Path-Finding Algorithms in Practice

EN 500.111 Week 3

Outline

- Time complexity in theory
- Heuristics
- A-Star Search
- Hierarchical Pathfinding
- Contraction Hierarchies

Example: Recipe for chocolate chip cookies

Inputs

1 cup chocolate chips

1.5 cups flour

.75 cups sugar

etc.

Series of Steps

Outputs

2 dozen cookies

https://simple-veganista.com/vegan-chocolate-chip-cookies/

What if we double the recipe? Does it take longer? What about making 100 batches?

Considering individual steps

Adding ingredients
5 minutes to find and measure them

Shaping dough
10 seconds per cookie, so 2 minutes

Mixing
1 minute

Baking
10 minutes

Total = 18 minutes

Which of these steps take longer if we make twice as many?

Doubling the recipe

Adding ingredients
Still 5 minutes if we have big enough measuring cups and bowls

Shaping dough
10 seconds per cookie, so **4 minutes instead of 2 minutes** now

Mixing
Still 1 minute if one bowl fits everything

Baking
Still 10 minutes if we have 2 pans and a big oven

What about 100 batches (assuming we somehow have equipment that keeps the time required to measure, mix, and bake the same)?

100 batches

__Adding ingredients__
Still 5 minutes if we have enormous measuring cups and bowls

__Mixing__
Still 1 minute if one bowl fits everything

__Shaping dough__
10 seconds per cookie, so **200 minutes instead of 2 minutes** now

__Baking__
Still 10 minutes if we have 100 pans and a really really big oven

216 minutes for 100 batches vs. 18 minutes for 1 batch

Modelling the time required

Adding ingredients
Alway 5 minutes

Shaping dough
2 minutes times the number of batches

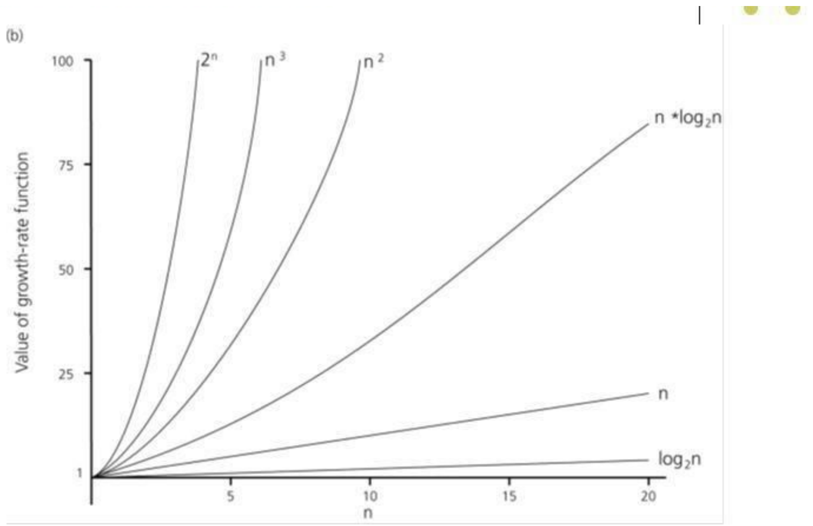
Mixing
Always 1 minute

Baking
Always 10 minutes

Total for n batches = 2n + 16 minutes

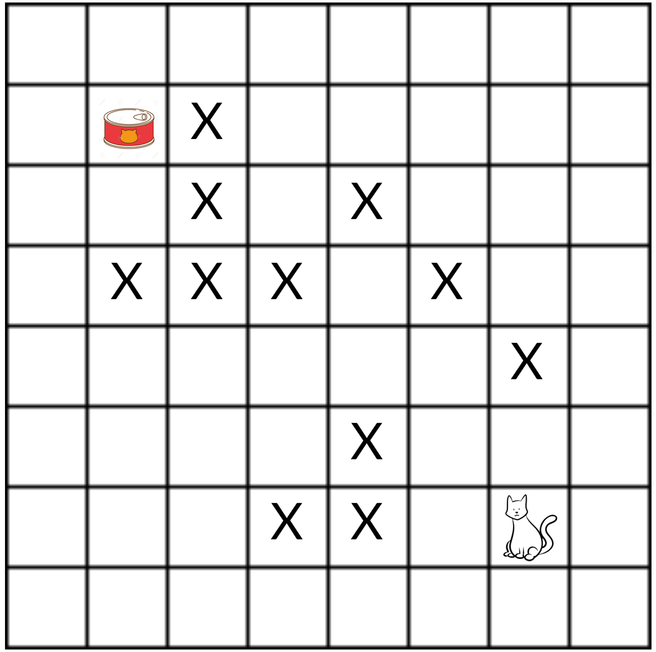
Modelling growth

- ➔ In runtime analysis, we want to understand how runtime grows as a function of the size of the input
- ➔ If we double the input size, does the algorithm take the same amount of time? Twice as long? 100 times as long?



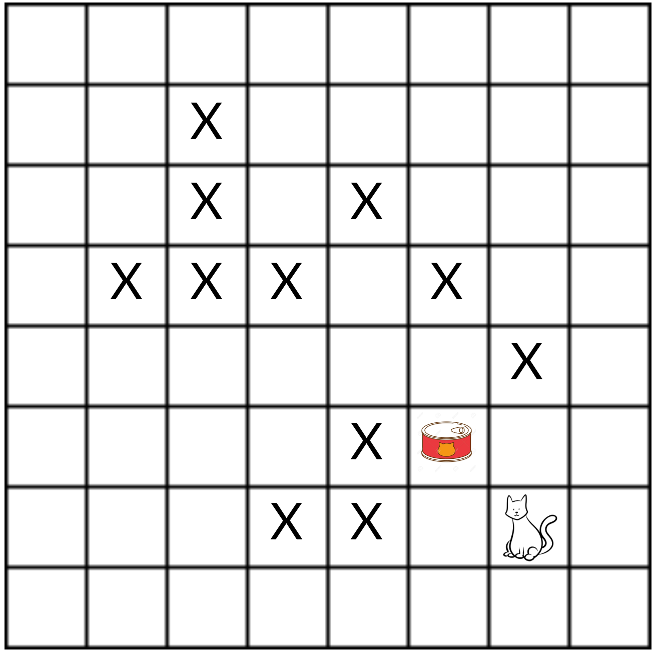
(https://slideplayer.com/slide/1473419/)

