Reusing Random Vectors in the Cut Matching Game

Lucas Kellar

University of Michigan, Ann Arbor, MI

1 INTRODUCTION

The Cut Matching Game is an algorithmic framework introduced by Khandekar, Rao, and Vazirani [5] that aims to find a ϕ-sparse cut[1] in a graph or certify it as a ϕ-expander. [2]

To run efficiently, the Cut Matching Game relies upon random projection to represent its internal state in near-linear time. Generating the random vectors for projection (and keeping them up-to-date) contributes a significant amount to the overall runtime asymptotically.

In this paper, we experimentally explore whether reusing random vectors for projection has a significant effect on the algorithm's efficiency compared to the asymptotic runtime savings it provides. We'll first provide an overview of the Cut Matching Game, then detail the modification we attempt to make.

2 CUT MATCHING GAME

At a high level, the game is processed in a series of rounds. In each round, the "Cut Player" will generate some cut in the graph, aiming to expose a sparse cut if it exists. Given the cut, the "Matching Player" will attempt to produce a perfect matching, mapping each node in the cut to a unique node outside the cut. This matching represents a flow comprised of unit-flow paths between each matched pair that routes $n/2$ flow with congestion $\frac{1}{\phi}$. If the matching player can produce such a matching $r = \log^2(n)$ times (assuming the cut player is using an optimal strategy), we can say with high probability that G'_r (the union of all the matchings) is a $\frac{1}{2}$-expander that we can embed within G with $r \cdot \frac{1}{\phi}$ congestion.

If during any of the rounds, the matching player cannot route $n/2$ flow, the cut provided is ϕ-sparse. [5] Execution stops, and the sparse cut is returned. The high-level algorithm is described in Algorithm 1. We'll explore the cut and matching stages in more detail below.

Algorithm 1 Cut Matching Game

```
1:  $M \leftarrow \{\}$
2:  $G_0 \leftarrow (V, \emptyset)$
3:  for each $i \in 1..\log^2(n)$ do
4:      $S \leftarrow$ CutPlayer$(G, M)$
5:      $(M_i, \text{flow}) \leftarrow$ MatchingPlayer$(M, S)$
6:      if flow $< n/2$ then
7:          return $S$                                  ▷ Return $\phi$-sparse cut
8:      end if
9:      $M \leftarrow M + M_i$
10:     $G_i = \leftarrow G_{i-1} \cup M_i$
11: end for
12: return $\emptyset$                 ▷ No sparse cut found. Graph is a $\phi$-expander
```

2.1 Cut Player

The cut player is aiming to expose a sparse cut if one exists or certify that an expander can be embedded in G with $r \cdot 1/\phi$ congestion in the fewest number of rounds possible. At a high level, the optimal strategy for the cut player is to separate nodes that are "far away" from each other. By far away, we mean that we aim to select nodes pairs that are many hops away from each other. If the graph is well connected,

[1]A ϕ-sparse cut is a cut $S \subset V$ where capacity$(S, V \setminus S) < \phi \min(|S|, |V \setminus S|)$. There exist other node weightings that weight nodes differently to change the definition of a sparse cut.

[2]A ϕ-expander is a graph where no ϕ-sparse cuts exist

routing flow between the pairs causes no problems, but if the graph is not well connected, then routing flow between these pairs will encounter some bottleneck, and the maximum flow will be less than targeted. See Figure 1 for a comparison between an optimal and poor cut (nodes in the cut marked red) on a barbell graph (two cliques connected by a single edge). The optimal cut isolates one of the cliques from the other, so a flow across the cut will be bottlenecked by the single edge, which will expose the sparse cut. In the poor cut, we can easily route flow between nodes inside and outside the cut, so we learn nothing about the sparse cut.

