



COMP 476

Advanced Game Development

Session 3

Pathfinding AI, World Representations

Pathfinding AI (Reading: AI for G, Millington § 4.1-4.4)

Lecture Overview

- ❑ **Pathfinding**
- ❑ **Dijkstra's Algorithm**
- ❑ **A* Algorithm**
- ❑ **Pathfinding Lists**
- ❑ **World Representation**

Pathfinding

If we need to design an AI...

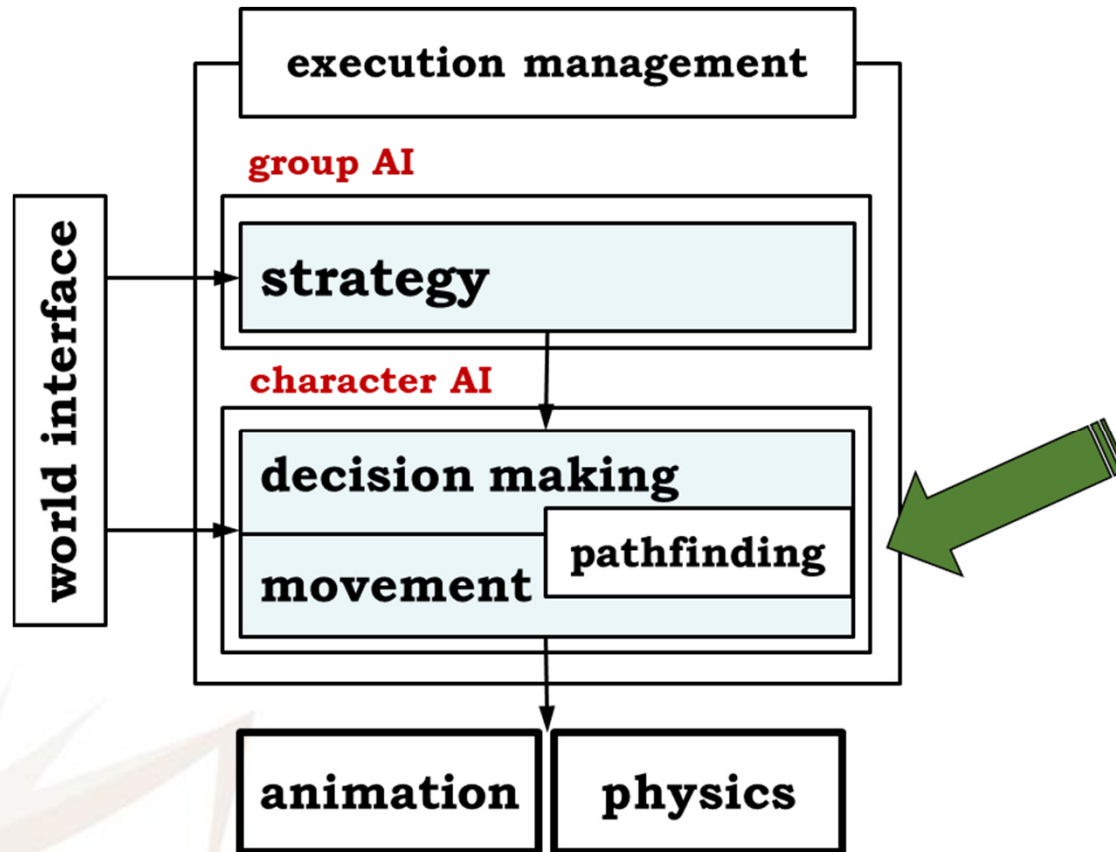
- ❑ That is able to calculate a suitable route through the game level to get from where it currently is to its goal position ...
- ❑ ... and that route is to be as short or rapid as possible, or at least, **looks smart enough** (!) ...
- ❑ ... we need to do ... **pathfinding**

Pathfinding

- ❑ Sometimes called **path planning** (but pathfinding is still the usual term used)
- ❑ Simple task: **Work out where to move to reach a goal**
- ❑ In Millington and Funge's AI model – sits on the border between decision-making and movement.
 - The **goal** is decided by decision-making AI
 - Pathfinding simply works out **how** to get there
 - Movement AI **gets** the character there

Pathfinding AI

Millington and Funge's Model



Using Pathfinding

- ❑ Vast majority of games use pathfinding solutions that are efficient and easy to implement – but pathfinding AI **cannot work directly with game level data**.
- ❑ Requires game level to be represented in a particular data structure:

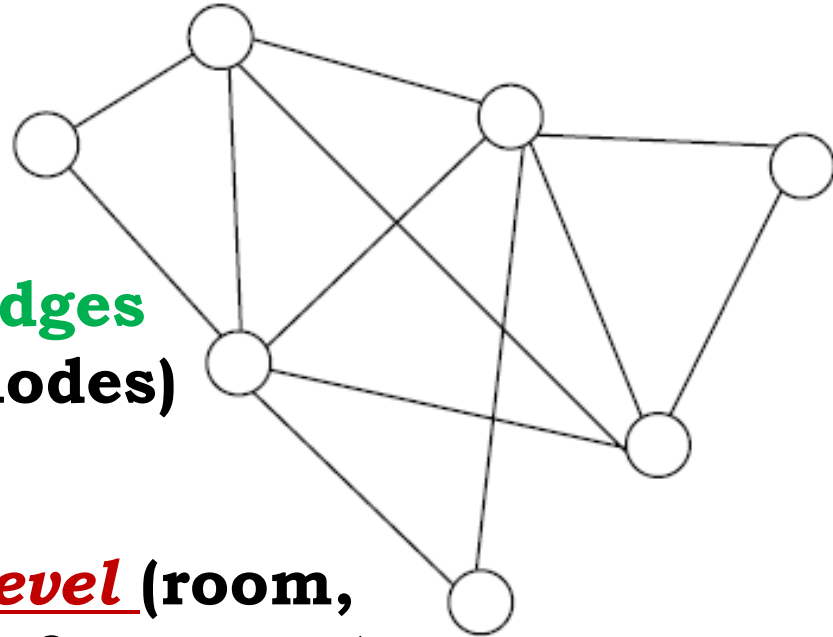
A directed non-negative weighted graph

- ❑ In particular, a graph is a simplified version of the level; the **better the simplification, the better the path**

Graphs

- **Formal representation:**

- A **set of nodes**
- A **set of connections/edges**
(an unordered pair of nodes)



- Each **node represents**

- A **region of the game level** (room, section of corridor, platform, etc.)

