CS4386 AI Game Programming

Lecture 07
Pathfinding: Dijkstra and A*

Semester B, 2020-2021
Department of Computer Science
City University of Hong Kong

Game Character Movement

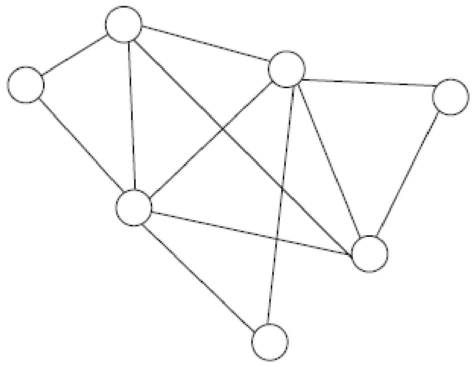
- Game characters usually need to move around their level
- Sometimes this movement is set in stone by the developers, such as
  1. a patrol route that a guard can follow blindly
     – Fixed routes are simple to implement, but can easily be fooled if an object is pushed in the way
  2. a small fenced region in which a dog can randomly wander around
     – Free wandering characters can appear aimless and can easily get stuck

Motivation for Pathfinding

- More complex characters do not know in advance where they will need to move, e.g.
  - A unit in a real-time strategy game may be ordered to any point on the map by the player at any time
  - a patrolling guard in a stealth game may need to move to its nearest alarm point to call for reinforcements
  - a platform game may require opponents to chase the player across a chasm using available platforms
- For each of these characters the AI must be able to calculate a suitable route through the game level to get from where it is now to its goal
  - the route should be sensible and as short or rapid as possible

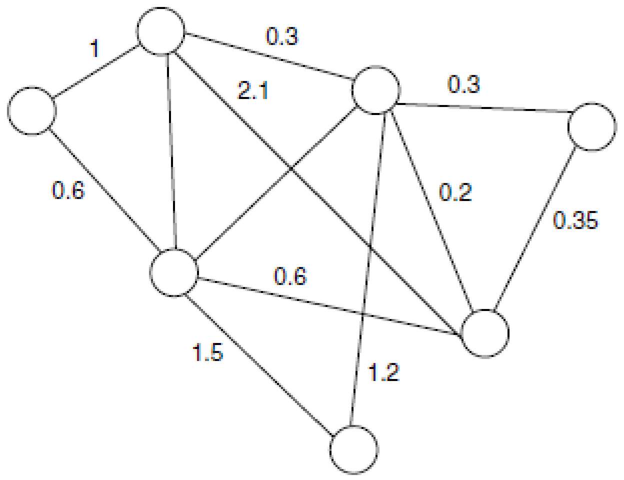
Graph

- Often a level needs to be simplified and represented in the form of a graph for pathfinding algorithms to work
- A graph is a mathematical structure often represented with a diagram
- A graph consists of two different types of element
  1. nodes (vertices) are often drawn as points or circles in a graph diagram
  2. connections (edges) link nodes together with lines
- For pathfinding
  – Each node usually represents a region of the game level, such as a room, a section of corridor, a platform, or a small region of outdoor space
  – Connections show which locations are connected
  – In this way the whole game level is split into regions, which are connected together



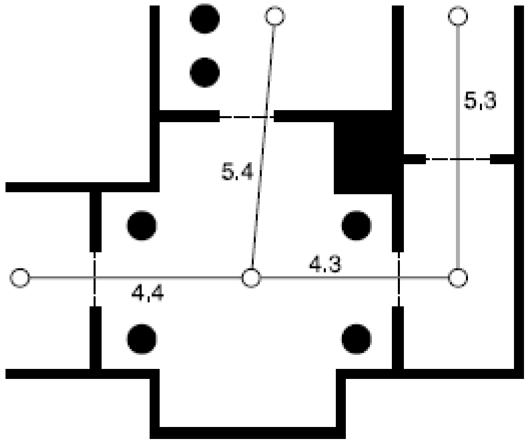
Weighted Graphs

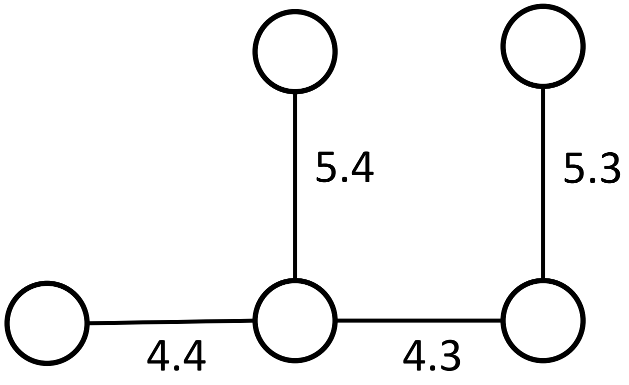
- A weighted graph is made up of nodes and connections, just like the general graph, and a numerical value is added to a pair of nodes for each connection
- In mathematical graph theory this is called the weight, and in game applications it is more commonly called the cost
- The costs in a pathfinding graph often represent time or distance, or a combination of time, distance, and other factors
- For a whole route through a graph, from a start node to a target node, we can work out the total path cost which is simply the sum of the costs of each connection in the route



Representative Points in a Region

- We tend to measure connection distances or times from a representative point in each region
- So we pick the center of the room and the center of the corridor
  - If the room is large and the corridor is long, then there is likely to be a large distance between their center points, so the cost will be large



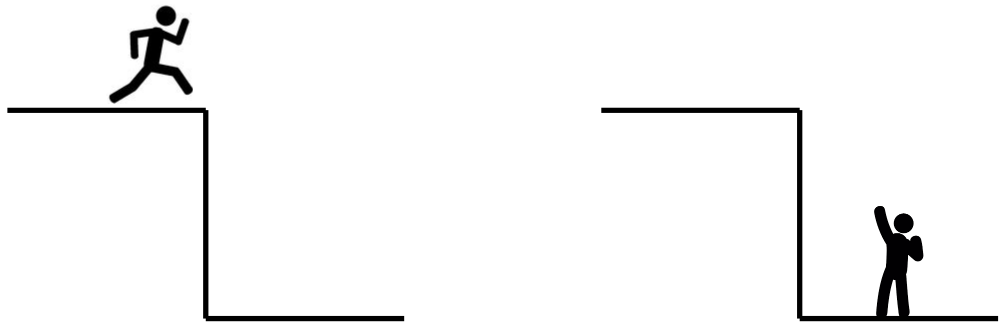


Non-Negative Constraint

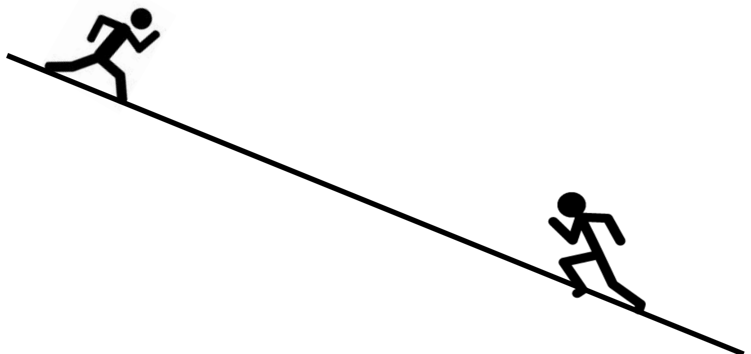
- Although Mathematical graph theory does allow negative weights and they have direct applications in some practical problems, these problems are entirely outside of normal game development
- Here we will only work with non-negative weights, i.e., the costs are always positive (or zeros)
  - The distance between 2 points cannot be negative
  - The amount of time taken to get from one point to another cannot be negative

