Star 18

Kipu Digitized Counterdiabatic Quantum Optimization - DCQO

This service implements Kipu's Digitized Counterdiabatic Quantum Optimization (DCQO) algorithm to solve optimization problems. It uses Kipu's technique of *algorithmic compression* by using counterdiabatic protocols.

Usage

Quantum computing can be used to find the ground state (minimum energy state) of an Ising Hamiltonian of the form

$$H = \sum_{i}^{N} h_i \sigma_i^z + \sum_{i}^{N} \sum_{j>i}^{N} J_{ij} \sigma_i^z \sigma_j^z$$

Many standard optimization problems from industrial use cases can be restated in Ising form. We assume that the user has already a problem in the form described above.

Our service requires you to provide

- the single body Ising coefficients h_i,
- and the two-body coefficients J_{ij},
- additional parameters (specified below).

The input data needs to be supplied in JSON format. A full JSON input for the service has the following form:

```
{
  "data": {
    "optimization": {
      "coefficients": {
        "(0,)": 0.05009721367530462,
        "(0, 1)": 0.1589373302733451,
        "(0, 2)": -0.1465938630044573,
        "(0, 3)": 0.1444447352374391,
        "(1,)": -0.008103805367511346,
        "(1, 2)": -0.10385855447216596,
        "(1, 3)": 0.014771937124279128,
        "(2,)": -0.5638808178837599,
        "(2, 3)": -0.1014721471152484,
        "(3,)": 0.023722798165331543
      },
      "annealing_time": 0.7,
      "trotter_steps": 2,
      "mode": "CD"
    }
  },
  "params": {
    "backend": "azure.ionq.simulator",
    "shots": 1024
  }
}
```

The `data` section contains the specification of the circuit to be constructed in the `optimization` attribute, including the Hamiltonian coefficients, annealing time, number of Trotter steps, and the mode of operation.

The `params` section includes the parameters for the submission of the circuit to a hardware backend, including the number of shots and the backend.

Hamiltonian coefficients

The Hamiltonian coefficients h_i and J_{ij} are specified in the `coefficients` attribute, where each key `(i,)` corresponds to the single body coefficient h_i and the `(i,j)` key corresponds to the two-body interaction coefficient J_{ij}. **Note that we use 0-indexing for the coefficients.**

DCQO Hyperparameters

Annealing time: `annealing_time`

The total time over which the quantum system evolves from the initial Hamiltonian to the problem Hamiltonian. It is a crucial parameter in quantum annealing and counterdiabatic quantum computing, as it determines the duration of the evolution process.

Trotter stteps: `trotter_steps`

The number of discrete time steps used to approximate the continuous evolution of the quantum system. This is based on the Trotter-Suzuki decomposition, which breaks down the exponential of a sum of non-commuting operators into a product of exponentials of the individual operators. More Trotter steps generally lead to a more accurate approximation of the continuous evolution but also increase the complexity of the quantum circuit.

Evolution Mode: `mode`

The mode determines the type of Hamiltonian evolution used in the algorithm. In the context of Counterdiabatic Quantum Computing (CQC), there are typically three modes:

- Counterdiabatic: `mode = CD`
  Only the counterdiabatic term is included in the Hamiltonian. This mode gives you Kipu's algorithmic compression and should be used with a small annealing time $0.5 < T < 1.5$.
- Full: `mode = FULL`
  Both the adiabatic and counterdiabatic terms are included in the Hamiltonian.

Circuit submission parameters

The `params` section includes the `backend` attribute to set the hardware backend for the quantum computation and the `shots` number of shots (repetitions) for the simulation. If these parameters are not specified, the service will not submit the circuit to a hardware provider and instead return a circuit in QASM format. See below for the output information.

Results

If no backend is specified in the input JSON, the service will compose a circuit and return it.
The circuit will be in the QASM text format that can be interpreted by different frameworks (Qiskit, Pennylane, Amazon Braket, etc). An example is shown below.

```
{'qasm2': 'OPENQASM 2.0;\ninclude "qelib1.inc";\nqreg q[4];\ncreg c[4];\nh q[0];\nh q[1];\nh q[2];\nh q[3];\nsdg q[0];\nh
q[0];\nrz(0.09988203608620727) q[0];\nh q[0];\ns q[0];\nsdg q[0];\nh q[0];\ncx q[0],q[1];\nrz(0.3168835748171212) q[1];\ncx q[0],q[1];\nh
q[0];\ns q[0];\nsdg q[0];\nh q[0];\ncx q[0],q[2];\nrz(-0.2922736104552165) q[2];\ncx q[0],q[2];\nh q[0];\ns q[0];\nsdg q[0];\nh q[0];
...
'}
```

If the backend and counts are in the input, the service will submit the circuit to the hardware backend and return a JSON file, with the bitstring counts in the `counts` attribute and information of the job submission (`job_id` , `backend_name` , `shots`).

Bitstring ordering and spin variable convention

The returned bitstrings follow the big endian convention, where the bit `x[i]` of a bitstring `x` corresponds to the spin variable (qubit) σ_i^z. The map from binary variables $x_i = 0, 1$ to spin variables $\sigma_i^z = 1, -1$ is $\sigma_i^z = 1 - 2x_i$.