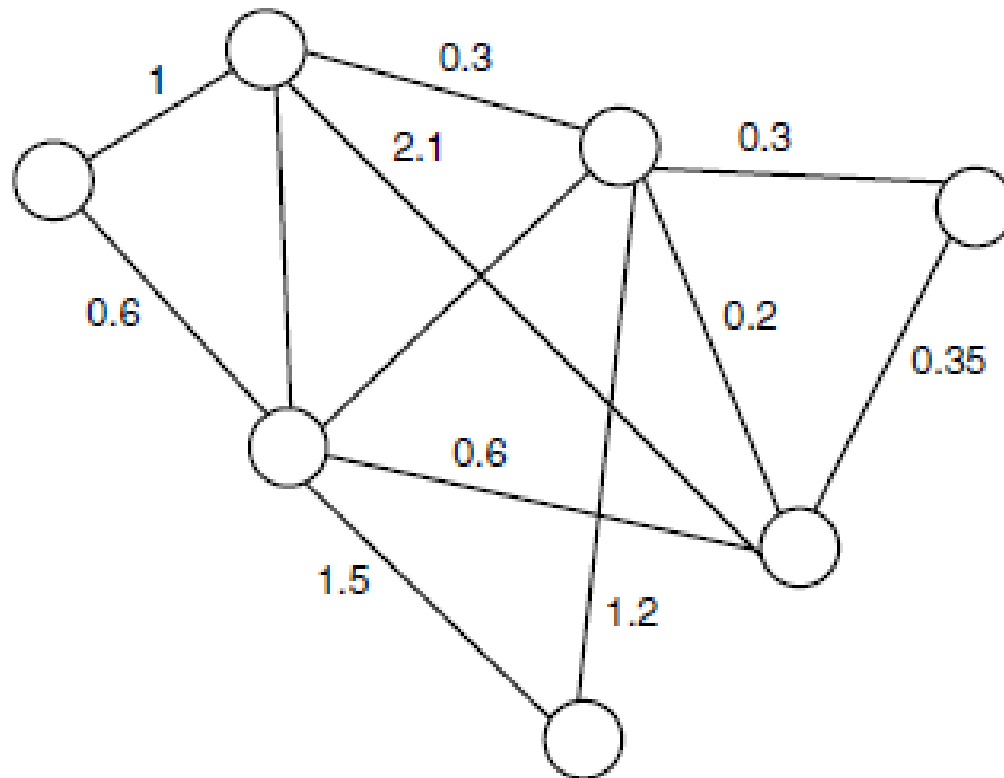
- **Connections represent**

- **Which locations (nodes) are connected**
 - E.g., if a room adjoins a corridor

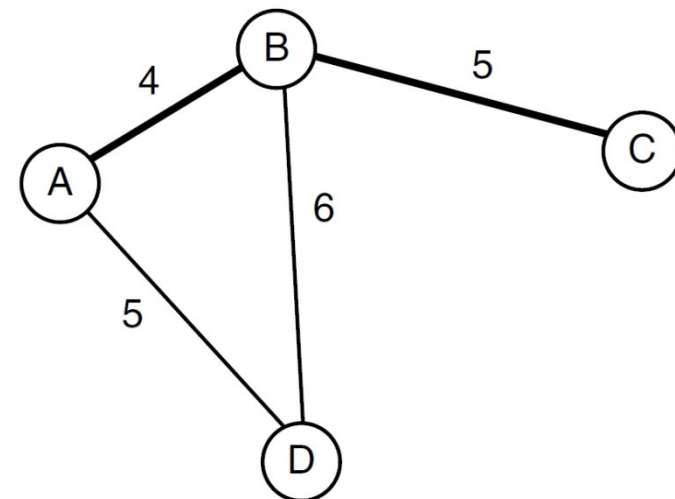
- A graph splits the whole game level into regions which are **connected together**

Weighted Graphs

- In addition to having nodes and connections, a numerical value or “**weight**” is used to label each connection with an associated cost value



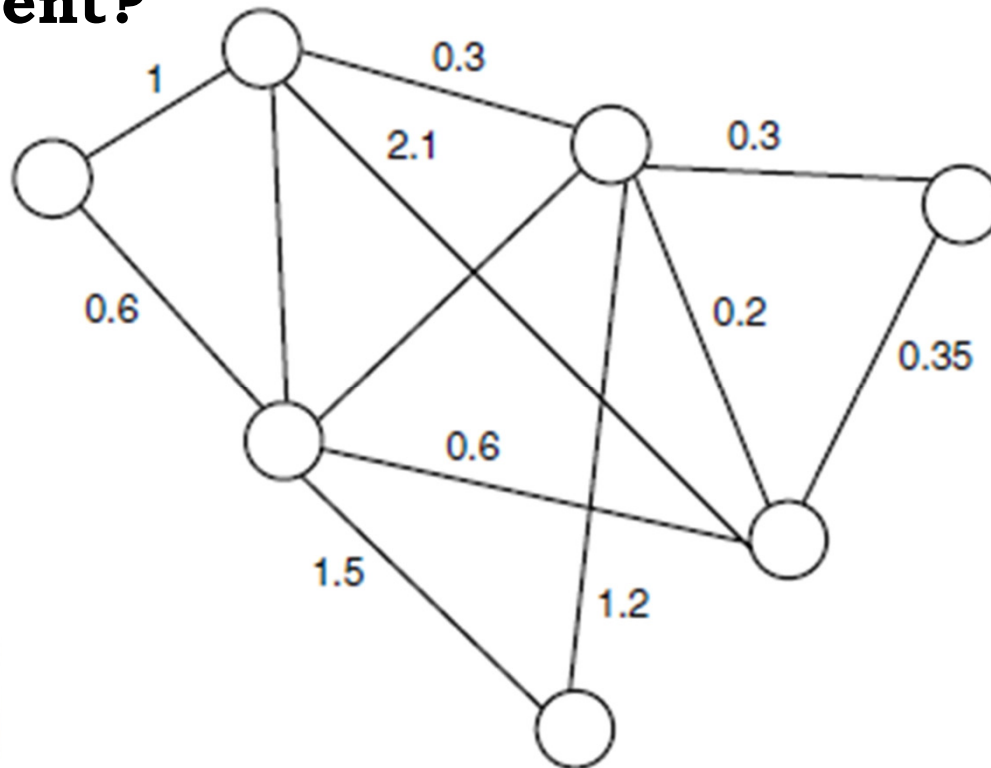
Total path cost



Graphs in Games

Weights?

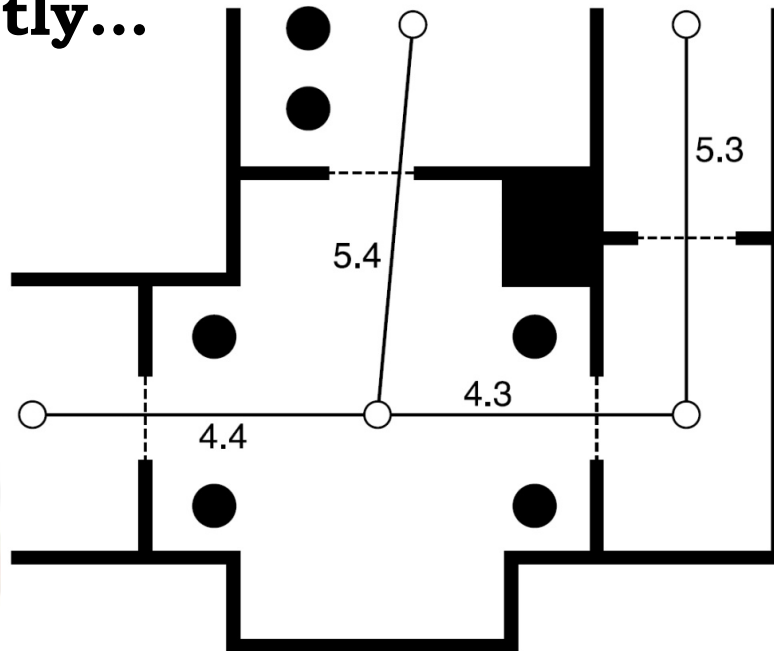
- In games, what can we use the weights (costs) to represent?



