Depth-first search, Recursion

6.100 LECTURE 13

SPRING 2026

Announcements

- Pset 3 checkoffs due tonight
- Pset 4 due Friday 3/20
- Pset 4 checkoffs throughout the week after break 3/30–4/3

Iteration vs Recursion

- **Recursion** simulates iteration by rolling out state in the **stack of function calls**
  - **base case** is essential for stopping!
  - to reach base case, **each recursive call should be performing a "smaller" task**
- **Any iterative program can be written recursively, and vice versa**
  - do not gain nor lose any computability power with either approach
- In practice, **recursion tends to have higher overhead** due to stack frames
  - Python's default limits the stack to 1000 frames, `RecursionError` if exceed
    - `import sys`
    - `sys.getrecursionlimit()`
- But thinking in terms of recursion can be very useful!

Depth-first search

DFS strategy

- BFS's strategy was to **build each successive frontier**
  - resulted in shortest paths
  - but always have to keep an entire frontier in memory
- DFS **prioritizes exploring away** from the start
  - rather than fill in an entire frontier at a time, just **keep following neighbors**
  - when you reach a dead end (no outgoing edges to unseen nodes), **backtrack to previous node in path so far,** explore next neighbor
- "Following your nose" and backtracking is inherently recursive!

DFS on trees

- Like with BFS implementation, start with **trees**
  - subtrees do not overlap
  - hence there's a **unique path** from the root to any node
- **Trees are structurally recursive!**
  - each neighbor of a node is the root of its own subtree
- **Recursive case**
  - **run DFS on each neighbor** of the root
  - if any neighbor reaches the goal, then so can the root
- **Base case**
  - when the root is the goal, stop searching
  - causes frames on stack to **unravel back to global frame**

Getting paths out of DFS on trees

- **Insight:** when unraveling, collect the subtree roots as you go
- **Recursive case**
  - when a DFS call on a neighbor returns a valid path, append the current root on the beginning
- **Base case**
  - when reach the goal, **return a path** consisting of the goal

DFS on graphs

- Like in BFS, should avoid visiting previously seen nodes
  - **possibility 1:** edge to a node in a previously explored subtree
    - would waste time exploring that subtree again
  - **possibility 2:** edge to a node in the current path from start
    - would result in an infinite loop → Python RecursionError
- Whereas BFS was a single function call, with a **`visited`** set created during execution, DFS operates as many function calls
  - how to maintain a single **`visited`** set across all those calls?
- Common strategy
  - make the **recursive DFS function a "private" helper** that takes in a **`visited`** set parameter
  - mutate that set as needed
  - the **"public" DFS function** has no such extra parameter, because users don't need to know about it
  - instead, it makes a single top-level call to the recursive helper with an **empty set to initialize `visited`**

Recursive list structure

Lists as trees/graphs

- Remember that lists store only references to other objects
  - all the arrows we've been drawing in our environment diagrams can be interpreted as graph edges!
- **Every object in heap memory is a node**
  - a **list object** is a node with edges (references) to other nodes
  - **non-list objects** (str, int, float) are nodes with no neighbors
  - (assume no other container types for now, e.g., dict, tuple, set)
- If there is **no aliasing** among nested lists, then they form a **tree structure**

Operations on nested lists

- **Exercise 1:** count all the non-list objects contained in the structure
  - **recursive case:** list object, add up the counts within all its items
  - **base case:** non-list object, count is 1
- **Exercise 2:** print a list with indentation for each level of nesting
  - use a recursive helper to collect all lines to print, then join them with newlines
  - **recursive case:** list object, collect lines for how each of its items would print, then indent them and wrap with square brackets
  - **base case:** non-list object, just stringify the item on a single line, but need to represent the line as a single-element list

Queue-based search

Queues and Stacks

- A **queue** is a list where we are only allowed to append at the end and remove from the beginning
  - **first-in first-out (FIFO)** operation
- A **stack** is a list where we are only allowed to append and remove from the same end
  - **last-in first-out (LIFO)** operation
  - a stack is often called a LIFO queue
- Alternate implementations of BFS and DFS
  - **BFS frontiers** can be simulated with a single FIFO queue
  - **DFS recursion** can be simulated with a single LIFO stack

FIFO queue-based implementation of BFS

- Store current frontier on queue
- Remove node from beginning, add its neighbors to the end
- At any point, the queue stores:
  - the **latter part of the current frontier** that hasn't been expanded yet
  - followed by **nodes of the next frontier being expanded to**
- Check for **visited nodes** as needed (i.e., avoid adding them back on the queue)

LIFO stack-based implementation of DFS

- Treat each element on stack as a stand-in for a recursive **`dfs()`** call
- To simulate recursive calls to a node's neighbors:
  - **pop a node** from the stack
    - this is a "pre-execution" of the stack frame going away
  - put **new nodes** on the stack for **each of its neighbors**
    - put them in **reverse order,** so when they are removed, they match the order in which **`dfs()`** would have been called on them
    - reverse order is not strictly necessary, just to make the execution consistent with the recursive version
    - when a node and all of its neighbors have been popped off the stack, it's "subtree" has been fully explored
    - now the **node's subsequent siblings** are ready to be popped
- Check for **visited nodes** as needed

Next time

- **Shortest paths on weighted graphs**
- For pre-lecture prep, read and work on **Section 3 of Pset 4**
- In lecture, we will extend that idea into **Dijkstra's algorithm**