CDMTCS
Research
Report
Series

Finding Maximum-sized Native Clique Embeddings: Implementing and Extending the Block Clique Embedding Algorithm

Puya Yao
Richard Hua
Department of Computer Science
University of Auckland
Auckland, New Zealand

CDMTCS-523
March 2018

Centre for Discrete Mathematics and
Theoretical Computer Science

Finding Maximum-sized Native Clique Embeddings: Implementing and Extending the Block Clique Embedding Algorithm

Puya (Amanda) Yao and Richard Hua
Department of Computer Science
University of Auckland, Auckland, New Zealand
{pyao017,rwan074}@aucklanduni.ac.nz

1 Introduction

Minor-embedding is one of the fundamental concepts in adiabatic quantum computing when the hardware structure does not support arbitrary qubit interactions. In particular, when minimizing the energy of an Ising spin configuration, the corresponding graph must be minor-embedded into a Chimera graph [1].

A minor embedding of a graph $G_1 = (V_1, E_1)$ onto a graph $G_2 = (V_2, E_2)$ is a function $f : V_1 \to 2^{V_2}$ that satisfies the following three conditions:

1. The sets of vertices $\{f(v) | v \in V_1\}$ are disjoint.
2. For all $v \in V_1$, there is a subset of edges $E' \in E_2$ such that $G' = (f(v), E')$ is connected.
3. If $\{u, v\} \in E_1$, then there exist $u', v' \in V_2$ such that $u' \in f(u)$, $v' \in f(v)$ and $\{u', v'\}$ is an edge in E_2.

Within the scope of a minor embedding, G_1 is referred to as the guest graph while G_2 is called the host graph [2].

This report follows closely to the paper, *Fast clique minor generation in Chimera qubit connectivity graphs* [1], and includes the implementation of the algorithm for finding one of the largest clique minors of any given Chimera graph. The result on a D-Wave 2X machine is included as well.

Below are some of the key definitions from [1].

A block clique embedding is a set χ of n ell blocks $\{(X_1, c_1), ..., (X_n, c_n)\}$ such that each X_i contains n unit cells (so ells have length $n+1$), and every distinct pair X_i, X_j in χ intersects at exactly one unit cell, which is in the horizontal component of one ell block and the vertical component of the other.

A native clique embedding respecting a block clique embedding χ is a collection β of ell bundles $\{B_1, ..., B_n\}$ such that for each i and for each $l \in B_i$, $(X_i, c_i) = (X(l), c(l))$, i.e. (X_i, c_i) is the ell block for each ell in B_i.

Given a set of ell bundles $\beta = \{B_1, B_2, \cdots, B_n\}$ where each B_i is contained in the ell block (X_i, c_i) and $\chi = \{(X_1, c_1), (X_2, c_2), \cdots, (X_n, c_n)\}$, we define $\parallel \chi \parallel = |\bigcup_{i=1}^{n} maxBundle(X_i, c_i)| \geq |\bigcup_{i=1}^{n} B_i|$.

A Chimera $C_{M,N,L}$ consists of $M \times N$ interconnected complete bipartite graph $K_{L,L}$. For a given integer n, every native clique embedding respecting the corresponding block clique embedding of size n has $L * n$ vertices, since every ell block contains L ells. However, in practice, not all physical qubits of the hardware structure are functional all the time so an algorithm is needed to find the block clique embedding of size n that contains the maximum-sized native clique embedding.

2 The NativeCliqueEmbed Algorithm and its Proof

The NativeCliqueEmbed algorithm uses dynamic programming to find the partial block clique embedding that contains the maximum-sized partial native clique embedding for each working rectangle R of height i, where i increases from 1 to $n-1$, hence finding the maximum-sized native clique embedding in polynomial time [1].

The correctness of the algorithm is proven using the following lemma and theorem (see [1] for a complete proof).

Lemma 1 *Let $\chi = \{(X_1, c_1), ..., (X_n, c_n)\}$ be a block clique embedding in $C_{n,n,L}$. Then the ell blocks of X have distinct heights.*

Theorem 1 *In a $C_{n,n,L}$ Chimera graph for $n \geq 2$, there are 4^{n-1} block clique embeddings that contain n ell blocks. In particular, they are in natural bijection with the set $\{E, W\} \times \{NE, NW, SE, SW\}^{n-2} \times \{N, S\}$.*

The first working rectangle R_1 is placed after the placement of the first ell block X_1 and the i-th ell block X_i placed has height i and width $n-i+1$. The first ell block X_1 has width $n-1$ and height 1. The first working rectangle R_1 placed intersects all the unit cells except for the corner cell. Every time another ell block is placed, the corresponding working rectangle gets one unit taller and one unit narrower, and it never covers the corner cell.

