



Chapter 1
Search

Paragraph 3
Constrained Optimization

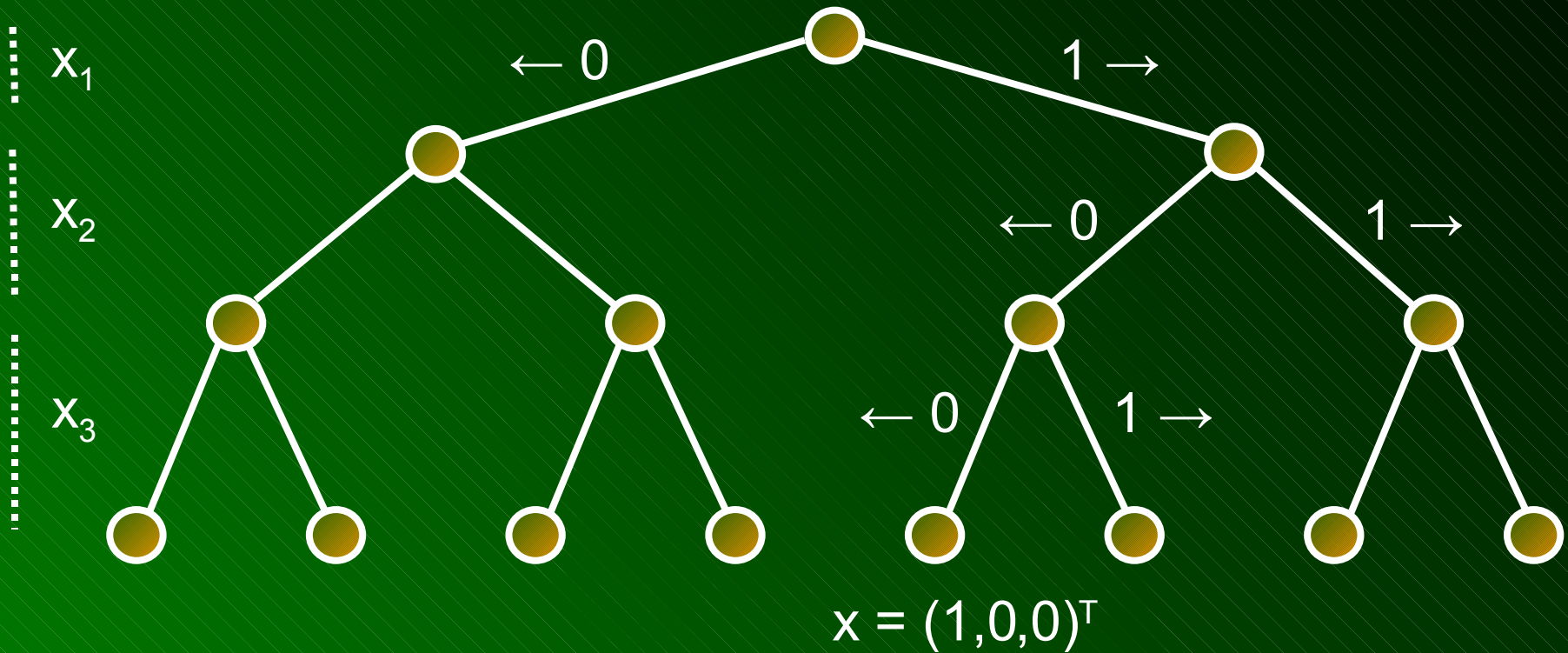
Tree Search

- As an example, assume we have to solve the Knapsack Problem.
- How many combinations of knapsack items are possible?
- The brute-force approach to solve the problem is to enumerate all 2^n combinations, see which ones are feasible, and which one of those achieves maximum profit.
- A systematic way of enumerating all solutions is via **backtracking**.