Time Complexity



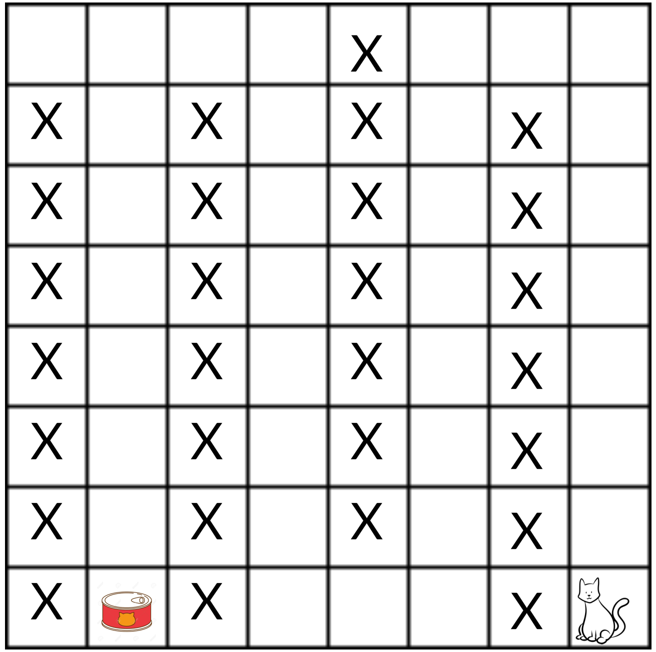
How long does Breadth-First Search take to find the shortest path from start to end path on an 8x8 grid?

Time Complexity



What about now?

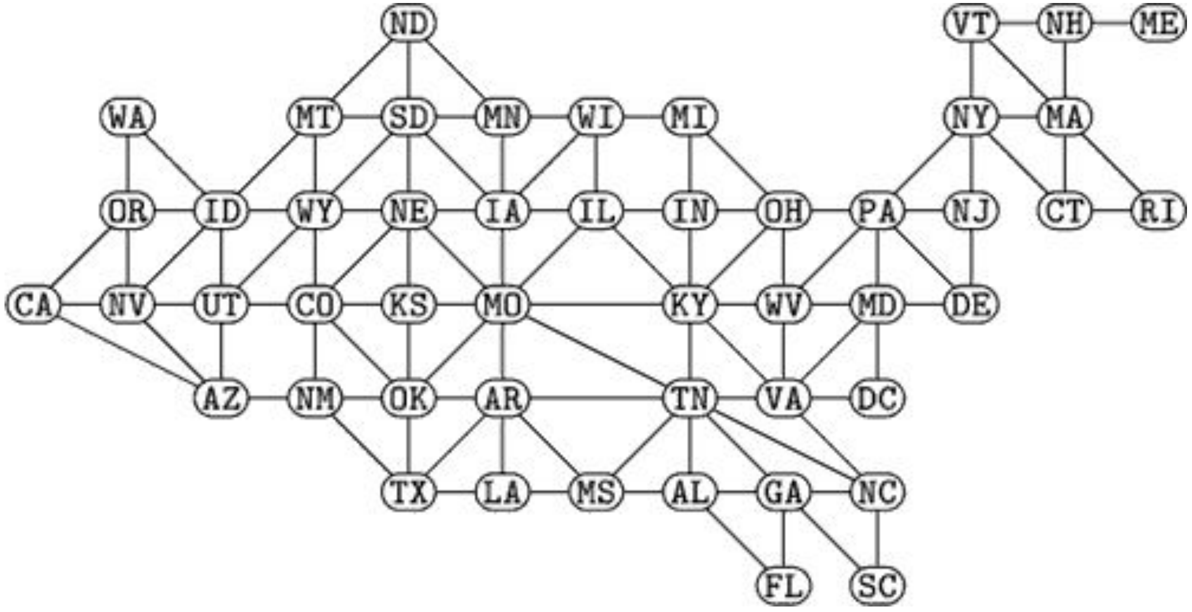
Time Complexity



What about now?

What about on general graphs?

- ➔ BFS can very quickly find a path from Maryland to Pennsylvania
- ➔ It takes a lot longer to find a path from Florida to Washington, even on the same graph



Worst-Case Analysis and Big-O Notation

- **Big-O Notation**: a way to compare how algorithms increase in runtime as the input gets larger
  - Assumes the most difficult possible graph/list/etc. of different sizes
  - Ignores constant factors because they are “irrelevant” for large enough inputs or hardware-specific
- A runtime function f(n) is called **O(n)** if there’s some constant c such that $f(n) \leq c*n$ for big enough n
- Similar definitions can be made for any functions of n such as n^2

Big-O runtime for cookie baking

Adding ingredients
Alway 5 minutes

Shaping dough
2 minutes times the number of batches

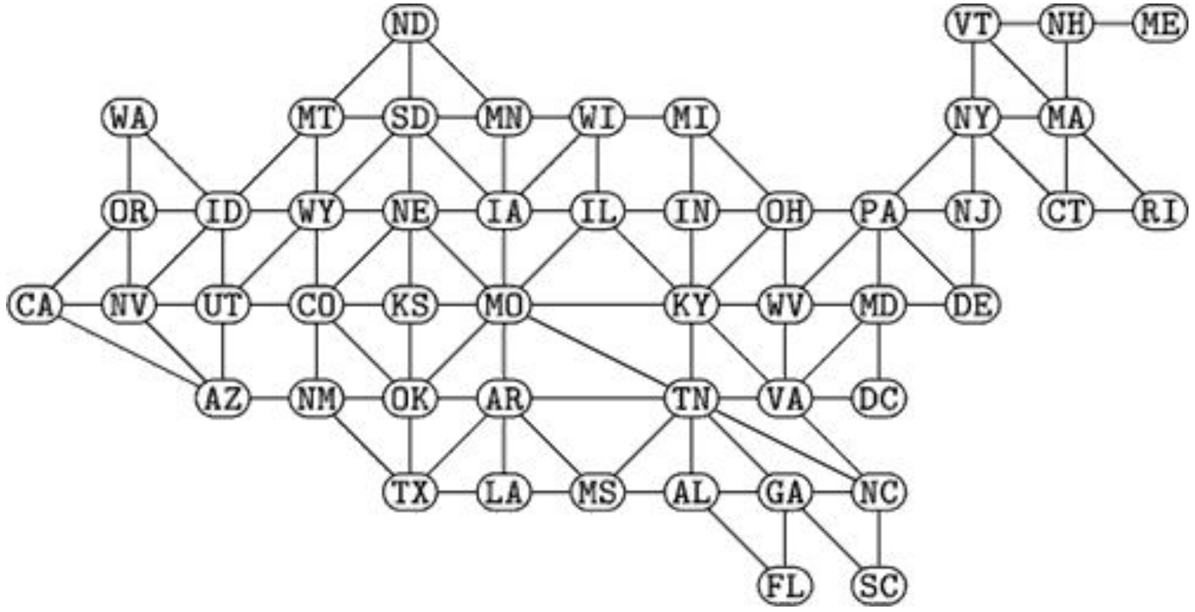
Mixing
Always 1 minute

Baking
Always 10 minutes

Total for n batches = 2n + 16 minutes
This is bounded by 3n for large n (n > 15), so the time is O(n)

What about BFS?

