SHORTEST PATHS

READINGS? CHAPTER 28

Lecture 20

CS2110 – Spring 2016

About A6

We give you class ArrayHeaps for a reason:

It shows the simplest way to write methods like bubble-up and bubble-down. It gives you a method to get the smaller child.

You can write A6 most easily by adapting the ArrayHeap methods to work in the new environment! Do the assignment without looking at ArrayHeap makes it MUCH harder!

Look at all the notes in the pinned Piazza note A6 FAQ before beginning —and then every other day to see whether new info has been added.

Shortest Paths in Graphs

Problem of finding shortest (min-cost) path in a graph occurs often

- Find shortest route between Ithaca and West Lafayette, IN
- Result depends on notion of cost
  - Least mileage… or least time… or cheapest
  - Perhaps, expends the least power in the butterfly while flying fastest
  - Many “costs” can be represented as edge weights

Every time you use googlemaps or the GPS system on your smartphone to find directions you are using a shortest-path algorithm

Dijkstra's shortest-path algorithm

Edsger Dijkstra, in an interview in 2010 (*CACM*):

… the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiance, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. [Took place in 1956]

Dijkstra, E.W. A note on two problems in Connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959).

Visit http://www.dijkstrascry.com for all sorts of information on Dijkstra and his contributions. As a historical record, this is a gold mine.

Dijkstra's shortest-path algorithm

Dijsktra describes the algorithm in English:

- When he designed it in 1956 (he was 26 years old), most people were programming in assembly language!
- Only *one* high-level language: Fortran, developed by John Backus at IBM and not quite finished.

No theory of order-of-execution time —topic yet to be developed. In paper, Dijkstra says, "my solution is preferred to another one … "the amount of work to be done seems considerably less."

Dijkstra, E.W. A note on two problems in Connexion with graphs. *Numerische Mathematik* 1, 269–271 (1959).

1968 NATO Conference on Software Engineering, Garmisch, Germany

Term “software engineering” coined for this conference

1968 NATO Conference on Software Engineering

- In Garmisch, Germany
- Academicians and industry people attended
- For first time, people admitted they did not know what they were doing when developing/testing software. Concepts, methodologies, tools were inadequate, missing
- The term *software engineering* was born at this conference.
- The NATO Software Engineering Conferences:

  http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html

  Get a good sense of the times by reading these reports!

1968 NATO Conference on Software Engineering, Garmisch, Germany

1968/69 NATO Conferences on Software Engineering

Editors of the proceedings

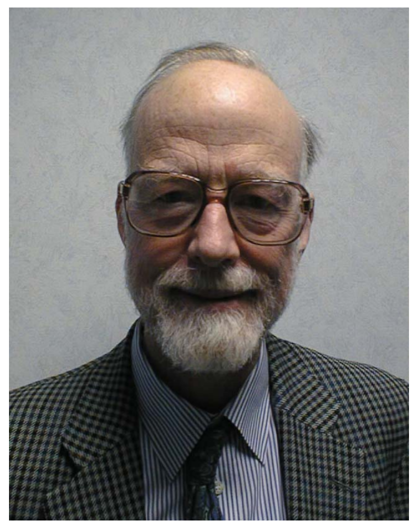
Beards

The reason why some people grow
aggressive tufts of facial hair
Is that they do not like to show
the chin that isn't there.

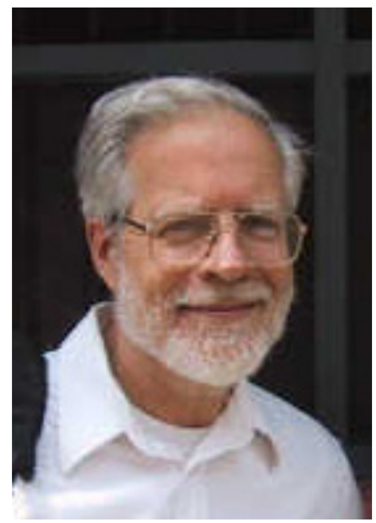
a grook by Piet Hein

Edsger Dijkstra

Niklaus Wirth

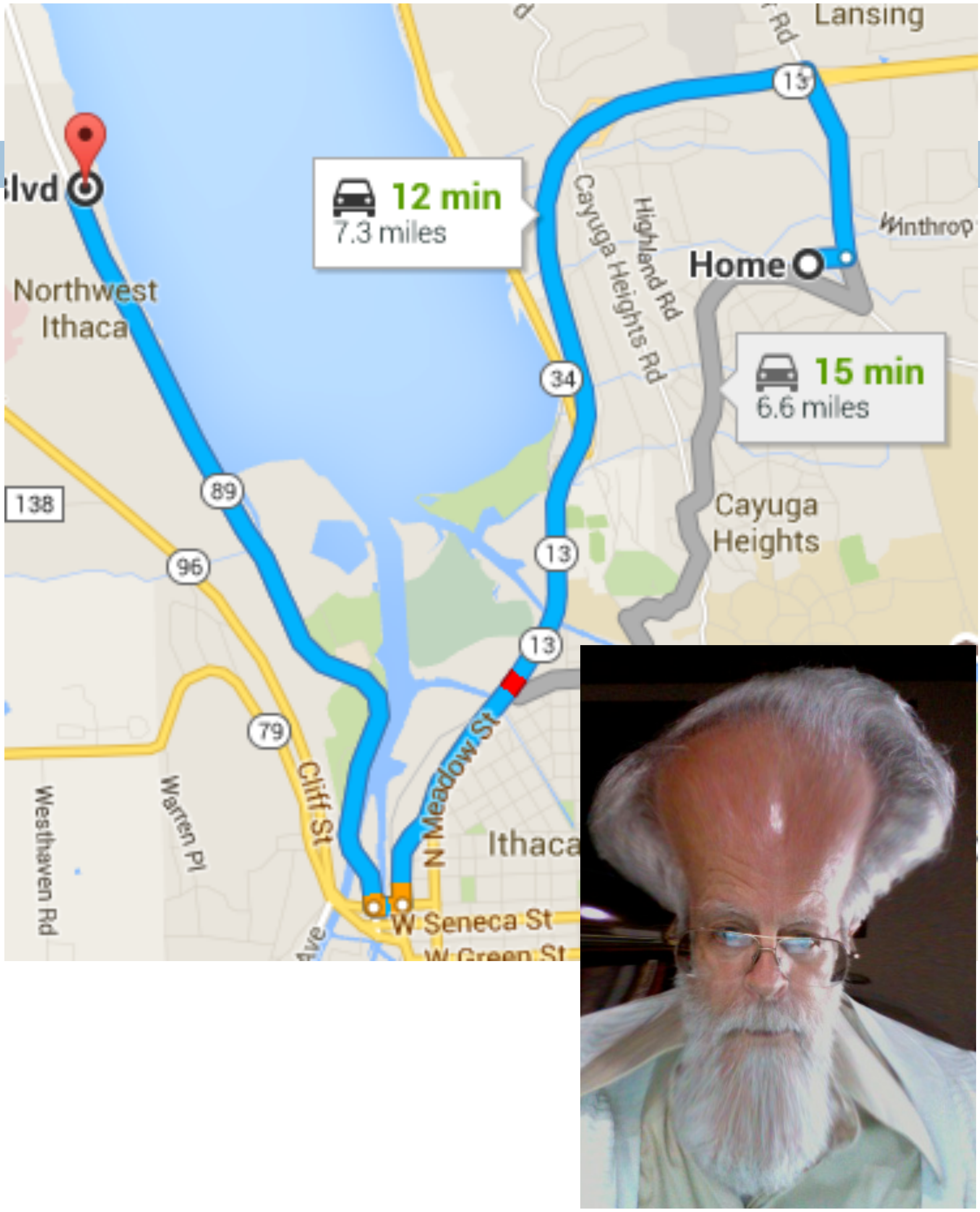


Tony Hoare



David Gries

From Gries to Tate



Googlemaps: find a route from Gries's to Tate's house.