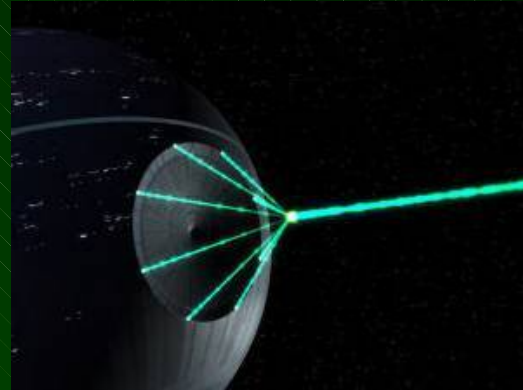
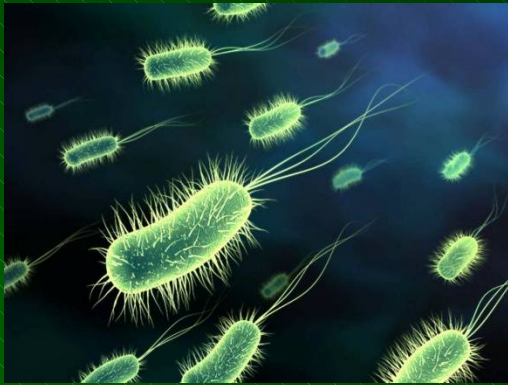
Tree Search

- Assume we order the variables x_1, \dots, x_n .
- A recursive way of enumerating all solutions is to set x_1 to 0 first and to recursively enumerate all solutions for $KP(x_2, \dots, x_n, p, w, C)$. Then we set x_1 to 1 and enumerate all solutions for $KP(x_2, \dots, x_n, p, w, C - w_1)$.
- This procedure yields to a **search tree!**

Tree Search



Combinatorial Explosions



1 2 4 8 16 32 64 128 ...

Will the Bacteria ever overtake the laser?
When will this happen? Take a guess!

Combinatorial Explosions

- Enumerating all possible solutions is of course not feasible when there are too many items.
- What is “too many”?
 - 500? 200? 100? 50? 10?
 - Take a guess!
- Assume we can investigate 1 solution per cpu cycle at a rate of 10 GHz (that’s 10 billion per second). Then, enumerating all Knapsacks with 50 items takes more than 100 years!
- This effect is called a combinatorial explosion.
- If $NP \neq P$, it cannot be avoided. However, we can aim at pushing the intractable instance sizes as far as possible – far enough to solve real-world instances. This is what combinatorial optimization is all about!

Implicit Enumeration

- We cannot afford to enumerate all combinations.
- We must try to enumerate the overwhelming part of all combinations implicitly!
- The only way to do this is by intelligent inference.
 - It is usually easy to find a first solution.
 - The core question to ask for an optimization problem is: Can we achieve a better solution?
 - Answering this question is of course NP-hard.
 - Consequently, we have to try to estimate intelligently.

Relaxations

- We can achieve an upper bound on an optimization problem like Knapsack by computing an optimal solution over a larger set of feasible solutions.
- We can allow more solutions by getting rid of some constraints - hopefully in such a way that the relaxed problem is easier to solve.
- This approach is generally called a **relaxation**.
- The milder the effect of a relaxation on the objective value, the better our estimate!

Linear Relaxation

- The most commonly used relaxation consists in dropping the constraint that variables be integer.
- In Knapsack for instance, we replace $x_i \in \{0, 1\}$ by $0 \leq x_i \leq 1$.
- Then, optimizing the relaxed problem calls for solving a linear program.

Linear Relaxation

- Definition:

A linear program consists of variables $x_1 \dots x_n$ and a linear objective function $z^T x$ that is to be maximized or minimized under linear constraints $Ax \leq a$, $Bx = b$, and $Cx \geq c$.