- What’s the worst case on a graph with n nodes and m edges?

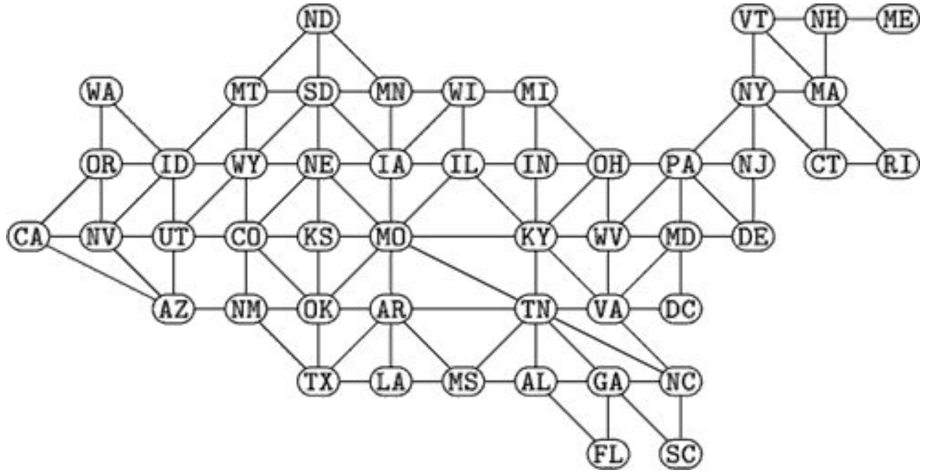


BFS is O(n + m)

The worst case is having to explore every single node and edge and the very last one gets us to the endpoint.

Relies on ability to quickly look up all of the neighbors of a given node

Dijkstra’s algorithm is slightly slower:
O((n + m) * log(n)) because of an extra data structure needed to keep track of the next-nearest node to process



Big-O Review

Which of these functions are in the same Big-O class of functions as each other?

- $a(n) = n$
- $b(n) = n^2 + 5n - 6$
- $c(n) = n + 123$
- $d(n) = 100n - 10$
- $e(n) = n^2$
- $f(n) = n^{10}$
- $g(n) = n * (n + 1) / 2$

Runtime in practice

- ➔ **In theory**, you can't do any better than BFS because the worse case will always require you to look at everything in the graph at least once
- ➔ But **in practice**, we care more about things like constant factors and the average-case runtime
- ➔ Heuristic: "shortcuts" that help out with commonly encountered case; they don't usually improve the worst case and may even lead to slightly suboptimal paths

What are some heuristics you could use to add numbers faster?

14

7

3

12

10

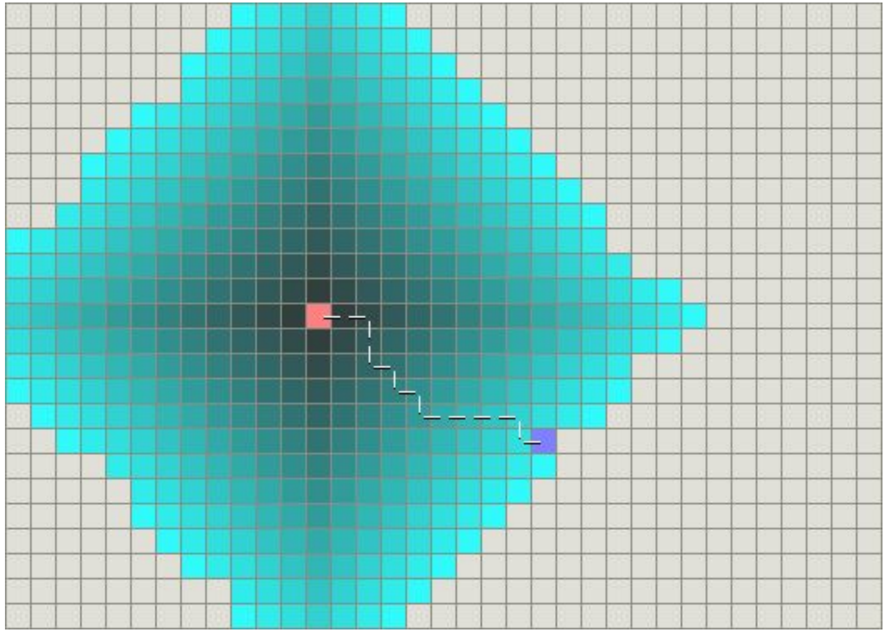
5

What about these numbers?

1	1	1	1	1	1	2	2	2	2	2	2
1	1	1	1	1	1	2	2	2	2	2	2
1	1	1	1	1	1	2	2	2	2	2	2
1	1	1	1	1	1	2	2	2	2	2	2
1	1	1	1	1	1	2	2	2	2	2	2

