CPEN513 Assignment 3: Branch and Bound Bi-Partitioning

Martin Chua
35713411
Mar 13, 2023

Summary

This report details the implementation of the branch and bound bi-partitioning algorithm and improvements. The brand and bound bi-partitioning algorithm is run on benchmark circuits given on Canvas. The algorithm is implemented in Python and is inspired by the recursive *steves_routine* algorithm given in the assignment. The graphics library is implemented using matplotlib. A copy of source code and documentation can be found in Appendix B. This report is divided into the following sections: base branch and bound bi-partitioning algorithm implementation, initiatives on top of the base algorithm to improve its performance, how the code was tested, and graphics.

Implementation of the base branch and bound bi-partioning algorithm

The branch and bound bi-partioning algorithm is implemented as follows in the pseudocode below:

```
Algorithm 1: branch_bound(current list, current node, nets, left count, right count, previous cost)
    1. Calculate current cost given previous cost, current node (algorithm 2)
    2. if current cost < best cost:
        a. append current node to current list
        b. if number of levels < total count:
            i. if left count < total count/2:
                1. branch_bound(left)
            ii. if right count < total count/2:
                1. branch_bound(right)
        c. else update best cost
```

The current list is defined as the list of nodes that have been assigned. Each node (as well as current node) within this list is a tuple of cell number and partition location (ie. left or right, encoded as 0 or 1). Nets is the list of all nets. There are also performance improvements for nets described in Algorithm 2. Left and right count describe the number of times the algorithm has gone left or right in the binary tree. Cost is defined as the number of nets that cross the partition boundary. The goal of the algorithm is to minimize the nets that cross/cut this boundary.

The initial best cost is based on the total number of nets from the current benchmark circuit. This is changed later in the initiatives section. Algorithm 1 includes several opportunities for pruning for performance benefits. (2) immediately stops the current node from proceeding if the current cost is greater than the global best cost. (2b) checks to see if there are no more leaves. The global best cost is updated if the current cost is < best cost. (2bi, 2bii) stops the current node from proceeding on the left or right branch if its left count or right count is greater than a balanced final partition, or half of the total cell count.

Algorithm 2 below describes the calculation of a node's current cost (label).

```
Algorithm 2: calculate_label(current list, current node, nets, cost)
    1. for net in nets:
        a. if current node exists in net and is on a different branch than other cells in the same net:
            i. update cost by 1
            ii. remove current net from nets list
    2. Return cost and nets
```

Algorithm 2 finds the cost of the current node by calculating only the cuts done by the current node in all existing nets. Since the previous cost does not need to be recomputed, we also minimize memory usage (and latency) of future nodes by removing nets that have already been computed (existing cut). In doing so, we can return the cost and current updated net list.

Table I: The min cut values for the base branch and bound bi-partitioning algorithm on provided benchmark circuits. Runtime given is the **base**[1] or **initiative**[2] results using an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz.

Circuit	**min cut**	**Runtime (seconds)**
con1	4	0.014[1]
twocm	N/A[1,2] (<10)[2]	core dumped[1,2]
cm82a	1	0.002[1]
cm162a	6	319.00[1]
ugly16	16	0.11[1]
cm138a	4	0.16[1]
z4ml	3	0.034[1]
cc	N/A[1] 4[2]	core dumped[1] 3616.85[2]
cm150a	6	798.21[1]
ugly8	8	0.0005[1]

Table I above outlines the minimum cut values found for each benchmark circuit. A runtime is given for comparison between circuit benchmarks. We see that several benchmark circuits take a long time to run. Unfortunately, twocm and cc were unable to be successfully run due to maximum cpu time limits imposed on the ECE server. Some improvements were made to decrease runtime, which in turn meant better performance with the same min-cut cost. Twocm was attempted using the base algorithm on a M2 macbook air but was cancelled (during which current best cost = 10) after running for hours due to concern for thermal limitations. Cc was later attempted on the ECE server with initiatives 1-4 and ran successfully after 1h.

