Optimized Graph traversing Algorithm for multi-weighted graphs with priority nodes and a constraint

Dilshan K.M.N.
Department of Computer Engineering
Faculty of Engineering
University of Sri Jayewardenepura
Nugegoda,Sri Lanka
en97569@sjp.ac.lk

***Abstract*—Graph traversal can be either simple or complex based on the context. This particular article is concerned with developing a traversal algorithm to suit multi-weighted graphs with node priorities and a constraint. The objective of this optimized algorithm is not to find the shortest path between two nodes, but to maximize the nodes explored while traversing from the source to the destination. The multi-weights in this scenario are denoted with edge weights and switch weights that are incurred when switching from a node to an adjacent edge. The constraint is the total permissible cost of the path. Each node in the network will be assigned with a rating value that is used to determine the nodes that need to be prioritized when maximizing the number of nodes explored within the given constraint limit. Hence, this work will be an optimized version of the graph traversal algorithms with particular fields of application.**

I. INTRODUCTION

Graph traversal has been an area of interest of various industries since a long time ago. Because of this, various graph traversal algorithms have been introduced and developed to meet various graph traversal requirements. Pathfinding and navigation, network analysis, logistics and supply chain management and bioinformatic are some applications where graph traversal algorithms contribute [1][2]. When considering the traversal requirements, various real world applications require various requirements. Graphs can either be weighted or non-weighted but for the majority of practical applications, graphs are always weighted. Some traversal algorithms seek to find a path between two given nodes. Such algorithms do not necessarily need to traverse the entire network. They will provide a usable output once a path from the source node to the destination node has been determined. But in other cases where it is necessary to find the shortest path between a given source and a destination node, the algorithm needs to traverse the entire network to come up with an accurate answer. More optimized versions of such traversal algorithms are even used to find the shortest path between any two nodes within a network of nodes.

The optimized algorithm proposed in this paper, takes a different set of requirements. Here, the objective is to maximize the number of nodes explored while keeping the total cost incurred under a given constraint. A new parameter called 'rating' is assigned to each node aside from the source and the destination to denote the priority level of each node. The algorithm will also take that rating value into consideration when maximizing the nodes explored. This doesn't mean the algorithm would definitely select the highest priority node, it depends on the other factors such as the constraint and the edge weights.

The implementation of the algorithm to serve this specific objective can be broken down into three main steps.

1. Develop an algorithm to maximize node exploration.
2. Develop an algorithm prioritize nodes based on rating value
3. Integration of the two algorithms to achieve the objective

II. METHODOLOGY

Given graph G = (V, E), with a set of vertices V and a set of edges E, where an edge can be directed or undirected. Each edge $e \in E$ is assigned a weight $w_e(u, v)$. The notation $w_v(v)$ denotes the additional switch weight associated with the node v[3][4]. Switch weights are not assigned to the source node and the destination node. For all intents and purposes, consider the weights are in units of time. The algorithms are to traverse from the source node *S* to the destination node *D* within the time *T*. The total_time (l(p)) must be equal or lesser than *T* at the end of the process.

$$l(p) = \sum_{j=1}^{h} w_e(e_j) + \sum_{j=1}^{h-1} w_v(u_j) \qquad (1)$$

A. Algorithm for maximizing the nodes explored

- Inputs
  - Graph $G = (V, E, w_e, w_v)$
  - Source vertex $s \in V$, Destination vertex $t \in V$
  - Time constraint T
- Outputs
  - best_path: The path from s to t while achieving maximum node exploration within the T constraint.
- Procedure
  - Precompute the shortest time t(v) for all v ∈ V using Dijkstra's algorithm on G [5] [6]

$$shortest\_time\_to\_t[v] = min(\sum w_e(e) + \sum w_v(u)) \quad (2)$$
path p: from v to t

- Utilize a priority queue to explore paths, prioritizing those with high number of nodes relative to the total time spent.

