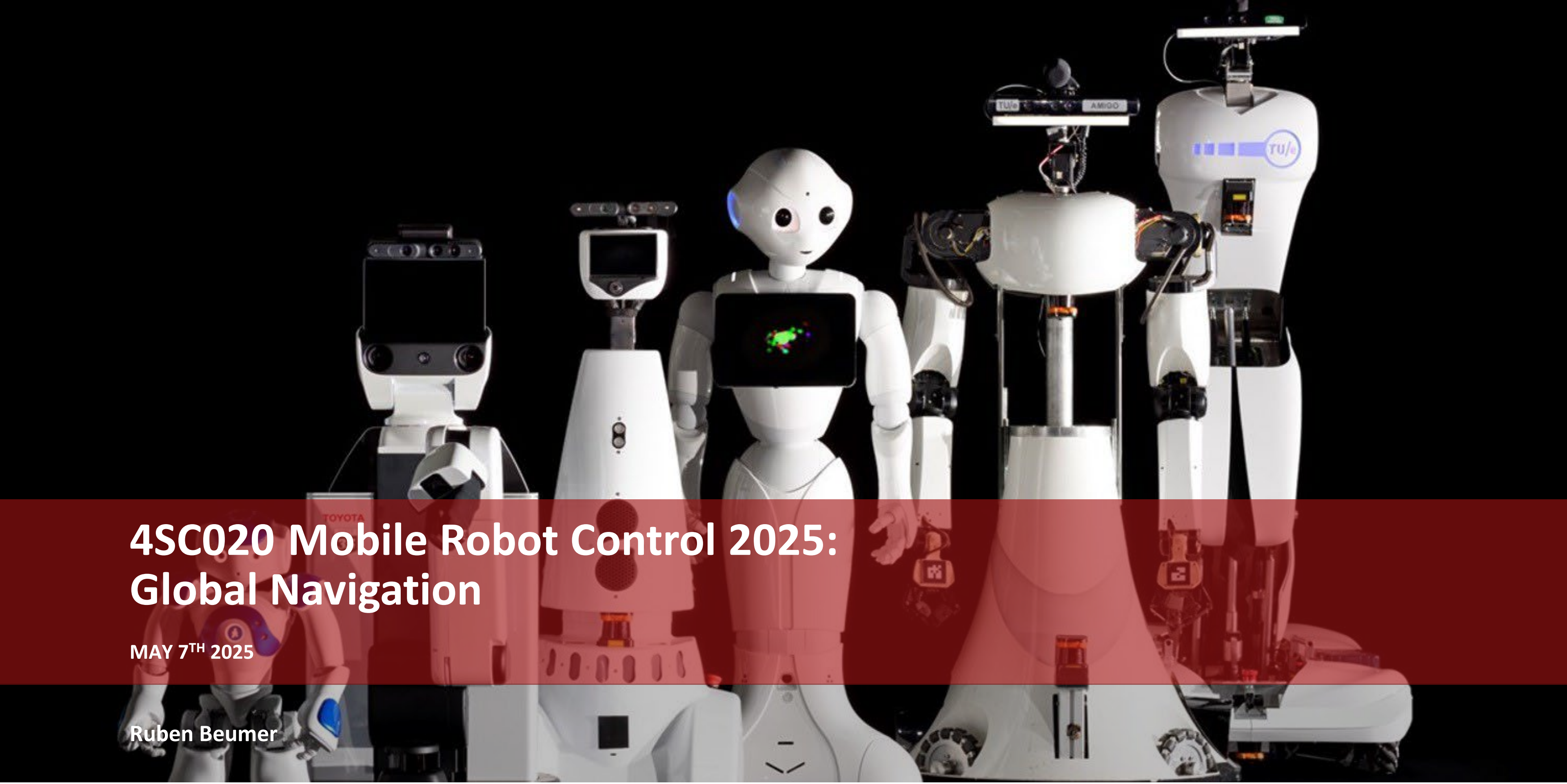


4SC020 Mobile Robot Control 2025: Global Navigation

MAY 7TH 2025

Ruben Beumer

Mechanical Engineering, Robotics

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

Outline

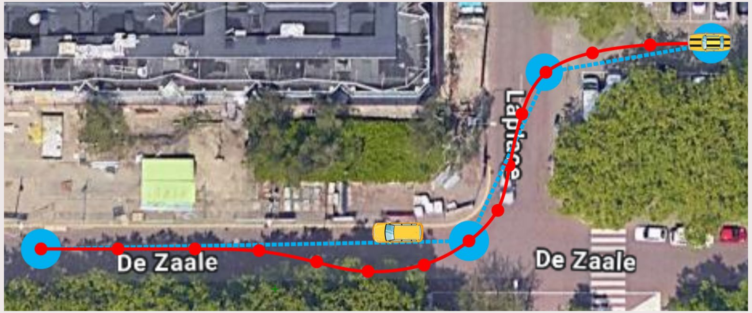
- Recap local navigation & intro global navigation
  - Robot navigation problem
    - Global vs local navigation
  - Global navigation problem
  - Motion planning algorithms: specifications and properties
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

Recap & intro / Robot navigation problem

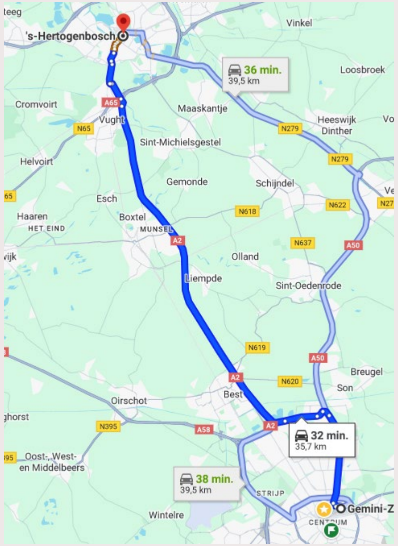
- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)

Recap & intro / Robot navigation problem

- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
  - Global: compute path from start to goal
  - Local: execute local part of global path while satisfying constraints



Local



Global

Recap & intro / Robot navigation problem

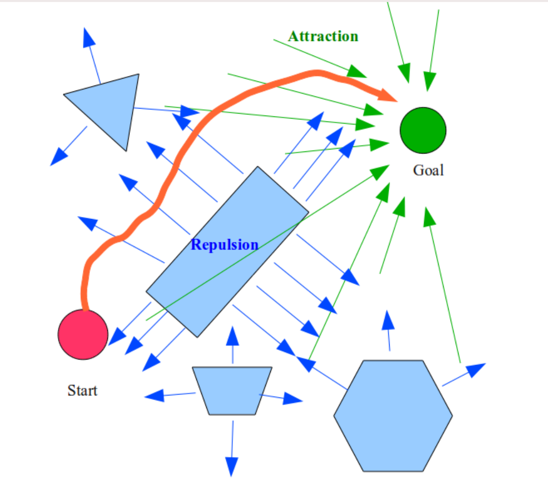
- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
  - Global: compute path from start to goal
  - Local: execute local part of global path while satisfying constraints
  - Reasons:
    - Reduce complexity
    - Static vs dynamic environment
    - Global world model often incomplete or unavailable

Recap & intro / Robot navigation problem

- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms

Recap & intro / Robot navigation problem

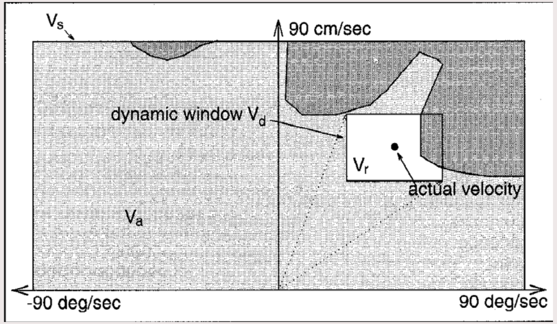
- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms
  - Artificial potential fields

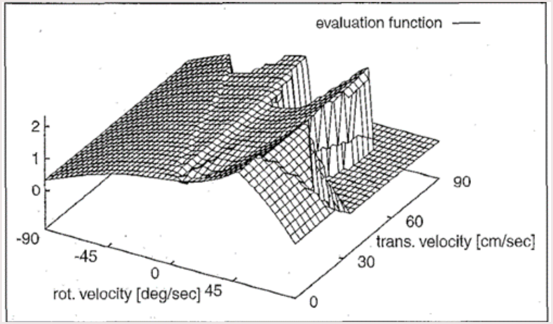


https://sudonull.com/post/62343-What-robotics-can-teach-gaming-AI

Recap & intro / Robot navigation problem

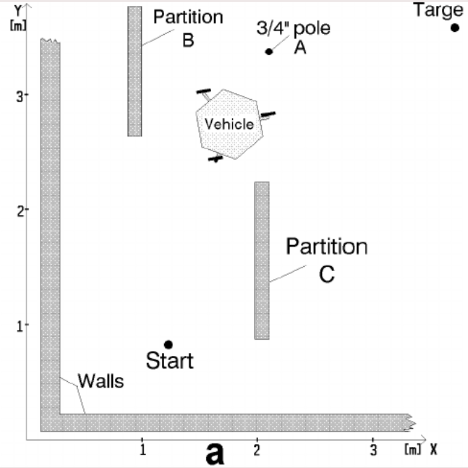
- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms
  - Artificial potential fields
  - Dynamic window approach

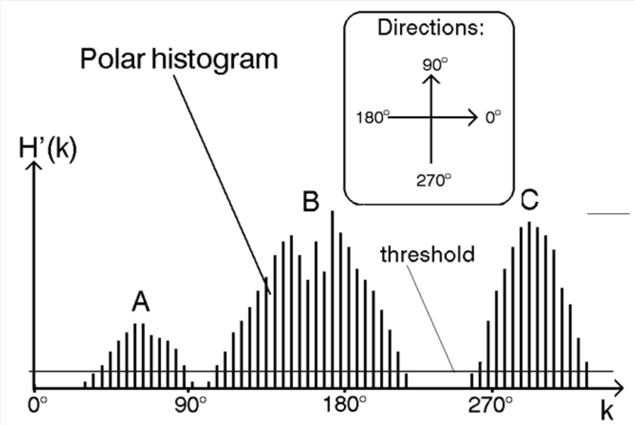




Recap & intro / Robot navigation problem

- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms
  - Artificial potential fields
  - Dynamic window approach
  - Vector field histograms





Recap & intro / Robot navigation problem

- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms
  - Artificial potential fields
  - Dynamic window approach
  - Vector field histograms
  - Optimization- and learning-based methods

Recap & intro / Robot navigation problem

- Goal: find a path or trajectory from a given initial pose (A) to the desired final pose (B)
- Division into global and local navigation
- Local navigation algorithms

Questions?

Recap & intro / Global navigation problem

- What is the global navigation problem?
  - Find a feasible path from A to B based on your current knowledge

- How does it complement local navigation?
  - Global path gives the direction to progress
  - Local navigation follows this direction safely, taking into account local objects

- What are the requirements?

Recap & intro / Motion planning algorithms: specs & properties

- Completeness: finding a path if one exists
- Optimality: finding the optimal path (time, energy, distance, …)
- Computational complexity: scalability
- Robustness against a dynamic environment
- Robustness against uncertainty
- Kinematic and dynamic constraints

Recap & intro / Motion planning algorithms: specs & properties

- Completeness: finding a path if one exists
- Optimality: finding the optimal path (time, energy, distance, …)
- Computational complexity: scalability

Also important for global planning!