Let $R_{from}(X,c)$ denotes the working rectangle that is placed immediately after the placement of the ell block (X,c). Let $R_{to}(X,c)$ denotes the working rectangle that is placed right before the placement of (X,c). From the proof of theorem one, each ell block (X,c) either has one unique $R_{from}(X,c)$ or one unique $R_{to}(X,c)$ or both.

Also, for each rectangle R, the sets $X_{from}(R) := \{(X,c)|R = R_{to}(X,c)\}$ and $X_{to}(R) := \{(X,c)|R = R_{from}(X,c)\}$ both have size at most four.

The algorithm is presented below:

Algorithm 1 The algorithm to find a maximum-sized native clique embedding in an induced subgraph of a Chimera graph.

function $NativeCliqueEmbed(G,n)$
    for $i = 1,...,n-1$ **do**
        for each rectangle R of height i and width $n-i$ **do**
            $maxPartialEmbedding(R) \leftarrow \phi$
        for each ell block (X,c) of height i and width $n-i+1$ **do**
            $\beta \leftarrow maxPartialEmbedding(R_{to}(X,c)) \bigcup \{(X,c)\}$
            if $\| maxPartialEmbedding(R_{from}(X,c)) \| < \| \beta \|$ **then**
                $maxPartialEmbedding(R_{from}(X,c)) \leftarrow \beta$
    $\beta_{max} \leftarrow \phi$
    for each ell block (X,c) of height n and width 1 **do**
        $\beta \leftarrow maxPartialEmbedding(R_{to}(X,c)) \bigcup \{(X,c)\}$
        if $\| \beta_{max} \| < \| \beta \|$ **then**
            $\beta_{max} \leftarrow \beta$
    return $\{maxBundle(X,c,G)|(X,c) \in \beta_{max}\}$

$maxPartialEmbedding(R)$ denotes the maximum partial block clique embeddings, $\chi_i = \{(X_1,c_1),...,(X_i,c_i)\}$, respecting working rectangle R of height i, where $R = R_{from}(X_i,c_i)$.

$maxBundle(X, c, G)$ denotes the maximum collection of ells that are contained in the ell block (X, c) in Chimera graph G.

Claim: At each iteration, all the maximum partial block clique embeddings respecting rectangles of height i are found.
At $i = 1$, all the maximum partial block clique embeddings of rectangles of height 1 are found.
Suppose the claim is true for $i = j$,

for $i = j + 1$, since $R_{to}(X_{j+1}, c_{j+1}) = R_{from}(X_j, c_j)$, therefore
at step 6: "$\beta \leftarrow maxPartialEmbedding(R_{to}(X, c) \bigcup \{(X, c)\}$",
$maxPartialEmbedding(R_{to}(X, c))$ contains the maximum
partial block clique embedding respecting $R_{to}(X, c)$, since $R_{to}(X, c)$ is of height j.

The algorithm goes through all the ell blocks (X, c) of height $j + 1$. Then for each rectangle R of height $j + 1$, it chooses the partial block clique embedding χ_{j+1} that contains the (X_{j+1}, c_{j+1}) which gives the maximum $\| maxPartialEmbedding(R_{to}(X_{j+1}, c_{j+1})) \bigcup \{(X_{j+1}, c_{j+1})\} \|$ out of all ell blocks from $X_{to}(R)$.
Let $\chi_{optimal}$ be an optimal partial block clique embedding respecting R, then $\chi_{optimal}$ must contain one (X, c) from $X_{to}(R)$. Therefore $\chi_{optimal} = (X, c) \bigcup$\{some partial block embedding respecting $R_{to}(X, c)$\}. Since $\| maxPartialEmbedding(R_{to}(X, c)) \| \geq \|$ \{some partial block embedding respecting $R_{to}(X, c)$\}$\|$, $\chi_{optimal} = (X, c) \bigcup maxPartialEmbedding(R_{to}(X, c))$. And since the algorithm selects the (X, c) that gives the maximum
$\| maxPartialEmbedding(R_{to}(X, c)) \bigcup \{(X, c)\} \|$, $\chi_{j+1} = \chi_{optimal}$.
Hence for $i = j + 1$ the claim is true as well.
Therefore by the end of step 8, the algorithm finds the maximum partial block clique embedding respecting all rectangles of height $n - 1$.
From step 10 to 14, it iterates through all the ell blocks of height n and chooses the block (X, c) which gives the maximum $\| maxPartialEmbedding(R_{to}(X, c)) \bigcup \{(X, c)\} \|$. Therefore it finds the maximum-sized native clique embedding.

