CSCI 136: Data Structures and Advanced Programming

Lecture 34

Dijkstra's Algorithm

Instructor: Kelly Shaw

Williams

Topics

Priority Queues

Dijkstra's algorithm

Your to-dos

1. **Review readings** from *Bailey*.
2. **Study** for the final exam.
   a. Pro tip: **review quizzes**.
   b. **Do problems** in study guide/practice exam.
   c. **Don't stress out!** Just be methodical and do your best.
3. **Work on resubmissions** you plan to submit.

Announcements

Amy Babay, University of Pittsburgh

Friday, Dec 9 @ 2:35pm (last colloquium of 2022!)
Computer Science Colloquium – Wege TCL 123
Toward Intrusion-Tolerant Critical Infrastructure

As critical infrastructure systems are becoming increasingly exposed to malicious attacks, it is crucial to ensure that they can withstand sophisticated attacks while continuing to operate correctly and at their expected level of performance.

In this talk, I will present our work on making intrusion-tolerant critical infrastructure systems possible and practical. I will start by discussing our Spire system, the first Supervisory Control and Data Acquisition (SCADA) system for the power grid that is resilient to both system-level compromises and sophisticated network-level attacks.

Then, I will present our recent work offering a practical deployment path for Spire and similar BFT-based systems through a new model for "intrusion tolerance as a service". The intrusion-tolerance-as-a-service model enables critical infrastructure operators to gain the resilience benefits of intrusion tolerance, while offloading significant parts of the system management to a service provider. Critically for practical acceptance, our work shows how these benefits can be achieved without requiring critical infrastructure operators to expose confidential or proprietary data and algorithms to the service provider.

Recall: with a heap, we can implement a priority queue.

Lots of interesting variants on heaps!

Summary of running times [edit]

In the following time complexities[5] $O(f)$ is an asymptotic upper bound and $\Theta(f)$ is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	find-min	delete-min	insert	decrease-key	merge
Binary[5]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[5]	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$[a]	$\Theta(\log n)$	$O(\log n)$[b]
Fibonacci[5][6]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Pairing[7]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$o(\log n)$[a][c]	$\Theta(1)$
Brodal[10][d]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[12]	$\Theta(1)$	$O(\log n)$[a]	$\Theta(1)$	$\Theta(1)$[a]	$\Theta(1)$
Strict Fibonacci[13]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap	?	$O(\log n)$[a]	$O(\log n)$[a]	$\Theta(1)$	?

a. ^ a b c d e f g h i Amortized time.

b. ^ n is the size of the larger heap.

c. ^ Lower bound of $\Omega(\log \log n)$,[8] upper bound of $O(2^{2\sqrt{\log \log n}})$.[9]

d. ^ Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with n elements can be constructed bottom-up in $O(n)$.[11]

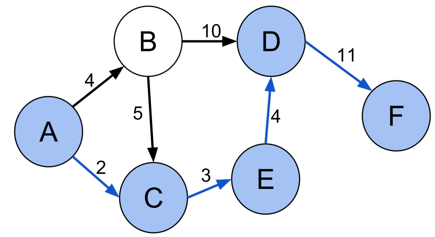
From **Wikipedia**: **priority queue** page.

Recall the example from our first class

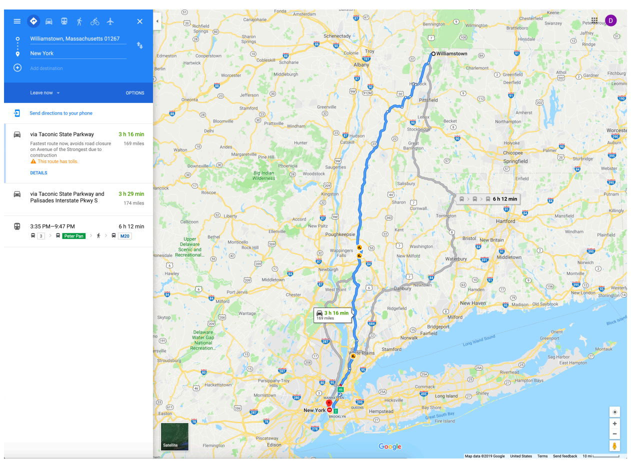
Graphs: shortest paths

Shortest path problem

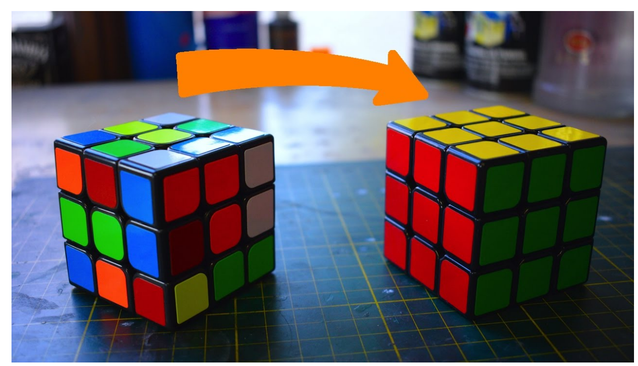
The **shortest path problem** is the problem of finding a **path between two vertices** in a graph such that **the sum** of the weights of its constituent edges **is minimized**.