Directed Weighted Graphs

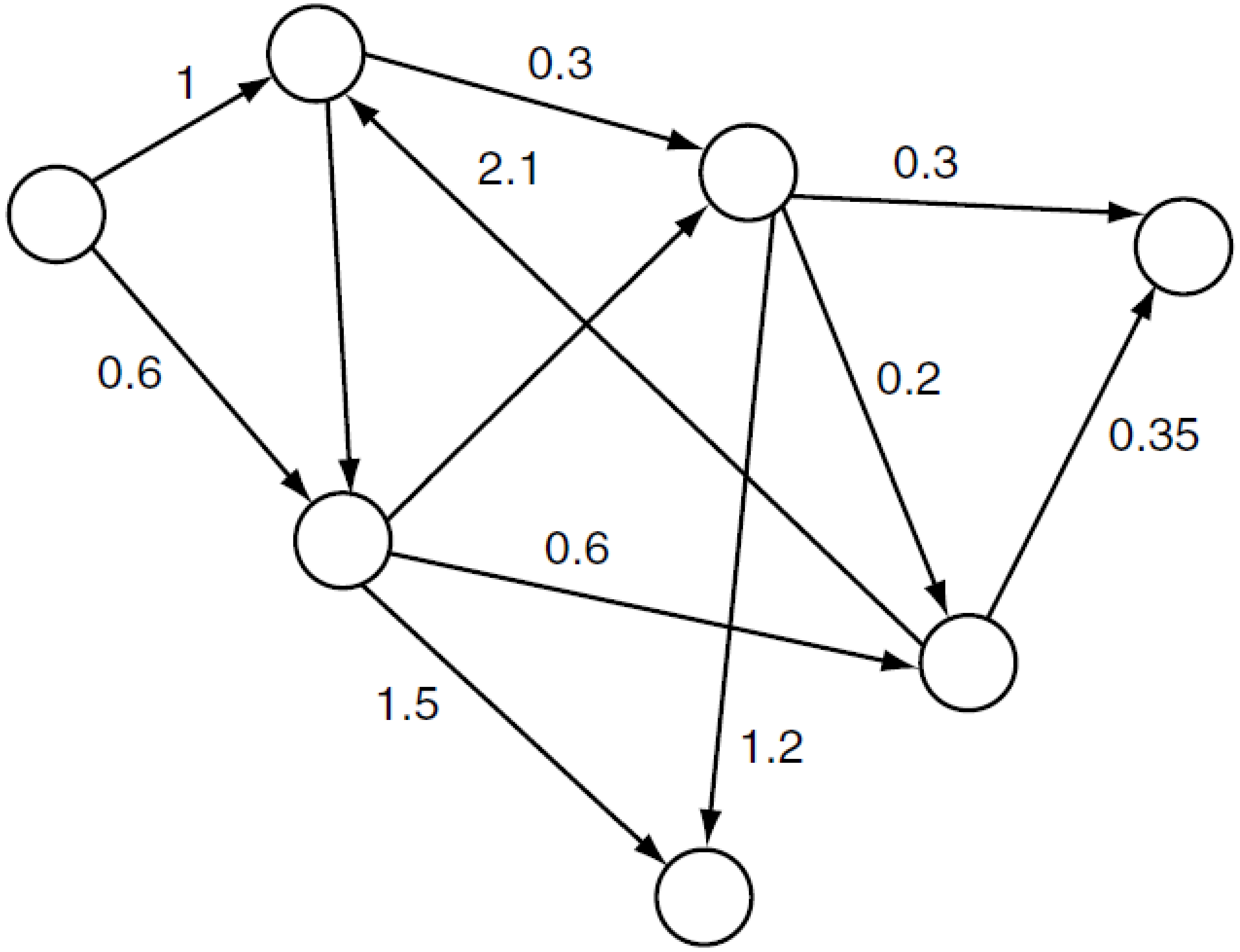
- The major pathfinding algorithms support the use of a more complex form of graph, the directed graph which assumes that the connections are in one direction only
  - the ability to move from A to B implies that B is reachable from A



  - 2 connections are required if it is possible to get from A to B and also from B to A, and they may have different costs



Example Directed Weighted Graph

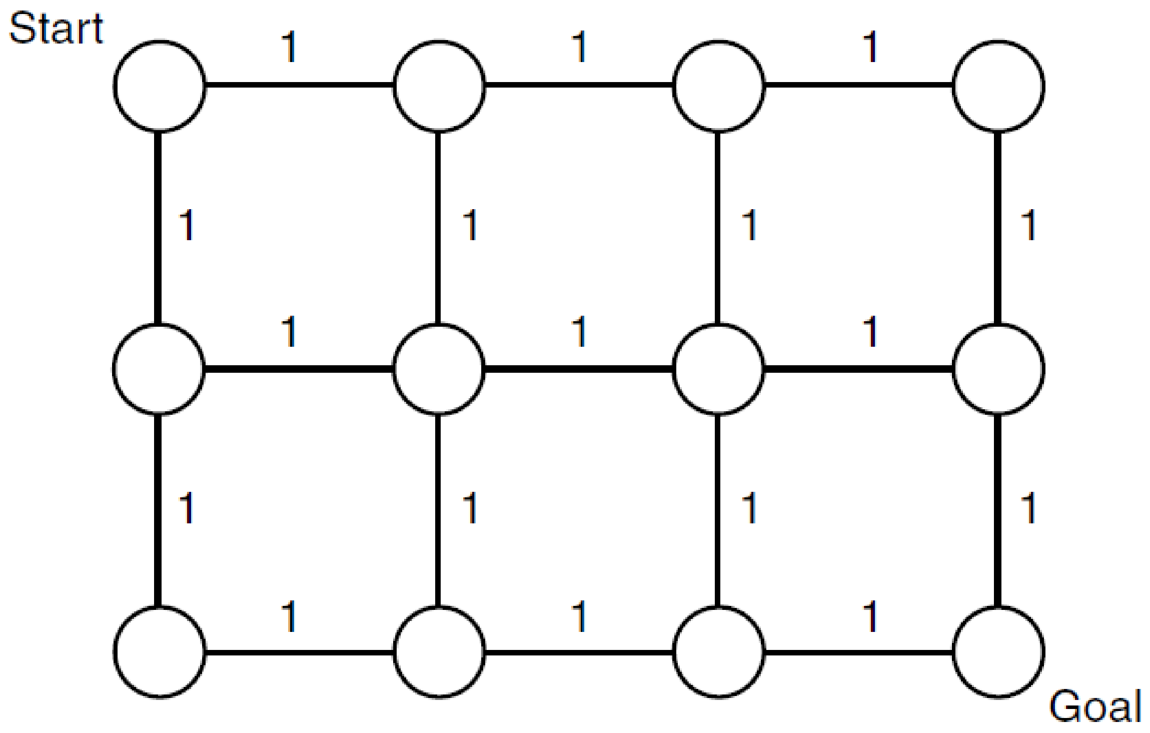


Dijkstra

- The Dijkstra algorithm is named for Edsger Dijkstra, the mathematician who devised it (and the same man who coined the famous programming phrase "GOTO considered harmful")
- Dijkstra's algorithm was originally designed to solve a problem in mathematical graph theory, confusingly called "shortest path"
- Where pathfinding in games has one start point and one goal point, the shortest path algorithm is designed to find the shortest routes to everywhere from an initial point
  - The solution to this problem will include a solution to the pathfinding problem

Problem

- Given a graph (a directed non-negative weighted graph) and two nodes (called start and goal) in that graph, we would like to generate a path such that the total path cost of that path is minimal among all possible paths from start to goal



Problem (cont.)

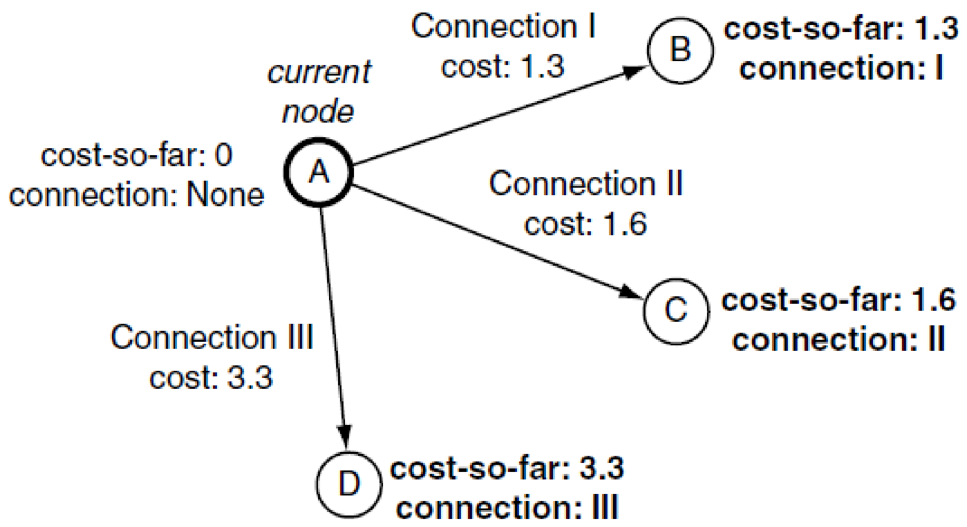
- The path we expect to be returned consists of a set of connections, not nodes
- Two nodes may be linked by more than one connection, and each connection may have a different cost (it may be possible to either fall off a walkway or climb down a ladder, for example)
- We therefore need to know which connections to use; a list of nodes will not suffice
- Many games do not make this distinction
  - There is, at most, one connection between any pair of nodes. After all, if there are two connections between a pair of nodes, the pathfinder should always take the one with the lower cost
  - In some applications, however, the costs change over the course of the game or between different characters, and keeping track of multiple connections is useful

Overall Algorithm

- Dijkstra works by spreading out from the start node along its connections
- As it spreads out to more distant nodes, it keeps a record of the direction it came from (imagine it drawing chalk arrows on the floor to indicate the way back to the start)
- Eventually, it will reach the goal node and can follow the arrows back to its start point to generate the complete route
- Because of the way Dijkstra regulates the spreading process, it guarantees that the chalk arrows always point back along the shortest route to the start

