D-Wave Hybrid

An Overview

Radomir Stevanovic

D-Wave
Ocean Software Stack

Ocean Software

Optimization
- Social Network Analysis
- Traffic Flow
- Web Advertising
- New Application

Constraint Satisfaction
- Portfolio Optimization
- Scheduling
- Circuit Fault Detection
- New Application

Applications

Mapping Methods
Uniform Sampler API

Samplers

Compute Resources

Ocean Software Stack

Graph Mapping
Constraint Compilation
New Mapping Method

Problem Suitable for QPU: Binary Quadratic Model (BQM)

Simulated Annealing
D-Wave API
Hybrid Sampler
New Sampler

CPUs and GPUs
QPU's
github.com/dwavesystems/dwave-hybrid

- Hybrid Asynchronous Decomposition Sampler framework
  - Minimal, Python, solver/sampler-building framework, built atop Ocean tools
  - Leverages **quantum** and **classical** resources
  - Independent parts are executed **concurrently**
  - Problems are **broken into pieces** that fit the compute resources
  - Uses sample sets (probabilistic approach)
Motivation

Algorithm 1 Partitioning algorithm implemented by qbsolv

1. Input: QUBO instance
2. # best.energy is the lowest value found to date
3. # best.solution is the solution bit vector corresponding to the lowest value so far
4. # index is the indices of the bits in the solution, sorted from most to least impact on value
5. # Get initial estimate of minimum value and backbone
6. solution ← random 0/1 vector
7. (best.energy, best.solution) ← TabuSearch(QUBO, solution)
8. index ← OrderByImpact(QUBO, best_solution)
9. passCount ← 0
10. solution ← best.solution
11. while passCount < numberRepeats do
12. change ← false
13. for i : 0 < fraction * Size(QUBO); i : subQUBOSize
14. # select subQUBO with other variables clamped
15. subQUBO ← Clamp(QUBO, solution, index, sub.index)
16. (sub.energy, sub.solution) ← DWaveSearch(subQUBO)
17. # project onto full solution
18. if solution[sub.index] ≠ sub.solution then
19. solution ← sub.solution
20. change ← true
21. end if
22. end for
23. if not change then
24. Randomize(solution[0:1-1])
25. end if
26. if energy(solution) < TabuSearch(QUBO, solution)
27. if energy < best.energy then
28. best.energy ← energy
29. best.solution ← solution
30. passCount ← 0
31. else
32. passCount ++
33. end if
34. index ← OrderByImpact(QUBO, solution)
35. end while
36. Output: best.energy, best.solution

Loop(RacingBranches(
  InterruptableTabuSampler(),
  EnergyImpactDecomposer(size=50)
  | QPUSubproblemAutoEmbeddingSampler()
  | SplatComposer()
)) | ArgMin()
Goals

• Code outlines/visualizes the algorithm
• Code is easy to tweak, extend, experiment, benchmark and profile
• Simplicity is balanced with expressiveness
• Library of building blocks provided, extendable by developers

Loop(RacingBranches(
    InterruptableTabuSampler(),
    EnergyImpactDecomposer(size=50)
    | QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()
) | ArgMin())

© D-Wave Systems Inc.
In [1]: import dimod, hybrid

...: bqm = dimod.BinaryQuadraticModel({}, {'ab': 1, 'bc': -1, 'ca': 1}, 0, dimod.SPIN)
...: state = hybrid.State.from_sample(hybrid.min_sample(bqm), bqm)
...:
...: workflow = hybrid.Loop(hybrid.RacingBranches(
...:     hybrid.InterruptTabuSampler(),
...:     hybrid.EnergyImpactDecomposer(size=1)
...: )
...:     hybrid.SplatComposer()
...: ) | hybrid.ArgMin(), convergence=3)

In [2]: state
Out[2]:
{'problem': BinaryQuadraticModel({'b': 0, 'a': 0, 'c': 0}, {'a', 'c'}: 1, {'a', 'b'}: 1, {'b', 'c'}: -1}, 0, Vartype.SPIN,
'samples': SampleSet(rec.array([[-1, 1, -1, 1, 1]]),
dtype=[('sample', 'i1', (3,)), ('energy', '<f8'), ('num_occurrences', '<i8')]), ['b', 'a', 'c'], [], 'SPIN')