Improving BFS in practice

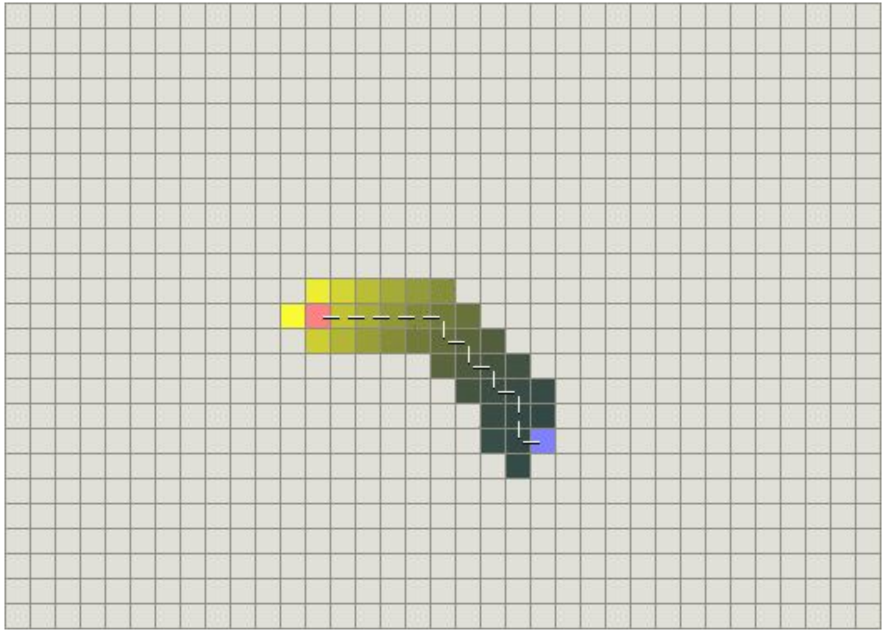
In this example, BFS does a lot of extra work to find a very simple path by looking at squares without considering if they'll help find a path to the goal



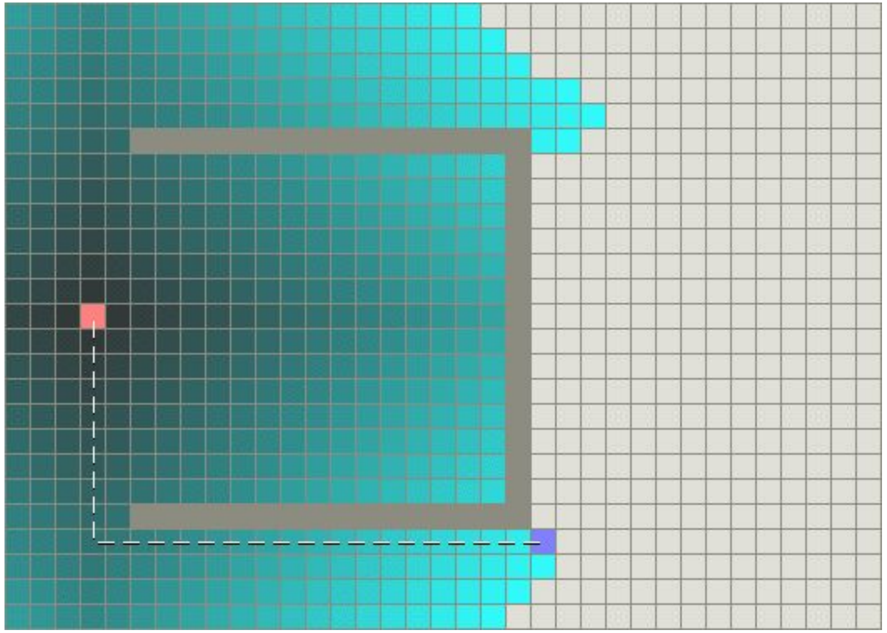
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

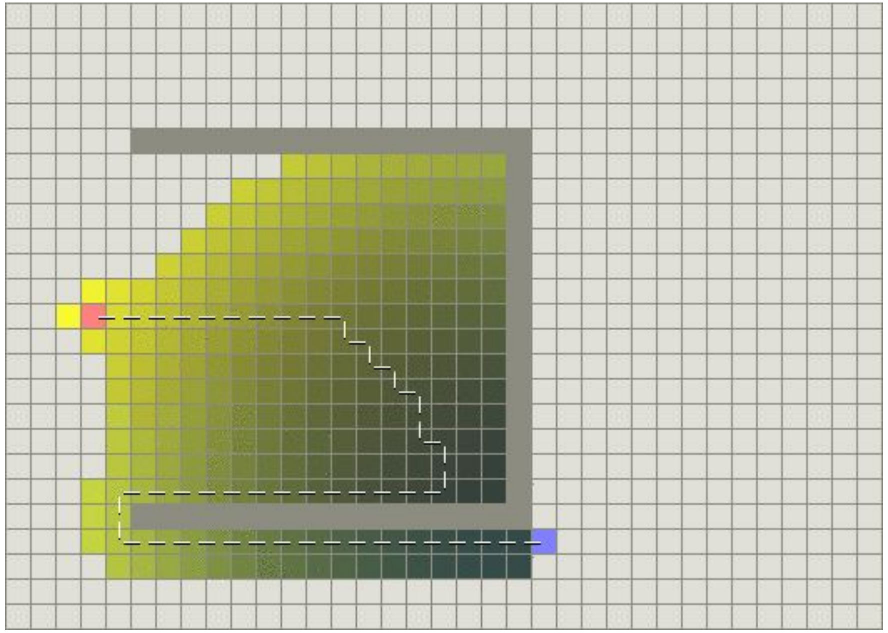
Best-First Search

What if we use an algorithm that preferentially explore nodes that are closer (as the crow flies) to the destination, and stops when it finds a path?

