Graph Traversal: DFS & BFS – Student Notes (with Code Appendix)

This document extends the previous notes with a detailed code appendix. The goal is to help you read, understand, and modify the Python simulation for DFS and BFS.

1. Short Recap

The program visualizes graph traversal using Depth-First Search (DFS) and Breadth-First Search (BFS). It shows the current node, the visited order, and the frontier data structure (stack for DFS, queue for BFS). You can step through the algorithm or run it automatically.

2. Full Code Listing

Below is the complete Python script used for the DFS/BFS simulation with Tkinter. You can copy this into a file named, for example, graph_simulation_stack_queue.py and run it with Python.

```
import tkinter as tk
from tkinter import ttk
from collections import deque

GRAPH = {
    "A": ["B", "C"],
    "B": ["A", "D", "E"],
    "C": ["A", "F"],
    "D": ["B"],
    "E": ["B", "F"],
    "F": ["C", "E"],
}

NODE_POSITIONS = {
    "A": (150, 80),
    "B": (80, 200),
    "C": (220, 200),
    "D": (40, 320),
    "E": (120, 320),
    "F": (260, 320),
}

NODE_RADIUS = 22

def build_dfs_states(graph, start):
    if start not in graph:
        return []
    stack = [start]
    visited = []
    visited_set = set()
    states = []
    while stack:
        u = stack.pop()
        if u in visited_set:
            states.append({
                "current": u,
                "frontier": list(stack),
                "visited": list(visited),
                "kind": "DFS",
            })
            continue
        visited_set.add(u)
        visited.append(u)
        for v in reversed(graph[u]):
            if v not in visited_set:
                stack.append(v)
        states.append({
            "current": u,
            "frontier": list(stack),
            "visited": list(visited),
            "kind": "DFS",
        })
    return states

def build_bfs_states(graph, start):
    if start not in graph:
        return []
    q = deque([start])
    visited_set = {start}
    visited = []
    states = []
    while q:
```

```
        u = q.popleft()
        visited.append(u)
        for v in graph[u]:
            if v not in visited_set:
                visited_set.add(v)
                q.append(v)
        states.append({
            "current": u,
            "frontier": list(q),
            "visited": list(visited),
            "kind": "BFS",
        })
    return states

class GraphCanvas:
    def __init__(self, root, graph, positions):
        self.graph = graph
        self.positions = positions
        self.node_items = {}
        self.canvas = tk.Canvas(root, width=500, height=400, bg="white")
        self.canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        self.draw_graph()

    def draw_graph(self):
        drawn_edges = set()
        for u, neighbors in self.graph.items():
            x1, y1 = self.positions[u]
            for v in neighbors:
                if (v, u) in drawn_edges:
                    continue
                x2, y2 = self.positions[v]
                self.canvas.create_line(x1, y1, x2, y2, width=2)
                drawn_edges.add((u, v))
        for node, (x, y) in self.positions.items():
            circle = self.canvas.create_oval(
                x - NODE_RADIUS,
                y - NODE_RADIUS,
                x + NODE_RADIUS,
                y + NODE_RADIUS,
                fill="#DDDDDD",
                outline="black",
                width=2,
            )
            text = self.canvas.create_text(x, y, text=node, font=("Arial", 14, "bold"))
            self.node_items[node] = (circle, text)

    def reset_colors(self):
        for node, (circle, _) in self.node_items.items():
            self.canvas.itemconfig(circle, fill="#DDDDDD")

    def highlight_node(self, node, color):
        if node in self.node_items:
            circle, _ = self.node_items[node]
            self.canvas.itemconfig(circle, fill=color)

    def emphasize_node(self, node):
        self.highlight_node(node, "#FFA500")

    def mark_visited(self, node):
        self.highlight_node(node, "#90EE90")

class AlgorithmAnimator:
    def __init__(self, root, graph_canvas, graph, controls_frame):
        self.root = root
        self.graph_canvas = graph_canvas
        self.graph = graph
        self.states = []
        self.index = 0
```

```
self.running = False
self.current_kind = None

tk.Label(controls_frame, text="Start node:", font=("Arial", 11, "bold")).pack(
    anchor="w", pady=(5, 0)
)
self.start_var = tk.StringVar(value="A")
start_nodes = sorted(self.graph.keys())
self.start_menu = ttk.Combobox(
    controls_frame,
    textvariable=self.start_var,
    values=start_nodes,
    state="readonly",
    width=5,
)
self.start_menu.pack(anchor="w", pady=(0, 5))

tk.Label(controls_frame, text="Algorithm:", font=("Arial", 11, "bold")).pack(
    anchor="w", pady=(5, 0)
)
self.algorithm_var = tk.StringVar(value="DFS")
self.algorithm_menu = ttk.Combobox(
    controls_frame,
    textvariable=self.algorithm_var,
    values=["DFS", "BFS"],
    state="readonly",
    width=7,
)
self.algorithm_menu.pack(anchor="w", pady=(0, 10))

btn_frame = tk.Frame(controls_frame)
btn_frame.pack(fill="x", pady=5)

self.btn_dfs = tk.Button(
    btn_frame, text="Run DFS", command=self.run_dfs, width=10
)
self.btn_dfs.grid(row=0, column=0, padx=2, pady=2)

self.btn_bfs = tk.Button(
    btn_frame, text="Run BFS", command=self.run_bfs, width=10
)
self.btn_bfs.grid(row=0, column=1, padx=2, pady=2)

self.btn_step = tk.Button(
