Reproducing *Fingerhuth, et. al. 2018*: A Jupyter Notebook Tutorial

Scott Longwell
Department of Bioengineering, Stanford University[*]
(Dated: June 9, 2019)

I. PROBLEM BACKGROUND: PROTEIN FOLDING AND BINDING

Understanding the principles that govern how proteins fold and bind one another - effectively intra- and intermolecular versions of the same process - would usher in a new era in drug design and personalized medicine [1]. It is also notoriously difficult to predict the trajectory of these processes from the sequence of amino acids that comprise proteins. In the case of protein folding, the Levinthal paradox notes that for a protein of length n amino acids, there are $n-1$ phi-angles and $n-1$ psi-angles within the protein backbone, each of which has ~ 3 possible values [2, 3]. In all, this leads to an exponential number of conformations which a folding protein can sample: $3^{2(n-1)}$, ignoring a large number of sterically impossible conformations. Despite this massive number, proteins fold to low-energy minimia on incredibly short timescales (10^3 down to 10^6 seconds), suggesting that there are heuristics and principles to efficiently traverse this exponentially large conformation space.

It is often stated that the massively parallel nature of quantum computing will be the advance that renders protein folding tractable to predict [4]. There are several possible embeddings of the protein folding process [5]. A simplistic but commonly used approach is to map the amino acids of a protein sequence onto a cubic or planar lattice by placing the first two amino acids at adjacent positions and placing the remaining amino acids sequentially, each time making a move in one of the 6 (or 4) possible absolute directions on the cubic (or planar) lattice. A cost function is then established that a) penalizes chain overlaps and b) accounts for interactions between adjacent amino acids. This embedding has lent itself well to quantum annealers, with Perdomo and colleagues demonstrating folding of short peptides on D-Wave systems [6, 7]. More recently, a team from ProteinQure demonstrated similar capabilities on a D-Wave 2000Q system [8], as well as the first implementation on a gate-based universal quantum computer (19-Q Acorn at Rigetti) [9].

II. PROBLEM APPROACH

In this project, I attempted to reproduce the work of Fingerhuth, Babej, and Ing (hereafter referred to as 'the authors') in their recent paper [9]. I selected this work because it a) was an interesting application of a quantum alternating operator ansatz (QAOA), and b) was uniquely composed to run on a gate-based quantum computer (*vs.* a quantum annealer). However, I found it difficult to recover the authors' QAOA implementation from the paper alone, and had to extensively refer to their other work [8] as well as works they cited [5–7]. Below, I attempt to succinctly describe the authors' approach to identify the ground state fold of a 4 residue protein (PSVK) using a QAOA simulated with Forest [10]. Much of the ansatz procedure generalizes to folding proteins of length N on lattices of dimension D. Thorough Jupyter notebook walk-throughs are available at `https://github.com/slongwell/qfold`.

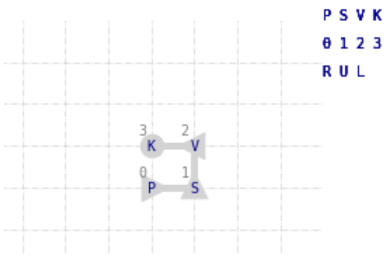

FIG. 1. Folding of the 4 amino acid protein *PSVK* on a 2D-lattice (own work). Only P and K interact.

A. Lattice-based turn-encoding

To render the protein folding tractable, it is common to model the problem as a self-avoiding walk (SAW) on a planar lattice (Fig. 1). Residues in the protein are sequentially placed at lattice vertices, with each sequential placement on the lattice corresponding to unitary movement along an edge (*i.e.* `right`, `up`, `left`, `down`). Thus, a protein fold is defined by a sequence of turns.

The authors use the following one-hot encoding for the 4 possible turn directions on a 2D-lattice, where k is a pointer to a corresponding bit flip:

$$\texttt{right} \mapsto k = 0 \ (1000) \tag{1}$$
$$\texttt{up} \mapsto k = 1 \ (0100) \tag{2}$$
$$\texttt{left} \mapsto k = 2 \ (0010) \tag{3}$$
$$\texttt{down} \mapsto k = 3 \ (0001) \tag{4}$$

Each turn is therefore represented as 4 bits, and a sequence of turns forms a *turn bitstring* of $4(N-1)$ bits.