- LPs can be solved in polynomial time! 😊

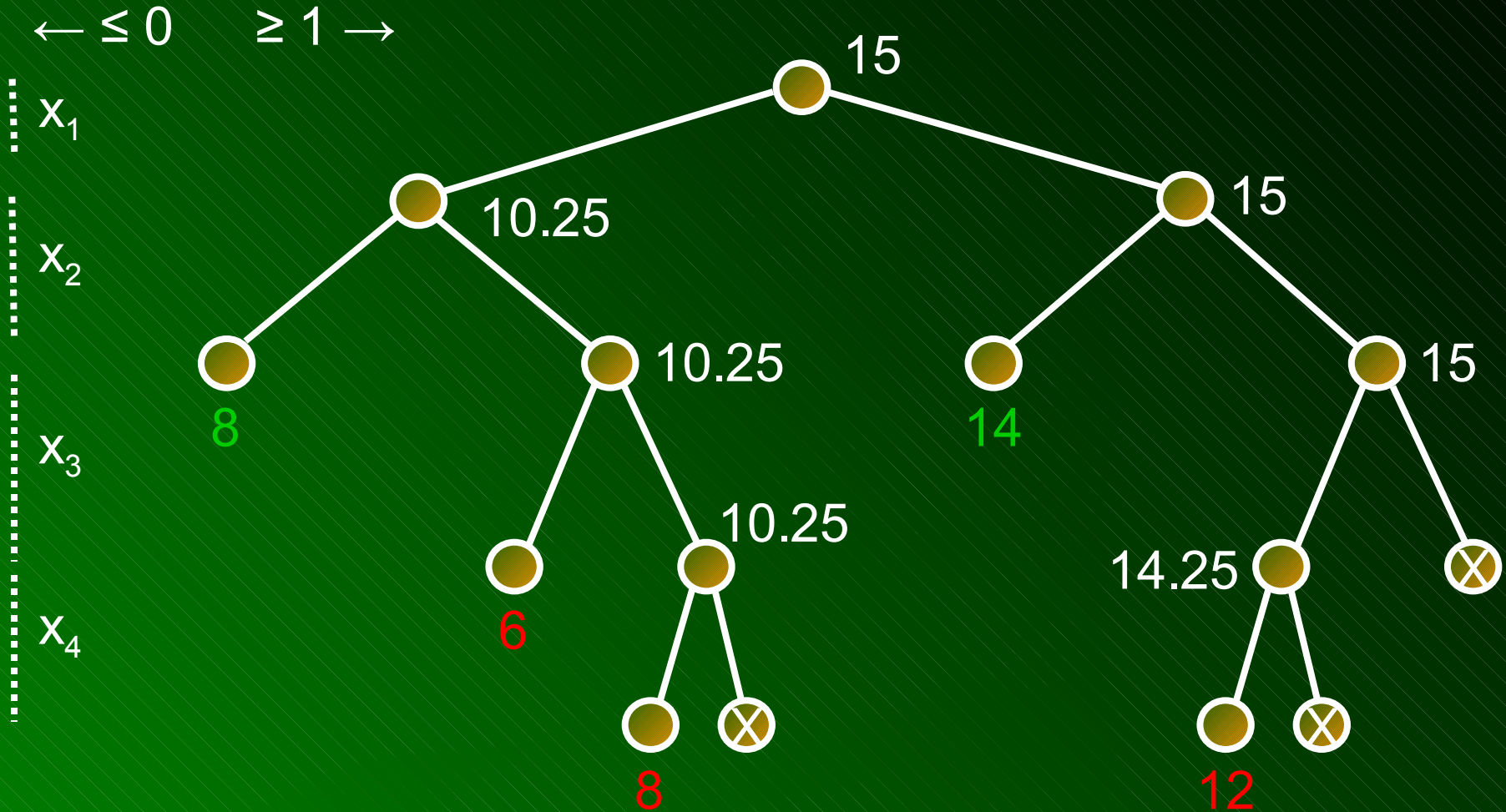
Relaxations

- What does a relaxation give us?
 - **Dominance**: If the relaxation value is lower (for minimization: greater) or equal than the best known solution
 - ⇒ All solutions with the current prefix are sub-optimal and need not be looked at at all!
 - **Optimality**: If the relaxation returns a feasible solution for our original problem
 - ⇒ This solution dominates all other feasible solutions, they need not be looked at at all!
 - **Infeasibility**: If the relaxation is infeasible
 - ⇒ There exists no feasible solution with the current prefix, all such combinations need not be looked at at all!
- In all these cases, we are not going to expand the search tree below the current node further ⇒ We **prune** the search!

