COMP 182 Algorithmic Thinking

Of Men and Mice… Rats and Chimps, and Algorithmic Thinking

Luay Nakhleh
Computer Science
Rice University

- A biologist is interested in the evolutionary history of humans, mice, rats, and chimps.
- Given that this evolutionary history spans tens of millions of years, we need to reason about the problem computationally and solve it from data that the biologist has.

Algorithmic Thinking Questions

- To solve the biologist's problem, we need to
  - understand the input data
  - what he/she means by evolutionary history
  - if more than a single history exists, which one(s) he/she "likes"
  - ...

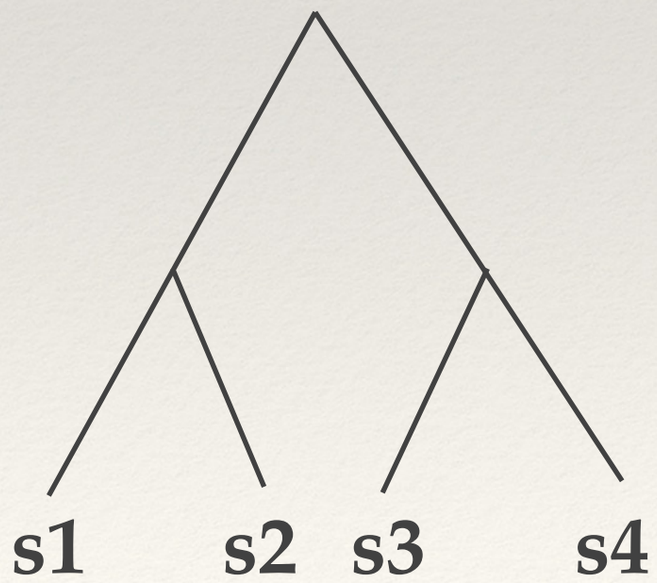
Algorithmic Thinking Questions

- What exactly is the input format?
  - a set S of n sequences of equal length, over an alphabet Σ

s1=	A	A	T
s2=	A	C	T
s3=	C	G	C
s4=	C	G	A

Algorithmic Thinking Questions

- What exactly is the output format?
  - a rooted, binary tree T, with n leaves, such that each leaf is labeled uniquely with one of the sequences in the input.



Algorithmic Thinking Questions

- So… the biologist's problem is:

Algorithmic Thinking Questions

- So… the biologist's problem is:

Input:

s1=	A	A	T
s2=	A	C	T
s3=	C	G	C
s4=	C	G	A

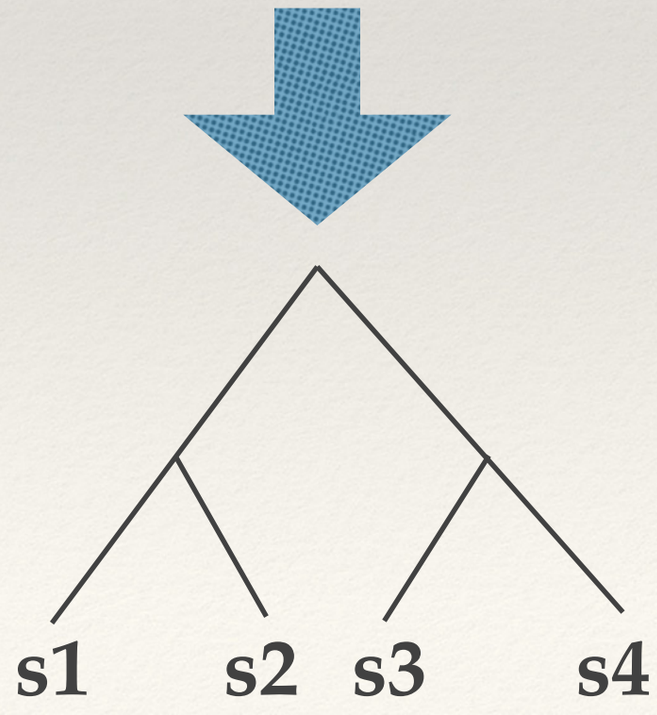
Algorithmic Thinking Questions

- So… the biologist's problem is:

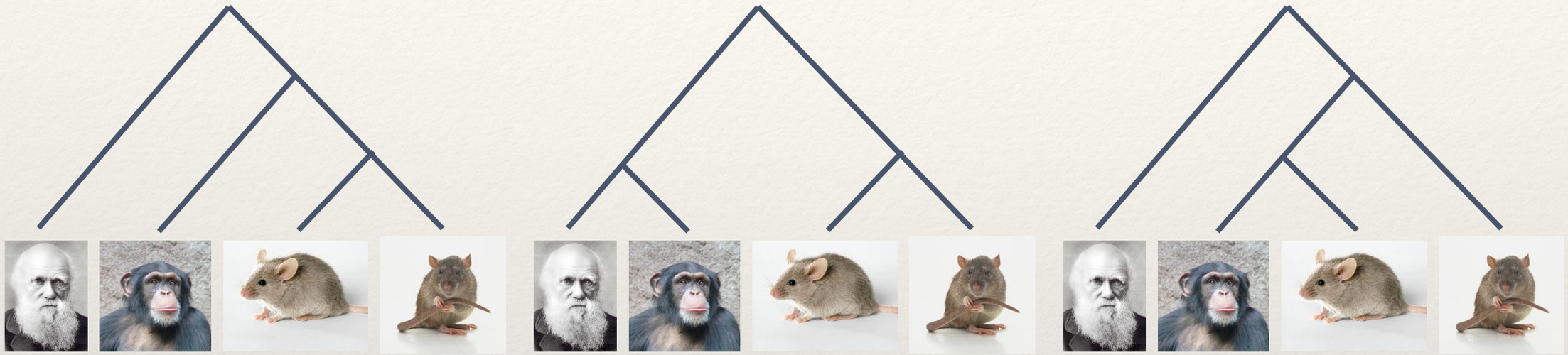
Input:

