Chapter 3: Classical Optimization Methods You Must Know

Quantum optimization does not replace classical optimization in one step. In fact, every practical variational quantum algorithm is partly classical.

A variational quantum algorithm prepares a quantum state, measures it, and then uses a classical optimizer to decide which parameters to try next. Even when the long-term goal is quantum advantage, the short-term reality is comparison: a quantum method must be tested against strong classical methods, not against weak or outdated ones.

This chapter gives you the classical optimization tools you must know before studying variational quantum optimization seriously. We will not cover every classical method. Instead, we will focus on the families that appear again and again when people design, benchmark, or criticize quantum optimization algorithms:

- gradient descent,
- simulated annealing,
- branch and bound,
- integer programming,
- heuristic search,
- approximation algorithms.

Each method represents a different way to think about search.

Gradient descent follows slopes. Simulated annealing explores by sometimes accepting worse moves. Branch and bound proves that parts of a search tree cannot contain the answer. Integer programming models discrete decisions with algebraic constraints. Heuristic search uses practical rules to find good solutions quickly. Approximation algorithms give provable quality guarantees even when exact optimization is too expensive.

By the end of this chapter, you should be able to look at a proposed quantum optimization experiment and ask:

> What classical methods should this be compared against?

That question is one of the most important habits in responsible quantum optimization.

3.1 Optimization Algorithms: What Are We Actually Comparing?

An optimization algorithm is a procedure for finding a good solution to an optimization problem.

Recall from Chapter 2 that an optimization problem usually has:

1. decision variables,
2. an objective function,
3. constraints.

For example, suppose we want to minimize

$$f(x) = (x - 3)^2.$$

The decision variable is x. The objective function is f(x). If there are no restrictions on x, then this is an unconstrained optimization problem. The best solution is

$$x = 3,$$

because

$$f(3) = 0.$$

This example is easy because the search space is one-dimensional and smooth. Real problems are often much harder. A delivery-routing problem may involve millions of possible routes. A factory-scheduling problem may involve thousands of tasks and many constraints. A portfolio-selection problem may involve binary choices, risk estimates, transaction limits, and regulatory rules.

When comparing optimization algorithms, we usually care about several different quantities.

The first is solution quality. Did the algorithm find the best solution, or only a good one?

The second is runtime. How long did the algorithm take?

The third is scaling behavior. What happens as the problem becomes larger?

The fourth is robustness. Does the algorithm work reliably on many different problem instances, or only on carefully chosen examples?

The fifth is cost of information. Some algorithms need exact objective values. Others can work with noisy estimates. This matters greatly for variational quantum algorithms because quantum measurements are statistical: repeated measurements are needed to estimate an expected value.

Classical optimization methods can be grouped in several useful ways.

An exact algorithm is designed to find a truly optimal solution, assuming enough time and memory. Branch and bound for integer programming is an example.

An approximate algorithm is designed to find a solution that is close to optimal, often with a mathematical guarantee. Approximation algorithms for problems such as MaxCut and vertex cover belong here.

A heuristic algorithm is designed to find good solutions in practice, but it may not provide a proof that the solution is optimal or near-optimal.

A deterministic algorithm gives the same result every time if started from the same input.

A randomized algorithm uses randomness, so different runs may produce different results.

These distinctions are not just vocabulary. They affect how we interpret quantum optimization results. If a quantum algorithm finds a good solution to a hard problem, that is interesting. But if a classical heuristic finds an equally good solution faster, then the quantum method has not yet shown practical value.

3.2 Gradient Descent: Following the Slope Downhill

Many optimization problems involve continuous variables. A variable is continuous if it can take values along a continuum, such as

$$x = 1.2, \quad x = 1.25, \quad x = 1.251, \quad x = 1.2517.$$

In continuous optimization, one of the most important ideas is the gradient.

