Closest-First Graph Traversal

COMPSCI 230 — Discrete Math

March 30, 2017
Overview

Closest-First Graph Traversal (a.k.a. Best-First Search or Dijkstra’s Algorithm)

1. Google Maps
2. Efficiency
3. Graph Implementation
4. Closest-First Traversal
Graph Traversal Application Example

• Google Maps finds the shortest route from \( a \) ("source") to \( b \) ("destination")
• Let us find some route first
• The graph \( G \) is the roadmap of the USA
• Vertices are intersections between roads (mostly cities)
• Edges are roads that connect intersections directly
• Traverse \( G \) from \( a \) until you encounter \( b \)
• Depth-first or breadth-first?
• Main problem with depth-first: The graph is very large
• Traveling from neighbor to neighbor can lead to very deep trees
• Breadth-first seems a much better idea
  [Tree \( T \) is partial (not spanning)]
NCAA Championship

What team is UNC playing in the Final Four?

Pick one

A: Gonzaga
B: Duke
C: Oregon
D: Who cares? We lost

(Not Graded)
Traversals

In the depth-first traversal of a graph, a vertex is *scanned* when ...

Pick one

A: the contents of the vertex have been accessed
B: the vertex is in the tree being grown
C: its neighbors are in the tree being grown
D: the vertex is part of a Hamilton path
E: the vertex has been removed from the graph
Trees

In graph theory, every tree ...

Pick one (Graded)

A: has a Hamilton path
B: has an Euler path
C: both A and B
D: has a root
E: has a unique, simple path from any vertex to any other vertex
A Small Example

STL = 'St. Louis'  
CHI = 'Chicago'  
IND = 'Indianapolis'  
NAS = 'Nashville'  
CLE = 'Cleveland'  
COL = 'Columbus'  
PIT = 'Pittsburgh'  
WDC = 'Washington'  
DUR = 'Durham'  
NYC = 'New York'  
PHI = 'Philadelphia'

Find the shortest route from St. Louis to New York
Efficiency Considerations

- For large graphs and frequent use, efficiency matters
- Consider breadth-first

```python
def gbf(G, u):
    q = []
    VT = []
    q.append(u)
    while len(q) > 0:
        u = q.pop(0)
        VT.append(u)
        for v in neighbors(u, G):
            if v not in VT and v not in q:
                q.append(v)
```

- Operations in red can be expensive for large graphs or long paths
- Need to access any neighbor of any node in a large graph, or any item in a list, quickly
- What to do? *Hashing*
The *Purpose* of Hashing

- Lists are slow: To access item $k$ takes $k$ steps
+ Lists need no addresses: Just follow `next` pointers
+ Arrays are fast: To access item $k$ takes 1 step
- Arrays require dense numerical addresses: $1, 2, \ldots, n$
  - *Hashing* is a method to convert anything (immutable) to a number
  - Think $\text{hash('St. Louis')} \rightarrow 37 \ldots$
  - But done so that all numbers are packed tightly
  - Then everything can be stored in “arrays” for fast access
  - We’ll see later in the course how this can be achieved
  - Hashing yields (almost) the best of both worlds: universal indexing, fast access
  - Comes for free in Python: `dict()`
Python Dictionaries

- Hashing, and the consequent ability to use dictionaries is why Python distinguishes between mutable and immutable objects
- You can hash a tuple ('a', 'b')
- You cannot hash a list ['a', 'b']
- `hash(( 'a' , 'b' ))` is always the same
- So we can access later what we store now
- `hash([ 'a' , 'b' ])` could change: disaster!
Python Dictionary Example

• Roads connect Pittsburgh and Philadelphia directly to New York City

NYCneigh = {'Pittsburgh':371, 'Philadelphia':97}

• Numbers here are distances in miles, not hash codes!