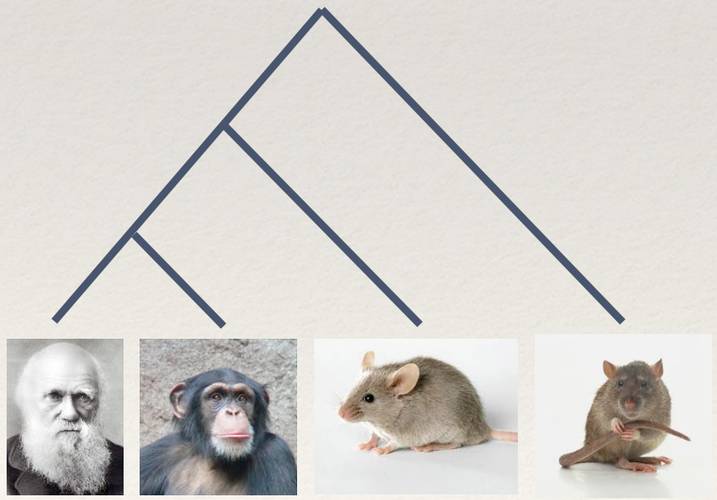
s1=	A	A	T
s2=	A	C	T
s3=	C	G	C
s4=	C	G	A

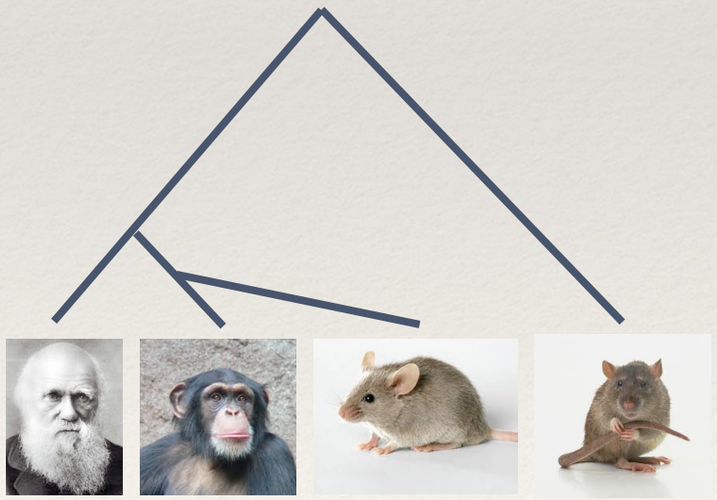
Output:



Which One?







●●●●

Which One?

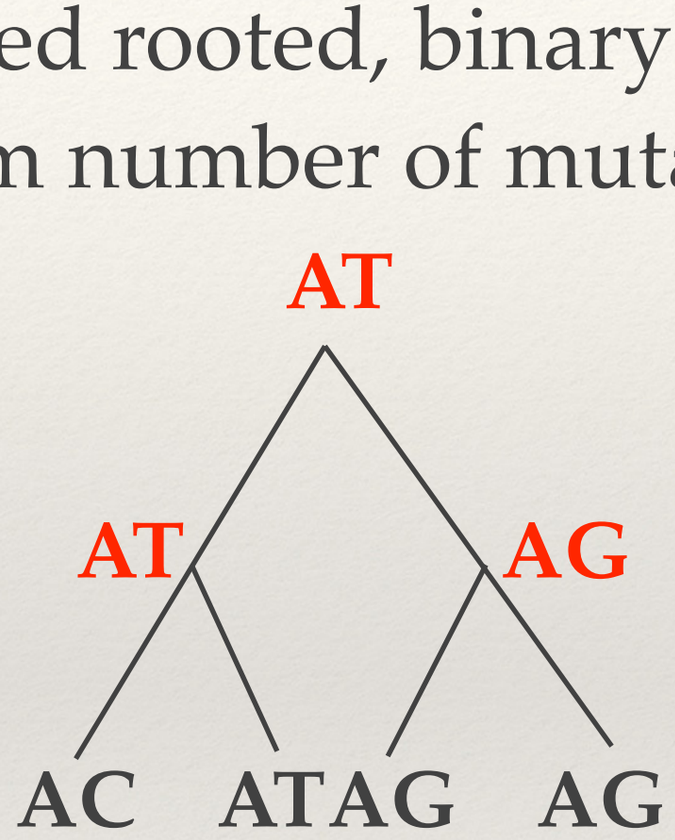
- The number of trees grows very fast with the number of leaves.
- The universe has about 10^{89} protons and has an age of about 5×10^{17} seconds. The number of trees with 60 leaves is about 5×10^{94}.
- (Don’t trust me? Do homework 5!)