- Robustness against a dynamic environment
- Robustness against uncertainty
- Kinematic and dynamic constraints

Mainly handled by local planning

TU/e

Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

Map representations

We often discretize the map to make the problem more manageable

Grid-based (equidistant cells)

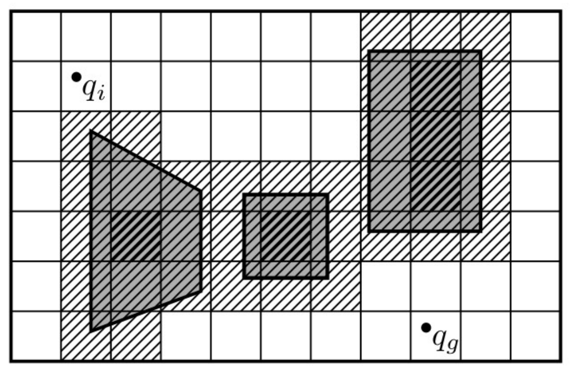
Cell-based

Graph-based

Map representations

We often discretize the map to make the problem more manageable

Grid-based (equidistant cells)



Coenen, S.A.M. (2012). Motion Planning for Mobile Robots - A Guide. Master's thesis

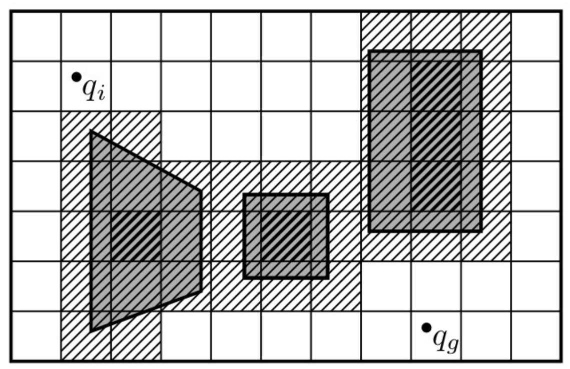
Cell-based

Graph-based

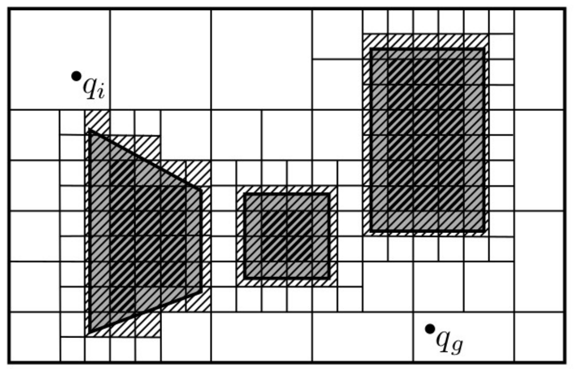
Map representations

We often discretize the map to make the problem more manageable

Grid-based (equidistant cells)



Cell-based



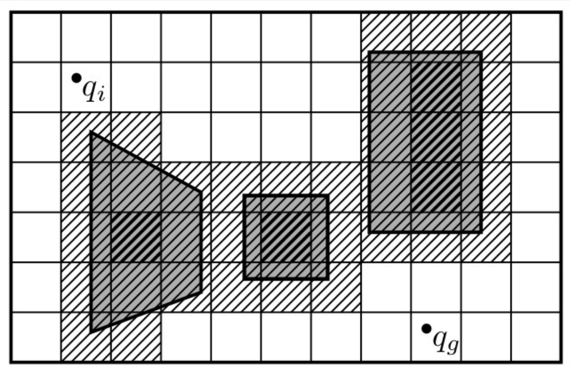
Graph-based

Coenen, S.A.M. (2012). Motion Planning for Mobile Robots - A Guide. Master's thesis

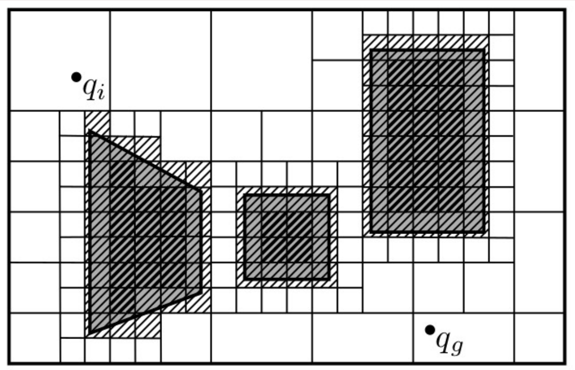
Map representations

We often discretize the map to make the problem more manageable

Grid-based (equidistant cells)

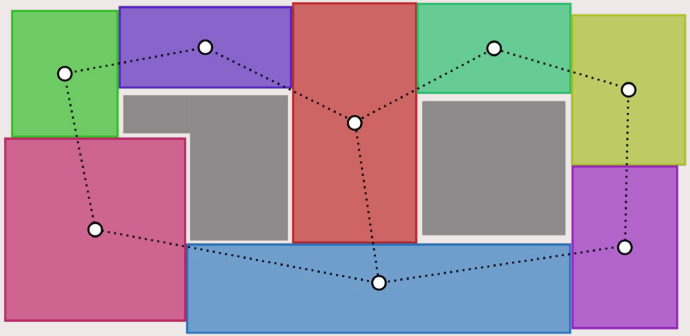


Cell-based



Coenen, S.A.M. (2012). Motion Planning for Mobile Robots - A Guide. Master's thesis

Graph-based



Blöchliger et al. (2017). Topomap: Topological Mapping and Navigation Based on Visual SLAM Maps. CoRR, http://arxiv.org/abs/1709.05533

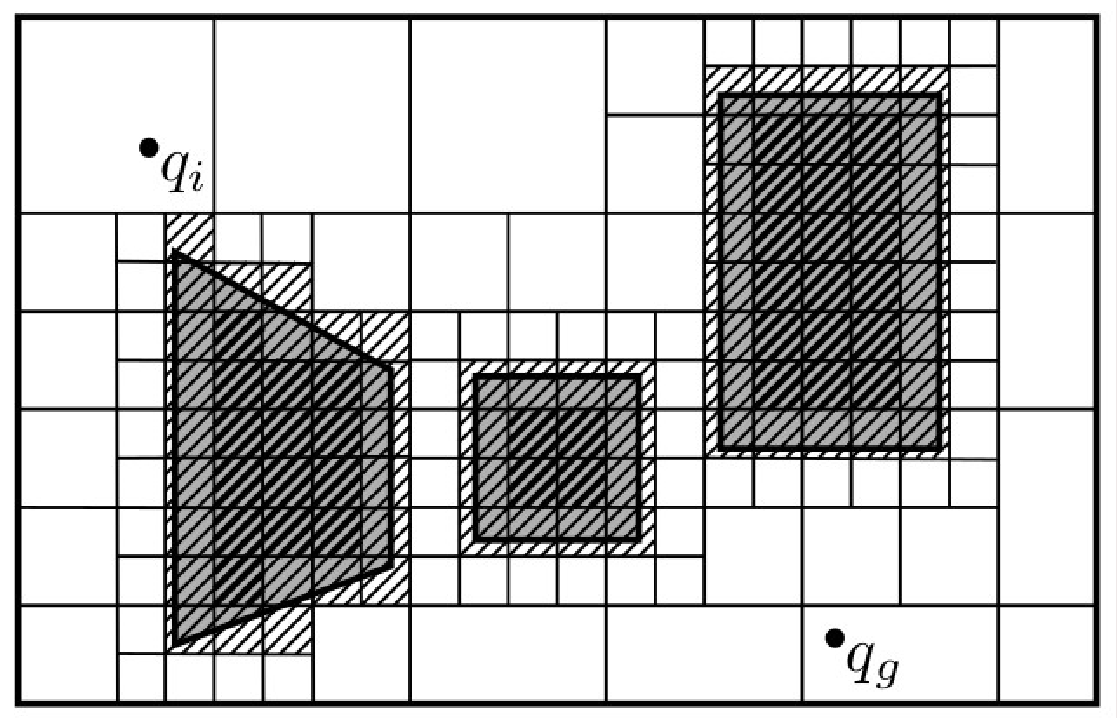
TU/e

Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

From map to graph

- Nodes
- Edges

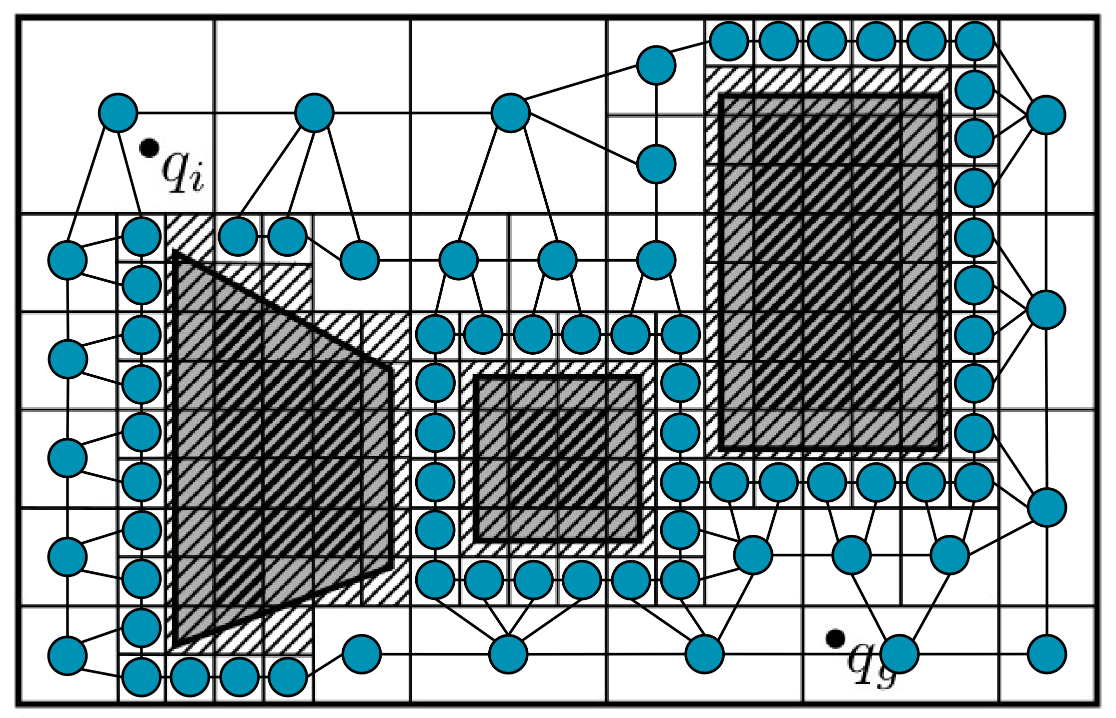


Coenen, S.A.M. (2012). Motion Planning for Mobile Robots - A Guide. Master's thesis

TU/e

From map to graph

- Nodes
- Edges

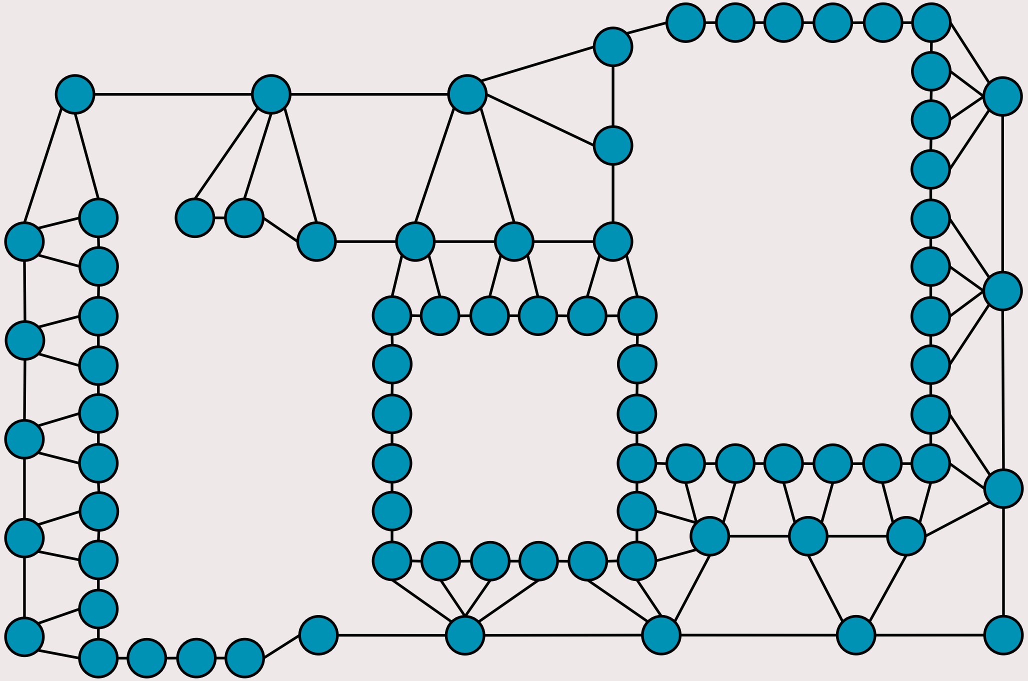


Coenen, S.A.M. (2012). Motion Planning for Mobile Robots - A Guide. Master's thesis

