Approaches to Identifying Kernels in Directed Graphs

Colin McClelland
CPSC 450
Fall 2024

Summary

In graph theory, a kernel exists in a directed graph if there exists a subset of vertices such that no pair of vertices in the set have an edge between them (independent), and every vertex not included in the set has an edge to a vertex in the set (dominating). The problem of finding kernels in a digraph is NP-Complete, but for graphs having certain properties, the problem can be solved in linear time relative to the size of the graph. In this work, I focus on finding kernels using three different approaches: a brute-force approach, a naive depth-first search approach, and an optimized approach that attempts to exploit the properties of the graph to find a kernel in linear time. The results of performance tests are discussed in section 3. The naive and optimized approaches improve upon the brute force approach by order of magnitude. The Java language was used to implement all three approaches, unit tests, and performance tests.

1. ALGORITHMS USED

1.1 Brute Force Approach

The brute force method evaluates all possible subsets of vertices to determine if any form a valid kernel. For each subset, the conditions of independence and domination are checked. Although guaranteed to find a solution if one exists, this approach is highly inefficient due to the exponential number of subsets.

Algorithm: **findKernelBruteForce**
Input*: A graph G = (V, E)*
Result*: kernel if one exists, otherwise empty set*

```
1)  Begin:
2)    C = all possible combinations of vertices
3)    for combination c ∈ C
4)      if c is independent and dominating
            return c
5)    return empty set
```

For a graph with V vertices, there are 2^V possible combinations of vertices. For each combination it is necessary to check the conditions of a valid kernel. The independence check requires that all vertices in the given combination be compared to all other vertices in the kernel. The domination checks require that every vertex outside of the kernel be compared to every vertex in the kernel. These operations give the brute force approach an overall complexity of $O(2^V \cdot V^2)$, making it impractical for even graphs with relatively few vertices.

1.2 Naive DFS Approach

The domination criteria for a kernel require that every vertex with no out-edges be included in the kernel. As such, these vertices are added to the kernel prior to traversal. The naive depth-first search (DFS) approach performs a full DFS from each node in the graph in attempt to find a valid kernel. A single pass of full DFS yields a *candidate kernel*, which is a subset of vertices that is independent, but may not be dominating. After each pass of full DFS it is necessary to check that each vertex excluded from the kernel has an edge to a vertex in the kernel.

Algorithm: **findKernelNaive**
Input*: A graph G = (V, E)*
Result*: subset of vertices K if a kernel exists, otherwise empty set*

```
1)  Begin:
2)    for vertex v ∈ V
3)      K = empty set
4)      add sinks to K
5)      exclude vertices adjacent to sinks from K
6)      perform full DFS starting at v
7)        if discovered vertex v_new is not excluded from K
8)          add v_new to K
9)      if all excluded vertices have an edge to K
10)       return K
11)   return empty set
```

The naive approach adds a given vertex to the candidate kernel if it does not neighbor a vertex that is already in the kernel, ensuring that the candidate kernel is independent. The check for domination must be done after the candidate is constructed because it is not possible to know if all vertices outside the candidate kernel are dominated by it. The search continues until a subset of vertices is found if one exists.

For any graph, there are |V| passes of full DFS in the worst case. On each iteration additional work is required beyond the DFS logic: finding the sinks and their adjacent vertices is O(V+E), however in the implementation, this is computed once and stored for restoring K if it fails the check for domination. Adding v_{new} to the kernel and its edges to the excluded set is O(V+E). The check for domination also has a cost of O(V+E). The overall cost of the algorithm is O(V(V+E)), which represents a significant improvement over brute force approach.

1.3 Optimized DFS Approach

The optimized approach also relies on a full DFS from every vertex in the kernel, however prior to traversal this approach attempts to exploit properties of the graph that can be computed in linear time. These checks include topological sort and a bipartite check. If the graph has a valid topological sort, then the graph must have a kernel All graphs that are two colorable must also contain a kernel. In the case of graph having neither a topological sort nor being bipartite, it becomes necessary to resort to the method described in section 1.2.

Algorithm: **findKernelOptimized**
Input*: A graph G = (V, E)*
Result*: subset of vertices K if a kernel exists, otherwise empty set*

```
1)  Begin:
2)      K = empty set
3)      if G has a topological sort T
4)          for vertex v ∈ reverse(T)
5)              if v is not excluded from K
6)                  add v to the kernel
7)                  exclude vertices adjacent to v from K
8)          return K
9)      if G is bipartite
10)         perform a single pass of full DFS from any vertex
11)             return K;
12)
13)     return findKernelNaive(G)
```

If a graph has a topological sort, or is two-colorable, then it contains a kernel that can be found in linear time. Obtaining a topological sort has a time cost of O(V+E). Once the order is obtained, the cost to iterate the graph is also O(V+E), meaning the overall cost for this case is O(V+E). Similarly, the cost to determine if the graph is bipartite is O(V+E) and requires only a single pass of full DFS if true, making the total cost of this case O(V+E). Despite the linear time cost for certain graphs, the overall cost remains O(V(V+E)) in the worst case.

