

CSCI203

Algorithms and Data Structures



Graphs (II)

Lecturer: Dr. Fenghui Ren

Room 3.203

Email: fren@uow.edu.au

Tweaking Dijkstra



- ▶ As we have already seen, Dijkstra's algorithm provides an effective solution strategy for solving the **single source-all destinations** version of the shortest path problem.
- ▶ We will now look at a few simple modification of Dijkstra that will:
 - Improve its practical performance;
 - and/or extend its range of applicability.

Recall Dijkstra's Algorithm

```
Procedure Dijkstra(G: array[1..n, 1..n]): array [2..n]
  D: array[2..n], P: array[2..n]
  C: set = {2, 3, ..., n}
  for i = 2 to n do
    D[i] = G[1, i]
    P[i]=1
  od
  repeat
    v = the index of the minimum D[v] not yet selected
    remove v from C // and implicitly add v to S
    for each u  $\in$  C do
      D[u] = min(D[u], D[v] + G[v, u])
      p[u] = v
    rof
  until C contains no reachable nodes
  return D
end Dijkstra
```

Overall Efficiency

- ▶ The algorithm as you have seen it so far has efficiency $O(|V|^2 + |E|)$.
- ▶ But it was $O(|V| * \log |V| + |E|)$, How come?
- ▶ The answer is simple:
 - As presented, we find the next vertex to select by searching a list of candidate vertices and selecting the vertex with minimum D value.
 - This is a linear search process; $O(|V|)$.
 - We do this for each vertex, also $O(|V|)$.
 - This is $O|V^2|$.
- ▶ So, how do we improve on this?

Reaching Peak Efficiency

- ▶ The answer is surprisingly simple.
- ▶ Replace the candidate list/array C with...
- ▶ ...a priority queue (or a heap), ordered on $D(v)$.
- ▶ Now:
 - Finding the best candidate is $O(1)$;
 - Updating C is $O(\log |V|)$.
- ▶ Now, over all vertices, we have $|V| * \log |V|$.

All Sources - Single Destination

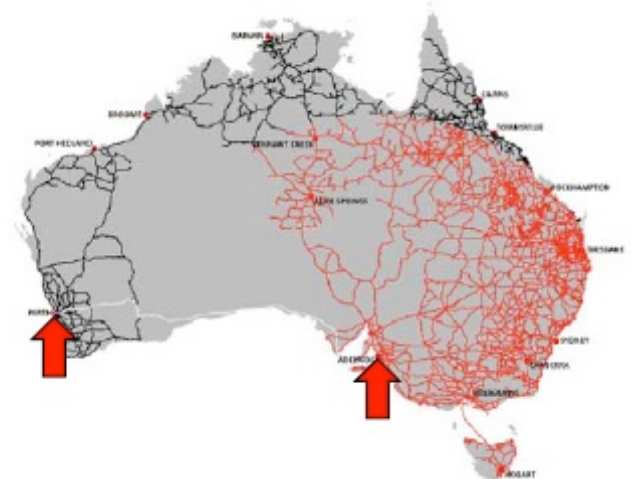
- ▶ In this case, rather than finding paths from a starting vertex, s , to all other reachable vertices we are looking for the shortest paths to some goal vertex, g , from all possible starting vertices.
- ▶ How do we do this?
 - Run *Dijkstra backwards*.
- ▶ Specifically:
 - Redefine $Adj(v)$ to be the list of set of vertices leading to vertex v ...
 - ...instead of reachable from v ;
 - Start with $D(g) = 0$...
 - ...instead of $D(s) = 0$;
 - Let $P(v)$ indicate the next vertex in the path...
 - ...instead of the prior vertex;
 - Let the selected set, S , start at $\{g\}$...
 - ...instead of $\{s\}$.

