# Geography and CS

Philip Chan

## Maps

- Problem 1
  - Where am I?
  - “Localization”
- Problem 2
  - How do I get there?
  - “Navigation”

# Localization

Problem 1

## Localization--Where am I?

- Cell phone
- GPS—Global Positioning System

## Localization--Where am I?

- Cell phone
  - Reference points:
- GPS—Global Positioning System
  - Reference points:

## Localization--Where am I?

- Cell phone
  - Reference points: cell towers
- GPS—Global Positioning System
  - Reference points: satellites

## Localization--Where am I?

- Cell phone
  - Reference points: cell towers
- GPS—Global Positioning System
  - Reference points: satellites
- How many reference points are needed to fix the location?

## Localization--Where am I?

- Cell phone
  - Reference points: cell towers
  - Need 3 reference points
- GPS—Global Positioning System
  - Reference points: satellites
- How many reference points are needed to fix the location?

## Localization--Where am I?

- Cell phone
  - Reference points: cell towers
  - Need 3 reference points
- GPS—Global Positioning System
  - Reference points: satellites
  - Need 4 reference points,
  - but 3 are ok if I know that I'm not floating in space above the satellites
- How many reference points are needed to fix the location?

## Localization [2D] (Problem Formulation)

- Given (input)
  - Coordinates of the reference points
  - Distances from the reference points
- Find (output)
  - Coordinates of the location

## Localization [2D] (Problem Formulation)

- Given (input)
  - Coordinates of the reference points
    - $(x_1, y_1)$, $(x_2, y_2)$, $(x_3, y_3)$
  - Distances from the reference points
    - $d_1$, $d_2$, $d_3$
- Find (output)
  - Coordinates of the location
    - $(x, y)$

## Algorithm

- What is the mathematical relationship among the variables?

## Algorithm

- What is the mathematical relationship among the variables?
- Hint: given two points [two pairs of (x,y) coordinates], what is the distance between them?

## Navigation

Problem 2

## Navigation [Problem understanding]

- Finding a route from the origin to the destination
- “Static” directions
  - Mapquest, Google maps
- “Dynamic” on-board directions
  - GPS navigation
    - if the car deviates from the route, it finds a new route

## Navigation [Problem Formulation]

- Given (input)
  - Map
  - Address of the origin
  - Address of the destination
- Find (output)
  - Turn-by-turn directions
- Simplification
  - In the same city, all two-way streets, all left and right turns are allowed, no overpass/tunnels…

## Navigation [Problem Formulation → Graph Problem]

- Given (input)
  - Map → ?
  - Address of the origin → ?
  - Address of the destination → ?
- Find (output)
  - Turn-by-turn directions → ?
- Simplification
  - In the same city, all two-way streets, all left and right turns are allowed, no overpass/tunnels…

## Navigation [Problem Formulation → Graph Problem]

- Given (input)
  - Map → edge=street, vertex=intersection, weight=length
  - Address of the origin → vertex
  - Address of the destination → vertex
- Find (output)
  - Turn-by-turn directions → ?
- Simplification
  - In the same city, all two-way streets, all left and right turns are allowed, no overpass/tunnels…

## Navigation
[Problem Formulation → Graph Problem]

- Given (input)
  - Map → edge=street, vertex=intersection, weight=length
  - Address of the origin → vertex
  - Address of the destination → vertex
- Find (output)
  - Turn-by-turn directions → shortest path
- Simplification
  - In the same city, all two-way streets, all left and right turns are allowed, no overpass/tunnels …

## Map/Street Data (input)

- Need more thoughts:
  - What do we need to know about the streets?
  - How could they be represented?

## Map/Street Data (input)

- Tessellation or Vector?
  - Tessellation:
  - Vector:

## Map/Street Data (input)

- Tessellation or Vector?
  - Tessellation: “image” of the streets
  - Vector: “description” of the streets

## Map/Street data (input)

- Vector
  - Name
  - Two end points in x,y coordinates
  - Range of house numbers
- What if the street is curvy (not straight)?

## Map/Street data (input)

- Vector
  - Name
  - Two end points in x,y coordinates
  - Range of house numbers
- What if the street is curvy (not straight)?
  - “Polyline”
  - Additional intermediate x,y coordinates and house numbers
- Street name, $(x_1, y_1, h_1)$, $(x_2, y_2, h_3)$, …

## Map/Street data (input)

- What if a straight street has multiple intersections?

## Map/Street data (input)

- What if a straight street has multiple intersections?
  - Polyline (like curvy street)
    - Additional x,y coordinates and house numbers

## Algorithm Overview

1. Preprocessing
   - Convert the map, origin & destination into a graph
2. Main algorithm
   - Dijkstra's shortest path algorithm
3. Postprocessing
   - Convert shortest path to turn-by-turn directions

## Vertices in the graph

- What should be a vertex?
  - Intersections
  - How about intermediate points in the polyline of a curvy street?

