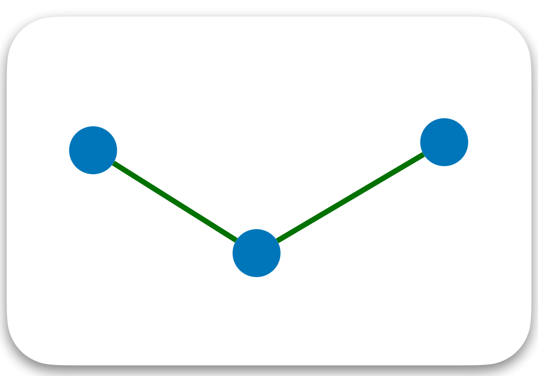
Introduction to graph-based planning

Overview of graph-based planning

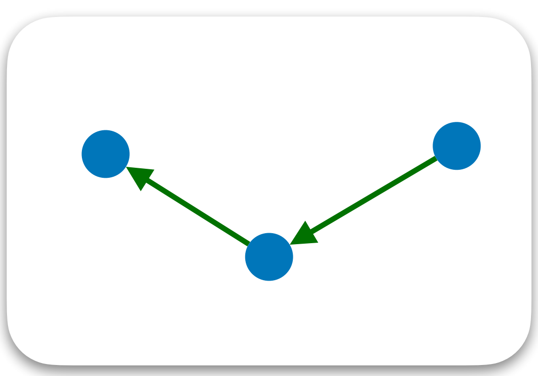
- **Graphs** as the easiest representation for planning.
  - Warning: graph nomenclature varies across fields.
  - Directed graph, labeled graph, multigraph, labeling, path, cycles, cycle bases.
- The basic **graph formulation of**
  - Nodes = states, edges = actions, paths = plans
- Specific examples:
  - **Road networks**
  - **Pose graphs** (useful for SLAM as well)
- **Classic algorithms** for path finding(Dijkstra, A*)

Basic graphs definitions

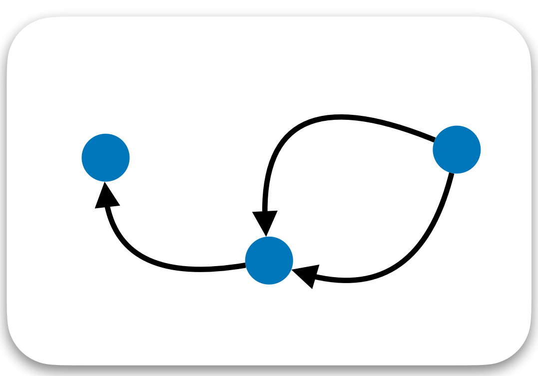
- A **graph** is a set of **nodes** (*vertices*) + a set of **edges.**
- In a **directed graph**, each edge has a direction (it's an arrow).
  - **Source** nodes have only outgoing edges, **sinks** have only incoming edges.
- In a **multigraph**, there might be multiple edges between two nodes with the same direction.



graph



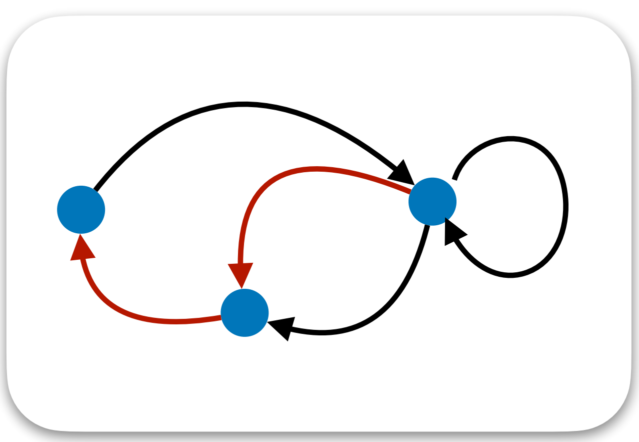
directed graph



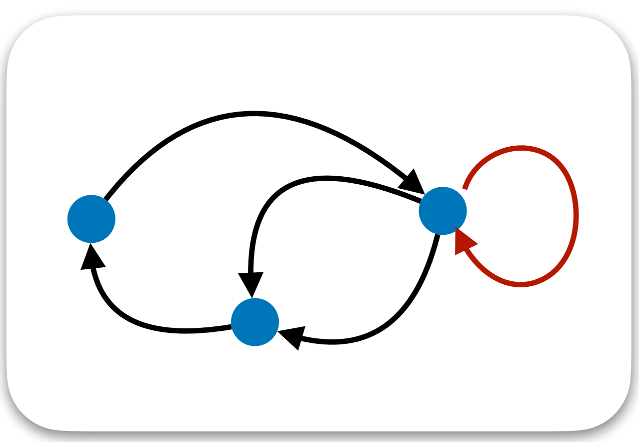
directed multigraph

Loops, paths, cycles

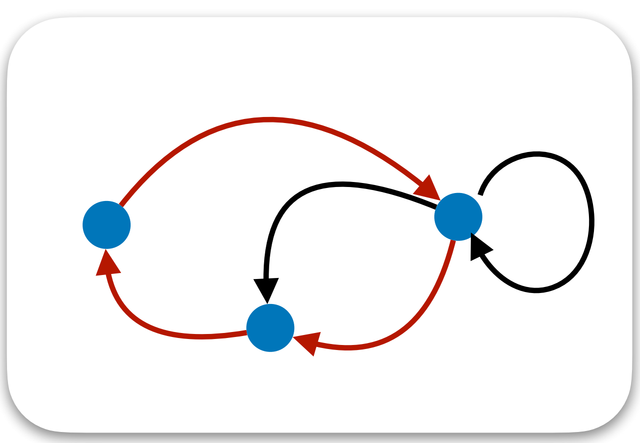
- A **path** is a sequence of consecutive edges.
- A **loop** is an edge for which the starting and ending node are the same.
- A **cycle** is a path that starts and ends with the same node.



path



loop

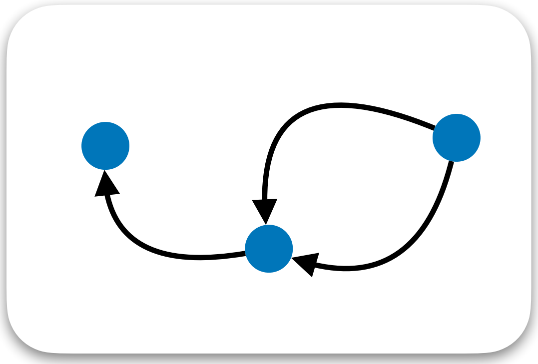


cycle

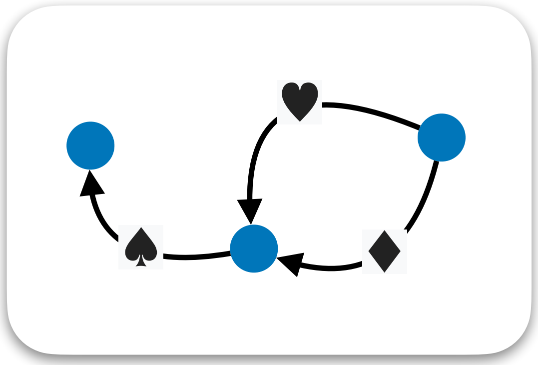
Labeled graphs

- A graph is **labeled** if there is some extra information (the “label”) associated to the edges and/or the nodes.

