Optimization and greedy approaches

Optimization problems

- An optimization problem is usually defined in general like this: given a function $c : \mathcal{X} \to \mathbb{R}$, where $\mathcal{X}$ is any set, find the value $x \in \mathcal{X}$ such that $c(x)$ is minimum.
  - You may be looking for the maximum instead, but that's just the same as finding the minimum of $-c(x)$, so let's stick with minimum from now on.
- In simpler cases, the set $\mathcal{X}$ is $\mathbb{R}^d$ for some d. Think of having a function c that represents some cost (money, time, energy consumption...) that you want to minimize, and depends a number d of parameters.
  - Already, this may be very challenging if c is not a "nice" function – we'll see this in the second half of the course.
- But usually there are additional restrictions. A common occurrence is that $\mathcal{X}$ is some subset of $\mathbb{R}^d$.
  - One or more of the parameters could have a limited range (e.g. a parameter could be "fraction of resources invested in xyz" and thus vary in $[0, 1]$)
  - Two or more parameters could have interactions (e.g. if two parameters are both "fraction of resources invested in something" then their sum cannot be larger than 1)
  - This can get arbitrarily complicated

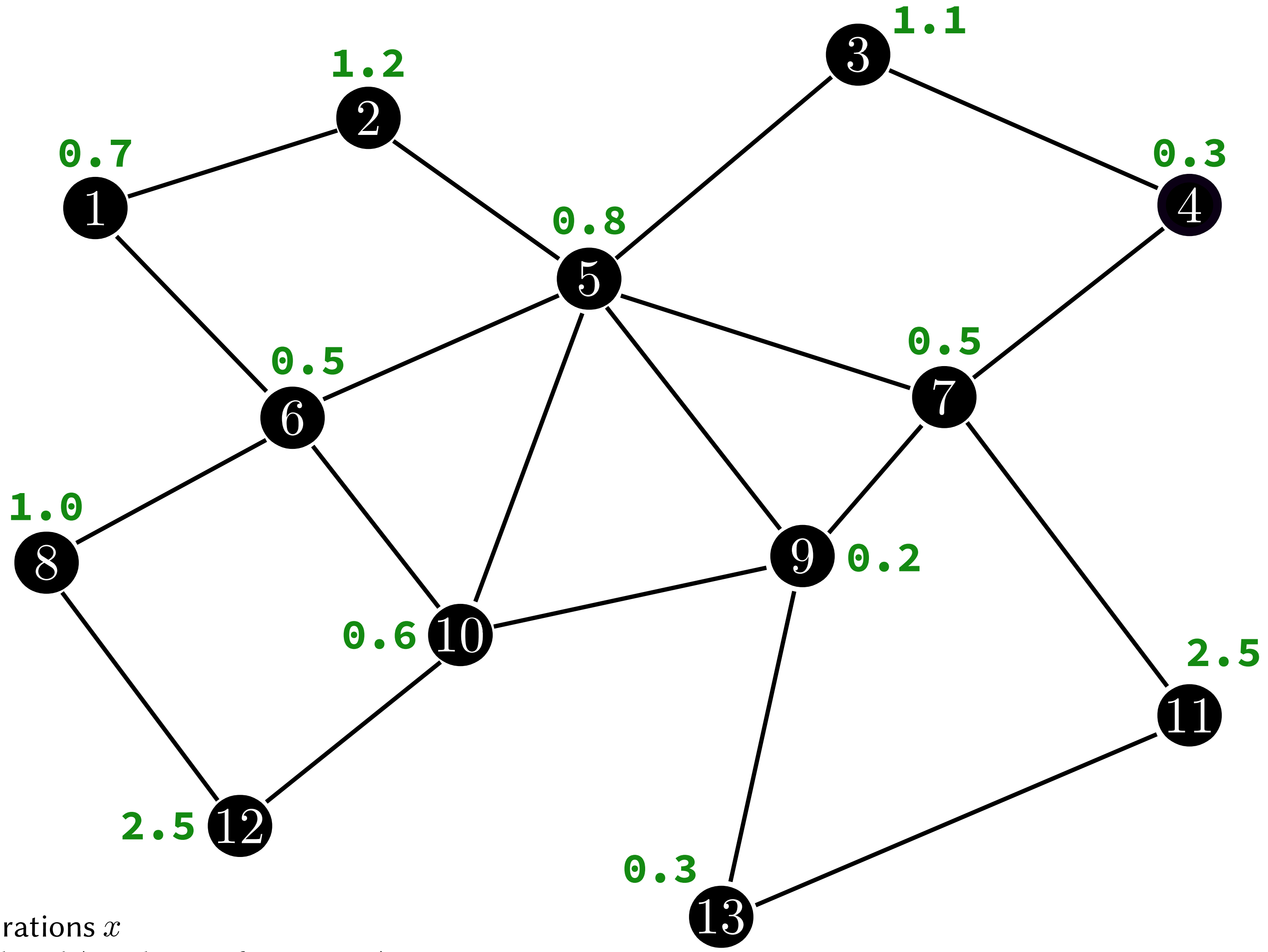
Discrete optimization problems

- A discrete optimization problem is a problem in which the space of parameters $\mathcal{X}$ is discrete
  - Example: given a directed weighted graph (e.g. a map of cities and roads), find the shortest path that goes from node A to node B (e.g. the route of minimum distance between two cities)
- We will focus on the case when $\mathcal{X}$ is discrete and finite. Therefore $c\left(x\right)$ can only take a finite number of values
- Things get interesting when $\mathcal{X}$ is very large. Usually, the elements of $\mathcal{X}$ are made up of various "pieces"
  - In the graph example, the elements of $\mathcal{X}$ are paths (routes), so the pieces are the individual edges. The number of all possible paths grows exponentially with the size of the graph.
  - Another simple-to-describe case is when $\mathcal{X}$ is a collection of binary strings of length n, which may represent for example a sequence of true/false decisions about something. The size is 2^n.
- You see from the examples that it's impractical to just check all possible values of $\mathcal{X}$ and see which is the best - this approach often scales like $O\left(2^n\right)$

Greedy approaches

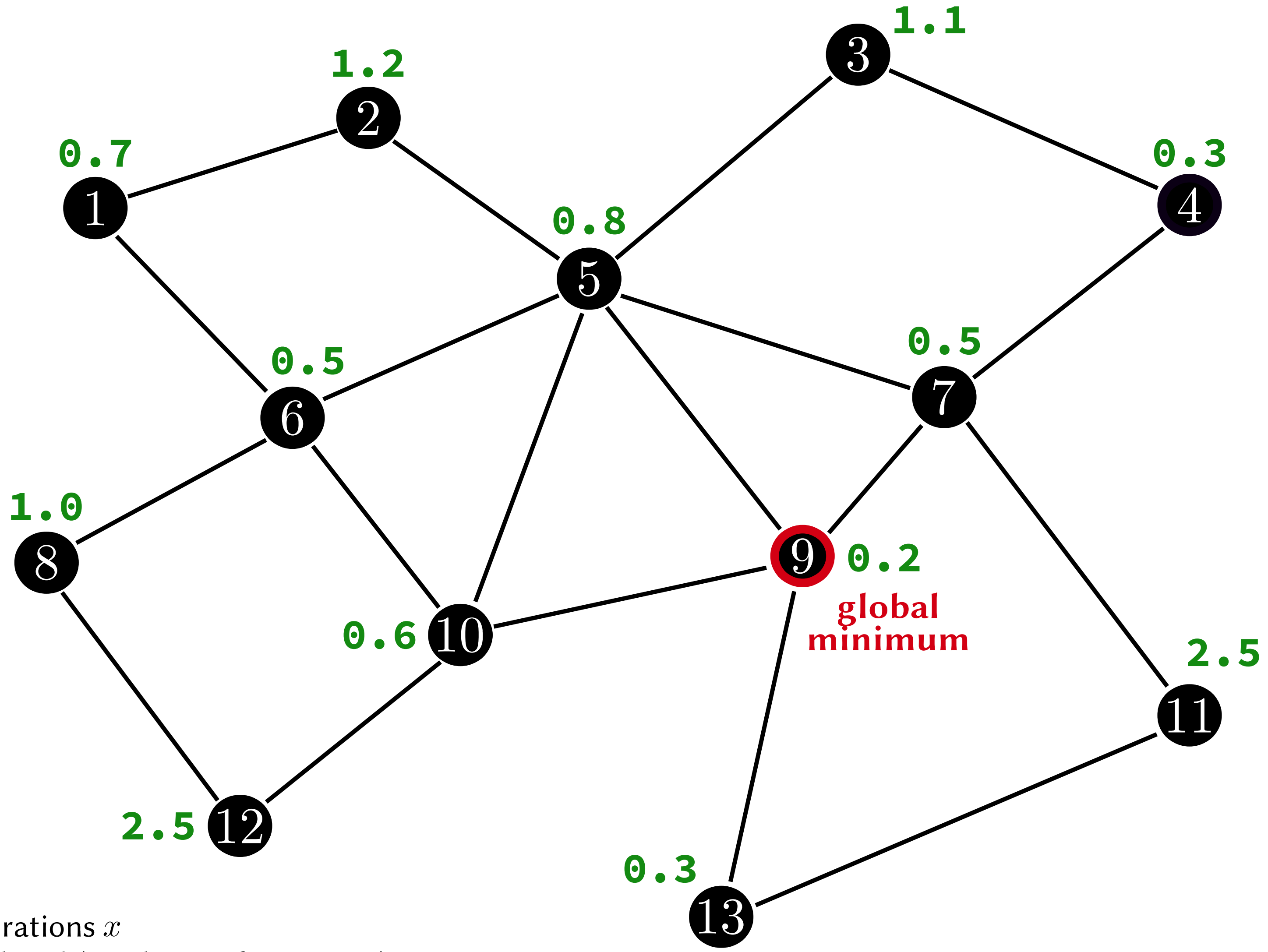
- Let's say that we have at least some notion of "neighborhood" within our $\mathcal{X}$, so that given two elements x and y we can tell whether they are "close" or not
  - Graph example: $\mathcal{X}$ is the set of paths from A to B, two paths are close if they differ only by one intermediate node
  - Binary string example: $\mathcal{X}$ is the set of sequences of the form `01100...110`, two sequences are close if they only differ in one bit.
- Then a type of greedy approach would go like this:
  - Pick an initial guess $x^{(0)}$ in $\mathcal{X}$ (somehow, maybe at random)
  - Repeat for all time steps t:
    - Pick a neighbor y of $x^{(t)}$ (somehow, maybe at random)
    - If $c\left(y\right) \leq c\left(x^{(t)}\right)$, then $x^{(t+1)} = y$, otherwise $x^{(t+1)} = x^{(t)}$
  - Keep going until you found an x such that all its neighbors are worse.
    This is a **"local minimum"**
- A greedy approach is usually quick, relatively easy to implement. It also often gives very poor results.