TU/e

From map to graph

- Nodes
- Edges



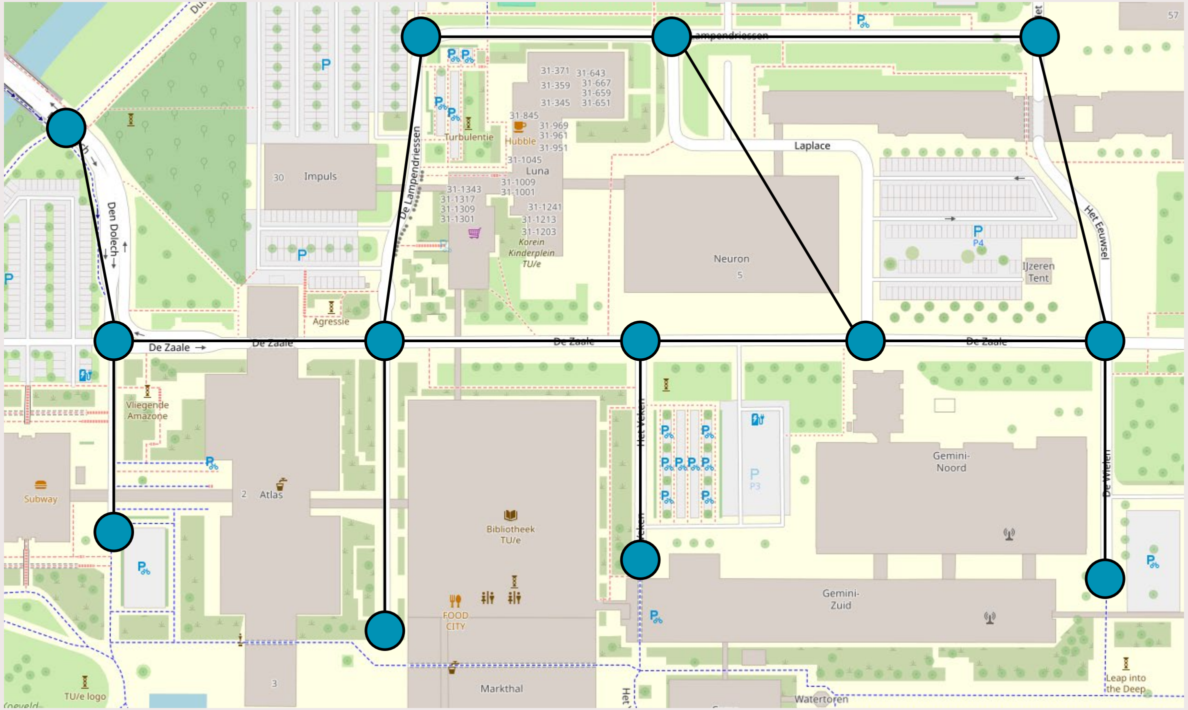
From map to graph

- Nodes
- Edges

Source: https://www.openstreetmap.org/

From map to graph

- Nodes
- Edges

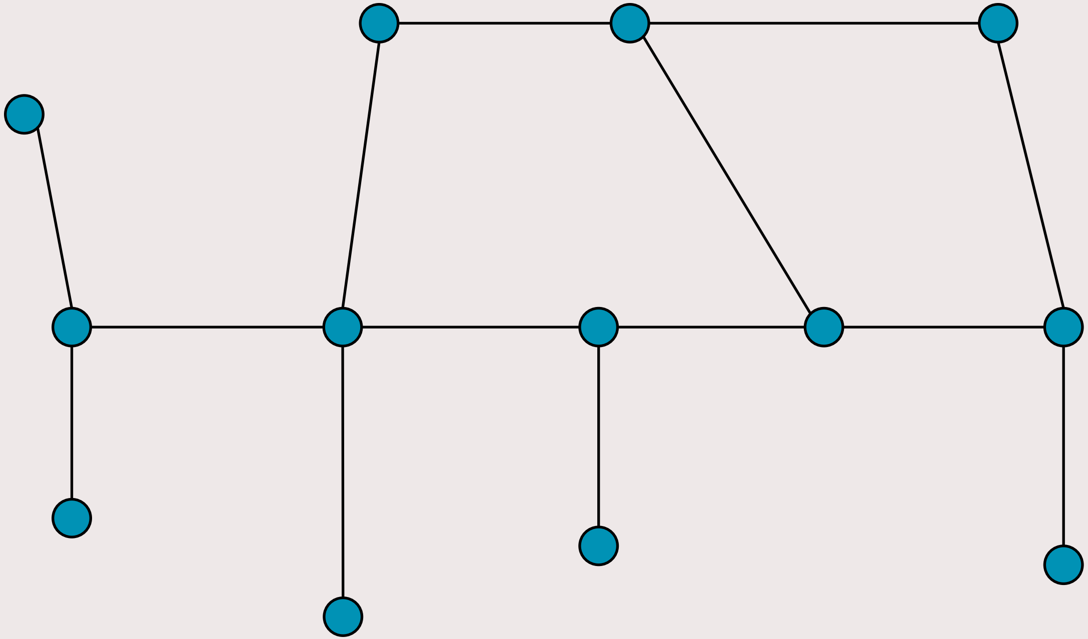


Source: https://www.openstreetmap.org/

TU/e

From map to graph

- Nodes
- Edges

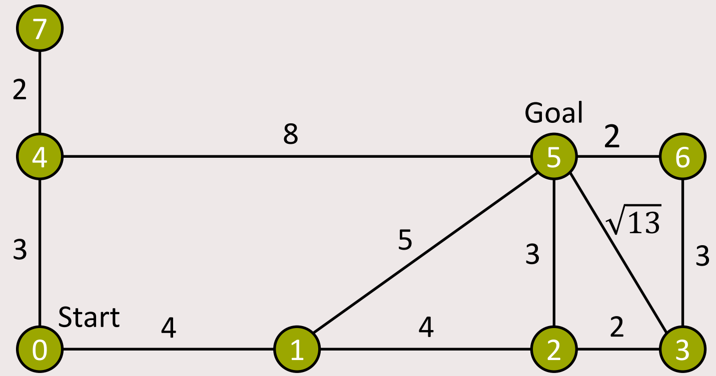


Outline

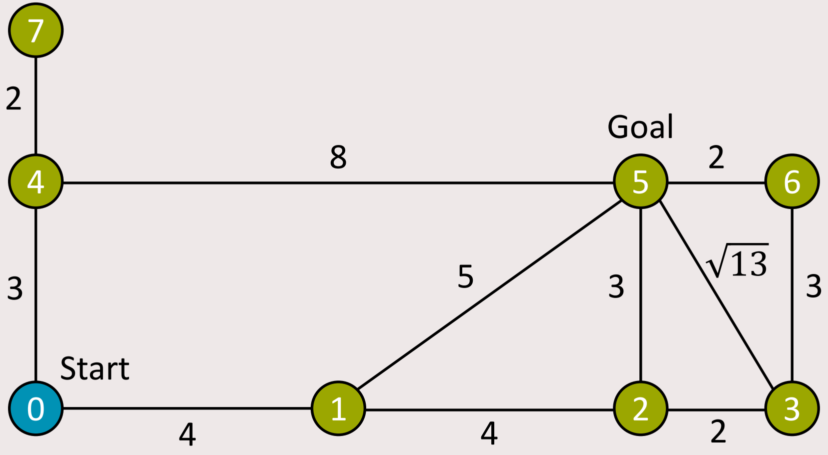
- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
  - Dijkstra’s algorithm
  - A* algorithm
- Efficient graph generation
- Summary
- Introduction to assignment

Dijkstra's algorithm

- Goal: find the shortest path from start to goal in a graph
- Two stages:
  - Exploration starting from start node
  - Tracing back the path from goal to start
- Guarantees optimality!



Dijkstra's algorithm / Exploration

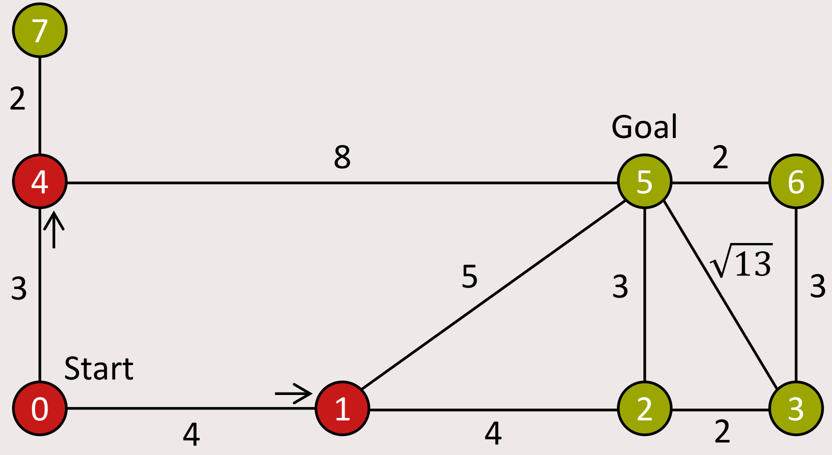


Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Exploration

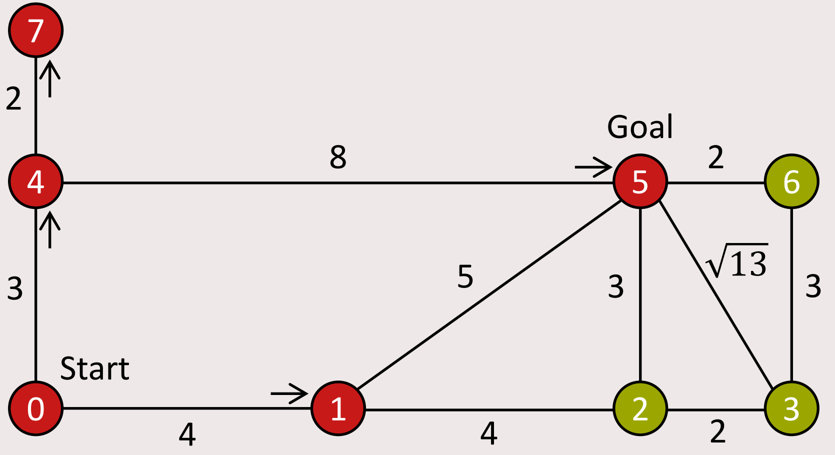


Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7
1	0	1 (4), 4 (3)	2, 3, 5, 6, 7

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Exploration



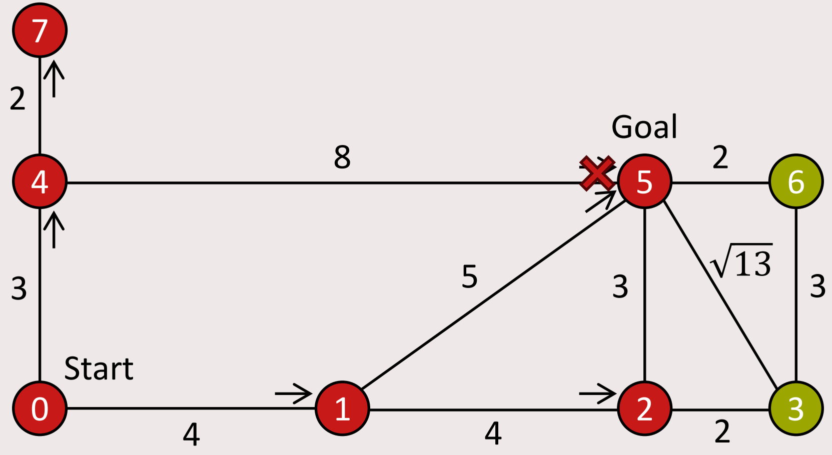
Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7
1	0	1 (4), 4 (3)	2, 3, 5, 6, 7
2	0, 4	1 (4), 5 (11), 7 (5)	2, 3, 6

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

TU/e

Dijkstra's algorithm / Exploration



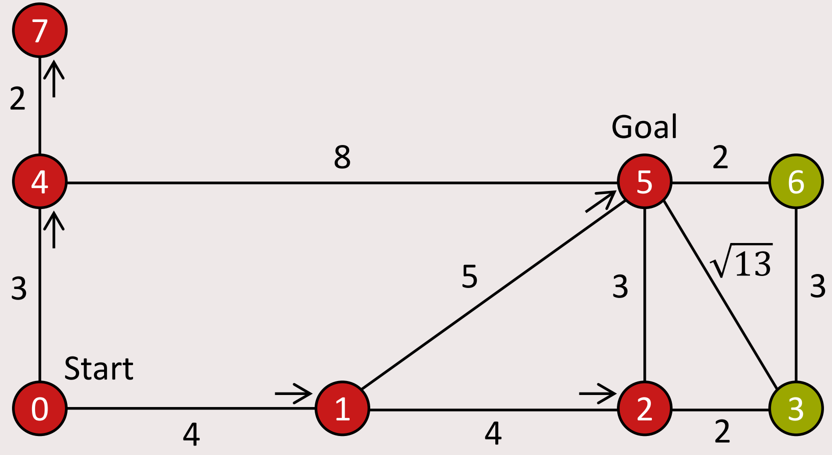
Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7
1	0	1 (4), 4 (3)	2, 3, 5, 6, 7
2	0, 4	1 (4), 5 (11), 7 (5)	2, 3, 6
3	0, 1, 4	2 (8), 5 (9), 7 (5)	3, 6