Gives two routes

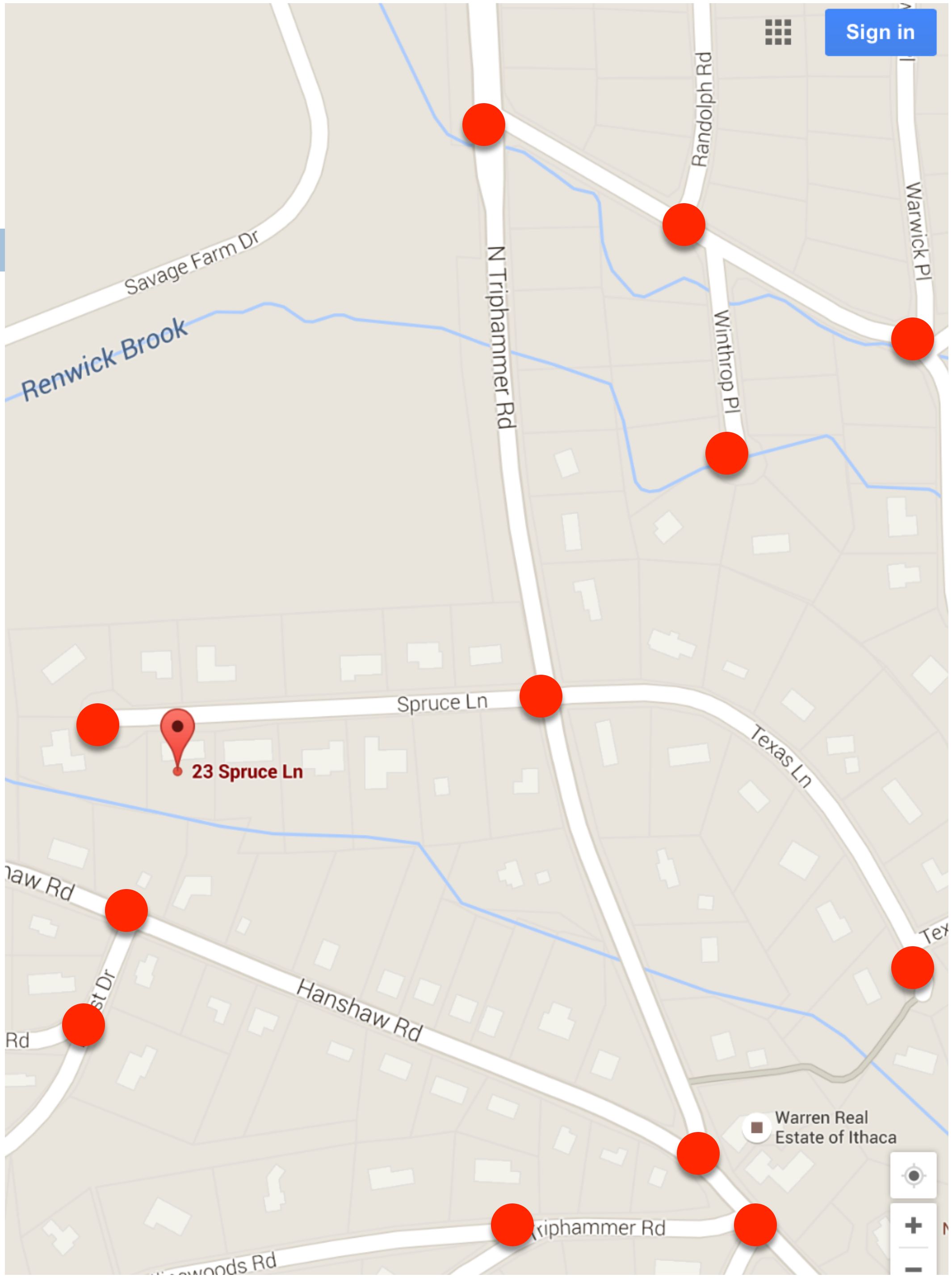
12 minutes, 7.3 miles

15 minutes, 6.6 miles

Shortest path?

Each intersection is a node of the graph, and each road between two intersections has a weight

distance?
time to traverse?
…

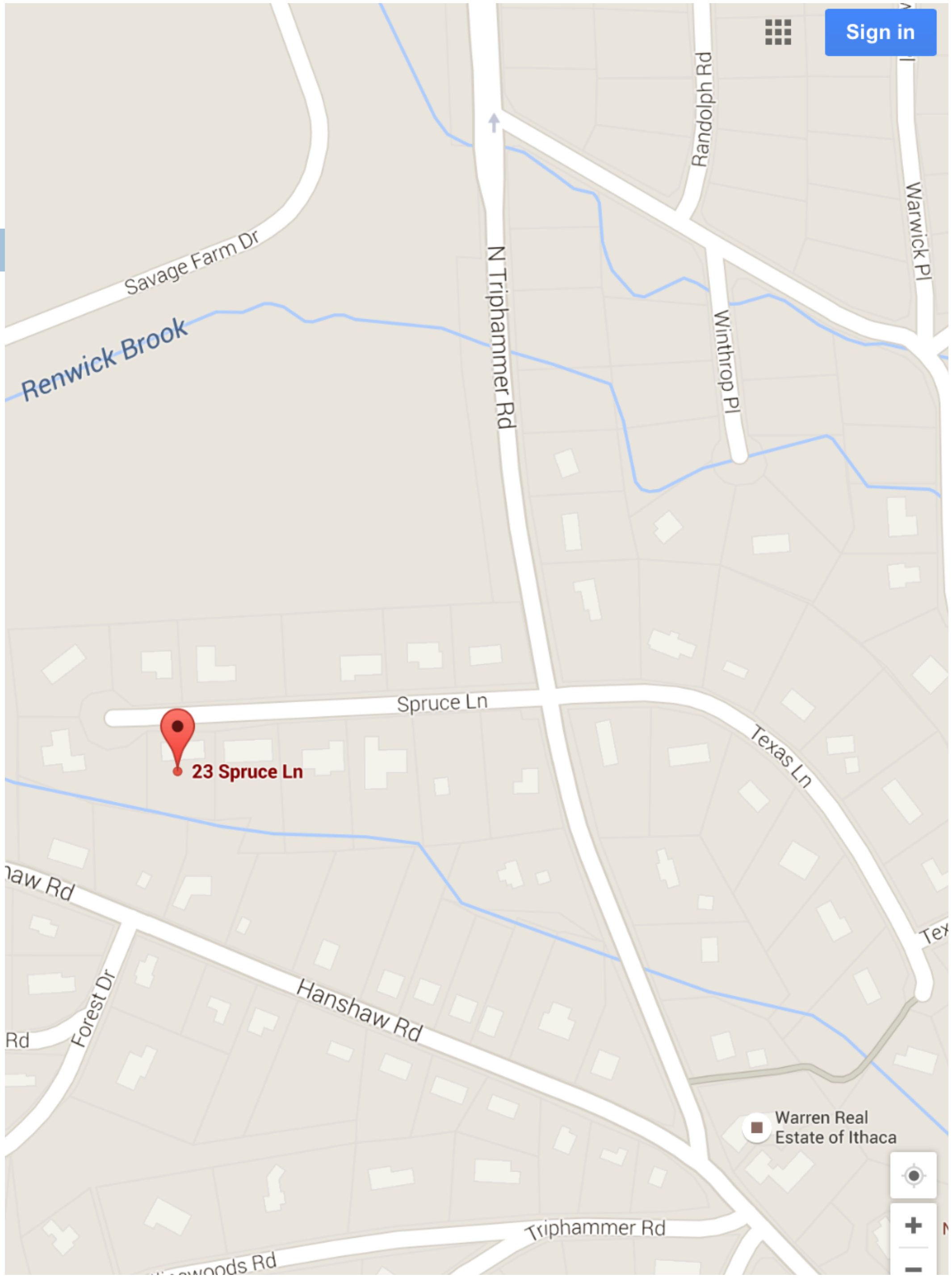


Shortest path?

Fan out from the start node (kind of breadth-first search)

Settled set: Nodes whose shortest distance is known.

Frontier set: Nodes seen at least once but shortest distance not yet known



Shortest path?

Settled set: we know their shortest paths
Frontier set: We know some but not all information

Each iteration:

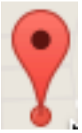
1. Move to the Settled set: a Frontier node with shortest distance from start node.