Algorithmic Thinking Questions

- If there is more than a single candidate for a solution, how do we distinguish among those candidates?
  - A general technique is to define an optimality criterion Φ, and seek the candidate, or candidates, that optimize that criterion.
- What is a good optimality criterion for evolutionary trees?
  - One such criterion seeks the tree that minimizes the number of mutations required to explain the evolution of the sequences at the leaves.

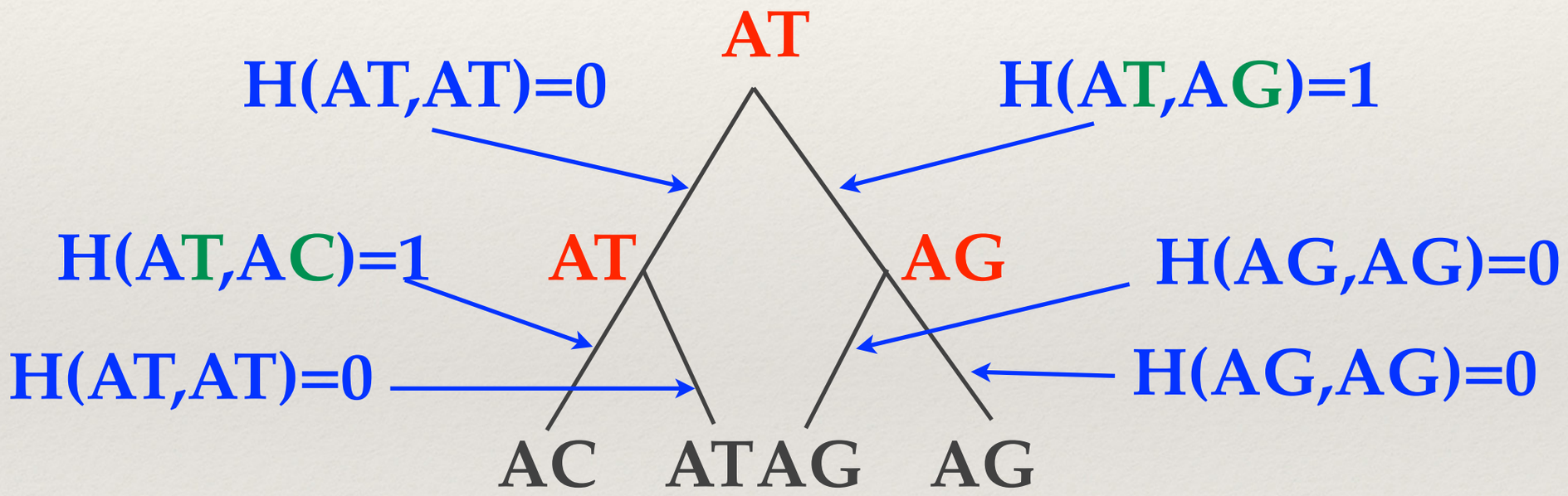
Algorithmic Thinking Questions

- Given a fully-labeled rooted, binary tree, how do we count the minimum number of mutations?



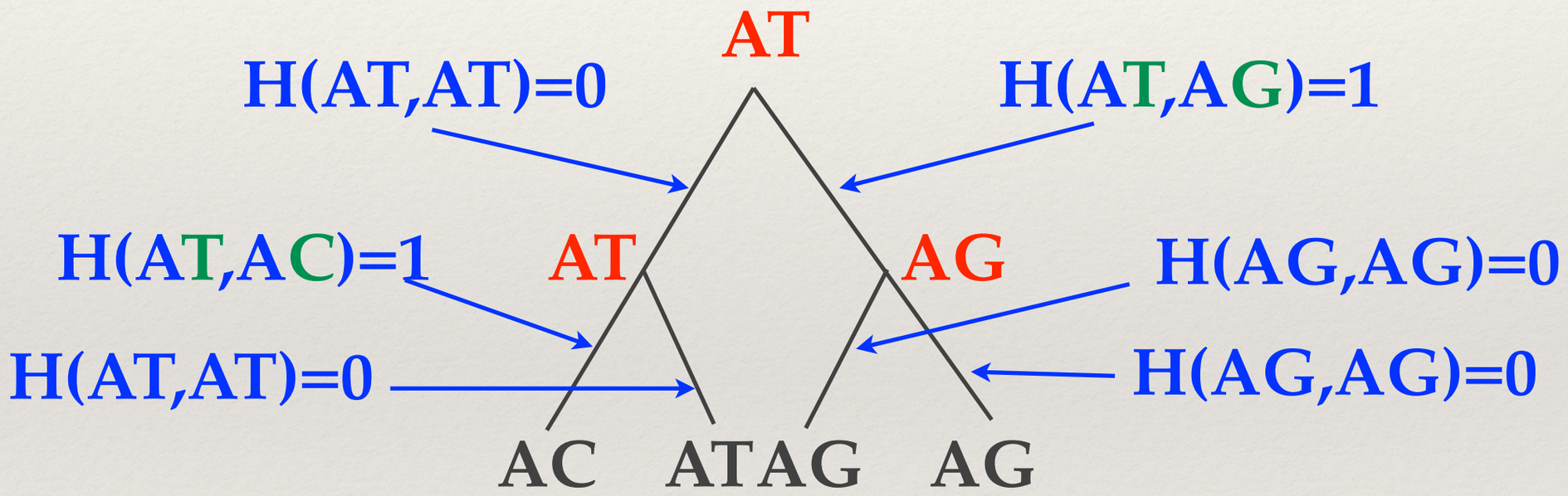
Algorithmic Thinking Questions

- Given a fully-labeled rooted, binary tree, how do we count the minimum number of mutations?