In [4]: f = workflow.run(state)
In [5]: f
Out[5]: <Future at 0x7f5e777a9710 state=running>
In [6]: f
Out[6]: <Future at 0x7f5e777a9710 state=finished returned State>
In [7]: f.result()
Out[7]:
{'problem': BinaryQuadraticModel({'b': 0, 'a': 0, 'c': 0}, {'a', 'c'}: 1, {'a', 'b'}: 1, {'b', 'c'}: -1}, 0, Vartype.SPIN,
'samples': SampleSet(rec.array([[-1, 1, -1, -3, 1]]),
dtype=[('sample', 'i1', (3,)), ('energy', '<f8'), ('num_occurrences', '<i8')]), ['b', 'a', 'c'], [], 'SPIN')
Framework Primitive: Runnable Type

Loop(RacingBranches(
  InterruptableTabuSampler(),
  EnergyImpactDecomposer(size=50)
  | QPUSubproblemAutoEmbeddingSampler()
  | SplatComposer()
) | ArgMin())

• All components implement the Runnable type
• Act on input State(s), produce output State(s)
• Execute asynchronously (.run() and .stop())
• Composable top-down (tree); traits constrain connectivity; profiled by default
Framework Primitive: State

- Immutable mapping type
- Passed between Runnable components, wrapped in Future
- Carries the problem, subproblem, samples, etc.
- Compliance with component's traits checked during runtime
Modifying Workflow Parameters

```python
subproblem = EnergyImpactDecomposer(size=50, rolling_history=0.15)
subsampler = QPUSubproblemAutoEmbeddingSampler()
    | SplatComposer()

iteration = RacingBranches(
    InterruptableTabuSampler(),
    subproblems | subsampler
) | ArgMin()

workflow = Loop(iteration, max_iter=1e3, convergence=3)
```

• Solve subproblems (of size 50 variables), at different points (samples), one per iteration
  – Keep unrolling (deconstructing) up to 15% of the input problem variables (in order of energy impact)

• Upper bound on loop count, terminate if no improvement after 3 iterations
Modifying Workflow Structure

```
subproblems = Unwind(
    EnergyImpactDecomposer(size=50, rolling_history=0.15, silent_rewind=False))

subsample = Map(QPUSubproblemAutoEmbeddingSampler())
    | Reduce(Lambda(merge_substates))
    | SplatComposer()

iteration = RacingBranches(
    InterruptableTabuSampler(),
    subproblems | subsample
) | ArgMin()

workflow = Loop(iteration, max_iter=1e3, convergence=3)
```

- Deconstruct 15% of the problem into multiple subproblems (at the same sample)
  - Solve them all in parallel on the QPU
  - Merge subsamples
  - Compose with the original sample
### Modifying Workflow Structure

| subproblems = Unwind( | Deconstruct 15% of the problem into subproblems |
| EnergyImpactDecomposer(size=50, rolling_history=0.15, silent_rewind=False)) |
| qpu = Map(QPUSubproblemAutoEmbeddingSampler()) |
| Reduce(Lambda(merge_substates)) |
| SplatComposer() |
| random = Map(RandomSubproblemSampler()) |
| Reduce(Lambda(merge_substates)) |
| SplatComposer() |
| subsampler = Parallel(qpu, random, endomorphic=False) | ArgMin() |
| iteration = RacingBranches( |
| InterruptableTabuSampler(), |
| subproblems | subsampler |
| ) | ArgMin() |
| workflow = Loop(iteration, max_iter=1e3, convergence=3) |

- Solve them all in parallel on the QPU
- But also solve them using a second/classical subsampler (random here)
- All in parallel
On Problem Decomposition

• When problem doesn't fit the computing device
  – Memory, parallelism/cores, bit-length, GPU pipeline, QPU size/structure

• Tailored to problem class and purpose/device
  – "no free lunch"
  – no "right", or general, approach to problem decomposition

• No shortage of ideas for decomposition:
  – Energy based, connectivity/structure based...

• (Or hybrid solvers):
  – Based on tabu search, parallel tempering, dialectic search, branch and bound, diversity-preserving sampling, genetic algorithms...
Constructing Runnables

• Extend hybrid.Runnable's methods:
  • init(), next(), error(), halt()

• Implement a flow control block, a sampler, or a problem decomposer tailored to your problem (class)

• Share it!
Contributions Welcome

- [https://github.com/dwavesystems/dwave-hybrid/issues](https://github.com/dwavesystems/dwave-hybrid/issues)
  - more samplers (parallel tempering, ICM, reverse anneal)
  - more decomposing strategies (e.g. BFS, PFS traversal in EID)
  - more composing strategies – better support for multiple samples per state (alternative to "best sample splat")
  - more flow control blocks
  - sample diversity-preserving sampleset pruning
  - CoW State
  - more Runnable Executors (celery, asyncio?)

- Developer survey
  - [https://www.surveymonkey.com/r/LJM96GT](https://www.surveymonkey.com/r/LJM96GT)