Hashing, Sets, Breadth-first search

6.100 LECTURE 12

SPRING 2026

Announcements

- Pset 3 checkoffs due next Monday 3/16
  - feedback and scores available on Scores page
- Pset 4 due next Friday 3/20
  - can make a lot of progress after today's lecture
- If you have concerns about your performance in the class, please see me before spring break
  - instructor office hours
  - by appointment

Hashing and Sets

Timing of `list` vs `dict` operations

- Python **`list`**s are contiguous blocks of references to other objects
  - makes looking up by index fast
  - but looking up by item value needs to scan list sequentially
- Preserving contiguity is important for fast index lookup
  - but **insertion** and **deletion** in middle of list requires shifting all subsequent cells
- When performing **`(key, value)`** insertions and deletions in a **`dict`**, runtime cost grows notably slower than for **`list`**s
  - what is the internal mechanism for Python **`dict`**s?

Designing for fast insert, lookup, delete

- **Property:** contiguity enables fast lookup by index
- **Issue:** contiguity make insertion and deletion of items expensive
- Idea 1: **treat data as an index**
  - any data can be interpreted as a number **x**
  - with a large enough contiguous block, just mark at that index **x** whether item is present
  - fast to insert, lookup, delete
- **Problem:** data "number" can get really large
  - wasted space, slow for iteration
  - not enough space

Designing for fast insert, lookup, delete

- **Property:** contiguity enables fast lookup by index
- **Issue:** contiguity make insertion and deletion of items expensive
- **Idea 2: map data down to a smaller range of indices**
  - hash function takes in data, outputs a valid index into a list of length $\boldsymbol{p}$, e.g.,

$$hash(\text{data}) = \left(\sum_{i}^{len(\text{data})} ord(\text{data}[i]) \times 2^{i}\right) \bmod p$$

  - list cell at index stores reference to actual data
  - preserves fast insert/check/delete
  - iteration is now fast, too
- **Problem:** different data may collide on same hash value
  - how to distinguish?

Designing for fast insert, lookup, delete

- **Property:** contiguity enables fast lookup by index
- **Issue:** contiguity make insertion and deletion of items expensive
- Idea 3: **chaining: list cells point to small collections of data**
  - use secondary lists for those collections
  - adds one more step for insert
  - requires == verification for lookup and delete
  - iteration requires scanning secondary lists
- **Note:** to maintain fast operations, second lists need to be small
  - hence periodically resize top-level hash list to be proportional in size to number of items
  - overall, only a couple extra steps per operation

Hash tables and sets

- **Hash tables** are very effective at implementing the mathematical notion of a **set**
  - an **unordered collection of items**
  - **no duplicates**
  - **operations:** union, intersection, difference, subset

- Why are hash tables **"unordered"?**
  - a good hash function redistributes the original data **uniformly** across the underlying index space
    - but **must not be random,** or else subsequent lookups would fail!
  - no guarantee that data "value" or insertion order would be preserved
  - hence, we **give up ordering for speed**

Python sets

- Syntax
  - **`{1, 2, 3}`** is a set
  - **`{1: "a", 2: "b", 3: "c"}`** is a dict
- Documentation
  - https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
  - https://docs.python.org/3/tutorial/datastructures.html#sets
- Binary operators
  - **`|`** (union), **`&`** (intersection), **`-`** (difference), **`<`** (subset)
- Empty containers
  - **`list()`**, **`dict()`**, **`tuple()`**, **`set()`**
  - **`[]`** is empty list
  - **`{}`** is empty dict, not set
- **`set`**s are mutable, use **`frozenset`** if want to use within **`set`**s or as **`dict`** keys

Back to Python dicts

- Original purpose of dictionaries is to maintain mappings
  - could implement as regular **`list`**s, but scanning for keys would be slow
  - hence, Python **`dict`**s borrow hashing idea from typical set implementations
- Conceptually, a Python **`dict`** is a set with **keys as items,** and a **value attached to each key**
  - **to look up by key:** first hash on key, check chaining for actual key entry, then return attached value
  - since Python 3.7, **`dict`**s actually use an alternative implementation that preserves insertion order

Graph Search

Directed vs undirected graphs

- When considering **connectivity** between two nodes, directed seems to be a special case of undirected
  - directed distinguishes between start/source/origin node and end/target/destination node of an edge
- But from a **representation** standpoint, undirected is a special case of directed
  - an undirected edge can be represented as two opposing directed edges
- Hence we will focus on **directed graphs (digraphs)**
- For now, we will also ignore edge weights
  - all edges effectively have unit weight 1

Traversing/exploring a graph

- Pre-lecture code demonstrated random traversal of edges
- Traversal is ultimately about exploring **connectivity**
  - **Question 1:** What nodes are **reachable** from a given node?
  - **Question 2:** If reachable, what is a **valid path** from a **start** to a **goal** node?
- Random traversal is not very systematic, nor efficient
  - relies on ***eventually*** exploring all neighbors of any node
  - when finally reach goa, path may be very convoluted

A more systematic way

- Simplify the problem at first, consider only **tree graphs**
  - **each node branches** to its own children, **no overlap** with other branches/subtrees
  - e.g., a **filesystem** of folders and files, files and empty folders are leaf nodes
- Consider a **layer-by-layer strategy**
  - explore all child nodes of start/root
  - then explore all the children of those nodes, i.e., start's grandchildren
  - essentially asking is the goal one step away, two steps away, etc.
- Each layer is a **frontier** in our search, called **breadth-first search (BFS)**

BFS implementation on trees

- Represent **current and next frontiers**
  - each frontier is a **list of nodes**
  - initial level-0 frontier contains only start node
- **Expand each node** on the current frontier into nodes on the next frontier
  - keep repeating until final frontier is empty
- This strategy will identify if a goal is reachable from the start, but **can't construct path from start to goal**
  - lost information when discarding previous frontiers
- Fix: **frontiers are lists of partial paths,** extend the paths onto the next frontier
  - represent a **path** as a **sequence/list of nodes**
  - **note:** frontiers actually just need to be sets, but we used lists to help trace verify deterministic neighbor ordering

BFS implementation on graphs

- In a **tree,** always a **unique path** between connected nodes
- Now consider **general graphs,** e.g., a network of flight routes
- Apply same strategy of frontiers, but now may re-encounter **previously seen nodes**
  - some nodes in a frontier may have edge to each other
  - same may even have edges back to previous frontiers
  - want to **avoid putting those neighbors on next frontier**
- **Solution:** maintain a seen/visited collection
  - excellent opportunity to use a **`set`,** need fast insert and fast check
  - when expanding to new node, **check before placing on next frontier**
  - put **actual new nodes** (previously unseen) in **visited set**

BFS and shortest paths

- **Key property:** level-n frontier contains all nodes whose shortest distance from start node is n edges away
  - hence the path found to any node will be a **shortest path**
  - there may be other paths from start to goal, some may be longer or equal length, none will be shorter
- **Proof sketch**
  - shortest path to start's neighbors is 1, property holds for level 1
  - consider level 2's nodes
    - already know there exists path of length 2, so can rule out 3 and above as shortest distances
    - what if some of them had a shortest distance of 1?
    - then they must have been encountered in level 1 instead
  - repeat reasoning for level 3, 4, etc.

Next week

- **Issue:** at any point, BFS requires memory to store a full frontier
  - many graphs are "radially dense," frontiers get ever wider
  - will waste time/memory exploring unnecessary portions of graph
- **Monday:** depth-first search (DFS), potential to save on memory
- **Wednesday:** Dijkstra's algorithm for shortest paths on weighted graphs
  - alternative BFS-based implementation on Pset 4