Database Systems Term Project Report

Ishan Shah
The University of Texas at Austin

1 Introduction

Cypher is a graph query language that matches patterns of nodes and relationships to resolve queries. One of the most common graph query problems is path planning. Cypher has built-in subroutines to easy determine shortest paths, such as Dijkstra's single-source shortest path algorithm and the Floyd Warshall all-pairs shortest path algorithm. Cypher users can also add predicates to shortest path queries. For example, consider a graph representing the road network of Texas, where each edge is a road and each node is an intersection. A query might ask for the shortest path from Dallas to San Antonio that visits Austin. In this case, the shortest path is from Dallas to San Antonio, and the predicate is that Austin must exist in this path.

Unfortunately, when there is a predicate that needs to see the entire path before being evaluated, Cypher defaults to exhaustive search. In the Texas road network example, instead of using Dijkstra's shortest path algorithm, Cypher considers every possible path from Dallas to San Antonio, selects the paths that include Austin, then finds the minimum cost path of all of those paths. Exhaustive search is extremely time consuming, but there is a simple solution that can use Dijkstra's shortest path algorithm as a subroutine. Notice that the shortest path from Dallas to San Antonio that must visit Austin is simply the shortest path from Dallas to Austin concatenated with the shortest path from Austin to San Antonio. An optimized query can directly ask for the shortest path from Dallas to Austin, ask for the shortest path from Austin to San Antonio, and concatenate the two paths. The optimized query uses Dijkstra's shortest path algorithm twice as a subroutine, but since neither subroutine needs to satisfy a special predicate, the optimized query does not need to perform an exhaustive search, and can save a lot of time.

In this work, we present 20 different path-based queries where Cypher defaults to exhaustive search, and we present 20 optimized queries that speed up execution time using some knowledge of graph theory. Each query is either a single-source single-target shortest path query, or an all-pairs shortest path query. Several queries relate to finding the shortest path that must satisfy a predicate based on key-value pairs of different nodes and edges on the path. We discuss the optimization strategies used in each of the queries and show some performance metrics. We provide source code for each straightforward and optimized query at https://github.com/ishanashah/Cypher-Query-Optimizations.

2 Methodology

We evaluate our queries on an undirected 5-node clique and an undirected 6-node clique. The initialization code for those graphs is available in the source code. To evaluate queries that deal with the key-value pairs on nodes and edges, we assign a color or red or blue to every node and every edge. The key is *color* and the value is either RED or $BLUE$. We evaluate our queries on small graphs because the straightforward exhaustive implementations tend to time-out. Although we only evaluate on small graphs and use colors as key-value pairs, our work is easily extendable to any size of graph with any key-value pairs.

We use two optimization strategies. The first strategy is using an unconditional shortest path query as a subroutine to avoid an exhaustive search. The three shortest path queries we use are single-source single-target, single-source all-targets, and all-pairs. Single-source single target shortest path is the fastest subroutine and finds the shortest path between a single pair of nodes. Single-source all targets is the second fastest subroutine and finds the shortest path between a fixed source node and all possible target nodes. All-pairs shortest path is the slowest subroutine and finds the shortest path between all possible source and target nodes. When these subroutines are used without any conditions on the path, they all perform significantly faster than any exhaustive search.

The second optimization strategy is creating a projection of the graph. A projection is a function that creates a new virtual graph based on the contents of the original graph, and a condition that selects which nodes and edges should appear in the virtual graph. For example, a projection might select

all nodes of the original graph but only the red edges of that graph, and a shortest path subroutine on this virtual graph would return the shortest path with the predicate that the path can only contain red edges.

3 Queries and Optimizations

We describe each of the 20 exhaustive and 20 optimized queries in this section. The queries can be grouped in pairs, where one query is the single-source single-target shortest path case, and the other query is the all-pairs shortest path case. Each of these pairs require the same predicate on all paths for both of its queries. There are 10 predicates in total, and each predicate has 4 queries: exhaustive straightforward single-source single-target shortest path, exhaustive straightforward all-pairs shortest path, optimized all-pairs shortest path, and optimized all-pairs shortest path. In this section, we group queries by the predicate that a path must satisfy.

For each predicate, we first describe what the predicate is, and how the straightforward exhaustive queries are implemented. Then, we describe a general optimization strategy. We explain how the optimized single-source single-target query and how the optimized all-pairs query satisfy the predicate. We show how all 4 queries perform on a 5-node and 6-node clique.