unlabeled



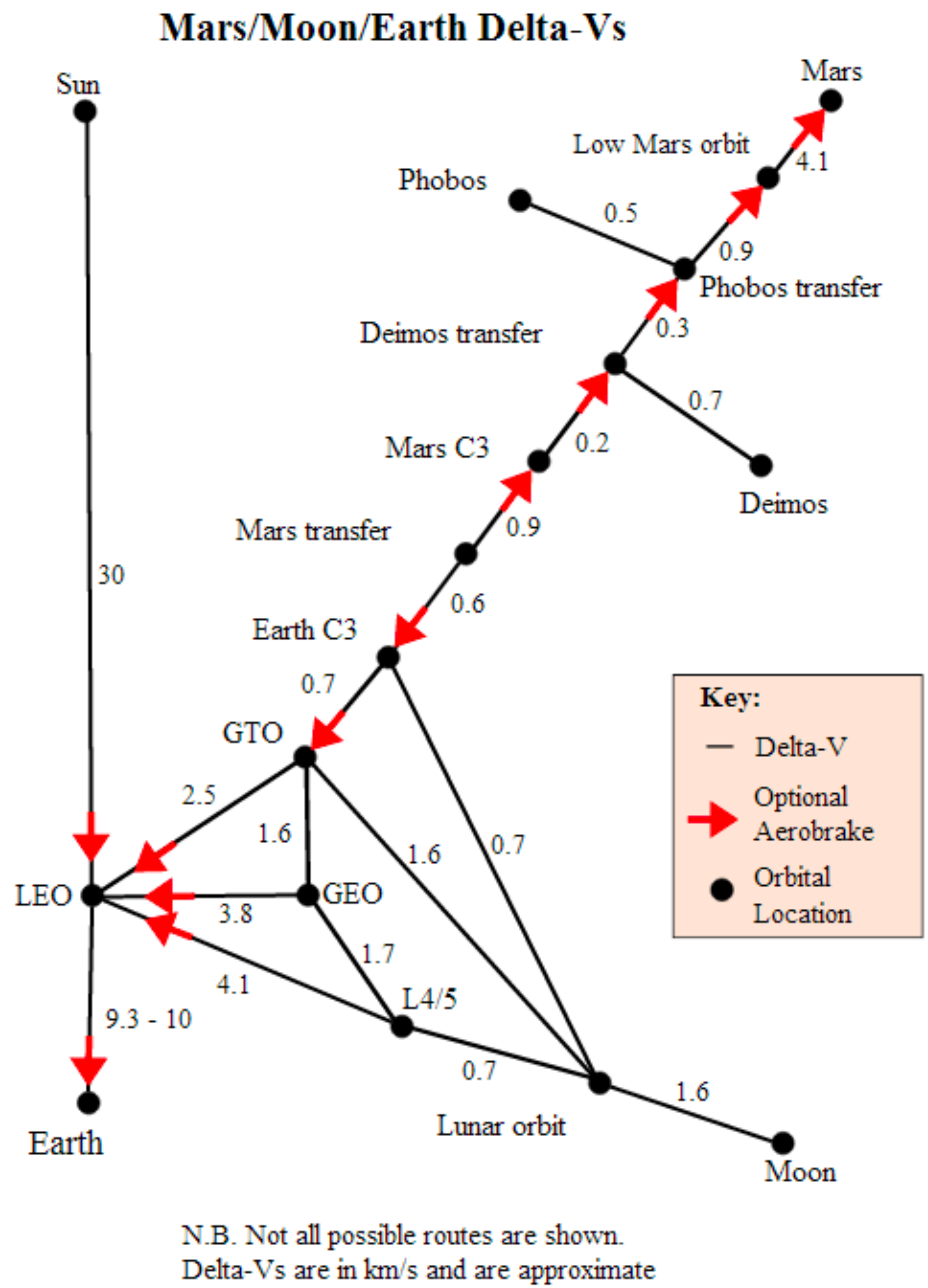
labeled



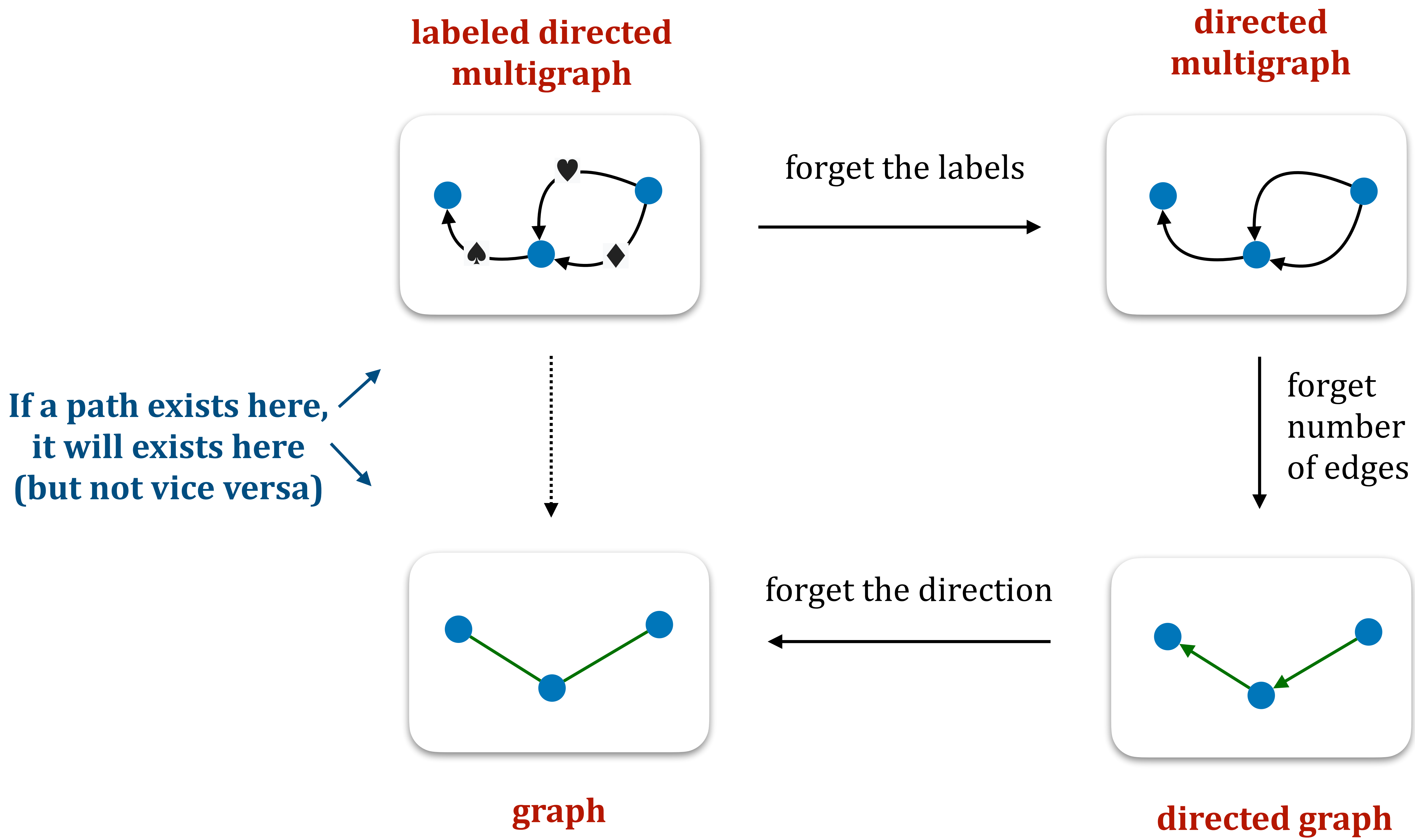
- In planning, edge labels are used to describe **costs.**
- Example: nodes = places; label = the length of a trip from place to place.

Example: Delta V maps

- Nodes = orbits.
- Edges = transfers.
- Labels = cost in “Delta V”.
    - ~= fuel to be spent

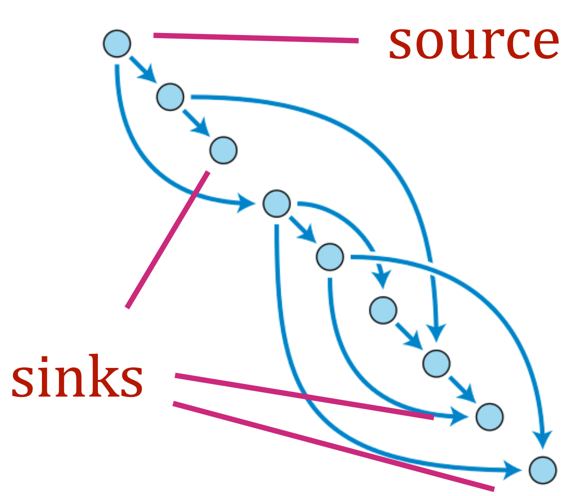


Forgetful graph transformations

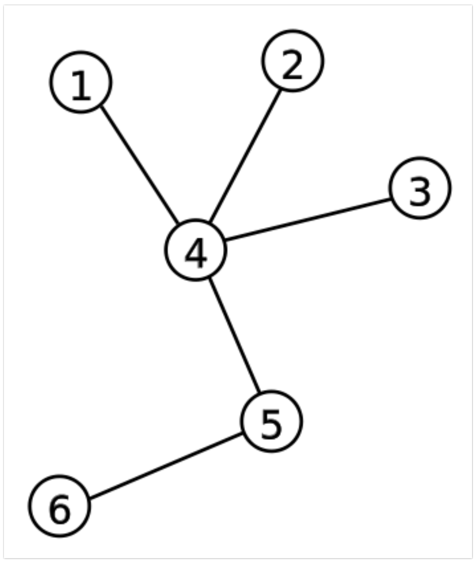


Special graphs

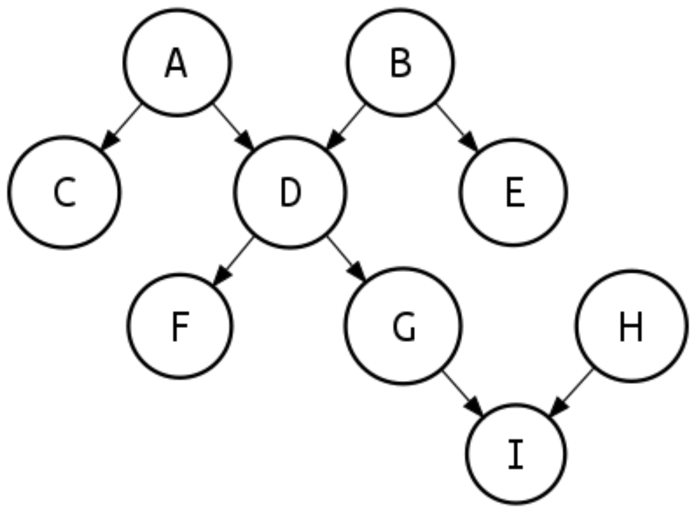
- **Directed acyclic graph (DAG)**: a graph that contains no cycles.
- **Tree:** Connected acyclic undirected graph
- **Polytree**: A directed graph whose underlying undirected graph is a tree.