Single Source - Single Destination

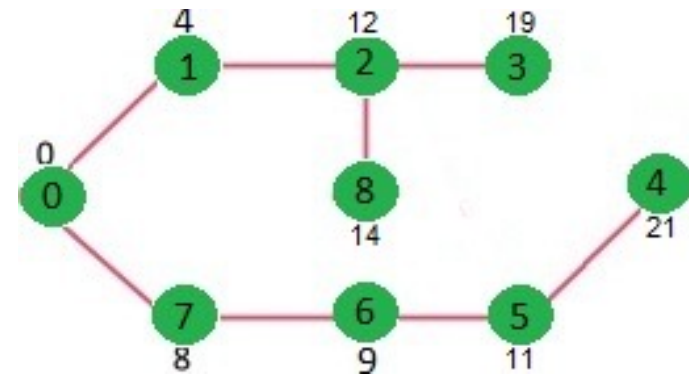
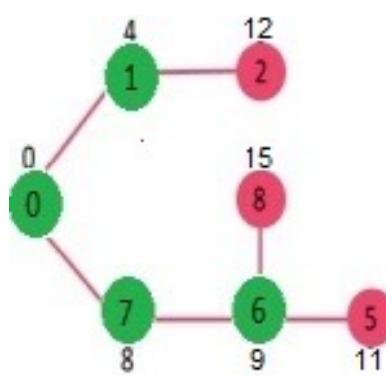
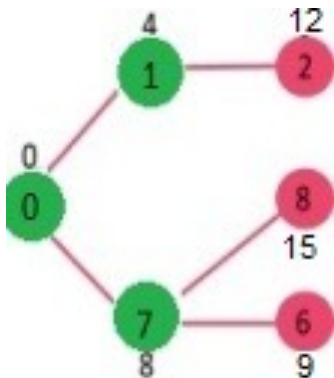
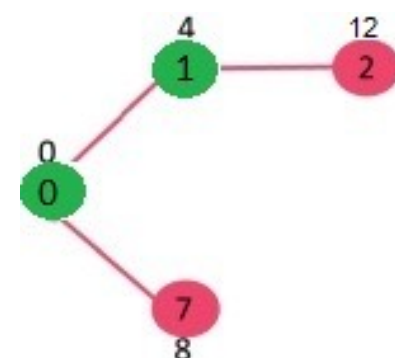
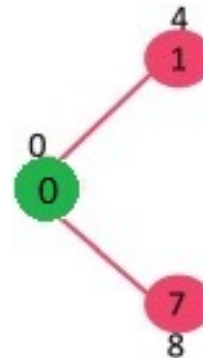
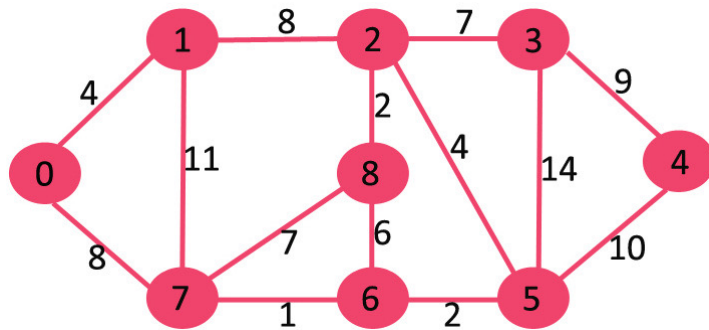
- ▶ What do we do if, rather than looking for the shortest paths from a start vertex, s , to all other vertices, we wish to find the shortest path from s to a specific goal vertex, g ?
- ▶ The answer is easy:
 - Stop when vertex g becomes a member of S , the selected set.
- ▶ This means that we do not waste time with any vertex further away from s than g .
- ▶ This usually reduces the total running time of the algorithm.
 - Why?

The Problem with Dijkstra's

- ▶ There is a big problem with using Dijkstra on the single source/single destination problem:
 - The order in which the vertices are added.
- ▶ Consider the following graph:
 - Say we want to get from Adelaide...
 - ...to Perth.
- ▶ Dijkstra will add all of the closer vertices first:
- ▶ Before we ever get close to the path we seek.