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

TU/e

Dijkstra's algorithm / Exploration

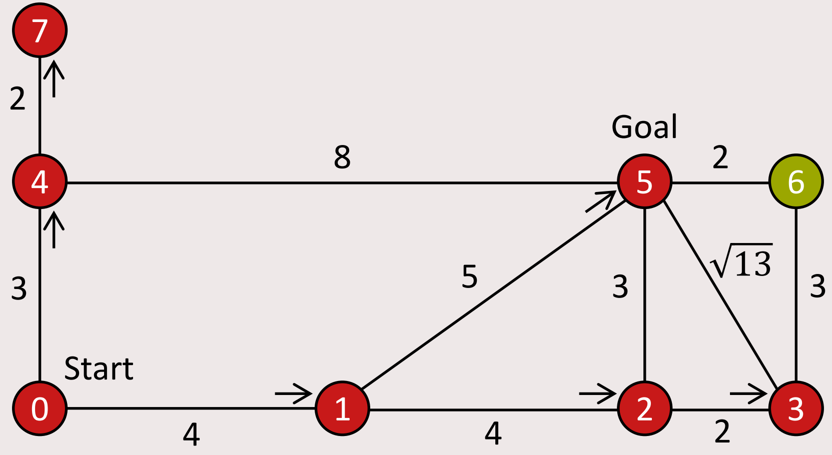


Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7
1	0	1 (4), 4 (3)	2, 3, 5, 6, 7
2	0, 4	1 (4), 5 (11), 7 (5)	2, 3, 6
3	0, 1, 4	2 (8), 5 (9), 7 (5)	3, 6
4	0, 1, 4, 7	2 (8), 5 (9)	3, 6

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Exploration



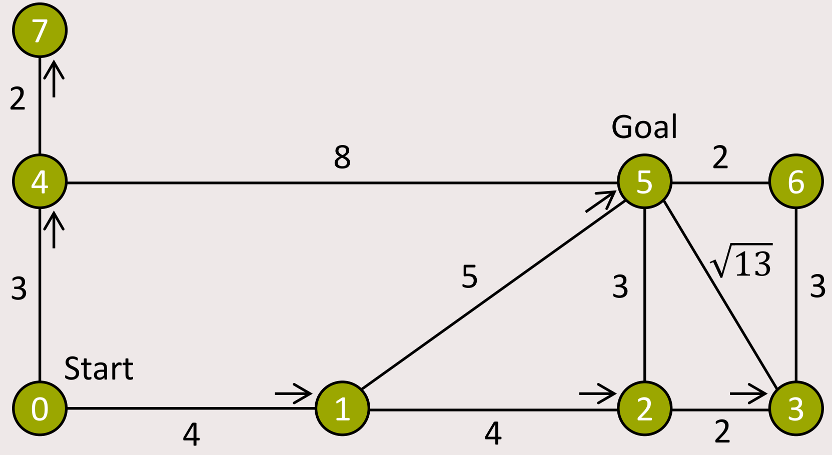
Step	Closed nodes $\{v \notin Q\}$	Open nodes (distance) $\{v \in Q \mid \text{dist}[v] < \infty\}$	Unvisited nodes $\{v \in Q \mid \text{dist}[v] = \infty\}$
0		0 (0)	1, 2, 3, 4, 5, 6, 7
1	0	1 (4), 4 (3)	2, 3, 5, 6, 7
2	0, 4	1 (4), 5 (11), 7 (5)	2, 3, 6
3	0, 1, 4	2 (8), 5 (9), 7 (5)	3, 6
4	0, 1, 4, 7	2 (8), 5 (9)	3, 6
5	0, 2, 1, 4, 7	5 (9), 3 (10)	6

```
function Dijkstra(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

Pseudo-code based on https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

TU/e

Dijkstra's algorithm / Trace path back



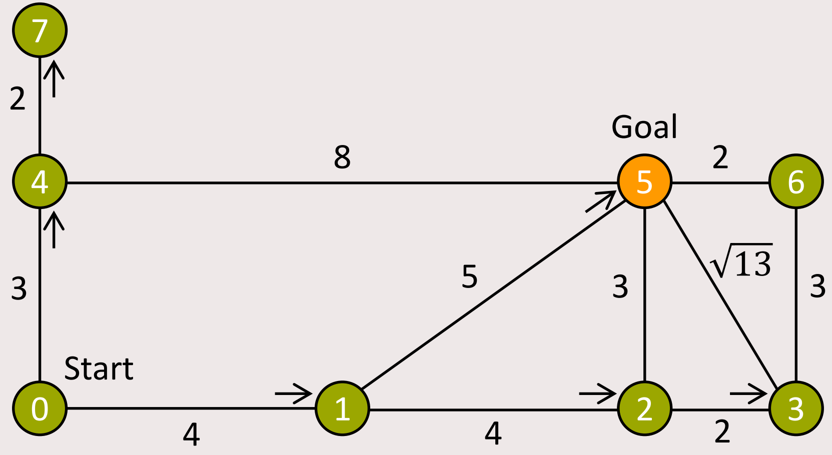
Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
    insert *u* at beginning of *Path*
    u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

TU/e

Dijkstra's algorithm / Trace path back

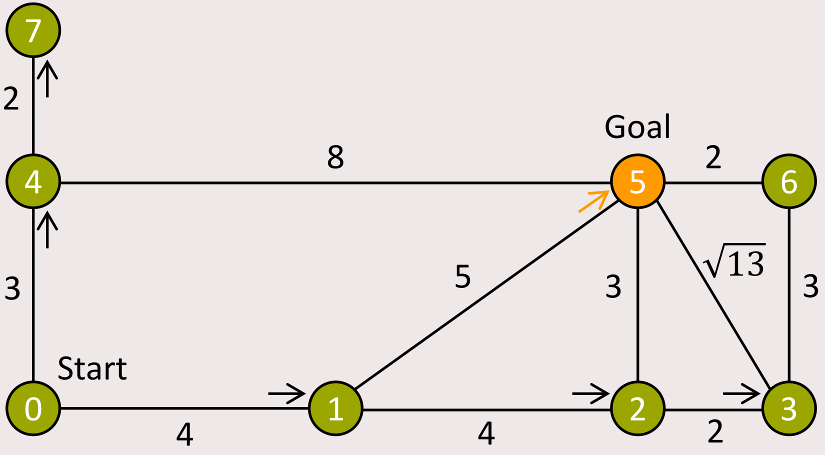


Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
    insert *u* at beginning of *Path*
    u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Trace path back

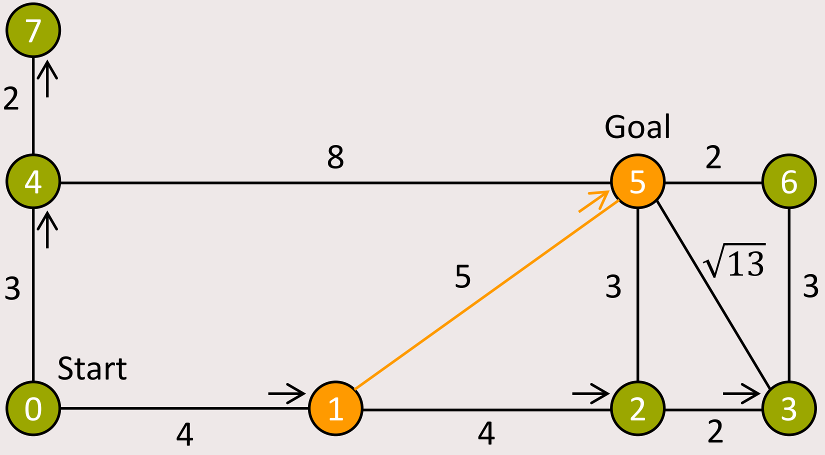


Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
  insert *u* at beginning of *Path*
  u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Trace path back

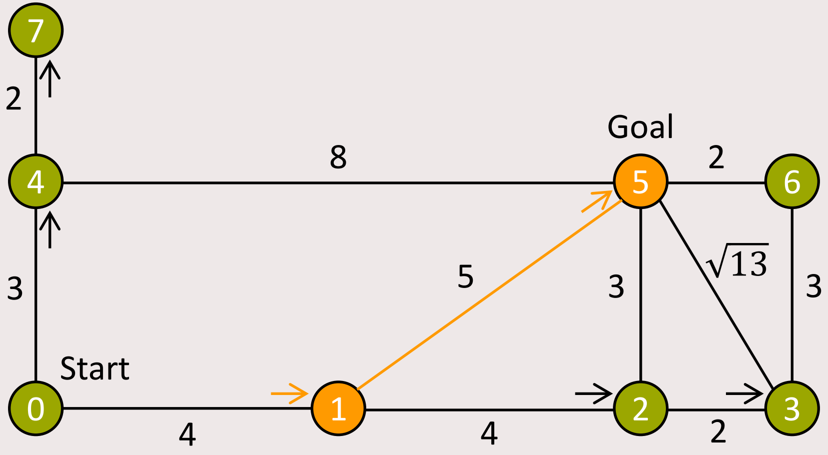


Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
  insert *u* at beginning of *Path*
  u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Trace path back

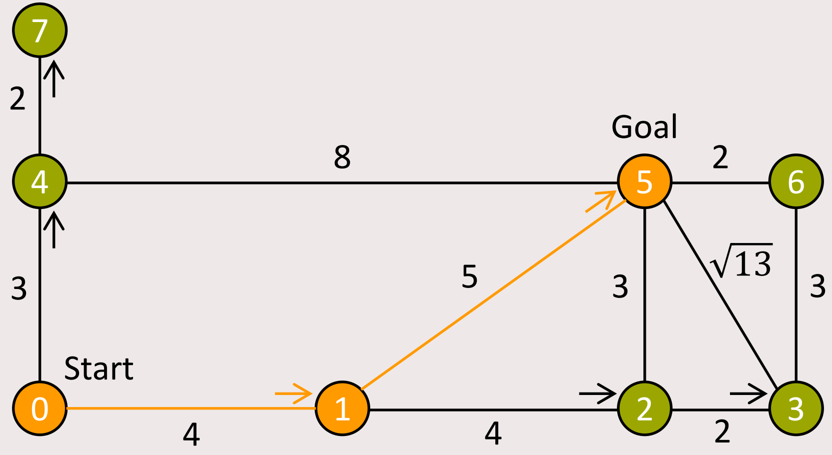


Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
  insert *u* at beginning of *Path*
  u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Trace path back

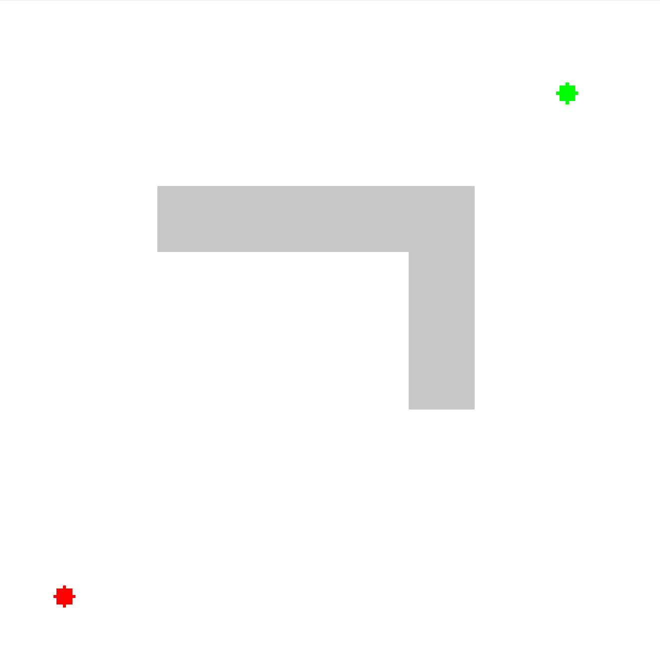


Path = empty sequence
u = *goal*

while prev[*u*] ≠ NONE **and** *u* = *start*:
  insert *u* at beginning of *Path*
  u = prev[*u*]

Pseudo-code based on
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Dijkstra's algorithm / Larger scale visualization



Dijkstra's algorithm. Source:
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

A* algorithm

"A* = Dijkstra + heuristic"

- $(v) \leq \text{cost\_to\_go}(v)$
- $H(goal) = 0$
- Example: Euclidean distance to goal

A* algorithm

"A* = Dijkstra + heuristic"

- Select open node with minimum:
  - Dijkstra: cost-to-come
  - A*: cost-to-come + heuristic cost-to-go
  - $(v) \leq \text{cost\_to\_go}(v)$
  - $H(goal) = 0$
- Example: Euclidean distance to goal

```
function Astar(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    heur[v] = H(v)
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u] + heur[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

A* algorithm

"A* = Dijkstra + heuristic"

- Select open node with minimum:
  - Dijkstra: cost-to-come
  - A*: cost-to-come + heuristic cost-to-go
- Heuristic approximates remaining distance to goal
  - $(v) \leq \text{cost\_to\_go}(v)$
  - $H(goal) = 0$
- Example: Euclidean distance to goal

```
function Astar(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    heur[v] = H(v)
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u] + heur[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

A* algorithm

ggooaall)=*0*

*cost*_to_go*(vv)*

“A* = Dijkstra + heuristic”

- Select open node with minimum:
  - Dijkstra: cost-to-come
  - A*: cost-to-come + heuristic cost-to-go
- Heuristic approximates remaining distance to goal
- Criteria for an admissible heuristic to guarantee optimality:
  - $H(goal) = 0$
  - $H(goal) = 0$

