技術・研究報告

組合せ最適化問題に対する量子アニーリング方式の試用と考察

若谷彰良

甲南大学 知能情報学部
神戸市東灘区岡本８－９－１, 658-8501

(受理日 2022 年 5 月 10 日)

概 要

近年, 量子計算が注目されている. 量子計算には量子ゲート方式と量子アニーリング方式があるが, 実用的に利用されているいるのは後者であり, デジタル方式のものも含め複数の実機が発表されている. しかし, 実問題をイジングモデルの相互作用とバイアスに変換することは煩雑であり, 自動的にイジング最適化を行なうソフトウェアの利用が重要である. 本稿では, そのためのソフトウェアの一つである Fixstars Amplify を用いて巡回セールスマン問題を Fixstars Amplify Annealing Engine 上で実行し, 古典的ソルバーの GLPK (GNU Linear Programming Kit) のオンプレミスの Linux マシン上での実行結果との比較を行い, その試用を通して実用性の考察を行なう.

キーワード: 量子計算，組合せ最適化，巡回セールスマン問題, イジングモデル, QUBO

1 はじめに

ムーアの法則が終焉に近づき, 新しい計算方式として量子計算に大きな注目が集まってきている. また, 情報処理学会にも量子ソフトウェアの研究会が発足する [1] など, その重要性は高まっている. 量子計算には, デジタルコンピュータの量子計算版である量子ゲート方式と, アナログコンピュータの量子計算版である量子アニーリング方式がある. 量子ゲート方式は商用化はされているが, 性能面で実用化のレベルには達しておらず, 実用化のカギはノイズへの対策であり, その誤り訂正技術が重要となる. 量子ゲート方式は, NISQ (Noisy Intermediate-Scale Quantum) と FISQ (Fault Tolerant Quantum Computer) に分類され, NISQ では計算途中で発生するノイズの影響を前提としたアルゴリズムに特化し, 古典コンピュータの組合せで用いることになる. また, 誤り訂正が実現できた場合の FISQ では, 因数分解の Shor のアルゴリズムや, 検索の Grover のアリゴリズムなどの本格的な使用が可能であるが, 現時点は実用レベルで FISQ のマシンは完成していない [2]. 例としては, IBM の Qiskit [3] を用いて, IBM Quantum を利用可能であり, 無料枠では 5 量子ビットまでの計算ができる.

一方, 量子アニーリング方式には, 超伝導磁束量子ビットを用いた D-Wave のマシンだけでなく, 量子アニーリング方式にインスパイアされた, CMOS 技術をベースとしたデジタルアニーラを含め, デジタルコンピュータベースの実用的ハードウェアがいくつかある [4]. D-Wave をはじめとする量子アニーリング方式のマシンは, イジングモデル上のスピングラス問題（NP 困難）を高速で解くことがで

きる. イジングモデルとは, スピンの状態 s_i が 1 もしくは -1 をとるとき, バイアス h_i と相互作用 J_{ij} を適切に設定し, $H = -\sum_{i=1}^{N} h_i s_i - \sum_{i<j}^{N} J_{ij} s_i s_j$ となるハミルトニアン H の最小化を計算するものである. つまり, イジングモデル上の相互作用 J_{ii} とバイアス h_i を問題にあわせて適切に設定することで, 任意の組合せ最適化問題を解け, これを量子アニーリングマシンで高速化することを目指している.

量子アニーリング方式では, 実問題に対応した相互作用とバイアスの設定が難しく, 解きたい組合せ最適化問題を制約付き最適化問題に変換し, それを QUBO (Quadratic Unconstrained Binary Optimization: 二次制約なし二値最適化問題) にすることが必要だが, 昨今は OpenJij [5] や Fixstars Amplify [6] (以下, Amplify と記す) などのソフトウェアの利用で簡素化がなされれている. なお, イジングマシンは QUBO を解くもので, 制約条件は付けられないが, 目的関数に制約条件をペナルティ項として付加することで制約条件を実現し, Amplify ではソフトウェア内で求解後の解が, 制約条件を満たしているか否かのチェックも自動的に行えるようになっている.

本稿では, Amplify を用いた量子アニーリング方式の有用性を確認するために, 2 章で Amplify の概要について述べ, 3 章で巡回セールスマン問題に対する Amplify と古典的ソルバーの比較を行い, 4 章でまとめを述べる.

2 Amplify プログラミング

2.1 概要

Amplify [6] は, イジングマシンのためのクラウドシステムで, Python から呼び出せる API で構成されるプログラミングプラットフォームである. Amplify では, Fixstars Amplify Annealing Engine や D-Wave だけでなく, 東芝の Simulated Bifurcation Machine [7], 富士通のデジタルアニーラ [8] や日立の CMOS アニーリングマシン [9] などにも対応している.