2. IMPLEMENTATION

For all of the listed approaches, the kernel is represented as a Java HashSet. The excluded set that is used to ensure independence is maintained as a boolean array, which provides constant time lookups and assignments. For the full DFS traversal, an iterative implementation of DFS is used. The graphs used in testing are all implemented as adjacency matrices for their constant time access of edges, which is important for checking the dominating criterion. No external libraries were used for this work.

A thorough set of nearly one-hundred unit tests were developed to ensure a working level of functionality for several variations of directed graphs. Positive test were created to confirm that the algorithms correctly identified a kernel if one was present, and negative tests were used to confirm that the algorithms did not erroneously detect a kernel when one was not present.

Tests were created for all three approaches to identify a kernel, ensuring that each was able to identify a kernel for the same graph. In many cases, a graph was altered in a minimal way such its resulting kernel was changed, but many of its edges and vertices remained unchanged. Tests with subtle changes that yielded a new kernel ensure that the algorithms identify the correct result.

Figure 1 shows a vertex which does not have a topological sort, nor can it be bipartite. In this case vertex 7 is a sink and is added prior to the start of full DFS along with its adjacent vertices being added to the exclusion set. A DFS traversal beginning at 0 adds 0 to the kernel then excludes vertices 1 and 2 as to not violate the constraint of independence. Vertex 1 is examined per DFS traversal, but it has been excluded. Figure 2 shows a graph with three possible kernels, all of which are correct results of the algorithms, but only one is returned. Figure 3 shows a star graphs which can not contain a kernel, and as such we assert that the empty set are returned. Figure 4 shows a modified version of the graph in figure 1. Additional vertices have been added along with corresponding edges, which result in a new kernel.

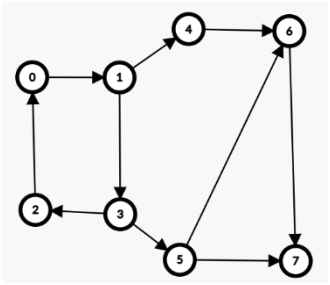

Figure 1. Eight Vertex cyclic graph containing kernel {0,3,4,7}

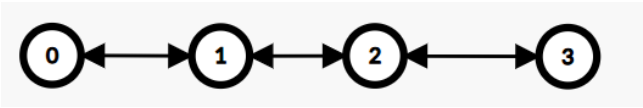

Figure 2. Graph containing kernels {0,2}, {0,3}, {1,3}

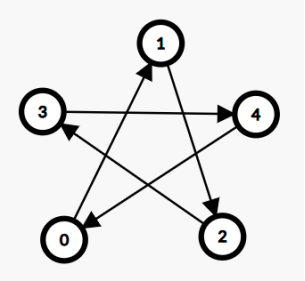

Figure 3. Graph containing no kernels

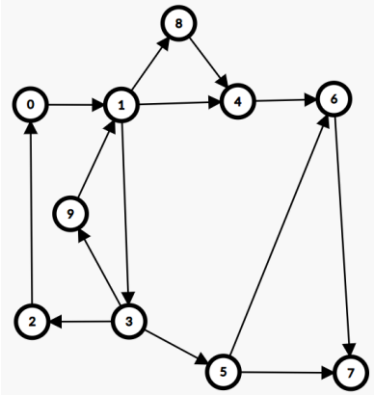

Figure 4. Modified figure 1 graph containing kernel {0,4,7,9}

3. PERFORMANCE TESTS

Performance testing evaluations were conducted between the brute force approach and the naive approach. As well as between the naive and topological approaches. Smaller steps of the graphs used in performance testing are seen in figure 5 and figure 6. The graphs continue to grow larger with the same pattern such that performance differences may eventually be observed.

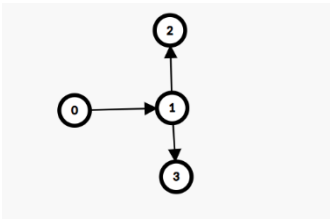

Figure 5

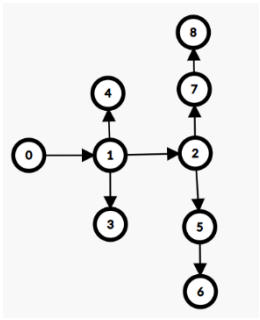

Figure 6

To adequately compare the differences between the naive an optimized implementation, all of the graphs including those seen above contain a topological sort order and a bipartite.