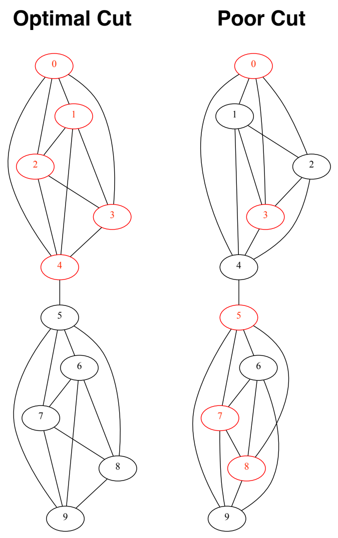

Figure 1. Optimal vs Poor cut for exposing a sparse cut

To represent the relative position of the nodes, we imagine an n-dimensional space where each node starts out with distance 1 in its specified dimension and 0 in all others. During each matching, the average position of each node pair is calculated and assigned to both nodes in the pair (to bring them to the same position). After many matchings, each node will be relatively close in position to each of the nodes it has matched with. We can then select half of the closest nodes in one direction for our cut. This forces the matching player to construct flows between pairs of nodes relatively “far away” from each other, resulting in a greater chance of exposing a sparse cut.

However, this is too slow to be practical. Each node will have n dimensions, so even representing the position matrix would take $O(n^2)$ time and space. Instead, we project the position of each node onto a random vector, generally preserving the relative distance between nodes (because squared distance is preserved during projection with high probability).

Specifically, we generate a random vector of size n and assign each node a random position. For each matching and matched pair, we assign each node in the pair the average position of the pair. We then find the median position and select every node with position less than or equal to the median position for our cut. Relative distances are preserved during projection, and a new random direction is selected each round, so this provides a high-quality approximation in much faster time. See Algorithm 2 for a high-level representation of the cut step.

Algorithm 2 Cut Player

```
1: $r \leftarrow \textsc{RandomVector}(n)$
2: for each $M_i \in M$ do
3:     for each $v_0, v_1 \in M_i$ do
4:         $avg \leftarrow (v_0 + v_1)/2$
5:         $r[v_0] = avg$
6:         $r[v_1] = avg$
7:     end for
8: end for
9: $x : |\{v|r[v] \leq x\}| = |\{v|r[v] > x\}|$        ▷ Identify the median element
10: $S \leftarrow \{v|r[v] \leq x\}$
11: return $S$
```

2.2 Matching Player

Given a cut $S \subset V$, the matching player is tasked with constructing a perfect matching between S and $\bar{S}$. The general idea is to attempt to route flow between S and $\bar{S}$ and construct a matching from the flow. [5] S and $\bar{S}$ both contain $n/2$ nodes, so we attempt to route $n/2$ units of flow.

First, we construct the flow network we will attempt to route within. Each edge in G is assigned a capacity of $1/\phi$. We then create a "super-source" node and "super-sink" node and attach them with unit capacity to each node in S and $\bar{S}$ respectively.

We then run a max flow algorithm on the network. If we can get a max flow of $n/2$, then we decompose the paths in the flow to generate a perfect matching between S and $\bar{S}$ (because each unit of flow sent through some $u \in S$ will end up at some unique $v \in \bar{S}$). If we get a max flow value less than $n/2$, then there must exist some ϕ-sparse cut between S and $\bar{S}$.

Recall that there exists a ϕ-sparse cut if there is a cut $S \subset V$ where capacity$(S, V \setminus S) < \phi \min(|S|, |V \setminus S|)$. We know the size of both $|S|$ and $|V \setminus S|$ is $n/2$ by construction. We've also scaled each edge by $1/\phi$ so we can show that if max flow is less than $n/2$, there exists a ϕ-sparse cut in G:

$$\begin{aligned} \frac{1}{\phi}\,\text{capacity}(S, V \setminus S) &< n/2 \\ \text{capacity}(S, V \setminus S) &< \phi \cdot n/2 \\ \text{capacity}(S, V \setminus S) &< \phi \min(|S|, |V \setminus S|) \end{aligned} \tag{1}$$

If this case occurs, we stop execution and return the cut. Otherwise, if we achieve max flow $= n/2$, we decompose the flows and return the matching. See Algorithm 3 for a high-level representation of the cut step.

