EDGE Algorithm Write-Up: ⟨Direction Optimizing BFS⟩

Sanjana S. Mali, Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Joel S. Emer, John D. Owens

1 Introduction

Direction-Optimizing Breadth-First Search, introduced by Beamer et al. [1], combines multiple traversal strategies to improve performance on low-diameter graphs.

In this work, we present a structured EDGE (Extended General Einsums) abstraction [3] formulation of the algorithm by walking through three parts:

- the classical *Top-Down* BFS,
- the *Bottom-Up* BFS variant, and
- the full *Direction-Optimizing (Hybrid)* algorithm that dynamically switches between the two, focusing on Beamer's original switching heuristic [1].

For each strategy, we first describe the algorithm in graph-theoretic terms and then translate it into its corresponding EDGE representation, making the connection between traversal intuition and algebraic formulation explicit.

With EDGE, we can apply a separation of concerns approach to understand the algorithm's design choices [2]. Using the TeAAL pyramid of abstractions [2], EDGE is the specification of the algorithmic space. Given an algorithm, we can then identify the space of *mapping* choices (e.g., traversal order, parallelism, data layout). A separate concern is the *format* space, representing data structures used for the EDGE tensors. Finally, hardware-specific concerns are captured in the *implementation* space, which includes optimizations and low-level details. This approach helps us understand the key aspects of an algorithm and which design choices at each level of the pyramid are responsible for its performance characteristics.

Traditionally, Bottom-Up BFS requires maintaining additional data structures (e.g., parent pointers) to enable efficient backward exploration, which can introduce overhead. Our EDGE formulation shows that algorithmically, this information is readily available within the Einsum, and the key difference boils down to an algebraic manipulation of Top-Down BFS combined with a mapping change (traversal order). Additionally, *switching* between the two strategies is not free, as it involves an additional algorithmic step to determine when to switch, and potentially requires changes to data formats or traversal order, which can introduce overhead.

1.1 Why This Write-Up?

This write-up belongs to the EDGE Zoo, a growing collection of worked examples that each express a well-known algorithm as an extended Einsum cascade. The Zoo targets a variety of algorithms, and each write-up follows the same structure so that the specifications can be read against one another.

A worked example, not an algorithm tutorial. We assume the reader already knows BFS. The emphasis here is on the *translation*: Section 4 runs one small graph through a conventional traversal trace and then through the same trace expressed as tensors, and every cascade is followed by a walk-through explaining each line in both graph-algorithm terms and Einsum terms. Reading the two views against each other is meant to make the notation concrete in a way that a bare specification cannot.

Switching is part of the specification. Beyond expressing plain BFS, we can also write the switch between the two directions in EDGE. The work estimates and the threshold comparison are Einsums over the same tensors the traversal already uses, and the choice of direction is made with a macro and a conditional cascade, so the traversal and the switching are written together in one specification (Section 10). We can express the other switching heuristics the same way: Section 14 writes the rules used by Gunrock, Ligra, and XBFS as EDGE cascades, which lets us compare them side by side.

2 Background

2.1 Classical BFS in EDGE

Although the focus of this work is on expressing Direction-Optimizing BFS in EDGE, we begin by revisiting the classical top-down and bottom-up BFS formulations.

The goal here is not to reintroduce BFS, but to establish a simple and consistent baseline using EDGE notation. This allows us to separate the core BFS logic from additional complexities such as parent tracking and switching heuristics, which are introduced later.

By first expressing BFS using a visited mask, we can more clearly understand how the bottom-up traversal relates to the top-down formulation, and how both can be derived from the same underlying computation. This also provides a useful reference point for the more complex formulations that follow.

2.2 Top-Down (Push BFS)

We begin with the classical top-down BFS, where we expand outward from the current frontier. At each iteration, every vertex in the frontier looks at its outgoing edges to discover new vertices.

In simple terms, we take all neighbors of the current frontier and add those that have not been visited before to the

next frontier. This process continues level by level until no new vertices are found.

To express this in EDGE, we use three main components:

- The graph G, which tells us which vertices are connected,
- The frontier F, which stores the vertices we are currently exploring (active set), and
- A visited mask P, which keeps track of which vertices have already been discovered.

Using these, we can directly write the EDGE formulation of top-down BFS:

▹ **Tensors**

$$G^{S,D} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,S} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$P^{I,S} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$P_{i,s} = \texttt{false}$$
$$G_{s,d} \longrightarrow \langle\text{user-specified}\rangle$$
$$F_{0,n} = \texttt{true}$$
$$P_{0,n} = \texttt{true}$$

▹ **Extended Einsums**

$$T_{i,d} = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap)\ \bigvee \text{OR}(\cup) \quad (2.1)$$
[Find neighbors of the current frontier]

$$F_{i+1,d} = T_{i,d} \cdot \neg P_{i,d} :: \bigwedge \text{AND}(\cap) \quad (2.2)$$
[Keep only vertices not yet visited]

$$P_{i+1,d} = P_{i,d} \cdot F_{i+1,d} :: \bigwedge \text{OR}(\cup) \quad (2.3)$$
[Update visited vertices]

$$\diamond :\ \|F_{i+1}\| \equiv 0 \quad (2.4)$$
[Stop when no new vertices are found]

Cascade 1. Top-Down BFS EDGE Cascade

This matches the standard BFS algorithm, written using EDGE. A more detailed explanation of this formulation is given in [4].

2.3 Bottom-Up (Pull BFS)

In the bottom-up traversal, we take a different view compared to top-down BFS. Instead of expanding outward from the frontier, we look at each vertex that has not yet been visited and check whether it has a parent in the current frontier.

In simple terms, for every unvisited vertex d, we ask: "Is there some vertex s in the frontier such that (s, d) exists?" If yes, then d is added to the next frontier.

This changes the order of computation: instead of starting from the frontier, we conceptually start from unvisited vertices and look for parents in the frontier.

EDGE formulation. We use the same tensors as in the top-down case:

- G for the graph,
- F for the frontier, and
- P for the visited mask.

▹ **Tensors**

$$G^{S,D} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,S} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$P^{I,S} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$P_{i,s} = \texttt{false}$$
$$G_{s,d} \longrightarrow \langle\text{user-specified}\rangle$$
$$F_{0,n} = \texttt{true}$$
$$P_{0,n} = \texttt{true}$$

▹ **Extended Einsums**

$$T_{i,s,d} = G_{s,d} \cdot \neg P_{i,d} :: \bigwedge \text{AND}(\cap) \quad (2.5)$$
[Find parents (incoming edges) of unvisited vertices]

$$F_{i+1,d} = T_{i,s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap)\ \bigvee \text{OR}(\cup) \quad (2.6)$$
[Check if any parent(s) of child(d) is in the frontier]

$$P_{i+1,d} = P_{i,d} \cdot F_{i+1,d} :: \bigwedge \text{OR}(\cup) \quad (2.7)$$
[Update visited set]

$$\diamond :\ \|F_{i+1}\| \equiv 0 \quad (2.8)$$
[Stop when no new vertices are found]

Cascade 2. Bottom-Up BFS EDGE Cascade

Einsum walkthrough. The first expression restricts the graph to edges whose destination vertex d has not yet been visited. This can be viewed as creating a subgraph over the *unvisited* vertices ($\neg P_{i,d}$).

The second expression checks, for each such vertex d, whether any of its parents s lie in the current frontier. If at least one such parent exists, d is added to the next frontier.

This matches the bottom-up intuition: instead of exploring outward from the frontier, each unvisited vertex checks whether it can be reached from the frontier.

Although this formulation uses the same tensors as the top-down case, the key difference is that we first restrict to unvisited vertices (before reduction), which changes the direction of the computation.

2.4 From Visited Mask to Parent Tracking

The formulations above describe BFS using an explicit visited mask P, which keeps track of whether a vertex has already been discovered.

In the Direction-Optimizing BFS of Beamer et al. [1], this information is not stored explicitly. Instead, the algorithm maintains a *parent array*, where each vertex records the vertex that first discovered it.

In this formulation, a vertex is considered visited if it has a parent recorded, and unvisited otherwise. In other words, the visited set is implicit in the parent array:

- If a vertex has a parent, it has been visited.
- If a vertex does not yet have a parent, it is unvisited.

This representation is used consistently in both the top-down and bottom-up phases of Beamer's algorithm. Even in the top-down case, where we conceptually expand from the frontier, the algorithm still records which parent discovered each vertex.

The advantage of this formulation is that it not only tracks reachability (as in the visited mask), but also captures the structure of the BFS tree. At the same time, it provides a natural way to express the bottom-up traversal, where each vertex checks whether any of its parents are in the frontier.

To express this behavior in EDGE, we replace the visited mask with a tensor that explicitly records parent-child relationships, *Tree*. This leads to the formulation described in the next section.

3 Problem Statement

3.1 Problem Description

Given an unweighted graph $G = (V, E)$ and a source vertex $s \in V$, the goal is to discover all vertices reachable from s and construct a Breadth-First Search (BFS) tree. The BFS tree records, for each reachable vertex, a parent vertex that discovered it during traversal.

The Direction-Optimizing BFS algorithm by Beamer et al. [1] improves performance on large graphs by dynamically switching between two traversal strategies:

- **Top-Down:** Expand outward from the current frontier (the set of vertices discovered at the current level, i.e., the active set) to discover new vertices.
- **Bottom-Up:** For each unvisited vertex, check whether any of its neighbors (parents) are in the current frontier.