Amplify は二次制約なし二値最適化問題, すなわち QUBO の問題解決のためのプログラミングを, 次の手順で解決していく. なお, 次章で巡回セールスマン問題を解くアプリケーションの性能評価を示すが, 以下の説明においても, そのアプリケーションに沿った例で示す.

2.2 決定変数

イジングモデルでは目的関数の値を最小にするスピンの状態を決定するものであるが, そのスピンの状態を決定変数として指定する. 決定変数には, 1 と -1 の値をとる `IsingPoly` と, 1 と 0 の値をとる `BinaryPoly` の 2 種類のタイプがあり, 解きたい問題に適したタイプを選んで, `SymbolGenerator` 関数で生成する. 図 1 では, `ncity×ncity` 個の `BinaryPoly` 型の決定変数を生成するプログラムを表わしている.

決定変数 `q[i][j]` は, `i` 番の巡回で `j` 番のノードを探索するか否かを示すもので, `k` と `l` のノード間のコストを `distance[k][l]` 関数で示すと, 最小化したい目的関数は図 2 で示されるプログラムで記述できる. この中で `sum_poly` 関数は, Python の sum 関数と同様で合計を示すものであるが,

Amplify の実行に適した最適化がされている, q[i][j] が 1 の時は, i 番目の巡回でノード j をたどるので, 1 の所同士のノード間の distance を合計したものが総コストになり, cost 関数を最小化することが目的となる.

```
gen = gen_symbols(BinaryPoly)
q = gen.array(ncity, ncity)
```

図 1: 決定変数

2.3 目的関数と制約条件

```
cost=sum_poly([distances[i][j] * q[n][i] * q[(n + 1) % ncity][j]
             for i in range(ncity)
             for n in range(ncity)
             for j in range(ncity)])
```

図 2: 目的関数

巡回セールスマン問題では, ひとつの巡回で訪問できるノードは 1 個 (制約 1) だけであり, また, すべてのノードは 1 回だけ訪問する (制約 2) という制約を満たすことが必要になるが, QUBO では制約条件を直接扱うことができない. そこで, equal_to 関数, greater_equal 関数, less_equal 関数, one_hot 関数などで制約を表し, 目的関数と一緒に最小化の対象として制約条件を満たすようにプログラムすることになる. equal_to 関数は決定変数もしくはその和が一定の値になるような条件を示し, greater_equal 関数や less_equal 関数は決定変数もしくはその和が一定の値よりも大きくもしくは小さくなるような条件を示す. また, one_hot 関数は equal_to 関数の特別なケースで, 1 になる場合の条件を示しており, 組合せ最適化では頻繁に用いられる制約条件である. なお, Amplify では, 上記の関数で表された制約条件は, 最小化の解に対して満たされているかのチェックを自動的に行なうので, プログラムで陽に記述する必要はない.

```
travel_c = [
    equal_to(sum_poly([q[n][i] for i in range(ncity)]), 1)
    for n in range(ncity)
]
node_c = [
    equal_to(sum_poly([q[n][i] for n in range(ncity)]), 1)
    for i in range(ncity)
]
```

図 3: 制約条件

図 3 の制約条件の例では, travel_c が制約 1 を表し, node_c が制約 2 を表している. つまり, travel_c では, q[n][i] の i に対する合計が 1 となる, すなわち n 番目の巡回で訪れるノードは 1 個だけになることを, すべての n で成り立つ条件を示している. また, node_c では, q[n][i] の n に対する合計が 1 となる, すなわち i 番目のノードは 1 回だけ訪れることを, すべての i で成り立つ条件を示している.

2.4 イジングマシンの実行

図 4 にイジングマシンの実行開始のプログラムを示す. model は目的関数と制約条件を表す関数の和であり, 制約条件を満たす強さを重み係数 w を乗じて加算している. つまり制約条件の部分は, 目的関数の補正項 (ペナルティ関数) となり, 重み係数 w は経験的に決めることになる. Fixstars Amplify Annealing Engine に接続するための token を 3 行目で設定しており, 実行時間の上限をミリ秒単位で 4 行目で設定している. 2022 年 5 月現在, 有料版では 1 分もしくは無制限の実行が可能であるが, 無料枠では 10 秒までの実行が可能となっている[1]. 5 行目で Solver のインスタンスが生成され, 6 行目の solve メソッドで量子アニーリングマシンにジョブが投入され, 解が求まるまで実行が続けられる.

```
1 model = cost + w*(sum(trave_c)+sum(node_c))
2 client = FixstarsClient()
3 client.token = "XXXXX"
4 client.parameters.timeout = 5000
5 solver = Solver(client)
6 result = solver.solve(model)
```

図 4: イジングマシンの実行

実行結果の表示には, 図 5 に示すように, 結果を決定変数と同じ形式で出力するための関数として decode 関数が準備されている. つまり, 変数 q_values[n][i] には, n 番目の巡回で i 番目のノードを訪問するか否かが 0 と 1 で示されるようになり, nonzero メソッドで非ゼロのインデックスを表示すれば, コストが最小化された経路が表示されることになる.

```
1 print('Size of results:', len(result))
2 q_values = q.decode(result[0].values)
3 answer = q_values.nonzero()[1]
4 print(answer)
```

図 5: 結果の表示

[1]現在, 無料枠でも 10 秒を超える時間設定が可能であり, この部分は今後どのようになるかは不明である.