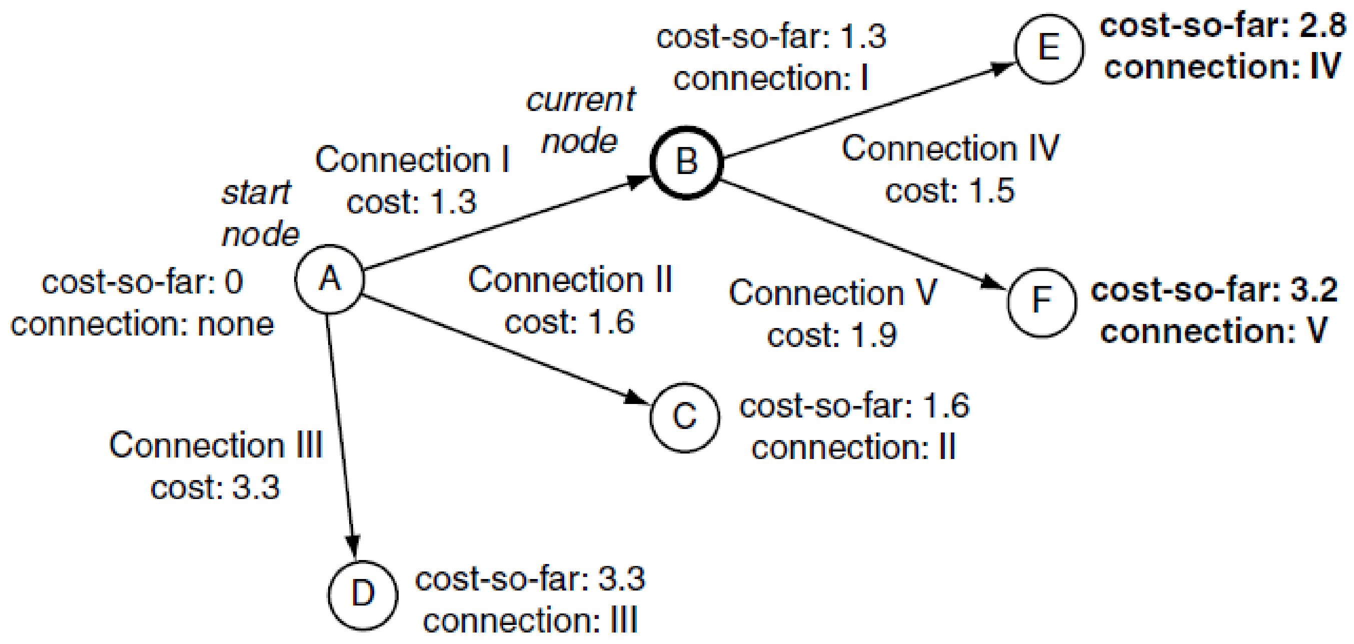
Processing the Current Node (1)

- During an iteration, Dijkstra considers each outgoing connection from the current node. For each connection it finds the end node and stores
  1. the total cost of the path so far
  2. the connection it arrived there from
- In the first iteration, where the start node is the current node, the total cost-so-far for each connection's end node is simply the cost of the connection



Processing the Current Node (2)

- For iterations after the first, the cost-so-far for the end node of each connection is the sum of the connection cost and the cost-so-far of the current node (i.e., the node from which the connection originated)



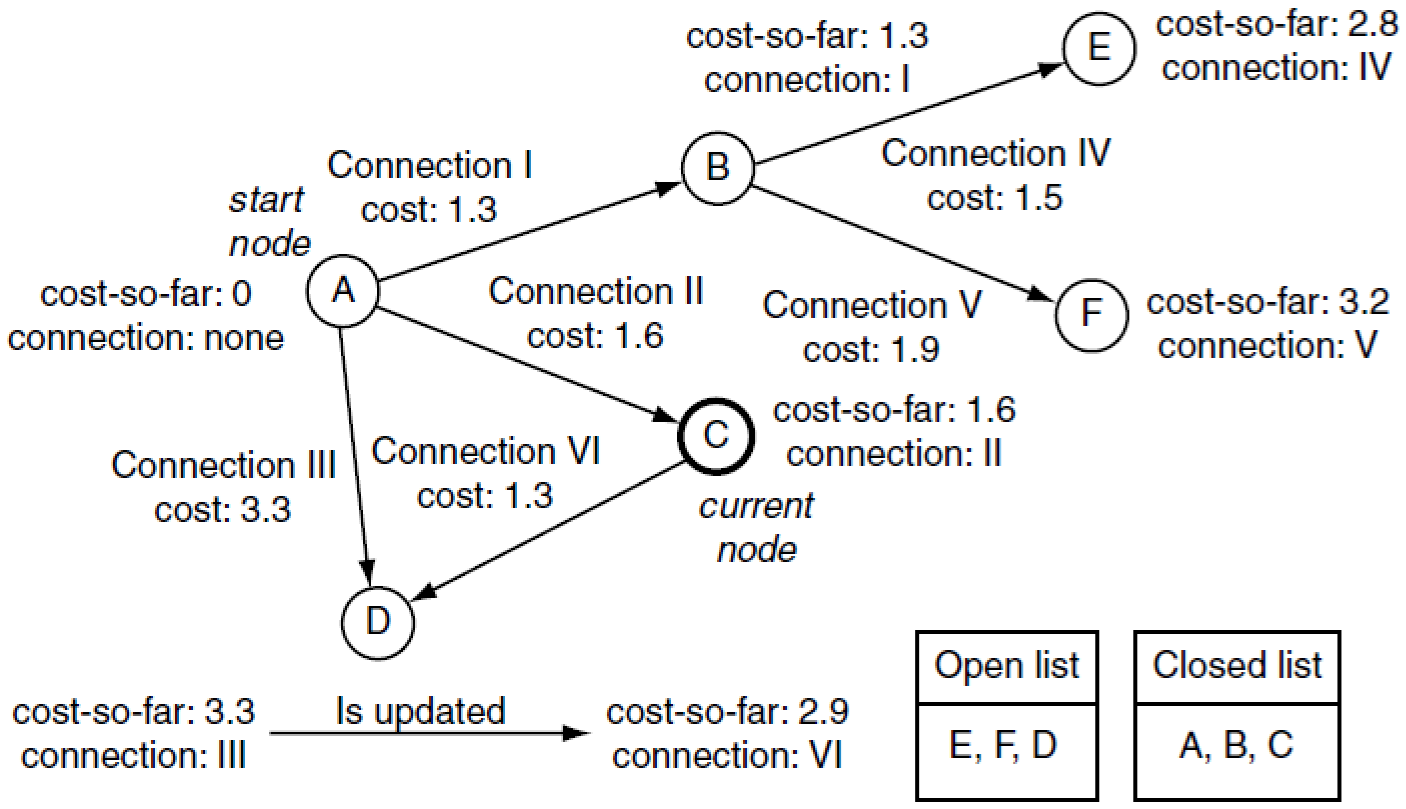
Node Lists

- Each node can be thought of as being in one of three categories:
  1. Closed list
  - Nodes that have been processed in their own iterations
  2. Open list
  - Nodes that have been visited from another node, but not yet processed in their own iterations
  3. Unvisited list
  - Nodes that are neither in the closed list nor in the open list
- To start with, the open list contains only the start node (with zero cost-so-far), and the closed list is empty
- At each iteration, the algorithm chooses the node from the open list that has the smallest cost-so-far. This is then processed in the normal way. The processed node is then removed from the open list and placed on the closed list

Calculating Cost-So-Far for Open and Closed Nodes

- If we arrive at an open or closed node during an iteration, then the node will already have a cost-so-far value and a record of the connection that led there
- We need to check if the route we have now found is better than the route that we have already found
- The cost-so-far value is calculated as normal
    1. if it is higher than the node's current cost-so-far, then do not update the node at all and do not change what list it is on
    2. If it 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

Example: Open Node Update



Open list
E, F, D

Closed list
A, B, C

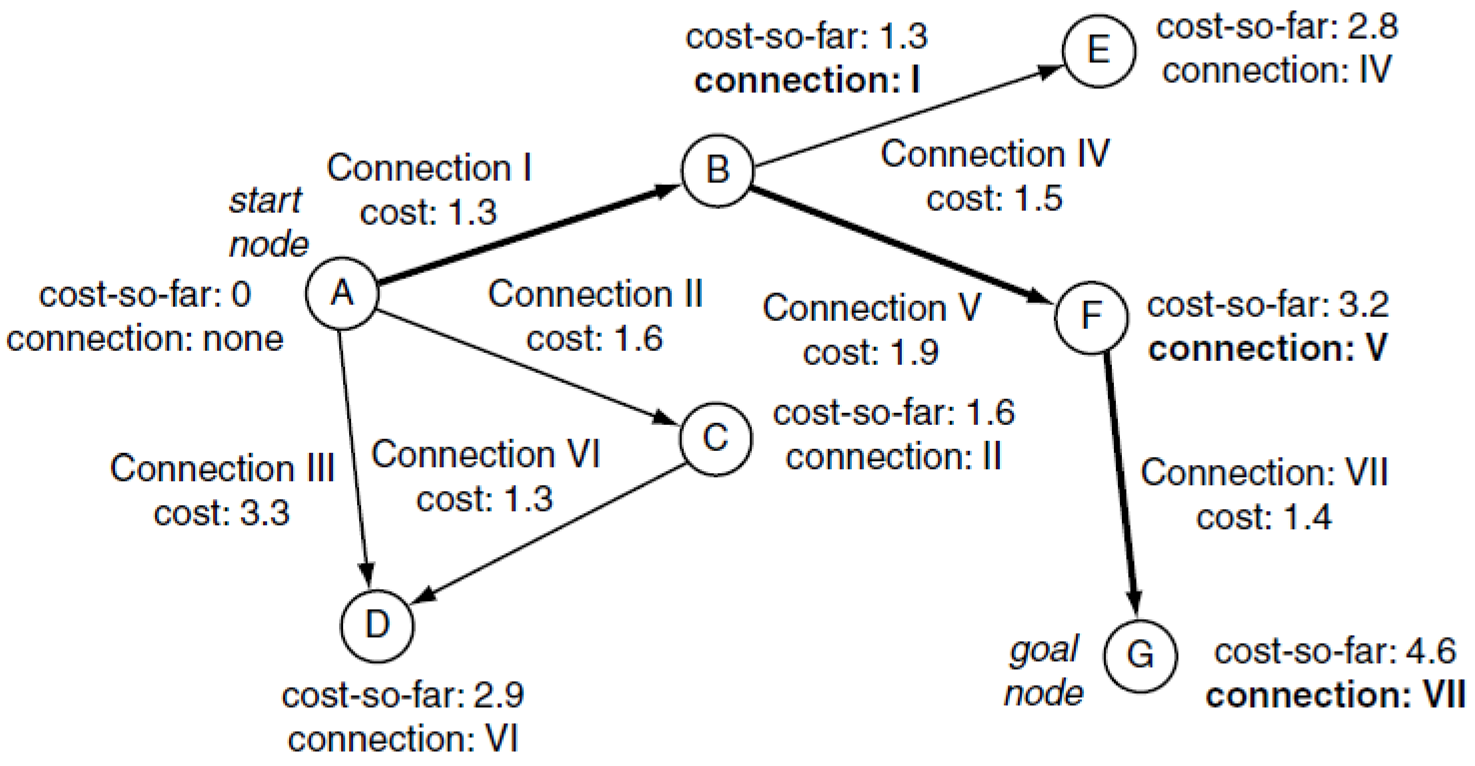
Terminating the Algorithm

- The basic Dijkstra algorithm terminates when the open list is empty: it has considered every node in the graph that can be reached from the start node, and they are all on the closed list
- For pathfinding, we are only interested in reaching the goal node, however, so we can stop earlier. The algorithm should terminate when the goal node is the smallest node on the open list