Algorithmic Thinking Questions

- Given a fully-labeled rooted, binary tree, how do we count the minimum number of mutations?



The score of the tree T with the 7 sequences S is PS(T,S)=0+1+0+1+0+0=2.

Algorithmic Thinking Questions

- More generally, given a tree T whose nodes are labeled by set S of sequences, each of the same length, the length of the tree T is

$$PS(T,S) = \sum_{(u,v)\in E(T)} H(s(u),s(v))$$

where E(T) is the set of T's edges, s(u) is the sequence labeling node u, and H(x,y) is the number of positions that sequences x and y differ at.

Algorithmic Thinking Questions

- How efficiently can we compute PS(T,S) when the sequences S are given for all nodes?
  - Answer: Easy, as you'll show us on homework 5!

Algorithmic Thinking Questions

- In practice, we have sequences only at the leaves of tree, but no sequences at the internal (non-leaf) nodes.
- In this case, given a tree T whose leaves are labeled uniquely by set S′ of sequences (each of length m), we seek set S″ of sequences to label the internal nodes, such that PS(T,S′∪S″) is minimum over all such sets S″.
- How efficiently can we solve this problem? That is, how efficiently can we find the set S″ that minimizes L(T,S′∪S″) given that we have tree T and set S′ of leaf sequences?
  - Answer: Dynamic Programming to the rescue; do homework 5!

Recall Our Main Problem

Recall Our Main Problem

Input:

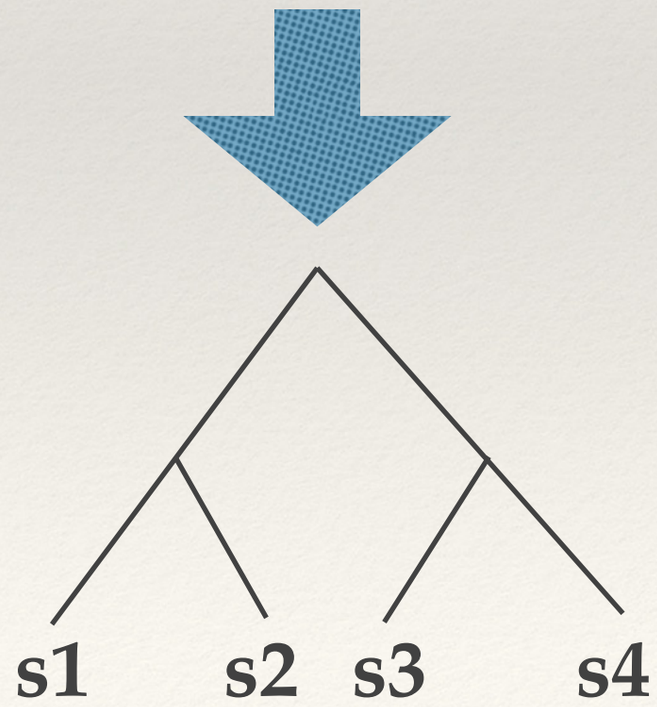
s1=	A	A	T
s2=	A	C	T
s3=	C	G	C
s4=	C	G	A

Recall Our Main Problem

Input:

s1=	A	A	T
s2=	A	C	T
s3=	C	G	C
s4=	C	G	A

Output:



Algorithmic Thinking Questions

- How can we make use of the optimization criterion, algorithms for scoring it efficiently, and methods for exploring the tree space, to solve the main problem?
  - Answer: Iterative improvement to the rescue; do homework 5!

Algorithmic Thinking

- To solve the problem, we had to
  - understand the input/output format (feasibility),
  - define a quality measure (optimality),
  - reason about the number of solution candidates,
  - devise an efficient algorithm to compute the optimality criterion,
  - try different algorithmic techniques for solving the main problem, and
  - eventually, start the implementation.

- So, what is iterative improvement?

Optimization Problems

- Computing an optimal (maximal, minimal, heaviest, lightest, shortest,...) solution from a set of feasible solution candidates.

Recall: Greedy Algorithms

- **Greedy** algorithms construct a solution to a problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step—and this is the central point of this technique—the choice made must be
  - feasible, i.e., it has to satisfy the problem's constraints
  - locally optimal, i.e., it has to be the best local choice among all feasible choices available on that step
  - irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Iterative Improvement Algorithms

- **Iterative improvement** algorithms start with a feasible solution candidate to the problem and proceeds to improve it by repeated application of some operation(s) that transforms the solution candidate.
- The improvement step typically involves a small, localized change yielding a feasible solution with an improved value of the objective function.
- When no such change improves the value of the objective function, the algorithm returns the last feasible solution as optimal and stops.