## Vertices in the graph

- What should be a vertex?
  - Intersections
  - How about intermediate points in the polyline of a curvy street?
    - No, fewer vertices, but need to sum segment distances
    - (Yes, make program simpler)
- Each vertex corresponds to a pair of x,y coordinates
- What is the weight of an edge?

## Curvy streets vs intersections

- An intermediate point of a polyline could be:
  - intersection → a vertex
  - part of a curvy street → not a vertex
- Vector representation:
  - Street name, $(x_1, y_1, h_1)$, $(x_2, y_2, h_3)$, …
- How could we tell the difference?

## Curvy Streets vs Intersections

- Additional info in vector representation
  - intersection: Pointer to the cross street *s* [assuming only one cross street; a list otherwise]
  - curvy street: no pointer
  - Street name, $(x_1, y_1, h_{1,} s_1)$, $(x_2, y_2, h_{2,} s_2)$, …

## Curvy Streets vs Intersections

- Additional info in vector representation
  - intersection: Pointer to the cross street *s* [assuming only one cross street; a list otherwise]
  - curvy street: no pointer
  - Street name, $(x_1, y_1, h_{1,} s_1)$, $(x_2, y_2, h_{2,} s_2)$, …
- No additional info in vector representation
  - Intersection: Two streets with the same vertex ID
  - A convenient vertex ID would be?

## Curvy Streets vs Intersections

- Additional info in vector representation
  - intersection: Pointer to the cross street *s* [assuming only one cross street; a list otherwise]
  - curvy street: no pointer
  - Street name, $(x_1, y_1, h_{1,} s_1)$, $(x_2, y_2, h_{2,} s_2)$, …
- No additional info in vector representation
  - Intersection: Two streets with the same vertex ID
  - A convenient vertex ID would be?
  - (concatenation of) x, y coordinates
- Time-space tradeoffs?

## Converting Address to Vertex

- For the origin and destination
- Given street name and house number
  - Create:
    - One temporary vertex (unless at an intersection)
    - Two temporary edges, why?

## Converting Address to Vertex

- For the origin and destination
- Given street name and house number
  - Create:
    - One temporary vertex (unless at an intersection)
    - Two temporary edges, why?
- What are the x,y coordinates of the new temporary vertex?
- What are the weights of the two new temporary edges?

## Converting Address to Vertex

- Tradeoffs between:
  1. Replace original edge with temporary vertex & edges [then reverse the process later]
  2. Add temporary vertex & edges [then reverse the process later]

## Main Algorithm

- If you do not know about Dijkstra's algorithm
  - How would you solve the shortest path problem?

## Main Algorithm—Greedy Algorithm

- Greedy algorithm
  1. Pick the closest vertex (shortest edge)
  2. Go to the vertex
  3. Repeat until the destination vertex is reached
- Does this always find the shortest path?
- If not, what could be a counter example?

## Main Algorithm-- Dijkstra's shortest path algorithm

- What are the key ideas?

## Main Algorithm-- Dijkstra's shortest path algorithm

- What are the key ideas?
  - Similar to BFS:
    - pick a leaf and expand its children
  - Different in which leaf to pick, how?

## Main Algorithm-- Dijkstra's shortest path algorithm

- What are the key ideas?
  - Similar to BFS:
    - pick a leaf and expand its children
  - Different in which leaf to pick, how?
    - the shortest length so far
    - instead of the fewest # of levels in BFS

## Main Algorithm-- Dijkstra's shortest path algorithm

- What are the key ideas?
  - Similar to BFS:
    - pick a leaf and expand its children
  - Different in which leaf to pick, how?
    - the shortest length so far
    - instead of the fewest # of levels in BFS
- BFS is a special case of Dijkstra's, why?

## Main Algorithm-- Dijkstra's shortest path algorithm

- What are the key ideas?
  - Similar to BFS:
    - pick a leaf and expand its children
  - Different in which leaf to pick, how?
    - the shortest length so far
    - instead of the fewest # of levels in BFS
- BFS is a special case of Dijkstra's, why?
  - fewest # of levels = shortest length so far
    - if edges are not weighted or have the same weight

## Main Algorithm-- Dijkstra's shortest path algorithm

- Why does it guarantee to find the shortest path?

## Main Algorithm-- Dijkstra's shortest path algorithm

- Why does it guarantee to find the shortest path?
  - The shortest path to vertex *A* is finalized
    - When?