3 性能比較

3.1 性能評価

前章で述べたように, 本章では, 巡回セールスマン問題を対象とし, Amplify による量子アニーリングの実行性能と古典的ソルバーでの実行性能を比較する. 古典的ソルバーには, 非商用として利用できるものとして, SCIP (Solving Constraint Integer Programs) [10] や GLPK (GNU Linear Programming Kit) [11] などがある. GLPK は, C 言語で記述され, 整数問題や混合整数問題を解くためのソフトウェアパッケージであり, 問題設定などは, GNU MathProg モデリング言語で記述する. 性能面では SCIP が優れているが, 本稿では GNU ライセンスで利用可能な GLPK を用いる.

GLPK での巡回セールスマン問題の MathProg プログラムの主要部分を図 6 に表わす. 図に示すように, 最小化したい関数を `minimize` キーワードで定義し, ノード `i` とノード `j` のリンク `link[i,j]` があれば, それにコスト `cost[i,j]` を乗じたものの総和が最小になるように, リンクを求めることになる. ここで制約条件を `s.t.` キーワードで定義し, それぞれのノードが, リンク元とリンク先のノードとなるのは 1 度だけになるような条件を与えている.

```
1 minimize total: sum{(i,j) in Edge} cost[i,j] * link[i,j];
2 s.t. leave{i in V}: sum{(i,j) in Edge} link[i,j] = 1;
3 s.t. enter{j in V}: sum{(i,j) in Edge} link[i,j] = 1;
```

図 6: MathProg の例

Amplify と GLPK の実行環境を表 1 に示す. Fixstars Amplify Annealing Engine は GPU をベースとしたデジタルマシンであるが, 量子アニーリングの確率的な振る舞いを模倣したシステムである.

表 1: 実行環境

Fixstars Amplify Annealing Engine	
形式	GPU
最大ビット数	100,000 以上
結合グラフ	全結合
GLPK	
CPU	Intel (R) Xeon (R) W-2123 (3,6 GHz)
メモリ	32 GB
OS	Ubuntu 20.04

これらの実行環境において, 訪問するノード数を, 32, 48, 64, 80, 96 にした場合の実行時間を図 7 に示す. Amplify では, 自明な問題, すなわち, 隣接間のノードのコストは 1 とし, それ以外のコストを 60 として, 自明な解に至るまでの実行時間を, timeout を変更しながら実行時間を計測することにする. ま

た, 実行時間は, `result = solver.solve(model)` を `time.time()` で囲んで, その差で計測する. また, GLPK では, ソルバー (glpsol) が実行結果とともに出力する実行時間を採用する.

ノード数が小さい場合は古典的ソルバーの方が実行時間が短いが, ノード数が大きくなるにつれて逆転し, 96 ノードでは Amplify の方が速くなったいる. 用いる実行環境によって実行時間の絶対値は変化するが, いずれの実行環境でも, ノード数の増加にしたがって実行時間の増加はみられる. GLPK においては, ノード数が 48 から 96 に 2 倍になった場合, 約 47.5 倍の増加になっている.

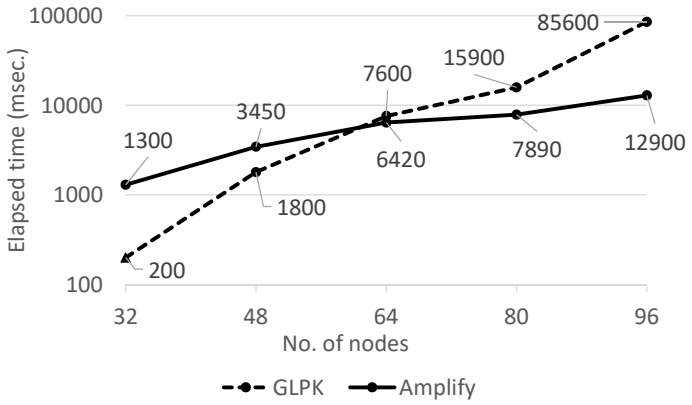

図 7: 実行時間比較 (ミリ秒)

3.2 考察

単純な総当たりによる求解であれば, ノード数の階乗に比例するので. 優れた解法を用いたとしても, ノード数に比例する以上の実行時間の増加は予想される. 一方, Amplify による量子アニーリングであれば, ノード数が 48 から 96 に増えても, 実行時間は 2 倍強の増加にとどまり, 問題サイズが大きくなるに連れて古典的なソルバーよりも優位になっており, 組合せ最適化問題に関しては将来的に有望なハードウェアであると言える. なお, ほぼ同じプログラムで, Fixstars Amplify Annealing Engine だけでなく, D-Wave 2000Q で実行できることになっているが, 本稿作成時の実行環境では, 問題サイズが適合しない, もしくは実行時間内に制約条件を満たす最適解が得られないなどの問題が生じ, 実行時間の測定には至らなかった.