3 An Extension of the Algorithm and the Correctness Proof

Algorithm 2 The algorithm to find all maximum-sized native clique embeddings in an induced subgraph of a Chimera graph.

function $NativeCliqueEmbedM(G, n)$
    for $i = 1, ..., n - 1$ **do**

3: **for** each rectangle R of height i and width $n-i$ **do**

4: $\texttt{maxPartialEmbedding}(R) \leftarrow \phi$

5: **for** each ell block (X,c) of height i and width $n-i+1$ **do**

6: $\beta \leftarrow \texttt{maxPartialEmbedding}(R_{to}(X,c)) \bigcup \{(X,c)\}$

7: **if** $\| \texttt{maxPartialEmbedding}(R_{from}(X,c)) \| < \| \beta \|$ **then**

8: $\texttt{maxPartialEmbedding}(R_{from}(X,c)) \leftarrow \beta$

9: **for** each ell block (X,c) of height i and width $n-i+1$ **do**

10: $\beta \leftarrow \texttt{maxPartialEmbedding}(R_{from}(X,c))$

11: **if** $\|\texttt{maxPartialEmbedding}(R_{to}(X,c)) \bigcup \{(X,c)\} \| = \| \beta \|$ **then**

12: $\texttt{allMaxPartialEmbedding}(R_{from}(X,c))$.add $(\texttt{partialEmbedding}_{max} \bigcup \{(X,c)\})$
for all $\texttt{partialEmbedding}_{max} \in \texttt{allMaxPartialEmbedding}(R_{to}(X,c))$

13: $\beta_{max} \leftarrow \phi$

14: **for** each ell block (X,c) of height n and width 1 **do**

15: $\beta \leftarrow \texttt{maxPartialEmbedding}(R_{to}(X,c)) \bigcup \{(X,c)\}$

16: **if** $\| \beta_{max} \| < \| \beta \|$ **then**

17: $\beta_{max} \leftarrow \beta$

18: **for** each ell block (X,c) of height n and width 1 **do**

19: $\beta \leftarrow \texttt{maxPartialEmbedding}(R_{to}(X,c))$

20: **if** $\| \beta_{max} \| = \| \beta \bigcup \{(X,c)\} \|$ **then**

21: maxClique . add$(\alpha \bigcup \{(X,c)\})$ **for all** $\alpha \in \texttt{allMaxPartialEmbedding}(R_{to}(X,c))$

22: **return** $\{\texttt{maxBundle}(X,c,G) | (X,c) \in \beta\}$ **for all** $\beta \in$ maxClique

Proof: Proof by induction.
Claim: For each value of i from 1 to $n-1$, the algorithm finds all the maximum partial native clique embeddings respecting to rectangles of height i.
Base case: For $i = 1$
Similar to Algorithm 1, when step 9 is reached, one maximum partial embedding $\texttt{maxPartialEmbedding}(R)$ for each rectangle R of height 1 is found.
For all ell blocks (X,c) of height 1, $\texttt{maxPartialEmbedding}(R_{to}(X,c)) = \emptyset$ since $R_{to}(X,c)$ does not exist. So as $\texttt{allMaxPartialEmbedding}(R_{to}(X,c))$.
Therefore by the end of the 3rd inner loop (from step 9 to step 12), $\texttt{allMaxPartialEmbedding}(R)$ contains all the ell blocks (X,c) where $(X,c) \in X_{to}(R)$ and $\| \{(X,c)\} \| = \|\texttt{maxPartialEmbedding}(R) \|$

for all R of height 1.

Hence the claim is true for $i = 1$.

Inductive step: Suppose the claim is true for $i = j$

For $i = j + 1$ where $j + 1 \leq n - 1$, when step 9 is reached, one maximum partial embedding for each rectangle R of height of $j + 1$ is found, denoted by $\texttt{maxPartialEmbedding}(R)$.

Since the working rectangle $R_{to}(X, c)$ is of height of j for each ell block (X, c) of height $j + 1$, for each working rectangle $R_{to}(X, c)$, $\texttt{allMaxPartialEmbedding}(R_{to}(X, c))$ contains all the maximum partial embeddings respecting that rectangle.

Suppose there is an optimal set $\texttt{optimalPartialEmbeddings}(R_{from}(X, c))$ that contains all the maximum partial embeddings respecting $R_{from}(X, c)$ and $\texttt{allMaxPartialEmbedding}(R_{to}(X, c))$ does not. Then we know that there is at least one maximum partial embedding that is contained in $\texttt{optimalPartialEmbeddings}(R_{from}(X, c))$ but not in $\texttt{allMaxPartialEmbedding}(R_{to}(X, c))$. Let α denotes that maximum partial embedding.

We know that $\| \alpha \| = \|\texttt{maxPartialEmbedding}(R_{from}(X, c)) \|$ and let (X_{j+1}, c_{j+1}) denotes the ell block of height $j + 1$ in α. Then α = one of the maximum partial embeddings respecting $R_{to}(X_{j+1}, c_{j+1}) \bigcup \{(X_{j+1}, c_{j+1})\}$. Since $\texttt{allMaxPartialEmbedding}(R)$ contains all the maximum partial embeddings respecting R for every R of height j, one of the maximum partial embeddings respecting $R_{to}(X_{j+1}, c_{j+1})$ should be contained in $\texttt{allMaxPartialEmbedding}(R_{to}(X_{j+1}, c_{j+1}))$. Therefore $\alpha \in \texttt{allMaxPartialEmbedding}(R_{to}(X, c))$. Hence a contradiction.