For a function of several variables,

$$f(x_1, x_2, \ldots, x_n),$$

the gradient is a vector that points in the direction of steepest increase of the function. It is written as

$$\nabla f(x).$$

If the gradient points uphill, then the negative gradient points downhill. This leads to the basic idea of gradient descent:

> Start from a point, compute the slope, and move a small step downhill.

The standard update rule is

$$x_{k+1} = x_k - \eta \nabla f(x_k),$$

where:

- x_k is the current point,
- x_{k+1} is the next point,
- $\nabla f(x_k)$ is the gradient at the current point,
- η is the learning rate or step size.

The step size η controls how far we move. If η is too small, progress may be slow. If η is too large, the algorithm may overshoot and fail to settle near a minimum. Modern numerical optimization texts treat step-size choice as a central practical issue, not a minor detail (Nocedal and Wright, 2006).

Let us see this with a simple one-variable example.

Suppose

$$f(x) = (x - 3)^2.$$

The derivative is

$$f'(x) = 2(x - 3).$$

In one dimension, the gradient is just the derivative. If we start at

$$x_0 = 0$$

and choose

$$\eta = 0.1,$$

then

$$f'(0) = 2(0 - 3) = -6.$$

The gradient descent update gives

$$x_1 = 0 - 0.1(-6) = 0.6.$$

Now we are closer to 3. At x = 0.6,

$$f'(0.6) = 2(0.6 - 3) = -4.8,$$

so

$$x_2 = 0.6 - 0.1(-4.8) = 1.08.$$

The sequence moves toward the minimum at x = 3.

Gradient descent is especially important in machine learning, where model parameters are adjusted to reduce a loss function. It also appears inside variational quantum algorithms, where the classical computer may adjust circuit parameters to reduce an estimated energy or cost value.

However, gradient descent is not magic.

If the objective function is convex, then every local minimum is also a global minimum. A convex function has a bowl-like structure: if you draw a line segment between two points on the graph, the graph lies below or on that segment. Convex optimization has a rich theory because this structure makes global optimization much more reliable (Boyd and Vandenberghe, 2004).

But many important problems are nonconvex. A nonconvex function may have many valleys. Gradient descent can get stuck in a local minimum or move slowly across flat regions.

For example, imagine hiking in fog. If your rule is “always walk downhill,” you may reach the bottom of a small valley, even though a deeper valley lies elsewhere. Gradient descent follows local information. It does not automatically know the full landscape.

This limitation matters for quantum optimization because variational quantum algorithms often produce complicated nonconvex parameter landscapes. Later, in Chapters 9 and 17, we will study why training these landscapes can be difficult.

3.3 Local and Global Thinking

Before moving to discrete methods, we need two central terms.

A local optimum is a solution that is best among nearby solutions.

A global optimum is a solution that is best among all feasible solutions.

For a minimization problem, x^star is a global minimum if

$$f(x^\star) \leq f(x)$$

for every feasible x.

It is a local minimum if the inequality holds only for points near x^star.

Consider the function

$$f(x) = x^4 - 4x^2 + x.$$

This function has multiple valleys and hills. A method that only uses local slope information may find one valley, but not necessarily the deepest one.

In discrete optimization, the same idea appears differently. Suppose a delivery route can be improved by swapping two stops. A route may be locally optimal if no single swap improves it. But there may be a very different route, requiring many changes at once, that is much better.

This difference between local and global search is one reason optimization is hard. Many algorithms are ways of balancing two activities:

- exploitation — improving the current promising solution,
- exploration — searching different regions of the solution space.

Gradient descent mostly exploits local slope information. Simulated annealing, which we study next, deliberately adds exploration.

3.4 Simulated Annealing: Escaping Bad Valleys

Simulated annealing is a randomized optimization method inspired by a physical process called annealing.

In metallurgy, annealing means heating a material and then cooling it slowly. At high temperature, atoms can move around more freely. As the material cools, the atoms settle into a lower-energy structure. Kirkpatrick, Gelatt, and Vecchi introduced simulated annealing as an optimization method by using this physical analogy: an algorithm can sometimes accept worse solutions early in the search, then become more selective as a “temperature” parameter is lowered (Kirkpatrick, Gelatt, and Vecchi, 1983).