Another Example

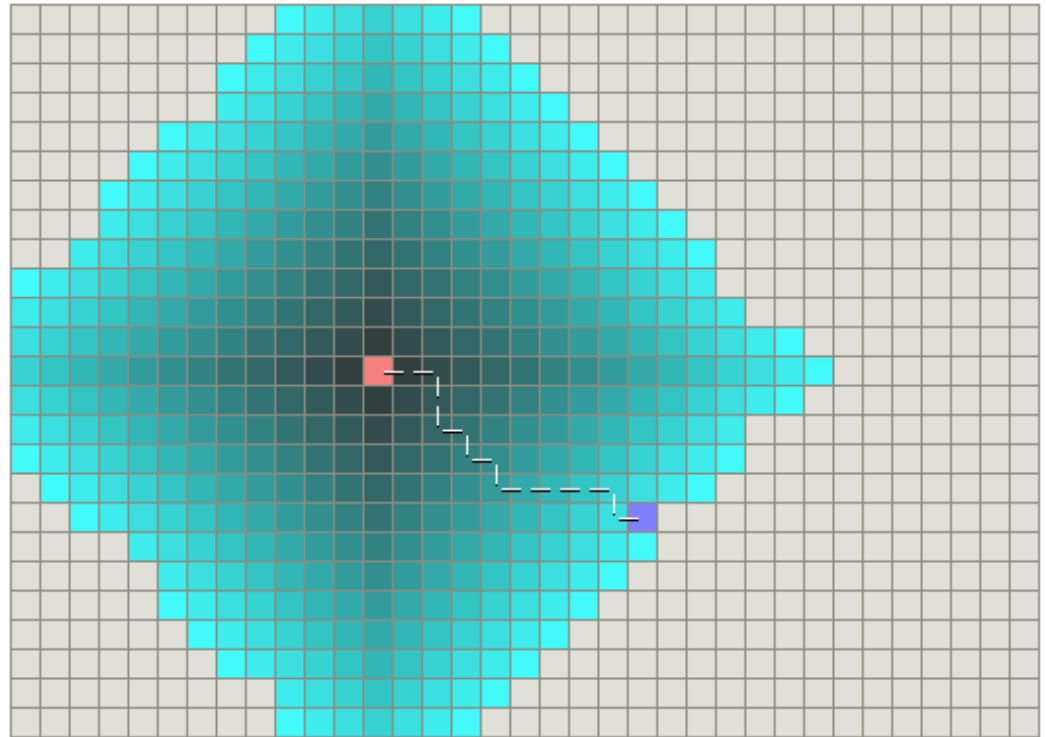


$$v_0 \rightarrow v_4$$

Another Example

Dijkstra Algorithm

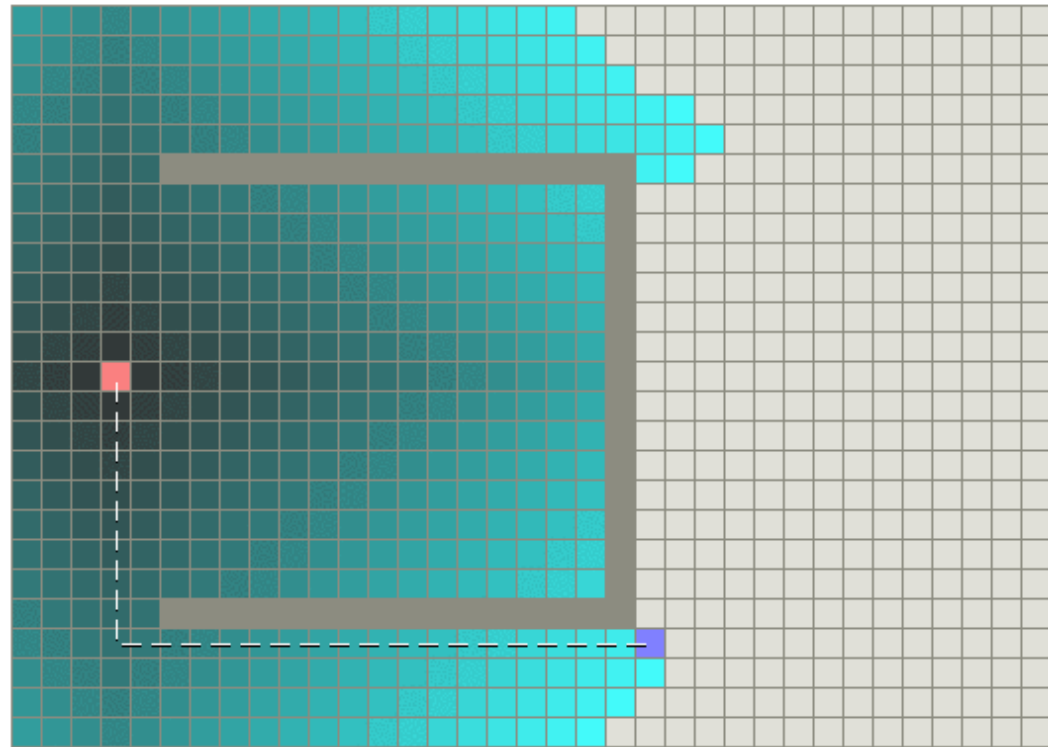
- Repeatedly examines the closest not-examined vertex
- Expands outwards until reach the goal
- Guarantee to find a shortest path
- But have a large teal area



Another Example

Dijkstra Algorithm

- The shortest path is not a straight line when has obstacles
- Works harder but is guaranteed to find a shortest path



Fixing the Problem

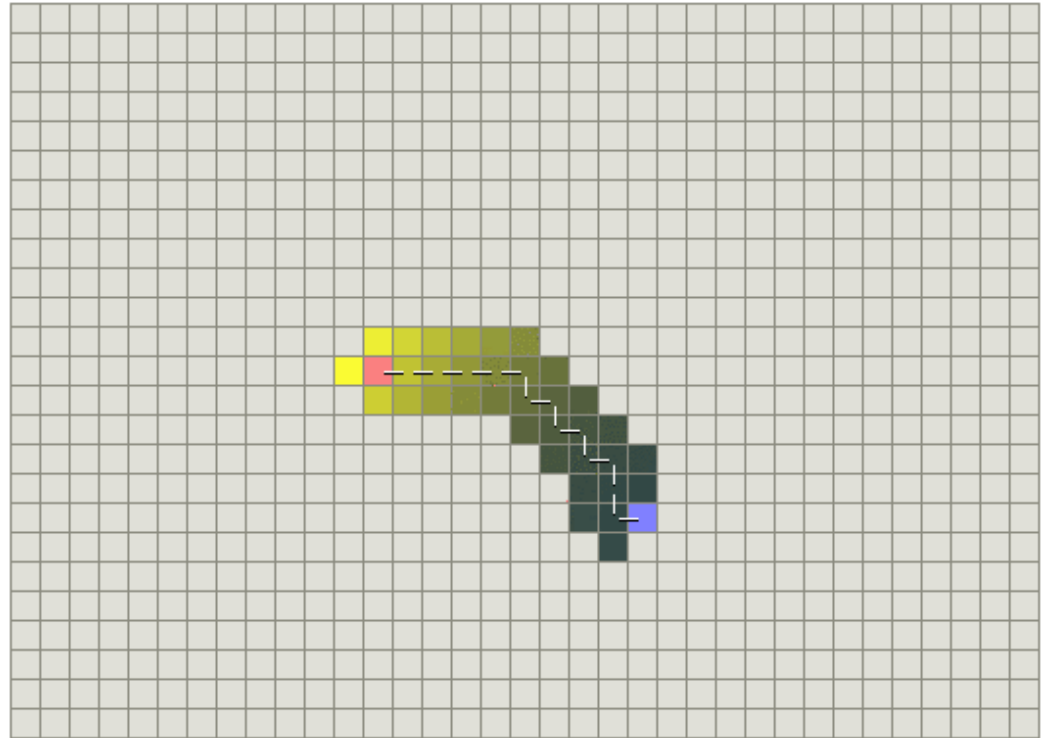
- ▶ So, how do we remedy this?
- ▶ Before we get to the answer let us take a step back.
- ▶ Let us **generalize** Dijkstra's algorithm.
- ▶ The key step in the algorithm is on the process by which we select the next vertex.
 - Specifically, we select the vertex in the candidate set, C , for which the overall distance to the vertex from the source vertex, s , is minimized.
 - $D(v) = P(s, v)$.
- ▶ Note that this does not involve the goal vertex, g .

