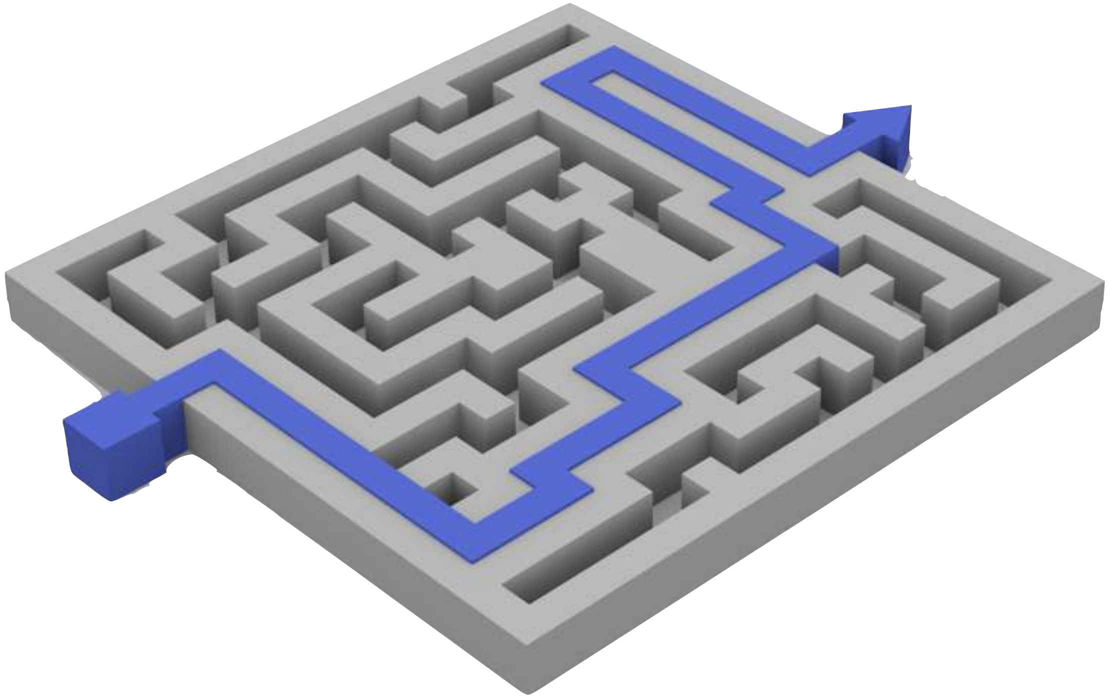
Discrete motion planning

TSFS12: Autonomous Vehicles –planning, control, and learning systems

Lecture 2: Erik Frisk <erik.frisk@liu.se>

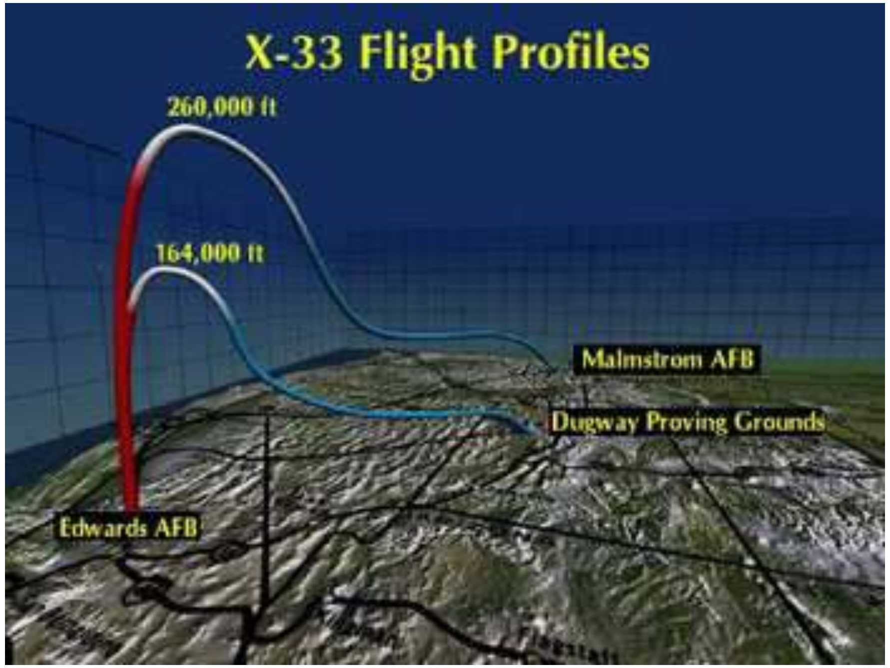
LINKÖPINGS UNIVERSITET

Motion planning, from discrete problems …



… to continuous, nonholonomic, systems with inertia

NASA/Lockheed Martin X-33



Re-entry trajectory

From "*Planning Algorithms*", S. LaValle, 2006.

Motion planning and discrete graph search

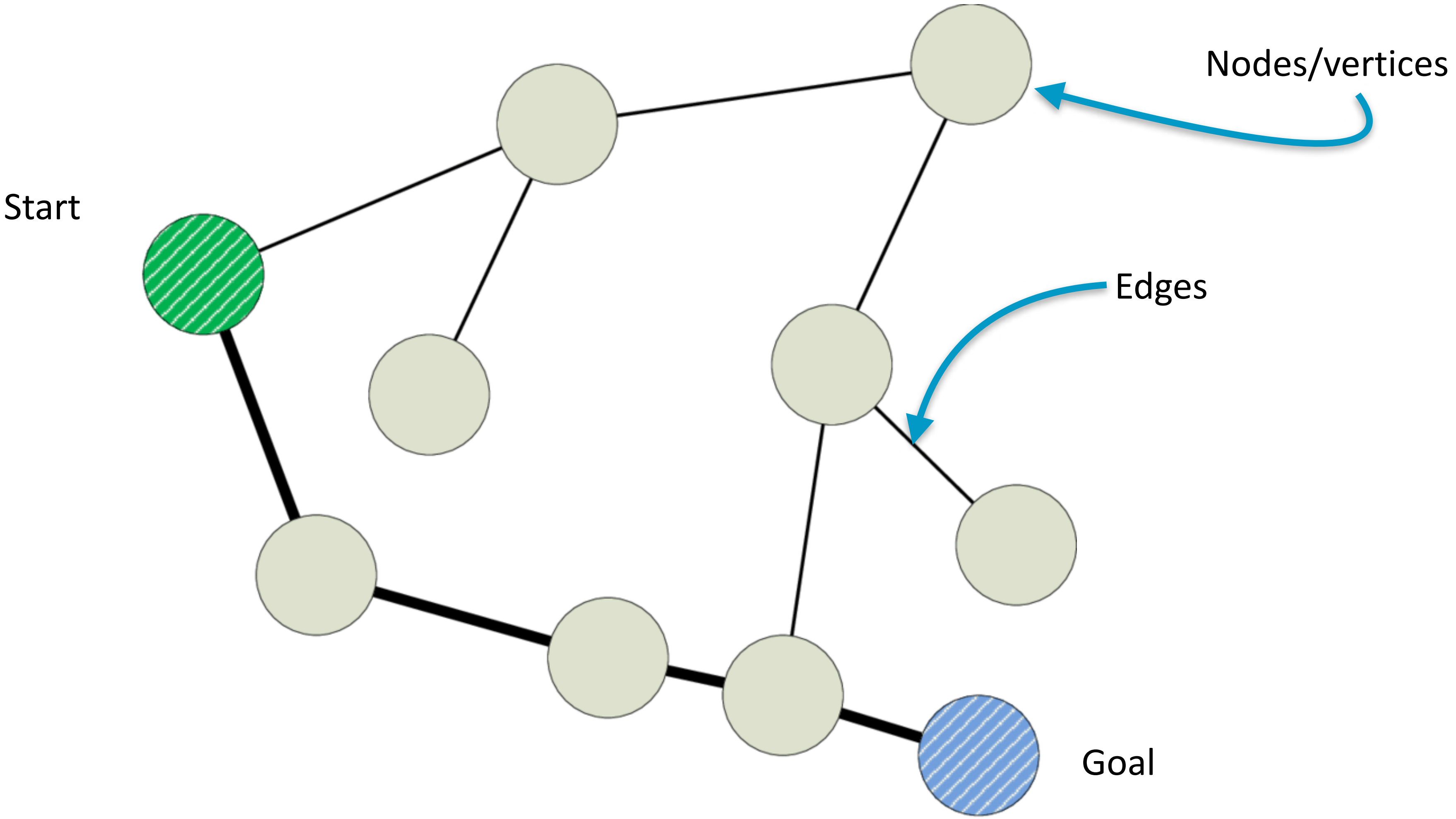
- Graph search algorithms are very useful for planning motion and trajectories for autonomous vehicles
- But, robots do not move on a graph?
  - Discretize
  - Use graph search as a component in a continuous planner, for example in so called lattice planners (Lecture 4)
- This lecture will focus on the graph search problem and introduce fundamental algorithms
- These algorithms are the topic of hand-in 1, and will be used also in hand-in 2.

Scope of this lecture

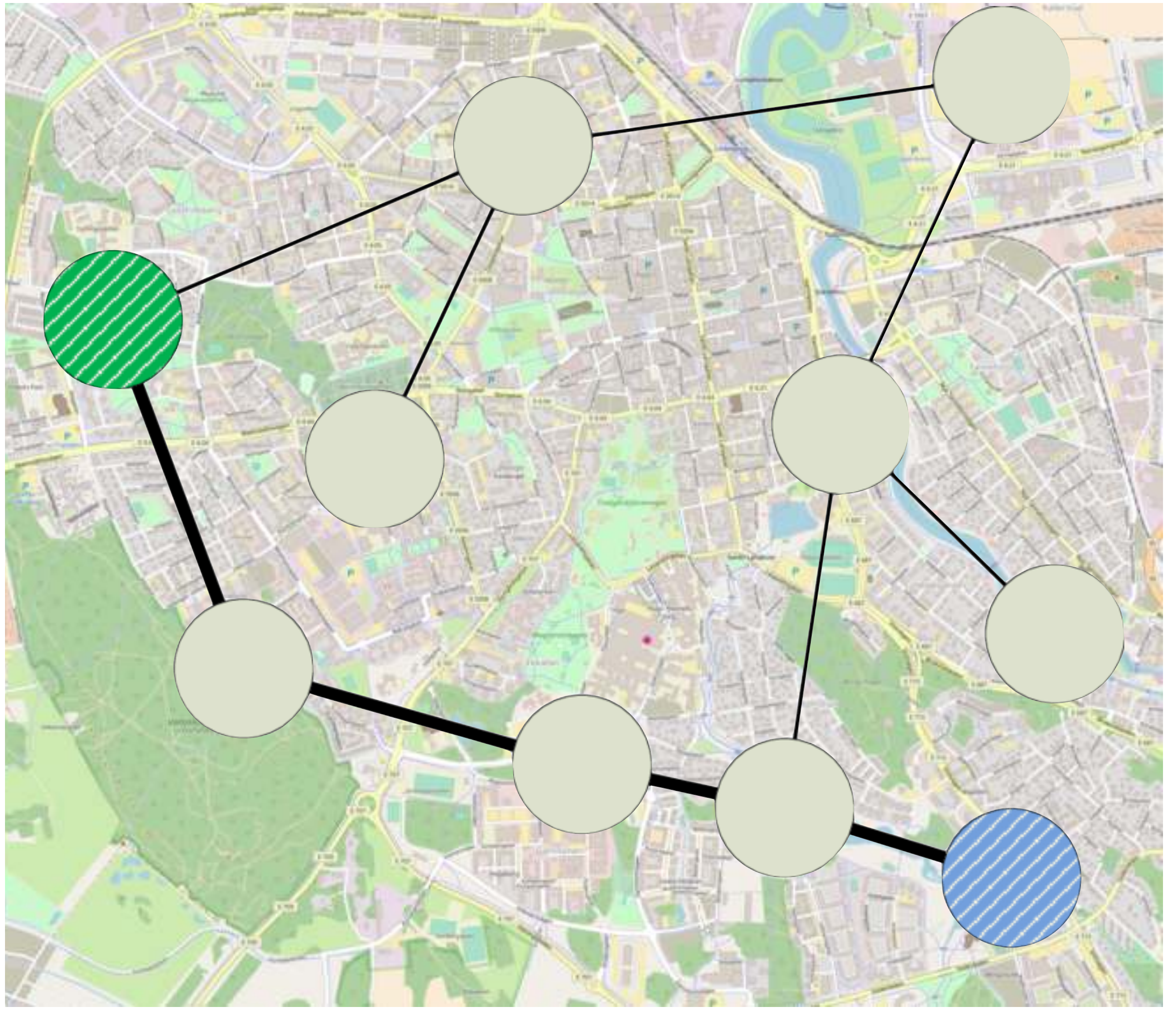
- Formalization of a planning problem as a graph search problem
- Main algorithms for graph search
    - Dijkstra’s algorithm
    - A*
- Properties of heuristics in A* to ensure optimality and efficiency
- Introduction to any-time planning using A* — ARA*