```
function Astar(Graph, start, goal):

for each node v in Graph.Nodes:
        dist[v] = INF
        heur[v] = H(v)
        prev[v] = NONE
        add v to Q
dist[start] = 0

while Q is not empty:
        u = node in Q with min dist[u] + heur[u]
        if u is goal:
                return dist, prev
        remove u from Q
        for each neighbor v of u still in Q:
                d = dist[u] + Graph.Edges(u, v)
                if d < dist[v]:
                        dist[v] = d
                        prev[v] = u
```

A* algorithm

$ggooaall)=0$

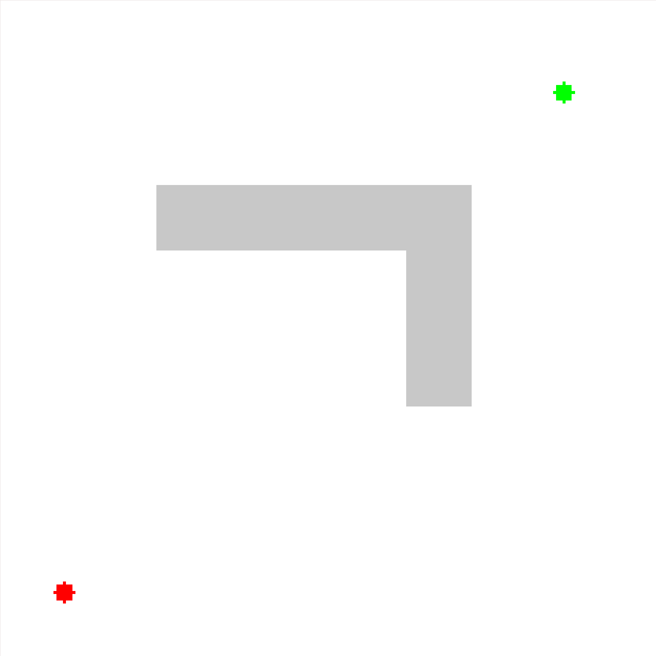
$cost\_to\_go(vv)$

"A* = Dijkstra + heuristic"

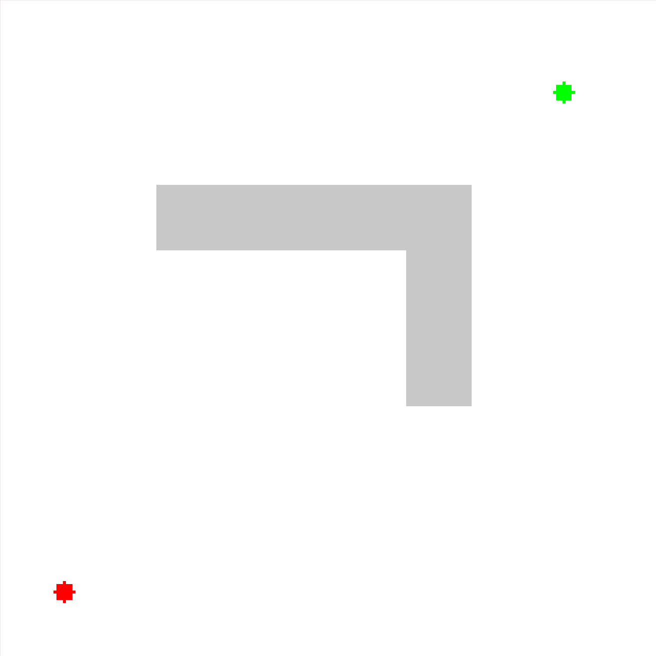
- Select open node with minimum:
  - Dijkstra: cost-to-come
  - A*: cost-to-come + heuristic cost-to-go
- Heuristic approximates remaining distance to goal
- Criteria for an admissible heuristic to guarantee optimality:
- Example: Euclidean distance to goal
  - $H(goal) = 0$

```
function Astar(Graph, start, goal):

for each node v in Graph.Nodes:
    dist[v] = INF
    heur[v] = H(v)
    prev[v] = NONE
    add v to Q
dist[start] = 0

while Q is not empty:
    u = node in Q with min dist[u] + heur[u]
    if u is goal:
        return dist, prev
    remove u from Q
    for each neighbor v of u still in Q:
        d = dist[u] + Graph.Edges(u, v)
        if d < dist[v]:
            dist[v] = d
            prev[v] = u
```

A* algorithm / Visualization compared to Dijkstra



A* algorithm. Source:
https://en.wikipedia.org/wiki/A*_search_algorithm



Dijkstra's algorithm. Source:
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Break

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

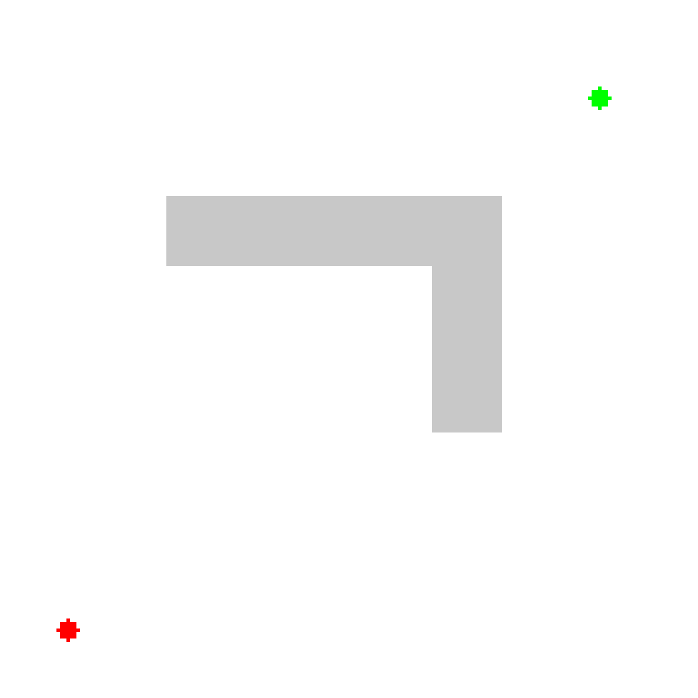
Any questions so far?

Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
  - Visibility graphs
  - Rapidly-exploring Random Tree (RRT)
  - Probabilistic Roadmap
- Summary
- Introduction to assignment

Efficient graph generation / Motivation

- Example: A* on grid maps
  - Advantages:
    + No need to create the graph in advance, because the grid is regular
    + Easy to calculate path cost
  - Disadvantages:
    - In general less efficient because not all nodes are necessary

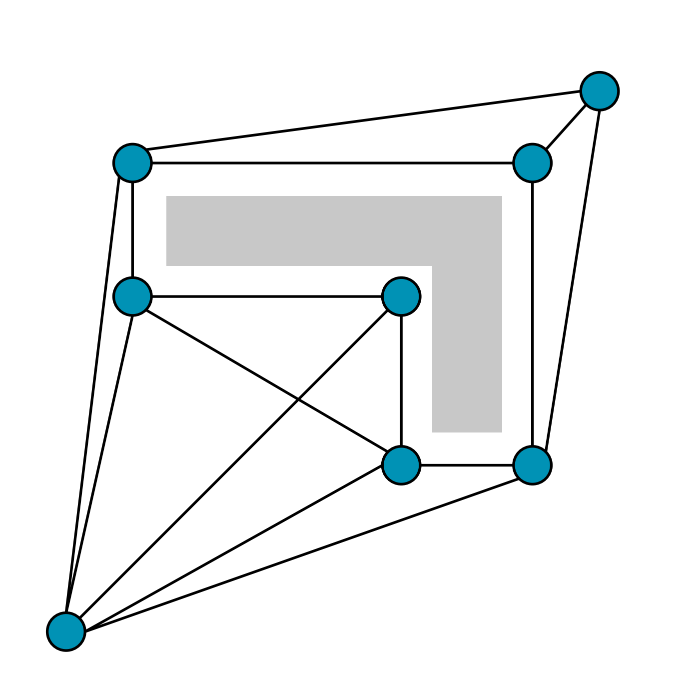


A* algorithm. Source: https://en.wikipedia.org/wiki/A*_search_algorithm

Efficient graph generation / Motivation

- Example: A* on grid maps
  - Advantages:
    + No need to create the graph in advance, because the grid is regular
    + Easy to calculate path cost
  - Disadvantages:
    - In general less efficient because not all nodes are necessary

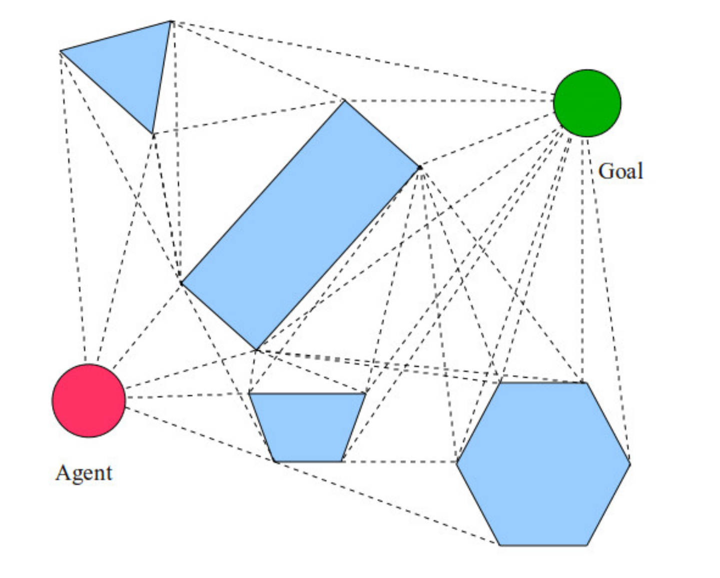
- More efficient graph desired!



A* algorithm. Source: https://en.wikipedia.org/wiki/A*_search_algorithm

Efficient graph generation / Visibility graph

- Nodes:
  - Vertices of obstacles
  - Agent
  - Goal
- Edges:
  - All straight lines between nodes that do not cross obstacles (visibility)

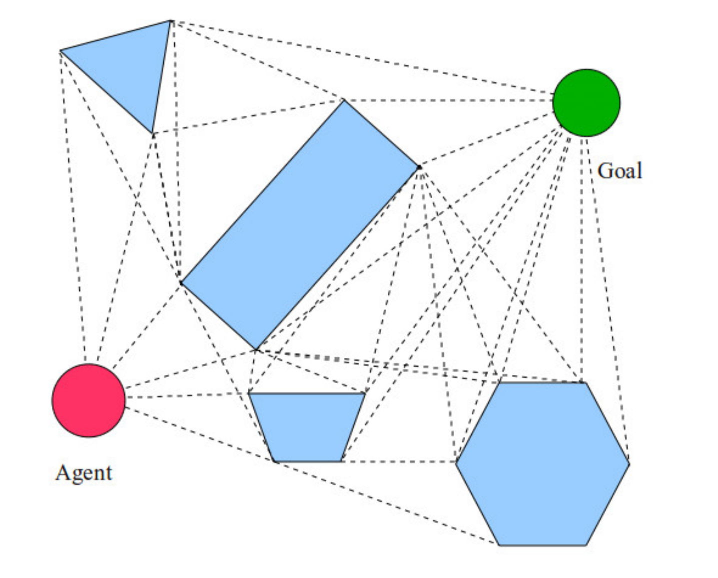


Niu, Hanlin & Lu, Yu & Savvaris, Al & Tsourdos, Antonios. (2018). An energy-efficient path planning algorithm for unmanned surface vehicles. Ocean Engineering. 161. 308-321. 10.1016/j.oceaneng.2018.01.025.

Efficient graph generation / Visibility graph

- Nodes:
  - Vertices of obstacles
  - Agent
  - Goal
- Edges:
  - All straight lines between nodes that do not cross obstacles (visibility)

- Scalability: maximum distance to create edge

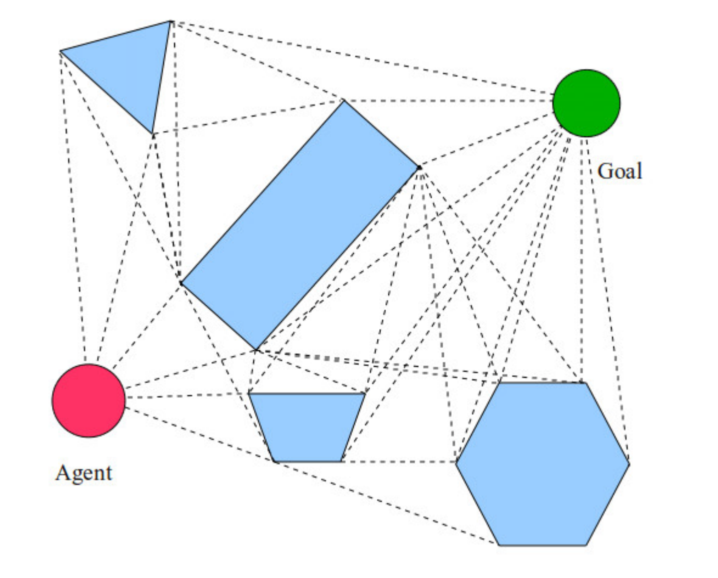


Niu, Hanlin & Lu, Yu & Savvaris, Al & Tsourdos, Antonios. (2018). An energy-efficient path planning algorithm for unmanned surface vehicles. Ocean Engineering. 161. 308-321. 10.1016/j.oceaneng.2018.01.025.

Efficient graph generation / Visibility graph

- Nodes:
  - Vertices of obstacles
  - Agent
  - Goal
- Edges:
  - All straight lines between nodes that do not cross obstacles (visibility)

- Scalability: maximum distance to create edge
- Robustness: inflate obstacles or handled by local planner



Niu, Hanlin & Lu, Yu & Savvaris, Al & Tsourdos, Antonios. (2018). An energy-efficient path planning algorithm for unmanned surface vehicles. Ocean Engineering. 161. 308-321. 10.1016/j.oceaneng.2018.01.025.

Efficient graph generation / RRT

- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state

Efficient graph generation / RRT

- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state

