# GETTING FROM A TO B:
## FAST ROUTE-FINDING USING SLOW COMPUTERS

Simon Peyton Jones (*giving the talk*)
Andrew Goldberg (who did all the work)

Microsoft Research

# www.computingatschool.org.uk

COMPUTING AT SCHOOL

EDUCATE · ENGAGE · ENCOURAGE

| HOME | THE CHALLENGE | JOINING CAS | DOCUMENTS | EVENTS | NEWS | ABOUT |

## Computing for the next generation ...

The "Computing At School" group (CAS) is a membership association run by BCS, The Chartered Institute for IT and supported by Microsoft Research and other industry partners. It was created to support and promote the teaching of computer science and other computing disciplines in UK schools. Our membership is broad, and includes teachers, examiners, parents, university faculty, and employers.

CAS was born out of our excitement with our discipline, combined with a serious concern that our brightest students are being turned off computing by a combination of factors that have conspired to make the subject seem dull and pedestrian. Our goal is to put the fun back into computing at school.

We see computing as a rich and deep discipline in its own right, like physics or mathematics. Like those subjects, computing explores foundational principles and ideas, rather than training

## Switched On


Keep up to date with all the latest news from CAS. Follow the links below to download a copy of the newsletter and also follow up the content with links to various external resources.

- Autumn 2009
- Summer 2010

# Please join in

The “Computing At School” group (CAS) is a membership association run by BCS, The Chartered Institute for IT and supported by Microsoft Research and other industry partners. It was created to support and promote the teaching of computer science and other computing disciplines in UK schools.  Our membership is broad, and includes teachers, examiners, parents, university faculty, and employers.

CAS was born out of our excitement with our discipline, combined with a serious concern that our brightest students are being turned off computing by a combination of factors that have conspired to make the subject seem dull and pedestrian. Our goal is to put the fun back into computing at school.

We see computing as a rich and deep discipline in its own right, like physics or mathematics. Like those subjects, computing explores foundational principles and ideas, rather than training

# Getting directions

# The work of a moment

# Finding shortest paths fast matters

- Well-known algorithms are fast enough on a reasonable computer but
  - A handheld is not a “reasonable computer”
  - Servers consume lots of energy (they are put next to hydro-electric power stations)
  - The “well known” algorithms are Terribly Wasteful
- The subject of this talk: can we do better?

# The story

My mother's method

Does not work

Labelling method

Works, slowly

Dijkstra

Dijkstra 1959

A bit faster still

Bidirectional Dijkstra

Hart, Nilsson, Raphael 1968

A* with Euclidean bounds

Still no cigar

Goldberg, Harrelson 2005

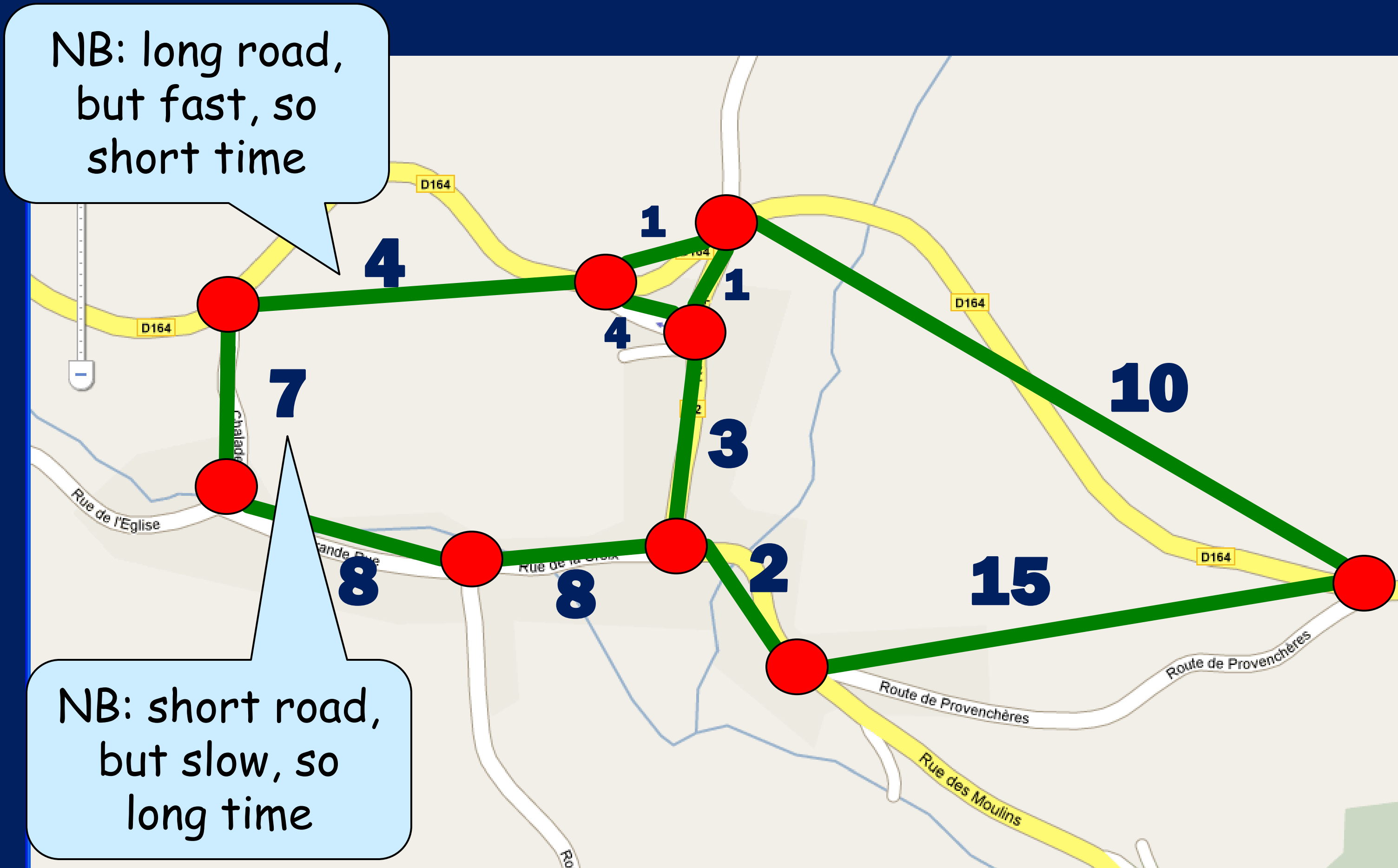
A* with landmarks

Goldberg, Kaplan, Werneck 2006

A* with landmarks and reach

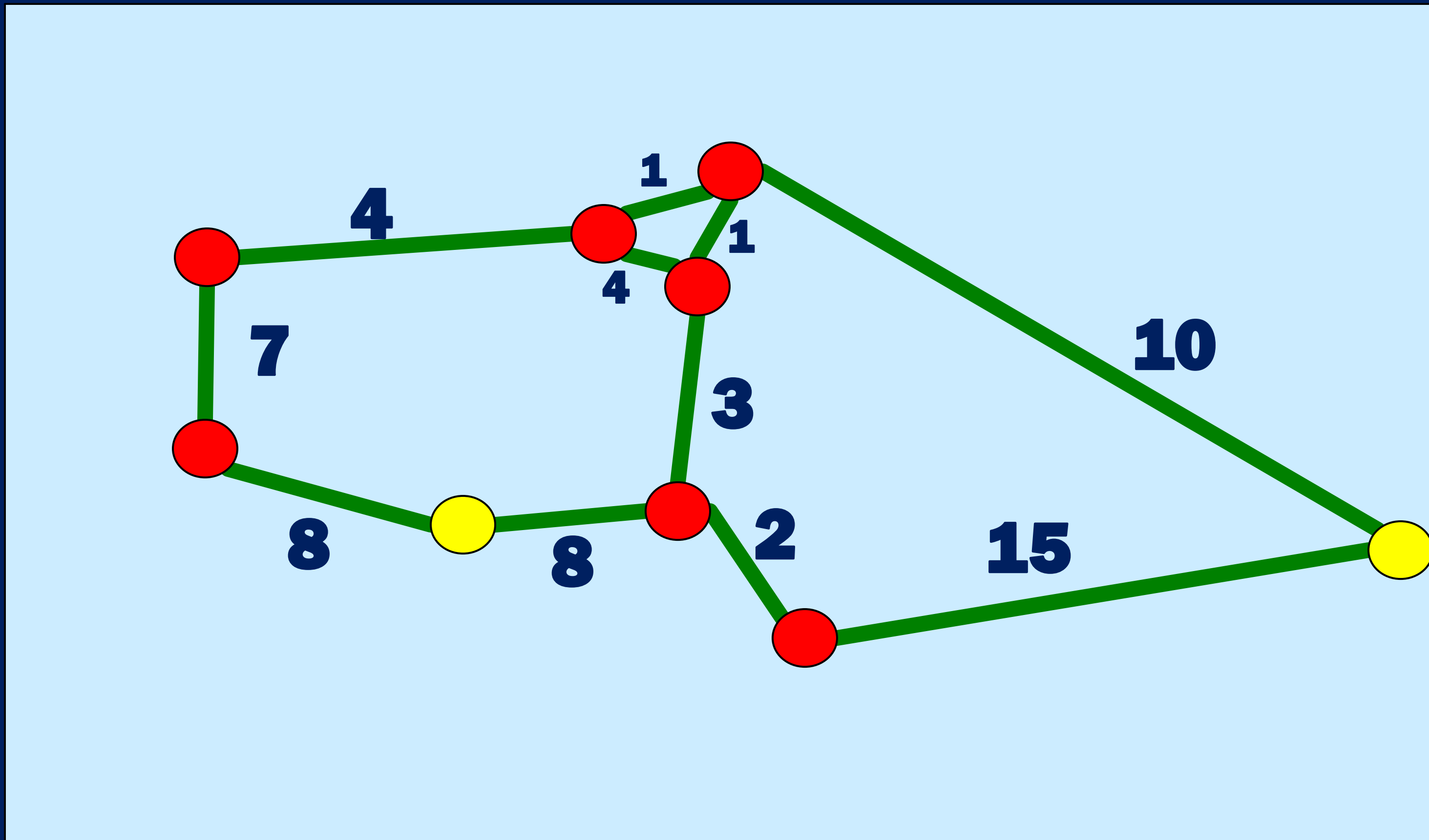
Andrew Goldberg
MSR Silicon Valley

# How it works




1. Put a **VERTEX** at every intersection
2. Put an **EDGE** between connected intersections
3. **LABEL** the edges with travel times