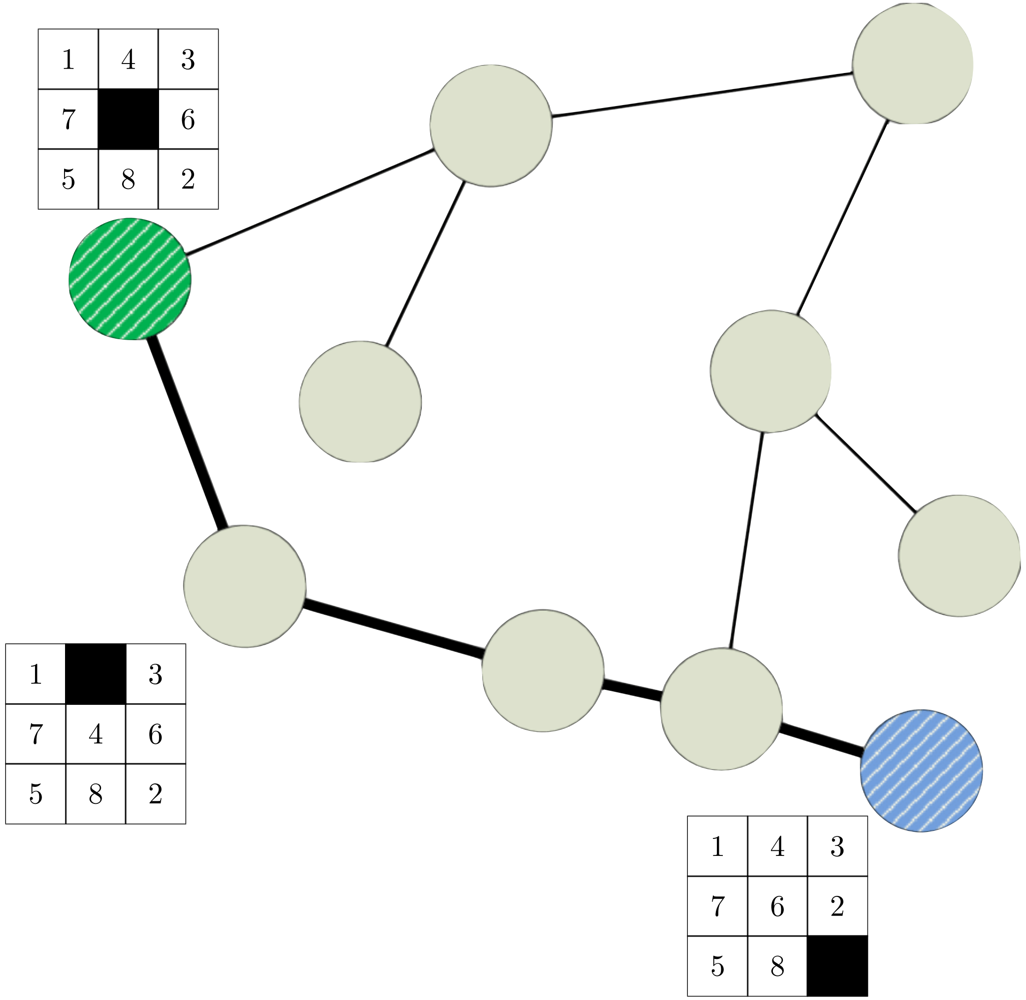
Graphs and discrete planning problems

Solving problems by searching in a graph



A node can represent any state in a search space





Formulating a planning problem as a graph search

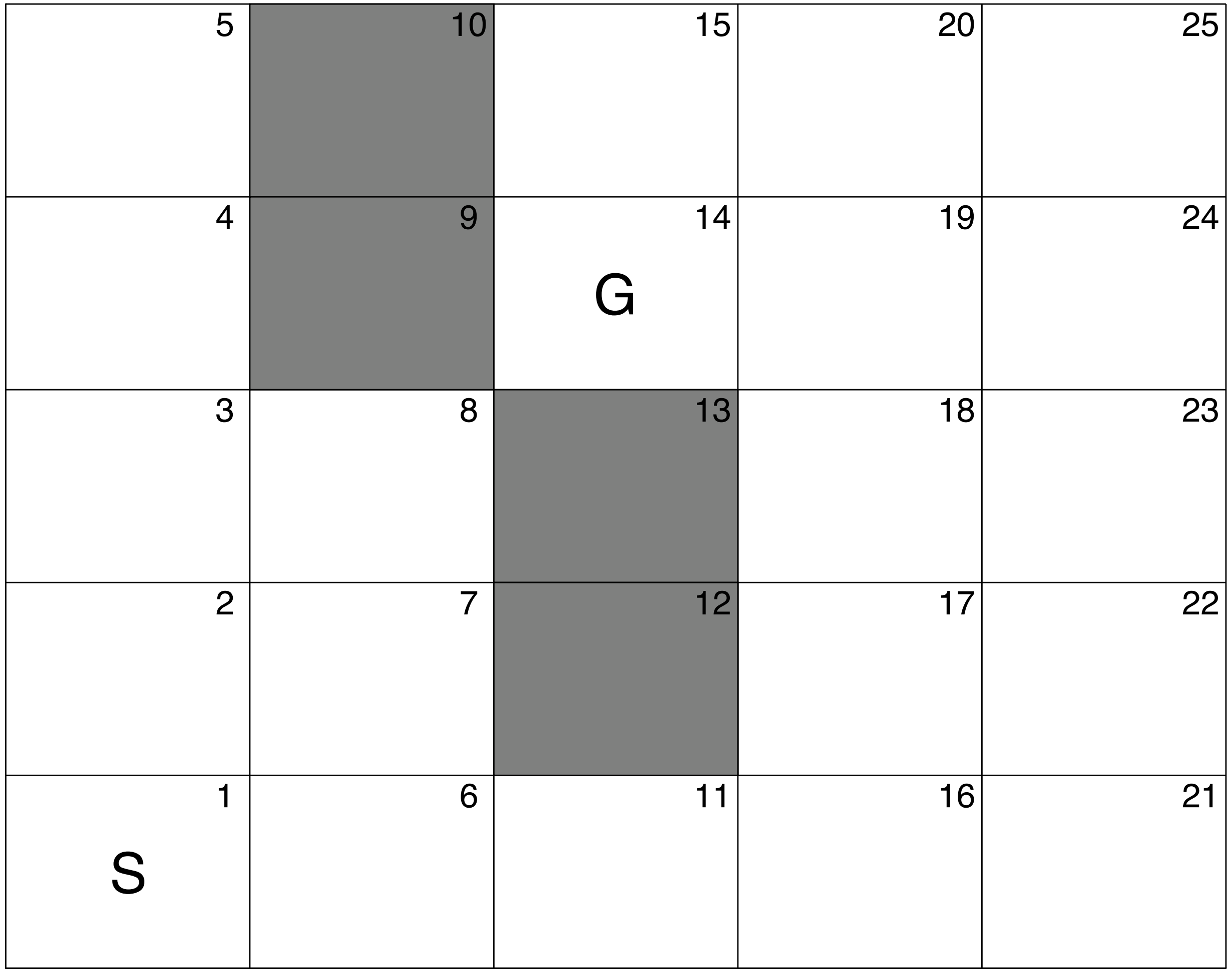
- Problem solving is sometimes well formulated as graph search problems
- Formulation of graph search problem requires
  - State-space $\mathscr{X}$
  - For each state $x \in \mathscr{X}$ there is an action space $\mathscr{U}(x)$
  - A state-transition function $x' = f(x, u) \in \mathscr{X}$ for each $u \in \mathscr{U}(x)$
- Initial state x_I and goal state x_G

Definition of a graph search problem

$$\mathcal{X} = \{1, \ldots, 25\}$$
$$U(x) \subseteq \{\mathrm{Up}, \mathrm{Down}, \mathrm{Left}, \mathrm{Right}\}$$
$$x_I = 1, \quad x_G = 14$$
$$x' = f(x, u) \in \mathcal{X}$$

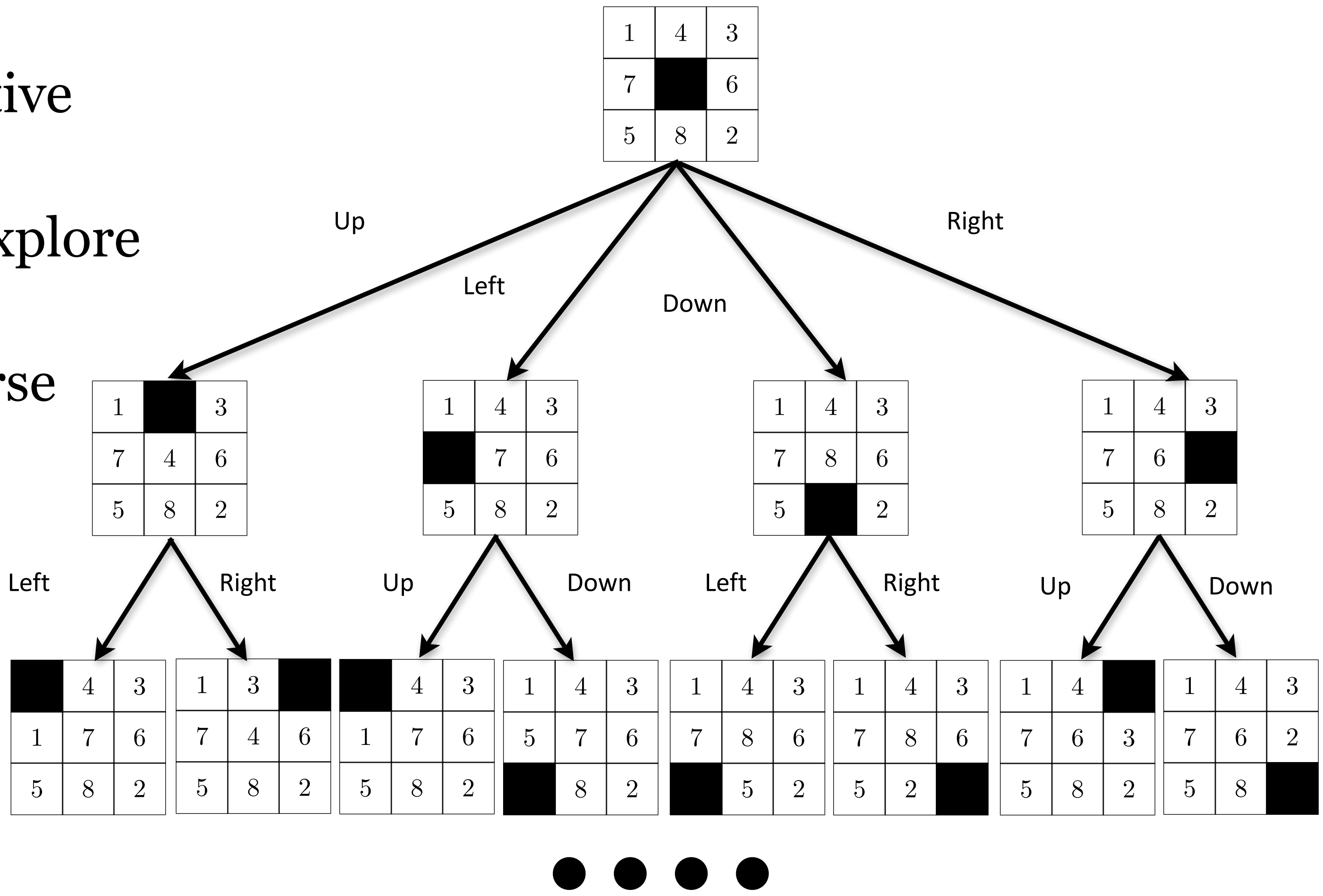
$$U(8) = \{\mathrm{Left}, \mathrm{Down}\}$$

$$f(17, u) = \begin{cases} 18 & u = \mathrm{Up} \\ 22 & u = \mathrm{Right} \\ 16 & u = \mathrm{Down} \end{cases}$$



State-space and search tree

- Naïve solution; exhaustive search
- Build search tree and explore until solution is found
- Different ways to traverse the tree
    - depth first
    - breadth first
    - …



Queues

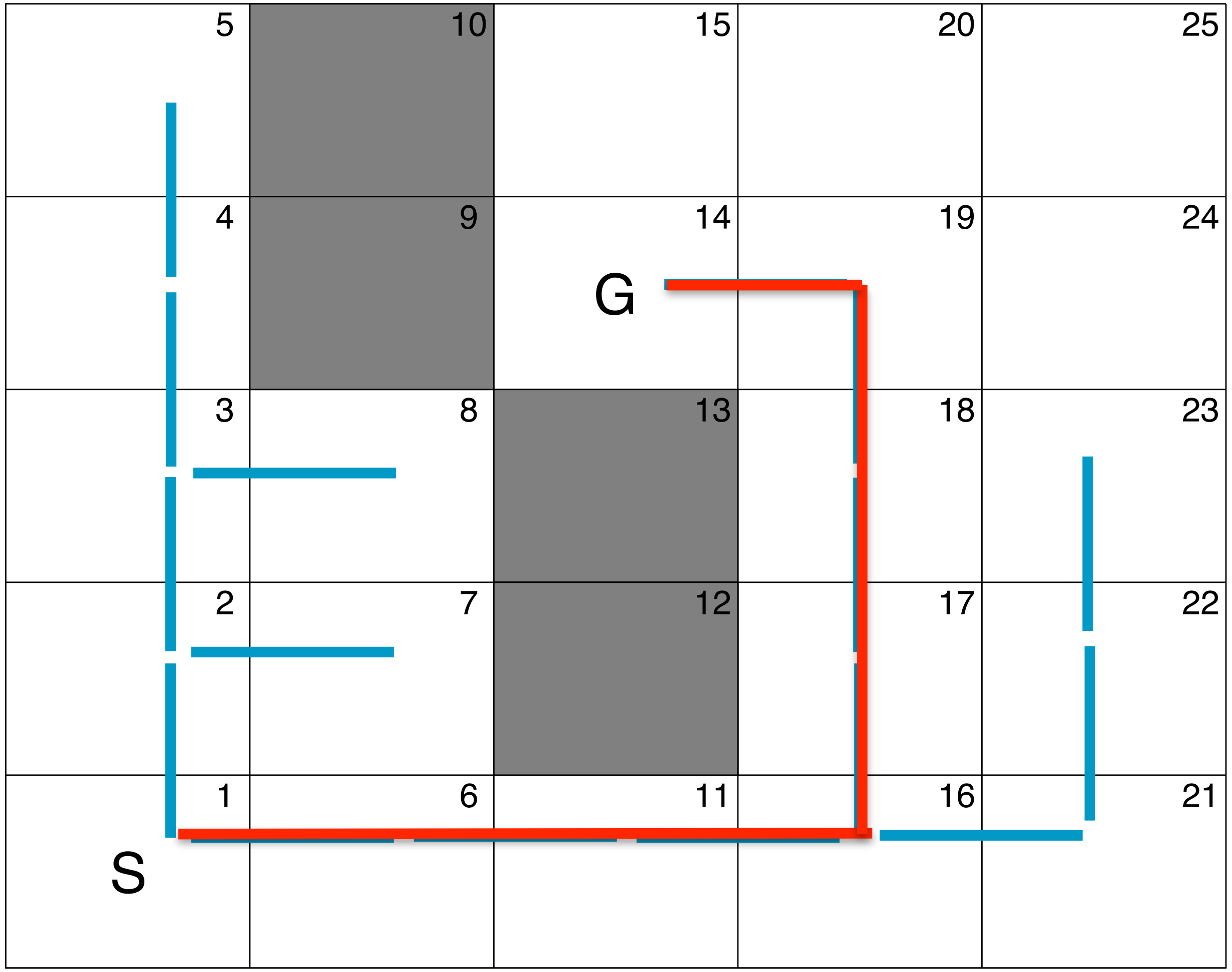
- A queue is a data structure where you can
  - Push (or insert) elements on the queue
  - Pop (or remove) elements from the queue
- Very useful for describing and implementing search algorithms
- Three different queues will be used
  - FIFO - First In First Out
  - LIFO - Last In First Out
  - Priority Queue - assign priority to each element
    ≈ keep the queue always sorted (but efficiently)
    Will return to this queue later.

