Tutorials /

Solve higher-order binary optimization problems with Q-CTRL's Optimization Solver

Beginner Optimization Functions

Usage estimate: 24 minutes on ibm_fez. (NOTE: This is an estimate only. Your runtime may vary.)

Background

This tutorial demonstrates how to solve a higher-order binary optimization (HOBO) problem using the Optimization Solver, a Qiskit Function by Q-CTRL Fire Opal. The example demonstrated in this tutorial is an optimization problem designed to find the ground-state energy of a random-bond 156-qubit Ising model possessing cubic terms. The Optimization Solver can be used for general optimization problems that can be defined as an objective function.

The Optimization fully automates the hardware-aware implementation steps of solving optimization problems on quantum hardware, and by leveraging Performance Management for the quantum execution, it achieves accurate solutions at utility scale. For a detailed summary of the full Optimization Solver workflow and benchmarking results, refer to the published manuscript.

This tutorial walks through the following steps:

1. Define the problem as an objective function
2. Run the hybrid algorithm using the Fire Opal Optimization Solver
3. Evaluate results

Requirements

Before starting this tutorial, ensure that you have the following installed:

– Qiskit Functions (`pip install qiskit-ibm-catalog`)

– SymPy (`pip install sympy`)

– Matplotlib (`pip install matplotlib`)

You will also need to get access to the Optimization Solver function. Fill out the form to request access.

Setup

First, import the required packages and tools.

```
# Qiskit Functions Catalog
from qiskit_ibm_catalog import QiskitFunctionsCatalog

# SymPy tools for constructing objective function
from sympy import Poly
from sympy import symbols, srepr

# Tools for plotting and evaluating results
import numpy as np
import matplotlib.pyplot as plt
from sympy import lambdify
```

No output produced

Define your IBM Quantum™ Platform credentials, which will be used throughout the tutorial to authenticate to Qiskit Runtime and Qiskit Functions.

```
# Credentials
token = "<YOUR_IQP_API_TOKEN>"
hub = "<YOUR_IQP_HUB>"
group = "<YOUR_IQP_GROUP>"
project = "<YOUR_IQP_PROJECT>"
```

No output produced

Step 1: Define the problem as an objective function

The Optimization Solver accepts either an objective function or a graph as input. In this tutorial, the Ising spin glass minimization problem is defined as an objective function, and it has been tailored for the heavy-hex topology of IBM devices.

Because this objective function contains cubic, quadratic, and linear terms, it falls into the HOBO class of problems, known to be considerably more complicated to solve than conventional quadratic unconstrained binary optimization (QUBO) problems.

For detailed discussion of the construction of the problem definition and previous results obtained from the Optimization Solver refer to this technical manuscript. The problem was originally defined and evaluated as part of a paper published by Los Alamos National Laboratory, and it has been adapted to leverage the full device width of the 156-qubit IBM Quantum Heron processors.

```
qubit_count = 156

# Create symbolic variables to represent qubits
x = symbols([f"x[{i}]" for i in range(qubit_count)])

# # Define a polynomial representing a spin glass model
spin_glass_poly = Poly(-4*x[0]*x[1] - 8*x[1]*x[2]*x[3] + 8*x[1]*x[2] + 4*x[1]*x[3] -
```

No output produced

Step 2: Run the hybrid algorithm using the Fire Opal Optimization Solver

Now use the Optimization Solver Qiskit Function to run the algorithm. Behind the scenes, the Optimization Solver takes care of mapping the problem to a hybrid quantum algorithm, running the quantum circuits with error suppression, and performing the classical optimization.

```
# Authenticate to the Qiskit Functions Catalog
catalog = QiskitFunctionsCatalog(token=token)

# Load the function
solver = catalog.load('q-ctrl/optimization-solver')
```

No output produced

Check to ensure that the chosen device has at least 156 qubits.

```
# Specify the target backend name
backend_name = "<CHOOSE_A_BACKEND>"
```

No output produced

The Solver accepts a string representation of the objective function.

```
# Convert the objective function to string format
spin_glass_poly_as_str = srepr(spin_glass_poly)
```

No output produced

```
# Run the problem
spin_glass_job = solver.run(
    problem = spin_glass_poly_as_str,
    instance = hub + "/" + group + "/" + project,
    run_options={
        "backend_name": backend_name
    }
)
```