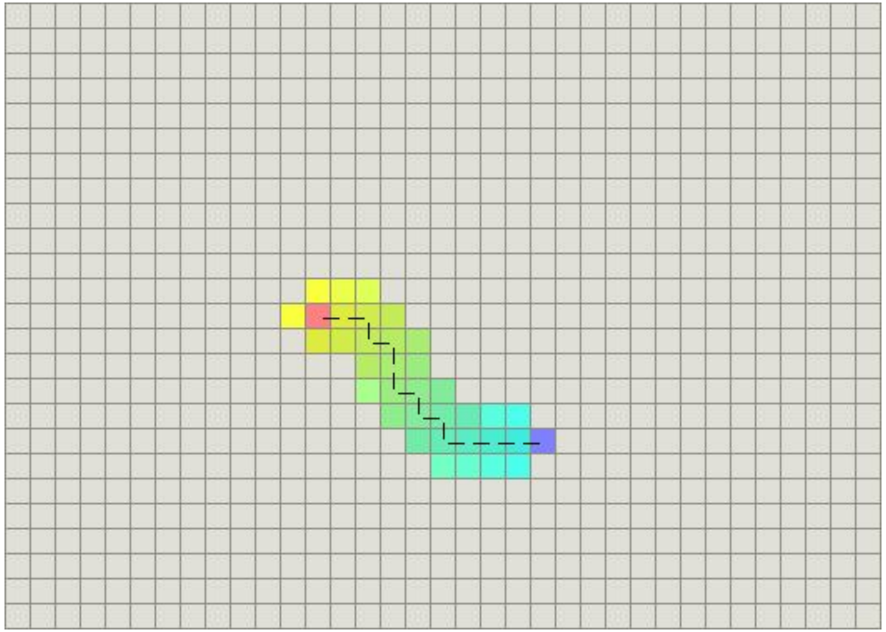


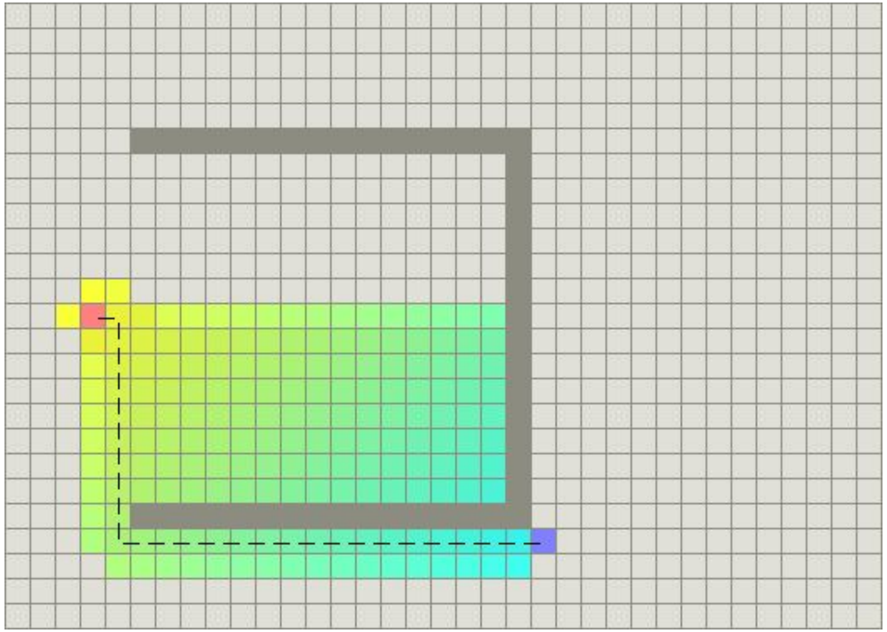
What about in harder cases?





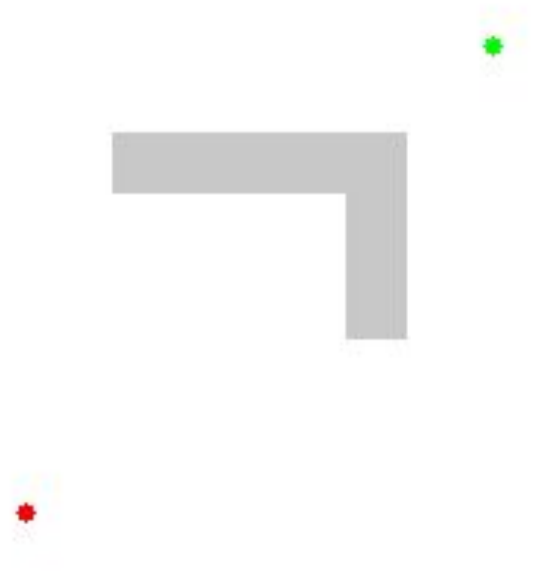
A^* - A hybrid shortest-path algorithm



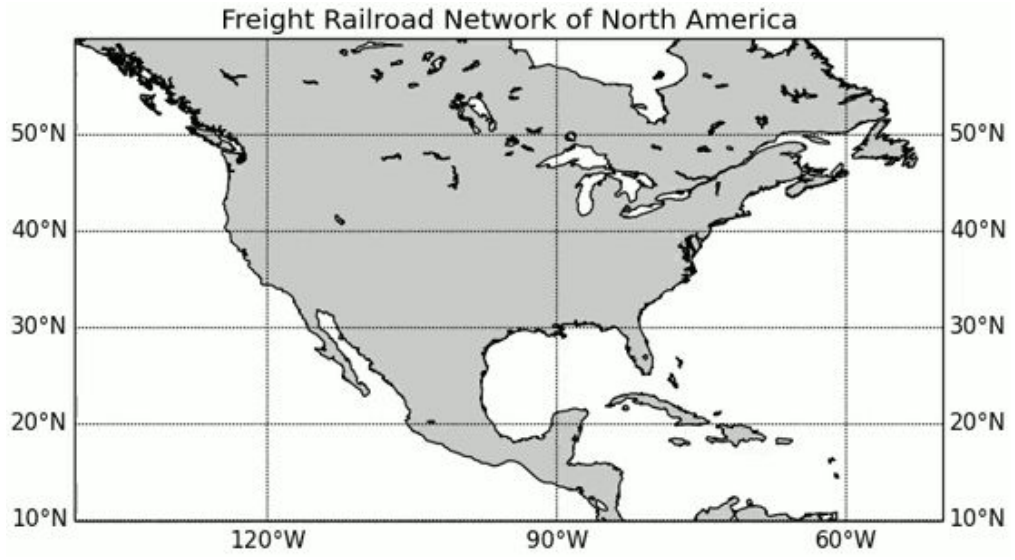


A* details

- Very similar to Dijkstra's algorithm, but each node has two values now:
  - Distance from start
  - An estimate of the distance to the end
- Nodes are processed in order of the sum of these two values
- Does more work than best-first but if the estimates never overestimate the true distance to the end, it's guaranteed to always be correct
- Choice of estimates very application-dependent


An example of A* getting around an obstacle

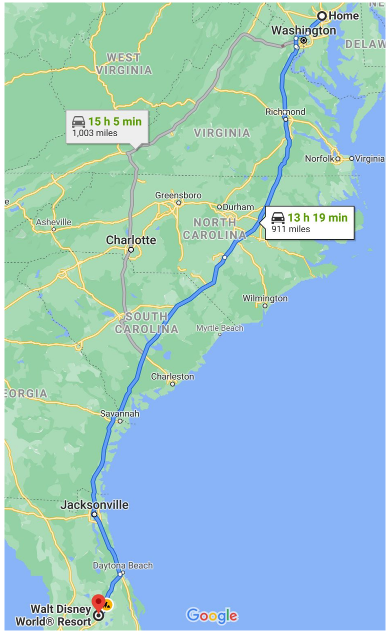
A* in practice



Finding a path from DC to Los Angeles via railroads (Wikipedia)

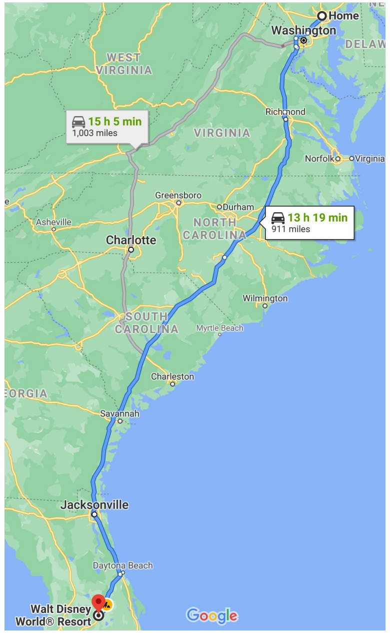
Further Improvements

- ➔ For long routes A* still has to explore a lot of nodes and edges, even with the heuristic directing its search
- ➔ What potentially useful properties would the fastest route from Baltimore to Disney World have that could help us narrow down the search more?