Greedy approach example



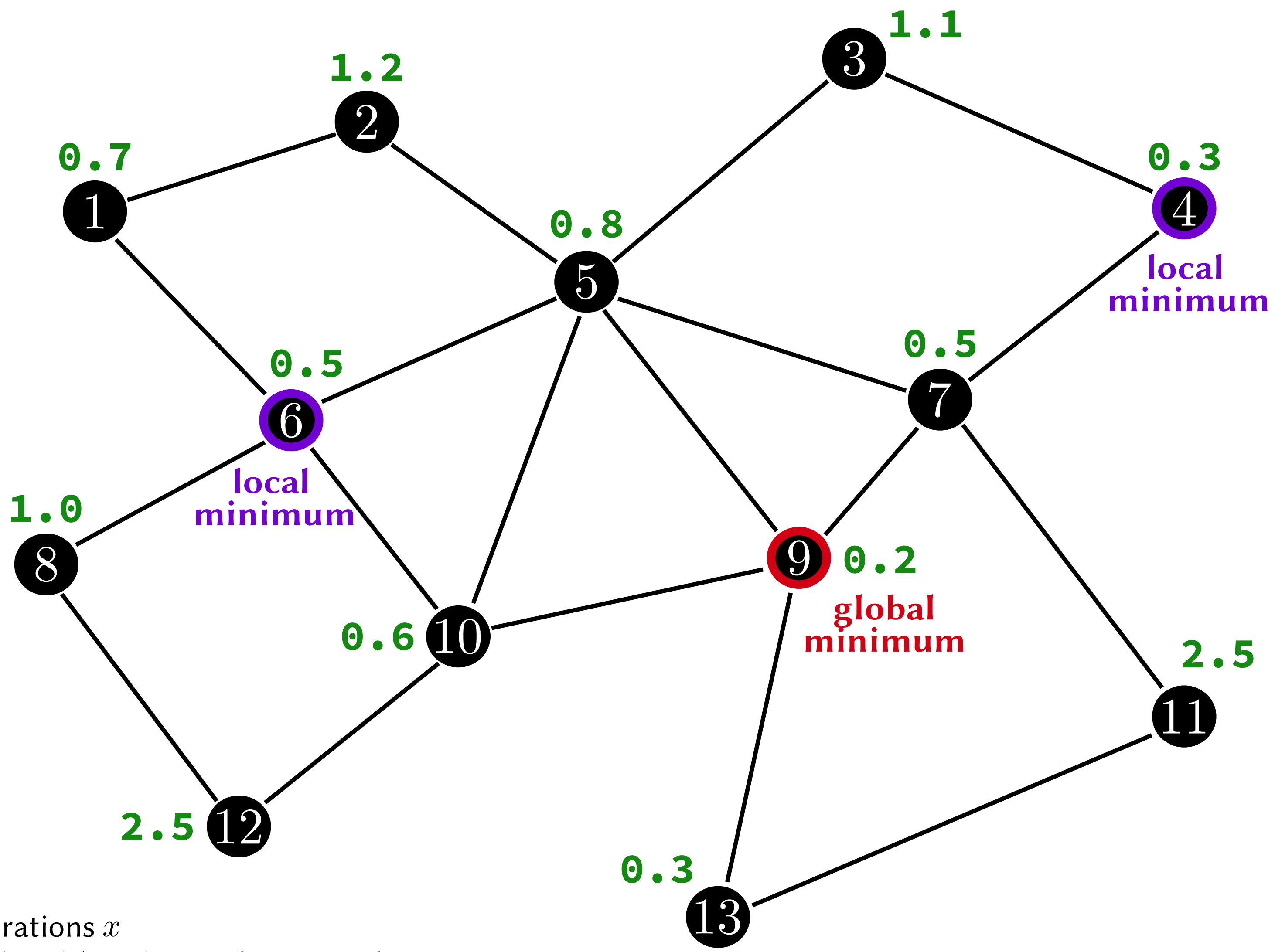
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



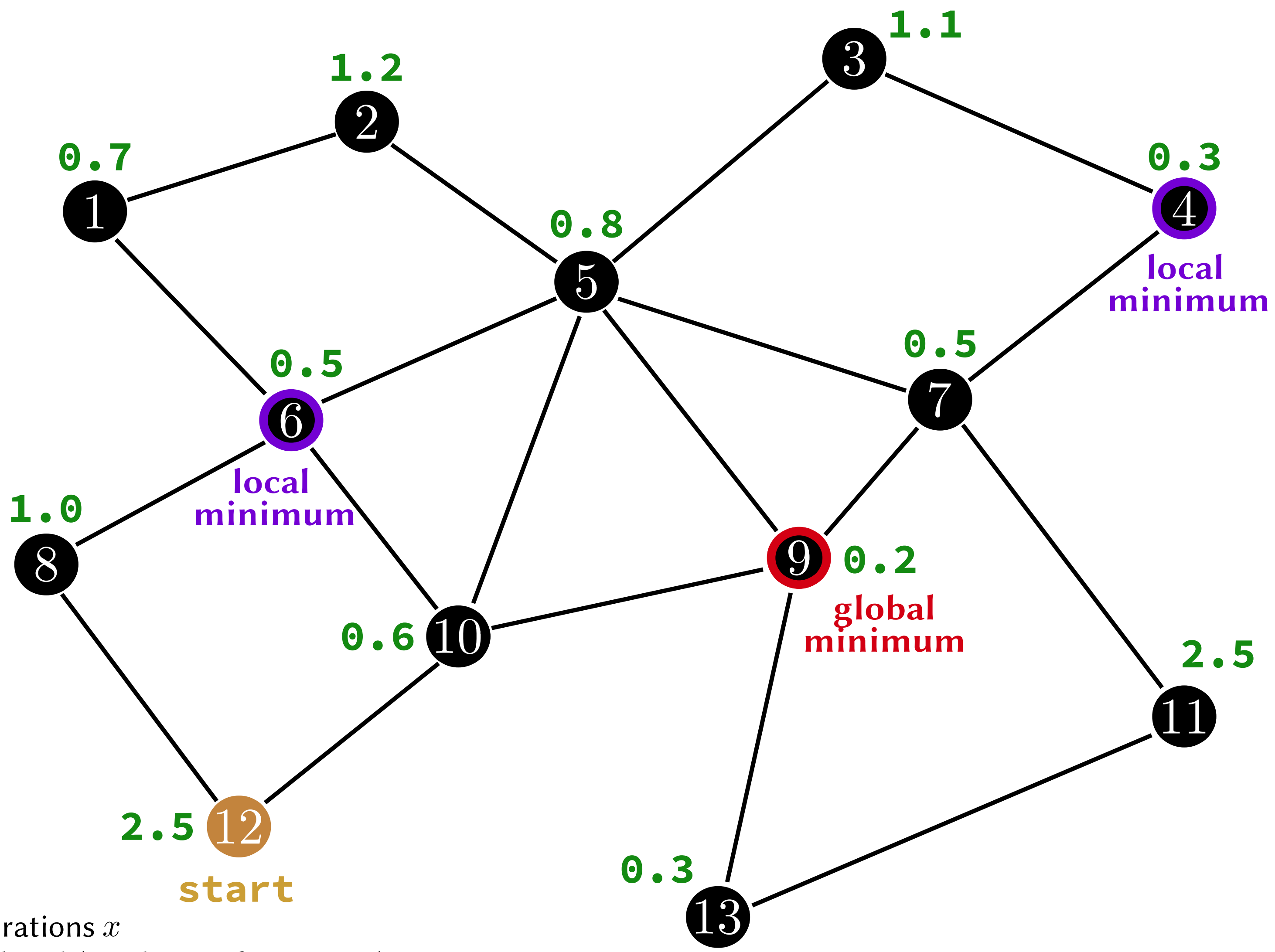
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



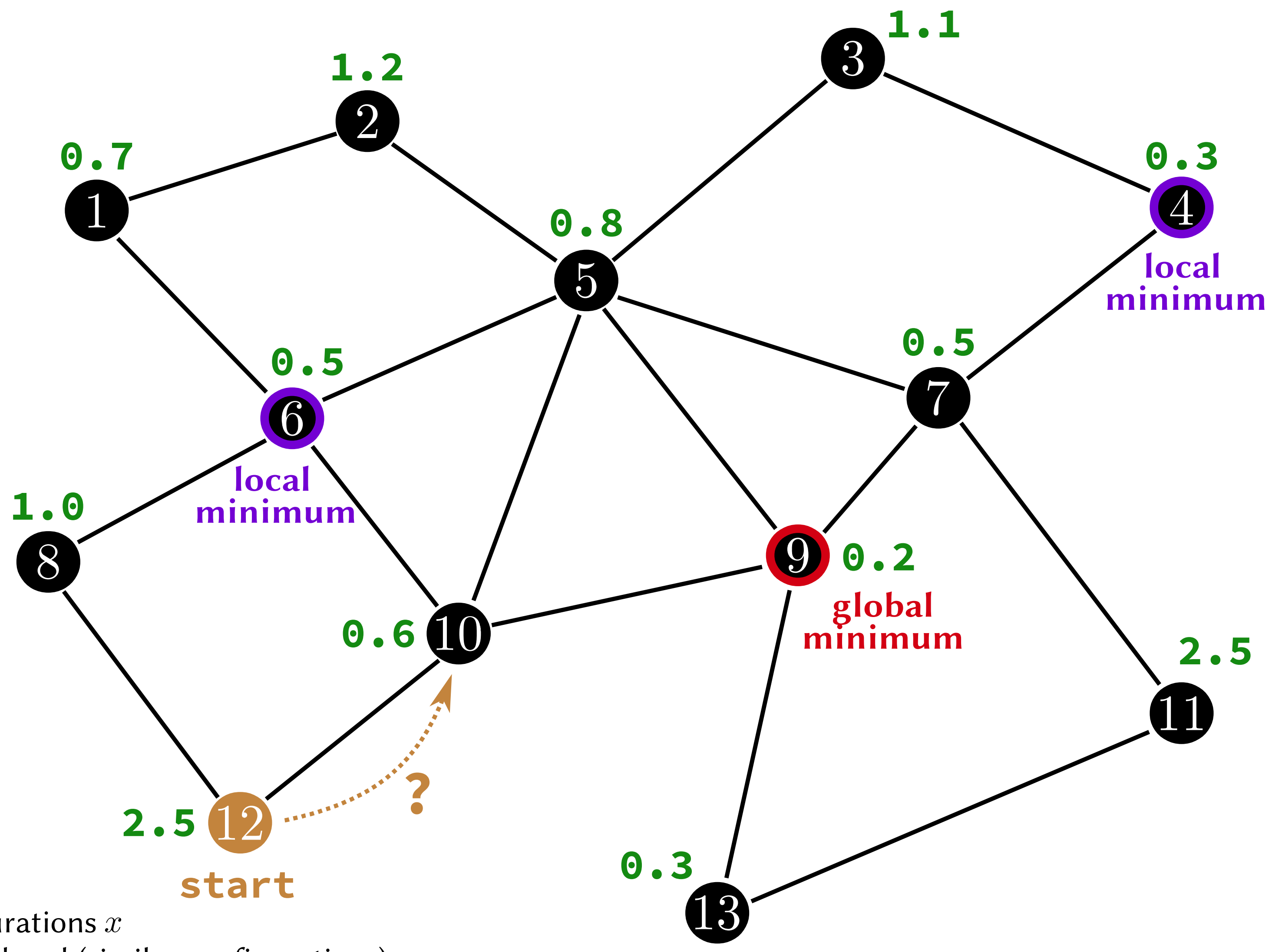
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example