General forward search (and keep track of visited nodes)

- During the search, define a mapping
  $x' = \text{previous}(x)$
- Keeps track of paths, node x is predecessor of node x'
- Keeps track of which nodes that are visited
- Depth first - LIFO queue
- Breadth First - FIFO queue

```
function ForwardSearch:
    Q.insert(x_I)

    while Q ≠ ∅
        x = Q.pop()
        if x = x_G
            return SUCCESS

        for u ∈ U(x)
            x' = f(x, u)
            if no previous(x')
                previous(x') = x
                Q.insert(x')

    return FAILURE
```

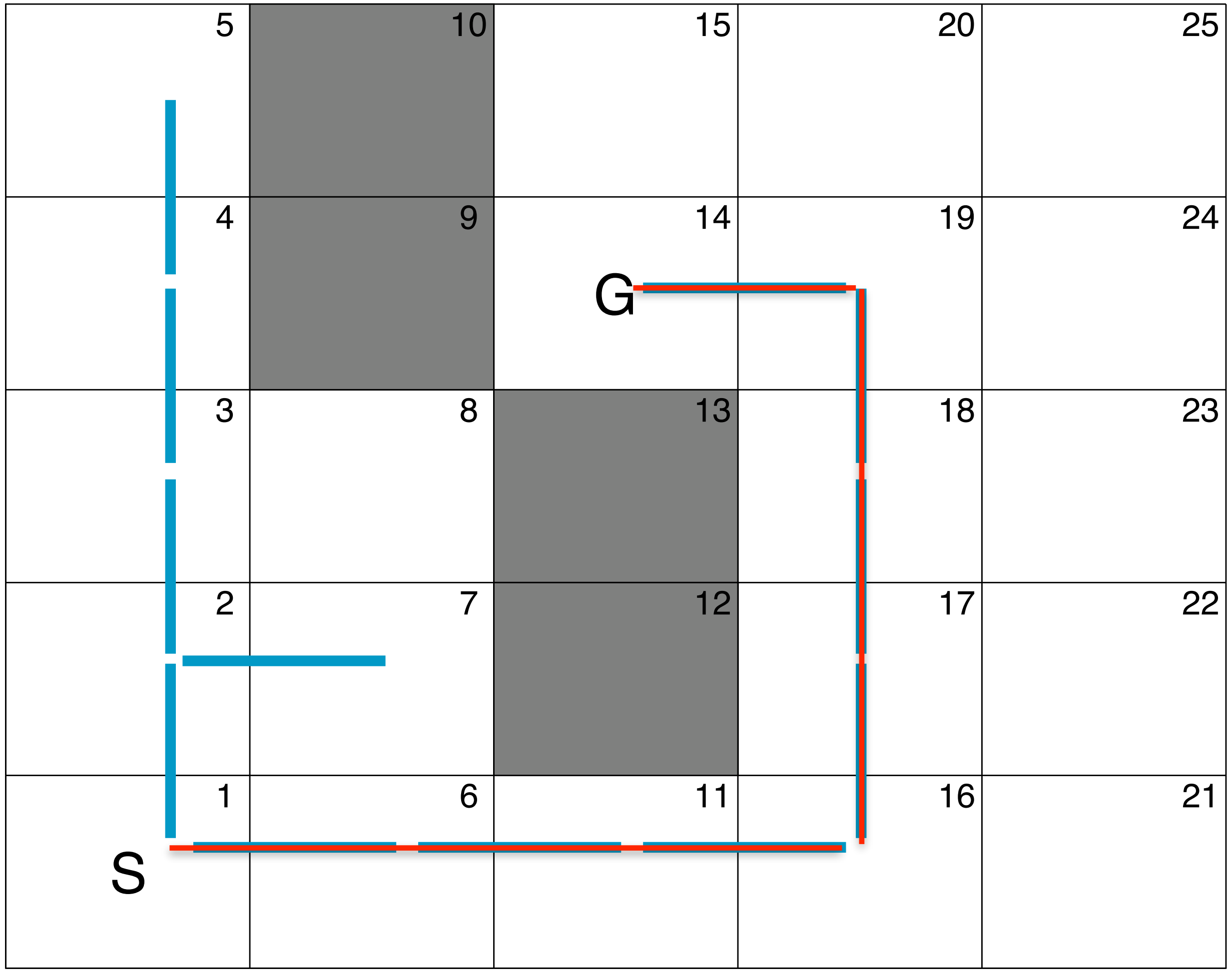
Breadth First search - FIFO queue



Node	Previous
1	-
2	1
3	2
4	3
5	4
6	1
7	2
8	3
9	
10	
11	6
12	
13	
14	19
15	
16	11
17	16
18	17
19	18
20	
21	16
22	21
23	22
24	
25	

LINKÖPINGS UNIVERSITET

Depth First search - LIFO queue



Generate path from visited mapping

- The mapping

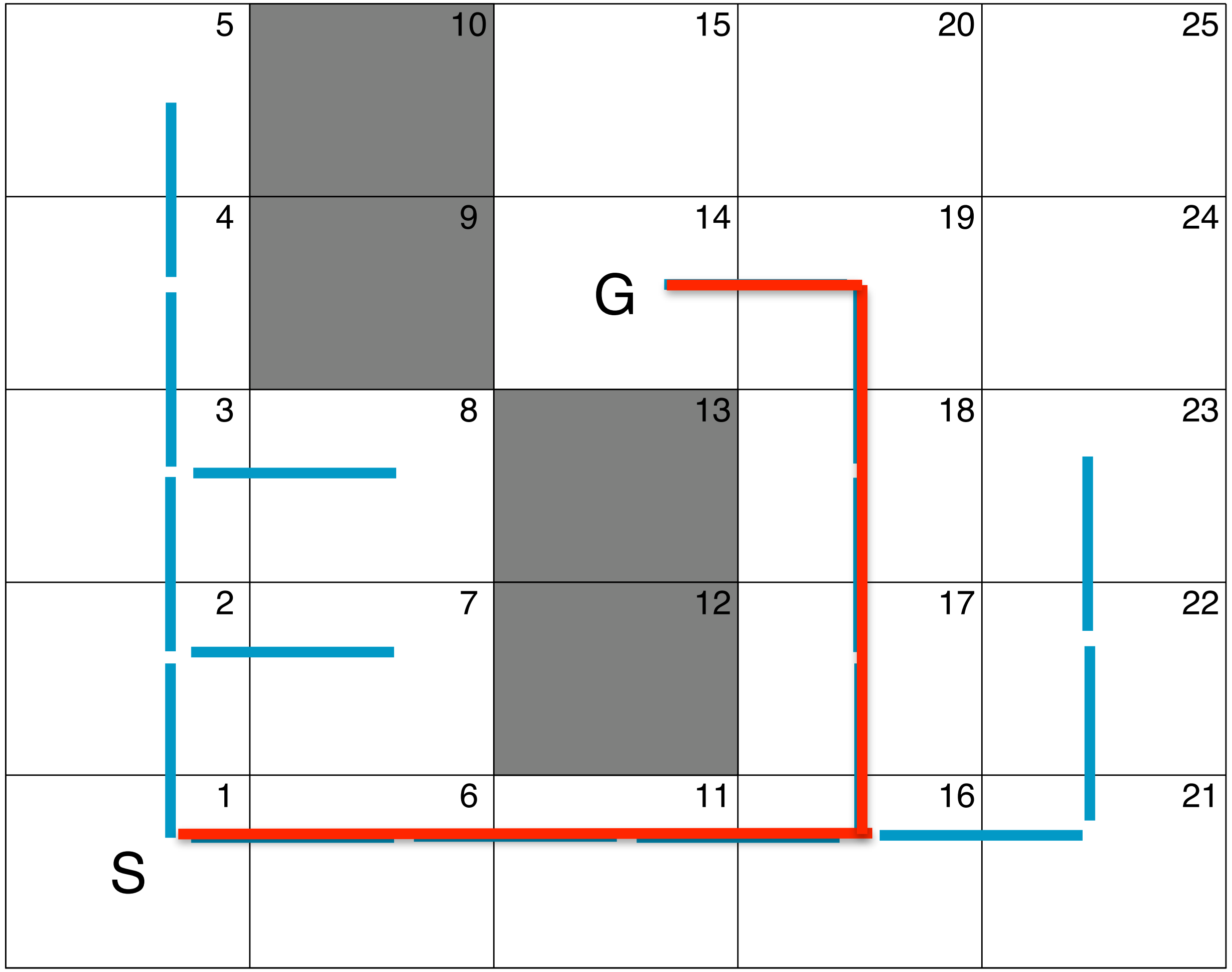
$$x' = \text{previous}(x)$$

defines the path.

- Node x is the predecessor of node x'
- Backtracking from goal to start then gives the path

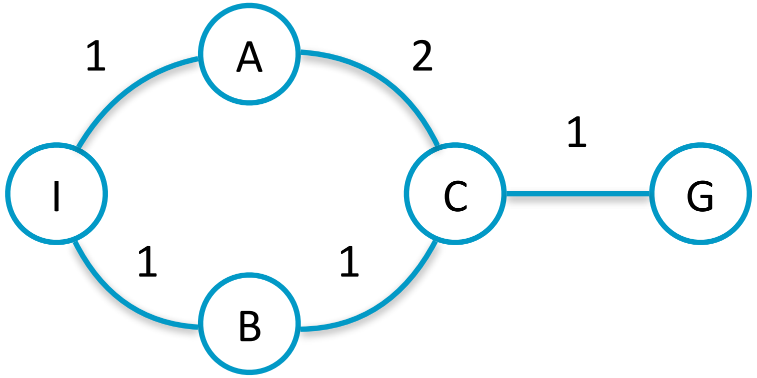
```
function Backtrack(visited, source, goal):
  if found
    p = ∅
    u = goal
    while previous[u] ≠ start
      insert u at the beginning of p
      u = previous[u]
    insert u at the beginning of p
```

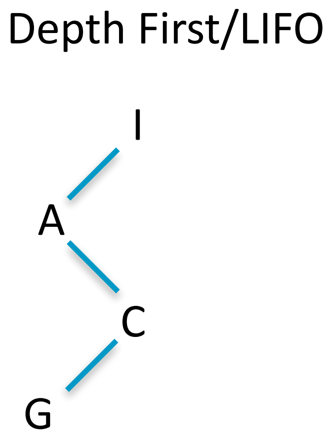
Backtrack generated path

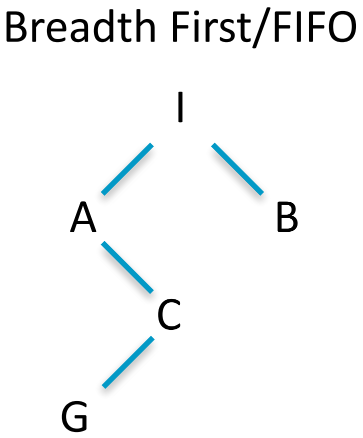


Node	Previous
1	-
2	1
3	2
4	3
5	4
6	1
7	2
8	3
9	
10	
11	6
12	
13	
14	19
15	
16	11
17	16
18	17
19	18
20	
21	16
22	21
23	22
24	
25	

What about quality of plan?







- Clearly neither depth-first nor breadth-first finds the shortest path in the graph
- Not surprising since there is no notion of distance/stage-cost in the search
- Next step is to find shortest paths …

Dijkstra's algorithm - finding shortest path

Dijkstra's algorithm

- Well known algorithm, first published in the 1950's
- Computes, not only the shortest path between two nodes, but the shortest path between a source node and *all* other nodes; *shortest path tree*
- Idea: keep track of cost-to-come for each visited node, and explore the tree search prioritized by cost-to-come
- Use Priority Queues instead of FIFO/LIFO

Priority Queue

- You can insert and pop (element, priority) pairs
- Here priority is typically path cost (length/time)
- Operations (for min-priority queue)
  - insert(element, priority) - insert pair into the queue
  - pop() - returns element and priority corresponding to the *lowest* priority
  - decrease_key(element, priority) - change priority for an element
- In general, you can decrease an elements priority by pushing it again
  - This is not strictly needed, I will come back to this; lazy delete
- Insert and pop are no longer constant time operations, typically $\mathcal{O}(\log n)$ for implementations based on a data-structure called heap

Dijkstra's algorithm

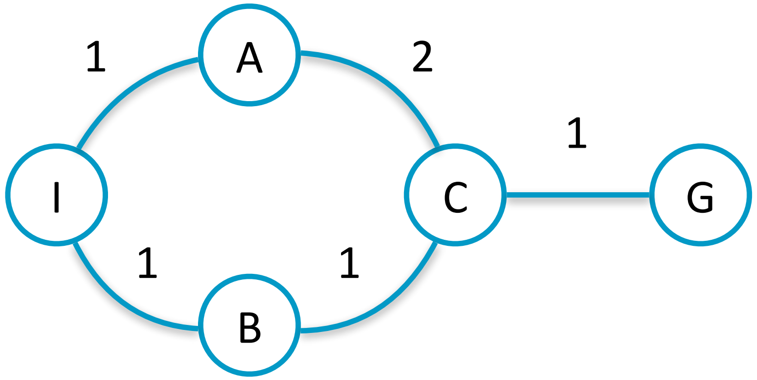
- $d(x, x')$ cost (~ length) to go from node x to x'
- During search, update mapping

  $$C(x)$$

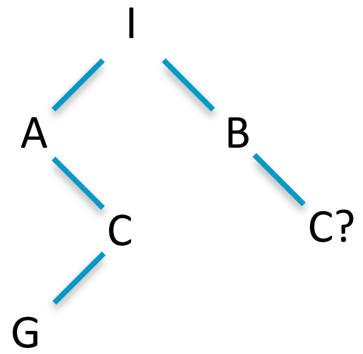
  that keeps track of *cost-to-come* to node x.
- Use a priority queue, with *cost-to-come* as priority to explore the shortest paths first
- Modify the search to rewire in case a cheaper path is found

```
function Dijkstra:
    C(x_I) = 0
    Q.insert(x_I, C(x_I))

    while Q ≠ ∅
        x = Q.pop()
        if x = x_G
            return SUCCESS

        for u ∈ U(x)
            x' = f(x, u)
            if no previous(x') or
                C(x') > C(x) + d(x, x')
                previous(x') = x
                C(x') = C(x) + d(x, x')
                Q.insert(x', C(x'))

    return FAILURE
```

