CSE 417
Branch & Bound (pt 4)
Branch & Bound

UNIVERSITY *of* WASHINGTON

Reminders

- HW8 due today
- HW9 will be posted tomorrow
  - start early
  - program will be slow, so debugging will be slow...

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 ≤ X
    - if you can solve X, then you can solve Y
  - X is NP-hard if every Y in NP is Y ≤ X

Review of previous lectures

more generally, only pay for distance from easy cases

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

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
   - example: register allocation
     - > model as a graph coloring problem
     - > use an approximation that works well on easy instances
     - > exponential time in distance from easy instances
   - branch & bound…

Outline for Today

- Branch & Bound
- Flow-Shop Scheduling
- Traveling Salesperson
- Integer Linear Programming

Generic Optimization Problem

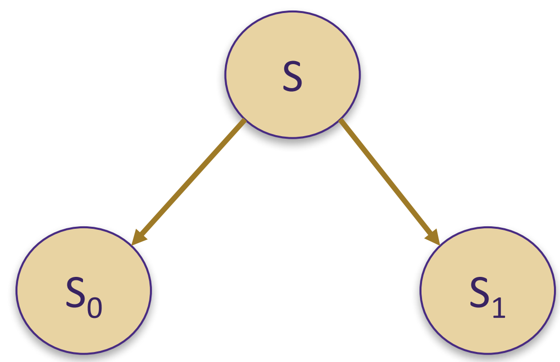
- **Generic Problem**: Given a cost function c(x) and (somehow) a set of possible solutions S, find the x in S with minimum c(x).
  - S must be described implicitly since it can be exponentially large

- Example: Knapsack
  - actual input is list of items: (w_i, v_i) pairs
  - S is the set of all subsets of the items
  - c(x) = –(sum of the values of the items)

Generic Optimization Problem

- **Generic Problem**: Given a cost function c(x) and (somehow) a set of possible solutions S, find the x in S with minimum c(x).
  - S must be described implicitly since it can be exponentially large
- Example: Traveling Salesperson Problem
  - actual input is a weighted graph
  - S is the set of all Hamiltonian cycles on the given nodes
  - c(x) is the sum of the edges along the cycle

Brute Force Search



> Enumerate solutions as leaves of a tree:
  - internal nodes split solution space into 2+ parts
  - e.g., root node splits S into S_0 and S_1

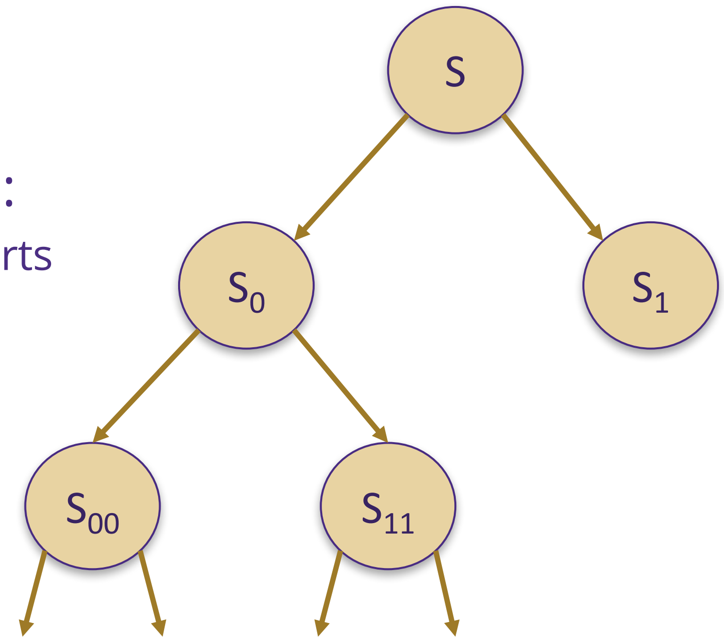
> Example: Knapsack
  - S0 = solutions that do not include item x_n
  - S1 = solutions that include item x_n