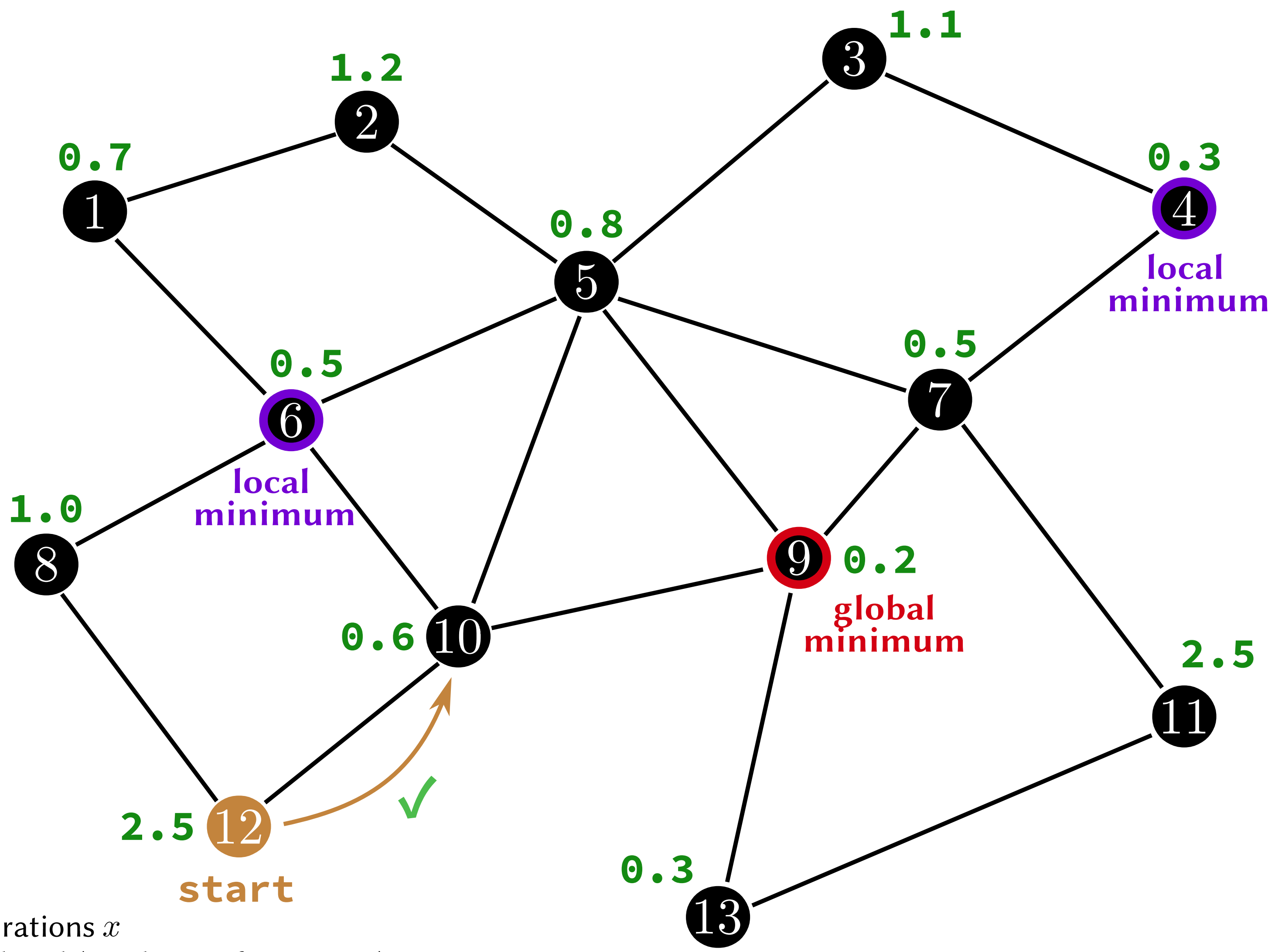


circles = configurations x

lines = neighborhood (similar configurations)