Retrieving the Path

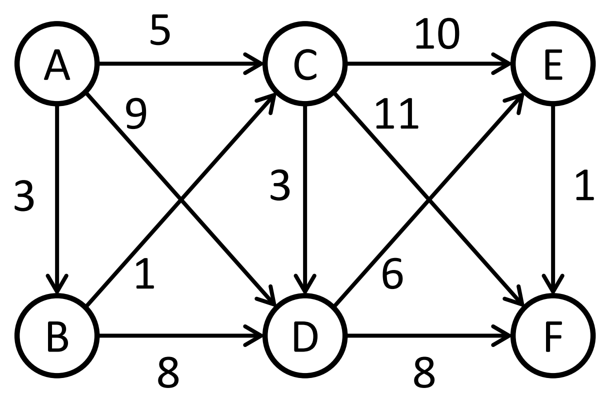
- The final stage is to retrieve the path
- Starting at the goal node, the connection that was used to arrive there was used to go back and the start node of that connection was examined to do the same
- This process is continued by back-tracking the connections, until the original start node is reached
- The list of connections is then reversed so that it is in correct order and the list is returned as the solution

Example: Retrieving the Path



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

Example 1: Dijkstra (1)



- Problem: Apply Dijkstra algorithm to find the optimal path from node A to node F
- Notations:

  N: Current Node

  C: Closed List

  O: Open List

  U: Unvisited List

  $\boxed{i}$: Iteration *i*

  ϕ: Nothing

  $X(Y, c)$: Connection to Node X is from Node Y with its cost-so-far c (assuming that there is at most one connection between any pair of nodes)

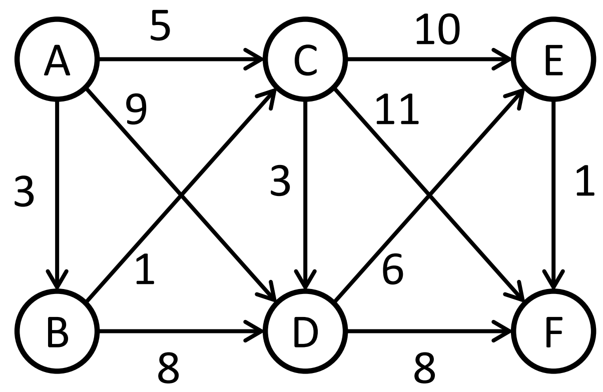
- Initial settings:

  C: ϕ

  O: A(ϕ,0)

  U: B, C, D, E, F

Example 1: Dijkstra (2)



Notations:
N: Current Node
C: Closed List
O: Open List
U: Unvisited List
$\boxed{i}$: Iteration *i*

$\boxed{1}$ **N**: A
C: A(ϕ,0)
O: B(A,3) C(A,5) D(A,9)
U: E, F

$\boxed{2}$ **N**: B
C: A(ϕ,0) B(A,3)
O: C(B,4) D(A,9)
U: E, F

$\boxed{3}$ **N**: C
C: A(ϕ,0) B(A,3) C(B,4)
O: D(C,7) E(C,14) F(C,15)
U: ϕ

$\boxed{4}$ **N**: D
C: A(ϕ,0) B(A,3) C(B,4) D(C,7)
O: E(D,13) F(C,15)
U: ϕ

$\boxed{5}$ **N**: E
C: A(ϕ,0) B(A,3) C(B,4) D(C,7) E(D,13)
O: F(E,14)
U: ϕ

Final Path:
F ← E ← D ← C ← B ← A

Performance

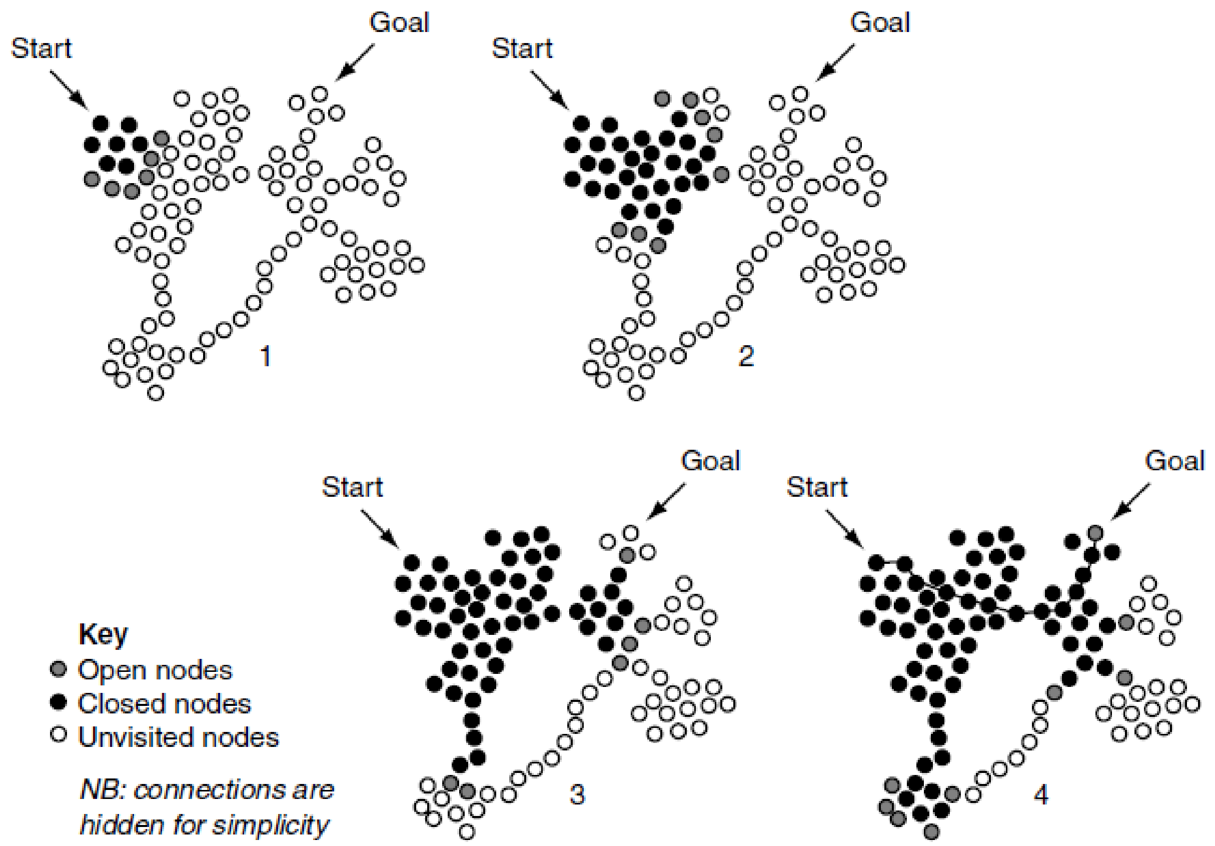
- Performance
  - $O(nm)$ in time

    where n is the number of nodes in the graph that is closer than the goal node, m is the average number of outgoing connections

    per node
  - The worst conceivable performance occurs when the graph is so densely connected that m = n, and the performance would be $O(n^2)$
- Weakness
  - it searches the entire graph indiscriminately for the shortest possible route. This is useful if we're trying to find the shortest path to every possible node (the problem that Dijkstra was designed for), but wasteful for point-to-point pathfinding

Illustration: Dijkstra in Steps

- The number of nodes that were considered, but never made part of the final route, is called the fill of the algorithm



A*

- Unlike the Dijkstra algorithm, A* is designed for point-to-point pathfinding and is not used to solve the shortest path problem in graph theory
- The problem is identical to that solved by the Dijkstra pathfinding algorithm:
  - Given a graph (a directed non-negative weighted graph) and two nodes in that graph (start and goal), we would like to generate a path such that the total path cost of that path is minimal among all possible paths from start to goal. The path should consist of a list of connections from the start node to the goal node