The algorithm selects the traversal direction based on the size of the frontier relative to the remaining unvisited vertices. Unlike top-down traversal, the bottom-up approach traditionally requires explicitly tracking parent information to determine whether a vertex has already been visited. (Beamer et al. [1], as discussed in the previous section, use a parent array for both top-down and bottom-up formulations.) Additionally, switching between the two strategies is not free, as it introduces overhead in evaluating when to switch and in maintaining the required data structures.

3.2 Inputs

- An unweighted graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges.
- A source vertex $s \in V$.

Assumptions:

- The graph may be directed or undirected.
- The graph is unweighted.

3.3 Outputs

- The set of vertices reachable from s.
- A BFS tree recording, for each reachable vertex $v \neq s$, a parent that discovered v.

3.4 Algorithmic Steps

High-Level Algorithm (Direction-Optimizing BFS)

1. Initialize the frontier with the source vertex s.
2. Mark s as visited and record it as its own parent.
3. While the frontier is not empty:
   a. If the frontier is small relative to the number of unvisited vertices, perform a **Top-Down** step:
      - For each vertex in the frontier, explore its outgoing edges. Each edge connects to a neighbor vertex.
      - For each unvisited neighbor, mark it visited, record its parent, and add it to the next frontier.
   b. Otherwise, perform a **Bottom-Up** step:
      - For each unvisited vertex, check whether any of its parents (incoming edges) are in the frontier.
      - If so, mark it visited, record one such neighbor as its parent, and add it to the next frontier.
   c. Replace the current frontier with the newly discovered vertices.
4. Terminate when the frontier becomes empty.

4 Algorithm Execution Flow: Graph View

Example Graph

Consider the following unweighted graph:

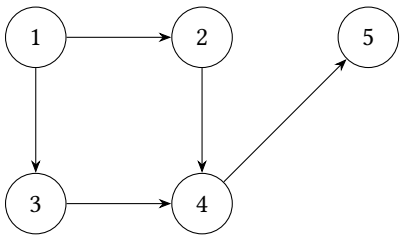

Figure 1. Example graph for Direction-Optimizing BFS

Let the source vertex be $s = 1$.

4.1 Top-Down Traversal

We now simulate BFS starting from vertex 1.

Iteration 0.

- Frontier: {1}
- Parent: 1 is its own parent

Iteration 1. From vertex 1, explore all outgoing edges.

- Neighbors discovered: {2, 3}
- Frontier becomes: {2, 3}
- Parents recorded:
  - Parent(2) = 1
  - Parent(3) = 1

Iteration 2. From vertices 2 and 3:

- Vertex 2 discovers 4
- Vertex 3 also reaches 4, but 4 is already discovered
- Frontier becomes: {4}
- Parent(4) = 2 (the first discoverer)

Iteration 3. From vertex 4:

- Neighbor discovered: {5}
- Frontier becomes: {5}
- Parent(5) = 4

Iteration 4. Vertex 5 has no new neighbors.

- Frontier becomes empty
- Algorithm terminates

4.2 When Would Bottom-Up Be Used?

In large graphs, when the frontier becomes very large (for example, when it contains a significant fraction of the vertices), it may be more efficient to reverse the search direction [1].

The key advantage of the bottom-up approach is that, for each unvisited vertex, the search can terminate as soon as a single parent is found (i.e., early exit). That is, once a vertex identifies one parent in the frontier, it does not need to examine any of its remaining incoming edges.

In contrast, the top-down approach explores all outgoing edges from the frontier, requiring every such edge to be examined. As a result, the bottom-up strategy can avoid visiting many edges entirely, leading to improved performance when the frontier is large.

4.3 Bottom-Up Traversal

We now simulate BFS starting from vertex 1 using the Bottom-Up strategy.

Iteration 0.

- Frontier: {1}
- Parent: 1 is its own parent

Iteration 1. Instead of expanding from the frontier, we examine each unvisited vertex and check whether it has a parent in the current frontier {1}.

- Vertex 2 checks its parents and finds 1 in the frontier
- Vertex 3 checks its parents and finds 1 in the frontier
- Vertices 4 and 5 also check their parents, but do not find any vertex in the frontier
- Frontier becomes: {2, 3}
- Parents recorded:
  - Parent(2) = 1
  - Parent(3) = 1

Iteration 2. Now the frontier is {2, 3}. Again, each unvisited vertex checks whether any of its parents are in the frontier.

- Vertex 4 finds that 2 is in the frontier
- Although 3 also connects to 4, the first detected parent is chosen
- Frontier becomes: {4}
- Parent(4) = 2

Iteration 3. Frontier is now {4}. Unvisited vertices check for parents in the frontier.

- Vertex 5 finds that 4 is in the frontier
- Frontier becomes: {5}
- Parent(5) = 4

Iteration 4.

- No unvisited vertex finds a parent in the frontier, so the next frontier is empty
- The algorithm terminates, indicating that no further vertices can be discovered

5 Algorithm Execution Flow: EDGE Tensor View

We now translate the same BFS example from Section 4 into EDGE tensors. The goal of this section is to show what each tensor represents and how the tensor values change across iterations, while gradually building toward the full EDGE expression.

5.1 Tensors Used

We use the following tensors (matching the Full EDGE description):

- $G^{S,D} \longrightarrow$ Boolean: adjacency matrix of the graph. An (s, d) point in G contains a `true` value if and only if there is an edge $s \longrightarrow d$.
- $F^{I,S} \longrightarrow$ Boolean: frontier indicator. $F_{i,s} = \texttt{true}$ iff vertex s is in the frontier at iteration i.
- $Tree^{I,S,D} \longrightarrow$ Boolean: BFS tree. $Tree_{i,s,d} = \texttt{true}$ means vertex s is recorded as the parent of vertex d by iteration i. Note that this is simply the tensor representation. Actual storage (the *format* space in the pyramid of abstractions [2]) may be optimized for sparsity; for example, as a list of parent–child pairs that is updated iteratively.

5.2 Why use a Tree tensor?

The Tree tensor comes from a simple design choice we follow in this work: each set of vertices that plays a particular role in the algorithm corresponds to a named tensor rank. For example, in the adjacency matrix $G^{S,D}$, the rank S indexes into source (parent) vertices and the rank D indexes into destination (child) vertices. A coordinate along S identifies a specific parent, and a coordinate along D identifies a specific child. This convention carries through all tensors: the frontier $F^{I,S}$ uses a named rank S because it indexes the same set of vertices in their role as potential parents.

Relationships between vertices must also be represented using multiple ranks. A parent-child relationship involves two vertices, so to represent the parent array structure we use a two-dimensional tensor:

$$Tree^{S,D}_{s,d} = \texttt{true}$$

means that vertex s is the parent of vertex d.

The tensor representation makes the relationship between vertices explicit and fits naturally within the EDGE formulation.

In practice, this tensor represents the same information as a parent array, and at the *format* level of abstraction, will likely be stored as such (see Section 1).

5.3 Graph Tensor G for the Example

For the example graph in Figure 1, the adjacency matrix G has `true` entries at:

$$(1, 2),\ (1, 3),\ (2, 4),\ (3, 4),\ (4, 5).$$

All other entries are `false`.

5.4 Top-Down: One Iteration in EDGE Form

We demonstrate how an iteration is computed using the Top-Down strategy.

We will use intermediate tensors that appear in the Full EDGE expression: *NV* (Not Visited vertices), *Nei* (neighbors of frontier), *NeiNV* (neighbors of frontier that are Not Visited), and *CandTree*/*Temp* (temporary tree updates).

Iteration 0: Initialize Frontier and Tree

The initial frontier contains only the source:

$$F_{0,1} = \texttt{true}, \quad F_{0,v} = \texttt{false} \text{ for } v \neq 1.$$

We also initialize the tree such that the source is its own parent:

$$Tree_{0,1,1} = \texttt{true}.$$

The above equation says "At iteration 0, record vertex 1 as the parent of itself."

Step A: Identify Not Visited Vertices (NV)

To determine whether a vertex is still Not Visited, we check whether it has any parent recorded in *Tree*. Reducing $Tree_{i,p,c}$ over the parent index p and negating the result yields a Boolean value for each vertex c:

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \mathrm{OR}(\cup) \tag{5.1}$$

So at iteration 0:

$$NV_{0,1} = \texttt{false}, \quad NV_{0,2} = NV_{0,3} = NV_{0,4} = NV_{0,5} = \texttt{true}.$$

Step B: Gather Neighbors of the Frontier (Nei)

Next we compute which destination vertices d are adjacent to at least one vertex in the frontier s:

$$Nei_{i,d} = G_{s,d} \cdot F_{i,s} :: \bigwedge \mathrm{AND}(\cap) \ \bigvee \mathrm{OR}(\cup) \tag{5.2}$$

At iteration 0, the frontier contains only vertex 1, so $Nei_{0,d}$ is true exactly for the outbound neighbors of 1 (vertices 2 and 3):

$$Nei_{0,2} = \texttt{true}, \quad Nei_{0,3} = \texttt{true}, \quad Nei_{0,4} = \texttt{false}, \quad Nei_{0,5} = \texttt{false}.$$

Step C: Filter to New Discoveries (NeiNV)

We only want newly discovered vertices, i.e., neighbors of the frontier that are still Not Visited. This is done by intersecting *Nei* from Eq. (5.2) with *NV* from Eq. (5.1):

$$NeiNV_{i,d} = Nei_{i,d} \cdot NV_{i,d} :: \bigwedge \mathrm{AND}(\cap) \tag{5.3}$$

At iteration 0, vertices 2 and 3 are Not Visited and are neighbors of the frontier, so:

$$NeiNV_{0,2} = \texttt{true}, \quad NeiNV_{0,3} = \texttt{true},$$

and all other entries are false.