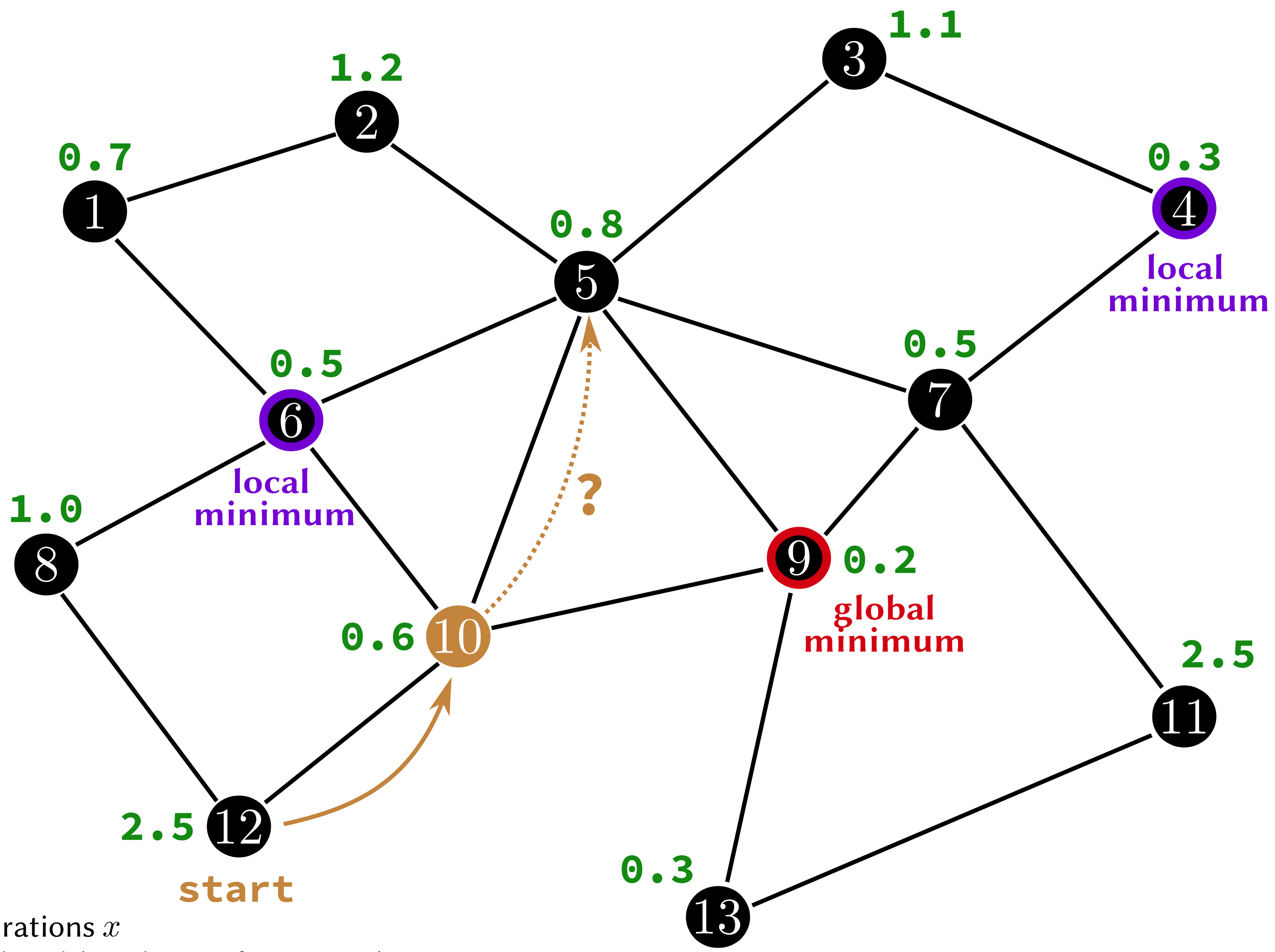
green numbers = costs

Greedy approach example



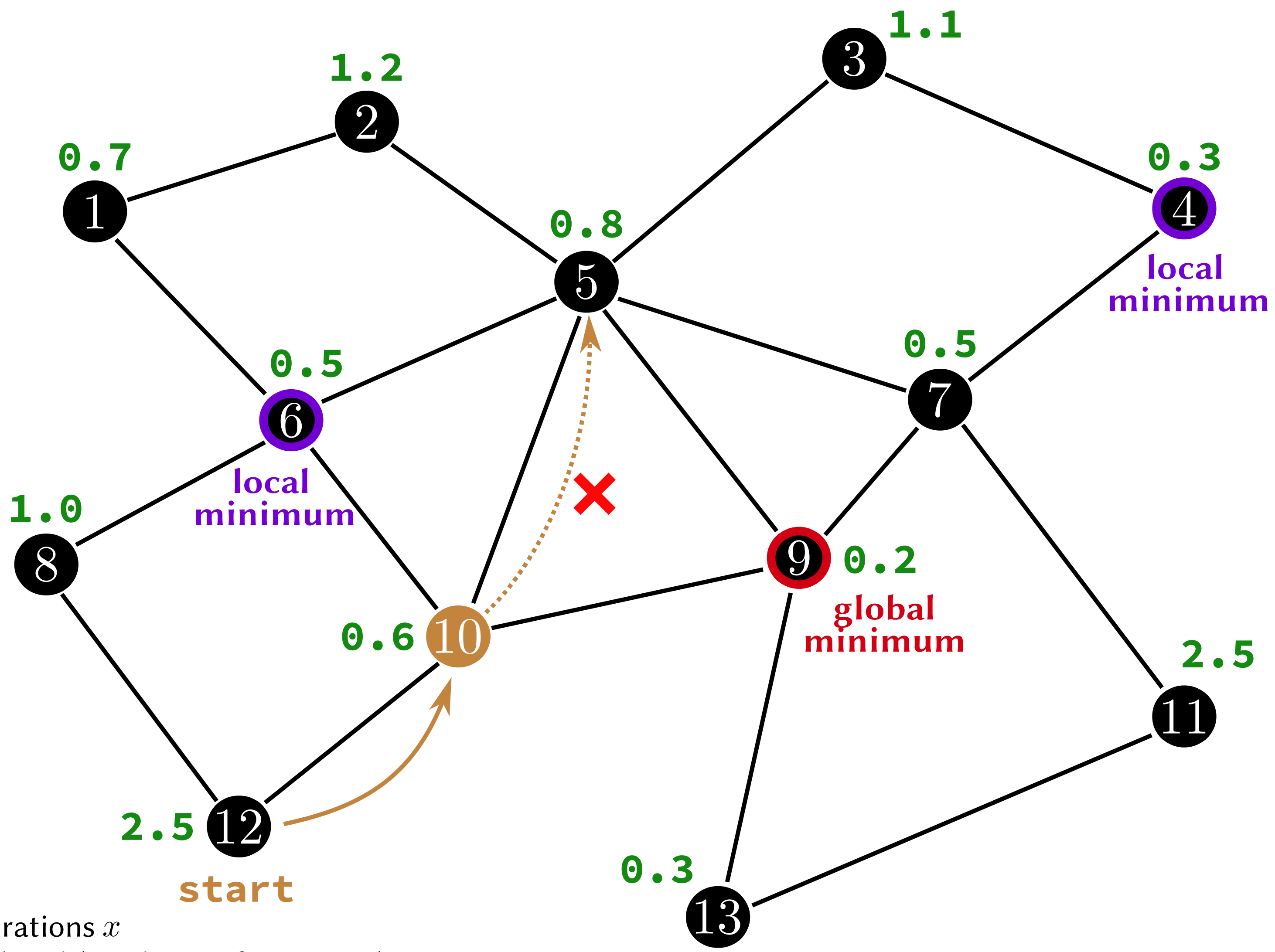
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



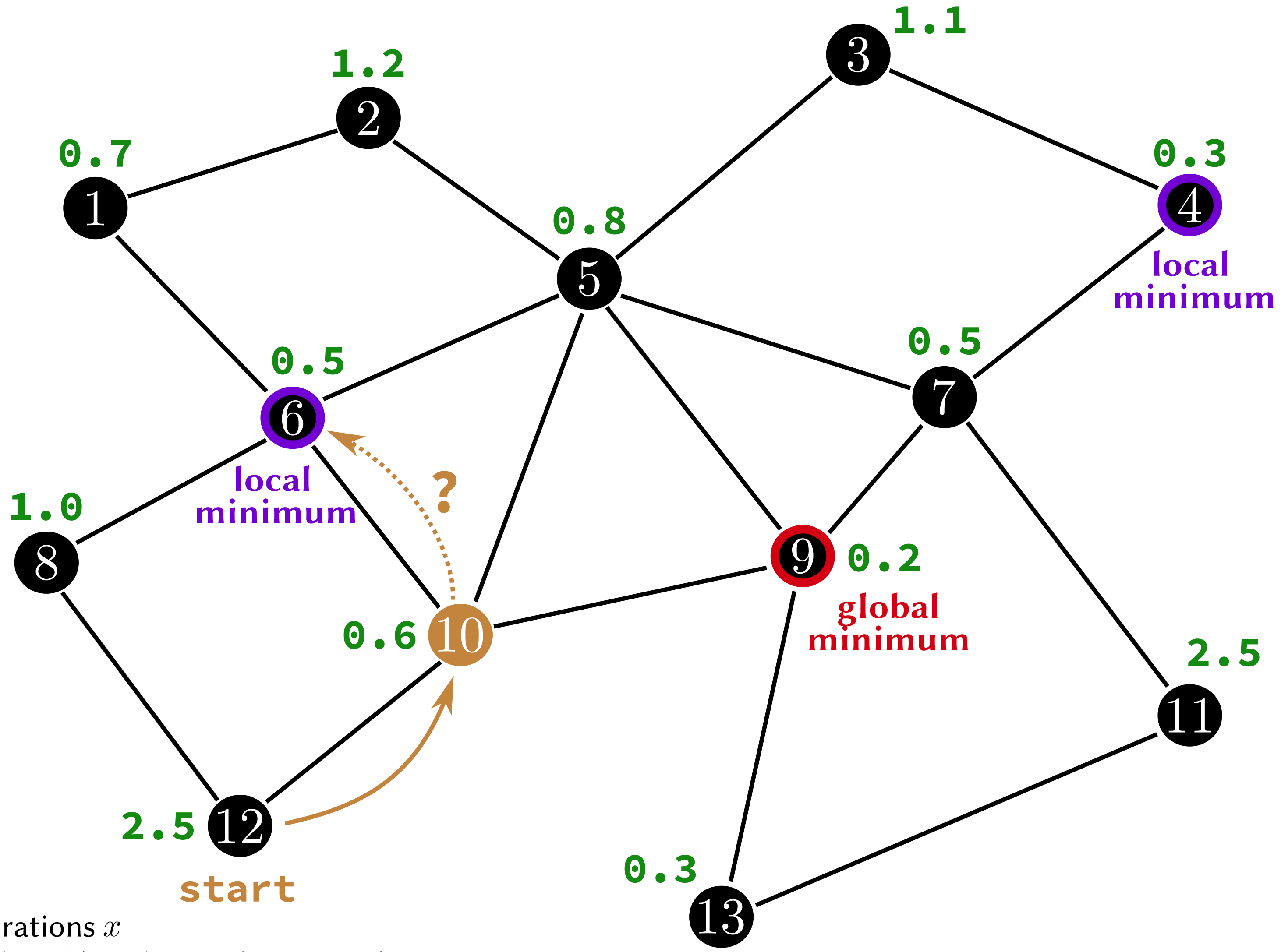
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



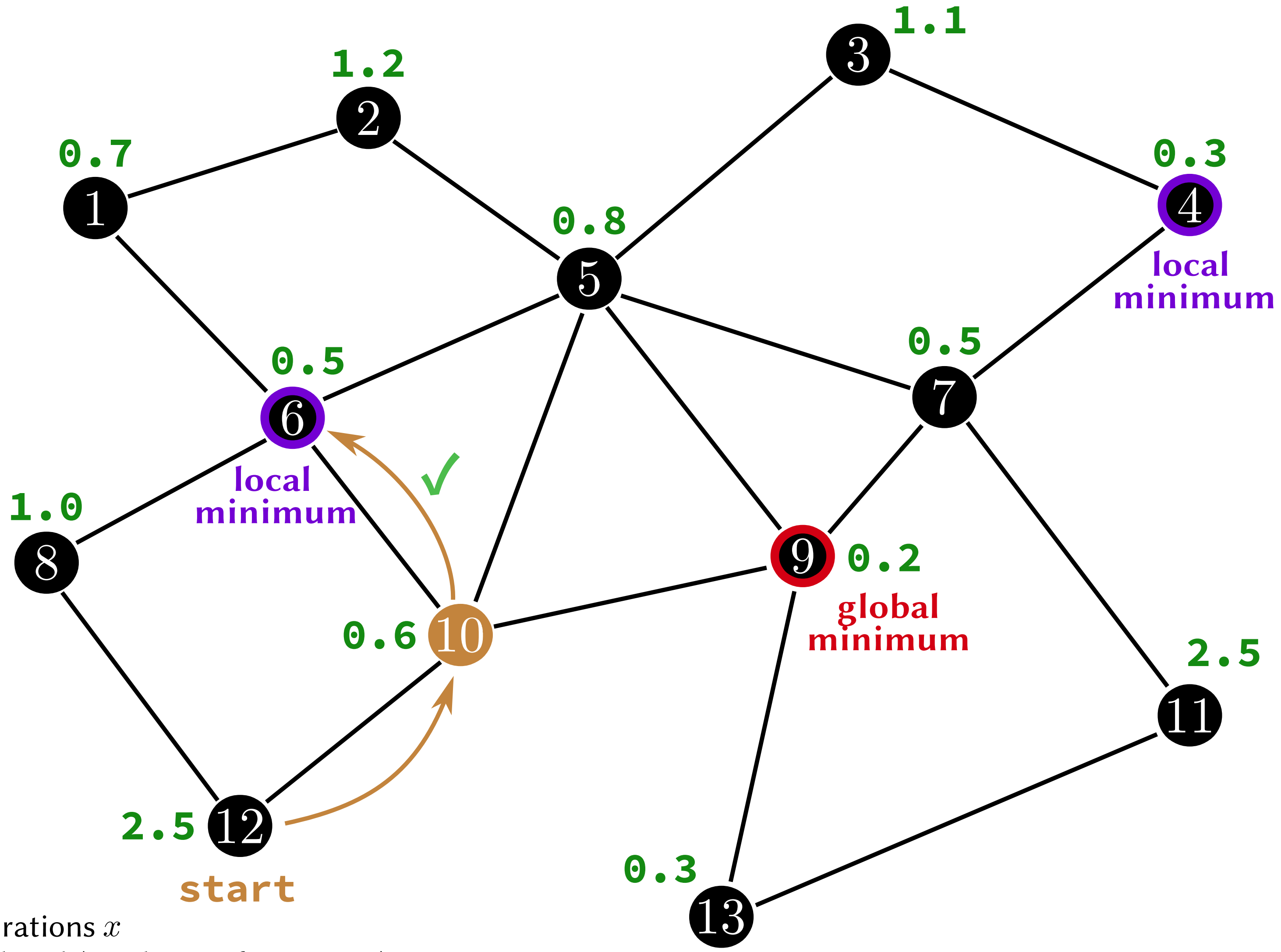
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