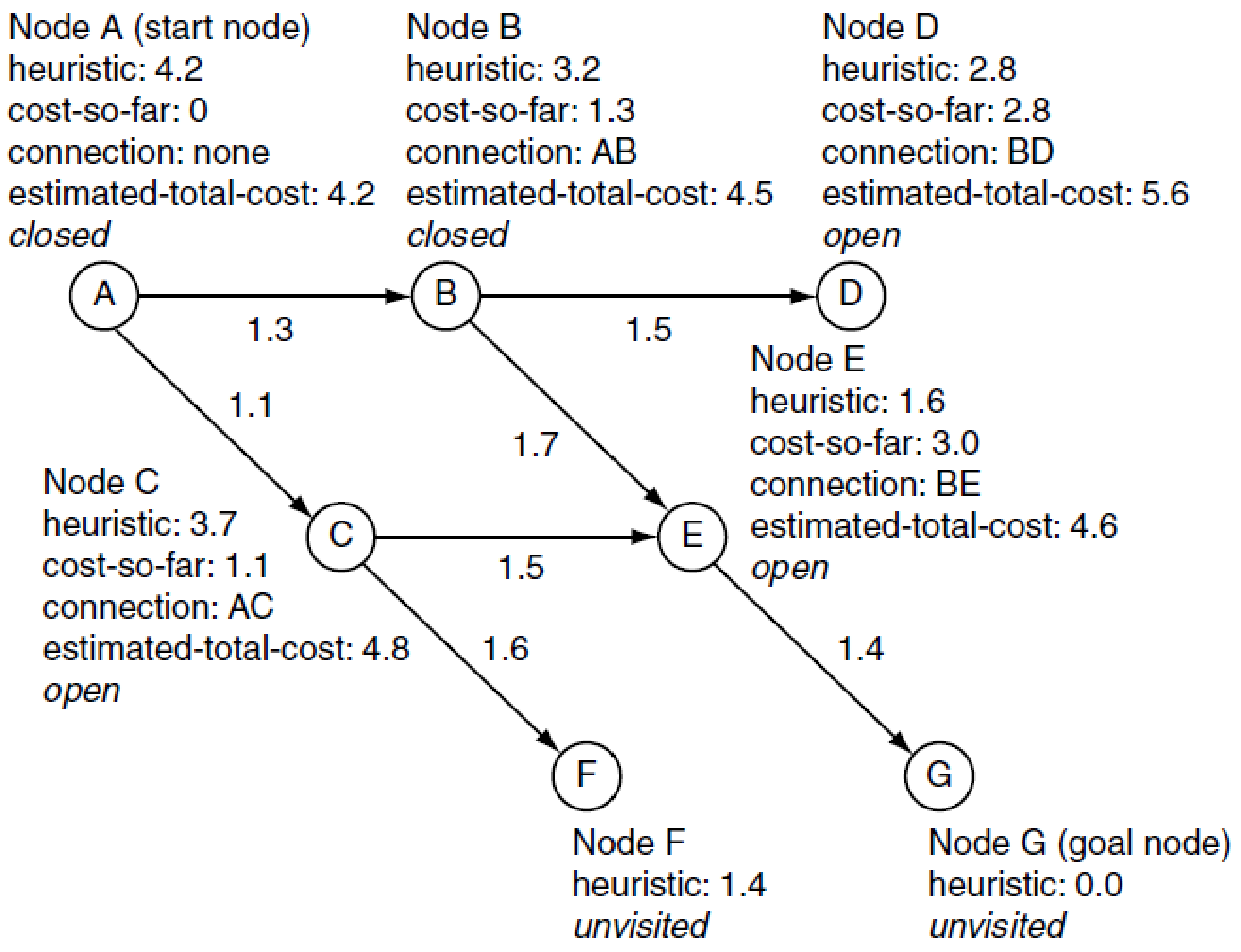
Overall Algorithm

- The algorithm works in much the same way as Dijkstra does
- Rather than always considering the open node with the lowest cost-so-far value, we choose the node that is most likely to lead to the shortest overall path. The notion of "most likely" is controlled by a heuristic
- If the heuristic is accurate, then the algorithm will be efficient. If the heuristic is terrible, then it can perform even worse than Dijkstra

Processing the Current Node

- During an iteration, A* considers each outgoing connection from the current node
- For each connection it finds the end node and stores:
  1. the total cost of the path so far
  2. the connection it arrived there from
  3. the estimate of the total cost for a path from the start node through this node and onto the goal
     - This estimate is the sum of two values: the cost-so-far and how far it is from the node to the goal
     - This estimate is generated by a separate piece of code and isn't part of the algorithm
     - These estimates are called the "heuristic value" of the node, and it cannot be negative

Illustration: A* Node Stored Values



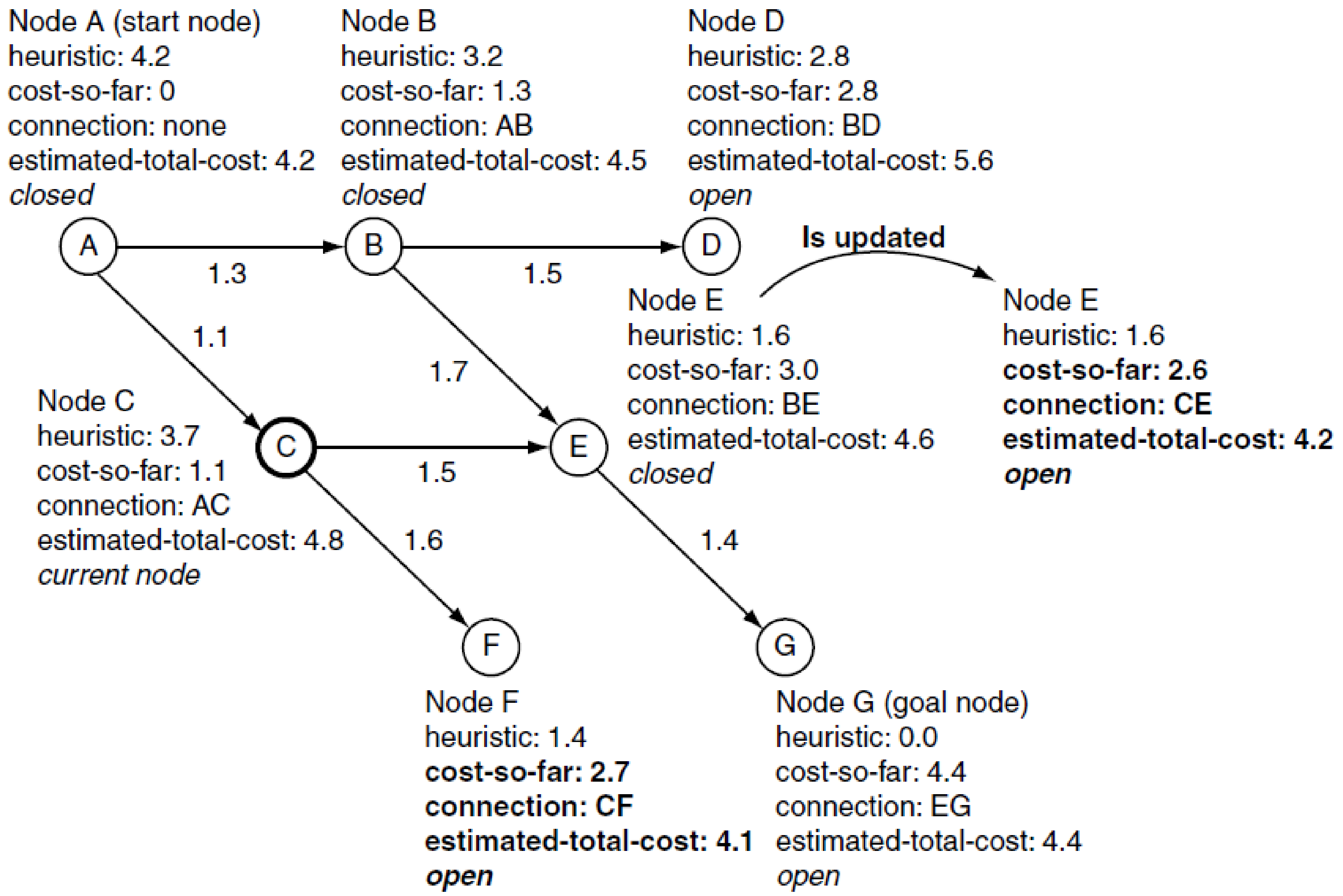
Node List

- Similar to Dijkstra
  - The algorithm keeps a list of open nodes that have been visited but not processed and closed nodes that have been processed
  - Nodes are moved onto the open list as they are found at the end of connections
  - Nodes are moved onto the closed list as they are processed in their own iteration
- Unlike Dijkstra
  - The node from the open list with the smallest **estimated-total-cost** (not cost-so-far) is selected at each iteration
    - This alteration allows the algorithm to examine nodes that are more promising first
    - If a node has a small estimated-total-cost, then it must have a relatively short cost-so-far and a relatively small estimated distance to go to reach the goal
    - If the estimates are accurate, then the nodes that are closer to the goal are considered first, narrowing the search into the most profitable area

Calculating Cost-So-Far for Open and Closed Nodes

- Similar to Dijkstra, the cost-so-far value is calculated as normal, and if the new value is lower than the existing value for the node, then it needs to be updated
- Unlike Dijkstra, the A* algorithm can find better routes to nodes that are already on the closed list
  - If a previous estimate was very optimistic, then a node may have been processed thinking it was the best choice when, in fact, it was not. We remove the node from the closed list and place it back on the open list
- Closed nodes that have their values revised are removed from the closed list and placed on the open list
- Open nodes that have their values revised stay on the open list, as before

Example: Closed Node Update



Terminating the Algorithm

- Dijkstra terminates when the goal node is the smallest node on the open list
- With A*
  - A node that has the smallest estimated-total-cost value (and will therefore be processed next iteration and put on the closed list) may later need its values revised
  - We can no longer guarantee, just because the node is the smallest on the open list, that we have the shortest route there
  - So terminating A* when the goal node is the smallest on the open list will not guarantee that the shortest route has been found
  - To generate a guaranteed optimal result, the algorithm only terminates when the node in the open list with the smallest cost-so-far (not estimated-total-cost) has a cost-so-far value greater than the cost of the path we found to the goal. Then can we guarantee that no future path will be found that forms a shortcut