Algorithm 3 Matching Player

```
1: Set each edge $e \in G$ to capacity $1/\phi$.
2: Add $G' = G \cup v_{\text{source}} \cup v_{\text{sink}}$        ▷ Add super-source/super-sink nodes to $G$
3: for each $u \in S$ do
4:     Add edge $(v_{\text{source}}, u)$ with capacity 1
5: end for
6: for each $u \in V \setminus S$ do
7:     Add edge $(v_{\text{sink}}, u)$ with capacity 1
8: end for
9: $f \leftarrow \textsc{MaxFlow}(G')$
10: if $|f| < n/2$ then
11:     return $S$        ▷ Return $\phi$-sparse cut
12: end if
13: $M_i \leftarrow \textsc{DecomposeFlow}(f)$
14: return $M_i$
```

3 REUSING RANDOM VECTORS

While we save time during the Cut stage by projecting the position of the nodes to a random vector, we still incur a non-negligible runtime. During each round i, we generate a random vector of size n in $O(n)$-time. We then apply each of the $i-1$ matchings to our vector, each of which contains $O(n)$ pairs to assign. If we run for $O(\log^2(n))$ rounds, the last round will take $O(n)+O(n)\cdot O(\log^2(n))$ time to generate the projection vector.

After each round, we get a total runtime of approximately $O(\log^2(n))\cdot O(n\cdot\log^2(n)) = O(n\cdot\log^4(n))$ time. Intuitively, each round, we must generate a fresh random vector and have it "catch up" with all the previous matchings.

If we were to use only some constant $k << \log^2(n)$ random vectors, we could reuse them throughout the rounds and reduce the total runtime to $O(n\cdot k\cdot\log^2(n))$ time. Specifically, assume we have k random projection vectors. Then, on round i, we just use vector i mod k to project upon.

Assume for simplicity we generate all k random vectors up front before any of the rounds begin. This will take time $O(nk)$. After each round, when the matching is received, we apply the matching to each random vector in time $O(nk)$. After $\log^2(n)$ rounds, we've spent $O(nk\cdot\log^2(n))$ time maintaining our random vectors, which is a substantial improvement as long as $k << \log^2(n)$.

As a sidenote, in cases where we expect to find sparse cuts in less than k rounds frequently, we could adopt an alternative strategy where in the first k rounds we generate the needed random vector, "catch it up" (apply the matchings), and then store it in the cache. After each round, we apply the received matching to each vector generated so far. When the game goes beyond k rounds, the effect and runtime is the same, but if the game terminates in $r < k$ rounds, we'll only spend $O(n\cdot r^2)$ time managing projection vectors. For simplicity, we'll ignore this case for now and assume we generate all k vectors up front.

We'll explore this idea experimentally, by constructing an implementation of the Cut Matching Game and comparing the number of rounds required to find sparse cuts with and without this random vector reuse system.

4 IMPLEMENTATION

We begin to discuss our implementation of the Cut Matching Game and modifications made for the experiment. First, we briefly look at some previous implementations.

4.1 Related Work

Ludwik Janiuk's implementation of the Cut Matching Game [4] is one of the earlier public implementations of the framework available. It heavily relies upon the LEMON graph framework [2] for managing the graphs and computing max flow (as well as other related calculations).[3] Janiuk's implementation generally follows the structure of the original framework introduced by Khandekar, Rao, and Vazirani [5] but additionally takes influence from an expander decomposition algorithm by Saranurak and Wang [6] that focuses more on an edge-based version of the game. This technique subdivides the graph before the game begins, converting every edge into a "split node", replacing an edge (u,v) with edges (u,w) and (w,v) where w is a new node with no other edges.