Graphs in Games

Nodes/Connections?

- ❑ **How shall we overlay** a weighted graph onto game level geometry?
- ❑ Suggest some ways to place nodes/connections correctly...

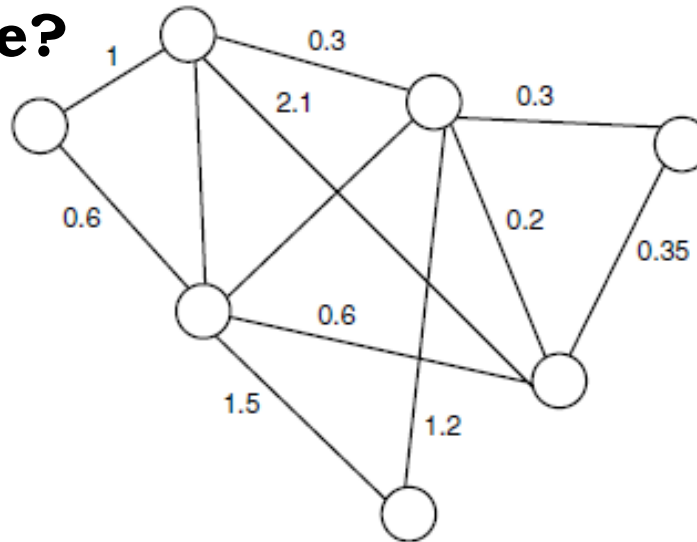


**Weighted graph
overlaid onto
level geometry**

Graphs in Games

Now, what's the whole idea?

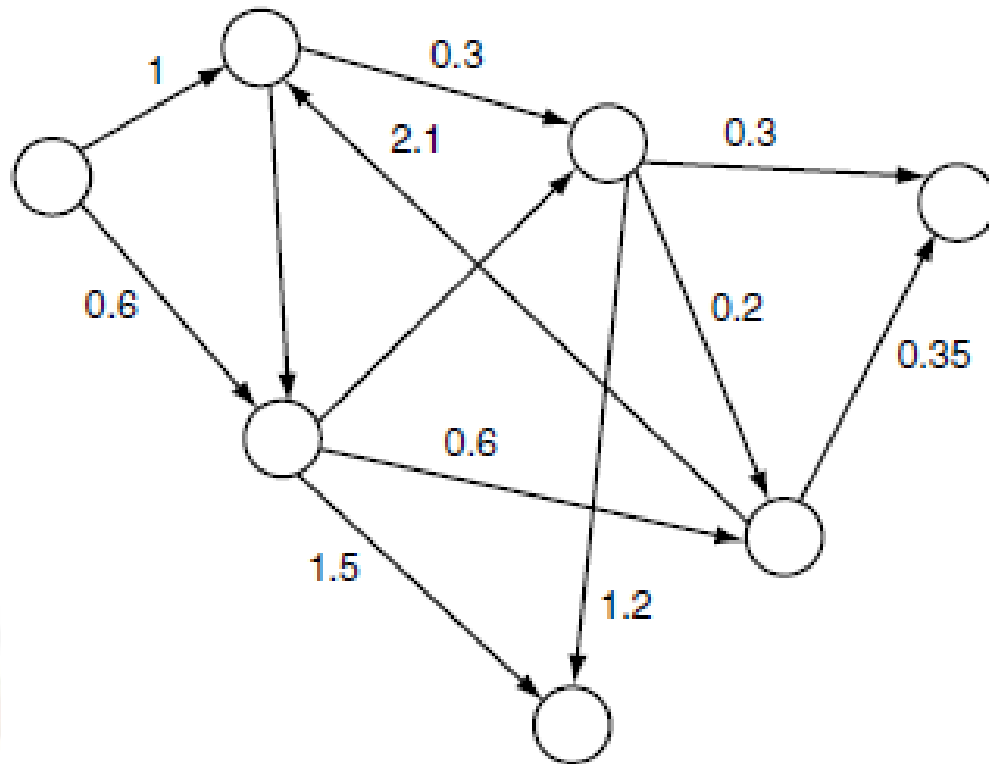
- So, back to the original idea of pathfinding: How shall we use a weighted graph (like below) to find paths in the game?



- Can we have **negative weights** in a graph?
- Can certain nodes be only connected **one way** (just like a one-way street)?

Directed Weighted Graphs

- Connections of graph are directed, or allows movement between 2 nodes to be from one direction only



Directed Weighted Graphs

- ❑ Can be useful in 2 special situations:
 - Reachability between two locations (node A can reach node B, but node B cannot reach node A)
 - Allow both *connections in opposite directions to have different weights* (the cost to move from node A to node B is different from the cost to move from node B to node A)



Graph Representations

Now, what's the whole idea?

- ❑ **Graph class** – store an array of connection objects for any node
- ❑ **Connection class**– store cost, 'from' node, 'to' node

```
class Graph:
    # Returns an array of connections (of class
    # Connection) outgoing from the given node
    def getConnections(fromNode)

class Connection:
    # Returns the non-negative cost of the connection
    def getCost()

    # Returns the node that this connection came from
    def getFromNode()

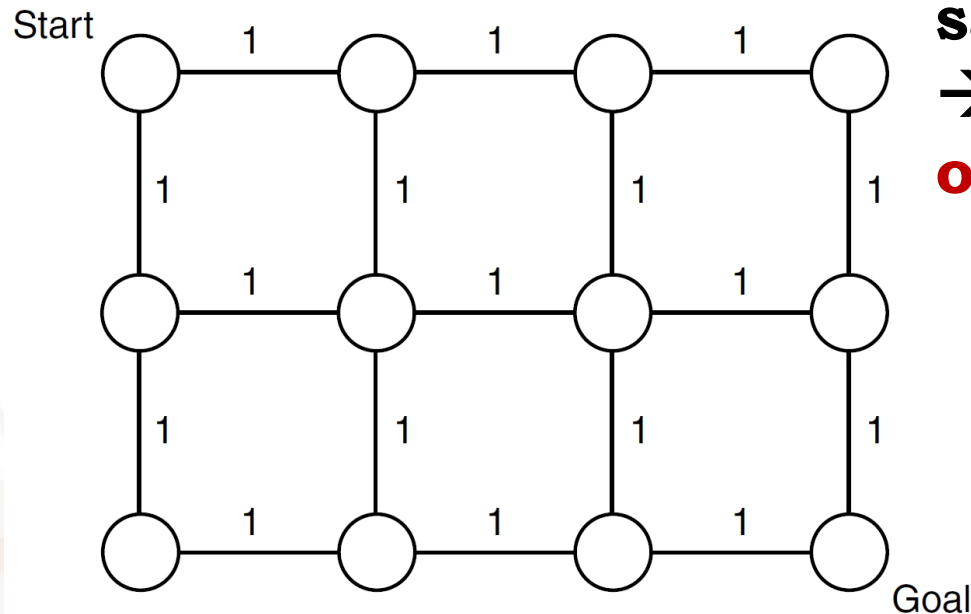
    # Returns the node that this connection leads to
    def getToNode()
```

