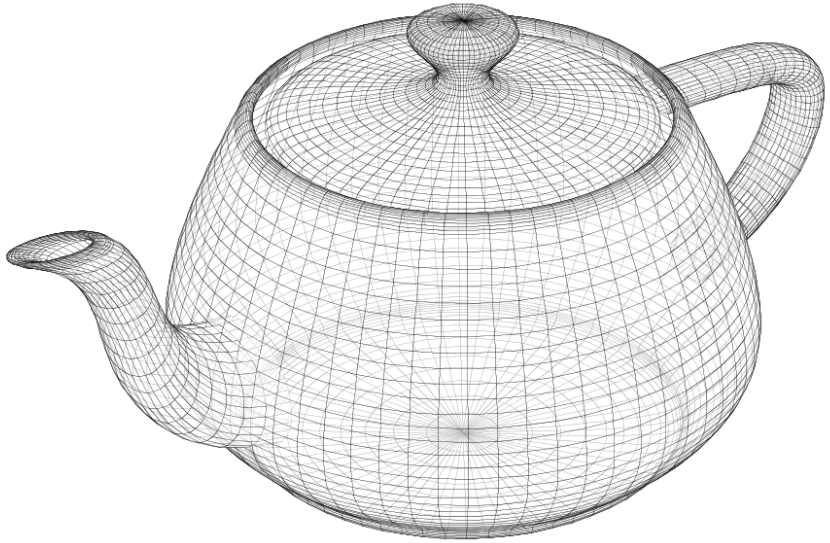


Game AI

Pathfinding: Shortest Path

Professor Eric Shaffer

ILLINOIS

Pathfinding in Games

Game characters typically need to move around a level

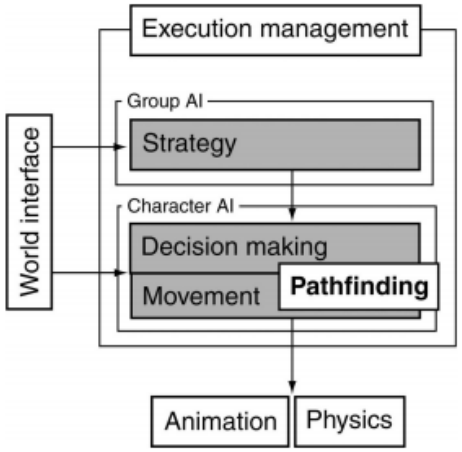
- Guards in a stealth game
- Units in a real-time strategy game

Scripted movement is easy for player to fool

Random wandering usually looks unrealistic…often unchallenging for player

ILLINOIS

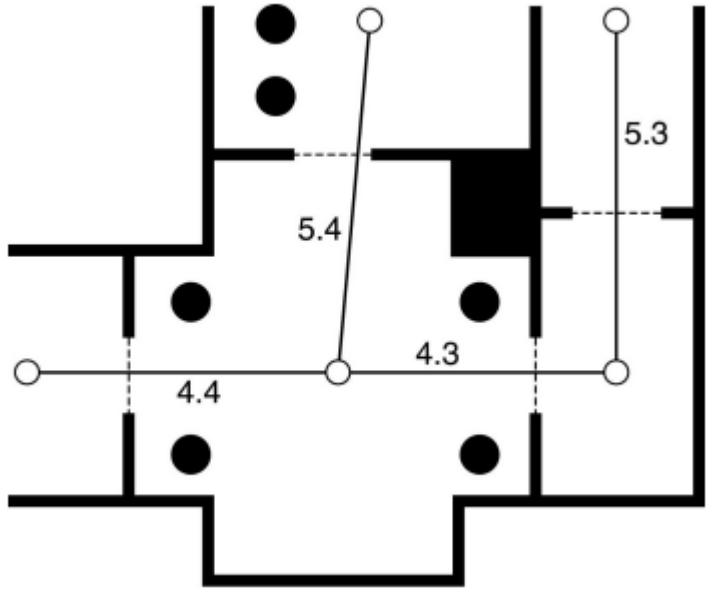
Pathfinding Algorithms



- Majority of games use some variation of an algorithm called A*
  - More complicated version of Djikstra's shortest path algorithm
- Djikstra does get used in tactical AI in games…we'll start with it

Pathfinding Graph

- Neither A* nor Dijkstra work directly on the geometry that makes up a game level
- They rely on a simplified version of the level represented in the form of a graph
- The simplification must be done well



- Pathfinding algorithms use a directed non-negative weighted graph.