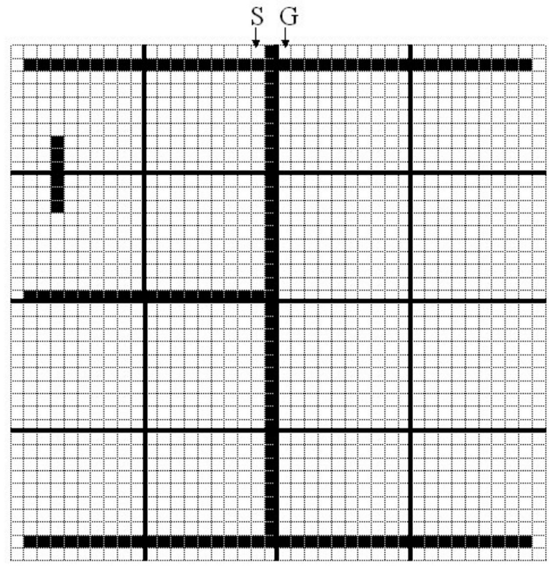
Using Highways

- ➔ A route likely has these three steps:
  1. Get from home to a highway going south
  2. Take a series of highways to Orlando
  3. Get from a highway to Disney World
- ➔ Solving these separately lets us ignore the side roads in the middle of the route that we would never want to use



Hierarchical Pathfinding

- ➔ Break grid up into subgrids
- ➔ Path is now decomposed into a few parts:
  - ◆ Get to the edge of S's subgrid
  - ◆ Travel through some intermediate subgrids to get to the subgrid with G
  - ◆ Travel within G's subgrid to get to G

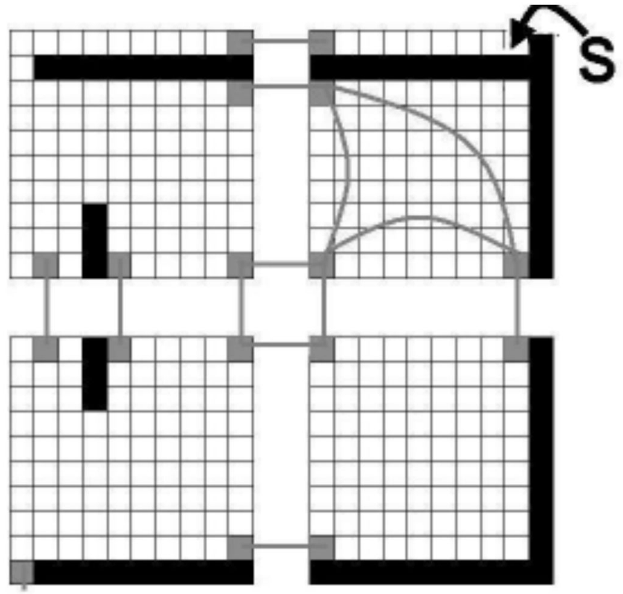


(a) The 40 × 40 maze used in our example. The obstacles are painted in black. S and G are the start and the goal nodes. (b) The bold lines show the boundaries of the 10 × 10 clusters.

(Near optimal hierarchical path-finding, Botea et. al., 2004)

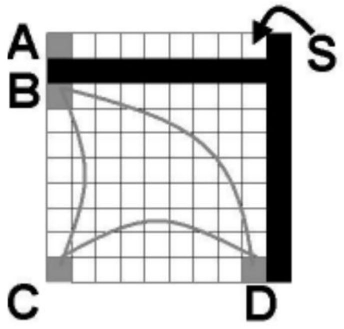
Precomputing intermediate subgrids

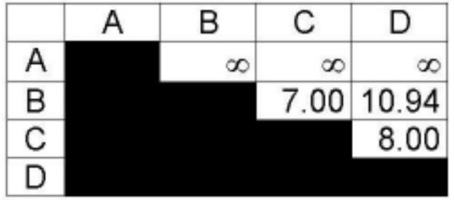
- ➔ For intermediate subgrids, there are a few key nodes which can together capture all of the paths which use the subgrid as an intermediate
- ➔ In this paper, they chose these key nodes by looking at each the width of each contiguous “entrance” and adding one or two nodes based on the entrance width



Precomputing intermediate subgrids

- Shortest path between each pair of entrances in a subgrid, forming a compressed graph to get through the intermediate subgrids