Eyes on the Goal

- ▶ What if we could **bias** the selection towards the goal in some way.
- ▶ We can!
- ▶ Essentially, we select the minimum not simply of $D(s, v)$ but, instead of $D(s, v) + H(v, g)$.
- ▶ This new function, H , is a *heuristic*; an estimate of the remaining distance from each candidate vertex, v , to the goal vertex, g .
- ▶ What is a good estimator?

Greedy Best-First-Search Algorithm

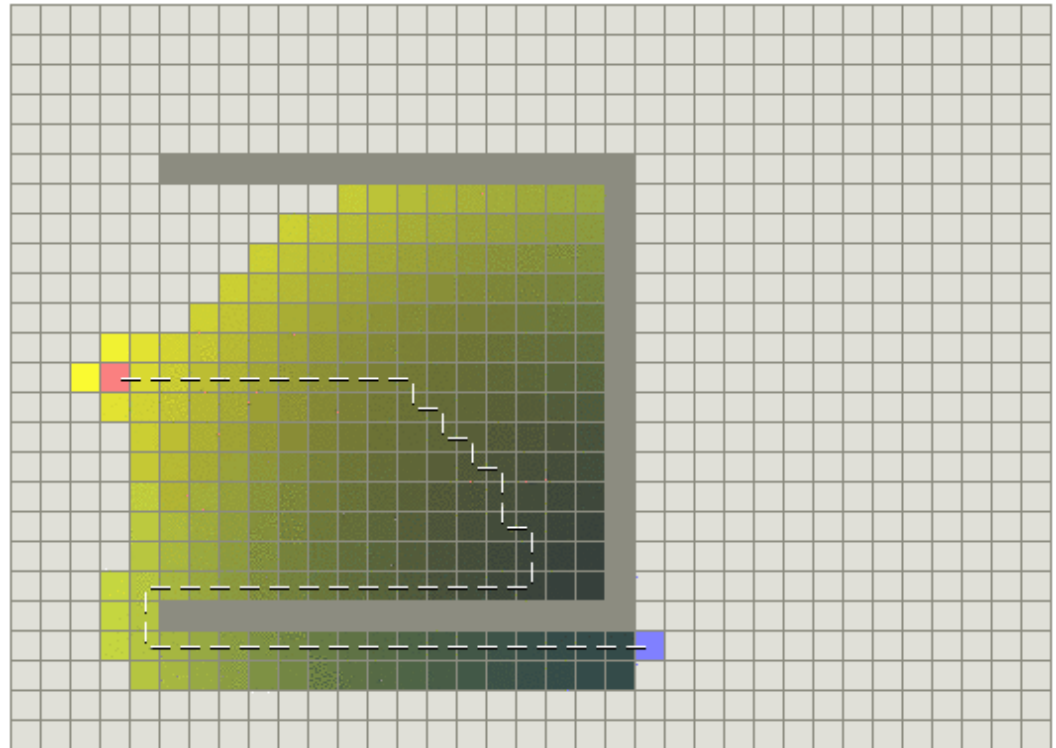
- Similar as the Dijkstra algorithm but with a heuristic
- Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal.
- Not guarantee to find a shortest path
- But it is much quicker



Greedy Best-First-Search Algorithm

- Does less work and faster
- But the path is not the shortest

Can we combine the Dijkstra and the Greedy algorithms?



Greedy Best-First-Search Algorithm

procedure GBS(start, target) is:

- mark start as visited

- add start to queue

- while queue is not empty do:

 - current_node \leftarrow vertex of queue with min distance to target

 - remove current_node from queue

 - foreach neighbor n of current_node do:

 - if n not in visited then:

 - if n is target:

 - return n

 - else:

 - mark n as visited

 - add n to queue

20/9/2022
return failure

A Good Heuristic

- ▶ We require $H(v, g)$ to have one key property:
 - $H(v, g) \leq P(v, g)$;
 - The heuristic estimate must never exceed the actual shortest path length.
 - This requirement guarantees that the final path we find is still the correct answer.
- ▶ In our example we have a ready-made heuristic...
- ▶ ...The Euclidean (straight-line) distance between v and g .
- ▶ Provided the graph behaves according the rules of geometry there can never be a shorter path than this.