ILLINOIS

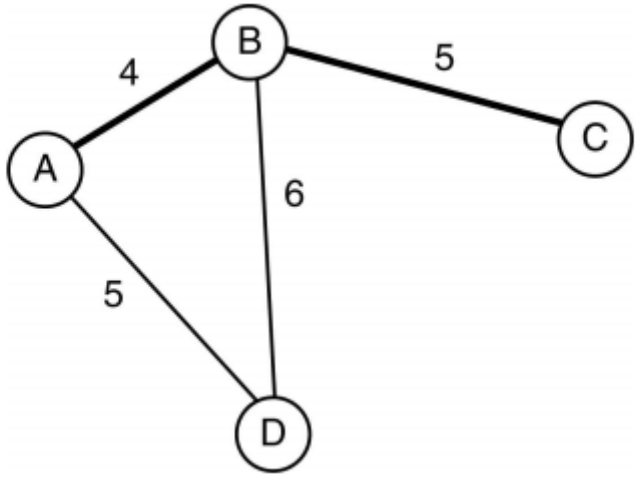
Representation and Terminology

```
class Graph:
    # An array of connections outgoing from the given node.
    function getConnections(fromNode: Node) -> Connection[]

class Connection:
    # The node that this connection came from.
    fromNode: Node

    # The node that this connection leads to.
    toNode: Node

    # The non-negative cost of this connection.
    function getCost( ) -> float
```



The way you represent the graph can significantly impact efficiency and storage cost...
…but we will ignore those considerations for now

No settled terminology for pathfinding graphs in games…we'll use nodes and connections

ILLINOIS

Djikstra's Algorithm

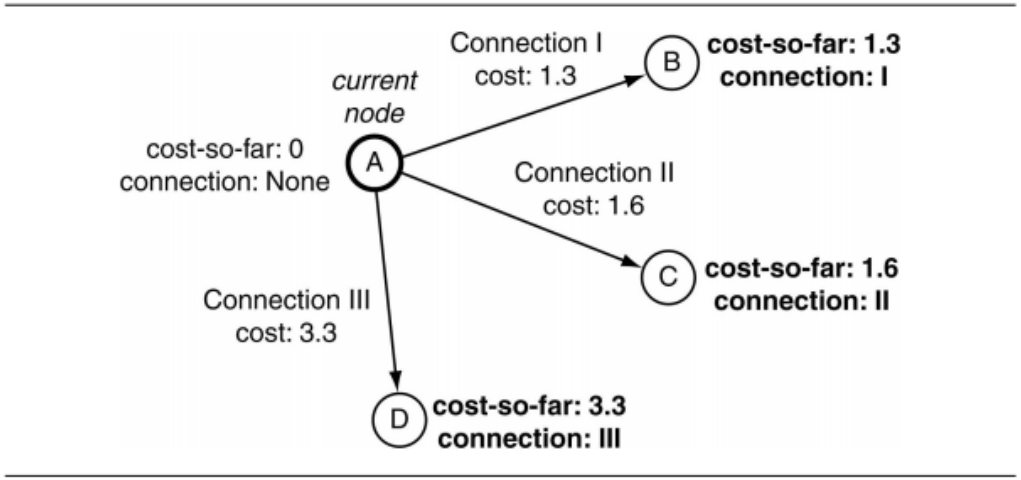
Informally, Dijkstra considers paths that spread from a start node along connections

As it spreads to more distant nodes, it keeps a record of the direction it came from

Eventually, one of these paths will reach the goal node

The algorithm can follow the path back to its start point to generate the complete route.

Because of the way Dijkstra regulates the spreading process,
it guarantees that the first valid path it finds will also be the cheapest



Algorithm

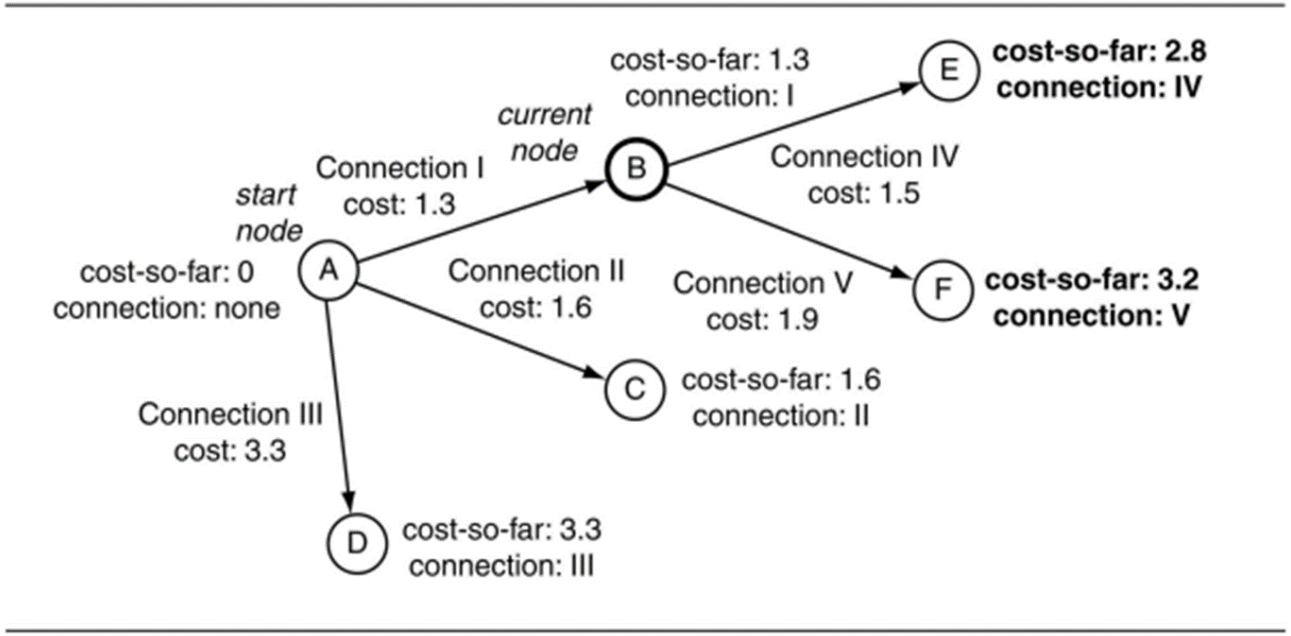
- Iterative algorithm
  - Explore neighbors of one node each iteration until termination
- Track cost-so-far for each node
- Keep 2 Lists
  - Open → all nodes that have been discovered but not explored
  - Closed → all nodes that have been explored
- Initialization
  - Open → start node
  - Closed → Empty
  - Start node cost-so-far = 0
- At each iteration explore the node from Open with lowest cost-so-far