No output produced

You can use the familiar Qiskit Serverless APIs to check your Qiskit Function workload's status:

```
# Get job status
spin_glass_job.status()
```

No output produced

The Solver returns a dictionary with the solution and associated metadata, such as the solution bitstring, number of iterations, and mapping of variables to bitstring. For a full definition of the Solver's inputs and outputs, check out the documentation.

```
# Poll for results
result = spin_glass_job.result()
```

No output produced

```
# Get the final bitstring distribution and set the number of shots
distribution = result['final_bitstring_distribution']
```

No output produced

Step 3: Evaluate results

```
# Get the solution ground state energy
print(f"Minimum ground state energy: {result["solution_bitstring_cost"]}")
```

Output:

```
Minimum ground state energy: -242.0
```

The Solver found the correct solution, which was validated using classical optimization software. The complexity of this utility-scale problem requires an advanced optimization software to be solved classically, such as IBM ILOG CPLEX Optimization Studio (CPLEX) or Gurobi Optimization.

As a visual analysis of the quality of results, you can plot the results by calculating the cost values from the bitstrings and their probabilities. For comparison, plot the results alongside a distribution of randomly sampled bitstrings, which is equivalent to a "brute-force" classical solution. If the algorithm consistently finds lower costs, it suggests the quantum algorithm is effectively solving the optimization problem.

```python
def plot_cost_histogram(costs, probs, distribution, qubit_count, bitstring_cost)
    """Plots a histogram comparing the cost distributions of Q-CTRL Solver and rando

    # Set figure DPI for higher resolution and font size for labels
    plt.rcParams['figure.dpi'] = 300
    plt.rcParams.update({'font.size': 6})  # Set default font size to 6

    # Define labels and colors for the plot
    labels = ["Q-CTRL Solver", "Random Sampling"]
    colors = ['#680CE9', '#E04542']

    # Calculate total shots (total number of bitstrings in the distribution)
    shots = sum(distribution.values())

    # Generate random bitstrings for comparison (random sampling)
    rng = np.random.default_rng(seed=0)
    random_array = rng.integers(0, 2, size=(shots, qubit_count))  # Generate random
    random_bitstrings = ["".join(row.astype(str)) for row in random_array]

    # Compute the cost for each random bitstring
    random_costs = [bitstring_cost(k) for k in random_bitstrings]

    # Set uniform probabilities for the random sampling
    random_probs = np.ones(shape=(shots,)) / shots  # Equal probability for each ran

    # Find the minimum and maximum costs for binning the histogram
    min_cost = np.min(costs)
    max_cost = np.max(random_costs)

    # Create a histogram plot with a smaller figure size (4x2 inches)
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(4, 2))

    # Plot histograms for the Q-CTRL solver and random sampling costs
    _, _, _ = ax.hist(
        [costs, random_costs],  # Data for the two histograms
```