To understand simulated annealing, imagine a discrete optimization problem. We have a current solution s, and we can make small changes to get neighboring solutions.

For example, in a routing problem, a neighbor might be created by swapping the order of two delivery stops.

Let

$$C(s)$$

be the cost of solution s. We want to minimize C.

At each step, simulated annealing proposes a neighboring solution s'. If s' is better, meaning

$$C(s') < C(s),$$

we accept it.

But if s' is worse, we may still accept it with probability

$$\exp\left(-\frac{C(s') - C(s)}{T}\right),$$

where T > 0 is the temperature.

This formula has an intuitive meaning.

If the worse move is only slightly worse, then C(s') - C(s) is small, so the probability of accepting it can be fairly high.

If the worse move is much worse, then C(s') - C(s) is large, so the probability is low.

If the temperature T is high, the algorithm is more willing to accept worse moves.

If T is low, the algorithm becomes conservative.

A simple simulated annealing procedure looks like this:

1. Start with an initial solution.
2. Choose an initial temperature.
3. Propose a small random change.
4. Accept the change if it improves the cost.
5. If it worsens the cost, accept it with a temperature-dependent probability.
6. Gradually lower the temperature.
7. Stop after enough steps or when improvement becomes unlikely.

The key idea is that accepting worse moves can help the algorithm escape local minima.

Suppose a scheduling algorithm has assigned tasks to machines. A single move may temporarily make the schedule worse, but that move might allow later changes that produce a much better schedule. A greedy algorithm that only accepts improvements would reject the first move. Simulated annealing might accept it early in the search.

Simulated annealing is important for quantum optimization for two reasons.

First, it is a strong classical baseline for many combinatorial problems. If a QAOA experiment claims good performance on MaxCut, scheduling, or QUBO problems, simulated annealing is often one of the classical methods worth comparing against.

Second, quantum annealing, discussed later in Chapter 15, is conceptually related but physically different. Simulated annealing uses thermal randomness in a classical algorithm. Quantum annealing uses quantum dynamics, often described with changing Hamiltonians. The analogy is useful, but the two methods are not the same.

3.5 Branch and Bound: Proving Where the Answer Cannot Be

Some optimization algorithms search intelligently by proving that large parts of the search space cannot contain the best answer.

This is the idea behind branch and bound.

Branch and bound is an exact framework for discrete optimization. It was developed as a systematic method for solving discrete programming problems, with the classic paper by Land and Doig giving an early formal version of the approach (Land and Doig, 1960).

To understand it, consider a binary optimization problem with variables

$$x_1, x_2, x_3, \ldots, x_n,$$

where each variable must be either 0 or 1.

A complete assignment might look like

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 1, \quad \ldots$$

If there are n binary variables, then there are

$$2^n$$

possible assignments. For n = 10, this is 1024 possibilities. For n = 100, it is about

$$1.27 \times 10^{30},$$

which is far too many to check one by one.

Branch and bound organizes the search as a tree.

At the top, no variables are fixed. Then the algorithm branches:

$$x_1 = 0$$

on one branch, and

$$x_1 = 1$$

on another. Each branch then splits again according to x_2, and so on.

If the algorithm searched the whole tree, it would still be brute force. The power comes from bounding.

A bound is a mathematical estimate of how good the best possible solution in a branch could be.

For a minimization problem, a lower bound says:

> No solution in this branch can have cost below this number.

Suppose we already have a feasible solution with cost 100. This is called the incumbent, meaning the best solution found so far.

Now suppose a branch has a lower bound of 120. Since we are minimizing, even the best possible solution in that branch cannot beat 100. Therefore, we can discard the entire branch. This is called pruning.

Here is a small example.

Suppose we are choosing items for a backpack. Each item has value and weight. The backpack has a maximum capacity. We want maximum value without exceeding the capacity. This is a version of the knapsack problem.