## Main Algorithm-- Dijkstra's shortest path algorithm

- Why does it guarantee to find the shortest path?
  - The shortest path to vertex *A* is finalized
    - when every path to the "non-finalized" vertices is longer

## Main Algorithm-- Dijkstra's shortest path algorithm

- Why does it guarantee to find the shortest path?
  - The shortest path to vertex *X* is finalized
    - when every path to the "non-finalized" vertices is longer
      - no way to get to vertex *X* with a shorter path via "non-finalized" vertices

## Main Algorithm-- Dijkstra's shortest path algorithm

- Can we potentially stop the algorithm early?

## Main Algorithm-- Dijkstra's shortest path algorithm

- Can we potentially stop the algorithm early?
  - Single source/origin—all destinations
  - Stop when our destination is reached
- Works with directed graphs too, why?


## Main Algorithm-- Dijkstra's shortest path algorithm

- Can we potentially stop the algorithm early?
  - Single source/origin—all destinations
  - Stop when our destination is reached
- Works with directed graphs too, why?

- Interesting applet to demonstrate the alg:
- http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html

## Turn-by-turn directions (output)

- "Turn LEFT onto COUNTRY CLUB RD. 0.2 mi"
  - Turn direction
  - Street name
  - Distance on the street

## Turn-by-turn directions (output)

- Given vertices on the shortest path and map, find:
  - Turn direction
    - How do you decide you're making a turn or not?
    - If making a turn, which direction is the turn?

## Turn-by-turn directions (output)

- Given vertices on the shortest path and map, find:
  - Turn direction
    - How do you decide you're making a turn or not?
    - If making a turn, which direction is the turn?
  - Street name
    - Lookup
  - Distance
    - Lookup/calculate (and possibly addition, why?)

## Summary of Algorithm

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances
   - Convert origin/destination addresses to vertices
2. Main Algorithm
   - Dijkstra's shortest path
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation

- Again to pick data structures for efficiency/speed
  - We analyze ? of the ?

## Implementation

- Again to pick data structures for efficiency/speed
  - We analyze key operations of the algorithm
  - These key operations could be time consuming for large amounts of data

## Implementation—Where are the bulk of data stored?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances
   - Convert origin/destination addresses to vertices
2. Main Algorithm
   - Dijkstra's shortest path
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation—Where are the bulk of data stored?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances
   - Convert origin/destination addresses to vertices
2. Main Algorithm
   - Dijkstra's shortest path
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation—What are the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections
   - Convert origin/destination addresses to vertices Main Algorithm
   - Dijkstra's shortest path
2. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation—What are the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections
   - Convert origin/destination addresses to vertices -> address to x,y
2. Main Algorithm
   - Dijkstra's shortest path
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation—What are the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections
   - Convert origin/destination addresses to vertices -> address to x,y
2. Main Algorithm
   - Dijkstra's shortest path -> children; pick a leaf
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance

## Implementation—What are the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections
   - Convert origin/destination addresses to vertices -> address to x,y
2. Main Algorithm
   - Dijkstra's shortest path -> children, pick a leaf
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance -> vertex to street name

## Implementation—How to prioritize the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections
   - Convert origin/destination addresses to vertices -> address to x,y
2. Main Algorithm
   - Dijkstra's shortest path -> children, pick a leaf
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance -> vertex to street name

## Implementation—How to prioritize the key operations?

1. Preprocessing (converting input)
   - Input the map--street names, end points, house numbers
   - Create the graph—vertices/intersections, edges/distances -> neighboring intersections #4 or 1.5?
   - Convert origin/destination addresses to vertices -> address to x,y #3
2. Main Algorithm
   - Dijkstra's shortest path -> children, pick a leaf #1
3. Postprocessing (converting output)
   - Turn by turn directions—turn direction, street name, distance -> vertex to street name #2

## Implementation—Selecting data structures

- Need to find neighbors (to become "children") quickly [in Dijkstra's]
- Which graph is sparser: friends or streets?
- Graph (input):
  - Adjacency Matrix?
  - Adjacency List?
- Time
- Space

## Implementation—Selecting data structures

- Need to find neighboring vertices quickly [in converting Map to Graph]
  1. intersections (& points on a curvy street) -> vertices
  2. neighboring vertices -> edges
- Map (input)
  - Street name, ($x_1$, $y_1$, $h_1$), ($x_2$, $y_2$, $h_2$), …
- Graph (output)
  - Adjacency list
- Time
- Space

## Summary

- Problem 1: Where am I?
  - Localization
    - Geometry
- Problem 2: How do I get there?
  - Navigation
    - Preprocessing to create the graph
    - Dijkstra's Shortest Path algorithm
    - Postprocessing to give turn by turn directions

## Reading Assignment

- Handout on the advertising portion in "Prepping the Google Rocket"
  - Ken Auletta
  - *Googled—The End of the World as We Know It*
  - Penguin Press, 2009