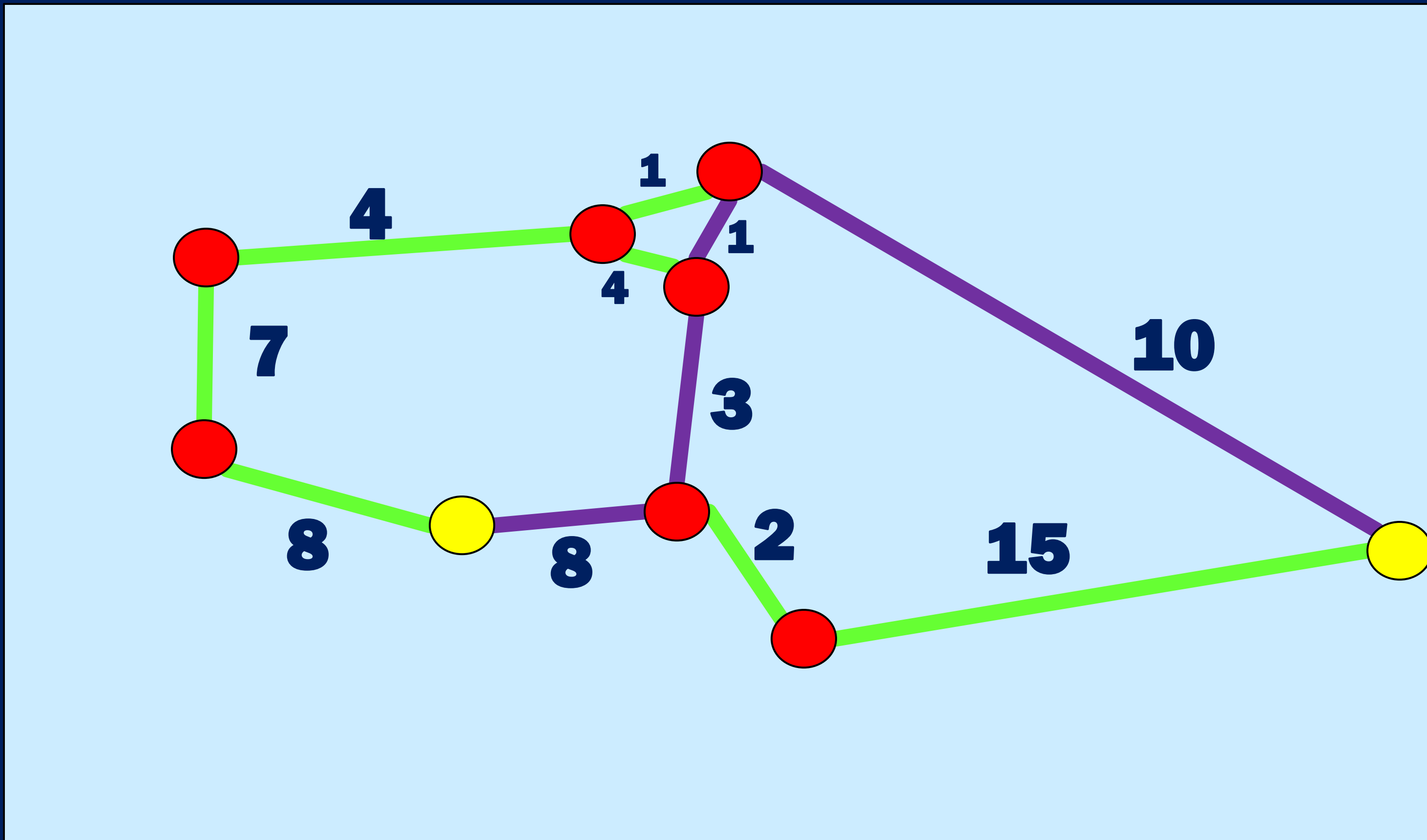
# How it works




4. Now ignore the original map

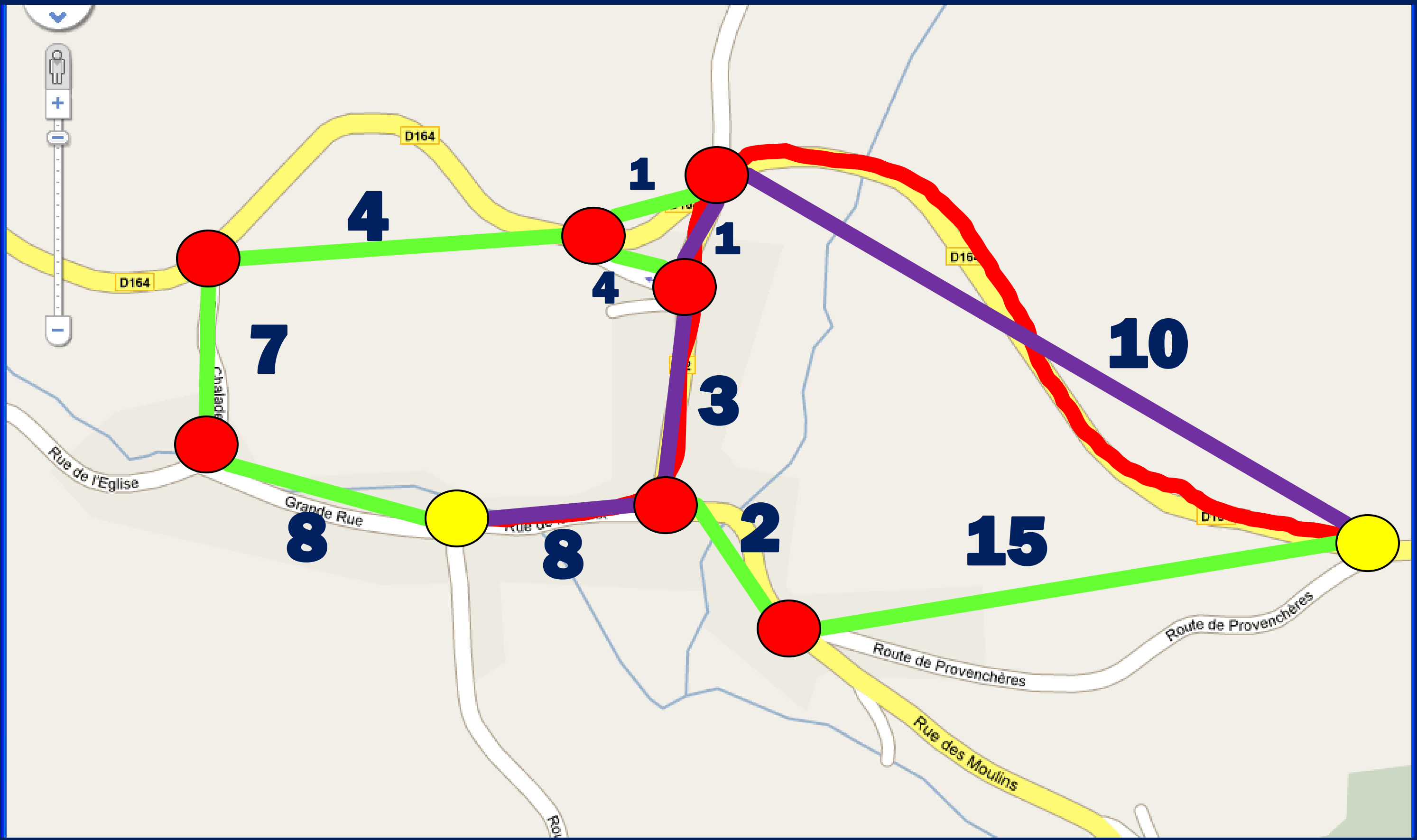
5. Choose START and END points

# How it works



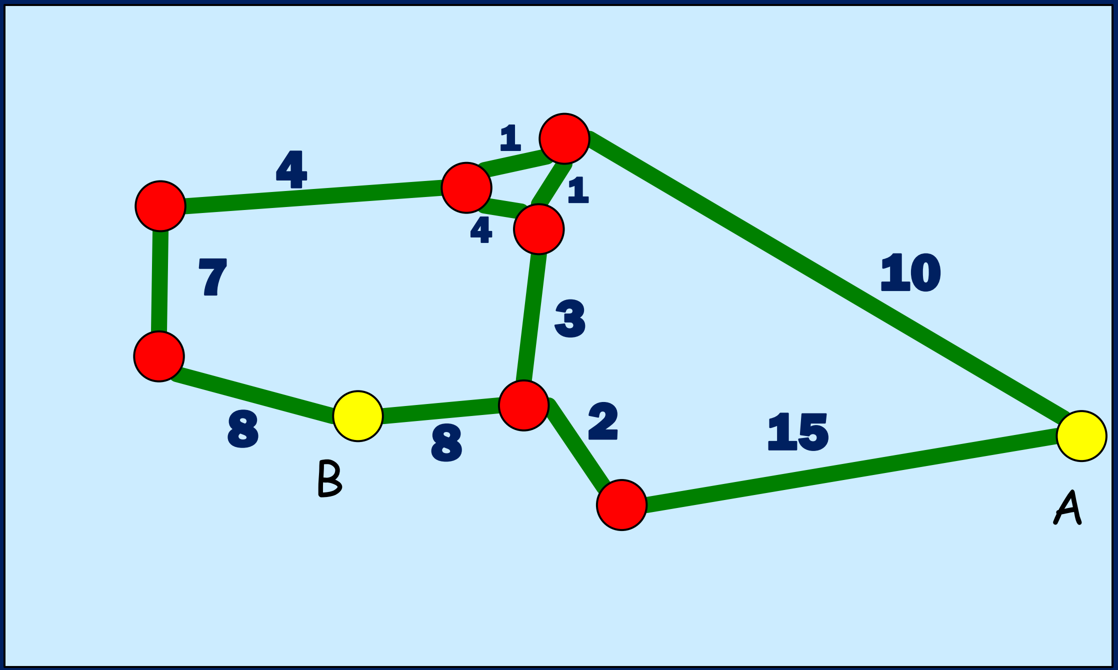

5. Choose **START** and **END** points
6. Find **shortest path**

# How it works



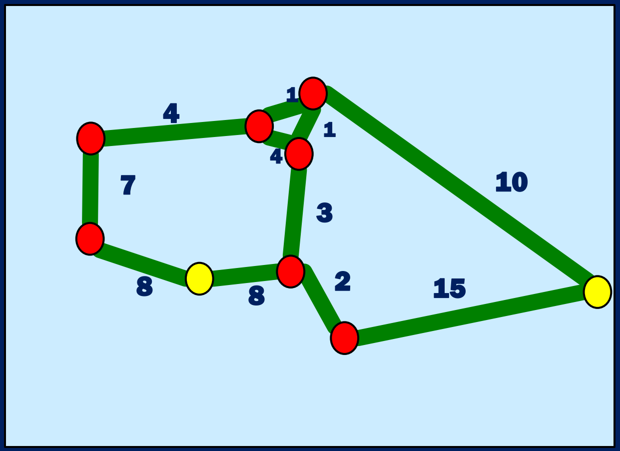

7. Draw on original map, following roads
8. Discard the vertices and edges

# So the real task is:




- Given a "edge-weighted un-directed graph"
- And any two points A and B
- Find the shortest path from A to B
- Length of shortest path = SP(A,B)

# Possible plans




- My mother:
  - Start at A
  - Drive around at random until you find B
- Slow, and even if she arrives, it definitely isn't by the shortest route

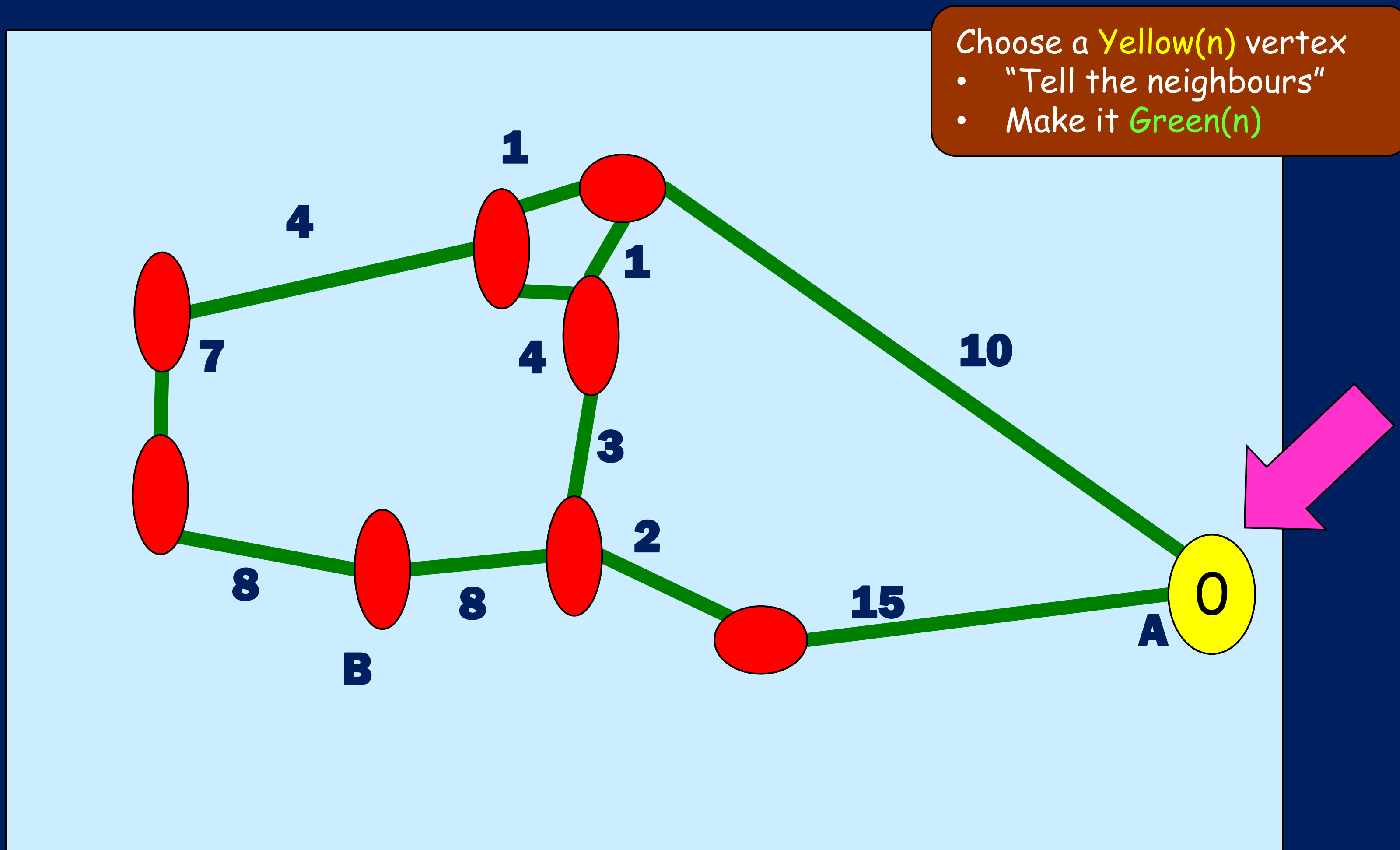
# Labelling method

Starting at A
Destination is B

- Divide vertices V into 3 groups
  - Red: V knows nothing
  - Yellow(n): V knows something:
    - the distance from A to V is no more than n
  - Green(n): V knows something, and so do V's neighbours:
    - the distance from A to V is no more than n
    - V's immediate neighbours know that fact
- Start with A = Yellow(0), everything else Red
- Choose any Yellow(n) vertex
  - Make it Green(n)
  - "Tell the neighbours"
- Stop when all are green

# Labelling method

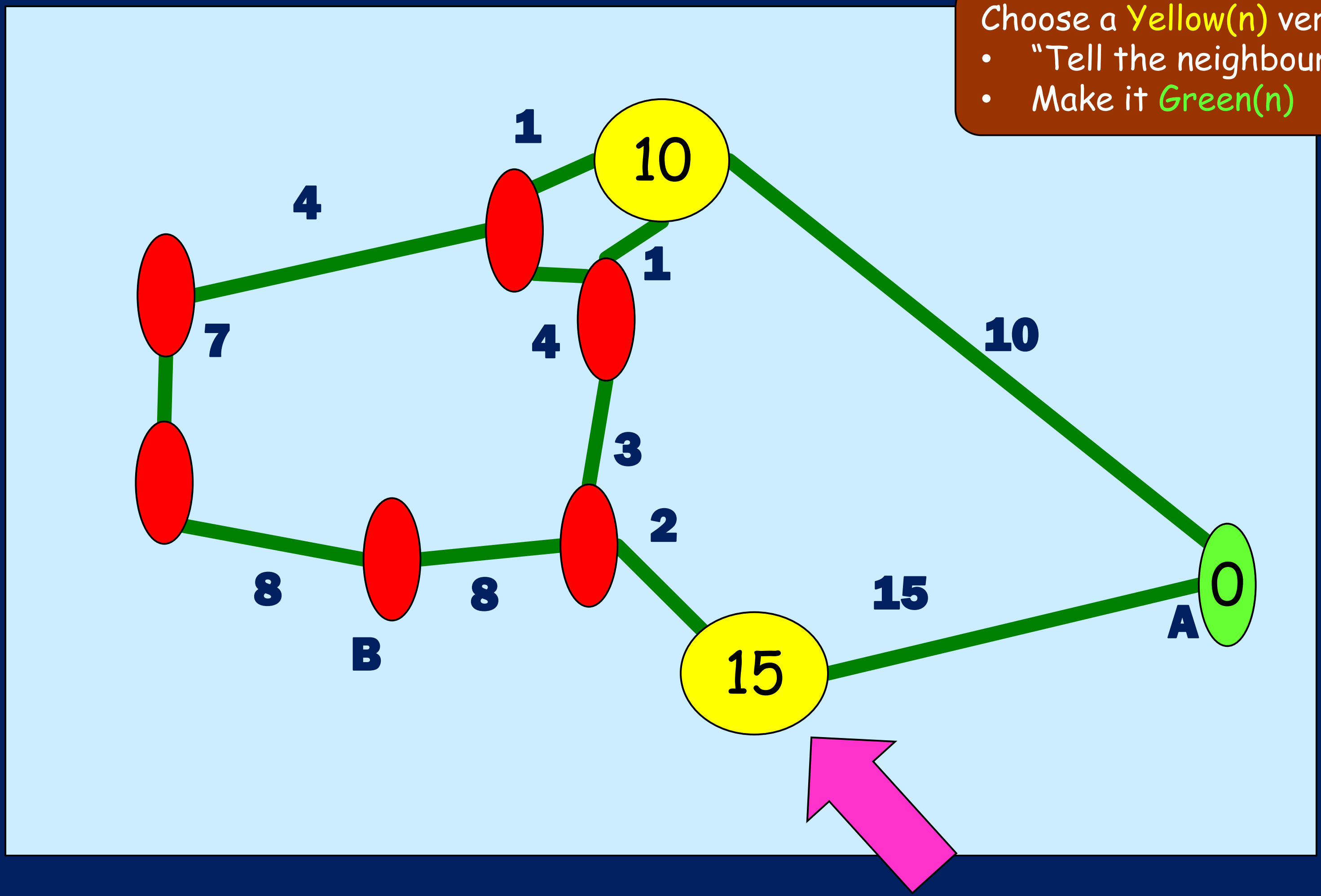
- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that



# Labelling method

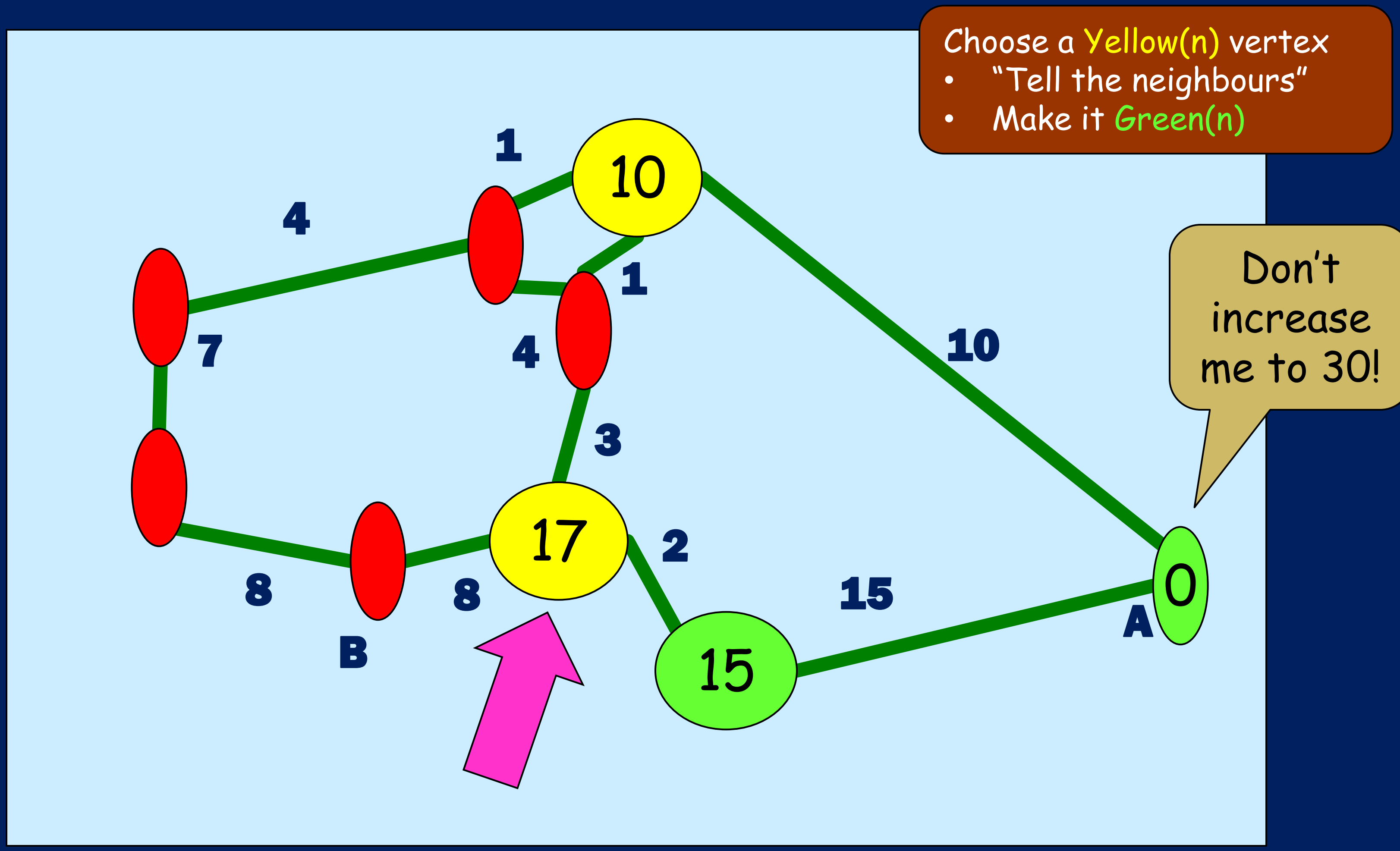
- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that

Choose a Yellow(n) vertex
- "Tell the neighbours"
- Make it Green(n)