Performance

- Performance
  - O(lm) in time

    where l is the number of nodes whose total estimated-path-cost is less than that of the goal, m is the average number of outgoing connections per node
  - In addition, the heuristic function often requires some processing and can dominate the execution load of the algorithm. It is rare, however, for its implementation to directly depend on the size of the pathfinding problem although it may be time-consuming

Choosing a Heuristic

- The more accurate the heuristic, the less fill A* will experience, and the faster it will run
- If you can get a perfect heuristic (one that always returns the exact minimum path distance between two nodes), A* will go straight to the correct answer: the algorithm becomes O(p), where p is the number of steps in the path
- However, to work out the exact distance between two nodes, you typically have to find the shortest route between them. This would mean solving the pathfinding problem—which is what we are trying to do in the first place!
- For non-perfect heuristics, A* behaves slightly differently depending on whether the heuristic is too low or too high

Underestimating Heuristic

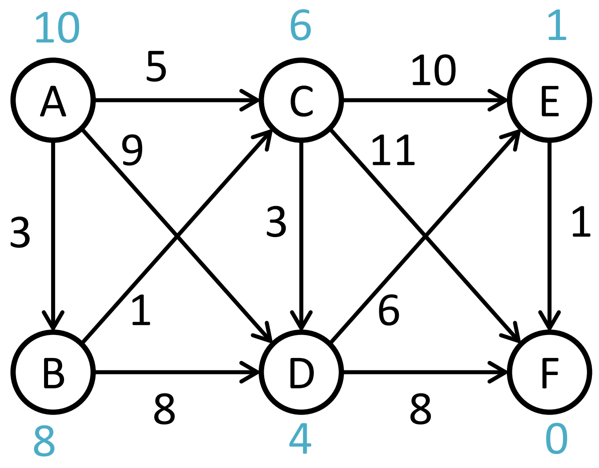
- If the heuristic is too low, so that it underestimates the actual path length, A* takes longer to run
- The estimated-total-cost will be biased toward the cost-so-far (because the heuristic value is smaller than reality)
  - So A* will prefer to examine nodes closer to the start node, rather than those closer to the goal
  - This will increase the time it takes to find the route through to the goal
- If the heuristic underestimates in all possible cases, then the result that A* produces will be the best path possible
  - It will be the exact same path that the Dijkstra algorithm would generate

Overestimating Heuristic

- If the heuristic is too high, so that it overestimates the actual path length, A* may not return the best path
  - A* will tend to generate a path with fewer nodes in it, even if the connections between nodes are more costly
  - The estimated-total-cost value will be biased toward the heuristic. The A* algorithm will pay proportionally less attention to the cost-so-far and will tend to favor nodes that have less distance to go
  - This will move the focus of the search toward the goal faster, but with the prospect of missing the best routes to get there
  - This means that the total length of the path may be greater than that of the best path
  - It can be shown that if the heuristic overestimates by at most x (i.e., x is the greatest overestimate for any node in the graph), then the final path will be no more than x too long

v: estimated cost from a node to the goal node

Example 2: A* (1)



Underestimating Heuristics

- Initial settings:

  C: ϕ

  O: A(ϕ,0,10)

  U: B, C, D, E, F

- Problem: Apply A* algorithm to find the optimal path from node A to node F
- Notations:

  N: Current Node

  C: Closed List

  O: Open List

  U: Unvisited List (will be omitted)

  $\boxed{i}$: Iteration *i*

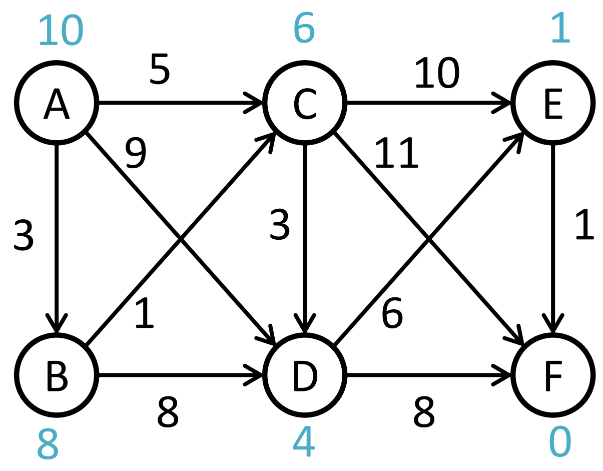
  ϕ: Nothing

  $X(Y, c, e)$: Connection to Node X is from Node Y with its cost-so-far c and estimated-total-cost e (assuming that there is at most one connection between any pair of nodes)

v: estimated cost from a node to the goal node

Example 2: A* (2)

Notations:
N: Current Node
C: Closed List
O: Open List
$\boxed{i}$: Iteration *i*



Underestimating Heuristics

$\boxed{1}$ **N**: A
C: A(ϕ,0,10)
O: B(A,3,11) C(A,5,11) D(A,9,13)

$\boxed{2}$ **N**: B
C: A(ϕ,0,10) B(A,3,11)
O: C(B,4,10) D(A,9,13)

$\boxed{3}$ **N**: C
C: A(ϕ,0,10) B(A,3,11) C(B,4,10)
O: D(C,7,11) E(C,14,15) F(C,15,15)

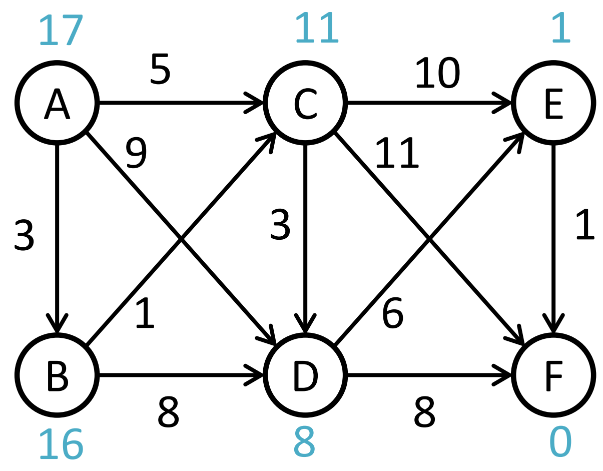
$\boxed{4}$ **N**: D
C: A(ϕ,0,10) B(A,3,11) C(B,4,10) D(C,7,11)
O: E(D,13,14) F(C,15,15)

$\boxed{5}$ **N**: E
C: A(ϕ,0,10) B(A,3,11) C(B,4,10) D(C,7,11) E(D,13,14)
O: F(E,14,14)

Final Path:
F ← E ← D ← C ← B ← A

v: estimated cost from a node to the goal node

Example 3: A* (1)



Overestimating Heuristics

- Initial settings:

  C: ϕ

  O: A(ϕ,0,17)

  U: B, C, D, E, F

- Problem: Apply A* algorithm to find the optimal path from node A to node F
- Notations:

  N: Current Node

  C: Closed List

  O: Open List

  U: Unvisited List (will be omitted)

  $\boxed{i}$: Iteration *i*

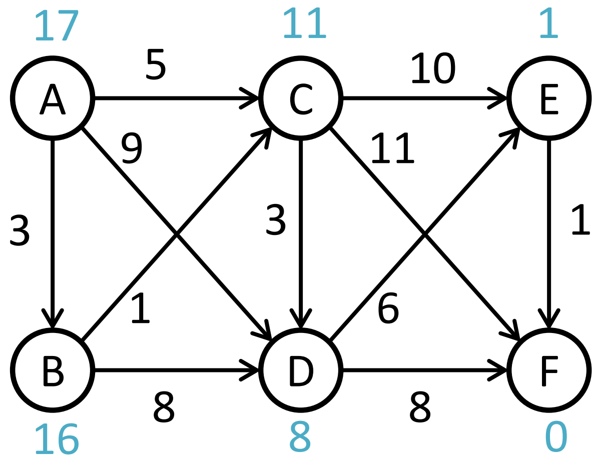
  ϕ: Nothing

  X(*Y*, *c*, *e*): Connection to Node *X* is from Node *Y* with its cost-so-far *c* and estimated-total-cost *e* (assuming that there is at most one connection between any pair of nodes)

v: estimated cost from a node to the goal node

Example 3: A* (2)

<u>Notations</u>:
N: Current Node
C: Closed List
O: Open List
$\boxed{i}$: Iteration *i*



Overestimating Heuristics

$\boxed{1}$ **N**: A
C: A(ϕ,0,17)
O: B(A,3,19) C(A,5,16) D(A,9,17)

$\boxed{2}$ **N**: C
C: A(ϕ,0,17) C(A,5,16)
O: B(A,3,19) D(C,8,16) E(C,15,16) F(C,16,16)

$\boxed{3}$ **N**: D
C: A(ϕ,0,17) C(A,5,16) D(C,8,16)
O: B(A,3,19) E(D,14,15) F(C,16,16)

$\boxed{4}$ **N**: E
C: A(ϕ,0,17) C(A,5,16) D(C,8,16) E(D,14,15)
O: B(A,3,19) F(E,15,15)