> Example: TSP
  - S_0 = Hamiltonian cycles not including edge (u,v)
  - S_1 = Hamiltonian cycles including edge (u,v)

Brute Force Search

> Enumerate solutions as leaves of a tree:
  - internal nodes split solution space into 2+ parts
  - e.g., root node splits S into S_0 and S_1
  - e.g., S_0 splits into even smaller S_{00} and S_{01}

> Leaves are nodes with only 1 solution



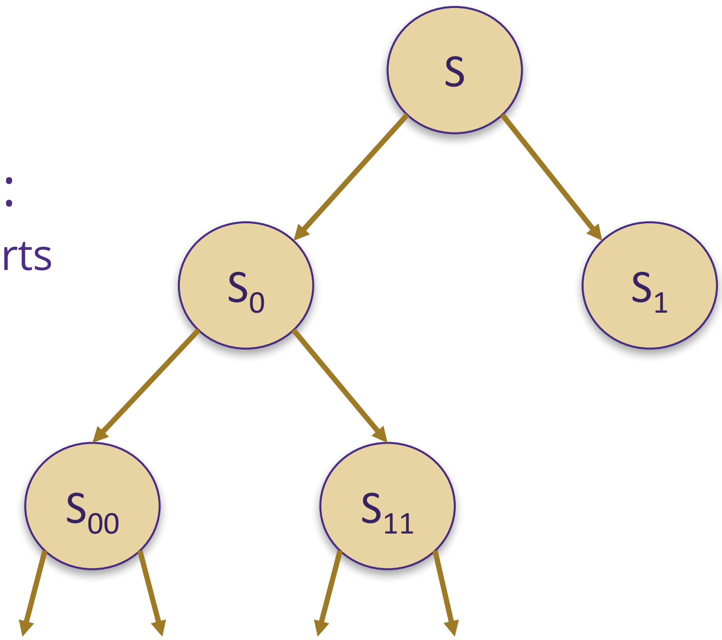
Brute Force Search

> Enumerate solutions as leaves of a tree:
  – internal nodes split solution space into 2+ parts
  – leaves are nodes with only 1 solution

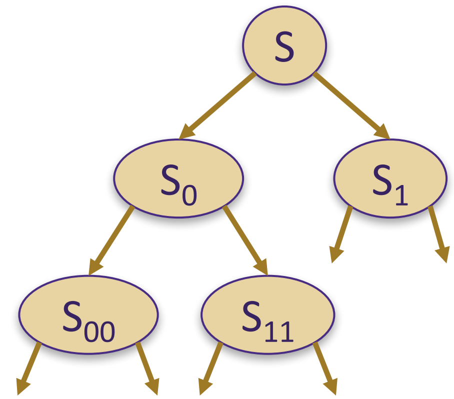
> Can implement this recursively
  – nodes correspond to recursive calls
  – pass along choices made so far

> Leaf node returns cost of its 1 solution, c(x)

> Internal node returns minimum cost from its children



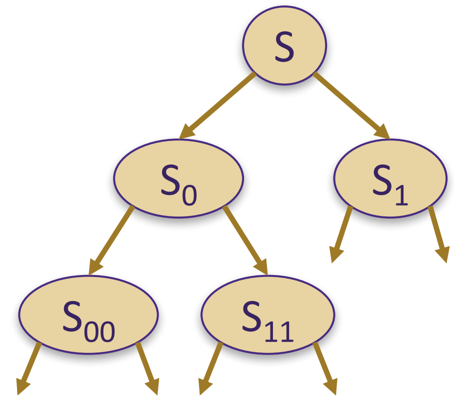
Brute Force Search



- Running time is O(|S|) · time per node
  - |S| part is exponentially large

- We need to find a way to avoid exploring all nodes...