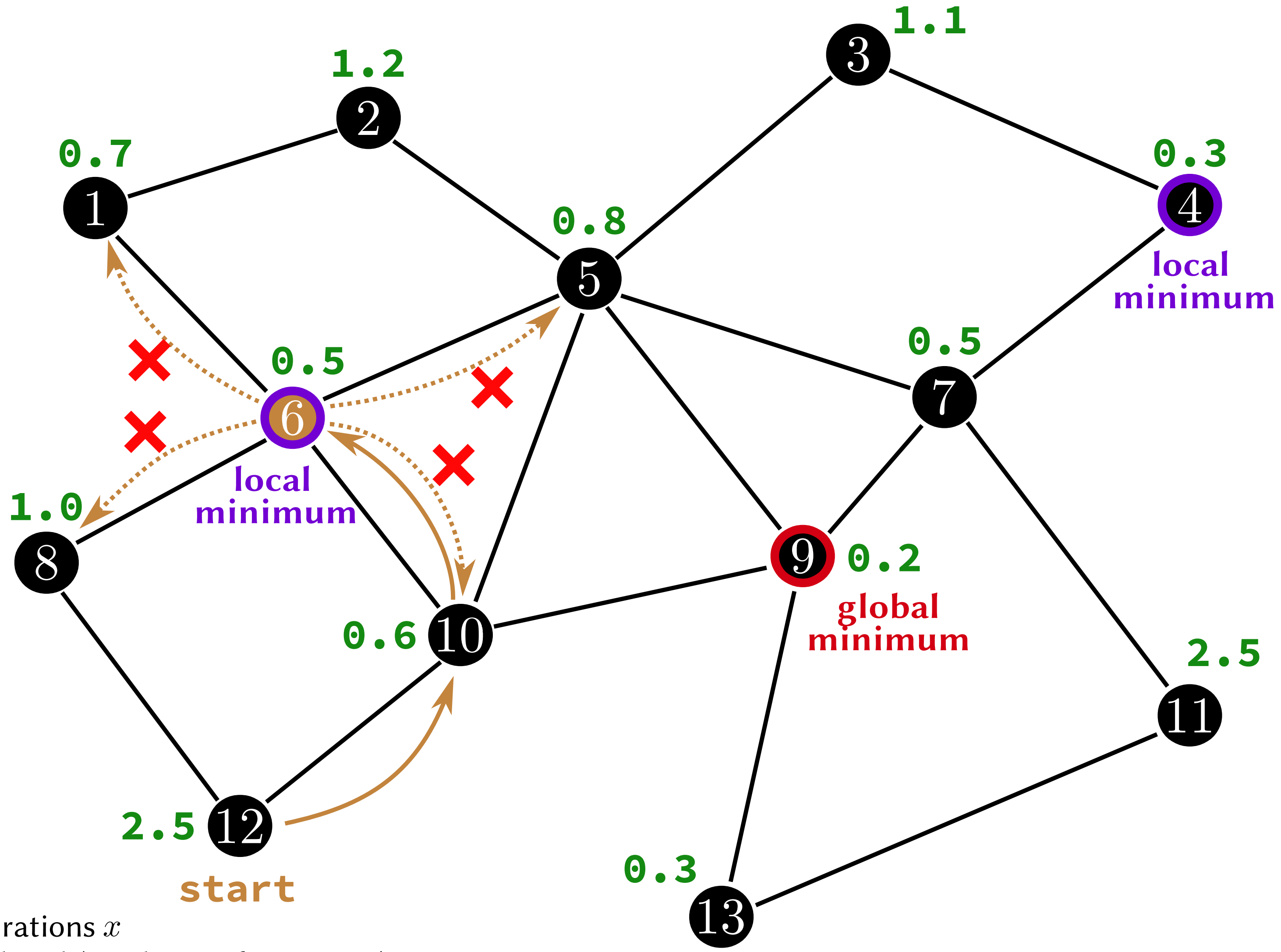
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



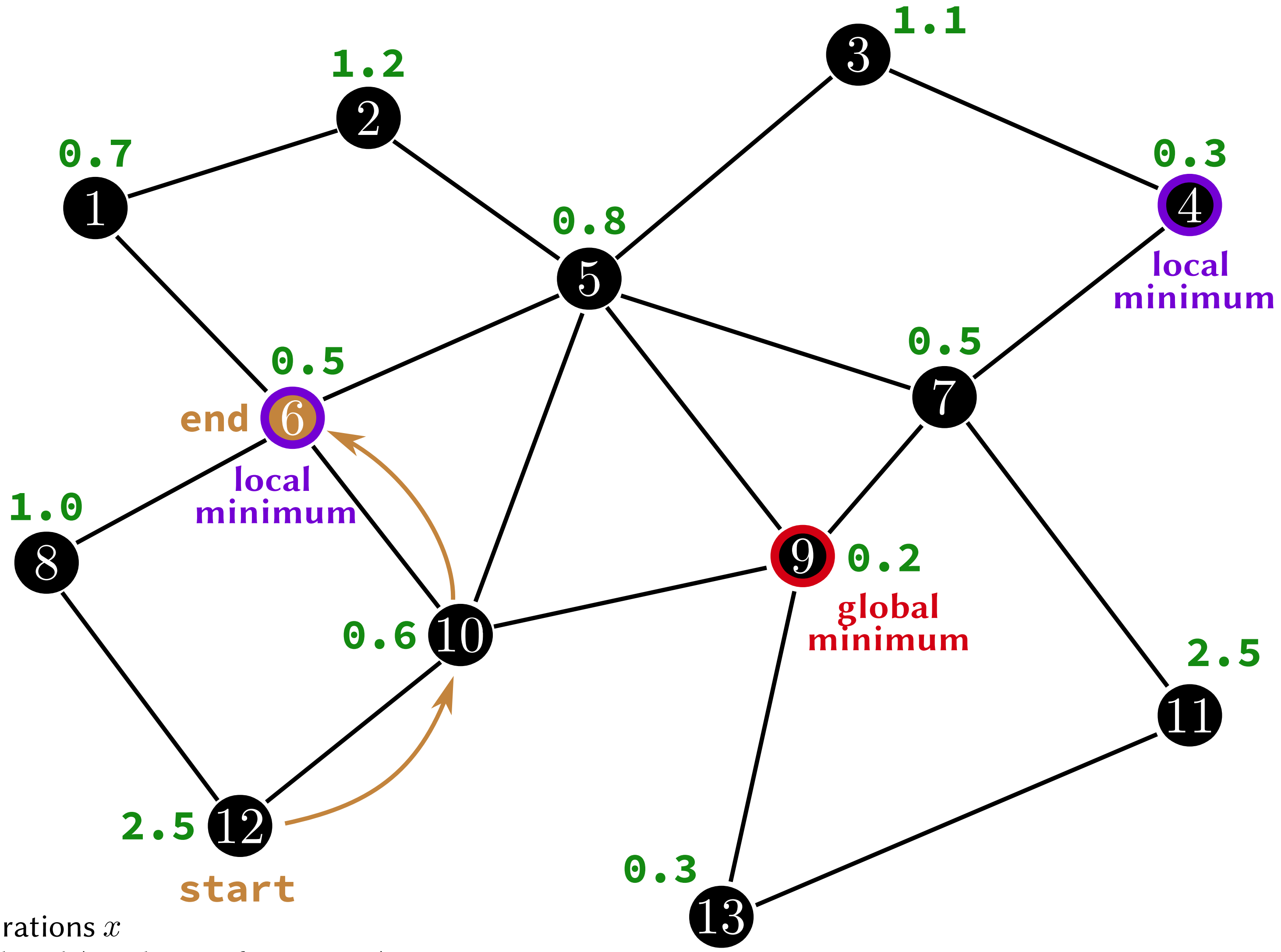
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



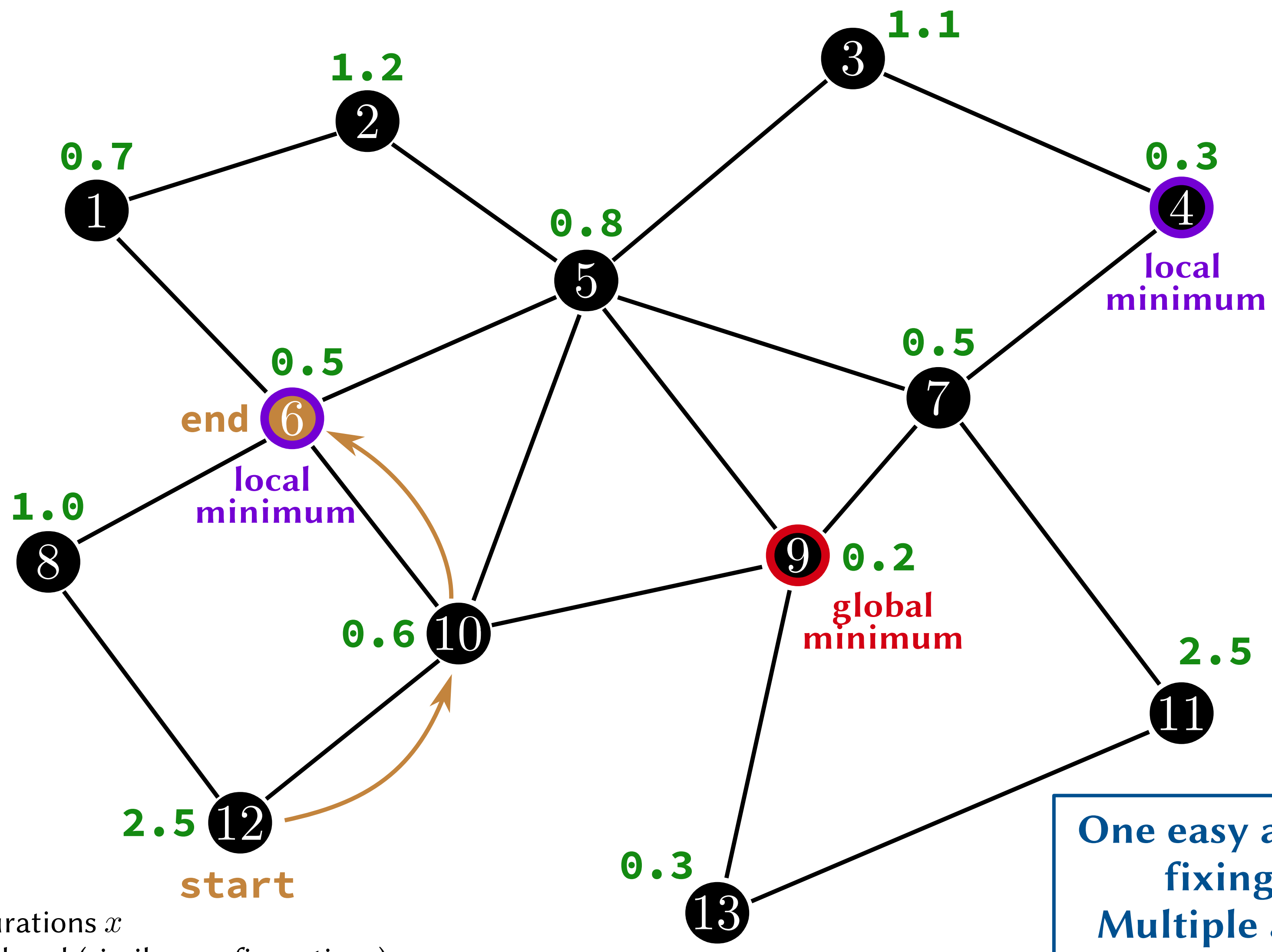
circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