- Algorithm
  - For each new edge e = (u,v) explored from current vertex u, the new total time is,

    $$total\_time = total\_time + w_e(e) + w_v(u_j) \quad (3)$$

  - The priority queue orders the paths based on their potential to maximize the number of nodes explored. The potential is denoted by the parameter 'score'. The higher it is, higher the potential.

    $$Priority\ \ Score = \frac{node\_count}{total\_time} + \alpha \cdot est\_nodes \quad (4)$$

    - node_count : Number of distinct nodes in the path up to the current point.
    - total_time : The sum of total time cost up to the current node.
    - est_nodes : An estimate of additional number of nodes that can be visited within the remaining time frame.

      $$est\_nodes = \frac{T - total_{time} - shortest\_time\_to\_t[v]}{average\_travel\_time} \quad (5)$$

    - T : Time constraint
    - shortest_time_to_t[v] : The precomputed shortest time from the node v to the destination t
    - average_travel_time : Mean edge weight. Calculated to get the average time per node.

      $$average\_travel\_time = \frac{\sum w_e}{|E|} + \frac{\sum w_v}{|V-2|} \quad (6)$$

    - Tuning Parameter (α) : Preferably set to 0.5 to balance the current progress and the future potential

- The feasibility of a path is determined based on the path's ability to reach the destination within the given time constraint. A path is pruned if it has no potential to reach the destination within the time constraint.

  $$total\_time + shortest\_time\_to\_t[v] > T \quad (7)$$

While the queue is not empty, extract the state with the highest score, check if it reaches within T, update the path based on the node count, prune paths that are not feasible to explore neighbors and calculate the score for them. Repeat until the best path is found otherwise indicate no feasible path.

```
MaxNodesRoute (G, s, t, T)
  Compute shortest_time_to_t[v] for all v ∈ V
  using Dijkstra's algorithm with edge cost: we(e) + wv(v)

  avg_travel_time = (sum of we(e) for all e ∈ E) / |E|
            + (sum of wv(v) for all v ∈ V, v ≠ s, t) / (|V| - 2)

  α =0.5
  PriorityQueue Q = empty
  Q.push((s, 0, 0, [s]))

  best_node_count = -∞
  best_path = null

  while Q is not empty do:
    (u, total_time, node_count, path) = Q.pop()

    if u = t and total_time ≤ T then
      if node_count > best_node_count then
        best_node_count = node_count
        best_path = path
      continue

    if total_time + shortest_time_to_t[u] > T then:
      continue

    for each edge e = (u, v) in E do
      if v is not in path then:
        new_time = total_time + we(e) + wv(v)

        if new_time + shortest_time_to_t[v] ≤ T then
          new_node_count = node_count + 1
          new_path = path + [v]
          est_nodes=(T − new_time − shortest_time_to_t[v]) / avg_travel_time
          score = new_node_count / new_time + α . est_nodes
          Q.push((v, new_time, new_node_count,
          new_path), score)

  if best_path is not null then
    return best_path
  else
    return "No feasible path exists"
```

- The time complexity of the above algorithm can be expressed as follows:

  Precomputation : O(|E| + |V|log|V|)

  Main Algorithm : O(|V|Tlog(|V|T))

  Total Complexity : O(|E| + |V|log|V| + |V|Tlog(|V|T))

B. Algorithm to maximize the priority score

- Inputs
  - Graph $G = (V,E,w_e,w_v)$, with edge weights we(e) and node-dependent switch weights wv(v) and node priority values R(v).
  - Source vertex $s \in V$, Destination vertex $t \in V$
  - Time constraint T

- Outputs
  - best_path : The path from s to t achieving maximum possible cumulative node priority value within the time constraint T.

- Procedure
  - Precompute the shortest time t(v) for all v ∈ V using Dijkstra's algorithm on G [5] [6]

$$Shortest\_time\_to\_t[v] = \min_{\text{path p: from v to t}}\left(\sum w_e(e) + \sum w_v(u)\right) \quad (2)$$

  - Utilize a priority queue to explore paths, prioritizing those with high number of nodes relative to the total time spent.

- Algorithm
  - The general procedure is the same as in the algorithm for maximizing the nodes explored with only the formula for calculating the score having slight adjustments.

$$Priority\ Score = \frac{sum\_rating}{total\_time} + \alpha \cdot est\_rate \quad (8)$$

sum_rating: The cumulative of the node priority values of the path being explored after adding the current node to the path.

$$sum\_rating = sum\_rating + R(v) \quad (9)$$

est_rate : The estimated future node rates that can be accumulated along the remaining path from the current node to the destination t.

$$est\_rate = max\_rating \cdot \frac{T - total_{time} - shortest\_time\_to\_t[v]}{average\_travel\_time} \quad (10)$$

max_rating : The maximum node rating among all the nodes.