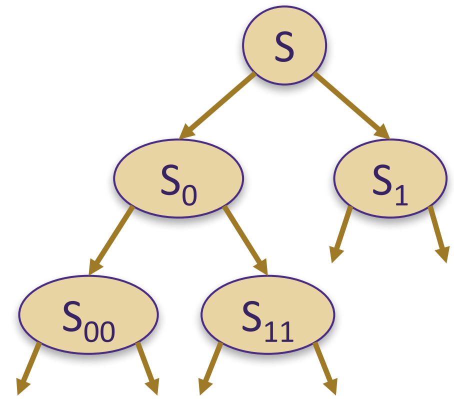
Branch & Bound



- Idea for avoiding skipping node for S_x:
  - get an upper bound, U, on the minimum solution value over all of S
  - get a lower bound, L, on the minimum solution value over just S_x
  - if $U < L$, then we can skip the node (and all children)

- opt solution value $\leq U < L \leq$ any solution in S_x
  - no solution in S_x could be optimal

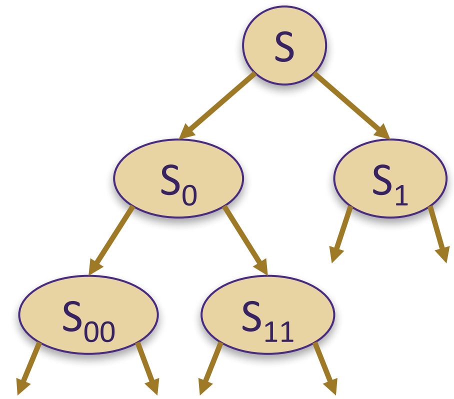
Branch & Bound



- Idea for avoiding skipping node for S_x:
  - get an upper bound, U, on the minimum solution value over <u>all of</u> S
  - get a lower bound, L, on the minimum solution value over <u>just</u> S_x
  - if U < L, then we can skip the node (and all children)

- Easy upper bound: best solution found so far
  - we know the opt solution must be at least that good as that (so opt ≤ U)
  - (can still implement recursively… store this in, say, a field of the class)

Branch & Bound



- Idea for avoiding skipping node for S_x:
  - get an upper bound, U, on the minimum solution value over all of S
  - get a lower bound, L, on the minimum solution value over just S_x
  - if U < L, then we can skip the node (and all children)

- Typical lower bound: drop hard constraints (“relaxation”)
  - opt value on new problem can only be ≤ than opt value on original problem
  - drop constraints so new problem is solvable efficiently
  - can be more than one way to do this
    - finding the best choice requires careful analysis / experimentation
    - this is where the creativity comes in

Knapsack + Vertex Cover

- **Problem**: Given a set of items $\{(w_i, v_i)\}$, a weight limit W, and a collection of pairs $\{(i, j)\}$, find the subset of items with largest total value subject to the constraints that:
  - total weight is under the limit
  - for each pair (i, j), either i or j (or both) is included

Knapsack + Vertex Cover

S = vertex covers of G
(only exponential part!)

```
KVC(G, I):
  solve knapsack on (items – I) with W – (weight of I)

  if best seen so far < knapsack value + (value of I):
    return –infinity

  else if some edge (u,v) is not covered by solution:
    return max(KVC(G – {u}, I + {u}),
               KVC(G – {v}, I + {v})

  else:
    return knapsack value + (value of T)
```

lower bound:
ignores vertex cover constraints

split into
- those that cover u
- those that cover v

(note: not a true split…
some duplicates!)

Branch & Bound in Practice

- Most successful approach for exactly solving hard problems
  - e.g., NP-complete problems
  - examples shown later: TSP & integer programming
- Example: can solve TSP instances with 10,000+ nodes
  - (that was true in 1990, and computers are 1000x faster now)
- Key point: spending **more** time per node is often faster
  - intuition: difficult part is not finding the optimal solution
    it is proving that the optimal solution is really optimal
    (i.e., ruling out all the other options)

Branch & Bound in Practice

> Can explore nodes of search tree in any order...