Step D: Record Parent Links in the Tree (CandTree, Temp, Tree Update)

To record parent-child edges for newly discovered vertices, we form a temporary tree tensor:

$$CandTree_{i,s,d} = (G_{s,d} \cdot F_{i,s})_{i,s,d} \cdot NeiNV_{i,d} :: \bigwedge \text{AND}(\cap) \bigwedge \text{AND}(\cap) \tag{5.4}$$

At iteration 0, only $s = 1$ is in the frontier, and the new children are $d \in \{2, 3\}$. Thus $CandTree_{0,1,2}$ and $CandTree_{0,1,3}$ become true (since edges $(1, 2)$ and $(1, 3)$ exist).

If multiple parents discover the same child in the same iteration, we select exactly one parent using the *populate* action [3]. See Section 12 for a detailed explanation of the populate action.

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{5.5}$$

Finally, we merge these new parent assignments into the main tree:

$$Tree_{i+1,s,d} = Tree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{5.6}$$

After iteration 0, the tree includes true at (s, d) locations:

$$(1, 1),\ (1, 2),\ (1, 3).$$

Step E: Update Frontier

The next frontier is obtained directly from the tensor $NeiNV_{i,d}$, which identifies all newly discovered vertices.

$$F_{i+1,d} = NeiNV_{i,d}. \tag{5.7}$$

Thus, the vertices that were newly discovered in this iteration become the frontier for the next iteration. For this example:

$$F_{1,2} = \texttt{true}, \quad F_{1,3} = \texttt{true}.$$

Termination is detected when the newly computed frontier becomes empty:

$$\diamond : \ \|F_{i+1}\| \equiv 0. \tag{5.8}$$

5.5 Bottom-Up: One Iteration in EDGE Form

We now demonstrate how the same iteration (starting from frontier $\{1\}$) is computed using the Bottom-Up strategy.

Intuitively, the bottom-up traversal asks: "For each Not Visited vertex, does it have a parent in the current frontier?"

We reuse the tensors G, F, $Tree$, and NV defined above.

Iteration 0 (Bottom-Up)

The initial frontier and tree are the same:

$$F_{0,1} = \texttt{true}, \quad Tree_{0,1,1} = \texttt{true}.$$

From Step A above (Eq. (5.1)), we already computed:

$$NV_{0,2} = NV_{0,3} = NV_{0,4} = NV_{0,5} = \texttt{true}.$$

Thus, vertices $\{2, 3, 4, 5\}$ are currently Not Visited.

Step B (Bottom-Up): Examine Unvisited Vertices

Instead of expanding from the frontier, the bottom-up strategy scans each Not Visited vertex d and checks whether it has any parent s (i.e., a vertex with an edge (s, d)) that lies in the current frontier.

Here, we first find the parents of the Not Visited vertices:

$$CandParents_{i,s,d} = G_{s,d} \cdot NV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{5.9}$$

At any iteration, the tensor $CandParents$ represents all candidate parent-child edges (s, d) where d is currently Not Visited (vertices $\{2, 3, 4, 5\}$).

Next we check which of those candidate parent vertices s lie in the current frontier.

$$InF_{i,s,d} = CandParents_{i,s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \tag{5.10}$$

Since only vertex 1 is in the frontier at iteration 0, $InF_{0,1,2}$ and $InF_{0,1,3}$ become true (because edges $(1, 2)$ and $(1, 3)$ exist).

No other InF entries are true.

While this expression is written over all (s, d) pairs, the actual bottom-up algorithm is implemented in a vertex-centric manner, where each Not Visited vertex examines its parents and may terminate early once one parent in the frontier is found. This optimization is not captured in the current formulation and is discussed separately in Section 13.

Step C: Select Parent and Update Tree

If multiple parents are found for a vertex in the same iteration, we select exactly one parent among the candidate parents using the populate action. Here, we select the first candidate parent encountered for each child vertex. See Section 12 for a detailed explanation of the populate action.

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{5.11}$$

Here only one valid parent exists for each of 2 and 3, so $Temp_{0,1,2}$ and $Temp_{0,1,3}$ are true.

We update the tree:

$$Tree_{i+1,s,d} = Temp_{i,s,d} \cdot Tree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{5.12}$$

Now the tree contains:

$$(1, 1),\ (1, 2),\ (1, 3).$$

Step D: Update Frontier

The next frontier consists of all vertices that found at least one parent in the frontier. This is obtained by reducing InF over the parent index:

$$F_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{5.13}$$

Thus:

$$F_{1,2} = \texttt{true}, \quad F_{1,3} = \texttt{true}.$$

This matches the Top-Down result for this iteration, but the computation order was reversed: instead of expanding from vertex 1, we scanned Not Visited vertices and checked whether they have a parent in the frontier.

As in standard BFS, the algorithm terminates when the newly computed frontier becomes empty:

$$\diamond : \ \|F_{i+1}\| \equiv 0. \tag{5.14}$$

Notes on Optimization Opportunities

The formulations presented in this section describe BFS in a conceptual and algebraic manner using EDGE tensors. These tensors capture the computation at a high level, but do not prescribe how the data must be stored in memory.

In practice, BFS is an iterative process where each step depends only on the previous iteration. As a result, it is not necessary to materialize all iterative tensors. For example, we can maintain the frontier using a ping-pong buffer:

$$F_{(i+1)\%2,d} = G_{s,d} \cdot F_{i\%2,s} \cdot NV_{i\%2,d} :: \bigwedge \text{AND}(\cap) \ \bigvee \text{OR}(\cup). \tag{5.15}$$

Thus, an implementation need only store the data required for the current and next iterations. A similar optimization is possible for the *Tree* tensor.

These observations highlight that the EDGE formulation captures the full computation for clarity, while practical implementations can apply standard optimizations such as double-buffering, sparsity-aware storage, and in-place updates.

6 EDGE Expression (Top-Down)

The full EDGE expression for top-down Direction-Optimizing BFS is as follows:

▹ **Tensors**

$$G^{S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean, empty} = \texttt{false}$$
$$F^{I,\ S\equiv|V|} \rightarrow \text{Boolean, empty} = \texttt{false}$$
$$Tree^{I,\ S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean, empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$Tree_{i,s,d} = \texttt{false}$$
$$G_{s,d} \rightarrow \langle\text{user-specified}\rangle$$
$$F_{0,n} = \texttt{true}$$
$$Tree_{0,n,n} = \texttt{true}$$

▹ **Extended Einsums**

$$Nei_{i,d} = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee \text{OR}(\cup) \tag{6.1}$$

[Gather neighbors of nodes in frontier]

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{6.2}$$

[Find unvisited nodes from the tree]

$$NeiNV_{i,d} = Nei_{i,d} \cdot NV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{6.3}$$

[Filter neighbors that are not yet visited]

$$CandTree_{i,s,d} = (G_{s,d} \cdot F_{i,s})_{i,s,d} \cdot NeiNV_{i,d} :: \bigwedge \text{AND}(\cap) \ \bigwedge \text{AND}(\cap) \tag{6.4}$$

[Record candidate parent-child edges]

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{6.5}$$

[Select exactly one parent per child]

$$Tree_{i+1,s,d} = Tree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{6.6}$$

[Update BFS tree]

$$F_{i+1,d} = NeiNV_{i,d} \tag{6.7}$$

[Set next frontier]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{6.8}$$

[Terminate when frontier becomes empty]

Cascade 3. Top-Down DO-BFS EDGE Cascade

7 EDGE Expression Walk-Through (Top-Down)

This section walks through the Full EDGE expression for Direction-Optimizing BFS (Top-Down) line by line. For each expression, we explain (1) what it means in graph-algorithm terms, and (2) what it means as an Einsum-style tensor computation. We use the same tensor names and operators as in the Full EDGE description.

Step 1: Gather neighbors of the current frontier (Nei). In the top-down direction, we expand outward from active frontier vertices. A vertex d is considered a neighbor if there exists at least one frontier vertex s such that an edge $s \rightarrow d$ exists.

Graph meaning: "From all vertices in the frontier, collect all outgoing neighbors."

Einsum meaning: Elementwise AND between $G_{s,d}$ and $F_{i,s}$ produces a tensor that identifies all candidate edges (s, d) from the current frontier. Reducing over s using OR then

determines whether each vertex d has at least one such incoming edge, thereby gathering all neighbors of the frontier.

$$Nei_{i,d} = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \bigvee \text{OR}(\cup) \tag{7.1}$$

Step 2: Compute the set of Not Visited vertices (NV). A vertex is considered Not Visited when it does not yet have any parent recorded in the BFS tree. We detect this by reducing $Tree_{i,p,c}$ over the parent index p and then negating the result.

Graph meaning: "Find every vertex that does not yet have a parent in the BFS tree."

Einsum meaning: Reduce the boolean tensor *Tree* over index p using `OR`, then negate the resulting mask.

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{7.2}$$

Step 3: Filter out previously visited neighbors (NeiNV). We only want to visit new vertices in the current iteration. Intersecting *Nei* from Eq. (7.1) with the Not Visited mask *NV* from Eq. (7.2) keeps only those neighbors that are not yet visited.

Graph meaning: "Keep only neighbors that have not been discovered before."