2. Add neighbors of the new Settled node to the Frontier set.

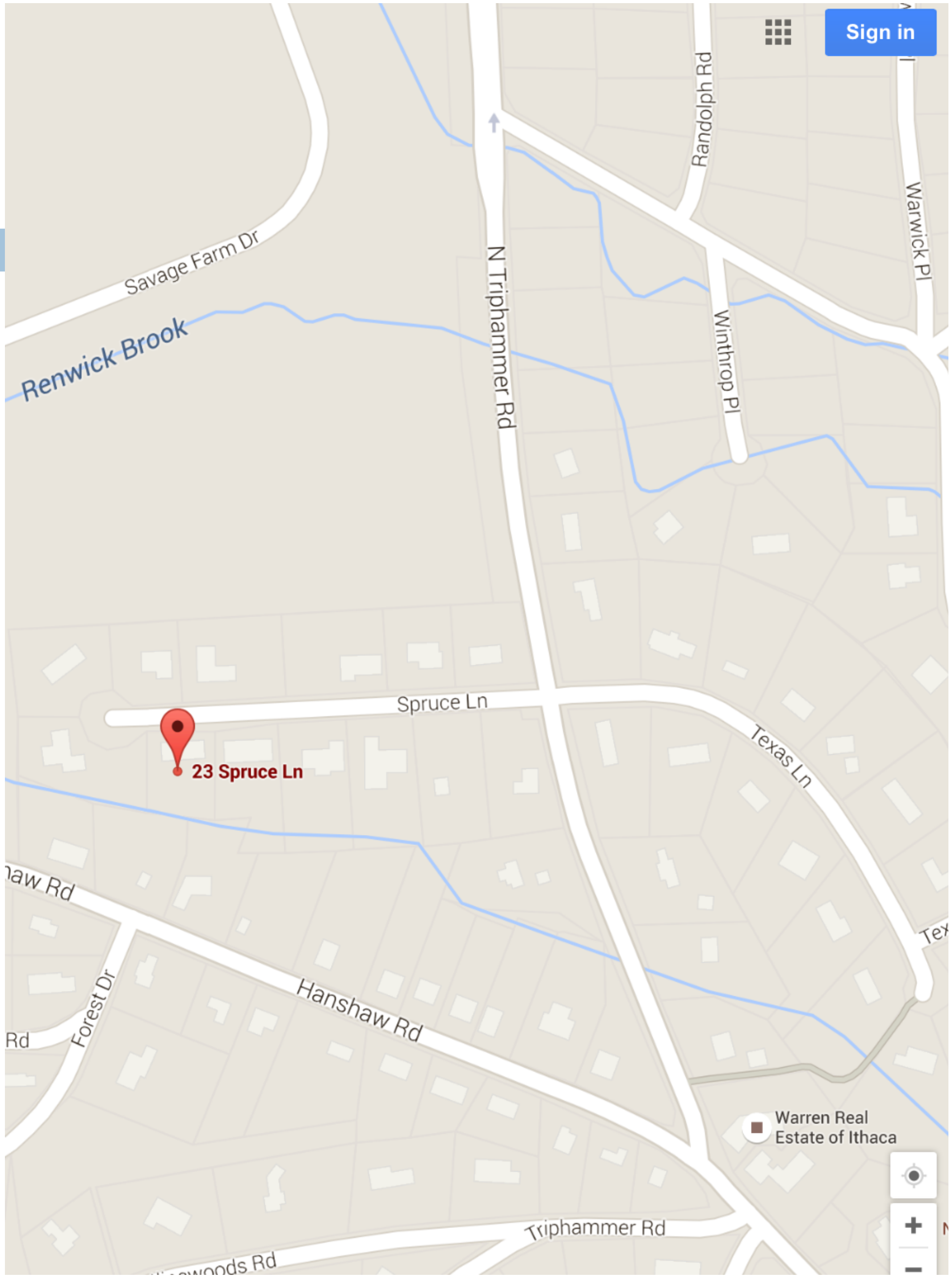
Shortest path?

Fan out from the start node (kind of breadth-first search). Start:

Settled set:

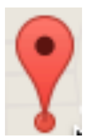
Frontier set: 

1. Move to Settled set the Frontier node with shortest distance from start

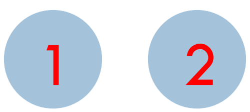


Shortest path?

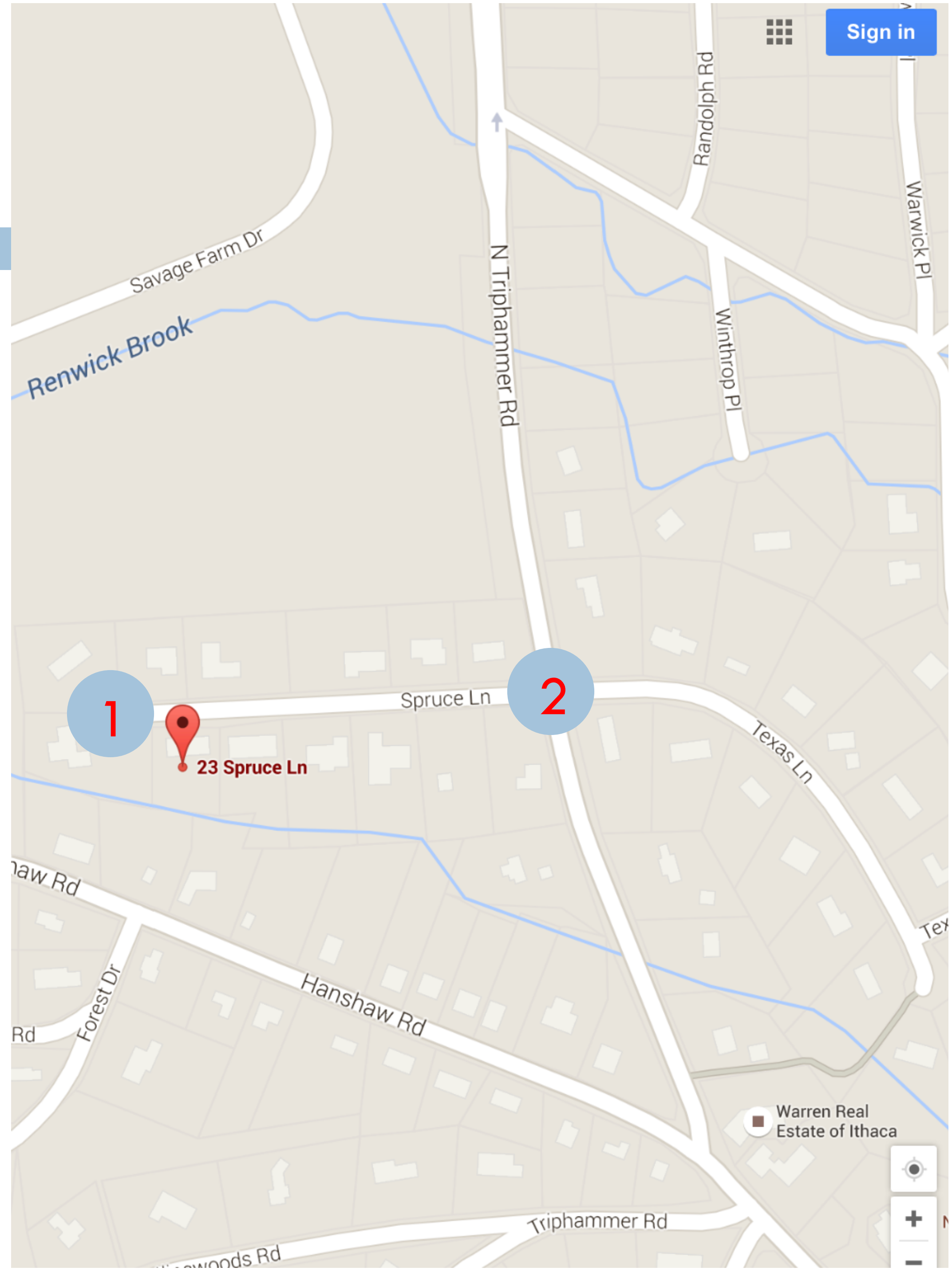
Fan out from start node. Recording shortest distance from start seen so far

Settled set: 

Frontier set:

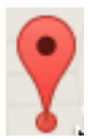


2. Add neighbors of new Settled node to Frontier



Shortest path?