Initial cost-so-far for nodes

Start node has cost-so-far of 0

Each time a new node is discovered:

the cost-so-far is the sum of the connection cost and cost-so-far of current node



Updating cost-so-far

If we arrive at an open or closed node during an iteration:

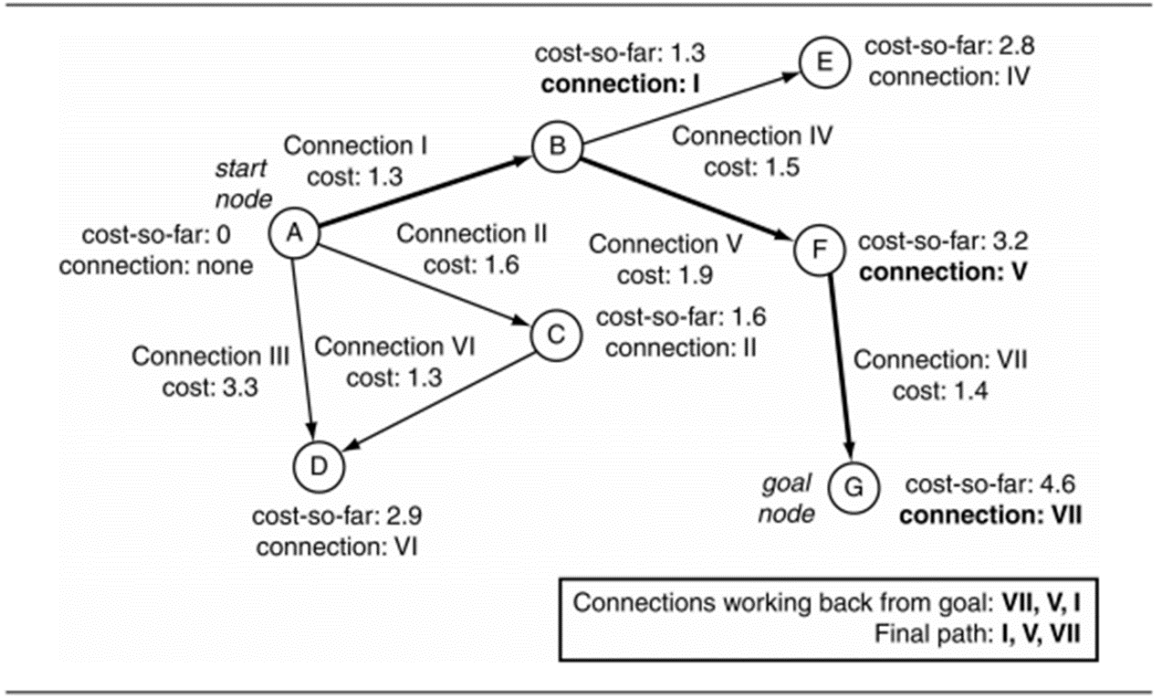
Calculate the cost-so-far value as normal,

- If the new cost-so-far value is smaller than the node's current cost-so-far, then update it with the better value, and set its connection record.
- The node should then be placed on the open list.
- If it was previously on the closed list, it should be removed from there.