A branch might represent decisions such as:

$$x_1 = 1, \quad x_2 = 0, \quad x_3 \text{ undecided}, \quad x_4 \text{ undecided}.$$

To bound this branch, we might temporarily relax the problem by allowing fractional items. That easier relaxed problem can give an optimistic estimate of the best value still possible. If even the optimistic estimate is worse than the best complete solution already found, the branch can be pruned.

Branch and bound is exact: if run to completion correctly, it can prove optimality. But in the worst case, the search tree may still be enormous. Many integer optimization problems are NP-hard, meaning that no polynomial-time algorithm is known for them, and the broader question of whether all NP problems can be solved in polynomial time is the famous unresolved P versus NP problem (Garey and Johnson, 1979).

This is one reason practical optimization often mixes exact methods, relaxations, heuristics, and approximation methods.

3.6 Integer Programming: Optimization with Whole-Number Decisions

Many real decisions are not continuous.

A warehouse is either opened or not opened. A worker is either assigned to a shift or not assigned. A stock is either included in a portfolio or not included. A machine either performs a job at a certain time or it does not.

These are integer or binary decisions.

An integer programming problem is an optimization problem in which some or all variables must take integer values. If the variables are restricted to 0 or 1, they are called binary variables.

A common form is:

$$\min c^T x$$

subject to

$$Ax \leq b,$$

$$x \in \mathbb{Z}.$$

Here:

- x is a vector of decision variables,
- $c^{(T)}$ x is a linear objective function,
- Ax ≤ b represents linear constraints,
- x_i ∈ mathbbZ means the variables must be integers.

If some variables are continuous and others are integer, the problem is called a mixed-integer programming problem, often abbreviated as MIP.

If the objective and constraints are linear, we get mixed-integer linear programming, or MILP. Integer and combinatorial optimization are major fields, with deep theory and highly developed software tools (Nemhauser and Wolsey, 1988).

Let us model a simple assignment problem.

Suppose there are three workers and three tasks. Let

$$x□□ = \text{begincases } 1, \text{ if worker i is assigned to task j, ; } 0, \text{ otherwise. endcases}$$

If every task must be assigned exactly once, then for each task j,

$$∑□ x□□ = 1.$$

If every worker can do at most one task, then for each worker i,

$$∑□ x□□ ≤ 1.$$

If c_{ij} is the cost of assigning worker i to task j, then the objective may be

$$\min ∑□ ∑□ c□□ x□□.$$

This model is simple, but it already shows the power of integer programming: logical decisions can be expressed algebraically.

Modern integer programming solvers combine many ideas, including branch and bound, cutting planes, relaxations, presolve rules, and primal heuristics. Linear optimization and integer optimization are therefore not just abstract mathematics; they are practical computational technologies used in logistics, energy, manufacturing, finance, and scheduling (Bertsimas and Tsitsiklis, 1997).

Integer programming is also important because many quantum optimization methods require problems to be written in binary form. In later chapters, we will study QUBO models, where variables are binary and the objective is quadratic. But QUBO is not automatically the best modeling language for every real problem. Integer programming can express constraints naturally, while converting everything into QUBO may require penalty terms and extra variables.

This is a recurring lesson:

> A quantum algorithm begins only after the modeling choices have already shaped the problem.

3.7 Heuristic Search: Practical Rules for Hard Problems

A heuristic is a practical rule or strategy that often works well, even if it does not guarantee the best answer.

The word “heuristic” should not be understood as “bad.” Many real-world optimization problems are too large, too noisy, or too time-sensitive for exact methods. In such cases, a good heuristic may be extremely valuable.

A basic example is local search.

In local search, we start with a solution and repeatedly move to a neighboring solution if it improves the objective.

For a routing problem, a neighbor might be made by reversing a segment of the route.

For a scheduling problem, a neighbor might be made by moving one job to a different time slot.

For a portfolio-selection problem, a neighbor might be made by removing one asset and adding another.

