A presentaion on A* search algorithm

Priyeta Saha - 1605094
Tasnim Rahman - 1605102
Syed Muhammad - 1605110

August 31, 2019

Table Of Contents

Introduction

Figure: Using Navigation Assistants

Definition

What is Graph Traversal?

Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph.

Definition

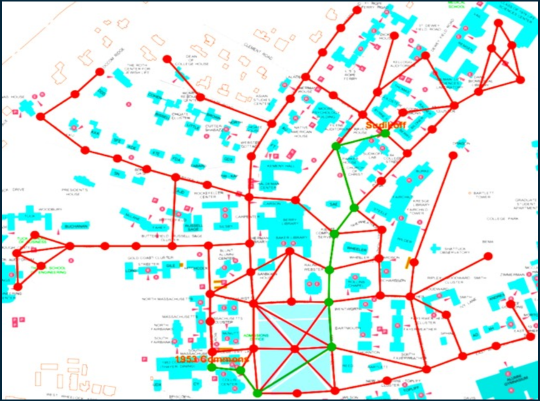


Figure: Using Graph Traversal Algorithms

Definition

What is A* search?

A* search is a path-finding and graph traversal algorithm which is basically an evolution of Dijkstra's algorithm

Why A* search?

As we have already mentioned, A* search is basically a modification of Dijkstra's algorithm.

- But why do we need A* search?

Why A* search?

As we have already mentioned, A* search is basically a modification of Dijkstra's algorithm.

- But why do we need A* search?
- What are the benefits of A* over Dijkstra's algorithm or other such algorithms which validates the modification?

A* search Overview

To answer that question we first need to know how the A* search algorithm works

- You see, Dijkstra's algorithm is a greedy algorithm and so it always immediately chooses the best possible path at every step

A* search Overview

To answer that question we first need to know how the A* search algorithm works

- You see, Dijkstra's algorithm is a greedy algorithm and so it always immediately chooses the best possible path at every step
- Even if the chosen path strays away from the destination, it isn't smart enough to detect that

A* search Overview

To answer that question we first need to know how the A* search algorithm works

- You see, Dijkstra's algorithm is a greedy algorithm and so it always immediately chooses the best possible path at every step
- Even if the chosen path strays away from the destination, it isn't smart enough to detect that
- That's why it needs to visit a lot more nodes to come to a conclusive decision

Figure: A* search vs Dijkstra

How does A* manage to be this efficient?

HEURISTICS

What is Heuristics?

- Heuristics is an approach to problem solving, learning, or discovery

What is Heuristics?

- Heuristics is an approach to problem solving, learning, or discovery
- It employs a practical method not guaranteed to be optimal or perfect

What is Heuristics?

- Heuristics is an approach to problem solving, learning, or discovery
- It employs a practical method not guaranteed to be optimal or perfect
- But it is sufficient for the immediate goals

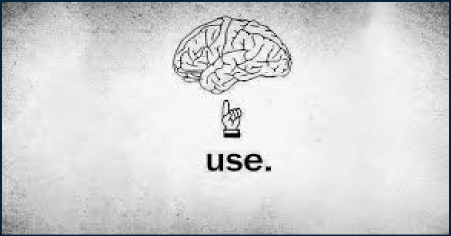


Figure: Using Heuristics

Let's come back to this diagram again

Figure: A* search vs Dijkstra

- How is Heuristics used in A* search?

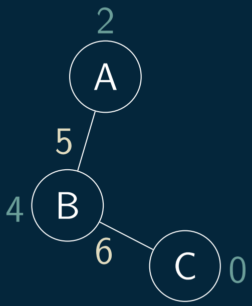
- How is Heuristics used in A* search?
- It can be of many different kinds

- How is Heuristics used in A* search?
- It can be of many different kinds
- For example, the Euclidean distance can be an obvious one

- How is Heuristics used in A* search?
- It can be of many different kinds
- For example, the Euclidean distance can be an obvious one
- The shorter the Euclidean distance of a node from the destination the more likely it is to be included in the optimal path