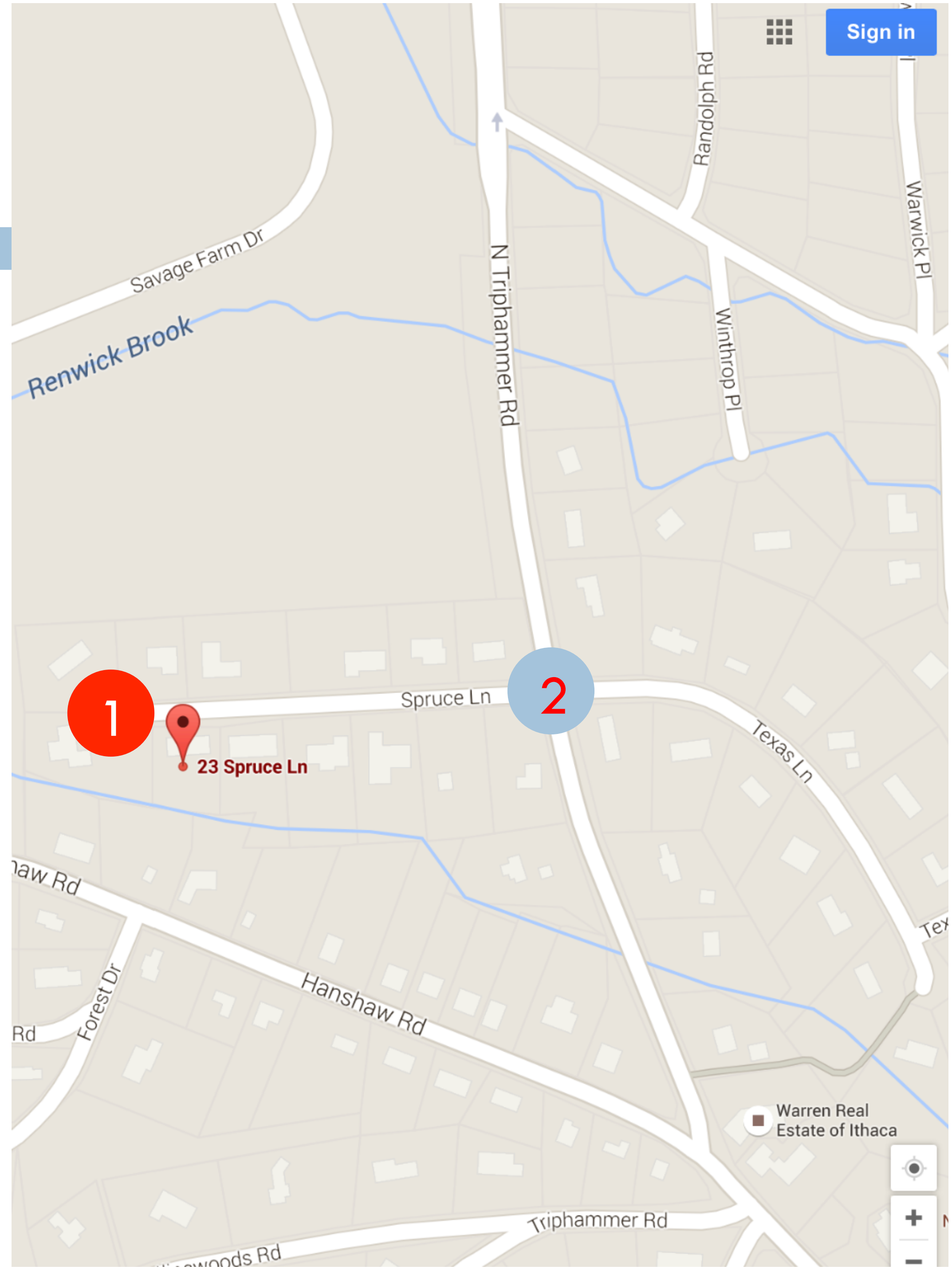
Fan out from start node. Recording shortest distance from start seen so far

Settled set:  1

Frontier set:

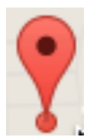
1 2

1. Move to Settled set a Frontier node with shortest distance from start



Shortest path?

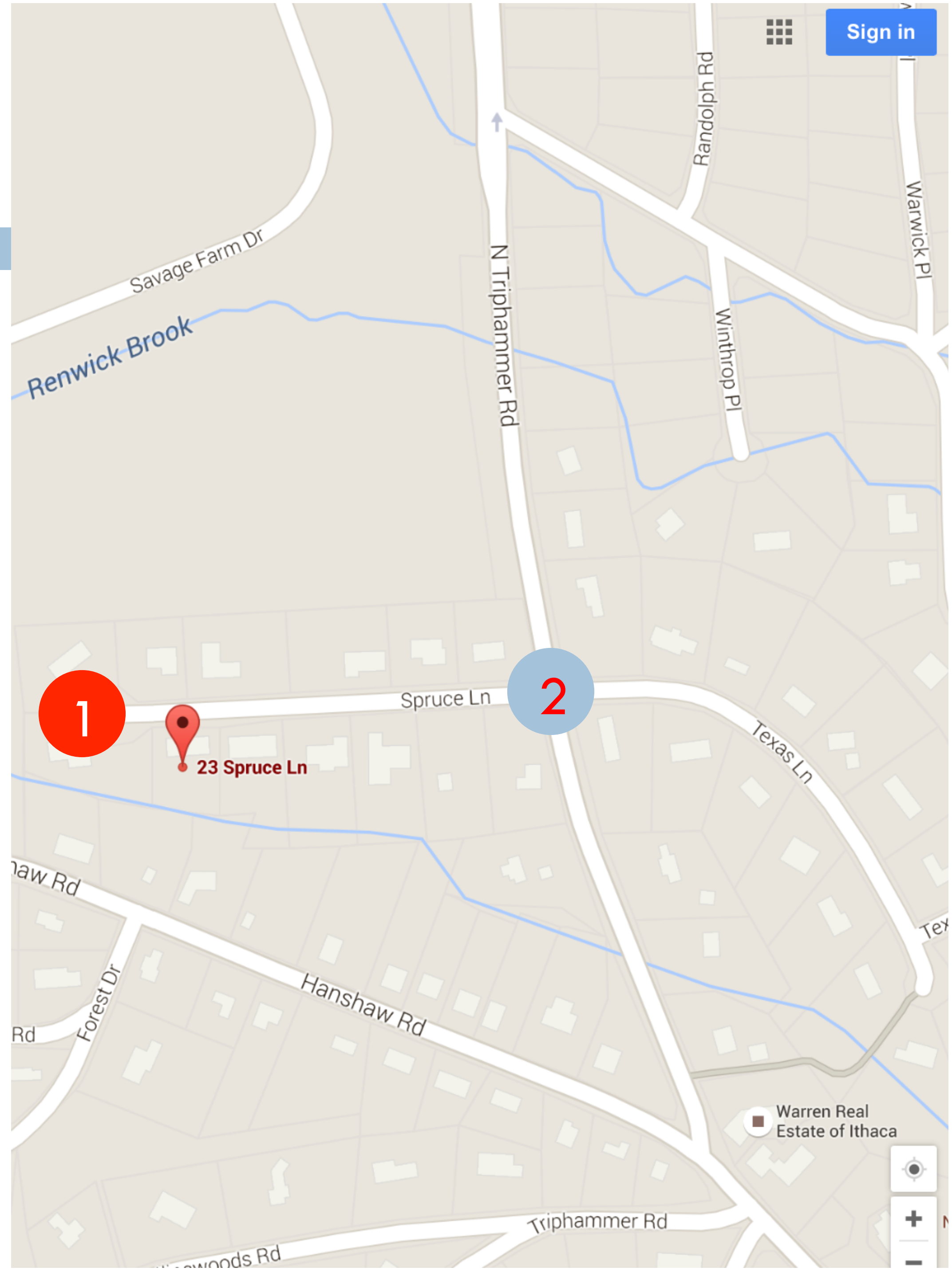
Fan out from start node. Recording shortest distance from start seen so far

Settled set:  1

Frontier set:

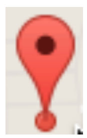
2

2. Add neighbors of new Settled node to Frontier (there are none)



Shortest path?