While the queue is not empty, extract the state with the highest score, check if it reaches within T, update the path based on the node count, prune paths that are not feasible to explore neighbors and calculate the score for them. Repeat until the best path is found otherwise indicate no feasible path.

```
MaxRateRoute (G, s, t, T)
  Compute shortest_time_to_t[v] for all v ∈ V
  using Dijkstra's algorithm with edge cost: w_e(e) + w_v(v)

  avg_travel_time = (sum of w_e(e) for all e ∈ E) / |E|
          + (sum of w_v(v) for all v ∈ V, v ≠ s, t) / (|V| - 2)
  max_rating = max {R(v) | v ∈ V}

  α = 0.5

  PriorityQueue Q = empty
  Q.push((s, 0, 0, [s]))

  best_s = -∞
  best_path = null

while Q is not empty do
    (u, total_time, score, path) = Q.pop()

    if u = t and total_time ≤ T then
      if score > best_s then
        best_s = score
        best_path = path
      continue

    if total_time + shortest_time_to_t[u] > T then
      continue

    for each edge e = (u, v) in E do
      if v is not in path then
        new_time = total_time + w_e(e) + w_v(v)

        if new_time + shortest_time_to_t[v] ≤ T then
          new_score = score + R(v)
          new_path = path + [v]
          est_rate = max_rating * (T - new_time -
shortest_time_to_t[v]) / avg_travel_time
          priority_score = (new_score / new_time) + α *
est_rate

          Q.push((v, new_time, new_score, new_path),
priority_score)

  if best_path is not null then
    return best_path
  else
    return "No feasible path exists"
```

The time complexity of the above algorithm can be expressed as follows:

Precomputation : O(|E| + |V|log|V|)

Main Algorithm : O(|V|Tlog(|V|T))

Total Complexity : O(|E| + |V|log|V| + |V|Tlog(|V|T))

C. *Algorithm to balance node exploration and prioritizing node ratings*

This algorithm is the final objective of the study. This is derived from the integration of the previous two algorithms to obtain the defining features of each.

- Inputs
  - Graph $G = (V,E,w_e,w_v)$, with edge weights $w_e(e)$ and node-dependent switch weights $w_v(v)$ and node priority values R(v).
  - Source vertex $s \in V$, Destination vertex $t \in V$
  - The trade-off parameter $\beta \in [0,1]$

- Time constraint T

- Outputs
  - best_path : The path from s to t achieving a balance between maximizing node exploration and prioritizing node ratings while adhering to the T constraint.

- Procedure
  - Precompute the shortest time t(v) for all v ∈ V using Dijkstra's algorithm on G [5] [6]

$$Shortest\_time\_to\_t[v] = \min_{\text{path p: from v to t}}\left(\sum w_e(e) + \sum w_v(u)\right) \quad (2)$$

  - Utilize a priority queue to explore paths, prioritizing those with high number of nodes relative to the total time spent.

- Algorithm
  - The general procedure is the same as in the algorithm for maximizing the nodes explored with only the formula for calculating the score having slight adjustments.

$$Priority\ Score = \frac{hybrid\_score}{total\_time} + \alpha \cdot est\_hyScore \quad (11)$$

$$hybrid\_score = \beta \cdot node\_count + (1-\beta) \cdot sum\_rating \quad (12)$$

$$est\_hyScore = \beta \cdot est\_nodes + (1-\beta) \cdot est\_rate \quad (13)$$

  - The trade-off parameter is used to balance the trade-offs between prioritizing node exploration and node ratings. Preferably it is set to 0.5 for unbiased computation.