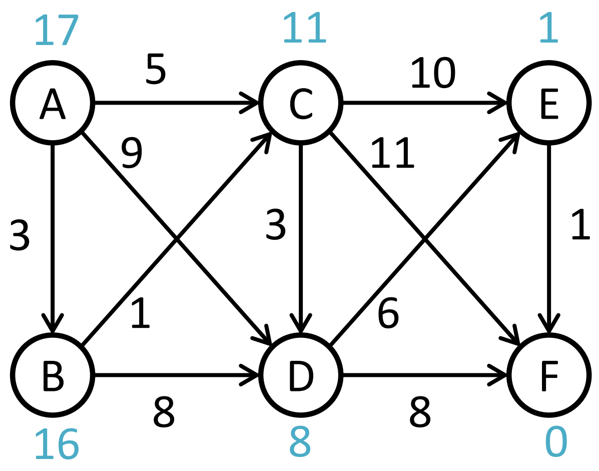
$\boxed{5}$ **N**: F
C: A(ϕ,0,17) C(A,5,16) D(C,8,16) E(D,14,15) F(E,15,15)
O: B(A,3,19)

$\boxed{6}$ **N**: B
C: A(ϕ,0,17) B(A,3,19) D(C,8,16) E(D,14,15) F(E,15,15)
O: C(B,4,15)

v: estimated cost from a node to the goal node

Example 3: A* (3)

Notations:
N: Current Node
C: Closed List
O: Open List
$\boxed{i}$: Iteration *i*



Overestimating Heuristics

$\boxed{6}$ **N**: B

C: A(ϕ,0,17) B(A,3,19) D(C,8,16) E(D,14,15) F(E,15,15)

O: C(B,4,15)

$\boxed{7}$ **N**: C

C: A(ϕ,0,17) B(A,3,19) C(B,4,15) E(D,14,15) F(E,15,15)

O: D(C,7,15)

$\boxed{8}$ **N**: D

C: A(ϕ,0,17) B(A,3,19) C(B,4,15) D(C,7,15) F(E,15,15)

O: E(D,13,14)

$\boxed{9}$ **N**: E

C: A(ϕ,0,17) B(A,3,19) C(B,4,15) D(C,7,15) E(D,13,14)

O: F(E,14,14)

Final Path:

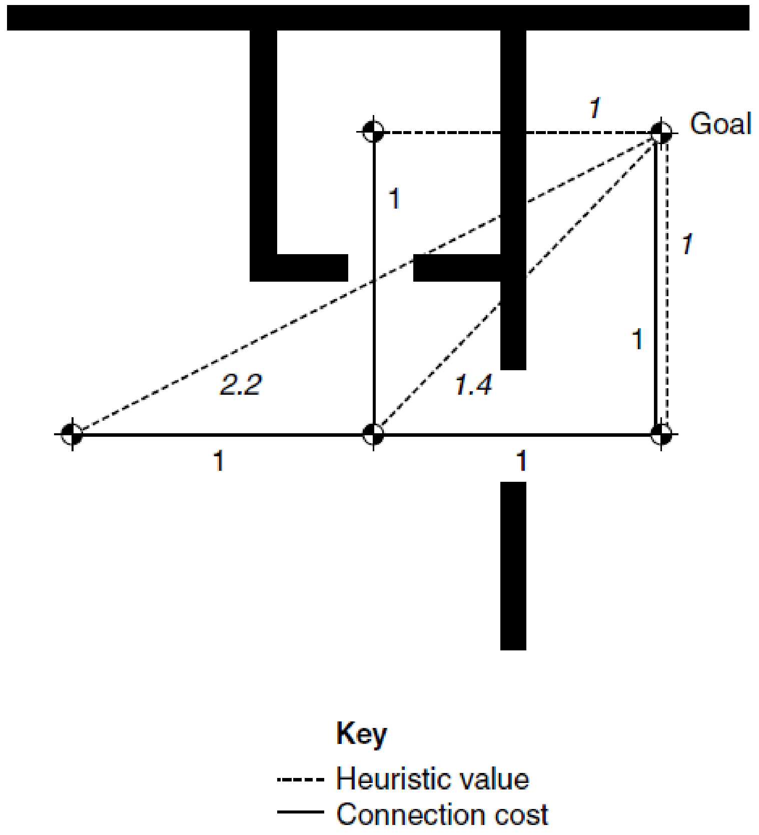
F ← E ← D ← C ← B ← A

Underestimating or Overestimating Heuristic?

- In applications where accuracy is more important than performance, it is important to ensure that the heuristic is underestimating
- In practice, try to resist dismissing overestimating heuristics outright
  - A game is not about optimum accuracy; it's about believability
- An overestimating heuristic is sometimes called an "inadmissible heuristic"
  - This does not mean you can't use it; it refers to the fact that the A* algorithm no longer returns the shortest path
- Overestimates can make A* faster if they are almost perfect, because they home in on the goal more quickly
  - If they are only slightly overestimating, they will tend to produce paths that are often identical to the best path, so the quality of results is not a major issue
  - But the margin for error is small. As a heuristic overestimates more, it rapidly makes A* perform worse
- Unless your heuristic is consistently close to perfect, it can be more efficient to underestimate, and you get the added advantage of getting the correct answer

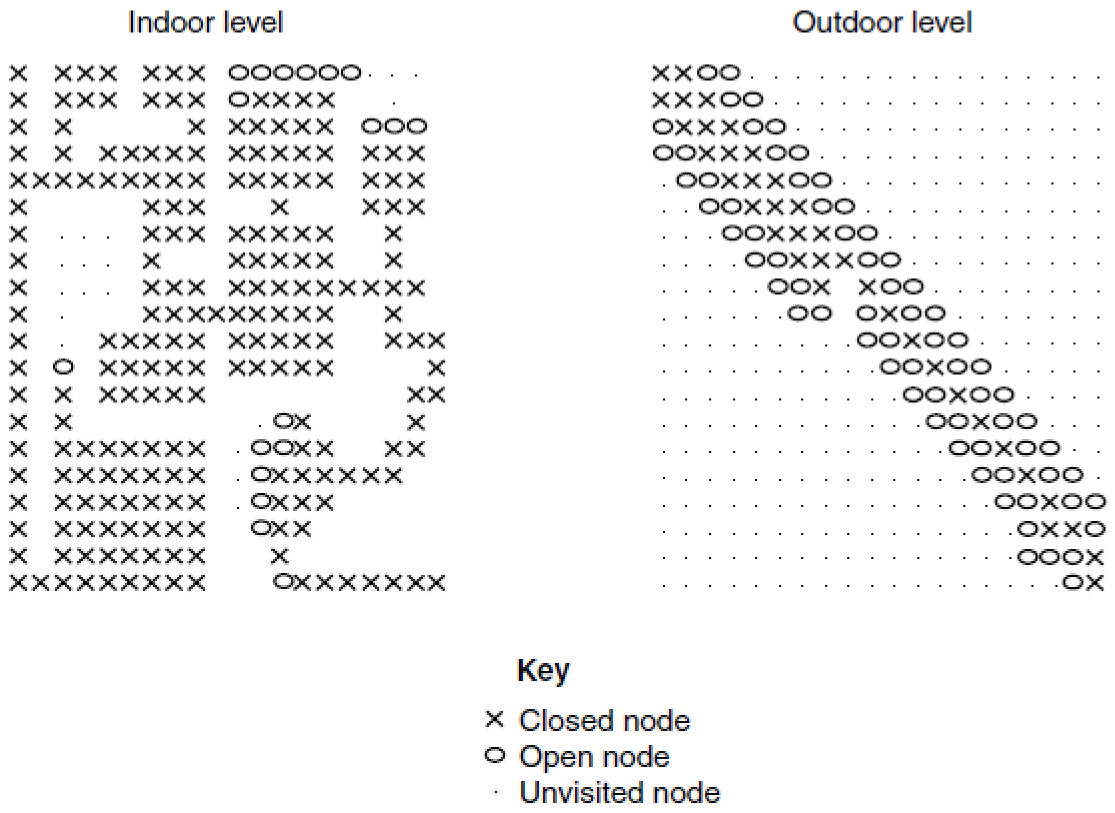
Euclidean Distance

- A common heuristic is Euclidean distance, which is guaranteed to be underestimating
- Euclidean distance is measured directly between two points in space, through walls and obstructions
- Euclidean distance is always either accurate or an underestimate
  - Traveling around walls or obstructions can only add extra distance
  - If there are no such obstructions, then the heuristic is accurate. Otherwise, it is an underestimate



Euclidean Distance: Indoor and Outdoor

- In outdoor settings, with few constraints on movement, Euclidean distance can be very accurate and provide fast pathfinding
- In indoor environments, it can be a dramatic underestimate, causing less than optimal pathfinding