All performance testing occurred on Windows 11 OS, on a machine equipped with an 11th Gen Intel Core i7 processor running at 2.90GHz and 16.0 GB of RAM. The system is a 64-bit operating system, x64-based processor.

4. EVALUATION RESULTS

Figure 7 shows the difference between the brute force approach and the naive full DFS approach. The brute force algorithm runs into significant performance slowdown after the graph reaches 10 nodes. The brute force algorithm could not process a graph larger than 11 nodes, largely in part due to its recursive implementation and the exponential number of calls made. Many more checks on the conditions of independence and domination must be performed.

Figure 8 shows the performance difference between the naive implementation and the optimized implementation. The major performance difference comes from the fact that the graphs used in the performance tests have a topological sort (the check for topological occurs before bipartite) so kernels are found within these graphs in linear time, whereas the naive requires potentially many DFS traversals and additional checks.

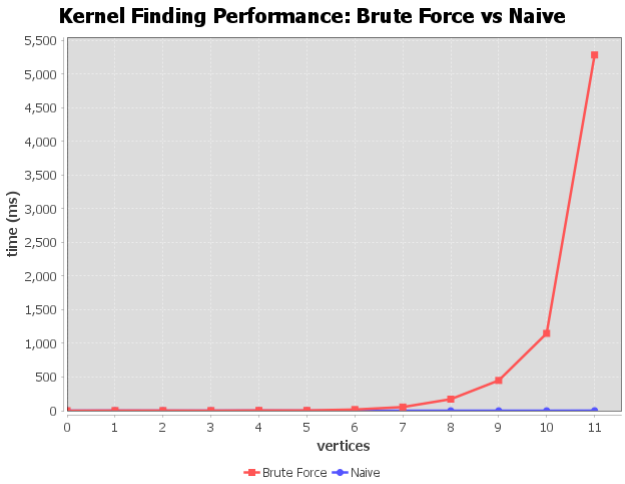

Figure 7: Brute Force vs Naive

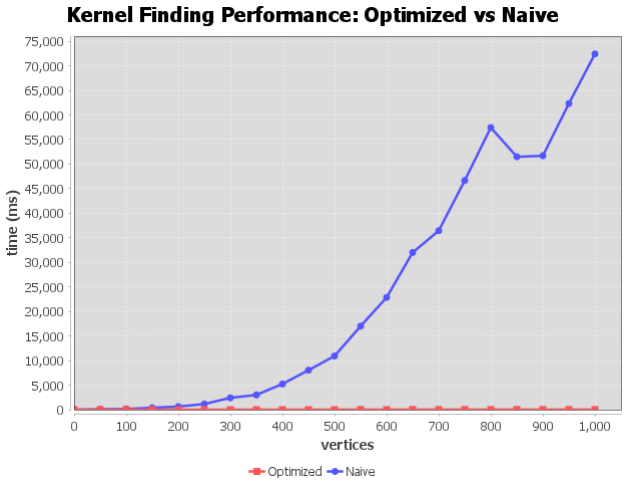

Figure 8: Naive vs Optimized

5. REFLECTION

I found that the problem of finding kernels in a digraph reinforced the notion of NP-Completeness. For this problem the candidate keys are a potential certificate that may satisfy the conditions of a kernel and can be verified in PTIME…O(V+E). I can imagine this problem being run on a non-deterministic Turing Machine or simulating on a deterministic TM. I also appreciated how this algorithm was able to reuse algorithms, which did not guarantee a satisfying result, but allowed for significant performance increase if present. The most challenging aspect of this problem was attempting to identify the properties of a graph that would allow for a linear time best case. I am sure that there are more than were captured in this implementation. If I had more time I would attempt to define more of these properties, or identify an optimal traversal of the graph, which would not lower the worst-case time cost, but could certainly reduce the cost in the average case.

6. RESOURCES

There is not an extensive number of resources available for the problem of finding kernels in a digraph. The resources that are available tend to describe an applied technique that is useful for a restricted subset of graphs where a kernel can be found quickly, with less emphasis being placed on the general case. The resources were helpful with familiarizing myself with the problem and employing the method of identifying certain graphs that can have a linear time result.

7. REFERENCES

[1] Bruno Jartoux. 2024. *Counting kernels in directed graphs with arbitrary orientations*. Discrete Applied Mathematics, 355:62–73. Elsevier B.V., doi:10.1016/j.dam.2024.04.018.Second reference, etc.

[2] Aviezri S. Fraenkel. 1996. *Combinatorial game theory foundations applied to digraph kernels*. *Electronic Journal of Combinatorics*, 4(2), 1-17. DOI: 10.37236/1325.