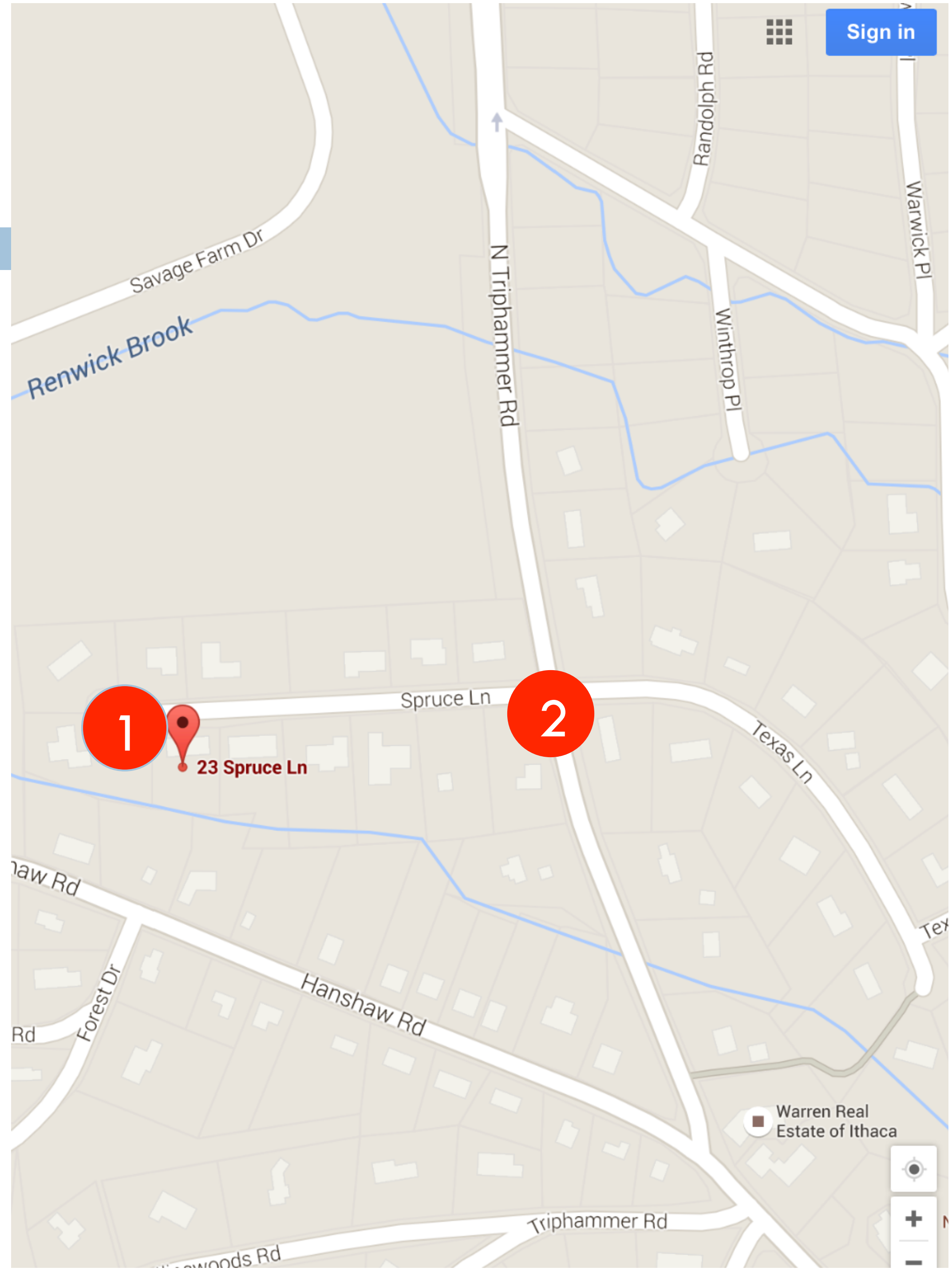
Fan out from start, recording shortest distance seen so far

Settled set:  1 2

Frontier set:

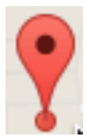
2

1. Move to Settled set a Frontier node with shortest distance from start



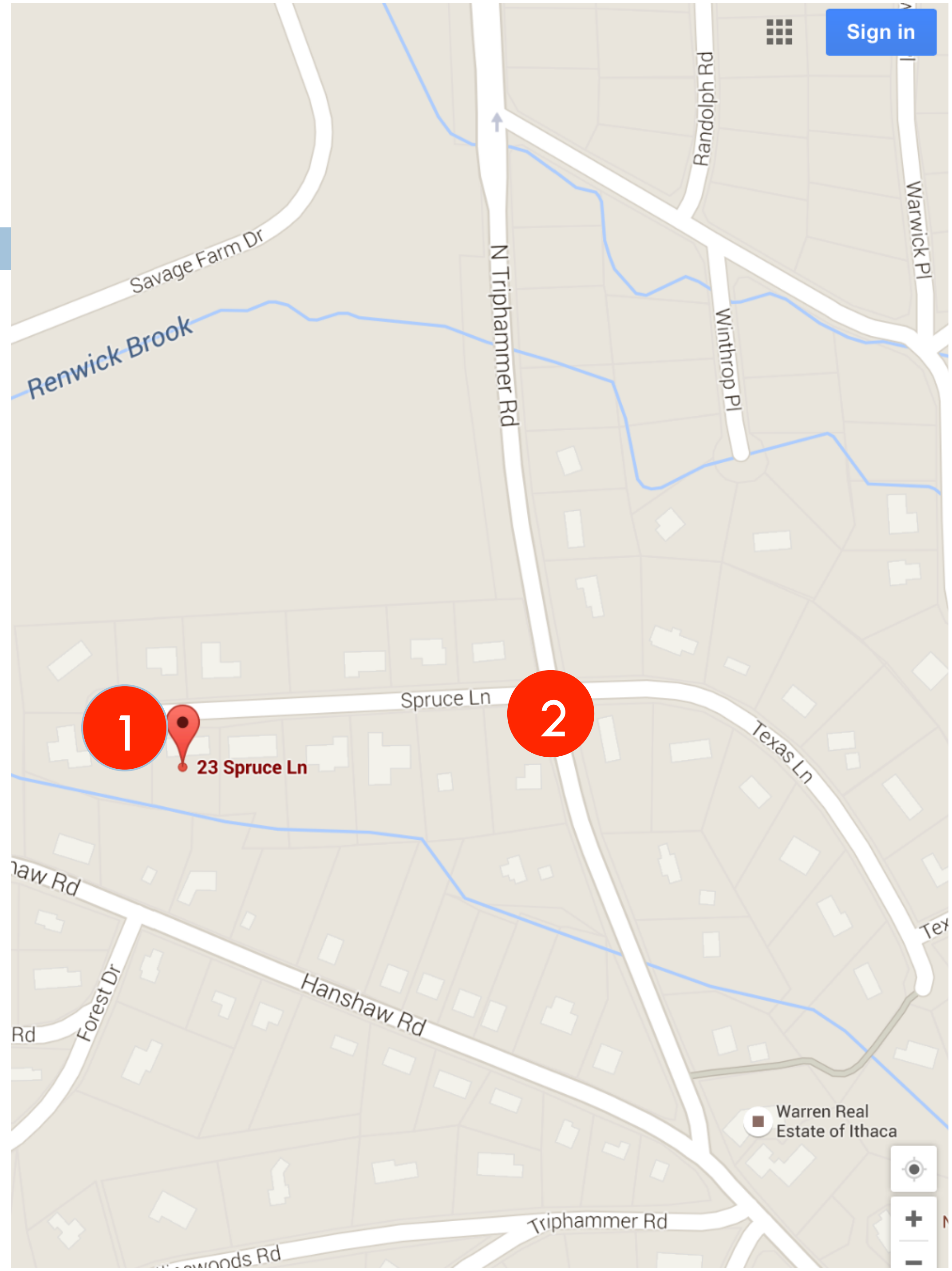
Shortest path?