Therefore the claim is true for $i = j + 1$.

Therefore when the algorithm reaches step 13, $\texttt{allMaxPartialEmbedding}(R_{from}(X, c))$ contains all the maximum partial native clique embeddings of size $n - 1$ for each rectangle $R_{from}(X, c)$ of height $n - 1$.

When the algorithm reaches step 18, β_{max} denotes one of the maximum native clique embeddings as proven in the proof of Algorithm 1.

Suppose there exists a maximum embedding α that does not belong to maxClique. By definition, $\| \alpha \| = \| \beta_{max} \|$. Let (X_n, c_n) denotes the ell block of height n in α. α consists of one maximum partial embedding respecting $R_{to}(X_n, c_n)$ and (X_n, c_n). Since any maximum partial embedding respecting R of height $n - 1$ is contained in $\texttt{allMaxPartialEmbedding}(R)$ and $R_{to}(X_n, c_n)$ is of height $n - 1$, $\alpha \in$ maxClique following the algorithm. Hence a contradiction.

Therefore Algorithm 2 finds all the maximum native clique embeddings of size n given a Chimera graph G and number n.

4 Conclusion

The actual D-Wave 2X hardware we have access to have faulty couplers as well as faulty qubits so the algorithm is modified as suggested in [1] to take into consideration the faulty couplers. However, this improvement is highly restricted, it will only work when there is at most one intra-cell faulty coupler per unit cell. For more general cases, a more sophisticated algorithm is needed. The $NativeCliqueEmbed$ algorithm is also extended to find all the maximum native clique embeddings instead of just one maximum native clique embedding. Implementations for both algorithms are included in the appendix.

Acknowledgements

This report was based on Amanda Yao's research done as a Science Faculty Summer Scholarship at the University of Auckland under the supervision of Cristian Calude and Michael J. Dinneen. The CDMTCS version of the final report was edited by Richard Hua.

References

[1] Tomas Boothby, Andrew D. King, and Aidan Roy. Fast clique minor generation in Chimera qubit connectivity graphs. *Quantum Information Processing*, 15(1):495–508, 2016.

[2] Cristian S. Calude, Michael J. Dinneen, and Richard Hua. QUBO formulations for the graph isomorphism problem and related problems. *Theoretical Computer Science*, 701:54–69, 2017.