Linear Programming

- Arguably, the most important problem that can be solved by iterative-improvement algorithms is linear programming, and the algorithm is called the Simplex Method (formal description of the algorithm is beyond the scope of this course).

$$
\begin{array}{rll}
\text{maximize (or minimize)} & c_1 x_1 + \cdots + c_n x_n & \\
\text{subject to} & a_{i1} x_1 + \cdots + a_{in} x_n \leq \text{ (or } \geq \text{ or } =) \; b_i & \text{for } i = 1, \ldots, m \\
& x_i \geq 0 & \text{for } i = 1, \ldots, n
\end{array}
$$

Linear Programming

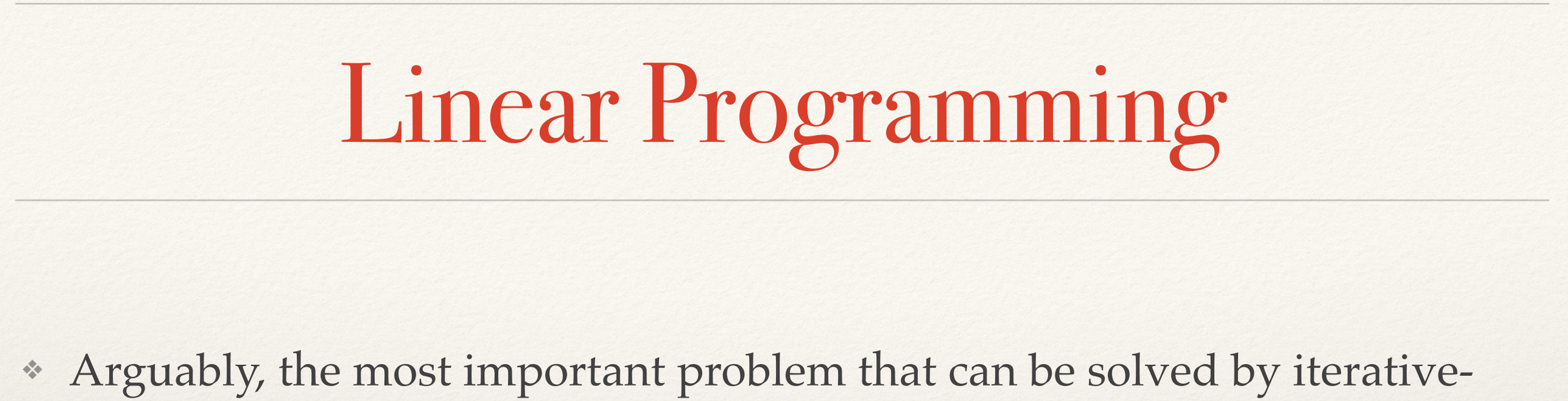
- Arguably, the most important problem that can be solved by iterative-improvement algorithms is linear programming, and the algorithm is called the Simplex Method (formal description of the algorithm is beyond the scope of this course).

$$
\begin{array}{rll}
\text{maximize (or minimize)} & c_1 x_1 + \cdots + c_n x_n & \\
\text{subject to} & a_{i1} x_1 + \cdots + a_{in} x_n \leq \text{ (or } \geq \text{ or } =) \; b_i & \text{for } i = 1, \ldots, m \\
& x_i \geq 0 & \text{for } i = 1, \ldots, n
\end{array}
$$

feasibility

Linear Programming

- Arguably, the most important problem that can be solved by iterative-improvement algorithms is linear programming, and the algorithm is called the Simplex Method (formal description of the algorithm is beyond the scope of this course).



Linear Programming

- The simplex method (1947):
  - Start by identifying an extreme point of the feasible region.
  - Then, check whether one can get an improved value of the objective function by going to an adjacent extreme point.
    - If it is not the case, the current point is optimal.
    - If it is the case, proceed to an adjacent extreme point with an improved value of the objective function.
- After a finite number of steps, the algorithm will either reach an extreme point where an optimal solution occurs, or determine that no optimal solution exists.

Linear Programming

- In the worst case, the simplex method takes exponential time.
- In 1979, L.G. Kachian introduced the ellipsoid method for solving LP in polynomial time (even in the worst case). But, the simplex method was faster in practice!
- In 1984, N. Karmarkar introduced a polynomial time algorithm for LP that is competitive with the simplex method in practice.

Linear Programming

- So, LP is polynomial.
- However, if we add the constraint that the variables are integer (Integer Linear Programming, or ILP), the problem becomes NP-hard.
- ILP has very powerful solvers (CPLEX, Gurobi,…).
- Many problems in computer science can be formulated as instances of ILP and solved using ILP solvers.

Linear Programming

- The Vertex Cover Problem:
  - Input: Graph $g=(V,E)$.
  - Output: A smallest subset $V' \subseteq V$ such that every edge $e \in E$ has at least one of its endpoints in V'.

Linear Programming

- We can solve the Vertex Cover Problem by casting it as an integer linear program (that can then be solved using an ILS solver).
- Introduce a variable x_i for every node v_i such that $x_i=1$ if the node v_i is part of the subset V' and $x_i=0$ otherwise.

Linear Programming

- The goal is to find set V' such that
  - every edge $\{x_i,x_j\}$ has at least one of its endpoints in V'.
    - This can be achieved via the constraints: $x_i+x_j \geq 1$.
- the smallest number of x_i's are set to 1.
  - This can be achieved via minimizing the sum of the x_i's.