A*



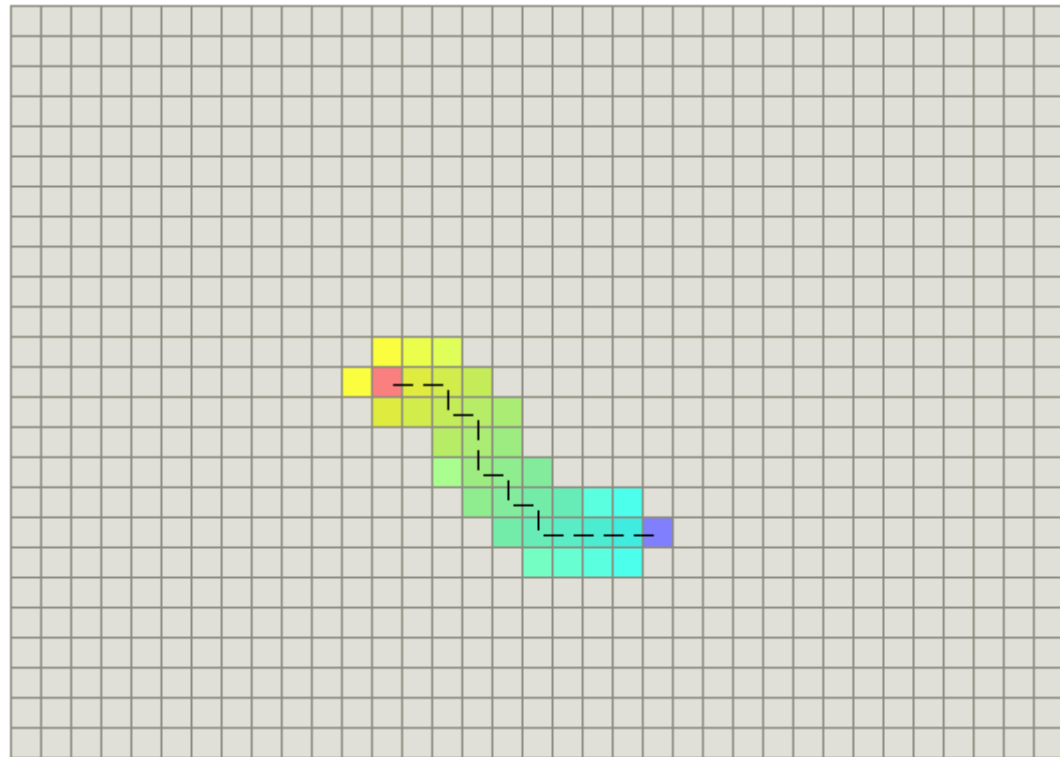
- ▶ The heuristic modification to the vertex selection rule changes Dijkstra's algorithm into an example of what is called the A^* algorithm.
- ▶ Although, in the worst case, A^* is no faster than Dijkstra in practice it will generally represent a substantial improvement.
- ▶ Note: the **trick to A^* is finding a good heuristic.**
- ▶ The nearer that $H(v, g)$, the estimated minimum path length from v to g , is to $P(v, g)$, the actual minimum path length, the faster A^* will find the solution.

Through the Looking Glass

- ▶ Because of the order in which we saw them, it is easy to think of A^* as a generalization of Dijkstra's algorithm.
- ▶ This is not the only, or perhaps even the best, way to view this.
- ▶ Consider instead this viewpoint:
- ▶ Dijkstra's algorithm is simply a special case of A^* :
- ▶ The one with the worst possible choice of H .
- ▶ Specifically, $H(v, g) = 0$.