We also provide a taxonomy for each predicate. We classify each predicate as either node-based or edge based. We classify each predicate based on whether it considers key-value pairs. We classify each predicate based on the number of projections the optimization strategy requires. Finally, we classify each predicate based on the shortest path subroutine used in the source-target case and the all-pairs case.

We use some notation to simplify understanding. A node can appear as a simple variable such as x. An edge is represented by a pair of nodes such as (u,v). In this case, edge (u,v) represents the edge that connects nodes u and v. $cost(u,v)$ represents the cost of edge (u,v). $sp\_cost(source,target)$ is a function that determines the shortest path cost between node *source* and node *target*. $sp\_cost(source,target,proj)$ is a function that determines the shortest path cost between node *source* and node *target* on a given projection $proj$.

3.1 Predicate 1

Query predicate 1 finds the shortest path cost between a source and a target, where the path must visit a given node x. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths that visit node x, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths that visit node x, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_1.

3.1.1 Optimization

The shortest path cost between a source and a target that must visit node x is $sp\_cost(source,x) + sp\_cost(x,target)$. In the source-target case, the optimized implementation computes the single-source single-target $sp\_cost(source,x)$ and $sp\_cost(x,target)$, and simply returns the sum.

In the all-pairs case, the optimized implementation computes the single-source all-targets $sp\_cost(x,y)$ for all target nodes y. For each pair of source and target nodes, the solution is $sp\_cost(x,y = source) + sp\_cost(x,y = target)$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	1.7s	TLE	55.1s	TLE
Optimized	0.01s	0.01s	0.4s	0.7s

Taxonomy	
Node/Edge Based	Node Based
Uses Key-Value Pairs	False
Uses Projections	0
Source-Target Subroutine	Single-Source Single-Target
All-Pairs Subroutine	Single-Source All-Targets

3.2 Predicate 2

Query pair 2 finds the shortest path cost between a source and a target, where the path must visit a given edge (u,v). In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths that visit edge (u,v), and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths that visit edge u,v, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_2.

3.2.1 Optimization

The shortest path cost between a source and a target that must visit edge (u,v) is $min(sp\_cost(source,u) + (u,v) + sp\_cost(v,target), sp\_cost(source,v) + (v,u) + sp\_cost(u,target))$. In the source-target case, the optimized implementation computes the single-source

single-target $sp\_cost(source, u)$, $sp\_cost(v, target)$, $sp\_cost(source, v)$, and $sp\_cost(v, target)$. The solution is then directly $min(sp\_cost(source, u) + (u, v) + sp\_cost(v, target), sp\_cost(source, v) + (v, u) + sp\_cost(u, target))$.

In the all-pairs case, the optimized implementation computes the single-source all-targets $sp\_cost(u, x)$ for all target nodes x, and computes the single-source all-targets $sp\_cost(v, y)$ for all target nodes y. For each pair of source and target nodes, the solution is $min(sp\_cost(u, x = source) + (u, v) + sp\_cost(v, y = target), sp\_cost(v, y = source) + (v, u) + sp\_cost(u, x = target))$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.01s	TLE	31.9s	TLE
Optimized	0.01s	0.01s	0.7s	1.1s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	False
Uses Projections	0
Source-Target Subroutine	Single-Source Single-Target
All-Pairs Subroutine	Single-Source All-Targets

3.3 Predicate 3

Query predicate 3 finds the shortest path cost between a source and a target, where the path must visit a red node. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths that visit a red node, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths that visit a red node, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_3.

3.3.1 Optimization

The shortest path cost between a source and a target that must visit red node is $min(sp\_cost(source, x) + sp\_cost(x, target))$ across all red nodes x. In the source-target case, the optimized implementation computes the single-source all-targets $sp\_cost(source, x)$ and $sp\_cost(target, x)$ for all target nodes x. The solution is then directly $min(sp\_cost(source, x) + sp\_cost(x, target))$ across all nodes x such that $color(x) = RED$.

In the all-pairs case, the optimized implementation computes the single-source all-targets $sp\_cost(x, y)$ for each red source node x and for all target nodes y. For each pair of source and target nodes, the solution is $min(sp\_cost(x, y = source) + sp\_cost(x, z = target))$ across all red nodes x.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	6.0s	TLE	166.6s	TLE
Optimized	0.03s	0.04s	1.5s	1.5s

Taxonomy	
Node/Edge Based	Node Based
Uses Key-Value Pairs	True
Uses Projections	0
Source-Target Subroutine	Single-Source All-Targets
All-Pairs Subroutine	Single-Source All-Targets

3.4 Predicate 4

Query predicate 4 finds the shortest path cost between a source and a target, where the path must visit a red edge. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths that visit a red edge, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths that visit a red edge, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_4.