Dijkstra's Algorithm


- ❑ Named after **Edsger Dijkstra**, a mathematician
- ❑ Originally designed to solve a problem in mathematical graph theory called “shortest path”, not for games
- ❑ **Idea**: Finding the *shortest path from one start point to one goal point*
- ❑ Dijkstra's Algorithm is designed to find the shortest routes to everywhere from an initial point – **Wasteful?**
- ❑ We will examine Dijkstra's because it is a **simpler version of** the main pathfinding algorithm **A***.

The Problem

- **Aim:** Given a graph and two nodes (start and goal), generate a path such that the total path cost of that path is minimal among all possible paths from start to goal
- There may be any number of paths with the same minimal cost
→ Just **returning any one** will do



Same Cost Paths

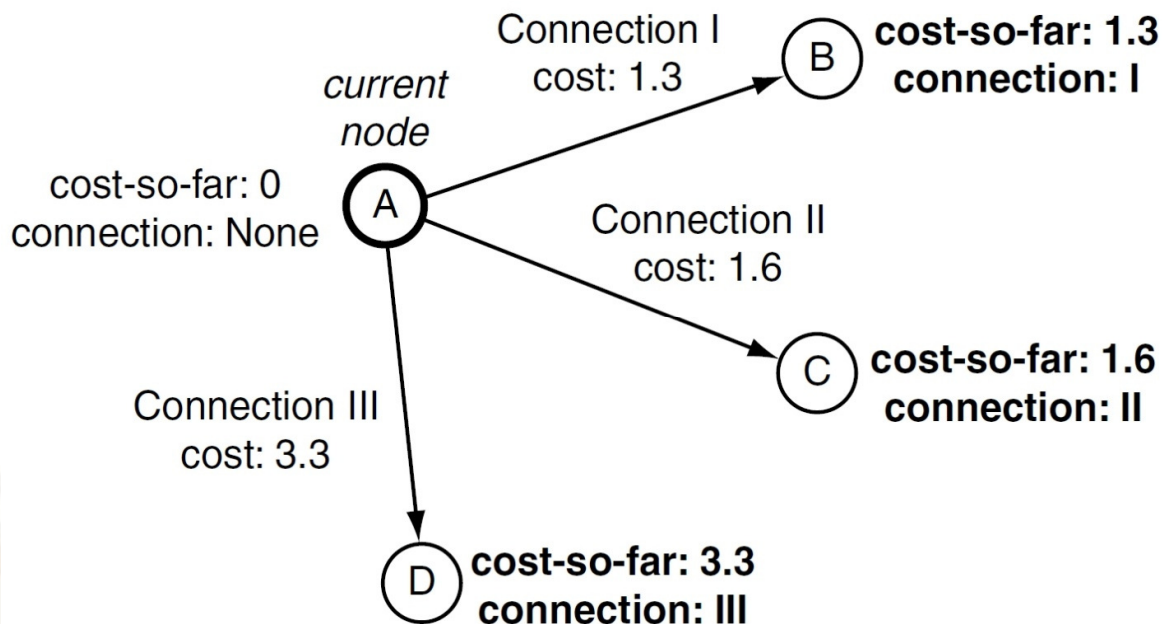
- ❑ The path we expect to be returned consists of a set of connections, not nodes.
- ❑ Many games do not consider having more than one connection between any pair of nodes (**only one possible path?**)

- ❑ With many connections between any pair of nodes, an optimum path is needed to make the **AI look smart!** So, keep track of multiple connections, use the connection of these with smallest cost.
- ❑ This is particularly useful if the costs **change over the course of the game** or between different characters

The Algorithm

- ❑ Spread out from the start node along its connections
- ❑ Each time, keep a record of the direction it came from with “arrows” (stored as directed edges)
- ❑ Eventually, when the goal is reached, follow the “arrows”, in reverse, back to the start point to generate the complete minimal route
- ❑ Works in iterations
- ❑ Each iteration: consider one node of a graph and follow its outgoing connections (to other nodes)
- ❑ Each iteration’s node is called the “*current node*”

The Algorithm

- ❑ Processing the Current Node at each iteration
 - Consider outgoing connections from current node
 - For each connection, **finds the end node** and stores the total cost of the path so far as the “**cost-so-far**” (from the start node)