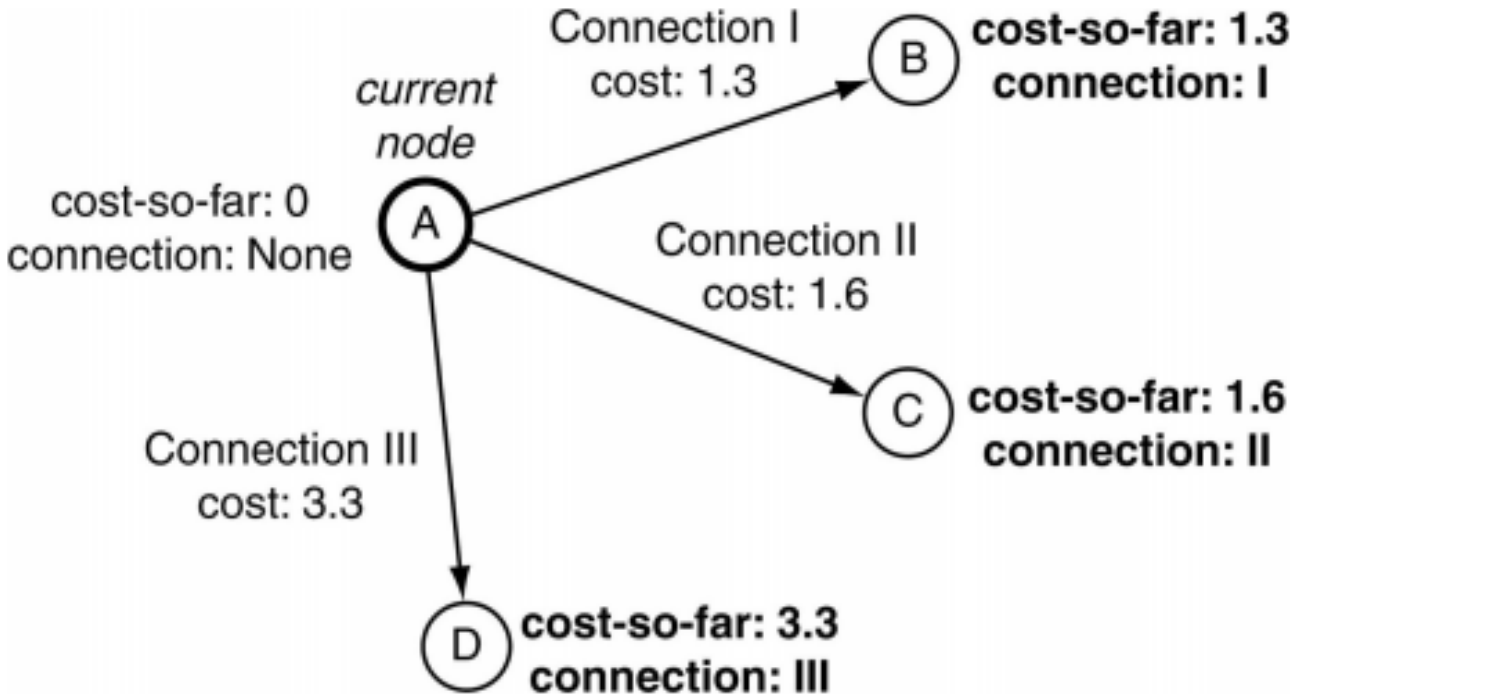
Updating a closed node won't happen in Djikstra but can a A*…so we will keep that logic for now

Termination

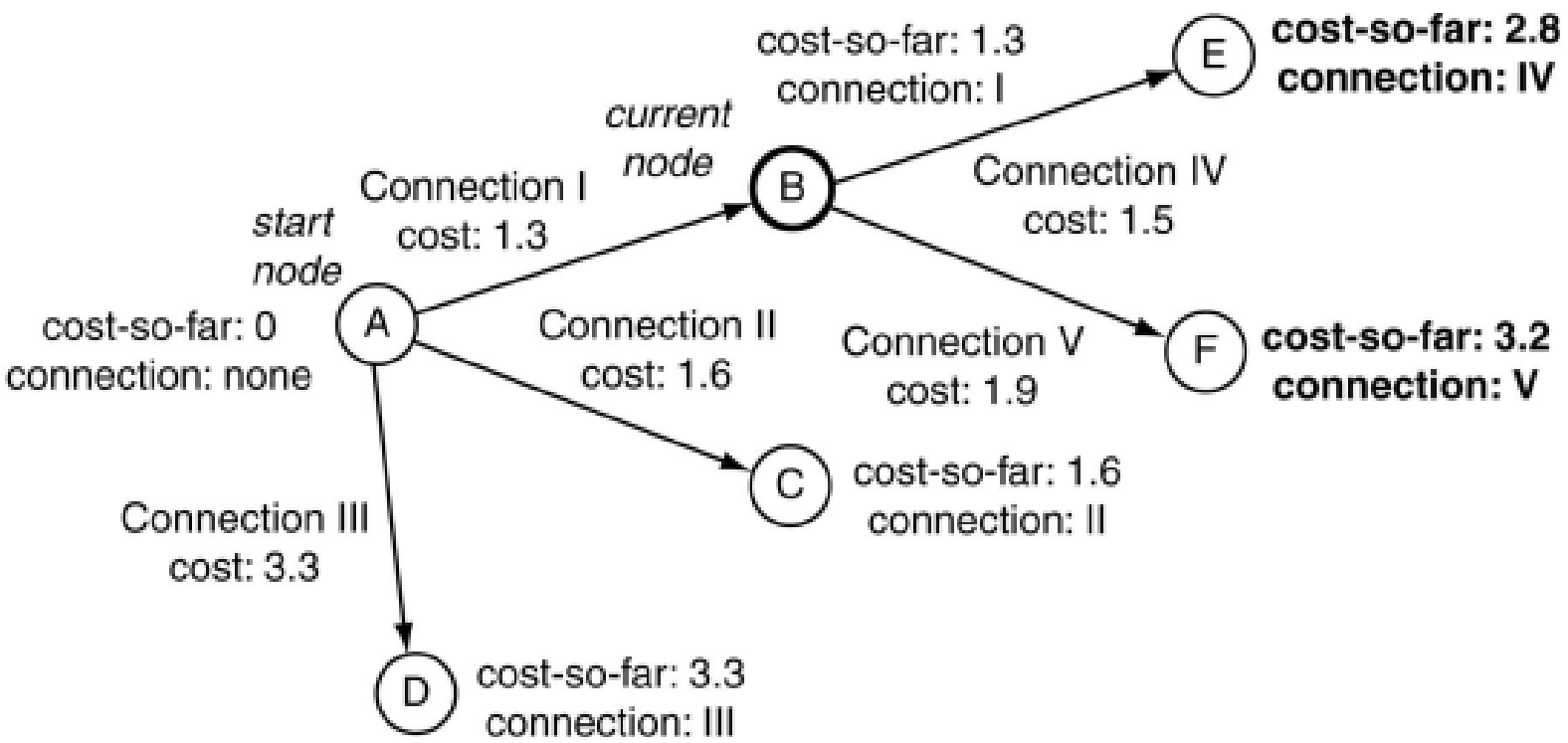
- The basic Dijkstra algorithm terminates when the open list is empty
- For pathfinding, we are only interested in reaching the goal node
  - Terminates when goal node is the smallest node on the open list.