3.4.1 Optimization

The shortest path cost between a source and a target that must visit a red edge is $min(sp\_cost(source, u) + (u, v) + sp\_cost(v, target))$ across all red edges (u, v). In the source-target case, the optimized implementation computes the single-source all-targets $sp\_cost(source, x)$ and $sp\_cost(target, x)$ for all target nodes x. The solution is then directly $min(sp\_cost(source, x = u) + cost(u, v) + sp\_cost(target, x = v))$ across all edges (u, v) such that $color(u, v) = RED$.

In the all-pairs case, the optimized implementation computes the single-source all-targets $sp\_cost(u, x)$ for each source node u with $color(u, v) = RED$ and for all target nodes x. The optimized implementation also computes the single-source all-targets $sp\_cost(v, y)$ for each source node v with $color(u, v) = RED$ and for all target nodes y. For each pair of source and target nodes, the solution is $min(sp\_cost(u, x =$

$source) + cost(u,v) + sp\_cost(v, y = target))$ across all red edges (u,v).

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	5.5s	TLE	162.6s	TLE
Optimized	0.02s	0.02s	4.8s	7.8s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	True
Uses Projections	0
Source-Target Subroutine	Single-Source All-Targets
All-Pairs Subroutine	Single-Source All-Targets

3.5 Predicate 5

Query predicate 5 finds the shortest path cost between a source and a target, where every node in the path must share the same color. Every node in the path must be red or every node in the path must be blue. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths where every node shares the same color, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths where every node shares the same color, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_5.

3.5.1 Optimization

The shortest path cost between a source and a target that only visits red nodes is $sp\_cost(source, target, proj\_red)$ on a projection $proj\_red$ of the original graph that only contains the red nodes. The shortest path cost between a source and a target that only visits blue nodes is $sp\_cost(source, target, proj\_blue)$ on a projection $proj\_blue$ of the original graph that only contains the blue nodes. The shortest path cost between a source and a target where every node in the path must share the same color is then $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$. In the source-target case, the optimized implementation computes $proj\_red$ as a projection of the original graph that only contains the red nodes, and computes $proj\_blue$ as a projection of the original graph that only contains the blue nodes. The optimized implementation computes the single-source single-target $sp\_cost(source, target, proj\_red)$ and $sp\_cost(source, target, proj\_blue)$. The solution is then directly $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$.

In the all-pairs case, the optimized implementation also computes $proj\_red$ as a projection of the original graph that only contains the red nodes, and also computes $proj\_blue$ as a projection of the original graph that only contains the blue nodes. The optimized implementation computes the all-pairs $sp\_cost(source, target, proj\_red)$ and $sp\_cost(source, target, proj\_red)$. For each pair of source and target nodes, the solution is $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.07s	1.3s	0.1s	38.0s
Optimized	0.01s	0.01s	0.1s	0.2s

Taxonomy	
Node/Edge Based	Node Based
Uses Key-Value Pairs	True
Uses Projections	2
Source-Target Subroutine	Single-Source Single-Target
All-Pairs Subroutine	All-Pairs

3.6 Predicate 6

Query predicate 6 finds the shortest path cost between a source and a target, where every edge in the path must share the same color. Every edge in the path must be red or every edge in the path must be blue. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths where every edge shares the same color, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths where every edge shares the same color, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_6.

3.6.1 Optimization

The shortest path cost between a source and a target that only visits red edges is $sp\_cost(source, target, proj\_red)$ on a projection $proj\_red$ of the original graph that only contains the red edges. The shortest path cost between a source and a target that only visits blue edges is $sp\_cost(source, target, proj\_blue)$ on a projection

proj_blue of the original graph that only contains the blue edges. The shortest path cost between a source and a target where every edge in the path must share the same color is then $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$. In the source-target case, the optimized implementation computes *proj_red* as a projection of the original graph that only contains the red edges, and computes *proj_blue* as a projection of the original graph that only contains the blue edges. The optimized implementation computes the single-source single-target $sp\_cost(source, target, proj\_red)$ and $sp\_cost(source, target, proj\_blue)$. The solution is then directly $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$.