# Labelling method

- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that



# Telling the neighbours

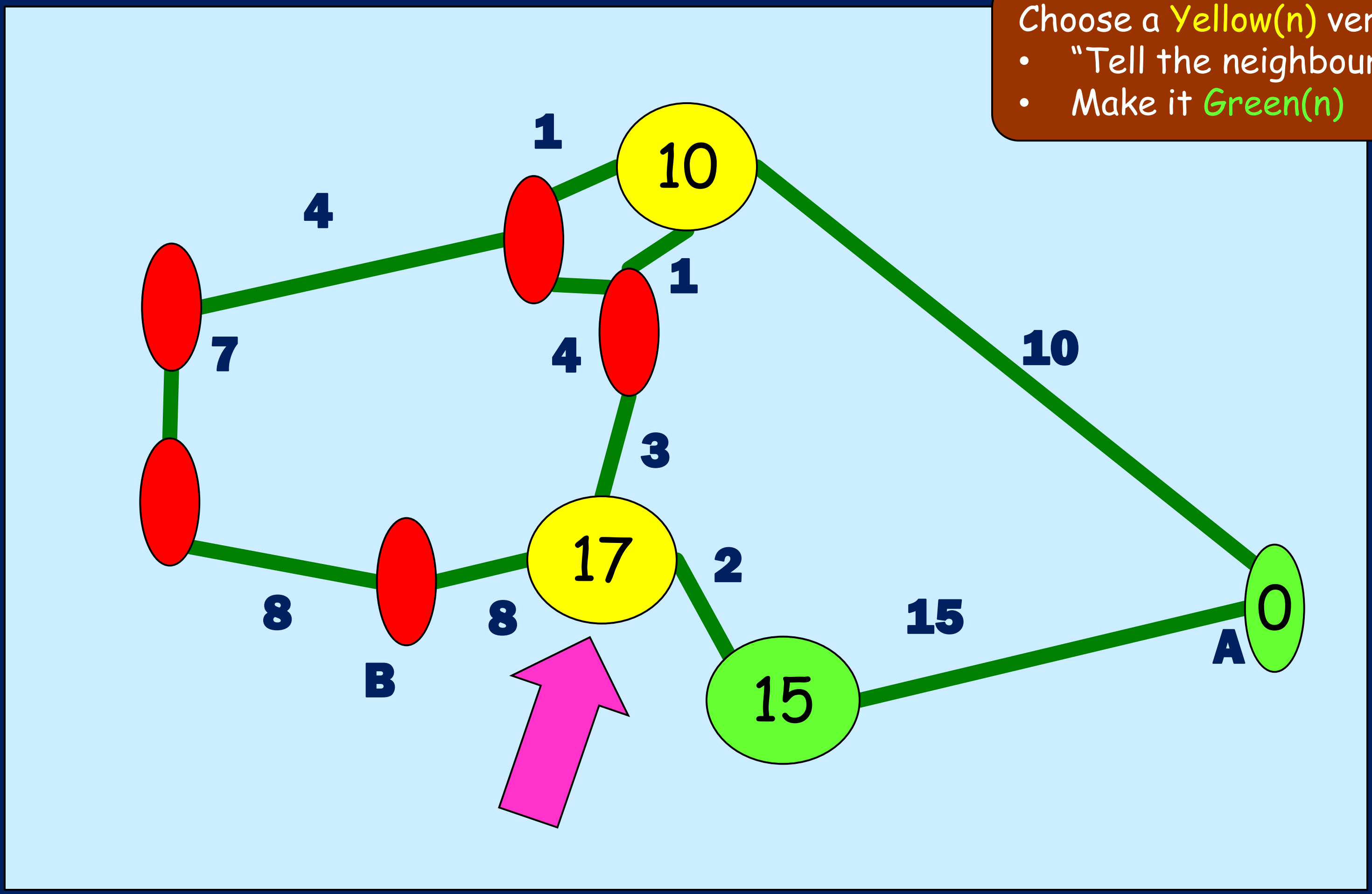
- When we turn V to Green(n)
- Tell the neighbours: for each neighbour W, change W as follows:


  - Yellow or Red -> Yellow( n+k )
  - Green(n): no change

# Labelling method

- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that

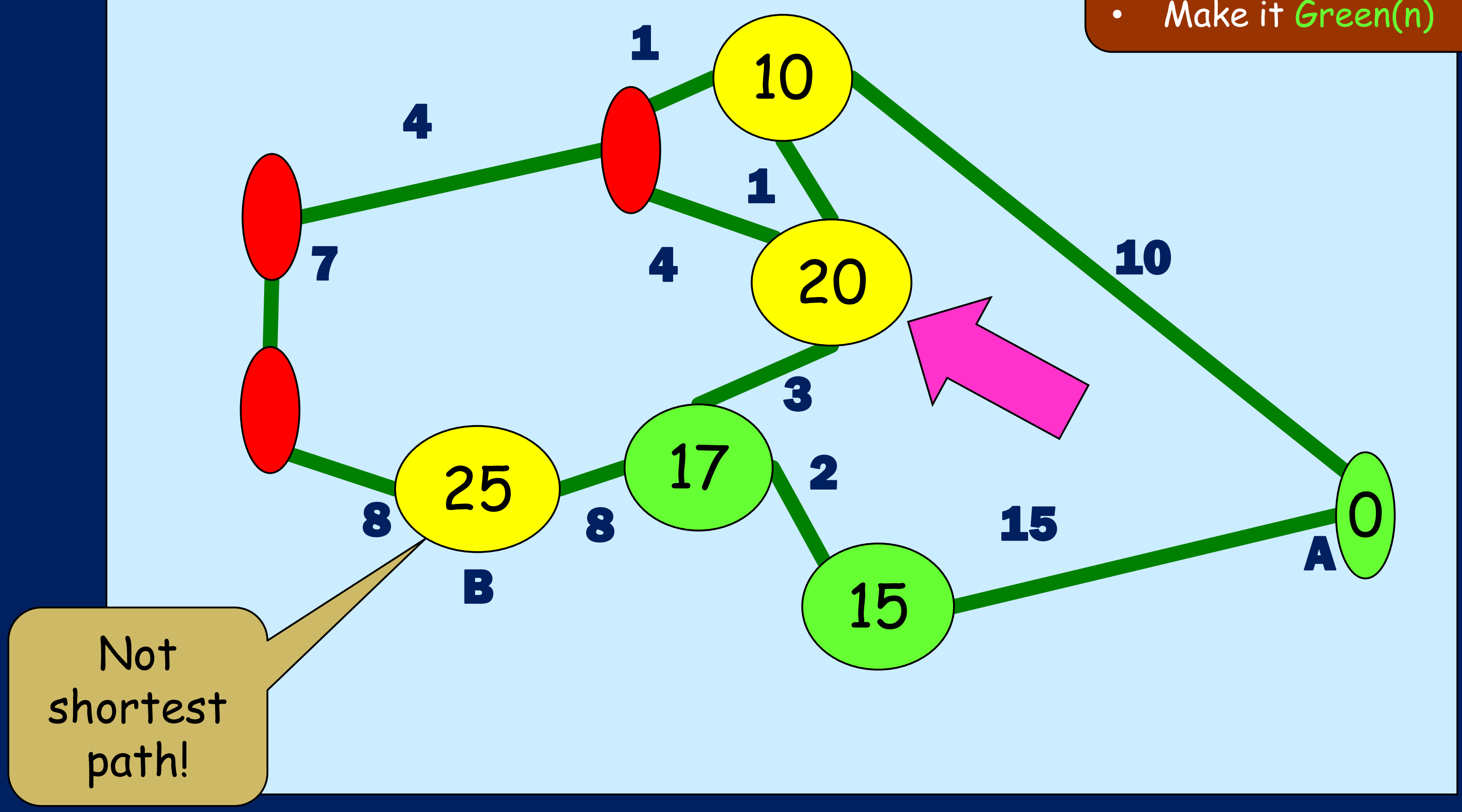
Choose a Yellow(n) vertex
- "Tell the neighbours"
- Make it Green(n)



# Labelling method

- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that

Choose a Yellow(n) vertex
- "Tell the neighbours"
- Make it Green(n)

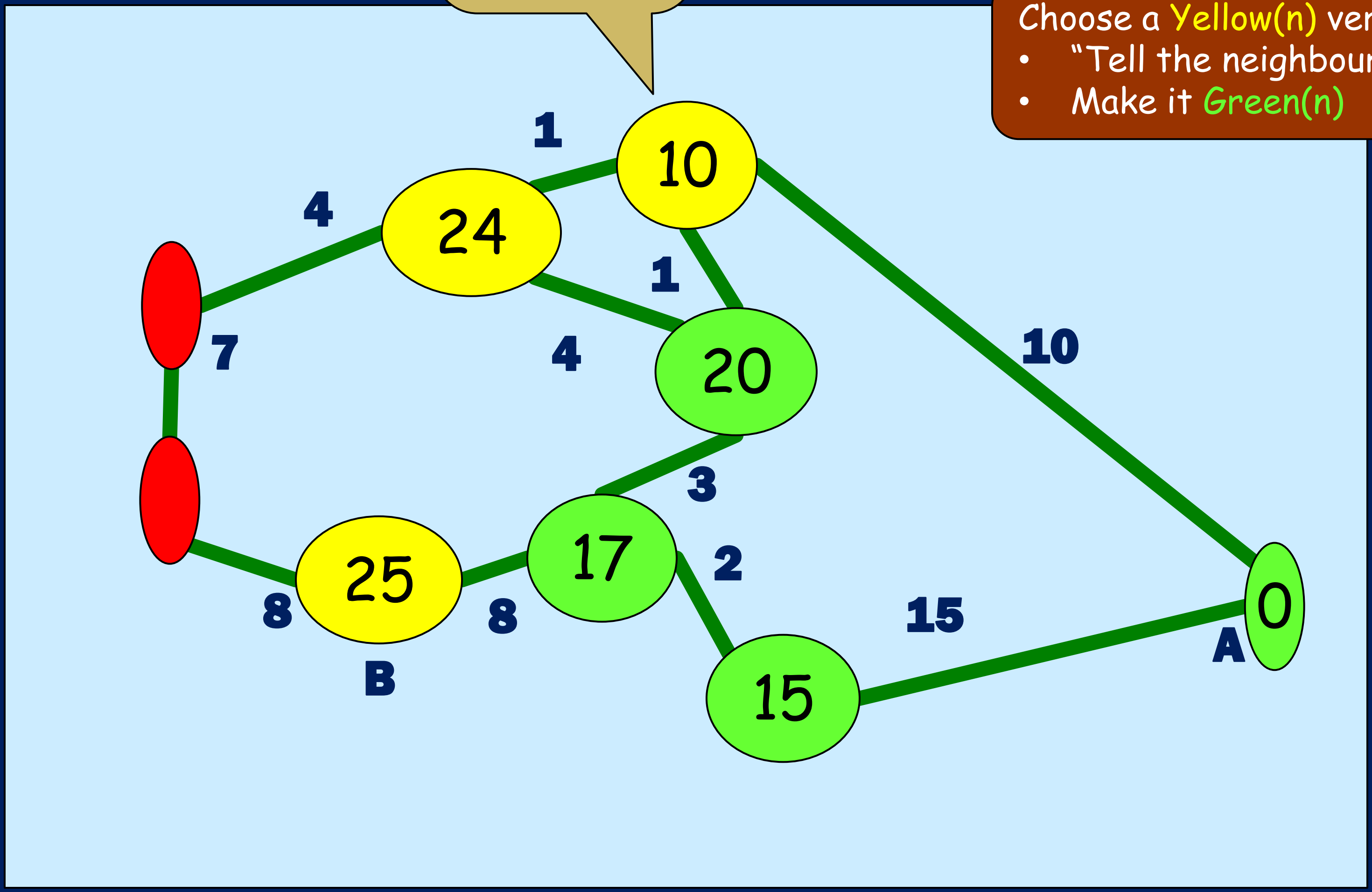


# Labelling me

- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that

Do not increase me from 10 to 21!

Choose a Yellow(n) vertex
- "Tell the neighbours"
- Make it Green(n)



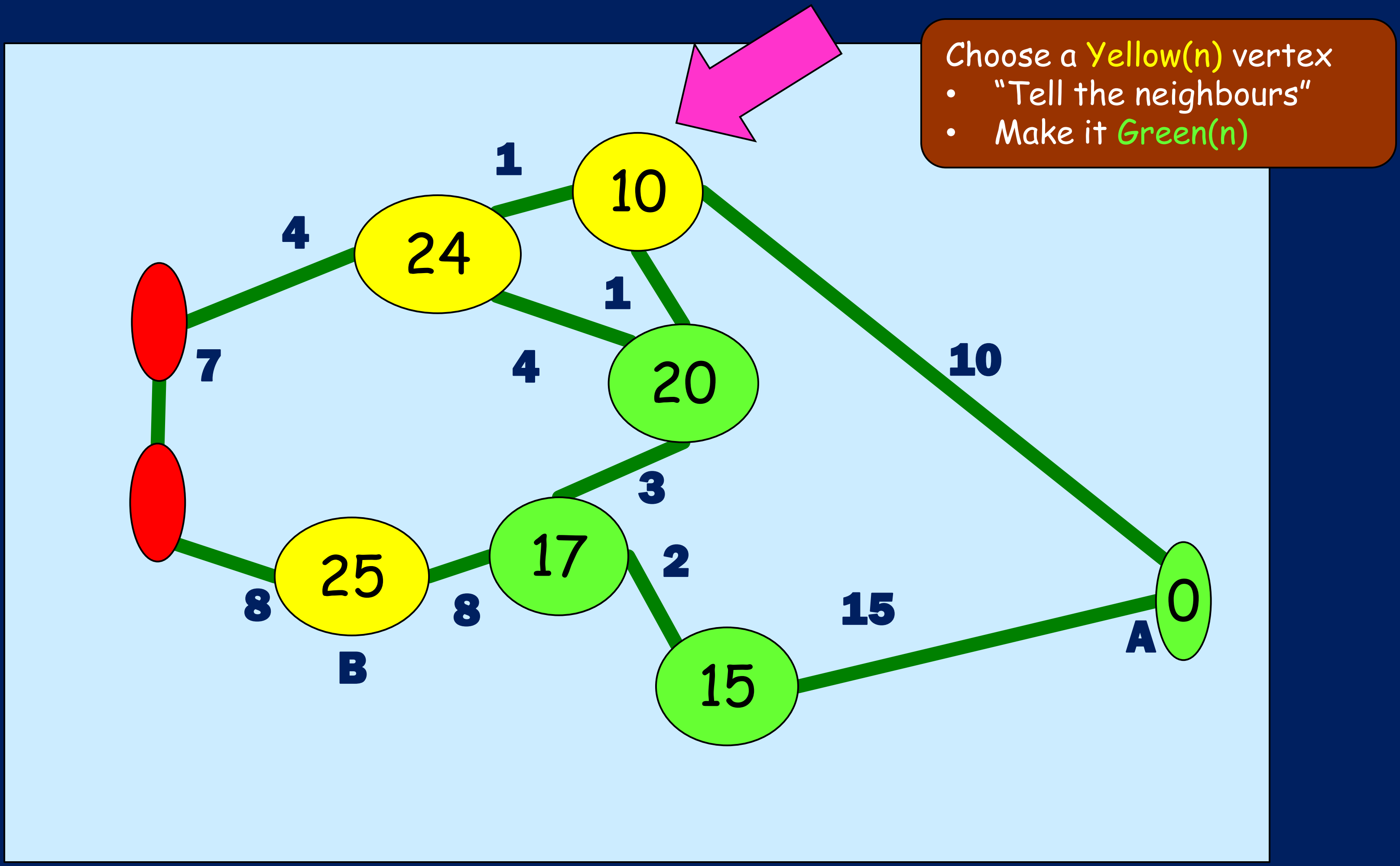
# Telling the neighbours

- When we turn V to Green(n)
- Tell the neighbours: for each neighbour W, change W as follows:


  - Red -> Yellow( n+k )
  - Yellow( m ) -> Yellow( n+k ), if n+k < m
  - Green(n): no change