```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)

        Graph.add_edge(q_near, q_rand)

    return Graph
```

Efficient graph generation / RRT

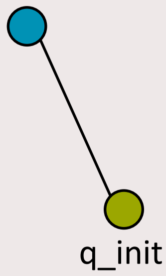
- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state

```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

  Graph.init(q_init)

  for k = 1 to K:
    q_rand ← RAND_CONF(Δq)
    q_near ← NEAREST_VERTEX(q_rand, Graph)

    Graph.add_edge(q_near, q_rand)

  return Graph
```

q_init

Efficient graph generation / RRT

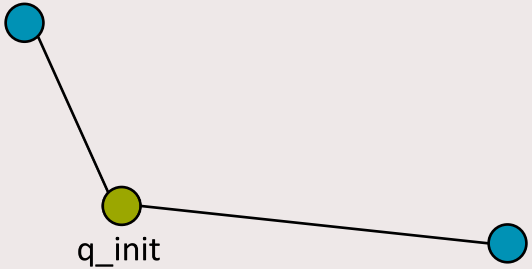
- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

   Graph.init(q_init)

   for k = 1 to K:
      q_rand ← RAND_CONF(Δq)
      q_near ← NEAREST_VERTEX(q_rand, Graph)

      Graph.add_edge(q_near, q_rand)

   return Graph
```

Efficient graph generation / RRT

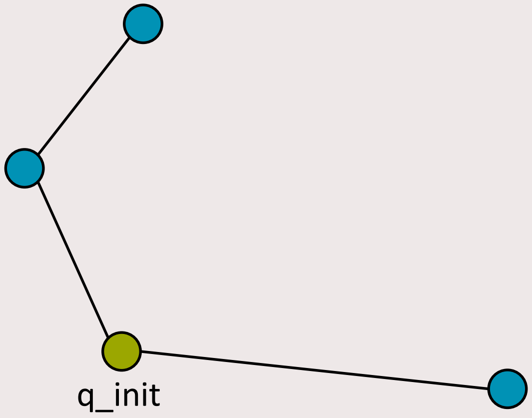
- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

  Graph.init(q_init)

  for k = 1 to K:
    q_rand ← RAND_CONF(Δq)
    q_near ← NEAREST_VERTEX(q_rand, Graph)

    Graph.add_edge(q_near, q_rand)

  return Graph
```

Efficient graph generation / RRT

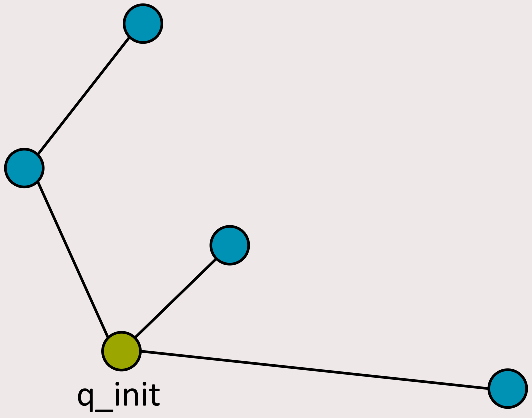
- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)

        Graph.add_edge(q_near, q_rand)

    return Graph
```

Efficient graph generation / RRT

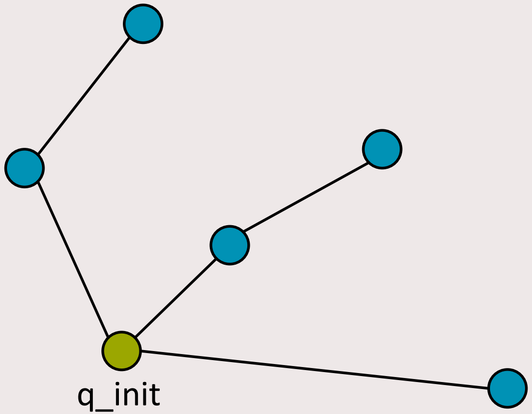
- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)

        Graph.add_edge(q_near, q_rand)

    return Graph
```

Efficient graph generation / RRT

- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state



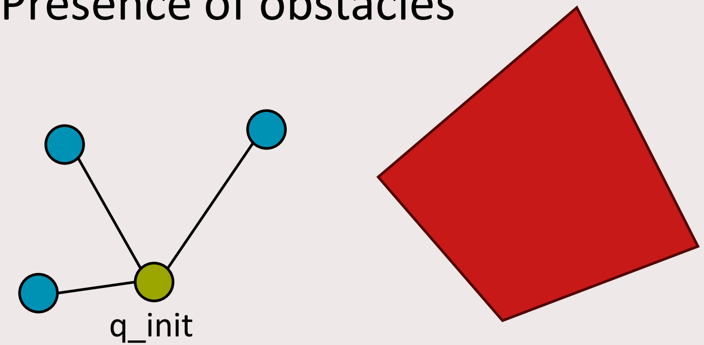

```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)

        Graph.add_edge(q_near, q_rand)

    return Graph
```

Efficient graph generation / RRT

- RRT = Rapidly-exploring Random Tree
- Construct a tree by random sampling, starting at the initial state
- Tree is dense in the limit

Efficient graph generation / RRT

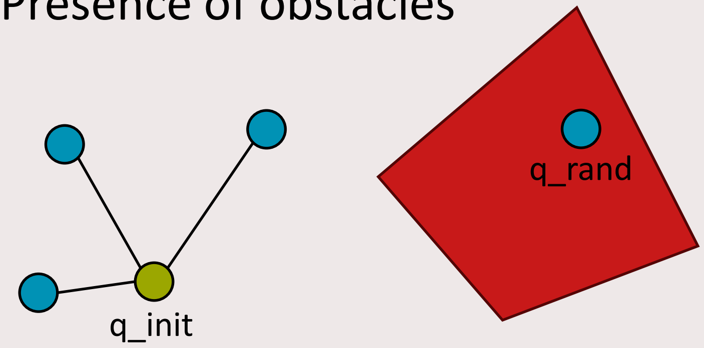
- Presence of obstacles




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

  Graph.init(q_init)

  for k = 1 to K:
    q_rand ← RAND_CONF(Δq)
    q_near ← NEAREST_VERTEX(q_rand, Graph)
    q_new ← STOPPING_CONFIG(q_near, q_rand)
    if q_new != q_near:
      Graph.add_edge(q_near, q_new)

  return Graph
```

Efficient graph generation / RRT

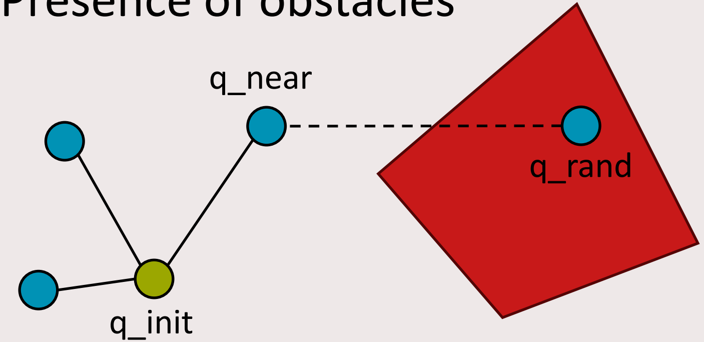
- Presence of obstacles




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)
        q_new ← STOPPING_CONFIG(q_near, q_rand)
        if q_new != q_near:
            Graph.add_edge(q_near, q_new)

    return Graph
```

Efficient graph generation / RRT

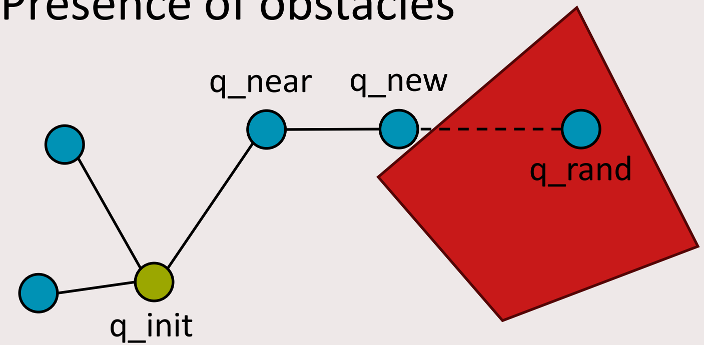
- Presence of obstacles




```
function BuildRRT(q_init,   // Initial configuration
                  K,        // number of vertices
                  Δq):      // incremental distance

   Graph.init(q_init)

   for k = 1 to K:
      q_rand ← RAND_CONF(Δq)
      q_near ← NEAREST_VERTEX(q_rand, Graph)
      q_new ← STOPPING_CONFIG(q_near, q_rand)
      if q_new != q_near:
         Graph.add_edge(q_near, q_new)

   return Graph
```

Efficient graph generation / RRT

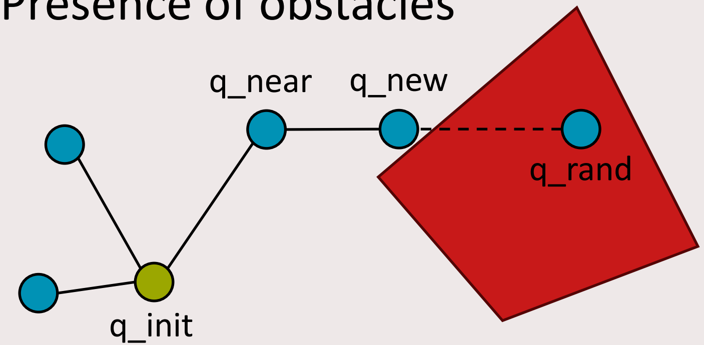
- Presence of obstacles




```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

  Graph.init(q_init)

  for k = 1 to K:
    q_rand ← RAND_CONF(Δq)
    q_near ← NEAREST_VERTEX(q_rand, Graph)
    q_new ← STOPPING_CONFIG(q_near, q_rand)
    if q_new != q_near:
      Graph.add_edge(q_near, q_new)

  return Graph
```

Efficient graph generation / RRT

- Presence of obstacles

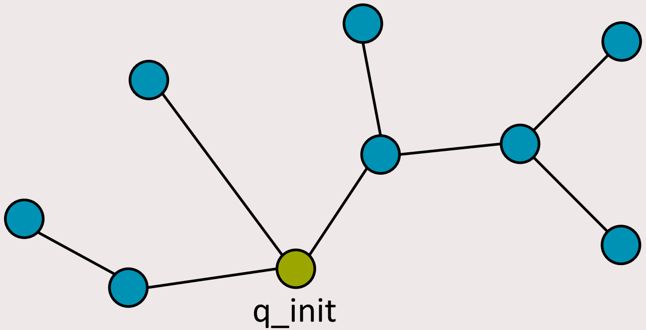


- We can stop if we can add the goal node to the tree

```
function BuildRRT(q_init,    // Initial configuration
                  K,         // number of vertices
                  Δq):       // incremental distance

