CPSC 427
Video Game Programming

AI and Strategy

Helge Rhodin

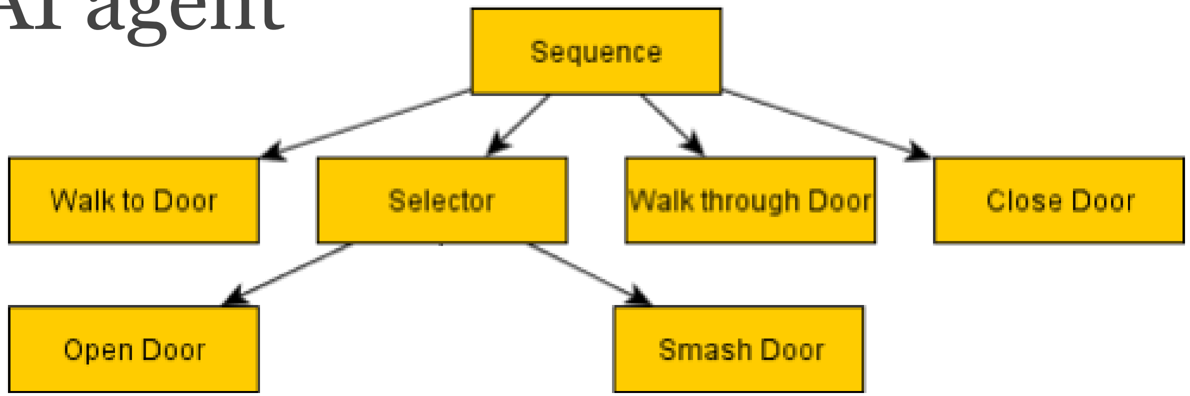
Overview

Learning outcome:

- ***Link data structure and algorithm knowledge to game dev.***
- ***Understand search algorithms (breadth first, depth first, A*, min max)***

Recap: Behaviour Trees

- flow of decision making of an AI agent
- tree structured
- ***Each frame:***
  - Visit nodes from root to leaves
    - *depth-first order*
    - *check currently running node*
      - succeeds or fails:
      - return to parent node and evaluate its Success/Failure
      - the parent may call new branches in sequence or return Success/Failure
      - continues running: recursively return Running till root (usually)



New: A leaf node with internal state

Example scenarios

1. ***Run three steps, turn around, run one step back***

2. ***Turn right, run three steps, turn around***

Live demo

Multiple components for one entity?

Classical ECS:

- ***Each entity***
  - has one ID
  - has or has not a certain component type
  - cannot store multiple components of the same type

Character inventory:

- A character should be able to hold multiple portions of the same type
- Solution:
  - Each item is its own entity
  - Introduce an inventory component that stores list of items (list of entities)

The same b-tree for multiple entities?

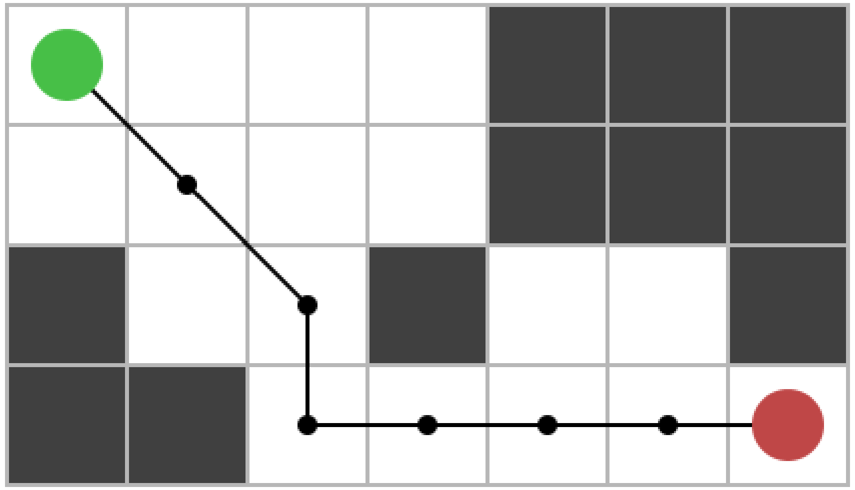
- ***How to store the state with each entity?***
- within the ECS registry?
  - *add a new state component for each b-tree node?*
    - what if multiple nodes of the same type run on the same entities?
- a custom data structure?
  - *a lookup table?*
    - conditioned on entity ID!