[*] longwell@stanford.edu

It is useful to fix turn $t = 0$ to `right` and limit $t = 1$ to {`right, up`}, as this reduces the occurrence of redundant folds with equivalent cost (*e.g.* those with rotational or planar symmetry). Consequently, the *turn bitstring* can be represented with 0 qubits for $t = 0$, 2 qubits for $t = 1$, and 4 qubits for $t \geq 2$, such that the total number of required qubits is $4N - 10$ (for proteins of length $N \geq 3$). The boolean for whether the protein turned direction k on turn t is therefore read as:

$$d_k^t = \begin{cases} q_k & t = 1, k \in \{0, 1\} \\ 0 & t = 1, k \notin \{0, 1\} \\ q_{4t+k-6} & else \end{cases} \tag{5}$$

B. Cost Hamiltonian

The cost ascribed to a fold (*turn bitstring*) considers two principles: a) that residues cannot overlap in space, and b) that adjacent residues will interact with an energetic reward/penalty specific to their identities. These two principles combine to form the total cost, such that:

$$H_{cost} = H_{overlap} + H_{pair} \tag{6}$$

Determining $H_{overlap}$ and H_{pair} first requires evaluating *turn bitstrings* to determine if pairs of residues indexed by i and i' are, respectively, a) overlapping or b) adjacent. To detect both of these cases, it is useful to convert the single *turn bitstring* between i and i' into 4 *sum strings* (one for each direction). Each *sum string* is a tally of the turns taken in a particular direction k between i and i', represented as a binary number:

$$s_k^r(i, i') = r^{th}\text{digit of} \sum_{p=i}^{i'-1} d_k^p \tag{7}$$

The *sum strings* can be calculated from the *turn bitstring* with a circuit of half-adders, with each half-adder acting on two bits to yield their `AND` and `XOR`:

$$\texttt{AND}(x, y) = xy \tag{8}$$
$$\texttt{XOR}(x, y) = x + y - 2xy \tag{9}$$
$$\texttt{HA}(x, y) = (\texttt{AND}(x, y),\ \ \texttt{XOR}(x, y)) \tag{10}$$

The full circuit of half-adders used to generate the *sum strings* is shown in Fig. 2. The authors note that this circuit may be reduced in complexity from quadratic to quasilinear by omitting half-adders that do not contribute to the sum bits (since the sum of m bits will require at most $\lceil log_2(m + 1) \rceil$ bits).

To determine whether two residues indexed by i and i' overlap, the *sum strings* representing the turns between them are paired off according to opposite directions (*i.e.* `left-right`, `up-down`) and compared. If (and only if) all *sum string* pairs are equivalent, it follows that the two residues occupy the same position on the lattice. This is implemented by comparing the pairs bit-by-bit with an equality gate, commonly known as `XNOR`. The product of over the `XNOR` of all directional *sum string* pairs and all valid sum bits indicates whether the two residues overlap:

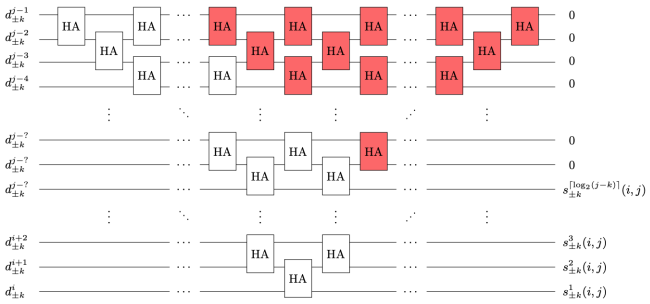

FIG. 2. Circuit of half-adders used to compute *sum strings*. Note that several of the half-adders can be removed without affecting the sum (adapted from [8]).

$$\texttt{XNOR}(x, y) = 1 - x - y + 2xy \tag{11}$$
$$h_{overlap}(i, i') = \prod_{k=1}^{D} \prod_{r=1}^{\lceil log_2(i'-i) \rceil} \texttt{XNOR}(s_k^r(i, i'), s_{k_o}^r(i, i')) \tag{12}$$

To arrive at the full overlap cost, this expression for two residues is summed over all possible overlapping residues and multiplied by a user-defined factor $\lambda_{overlap} \sim 10$ (see Babbush, *et. al.* [5] for a description of how to set):

$$H_{overlap} = \lambda_{overlap} \sum_{i=0}^{N-3} \sum_{i'=1}^{\lfloor \frac{1}{2}(N-i-1) \rfloor} h_{overlap}(i, i + 2i') \tag{13}$$

It is more involved to determine whether two residues indexed by i and i' are adjacent based on the *sum strings* between them. The corresponding adjacency function $a_k(i, i')$ comprises several products and sums of `XOR` and `XNOR` terms (Fig. 3). A thorough description is found in Babbush, *et. al.* [5].