Dijkstra on the small graph



Breadth First/FIFO



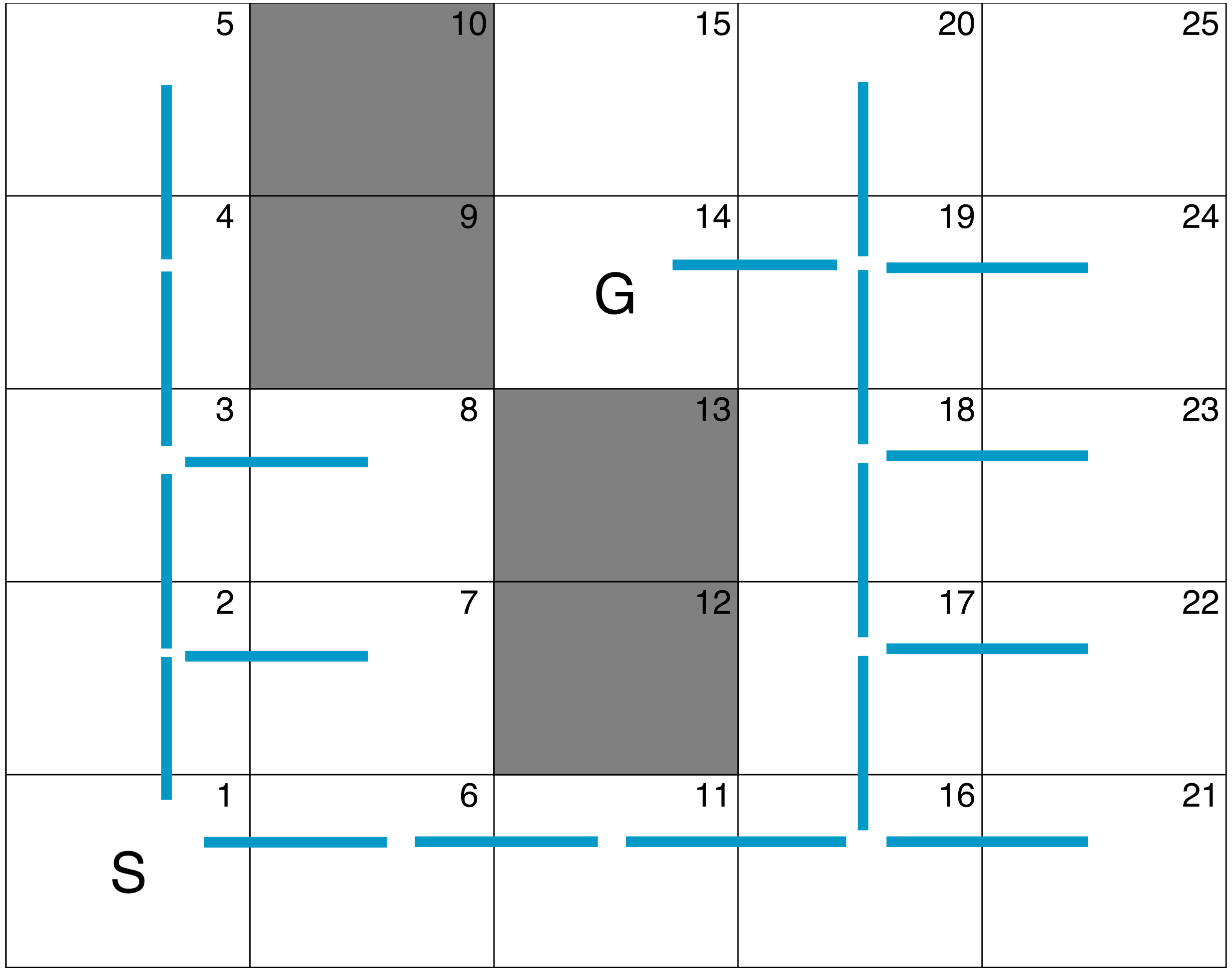

```
Start:    I
  Prio    0

Pop I:    A B
  Prio    1 1

Pop A:    B C
  Prio    1 3

Pop B:    C   alt. C C
  Prio    2        2 3

Pop C:    G   alt G C
  Prio    3       3 3

Pop G:   Goal
```

Previous(x)

I	-
A	I
B	I
C	~~A~~ B
G	C

Here path to C is rewired

Dijkstra's algorithm




```
Start:   1
  Prio   0

Pop 1:   2 6
  Prio   1 1

Pop 2:   6 3 7
  Prio   1 2 2

Pop 6:   3 7 11
  Prio   2 2 2

Pop 3:   7 11 4 8
  Prio   2  2 3 3

Pop 7:  11 4 8
  Prio   2 3 3

Pop 11:  4 8 16
  Prio   3 3  3

Pop 4:   8 16 5
  Prio   3  3 4

Pop 8:  16 5
  Prio   3 4

Pop 16:  5 17 21
  Prio   4  4  4

Pop 5:  17 21
  Prio   4  4

Pop 17: 21 18 22
  Prio   4  5  5

Pop 21: 18 22
  Prio   5  5

Pop 18: 22 19 23
  Prio   5  6  6

Pop 22: 19 23
  Prio   6  6

Pop 19: 23 14 20 24
  Prio   6  7  7  7

Pop 23: 14 20 24
  Prio   7  7  7

Pop 14: Goal
```

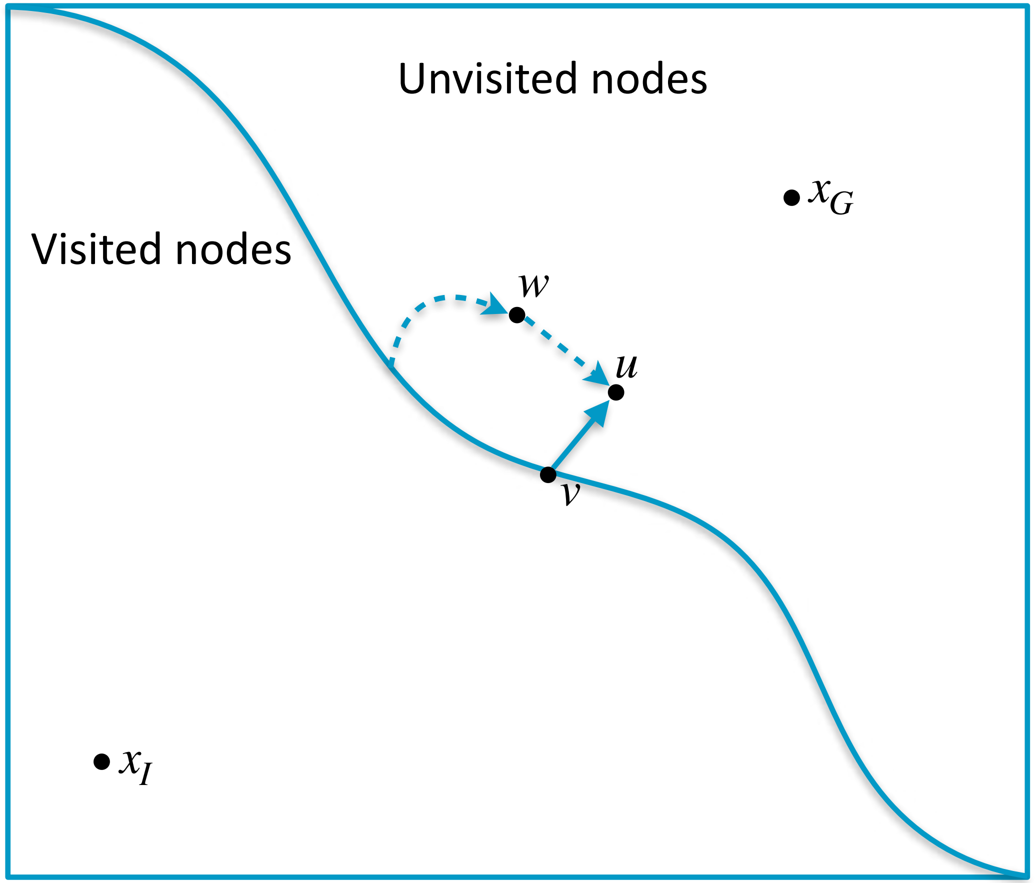
Example, backtracking path

- During search, two mappings are updated
  - Previous(x) - keep track of parent node
  - Cost(x) - current cost to come
- Using Visited(x), backtracking gives the resulting path
  - 14 - 19 - 18 - 17 - 16 - 11 - 6 - 1
- Note that you get minimum length path from start to all nodes

Node	Previous	Cost
1	-1	0
2	1	1
3	2	2
4	3	3
5	4	4
6	1	1
7	2	2
8	3	3
9		
10		
11	6	2
12		
13		
14	19	7
15		
16	11	3
17	16	4
18	17	5
19	18	6
20	19	7
21	16	4
22	17	5
23	18	6
24	19	7
25		

Sketch of proof of optimality

- Typically a proof by induction
- Assume $C(x)$ is minimal for all visited nodes
- Take an edge to an unvisited node u with *the lowest* $C(x)$ (corresponds to the pop-operation in the priority queue)
- Then, $C(u) = C(v) + d(v, u)$ is minimal
- If there was a shorter path
  - via visited nodes, that edge would have been chosen
  - Via unvisited nodes, that edge would have been explored before

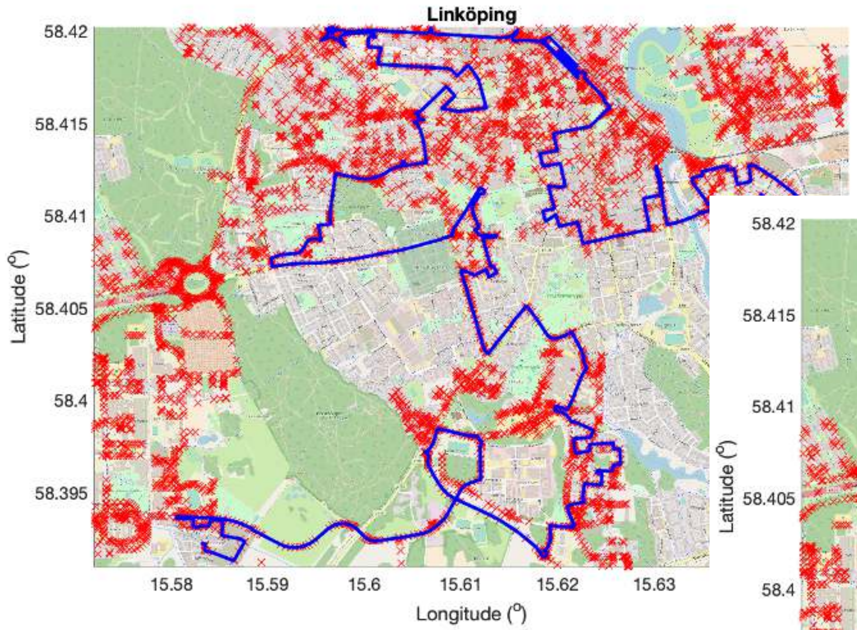


Properties of Dijkstra's algorithms

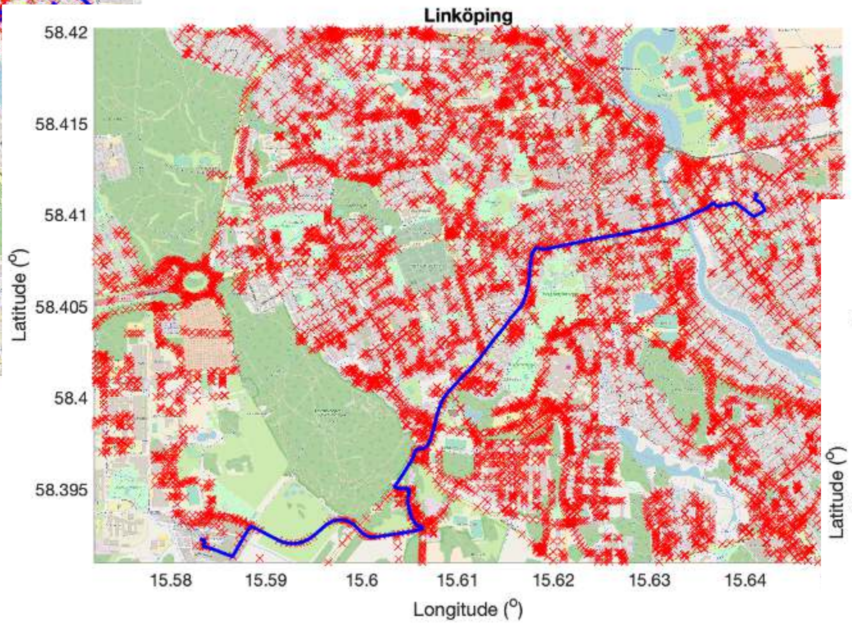
- Once you pop the goal node (on line 6), you are sure you've found the optimal path
- Complexity properties ≈ edges gives insertions and nodes pops:
  $$\mathcal{O}(|E|T_{\text{insert}} + |V|T_{\text{pop}})$$
- With balanced binary heap, both operations are $\mathcal{O}(|V|\log|V|)$ and $|E| = \mathcal{O}(|V|^2)$ so resulting
  $$\mathcal{O}(|V|^2\log|V|)$$
- Worst-case bounds for fully connected graphs maybe not that relevant; nodes are typically only connected to a few nodes.

```
1  function Dijkstra:
2      C(x_I) = 0
3      Q.insert(x_I, C(x_I))
4
5      while Q ≠ ∅
6          x = Q.pop()
7          if x = x_G
8              return SUCCESS
9
10         for u ∈ U(x)
11             x' = f(x, u)
12             if no previous(x') or
13                 C(x') > C(x) + d(x, x')
14                 previous(x') = x
15                 C(x') = C(x) + d(x, x')
16                 Q.insert(x', C(x'))
17
18     return FAILURE
```