Initiatives

Several ideas were brainstormed to improve the runtime of the algorithm. Since the algorithm always provides an optimal value, we can only attempt to improve runtime. The details of their implementation are below:

1. Parallel processing by adapting the baseline to use Python's built-in multiprocessing package. Care must be used to synchronize global values (ie. best cost). This is done using multiprocessing's Process and Value objects. Although we can parallelize up to N, only 2 processes were spawned due to physical core limitations.
2. Initial random heuristics. As the initial state is non-optimal, we can try to find a better initial non-optimal partition out of the best Y random iterations. This effectively allows for early pruning of the entire search space by determining a better initial best cost. Initially, Y is set to a constant, 1000, and performs reasonably fast. However, this increases runtime for small circuits as seen in Table II. As a result, we arbitrarily use the cost function $Y = (N/10)^X$, where X is experimentally set to 7 and N is the number of nets in the circuit, from the insight that the number of nets exponentially increases the search time more than the number of cells in the benchmark circuits provided. Since this is an arbitrary cost function, it is likely there are better functions that more adequately map to general circuits beyond the 10 benchmark circuits given. We set a max threshold to 15,000 for larger circuits.
3. Cyclic best-first-search (C-BFS)[1]. Instead of the baseline DFS starting from the left-most branch and iteratively performing DFS from the left with pruning, we can attempt to use C-BFS to speed-up convergence by greedily prioritizing and traversing the search space between left and right branches based on which branch node has the least cost. This is done by modifying the recursive DFS to recursive C-BFS, by "peeking" and preemptively computing cost for leaf nodes. To prevent the same label calculation in the leaf, we directly compute the leaf cost during peeking, modifying Algorithm 1 + 2. After peeking, we prioritize the node with the smaller cost first.
4. Initial low-temperature simulated annealing heuristic, to aid in pruning the search space for large circuits such as cc and twocm. We experimentally found that this heuristic decreases the initial cost by up to 12 before bi-

[1] Cyclic best-first search as described in: D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, vol. 19, pp. 79–102, 2016.

partitioning. This is implemented by swapping one random left cell with the right, then computing cost. If cost goes down, the swap is taken. This is performed after Initiative 2, and the annealing iterations are set to half of the random iteration value as the local minima converges quickly; note that the local minima is not used during bi-partitioning. Only the cost for pruning is saved, as the runtime cost of nodes visited (<7500) is trivially small compared to the total nodes within larger circuits.

5. The following were not implemented:
    a. Early exiting. As the branch and bound algorithm finds an optimal result, early exiting can prevent the global optimum from being found if convergence takes a long time (especially for larger circuits where the search space is exponentially larger). As a result, this was not implemented.
    b. Compiled code. Compiled languages such as C (harder) will perform much faster than Python (easier).

Table II: Comparison of the base single-threaded branch and bound bi-partitioning algorithm versus initiatives. Run on select circuits using an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz.

	Base (s)	**Initiative 1+2 with initial constant Y=1000 iterations**	**Initiative 1+2 with initial dynamic Y= $(N/10)^X$ iterations**	**Initiative 1-3**	**Initiative 1-4**
cm138a	0.18	0.21 (+16%)	0.08 (-56%)	0.077 (-57.3%)	0.0805 (-55.3%)
cm162a	319.00	159.74 (-50.0%)	161.36 (-49.9%)	160.15 (-49.8%)	165.36 (-48.1%)
cm150a	798.21	474.72 (-40.6%)	483.00 (-39.5%)	388.12 (-51.4%)	410.58 (-48.6%)

Table II outlines the initiatives on several benchmark circuits. The increase in time spent on smaller circuits such as cm138a, for initiative 1+2 can be due to the overhead in instantiating the Python parallel processing packages + variable synchronization and random initialization of 1000 iterations. The latter is mitigated using a dynamic initial random heuristic. Regardless of the dynamic or static initial heuristic, the runtime of larger circuits such as cm150a and cm162a is significantly reduced by 40-50%.

With C-BFS, the runtime of smaller circuits is reduced significantly. But, for larger circuits where the time spent traversing nodes dominates convergence time, there is less of a significant difference between C-BFS and DFS as seen in cm162 (0.01%) or cm150a (12%). The difference in percentage gap between these two circuits could possibly be due to structural differences in the circuit and nets, where cm150a sees a more noticeable improvement by a decrease in convergence time. This makes sense because converging faster within a larger search space prunes more nodes before visiting them.