Strategy

- Given current state, determine **BEST** next move
- Short term: best among immediate options
- Long term: what brings something closest to a goal
  - *How?*
    - Search for path to best outcome
      - Across states/state parameters

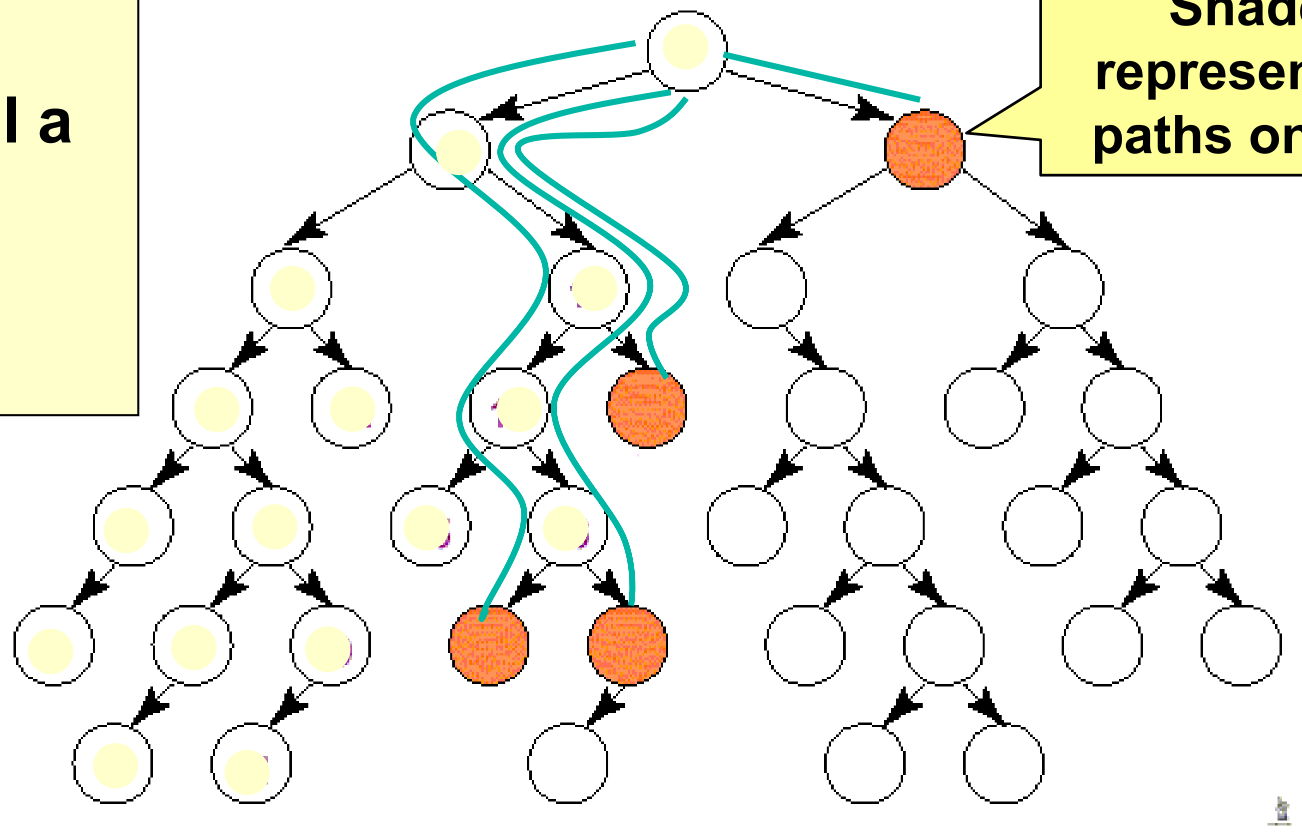
Pathfinding

- **How do I get from point A to point B?**



DFS: Depth-first search

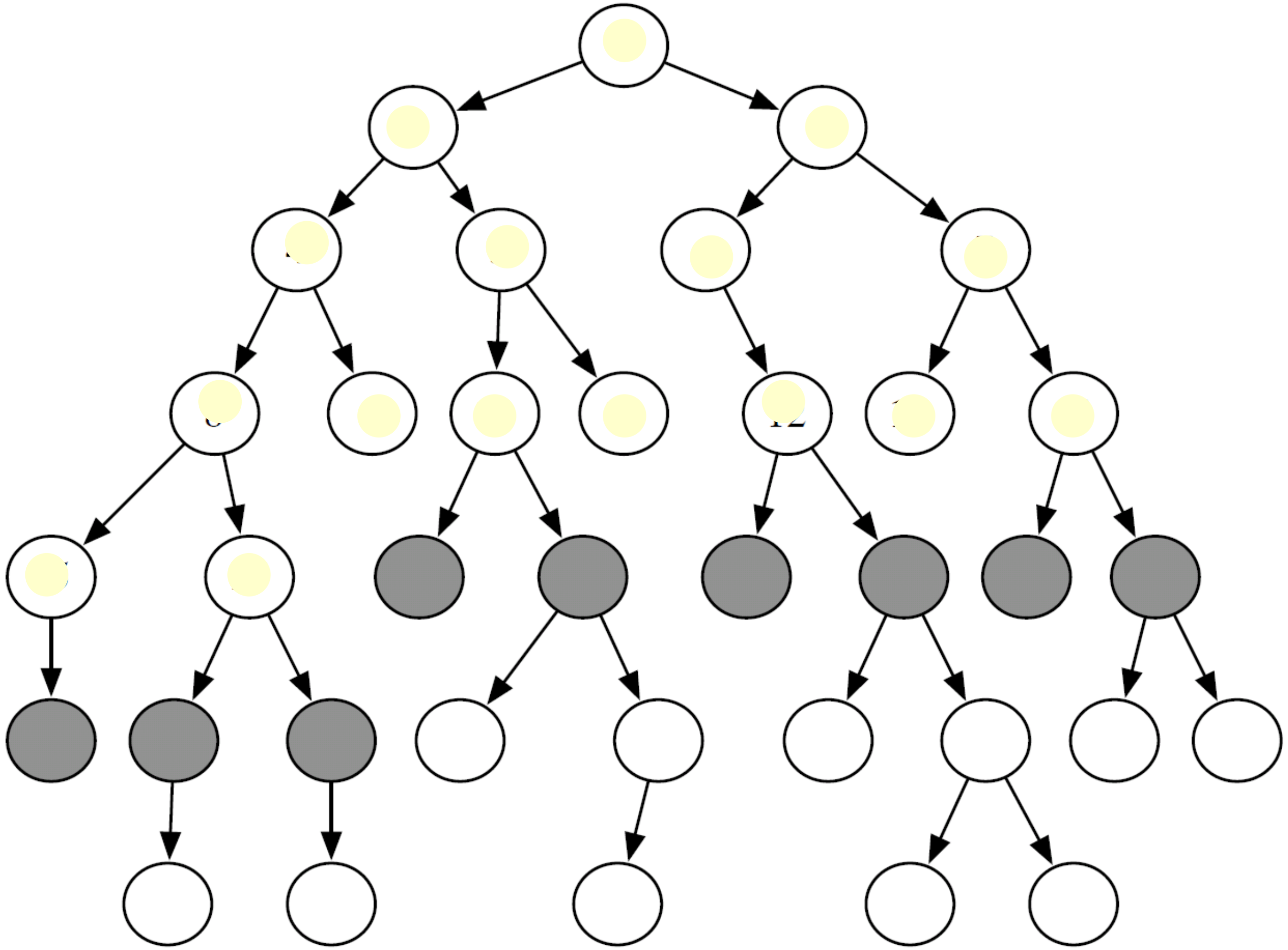
Explore each path on the frontier until its end (or until a goal is found) before considering any other path.