Dijkstra optimal in length, but not in performance

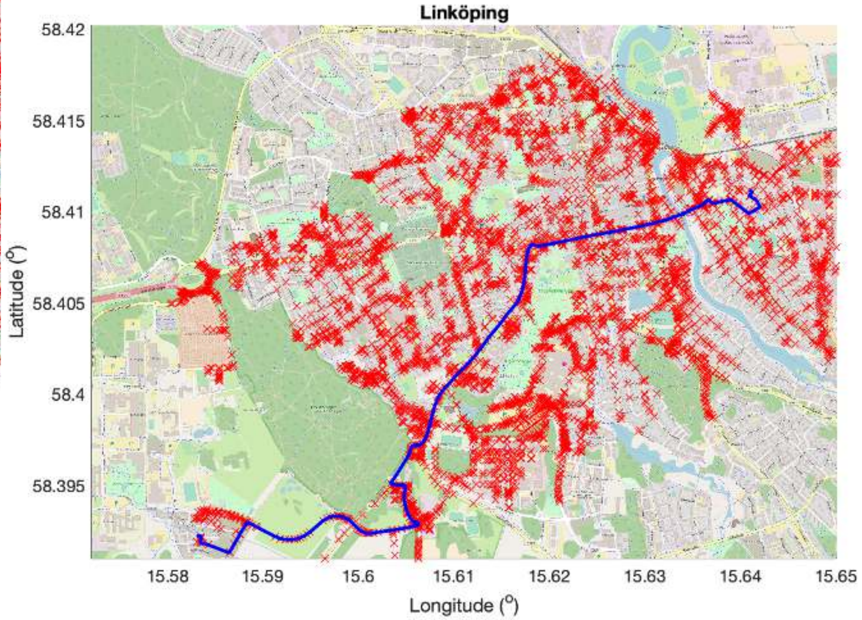


Depth First
26 km path
7570 nodes visited



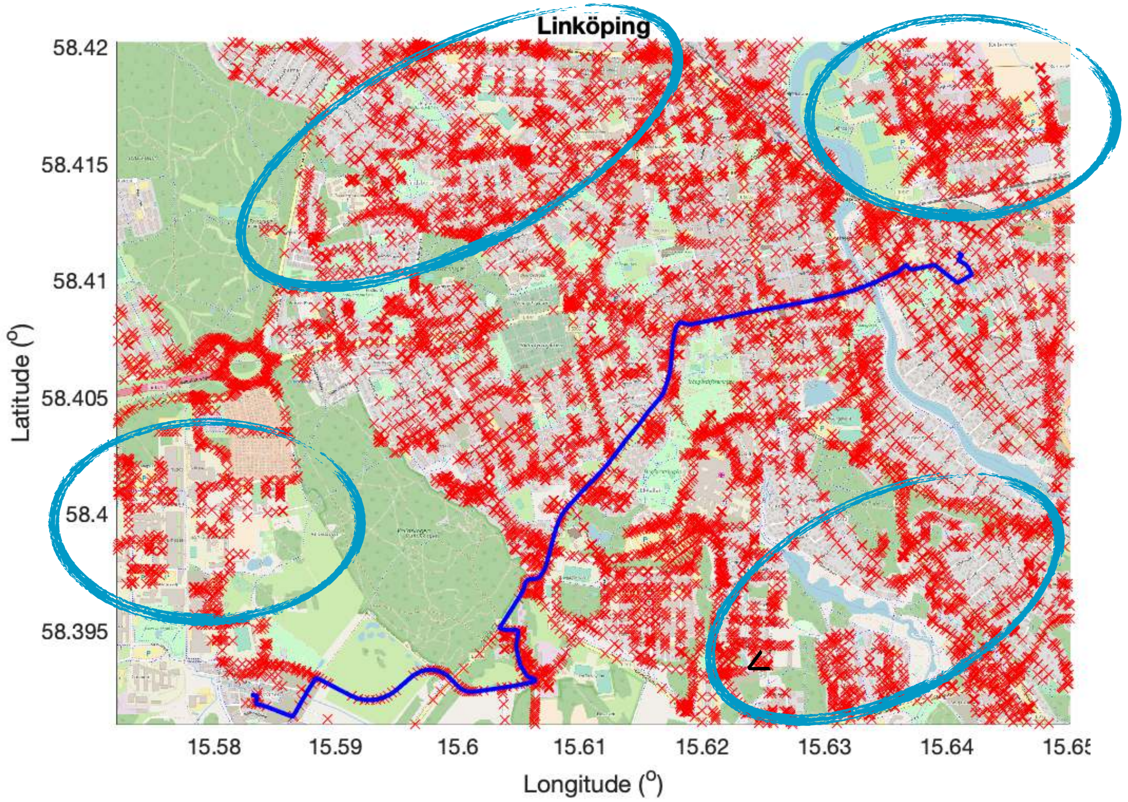
Dijkstra
5.38 km path
11626 nodes visited

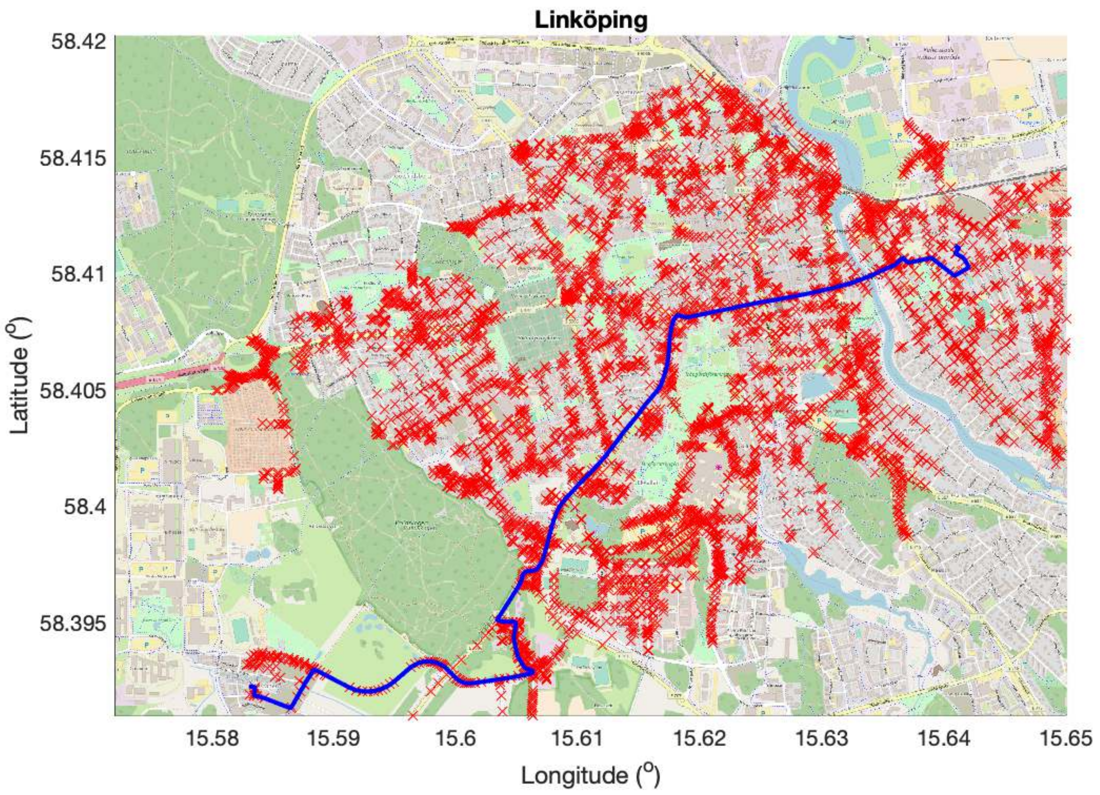
A*
5.38 km path
6594 nodes visited



A* - efficiently finding an optimal path

Keep optimality but reduce the number of visited nodes





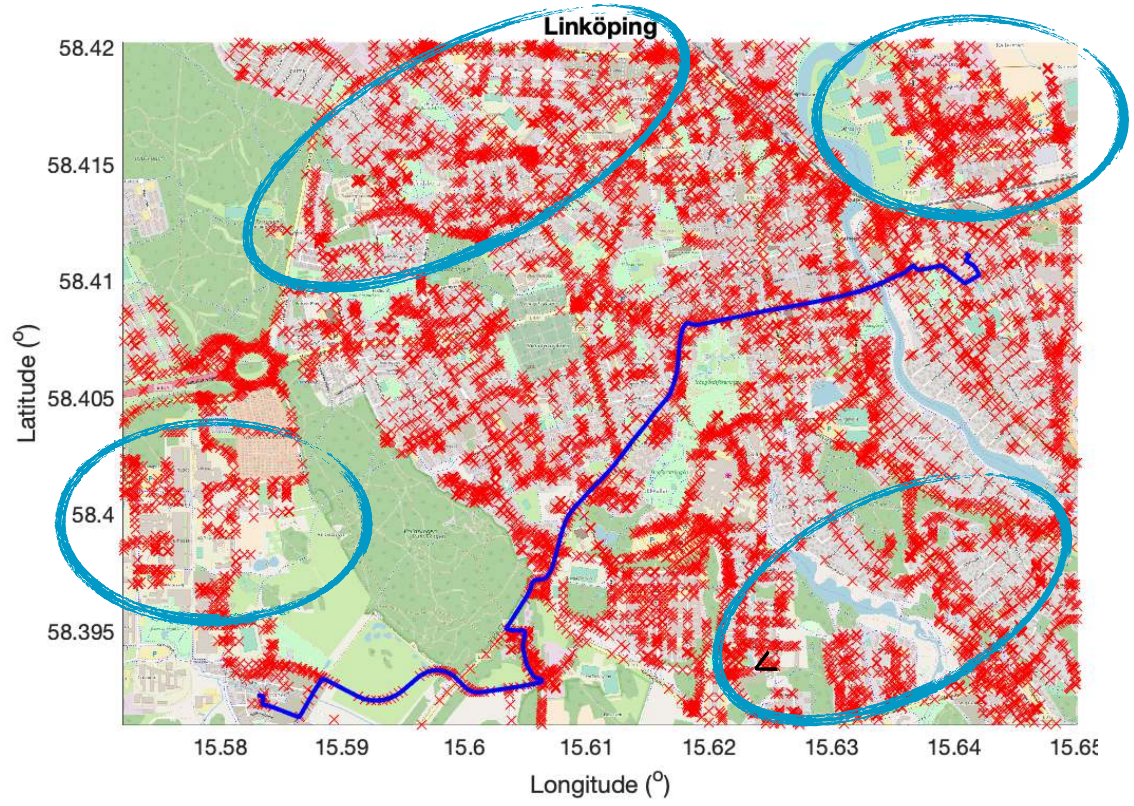
Strategy:
1. Prioritize nodes according to **estimated** final length
2. Explore nodes in the search that have high chance to be in optimal path.

Estimated final length

- Let $C(x)$ be the cost to come as before
- Let $h(x) \geq 0$ be an estimate of cost to go to the goal; called a *heuristic* function
- The estimated total length is then

$$C(x) + h(x)$$

  used in the priority queue
- Means; explore nodes that have a low **estimated** final length
- With a good heuristic, we will find a solution without exploring too many nodes

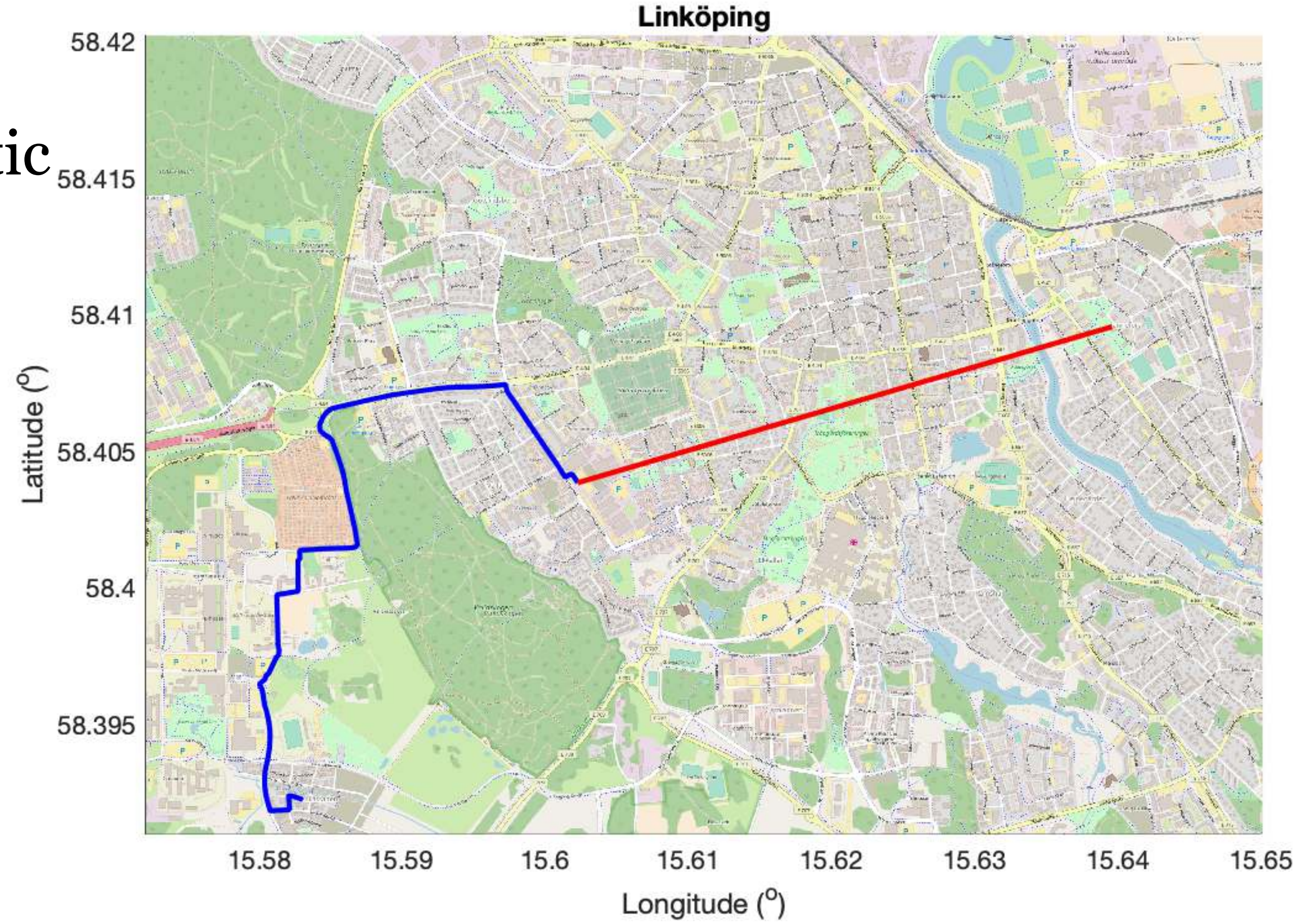


Estimated final length – heuristics

- Let $C(x)$ be the cost to come as before
- Let $h(x)$ be an estimate of cost to go to the goal; called a heuristic function
- The estimated length is then

$$C(x) + h(x)$$

  used in the priority queue
- In the example here, the Euclidean distance to the goal is used as heuristic



Dijkstra vs A* – very similar just a change of priority

```
function Dijkstra:
    C(x_I) = 0
    Q.insert(x_I, C(x_I))

    while Q ≠ ∅
        x = Q.pop()
        if x = x_G
            return SUCCESS

        for u ∈ 𝒰(x)
            x' = f(x, u)
            if no previous(x') or
                C(x') > C(x) + d(x, x')
                 previous(x') = x
                 C(x') = C(x) + d(x, x')
                 Q.insert(x', C(x'))

    return FAILURE
```

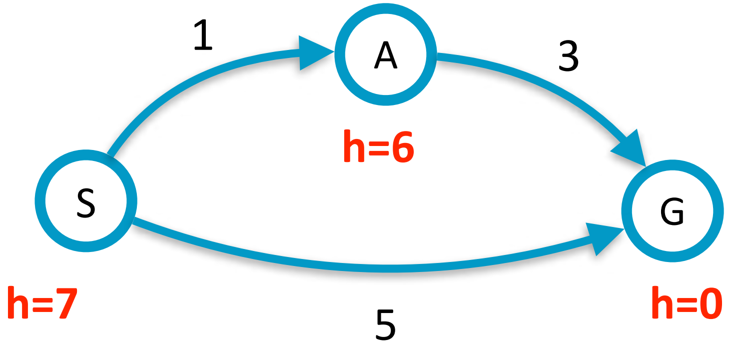
```
function Astar:
    C(x_I) = 0
    Q.insert(x_I, C(x_I) + h(x_I))

    while Q≠ ∅
        x = Q.pop()
        if x = x_G
            return SUCCESS

        for u ∈ 𝒰(x)
            x' = f(x, u)
            if no previous(x') or
                C(x') > C(x) + d(x, x')
                 previous(x') = x
                 C(x') = C(x) + d(x, x')
                 Q.insert(x', C(x') + h(x'))

    return FAILURE
```

Does A* find the optimal path?