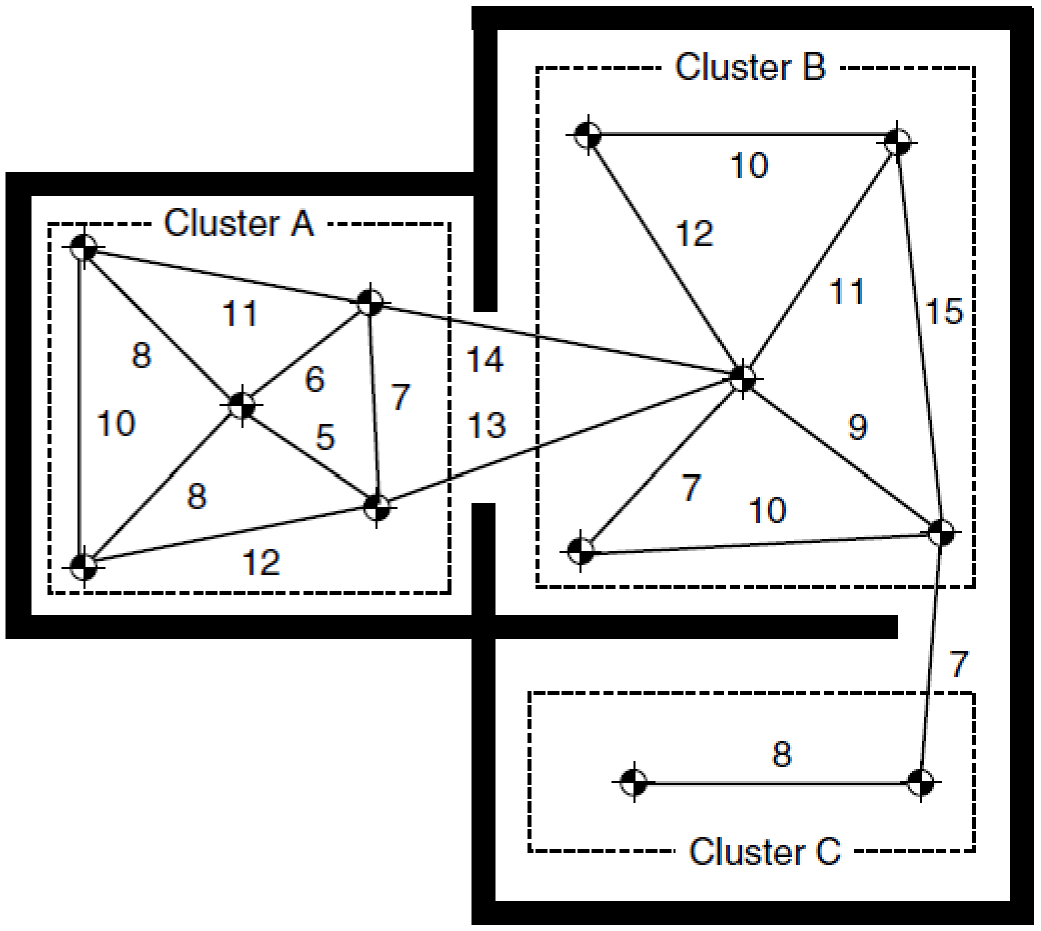
Cluster Heuristic (1)

- The cluster heuristic works by grouping nodes together in clusters
- The nodes in a cluster represent some region of the level that is highly interconnected
- Clustering can be done automatically using graph clustering algorithms, or manually, or a by-product of the level design (portal-based game engines lend themselves well to having clusters for each room)
- A lookup table is then prepared that gives the smallest path length between each pair of clusters
- This is an offline processing step that requires running a lot of pathfinding trials between all pairs of clusters and accumulating their results
- A sufficiently small set of clusters is selected so that this can be done in a reasonable time frame and stored in a reasonable amount of memory
- When the heuristic is called in the game, if the start and goal nodes are in the same cluster, then Euclidean distance is used to provide a result. Otherwise, the estimate is looked up in the table

Cluster Heuristic (2)

- The cluster heuristic often dramatically improves pathfinding performance in indoor areas over Euclidean distance, because it takes into account the convoluted routes that link seemingly nearby locations (the distance through a wall may be tiny, but the route to get between the rooms may involve lots of corridors and intermediate areas)
- Because all nodes in a cluster are given the same heuristic value, the A* algorithm cannot easily find the best route through a cluster
  - Visualized in terms of fill, a cluster will tend to be almost completely filled before the algorithm moves on to the next cluster
- If cluster sizes are small, then the accuracy of the heuristic can be excellent. On the other hand, the lookup table will be large (and the pre-processing time will be huge)
- If cluster sizes are too large, then there will be marginal performance gain, and a simpler heuristic would be a better choice

Illustration: Cluster Heuristic

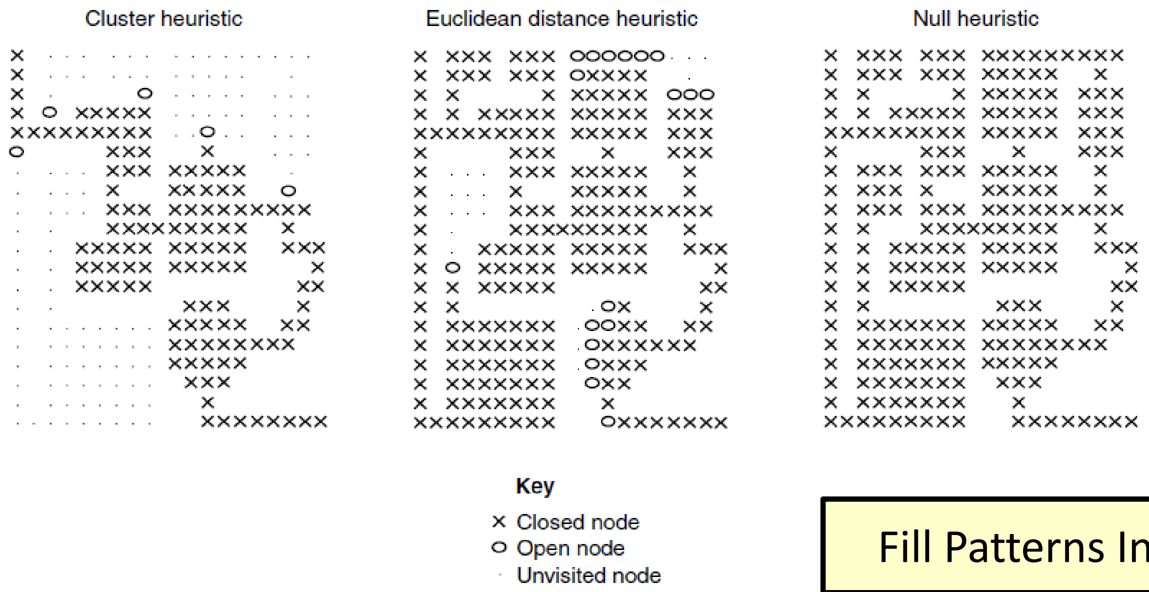


	A	B	C
A	x	13	29
B	13	x	7
C	29	7	x

Lookup table

Fill Patterns in A* (1)

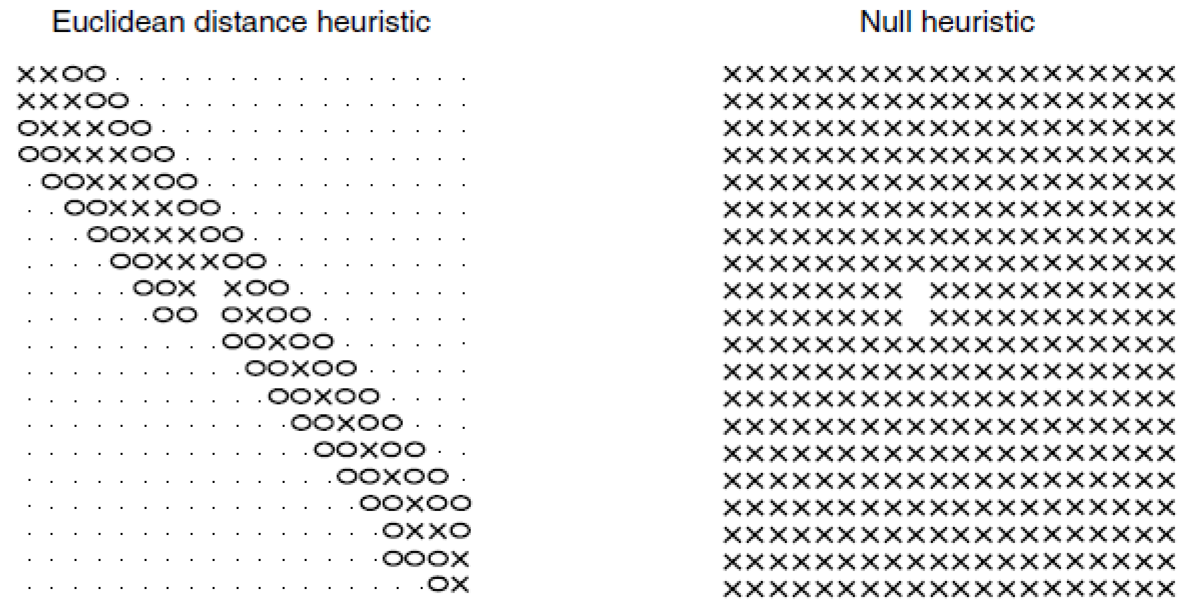
- Knowledge vs. search trade-off: If the heuristic is more complex and more tailored to the specifics of the game level, then the A* algorithm needs to search less. It provides a good deal of knowledge about the problem
- The Euclidean distance provides a little knowledge. It knows that the cost of moving between two points depends on their distance apart. This little bit of knowledge goes a long way, but still requires more searching than the perfect heuristic
- The zero heuristic has no knowledge, and it requires lots of search



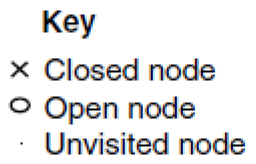
Fill Patterns Indoors

Fill Patterns in A* (2)

- While indoor where there are large obstructions, the Euclidean distance is not the best indicator of the actual distance.
- In outdoor maps, it is far more accurate. Now the Euclidean heuristic is more accurate, and the fill is correspondingly lower



Fill Patterns Outdoors



Dijkstra is A*

- Dijkstra algorithm is a subset of the A* algorithm
  - In A* we calculate the estimated-total-cost of a node by adding the heuristic value to the cost-so-far
  - A* then chooses a node to process based on this value
  - If the heuristic always returns 0, then the estimated-total-cost will always be equal to the cost-so-far
  - When A* chooses the node with the smallest estimated-total-cost, it is choosing the node with the smallest cost-so-far. This is identical to Dijkstra
- A* with a zero heuristic is the pathfinding version of Dijkstra

Reference

- Artificial Intelligence for Games
  - Chapter 4.1-4.3