Similar to $H_{overlap}$, this adjacency function is then summed over all possible adjacent residues, but this time multiplied by a contact energy $P_{i,i'}$ specific to the identities of the two adjacent residues (*e.g.* as defined by the Miyazawa-Jernigan model [11–13]):

$$H_{pair} = \sum_{i=1}^{N-3} \sum_{i'=1}^{\lfloor \frac{1}{2}(N-i-1) \rfloor} P_{i,1+i+2i'} \sum_{k=1}^{D} a_k(i, 1 + i + 2i') \tag{14}$$

$$
\begin{aligned}
a_k(i,j) = &\Bigg[\prod_{w \neq k} \Bigg(\prod_{r=1}^{\lceil \log_2(j-i) \rceil} \mathrm{XNOR}(s_w^r(i,j), s_{\bar{w}}^r(i,j)) \Bigg) \Bigg] \quad \texttt{t0} \\
&* \Bigg[\mathrm{XOR}(s_k^1(i,j), s_{\bar{k}}^1(i,j)) \quad \texttt{t1} \quad \texttt{t10} \\
&* \prod_{r=2}^{\lceil \log_2(j-i) \rceil} \mathrm{XNOR}(s_k^r(i,j), s_{\bar{k}}^r(i,j)) \quad \texttt{t11} \\
&+ \sum_{p=2}^{\lceil \log_2(j-i) \rceil} \Big(\mathrm{XOR}(s_k^{p-1}(i,j), s_{\bar{k}}^p(i,j)) \quad \texttt{t12} \quad \texttt{t120} \\
&* \prod_{r=1}^{p-2} \mathrm{XNOR}(s_k^r(i,j), s_k^{r+1}(i,j)) \quad \texttt{t121} \\
&* \prod_{r=1}^{p} \mathrm{XOR}(s_{\bar{k}}^r(i,j), s_{\bar{k}}^r(i,j)) \quad \texttt{t122} \\
&* \prod_{r=p+1}^{\lceil \log_2(j-i) \rceil} \mathrm{XNOR}(s_{\bar{k}}^r(i,j), s_{\bar{k}}^r(i,j)) \Big) \Bigg] . \quad \texttt{t123}
\end{aligned} \tag{26}
$$

FIG. 3. Annotated adjacency function for two residues indexed by i and j (adapted from [9]).

C. Mixer (Driver) Hamiltonians

In quantum annealers, the mixer is typically fixed to a simple Hamiltonian such as:

$$X_{mixer} = \sum_i X_i \tag{15}$$

One of the advantages of gate-based quantum computers over quantum annealers is that they allow a user to flexibly define the mixer Hamiltonian to encode 'hard', inviolable constraints (*i.e.* residue overlaps), allowing the cost Hamiltonian to be relaxed to only consider 'soft' constraints (*i.e.* residue interaction energy). The authors explore 7 mixer Hamiltonians, many of which prevent overlapping turns so long as the initial quantum state does not encode an overlap. One such mixer that I attempted to implement is given by:

$$\texttt{XY}_{i,i'} = \frac{1}{2}(X_i X_{i'} + Y_i Y_{i'}) \tag{16}$$

$$M_t = \sum_{i=0}^{n-2} \sum_{i'=i+1}^{n-1} \texttt{XY}_{i,i'} \tag{17}$$

$$XY_{simple} = \sum_{t=1}^{N-2} M_t \tag{18}$$

Here, $\texttt{XY}_{i,i'}$ has similar properties to a `SWAP` gate, but it does not commute with the classical, Z-based cost Hamiltonian (X_i, Y_i, and Z_i are all Pauli matrices). Hadfield, *et. al.* refer to this gate as $\texttt{XY}_{i,i'}$ [14], while the authors redefine the `SWAP` gate to this definition.

D. QAOA Simulation

The authors considered a range of ansatz architectures, spanning 7 mixer Hamiltonians, 2 circuit depths ($p = 1$, $p = 2$), and 2 initial quantum states (superposition over *all* bitstrings or *feasible* bitstrings). They performed Nelder-Mead optimization of these ansatzes with a tolerance of 0.001 and 50 to 100 runs on a fully-connected, noiseless simulated quantum computer using Forest.

III. PROGRESS AND FUTURE WORK

All work to date is accessible at `https://github.com/slongwell/qfold`, with the majority of the tutorial in the Jupyter notebook `final.ipynb`. The notebook first demonstrates the 2D-lattice encoding scheme, using a visualization tool to render an example folded protein (as in Fig. 1). Along with the visualization function, the direction dictionaries to implement the authors' turn-based 2D-lattice scheme are included as an importable module in `lattice.py`. The notebook then loads and shows the Miyazawa-Jernigan matrix. As this matrix was not provided from the authors or easily accessible from the original publication, I opted to obtain it with some adaptation from LightDock. The full matrix is subsetted to create the PSVK-specific $P_{i,i'}$ for use in QAOA.