While the queue is not empty, extract the state with the highest score, check if it reaches within T, update the path based on the node count, prune paths that are not feasible to explore neighbors and calculate the score for them. Repeat until the best path is found otherwise indicate no feasible path.

```
MaxHybridRoute (G, s, t, T)
  Compute shortest_time_to_t[v] for all v ∈ V
  using Dijkstra's algorithm with edge cost: we(e) + wv(v)

  avg_travel_time = (sum of we(e) for all e ∈ E) / |E|
          + (sum of wv(v) for all v ∈ V, v ≠ s, t) / (|V| - 2)

  max_rating = max { R(v) | v ∈ V }

  α = 0.5
  β = 0.5
  PriorityQueue Q = empty
  Q.push((s, 0, 0, 0, [s]))

  best_h = -∞
  best_path = null

  while Q is not empty do
    (u, total_time, node_count, score, path) = Q.pop()
    hybrid_score = β * node_count + (1 - β) * score

    if u = t and total_time ≤ T then
      if hybrid_score > best_h then
        best_h = hybrid_score
        best_path = path
      continue

    if total_time + shortest_time_to_t[u] > T then
      continue

    for each edge e = (u, v) in E do
      if v is not in path then
        new_time = total_time + we(e) + wv(v)

        if new_time + shortest_time_to_t[v ≤ T then
          new_node_count = node_count + 1
          new_score = score + R(v)
          new_hybrid = β * new_node_count + (1 - β) *
new_score
          new_path = path + [v]
          est_nodes = (T - new_time -
shortest_time_to_t[v]) / avg_travel_time

          est_score = max_rating * est_nodes
          est_hyScore = β * est_nodes + (1 - β) *
est_score
          priority_score = (new_hybrid / new_time) + α *
est_hybrid
          Q.push((v, new_time, new_node_count,
new_score, new_path), priority_score)

  if best_path is not null then
    return best_path
  else
    return "No feasible path exists"
```

The time complexity of the above algorithm can be expressed as follows:

Precomputation: O(|E| + |V|log|V|)

Main Algorithm: O(|V|Tlog(|V|T))

Total Complexity: O(|E| + |V|log|V| + |V|Tlog(|V|T))

III. Results and Discussion

For the given graph G = (V,E) in the Fig. 1, the |V| = 6 nodes and |E| = 9 edges and a time constraint (T) of 17.

Trade-off parameter (β) = 0.5

The Table 01 and Table 02 contains the node and edge data respectively.

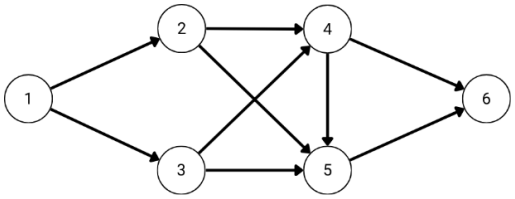

Fig. 1. The graph G = (V,E) containing the nodes and edges and their connections

TABLE 01 NODE DATA

Node	Switch Weight (w_v)	Rating (R_v)
1	0	0
2	2	3
3	1	4
4	3	2
5	2	5
6	0	0

TABLE 02 EDGE DATA

Edge (U,V)	w_e
(1,2)	3
(1,3)	4
(2,4)	2
(2,5)	5
(3,4)	3
(3,5)	2
(4,5)	1
(4,6)	4
(5,6)	3

A. Summary of the expected results

Table 03 contains the results when the graph G = (V, E) in Fig. 1 is processed with the three algorithms separately. The first column displays the algorithm used while the second column displays the resulting output for that particular algorithm. Total Rate is the sum of the rating values of all the nodes traversed excluding the source and the destination while total time represents the total time taken by the particular path.

TABLE 03 SUMMARY OF RESULTS

Algorithm	Path	Total Rate	Total Time
Max Nodes	[1, 2, 4, 5, 6]	-	16
Max-Rate	[1, 3, 4, 5, 6]	11	17
Hybrid	[1, 3, 4, 5, 6]	11	17

All the results obtained have adhered to the time constraint T by maintaining a traversal time of less or equal to 17.

The Max nodes algorithm has generated a path with five nodes, which is the maximum possible number of nodes a single path can explore while at the same time minimizing the cost.

The Max-Rate algorithm has generated a new path prioritizing the node rates over traversal cost. The path generated has the highest cumulative rate a single path can possibly possess in this context.

The Hybrid implementation algorithm has generated the same path as the Max-Rate algorithm as it has the maximum cumulative rate as well as the maximum number of nodes a single path can traverse in this graph.

Although the computation of estimation values in each algorithm introduces variability and thus does not guarantee the optimal path, in this context the paths generated by each algorithm are optimal according to the requirements presented.