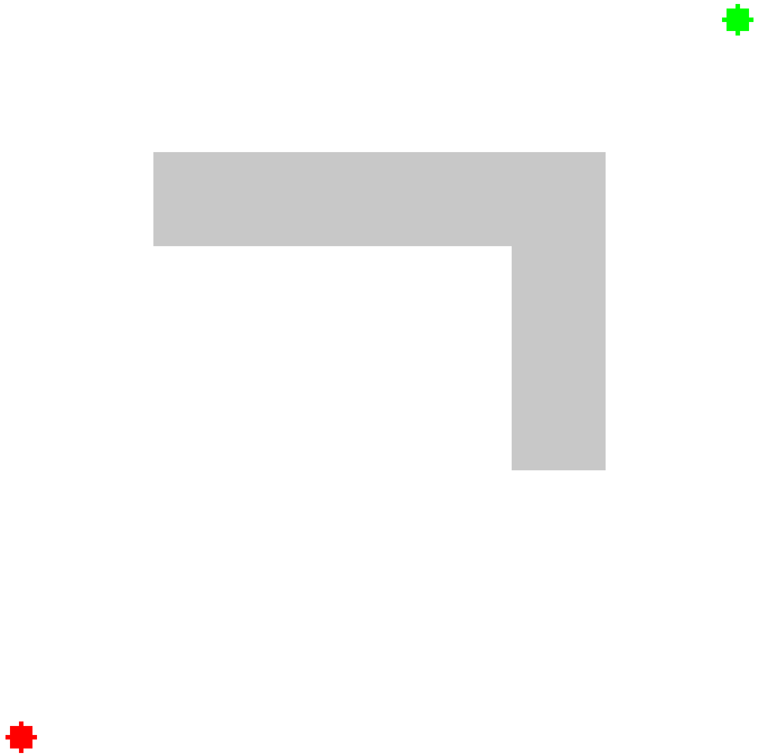
Shaded nodes represent the end of paths on the frontier

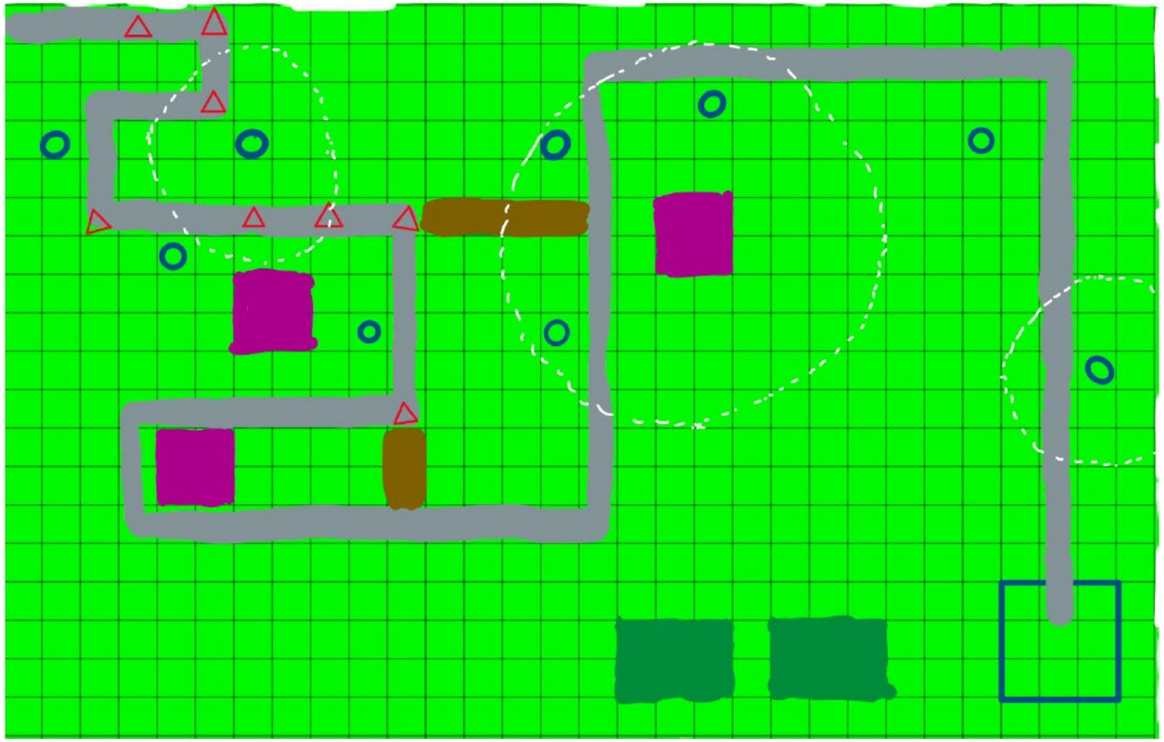
Breadth-first search (BFS)

- **Explore all paths of length L on the frontier, before looking at path of length *L + 1***



Breadth-first





Project pitch Team 4

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

When to use BFS vs. DFS?