Example

- Knapsack Instance
 - Maximize
 - $9x_1 + 3x_2 + 5x_3 + 3x_4$
 - such that
 - $5x_1 + 2x_2 + 5x_3 + 4x_4 \leq 10$
 - $x_1, x_2, x_3, x_4 \in \{0, 1\}$
- LP Relaxation
 - Maximize
 - $9x_1 + 3x_2 + 5x_3 + 3x_4$
 - such that
 - $5x_1 + 2x_2 + 5x_3 + 4x_4 \leq 10$
 - $0 \leq x_1, x_2, x_3, x_4 \leq 1$

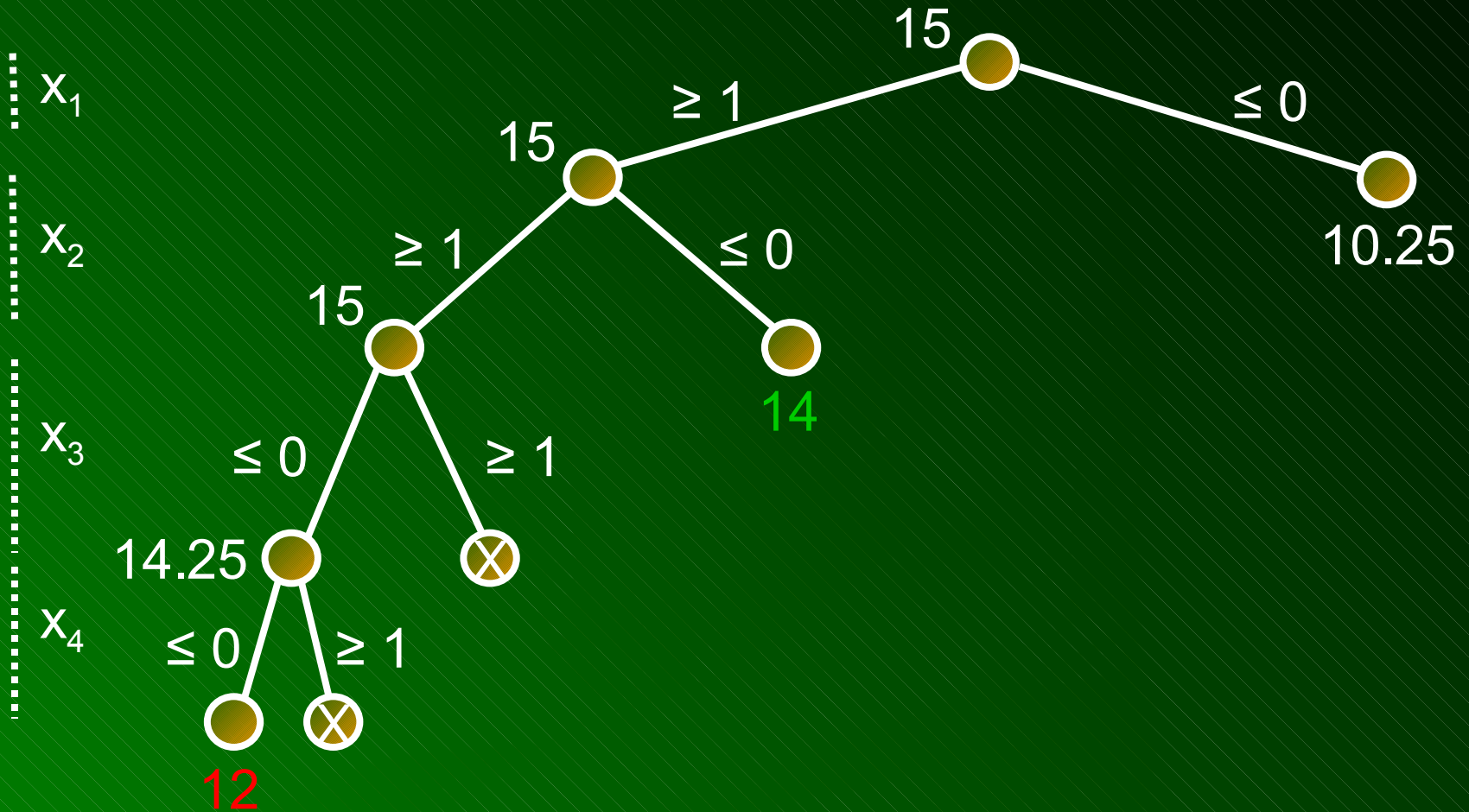
Example



Branching Direction Selection

- In our general Branch-and-Bound scheme, we have some liberty:
 - Which node shall we look at next?
 - Which variable should we branch on?
- We would like to dive into the search tree in order to find a feasible solution (a lower bound) quickly.
- When diving, the question which node to pick next comes down to: which of the two son nodes shall we follow first?