Appendix

Python Program that Implements NativeCliqueEmbed Algorithm

```
import sys
from dwave_sapi2.util import get_chimera_adjacency
import networkx as nx
from itertools import product, combinations
from collections import Counter
from dwave_sapi2.util import chimera_to_linear_index
from dwave_sapi2.util import linear_index_to_chimera
import random
import math
```

```
import itertools
import json

order=int(input())
[M,N,L]=[int(math.sqrt(order/8)), int(math.sqrt(order/8)), 4]
A=get_chimera_adjacency(M,N,L)
G = nx.empty_graph(order)
G.add_edges_from(A)

#code taken from chimera_graph.py
C = nx.empty_graph(order)

for value in range(order):
    b = raw_input()
    b = b.split()
    for value2 in b:
        C.add_edge(value, int(value2))

faultyQubits = [v for v in C.nodes() if len(C[v])==0]
missingE = []
for u in range(order-1):
    if u in faultyQubits: continue
    for v in range(u+1,order):
         if v in faultyQubits: continue
         if v in G[u] and v not in C[u]: missingE.append([u,v])

missingEdges = []
for e in missingE:
    ins = linear_index_to_chimera(e, M, N, L)
    missingEdges.append(ins)

n = int(sys.argv[1])

def maxBundle(X,c, faultyQubits, missingEdges, M, N, L):
    xCoordinate = c[0]
    yCoordinate = c[1]
    hList = []
    vList = []
    for i in X:
        if i[0] == xCoordinate:
            vList.append(i)
        if i[1] == yCoordinate:
            hList.append(i)
```

```python
    directionH = 0
    directionV = 0
    for e in hList:
        if e[0] > xCoordinate:
            directionH = 1
            break
    for e in vList:
        if e[1] > yCoordinate:
            directionV = 1
            break
    posHFaultyQubits = []

    for e in hList:
        for i in range(L):
            x = [e[0]]
            y = [e[1]]
            u = [1]
            k = [i]
            ind = chimera_to_linear_index(x,y,u,k,M,N,L)
            ind = int(''.join(map(str,ind)))
            if ind in faultyQubits and i not in posHFaultyQubits:
                posHFaultyQubits.append(i)
    posVFaultyQubits = []
    for e in vList:
        for i in range(L):
            x = [e[0]]
            y = [e[1]]
            u = [0]
            k = [i]
            ind = chimera_to_linear_index(x,y,u,k,M,N,L)
            ind = int(''.join(map(str,ind)))
            if ind in faultyQubits and i not in posVFaultyQubits:
                posVFaultyQubits.append(i)

    # Difference starts here
    counterH = 0
    for i in range(L):
        if i not in posHFaultyQubits:
            if directionH == 1:
                maxBH[counterH] = [(x, yCoordinate, 1, i) for x in range(
    xCoordinate, xCoordinate + len(hList))] #depends on direction
                for h in missingEdges:
```

```
                    if tuple(h[0]) in maxBH[counterH] and tuple(h[1]) in maxBH[counterH]:
                        maxBH[counterH] = []
                        counterH = counterH - 1
                        break

            else:
                maxBH[counterH] = [(x, yCoordinate, 1, i) for x in range(xCoordinate - len(hList) + 1, xCoordinate + 1)]
                for h in missingEdges:
                    if tuple(h[0]) in maxBH[counterH] and tuple(h[1]) in maxBH[counterH]:
                        maxBH[counterH] = []
                        counterH = counterH - 1
                        break
            counterH = counterH + 1

    noH = counterH

    maxBV = {}
    counterV = 0
    for i in range(L):
        if i not in posVFaultyQubits:
            if directionV == 1:
                maxBV[counterV] = [(xCoordinate, y, 0, i) for y in range(yCoordinate, yCoordinate + len(vList))] #depends on direction
                for h in missingEdges:
                    if tuple(h[0]) in maxBV[counterV] and tuple(h[1]) in maxBV[counterV]:
                        maxBV[counterV] = []
                        counterV = counterV - 1
                        break
            else:
                maxBV[counterV] = [(xCoordinate, y, 0, i) for y in range(yCoordinate - len(vList) + 1, yCoordinate + 1)]
                for h in missingEdges:
                    if tuple(h[0]) in maxBV[counterV] and tuple(h[1]) in maxBV[counterV]:
                        maxBV[counterV] = []
                        counterV = counterV - 1
                        break
            counterV = counterV + 1
```

```python
    noV = counterV
    size = min([noV,noH])
    maxB = {}
    missingIndex = -1
    #fCount = 0
    sizeF = 0
    for i in range(size):
        maxB[i] = maxBV[i] + maxBH[i]
        for h in missingEdges:
            if tuple(h[0]) in maxB[i] and tuple(h[1]) in maxB[i]:
                maxB[i] = []
                missingIndex = i