- Efficiency of A* depends on the heuristic; the better estimate of cost-to-go, the more efficient search
- The heuristic helps us prioritize; do not prioritize nodes that probably is not part of the solution
- The priority in Dijkstra is $C(x)$ and $C(x) + h(x)$ in A*
- Clearly, for $h(x) = 0$ both algorithms give the same result and explore exactly the same search space
- The higher the value of cost-to-go for a node, the lower priority in the search. Note that no node is excluded from the search, it is just put way back in the queue if the expected cost to go through that node is high!

Does A* find the optimal path?

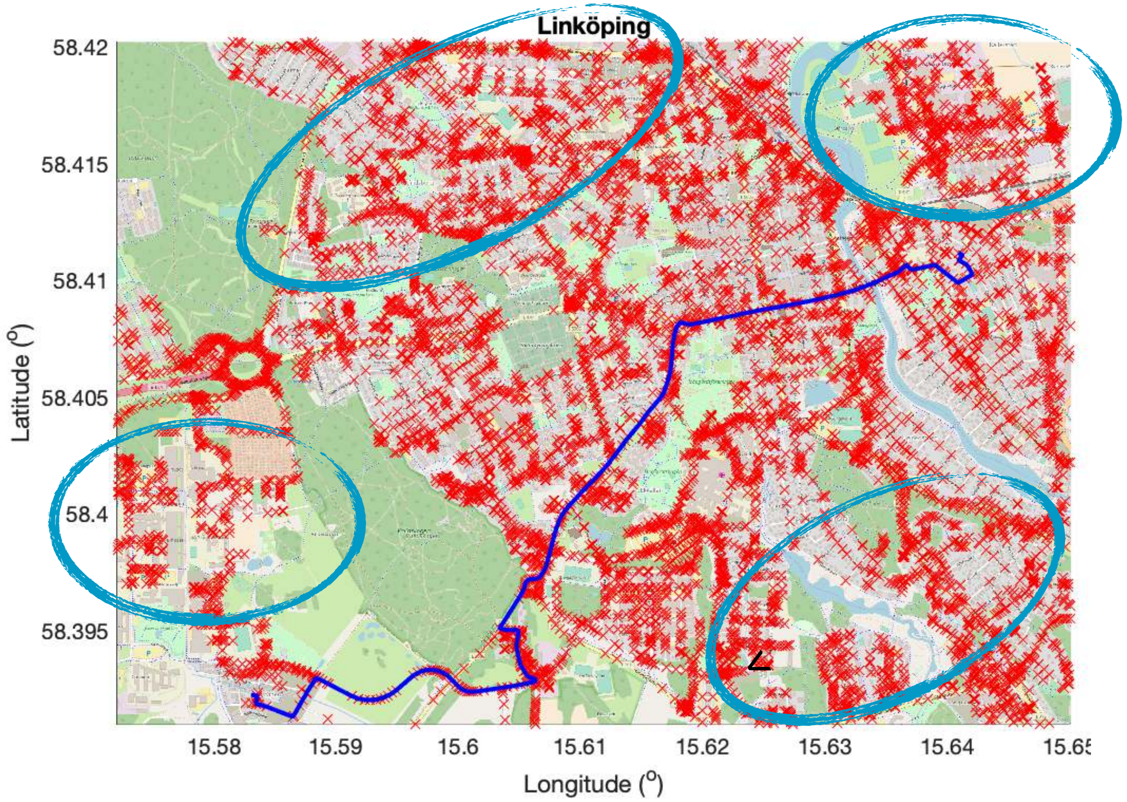
- Edge costs in black, heuristic in red
- A* will find path S—G
- What went wrong here? Found sub-optimal path.
- Heuristic seems to be the reason!

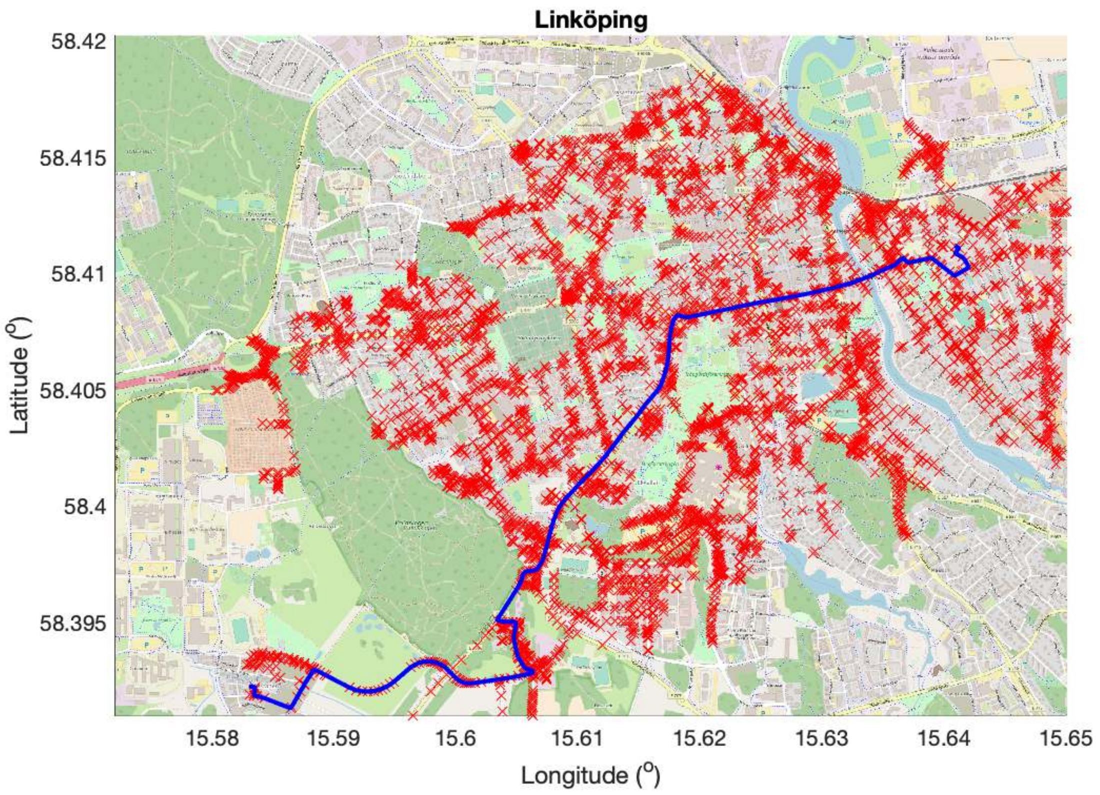


Does A* find the optimal path?

- Let $h^*(x)$ be the (unknown) true cost-to-go function
- A heuristic that satisfies
$$h(x) \leq h^*(x)$$
is called admissible
- With an admissible heuristic, once the goal node is popped the optimal solution is found.
- Optimality can be proven with a similar argument as for Dijkstra, not covered now.

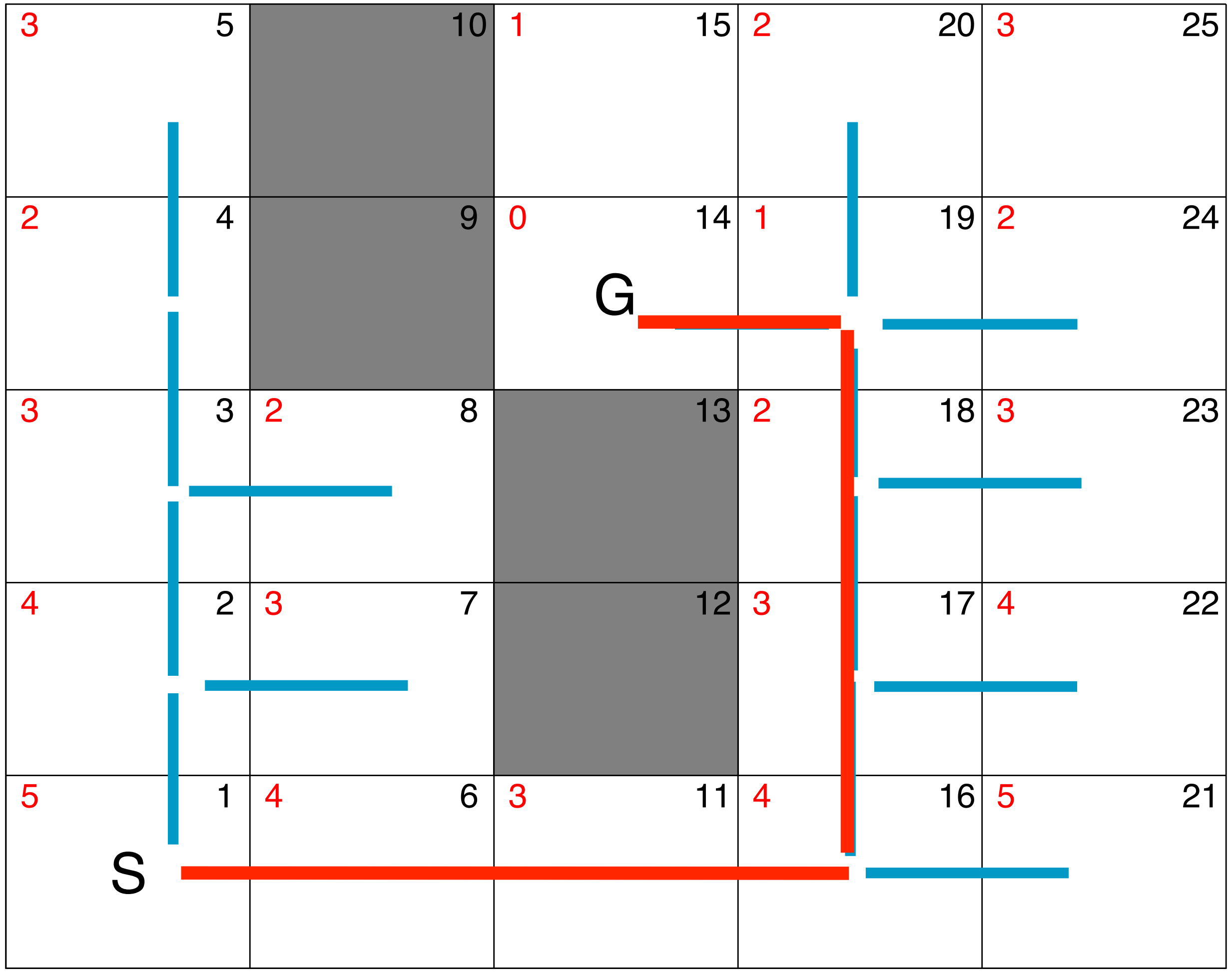
With admissible heuristic, optimality is preserved





Explore nodes in the search that have high chance to be in optimal path; here means *explore nodes that, with underestimated cost-to-go, is cheaper than others*

A* search with manhattan heuristic



Start: 1
Prio 5

Pop 1: 2 6
Prio 5 5

Pop 2: 3 6 7
Prio 5 5 5

Pop 3: 4 6 7 8
Prio 5 5 5 5

Pop 4: 6 7 8 5
Prio 5 5 5 7

Pop 6: 7 8 11 5
Prio 5 5 5 7

Pop 7: 8 11 5
Prio 5 5 7

Pop 8: 11 5
Prio 5 7

Pop 11: 5 16
Prio 7 7

Pop 5: 16
Prio 7

Pop 16: 17 21
Prio 7 9

Pop 17: 18 21 22
Prio 7 9 9

Pop 18: 19 21 22 23
Prio 7 9 9 9

Pop 19: 14 20 21 22 23 24
Prio 7 9 9 9 9 9

Pop 14: Goal!