# Labelling method

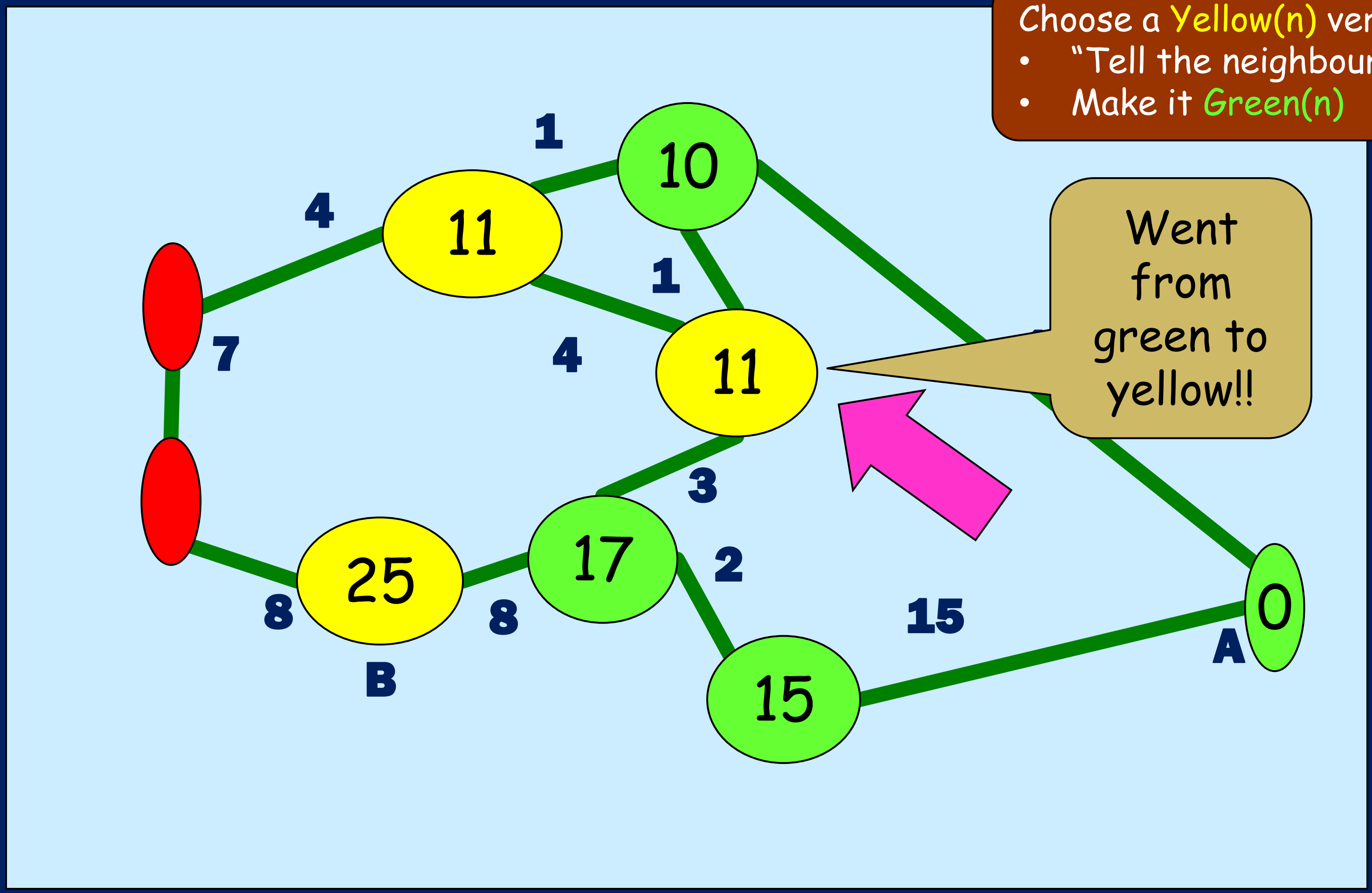
- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that



# Labelling method

- Red: V knows nothing
- Yellow(n): SP(A,V) ≤ n
- Green(n): SP(A,V) ≤ n, and V's neighbours know that

Choose a Yellow(n) vertex
- "Tell the neighbours"
- Make it Green(n)



# Telling the neighbours

- When we turn V to Green(n)
- Tell the neighbours: for each neighbour W, change W as follows:


  - Red -> Yellow( n+k )
  - Yellow( m ) -> Yellow( n+k ), if n+k < m
  - Green( m ) -> Yellow( n+k ), if n+k < m

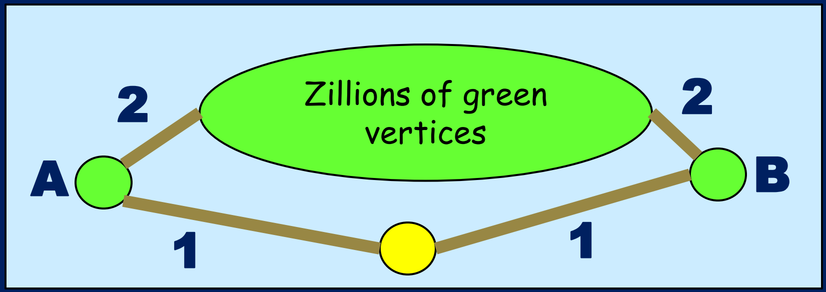
General rule:
Vertex turns yellow when its n decreases
Think of Red as ∞

# Labelling method works

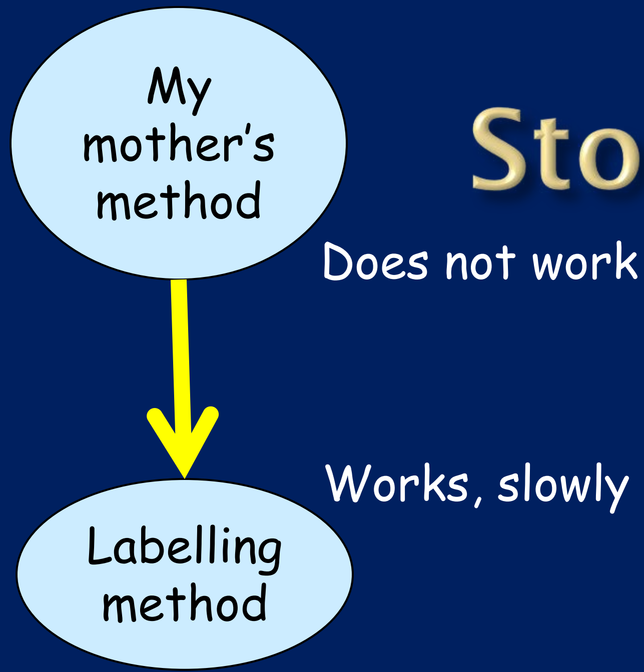
- **Claim**: when all vertices are green, every vertex knows its exact shortest path
- NOT OBVIOUS
- But true. [Exercise: prove it]

# Labelling method works... badly

- **Claim**: when all vertices are green, every vertex knows its **exact** shortest path [Exercise: prove it]
- BUT this is stupid
  - We may visit each vertex **many times** (because of Red -> Yellow -> Green -> Yellow -> Green -> Yellow)
  - We must visit **every vertex** (eg examine all roads in Glasgow when finding a route from London to Reading)



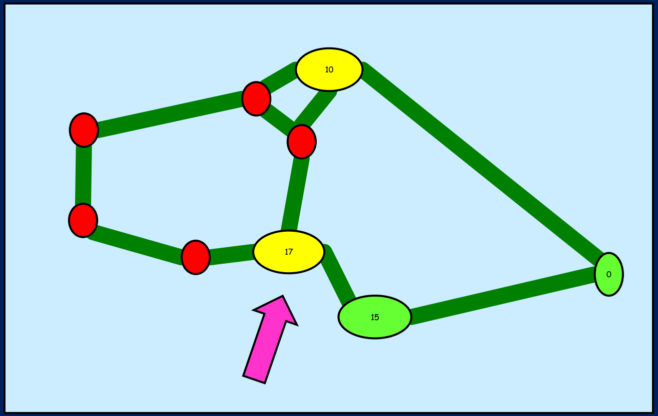
# Story so far


My mother's method
Does not work
Works, slowly
Labelling method

# Hey, computers are fast

- Europe
  - 18M vertices
  - 43M edges
- Visiting all edges and vertices is no big deal
- But it is **STILL** stupid:
  - Slow on a hand-held (seconds or minutes to re-plan your route)
  - Would you prefer a server farm with 500 servers? Or 5?

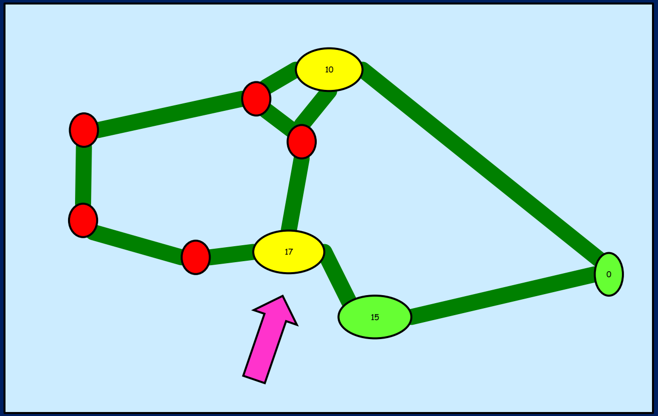
# Labelling method




- Divide vertices into 3 groups
  - Red
  - Yellow(n)
  - Green(n)
- Start with A = Yellow(0), everything else Red
- Choose **any** Yellow(n) vertex
  - “Tell the neighbours
  - Make it Green(n)
- Stop when all are green

IDEA: Choose a good vertex!

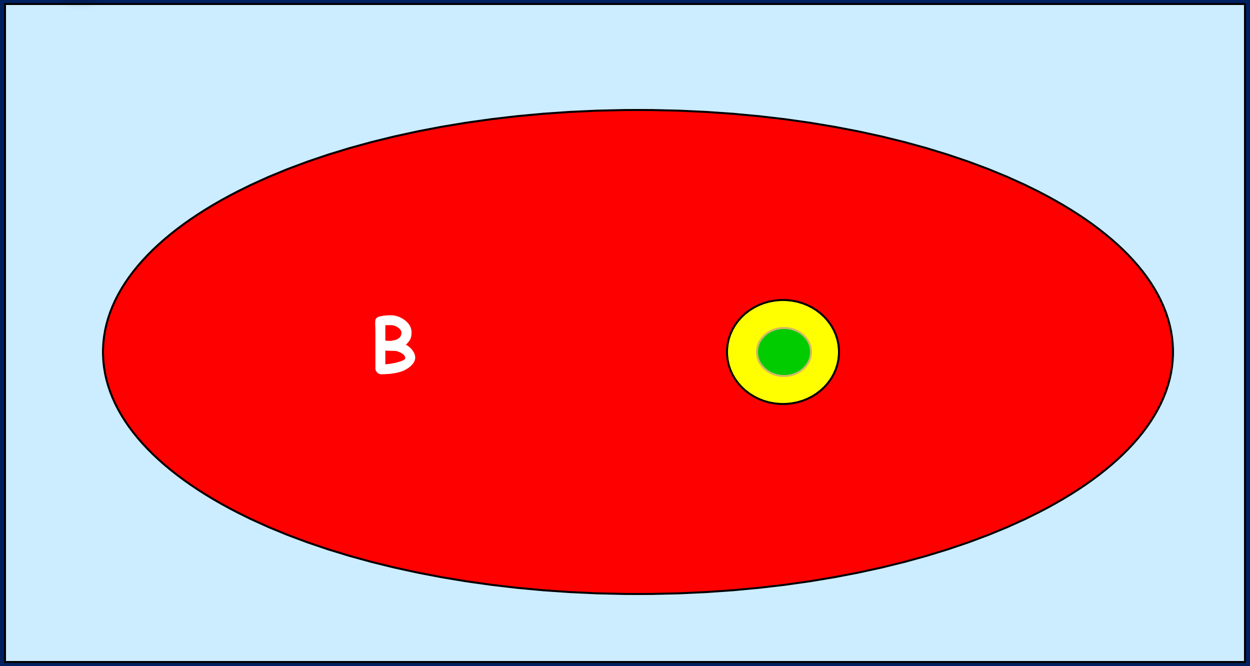
# Dijkstra




- Divide vertices into 3 groups
  - Red
  - Yellow(n)
  - Green(n)
- Start with A = Yellow(0), everything else Red
- Choose ~~any~~ **the** Yellow(n) vertex **with smallest n**
  - "Tell the neighbours"
  - Make it Green(n)
- Stop when all are green

Choose a good one!
"Good" ones are close to A

# Dijkstra does ink-blotting




- Roughly speaking
  - Take nearest yellow vertex
  - Turn it green, and its more distant neighbours yellow

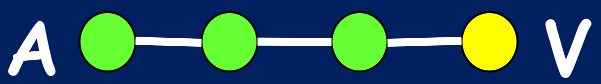
# Two big advantages

1. No vertex changes Green -> Yellow, so we visit each vertex at most once
2. Can stop when **B** becomes Green (rather than when **all** vertices become Green)

WHY?

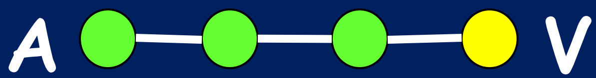
# Why Dijkstra works

- Dijkstra properties
  - Yellow(n): shortest green-only path A to V is **exactly** n
  - Green(n): shortest path A to V is **exactly** n




A “green-only path”

# Why Dijkstra works



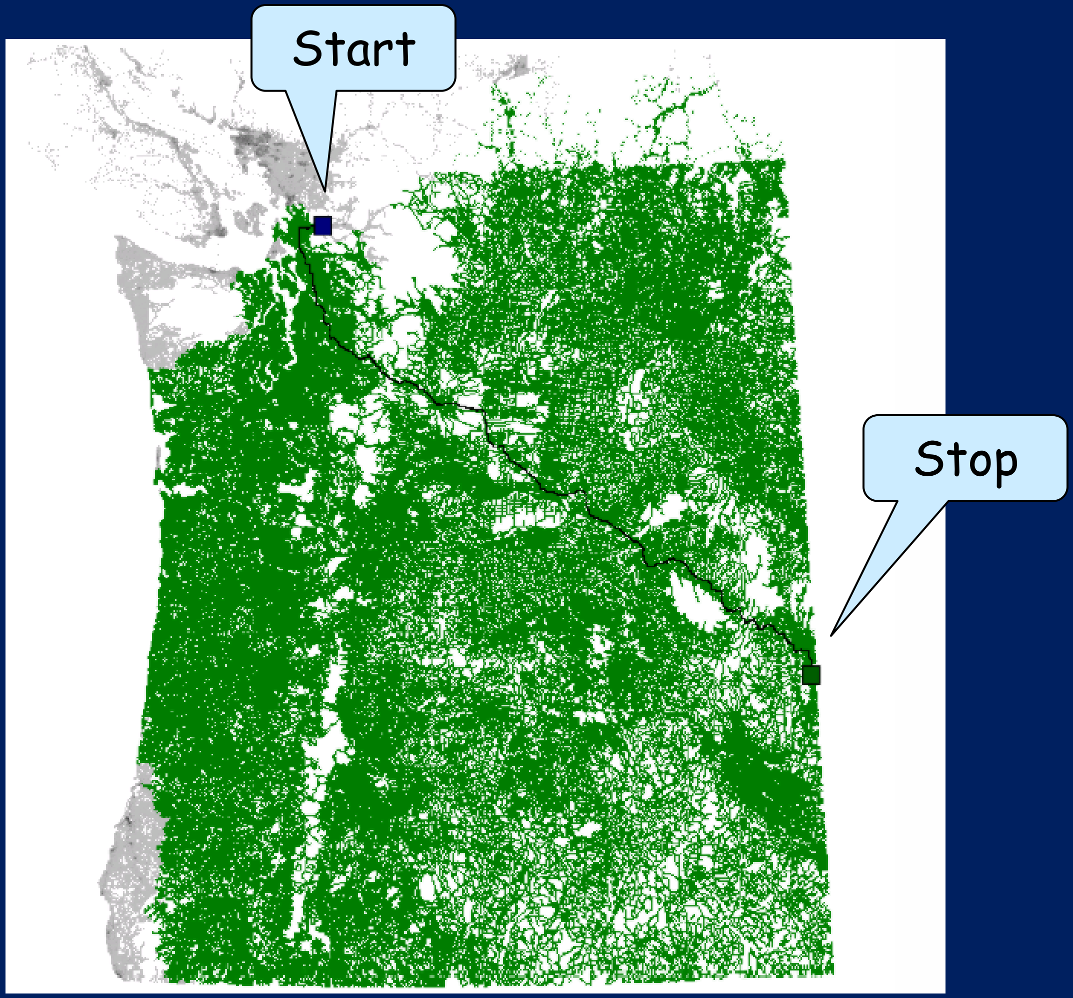

- Dijkstra properties
  a. Yellow(n): shortest green-only path A to V is exactly n
  b. Green(n): shortest path A to V is exactly n
- True initially: A = Yellow(0)
- Choose the Yellow(n) vertex V with smallest n
  - Make it Green(n): (b) remains true
    - We know shortest green-only A-V path is n
    - Any shorter A-V path must go GGGGY(m)....Y(n)
    - Hence m<n, contradication!
  - "Tell the neighbours": makes (a) true again

# Two big advantages

1. No vertex changes Green -> Yellow, so we visit each vertex at most once
   - Because once Green, it has the right n, so n cannot decrease any more
2. Can stop when B becomes Green (rather than when **all** vertices become Green)
   - Because Green vertices have exact shortest path.