Overall, initiatives 1-3 substantially decrease runtime by around 50-57% as compared to baseline, although there is a tradeoff to consider for initiative 4. For larger circuits (ie. twocm), the initial heuristics drop the best cost from random=23 to annealing=11. This is a substantial decrease given that when run using the base algorithm, it took many hours to go from best cost=27 to best cost=20 during bi-partitioning. However, this still took too long to run with initiatives. Although Initiative 1-3 performs better than Initiative 1-4, the latter is kept. To improve robustness of results, averaging could have been done, however run-to-run variations did not differ much (~ ±2%) and larger circuits took too long to run repeatedly.

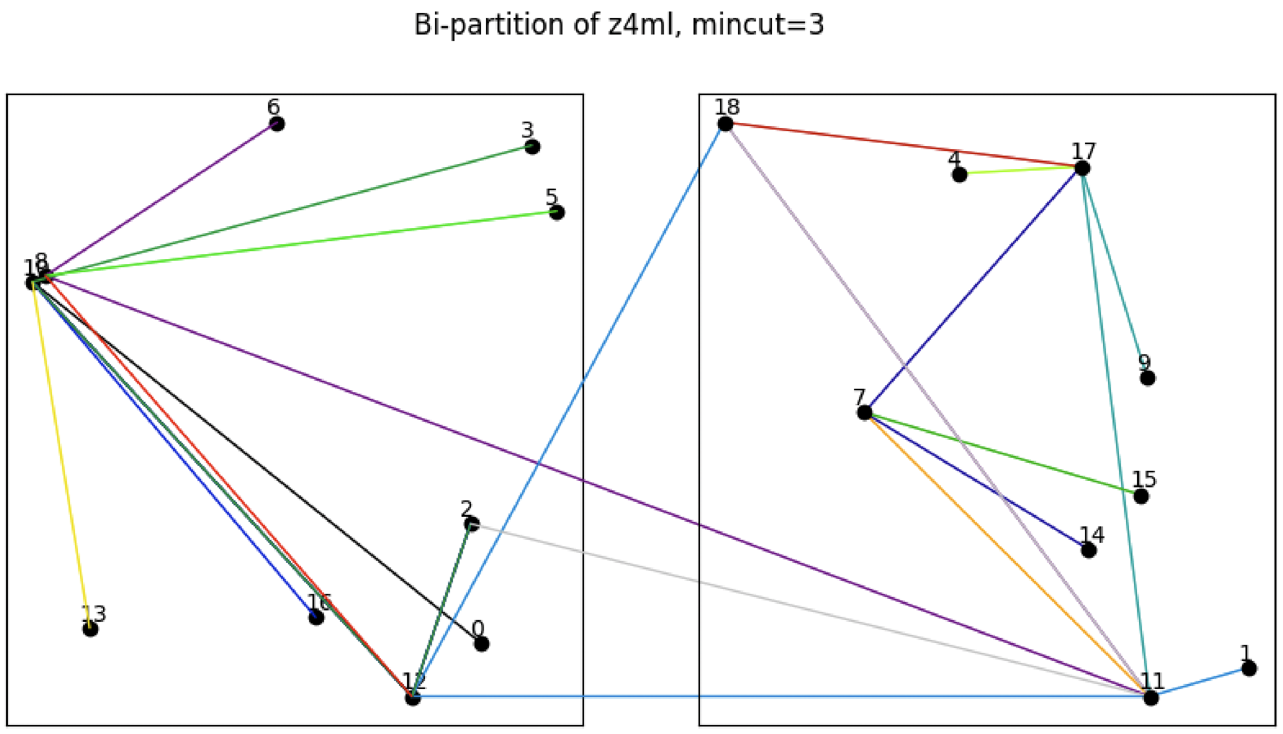
Lastly, originally Python's deepcopy to propagate lists was used in the baseline. Deepcopy is a major bottleneck due to latency of off-chip memory accesses. After switching to Python's copy it is significantly faster. This was implemented as part of the baseline algorithm due to the problem being a Python optimization rather than algorithmic.