Fan out from start, recording shortest distance seen so far

Settled set:  1 2

Frontier set:

2. Add neighbors of new Settled node to Frontier



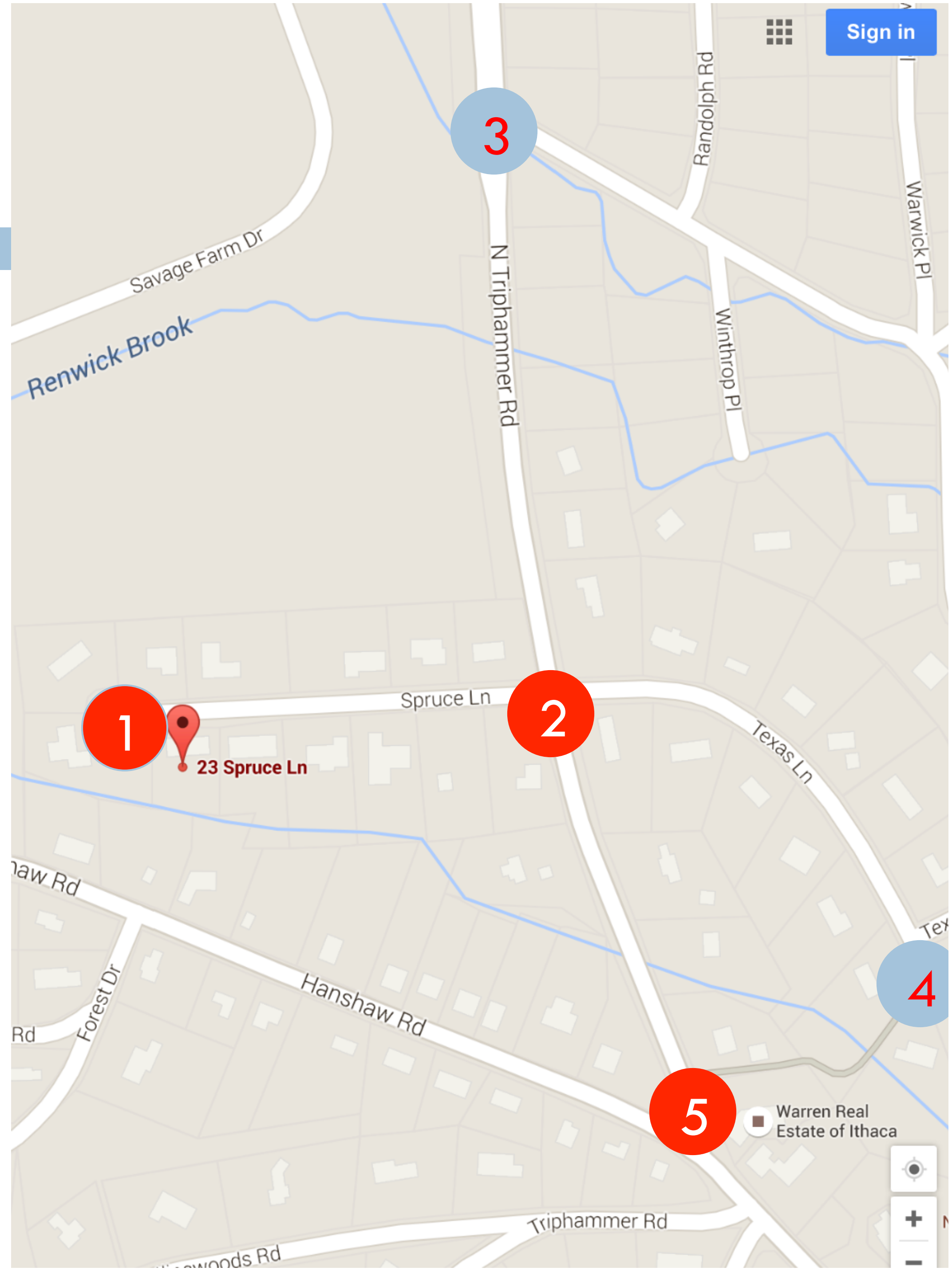
Shortest path?

Fan out from start, recording shortest distance seen so far

Settled set: 1 2 5

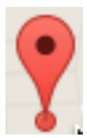
Frontier set: 3 4 5

1. Move to Settled set a Frontier node with shortest distance tfrom start



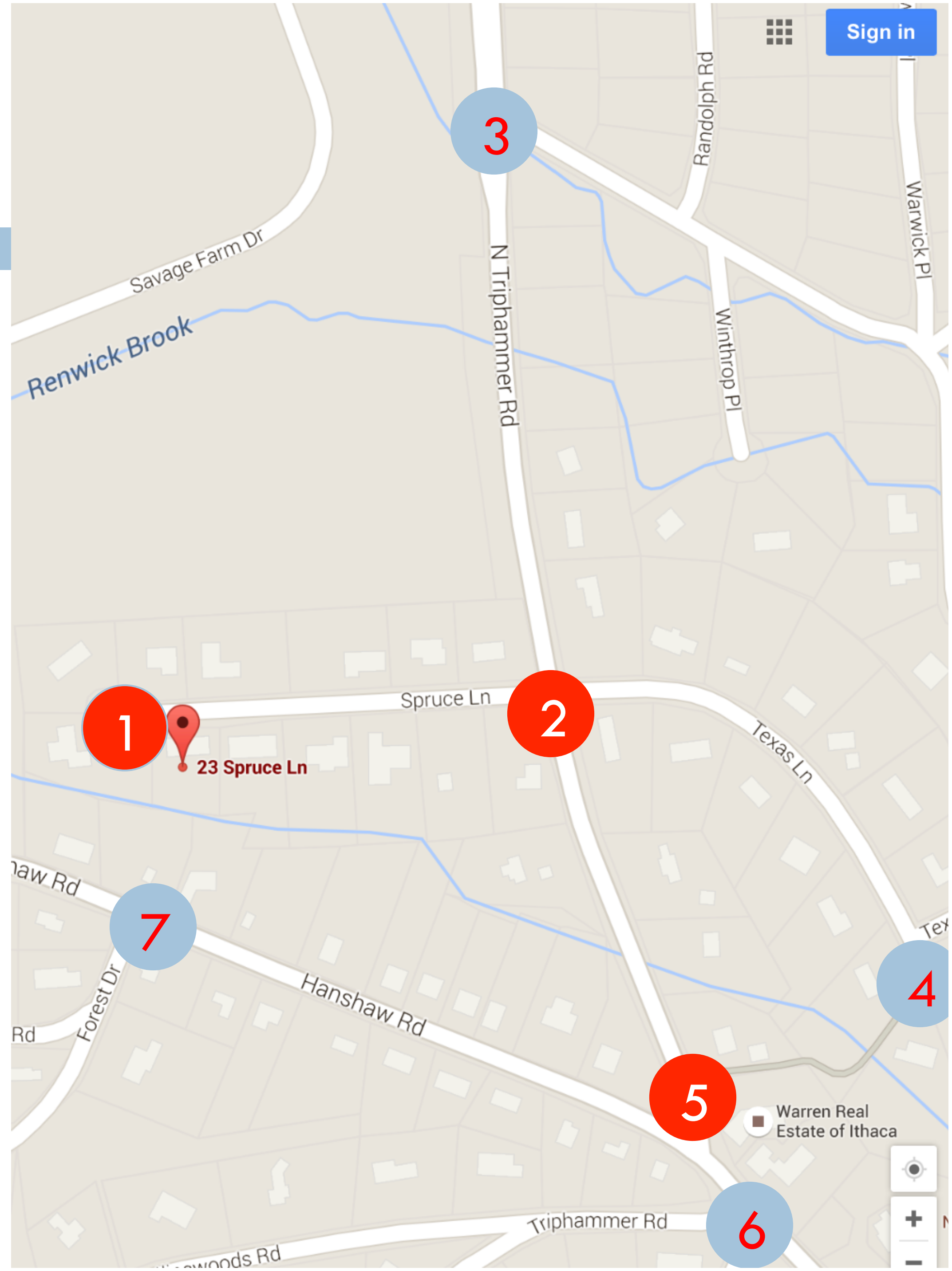
Shortest path?

Fan out from start, recording shortest distance seen so far

Settled set:  1 2 5

Frontier set: 7 3 4 6

1. Add neighbors of new Settled node to Frontier



Dijkstra's shortest path algorithm

The n (> 0) nodes of a graph numbered 0..n-1.

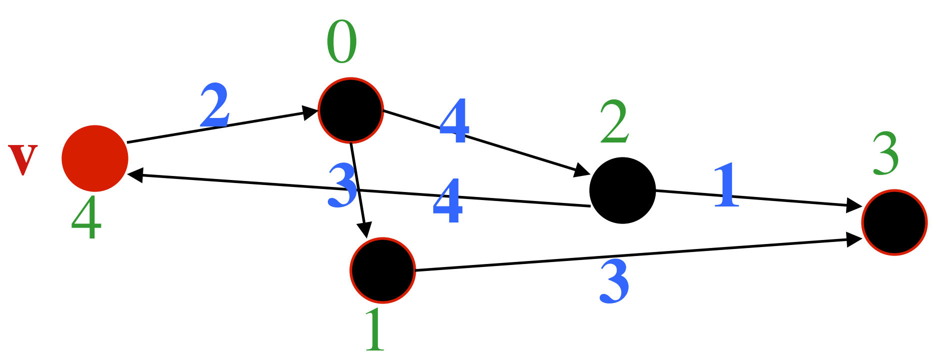
Each edge has a positive weight.

wgt(v1, v2) is the weight of the edge from node v1 to v2.

Some node v be selected as the *start* node.

Calculate length of shortest path from v to each node.