Additionally, the Janiuk implementation uses a different technique for the cut. Instead of generating a random vector, each node is randomly assigned a value of $\frac{1}{n}$ or $-\frac{1}{n}$. Matchings are then applied as usual (by averaging values). Janiuk also uses LEMON's implementation of the Push-Relabel algorithm, which is the fastest max-flow algorithm included with a stated runtime of $O(n^2\sqrt{m})$ [3].

4.2 Our Implementation

Our implementation[4] takes inspiration from the Janiuk implementation but diverges in a few aspects:

- **Not using LEMON**: For simplicity, we used pure C++, implementing our own graphs (via adjacency lists) and max flow algorithms. This allowed for a less complex program, easier debugging, and better visibility, especially because we weren't aiming to be as ambitious as other implementations

[3] At the time of this writing, I was unable to get LEMON compiled and so unfortunately couldn't run Janiuk's implementation as a comparison.

[4] The implementation can be found on Github at https://github.com/lkellar/cse-598-cut-matching-game

- **Edmonds-Karp for Max Flow**: We implemented both Edmonds-Karp[1] and Push-Relabel algorithms for max flow, but at the time of this writing, the Edmonds-Karp algorithm was more performant than the Push-Relabel implementation. Further work could be done to enhance our Push-Relabel implementation. However, we were able to extract the matching from the Edmonds-Karp algorithm directly, eliminating the need for an additional flow decomposition step
- **Not using subdivision**: The subdivision process was initially implemented before the game, but we found that it didn't seem to affect our use case of the game, and it increased complexity (by adding more edges). The subdivision code is still present in the implementation, but is currently disabled.

In addition to these differences, there are some other highlights of note. We implement the cut generation step exactly like the original framework [5] and have an option to enable random vector caching. Specifically, when a number of random vectors k is provided, we generate them ahead of time and then update all the vectors with the matching generated after each round.

We also use the CHACO format for graphs described in Chris Walshaw's JOSTLE user guide [8], which allows us to easily test our algorithm on Walshaw's Graph Partitioning Archive [7].

At this time, our algorithm only supports unweighted graphs and the 1_V node weighting (where each node is weighted equally). Additionally, for brevity, the algorithm doesn't report the details of the cut when found, but just outputs that one was found. Future work could expand to include these features without too much additional effort.

4.2.1 Edmonds-Karp Matching Extraction

When no sparse cut exists, the matching player must verify $n/2$ flow can be routed across the cut with congestion at most $1/\phi$ and then decompose the resulting flow to find a perfect matching between nodes in the cut and nodes outside of the cut.

However, we note an interesting property that allows us to skip the flow decomposition stage when edges are unweighted and every node has equal weight. Consider the following graph in Figure 2 with the cut $S = \{0,1,2,3,4\}$ and $V \setminus S = \{5,6,7,8,9\}$. In this scenario, we have a $\phi = \frac{1}{2}$, so all original edges in G are set to capacity $\frac{1}{\phi} = 2$. We then connect nodes in the cut to a super source s and nodes outside the cut to a super sink t with unit capacity edges.

Running a max flow algorithm could give us a resulting flow as in Figure 2, where the source is able to successfully send $n/2 = 5$ units of flow to the sink. However, because each edge from super source/sink has capacity 1, each resulting edge between node and super source/sink will handle exactly 1 flow path (and no flow path will traverse more than one of each type of edge). We can then say that each flow path uniquely passes through one super-source / node edge and one super-sink / node edge. If we were able to recover these unique pairs, we could use this as our matching (and that's what typical flow decomposition does).

In the Edmonds-Karp max-flow algorithm [1], augmenting paths are found one by one. Once an augmenting path is found, each edge traveled by the path is updated (and if an edge becomes at capacity, future paths will not try to traverse the edge in that direction). Therefore, along standard processing of the augmenting path, we will encounter these edges between super source/sink and nodes.