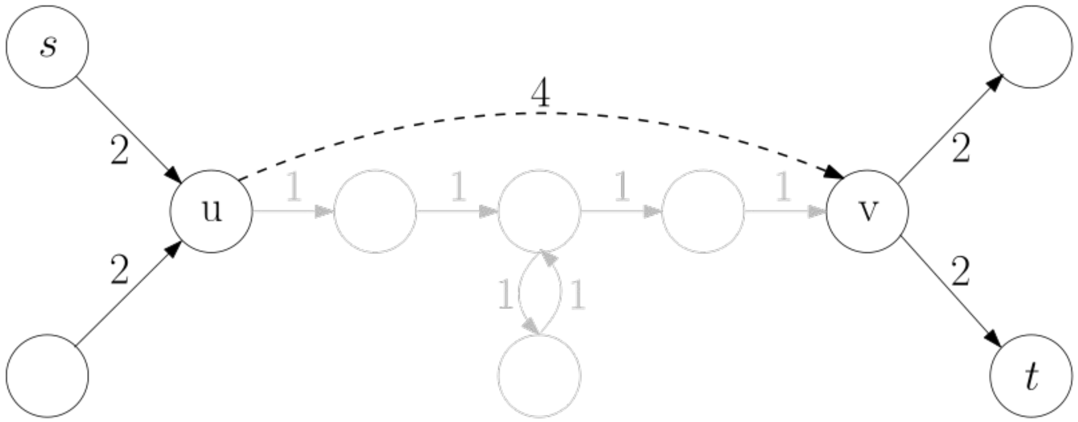




	A	B	C	D
A	⬛	∞	∞	∞
B	⬛	⬛	7.00	10.94
C	⬛	⬛	⬛	8.00
D	⬛	⬛	⬛	⬛

Contraction Hierarchies

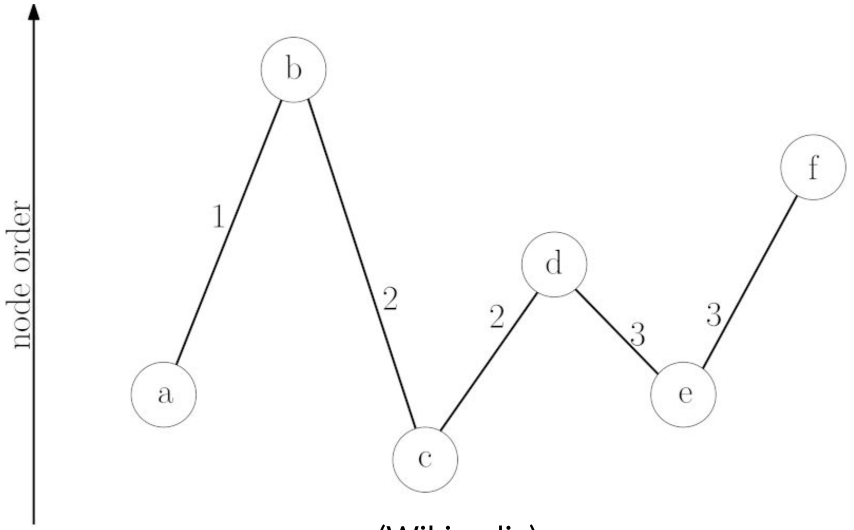
Extends the idea of hierarchical pathfinding to general graphs by adding in “short-cuts” as a pre-processing step



(Wikipedia)

Building a hierarchy of compressed edges

- ➔ Vertices are “contracted” one at a time
- ➔ When contracting a vertex v, it’s temporarily removed from the graph and edges are added between pairs of its neighbors if all shortest paths between them go through v
- ➔ Number of shortcuts depends on the order in which vertices are contracted
- ➔ In practice on large geographical maps it about doubles the size of the graph

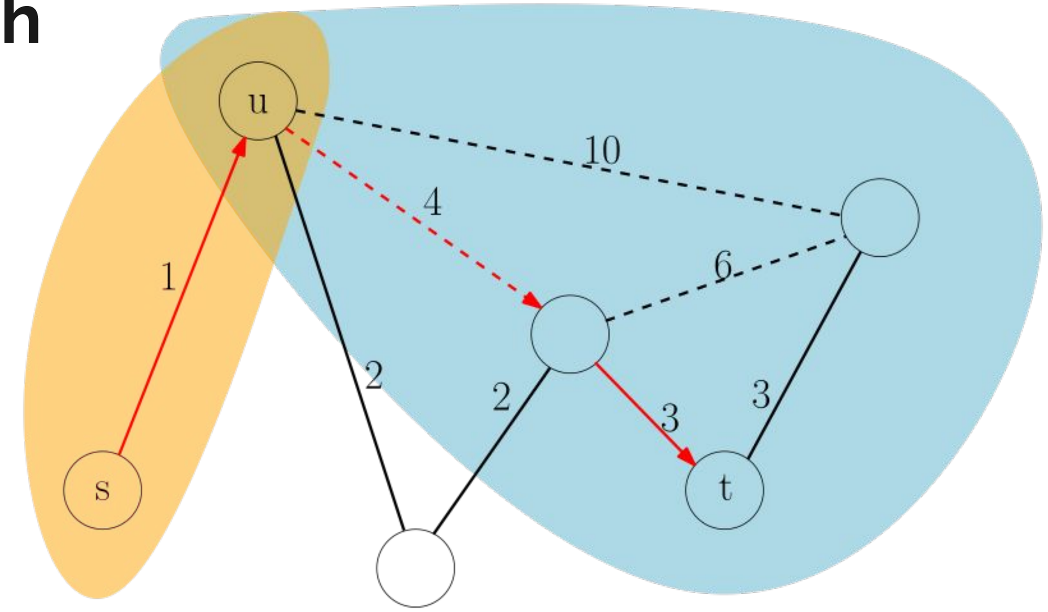


(Wikipedia)

Bidirectional search

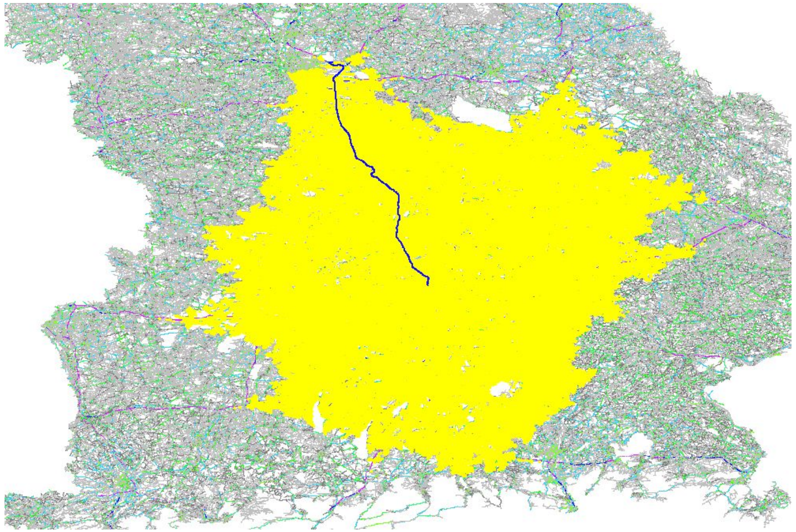
Next step is to search from s and t, always going farther up the hierarchy