DAG



Tree

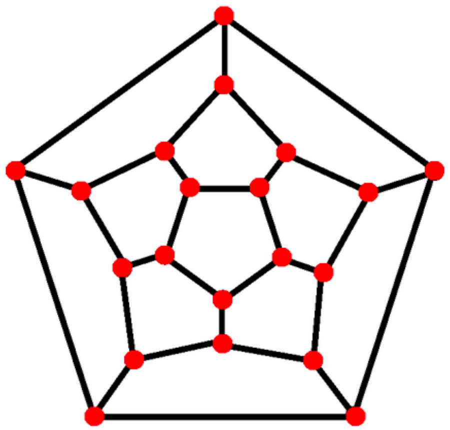


Polytree

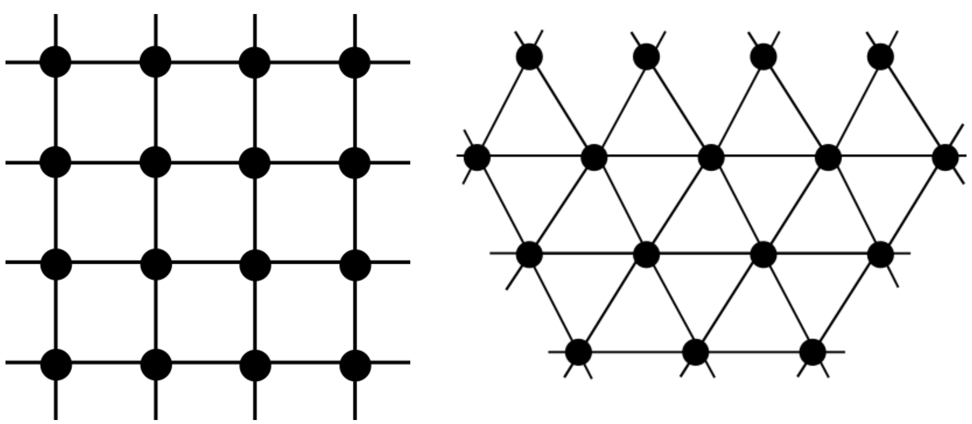
Planar graphs

- **Planar graph**: A graph that can be "embedded in the plane", in the sense that it can be drawn without any intersection of edges.

planar graphs

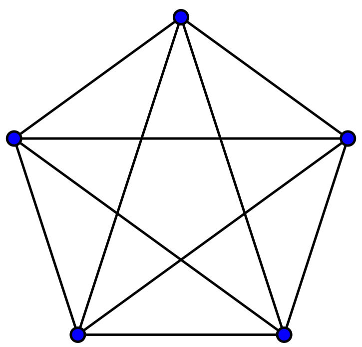


dodecahedron



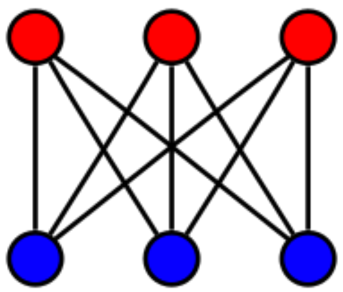
"tilings"

non-planar graphs



pentagon

K5



"utility graph"

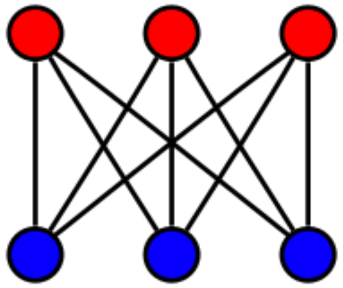
K3,3

Kuratowski's theorem: if a graph is nonplanar, it's because it contains a copy of K3,3 or K5.

The utilities problem

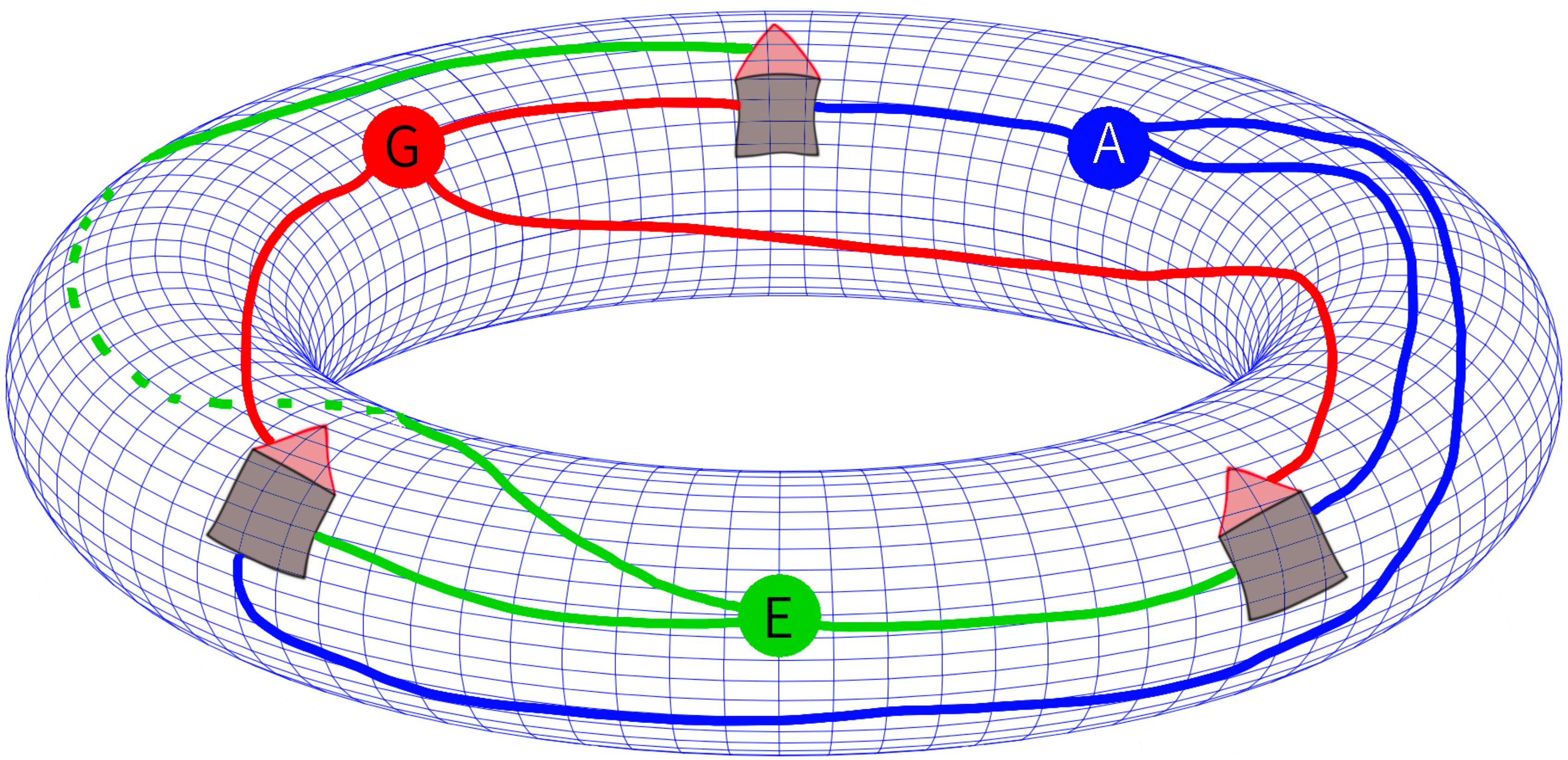
- Can you connect each house to each utility without any wires crossing?