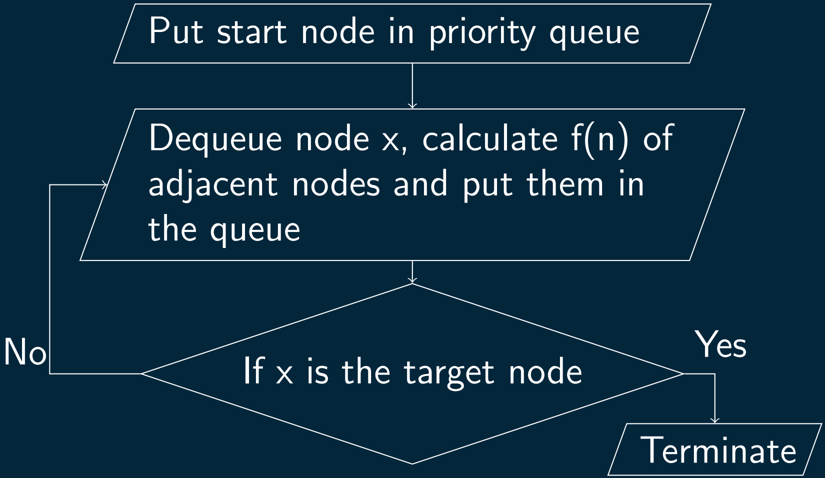
Figure: Using Euclidean Distance as a Heuristic measure

A* Search Algorithm



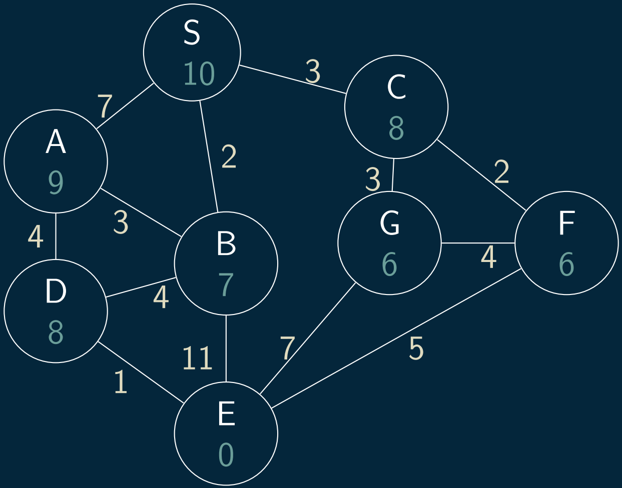
$$f(n) = d(n) + h(n)$$

Flow Chart

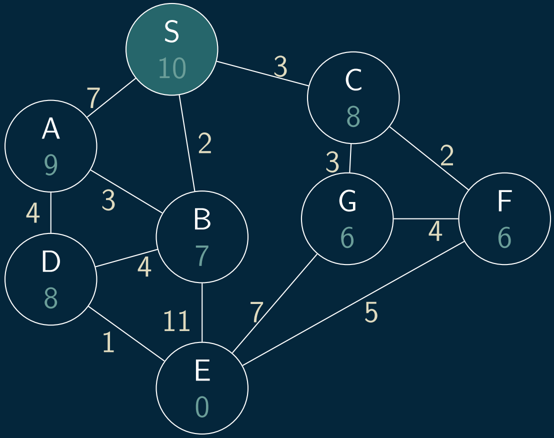

Put start node in priority queue
Dequeue node x, calculate f(n) of adjacent nodes and put them in the queue
If x is the target node
No
Yes
Terminate