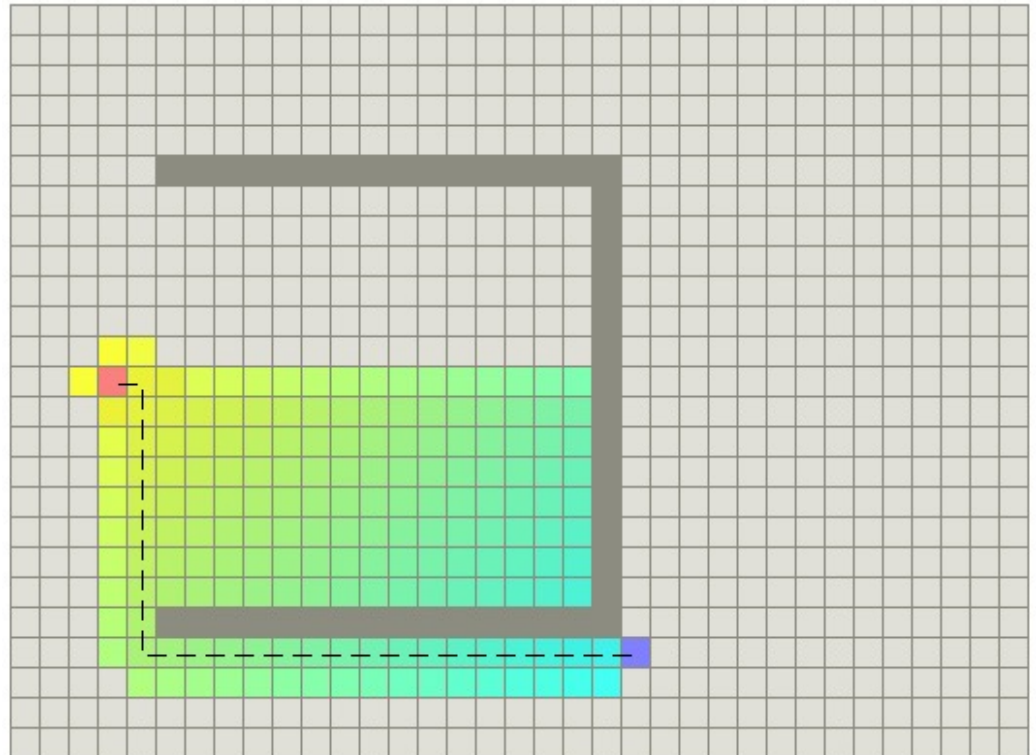
A*

- ▶ A* algorithm is like Dijkstra's algorithm to find the shortest path
- ▶ A* algorithm also is like Greedy Best-First-Search in that it can use a heuristic to guide itself
- ▶ In short, A* algorithm is as reliable as Dijkstra's algorithm and as fast as Greedy Best-First-Search algorithm

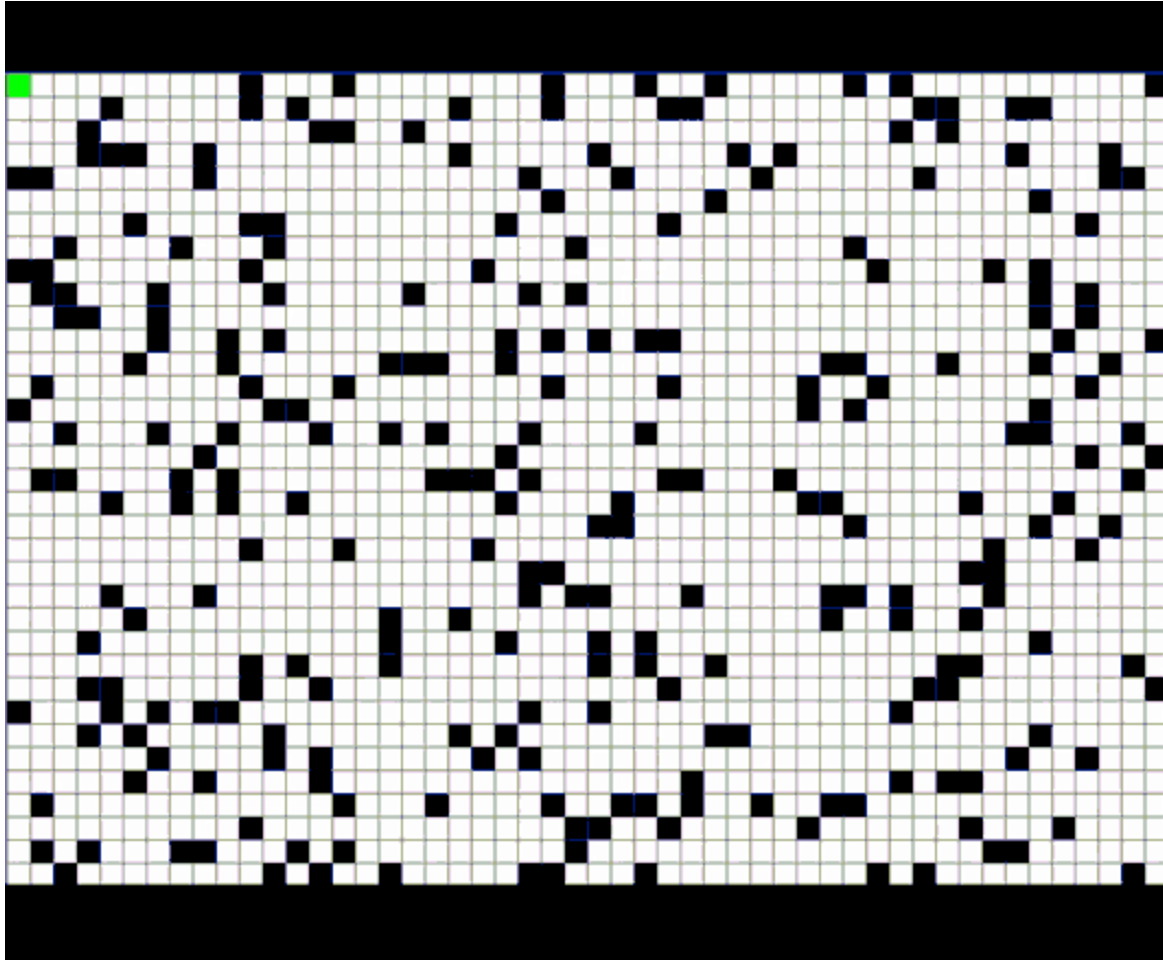


A*

- ▶ A* algorithm finds a path as good as what Dijkstra's algorithm found
- ▶ But A* algorithm is faster.
- ▶ Because it combines the Dijkstra's algorithm with the heuristics.




