core