                break
    if size > 1 and missingIndex != -1:
        if missingIndex == 0:
            maxB[0] = maxBV[0] + maxBH[1]
            maxB[1] = maxBV[1] + maxBH[0]
        else:
            maxB[missingIndex] = maxBV[missingIndex] + maxBH[missingIndex - 1]
            maxB[missingIndex - 1] = maxBV[missingIndex - 1] + maxBH[missingIndex]
        sizeF = len(maxB)
    else:
        if size == 1 and missingIndex != -1:
            if noV > 1 :
                maxB[0] = maxBV[1] + maxBH[0]
                sizeF = len(maxB)
            else:
                if noH > 1:
                    maxB[0] = maxBV[0] + maxBH[1]
                    sizeF = len(maxB)
                else:
                    sizeF = 0
                    maxB = {}
        else:
            sizeF = len(maxB)
    return (maxB, sizeF)

def size(lis):
    res = 0
    for e in lis:
```

```
        res = res + maxBundle(e[0],e[1],faultyQubits,missingEdges,M,N,L)[1]
    return res

maxPartialEmbedding = {}
R = {}
From = {}
To = {}
Rto = {}
Rfrom = {}

# Enumerate and store all rectangles and ell blocks
for i in range(1,n):
    for j in range(M-n+i+1):
        for k in range(N-i+1):
            R[i,j,k] = ((j,k),(j+n-i-1,k+i-1))
            cur = R[i,j,k]
            To[cur] = []
            if j-1 >= 0:
                c1 = (j-1, k)
                c2 = (j-1, k+i-1)
                X1 = list(set().union(*[[(j-1,b) for b in range(k, k+i)],[(a,k) for a in range(j-1, j+n-i)]]))
                X2 = list(set().union(*[[(j-1,b) for b in range(k, k+i)],[(a,k+i-1) for a in range(j-1, j+n-i)]]))
                To[cur].append((X1, c1))
                Rfrom[(tuple(X1),c1)] = cur
                To[cur].append((X2, c2))
                Rfrom[(tuple(X2),c2)] = cur
            if j+n-i <= M - 1:
                c1 = (j+n-i, k)
                c2 = (j+n-i, k+i-1)
                X1 = list(set().union(*[[(j+n-i,b) for b in range(k, k+i)],[(a,k) for a in range(j, j+n-i+1)]]))
                X2 = list(set().union(*[[(j+n-i,b) for b in range(k, k+i)],[(a,k+i-1) for a in range(j, j+n-i+1)]]))
                To[cur].append((X1, c1))
                Rfrom[(tuple(X1),c1)] = cur
                To[cur].append((X2, c2))
                Rfrom[(tuple(X2),c2)] = cur
            To[cur].sort()
            To[cur] = list(To[cur] for To[cur],_ in itertools.groupby(To[cur]))
```

```python
            From[cur] = []
            if k-1 >= 0:
                c1 = (j,k-1)
                c2 = (j+n-i-1, k-1)
                X1 = list(set().union(*[[(j,b) for b in range(k-1, k+i)],[(a,k-1) for a in range(j, j+n-i)]]))
                X2 = list(set().union(*[[(j+n-i-1, b) for b in range(k-1, k+i)],[(a,k-1) for a in range(j, j+n-i)]]))
                From[cur].append((X1, c1))
                Rto[(tuple(X1),c1)] = cur
                From[cur].append((X2, c2))
                Rto[(tuple(X2),c2)] = cur
            if k+i <= N - 1:
                c1 = (j, k+i)
                c2 = (j+n-i-1, k+i)
                X1 = list(set().union(*[[(j, b) for b in range(k, k+i+1)],[(a, k+i) for a in range(j, j+n-i)]]))
                X2 = list(set().union(*[[(j+n-i-1, b) for b in range(k, k+i+1)],[(a, k+i) for a in range(j, j+n-i)]]))
                From[cur].append((X1, c1))
                Rto[(tuple(X1),c1)] = cur
                From[cur].append((X2, c2))
                Rto[(tuple(X2),c2)] = cur
            From[cur].sort()
            From[cur] = list(From[cur] for From[cur],_ in itertools.groupby(From[cur]))

# Algorithm 1
for i in range(1,n):
    for j in range(M-n+i+1):
        for k in range(N-i+1):
            cur = R[i,j,k]
            maxPartialEmbedding[cur] = []
            for e in To[cur]:
                if i == 1:
                    Beta = [e]
                else:
                    Beta = maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]+[e]
                if size(maxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]]) < size(Beta):
                    maxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]] = Beta
BetaMax = []
```

```
for j in range(M):
    for k in range(N-n+2):
        cur = R[n-1,j,k]
        for e in From[cur]:
            Beta = maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]+[e]
            if size(BetaMax) < size(Beta):
                BetaMax = Beta

print ("Chain length: " + str(n + 1) + "\n")
print ("Max clique order: " + str(size(BetaMax)) + "\n")
for e in BetaMax:
    temp = []
    print ("Ell block: " + str(e) + "\n")
    b = maxBundle(e[0],e[1],faultyQubits,missingEdges,M,N,L)[0]
    for j in b.values():
        temp.append(chimera_to_linear_index(j, 12, 12, 4))
    print ("Ells: " + str(temp) + "\n")
```