# Still too slow

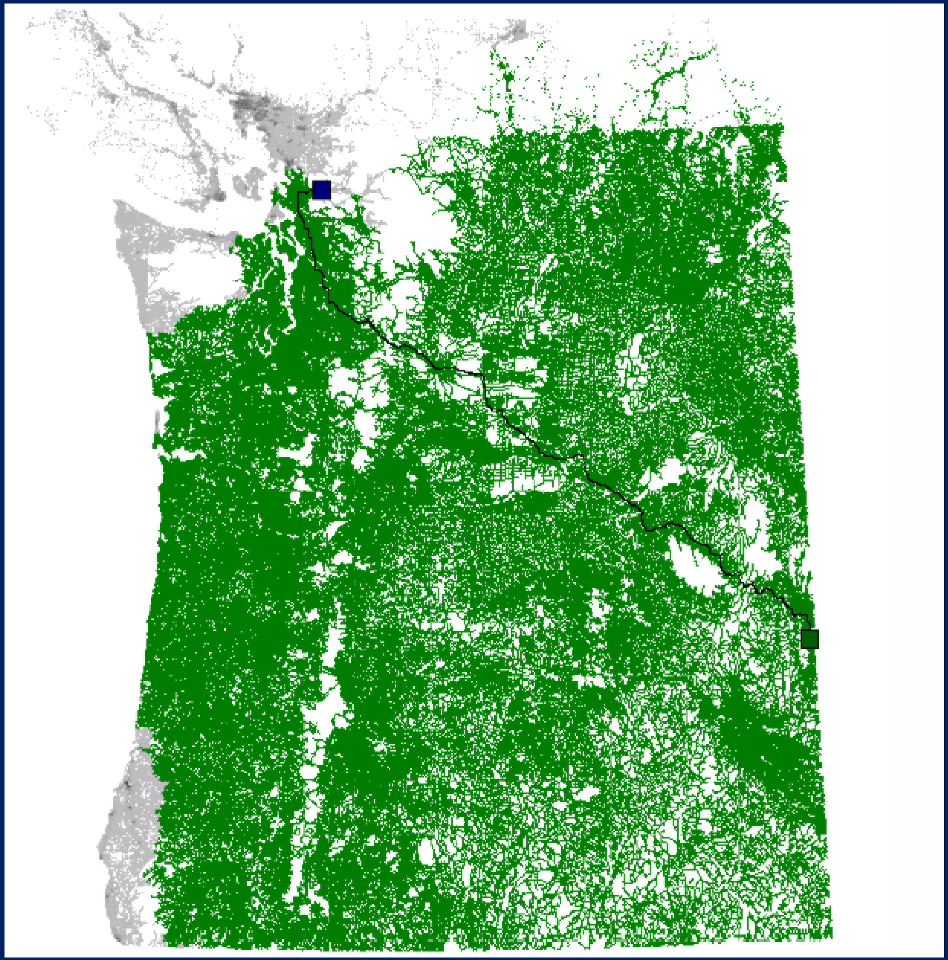
- Far, far too many roads explored!




Pacific North West USA

# Still too slow

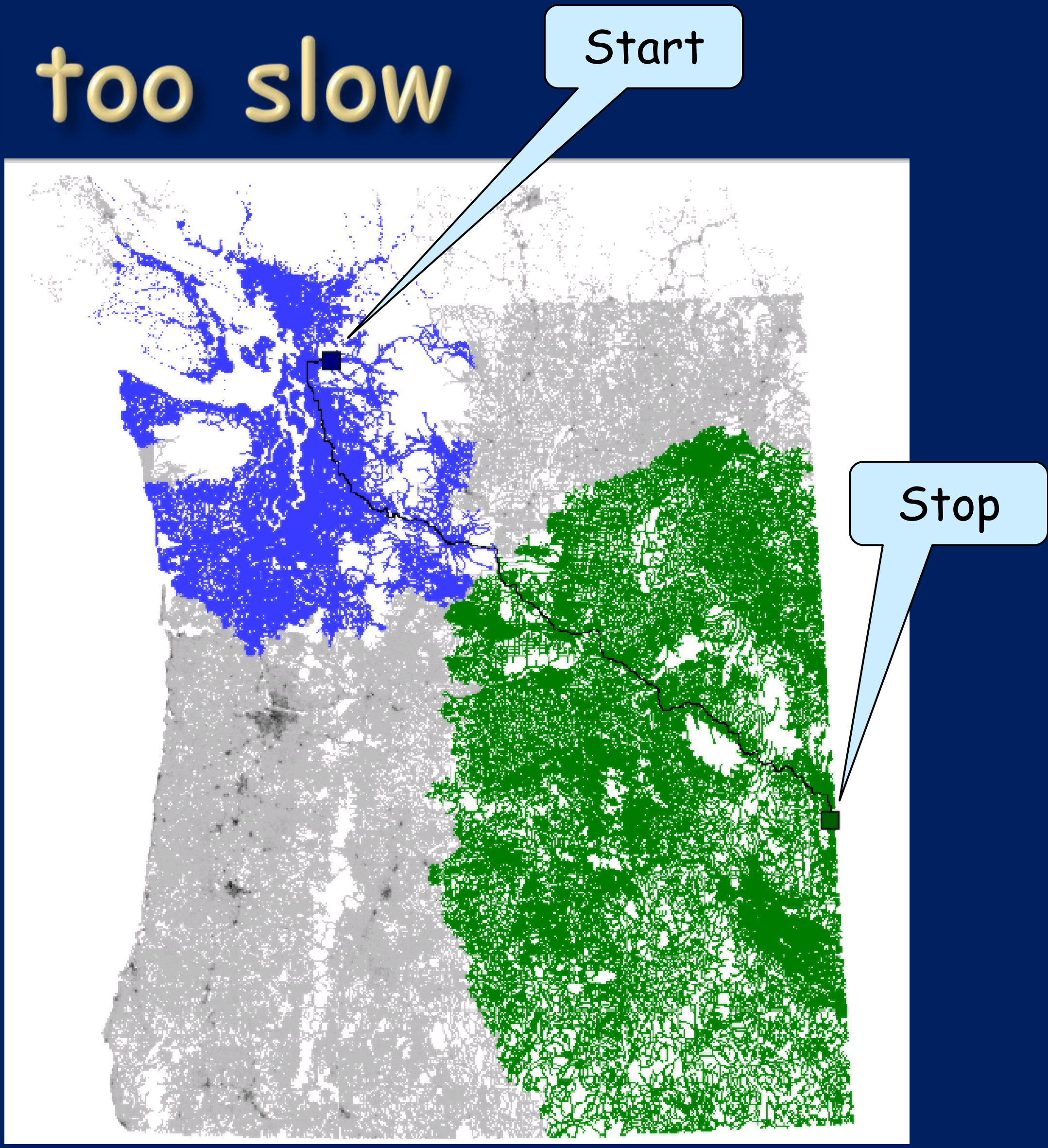
- Idea: start from both ends and work towards the middle



Pacific North West USA

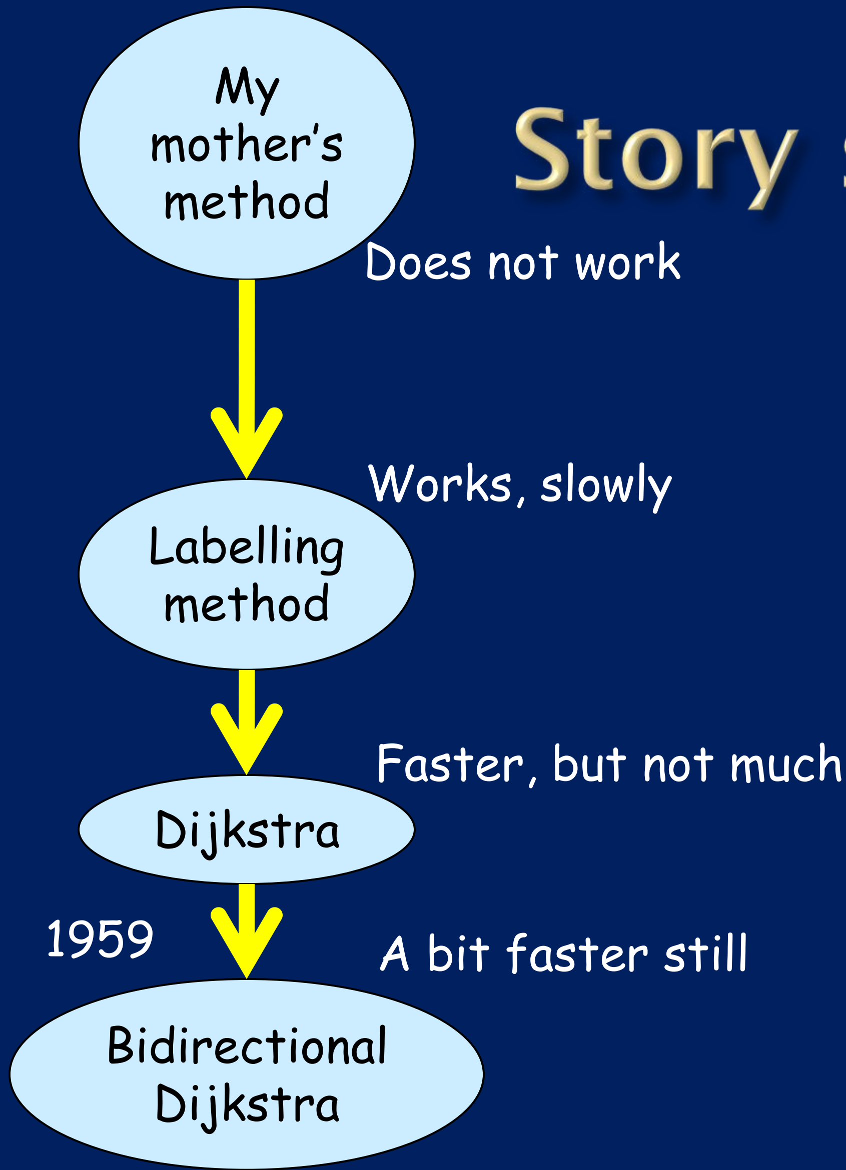
# Still too slow

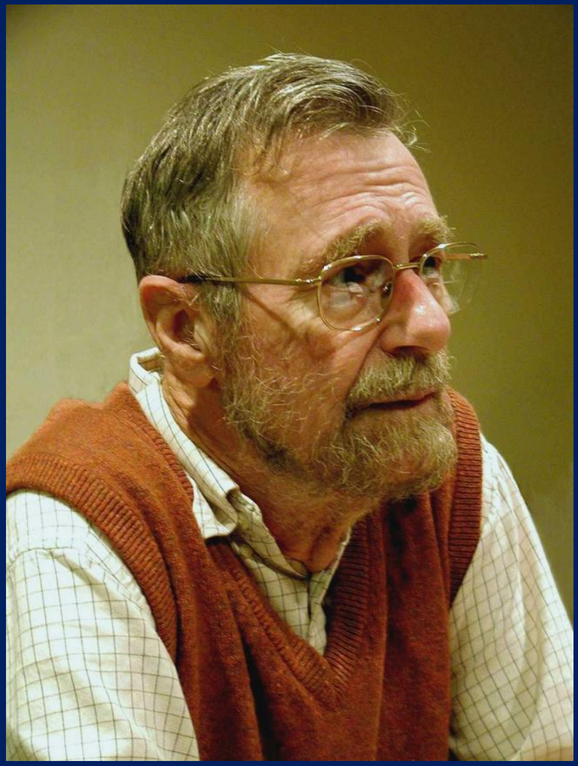
- Idea: start from both ends and work towards the middle
- Better
- But still far too many roads explored




Pacific North West USA

# Story so far



Edsger Dijkstra 1930-2002

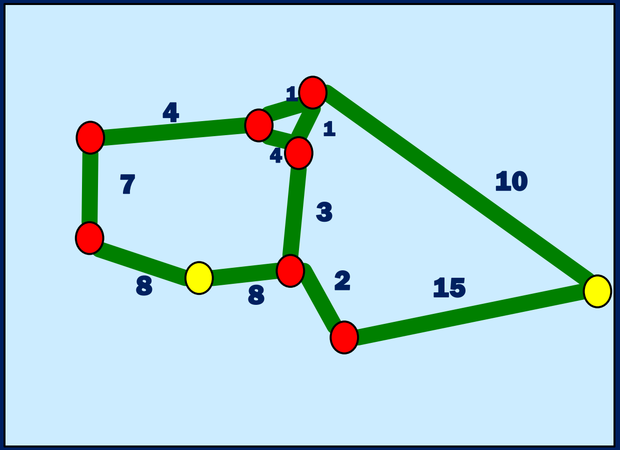
# Still too slow

- Still far too many roads explored
- Why???
- Space warp!

**Idea**
Exploit our knowledge of **lower bounds**
"The shortest path from W to V cannot be less than ...."


Pacific North West USA

# Dijkstra

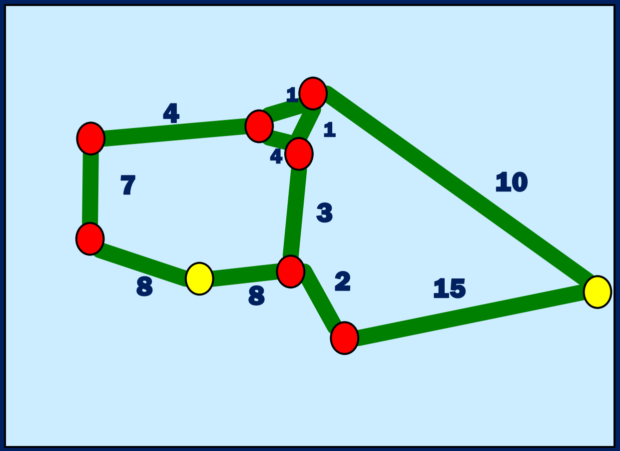



- Divide vertices into 3 groups
  - Red
  - Yellow(n)
  - Green(n)
- Start with A = Yellow(0), everything else Red
- Choose Yellow(n) vertex V with smallest n
  - Make it Green(n)
  - Tell its neighbours
- Stop when B is Green

"Good" ones are close to A

Not necessarily!

# ~~Dijkstra~~ A* search




- Divide vertices into 3 groups
  - Red
  - Yellow(n)
  - Green(n)
- Start with A = Yellow(0), everything else Red
- Choose Yellow(n) vertex V with smallest **(n+L(V))**
  - Make it Green(n)
  - Tell its neighbours
- Stop when B is Green

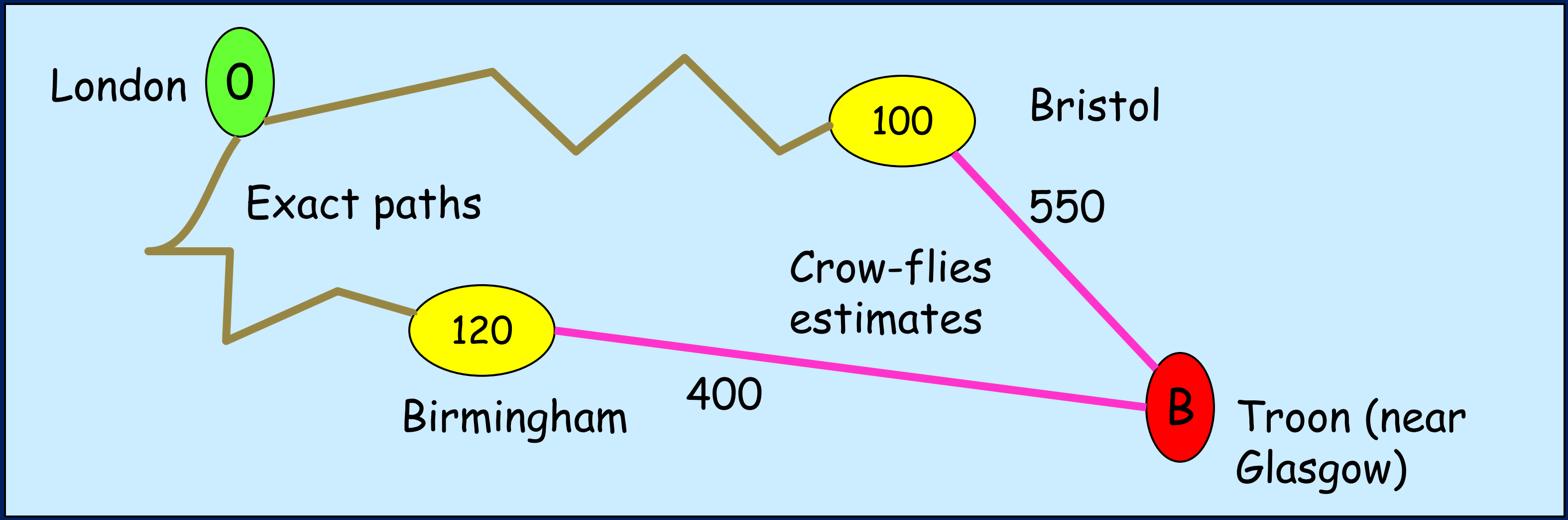
**L(V)** is a **lower bound for** shortest path V-B

"Good" node has smallest estimated **complete** path

L(V)=0 gives Dijkstra

# Euclidean bounds

- L(V) = Euclidean distance CrowFlies(V,B)

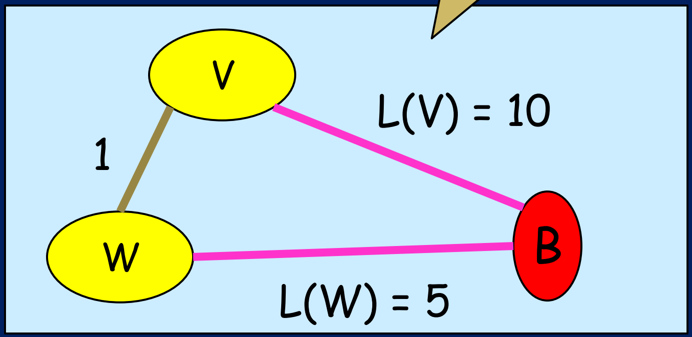



- Birmingham: 120+400 = 520
- Bristol: 100+550 = 650
- So work on Birmingham first (unlike Dijkstra)

# Does A* still work?

- Provided the lower bound estimate L(V) is "sensible" (which CrowFlies is) then yes, A* finds the exact shortest path
- NOT OBVIOUS [prove it!]
- "Sensible" iff
  - L(B)=0
  - It respects the triangle inequality: $L(V) \leq VW + L(W)$

L(V) is not sensible




- Thm: if L is "sensible" then $L(v) \leq SP(V,B)$

# ALAS!

- Euclidean bounds do not work well for route-finding in road maps
- Why? Because motorways are like space warps!
- Need CrowFlies(A,B) ≤ SP(A,B), so the crow must fly at the fastest motorway speed
- Which means that CrowFlies(A,B) is small
- Which is bad, bad, bad.