Verification

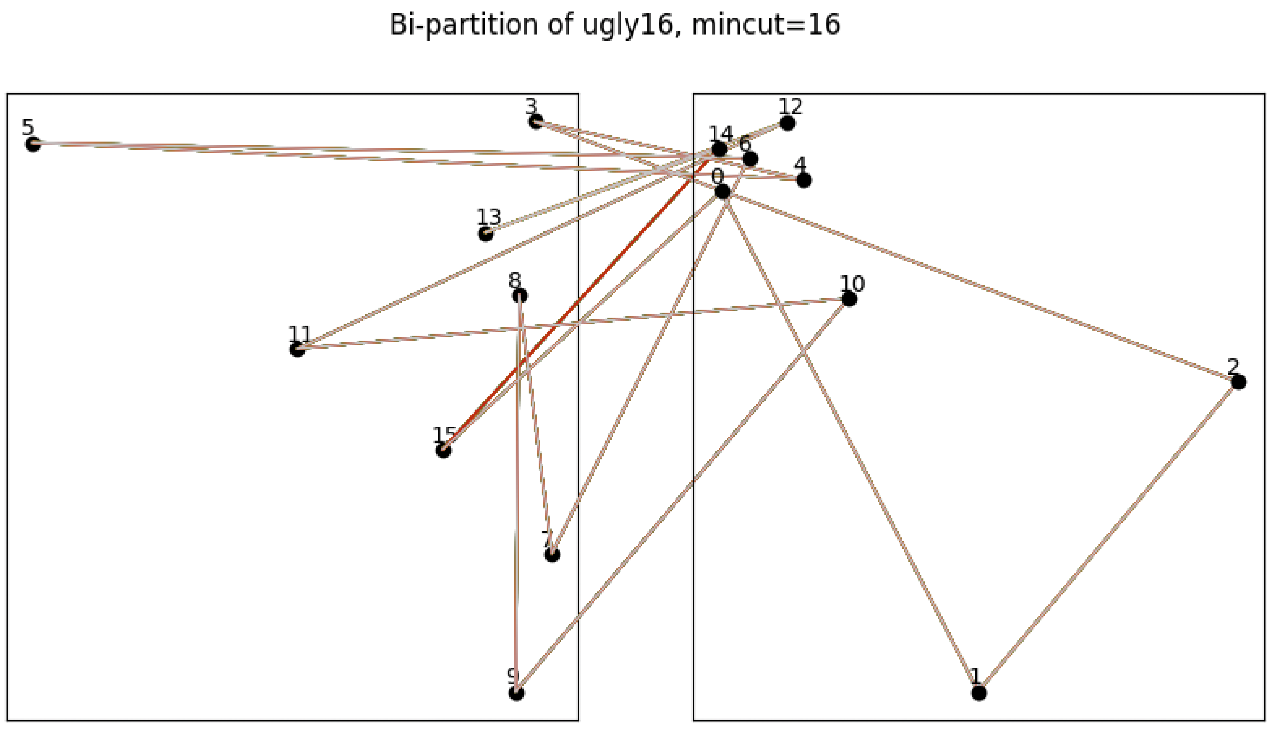
Testing was done in two parts: manually using the debugger + breakpoints and automatically using assertions/if-else statements for unit testing individual functions (rather than explicit tests using Pytest as done in assignment 1). Small checks are also within the algorithm code to ensure correct functionality, and the code is well-documented. For example, the branch and bound function checks cell size and records the best cost only if left and right are equal or at most by 1, and otherwise there is an assertion.