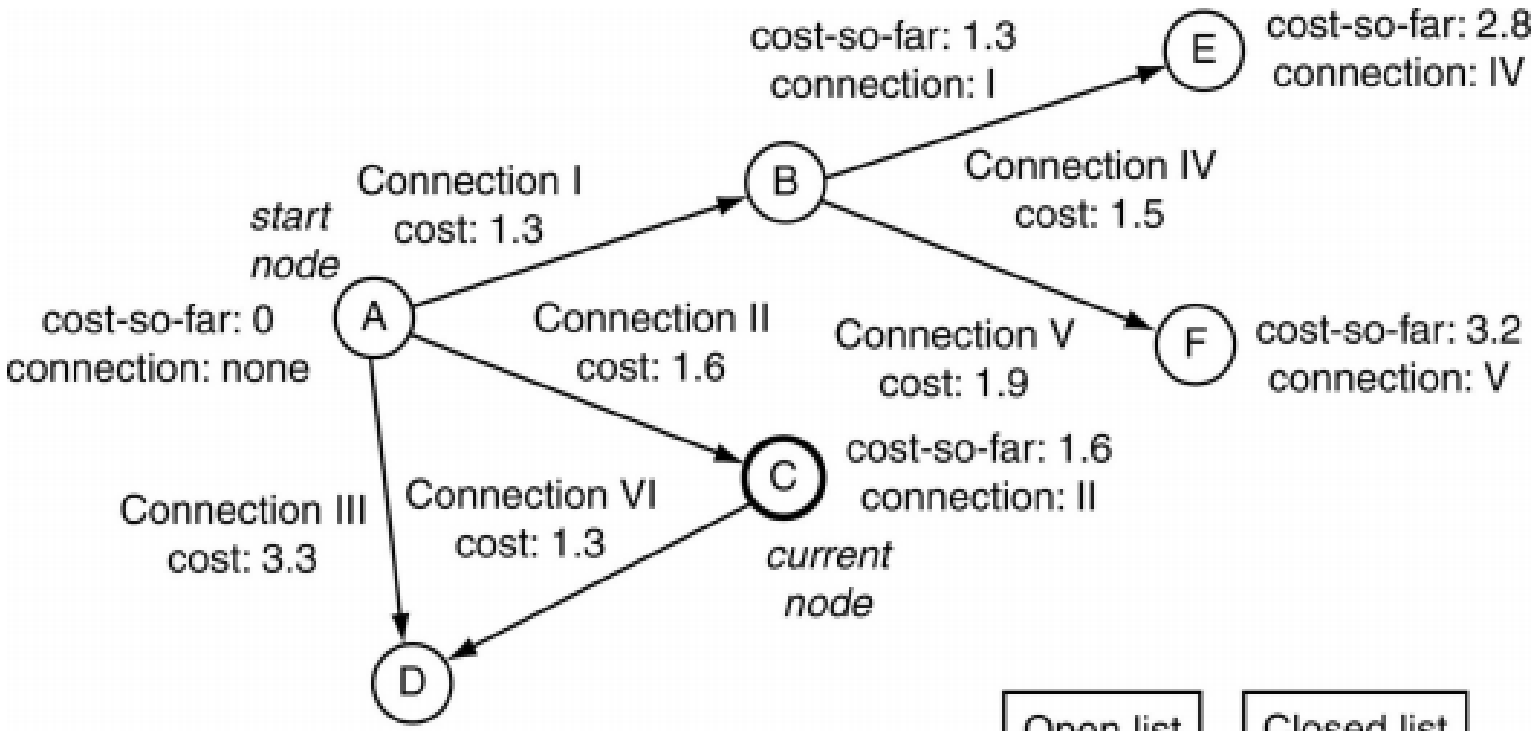
Example


Connection I
cost: 1.3
B
cost-so-far: 1.3
connection: I
current
node
cost-so-far: 0
connection: None
A
Connection II
cost: 1.6
C
cost-so-far: 1.6
connection: II
Connection III
cost: 3.3
D
cost-so-far: 3.3
connection: III

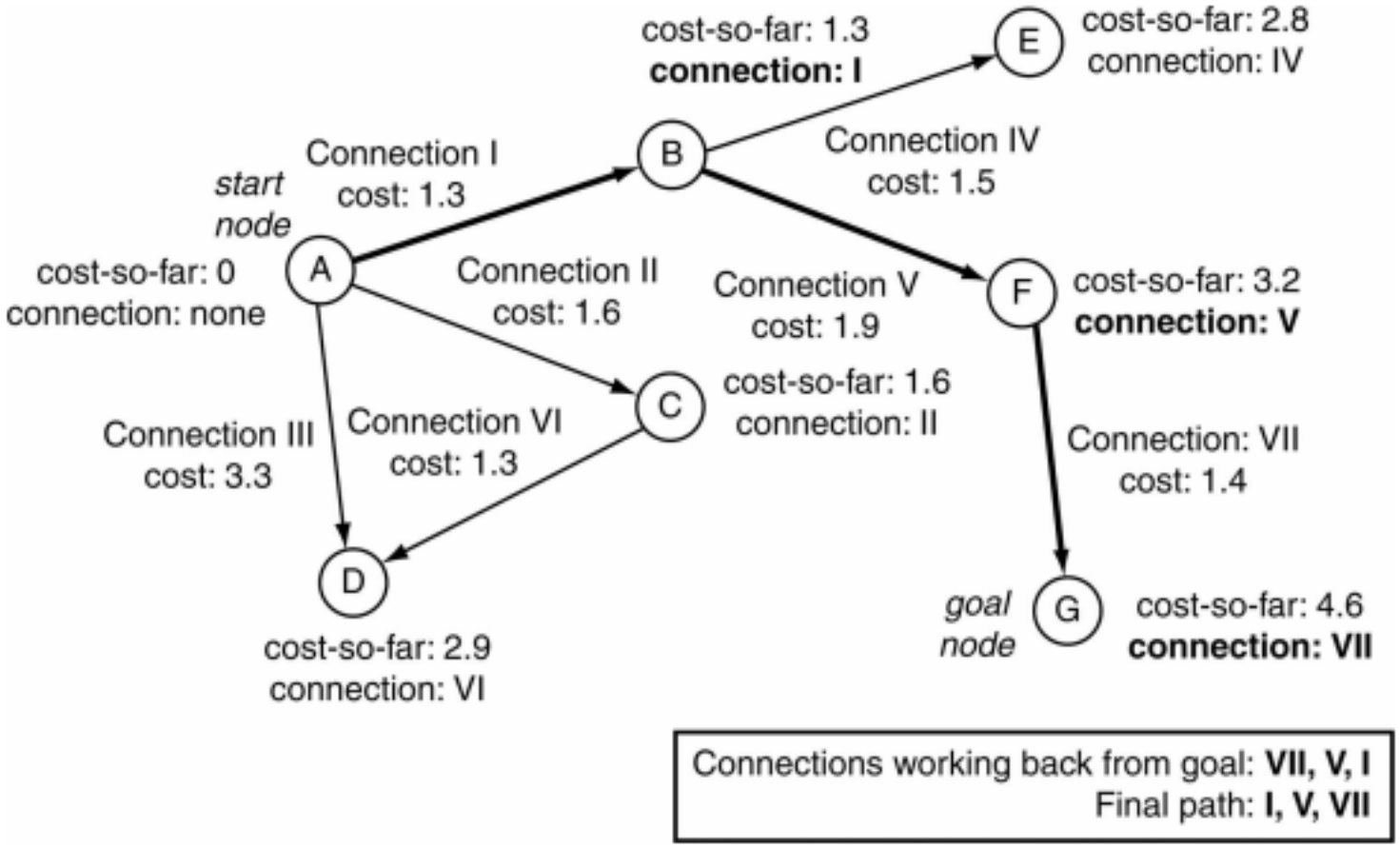
Example


cost-so-far: 1.3
connection: I
E
cost-so-far: 2.8
connection: IV
current
node
B
Connection I
cost: 1.3
Connection IV
cost: 1.5
start
node
A
cost-so-far: 0
connection: none
Connection II
cost: 1.6
Connection V
cost: 1.9
F
cost-so-far: 3.2
connection: V
C
cost-so-far: 1.6
connection: II
Connection III
cost: 3.3
D
cost-so-far: 3.3
connection: III