> Heuristic: explore the one with lowest upper bound
  - ideally, will reduce the global upper bound the fastest, reducing tree size

> OTOH, depth first search is easier to code
  - just use recursion

> In practice: DFS works just as well
  - no point trying to explore the tree in a smart way

Outline for Today

> Branch & Bound
> Flow-Shop Scheduling
> Traveling Salesperson
> Integer Linear Programming

Flow-Shop Scheduling

> **Problem**: Given a sequence of jobs 1 .. n where
> – each job has two parts that need to be run on machines A and B
> – the part on machine A must finish before starting the part on machine B
> – the machine time required for the two parts are A_i and B_i, respectively
>
> find the schedule minimizing the sum of completion times.

> Example: A is a computer and B is a printer
> – need to run the program to get the file for the printer

Flow-Shop Scheduling Example

> Consider the following inputs:

	A	B
Job 1	2 mins	1 min
Job 2	3 mins	1 min
Job 3	2 mins	3 mins

> Running 1 then 3 then 2 is optimal…
  – (this is in no way obvious…)

Flow-Shop Scheduling

- Some facts about the problem…
  - from "Combinatorial Optimization" by Papadimitriou & Steiglitz
- Flow-shop scheduling is NP-complete
  - maybe not surprising
- There exists an optimal solution where:
  - the jobs are run on on the two machines in the same order
    - no idle time on machine A
    - only idle on B waiting for previous item in order to finish
  - <u>result</u>: we can limit our search space to permutations

Flow-Shop Scheduling

- Branching (searching over permutations):
  - nodes correspond to permutations that start with a particular prefix
  - branching factor of the tree is n, not 2
- Bounding:
  - need to throw away enough constraints to make this solvable…

Flow-Shop Scheduling

- **Idea**: let multiple jobs use B simultaneously
  - dropping the constraint that jobs must run sequentially on B
  - keeping the constraint that they must run sequentially on A
  - keeping the constraint that the first part must run before the second

- Suppose the first j items are fixed so far...
  - time when $k > j$ finishes on B is $(A_1 + \ldots + A_j) + A_{j+1} + \ldots + A_k + B_k$
    - can always run B part immediately due to dropped constraint
  - first part, $A_1 + \ldots + A_j$, is always included
  - last part, B_k, is always include
  - middle part, $A_{j+1} + \ldots + A_k$, can improve with better order

Flow-Shop Scheduling

- Suppose the first k items are fixed so far…
  - time when $k > j$ finishes on B is $(A_1 + \ldots + A_j) + A_{j+1} + \ldots + A_k + B_k$
    - can always run B part immediately due to dropped constraint
  - first part, $A_1 + \ldots + A_j$, is always included
  - last part, B_k, is always include
  - middle part, $A_{j+1} + \ldots + A_k$, can improve with better order

- **Fact**: minimized if we order the elements by increasing A_i
  - one run first shows up in every sum
  - one run second shows up all but one sum
  - etc.

Flow-Shop Scheduling

> **Idea**: let multiple jobs use B simultaneously

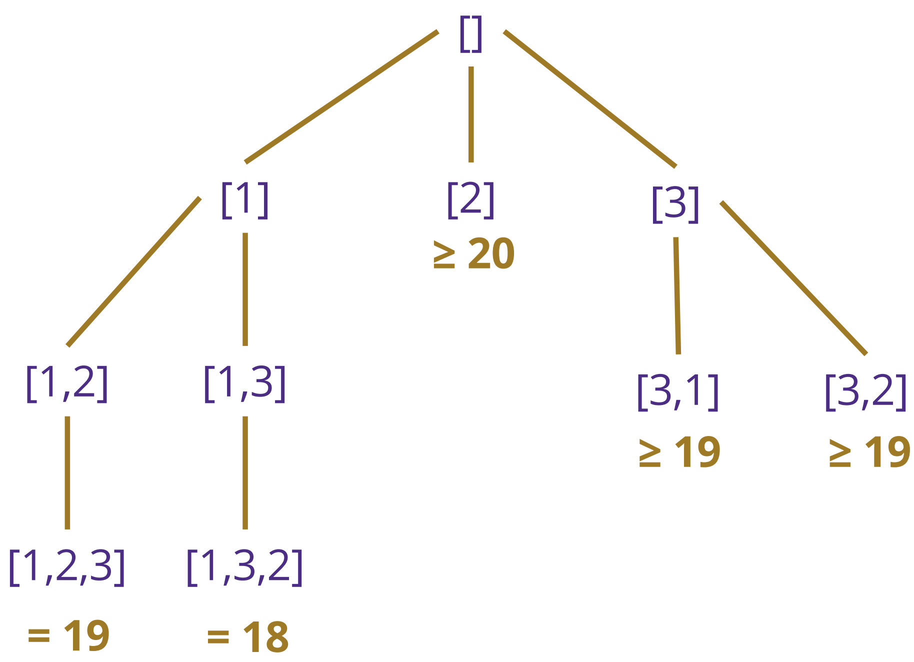
> Get a lower bound by taking the items in order of ascending A_i