Einsum meaning: Elementwise `AND` between $Nei_{i,d}$ and $NV_{i,d}$.

$$NeiNV_{i,d} = Nei_{i,d} \cdot NV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{7.3}$$

Step 4: Record parent-child relationships (CandTree). For each newly discovered child d, we want to record which frontier vertex or vertices s discovered it. A parent-child entry should be set when:

- s is in the frontier at iteration i,
- an edge $s \to d$ exists in the graph, and
- d is newly discovered in this iteration.

Graph meaning: "Record all parents for newly discovered vertices from the frontier."

Einsum meaning: Compute $(G_{s,d} \cdot F_{i,s})$ and broadcast it into the (i, s, d) space, then `AND` with $NeiNV_{i,d}$.

$$\begin{aligned} CandTree_{i,s,d} &= (G_{s,d} \cdot F_{i,s})_{i,s,d} \cdot NeiNV_{i,d} \\ &:: \bigwedge \text{AND}(\cap) \bigwedge \text{AND}(\cap) \end{aligned} \tag{7.4}$$

Step 5: Select exactly one parent when there are multiple candidates (Temp). A newly discovered vertex may be a child of multiple frontier vertices. BFS stores only one parent per vertex. The populate action enforces this by selecting a single parent index s^* for each child d. Here, we decide a parent by choosing the one with the smallest index. See Section 12 for details on the implementation of populate with the coordinate operator `pick-parent`.

Graph meaning: "If multiple frontier vertices discover the same child, choose the one with the smallest index as the parent."

Einsum meaning: Populate a new tensor by applying a coordinate operator `pick-parent` to choose one s per d.

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{7.5}$$

Step 6: Update the BFS tree (Tree). We merge the newly selected parent assignments with the existing tree so that parents recorded in previous iterations are preserved.

Graph meaning: "Add the newly discovered parent-child edges into the BFS tree."

Einsum meaning: Elementwise `OR` between the previous tree and the new assignments.

$$Tree_{i+1,s,d} = Tree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{7.6}$$

Step 7: Update the frontier for the next iteration. The next frontier is exactly the set of newly discovered vertices.

Graph meaning: "Set the next frontier to all newly discovered vertices."

Einsum meaning: Copy the boolean mask $NeiNV_{i,d}$ into $F_{i+1,d}$.

$$F_{i+1,d} = NeiNV_{i,d} \tag{7.7}$$

Step 8: Terminate when no new vertices are discovered. The BFS traversal stops when the new frontier is empty.

Graph meaning: "Stop when there are no newly discovered vertices."

Einsum meaning: Check whether any entry in F_{i+1} is true.

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{7.8}$$

8 EDGE Expression (Bottom-Up)

The full EDGE expression for bottom-up Direction-Optimizing BFS is as follows:

▹ **Tensors**

$$G^{S\equiv|V|,\ D\equiv|V|} \to \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,\ S\equiv|V|} \to \text{Boolean},\ \text{empty} = \texttt{false}$$
$$Tree^{I,\ S\equiv|V|,\ D\equiv|V|} \to \text{Boolean},\ \text{empty} = \texttt{false}$$
$$CandParents^{I,\ S,\ D} \to \text{Boolean},\ \text{empty} = \texttt{false}$$
$$InF^{I,\ S,\ D} \to \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$Tree_{i,s,d} = \texttt{false}$$
$$G_{s,d} \longrightarrow \langle \text{user-specified} \rangle$$
$$CandParents_{i,s,d} = \texttt{false}$$
$$InF_{i,s,d} = \texttt{false}$$
$$F_{0,n} = \texttt{true}$$
$$Tree_{0,n,n} = \texttt{true}$$

▹ **Extended Einsums**

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{8.1}$$
[Find unvisited nodes directly from the tree]

$$CandParents_{i,s,d} = G_{s,d} \cdot NV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{8.2}$$
[Find candidate parents of Not Visited nodes]

$$InF_{i,s,d} = CandParents_{i,s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \tag{8.3}$$
[Check whether any parent is in frontier (InF)]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{8.4}$$
[Select exactly one parent per child]

$$Tree_{i+1,s,d} = Temp_{i,s,d} \cdot Tree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{8.5}$$
[Update BFS tree]

$$F_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{8.6}$$
[Set next frontier]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{8.7}$$
[Terminate when frontier becomes empty]

Cascade 4. Bottom-Up DO-BFS EDGE Cascade

9 EDGE Expression Walk-Through (Bottom-Up)

Now we write the EDGE expressions for the Bottom-Up traversal, with English explanations for what it means in the application domain, and what it means as Einsums. In the bottom-up traversal, instead of expanding outward from the frontier, we iterate over Not Visited vertices and check whether any of their parents are in the current frontier. This reverses the direction of exploration and can significantly reduce edge examinations when the frontier becomes large.

Step 1: Compute Not Visited vertices (NV). As in the top-down case, we derive the Not Visited set directly from the BFS tree.

Graph meaning: "Find vertices that have not yet been discovered."

Einsum meaning: Reduce the Tree tensor over the parent index using `OR`, then negate the result.

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{9.1}$$

Step 2: Find candidate parents of Not Visited vertices (CandParents). Instead of expanding from the frontier, we now examine each Not Visited vertex d and find the vertices s that are parents to it.

Graph meaning: "For each Not Visited vertex, identify all of its potential parents." Note that this differs from the top-down traversal. In the top-down case, edges are explored with frontier vertices acting as sources, discovering their outgoing neighbors. Here the perspective is reversed: for each Not Visited vertex d, we examine its incoming edges (s, d) to determine whether any adjacent vertex s could serve as its parent.

Einsum meaning: Elementwise `AND` between the graph tensor and the Not Visited mask, $CandParents_{i,s,d}$ is true exactly when the edge (s, d) exists and d is currently Not Visited.

$$CandParents_{i,s,d} = G_{s,d} \cdot NV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{9.2}$$

Step 3: Check whether any of those parents are in the frontier (InF). For each Not Visited vertex d, we now check whether any of its parent vertices s are currently active in the frontier.

Graph meaning: "A Not Visited vertex becomes discovered if at least one of its parents is in the frontier."

Einsum meaning: Intersect $CandParents$ with the frontier tensor F on rank variable s, keeping only edges where the source is in the frontier.

$$InF_{i,s,d} = CandParents_{i,s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \tag{9.3}$$

Step 4: Select one parent per newly discovered vertex (Temp). If multiple frontier vertices are adjacent to the same Not Visited vertex, we must choose exactly one as its parent. Here we will use the populate action with a coordinate operator `pick-parent` to select a parent. A common optimization in bottom-up exit is early-exit, where a vertex can stop examining its parents as soon as it finds one in the frontier. Thus, the only constraint on `pick-parent` is that it must select at least one parent from the frontier if any exist, but it can be implemented in a way that allows for early-exit. For example, one may use the smallest-index parent as a

deterministic tie-breaking rule, although any valid parent would produce a correct BFS tree.

Graph meaning: "Choose a single parent for each newly discovered vertex."

Einsum meaning: Use the populate action with the coordinate operator `pick-parent` to enforce one parent per child.

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{9.4}$$

Step 5: Update the BFS tree. Merge the newly selected parent-child edges into the existing tree.

Graph meaning: "Record newly discovered vertices in the BFS tree."

Einsum meaning: Elementwise `OR` between previous *Tree* and *Temp*.

$$Tree_{i+1,s,d} = Temp_{i,s,d} \cdot Tree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{9.5}$$

Step 6: Update the frontier. The next frontier consists of vertices that were newly discovered in this iteration. These are exactly the d vertices for which at least one entry in $InF_{i,s,d}$ was true.

Graph meaning: "The new frontier contains vertices that were just discovered."

Einsum meaning: Reduce *InF* over the parent dimension using `OR`. Note that we can early-exit here as well: `OR` is a *bear-trap* function, meaning that it can return `True` as soon as it encounters the first `True` entry. Thus, we can stop checking parents for a child as soon as we find one in the frontier.

$$F_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{9.6}$$

Step 7: Termination condition. The algorithm stops when no new vertices are added to the frontier, i.e., when the frontier tensor contains no `true` entries (equivalently, the frontier is empty).

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{9.7}$$

9.1 Comparing Top-Down and Bottom-Up in DO-BFS

In the Direction-Optimizing BFS of Beamer et al. [1], both top-down and bottom-up traversals operate on the same underlying tensors, including the frontier and the Tree tensor that encodes parent-child relationships.

In both cases, the goal is to identify edges (s, d) such that s lies in the current frontier and d does not yet have a parent. This condition is the same in both formulations.

In the top-down phase, the computation is organized around the source rank s. The Einsum first restricts the graph using the frontier along s, and then filters destinations d that do not yet have a parent. Conceptually, this can be written as:

$$CandTree_{i,s,d} = \left(G_{s,d} \cdot F_{i,s}\right) \cdot \text{NV}_{i,d}, \tag{9.8}$$

where $\text{NV}_{i,d}$ is true exactly when d has not yet been visited.

In contrast, the bottom-up phase is organized around the destination rank d. The Einsum first restricts attention to vertices that do not yet have a parent, and then checks whether any parent s lies in the frontier:

$$CandTree_{i,s,d} = \left(G_{s,d} \cdot NV_{i,d}\right) \cdot F_{i,s}. \tag{9.9}$$

Although these expressions look similar, the difference lies in how they are evaluated. In the top-down case, the frontier mask $F_{i,s}$ is applied first, selecting edges outgoing from the frontier. In the bottom-up case, the mask $NV_{i,d}$ is applied first, restricting the computation to vertices that do not yet have a parent.