# Story so far

My mother's method

Does not work

Labelling method

Works, slowly

Dijkstra

Faster, but not much

Dijkstra 1959

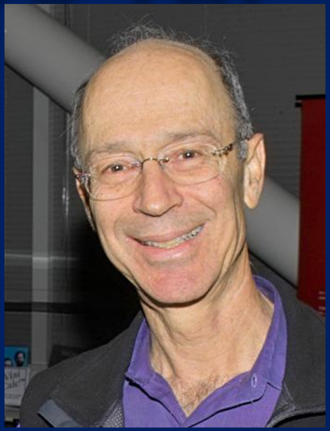
Bidirectional Dijkstra

A bit faster still

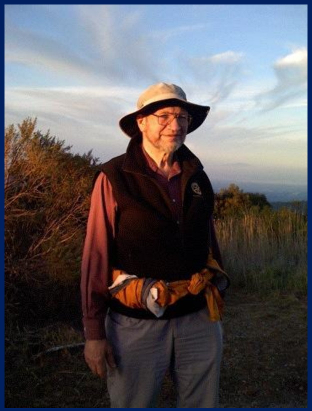
Still no cigar

A* with Euclidean bounds

Hart, Nilsson, Raphael 1968



Peter Hart 1940ish-

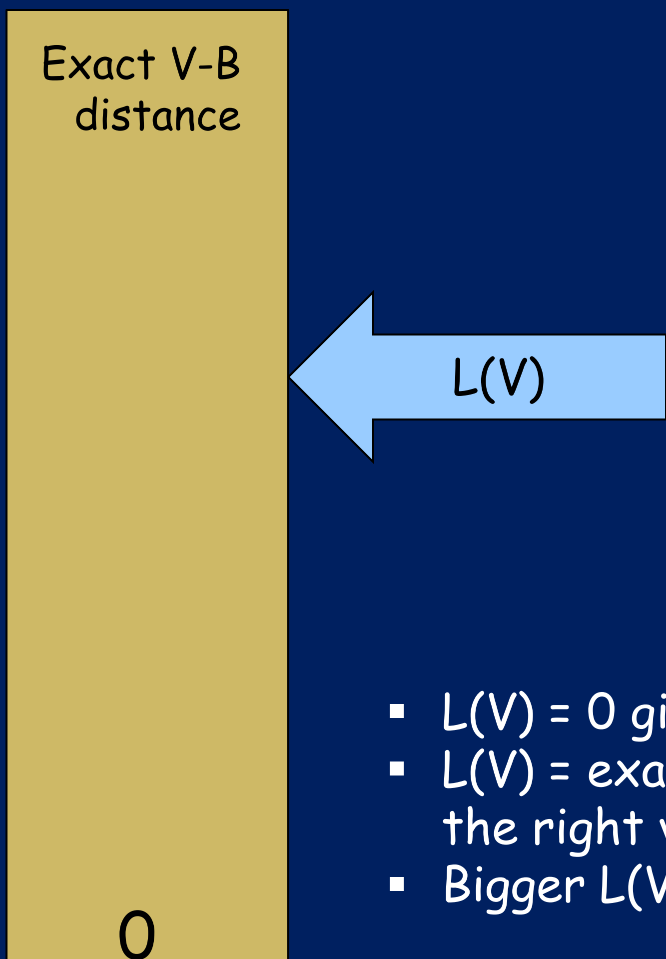


Bert Raphael 1936-

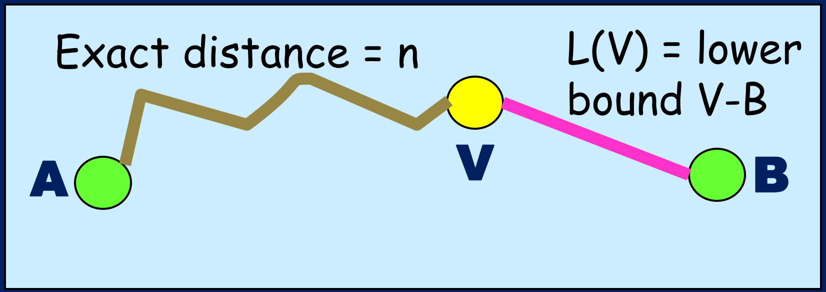
Nils Nilsson 1940ish-

# Better lower bounds

- Choose Yellow(n) vertex V with smallest **(n+L(V))**



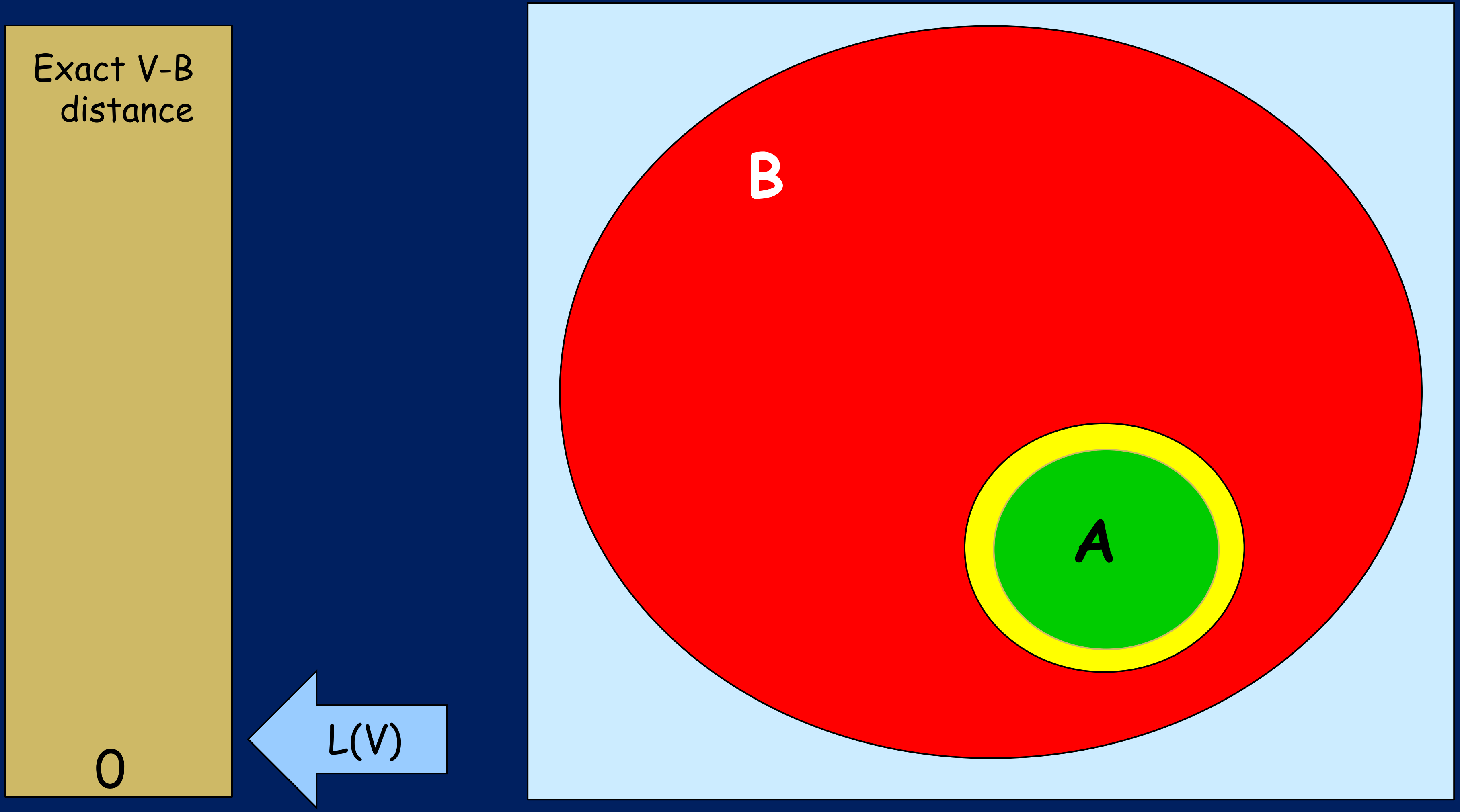

- L(V) is less than the exact distance V-B
- n is the exact distance A-V
- n+L(V) is less than the exact distance A-B




- L(V) = 0 gives Dijkstra
- L(V) = exact distance V-B means we always pick precisely the right vertex
- Bigger L(V) is better (provided always < exact distance)

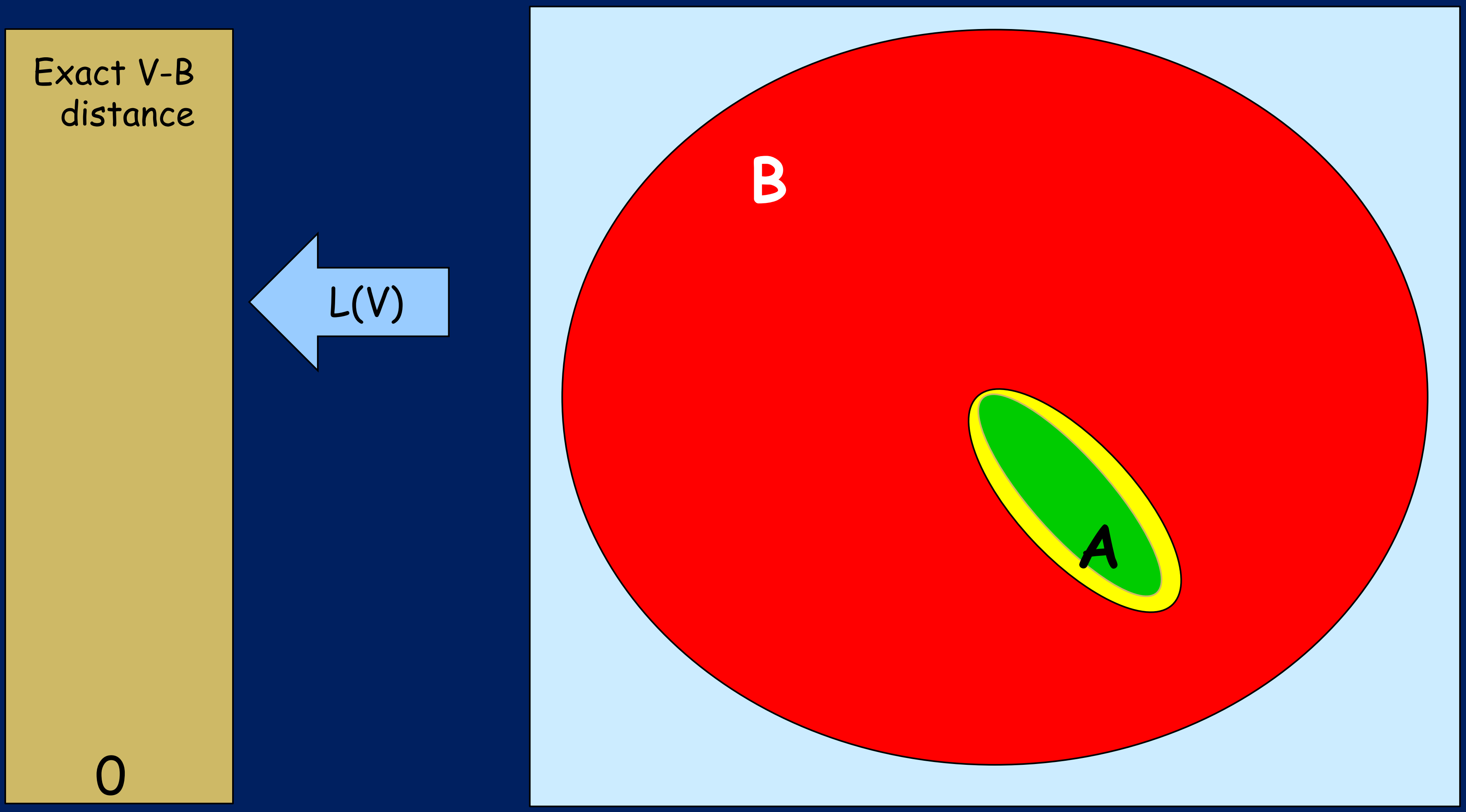
# Better lower bounds

- Choose Yellow(n) vertex V with smallest (n+L(V))



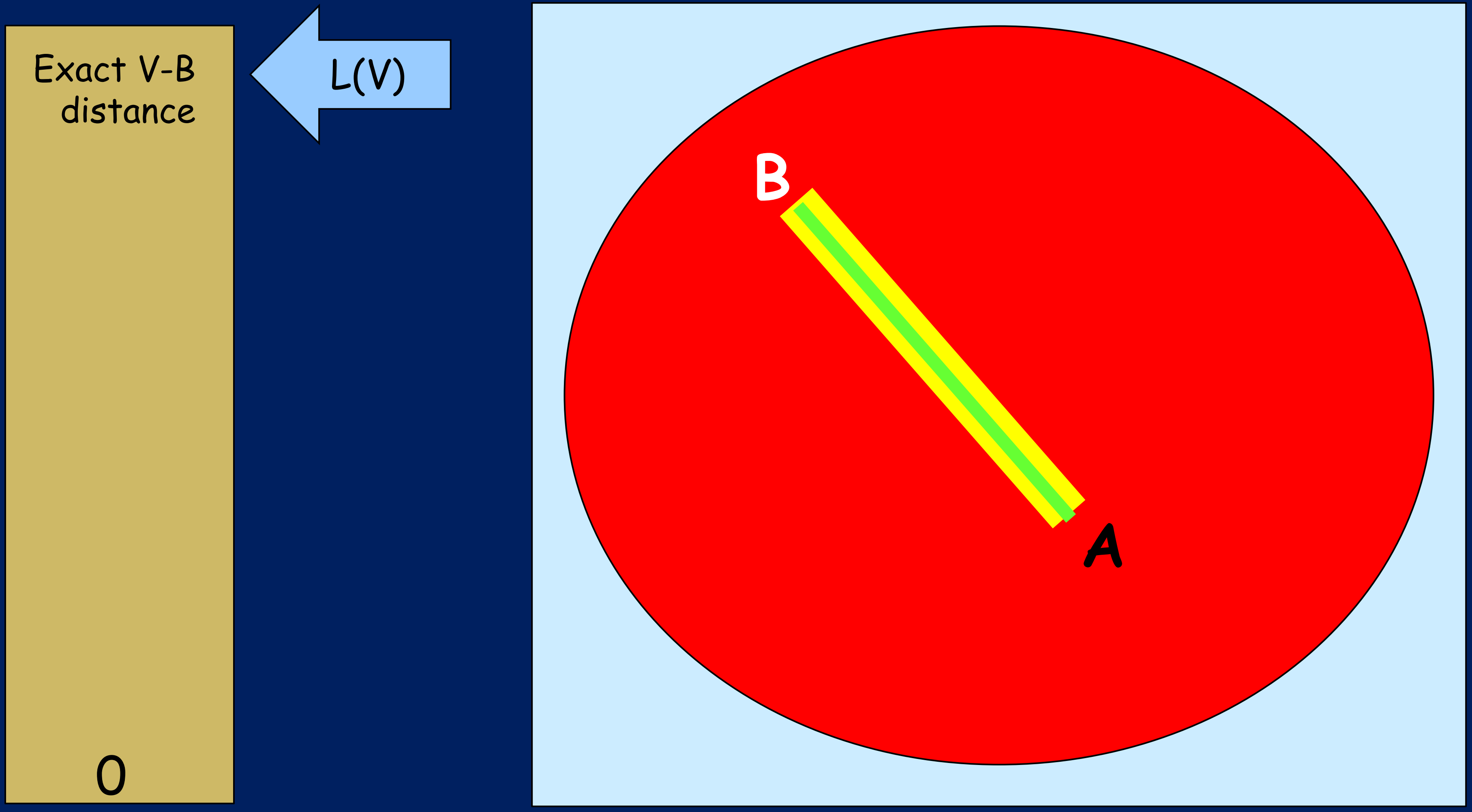
# Better lower bounds

- Choose Yellow(n) vertex V with smallest (n+L(V))