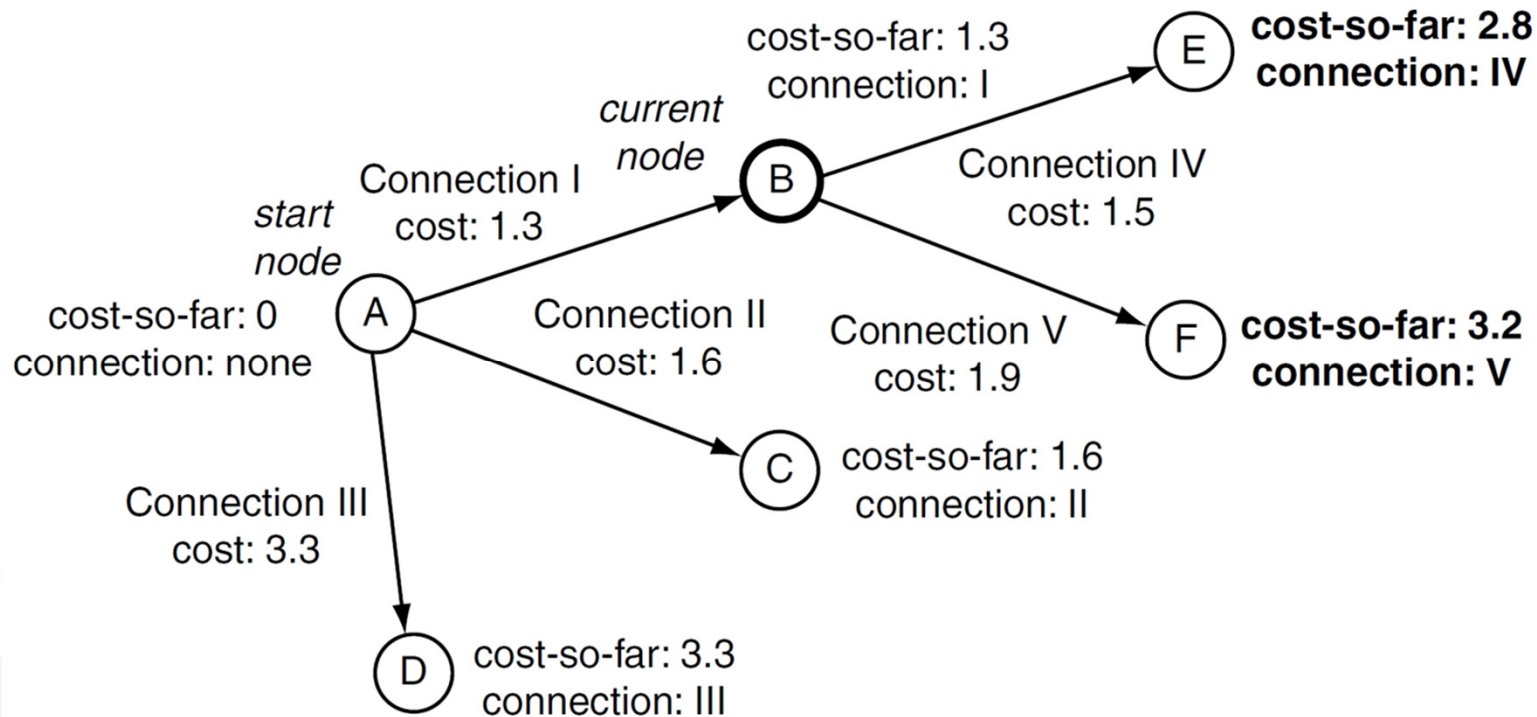


- Each connected node stores a) cost-so-far and b) which connection to the node was used to get the cost-so-far

The Algorithm

- After 1st iteration, for neighboring unvisited nodes

Cost-so-far = Sum of **connection cost** + **cost-so-far of current node**



Node Lists

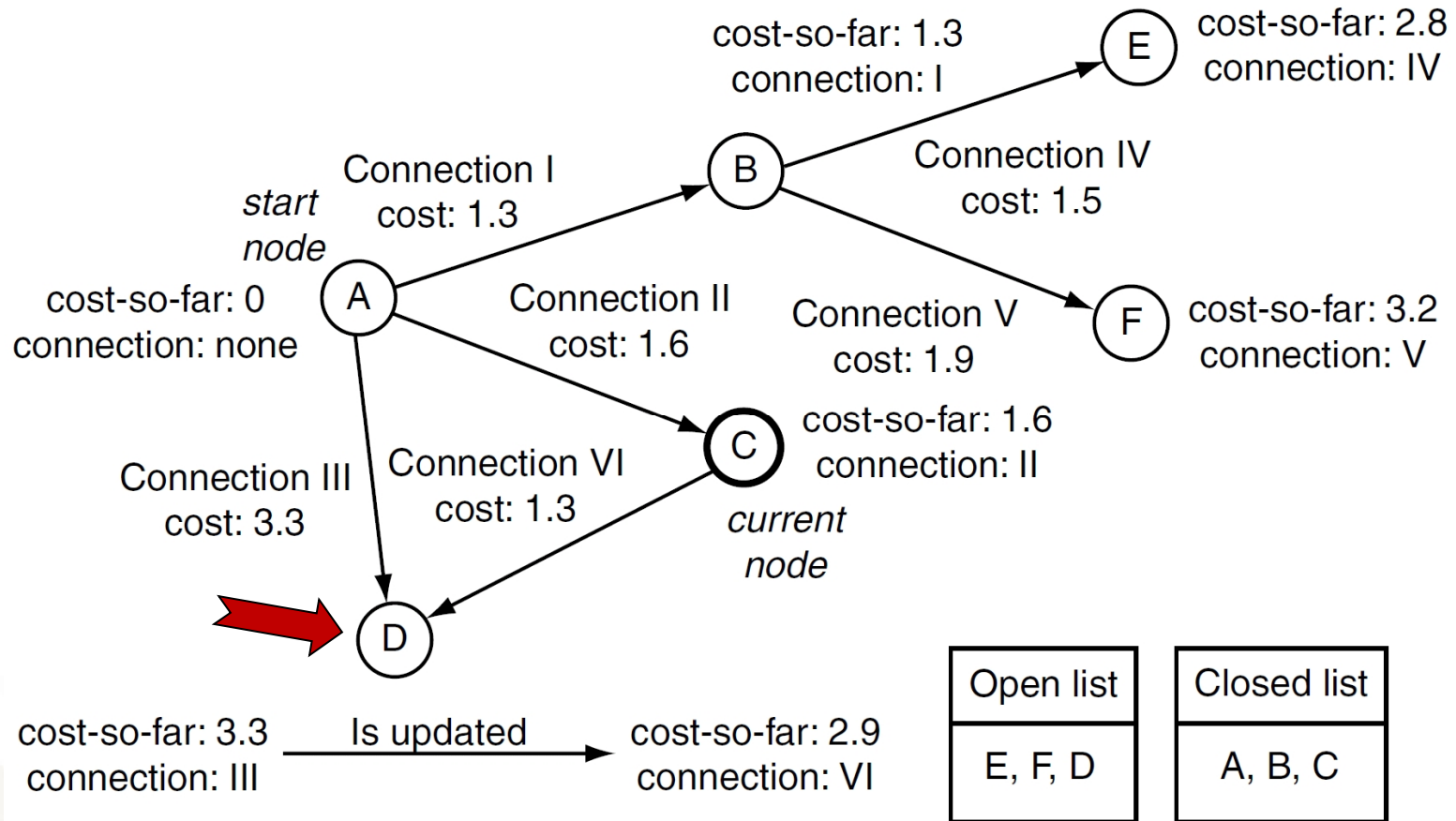
- ❑ Keeps track of all the nodes seen so far in 2 lists:
 - **Open List**
 - Records all nodes seen, but haven't been processed in its own iteration yet
 - **Closed List**
 - Records all nodes that have been processed in its own iteration (all “current node” s)
- ❑ Nodes in neither list are “**unvisited**”
- ❑ At each iteration, a node from the open list with **smallest cost-so-far** is chosen (and processed), then removed from the open list and placed in the closed list

Cost-so-far for Open and Closed Nodes

- ❑ What happens when we arrive at an open or closed node (**again**) during an iteration?
- ❑ If there are new values of the cost-so-far that are lower than the existing value of node, an **update** of that node is needed, regardless of which list it's in:
 - The **open nodes** that have their values updated simply stay on the open list
 - For **closed** nodes — simply force the algorithm to recalculate and re-propagate new values:
 - ✓ **Remove node from closed list, place back into open list with its values revised**
 - ✓ It will be **processed once again** and have its connections reconsidered

A few iterations later...

