SSSP ALGORITHMS IN GRAPHBLAS

Mia Campdera-Pulido

INTRODUCTION

This project consists of several single-source shortest path (SSSP) algorithms: unweighted SSSP, weighted SSSP, and batch SSSP. The algorithms are solved using linear algebra techniques implemented in the GraphBLAS C API specification and support weighted and unweighted graphs in the form of adjacency matrices.

The project aims to use the algorithms defined to solve board games and find their remoteness[1]. Non-loopy board games, such as tic tac toe, can be defined as a tree and loopy board games, such as chess, as a graph. By putting board game graphs in the form of adjacency matrices we can run the SSSP algorithms and find the desired information.

BACKGROUND

The main goal of SSSP algorithms, as the name suggests, is to calculate the shortest distance/path from a source to another node in a graph. The application of linear algebra operations to solve SSSP algorithms is based on the fact that performing the multiplication of a zero-valued vector where only the source node position contains a 1 and the transpose of an adjacency matrix returns the neighbours of the source node, or the node in question. This multiplication operation does not correspond to the common multiplication; it uses semiring logic. Semirings are a set of two operations, one of additive and the other of multiplicative nature (each of these are known as monoids, single operations).

The project implements these ideas using the GraphBLAS C API specification, which can be found in the references section. The GraphBLAS is an API specification which has defined linear algebra objects such as vector, matrix, and index and several linear algebra operations such as vector-vector multiplication or vector-matrix multiplication. With all of these it is possible to define graph algorithms. Additionally, GraphBLAS methods support OpenMP so large graphs can be correctly handled.

PROJECT SPECIFICATION AND DESIGN

The project supports three SSSP algorithms: unweighted SSSP, weighted SSSP, and batch SSSP.

The unweighted SSSP algorithm calculates the distances between a specified source node to every other node in the graph, for an unweighted graph. Observing the fact that calculating the distances between a source node and every other node where there is no weight between the edges of the nodes (the weight can be interpreted as constant or equivalent to 1) is the same as

[1] Number of moves until a board game is over.

performing breadth first search, BFS, the unweighted SSSP algorithm was designed as a BFS algorithm. The BFS algorithm calculates the levels of a graph; the source node corresponds to level 0, those nodes directly connected to it (neighbors) correspond to level 1, those nodes connected to the neighbours are on level 2, and so on until all nodes have been visited.

The weighted SSSP algorithm calculates the distances between a specified source node to every other node in the graph, for a weighted graph. Taking as inspiration Dijkstra's algorithm implementation with a minimum priority queue (see annex), the weighted SSSP algorithm was designed with a min-plus semiring which is further discussed in the implementation section.

The batch SSSP algorithm calculates the distances of every node to every other node. The design of the algorithm uses the unweighted SSSP algorithm to perform calculations, however, by just changing a single line of code, the weighted SSSP algorithm can be used.

IMPLEMENTATION

The unweighted SSSP algorithm takes as input a result vector which will be filled out during the computation of the algorithm, a source node which corresponds to the node from which all distances to other nodes are calculated, and an unweighted graph in which all the nodes are contained as an adjacency matrix[2]. The output of the SSSP algorithm is the result vector with all the distances from the source to each node where the index of the vector corresponds to the node number. The algorithm traverses through all of the graph by iteration through every set of neighbors. To avoid node repetition, the visited graphBLAS vector keeps track of visited nodes. This is done by comparing the current frontier nodes (the ones being visited during that iteration of the graph traversal) and the ones already visited, if not seen already they are added to the visited vector (method: eWiseAdd). This visited vector is used as a mask in the matrix-vector multiplication, which uses a logic semiring since the input graph is unweighted and node connections (edges) can be treated as connected (1) or not (0) (method: mxv). Finally, since each iteration corresponds to a further level in BFS, a distance is computed and assigned to each node (method: assign).