    btn_frame, text="Step ■", command=self.step_once, width=10
)
self.btn_step.grid(row=1, column=0, padx=2, pady=2)

self.btn_reset = tk.Button(
    btn_frame, text="Reset", command=self.reset, width=10
)
self.btn_reset.grid(row=1, column=1, padx=2, pady=2)

tk.Label(
    controls_frame, text="Current algorithm:", font=("Arial", 11, "bold")
).pack(anchor="w", pady=(10, 0))
self.algorithm_label = tk.Label(controls_frame, text="-", font=("Arial", 11))
self.algorithm_label.pack(anchor="w")

tk.Label(
    controls_frame, text="Currently visiting:", font=("Arial", 11, "bold")
).pack(anchor="w", pady=(10, 0))
self.current_label = tk.Label(controls_frame, text="-", font=("Arial", 11))
self.current_label.pack(anchor="w")

tk.Label(
    controls_frame, text="Visited order:", font=("Arial", 11, "bold")
).pack(anchor="w", pady=(10, 0))
```

```
        self.visited_label = tk.Label(
            controls_frame, text="[]", font=("Consolas", 10),
            wraplength=230, justify="left"
        )
        self.visited_label.pack(anchor="w")

        tk.Label(
            controls_frame, text="Frontier (stack/queue):", font=("Arial", 11, "bold")
        ).pack(anchor="w", pady=(10, 0))
        self.frontier_label = tk.Label(
            controls_frame, text="[]", font=("Consolas", 10),
            wraplength=230, justify="left"
        )
        self.frontier_label.pack(anchor="w")

        tk.Label(
            controls_frame,
            text="DFS: stack (top → right)\nBFS: queue (front → left)",
            font=("Arial", 9),
            justify="left",
            fg="gray25",
        ).pack(anchor="w", pady=(5, 0))

        tk.Label(
            controls_frame, text="Animation delay (ms):", font=("Arial", 11, "bold")
        ).pack(anchor="w", pady=(10, 0))
        self.speed_var = tk.IntVar(value=800)
        self.speed_scale = tk.Scale(
            controls_frame,
            from_=100,
            to=2000,
            orient=tk.HORIZONTAL,
            variable=self.speed_var,
        )
        self.speed_scale.pack(fill="x")

    def prepare_algorithm(self, kind):
        start = self.start_var.get()
        if start not in self.graph:
            return
        self.reset_visual_only()
        if kind == "DFS":
            self.states = build_dfs_states(self.graph, start)
        else:
            self.states = build_bfs_states(self.graph, start)
        self.current_kind = kind
        self.algorithm_label.config(text=kind)
        self.index = 0
        self.running = False
        self.update_ui_for_state(None)

    def run_dfs(self):
        self.algorithm_var.set("DFS")
        self.prepare_algorithm("DFS")
        self.running = True
        self._auto_step()

    def run_bfs(self):
        self.algorithm_var.set("BFS")
        self.prepare_algorithm("BFS")
        self.running = True
        self._auto_step()

    def step_once(self):
        selected_kind = self.algorithm_var.get()
        if not self.states or self.current_kind != selected_kind:
            self.prepare_algorithm(selected_kind)
            if not self.states:
```

```
                return
            self._do_step()

        def _do_step(self):
            if self.index >= len(self.states):
                self.running = False
                self.current_label.config(text="Finished")
                return
            state = self.states[self.index]
            current = state["current"]
            self.graph_canvas.emphasize_node(current)
            self.current_label.config(text=current)
            for node in state["visited"]:
                self.graph_canvas.mark_visited(node)
            self.update_ui_for_state(state)
            self.index += 1

        def _auto_step(self):
            if not self.running:
                return
            self._do_step()
            if self.index < len(self.states):
                delay = self.speed_var.get()
                self.root.after(delay, self._auto_step)

        def update_ui_for_state(self, state):
            if state is None:
                self.visited_label.config(text="[]")
                self.frontier_label.config(text="[]")
                return
            visited = state["visited"]
            frontier = state["frontier"]
            self.visited_label.config(text=str(visited))
            if state["kind"] == "DFS":
                self.frontier_label.config(
                    text=f"{frontier}   (stack, top → right)"
                )
            else:
                self.frontier_label.config(
                    text=f"{frontier}   (queue, front → left)"
                )

        def reset_visual_only(self):
            self.graph_canvas.reset_colors()
            self.current_label.config(text="-")
            self.visited_label.config(text="[]")
            self.frontier_label.config(text="[]")

        def reset(self):
            self.reset_visual_only()
            self.states = []
            self.index = 0
            self.running = False
            self.current_kind = None
            self.algorithm_label.config(text="-")

    def main():
        root = tk.Tk()
        root.title("Graph Traversal Simulator (DFS/BFS + Stack/Queue)")
        graph_canvas = GraphCanvas(root, GRAPH, NODE_POSITIONS)
        controls_frame = tk.Frame(root, padx=10, pady=10)
        controls_frame.pack(side=tk.RIGHT, fill=tk.Y)
        AlgorithmAnimator(root, graph_canvas, GRAPH, controls_frame)
        root.mainloop()

    if __name__ == "__main__":
        main()
```