A^*



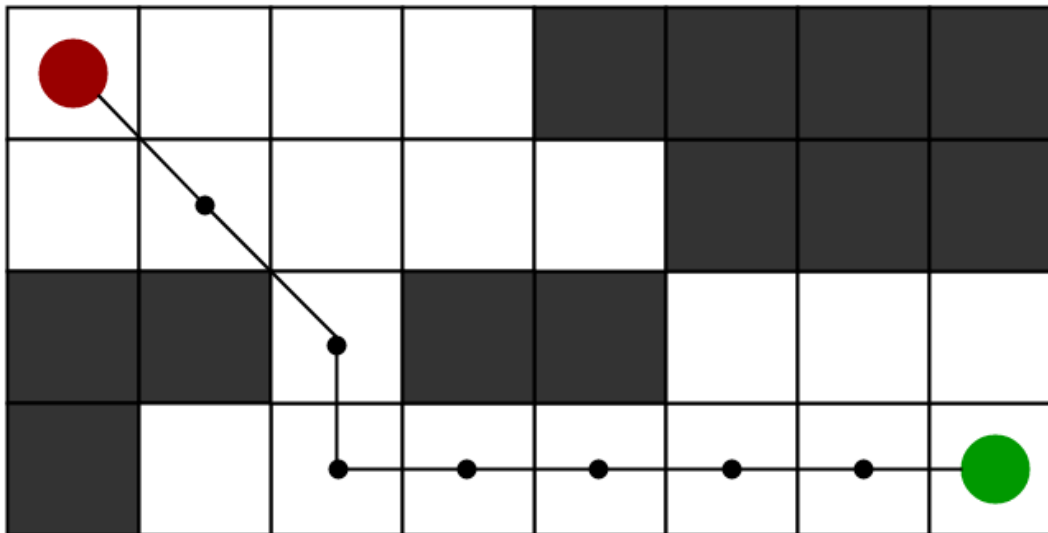
// A* Search Algorithm

1. Initialize the open list - a list of nodes to be explored
 2. Initialize the closed list - a list of nodes in the path so far
put the starting node on the open list (you can leave its f at zero)
 3. while the open list is not empty
 - a) find the node with the least f on the open list, call it "q"
 - b) pop q off the open list
 - c) generate q's non-blocked successors from the 8 ones and set their parents to q
 - d) for each successor
 - i) if successor is the goal, stop search
successor.g = q.g + distance between successor and q
successor.h = distance from goal to successor
// This can be done using many ways, we will discuss three heuristics-
// Manhattan, Diagonal and Euclidean Heuristics
successor.f = successor.g + successor.h
 - ii) if a node with the same position as successor is in the OPEN list
which has a lower f than successor, skip this successor
 - iii) if a node with the same position as successor is in the CLOSED list
which has a lower f than successor, skip this successor otherwise,
add the node to the open list
 - e) push q on the closed list
- end (while loop)

1	2	3
8		4
7	6	5

An Example

- ▶ We can consider a 2D Grid having several obstacles and we start from a source cell (green), s , to reach towards a goal cell (red), z
- ▶ We want to reach the target cell (if possible) from the starting cell as quickly as possible.



1	2	3
8	■	4
7	6	5

An Example

In the A^* Algorithm

- ▶ At each step, it picks the node (while cell), x according to a value $f(s, z) = g(s, x) + h(x, z)$
 - $g(s, x)$ - cost/distance to moved from s to x
 - $h(x, z)$ - estimated cost/distance, heuristic, to move from x to z
- ▶ We don't know the actual $f(s, z)$ until we find the path
- ▶ There are many ways to calculate $h()$

Heuristic h



- ▶ We can calculate g but how to calculate h ?
- ▶ We can do things.
 - Either calculate the exact value of h (which is certainly time consuming). OR
 - Approximate the value of h using some heuristics (less time consuming).

Heuristic h - Exact Heuristic



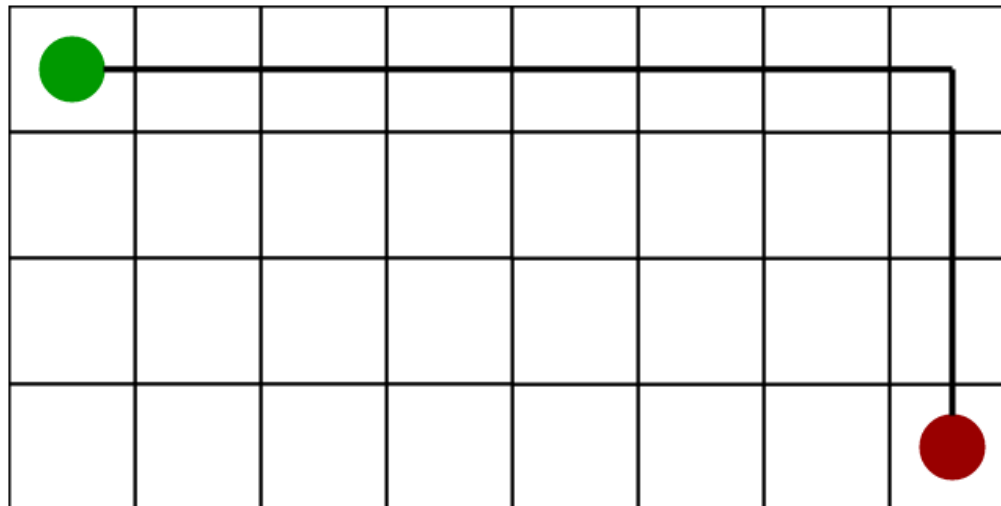
- ▶ We can find exact values of h , but that is generally very time consuming.
- ▶ e.g. Pre-compute the distance between each pair of cells before running the A^* Search Algorithm.