Local search is simple:

1. Start with a feasible solution.
2. Generate neighboring solutions.
3. Move to a better neighbor.
4. Repeat until no simple improvement is found.

The final solution is often locally optimal, but not necessarily globally optimal.

To improve exploration, many heuristic methods add memory or randomness.

Tabu search keeps a memory of recent moves and temporarily forbids reversing them, helping the search avoid cycling.

Genetic algorithms maintain a population of candidate solutions and combine parts of them, inspired loosely by biological evolution.

Random restart methods run local search many times from different starting points.

Greedy algorithms build a solution one step at a time by choosing the option that looks best immediately.

The broad area of stochastic local search studies randomized search procedures for combinatorial problems and analyzes how practical choices such as neighborhood design, restart rules, and acceptance criteria affect performance (Hoos and Stützle, 2004).

Here is a simple greedy example.

Suppose we want to pack items into a box and each item has value and weight. A greedy rule might choose items in order of value-to-weight ratio. This can work well, but it can also fail. A single heavy item might have the highest total value, while several smaller items might have better ratios. The greedy rule may miss the best combination.

Heuristics are central in quantum optimization benchmarking. A weak comparison might show that a quantum method beats random guessing. But random guessing is rarely the relevant competitor. For practical value, a quantum method must be compared with serious classical heuristics and exact solvers when available.

This is especially important because many near-term quantum experiments use small problem sizes. Small instances can often be solved exactly by classical methods, so the interesting question is not merely whether the quantum method finds a good answer. The question is whether its performance gives evidence of useful scaling, better sampling, or some other advantage that could matter on larger instances.

3.8 Approximation Algorithms: Good Solutions with Guarantees

Sometimes exact optimization is too expensive, but we still want a mathematical guarantee.

An approximation algorithm is an algorithm that runs efficiently and returns a solution whose quality is provably close to optimal for a class of problems.

To explain this, suppose we have a minimization problem. Let

$$C_{alg}$$

be the cost found by the algorithm, and let

$$C_{opt}$$

be the true optimal cost.

An algorithm is a 2-approximation if it always returns a solution satisfying

$$C_{alg} \leq 2\, C_{opt}.$$

That means the algorithm’s solution costs at most twice the optimum.

For a maximization problem, the guarantee is usually written differently. If the algorithm always returns a value at least α times the optimum, where

$$0 < \alpha \leq 1,$$

then it has approximation ratio α. For example,

$$C_{alg} \geq 0.8\, C_{opt}$$

means the algorithm always gets at least 80% of the optimal value.

Approximation algorithms are important because many famous combinatorial optimization problems are NP-hard. When exact polynomial-time algorithms are not known, approximation algorithms can still provide reliable performance guarantees. Vazirani’s textbook gives a systematic treatment of this field and its central techniques (Vazirani, 2001).

A famous example is MaxCut.

In the MaxCut problem, we are given a graph. A graph has vertices, also called nodes, and edges connecting pairs of vertices. The goal is to split the vertices into two groups so that as many edges as possible cross between the groups.

For example, imagine four cities connected by roads. We want to color each city red or blue so that as many roads as possible connect cities of different colors. Each coloring gives a cut. The objective is to maximize the number, or weight, of crossing edges.

MaxCut is important in quantum optimization because QAOA was originally introduced using problems such as MaxCut. It is also important classically because it has a celebrated approximation algorithm by Goemans and Williamson. Their algorithm uses semidefinite programming and achieves an approximation ratio of about 0.878 for MaxCut (Goemans and Williamson, 1995).

That number matters. If a quantum algorithm claims to perform well on MaxCut, it should be compared not only with random guessing but also with strong classical methods, including algorithms with known approximation guarantees.

Approximation algorithms teach a broader lesson:

> “Good” is not enough. We should ask, “Good compared to what?”

In optimization, quality is meaningful only relative to a baseline: the true optimum, a lower or upper bound, a classical heuristic, an approximation guarantee, or the best known solver performance.

3.9 Relaxations: Making a Hard Problem Easier

A relaxation is a simpler version of an optimization problem obtained by removing or weakening some constraints.