This leads to the following key differences:

- **Traversal order:** top-down evaluates edges as $s \to d$, while bottom-up evaluates them as $d \to s$,
- **Masking:** top-down applies the frontier mask on s first, while bottom-up applies the Not Visited mask on d first,
- **Reduction:** both formulations reduce over s to determine whether a valid parent exists, but this reduction occurs after different masking steps.

From the EDGE perspective, this shows that the two traversal strategies are not fundamentally different. They correspond to different evaluation orders of the same underlying computation. This is why bottom-up BFS can be derived from top-down BFS through algebraic reorganization and changes in traversal order.

10 EDGE Expression (Hybrid)

The full EDGE expression for the hybrid Direction-Optimizing BFS is as follows:

▹ **Tensors**

$G^{S\equiv|V|,\ D\equiv|V|} \to$ Boolean, empty = `false`

$F^{I,\ S\equiv|V|} \to$ Boolean, empty = `false`

$Tree^{I,\ S\equiv|V|,\ D\equiv|V|} \to$ Boolean, empty = `false`

$CandParents^{I,\ S,\ D} \to$ Boolean, empty = `false`

$InF^{I,\ S,\ D} \to$ Boolean, empty = `false`

$TopDown^{I} \to$ Boolean, empty = `false`

▹ **Initializations**

$$
\begin{aligned}
F_{i,s} &= \texttt{false} \\
Tree_{i,s,d} &= \texttt{false} \\
G_{s,d} &\rightarrow \langle \text{user-specified} \rangle \\
CandParents_{i,s,d} &= \texttt{false} \\
InF_{i,s,d} &= \texttt{false} \\
F_{0,n} &= \texttt{true} \\
Tree_{0,n,n} &= \texttt{true} \\
TopDown_0 &= \texttt{true}
\end{aligned}
$$

▹ **Extended Einsums**

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{10.1}$$

[Find Not Visited nodes from the tree]

$$
\begin{aligned}
Tree_{i+1,s,d}, \\
F_{i+1,d}, \\
TopDown_{i+1}
\end{aligned}
= \begin{cases}
\text{TOP-DOWN}(G_{s,d}, F_{i,s}, & \text{if } TopDown_i, \\
Tree_{i,s,d}, NV_{i,c}, MF_i, MU_i) & \equiv \texttt{true} \\
\text{BOTTOM-UP}(G_{s,d}, F_{i,s}, & \text{if } TopDown_i \\
Tree_{i,s,d}, NV_{i,c}, NF_i) & \equiv \texttt{false}
\end{cases}
$$

[Choose direction and compute next tensors]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{10.2}$$

[Terminate when new frontier is empty]

Cascade 5. Hybrid DO-BFS EDGE Cascade

A macro in EDGE [3] is a named, parameterized Einsum template. Invoking a macro expands it into one or more Einsums via lexical substitution of its tensor and rank-variable expression parameters.

▹ **MACRO**

TOP-DOWN
$(TG_{s,d}, TF_{i,s}, TTree_{i,s,d}, TNV_{i,c}, TMF_i, TMU_i)$

$$\mapsto TTree_{i+1,s,d}, TF_{i+1,d}, TTopDown_{i+1}$$

$$Nei_{i,d} = TG_{s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee \text{OR}(\cup) \tag{10.3}$$

[Gather neighbors of nodes in frontier]

$$NeiNV_{i,d} = Nei_{i,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{10.4}$$

[Filter neighbors that are not yet visited]

$$CandTree_{i,s,d} = (TG_{s,d} \cdot TF_{i,s})_{i,s,d} \cdot NeiNV_{i,d} \tag{10.5}$$

$$:: \bigwedge \text{AND}(\cap) \ \bigwedge \text{AND}(\cap)$$

[Record candidate parent-child edges]

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{10.6}$$

[Select exactly one parent per child]

$$TTree_{i+1,s,d} = TTree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{10.7}$$

[Update BFS tree]

$$TF_{i+1,d} = NeiNV_{i,d} \tag{10.8}$$

[Set next frontier]

$$TMF_i = TG_{s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{10.9}$$

[m_f: sum of degrees of vertices in the frontier]

$$TMU_i = TG_{s,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{10.10}$$

[Calculate sum of degrees of Not Visited vertices]

$$TB_i = TMF_i \cdot (TMU_i/\alpha) :: \bigwedge > (\cap) \tag{10.11}$$

[Condition to switch from Top-Down to Bottom-Up]

$$TTopDown_{i+1} = \neg TB_i \tag{10.12}$$

[Update traversal mode for the next iteration]

▹ **MACRO**

BOTTOM-UP$(TG_{s,d}, TF_{i,s}, TTree_{i,s,d}, TNV_{i,c}, TNF_i)$

$$\mapsto TTree_{i+1,s,d}, TF_{i+1,d}, TTopDown_{i+1}$$

$$CandParents_{i,s,d} = TG_{s,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{10.13}$$

[Find candidate parents of Not Visited nodes]

$$InF_{i,s,d} = CandParents_{i,s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \tag{10.14}$$

[Check whether any parent is in frontier (InF)]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{10.15}$$

[Select exactly one parent per child]

$$TTree_{i+1,s,d} = Temp_{i,s,d} \cdot TTree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{10.16}$$

[Update BFS tree]

$$TF_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{10.17}$$

[Set next frontier]

$$TNF_i = TF_{i,s} :: \bigvee +(\cup) \tag{10.18}$$

[Count the number of vertices in the frontier]

$$BT_i = TNF_i \cdot (|V|/\beta) :: \bigwedge < (\cap) \tag{10.19}$$

[Condition to switch from Bottom-Up to Top-Down]

$$TTopDown_{i+1} = BT_i \tag{10.20}$$

[Update traversal mode for the next iteration]

Cascade 6. Hybrid EDGE Macro Cascade

11 EDGE Expression Walk-Through (Hybrid)

The hybrid Direction-Optimizing BFS (Beamer et al. [1]) maintains an explicit traversal-mode tensor, $TopDown_i$, that records whether iteration i executes the Top-Down or Bottom-Up macro. The switching conditions are computed from frontier work estimates and are used inside the selected macro to determine the traversal mode for the next iteration.

Step 1: Identify Not Visited vertices (NV). As in both traversal strategies, we first determine which vertices do not yet have parents recorded in the BFS tree.

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{11.1}$$

Step 2: Record the current traversal mode (TopDown). The tensor $TopDown_i$ stores the direction used at iteration i. If $TopDown_i$ = `true`, the algorithm executes the Top-Down macro. If $TopDown_i$ = `false`, it executes the Bottom-Up macro. Initially, the traversal begins in Top-Down mode:

$$TopDown_0 = \texttt{true} \tag{11.2}$$

Step 3: Select the macro for this iteration. The traversal-mode tensor determines which macro is executed at iteration i. The selected macro computes three outputs simultaneously: the updated BFS tree, the next frontier, and the traversal mode for the next iteration.

$$\begin{matrix} Tree_{i+1,s,d}, \\ F_{i+1,d}, \\ TopDown_{i+1} \end{matrix} = \begin{cases} \text{TOP-DOWN}(G_{s,d}, F_{i,s}, & \text{if } TopDown_i, \\ \quad Tree_{i,s,d}, NV_{i,c}, MF_i, MU_i) & \equiv \texttt{true} \\ \text{BOTTOM-UP}(G_{s,d}, F_{i,s}, & \text{if } TopDown_i. \\ \quad Tree_{i,s,d}, NV_{i,c}, NF_i) & \equiv \texttt{false} \end{cases} \tag{11.3}$$

Step 4: Top-Down macro. When $TopDown_i$ is true, the algorithm executes the Top-Down macro. Inside the macro template, the formal input and output tensors are written with a 'T' prefix to distinguish them from the actual tensors at the call site. The first part of this macro is unchanged from the Top-Down formulation: gather neighbors, filter them by the Not Visited mask, record candidate parent-child edges, select one parent, and update the tree and frontier.

$$Nei_{i,d} = TG_{s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \bigvee \text{OR}(\cup) \tag{11.4}$$

$$NeiNV_{i,d} = Nei_{i,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{11.5}$$

$$\begin{aligned} CandTree_{i,s,d} = (TG_{s,d} \cdot TF_{i,s})_{i,s,d} \cdot NeiNV_{i,d} \\ :: \bigwedge \text{AND}(\cap) \bigwedge \text{AND}(\cap) \end{aligned} \tag{11.6}$$

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{11.7}$$

$$TTree_{i+1,s,d} = TTree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{11.8}$$

$$TF_{i+1,d} = NeiNV_{i,d} \tag{11.9}$$

Step 5: Compute the Top-Down switching condition (TB). At the end of the Top-Down macro, we estimate whether the next iteration should remain Top-Down or switch to Bottom-Up. This requires two work estimates:

$$TMF_i = TG_{s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \bigvee +(\cup) \tag{11.10}$$

$$TMU_i = TG_{s,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \bigvee +(\cup) \tag{11.11}$$

$$TB_i = TMF_i \cdot (TMU_i/\alpha) :: \bigwedge > (\cap) \tag{11.12}$$

If TB_i is true, the Top-Down macro sets the next traversal mode to Bottom-Up. Otherwise, it keeps the next iteration in Top-Down mode:

$$TTopDown_{i+1} = \neg TB_i \tag{11.13}$$

Step 6: Bottom-Up macro. When $TopDown_i$ is false, the algorithm executes the Bottom-Up macro. As before, it first finds candidate parents for Not Visited vertices, checks whether any such parent is in the frontier, selects one parent, updates the tree, and reduces to form the next frontier.

$$CandParents_{i,s,d} = TG_{s,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \tag{11.14}$$

$$InF_{i,s,d} = CandParents_{i,s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \tag{11.15}$$

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{11.16}$$

$$TTree_{i+1,s,d} = Temp_{i,s,d} \cdot TTree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{11.17}$$

$$TF_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{11.18}$$

Step 7: Compute the Bottom-Up switching condition (BT). At the end of the Bottom-Up macro, the algorithm checks whether the frontier has become small enough to switch back to Top-Down. This uses the frontier-size estimate:

$$TNF_i = TF_{i,s} :: \bigvee +(\cup) \tag{11.19}$$

$$BT_i = TNF_i \cdot (|V|/\beta) :: \bigwedge < (\cap) \tag{11.20}$$

If BT_i is true, the Bottom-Up macro sets the next traversal mode to Top-Down. Otherwise, it keeps the next iteration in Bottom-Up mode:

$$TTopDown_{i+1} = BT_i \tag{11.21}$$

Step 8: Termination. As in both individual strategies, the hybrid algorithm terminates when the newly computed frontier is empty.