pythonForIncompleteEdges_1_.py

Python Program that Implements NativeCliqueEmbedM Algorithm

```
import sys
from dwave_sapi2.util import get_chimera_adjacency
import networkx as nx
from itertools import product, combinations
from collections import Counter
from dwave_sapi2.util import chimera_to_linear_index
from dwave_sapi2.util import linear_index_to_chimera
import random
import math
import itertools
import json

order=int(input())
[M,N,L]=[int(math.sqrt(order/8)), int(math.sqrt(order/8)), 4]
A=get_chimera_adjacency(M,N,L)
G = nx.empty_graph(order)
G.add_edges_from(A)

#code taken from chimera_graph.py
C = nx.empty_graph(order)
```

```python
for value in range(order):
    b = raw_input()
    b = b.split()
    for value2 in b:
        C.add_edge(value, int(value2))

faultyQubits = [v for v in C.nodes() if len(C[v])==0]
missingE = []
for u in range(order-1):
    if u in faultyQubits: continue
    for v in range(u+1,order):
        if v in faultyQubits: continue
        if v in G[u] and v not in C[u]: missingE.append([u,v])

missingEdges = []
for e in missingE:
    ins = linear_index_to_chimera(e, M, N, L)
    missingEdges.append(ins)

n = int(sys.argv[1])

def maxBundle(X,c, faultyQubits, missingEdges, M, N, L):
    xCoordinate = c[0]
    yCoordinate = c[1]
    hList = []
    vList = []
    for i in X:
        if i[0] == xCoordinate:
            vList.append(i)
        if i[1] == yCoordinate:
            hList.append(i)
    directionH = 0
    directionV = 0
    for e in hList:
        if e[0] > xCoordinate:
            directionH = 1
            break
    for e in vList:
        if e[1] > yCoordinate:
            directionV = 1
            break
    posHFaultyQubits = []
```

```
    for e in hList:
        for i in range(L):
            x = [e[0]]
            y = [e[1]]
            u = [1]
            k = [i]
            ind = chimera_to_linear_index(x,y,u,k,M,N,L)
            ind = int(''.join(map(str,ind)))
            if ind in faultyQubits and i not in posHFaultyQubits:
                posHFaultyQubits.append(i)
    posVFaultyQubits = []
    for e in vList:
        for i in range(L):
            x = [e[0]]
            y = [e[1]]
            u = [0]
            k = [i]
            ind = chimera_to_linear_index(x,y,u,k,M,N,L)
            ind = int(''.join(map(str,ind)))
            if ind in faultyQubits and i not in posVFaultyQubits:
                posVFaultyQubits.append(i)

# Difference starts here
    maxBH = {}
    counterH = 0
    for i in range(L):
        if i not in posHFaultyQubits:
            if directionH == 1:
                maxBH[counterH] = [(x, yCoordinate, 1, i) for x in range(xCoordinate, xCoordinate + len(hList))] #depends on direction
                for h in missingEdges:
                    if tuple(h[0]) in maxBH[counterH] and tuple(h[1]) in maxBH[counterH]:
                        maxBH[counterH] = []
                        counterH = counterH - 1
                        break

            else:
                maxBH[counterH] = [(x, yCoordinate, 1, i) for x in range(xCoordinate - len(hList) + 1, xCoordinate + 1)]
                for h in missingEdges:
                    if tuple(h[0]) in maxBH[counterH] and tuple(h[1]) in maxBH[counterH]:
```

```
                        maxBH[counterH] = []
                        counterH = counterH - 1
                        break
            counterH = counterH + 1

    noH = counterH

    maxBV = {}
    counterV = 0
    for i in range(L):
        if i not in posVFaultyQubits:
            if directionV == 1:
                maxBV[counterV] = [(xCoordinate, y, 0, i) for y in range(yCoordinate, yCoordinate + len(vList))] #depends on direction
                for h in missingEdges:
                    if tuple(h[0]) in maxBV[counterV] and tuple(h[1]) in maxBV[counterV]:
                        maxBV[counterV] = []
                        counterV = counterV - 1
                        break
            else:
                maxBV[counterV] = [(xCoordinate, y, 0, i) for y in range(yCoordinate - len(vList) + 1, yCoordinate + 1)]
                for h in missingEdges:
                    if tuple(h[0]) in maxBV[counterV] and tuple(h[1]) in maxBV[counterV]:
                        maxBV[counterV] = []
                        counterV = counterV - 1
                        break
            counterV = counterV + 1

    noV = counterV
    size = min([noV,noH])
    maxB = {}
    missingIndex = -1
    sizeF = 0
    for i in range(size):
        maxB[i] = maxBV[i] + maxBH[i]
        for h in missingEdges:
            if tuple(h[0]) in maxB[i] and tuple(h[1]) in maxB[i]:
                maxB[i] = []
                missingIndex = i
                break
```

```python
    if size > 1 and missingIndex != -1:
        if missingIndex == 0:
            maxB[0] = maxBV[0] + maxBH[1]
            maxB[1] = maxBV[1] + maxBH[0]
        else:
            maxB[missingIndex] = maxBV[missingIndex] + maxBH[missingIndex - 1]
            maxB[missingIndex - 1] = maxBV[missingIndex - 1] + maxBH[missingIndex]
        sizeF = len(maxB)
    else:
        if size == 1 and missingIndex != -1:
            if noV > 1 :
                maxB[0] = maxBV[1] + maxBH[0]
                sizeF = len(maxB)
            else:
                if noH > 1:
                    maxB[0] = maxBV[0] + maxBH[1]
                    sizeF = len(maxB)