• 'Pittsburgh' is the key, 371 is the value

• Hashing occurs under the hood: Python converts 'Pittsburgh' to some internal number

• Alternative syntax:

NYCneigh = dict()
NYCneigh['Pittsburgh'] = 371
NYCneigh['Philadelphia'] = 97

• Either way, NYCneigh['Philadelphia'] returns 97

• Access is constant-time on average
A Simple Graph Implementation

- Use a `dict` of `dicts` to store edges and distances
- (Say `STL = 'St. Louis'` to save typing)

```python
d = {
    CHI: {STL: 297, IND: 183, CLE: 345},
    CLE: {CHI: 345, IND: 317, COL: 142, PIT: 134},
    WDC: {COL: 397, PIT: 242, PHI: 139, DUR: 258},
    DUR: {NAS: 516, COL: 458, WDC: 258},
    NYC: {PIT: 371, PHI: 97},
    PHI: {PIT: 304, WDC: 139, NYC: 97}
}
```

- Think of it as a sparse $11 \times 11$ matrix of distances where rows and columns are indexed by city names
- Call it `d` instead of `G` to emphasize distances

```python
d['Nashville']['Durham'] = 516
```

`d['Nashville']['New York']` generates a key error
How Do We Store the Tree?

- Traversal of a graph $G$ yields a tree $T$, at least conceptually.
- $T$ stores paths from St. Louis to all other cities.
- Once we find $b$, we only need a tree $T$ to trace our way back from $b$ to $a$ in $T$ to form the route.
- If we only traverse the tree from leaf to root, we don’t need to know where the root is.
- Each vertex in $G$ just needs to know its parent.

```
[Diagram of a tree with vertices a and b]
```

- From $b$, we can follow the pointers back to $a$.
- This is an *upward-linked tree*: One pointer up rather than $n$ pointers down: lightweight!
Upward-Linked Tree Implementation

- A dictionary for the parent of each node traversed:
  \[
  \text{parent} = \text{dict}()
  \]
- If \( u \) is the parent of \( v \):
  \[
  \text{parent}[v] = u
  \]
  For the first node traversed:
  \[
  \text{parent}[a] = \text{None}
  \]
- To trace our steps back from \( b \) to \( a \):
  \[
  \text{path} = \text{backtrack}(b, \text{parent})
  \]
  where
  \[
  \text{def backtrack(dest, parent):
    path = []
    u = dest
    while u:
      path.insert(0, u)
      u = parent[u]
    return path}
  \]
  \[
  \text{Insert } u \text{ at the } \text{beginning} \text{ of } \text{path} \text{ because we are going backwards}
  \]
  \[
  \text{Simple, no?}
  \]
A First Google Maps

- Find some route between $a$ and $b$
- Breadth-first traversal, stop at $b$ (partial tree only)
- Upwards-linked tree
- Use `dict` to store VT for efficiency

```python
def route(d, a, b):
    q = []  # queue
    q.append(a)
    parent = dict()  # upward-linked tree
    parent[a] = None  # root has no parent
    VT = dict()  # vertices traversed (dictionary!)
    while len(q) > 0:
        u = q.pop(0)  # fewest edges from $a$
        VT[u] = True  # value is irrelevant
        if u is b: return backtrack(u, parent)  # Done!
        for v in neighbors(u, d):  # Fast!
            if v not in VT and v not in q:  # Fast!
                q.append(v)
                parent[v] = u
    return None  # Did not find $b$ anywhere
```
A More Useful Google Maps?

STL = 'St. Louis'
CHI = 'Chicago'
IND = 'Indianapolis'
NAS = 'Nashville'
CLE = 'Cleveland'
COL = 'Columbus'
PIT = 'Pittsburgh'
WDC = 'Washington'
DUR = 'Durham'
NYC = 'New York'
PHI = 'Philadelphia'

```
print(*route(d, STL, NYC), sep=', ')
```

St. Louis, Chicago, Cleveland, Pittsburgh, New York

- Seems a bit long, but it’ll get you there
- How to find the **shortest** route?
Shortest Route (in Miles)

- Key change: **closest-first** rather than breadth-first
- Among the cities on the current queue, ...
- Breadth-first: ... pick the city that is the fewest edges away from \( a \)
- This is what the queue ensures:
  few edges from \( a \rightarrow \) ahead in the queue
- Closest-first: pick the city that is closest to \( a \) in miles
- Need to manage the queue differently
The Priority Queue

• We want an “unfair queue” that always returns the city that is closest to \( a \), as opposed to the first one to be queued.