Appendix A: Graphics

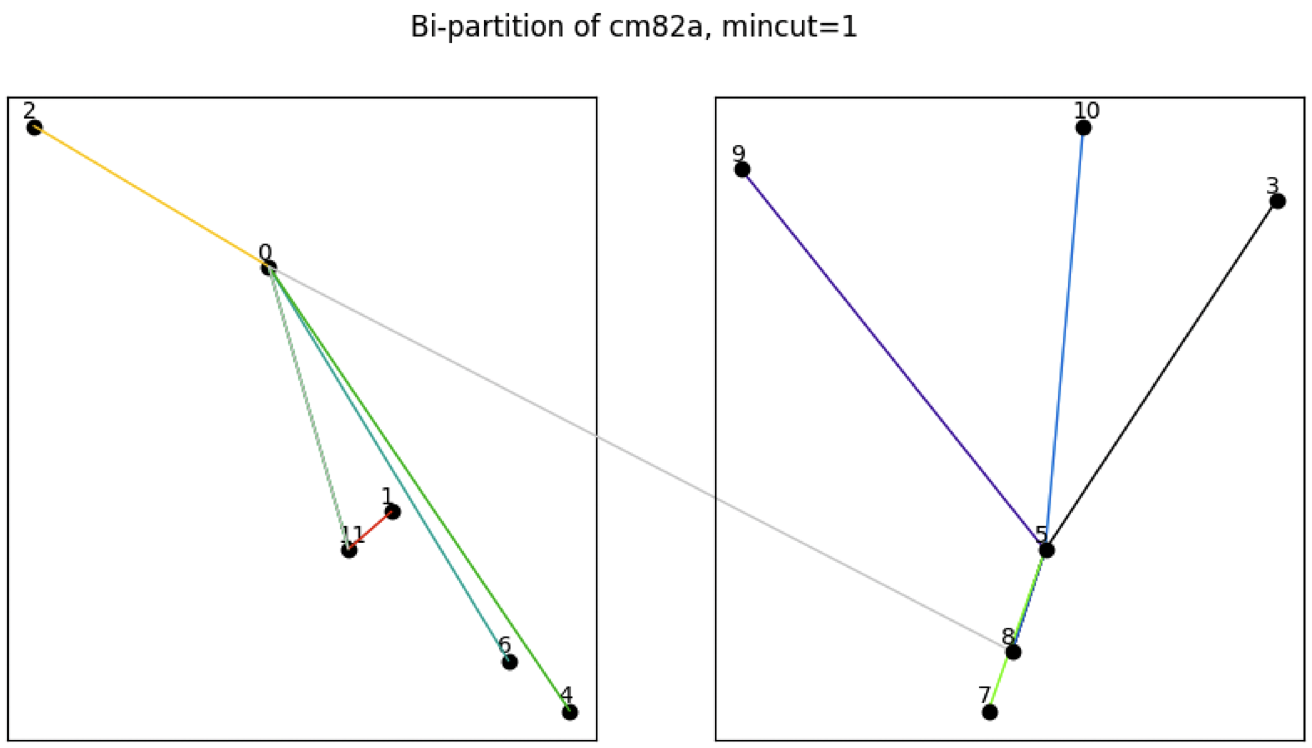
The asn3.py script can be run via command line or an IDE such as Visual Studio Code + Python extension. Examples of graphics are below.



