C.6. Construction of a multitask model

This section describes the methodology for constructing a unified model capable of executing multiple graph algorithms. To this end, some initial considerations are necessary. Each algorithm requires a distinct set of variables and functions. In our constructions, this is reflected as specific columns in the input matrix and the different implemented layers, respectively. Therefore, a model that unifies different algorithms must accommodate these individual components without compromising the execution of any individual algorithm. Nevertheless, some algorithms can have common structures, sharing similar functions or variables utilized in analogous ways. The challenge of multitasking extends beyond just encapsulating different executions within the same model. It also involves efficiently reusing shared variables and functions to avoid redundancy, which also significantly reduces overhead in terms of memory (number of columns of X) or runtime (number of layers).

Below, we describe constructing a multitask model that executes three distinct graph algorithms: Breadth-first search, Depth-first search, and Dijkstra's shortest path algorithm. We limit our scope to these three algorithms, considering the complexity and minimal additional insights gained from incorporating more algorithms. However, this same design principle can be applied to a broader range of algorithms, including Strongly Connected Components (referenced as Appendix C.5). Incorporating a new set of algorithms could potentially incur the introduction of new functions and a distinct set of variables to be integrated into the matrix X.

Our goal is to provide a single implementation capable of executing one of these three algorithms given the appropriate input configuration. The structure of the input is the same for all three algorithms. However, its configurations slightly change for the execution of each algorithm. While Breadth & Depth-first search operate on unweighted graphs, the execution of Depth-first search is distinguished by the activation of a specific flag in X, denoted by γ_s. For Breadth-first search and Dijkstra's algorithm, the configuration of X is exactly the same. What sets Dijkstra's algorithm apart from BFS is its operation on a weighted graph, as opposed to the unweighted graphs used by the other two algorithms. This distinction also highlights the fact that Breadth-first search can be considered a special case of Dijkstra's algorithm when applied to unweighted graphs. Consequently, we can leverage a single execution for both breadth-first search and Dijkstra's algorithm.

Furthermore, all three algorithms share a large portion of similar functions. For example, they all employ the minimum function during the initial phase of iteration and utilize a similar termination criterion. Our implementation strategy consists of leveraging this shared structure while individually accommodating the unique functions of each algorithm. The selection of specific elements necessary for executing a particular algorithm is managed by a selector function. This function determines the variables that need to be updated, thus ensuring the execution reflects the intended algorithmic behavior.

The comprehensive structure of our implementation is illustrated in Algorithm 7. Non-highlighted lines indicate the shared structural components common to all three algorithms. In contrast, colored lines denote algorithm-specific adaptations. Specifically, lines highlighted in blue (18, 22, 29, 38, and 39) are modifications for Depth-first search. Lines in red (24, 25, 30, 36, and 37) indicate the adaptations for both Dijkstra's and Breadth-first search. Lastly, the lines highlighted in orange (42-45) represent the conditional selection mechanism. This mechanism is crucial for dynamically selecting the algorithm-specific elements and the boolean variables that trigger these adaptations. Through this structured implementation, we establish the following remark:

Remark C.1. There exists a looped transformer h_T in the form of (2), with 19 layers, 3 attention heads and layer width $O(1)$ that (i) Simulates Depth-First Search (DFS) and Breadth-First Search (BFS) for unweighted graphs with up to $\min(O(\hat{\delta}^{-1}), O(\Omega))$ nodes; and (ii) Simulates Dijkstra's shortest path algorithm for weighted graphs with rational edge-weights with up to $O(\hat{\delta}^{-1})$ and graph diameter of $O(\Omega)$.

Since the implementations directly follow the specifications outlined in previous sections, the guarantees for each algorithm are established according to their respective designs (refer to Appendix C.3, Appendix C.4, and Appendix C.2). Furthermore, except for the selector function process, the details of each algorithmic step have been thoroughly discussed earlier. In the following, we present the implementation of the selector function, along with a detailed description of the algorithm.

Additionally, we also conduct empirical validation, as detailed in Appendix B.3. This validation confirms the robustness of our unified implementation, described in Algorithm 7, which demonstrates a 100% accuracy across all tested instances of the three algorithms.

C.6.1. Update priority factor: Step (12)

As previously discussed in Appendix C.4.1, for the execution of Depth-first search, the priority variable `order` must be decreased at each iteration. However, for the multitask model, this process should not be carried out if the model is executing a different algorithm. To this end, we introduce a condition for updating the priority factor.