Example


cost-so-far: 1.3
connection: I
E
cost-so-far: 2.8
connection: IV
Connection I
cost: 1.3
B
Connection IV
cost: 1.5
start
node
cost-so-far: 0
connection: none
A
Connection II
cost: 1.6
Connection V
cost: 1.9
F
cost-so-far: 3.2
connection: V
cost-so-far: 1.6
connection: II
C
Connection III
cost: 3.3
Connection VI
cost: 1.3
current
node
D
cost-so-far: 3.3
connection: III
Is updated
cost-so-far: 2.9
connection: VI
Open list
E, F, D
Closed list
A, B, C

Example



Performance

- Graph has n nodes
- A node on average has m connections
- Algorithm executes in O(nm) time
- Storage is O(nm) elements in the lists

- For a game, we expect m<<n…
- So typical $O(n^2)$ time cited for Djikstra would be misleading

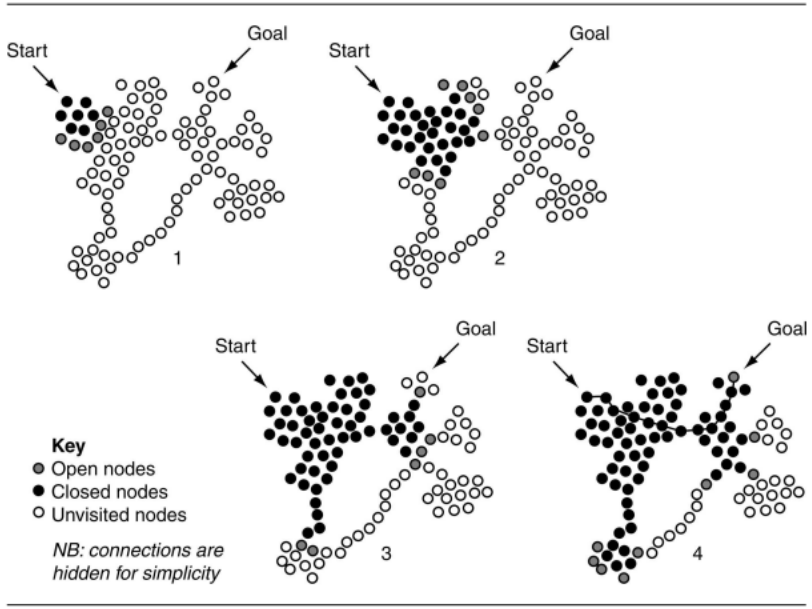
- Could do O(m + n log n) time using Fibonacci heap….

Weaknesses

Dijkstra searches entire graph indiscriminately for the shortest possible route.

Fine if we are trying to find the shortest path to every possible node

Wasteful for point-to-point pathfinding



The number of nodes explored, but not in final route, is called the fill

Want to explore as few nodes as possible, because each takes time to process.

Typically Dijkstra suffers from big fill…hence a preference for A*

ILLINOIS

Djikstra Unplugged

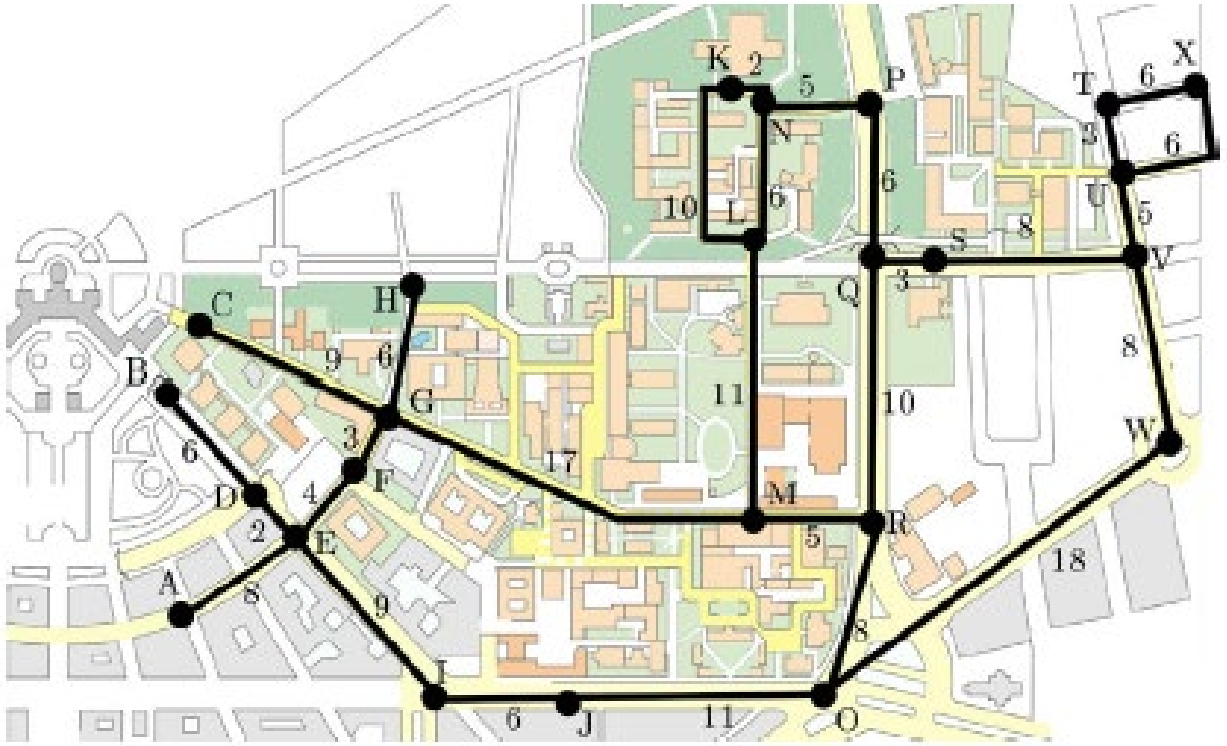
Can run Djikstra without a computer.