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{11.22}$$

Switching truth table. The traversal mode for iteration $i+1$ depends both on the current mode and on which switching condition is computed inside the active macro. Table 1 summarizes the behavior.

$TopDown_i$	Cond.	Val.	$TopDown_{i+1}$	Meaning
`true`	TB_i	`false`	`true`	stay Top-Down
`true`	TB_i	`true`	`false`	switch to Bottom-Up
`false`	BT_i	`false`	`false`	stay Bottom-Up
`false`	BT_i	`true`	`true`	switch to Top-Down

Table 1. Traversal-mode update in the Hybrid DO-BFS cascade.

12 Populate with the Coordinate Operator `pick-parent`

12.1 Coordinate Operator: `pick-parent`

In several BFS formulations, a child vertex may be discovered by multiple candidate parents in the same iteration. However, the BFS tree must store exactly one parent for each child. In EDGE, we enforce this constraint using the *populate* action together with a coordinate operator called `pick-parent`.

Populate processes one child coordinate at a time. For a fixed child d, it may receive multiple candidate parents s (coming from different frontier vertices). The role of the coordinate operator `pick-parent` is to choose exactly one of those candidates and discard the others. In this tutorial, we enforce early exit in the coordinate operator, by selecting the first candidate parent encountered for each child vertex.

Algorithm 1 The `pick-parent` coordinate operator

```
for each child vertex d do
    for each candidate parent s do
        if no parent is currently recorded for d then
            record s as the parent of d
            break
        end if
    end for
end for
```

After populate completes, each vertex has at most one recorded parent, ensuring that the *Tree* tensor represents a valid BFS tree rather than a general directed acyclic graph in which a vertex could have several parents.

Example: The `pick-parent` Coordinate Operator

The `pick-parent` function is used by the populate action to enforce that each child vertex selects exactly one parent during a given iteration.

In the EDGE expression:

$$Temp_{i,s^*,d} = CandTree_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}), \tag{12.1}$$

the coordinate operator `pick-parent` is applied along the parent dimension s for a fixed child vertex d.

Inputs to `pick-parent`:

- `z_fiber`: the set of currently selected parent coordinates for child d (i.e., existing true entries in $Temp_{i,s,d}$). This is an S-fiber of *Temp*.
- `rhs_coord`: the candidate parent coordinate s.
- `rhs_val`: the value of candidate parent s coming from $CandTree_{i,s,d}$.

Outputs:

- `valid_coords`: the parent coordinate that should remain true.
- `dead_coords`: parent coordinates that must be removed.

The rule implemented here selects the first-seen parent. Other tie-breaking rules may select the parent with the smallest vertex id, but doing so prevents early exit.

```
def pick_parent(z_fiber, rhs_coord, rhs_val):
    # Candidate parent coming from CandTree_{i,s,d}
    cand_s = rhs_coord

    # Check if any parent is already recorded for this child d
    has_parent = len(z_fiber) > 0

    if has_parent:
        # Already have a parent: reject the candidate
        valid_coords = []
        dead_coords = []
    else:
        # No parent yet: accept the first candidate seen
        valid_coords = [cand_s]
        dead_coords = []

    return valid_coords, dead_coords
```

Listing 1. Reference implementation of the pick-parent coordinate operator.

13 Other Notes

- **BFS tree and path reconstruction.** The tensor $Tree_{i,s,d}$ records parent-child relationships discovered during traversal. For each vertex d, exactly one parent s is selected (except for the source, which is its own parent). The collection of these parent assignments forms a BFS tree rooted at the source vertex.
  Maintaining the BFS tree makes it straightforward to reconstruct paths. To generate the path from the source s to any reachable vertex v, one can repeatedly follow the recorded parent pointers: start at v, look up its parent in the Tree tensor, then continue recursively until the source is reached. Reversing this sequence yields the shortest path from s to v in an unweighted graph.
  Because BFS explores vertices level by level, the depth of a vertex in the tree corresponds to its shortest-path distance (in number of edges) from the source.
  In many traditional implementations, instead of storing a full parent-child tensor, a simpler *predecessor array* (or parent array) is maintained. Such an array stores, for each vertex v, a single parent pointer $\text{pred}[v]$. The EDGE Tree tensor generalizes this idea by representing parent relationships in tensor form, while still allowing efficient extraction of a predecessor array if needed.
- **Populate and parent choice.** The populate step enforces the BFS-tree constraint "one parent per vertex." The coordinate operator `pick-parent` can implement different deterministic tie-breaking rules (e.g., minimum parent id, first-seen in a traversal order, or any random selection). Different rules may produce different valid BFS trees but the reachable set is unchanged.
- **Switch thresholds (α and β).** The hybrid algorithm uses two thresholds: α to decide when to switch from Top-Down to Bottom-Up, and β to decide when to switch back. These parameters are typically chosen empirically and may depend on graph structure (diameter, degree distribution) and hardware (memory bandwidth, cache behavior).
- **Work estimates (MF, MU).** MF_i approximates work for Top-Down by measuring total frontier degree. MU_i approximates work for Bottom-Up by measuring total degree of Not Visited vertices. These are proxy cost models; exact runtime depends on memory access patterns and representation.
- **Termination check.** The condition $\|F_{i+1}\| \equiv 0$ is written abstractly. Implementations typically check whether the next frontier has any elements (or whether a frontier count is zero).

14 Variants of Switching Heuristics

While the direction-optimizing BFS described by Beamer et al. [1] provides a widely used approach for switching between top-down (push) and bottom-up (pull) traversals, it is not the only strategy. A number of graph processing frameworks propose alternative switching heuristics that differ in how they estimate the work of each traversal direction and when they decide to switch.

At a high level, all of these methods aim to answer the same question: given the current frontier, which traversal direction will be more efficient in the next iteration?

While these heuristics differ in formulation, they are all based on a common trade-off. Top-down traversal is efficient when the frontier is small, since only a small number of outgoing edges are explored. In contrast, bottom-up traversal becomes advantageous when the frontier is large and the number of visited vertices is high, as it avoids repeatedly visiting many neighbors and instead checks incoming edges from a smaller set of unvisited vertices.

Each framework therefore designs its heuristic to approximate the relative cost of these two strategies while minimizing the overhead of computing the decision itself. The choice of heuristic reflects a balance between the accuracy of the cost estimate and the computational cost of calculating it.

However, these methods differ in the values they compute, the level of approximation they use, and how the decision is structured (e.g., single threshold, multiple thresholds, or multi-stage decisions).

In this section, we explore several representative switching heuristics from prior work and express each of them using EDGE notation. The goal is not to exhaustively cover all

possible strategies, but rather to demonstrate that EDGE can naturally capture a variety of switching formulations, from simple threshold-based rules to more structured, multi-stage decisions.

14.1 Gunrock: Approximate Switching Heuristic

Gunrock [6] adopts a direction-optimizing BFS strategy similar to that of Beamer et al. [1], but modifies the switching heuristic to better suit parallel execution on GPUs. Instead of computing exact edge-based quantities (such as the number of outgoing edges of the frontier or unvisited vertices), Gunrock uses simple approximations based on global graph statistics.

In particular, Gunrock estimates the work of the push and pull phases using the following quantities:

$$m_f = \frac{n_f \cdot m}{n}, \qquad m_u = \frac{n_u \cdot n}{n - n_u}, \tag{14.1}$$

where n_f is the number of vertices in the current frontier, n_u is the number of unvisited vertices, $m = |E|$ is the total number of edges, and $n = |V|$ is the number of vertices in the graph.

The switching conditions are then defined as:

$$m_f > m_u \cdot do_a \quad \text{(switch to bottom-up)}, \tag{14.2}$$

$$m_f < m_u \cdot do_b \quad \text{(switch back to top-down)}, \tag{14.3}$$

We now express the same heuristic using EDGE notation.