- Answer: no, because K3,3 is not a planar graph



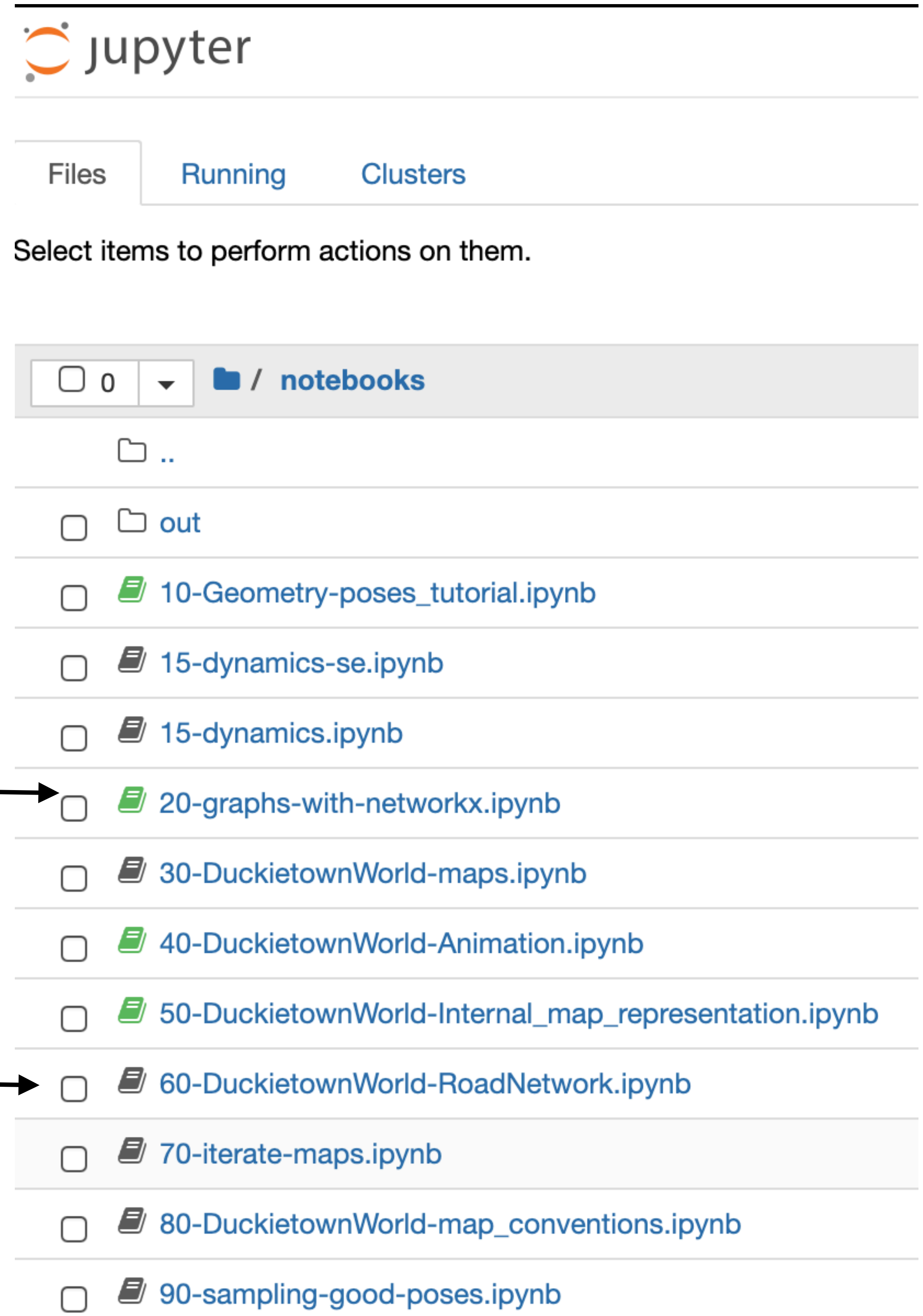
The utilities problem

- Yes, if you live on a torus.



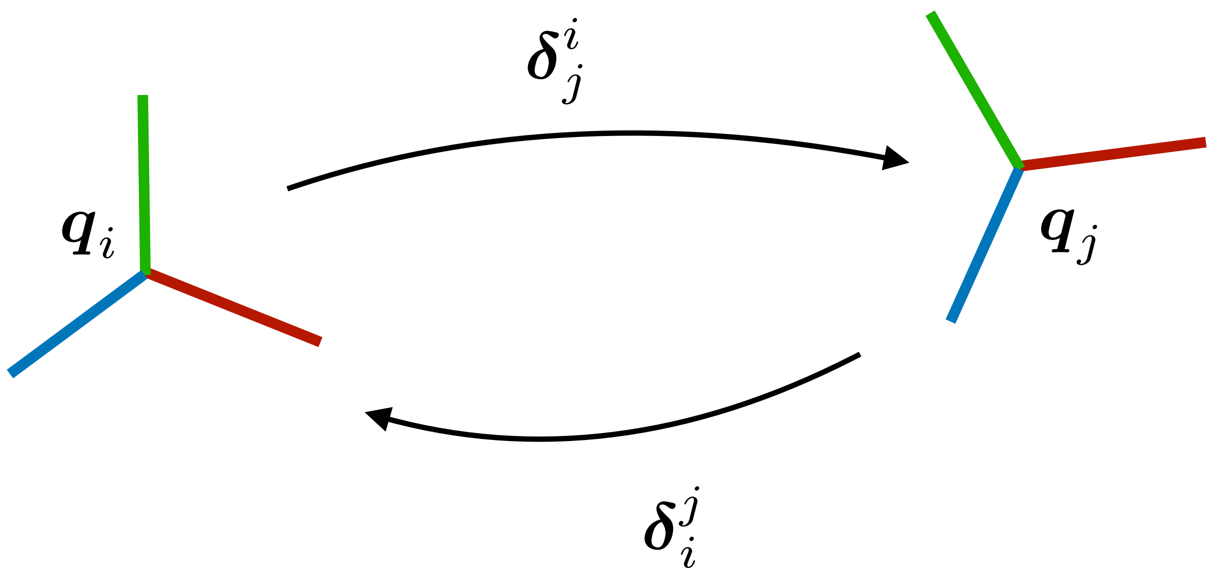
Pose graph notebook

- In **duckietown-world** there are several notebooks available.
- They show how to represent and manipulate Duckietown maps, road networks, pose graphs, etc.



Pose graph

- A **pose graph** is a graph where:
  - each **node** is labeled with a **pose** $\boldsymbol{q} \in \mathsf{SE}(3)$
  - each **edge** is labeled with the **difference between two poses** $\boldsymbol{\delta}_j^i = (\boldsymbol{q}_i)^{-1}\boldsymbol{q}_j \in \mathsf{SE}(3)$



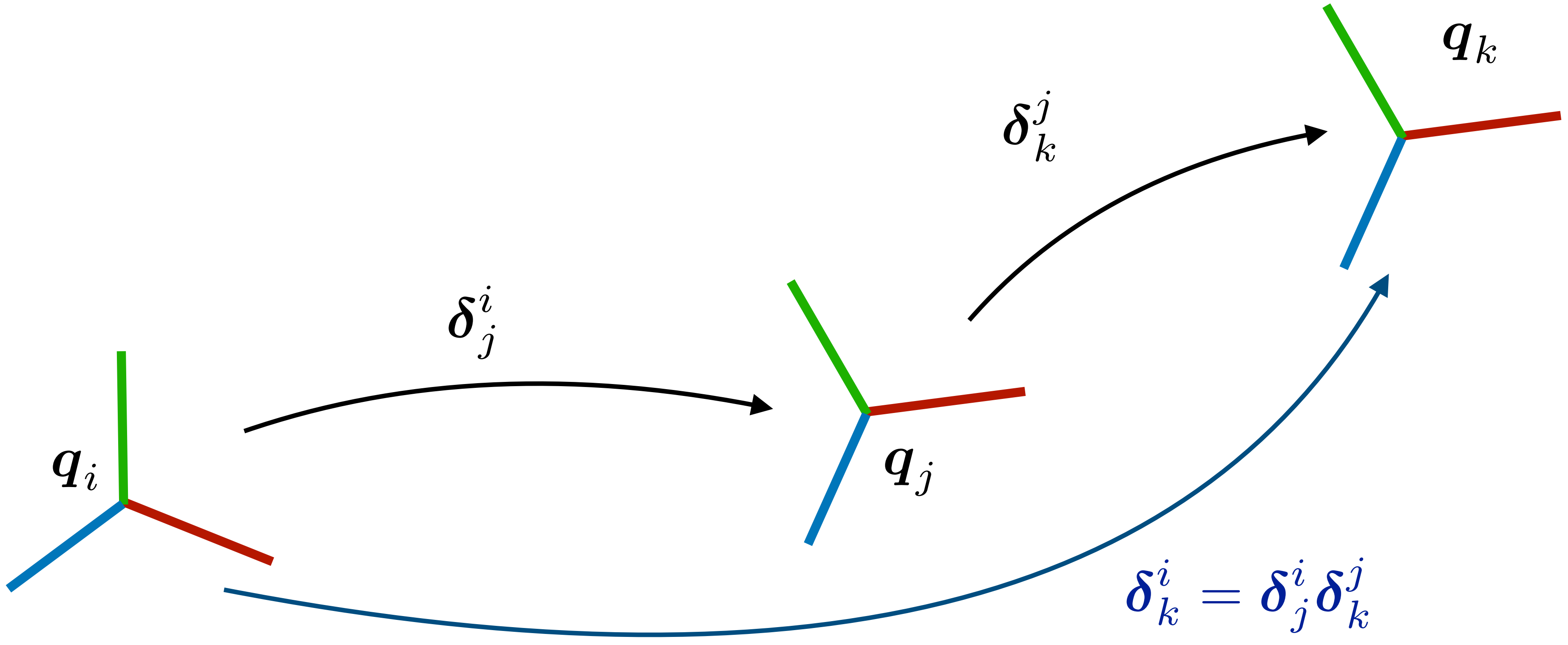
implications of definition:

$$\boldsymbol{\delta}_j^i = (\boldsymbol{\delta}_i^j)^{-1}$$

$$\boldsymbol{\delta}_i^i = \mathsf{Id}$$

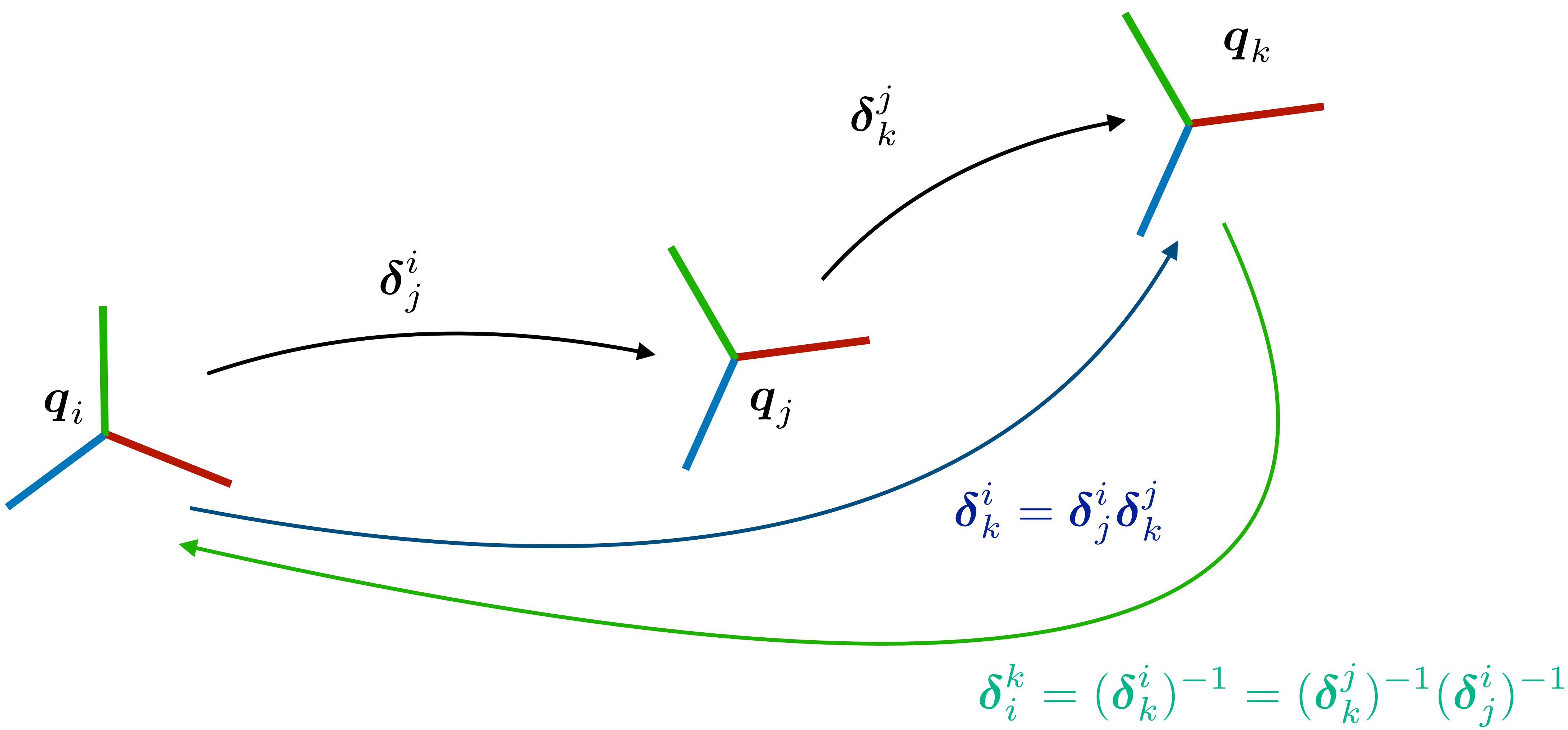
Pose graph

- If there is an edge from *i* to *j* and one from *j* to *k*, you can derive an edge from *i* to *k*.
- Compose the differences: $\boldsymbol{\delta}_k^i = \boldsymbol{\delta}_j^i \boldsymbol{\delta}_k^j$



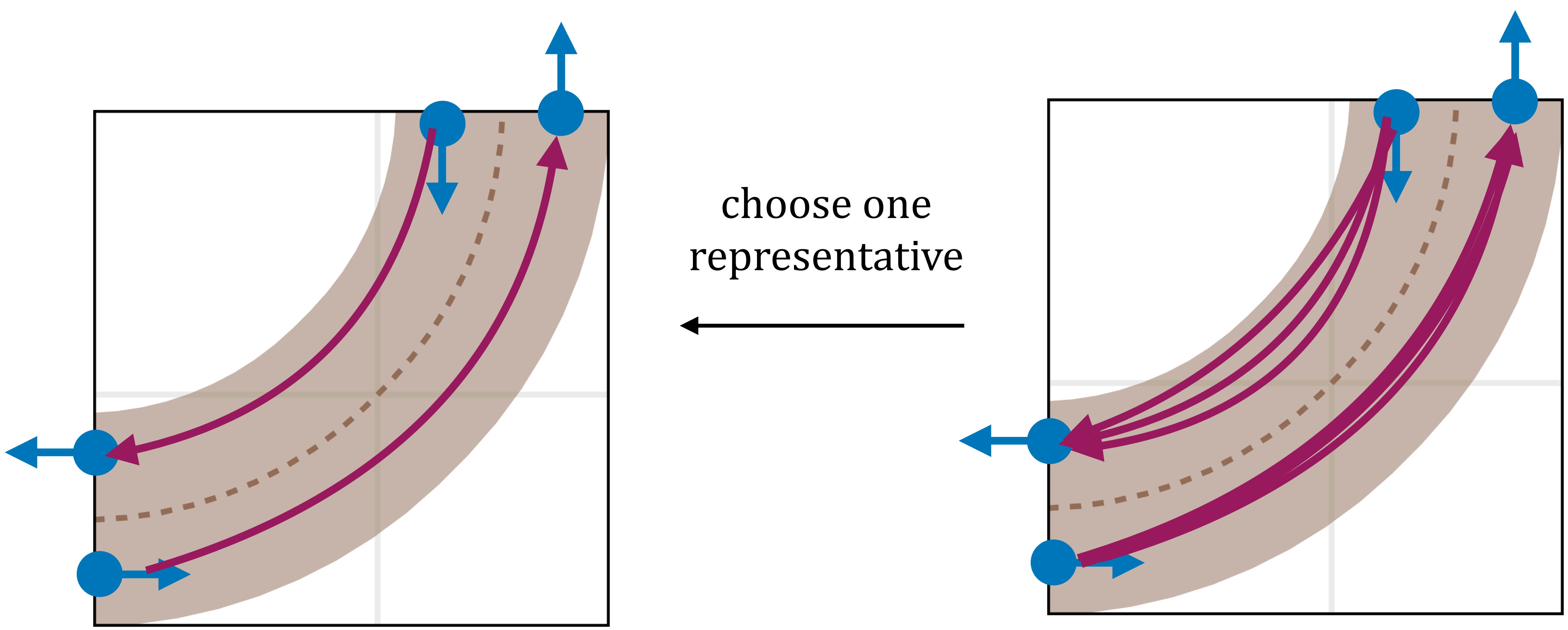
Cycles in pose graphs

- **Cycles represent constraints:**
  - **The product of the pose differences along a cycle is the identity.**
  - Think first: the sum of the position difference is 0,0,0. Generalize to SE(3).



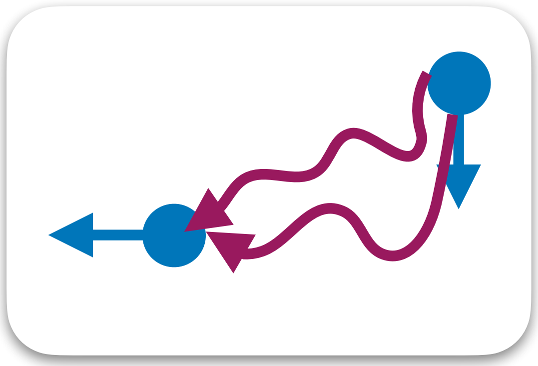
Road networks

- A **road network** is a directed graph where:
  - the **nodes** are "canonical" Duckiebot poses
  - the **edges** are representative trajectories to go from one pose to another



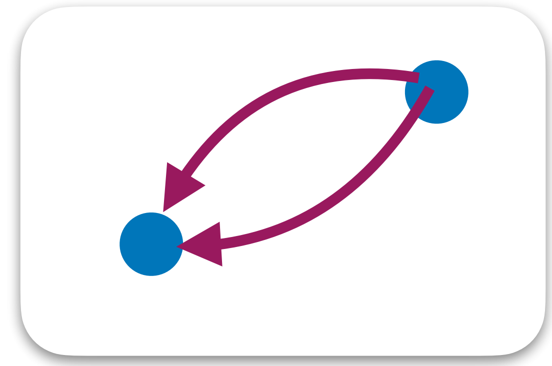
Forgetful graph transformations

road network with pose information

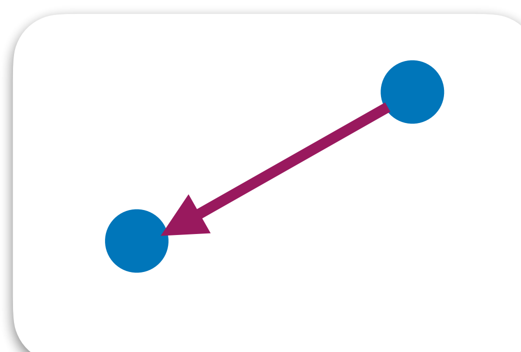


forget the poses

roads graph (with multiple edges)