□ Notice the updating done on node D



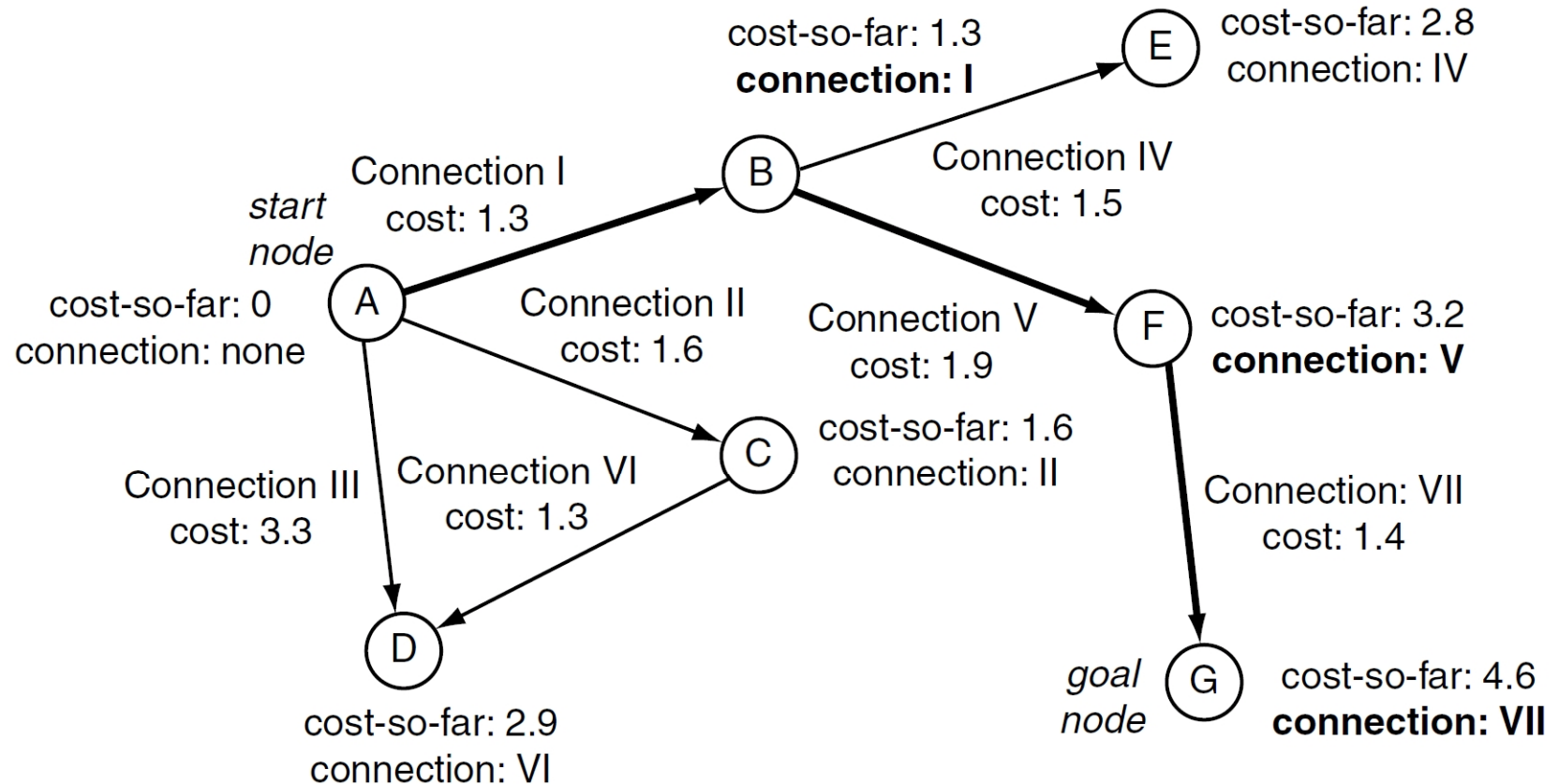
Terminating the Algorithm

- ❑ Basic Dijkstra's Algorithm – terminates when the open list is empty (no more nodes to process), *and all nodes are in the closed list*
- ❑ In practical pathfinding, we can terminate earlier once we have
 - placed the goal node on the open list, **and**
 - the **goal node** is the node with the **smallest** cost-so-far *on the open list* (an added **heuristic**)
- ❑ Why do we add this heuristic rule?

Retrieving the Path

- ❑ Final step
- ❑ Start from goal node, look back at the connection that was used to arrive to this node; then for the node at the other end of the connection, look at the connection used to arrive at that node; etc. Continue until reaching the start goal
- ❑ The list of connections need to be reversed to obtain the right path order

Final Result



Connections working back from goal: **VII, V, I**
Final path: **I, V, VII**

Data Structures & Interfaces

- ❑ Graph – **Rarely** a performance bottleneck (can be created offline)
- ❑ Simple List (for reporting connections in Graph)
 - Not performance critical, **use a basic linked list (list) or resizable array (vector)**
- ❑ Pathfinding Lists
 - Large Open and Closed lists **can** affect performance. **Need optimization to perform these operations:**
 - A.** Adding/removing entry,
 - B.** Finding smallest element,
 - C.** Finding entry in list corresponding to a particular node

Practical Performance of Dijkstra's

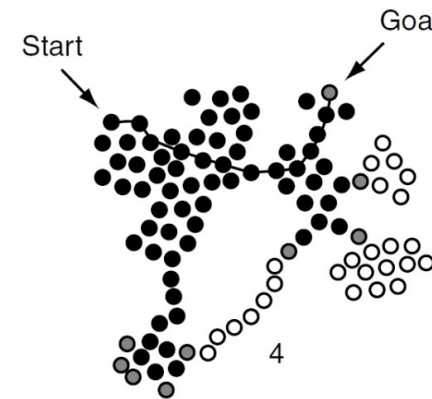
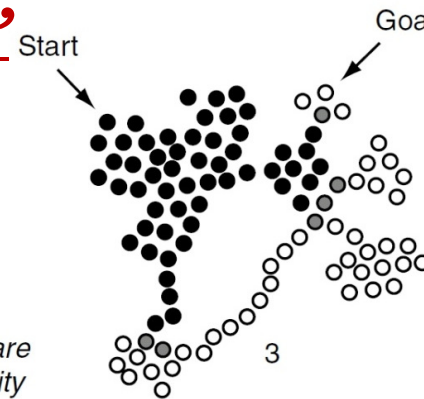
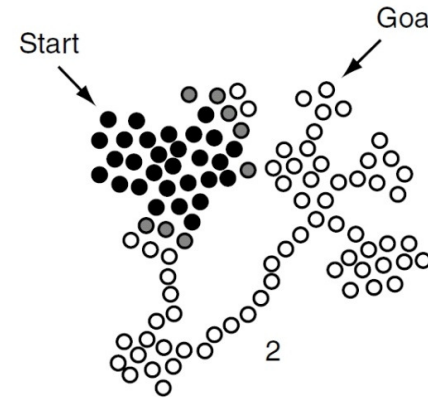
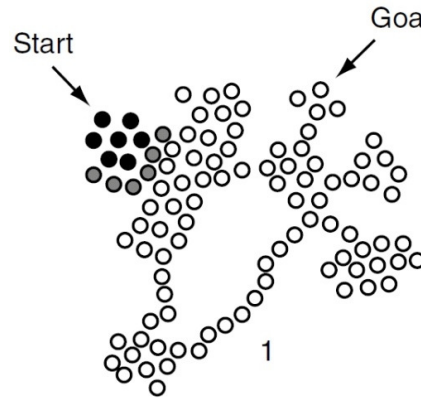
- ❑ Depends mostly on **performance of operations** in pathfinding list(s) data structure
- ❑ Theoretically (**n nodes**, **m average** number of connections per node):
 - Time complexity (execution speed): $O(nm)$
 - Space complexity (memory): $O(nm)$ worst-case
- ❑ Select implementations:
 - Min Priority Queue: $O(n^2 + nm)$
 - Fibonacci Heap: $O(n \log n + nm)$
- ❑ Note: **$1 < m \leq n-1$** for a connected graph with **$n > 2$**