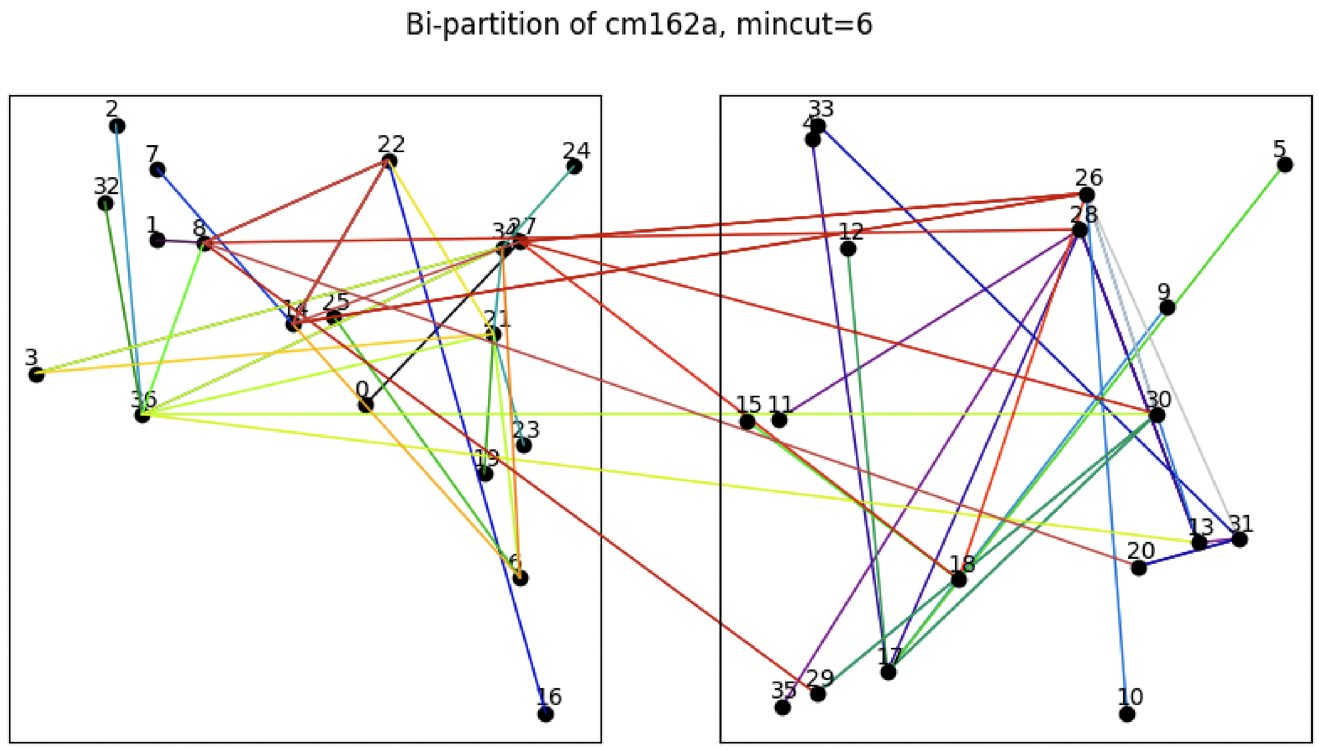
^ three different colours crossing



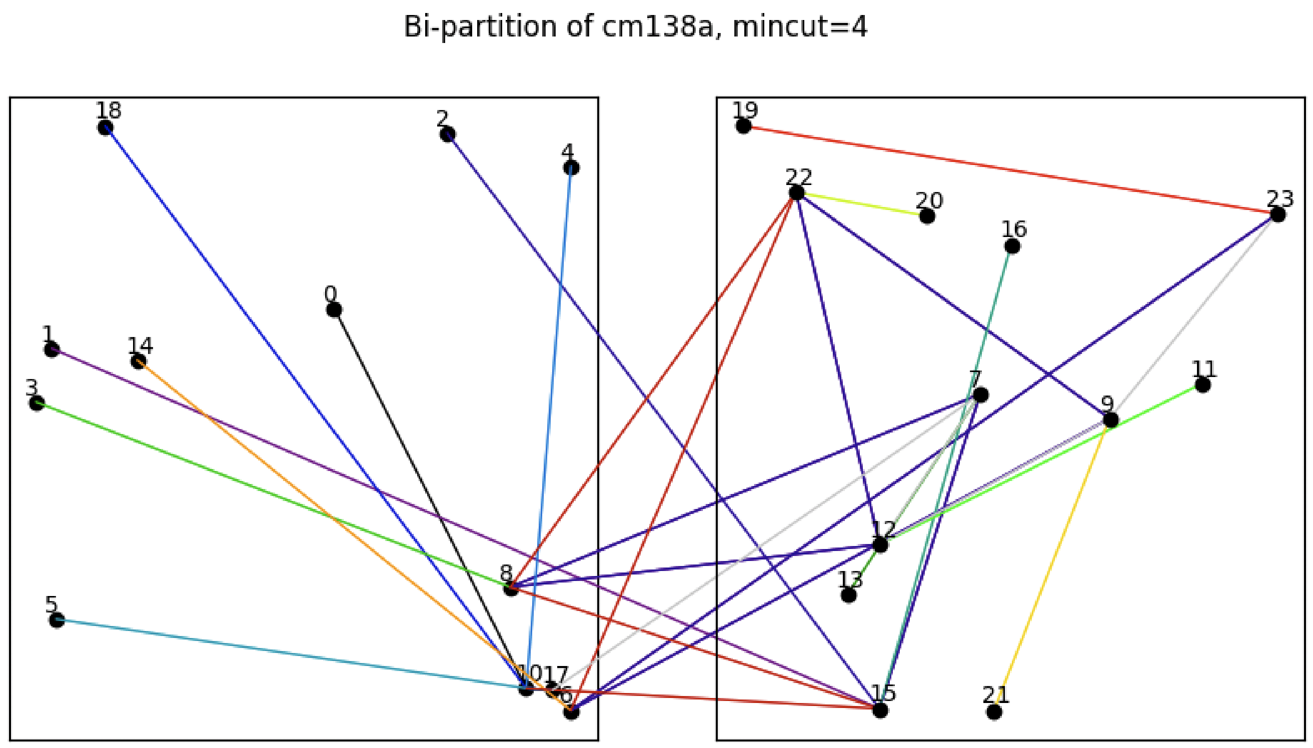
^ multiple overlaps



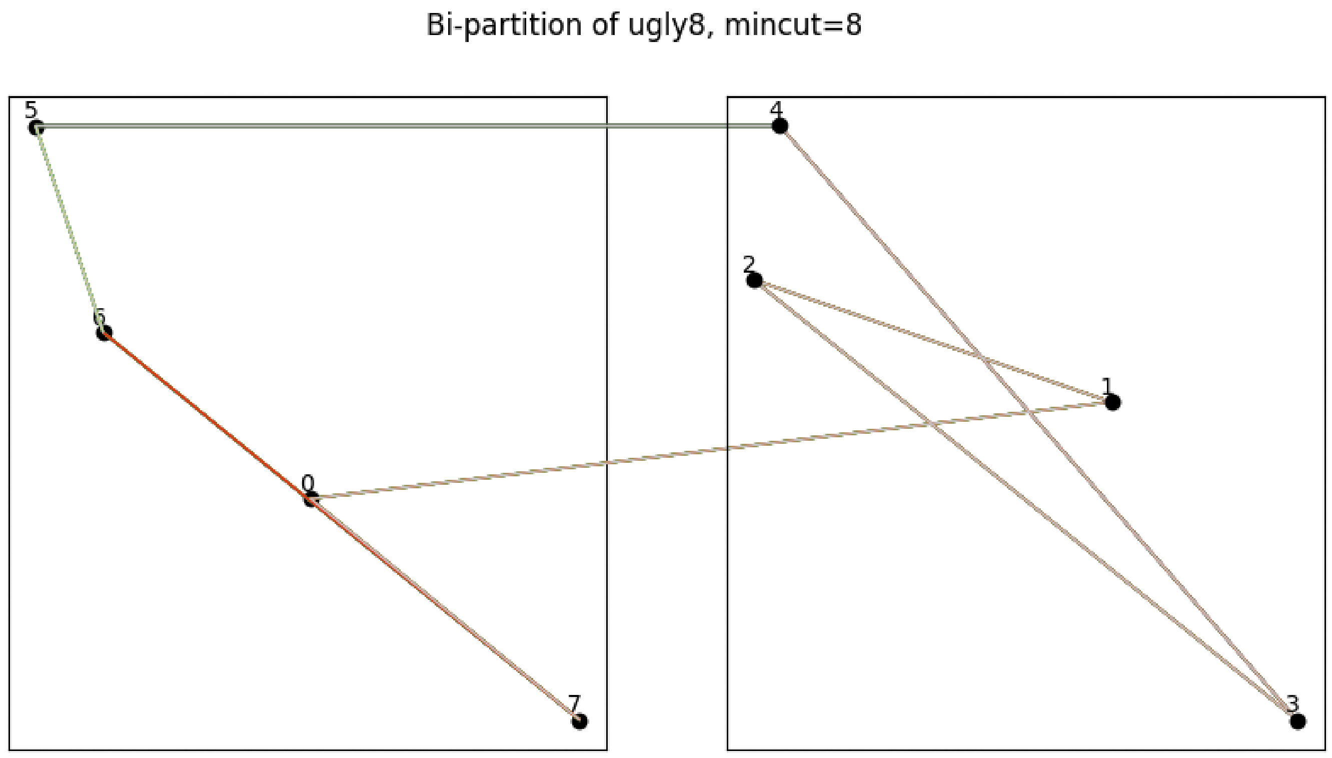
^ one crossing



^ 6 different colours of crossings



^ 4 different colours of crossings



^ multiple crossings overlap

Appendix B: Source code
Source code and documentation can be found here: https://github.com/mchuahua/CPEN513/tree/master/asn3