                else:
                    sizeF = 0
                    maxB = {}
        else:
            sizeF = len(maxB)
    return (maxB, sizeF)

def size(lis):
    res = 0
    for e in lis:
        res = res + maxBundle(e[0],e[1],faultyQubits,missingEdges,M,N,L)[1]
    return res

allMaxPartialEmbedding = {}
maxPartialEmbedding = {}
R = {}
From = {}
To = {}
Rto = {}
Rfrom = {}

#Enumerate and store all rectangles and ell blocks
for i in range(1,n):
```

```
    for j in range(M-n+i+1):
        for k in range(N-i+1):
            R[i,j,k] = ((j,k),(j+n-i-1,k+i-1))
            cur = R[i,j,k]
            To[cur] = []
            if j-1 >= 0:
                c1 = (j-1, k)
                c2 = (j-1, k+i-1)
                X1 = list(set().union(*[[(j-1,b) for b in range(k, k+i)],[(a,k) for a in range(j-1, j+n-i)]]))
                X2 = list(set().union(*[[(j-1,b) for b in range(k, k+i)],[(a,k+i-1) for a in range(j-1, j+n-i)]]))
                To[cur].append((X1, c1))
                Rfrom[(tuple(X1),c1)] = cur
                To[cur].append((X2, c2))
                Rfrom[(tuple(X2),c2)] = cur
            if j+n-i <= M - 1:
                c1 = (j+n-i, k)
                c2 = (j+n-i, k+i-1)
                X1 = list(set().union(*[[(j+n-i,b) for b in range(k, k+i)],[(a,k) for a in range(j, j+n-i+1)]]))
                X2 = list(set().union(*[[(j+n-i,b) for b in range(k, k+i)],[(a,k+i-1) for a in range(j, j+n-i+1)]]))
                To[cur].append((X1, c1))
                Rfrom[(tuple(X1),c1)] = cur
                To[cur].append((X2, c2))
                Rfrom[(tuple(X2),c2)] = cur
            To[cur].sort()
            To[cur] = list(To[cur] for To[cur],_ in itertools.groupby(To[cur]))
            From[cur] = []
            if k-1 >= 0:
                c1 = (j,k-1)
                c2 = (j+n-i-1, k-1)
                X1 = list(set().union(*[[(j,b) for b in range(k-1, k+i)],[(a,k-1) for a in range(j, j+n-i)]]))
                X2 = list(set().union(*[[(j+n-i-1, b) for b in range(k-1, k+i)],[(a,k-1) for a in range(j, j+n-i)]]))
                From[cur].append((X1, c1))
                Rto[(tuple(X1),c1)] = cur
                From[cur].append((X2, c2))
                Rto[(tuple(X2),c2)] = cur
            if k+i <= N - 1:
```

```
                c1 = (j, k+i)
                c2 = (j+n-i-1, k+i)
                X1 = list(set().union(*[[(j, b) for b in range(k, k+i+1)],[(a, k+i) for a in range(j, j+n-i)]]))
                X2 = list(set().union(*[[(j+n-i-1, b) for b in range(k, k+i+1)],[(a, k+i) for a in range(j, j+n-i)]]))
                From[cur].append((X1, c1))
                Rto[(tuple(X1),c1)] = cur
                From[cur].append((X2, c2))
                Rto[(tuple(X2),c2)] = cur
            From[cur].sort()
            From[cur] = list(From[cur] for From[cur],_ in itertools.groupby(From[cur]))

#Algorithm 2
for i in range(1,n):
    for j in range(M-n+i+1):
        for k in range(N-i+1):
            cur = R[i,j,k]
            maxPartialEmbedding[cur] = []
            for e in To[cur]:
                if i == 1:
                    Beta = [e]
                else:
                    Beta = maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]+[e]
                if size(maxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]]) < size(Beta):
                    maxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]] = Beta
    for j in range(M-n+i+1):
        for k in range(N-i+1):
            cur = R[i,j,k]
            allMaxPartialEmbedding[cur] = []
            for e in To[cur]:
                Beta = maxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]]
                if i == 1:
                    if size([e]) == size(Beta):
                        allMaxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]].append([e])
                else:
                    if size(maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]+[e]) == size(Beta):
```

```python
                        for maxP in allMaxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]:
                            allMaxPartialEmbedding[Rfrom[(tuple(e[0]),e[1])]].append(maxP + [e])
BetaMax = []
maxClique = []
for j in range(M):
    for k in range(N-n+2):
        cur = R[n-1,j,k]
        for e in From[cur]:
            Beta = maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]+[e]
            if size(BetaMax) < size(Beta):
                BetaMax = Beta
for j in range(M):
    for k in range(N-n+2):
        cur = R[n-1,j,k]
        for e in From[cur]:
            Beta = maxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]
            if size(BetaMax) == size(Beta + [e]):
                for maxP in allMaxPartialEmbedding[Rto[(tuple(e[0]),e[1])]]:
                    maxClique.append(maxP + [e])

c = 1
print ("Chain length: " + str(n + 1) + "\n")
print ("Max clique order: " + str(size(BetaMax)) + "\n")
for beta in maxClique:
    print ("Maximum Clique Embedding " + str(c) + ": \n")
    print
    for e in beta:
        temp = []
        print ("Ell block: " + str(e) + "\n")
        b = maxBundle(e[0],e[1],faultyQubits,missingEdges,M,N,L)[0]
        for j in b.values():
            temp.append(chimera_to_linear_index(j, 12, 12, 4))
        print ("Ells: " + str(temp) + "\n")
    print
    print
    c = c + 1
```

pythonForIncompleteEdges_1_E.py