> **Idea**: let multiple jobs use A simultaneously

> Get a lower bound by taking the items in order of ascending B_i

> Take the **larger** of those two bounds
  - reportedly very effective in practice

Flow-Shop Scheduling



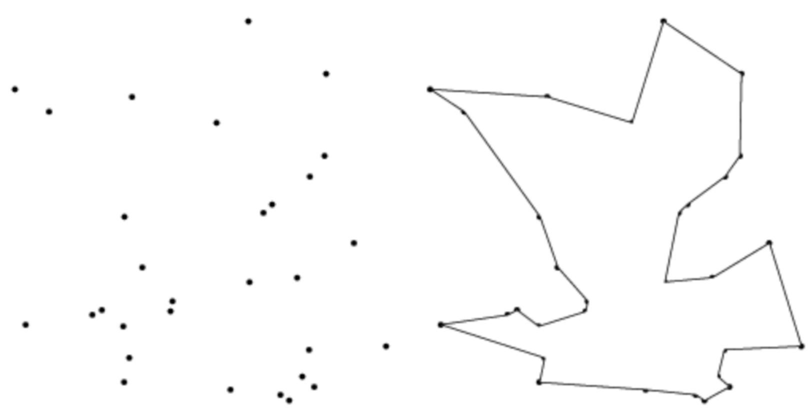
	A	B
Job 1	2 mins	1 min
Job 2	3 mins	1 min
Job 3	2 mins	3 mins

Outline for Today

- Branch & Bound
- Flow-Shop Scheduling
- Traveling Salesperson
- 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



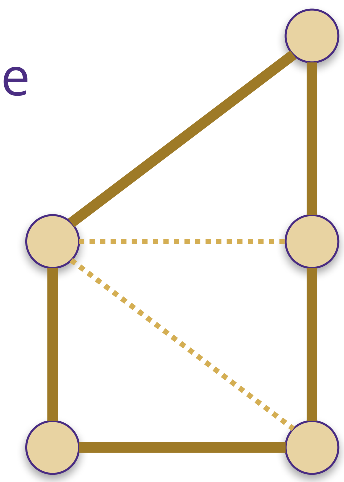
from http://mathworld.wolfram.com/TravelingSalesmanProblem.html

Traveling Salesperson Problem

- Branching:
  - nodes fix a subset of the edges to be included or excluded
  - put the edges in a fixed order
  - level i in the tree branches using the i-th edge
    - first branch is forced to use that edge
    - second branch is disallowed from using it (can remove it from the graph)

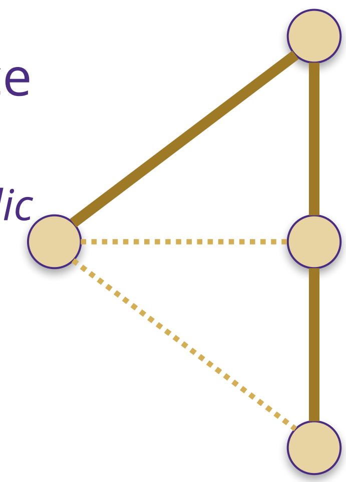
Traveling Salesperson Problem

- Bound 1: remove restriction of only using a node once
  - Hamiltonian cycle is a cycle including every node