Weaknesses

- ❑ Searches whole graph indiscriminately for shortest path

- ❑ ***Wasteful*** for point-to-point pathfinding

- ❑ Suffers from ***terrible amount of “fill”*** (nodes that were considered but never made part of the final route)



Key
● Open nodes
● Closed nodes
○ Unvisited nodes

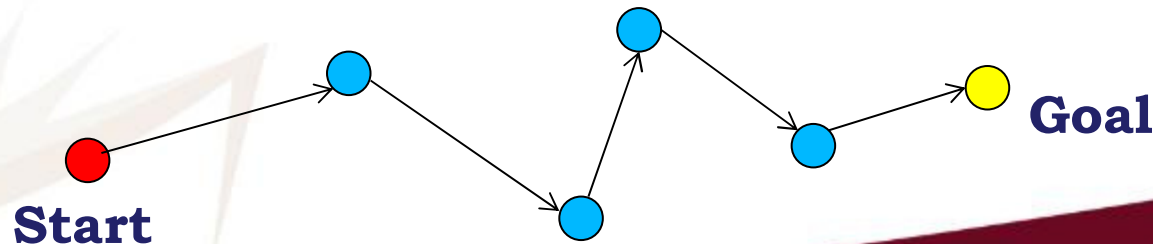
NB: connections are hidden for simplicity

A* Algorithm

- ❑ Most widely used for pathfinding in games
 - *Simple to implement*
 - *Efficient*
 - Lots of scope for optimization
- ❑ Unlike Dijkstra algorithm, A* is designed for point-to-point pathfinding, **not** for solving the shortest path problem in graph theory.
 - Always returns a single path from source to goal
 - The “best” possible path

The Problem

- ❑ Given a graph (a **directed non-negative weighted graph**) and two nodes in that graph (a start and a goal node)
- ❑ Generate a path → the ***total path cost of the path is minimal*** among all paths from start to goal
- ❑ If there are more than one possible solution, ***any minimal path will do***
- ❑ Path ***should consist of a list of connections from the start node to goal node***



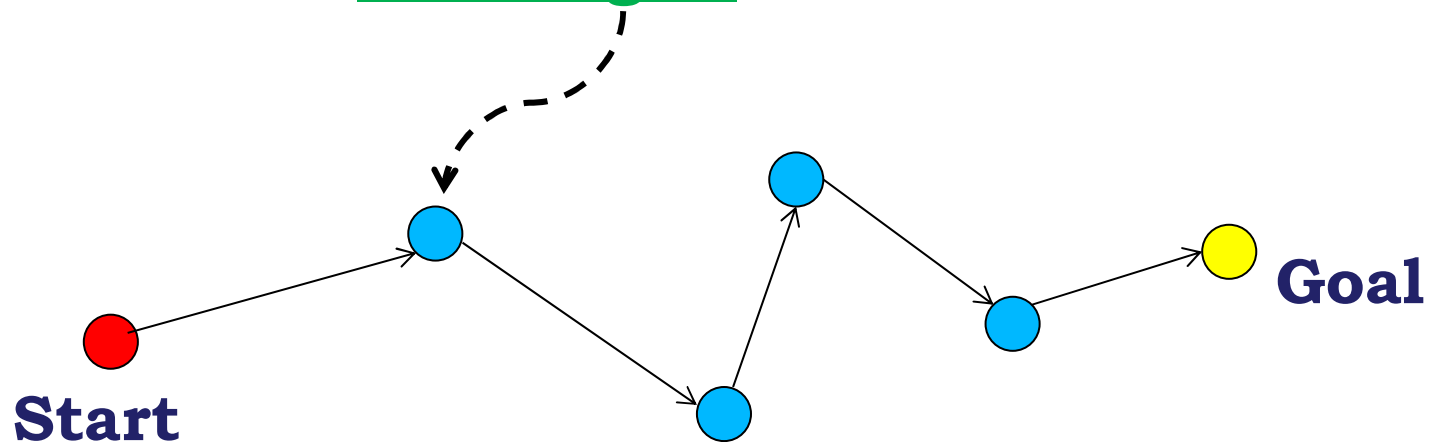
Differences from Dijkstra

- ❑ **Dijkstra** → *Always considered the open node with lowest cost-so-far* for processing
- ❑ **A*** → Consider node that **most likely** to lead to shortest overall path
- ❑ “most likely” → controlled by a “**heuristic**”, an *estimated rule of thumb*
- ❑ If accurate heuristic used: **Efficient**
- ❑ If bad heuristic used: **May perform worse than Dijkstra!**

“heuristic” for A*

- Think about **how** an “estimated total cost” should be calculated?...

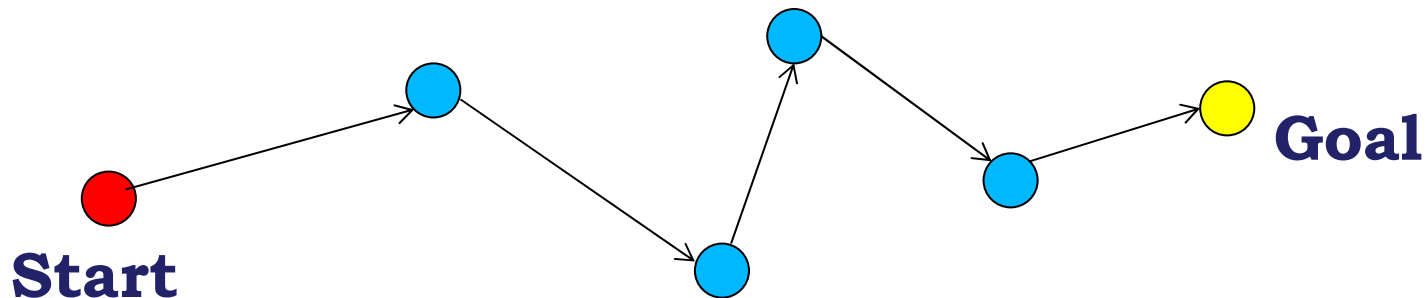
How to estimate total cost from here to goal for this node?



- Think of a good heuristic to use?