Resulting path

- During A* search, two functions are updated
  - Previous(x) - keep track of parent node
  - Cost(x) - current cost to come
- Using Visited(x), backtracking gives the resulting path
  - 14 - 19 - 18 - 17 - 16 - 11 - 6 - 1

Node	Previous	Cost
1	1	0
2	1	1
3	2	2
4	3	3
5	4	4
6	1	1
7	2	2
8	3	3
9		
10		
11	6	2
12		
13		
14	19	7
15		
16	11	3
17	16	5
18	17	5
19	18	6
20	19	7
21	16	5
22	17	5
23	18	6
24	19	7
25		

A closer look at heuristics – consistent and admissible heuristics

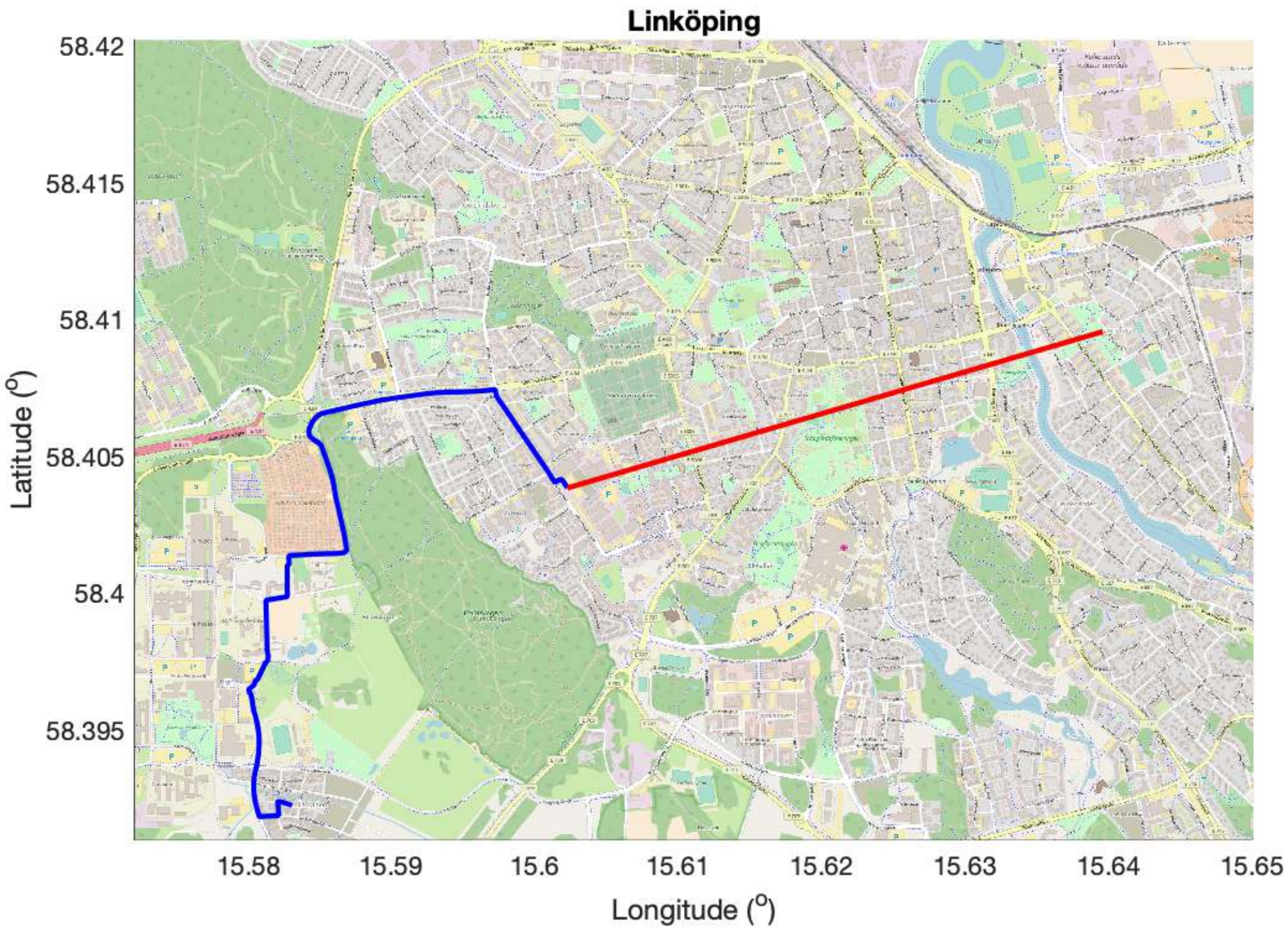
Estimated final length - example heuristic

- Clearly the heuristic is important to gain efficiency in the search
- In complex search problems, this can be really difficult
- In simple path planning, use for example the Euclidean distance

$$h(x) = |x - x_G|$$

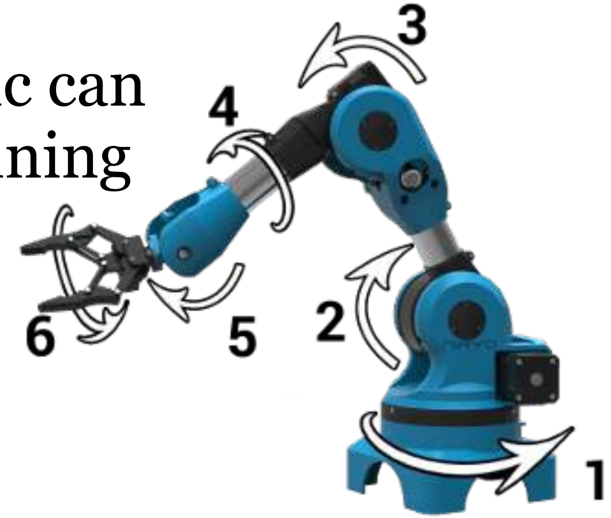
- Heuristic trivially admissible

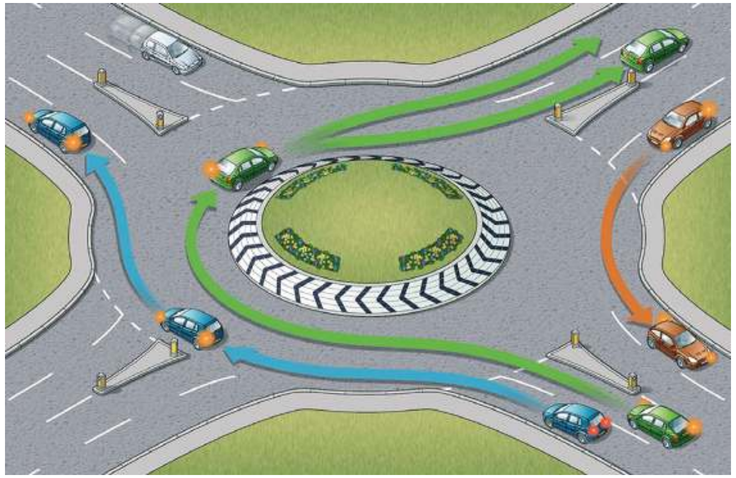
$$h(x) \leq h^*(x)$$



Heuristics, not always so simple

- Euclidean distance as heuristic can be a good choice for path planning
- Maze or cube-like problems
- Nonholonomic vehicles, e.g., parking maneuver of a car
- High-degree of freedom problems, e.g., positioning of a robotic arm





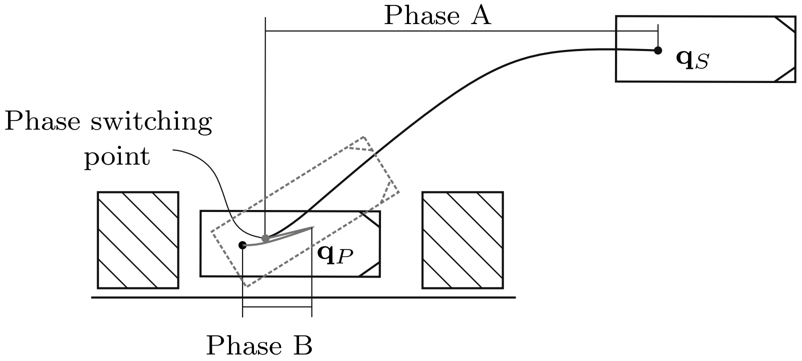
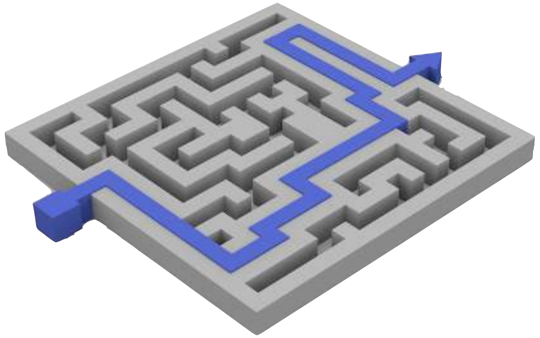
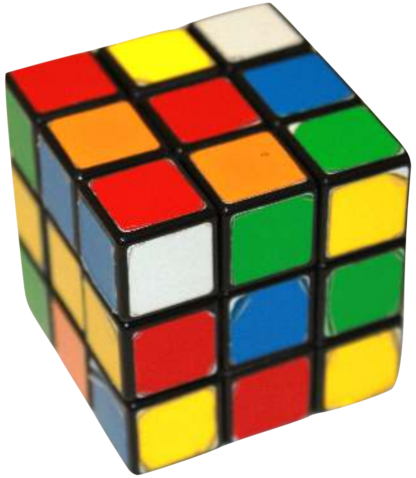


Figure from: *"Optimisation based path planning for car parking in narrow environments",* P. Zips et.al., Robotics and Autonomous Systems, 2016

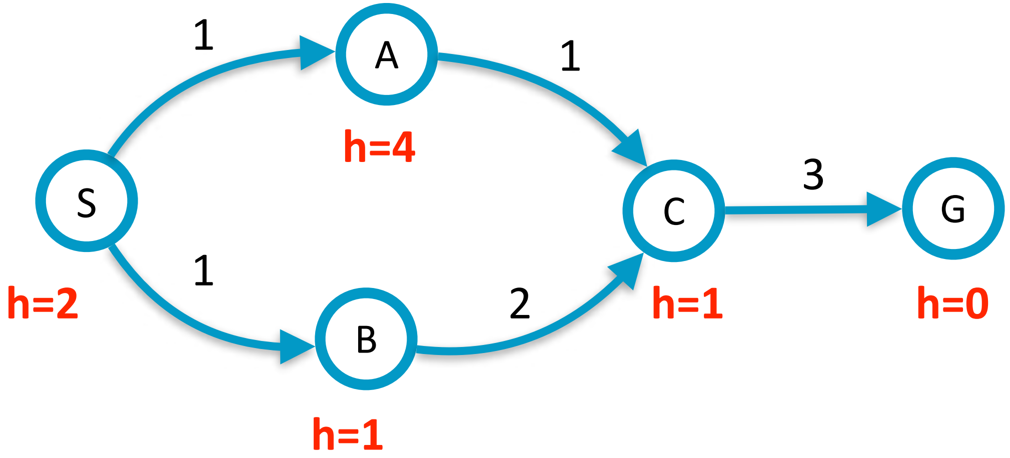




What happens here?

- Heuristic is admissible, so A* will find the optimal path S-A-C-G
- Perform A*, you will see that node C returns to the priority queue during search
- Consistent heuristic

  $$h(x) - h(x') \le d(x, x')$$



- Consistency means that the estimate, i.e., heuristic, becomes better and better along the path to the goal, $h(x') \ge h(x) + d(x, x')$The problem is the poor heuristic at C

  $$h(A) - h(C) \ge d(A, C)$$

Consistency has to do with efficiency, not optimality

- The search will find an optimal solution, regardless if the heuristic is consistent or not
- Inconsistency might lead to inefficiency, in a worst case exponential increase in node expansions

  Martelli, Alberto. "*On the complexity of admissible search algorithms*" Artificial Intelligence 8.1 (1977): 1-13.

Properties of consistent heuristics

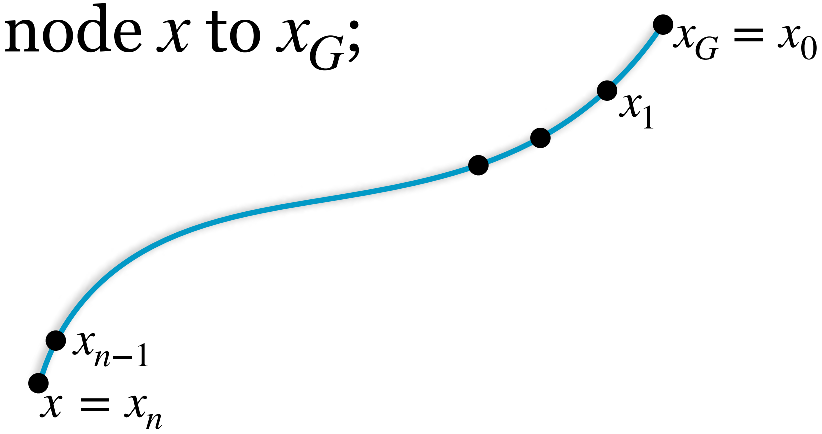
- The Euclidean heuristic is consistent

$$h(x) = |x - x_G| = |(x - x') - (x_G - x')| \leq |x - x'| + |x' - x_G| \leq$$
$$\leq |x - x'| + h(x') \leq d(x, x') + h(x')$$

- Consistent heuristic implies admissible (triangle equality is necessary and sufficient): Pearl, Judea. "*Heuristics: intelligent search strategies for computer problem solving*." (1984).

- Proof sketch: Let the path be the optimal path from node x to x_G; then the induction step is given by:

$$h(x_n) \leq h(x_{n-1}) + d(x_n, x_{n-1}) \leq$$
$$h^*(x_{n-1}) + d(x_n, x_{n-1}) = h^*(x_n)$$

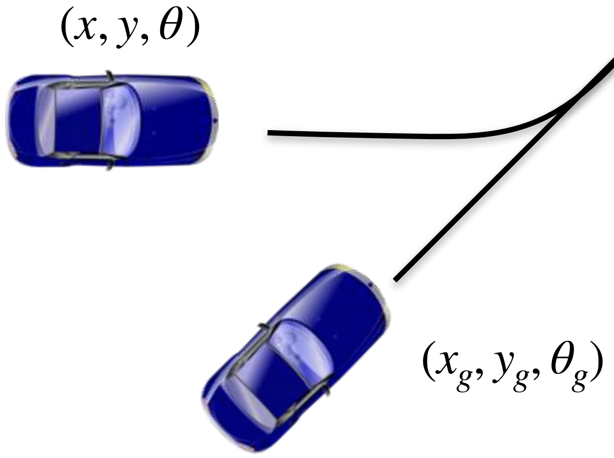


Short summary on heuristics

- Heuristic function $h(x)$ estimates distance from goal state
- Two properties
  - $h(x) \leq h^*(x)$ — admissibility, implies optimality of solution
  - $h(x) - h(x') \leq d(x, x')$ — consistency, efficiency (nodes cannot re-appear in search after popped)
- The closer $h(x)$ is to the true distance $h^*(x)$, the better. Dijkstra corresponds to the trivial heuristic $h(x) = 0$
- Consistency implies admissibility
- Heuristic that fulfills triangle inequality, e.g., Euclidean distance is consistent

Heuristics not always so easy

$$h(x) = |x - x_g|$$



$$h(x) = ?$$

Need to solve problem to get accurate heuristic

Any-time planning

What about a non-admissible heuristic?