Initiatives 1 + 2 are in initiatives.py
Initiatives 1 + 2 + 3 + 4 are in initiatives2.py

Appendix C: Final assignments
Format: [cell, left/right]

z4ml
```
[[0, 0], [1, 1], [2, 0], [3, 0], [4, 1], [5, 0], [6, 0], [7, 1], [8, 0], [9, 1], [10, 0], [11,
1], [12, 0], [13, 0], [14, 1], [15, 1], [16, 0], [17, 1], [18, 1]]
best cost: 3
```

con1
```
[[0, 0], [1, 1], [2, 0], [3, 0], [4, 0], [5, 1], [6, 0], [7, 0], [8, 1], [9, 0], [10, 1], [11,
1], [12, 1], [13, 1]]
best cost: 4
```

ugly16
all permutations best cost 16

ugly8
all permutations best cost 8

cm138a
```
[[0, 0], [1, 1], [2, 1], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0], [8, 0], [9, 1], [10, 0], [11,
1], [12, 1], [13, 0], [14, 0], [15, 1], [16, 1], [17, 0], [18, 0], [19, 1], [20, 1], [21, 1],
[22, 1], [23, 1]]
best cost: 4
```

cm82a
```
[[0, 0], [1, 0], [2, 0], [3, 1], [4, 0], [5, 1], [6, 0], [7, 1], [8, 1], [9, 1], [10, 1], [11,
0]]
best cost: 1
```

cm162a
```
[[0, 0], [1, 1], [2, 0], [3, 0], [4, 0], [5, 0], [6, 0], [7, 0], [8, 0], [9, 1], [10, 1], [11,
1], [12, 1], [13, 1], [14, 0], [15, 1], [16, 0], [17, 1], [18, 1], [19, 0], [20, 1], [21, 0],
[22, 0], [23, 0], [24, 0], [25, 0], [26, 1], [27, 1], [28, 1], [29, 1], [30, 1], [31, 1], [32,
0], [33, 1], [34, 0], [35, 1], [36, 0]]
best cost: 6
```

cm150a
```
[[0, 0], [1, 1], [2, 0], [3, 1], [4, 0], [5, 1], [6, 1], [7, 1], [8, 1], [9, 1], [10, 0], [11,
0], [12, 0], [13, 0], [14, 1], [15, 1], [16, 1], [17, 1], [18, 0], [19, 0], [20, 0], [21, 0],
[22, 0], [23, 0], [24, 1], [25, 1], [26, 0], [27, 1], [28, 0], [29, 1], [30, 0], [31, 1], [32,
0], [33, 1], [34, 1], [35, 0]]
best cost: 6
```

cc

```
[[0, 0], [1, 0], [2, 0], [3, 1], [4, 0], [5, 0], [6, 0], [7, 0], [8, 1], [9, 1], [10, 0], [11,
1], [12, 0], [13, 0], [14, 1], [15, 1], [16, 1], [17, 0], [18, 1], [19, 1], [20, 0], [21, 0],
[22, 1], [23, 1], [24, 1], [25, 1], [26, 1], [27, 1], [28, 0], [29, 0], [30, 1], [31, 0], [32,
0], [33, 0], [34, 0], [35, 0], [36, 1], [37, 0], [38, 1], [39, 1], [40, 1], [41, 1], [42, 1],
[43, 1], [44, 0], [45, 0], [46, 0], [47, 1], [48, 1], [49, 1], [50, 0], [51, 1], [52, 0], [53,
1], [54, 0], [55, 0], [56, 1], [57, 0], [58, 0], [59, 0], [60, 1], [61, 1]]
best cost: 4
```

twocm

```
CPU time limit exceeded (core dumped)
```