Traveling Salesperson Problem

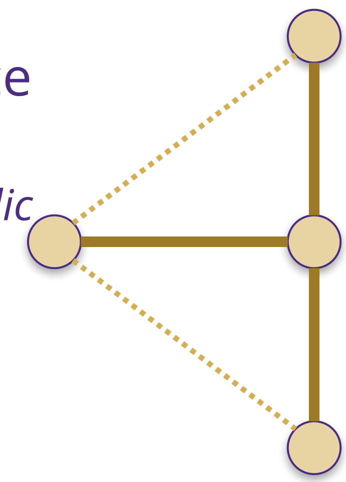
> Bound 1: remove restriction of only using a node once
  - Hamiltonian cycle is a cycle including every node
  - removing a node leaves a subgraph that is *connected and acyclic*



> **Q**: What do we call the least cost collection of edges that connect all the nodes without cycles?

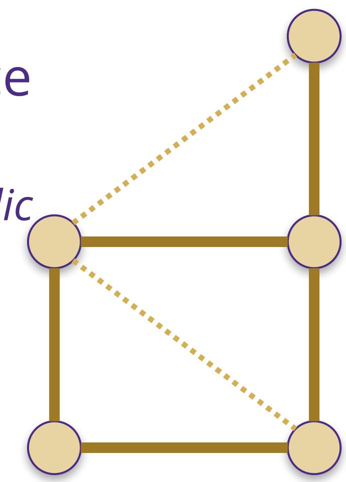
Traveling Salesperson Problem

- Bound 1: remove restriction of only using a node once
  - Hamiltonian cycle is a cycle including every node
  - removing a node leaves a subgraph that is *connected and acyclic*
- **Q**: What do we call the least cost collection of edges that connect all the nodes without cycles?
- **A**: Minimum spanning tree



Traveling Salesperson Problem

> Bound 1: remove restriction of only using a node once
  – Hamiltonian cycle is a cycle including every node
  – removing a node leaves a subgraph that is *connected and acyclic*



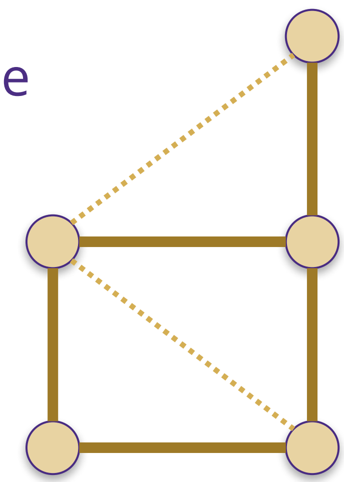
> **Q**: What do we call the least cost collection of edges that connect all the nodes without cycles?

> **A**: Minimum spanning tree

> Adding back the node & 2 edges may not be a cycle
  – there may also be multiple ways to include the node…

Traveling Salesperson Problem

> Bound 1: remove restriction of only using a node once
  - compute an MST on nodes 2 .. n
  - add node 1 by connecting it to its two closest neighbors
  - result cannot be longer than the shortest Hamiltonian cycle

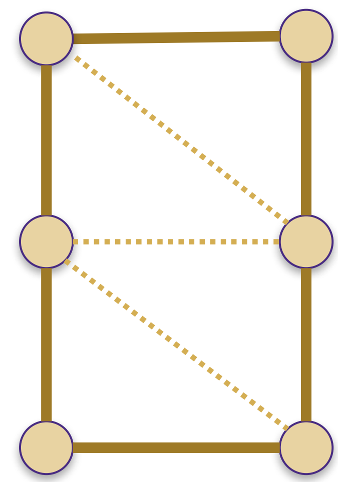


> (If we want, we can try this also with node 1 replaced by node 2, 3, ..., to see if we can improve the bound.)