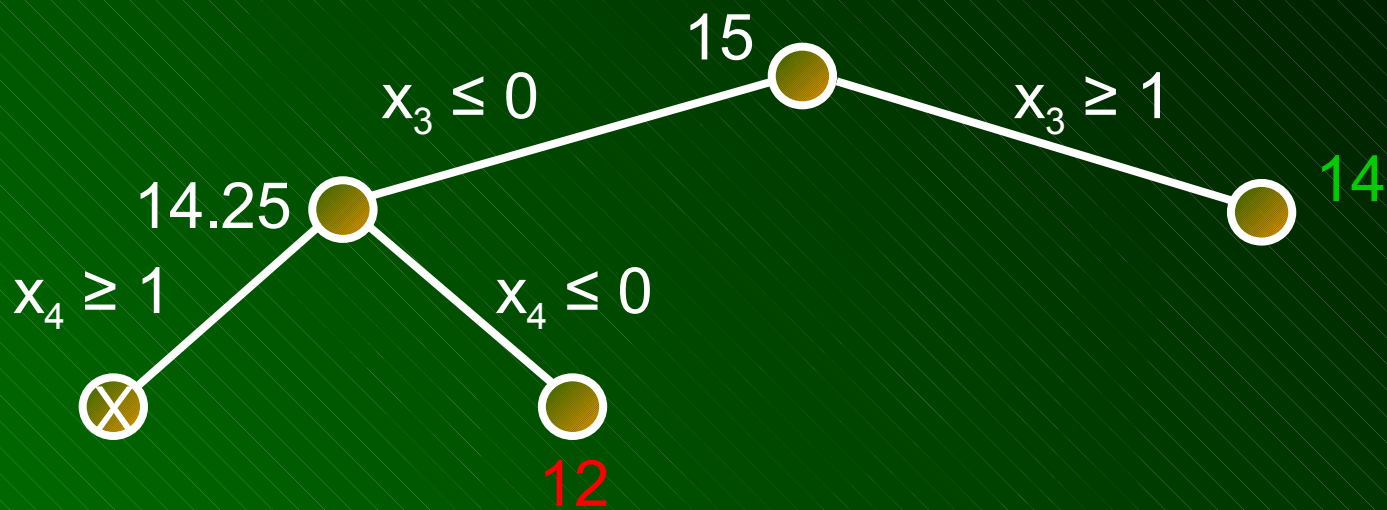
Example



Branching Variable Selection

- In our general Branch-and-Bound scheme, we have some liberty:
 - Which node shall we look at next?
 - Which variable should we branch on?
- In order to have a chance of improving our upper bound, we need to branch on a fractional variable.
- In KP, there is exactly one.

Example



Liberties in B&B

- So far, we took the liberty to select our own branching values and variables.
 - Value selection is a special case of node selection in depth first search.
 - The way how we traverse the search tree is generally determined by our **search strategy**.
 - Variable selection is a special case of **branching constraint selection**.
 - Very many different ways to partition the search space are possible.

Search Strategies

- When choosing the next node, we would like:
 - to find a near optimal solution quickly
(**lower bound improvement** in maximization)
 - not to jump too much to make use of incremental data-structures and keep the **memory requirements** in limits.

Search Strategies

– Depth First Search

- Finds feasible solutions quickly.
- Is very memory efficient.
- Can easily get stuck in sub-optimal parts of the search space.

– Best First Search

- Look at the node with best relaxation value next.
- Is provably optimal in the sense that it never visits a node that could be pruned otherwise.
- A lot of jumping is necessary and memory requirements are prohibitively large (often search degenerates to breadth first search).

Search Strategies

– Depth First Search with Best Backtracking

- Is a mix of both depth and best first search: perform depth first search until a leaf is found, then backtrack to the node with best relaxation value and so on.
- Much less jumping than best first search.
- Is more memory efficient than best first search, but less than DFS – could still be very memory intensive.

– Least Discrepancy Search

- Follow DFS with heuristic branching direction selection. Investigate leaves in order of increasing discrepancy wrt that heuristic.
- Memory requirements are within limits.
- Often finds good solutions early in the search.

Branching Constraint Selection

- When partitioning the search space, we would like:
 - to reduce the relaxation value as quickly as possible (**upper bound improvement** in maximization)
 - to avoid to double our workload which can happen for example when choosing the wrong branching variable
- The easiest way to partition the search is by branching on one variable.