Greedy approach example



circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

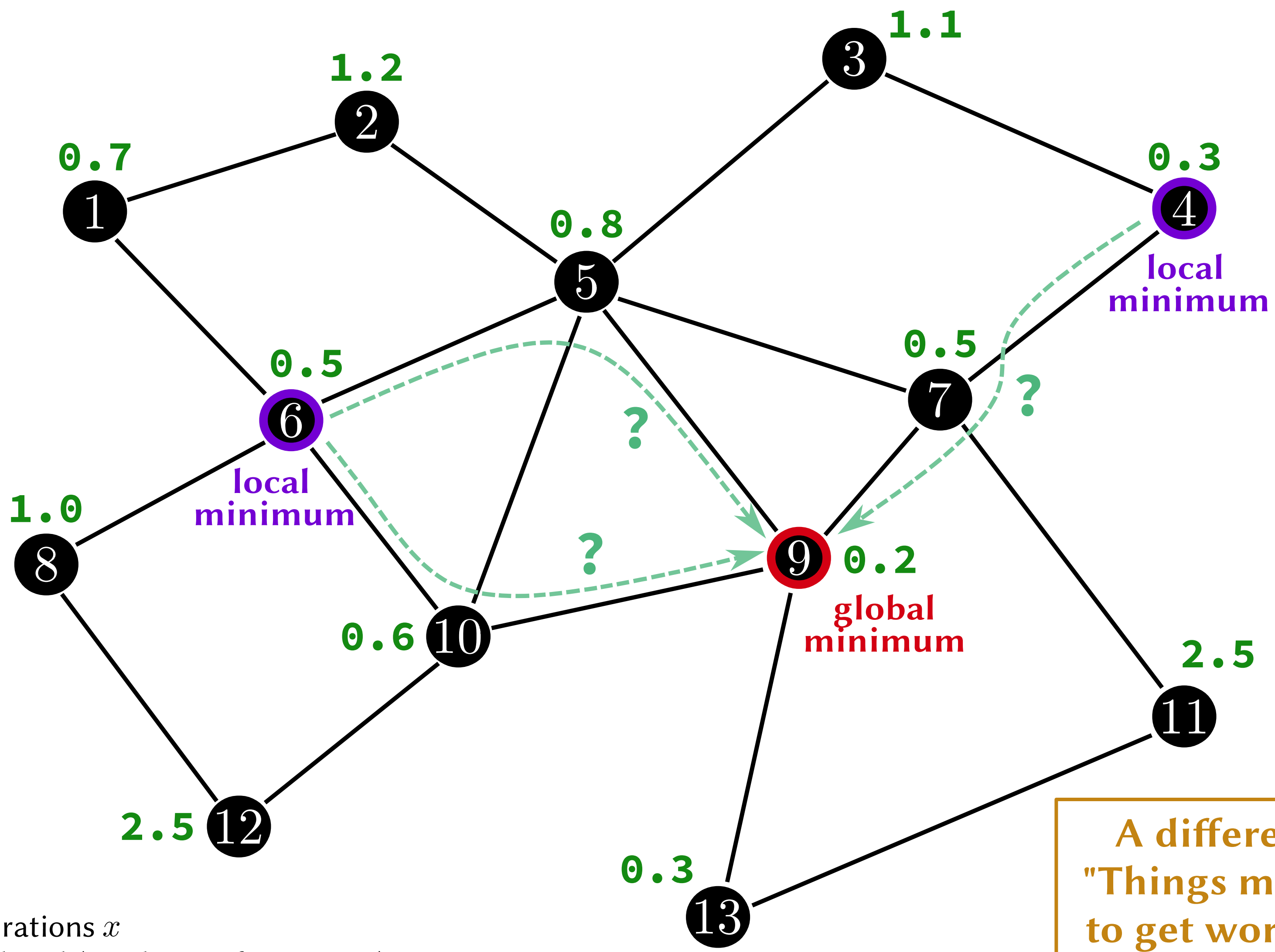
Greedy approach example



circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

One easy attempt at fixing this: Multiple attempts (restarts)

Greedy approach example



circles = configurations x
lines = neighborhood (similar configurations)
green numbers = costs

A different idea: "Things might need to get worse before they get better"

What's needed for this greedy approach?

- To code something like this, we must be able to:
- Represent the configuration x internally in the memory (choose a representation)
- Compute the cost (could be expensive to do, not a big problem itself)
- Pick an initial configuration at random
- Pick a random neighbor for any given configuration (propose a move)
- Decide whether the new configuration is better ("delta-cost")
  - Can't we just compute the new cost and see if it's lower than the old?
  - Yes, but it is often the case that it's way more efficient to compute the *cost difference* instead. Example: given a route from A to B, change only one intermediate node:

    A⟶...⟶X⟶Y⟶Z⟶...⟶B

    A⟶...⟶X⟶W⟶Z⟶...⟶B

    Only 4 pieces of route are involved. One might decide to remove an intermediate node, or add one: similar arguments apply. In general we'd like that "local" moves imply $O\left(1\right)$ computational effort in updating the cost.

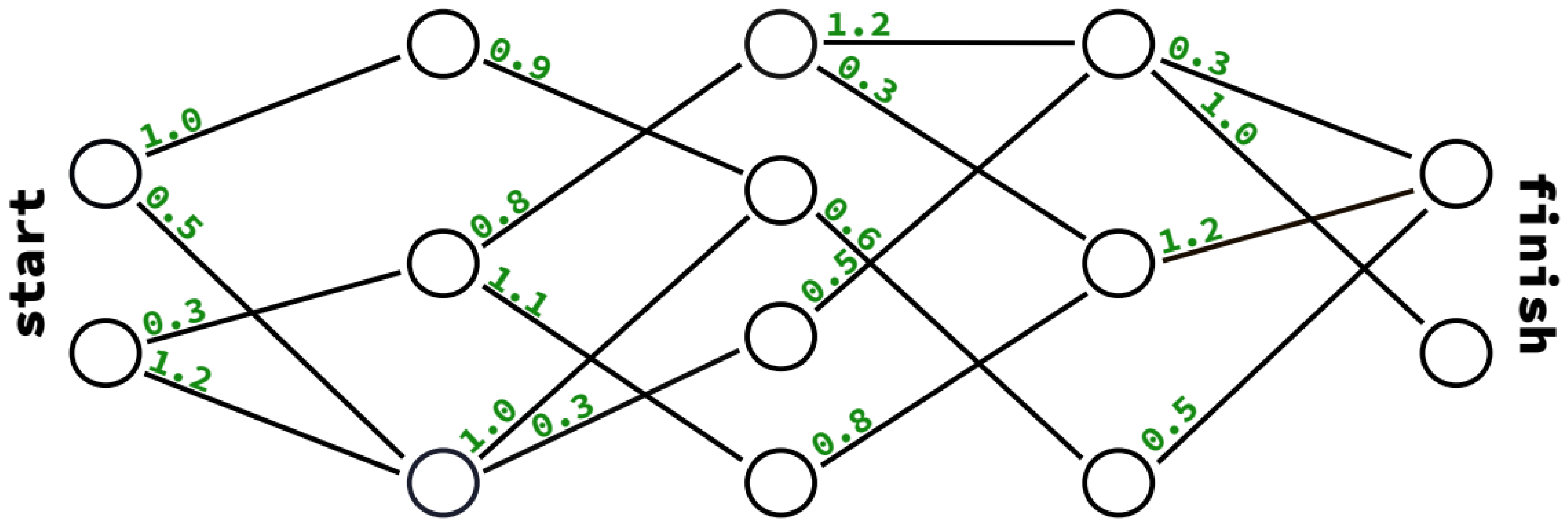
We're going to implement this!

More greedyness

("greedy" is a loose term, not a well-defined class)

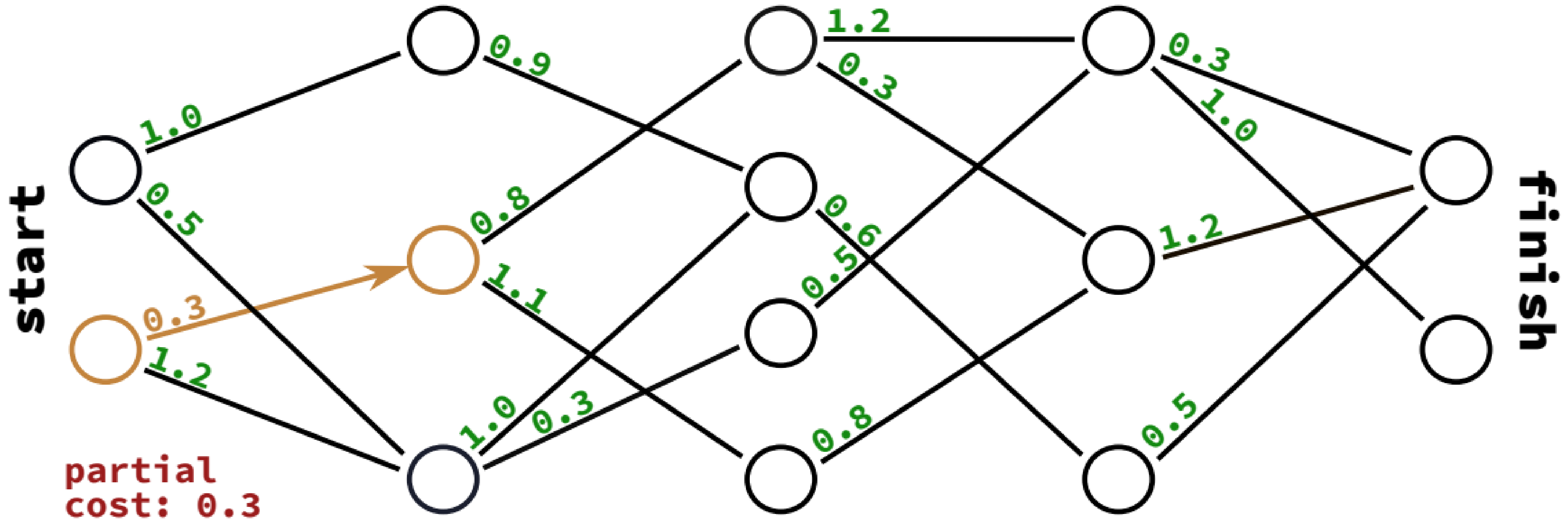
- Another possible kind of greedy approach is applicable when the elements of $\mathcal{X}$ are made of "pieces" and you can define your cost function c on just a part of x:
  - In the graph example, you can have a path that starts from A and has not reached B. You can still define its cost (the distance)
  - In the binary string example, maybe your function can be computed even if you only have the beginning of the string
- Then you can build your proposed solution one piece at a time; every time you choose the option that gives you the minimum cost so far:
  - Graph example: pick the shortest outgoing edge from A, then the shortest from there (as long as it doesn't go back to a previous node), etc. Maybe you'll get to B?
  - Binary string example: Choose the first bit between `0` and `1`. Say that `0` is better. Then choose between `00` and `01`. Say that `01` is better. Then choose between `010` and `011`. Say that `011` is better. And so on...
- Same deal as before: very quick, also usually easy to implement; often very poor results.