▹ **Initializations**

$$N = |V|$$
$$M = |E|$$
$$do_a \longrightarrow \langle \text{user-specified threshold} \rangle$$
$$do_b \longrightarrow \langle \text{user-specified threshold} \rangle$$
$$TopDown_0 = \texttt{true}$$

▹ **Extended Einsums**

$$NF_i = F_{i,s} :: \bigvee + (\cup) \tag{14.4}$$
[Count frontier vertices]

$$NU_i = NV_{i,s} :: \bigvee + (\cup) \tag{14.5}$$
[Count Not Visited vertices]

$$MF_i = NF_i \cdot (M/N) :: \bigwedge \times (\cap) \tag{14.6}$$
[Approximate push work]

$$MU_i = NU_i \cdot \left(\frac{N}{N - NU_i}\right) :: \bigwedge \times (\cap) \tag{14.7}$$
[Approximate pull work]

$$TB_i = MF_i \cdot (MU_i \cdot do_a) :: \bigwedge > (\cap) \tag{14.8}$$
[Switch to bottom-up when push exceeds pull]

$$BT_i = MF_i \cdot (MU_i \cdot do_b) :: \bigwedge < (\cap) \tag{14.9}$$
[Switch back when push work drops below pull]

$$Tree_{i+1,s,d},\ F_{i+1,d},\ TopDown_{i+1} = \begin{cases} \text{TOP-DOWN}(\cdots) & \text{if } TopDown_i, \\ & \equiv \texttt{true} \\ \text{BOTTOM-UP}(\cdots) & \text{if } TopDown_i \\ & \equiv \texttt{false} \end{cases}$$
[Choose traversal using Gunrock]

$$TopDown_{i+1} = \begin{cases} \neg TB_i & \text{if } TopDown_i, \\ & \equiv \texttt{true} \\ BT_i & \text{if } TopDown_i \\ & \equiv \texttt{false} \end{cases}$$
[Update traversal mode for the next iteration]

Cascade 7. Gunrock EDGE Cascade

Compared to Beamer's formulation, Gunrock replaces exact edge-based counts with simple approximations, avoiding additional graph traversals. This makes the heuristic more suitable for GPUs, where computing precise quantities can be more expensive. By using global statistics such as $|V|$ and $|E|$, Gunrock trades accuracy for efficiency while still capturing the relative cost of push and pull phases.

14.2 Ligra: Threshold-Based Switching Heuristic

Ligra [5] adopts a direction-optimizing traversal strategy similar to Beamer et al. [1], switching between top-down (sparse) and bottom-up (dense) traversal depending on the amount of work required in each step. The goal is to choose the traversal direction that is more efficient for the current frontier.

Unlike Beamer's formulation, which is defined specifically for BFS, Ligra provides a generalized abstraction for switching through its `EDGEMAP` procedure. This abstraction is used across multiple graph algorithms and is not limited to BFS.

In the `EDGEMAP` procedure, the decision between sparse (top-down) and dense (bottom-up) traversal is made using the following condition:

$$|U| + \sum_{u \in U} \deg(u) > \text{threshold}, \tag{14.10}$$

where U denotes the set of active vertices (e.g., the frontier), and $\deg(u)$ is the out-degree of vertex u. If this condition holds, Ligra uses the dense (bottom-up) traversal; otherwise, it uses the sparse (top-down) traversal.

We now express the same heuristic using EDGE notation.

▹ **Initializations**

$$\text{threshold} \longrightarrow \langle \text{user-specified threshold} \rangle$$

▹ **Extended Einsums**

$$NF_i = F_{i,s} :: \bigvee +(\cup) \tag{14.11}$$

[Count active frontier vertices]

$$MF_i = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{14.12}$$

[Count outgoing edges of the frontier]

$$TB_i = (NF_i + MF_i) \cdot \text{threshold} :: \bigwedge > (\cap) \tag{14.13}$$

[Switch to bottom-up when active work exceeds threshold]

$$Tree_{\substack{i+1,s,d,\\ F_{i+1,d}}} = \begin{cases} \text{BOTTOM-UP}(\cdots) & \text{if } TB_i, \\ & \equiv \texttt{true} \\ \text{TOP-DOWN}(\cdots) & \text{otherwise.} \end{cases}$$

[Choose traversal using Ligra heuristic]

Cascade 8. Ligra EDGE Cascade

This formulation shows that Ligra generalizes the switching decision beyond BFS by combining both the size of the active set and the number of outgoing edges from it. Both quantities can be naturally expressed using EDGE, demonstrating that the same tensor primitives can capture more general switching abstractions.

This heuristic is motivated by the observation that the cost of a top-down step depends not only on the number of active vertices but also on the number of edges they induce. By combining both quantities, Ligra captures the total amount of work required for a sparse (top-down) traversal. When this work becomes large, it is more efficient to switch to a dense (bottom-up) traversal, which avoids repeatedly exploring many outgoing edges.

Additionally, Ligra uses a single threshold to keep the decision simple and general across different graph algorithms, rather than tailoring the heuristic specifically to BFS.

14.3 XBFS: Multi-Strategy Switching Heuristic

XBFS extends the direction-optimizing BFS approach by introducing multiple traversal strategies instead of a single top-down and bottom-up switch [7]. In addition to bottom-up traversal, XBFS distinguishes between different top-down execution modes and dynamically selects among them based on the characteristics of the current frontier.

Similar to Beamer's approach, XBFS uses the amount of work induced by the frontier to determine when to switch to bottom-up traversal. In particular, it evaluates the total number of neighbors of the frontier relative to the total number of edges in the graph. The bottom-up traversal is selected when:

$$\frac{|E_f|}{|E|} > \alpha, \tag{14.14}$$

where $|E_f|$ denotes the number of outgoing edges from the frontier, $|E|$ is the total number of edges in the graph, and α is a tunable parameter [7].

If this condition is not satisfied, XBFS remains in top-down mode and selects among different push-based execution strategies based on additional heuristics related to the frontier.

We now express the bottom-up switching condition in EDGE notation.

▹ **Initializations**

$$M = |E|$$

$$\alpha \longrightarrow \langle \text{user-specified threshold} \rangle$$

▹ **Extended Einsums**

$$MF_i = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{14.15}$$

[Count edges incident to frontier]

$$BU_i = MF_i \cdot (M \cdot \alpha) :: \bigwedge > (\cap) \tag{14.16}$$

[Switch to bottom-up when frontier work exceeds threshold]

$$Tree_{\substack{i+1,s,d,\\ F_{i+1,d}}} = \begin{cases} \text{BOTTOM-UP}(\cdots) & \text{if } BU_i, \\ & \equiv \texttt{true} \\ \text{TOP-DOWN-VARIANT}(\cdots) & \text{otherwise.} \end{cases}$$

[Choose bottom-up or keep top-down]

Cascade 9. XBFS EDGE Cascade

The motivation behind this heuristic is to measure how large the current frontier is relative to the graph. Instead of using absolute counts, XBFS compares the number of edges

coming from the frontier to the total number of edges in the graph. This makes the decision more robust across graphs of different sizes.

If a large fraction of the graph is being explored through the frontier, continuing with top-down traversal would result in repeatedly visiting many edges. In this case, switching to bottom-up is more efficient, since it avoids exploring all outgoing edges and instead checks a smaller set of unvisited vertices.

XBFS also observes that top-down traversal itself can be implemented in different ways. Rather than treating it as a single strategy, it selects among multiple top-down variants when not switching to bottom-up, allowing it to better adapt to the structure of the frontier. This shows that EDGE can express not only different switching conditions, but also more structured decision processes built on the same underlying tensor computations.

14.4 Discussion

The switching heuristics presented in this section share a common goal: to select the traversal direction that minimizes the work performed in each iteration. While different frameworks propose different formulations, they are all based on estimating the amount of work induced by the current frontier.

Across Ligra, Gunrock, and XBFS, a common quantity that appears is the total number of outgoing edges from the frontier vertices. In EDGE notation, this is captured by the expression:

$$MF_i = G_{s,d} \cdot F_{i,s} :: \bigwedge \text{AND}(\cap) \bigvee +(\cup). \tag{14.17}$$

Each framework uses this quantity differently. Ligra compares it against a fixed threshold, Gunrock uses approximations based on global graph statistics, and XBFS incorporates it into a multi-stage decision process. Despite these differences, the underlying tensor computation remains the same.

This observation highlights an important aspect of the EDGE representation: a single tensor expression can support multiple switching heuristics without modification. The differences between these methods are not from new computations, but from how the computed quantities are used.

The key difference between these heuristics lies in how they balance the trade-off between accuracy and overhead. More precise estimates (such as exact edge counts) can lead to better switching decisions, but may introduce additional computational cost. Simpler heuristics, while less accurate, are often preferred in practice when they can be computed with minimal overhead.

This highlights that the effectiveness of a switching heuristic is not only determined by how well it models the underlying work, but also by how efficiently that model can be evaluated within each iteration.

Overall, these examples demonstrate that EDGE can express a range of switching strategies, from simple threshold-based rules to more structured decision processes, while reusing the same core tensor operations.

15 Simplified and Compact EDGE Formulation

The EDGE expressions presented so far closely follow the step-by-step construction of BFS and introduce intermediate tensors to make each stage of the computation explicit. While this improves intuition, it also results in redundant computations and additional tensors that are not strictly necessary.

We can simplify the EDGE formulation for BFS by combining these intermediate steps into more direct expressions. In particular, we avoid repeatedly computing neighbor sets and instead directly express the conditions required for discovering new vertices and assigning parents.

In this simplified formulation, the structure of the computation becomes clearer, and the difference between top-down and bottom-up traversals is more explicit. Both can be viewed as computing the same set of candidate parent-child edges, but with different evaluation orders and masking strategies.