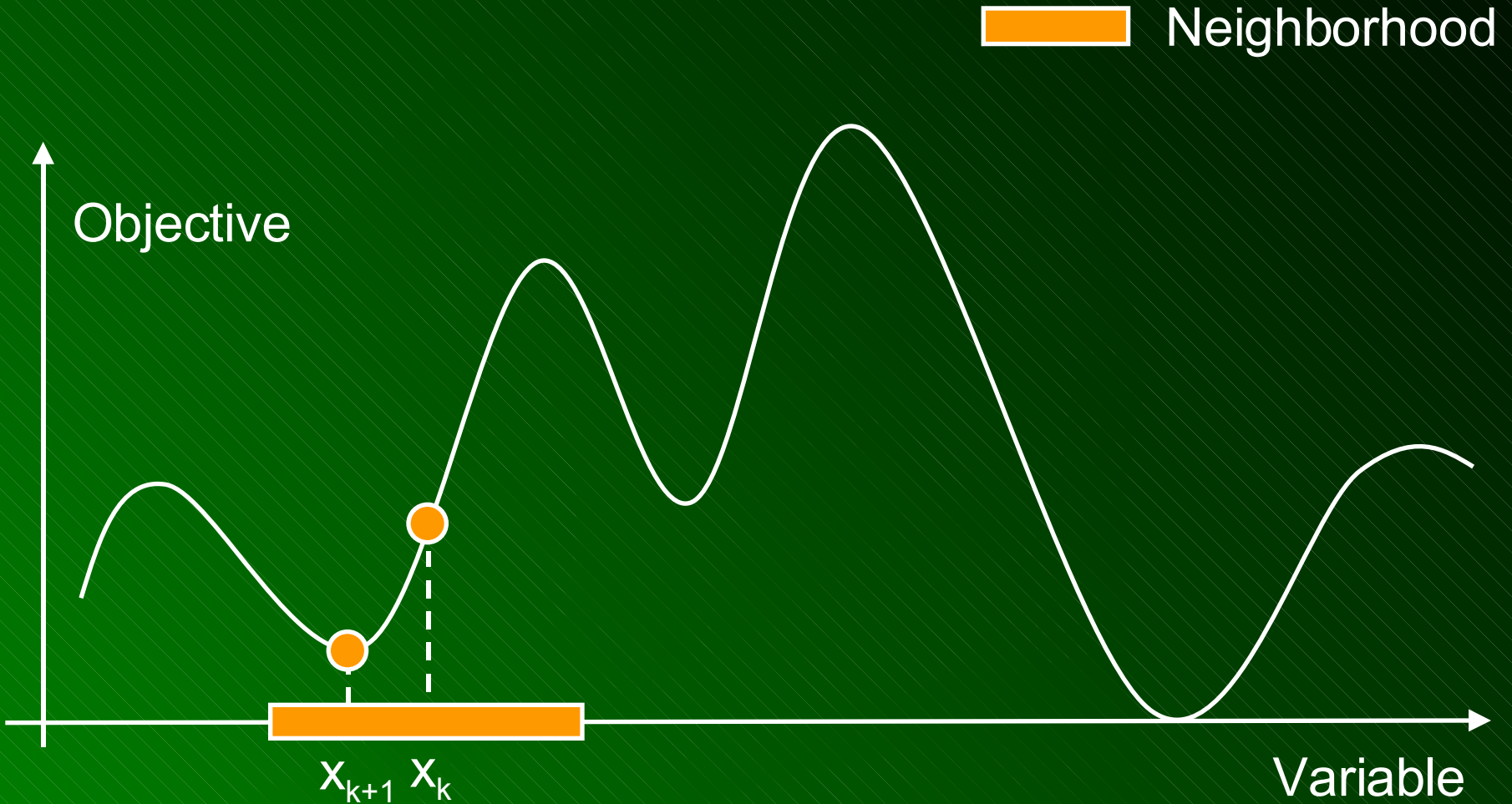
Branching Constraint Selection

- Unary Branching Constraints
 - Choose the variable which has a fractional part closest to $\frac{1}{2}$.
 - Try to estimate how much enforcing the integrality of a variable will cost at least – degradation method.
 - Follow user-defined priorities.
 - Choose a random variable and combine with restarts.
- Empirically, we prefer balanced search trees over degenerated branches.