The weighted SSSP algorithm takes as input a result vector which will be filled out during the computation of the algorithm, a source node which corresponds to the node from which all distances to other nodes are calculated, and an unweighted graph in which all the nodes are contained as an adjacency matrix. The output of the SSSP algorithm is the result vector with all the distances from the source to each node where the index of the vector corresponds to the node number. The algorithm traverses through all of the graph by iteration through every set of neighbors. To avoid node repetition, the visited GraphBLAS vector keeps track of visited nodes. This is done by comparing the current frontier nodes (the ones being visited during that iteration of the graph traversal) and the ones already visited, if not seen already they are added to the visited vector (method: eWiseAdd). This visited vector is used as a mask in the matrix-vector

[2] An adjacency matrix is generated by taking each row index as the corresponding node number and setting the column index of the matrix to the edge weight between the corresponding nodes of the row index and the column index. If an edge doesn't exist a 0 is imputed (it could also be set to null).

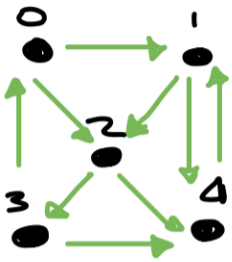
multiplication, which uses the min-plus semiring (method: mxv). The min-plus semiring chooses the minimum weight edge between two vectors and adds the past distance traveled to the new minimum weighted edge found. Thus, it is possible to compute the distance between the source and every other node.

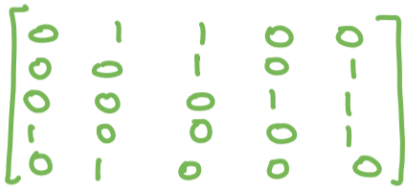
The batch SSSP algorithm takes as input a distances matrix and a graph over which the algorithm will calculate the distances. The output of the SSSP algorithm is the distances matrix, where the row number corresponds to the node which acted as the source and the row itself is the distance vector for that node. The algorithm calculates the unweighted SSSP for each node in the graph. It is possible to parallelize the iteration over all nodes in the graph.

RESULTS AND EVALUATION

We will use a small example graph to showcase the functionality of the three algorithms. We will analyze an unweighted and a weighted graph, both directed since board games have moves that may not be reversible. If a move were to be reversible then the corresponding edge between the two states of the game (nodes) would be bidirectional. The algorithms will also work for undirected graphs, granted that the adjacency matrix is set up accordingly.

Unweighted SSSP: Graph and adjacency matrix



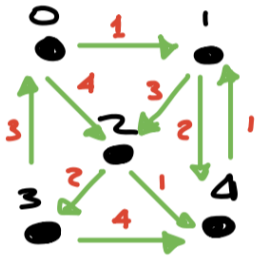


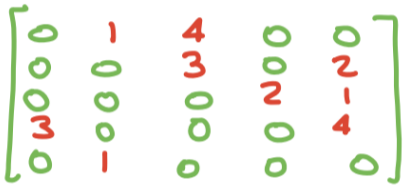
The main.c has a method called run_uw(). If we call this method the uw_sssp is run with input source node 0 and the drawn graph. The output of this is: [0, 1, 1, 2, 2], which is correct. The uw_sssp function also prints out the input graph:

Matrix: Graph =
[-, 1, 1, -, -]
[-, -, 1, -, 1]
[-, -, -, 1, 1]
[1, -, -, -, 0]
[-, 1, -, -, -]

As you can see GrapBLAS uses sparse adjacency matrices (non existing edges are not represented as 0 but as nonexistent). As you can see there is an existing edge that is represented with a 0, this corresponds to the connection of node 3 to 4. I was not able to figure out why this happened. When trying out several other graphs, some of the edges also came out as 0. This may lead to error when calculating the unweighted SSSP distances since the algorithm relies on binary logic. However, this is not the case in this example.

Weighted SSSP: Graph and adjacency matrix



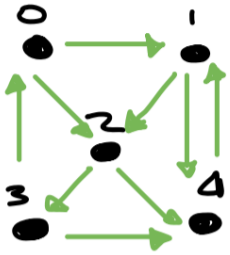


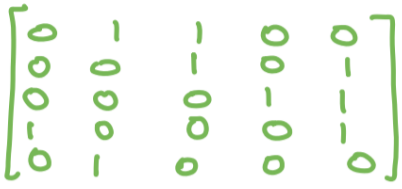
The main.c has a method called run_w(). If we call this method the w_sssp is run with input source node 0 and the drawn graph. The output of this is: [1, 2, 5, 7, 4]. Due to the functionality of the min-plus semiring, the start distance had to be set as 1 instead of 0. This is why the distance vector outputs a distance of length 1 greater than the real distances. The w_sssp function also prints out the input graph:

Matrix: Graph =
[-, 1, 4, -, -]
[-, -, 3, -, 2]
[-, -, -, 2, 1]
[3, -, -, -, 4]
[-, 1, -, -, -]

This is the correct corresponding sparse adjacency matrix, no errors have been found with the way GraphBLAS interpret weighted matrices.