    Graph.init(q_init)

    for k = 1 to K:
        q_rand ← RAND_CONF(Δq)
        q_near ← NEAREST_VERTEX(q_rand, Graph)
        q_new ← STOPPING_CONFIG(q_near, q_rand)
        if q_new != q_near:
            Graph.add_edge(q_near, q_new)

    return Graph
```

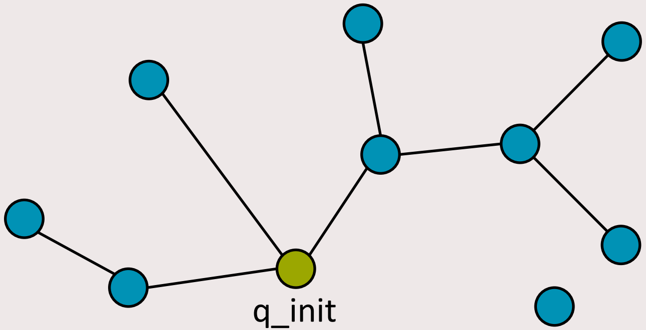
Efficient graph generation / RRT*

- RRT*: RRT with rewiring for shorter paths
  - Similar to Dijkstra and A*



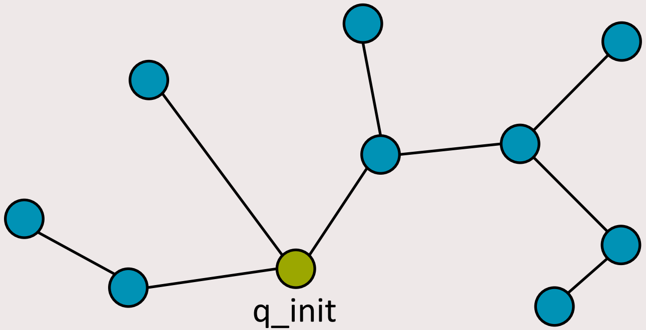
Efficient graph generation / RRT*

- RRT*: RRT with rewiring for shorter paths
  - Similar to Dijkstra and A*



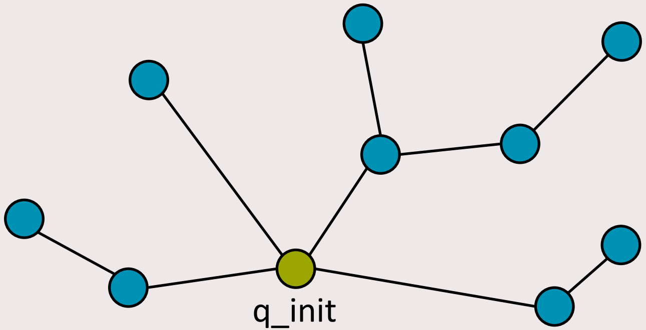
Efficient graph generation / RRT*

- RRT*: RRT with rewiring for shorter paths
  - Similar to Dijkstra and A*



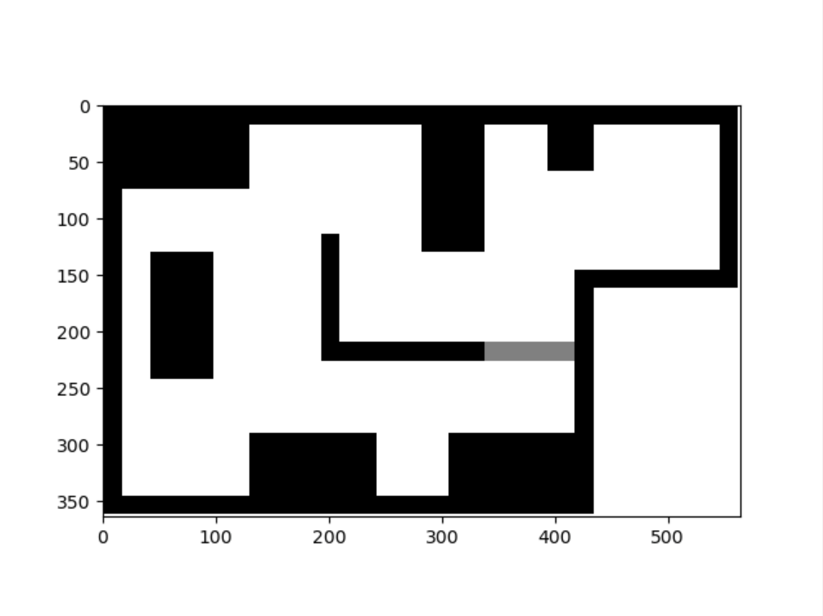
Efficient graph generation / RRT*

- RRT*: RRT with rewiring for shorter paths
  - Similar to Dijkstra and A*



Efficient graph generation / Probabilistic roadmap

- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes



Efficient graph generation / Probabilistic roadmap

- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes

```
function Generate_PRM(Map, N_vertices):

  G.init()

  for i = 0 to N_vertices:
    c ← a free configuration in Map
    G.add_vertex(c)

    for each q in neighbours(c, G):
      if connect(c, q):
        G.add_edge(c, q)
```

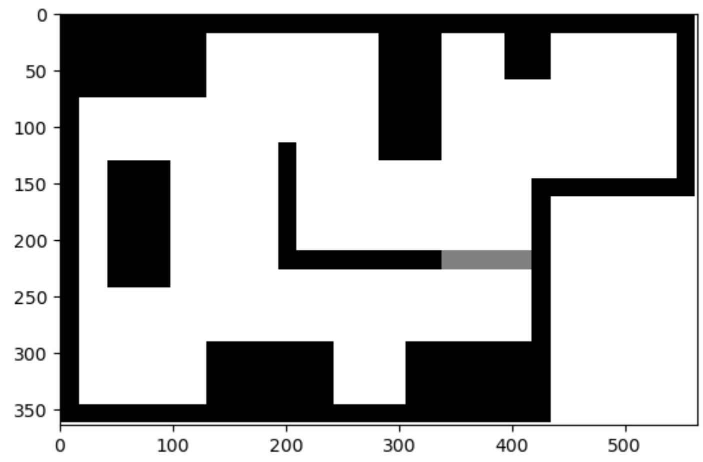
Efficient graph generation / Probabilistic roadmap

- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes
- *Note: Different strategies of sampling, neighborhood or connections might be more appropriate*

```
function Generate_PRM(Map, N_vertices):

    G.init()

    for i = 0 to N_vertices:
        c ← a free configuration in Map
        G.add_vertex(c)