Local Search

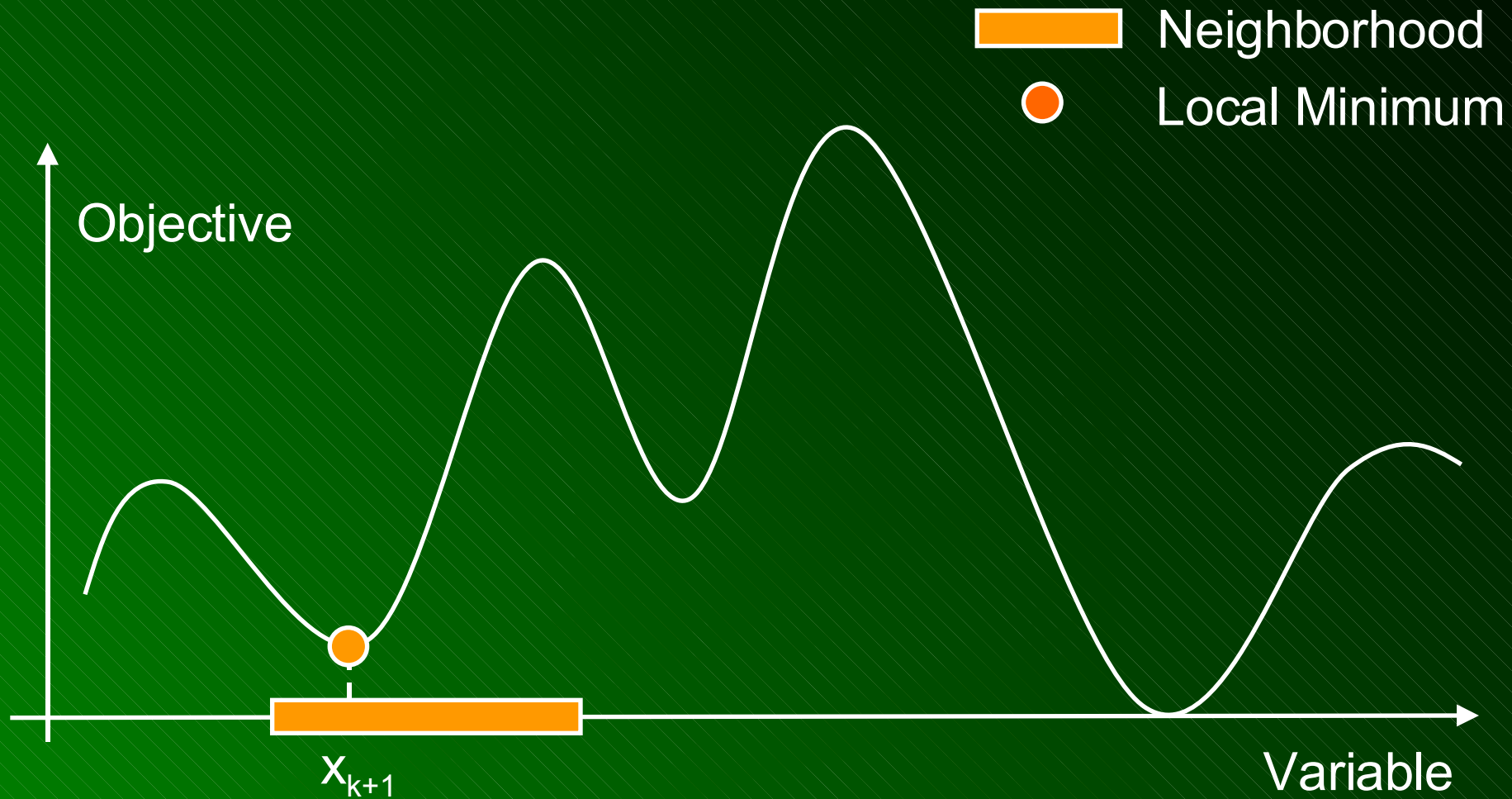
Local Search Approach

Simulated Annealing



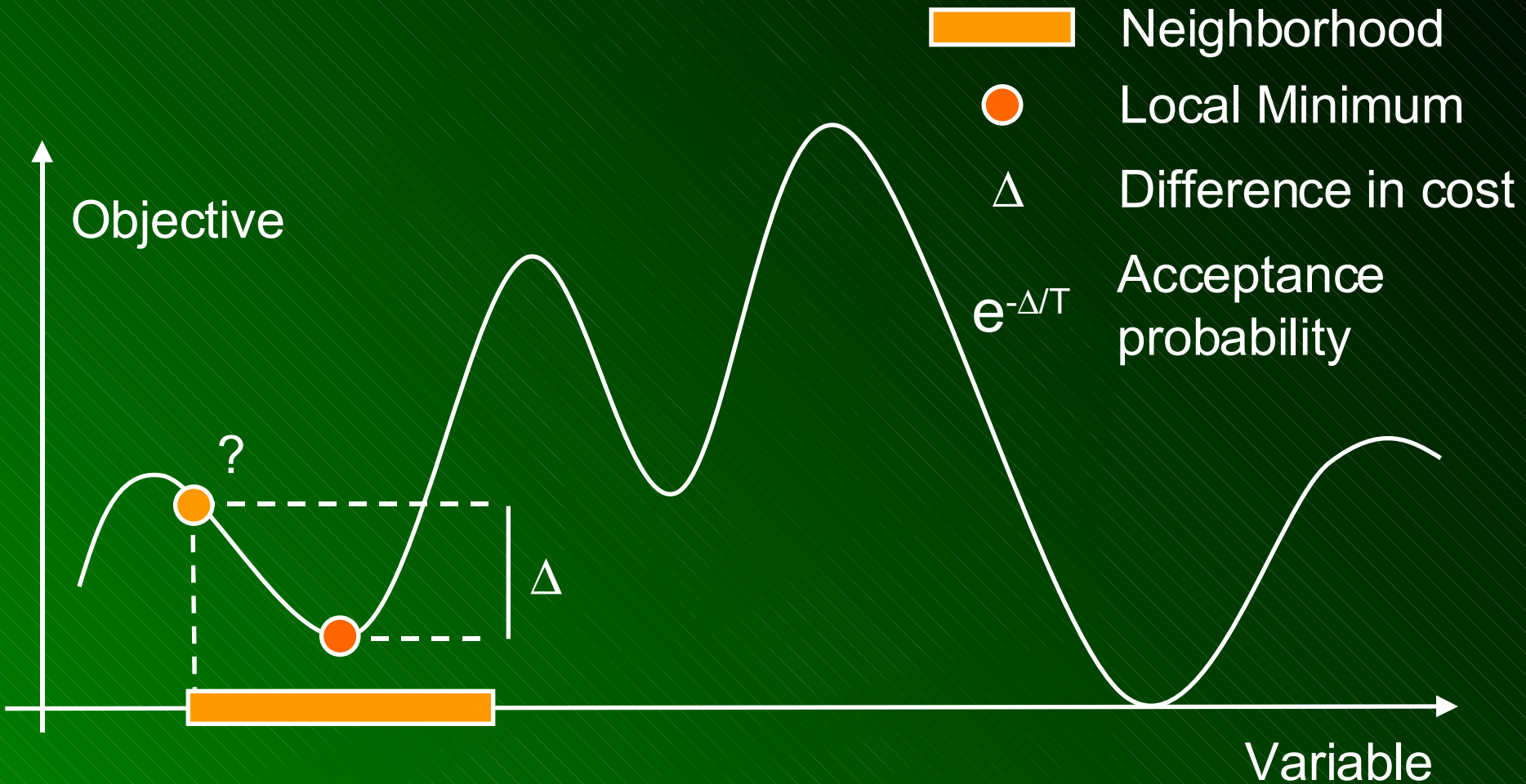
Local Search Approach

Simulated Annealing



Local Search Approach

Simulated Annealing



Local Search Approach

Simulated Annealing

- Must choose **Neighborhood** and **Objective function**
- Neighborhood
 - Start with random variable instantiation.
 - Keep the neighborhood reasonably simple.
 - Make sure every part of the search space can be reached.
- Objective
 - Avoid large plateaus where search is inherently unguided.
 - Avoid spiky landscapes packed with local optima.

The End