Partial buildup greedy approach example



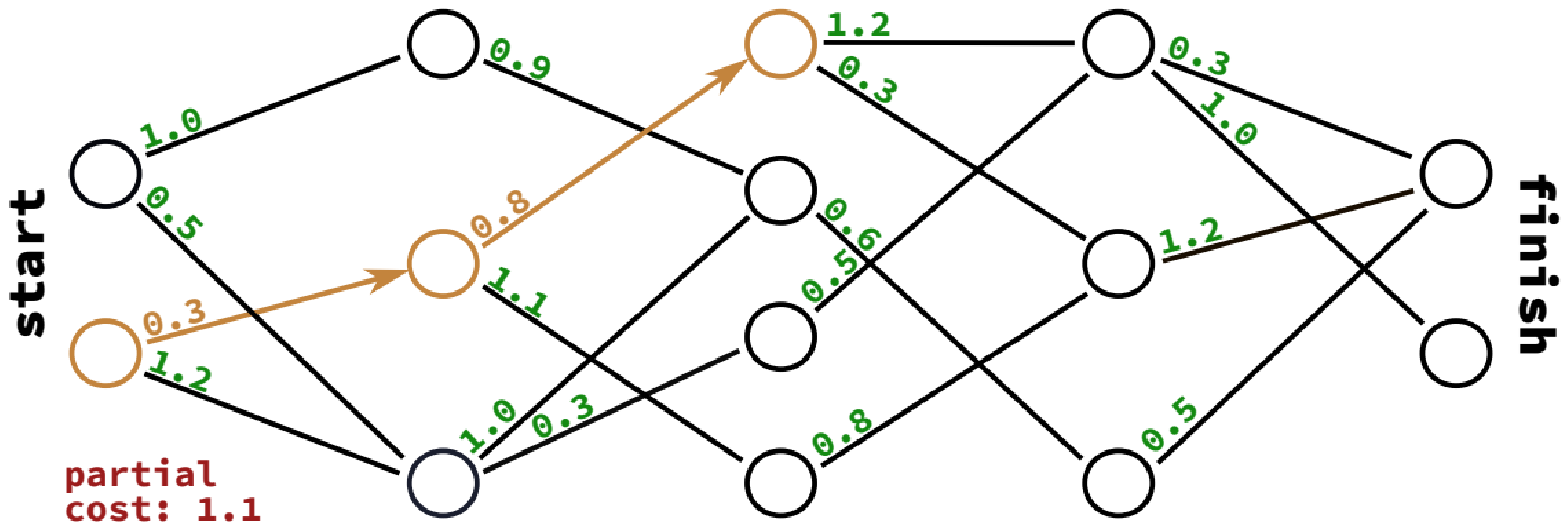
goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Partial buildup greedy approach example



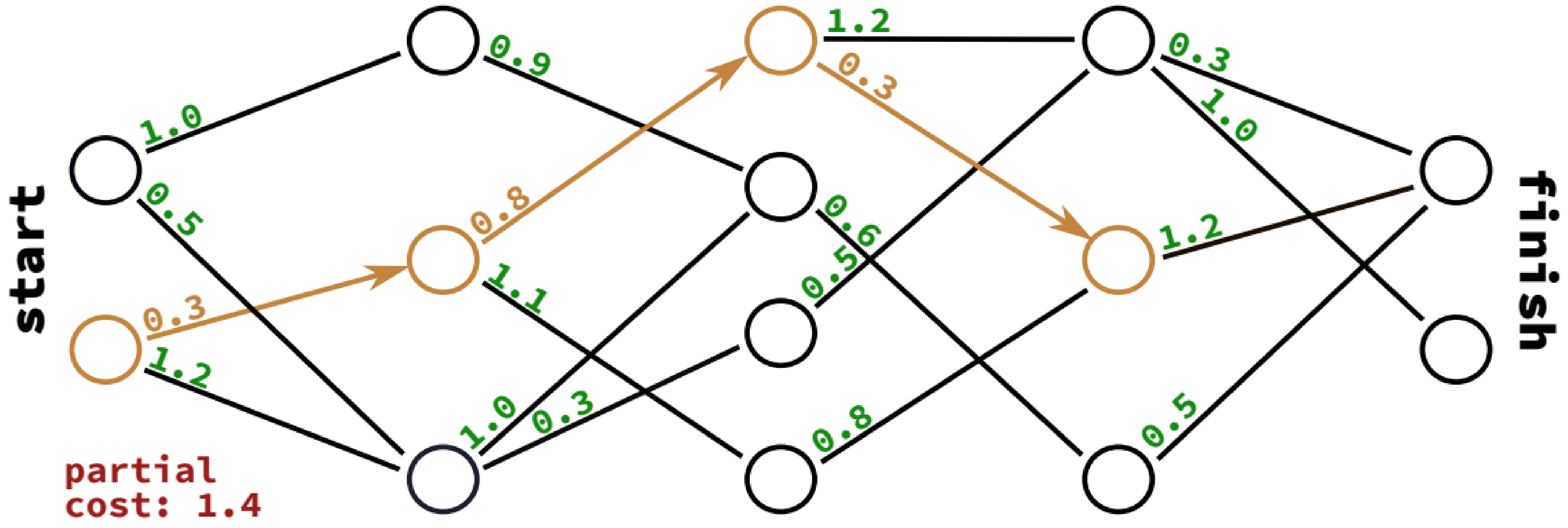
goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Partial buildup greedy approach example



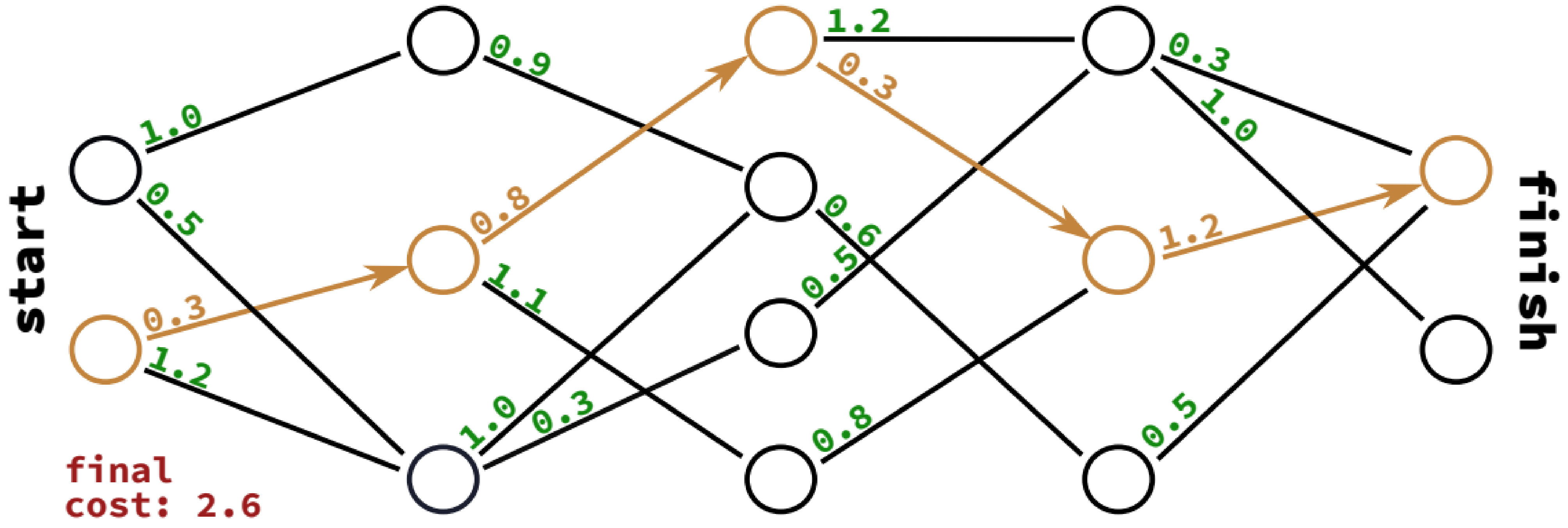
goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Partial buildup greedy approach example



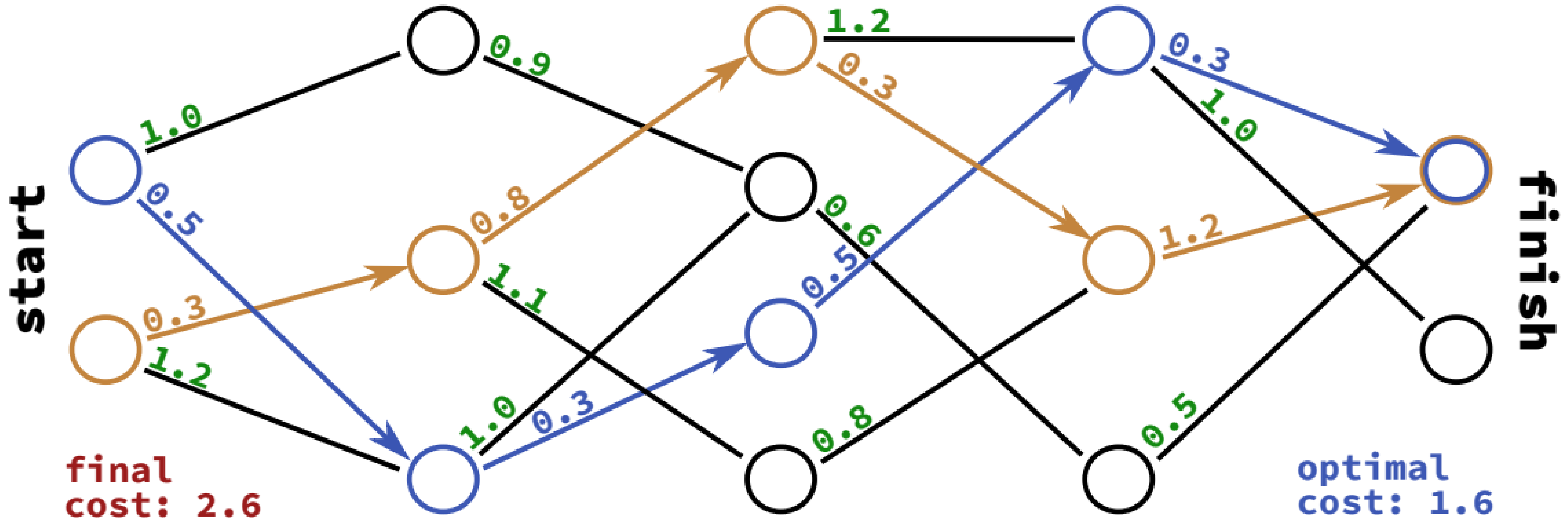
goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Partial buildup greedy approach example



goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Partial buildup greedy approach example