In our implementation, we modify the algorithm to mark the nodes attached to these edges and cache them until max flow is found. This can be done efficiently with a C++ unordered_map with little overhead. If the algorithm can successfully route $n/2$ flow, we just use this cache as our matching. In our testing, this resulted in no detriment in the number of rounds needed to find cuts and meaningfully sped up running time by skipping the flow decomposition step (as well as reduced complexity).

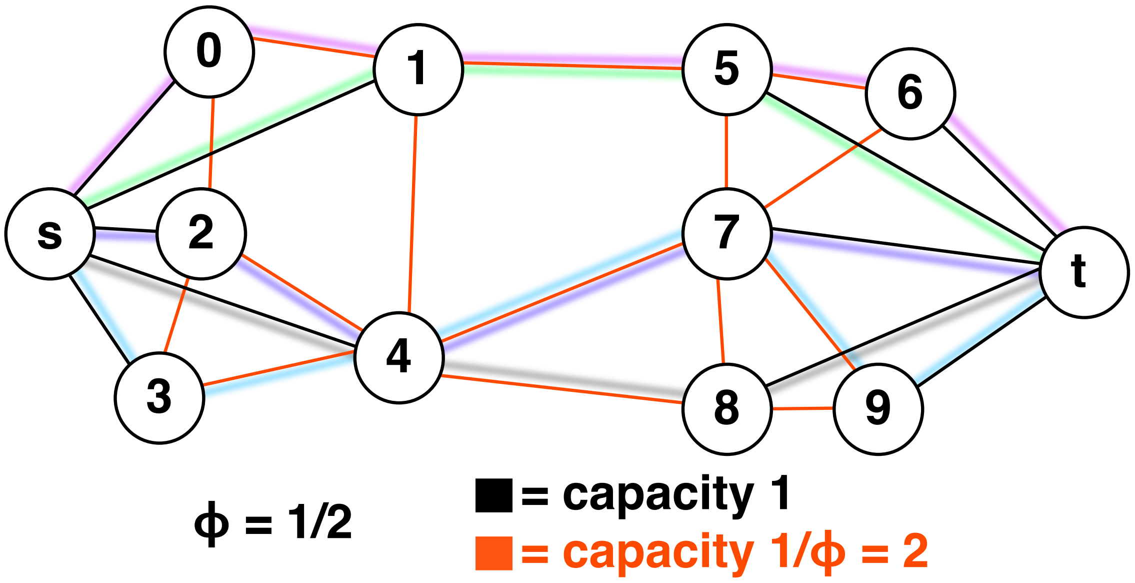

Figure 2. Example flow

5 RESULTS

Below we present two selected result tables from running on Walshaw's partitioning archive [7]. Unfortunately, we were only able to feasibly test constants up to $k = 80$. It really only makes sense to test constants where random vectors will be reused, but due to compute constraints, we could only feasibly test iterations that ran up to around $150 - 200$ rounds on average.

First, we tested small graphs and kept the number of random vectors constant k low. We selected a target ϕ by trying to get almost (but not exactly) as large of a ϕ as we could to maximize the average rounds. This both allowed us to determine the expansion parameter for each graph in our dataset and gave us the most rounds to work with, increasing random vector reuse. See details in Figure 3.

In the table, the columns with k headers indicate the number of average rounds that the algorithm took to complete when using at most k random vectors. $k = \infty$ means there was no reuse of random vectors. Additionally, the program has a default round setting of $r = \log^2(n)$, but we allowed it a maximum of $3r$ iterations for the testing. Entries marked N/A represent tests that failed to find the cut in $3r$ rounds more than 20% of the time.