Heuristic h - Approximation

► Manhattan Distance -

$$h = \text{abs}(\text{current_cell}.x - \text{goal}.x) + \text{abs}(\text{current_cell}.y - \text{goal}.y)$$

- The sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively,
- When to use this heuristic? - When we are allowed to move only in four directions only (right, left, top, bottom)



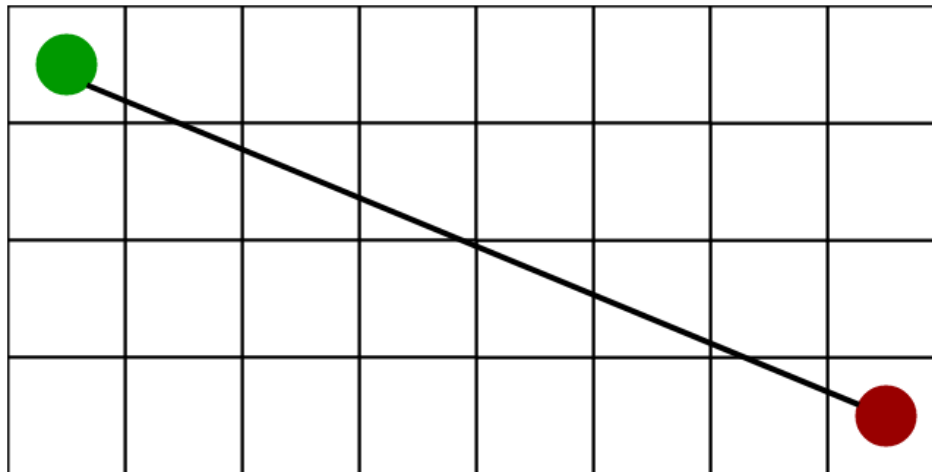
Heuristic...Approximation

► Euclidean Distance-

- the distance between the current cell and the goal cell using the distance formula

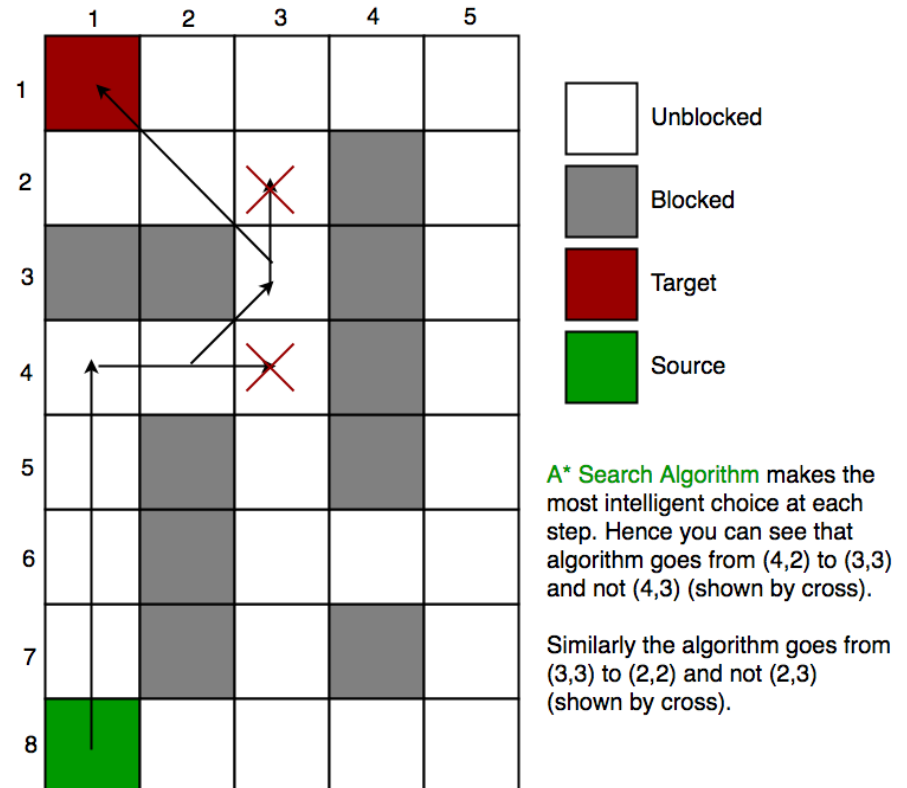
$$h = \text{sqrt} ((\text{current_cell.x} - \text{goal.x})^2 + (\text{current_cell.y} - \text{goal.y})^2)$$

- When to use this heuristic? - When we are allowed to move in any directions.



An Example

- ▶ A* Search algorithm would follow path as shown below if Euclidean Distance is chosen as a heuristics.



Related References



- ▶ Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
 - Chapters 9.3
- ▶ Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
 - Chapters 24.1 and 24.3