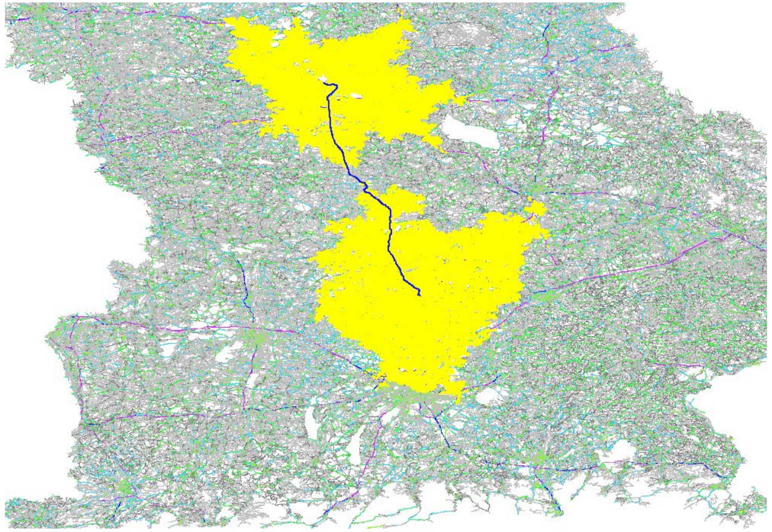
Run Dijkstra's from both nodes at the same time until a node has been hit by both of them



(Wikipedia)

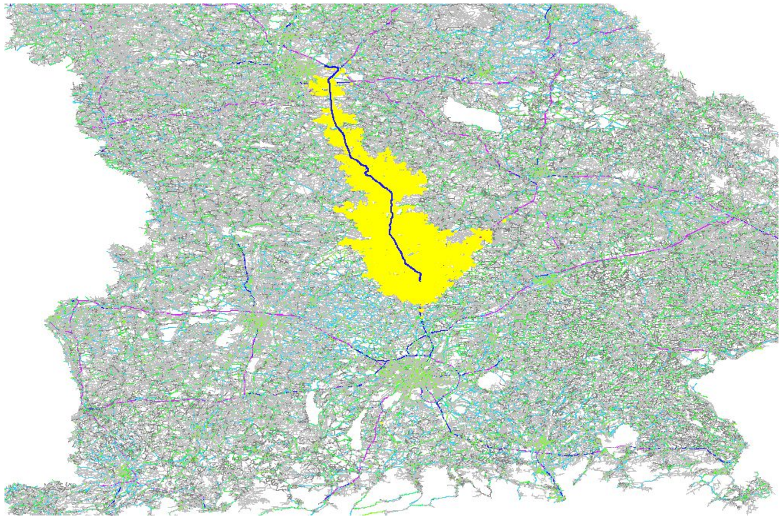
Comparison of methods in practice


Dijkstra’s algorithm (~500k nodes)

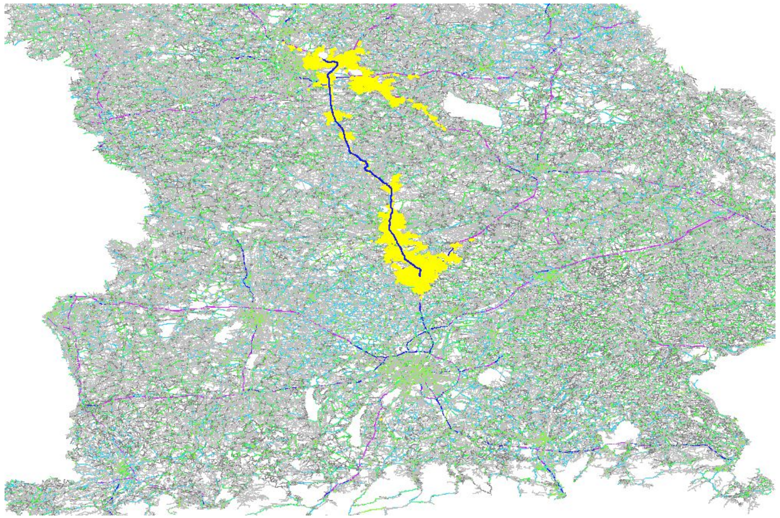

Bidirectional Dijkstra’s algorithm (~220k nodes)

(https://www.graphhopper.com/blog/2017/08/14/flexible-routing-15-times-faster/)

Comparison of methods in practice

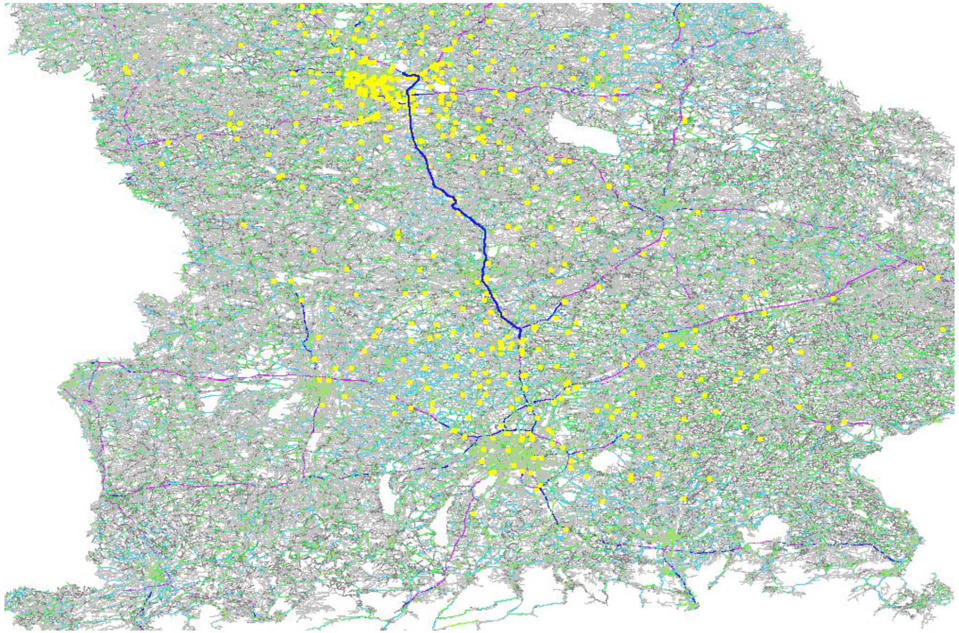


A* algorithm (~50k nodes)



Bidirectional A* algorithm (~25k nodes)

Comparison of methods in practice



Contraction hierarchies (~600 nodes)

Any questions?

- ➔ For a more advanced look at navigation algorithms, I recommend the online lectures/resources from Professor Hannah Bast’s (University of Freiburg) 2011 course titled “Efficient Route Finding”: https://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2011
- ➔ **Next week**: Social Network Graphs