The vast majority of effort went into faithfully retrieving and implementing the cost Hamiltonians, as well as 2 of the mixer Hamiltonians. I first transcribed these Hamiltonians and related functions as MathJax in the notebook `hamiltonians.ipynb` before implementing them as Python functions. As mentioned, the adjacency function $a_k(i, i')$ was particularly difficult to parse, so it was broken into several terms (Fig. 3). To discern how *sub strings* worked, the work by Babbush, *et. al.* was particularly helpful [5]. Within the text, I searched for, but found no explicit approach to express the equations as Pauli sums. While the approach was fairly clear for the mixer Hamiltonians, it was less so for the cost Hamiltonians, and remaining errors are likely due to my failure to correctly convert the equations to Pauli sums.

With the Hamiltonians implemented (and global varibles added, such as protein length N, number of dimensions D, and number of qubits n), I converted the notebook into the module `hamiltonians.py` so that they could be imported into `final.ipynb`. Adapting the implementation from QAOA example notebook from lecture 13, I then attempted to run QAOA using PyQuil's `exponentiate_commuting_pauli_sum()` function and using the Nedler-Mead optimization employed by the authors. While the optimization terminated successfully, it gave an unreasonably large function value (again, mostly likely due to an error in the way I implemented the Pauli

sums). Finally, I enumerated all the possible folds of the PSVK protein for (anticipated) comparison with the results from QAOA.

At the end of the notebook, I outline several possible pedagogical ideas (*e.g.* cubic lattice implementation, improved visualizations, running on a quantum computer) as well as possible improvements to the approach. My hope is that this notebook tutorial will make QAOA approaches to folding proteins much more accessible than I found them in literature (and perhaps be useful as a course problem set with an interesting application).

---

[1] K. A. Dilland J. L. MacCallum, The Protein-Folding Problem, 50 Years On, Science **338**, 1042 (2012).

[2] C. Levinthal, How to fold graciously, Mossbauer spectroscopy in biological systems **67**, 22 (1969).

[3] A. V. Melkikhand D. K. F. Meijer, On a generalized Levinthal's paradox: The role of long- and short range interactions in complex bio-molecular reactions, including protein and DNA folding, Progress in Biophysics and Molecular Biology **132**, 57 (2018).

[4] Y. Cao, J. Romero, and A. Aspuru-Guzik, Potential of quantum computing for drug discovery, IBM Journal of Research and Development **62**, 6:1 (2018), good overview.

[5] R. Babbush, A. Perdomo-Ortiz, B. O'Gorman, W. Macready, and A. Aspuru-Guzik, Construction of Energy Functions for Lattice Heteropolymer Models: A Case Study in Constraint Satisfaction Programming and Adiabatic Quantum Optimization, arXiv e-prints (2014), arXiv:1211.3422.

[6] A. Perdomo, C. Truncik, I. Tubert-Brohman, G. Rose, and A. Aspuru-Guzik, Construction of model Hamiltonians for adiabatic quantum computation and its application to finding low-energy conformations of lattice protein models, Physical Review A **78**, 012320 (2008).

[7] A. Perdomo-Ortiz, N. Dickson, M. Drew-Brook, G. Rose, and A. Aspuru-Guzik, Finding low-energy conformations of lattice protein models by quantum annealing, Scientific Reports **2**, 571 (2012).

[8] T. Babej, C. Ing, and M. Fingerhuth, Coarse-grained lattice protein folding on a quantum annealer, arXiv e-prints (2018), arXiv:1811.00713.

[9] M. Fingerhuth, T. Babej, and C. Ing, A quantum alternating operator ansatz with hard and soft constraints for lattice protein folding, arXiv e-prints (2018), arXiv:1810.13411.

[10] R. S. Smith, M. J. Curtis, and W. J. Zeng, A Practical Quantum Instruction Set Architecture, arXiv e-prints (2016), arXiv:1608.03355.

[11] S. Miyazawaand R. L. Jernigan, Estimation of effective interresidue contact energies from protein crystal structures: quasi-chemical approximation, Macromolecules **18**, 534 (1985).

[12] S. Miyazawaand R. L. Jernigan, Residue–residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading, Journal of molecular biology **256**, 623 (1996).

[13] S. Miyazawaand R. L. Jernigan, Self-consistent estimation of inter-residue protein contact energies based on an equilibrium mixture approximation of residues, Proteins: Structure, Function, and Bioinformatics **34**, 49 (1999).

[14] S. Hadfield, Z. Wang, B. O'Gorman, E. G. Rieffel, D. Venturelli, and R. Biswas, From the quantum approximate optimization algorithm to a quantum alternating operator ansatz, Algorithms **12**, 34 (2019).