Example

Graph



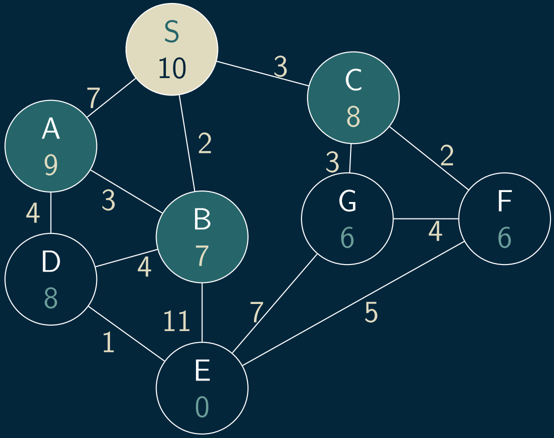
Example



Priority Queue

S(NULL) -> 0 -> 10

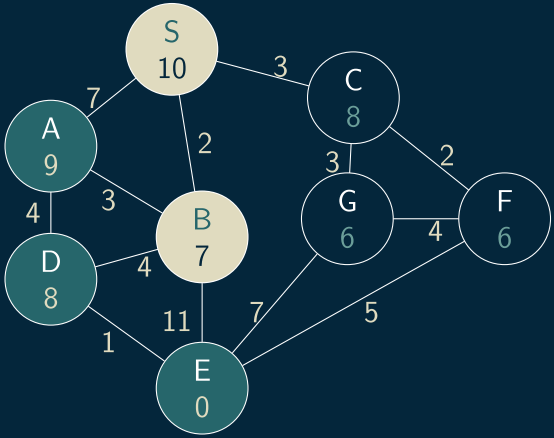
Example



Priority Queue

B(S) -> 2 -> 9
C(S) -> 3 -> 11
A(S) -> 7 -> 16

Example



Priority Queue

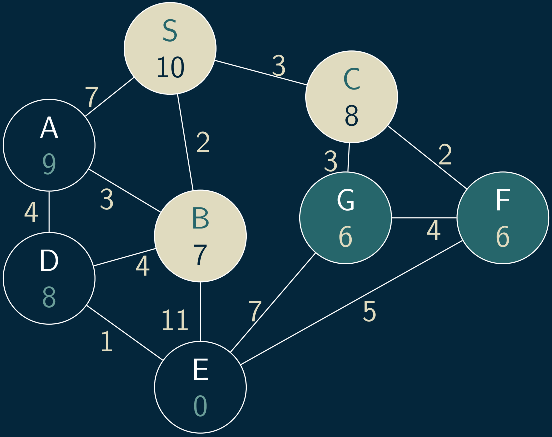
C(S) -> 3 -> 11

E(B) -> 13 -> 13

A(B) -> 5 -> 14

D(B) -> 6 -> 14

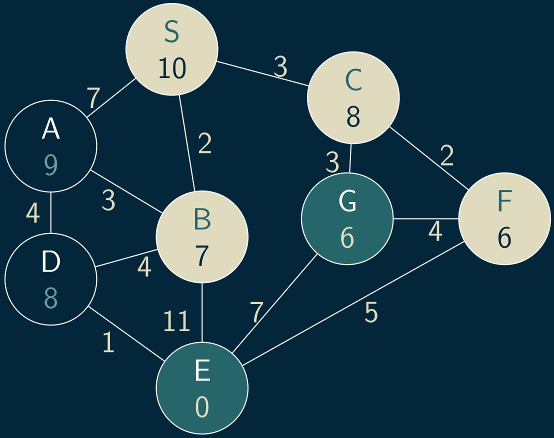
Example



Priority Queue

F(C) -> 5 -> 11
G(C) -> 6 -> 12
E(B) -> 13 -> 13
A(B) -> 5 -> 14
D(B) -> 6 -> 14

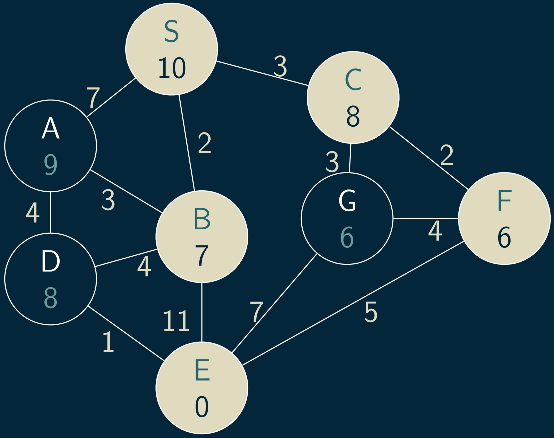
Example



Priority Queue

E(F) -> 10 -> 10
G(C) -> 6 -> 12
A(B) -> 5 -> 14
D(B) -> 6 -> 14

Example



Priority Queue

G(C) -> 6 -> 12
A(B) -> 5 -> 14
D(B) -> 6 -> 14

Example

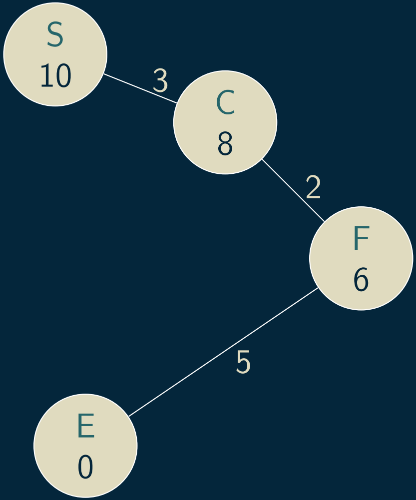


Figure: Expected shortest path

Exact Heuristics

Using exact distance from any node n to goal node as $h(n)$

- Precompute distances between each pair of nodes

Exact Heuristics

Using exact distance from any node n to goal node as $h(n)$

- Precompute distances between each pair of nodes
- Time-consuming

Exact Heuristics

Using exact distance from any node n to goal node as $h(n)$

- Precompute distances between each pair of nodes
- Time-consuming
- Equal to Euclidean distance if there is no other node on the direct path

Manhattan Distance

$$h(n) = \mid n.x - g.x \mid + \mid n.y - g.y \mid$$

- Suitable for 2D grids
- Movement is allowed only in x or y axis (4 directions)

Diagonal Distance

$$h(n) = \max\{|\ n.x - g.x\ |, |\ n.y - g.y\ |\}$$

- Suitable for games like chess
- Both axial and diagonal movements are allowed (8 directions)

Euclidean Distance

$$h(n) = \sqrt{(n.x - g.x)^2 + (n.y - g.y)^2}$$

- Suitable for graphs
- Movement in any direction is allowed

Implementation

Implementation details that can affect the performance of A*

- Breaking ties in deciding minimum element of priority queue

Implementation

Implementation details that can affect the performance of A*

- Breaking ties in deciding minimum element of priority queue
- Alternative heap implementations of priority queue

Breaking Ties

How to break ties in priority queue?

- **lowest heuristic cost**
  - Prefer the nodes closest to the goal