Relaxations are used throughout classical optimization. They are especially important in branch and bound and integer programming.

Suppose a problem has binary variables:

$$x_i \in \{0,1\}.$$

A common relaxation replaces this with

$$0 \le x_i \le 1.$$

Now the variable can take fractional values such as 0.3 or 0.8. The relaxed problem is usually easier to solve.

Why is this useful?

For a minimization problem, if the relaxed problem has a minimum value of 50, then the original problem cannot have a value below 50. The relaxed problem gives a lower bound.

For a maximization problem, a relaxation often gives an upper bound.

Here is a simple example.

Suppose we must choose either 0 or 1 for each variable, but the relaxed solution says

$$x_1 = 0.7,\ x_2 = 1,\ x_3 = 0.2.$$

This fractional solution may not be feasible for the original problem. But its objective value can still tell us something about what is possible.

Relaxations are one of the main ways classical solvers reason about huge search spaces. Instead of checking every possible binary assignment, they solve easier related problems to prove that some regions are promising and others are not.

This idea will return when we discuss quantum optimization. Mapping a problem to QUBO or Ising form is not the only possible path. Sometimes a classical relaxation gives excellent bounds or even a near-complete solution before any quantum computation is considered.

3.10 Why Classical Methods Matter for Variational Quantum Algorithms

Variational quantum algorithms are hybrid. This means classical optimization is not outside the quantum algorithm; it is part of the algorithm.

A typical variational quantum optimization loop works like this:

1. Choose parameters for a quantum circuit.
2. Run the circuit many times.
3. Measure bitstrings or expectation values.
4. Estimate a cost.
5. Use a classical optimizer to choose new parameters.
6. Repeat.

The quantum computer does not usually decide its own next parameters. The classical optimizer does.

For continuous circuit parameters, methods related to gradient descent may be used. In noisy settings, gradient-free or stochastic methods may be preferred. Chapter 9 will study those choices carefully.

For the underlying discrete optimization problem, methods such as simulated annealing, local search, branch and bound, and integer programming often provide baselines.

This creates two layers of classical comparison.

The first layer is parameter optimization:

> How well does the classical optimizer train the quantum circuit?

The second layer is problem optimization:

> How well does the final quantum method solve the original optimization problem compared with classical solvers?

Both questions matter.

A QAOA experiment may fail because the quantum circuit is not expressive enough, because the measurements are too noisy, because the classical optimizer gets stuck, or because the problem is simply easier for a classical solver. Without classical optimization knowledge, these causes are easy to confuse.

3.11 A Small Example: Comparing Methods on a Binary Problem

Let us use a tiny binary optimization problem to see how the methods differ.

Suppose we have four binary variables,

$$x_1, x_2, x_3, x_4 \in \{0,1\}$$

and we want to minimize

$$C(x) = -3x_1 - 2x_2 - 4x_3 - x_4 + 5x_1x_2 + 2x_3x_4.$$

This is a quadratic binary objective because it contains products such as x_1x_3. Problems of this kind are closely related to QUBO models, which we will study in Chapter 11.

There are only

$$2^4 = 16$$

possible assignments, so we could solve this by brute force. But imagine there were 400 variables instead of 4. Then brute force would be impossible.

Different classical methods would approach the problem differently.

A local search heuristic might start from

$$x = (0,0,0,0)$$

and flip one bit at a time if the flip improves the cost.

Simulated annealing might also flip bits, but it might sometimes accept a worse flip to escape a local minimum.

Branch and bound would build a search tree by fixing variables and pruning branches whose bounds show they cannot beat the current best solution.

Integer programming would express the binary variables and quadratic terms in a solver-friendly form, possibly using additional variables to linearize products.

An approximation algorithm might apply if the problem belongs to a class with known guarantees.

A variational quantum algorithm would encode the cost into a quantum measurement problem, use a parameterized circuit to sample bitstrings, and then use a classical optimizer to adjust circuit parameters.