In our construction, we substitute the conditional form for an equivalent expression: $\text{order} = \text{order} - \phi(\text{term}_{\text{min}} + \gamma_s - 1)$. This ensures that the variable `order` is only updated if $\texttt{term}_{\texttt{min}}$ and γ_s are activated. We implement this condition by setting the parameters of f_{attn} to zero, and we define the parameters of f_{MLP} as follows:

$$(W^{(1)})_{i,j} = \begin{cases} 1 & \text{if } i \in \{\text{term}_{\text{min}}, \gamma_s\}, j = \text{order} \\ -1 & \text{if } i = B_{\text{global}}, j = \text{order} \\ 0 & \text{otherwise,} \end{cases} \qquad (W^{(2,3)})_{i,j} = \begin{cases} 1 & \text{if } i, j = \text{order} \\ 0 & \text{otherwise.} \end{cases}$$

$$(W^{(4)})_{i,j} = \begin{cases} -1 & \text{if } i, j = \text{order} \\ 0 & \text{otherwise.} \end{cases}$$

The output of the last layer of f_{MLP} is directly added to the residual connection X, effectively replicating the expression above.

C.6.2. Select candidates and changes variables: Step (17)

The variable `candidates` represents the values used for updating the current distances or priorities, essential for the Dijkstra/BFS and DFS algorithms. Specifically, $\texttt{candidates}_1$ refers to the candidate values for Dijkstra/BFS, while $\texttt{candidates}_2$ indicates those for DFS. Similarly, the `changes` variable is a boolean-flag array containing flags that indicate which values require updating. The variables $\texttt{changes}_1$ and $\texttt{changes}_2$ correspond to the update flags for Dijkstra/BFS and DFS, respectively.

Finally, during the algorithm's execution, we must determine which variables are going to be chosen: $\texttt{changes}_1$ and $\texttt{candidates}_1$ or $\texttt{changes}_2$ and $\texttt{candidates}_2$. This decision is guided by the boolean flag γ_s, which, when activated, indicates that the DFS routine should be executed, thereby selecting the second set of variables; otherwise, the first set is chosen. This expression is implemented as $\texttt{cond-select(X, [change}_2\texttt{, candidates}_2\texttt{], [change}_1\texttt{, candidates}_1\texttt{], }\gamma_s\texttt{, [change, candidates]}$, utilizing the conditional selection function described in Appendix C.1.1. Here, the variable γ_s is also repeated along the last n rows during step (14) of the algorithm, whose objective is to replicate the top row value along these rows.

Algorithm 7 General algorithm for DFS/BFS/Dijkstra

Input: integer start
Input: bool γ_s, switch flag
Input: matrix A, size $n \times n$

---

visit[start], order, term = 0, 0, **false**
prev, visit, dists, $\text{dists}_{\text{masked}}$, changes, is_zero, candidates = arrays of size n
for $i = 1$ **to** n **do**
  visit[i], dists[i], prev[i] = **false**, $\hat{\Omega}$, i
end for
$\cdots$ *Initialization of min-variables*

---

while term is **false do**
  for $i = 1$ **to** n **do**
    if visit[i] is **true then**
      $\text{dists}_{\text{masked}}$[i] = Ω *(1) Mask visited nodes* [C.2.1]
    else
      $\text{dists}_{\text{masked}}$[i] = dists[i]
    end if
  end for

---

  get_minimum($\text{dists}_{\text{masked}}$) *(2-8) Find minimum value* [C.1]

---

  if term_{min} is **true then**
    node = idx_{best}
    dist = val_{best} *(9) Get minimum values* [C.2.2]
  end if
  A_{row} = A[node, :] *(10) Get row of A* [C.2.3]
  for $i = 1$ **to** n **do**
    is_zero[i] = (A_{row}[i] $\leq$ 0) *(11) Mark non-neighbors* [C.2.4]
  end for
  if γ_s is **true then**
    order = order - term_{min} *(12) Update priority factor* [C.6.1]
  end if
  visit[node] = visit[node] + term_{min} *(13) Visit node* [C.2.9]
  for $i = 1$ **to** n **do**
    candidates_1[i] = A_{row}[i] + dist *(14) Build candidates* [C.2.5]
    candidates_2[i] = order
  end for
  for $i = 1$ **to** n **do**
    changes_1[i] = candidates_1[i] $<$ dists[i] *(15) Identify updates* [C.2.6]
  end for
  for $i = 1$ **to** n **do**
    change_2 = term_{min} is **true and** visit[i] is **false** *(16) Build flags* [C.2.7/C.4.2]
    changes_2[i] = change_2 is **true and** A_{row}[i] is 1
    if term_{min} is **false and** is_zero[i] is **true then**
      changes_1[i] = 0
    end if
  end for
  if γ_s is **true then**
    candidates, changes = candidates_2, changes_2 *(17) Select candidates/changes* [C.6.2]
  else
    candidates, changes = candidates_1, changes_1
  end if
  for $i = 1$ **to** n **do**
    if changes[i] is **true then**
      prev[i], dists[i] = node, candidates[i] *(18) Update variables* [C.2.8]
    end if
  end for
  term = **not** (**false** in visit) *(19) Trigger termination* [C.2.10]
end while
return prev, dists