Benchmark	Nodes	Edges	Target $\frac{1}{\phi}$	Iterations	$k = \infty$	$k = 5$	$k = 10$	$k = 20$
add20	2395	7462	6	50	21	29	20	18
uk	4824	6837	175	50	59	99	67	45
add32	4960	9462	175	50	51	N/A	N/A	49
3elt	4720	13722	14	20	105	103	108	102

Figure 3. Small Graph Testing

We then test some larger graphs that take more rounds for our original algorithm to process. These allow us to test some larger constants for vector reuse. It's worth noting that smaller constants than those listed (such as $5, 10, 20$) did not tend to cross the inclusion threshold (80% success rate) enough to be recorded. Unfortunately, due to compute constraints, we weren't able to run benchmarks significantly larger than these. See details in Figure 4.

Benchmark	Nodes	Edges	Target $\frac{1}{\phi}$	Iterations	k = ∞	$k = 40$	$k = 80$
whitaker3	9800	28989	20	10	124	N/A	N/A
crack	10240	30380	15	10	79	99	71
cti	16840	48232	15	10	119	132	129

Figure 4. Large Graph Testing

6 CONCLUSION

6.1 Preliminary Findings

Based on the data we've found, it appears that when k (number of random vectors generated) is large enough that each random vector will be reused only a couple of times, the effect on the number of rounds isn't too severe. However, if k is kept small and the number of rounds needed increases, we tend to see rapid increases in the rounds required to complete (if it completes at all).

It's worth noting that we were only able to collect a relatively small amount of data, due to the compute restraints on our algorithm. Additionally, no matter the compute available, no amount of testing would be enough to formally prove the effect of caching random vectors, though larger datasets would provide a better picture.

However, based on the limited data and testing we've completed so far, it appears that there most likely does not exist any constant k that will allow the Cut Matching Game to complete in a comparable number of rounds as n and the baseline number of rounds grows. Using a k value proportional to the size of the graph may be safe, but it would cause the space required to grow, and would not reduce the asymptotic running time of the algorithm, so it most likely is not worth the tradeoff.

6.2 Future work

Aside from the compute constraints, the current iteration of the project also has some areas that could be improved to support future work

- **Support weighted graphs**: Out of simplicity, the implementation only processes unweighted graphs. Support for weighted graphs would require some changes in how edges are scaled during the matching phase.

- **Support different node weightings**: The implementation only considers the 1_V node weighting where every node is weighted equally. Support for other node weightings would require more node data overhead and different logic throughout the cut and matching steps.

- **Use Push-Relabel max flow**: At the moment, Edmonds-Karp is the most performant max flow we've implemented, despite Push-Relabel having better asymptotic performance. Work can be done to improve the efficiency of our Push-Relabel implementation.

- **Report the cut**: Right now, the algorithm doesn't report the cut it found before terminating. A very simple change could result in the cut being output when found.

The decision to forgo a graph framework makes some of these changes more difficult and tedious (such as manually tuning a Push-Relabel algorithm when off-the-shelf solutions exist), but the upside of the pure C++ approach allows for some improvements, like being able to use Edmonds-Karp directly to provide a matching, as well as a better understanding of the code and a more direct interface to the algorithm.

Improvements to this implementation aside, there is future work to be done in determining the efficacy of random vector reuse. While our limited dataset implies that a constant number of random vectors for all n is unlikely, further experimentation on larger datasets could provide confidence. On the other hand, a formal mathematical proof would provide a much stronger proof of the efficacy and would likely be a more fruitful path to take.

REFERENCES

[1] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, Apr. 1972.

[2] Egerváry Research Group on Combinatorial Optimization. LEMON, 2014.

[3] Egerváry Research Group on Combinatorial Optimization. LEMON: Preflow Class Reference, 2014.

[4] L. Janiuk. Cut Matching Game, July 2010.

[5] R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. *Journal of the ACM*, 56(4):1–15, June 2009.

[6] T. Saranurak and D. Wang. Expander decomposition and pruning: Faster, stronger, and simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2616–2635, 2019.

[7] A. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Journal of Global Optimization*, 29(2):225–241, June 2004.

[8] C. Walshaw. JOSTLE executable user guide, July 2005.