In the all-pairs case, the optimized implementation also computes *proj_red* as a projection of the original graph that only contains the red edges, and also computes *proj_blue* as a projection of the original graph that only contains the blue edges. The optimized implementation computes the all-pairs $sp\_cost(source, target, proj\_red)$ and $sp\_cost(source, target, proj\_blue)$. For each pair of source and target nodes, the solution is $min(sp\_cost(source, target, proj\_red), sp\_cost(source, target, proj\_blue))$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.07s	1.3s	0.2s	38.0s
Optimized	0.01s	0.01s	0.1s	0.1s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	True
Uses Projections	2
Source-Target Subroutine	Single-Source Single-Target
All-Pairs Subroutine	All-Pairs

3.7 Predicate 7

Query predicate 7 finds the shortest path cost between a source and a target, where no two adjacent nodes in the path share the same color. Every red node in the path must be adjacent to blue nodes, and every blue node in the path must be adjacent to red nodes. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths where no two adjacent nodes share the same color, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths where no two adjacent nodes share the same color, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_7.

3.7.1 Optimization

The shortest path cost between a source and a target where no two adjacent nodes share the same color is $sp\_cost(source, target, proj\_alt)$ on a projection *proj_alt* of the original graph that only contains edges where two adjacent nodes have different colors. In the source-target case, the optimized implementation computes *proj_alt* as a projection of the original graph that only contains the edges where the two adjacent nodes have different colors. The optimized implementation then directly computes the single-source single-target $sp\_cost(source, target, proj\_alt)$.

In the all-pairs case, the optimized implementation also computes *proj_alt* as a projection of the original graph that only contains the edges where the two adjacent nodes have different colors. The optimized implementation then directly computes the all-pairs $sp\_cost(source, target, proj\_alt)$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.07s	1.9s	1.0s	TLE
Optimized	0.02s	0.01s	0.01s	0.01s

Taxonomy	
Node/Edge Based	Node Based
Uses Key-Value Pairs	True
Uses Projections	1
Source-Target Subroutine	Single-Source Single-Target
All-Pairs Subroutine	All-Pairs

3.8 Predicate 8

Query predicate 8 finds the shortest path cost between a source and a target, where the color of every edge in the path must be monotonic. More specifically, every edge in the beginning of the path must be red, and every edge in the end of the path must be blue. There should be no instance where a red edge appears after a blue edge in a path. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the monotonic paths, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the monotonic paths, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/

Cypher-Query-Optimizations/tree/main/src/query_8.

3.8.1 Optimization

The shortest path cost between a source and a target requires a projection $proj\_red$ of the original graph that only contains the red edges, and a projection $proj\_blue$ of the original graph that only contains the red edges. The shortest path cost is then $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue))$ across all nodes x. In the source-target case, the optimized implementation computes $proj\_red$ as a projection of the original graph that only contains the red edges, computes $proj\_blue$ as a projection of the original graph that only contains the blue edges, and computes the single-source all-targets $sp\_cost(source, x, proj\_red)$ and $sp\_cost(target, x, proj\_blue)$ for all target nodes x. The solution is then directly $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue))$ across all nodes x.

In the all-pairs case, the optimized implementation also computes $proj\_red$ as a projection of the original graph that only contains the red edges and $proj\_blue$ as a projection of the original graph that only contains the blue edges. The optimized implementation computes the all-pairs $sp\_cost(source, x, proj\_red)$ and $sp\_cost(x, target, proj\_blue)$. For each pair of source and target nodes, the solution is $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue))$ across all nodes x.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.07s	1.9s	0.4s	60.8s
Optimized	0.01s	0.03s	0.03s	0.05s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	True
Uses Projections	2
Source-Target Subroutine	Single-Source All-Targets
All-Pairs Subroutine	All-Pairs

3.9 Predicate 9

Query predicate 9 finds the shortest path cost between a source and a target, where the color of every edge in the path must be bitonic. More specifically, either every edge in the beginning of the path must be red and every edge in the end of the path must be blue, or every edge in the beginning of the path must be blue and every edge in the end of the path must be red. There should be no instance where a red edge appears before and after a blue edge in the path, and there should also be no instance where a blue edge appears before and after a red edge in the path. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the bitonic paths, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the bitonic paths, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_9.