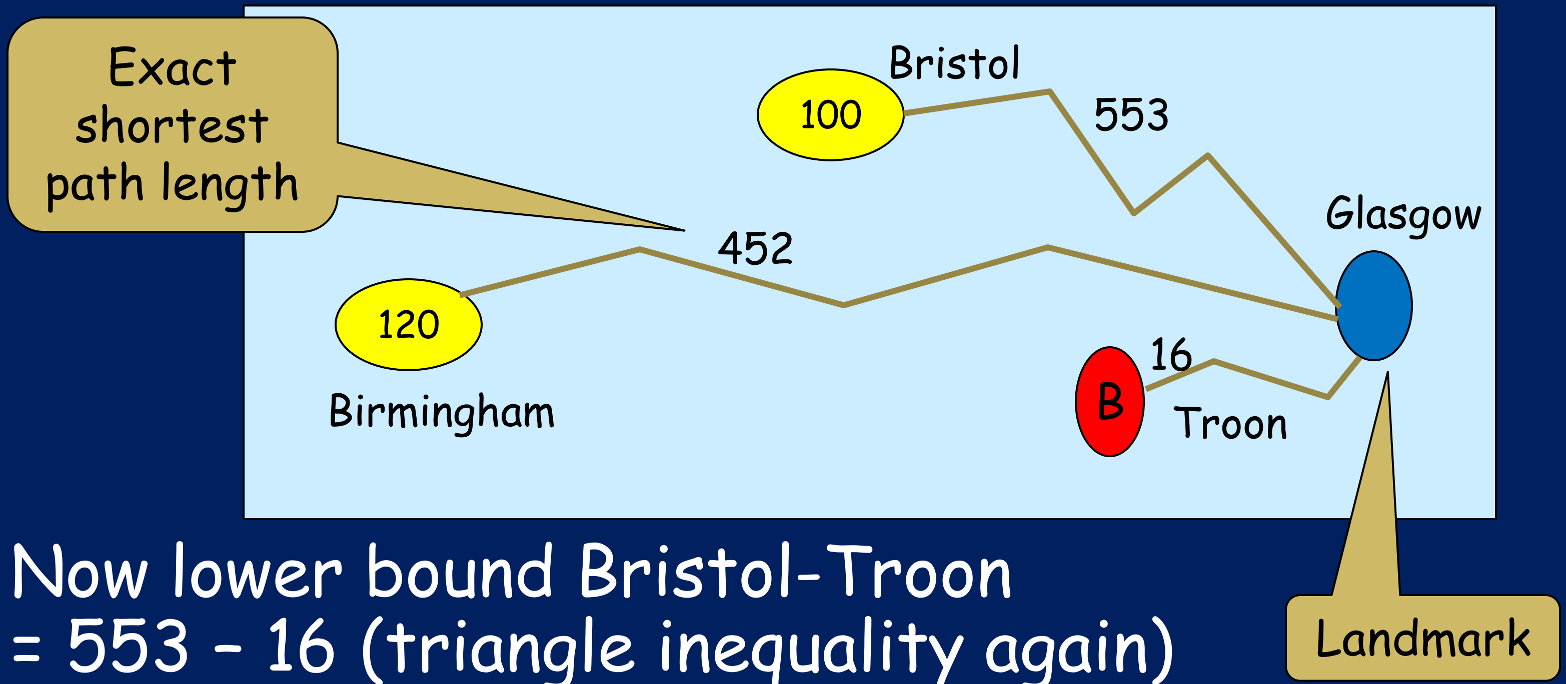
# Better lower bounds

- Choose Yellow(n) vertex V with smallest (n+L(V))



# Wanted: better lower bounds

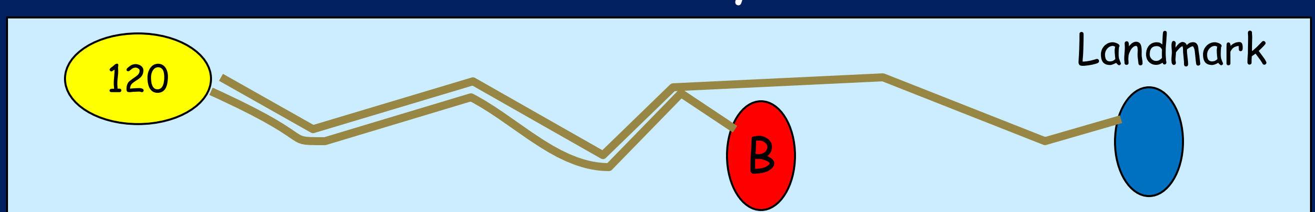
- Idea!
  - Fix a few **landmarks**
  - Pre-compute **exact shortest paths** SP(V,L) from every vertex V to each landmark L




- Now lower bound Bristol-Troon = 553 – 16 (triangle inequality again)

# Wanted: better lower bounds

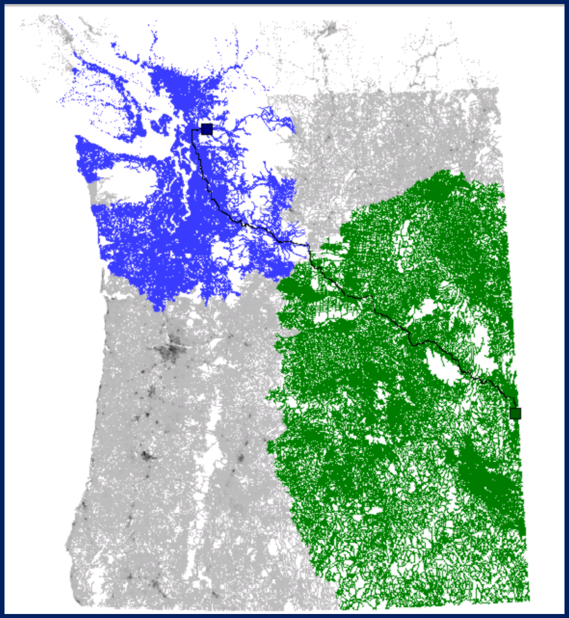
- Idea!
  - Fix a few **landmarks** (eg a dozen or two)
  - Pre-compute and store **exact shortest paths SP(V,L)** from every vertex V to each landmark L
- Main point: the pre-computation takes account of motorways
- Good lower bound if
  - Destination is near the landmark
  - Or the landmark is “beyond” destination



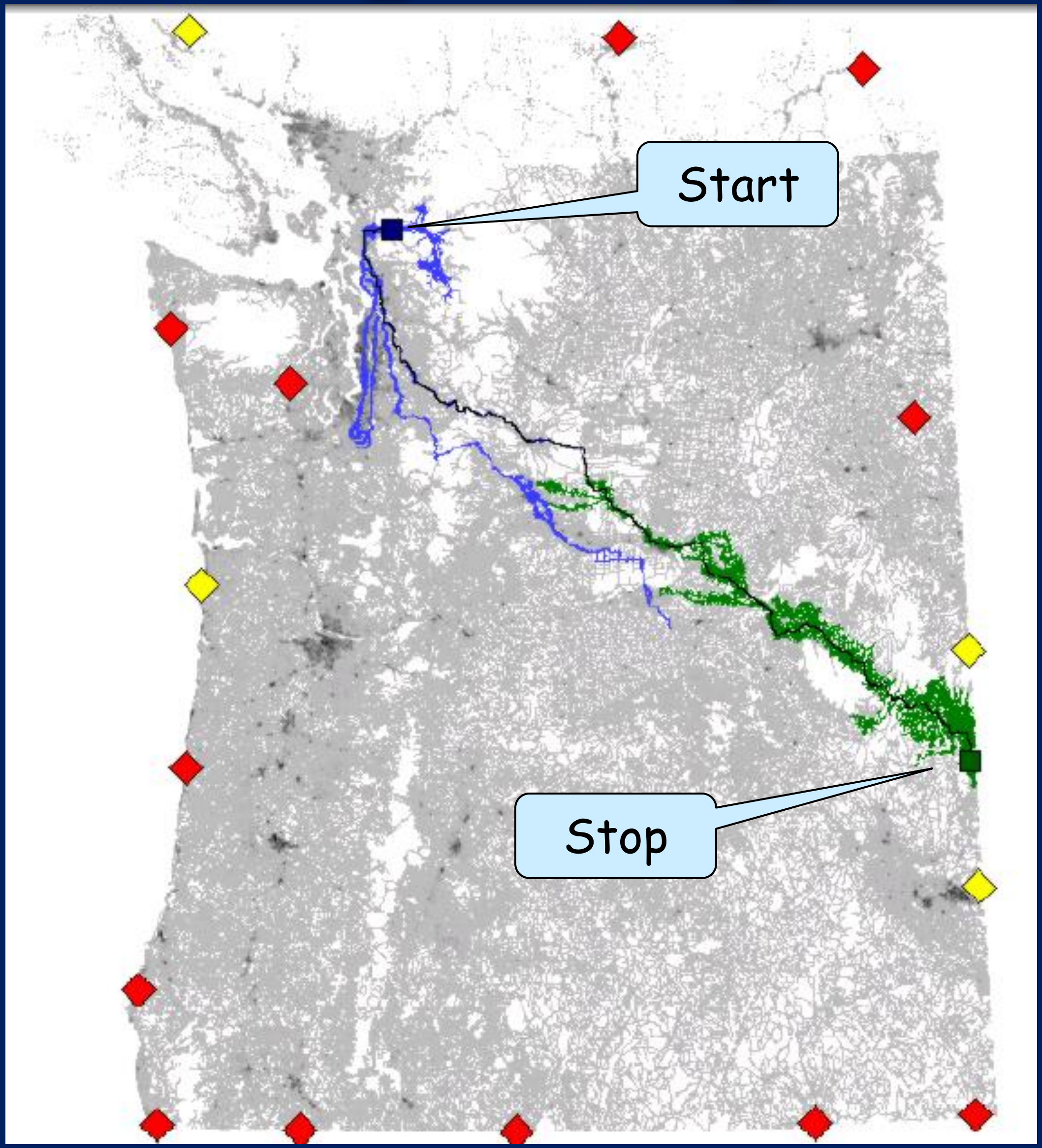
# Wanted: better lower bounds

- Good lower bound if
  - Destination is near the landmark
  - Or the landmark is "beyond" destination
- How do we find a landmark that is "beyond" the destination? Just picking one may sometimes be bad. So use several!
- Easy to use lots of landmarks at once:
  - $L(V) = \max( L_1(V), L_2(V), \ldots, L_n(V) )$
  - The max of a set of lower bounds is still a lower bound

# Now we're cooking with gas!



Much, much better!



# Results

Northwest (1.6M vertices), random queries, 16 landmarks.

| method | preprocessing minutes | preprocessing MB | query avgscan | query maxscan | query ms |
|---|---|---|---|---|---|
| Bidirectional Dijkstra | — | 28 | 518 723 | 1 197 607 | 340.74 |
| ALT | 4 | 132 | 16 276 | 150 389 | 12.05 |

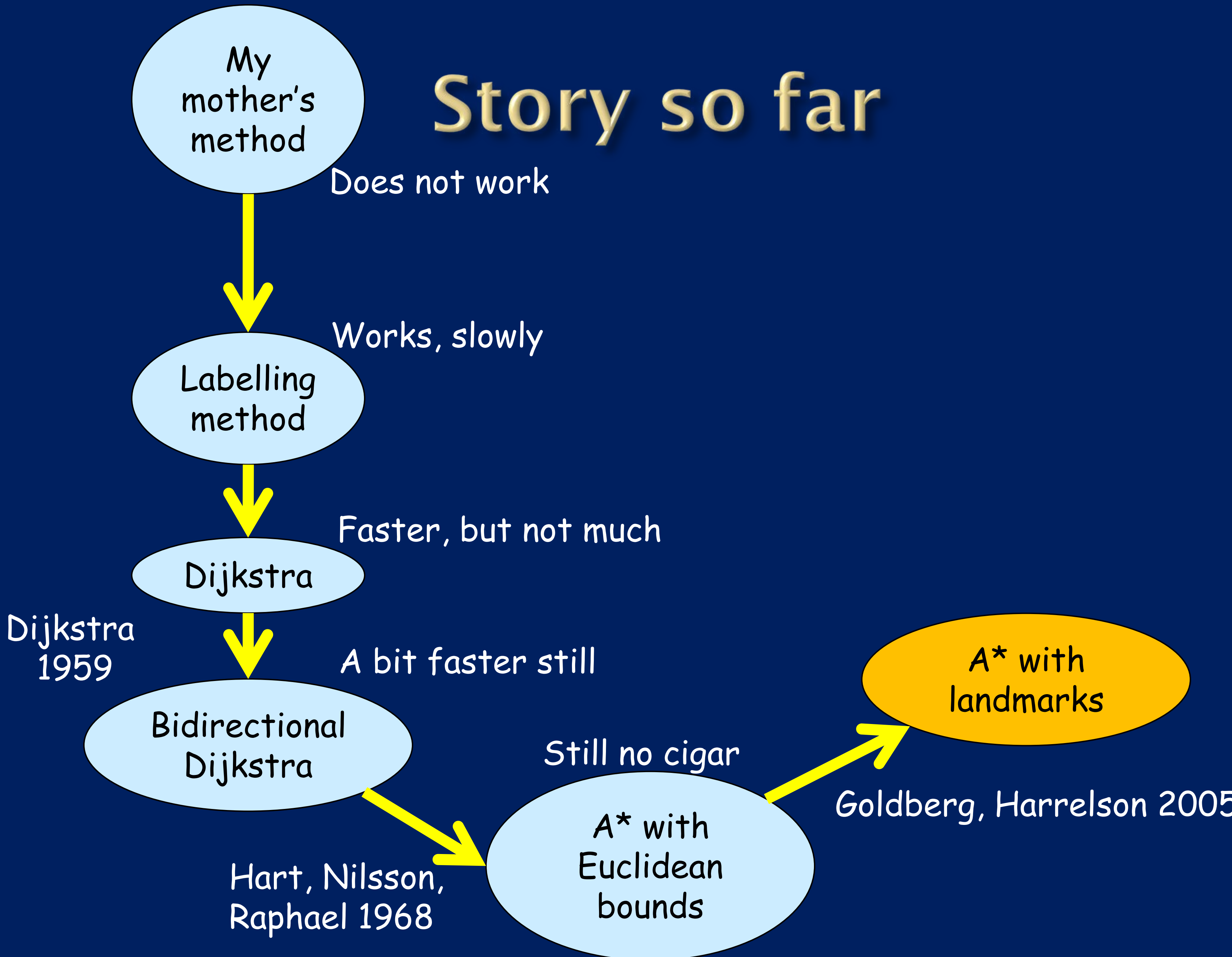
Landmarks

Average number of vertices scanned

Maximum number of vertices scanned

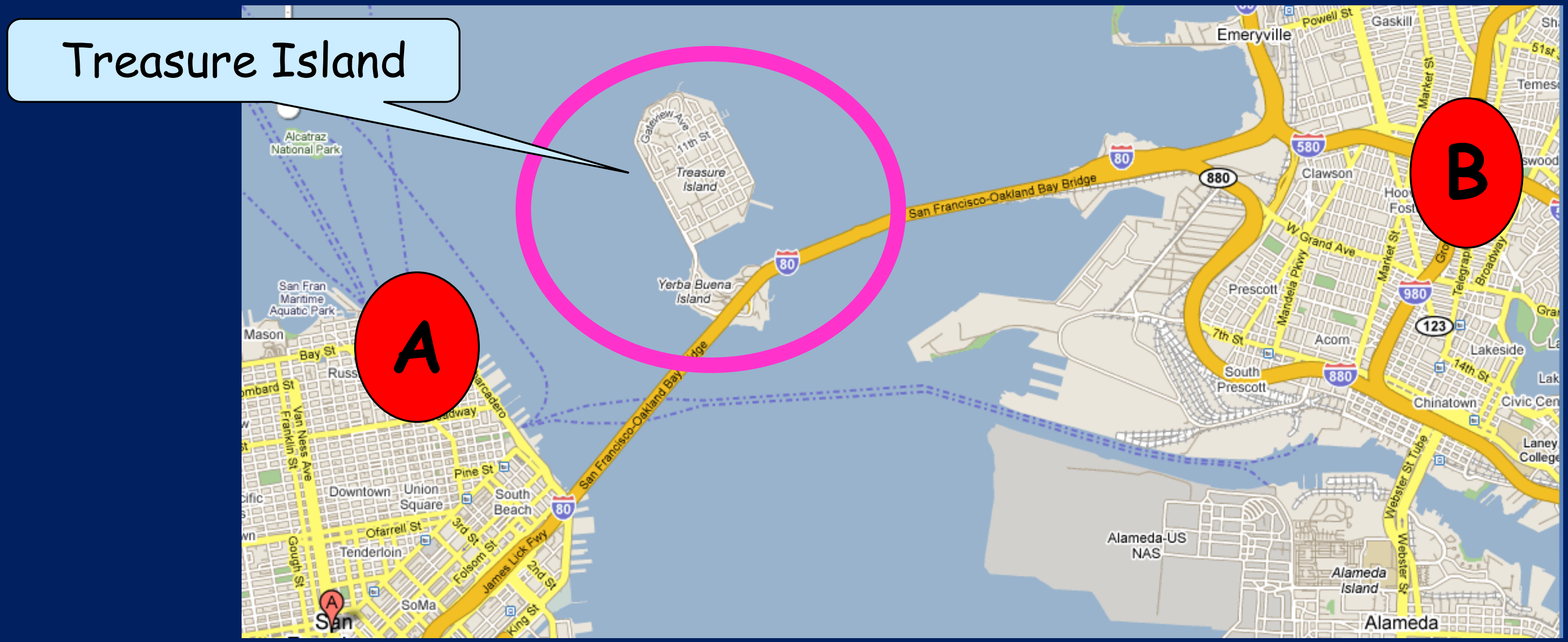
30 times faster
4 times as much memory used