16 Compact EDGE Expression (Top-Down)

The full EDGE expression for top-down Direction-Optimizing BFS is as follows:

▹ **Tensors**

$$G^{S\equiv|V|,\, D\equiv|V|} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,\, S\equiv|V|} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$Tree^{I,\, S\equiv|V|,\, D\equiv|V|} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$InF^{I,\, S,\, D} \longrightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$Tree_{i,s,d} = \texttt{false}$$
$$G_{s,d} \longrightarrow \langle\text{user-specified}\rangle$$
$$InF_{i,s,d} = \texttt{false}$$
$$F_{0,n} = \texttt{true}$$
$$Tree_{0,n,n} = \texttt{true}$$

▹ **Extended Einsums**

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{16.1}$$

[Find Not Visited vertices]

$$InF_{i,s,d} = (G_{s,d} \cdot^1 F_{i,s})_{i,s,d} \cdot^2 NV_{i,d} \tag{16.2}$$

$$:: \overset{1}{\bigwedge} \text{AND}(\cap) \ \overset{2}{\bigwedge} \text{AND}(\cap)$$

[Find neighbors of frontier vertices and check if unvisited]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{16.3}$$

[Select exactly one parent per child]

$$Tree_{i+1,s,d} = Tree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{16.4}$$

[Update BFS tree]

$$F_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{16.5}$$

[Set next frontier from discovered vertices]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{16.6}$$

[Stop when the next frontier is empty]

Cascade 10. Compact Top-Down DO-BFS EDGE Cascade

$$:: \overset{1}{\bigwedge} \text{AND}(\cap) \ \overset{2}{\bigwedge} \text{AND}(\cap)$$

[Check whether a parent exists in frontier]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{17.3}$$

[Select exactly one parent per child]

$$Tree_{i+1,s,d} = Temp_{i,s,d} \cdot Tree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{17.4}$$

[Update BFS tree]

$$F_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{17.5}$$

[Set next frontier]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{17.6}$$

[Stop when the next frontier is empty]

Cascade 11. Compact Bottom-Up DO-BFS EDGE Cascade

17 Compact EDGE Expression (Bottom-Up)

The full EDGE expression for bottom-up Direction-Optimizing BFS is as follows:

▹ **Tensors**

$$G^{S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,\ S\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$Tree^{I,\ S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$InF^{I,\ S,\ D} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$Tree_{i,s,d} = \texttt{false}$$
$$G_{s,d} \rightarrow \langle\text{user-specified}\rangle$$
$$InF_{i,s,d} = \texttt{false}$$
$$F_{0,n} = \texttt{true}$$
$$Tree_{0,n,n} = \texttt{true}$$

▹ **Extended Einsums**

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{17.1}$$

[Find Not Visited vertices]

$$InF_{i,s,d} = (G_{s,d} \cdot^1 NV_{i,d})_{i,s,d} \cdot^2 F_{i,s} \tag{17.2}$$

18 Compact EDGE Expression (Hybrid)

The full EDGE expression for the hybrid Direction-Optimizing BFS is as follows.

▹ **Tensors**

$$G^{S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$F^{I,\ S\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$Tree^{I,\ S\equiv|V|,\ D\equiv|V|} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$InF^{I,\ S,\ D} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$
$$TopDown^{I} \rightarrow \text{Boolean},\ \text{empty} = \texttt{false}$$

▹ **Initializations**

$$F_{i,s} = \texttt{false}$$
$$Tree_{i,s,d} = \texttt{false}$$
$$G_{s,d} \rightarrow \langle\text{user-specified}\rangle$$
$$InF_{i,s,d} = \texttt{false}$$
$$F_{0,n} = \texttt{true}$$
$$Tree_{0,n,n} = \texttt{true}$$
$$TopDown_0 = \texttt{true}$$

▹ **Extended Einsums**

$$NV_{i,c} = \neg Tree_{i,p,c} :: \bigvee \text{OR}(\cup) \tag{18.1}$$

[Find Not Visited vertices from the tree]

$$Tree_{i+1,s,d},\ F_{i+1,d},\ TopDown_{i+1} = \begin{cases} \text{TOP-DOWN}(G_{s,d}, F_{i,s}, Tree_{i,s,d}, NV_{i,c}, MF_i, MU_i) & \text{if } TopDown_i \equiv \texttt{true} \\ \text{BOTTOM-UP}(G_{s,d}, F_{i,s}, Tree_{i,s,d}, NV_{i,c}, NF_i) & \text{if } TopDown_i \equiv \texttt{false} \end{cases}$$

[Choose direction and compute next tensors]

$$\diamond : \ \|F_{i+1}\| \equiv 0 \tag{18.2}$$

[Stop when the next frontier is empty]

Cascade 12. Compact Hybrid DO-BFS EDGE Cascade

▹ **MACRO**

TOP-DOWN

$$(TG_{s,d}, TF_{i,s}, TTree_{i,s,d}, TNV_{i,c}, TMF_i, TMU_i) \mapsto TTree_{i+1,s,d}, TF_{i+1,d}, TTopDown_{i+1}$$

$$InF_{i,s,d} = (TG_{s,d} \cdot^1 TF_{i,s})_{i,s,d} \cdot^2 TNV_{i,d} \tag{18.3}$$
$$:: \overset{1}{\bigwedge} \text{AND}(\cap) \ \overset{2}{\bigwedge} \text{AND}(\cap)$$

[Find frontier neighbors that are unvisited]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{18.4}$$

[Select exactly one parent per child]

$$TTree_{i+1,s,d} = TTree_{i,s,d} \cdot Temp_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{18.5}$$

[Update BFS tree]

$$TF_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{18.6}$$

[Set next frontier from discovered vertices]

$$TMF_i = TG_{s,d} \cdot TF_{i,s} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{18.7}$$

[m_f: total degree of frontier vertices]

$$TMU_i = TG_{s,d} \cdot TNV_{i,d} :: \bigwedge \text{AND}(\cap) \ \bigvee +(\cup) \tag{18.8}$$

[Sum degrees of Not Visited vertices]

$$TB_i = TMF_i \cdot (TMU_i/\alpha) :: \bigwedge > (\cap) \tag{18.9}$$

[Switch from Top-Down to Bottom-Up]

$$TTopDown_{i+1} = \neg TB_i \tag{18.10}$$

[Update traversal mode for the next iteration]

▹ **MACRO**

BOTTOM-UP$(TG_{s,d}, TF_{i,s}, TTree_{i,s,d}, TNV_{i,c}, TNF_i) \mapsto TTree_{i+1,s,d}, TF_{i+1,d}, TTopDown_{i+1}$

$$InF_{i,s,d} = (TG_{s,d} \cdot^1 TNV_{i,d})_{i,s,d} \cdot^2 TF_{i,s} \tag{18.11}$$
$$:: \overset{1}{\bigwedge} \text{AND}(\cap) \ \overset{2}{\bigwedge} \text{AND}(\cap)$$

[Check whether the parent exists in frontier]

$$Temp_{i,s^*,d} = InF_{i,s,d} \lll_{s^*} \mathbb{1}(\text{pick-parent}) \tag{18.12}$$

[Select exactly one parent per child]

$$TTree_{i+1,s,d} = Temp_{i,s,d} \cdot TTree_{i,s,d} :: \bigwedge \text{OR}(\cup) \tag{18.13}$$

[Update BFS tree]

$$TF_{i+1,d} = InF_{i,s,d} :: \bigvee \text{OR}(\cup) \tag{18.14}$$

[Set next frontier]

$$TNF_i = TF_{i,s} :: \bigvee +(\cup) \tag{18.15}$$

[Count vertices in the frontier]

$$BT_i = TNF_i \cdot (|V|/\beta) :: \bigwedge < (\cap) \tag{18.16}$$

[Switch from Bottom-Up to Top-Down]

$$TTopDown_{i+1} = BT_i \tag{18.17}$$

[Update traversal mode for the next iteration]

Cascade 13. Compact Hybrid EDGE Macro Cascade

Note on *InF*. Both macros compute the same tensor $InF_{i,s,d}$: the edges that can extend the BFS tree at iteration i, meaning the edge (s, d) exists, s is in the frontier, and d is not yet visited. The two differ only in where the parentheses fall — top-down applies $F_{i,s}$ first, bottom-up applies $NV_{i,d}$ first. This makes it clear that even when recording parents, the difference between top-down and bottom-up boils down to an algebraic manipulation when collecting neighbors, as shown for push/pull BFS in [4].

References

[1] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)* (Salt Lake City, Utah, USA). IEEE, 12:1–12:10. doi:10.1109/SC.2012.50

[2] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. 2023. TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)*. ACM, 1255–1270. doi:10.1145/3613424.3623791

[3] Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Joel S. Emer, and John D. Owens. 2026. The EDGE Language: Extended General Einsums for Graph Algorithms. arXiv:2404.11591v2 [cs.DS] https://arxiv.org/abs/2404.11591v2

[4] Toluwanimi O. Odemuyiwa, Serban D. Porumbescu, Muhammad Osama, Joel S. Emer, and John D. Owens. 2026. The Einsum-Enabled Design Space for Graph Algorithms: A BFS Case Study. arXiv:2607.17106 [cs.DS] https://arxiv.org/abs/2607.17106

[5] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 135–146. doi:10.1145/2442516.2442530

[6] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. doi:10.1145/3108140

[7] Hao Yang, Haicheng Lu, Nadeem S. Sattar, Hang Liu, and Feifei Wang. 2024. Establish the Basis for Breadth-First Search on Frontier System: XBFS on AMD GPUs. In *Proceedings of the SC24 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W)*. 650–658. doi:10.1109/SCW63240.2024.00090