Linear Programming

- Putting it all together, solving the Vertex Cover Problem for graph $g=(V,E)$ where $V=\{v_1,v_2,\ldots,v_n\}$ can be achieved by solving the following instance of ILP:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} x_i \\
\text{subject to} \quad & \\
& x_i + x_j \geq 1 \quad \forall \{v_i, v_j\} \in E \\
& x_i \in \{0,1\} \quad \forall v_i \in V
\end{aligned}
$$

Linear Programming

- Once the ILP instance is solved, the vertex cover V' is obtained as follows:

$$V' = \{v_i \in V : x_i = 1\}$$

Reductions

- This technique of obtaining the solution to a problem (Vertex Cover in our example) in terms of the solution to another problem (ILS in our example) is called reduction.

Reductions

Reductions

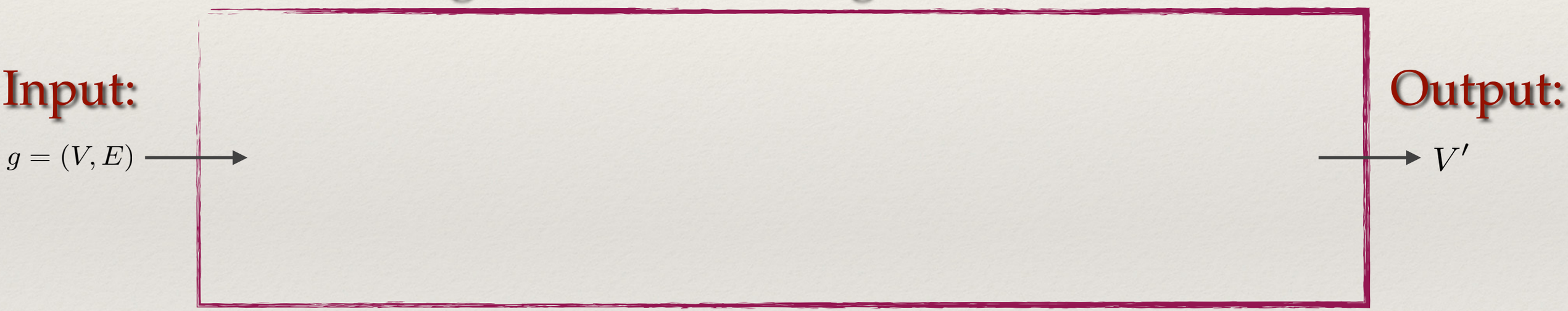
Input:

$g = (V, E)$

Output:

V'

Reductions



Reductions

Algorithm for solving Vertex Cover

Input:

$g = (V, E)$

Translate

minimize $\sum_{i=1}^{n} x_i$

subject to

$x_i + x_j \geq 1 \quad \forall \{v_i, v_j\} \in E$

$x_i \in \{0, 1\} \quad \forall v_i \in V$

Output:

V'

Reductions

Algorithm for solving Vertex Cover

Input:

$g = (V, E)$

Translate

minimize $\sum_{i=1}^{n} x_i$

subject to

$x_i + x_j \geq 1 \quad \forall \{v_i, v_j\} \in E$

$x_i \in \{0, 1\} \quad \forall v_i \in V$

Algorithm for solving ILP

$x_1 = 0$

$x_2 = 1$

$x_3 = 1$

$x_4 = 0$

$\ldots$

Output:

V'

Reductions

Algorithm for solving Vertex Cover

Input:

$g = (V, E)$

Translate

minimize $\sum_{i=1}^{n} x_i$

subject to

$x_i + x_j \geq 1 \quad \forall \{v_i, v_j\} \in E$

$x_i \in \{0, 1\} \quad \forall v_i \in V$

Algorithm for solving ILP

$x_1 = 0$

$x_2 = 1$

$x_3 = 1$

$x_4 = 0$

$\ldots$

Translate

$V' = \{v_i \in V : x_i = 1\}$

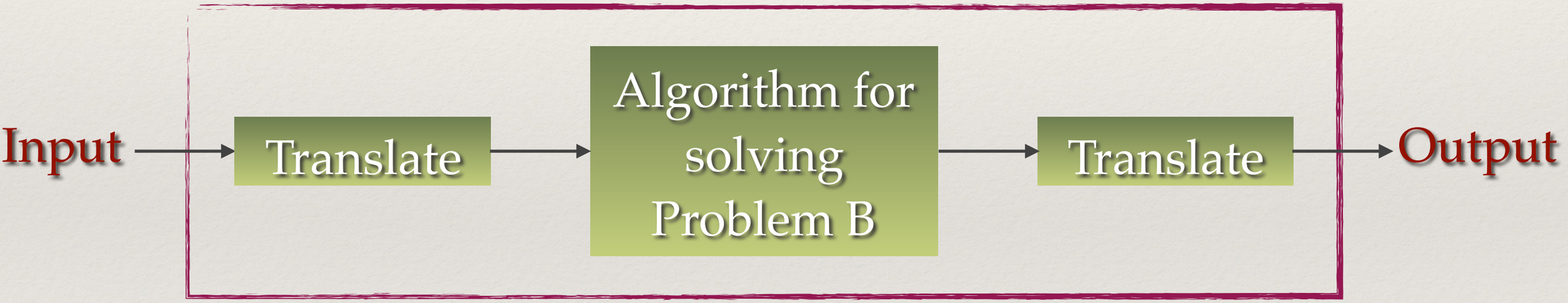
Output:

V'

Reductions

Reducing Problem A to Problem B

Algorithm for solving Problem A



Reductions

- Reductions play a central role in computer science.
- If we can reduce Problem A to Problem B, then:
  - What can we say about Problem A if Problem B has a polynomial time solution and it takes polynomial time to compute the reduction function (the "translation")?
  - What can we say about Problem B if there is an algorithm to compute the reduction function (the "translation") but there is no algorithm for Problem A?