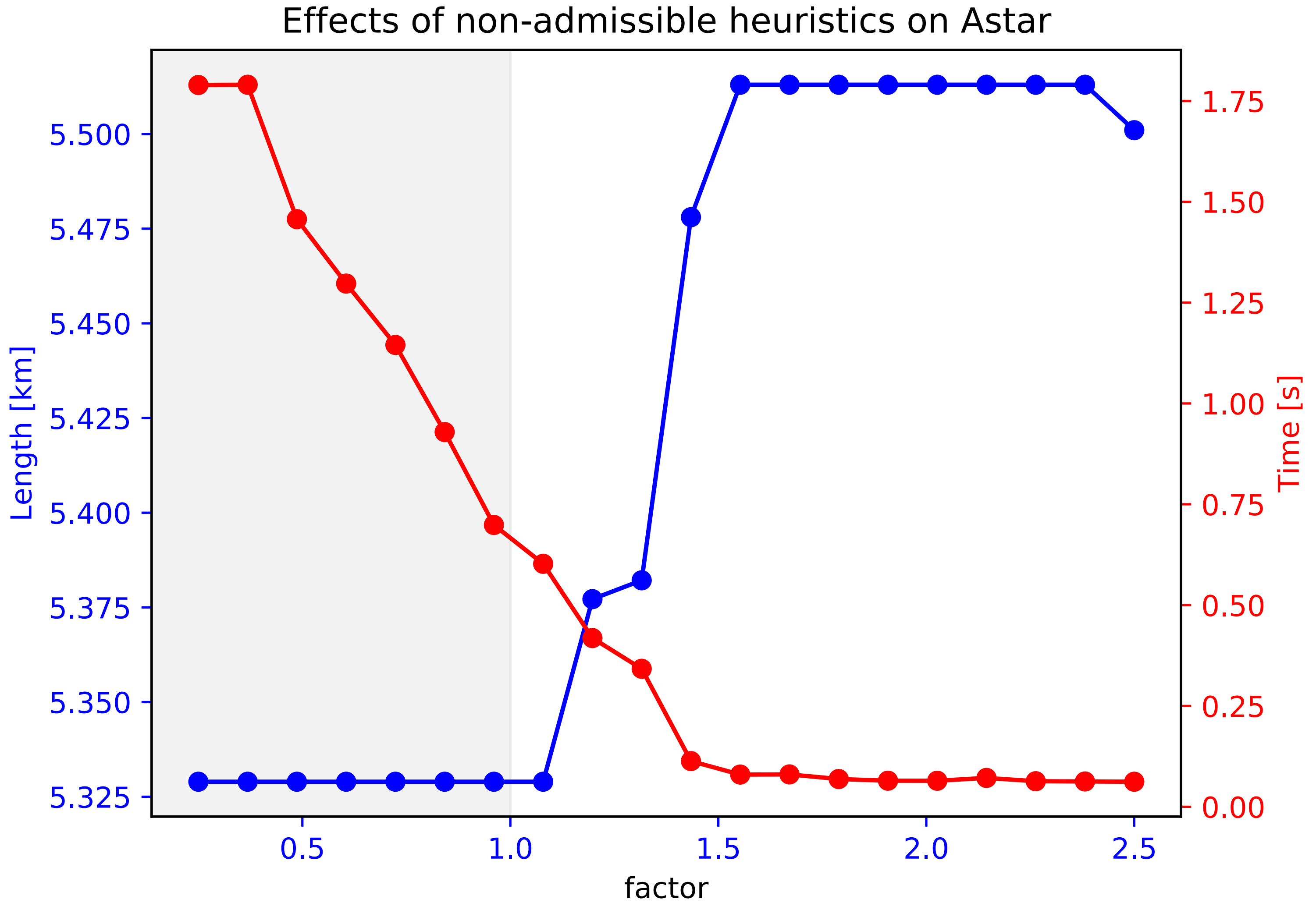
- What happens with a non-admissible heuristic, i.e., doesn’t satisfy

$$h(x) \leq h^*(x)$$

- With a non-admissible heuristic, a solution will be found but may not be optimal.
- The solution may be found faster though!

Effects of non-admissible heuristics in map routing

$$h_c(x) = c\,h(x)$$



ARA* - Anytime A*, basic principle

- Basic principle
  1. Find a solution with an inflated heuristic
  2. Lower inflation factor
  3. Reuse previous computations and compute a new solution
  4. Finish if satisfied with solution (or out of time), else go to 2
- Likhachev et.al. "*ARA*: Anytime A* with provable bounds on sub-optimality*."
  Advances in neural information processing systems, 2004.
- Connects to receding horizon control and replanning; this will be returned to later in the course

Best First Search

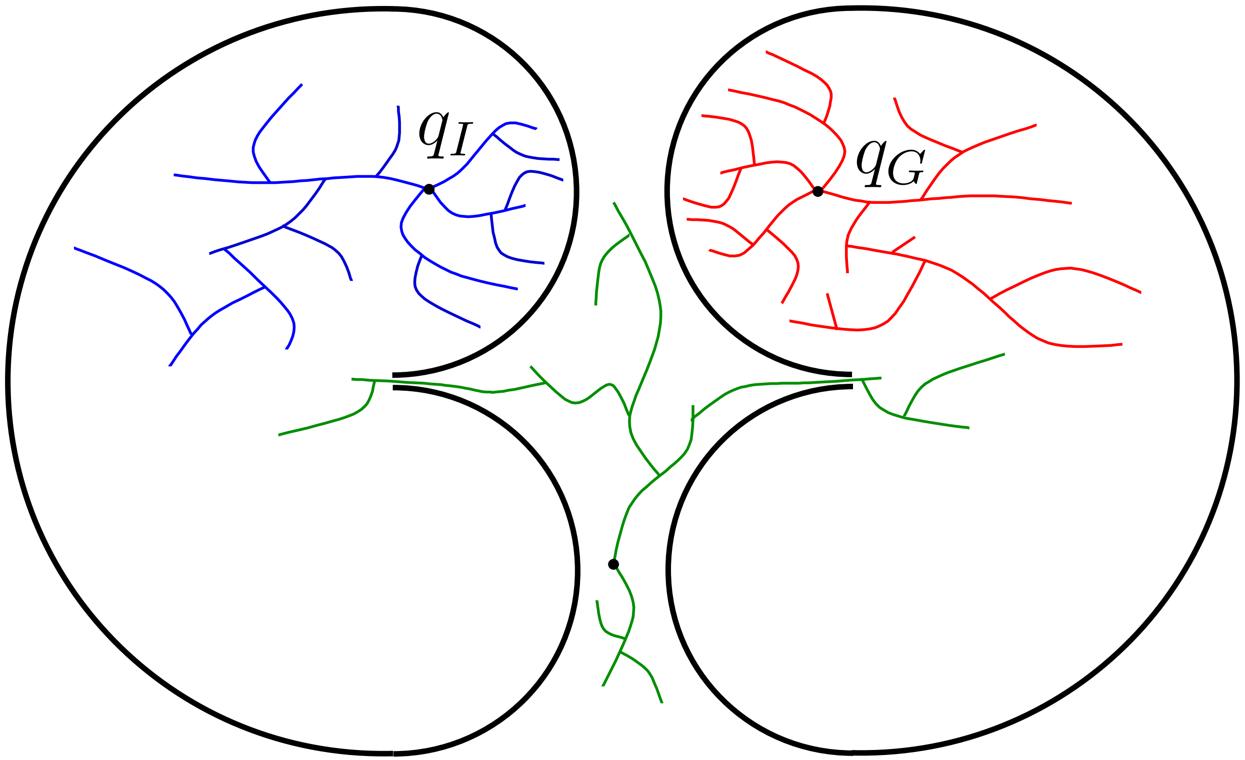
- Assume your heuristic is very good, i.e. , close to the real cost-to-go.
- Then it makes sense to expand node *x* with lowest *h*(*x*), i.e., use heuristic *h*(*x*) as priority in the queue.

```
function BestFirst:
    Q.insert(x_I, h(x_I))

    while Q ≠ ∅
        x = Q.pop()
        if x = x_G
            return SUCCESS

        for u ∈ U(x)
            x' = f(x, u)
            if no previous(x')
                previous(x') = x
                Q.insert(x', h(x'))

    return FAILURE
```

Some concluding comments

Forward-, backward-, and bi-drectional search

- Sometimes it is better to search in a particular direction
- Backward search
- Forward search
- Bi-directional search



Reading instructions

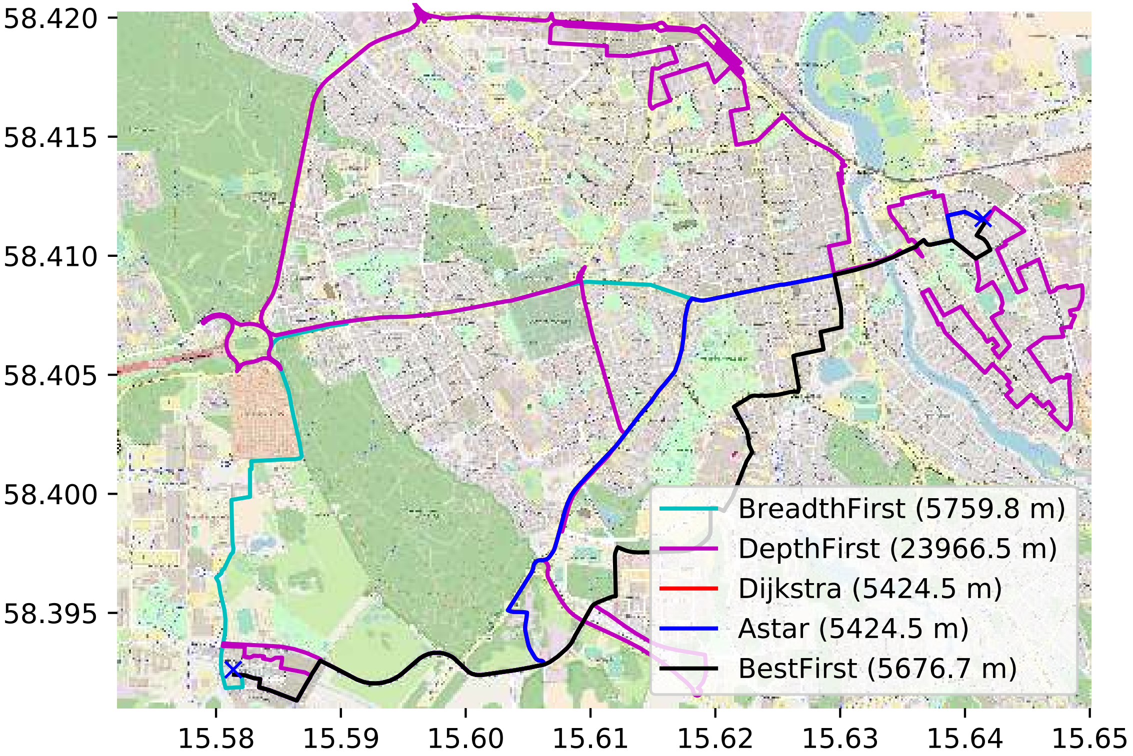
- "*Planning Algorithms*", Chapter 2 (mainly sections 2.1-2.3), S. LaValle.
- Want to dig a little deeper? Here's some extra reading…
  - "*ARA*: Anytime A* with provable bounds on sub-optimality.*", Likhachev et al. Advances in neural information processing systems, 2004.
  - "*Priority queues and Dijkstra's algorithm*", *Chen, Mo, et al.*. Computer Science Department, University of Texas at Austin, 2007.

Some take-home messages

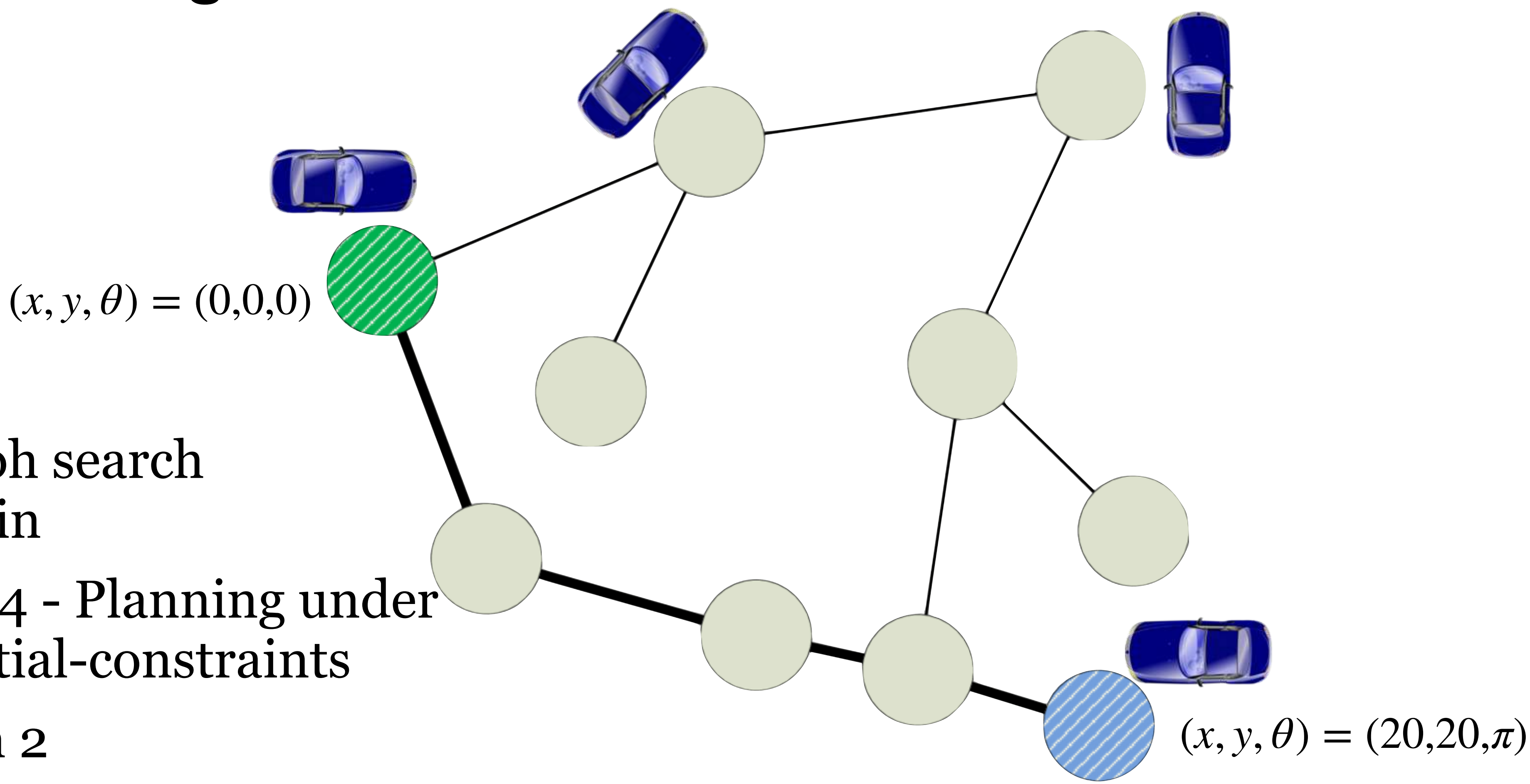
- How to formulate path planning as a search on a graph
- Basic search algorithms for motion planning in discrete graphs, in particular A*
- The heuristic function used in A*, and how it affects search efficiency
- Discrete graph search algorithms will be directly useful for motion planning with motion models
- There are *many* extensions to the basic A*

Hand-in 1

Discrete planning in a structured road network



Graph planning with motion models



- Revisit graph search algorithms in
  - Lecture 4 - Planning under differential-constraints
  - Hand-in 2

www.liu.se