The important point is not that one method is always best. The important point is that each method has a different kind of strength.

Gradient descent is natural for smooth continuous landscapes.

Simulated annealing is simple and good at randomized exploration.

Branch and bound can prove optimality.

Integer programming provides a powerful modeling and solving framework.

Heuristics can be fast and flexible.

Approximation algorithms can provide mathematical guarantees.

Quantum optimization must be evaluated in this ecosystem, not in isolation.

3.12 Common Mistakes When Comparing Quantum and Classical Optimization

Before leaving this chapter, let us name several common mistakes.

The first mistake is comparing a quantum method only against random guessing. Random guessing is rarely a strong baseline.

The second mistake is using a weak classical optimizer and concluding that the quantum method is strong. If the classical comparison is poorly tuned, the conclusion is unreliable.

The third mistake is ignoring problem structure. A generic QUBO solver may perform poorly on a problem that a specialized classical algorithm solves efficiently.

The fourth mistake is comparing only final solution quality while ignoring runtime, measurement cost, and hardware overhead.

The fifth mistake is reporting success only on very small instances. Small examples are useful for learning and debugging, but practical claims require scaling evidence.

The sixth mistake is forgetting that variational quantum algorithms themselves depend on classical optimizers. If the classical outer loop fails, the quantum algorithm may fail even if the circuit family is theoretically capable of representing a good solution.

These mistakes do not mean quantum optimization is unimportant. They mean that careful comparison is part of the science.

3.13 Chapter Summary

Classical optimization is the foundation on which quantum optimization must be understood.

In this chapter, we studied several essential methods.

Gradient descent uses local slope information to improve continuous variables. It is central in machine learning and variational circuit training.

Simulated annealing uses randomness and a decreasing temperature to explore discrete search spaces and escape local minima.

Branch and bound searches a tree of decisions while pruning branches that cannot contain an optimal solution.

Integer programming models whole-number and binary decisions using algebraic objectives and constraints.

Heuristic search uses practical rules to find good solutions quickly, often without formal guarantees.

Approximation algorithms provide provable guarantees for solution quality, especially for hard combinatorial problems.

The main lesson is simple but powerful:

> Quantum optimization methods must be compared against strong classical optimization methods.

In the next chapter, we turn to quantum computing itself. We will build the basic ideas of qubits, measurement, superposition, amplitudes, tensor products, entanglement, gates, and circuits. Then we will be ready to understand how quantum algorithms create and sample structured probability distributions.

References

- Bertsimas, D., & Tsitsiklis, J. N. (1997). Introduction to Linear Optimization. Athena Scientific.

- Boyd, S., & Vandenberghe, L. (2004). Convex Optimization. Cambridge University Press.

- Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.

- Goemans, M. X., & Williamson, D. P. (1995). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM, 42(6), 1115–1145. https://doi.org/10.1145/227683.227684

- Hoos, H. H., & Stützle, T. (2004). Stochastic Local Search: Foundations and Applications. Morgan Kaufmann.

- Kirkpatrick, S., Gelatt, C. D., Jr., & Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598), 671–680. https://doi.org/10.1126/science.220.4598.671

- Land, A. H., & Doig, A. G. (1960). An automatic method of solving discrete programming problems. Econometrica, 28(3), 497–520.

- Nemhauser, G. L., & Wolsey, L. A. (1988). Integer and Combinatorial Optimization. Wiley.

- Nocedal, J., & Wright, S. J. (2006). Numerical Optimization (2nd ed.). Springer.

- Vazirani, V. V. (2001). Approximation Algorithms. Springer.

Document information

Chapter 3: Classical Optimization Methods You Must Know

Project	Variational Quantum Algorithms for Optimization
Document	Document 1.7
Author	phone
Verifier	Not verified
Downloaded	July 04, 2026 22:51 KST
Status	Working
Document link	https://theorytrace.com/projects/variational-quantum-algorithms-for-optimization/documents/chapter-3-classical-optimization-methods-you-must-know/