Similarly, if a circuit is returned qubit i in the circuit will correspond to the i-th variable in the input data.

```
{'backend_name': 'azure.ionq.simulator',
 'job_id': '27a9e8c9-b50a-446b-86b7-706cefd3a707',
 'counts': {'1000': 306,
  '1110': 41,
  '1011': 44,
  '1010': 13,
  '0010': 1,
  '0101': 209,
  '0110': 5,
  '0011': 5,
  '0000': 19,
  '0100': 103,
  '1111': 38,
  '1100': 68,
  '0001': 87,
  '1001': 67,
  '0111': 18},
 'shots': 1024}
```


How to use it

After subscribing to the service via an application, one can use the service with the following code

```
from planqk.service.client import PlanqkServiceClient

consumer_key = "your_consumer_key"
consumer_secret = "your_consumer_secret"
service_endpoint = "https://gateway.platform.planqk.de/kipu-quantum/kipu-digitized-counterdiabatic-quantum-optimization---dcqo/1.0.0"

client = PlanqkServiceClient(service_endpoint, consumer_key, consumer_secret)

 data =  {
    "optimization": {
      "coefficients": {
        "(0,)": 0.05009721367530462,
        "(0, 1)": 0.1589373302733451,
        "(0, 2)": -0.1465938630044573,
        "(0, 3)": 0.1444447352374391,
        "(1,)": -0.008103805367511346,
        "(1, 2)": -0.10385855447216596,
        "(1, 3)": 0.014771937124279128,
        "(2,)": -0.5638808178837599,
        "(2, 3)": -0.1014721471152484,
        "(3,)": 0.023722798165331543
      },
      "annealing_time": 0.7,
      "trotter_steps": 2,
      "mode": "CD"
    }

params = {
    "backend": "azure.ionq.simulator",
```

API

ID	92197339-c134-4feb-8535-09ad1d90fb88
Endpoint	https://gateway.platform.planqk.de/kipu-quantum/kipu-digitized-counterdiabatic-quantum-optimization---dcqo/1.0.0

Status API

GET / Health checking endpoint

Parameters

No parameters

Responses

Code	Description	Links
200	OK Media type application/json Controls `Accept` header. Example Value Schema	*No links*

```
{
  "status": "Service is up and running"
}
```

Service API

POST / Asynchronous execution of the service

GET /{id} Check execution status

Parameters

Name	Description
id * required `string` *(path)*	The ID of a certain execution id

Responses

Code	Description	Links
200	Ok Media type application/json Controls `Accept` header. Example Value Schema	*No links*

```
job {
   id            string
   status        string
                 Enum:
                   Array [ 6 ]
   createdAt     string
   startedAt     string
   endedAt       string
}
example: OrderedMap { "id": "87cb778e-ac43-11ec-b909-0242ac120002", "status": "SUCCEEDED", "createdAt": "2022-01-01 22:38:08", "startedAt": "2022-01-01 22:39:08", "endedAt": "2022-01-01 22:40:08" }
```

Code	Description	Links
401	Unauthorized	*No links*
403	Forbidden	*No links*
404	Not found	*No links*
500	Internal server error	*No links*

GET `/{id}/result` Get the result of an execution

Parameters

Name	Description
id * required `string` *(path)*	The ID of a certain execution id

Responses

Code	Description	Links
200	Ok Media type application/json Controls `Accept` header. **Example Value** **Schema** `{ "qasm2": "OPENQASM 2.0" }`	*No links*
401	Unauthorized	*No links*
403	Forbidden	*No links*
404	Not found	*No links*
500	Internal server error	*No links*

GET `/{id}/interim-results` Get the last or a list of interim results of an execution

PUT `/{id}/cancel` Cancel an execution

How to use this Service

The PlanQK Service SDK serves as your gatew Basic Account: Don't worry, using PlanQK won't cost you a dime! ice through Python. Follow the steps below for a smooth integration experience.

Start by installing the PlanQK Service SDK Python library with the following command:

```
pip install --upgrade planqk-service-sdk
```

Prepare the consumer key and consumer secret by subscribing to this service using an application. [Applications](https://platform.planqk.de) on the PlanQK Platform hold all necessary information for secure communication with the service. Just use the *Subscribe* button on the right and select the appropriate application.

Replace `your_consumer_key` and `your_secret_key` in the code snippet below with the credentials provided in your application:

```
from planqk.service.client import PlanqkServiceClient

consumer_key = "your_consumer_key"
consumer_secret = "your_consumer_secret"
service_endpoint = "https://gateway.platform.planqk.de/kipu-quantum/kipu-digitized-counterdiabatic-quantum-optimization--
-dcqo/1.0.0"

client = PlanqkServiceClient(service_endpoint, consumer_key, consumer_secret)

# prepare your input data and parameters
data = {"optimization":{"coefficients":{"(0,)":0.05009721367530462,"(0, 1)":0.1589373302733451,"(0,
2)":-0.1465938630044573,"(0, 3)":0.1444447352374391,"(1,)":-0.008103805367511346,"(1, 2)":-0.10385855447216596,"(1,
3)":0.014771937124279128,"(2,)":-0.5638808178837599,"(2, 3)":-0.1014721471152484,"
(3,)":0.023722798165331543},"annealing_time":0.7,"trotter_steps":2,"mode":"CD"}}
params = {"backend":"azure.ionq.simulator","shots":1024}

# start the execution
job = client.start_execution(data=data, params=params)

# wait for the result
result = client.get_result(job.id)
```