Back to Iterative Improvement: Main Ingredients

- Starting solution candidate
- An operation (or set of operations) to modify a solution candidate
- A method to score the solution candidate for optimality
- A method to search the solution space

Iterative Improvement: Different Flavors

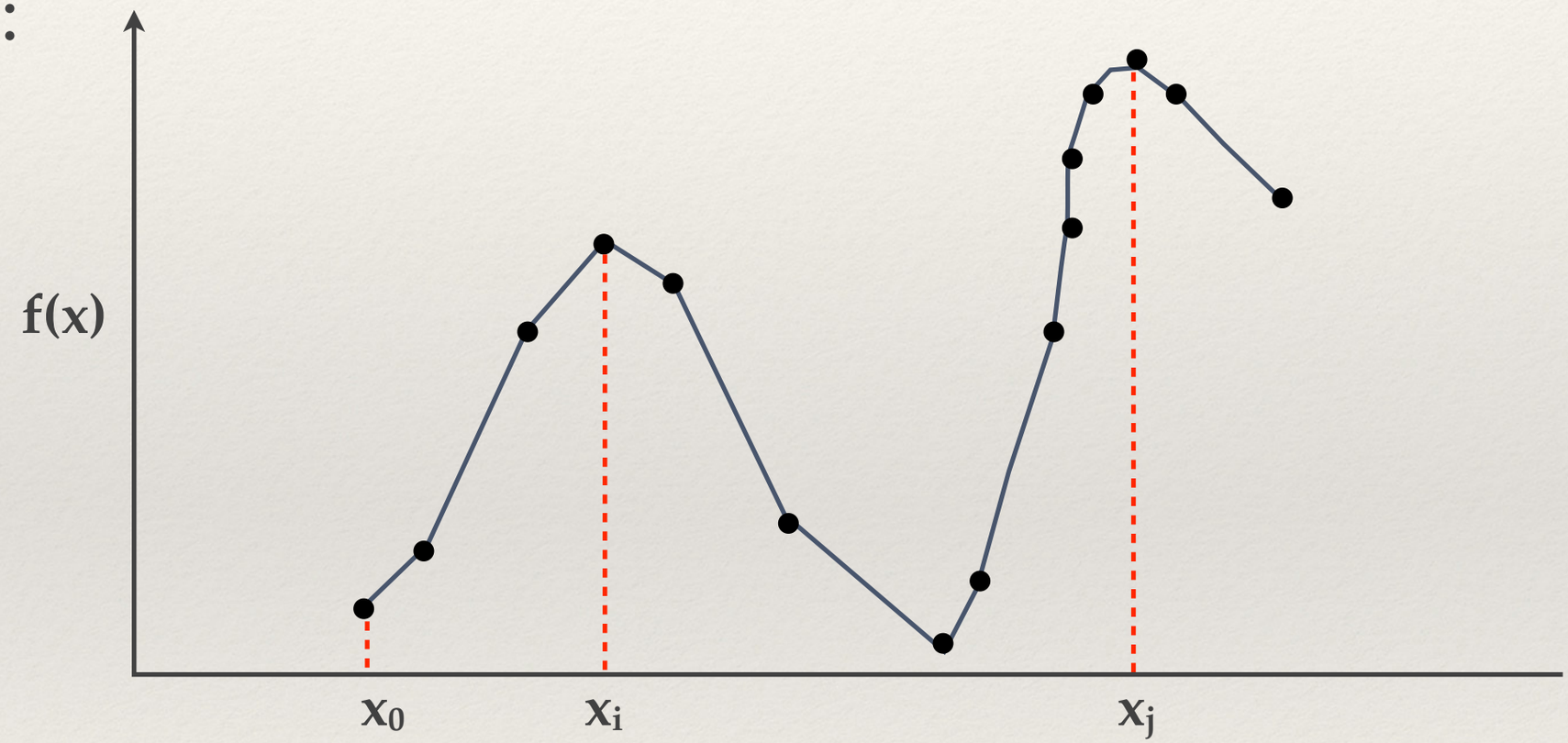
- Hill climbing
- Steepest ascent hill climbing
- Random-restart (steepest ascent) hill climbing
- Stochastic hill climbing
- ...

Hill Climbing

- In every step, choose a neighbor of the current solution that is closer to the optimal solution.
- Gets stuck in local optima (that is, solution candidates whose neighbors are no closer than it to the optimal solution).

Hill Climbing

- Illustrating hill climbing, where **x** is of length 1 and has real values:



Hill climbing starts at arbitrary value $\mathbf{x}=\mathbf{x}_0$.
Stops at value $\mathbf{x}_i$, since neighboring values have lower value of f.
Notice the issue of local ($f(x_i)$) vs. global ($f(x_j)$) maximum.

Steepest Ascent Hill Climbing

- A variant of hill climbing where in each step the neighbor that is closest to the optimal solution is chosen (that is, choose the “best” neighbor).
- Does not overcome the local optima problem.

Random-Restart (Steepest Ascent) Hill Climbing

- Runs (steepest ascent) hill climbing multiple times, each from a different starting point, and record/return the optimal solution found.