Finally, all the three algorithms display the same time complexities. Although the algorithms differ somewhat with differing objective, each of them explore the same underlying state space paths from the source node s to the destination node t while adhering to the same constraint and the same pruning rule. Their differences in objective effect only the order in which they explore the nodes.

IV. Further Improvements and Optimizations

A. Improved Traversal

The performance of the algorithms is improved using a pruning technique. Pruning is essential for the above algorithms to sufficiently decrease the complexity of the problem. Without pruning, an exhaustive search of the graph would result in a NP-hard problem with increased complexity much like in the Travelling Salesman problem.

These algorithms can be further improved by developing them in a thread-based structure so that the algorithm can evaluate multiple feasible paths at once to reduce the time spent.

B. *Heterogenous weights and multiple constraints*

The optimized algorithms are based on graphs with homogenous weights and constraints meaning the weights and the constraints must always be of the same type. But in practical scenarios there can be many types of weights and each of them have the possibility of having a separate constraint for their type of weight. The above algorithms can be further improved to integrate heterogenous weights and multiple constraints.

C. *Alternative approach for avg_travel_time:*

The avg_travel_time is computed considering the global aspect of the scenario. But this can result in slight deviations from the optimal solution. This can be mitigated by either computing the avg_travel_time by considering a local neighbourhood of nodes or updating it dynamically as the traversal progresses [7].

D. *Edge dependant switch weights:*

The algorithms developed assume that the switch weights are entirely dependent on the node considered and is not affected by the edges used to reach that particular node or the edge that is supposed to be explored next after the node in question has been visited. This in turn is also an assumption that can deviate from certain practical scenarios. To accommodate such scenarios, the algorithms can be further improved to consider edge dependent switch weights.

V. CONCLUSION

The algorithms, node maximizing, rate maximizing and hybrid implementation, discussed in this study are developed to address the challenges of path optimization of graphs with node dependent switch weights. The objective of their development has been to introduce a methodology to traverse graphs prioritizing certain nodes over others while maximizing the total number of nodes traversed within the given constraint. The node maximizing and rate maximizing algorithms are sub algorithms developed to integrate into the final result which is the hybrid implementation that introduces a trade-off parameter for the user to control the main aspect of the algorithm, whether to be biased towards maximizing nodes explored or prioritizing highly rated nodes or choosing a middle ground between the two options.

The performance of the developed algorithms depends on the context of the graph as there are several estimations that are made during the computations. In other words, these algorithms are not guaranteed to provide the optimal solution but depend on the conditions of the graph. Even then, the output obtained from the algorithms is of significance for various applications such as tour planning, emergency rescue and delivery route plannings.

REFERENCES

[1]S. Prakash, “GRAPH TRAVERSALS AND ITS APPLICATIONS,” Fully Refereed and Peer Reviewed Journal), vol. 10, no. 7, 2022, doi: https://doi.org/10.5281/zenodo.6838779.

[2]C. Monga and Richa, “GRAPH TRAVERSALS AND ITS APPLICATIONS IN GRAPH THEORY,” International Journal of Computer Science and Mobile Applications, vol. 6, no. 1, 2018.

[3]C. T. Quoc and H. H. Van, “Applying algorithm finding shortest path in the multiple- weighted graphs to find maximal flow in extended linear multicomodity multicost network,” EAI Endorsed Transactions on Industrial Networks and Intelligent Systems, vol. 4, no. 11, p. 153499, Dec. 2017, doi: https://doi.org/10.4108/eai.21-12-2017.153499.

[4]I. Basov and A. Vainshtein, “Approximation algorithms for multi-parameter graph optimization problems,” Discrete Applied Mathematics, vol. 119, no. 1–2, pp. 129–138, Jun. 2002, doi: https://doi.org/10.1016/s0166-218x(01)00269-4.

[5]G. Y. Handler and I. Zang, “A dual algorithm for the constrained shortest path problem,” Networks, vol. 10, no. 4, pp. 293–309, Dec. 1980, doi: https://doi.org/10.1002/net.3230100403.

[6]A. Javaid, Understanding Dijkstra’s Algorithm. Soft Computing, 2014.

[7]J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, “Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency,” 2012 IEEE 26th International Parallel and Distributed Processing Symposium, vol. 1, no. 26, pp. 378–389, May 2012, doi: https://doi.org/10.1109/ipdps.2012.43.