“heuristic” for A*

- A node with smallest estimated-total-cost should have:
 - A relatively small **cost-so-far** (**Dijkstra only has this**)
 - A relatively small **estimated distance to go to reach goal** (**the heuristic**)



- **estimated-total-cost = cost-so-far + heuristic cost**

Pathfinding Lists

- ❑ Just like Dijkstra, A* also keeps an **open** list of nodes and a **closed** list of nodes
 - Nodes are moved to open list as they are found at end of connections (previously “unvisited”)
 - Nodes are moved to closed list *as they are processed in their own iteration*
- ❑ Unlike Dijkstra, node with **smallest estimated-total-cost** is selected from the open list, not the node with smallest cost-so-far

Using the “heuristic”

- **A*** stores 2 additional values – heuristic value and *estimated total cost*

Node A (start node)

heuristic: 4.2

cost-so-far: 0

connection: none

estimated-total-cost: 4.2

closed

Node B

heuristic: 3.2

cost-so-far: 1.3

connection: AB

estimated-total-cost: 4.5

closed

Node D

heuristic: 2.8

cost-so-far: 2.8

connection: BD

estimated-total-cost: 5.6

open

A

1.3

B

1.5

D

1.1

1.7

E

heuristic: 1.6

cost-so-far: 3.0

connection: BE

estimated-total-cost: 4.6

open

Node C

heuristic: 3.7

cost-so-far: 1.1

connection: AC

estimated-total-cost: 4.8

open

C

1.5

1.6

F

Node F

heuristic: 1.4

unvisited

G

Node G (goal node)

heuristic: 0.0

unvisited

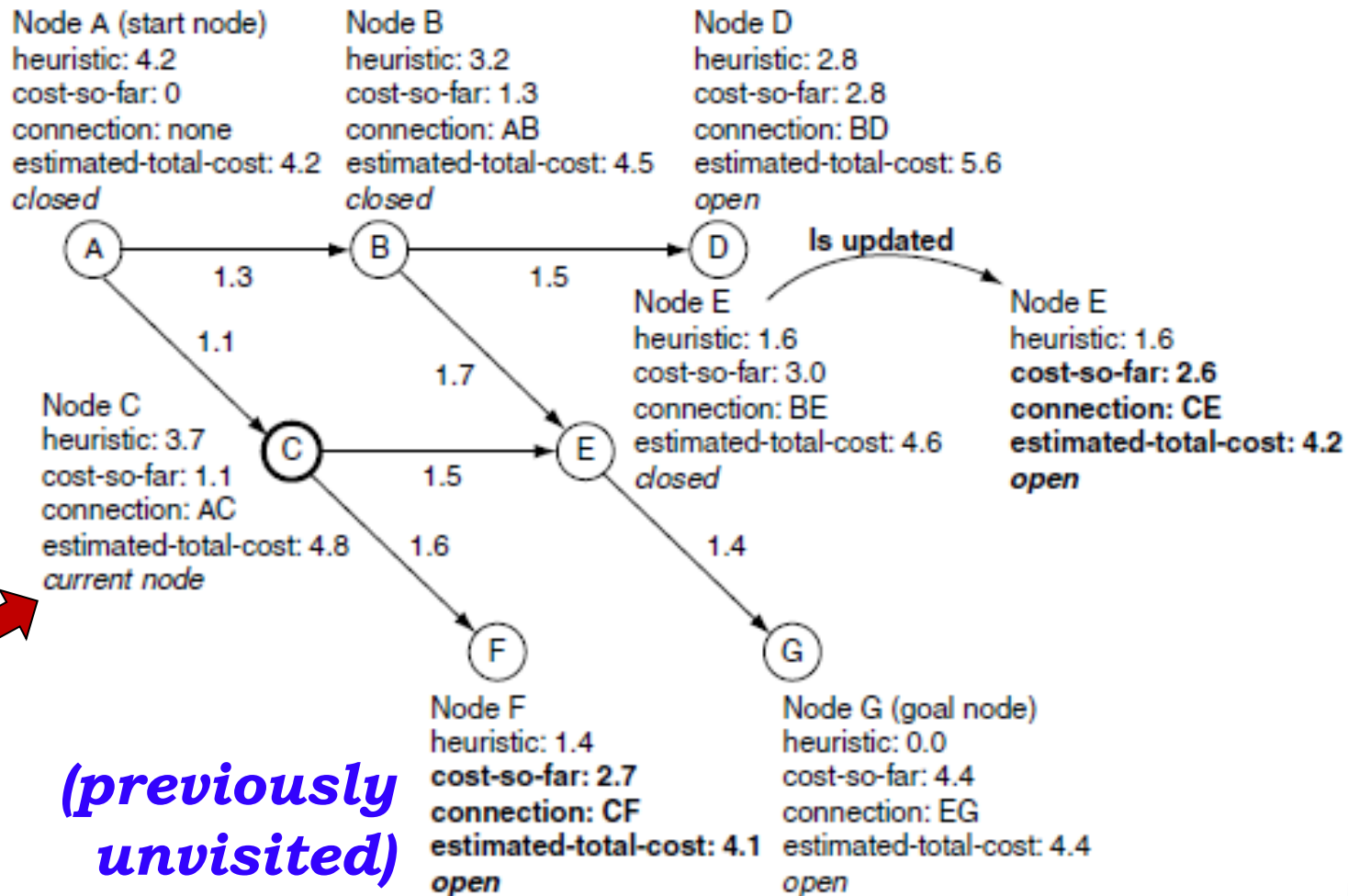
Calculating Costs

- ❑ **estimated-total-cost = cost-so-far + Heuristic cost**
 - As with Dijkstra's algorithm, the cost-so-far is calculated using the actual *total cost of the path* travelled so far from the start to the current node
 - The heuristic cost **cannot be negative** (obviously!)
- ❑ **To update a previously visited node**
 - Compare cost-so-far, not the estimated total cost (with heuristic added)
 - This is the only reliable “real” value. Heuristics involve estimation (and are usually fixed values)

Updating Costs

- If there are **new values** of the cost-so-far that are lower than the existing value of node, an **update** of that node is needed, as with Dijkstra, regardless of which list it is in:
 - The **open nodes** that have their values revised simply stay on the open list
 - For **closed** nodes, remove the node from closed list, **place it back** into open list with its values revised - to be processed once again and have its connections reconsidered

Updating Costs



Terminating the Algorithm

- ❑ A* terminates when the goal node is the smallest node on the open list (with smallest estimated-total-cost)
- ❑ This does not guarantee that shortest path is found. Why?
- ❑ It is natural for A* to run a bit longer to generate **a guaranteed optimal result**
- ❑ One way: Require that A* terminates when node in open list with smallest cost-so-far has a cost-so-far greater than cost of path found to goal (almost like Dijkstra)
- ❑ Drawback: **Same fill problem as Dijkstra!!**

Algorithm Performance

- Depends on data structure used

- Time complexity, $O(lm)$
- Space complexity, $O(lm)$
- l : number of nodes whose total estimated-path-cost is less than that of goal, $< n$ from Dijkstra
- m : *average number of outgoing connections from each node*

l = amount
of “fill”

- Heuristic calculation is usually $O(1)$ execution time and memory, so **no effect on overall performance**. Can be even calculated offline in some cases.
- Refer to [M&F] for more detail on pseudocode and data structures involved