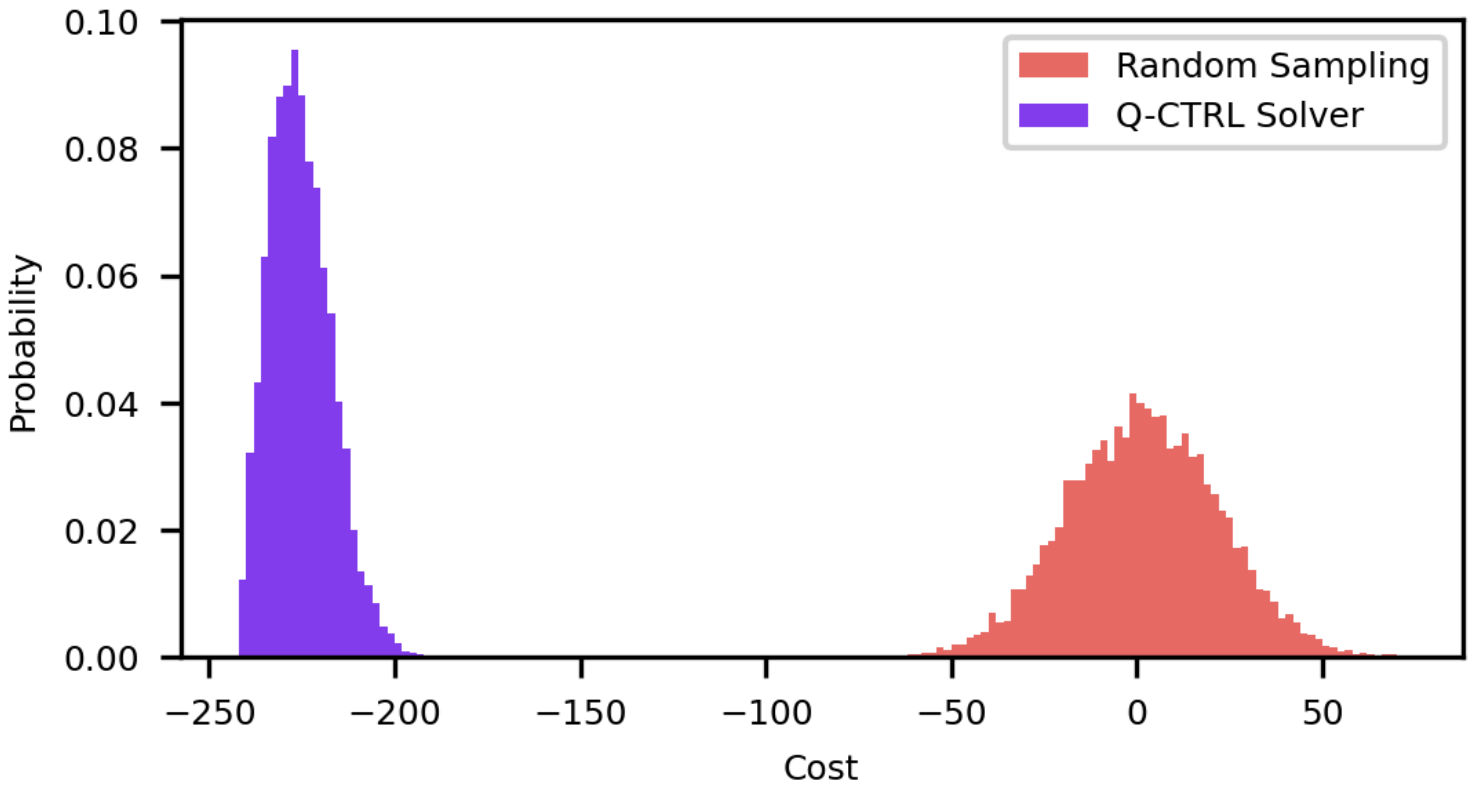
```
        np.arange(min_cost, max_cost, 2),  # Bins for the histogram
        weights=[probs, random_probs],  # Probabilities for each data set
        label=labels,  # Labels for the legend
        color=colors,  # Colors for each histogram
        histtype='stepfilled',  # Filled step histogram
        align="mid",  # Align bars to the bin center
        alpha=0.8,  # Transparency
    )

    # Set the x and y labels for the plot
    ax.set_xlabel("Cost")
    ax.set_ylabel("Probability")

    # Add the legend to the plot
    ax.legend()

    # Show the plot
    plt.show()
```

No output produced

```
# Convert spin_glass_poly into a NumPy-compatible function
poly_as_numpy_function = lambdify(x, spin_glass_poly.as_expr(), "numpy")

# Function to compute the cost of a given bitstring using spin_glass_poly
def bitstring_cost(bitstring: str) -> float:
    # Convert bitstring to a reversed list of integers (0s and 1s)
    return float(poly_as_numpy_function(*[int(b) for b in str(bitstring[::-1])]))

# Calculate the cost of each bitstring in the distribution
costs = [bitstring_cost(k) for k, _ in distribution.items()]

# Extract probabilities from the bitstring distribution
probs = np.array([v for _, v in distribution.items()])
```

```
probs = probs / sum(probs)  # Normalize to get probabilities

plot_cost_histogram(costs, probs, distribution, qubit_count, bitstring_cost)
```

Output:



Since the goal of this optimization algorithm is to find the minimum ground state of the Ising model, lower values indicate better solutions. Therefore, it's visually apparent that the solutions generated by the Fire Opal

Optimization Solver far outperform random selection.

Was this page helpful?