Breaking Ties

How to break ties in priority queue?

- **lowest heuristic cost**
  - Prefer the nodes closest to the goal
- **cross product of s-g and n-g vectors**
  - Prefer paths on the start-goal straight line

Breaking Ties

How to break ties in priority queue?

- **lowest heuristic cost**
  - Prefer the nodes closest to the goal
- **cross product of s-g and n-g vectors**
  - Prefer paths on the start-goal straight line
- **LIFO like DFS**
  - Prefer new insertions to old insertions

Which heap to use?

Binary Heap

VS

Fibonacci Heap

Which heap to use?

Operation	Delete-min	Insert	Decrease-key
Binary Heap	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Fibonacci Heap	Amortized $\mathcal{O}(\log n)$	$\Theta(1)$	Amortized $\Theta(1)$

Table: Time-complexity comparisons

Which heap to use?

Binary Heap

VS

Fibonacci Heap

Time Complexity

Branching Factor (b)

Average number of neighbours at each node

Time Complexity

Branching Factor (b)

Average number of neighbours at each node

Depth of Solution (d)

Number of nodes in the shortest path

Time Complexity

Branching Factor (b)

Average number of neighbours at each node

Depth of Solution (d)

Number of nodes in the shortest path

Time Complexity : $\mathcal{O}(b^d)$ or $\mathcal{O}(\mid E \mid)$

Where do we use A*?

Figure: Tower Defense

Where do we use A*?

- **Finding shortest path in games**

  - mostly single source, single destination

Where do we use A*?

- **Finding shortest path in games**
  - mostly single source, single destination
- **Graph Traversal**
  - originally designed for this purpose

Where do we use A*?

- **Finding shortest path in games**
  - mostly single source, single destination
- **Graph Traversal**
  - originally designed for this purpose
- **Artificial Intelligence**
  - informed search algorithm for agents

Where do we use A*?

- **Finding shortest path in games**
  - mostly single source, single destination
- **Graph Traversal**
  - originally designed for this purpose
- **Artificial Intelligence**
  - informed search algorithm for agents
- **Navigation**
  - Google Maps & other digital maps

THE END

Thank You !

Feel free to ask any questions