量子アニーリング及び Amplify のように確率的な振る舞いをデジタルマシンで模倣したハードウェアにおいては解に至る動作が実行毎に異なる. つまり, 求まる解が実行毎に異なったり, それに至るまでの実行時間にバラツキがある. 上記の実行時間測定においても, 数回の実行を行い, 最適解に至った実行時間の中間値を実験結果としている. このような振る舞いは, ベンチマーキングや試用レベルだけでなく, 実用レベルの利用においても同様である. 量子アニーリング方式は, 総じて高速に解を求めることはできるものの, 一定時間に確実に解が求まることは保証されない. したがって, 確実性や安定性を求めるアプリケーションには必ずしも向いていない場合もあることが予想され, 将来的には有望な手法ではあるものの, 量子的なアプローチか古典的なアプローチかは, 利用する場面に合わせた選択が必要であると考えられる.

4 おわりに

次世代の計算方式として量子計算は注目され, その中でも量子アニーリング方式は組合せ最適化問題を高速に解くことができた. 本方式は実用化されているハードウェアもいつくかあり, 実用的なアプリケーションに応用されつつある. 本稿では, Fixstars が提供している Amplify を用いて巡回セールスマン問題を実装し, 古典的ソルバーの GLPK と性能比較を行った. 性能比較においては, 訪問するノード数の増加に対し, 古典的ソルバーの実行時間は膨大に増加するのに対し, 量子アニーリング方式においてはノード数に比例する程度の実行時間の増加に抑えられ, 実行時間の短縮効果が確認された. すなわち, ノード数が大きい場合は, 量子アニーリング方式が優位になる. また, Amplify を用いることでプログラミングは簡素化され, イジングモデルにおける相互作用やバイアスなどを意識せずにプログラミングができるので, 組合せ最適化問題に容易に適用できる.

その一方で, 量子アニーリング方式は確率的振る舞いによって最適解を求めているので, 最適解に至らない場合や, 最適解に至る時間のバラつきなどの不確実で不安定な動作も散見された. また, D-Wave マシンの利用も可能であるとのことであったが, Fixstars Amplify Annealing Engine で実行できたのと同じ問題サイズでは実行が行えなかったので, さらにより詳しいプログラミング及びその性能評価が必要であることも分かった.

今後は, 古典的アプローチと量子的アプローチの使い分けの基準を検討することが必要である. また, より実用的な問題に対し Amplify による量子アニーリング方式が有効であるかを引き続き検討していくことが重要である.

謝辞

本研究の一部は JSPS 科学研究費 (基盤研究 (C) 18K02920 (2018-2022), 基盤研究 (C) 19K03018 (2019-2022)) 及び私立大学等経常費補助金特別補助「大学間連携等による共同研究」による.

参考文献

[1] “量子ソフトウェア研究会,” https://sigqs.ipsj.or.jp/ (As of 2022/5/1).

[2] S. S. Gill, A. Kumar, H. Singh, M. Singh, K. Kaur, M. Usman and R. Buyya, “Quantum computing: A taxonomy, systematic review and future directions,” https://arxiv.org/ftp/arxiv/papers/2010/2010.15559.pdf (As of 2022/5/1).

[3] “Open-source quantum development,” https://qiskit.org/ (As of 2022/5/1).

[4] “量子コンピューティング研究調査,” https://amplify.fixstars.com/ja/techresources/research/ (As of 2022/5/1).

[5] “OpenJij,” https://openjij.org/ (As of 2022/5/1).

[6] “Amplify,” https://amplify.fixstars.com/ja/ (As of 2022/5/1).

[7] “量子インスパイアード最適化ソリューション SQMB+,” https://www.global.toshiba/jp/products-solutions/ai-iot/sbm.html (As of 2022/5/1).

[8] “Digital annealer,” https://www.fujitsu.com/jp/digitalannealer/ (As of 2022/5/1).

[9] “Annealing cloud web,” https://annealing-cloud.com/ja/index.html (As of 2022/5/1).

[10] “SCIP (Solving Constraint Integer Programs),” https://scipopt.org/ (As of 2022/5/1).

[11] “GLPK (GNU Linear Programming Kit),” https://www.gnu.org/software/glpk/ (As of 2022/5/1).