Pin a map to a table.

Run string along streets.

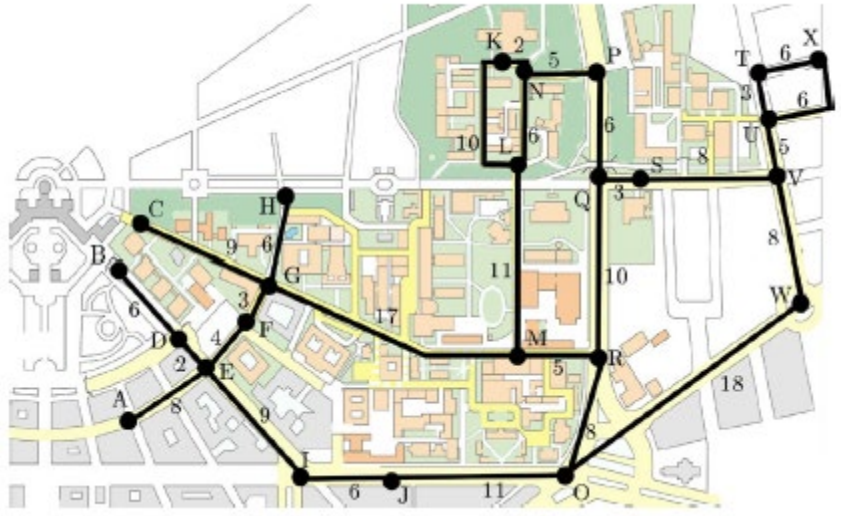
Put labeled knots at:

- start point
- end point
- dead ends
- intersections



ILLINOIS

Djikstra Unplugged

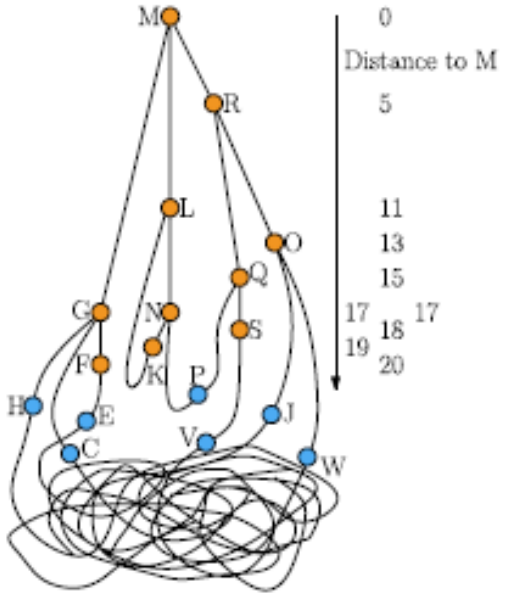


Lift up string from starting knot

Trace path the end knot following straight threads

Dijkstra's Algorithm in Thread Terminology

```
1  all knots are waiting, all d[v] are infinite, only d[starting knot] = 0
2  while there are waiting knots do
3     v := the waiting knot with the smallest d[v]
4     Turn v hanging
5     for all threads from v to a neighbor u of the length ℓ do
6        if d[v] + ℓ < d[u], then d[u] := d[v] + ℓ
            //   found shorter path to u, leads via v
```



Each iteration of the while-loop corresponds to the transition of knot *v* from waiting to hanging. The next knot lifted in turn is the waiting knot *v* of the smallest value *d*[*v*]. This value is the height to which we have to lift the starting knot to make *v* hanging. Since other threads lifted to this height later on cannot decrease it, *d*[*v*] is the definitive distance from the starting knot to *v*.

ILLINOIS