3.9.1 Optimization

The shortest path cost between a source and a target requires a projection $proj\_red$ of the original graph that only contains the red edges, and a projection $proj\_blue$ of the original graph that only contains the red edges. The shortest path cost is then $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue), sp\_cost(source, x, proj\_blue) + sp\_cost(x, target, proj\_red))$ across all nodes x. In the source-target case, the optimized implementation computes $proj\_red$ as a projection of the original graph that only contains the red edges and $proj\_blue$ as a projection of the original graph that only contains the blue edges. The optimized implementation computes the single-source all-targets $sp\_cost(source, x, proj\_red)$, $sp\_cost(target, x, proj\_blue)$, $sp\_cost(source, x, proj\_blue)$, and $sp\_cost(target, x, proj\_red)$ for all target nodes x. The solution is then directly $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue), sp\_cost(source, x, proj\_blue) + sp\_cost(x, target, proj\_red))$ across all nodes x

In the all-pairs case, the optimized implementation also computes $proj\_red$ as a projection of the original graph that only contains the red edges and $proj\_blue$ as a projection of the original graph that only contains the blue edges. The optimized implementation computes the all-pairs $sp\_cost(source, x, proj\_red)$, $sp\_cost(x, target, proj\_blue)$, $sp\_cost(source, x, proj\_blue)$, and $sp\_cost(x, target, proj\_red)$. For each pair of source and target nodes, the solution is $min(sp\_cost(source, x, proj\_red) + sp\_cost(x, target, proj\_blue), sp\_cost(source, x, proj\_blue) + sp\_cost(x, target, proj\_red))$ across all nodes x

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.05s	6.1s	0.6s	191.3s
Optimized	0.02s	0.02s	1.7s	3.2s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	True
Uses Projections	2
Source-Target Subroutine	Single-Source All-Targets
All-Pairs Subroutine	All-Pairs

3.10 Predicate 10

Query predicate 10 finds the shortest path cost between a source and a target, where the path must visit at least one edge where the colors of the two adjacent nodes are different. In the source-target case, the straightforward implementation exhaustively searches all paths starting at the source and ending at the target, selects the paths that visit an edge where the colors of the two adjacent nodes are different, and finds the minimum cost of all those paths. In the all-pairs case, the straightforward implementation exhaustively searches all paths between all pairs of nodes, selects the paths that visit an edge where the colors of the two adjacent nodes are different, and finds the minimum cost for each pair of nodes. The source code for straightforward and optimized implementations of the source-target and all-pairs cases can be found at https://github.com/ishanashah/Cypher-Query-Optimizations/tree/main/src/query_10.

3.10.1 Optimization

The shortest path cost between a source and a target that must visit an edge where the colors of the two adjacent nodes are different is $min(sp\_cost(source,u) + cost(u,v) + sp\_cost(v,target))$ across all edges (u,v) where $color(u) \neq color(v)$. In the source-target case, the optimized implementation computes the single-source all-targets $sp\_cost(source,x)$ and $sp\_cost(target,x)$ for all target nodes x. The solution is then directly $min(sp\_cost(source,x = u) + cost(u,v) + sp\_cost(target,x = v))$ across all edges (u,v) where $color(u) \neq color(v)$.

In the all-pairs case, the optimized implementation computes the single-source all-targets $sp\_cost(u,x)$ for each source node u adjacent to another node u with $color(u) \neq color(v)$ and for all target nodes x. The optimized implementation also computes the single-source all-targets $sp\_cost(v,y)$ for each source node v adjacent to another node u with $color(v) \neq color(u)$ and for all target nodes y. For each pair of source and target nodes, the solution is $min(sp\_cost(u,x = source) + cost(u,v) + sp\_cost(v,y = target))$ across all edges (u,v) where $color(u) \neq color(v)$.

Performance				
	Source-Target		All-Pairs	
	5 Nodes	6 Nodes	5 Nodes	6 Nodes
Original	0.07s	3.4s	0.3s	TLE
Optimized	0.09s	0.3s	4.7s	12.5s

Taxonomy	
Node/Edge Based	Edge Based
Uses Key-Value Pairs	True
Uses Projections	0
Source-Target Subroutine	Single-Source All-Targets
All-Pairs Subroutine	Single-Source All-Targets

4 Conclusion

We have found 20 predicate-based shortest path Cypher queries that rely on slow, exhaustive search. We have accordingly found 20 optimized queries that use either shortest path subroutines, projections, or both to eliminate the need for exhaustive search. We have shown that our optimized implementations massively outperform the straightforward implementations. In many cases, the straightforward implementation times out on a 6-node clique, when the optimized implementation finishes in a few seconds. We show several cases where a predicate that uses the key-value pairs of nodes or edges forces Cypher to rely on exhaustive search, and for each of those cases we apply graph theory to show a significant optimization. We hope that our application of projections and unconditional subroutines can be useful for an optimizing compiler in the future.