• This is called a priority queue, a data structure that works on keys (city names in the example) and values (distances from \( a \)) and performs at least the following operations efficiently:
  - Make an empty queue
  - Add a key to the queue and associate a value to it
  - Return the key with lowest value in the queue, and remove it from the queue
  - Change the value (and therefore the position in the queue) for a key already in the queue
  - Delete the queue
The Python `heapdict` Module

- The Python `heapdict` module implements a priority queue as a dictionary, so we can also quickly find the value of any key.
  - Not a standard module, so you need to first run `python3 -m pip install heapdict`.
  - Then, use `from heapdict import *`.
  - The syntax is beautifully simple:
    - `pq = heapdict()` creates a priority queue.
    - `pq[key] = value` inserts a key and its value.
    - `pq[key] = value` also changes the value of a key already in the queue.
    - `key, value = pq.popitem()` returns the key, value pair with lowest value and removes it from the queue.
    - `value = pq[key]` returns the value of the key (and leaves the queue unmodified).
    - `del pq` deletes the queue.
Closest-First Traversal: Terminology

- Source city: \( a \)
- Destination city: \( b \)
- \( d(u, v) \): *edge distance*, miles between \( u \) and \( v \), encoded as weights of the road-graph edges
- \( D_v \): at the current time in the algorithm, length of the shortest route found so far from \( a \) to \( v \)
- \( \text{parent}(v) \): the city just before \( v \) in the shortest route so far from \( a \) to \( v \)
Closest-First Traversal: Three Key Sets

- **C**: (Completed): Cities for which the final shortest route has been found
- **Q**: (Queued): All cities (i) whose parent is in $D$, and (ii) is still under consideration
- **U**: Untraversed cities
- **U** should not figure anywhere in the algorithm (too large!)
Example

STL = 'St. Louis'
CHI = 'Chicago'
IND = 'Indianapolis'
NAS = 'Nashville'
CLE = 'Cleveland'
COL = 'Columbus'
PIT = 'Pittsburgh'
WDC = 'Washington'
DUR = 'Durham'
NYC = 'New York'
PHI = 'Philadelphia'

Nodes: Queued, Complete, Untraversed.
Green Edges: Part of Final Tree
Closest-First Traversal: Algorithmic Idea

• Put $a$ in $Q$ (priority queue), with distance $D_a = 0$
• As long as the queue is not empty:
  • Pop $(u, D_u)$ from $Q$. This is the closest city in the queue that is reachable from $a$ and is still under consideration
  • So put $u$ into $C$
• If $u$ is $b$, we are done: trace the route back from $b$ to $a$
• Otherwise, for all neighbors $v$ of $u$ not in $C$:
  • The shortest route from $a$ to $v$ such that $u = \text{parent}(v)$ has length $\delta = D_u + d(u, v)$
  • Case 1: $v$ already in $Q$: update $D_v$ to $\delta$ if $\delta < D_v$
  • Case 2: $v$ in $U$: add it to $Q$ with $D_v = \delta$
  • If $D_v$ has changed, record $\text{parent}(v) = u$
Closest-First Traversal: Key Point

• When we pop \( u \) off \( Q \), we put it into \( C \), the set of nodes we no longer touch (“complete”)
• Why?
• There is no way to decrease \( D_u \) further, because
  • Routes through other cities in \( Q \) are longer
  • We must go through some city in \( Q \) to reach \( u \) from \( a \)

• Therefore, we are done with \( u \) forever
Closest-First Traversal: Code

def dijkstra(d, a, b):
    Q = heapdict()
    parent = dict()
    C = dict()
    Q[a], parent[a] = 0, None
    while len(Q):
        u, Du = Q.popitem()
        C[u] = True
        if u is b: return backtrack(u, parent), Du
        for v in neighbors(u, d):
            if v not in C:
                Dv = Q[v] if v in Q else inf
                delta = Du + d[u][v]
                if delta < Dv:
                    Q[v], parent[v] = delta, u
    return None, inf
Some route: route(d, STL, NYC)

['St. Louis', 'Chicago', 'Cleveland', 'Pittsburgh', 'New York']

Computing distance by hand yields $D(a, b) = 1147$ miles

Shortest route: dijkstra(d, STL, NYC)

(['St. Louis', 'Indianapolis', 'Columbus', 'Pittsburgh', 'New York'], 974)