Stochastic Hill Climbing

- Of all neighbors that are closer to the optimal solution, choose one at random.
- The choice can be uniformly at random or with probability that's proportional to the amount of improvement.

Hill Climbing Variants

- Whenever a local optimum is reached, the search is stopped.
- It is also common to pre-specify a limit on the number of steps or amount of time to terminate the search (in this case, the algorithm terminates whenever it reaches a local optimum or the limit, whichever happens first).
- If the problem concerns minimization, the algorithm should descend, rather than ascend, towards optimality.

A Hill Climbing Algorithm for TSP

- Start with an arbitrary tour t0 of the n cities, and compute its length f(t0).
- Make a change to the tour, so that you obtain tour t1, and compute its length f(t1). If f(t1)<f(t0), continue with f(t1); otherwise, go back to t0 and try another neighboring tour (i.e., tour that differs from the “current” one by a single change).
- Repeat as long as at least one neighbor improves the length of the tour.

A Hill Climbing Algorithm for TSP

A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A1: Easy! It's the sum of the lengths of the tour's edges.

A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A1: Easy! It's the sum of the lengths of the tour's edges.

Q2: How do we change the current tour to get the next one?

A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A1: Easy! It's the sum of the lengths of the tour's edges.

Q2: How do we change the current tour to get the next one?

A2: There are many ways! One of them is just swap two pairs of neighbors.

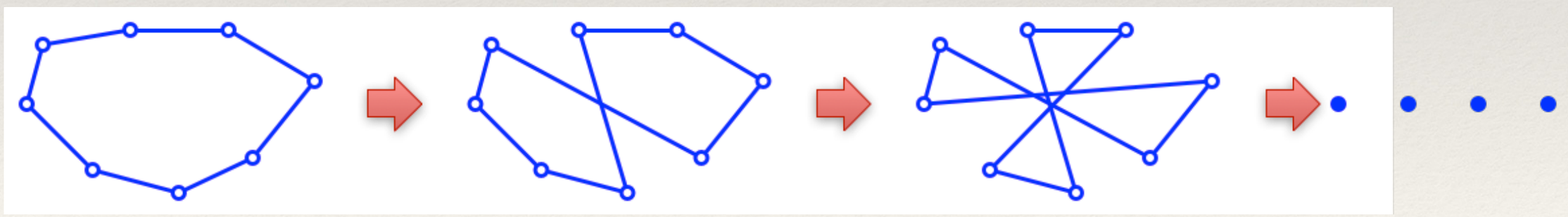
A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A1: Easy! It's the sum of the lengths of the tour's edges.

Q2: How do we change the current tour to get the next one?

A2: There are many ways! One of them is just swap two pairs of neighbors.



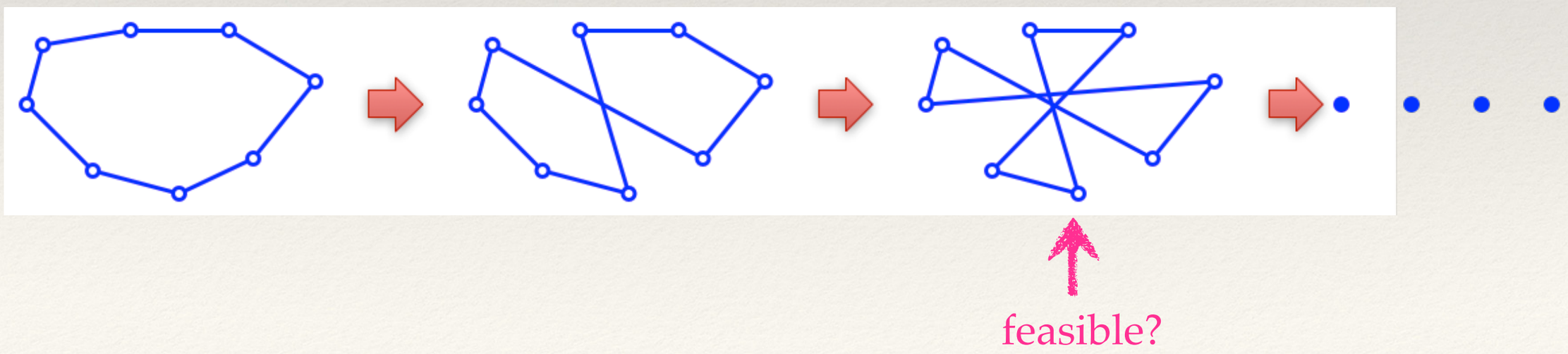
A Hill Climbing Algorithm for TSP

Q1: How do we compute the length of a tour?

A1: Easy! It's the sum of the lengths of the tour's edges.

Q2: How do we change the current tour to get the next one?

A2: There are many ways! One of them is just swap two pairs of neighbors.



A Hill Climbing Algorithm for TSP

- Is this algorithm guaranteed to find an optimal solution to TSP?

- Answer: In some cases yes, but in general no.

- TSP is NP-hard (to learn more about this, take COMP 481 or COMP 482, or even better, take both).

Vertex-Cover and Max-SAT

- Vertex-Cover: find the smallest set U of nodes such that each edge of the graph has at least one endpoint in U.
- Max-SAT: find a truth assignment to a CNF formula that satisfies the maximum number of clauses.
- Describe hill climbing (different variants) algorithms for these two problems.

Questions?