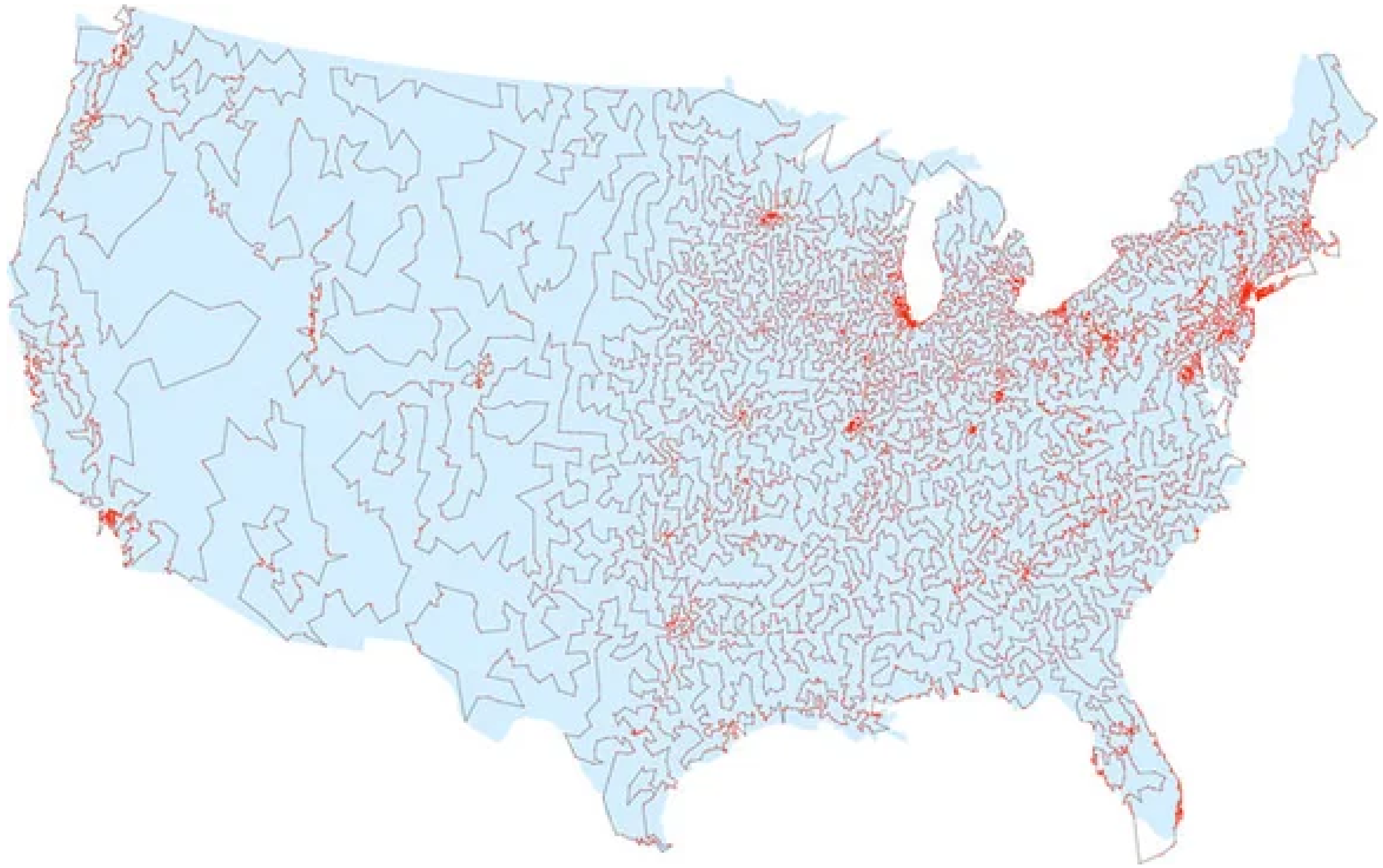


goal = find the best path that goes from the left to the right
circles = path nodes
lines = available connections
green numbers = costs (partial)
one whole path = configuration x

Don't be greedy

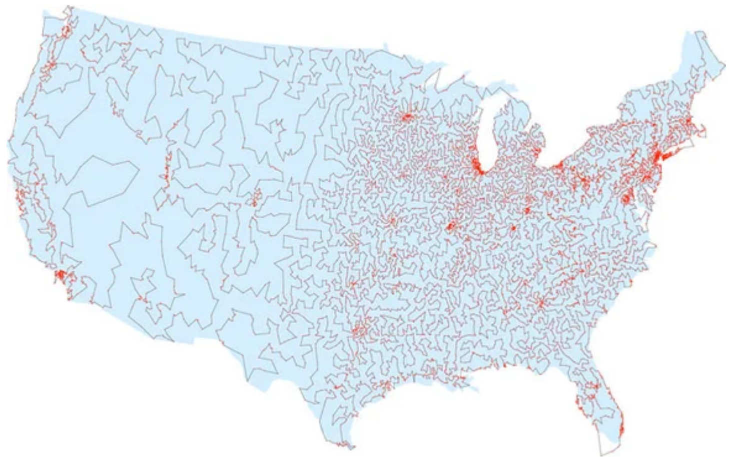
- For some selected problems, there are sophisticated non-greedy (or not-so-greedy) approaches
  - In the "best path from A to B" graph example, Dijkstra's algorithm
  - We'll see the dynamic programming approach (and solve the latter "optimal left-to-right route" example...)
- For some problems, no fast algorithm that guarantees to find the optimum exists, and likely never will (NP-C and harder problems)
- We are going to study an approximate but general approach, a metaheuristic called **Simulated Annealing**

The Travelling Salesman Problem (TSP)



The Travelling Salesman Problem (TSP)

- Celebrated NP-C problem (no efficient way to reach the optimum in the worst-case)
- Given: n cities (nodes), and the distance d_{ij} between any two cities i and j (symmetric)
- Find the shortest route that goes through every city exactly once and goes back to the starting point ("shortest Hamiltonian path on a graph")
- $\mathcal{X} =$ space of all possible Hamiltonian paths
- $c(x) =$ total distance along the path
- $|\mathcal{X}| = n!/(2n) =$ num. of ways to permute the cities ($n!$), ignoring differences in the starting point (divide by n) and the direction (divide by 2)



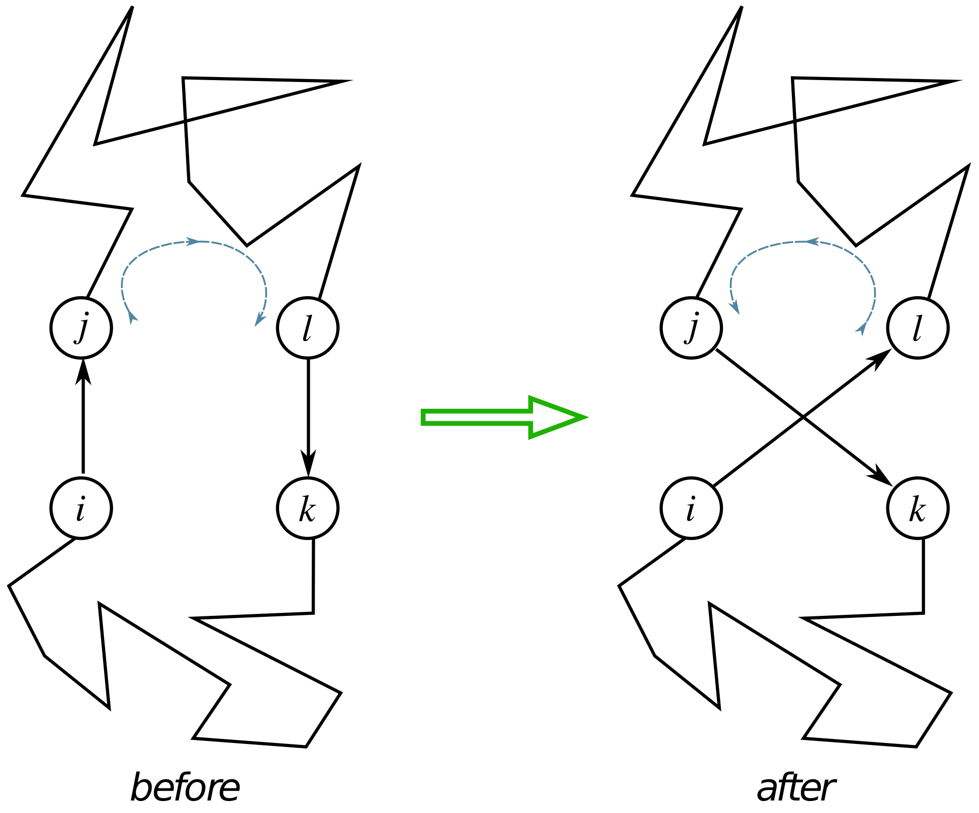
Greedy approaches to TSP

- $\mathcal{X} =$ space of all possible Hamiltonian paths
- $c\left(x\right) =$ total distance along the path
- Partial-buildup greedy approach: start from a city at random, choose each new city as the closest to the last chosen one, excluding the cities already visited. Proceed until all cities are chosen, at that point go back to the first.
  - This works awfully
- Random-search greedy:
  - How do we represent the route?
    - A permutation of the cities' indices will work fine.
  - How do we define when two routes are "close", or "neighbors"?
    - Several possible ways. We'll use the "swap-paths" rule. From any given route, pick two edges `i→j` and `l→k` and swap them to `i→l` and `j→k`.

TSP, swap paths move

- From any given route, pick two edges `i→j` and `l→k` and swap them to `i→l` and `j→k`.

  (note that the `j→...→l` piece of the route gets reversed)



only 4 edges involved in the delta-cost computation