        for each q in neighbours(c, G):
            if connect(c, q):
                G.add_edge(c, q)
```

Efficient graph generation / Probabilistic roadmap

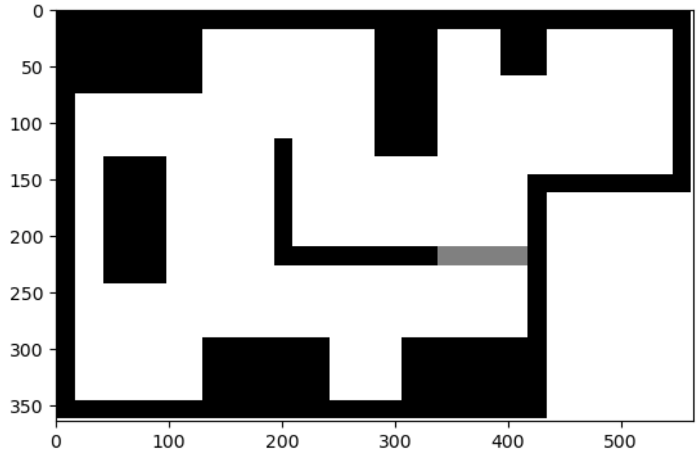
- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes
- *Note: Different strategies of sampling, neighborhood or connections might be more appropriate*



TU/e

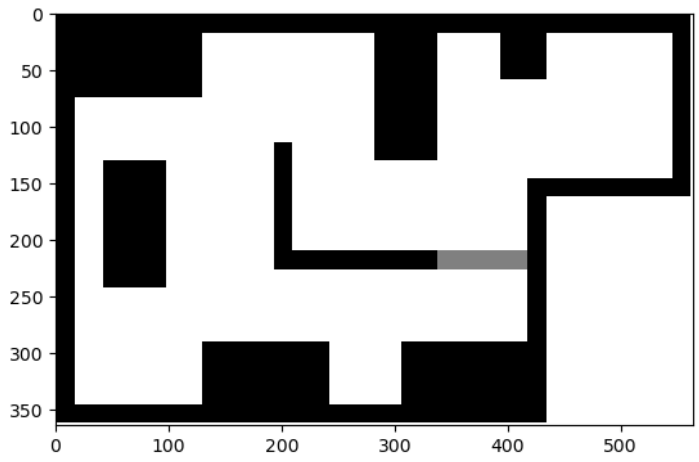
Efficient graph generation / Probabilistic roadmap

- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes
- *Note: Different strategies of sampling, neighborhood or connections might be more appropriate*
- Planning: e.g. Dijkstra or A*



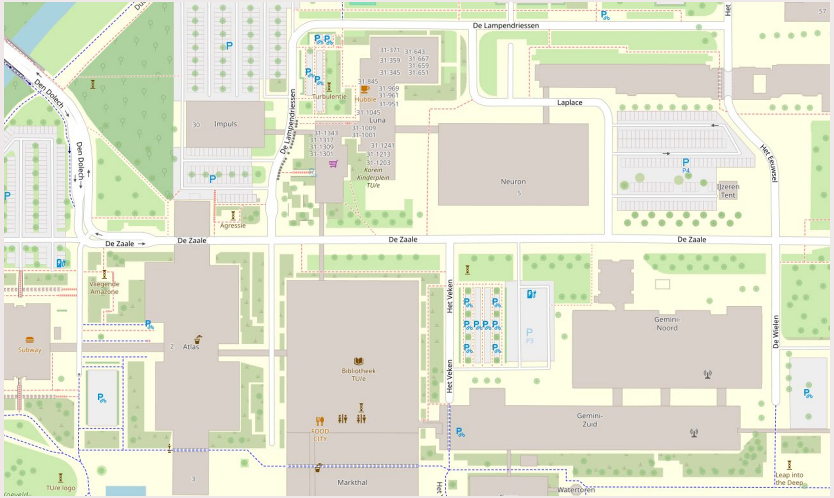
Efficient graph generation / Probabilistic roadmap

- Nodes: randomly generated (valid) configurations
- Edges: (straight-line) collision-free connections between nodes
- *Note: Different strategies of sampling, neighborhood or connections might be more appropriate*
- Planning: e.g. Dijkstra or A*
- Roadmap can be independent of start and final configurations



Following a global path with a local planner

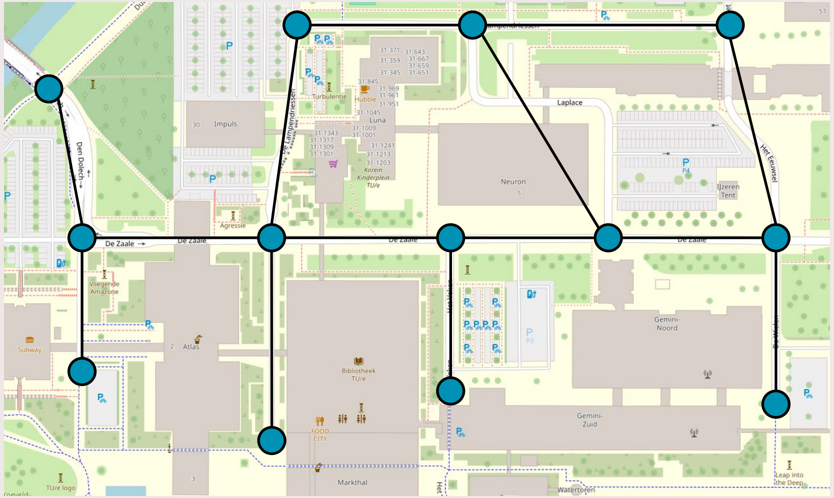
- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)



Source: https://www.openstreetmap.org/

Following a global path with a local planner

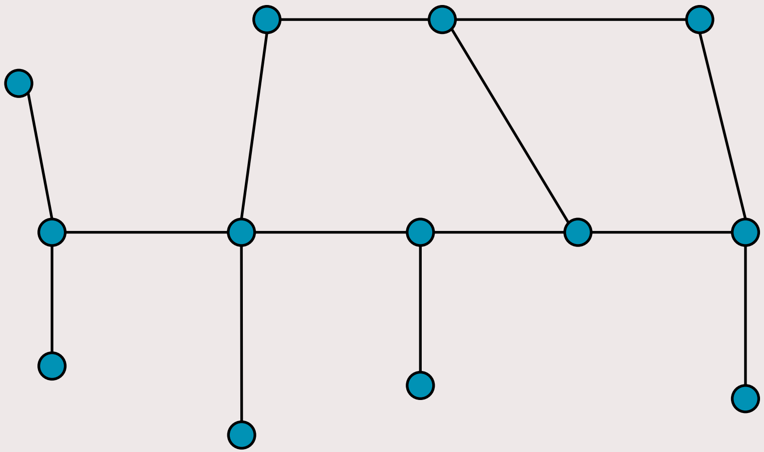
- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)



Source: https://www.openstreetmap.org/

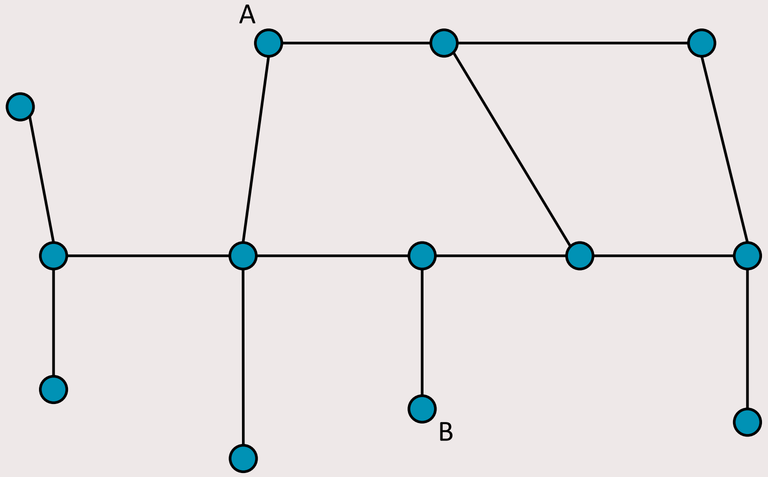
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)



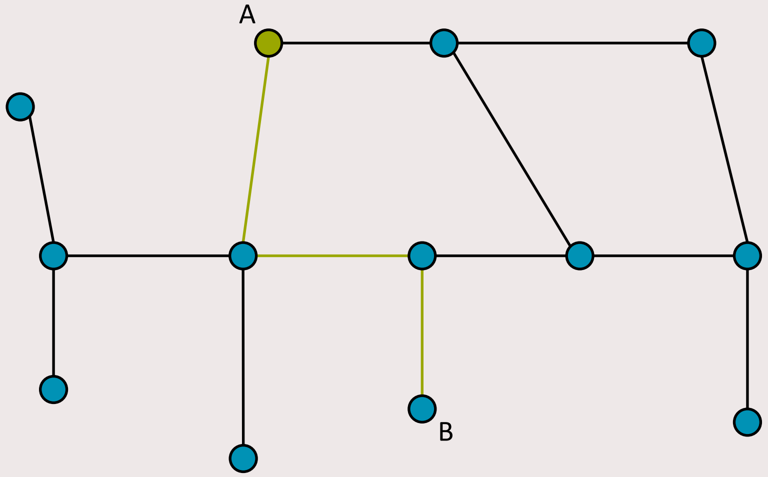
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)



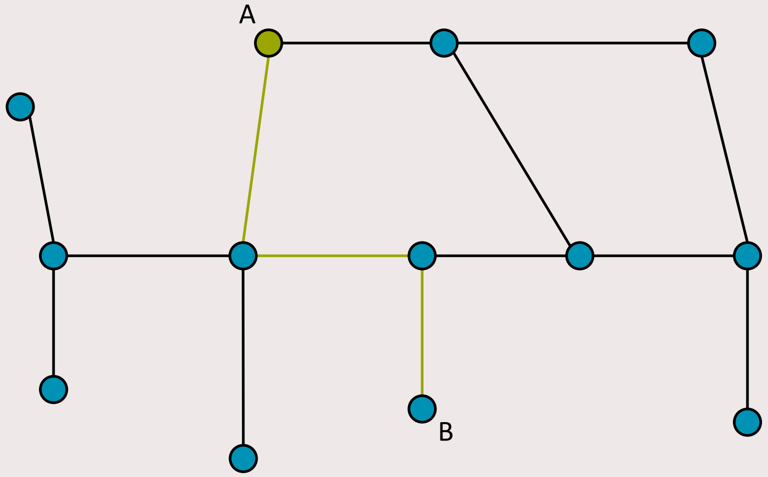
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)



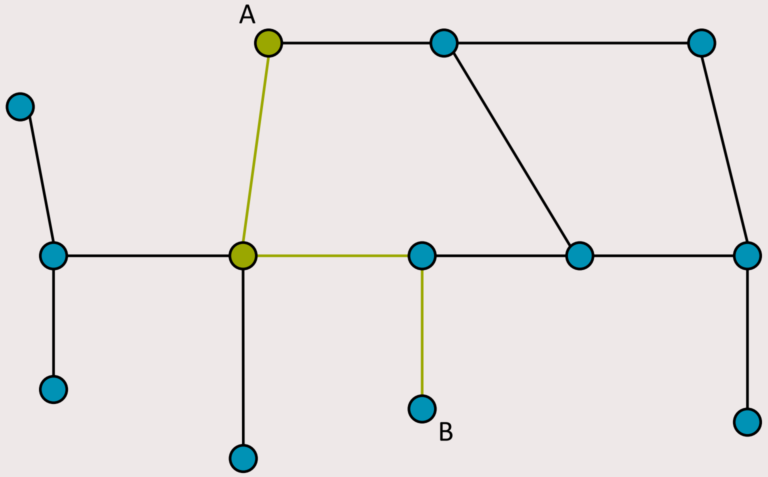
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still



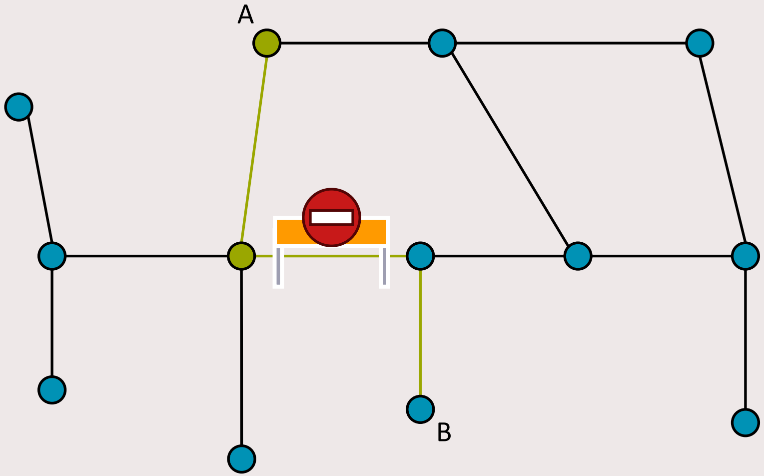
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still



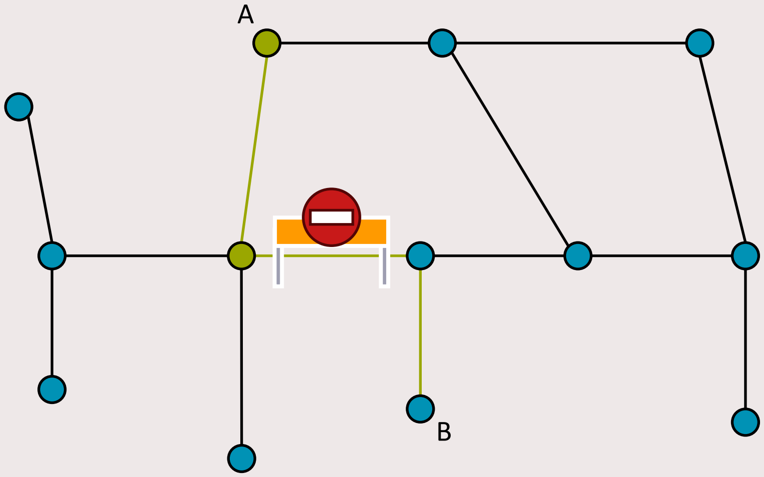
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still



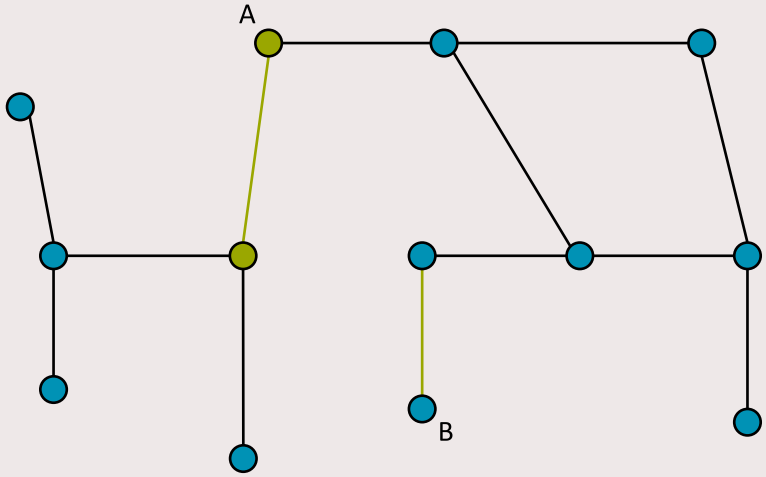
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still
- Update worldmodel/graph and replan



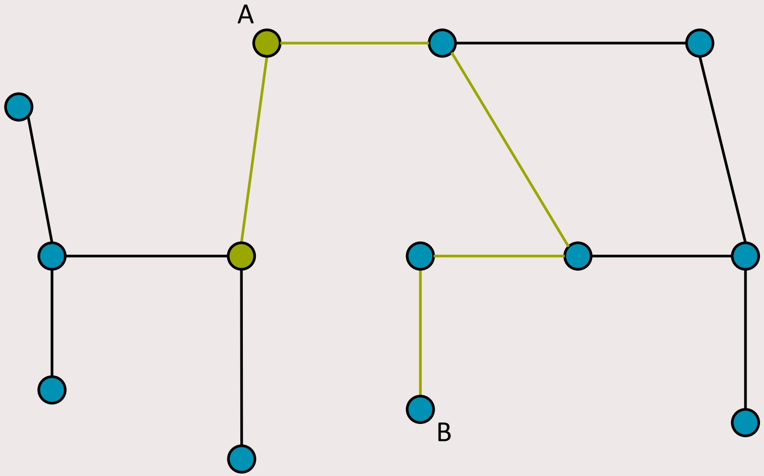
Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still
- Update worldmodel/graph and replan



Following a global path with a local planner

- Exact trajectory planning is often infeasible
- Would lead to constant replanning
- Local planner needed (previous lecture)
- Monitor when the plan is blocked
  - Robot stands still
- Update worldmodel/graph and replan



Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

Summary

- Global (vs local) navigation

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph
- Path planning in graphs
  - Dijkstra & A*: complete and optimal

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph
- Path planning in graphs
  - Dijkstra & A*: complete and optimal
- Grids are simple but often inefficient. Alternatives:

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph
- Path planning in graphs
  - Dijkstra & A*: complete and optimal
- Grids are simple but often inefficient. Alternatives:
  - Visibility graph: short paths, has to be recomputed when map is updated

Summary

- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph
- Path planning in graphs
  - Dijkstra & A*: complete and optimal
- Grids are simple but often inefficient. Alternatives:
  - Visibility graph: short paths, has to be recomputed when map is updated
  - RRT(*): creates graph and finds path, but very specific for start location, completeness only guaranteed in the limit

Summary

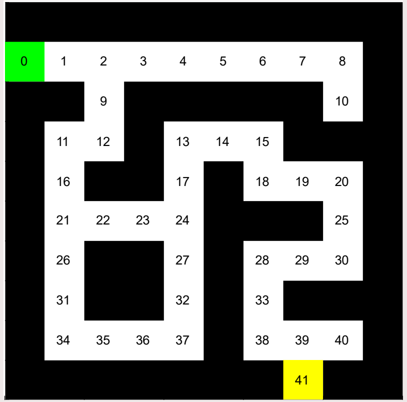
- Global (vs local) navigation
- Map representations
  - Discretizations: Cell-based, grid-based and graph-based
- Discretized map → graph
- Path planning in graphs
  - Dijkstra & A*: complete and optimal
- Grids are simple but often inefficient. Alternatives:
  - Visibility graph: short paths, has to be recomputed when map is updated
  - RRT(*): creates graph and finds path, but very specific for start location, completeness only guaranteed in the limit
  - Probabilistic roadmap: completeness only guaranteed in the limit

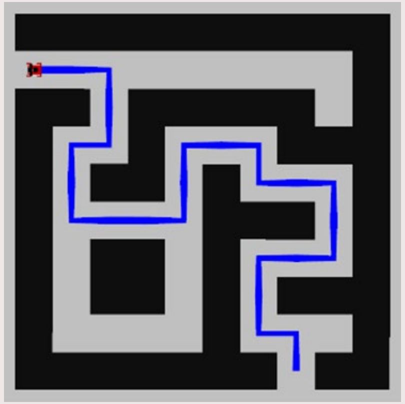
Outline

- Recap local navigation & intro global navigation
- Map representations
- From map to graph
- Path planning in graphs
- Efficient graph generation
- Summary
- Introduction to assignment

Assignment part 1

- Complete an implementation of the A* algorithm to find the shortest path from start to finish in a maze
- Provided: list of nodes and edges, index of start and finish nodes
- Required: sequence of node indices that form the shortest path from start to finish





Assignment part 2

- Complete the implementation of generating a graph that represents a Probabilistic Roadmap (PRM)
- When part 1 of the assignment is also finished, a path from start to goal in this PRM can be found using your A* algorithm

Assignment part 3

- Connect global and local planner