3. Section-by-Section Explanation

3.1 Graph and Positions

The GRAPH dictionary is an adjacency list representation of the graph. Each key is a node label and the corresponding value is a list of neighboring nodes. NODE_POSITIONS defines where each node will be drawn on the canvas, using (x, y) coordinates.

3.2 DFS and BFS State Builders

Both build_dfs_states and build_bfs_states do not directly draw anything on the screen. Instead, they construct a list of 'states'. Each state stores the current node (current), the frontier (stack or queue), and the visited list at that moment. This design is very powerful for simulation, because you can run the algorithm step-by-step by iterating over the states.

3.3 GraphCanvas Class

GraphCanvas encapsulates all drawing logic using Tkinter's Canvas widget. The draw_graph method first draws all edges as lines, then all nodes as circles with labels. The node_items dictionary maps node labels to their corresponding canvas item IDs, so we can change their colors later. emphasize_node colors the current node orange, while mark_visited colors visited nodes light green.

3.4 AlgorithmAnimator Class

AlgorithmAnimator connects the user interface with the algorithm logic. It creates dropdowns for the start node and algorithm type, buttons for running DFS/BFS, a step button, a reset button, and labels to show the current node, visited order, and frontier contents. The prepare_algorithm method builds the list of states depending on whether DFS or BFS is selected.

The step_once method either prepares the states (if needed) or advances by one step, calling _do_step. The _auto_step method repeatedly calls _do_step with a delay, controlled by the animation speed slider. update_ui_for_state updates the labels so that students can see how visited and frontier change over time.

3.5 Main Function

The main function builds the root Tkinter window, initializes the GraphCanvas and AlgorithmAnimator, and starts the Tkinter event loop (root.mainloop()). This is the standard structure for a Tkinter GUI program.

4. Suggested Student Activities with the Code

1. Change the graph structure (add nodes, remove edges) and observe how DFS and BFS orders change. 2. Modify colors used for current and visited nodes. 3. Add a label showing the size of the frontier at each step. 4. Extend the program to highlight the edge being traversed in each step. 5. Implement a 'Reset to Original Graph' button that restores the default graph definition.

End of Code Appendix