Use an array L[0..n-1]: for **each** node w, store in L[w] the length of the shortest path from v to w.



L[0] = 2
L[1] = 5
L[2] = 6
L[3] = 7
L[4] = 0

Dijkstra's shortest path algorithm

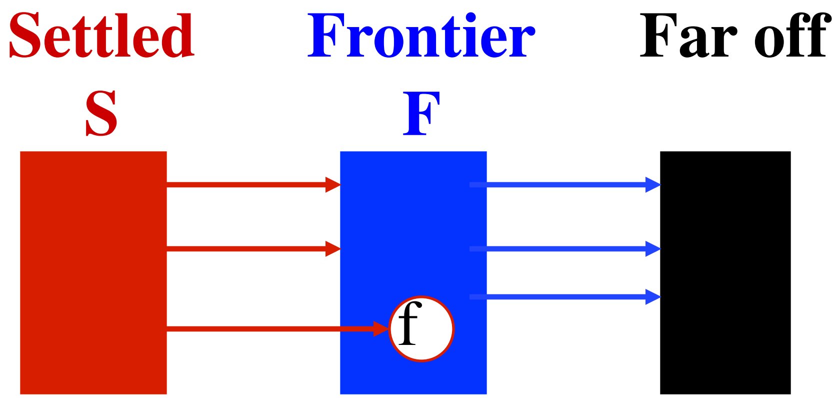
Develop algorithm, not just present it.

Need to show you the state of affairs —the relation among all variables— just before each node i is given its final value L[i].

This relation among the variables is an *invariant*, because it is always true.

Each node i (except the first) is given its final value L[i] during an iteration of a loop, so the *invariant* is called a *loop invariant*.

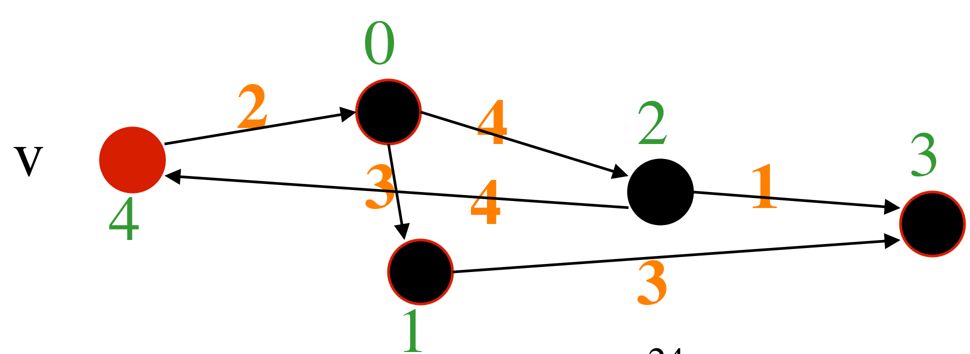
L[0] = 2
L[1] = 5
L[2] = 6
L[3] = 7
L[4] = 0

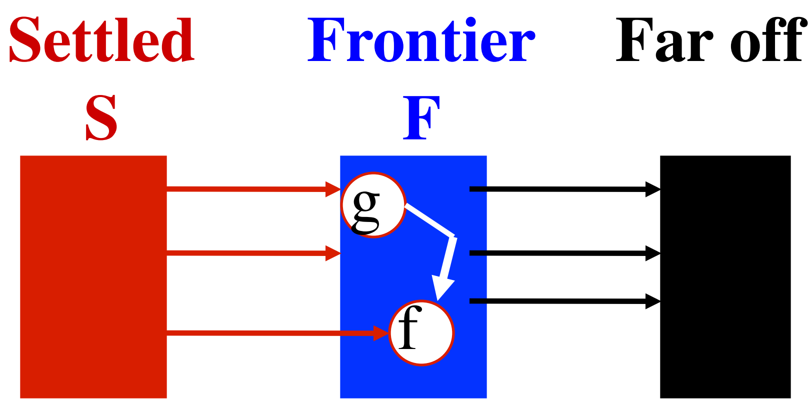


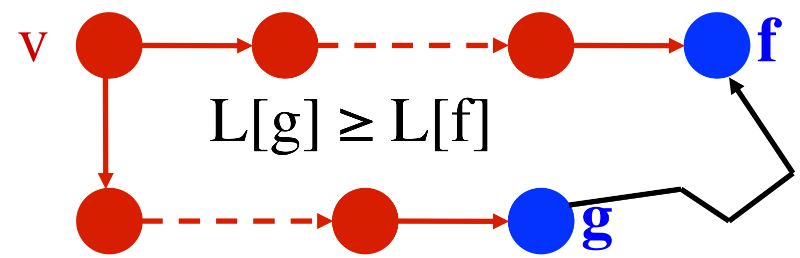
The loop invariant

(edges leaving the Far off set and edges from the Frontier to the Settled set are not shown)

1. **For a Settled node s, L[s]** is length of shortest v → s path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f] is length of shortest v → f path using only red nodes (except for f)**







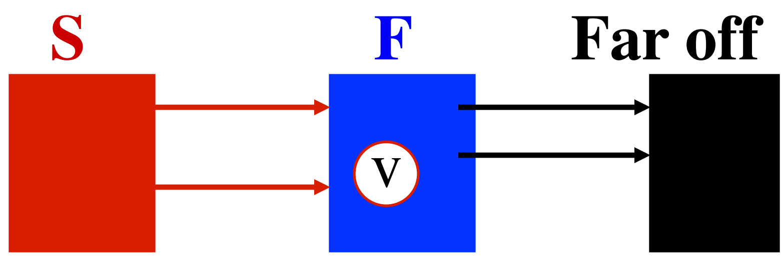
1. **For a Settled node s, L[s]** is length of shortest v → r path.
2. **All edges leaving S go to F.**
3. **For a Frontier node f, L[f]** is length of shortest v → f path using only Settled nodes (except for f).

Theorem. For a node **f** in **F** with minimum L value (over nodes in **F**), **L[f]** is the length of a shortest path from **v** to **f**.

Case 1: **v** is in **S**.

Case 2: **v** is in **F**. Note that L[v] is 0; it has minimum L value

The algorithm



1. **For s**, **L[s]** is length of shortest v→ s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

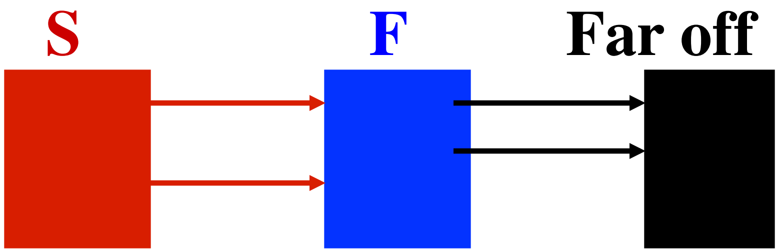
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

S= { }; F= { v }; L[v]= 0;

Loopy question 1:

How does the loop start? What is done to truthify the invariant?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

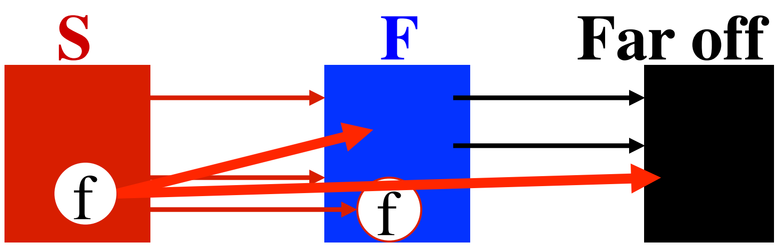
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

S= { }; F= { v }; L[v]= 0;
while F ≠ {} {

}

Loopy question 2:
When does loop stop? When is array L completely calculated?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F.**
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

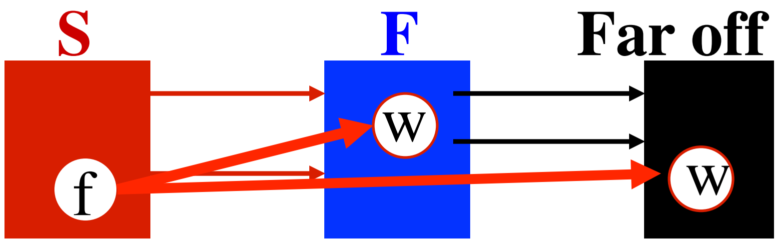
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

S= { }; F= { v }; L[v]= 0;
while F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;

}

Loopy question 3: Progress toward termination?