compress all edges between two nodes into one

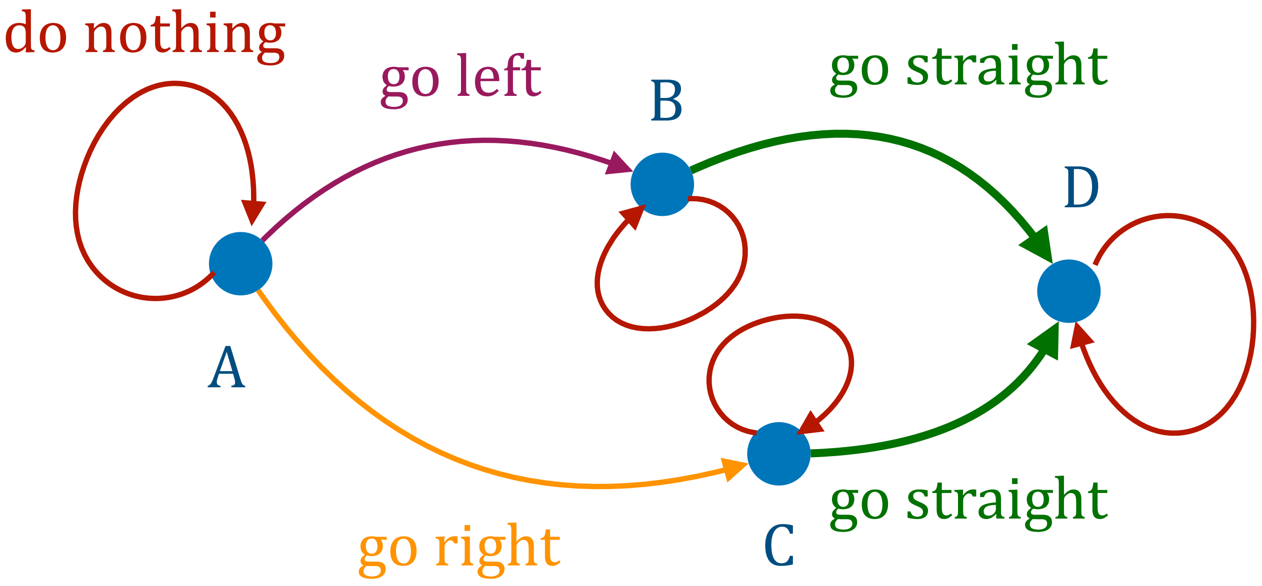


connectivity graph

If a path exists here, it will exists here (but not vice versa)

Graph-based planning

- The basic graph-based abstract formalization of a planning problem:
  - Each **node** is a **state**.
  - Each **edge** represents the effect of an **action**.
  - Edges are **labeled** with the **cost** of actions.



- Notice:
  - no uncertainty for evolution
  - no uncertainty for observations
  - no continuous dynamics

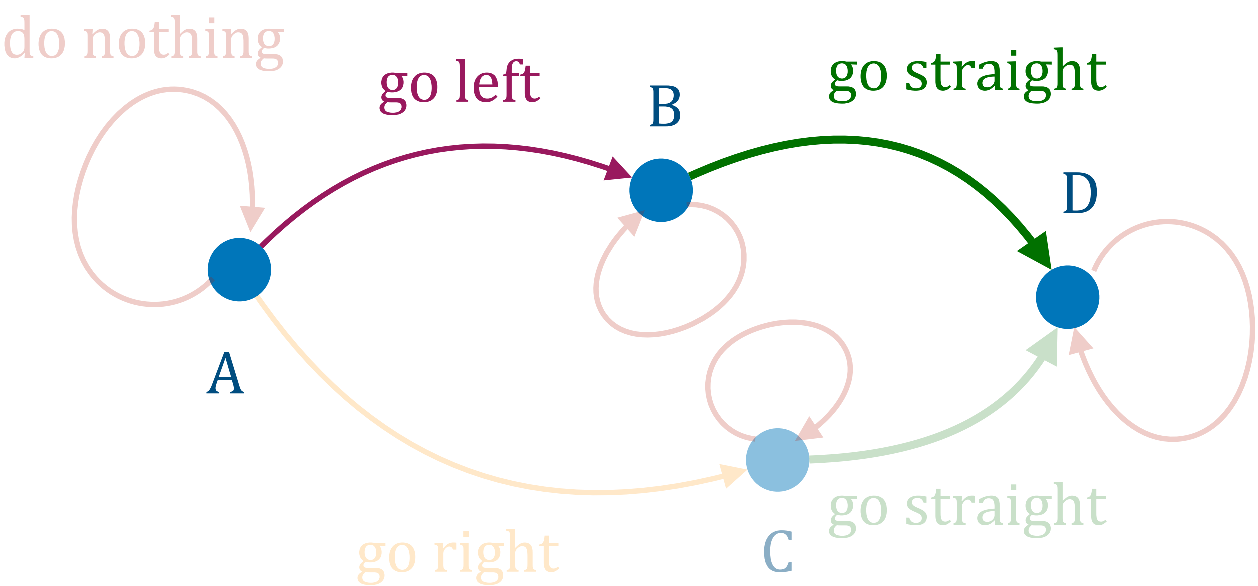
Graph-based planning

- The basic graph-based abstract formalization of a planning problem:
  - Each **node** is a **state**.
  - Each **edge** represents the effect of an **action**.
  - Edges are **labeled** with the **cost** of actions.

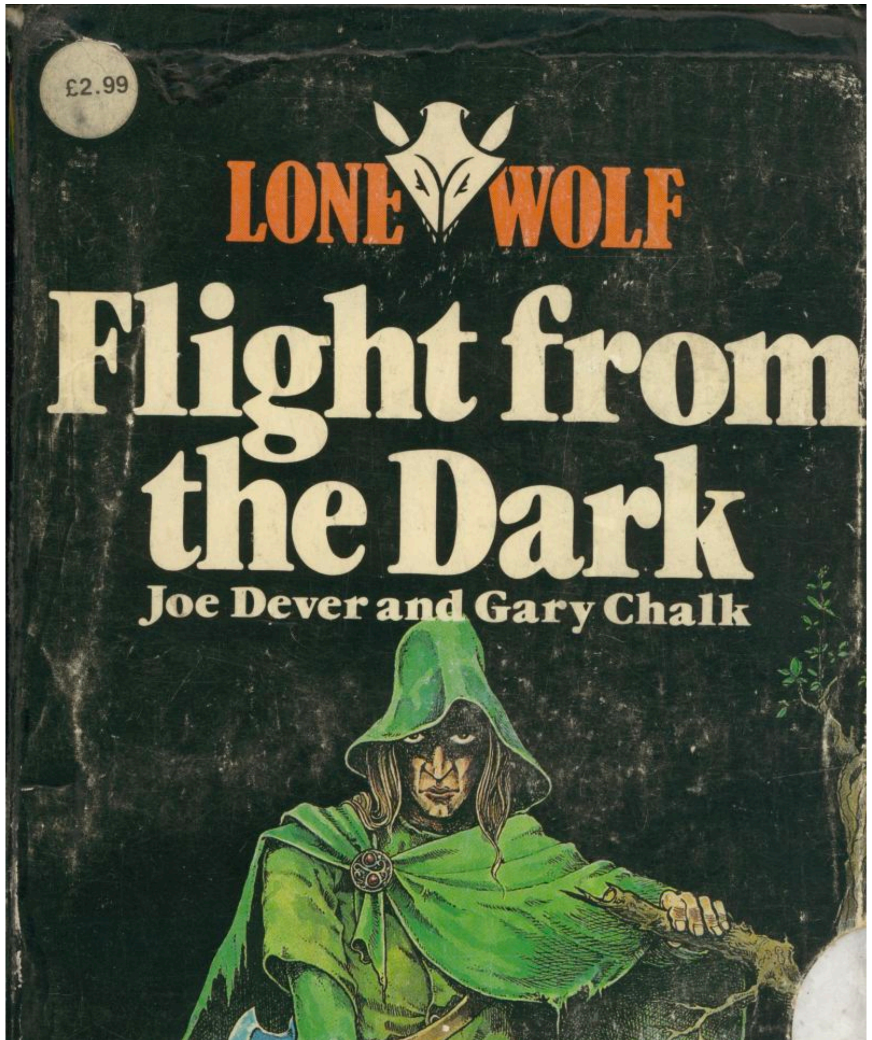
- **Shortest path finding**:

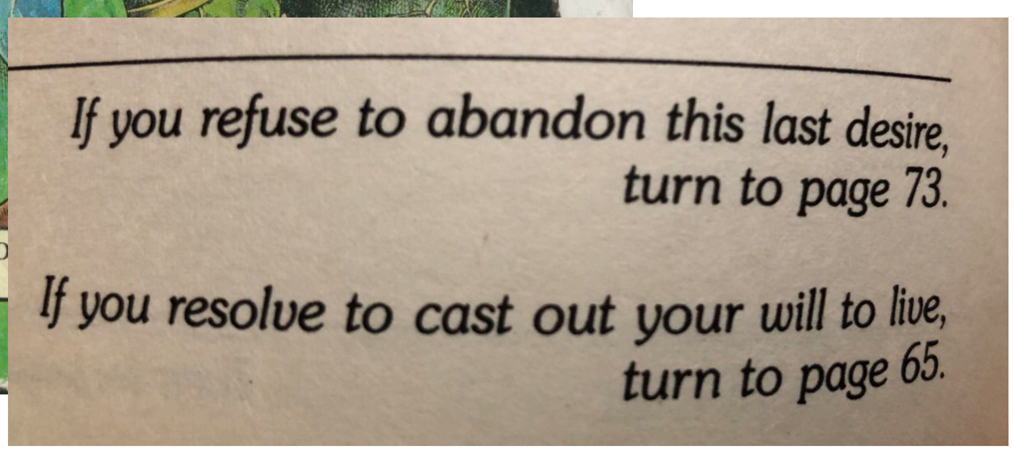
  Find a way to get from a node to another in the least amount of steps (least total cost).