Batch SSSP: Graph and adjacency matrix





The batch SSSP algorithm uses the same input matrix as the unweighted SSSP algorithm. It outputs the following distances matrix:

Matrix: distances =
[0, 1, 1, 2, 2]
[3, 0, 1, 2, 1]
[2, 2, 0, 1, 1]
[1, 2, 2, 0, 3]
[4, 1, 2, 3, 0]

As you can see, since the edge from 3 to 4 was interpreted as 0 by GraphBLAS (see unweighted SSSP section), the distances from node 3 to 4 is incorrect: in row 3 (starting with 0 index) we can see that the distance is 3, while if we check the graph we can clearly see that it is 1.

FUTURE WORK

Linear algebra, although being a runtime efficient implementation for SSSP algorithms, does have limitations. Algebraic operations over matrices must continue throughout every element of a row/column. Since each element in a matrix represents a node in a graph, when using linear algebra to solve graph algorithms the entire graph must be traversed. This prevents the implementation of those algorithms that do not traverse the entire graph, such as a* and other algorithms dependent on a heuristic function. The application of linear algebra to artificial intelligence is still a topic being researched and one that graph algorithms could clearly benefit from.

CONCLUSION

The SSSP algorithms are very effective for solving board games and are a clear example of the power of linear algebra and its applicability to graph algorithms. The batch SSSP algorithm, which can use either the weighted or unweighted SSSP algorithms is very useful for storing the complete information of a game and having quick access to distances, and deciding on future moves from any position (node) in a game (graph). An additional function to convert board game graphs into GraphBLAS adjacency matrices might be useful to ease the application of the SSSP algorithms from the project to board games.

REFLECTION

The project demanded a lot of research on solving single-source shortest path algorithms with linear algebra techniques. Additionally, a clear understanding of the GraphBLAS C API specification was crucial. Although gathering the necessary information to complete the project

was very challenging and time-consuming, I believe that it was worth it and it allowed me to complete the project more clearly and efficiently. Overall, I have learned a lot about the application of linear algebra to the SSSP algorithms I had implemented via different methods. I have realized the power of linear algebra beyond class projects and its applicability to algorithms that can be used to solve current problems the world is facing today. I have also gained a lot of knowledge on GraphBLAS and want to continue exploring it and the opportunities it provides me to further pursue my interest in board game solving.

REFERENCES

https://people.eecs.berkeley.edu/~aydin/GraphBLAS-Math.pdf

https://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf

CODE ACCESS

1. Clone repo: https://gitlab.bsc.es/mpulido/bsc-graphblas

2. The CTE-AMD machine has GraphBLAS installed, run: module load graphblas/5.1.5

3. If it errors, transfer the graphblas.h file from https://github.com/DrTimothyAldenDavis/GraphBLAS into a personal folder.

4. When compiling use:

gcc -I <location of graphblas.h file> -L /apps/GRAPHBLAS/5.1.5 -l graphblas main.c uw_sssp.c w_sssp.c batch_sssp.c

ANNEX

Pseudocode for Dijkstra's implementation using a minimum priority queue:

```
Dijkstra(G, target):
	PQ.add(s, 0)
	for all other vertices v:
		PQ.add(v, ∞)
	edgeTo[s] = null
	while PQ is not empty:
		v = PQ.removeSmallest()
		if v == target:
			break
		for all edges(v, w):
		// means we have found a shorter path
			if doFringe[w] > distTo[v] + edgeWeight(v, w):
```

```
distTo[w] = distTo[v] + edgeWeight(v, w)
doFringe[w] = distTo[w]
edgeTo[w] = v
PQ.changePriority(w, doFringe[w])
```