# Story so far



My mother's method
Does not work
Works, slowly
Labelling method
Faster, but not much
Dijkstra
Dijkstra 1959
A bit faster still
Bidirectional Dijkstra
Still no cigar
A* with Euclidean bounds
Hart, Nilsson, Raphael 1968
A* with landmarks
Goldberg, Harrelson 2005

# Can we do better?



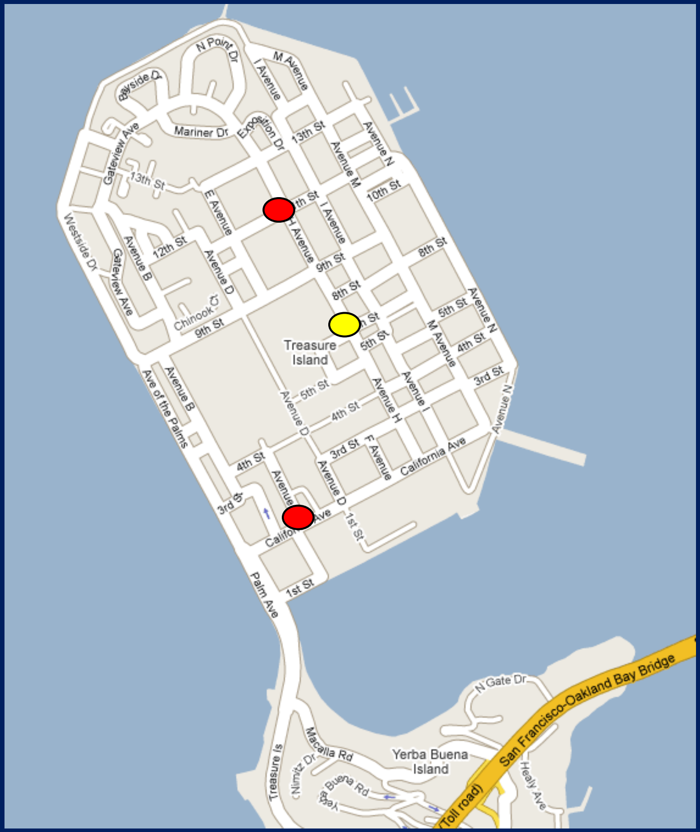

- Intuition: no point in exploring Treasure Island (at all) when finding path A-B
- Why not?
- Because

No vertex in Treasure Island is "on the way to anywhere else"

# Reach

No vertex in Treasure Island is "on the way to anywhere else"

- What does it mean to say "V is on the way to somewhere?"
- Obviously **IS** on the way from to !
- The "**reach**" of V is
  - big if V is on the way between far-away places
  - small if V is only on the way between nearby places
- "on the way" means "on the shortest path"

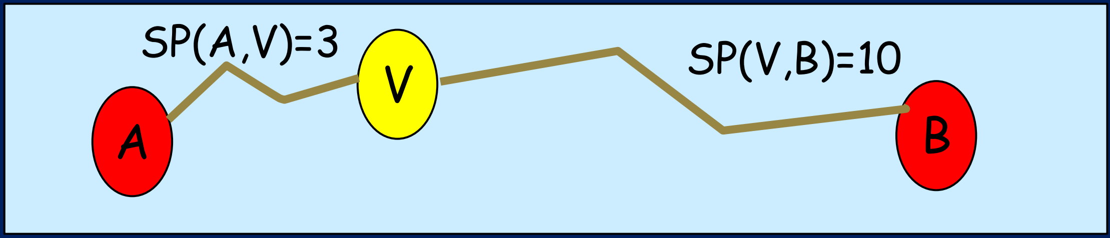


# Reach

- The "**reach**" of V is
  - big if V is on the way between far-away places
  - small if V is only on the way between nearby places

Vertices with small reach are not on the way to anywhere

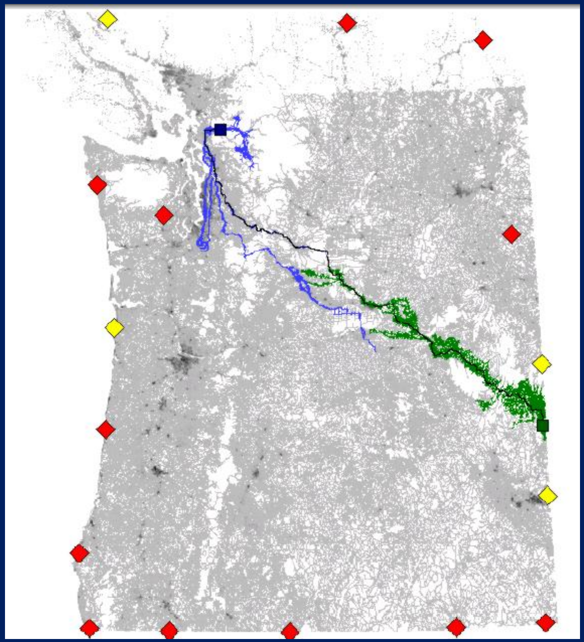
- Small if **all** shortest paths involving V have **one end** near V
- Reach(V) = max { min(SP(A,V), SP(V,B)) | A,B are vertices, and V is on shortest path A-B }



# Pruning A*

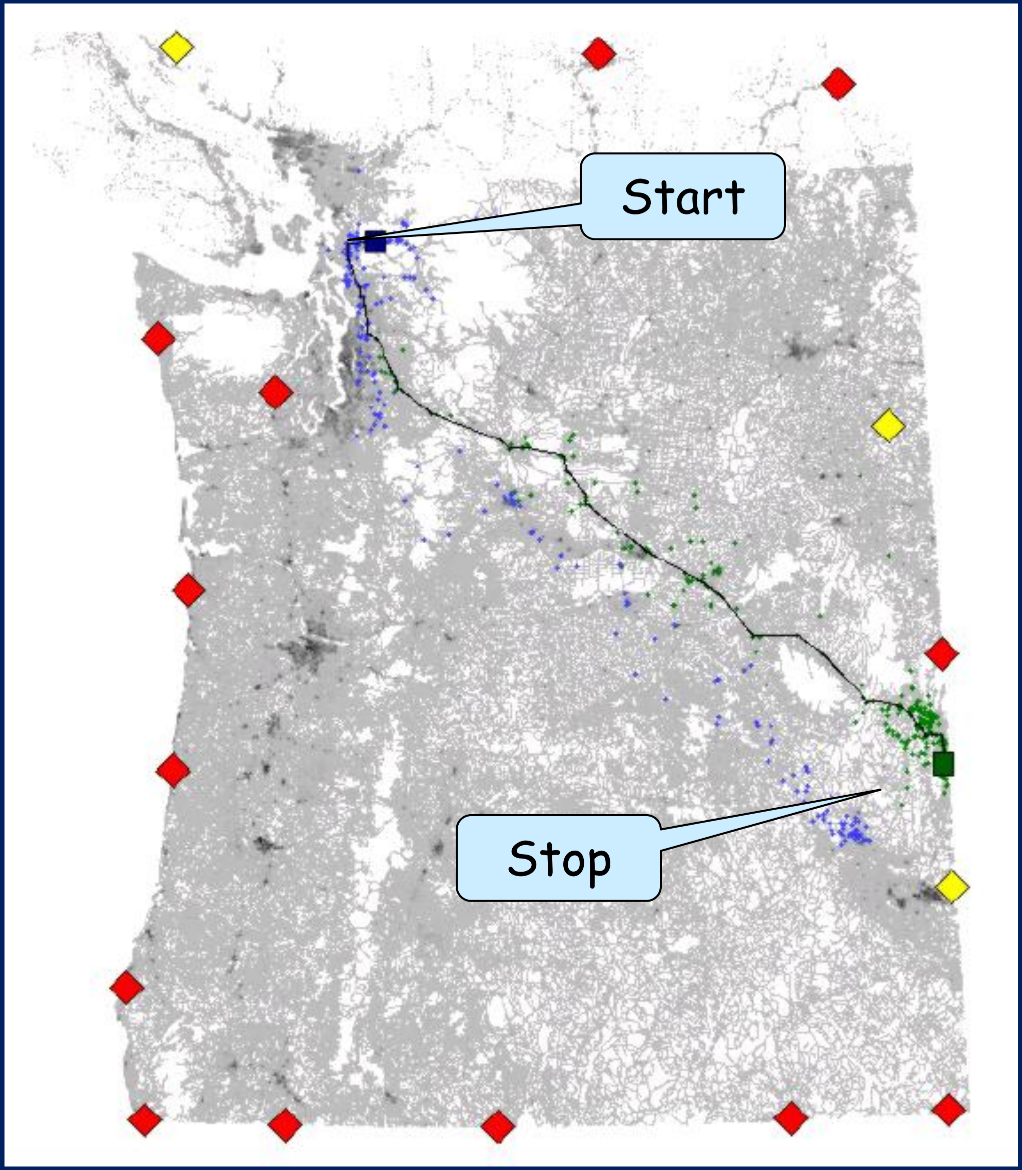
- Choose Yellow(n) vertex V with smallest **n+L(V)**, **unless Reach(V)<n and Reach(V)<L(V)**
- **Why?**
  - One end of any shortest path going through V must be within Reach(V) of V [defn of Reach]
  - But if Reach(V) < n then A is not within Reach(V) of V (since SP(A,V) = n)
  - And if Reach(V) < L(V) then B is not within Reach(V) of V (since L(V) ≤ SP(V,B) )
  - So neither A nor B are within Reach(V) of V
  - So the shortest path A-B cannot go through V

# Those side alleys make a difference!



ALT (A* + landmarks)

A* + landmarks + Reach



# Results

Northwest (1.6M vertices), random queries, 16 landmarks.

| method | preprocessing | | query | | |
|---|---|---|---|---|---|
| | minutes | MB | avgscan | maxscan | ms |
| Bidirectional Dijkstra | — | 28 | 518 723 | 1 197 607 | 340.74 |
| ALT | 4 | 132 | 16 276 | 150 389 | 12.05 |
| Reach | 1 100 | 34 | 53 888 | 106 288 | 30.61 |
| Reach+Short | 17 | 100 | 2 804 | 5 877 | 2.39 |
| Reach+Short+ALT | 21 | 204 | 367 | 1 513 | 0.73 |

50x fewer vertices than ALT

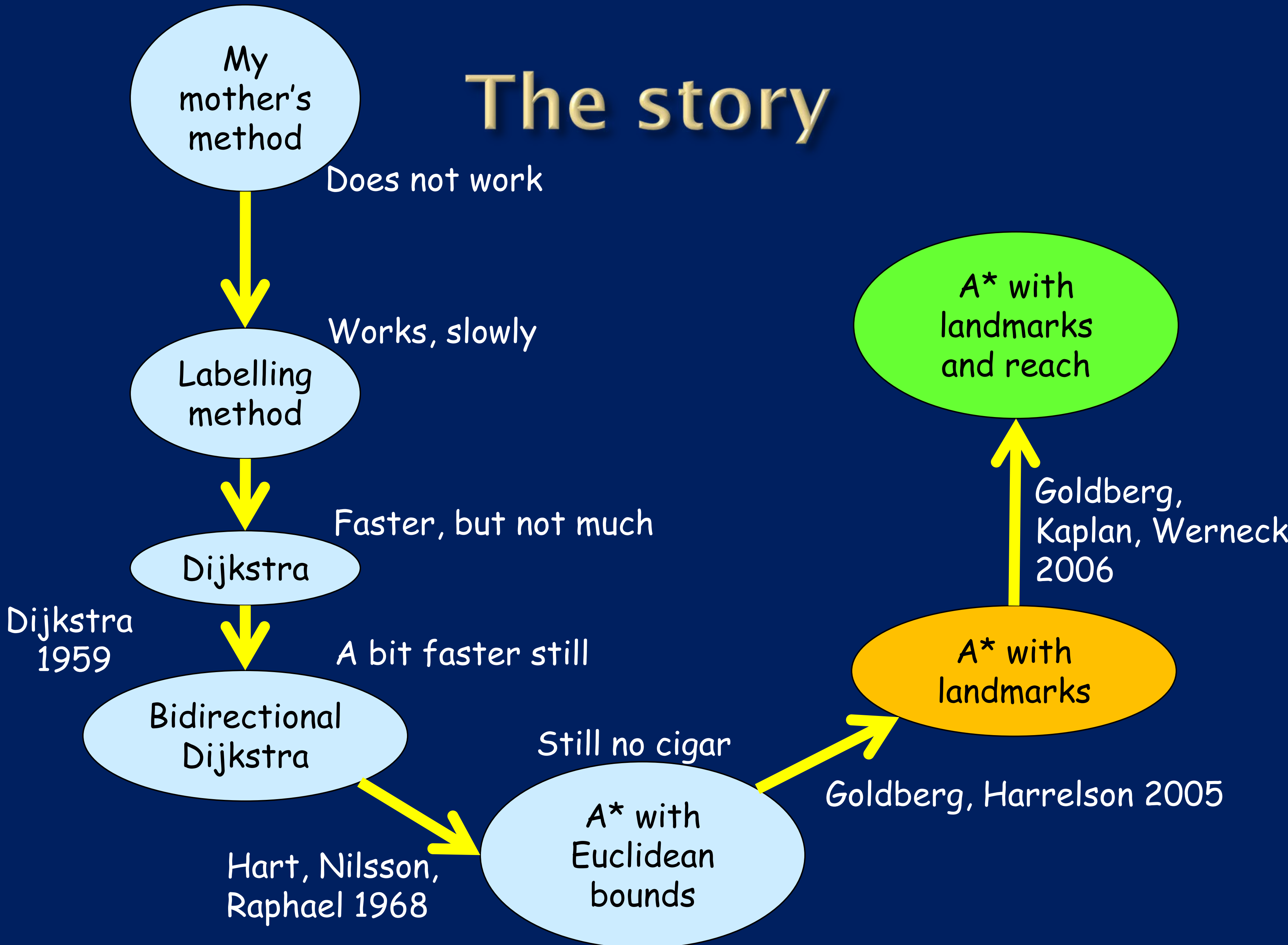
20x faster

# Europe!

Europe: 18M vertices, 43M arcs, time metric, random queries.

| method | preprocessing | | query | | |
|---|---|---|---|---|---|
| | min | KB | avgscan | maxscan | ms |
| Dijkstra | — | 393 | 8 984 289 | — | 4 365.81 |
| ALT(16) | 12.5 | 1 597 | 82 348 | 993 015 | 120.09 |
| Reach | impractical | | | | |
| Reach+Short | 45.1 | 648 | 4 371 | 8 486 | 3.06 |
| Reach+Short+ALT(16,1) | 57.7 | 1 869 | 714 | 3 387 | 0.89 |
| Reach+Short+ALT(64,16) | 102.6 | 1 037 | 610 | 2 998 | 0.91 |

# The story



My mother's method
Does not work
Labelling method
Works, slowly
Dijkstra
Faster, but not much
Dijkstra 1959
A bit faster still
Bidirectional Dijkstra
Hart, Nilsson, Raphael 1968
Still no cigar
A* with Euclidean bounds
Goldberg, Harrelson 2005
A* with landmarks
Goldberg, Kaplan, Werneck 2006
A* with landmarks and reach

# Summary

- A problem that spans 50+ years, still active today
- ONE algorithm, with a variety of "choose the next vertex to work on" heuristics
- An ounce of cunning is worth a tonne of brute force: fantastic gains from simple insights
- Abstraction is the key:
  - Boil away the detail to leave an abstract problem
  - Clever algorithms underpinned by formal reasoning
- Computer science is a lot more than programming!