  - A **path** is a **plan**.



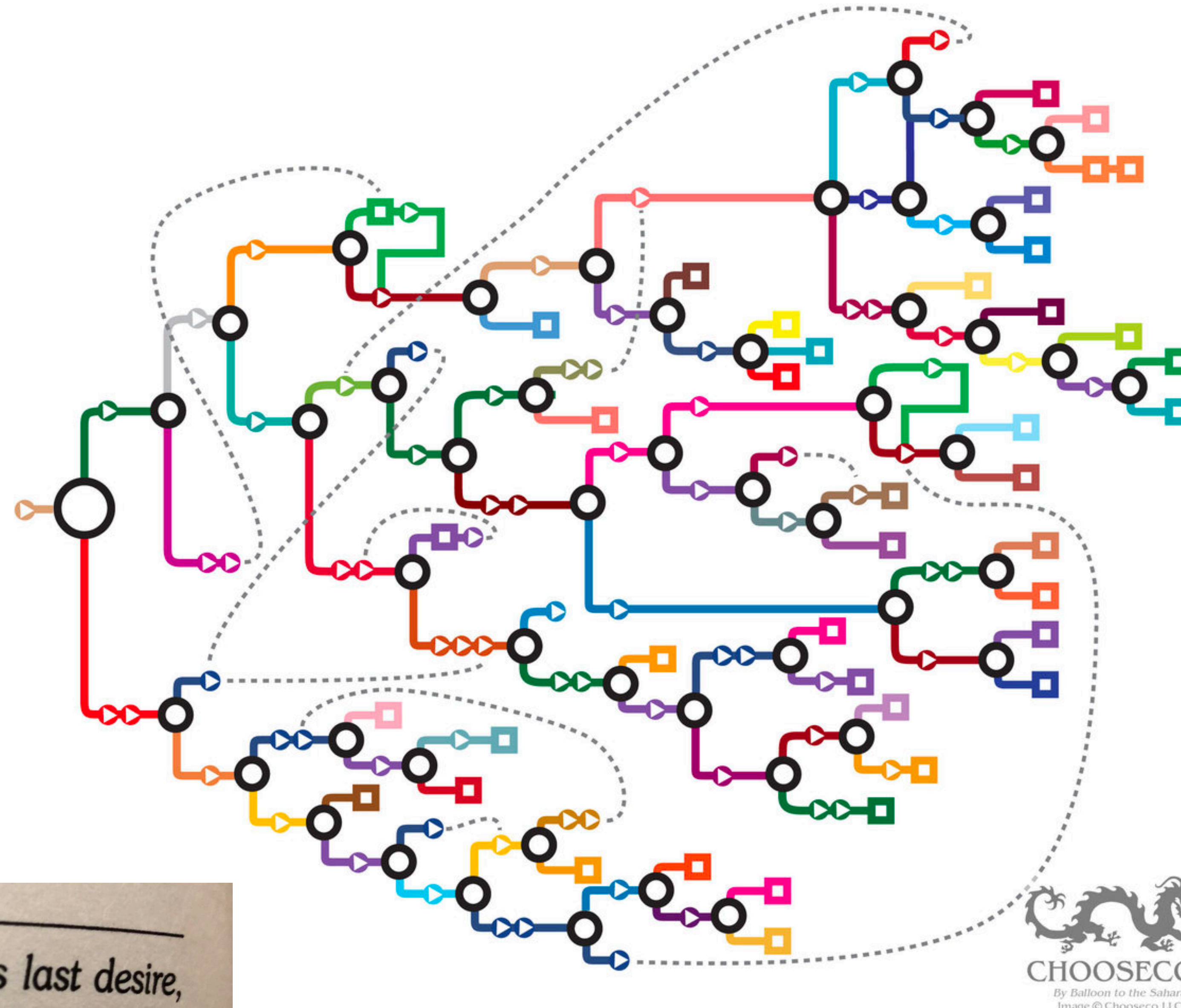
Example from the 70s: choose your own adventure books





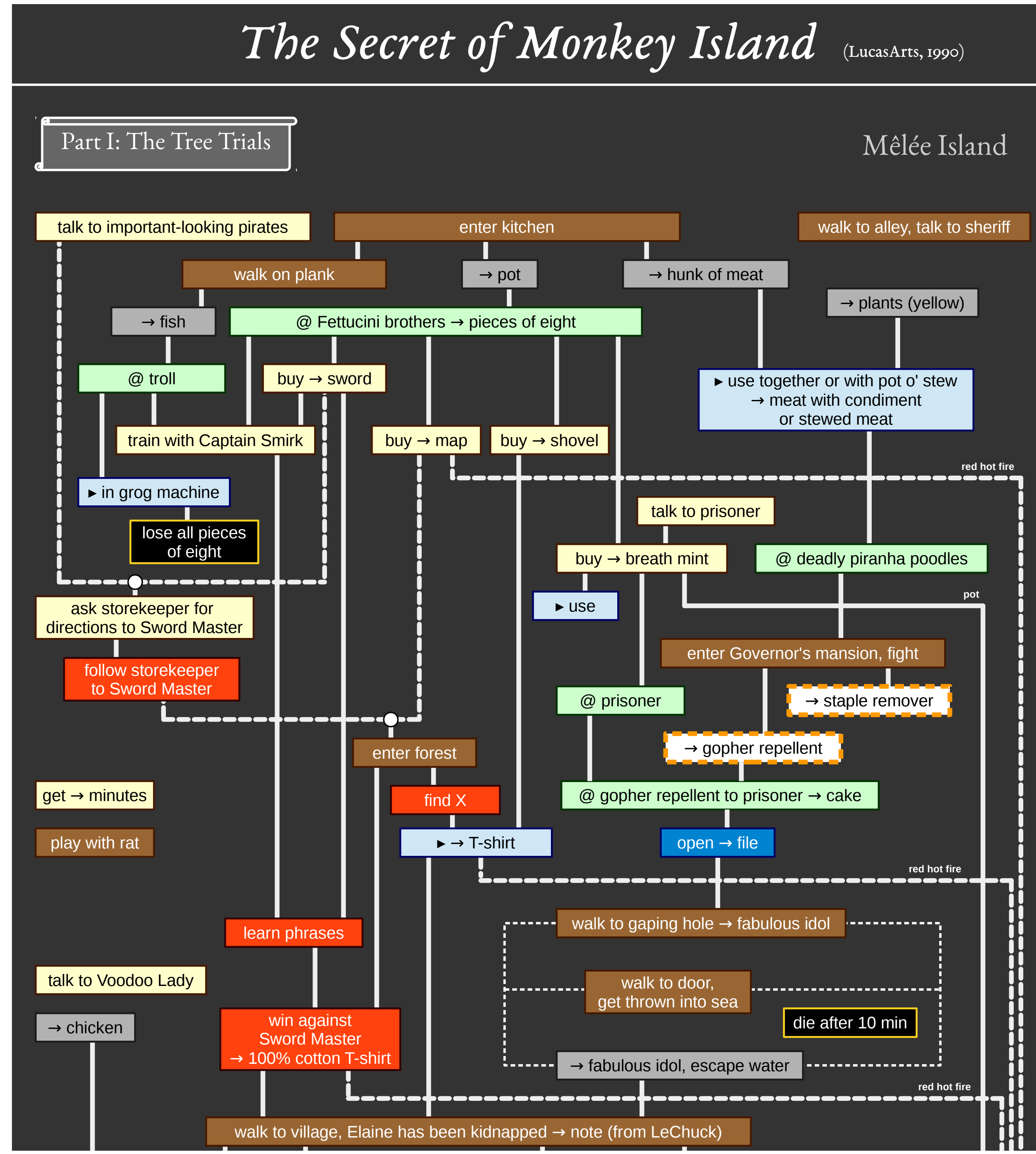
If you refuse to abandon this last desire,
turn to page 73.

If you resolve to cast out your will to live,
turn to page 65.



Example from the 80s: Puzzle dependencies charts

- Used to design puzzle games.



- **Liveness property**: there is a path from each node to one of the endings.

- That is, you cannot get stuck.