> Gets more complicated once some edges are fixed
  - can modify an MST algorithm to start with some included

Traveling Salesperson Problem

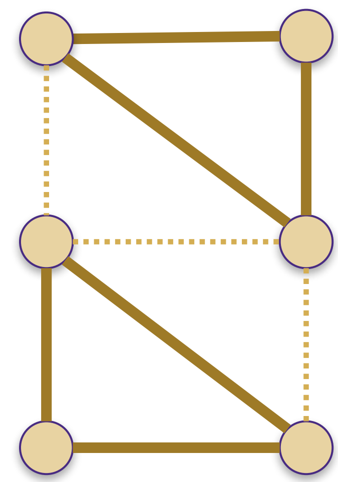
> Bound 2: remove restriction of being connected
  – Hamiltonian cycle is a connected subgraph
    where every node has exactly two incident edges



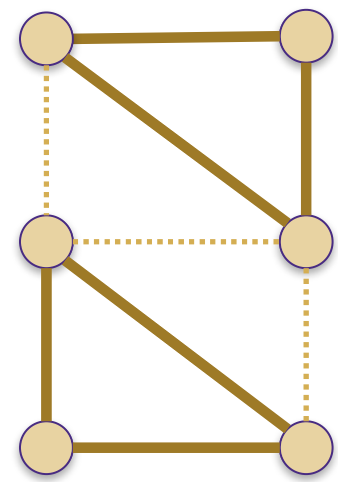
Traveling Salesperson Problem

> Bound 2: remove restriction of being connected
  – Hamiltonian cycle is a connected subgraph where every node has exactly two incident edges
  – without connectivity requirement, result may be a *collection* of disjoint cycles
  – this is sometimes called a "2-factor"

> **Q**: How do we find the least cost 2-factor?



Traveling Salesperson Problem

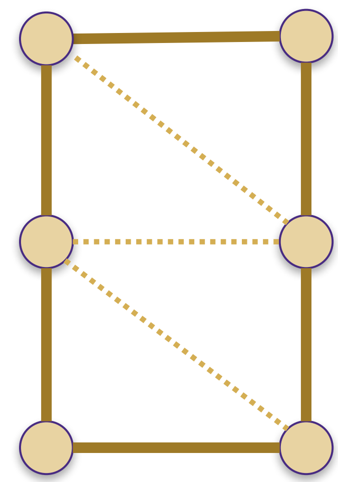


- Bound 2: remove restriction of being connected
  - Hamiltonian cycle is a connected subgraph
    where every node has exactly two incident edges
  - without connectivity requirement,
    result may be a *collection* of disjoint cycles
  - this is sometimes called a “2-factor”

- **Q**: How do we find the least cost 2-factor?
- **A**: It’s a min cost flow problem!
  - put upper and lower bounds of 1 on node capacities
  - every node has one incoming and one outgoing flow

Traveling Salesperson Problem

- Bound 2: remove restriction of being connected
  - least cost 2-factor gives a lower bound on TSP
  - easy to include the fixed edges:
    - set lower bounds on those as well
- Even though it may take $\Omega(nm)$ time to compute the lower bound, that can easily pay for itself...



Additional Results (out of scope)

- MST lower bound can be improved (Held-Karp)
  - increasing the length of every edge into u by T does not change opt cycle
    - every cycle must use 2 such edges, so all are increased by 2T
  - however, this can change the MST
  - repeatedly apply this to MST nodes with degree > 2 to eliminate them
    - stop when it's not improving much anymore

- 2-factor lower bound can be improved
  - re-write as an LP
  - add constraints to eliminate "sub-tours"
  - potentially need 2^n... and result still may be fractional

Additional Results (out of scope)

- MST lower bound can be improved (Held-Karp)
- 2-factor lower bound can be improved
- **Theorem** (Held-Karp): lower bounds produced by these two techniques are identical
  - in practice, the iterative Held-Karp approach is faster

Traveling Salesperson Problem

- Algorithm works extremely well in practice
  - solved problems with 10k+ nodes 20+ years ago
  - on one instance with 1k+ cities, searched only 25 tree nodes
    - versus a potential of $> 2^{1000}$ nodes

- Key point: more expensive lower bounds can easily pay for themselves by reducing the size of the search tree