The algorithm



1. **For s**, **L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

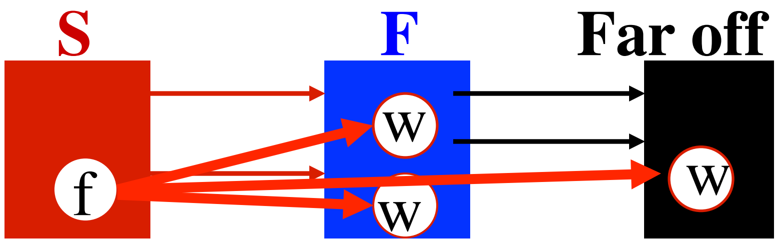
```
S= { }; F= { v }; L[v]= 0;
while  F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f, w) {
        if (w not in S or F) {


        } else {


        }
    }
}
```

Loopy question 4: Maintain invariant?

The algorithm



1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F**.
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

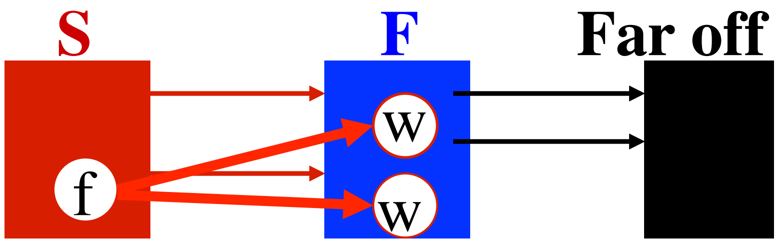
Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
S= { }; F= { v }; L[v]= 0;
while  F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f, w) {
        if (w not in S or F) {
            L[w]= L[f] + wgt(f, w);
            add w to F;
        } else {


        }
    }
}
```

Loopy question 4: Maintain invariant?

The algorithm



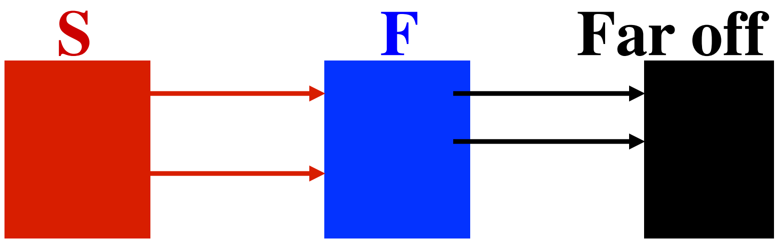
1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F.**
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
S= { }; F= { v }; L[v]= 0;
while  F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f, w) {
        if (w not in S or F) {
            L[w]= L[f] + wgt(f, w);
            add w to F;
        } else {
            if (L[f] + wgt (f,w) < L[w])
                L[w]= L[f] + wgt(f, w);
        }
    }
}
```

Loopy question 4: Maintain invariant?

The algorithm

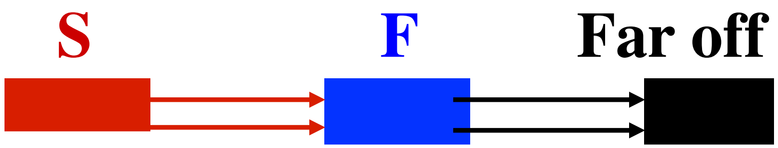


1. **For s, L[s]** is length of shortest v → s path.
2. **Edges leaving S go to F.**
3. **For f, L[f]** is length of shortest v → f path using red nodes (except for f).

Theorem: For a node **f** in **F** with min L value, L[f] is shortest path length

```
S= { }; F= { v }; L[v]= 0;
while   F ≠ {} {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f, w) {
        if (w not in S or F) {
            L[w]= L[f] + wgt(f, w);
            add w to F;
        } else {
            if (L[f] + wgt (f,w) < L[w])
                L[w]= L[f] + wgt(f, w);
        }
    }
}
```

Algorithm is finished!




```
S=  { }; F=  { v }; L[v]= 0;
while   F ≠  {}    {
    f= node in F with min L value;
    Remove f from F, add it to S;
    for each edge (f, w) {
        if (w not in S or F) {
            L[w]=  L[f] + wgt(f, w);
            add w to F;
    } else {
        if (L[f] + wgt (f,w) < L[w])
            L[w]= L[f] + wgt(f, w);
    }
}}
```

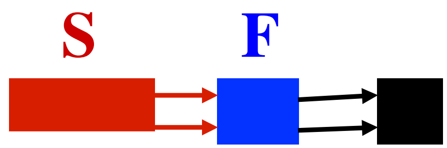
Implement F using a min-heap, priorities are L-values

Need L-values of nodes in S

Need to tell quickly whether a node is in S or F

```
class SFInfo {
    // this node's L-value
    int distance;
}  more fields later
```

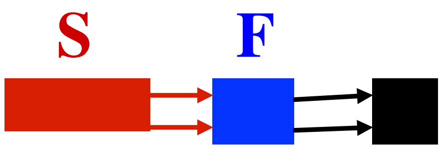
```
// entries for nodes in S or F
HashMap<Node, SFInfo>
                          map;
```



~~S= {};~~ F= { v }; L[v]= 0; add v to map
while F ≠ {} {
f= node in F with min L value;
Remove f from F, ~~add it to S~~;
for each edge (f, w) {
if (w not in ~~F or S~~ map) {
L[w]= L[f] + wgt(f, w);
add w to F; add w to map
} **else** {
if (L[f] + wgt (f,w) < L[w])
L[w]= L[f] + wgt(f, w);
}
}}

class SFInfo {
// this node's L-value
int distance;
} *more fields later*

// entries for nodes in S or F
HashMap<Node, SFInfo>
map;

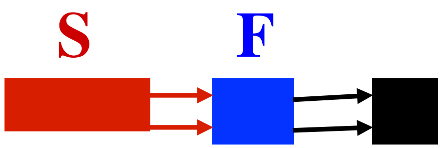


Final algorithm

```
F= { v }; L[v]= 0; add v to map
while  F ≠ {}   {
   f= node in F with min L value;
   Remove f from F;
   for each edge (f, w) {
      if (w not in map) {
          L[w]=  L[f] + wgt(f, w);
          add w to F; add w to map;
   } else {
      if (L[f] + wgt (f,w) < L[w])
          L[w]= L[f] + wgt(f, w);
   }
}}
```

```
class SFInfo {
   // this node's L-value
   int distance;
}  more fields later
```

```
// entries for nodes in S or F
HashMap<Node, SFInfo>
                        map;
```



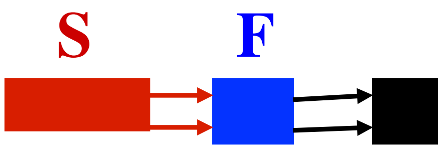
n nodes, reachable from v. $e \geq n-1$ edges.
$n-1 \leq e \leq n*n$

```
F= { v }; L[v]= 0; add v to map
while   F ≠ {}   {
   f= node in F with min L value;
   Remove f from F;
   for each edge (f, w) {
      if (w not in map) {
         L[w]=  L[f] + wgt(f, w);
         add w to F; add w to map;
   } else {
      if (L[f] + wgt (f, w) < L[w])
         L[w]= L[f] + wgt(f, w);
   }
}}
```

For each statement, calculate the average TOTAL time it takes to execute it.

Examples:
F ≠ {} is evaluated n+1 times. O(n)

w not in map is evaluated e times (once for each edge).
It's true n-1 times
It's false e – (n-1) times



n nodes, reachable from v. $e \geq n-1$ edges.
$n-1 \leq e \leq n*n$

```
F= { v }; L[v]= 0; add v to map                  O(1)
while  F ≠ {}   {                                O(n)
   f= node in F with min L value;                O(n)
   Remove f from F;                              O(n log n)
   for each edge (f, w) {                        O(n + e)
      if (w not in map) {                        O(e)
         L[w]= L[f] + wgt(f, w);                 O(n)
         add w to F; add w to map;               O(n log n)
      } else {
         if (L[f] + wgt (f, w) < L[w])
            L[w]= L[f] + wgt(f, w);              O((e-n) log n)
      }
}}
```

true n-1 times
false e-(n-1)

outer loop:
n iterations.
Condition evaluated n+1 times.

inner loop:
e iterations.
Condition evaluated n + e times.

Complete graph: $O(n^2 \log n)$. Sparse graph: $O(n \log n)$