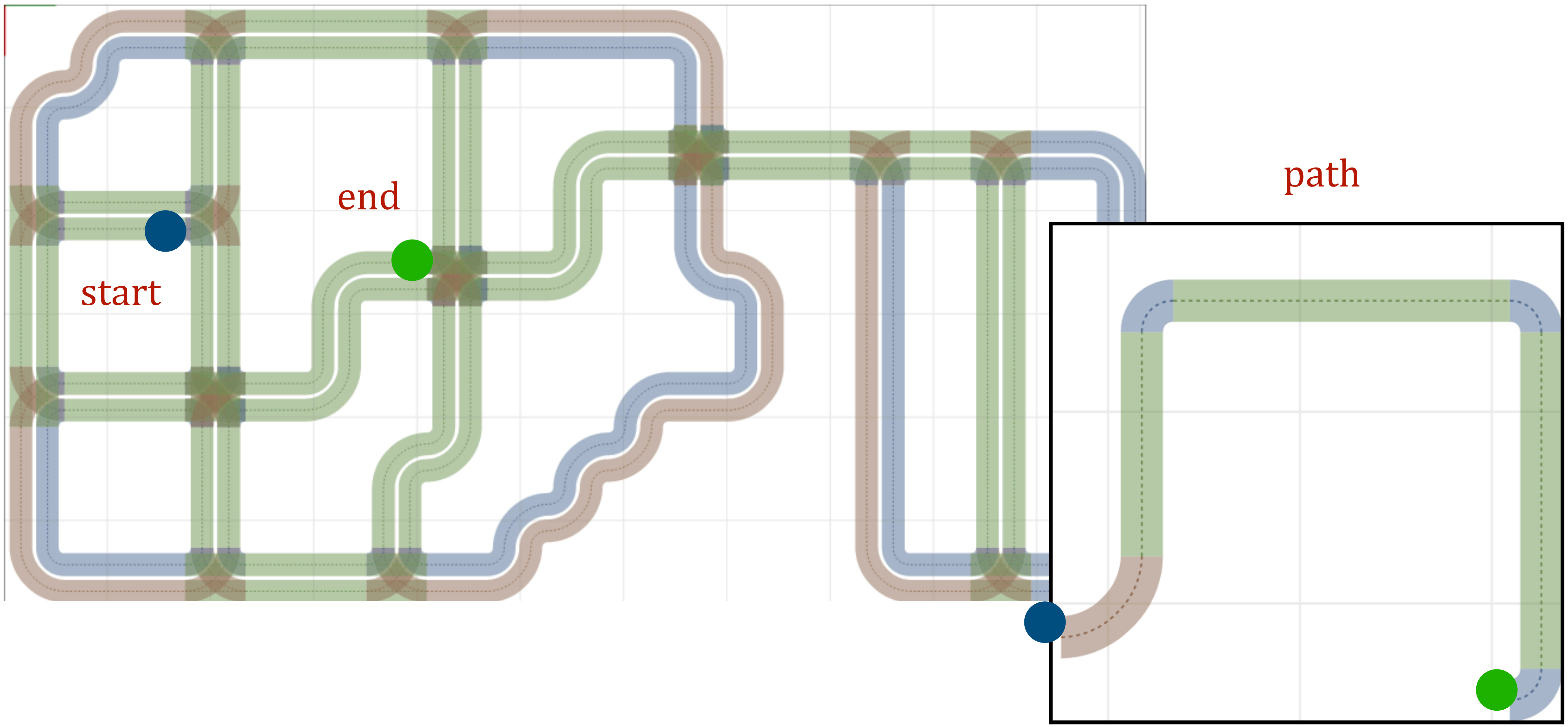
https://grumpygamer.com/puzzle_dependency_charts

How to solve a path finding problem

- There are **some classical algorithms** that you will find implemented in all graph libraries. In practice:
  - If your graph is small (<10,000 nodes), you're done.
  - If your graph is planar (2D, regular cells), you're done.
- **Dijkstra's algorithm**: find the shortest path from one node to every other node
  - In robotics, use the reverse: find the shortest path from any node **to** the goal.
- **A* ("A star")**: uses a heuristics to drive the search.
  - Very efficient, single query.
- **D***: variation of A* that allows replanning when the graph changes
  - Developed for field robotics (Stentz, CMU)

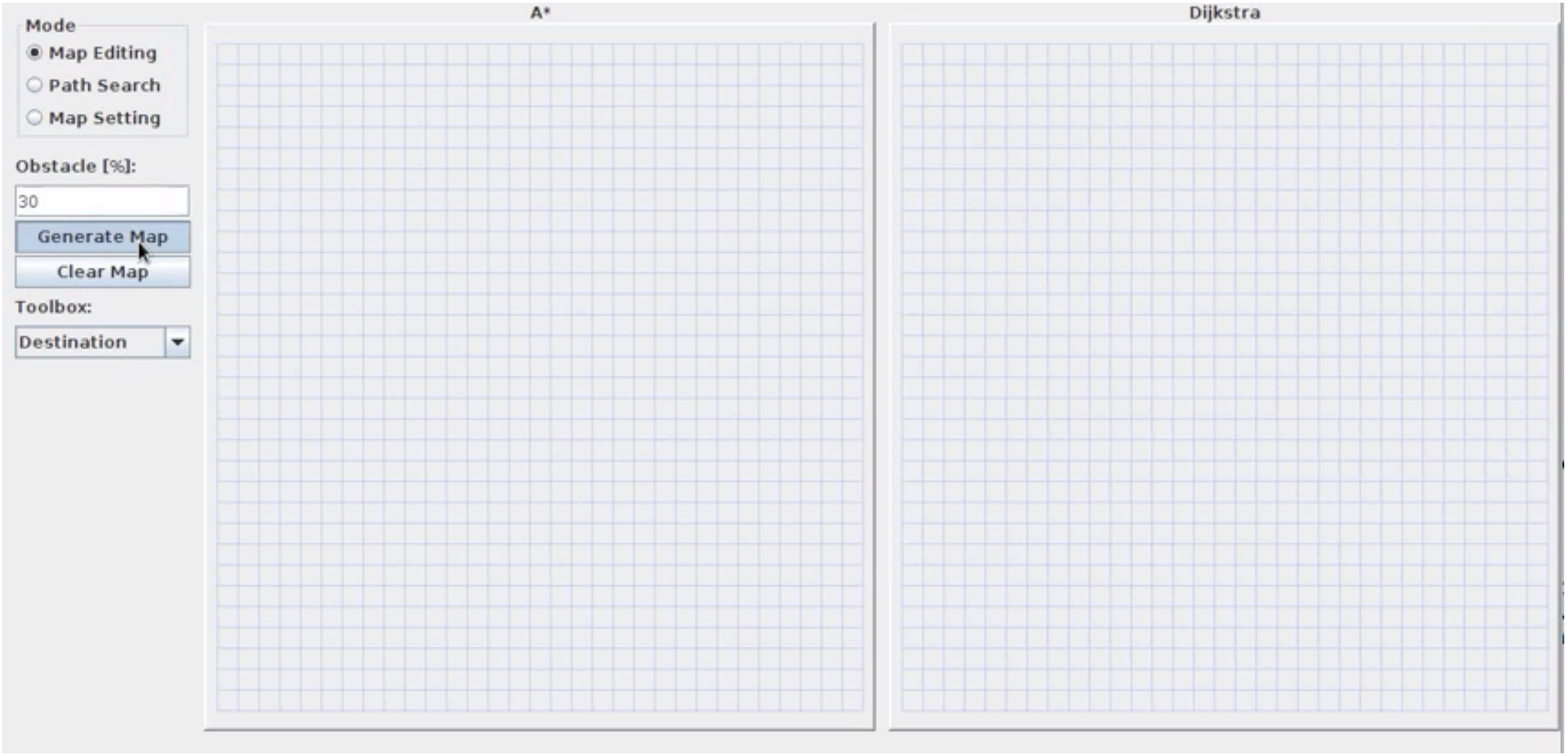
Solving a path-finding problem

- See notebook for an example of planning with the Duckietown maps:



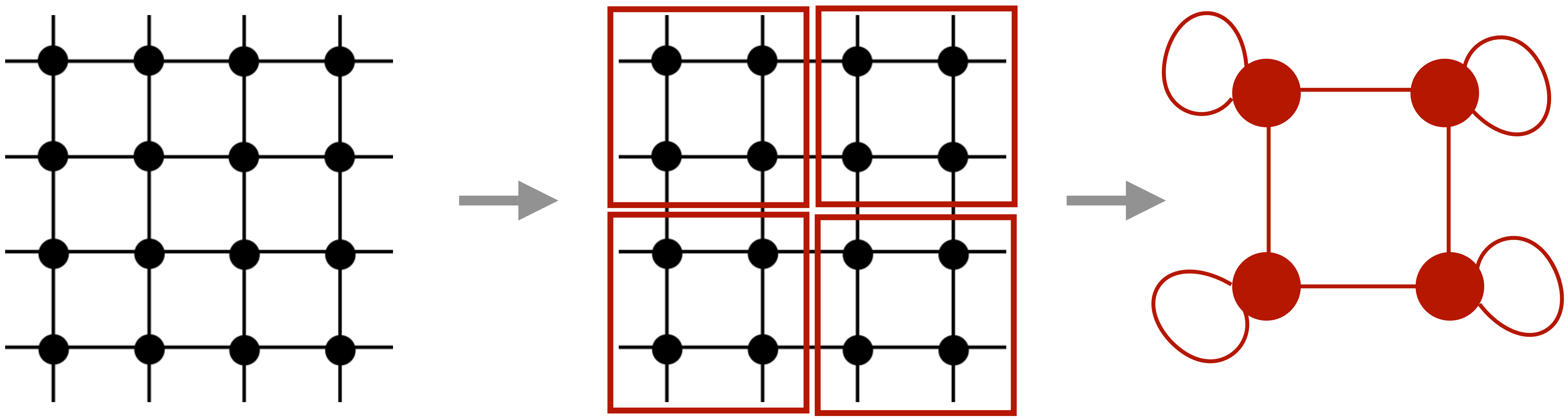
Animations of A* and Dijkstra's algorithm

- Source code / Java jar file https://github.com/kevinwang1975/PathFinder



Multi-scale planning

- Planar graphs suggest immediate approximation algorithms using a **multi-scale representations and planning.**



- It works because a planar graph is a triangulation of the plane.
  It can be extended to 3D / nD.
  - Insight: **Robot motion planning is inherently continuous** (you cannot teleport), therefore the graphs are triangulations.