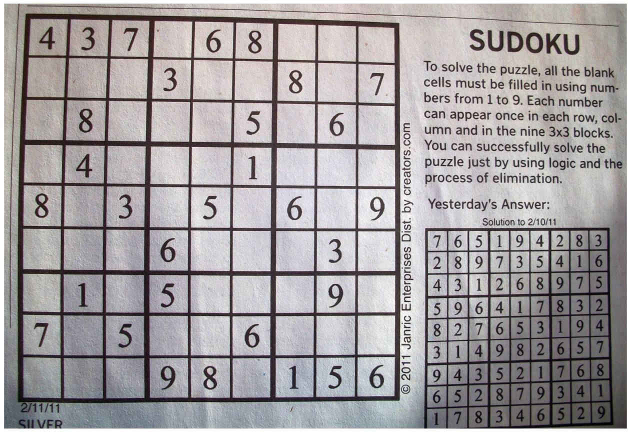
Applications



Applications

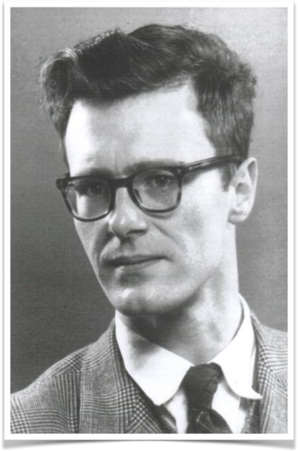


Applications



Applications

Dijkstra's algorithm



- Invented by Edsgar Dijkstra in 1959.
- The original version used a min-priority queue.
- Designed using pencil and paper; algorithm was intended to demonstrate to non-technical people how computers could be useful.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

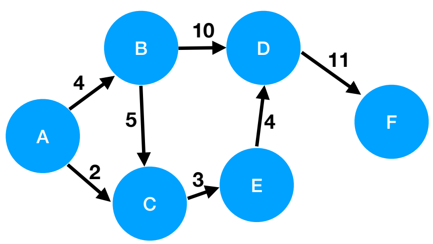
A	∞
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{A, B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

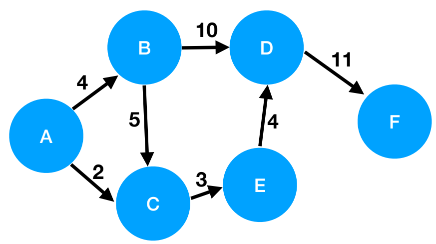
A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{A, B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

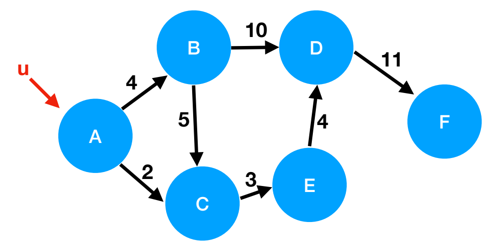
A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

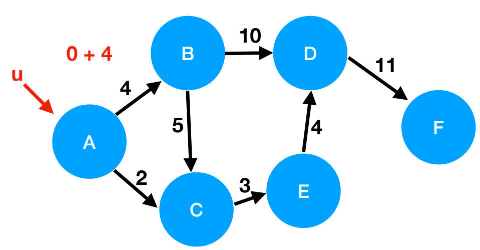
A	0
B	4
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	undef
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

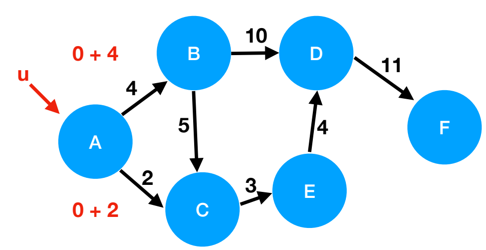
A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef

Q

{B, C, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

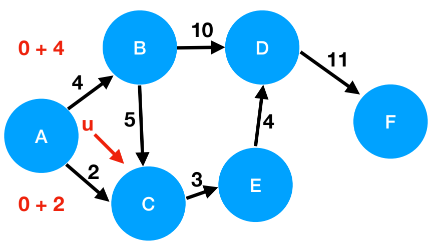
A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef

Q

{B, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

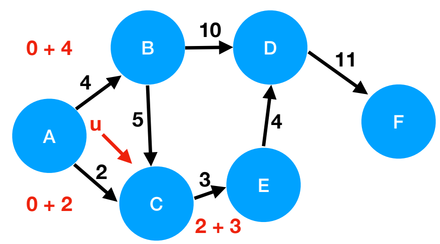
A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef

Q

{B, D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

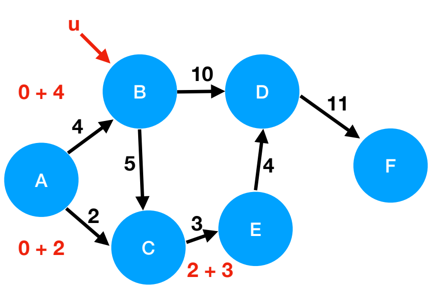
A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef

Q

{D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

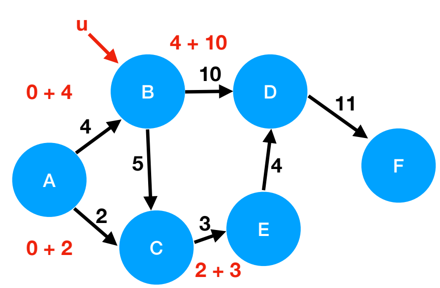
A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

Q

{D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

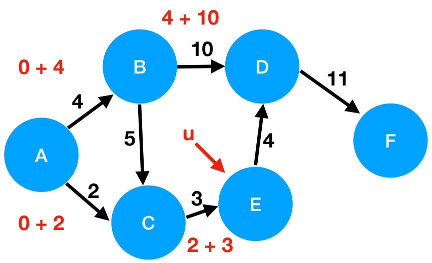
A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

Q

{D, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

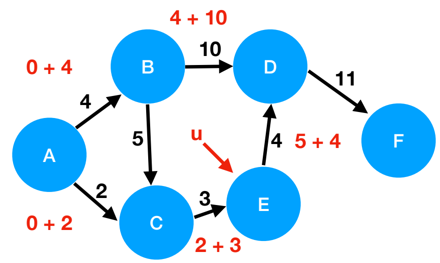
A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

Q

{D, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

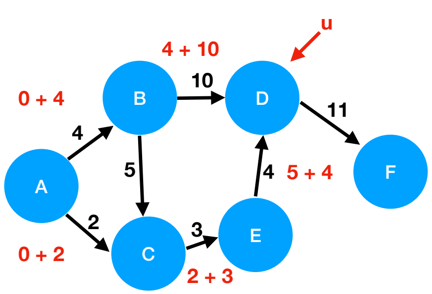
A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

Q

{F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

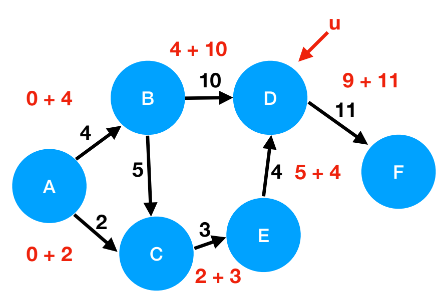
A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

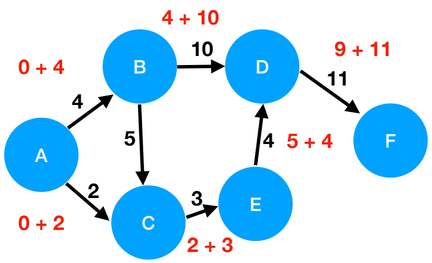
prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{}

Done!



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

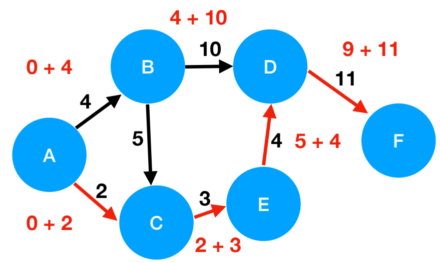
prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{}

Done!



Read prev backward from F and reverse.

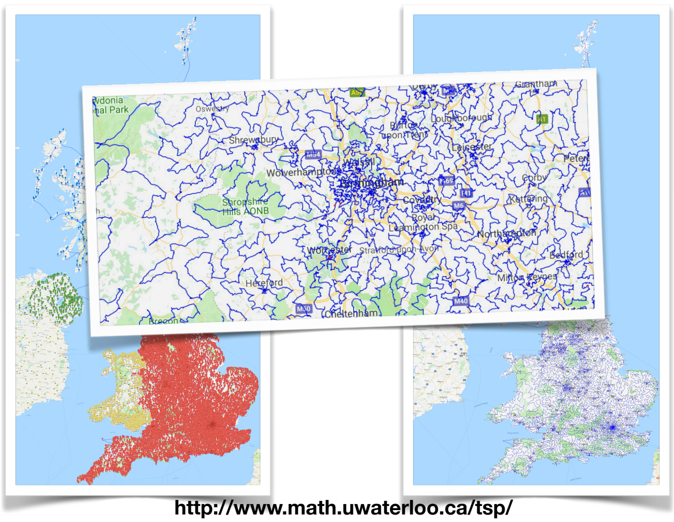
Graphs: traveling salesperson

Applications

Delivery routes.

Applications

Optimal 49,687-stop pub crawl



http://www.math.uwaterloo.ca/tsp/

Recap & Next Class

Today:

Dijkstra’s algorithm

Next class:

Final exam review