- ***The search graph has cycles or is infinite***

  BFS

- ***We need the shortest path to a solution***

  BFS

- ***There are only solutions at great depth***

  DFS

- ***There are some solutions at shallow depth***

  BFS

- ***No way the search graph will fit into memory***

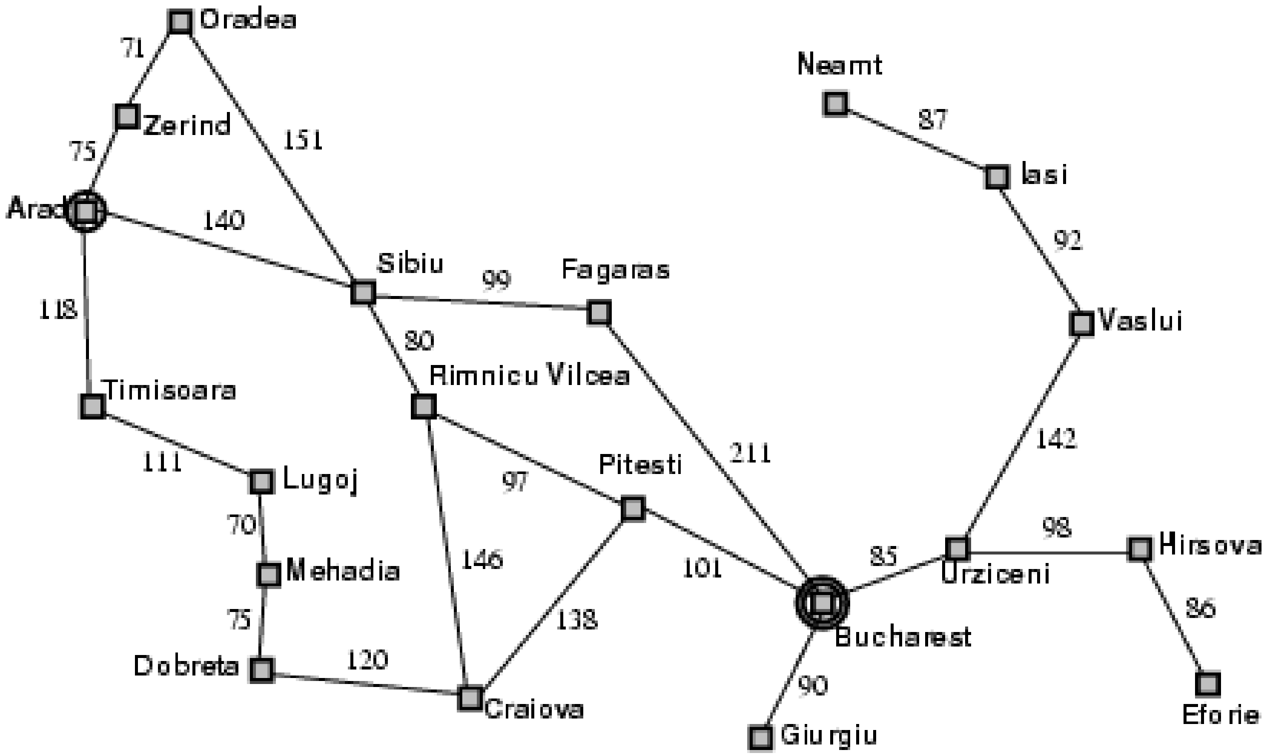
  DFS

Search with Costs

Def.: The **cost of a path** **is the sum of the** **costs of its arcs**

$$\text{cost}(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} \text{cost}(\langle n_{i-1}, n_i \rangle)$$

Want to find the solution that minimizes cost



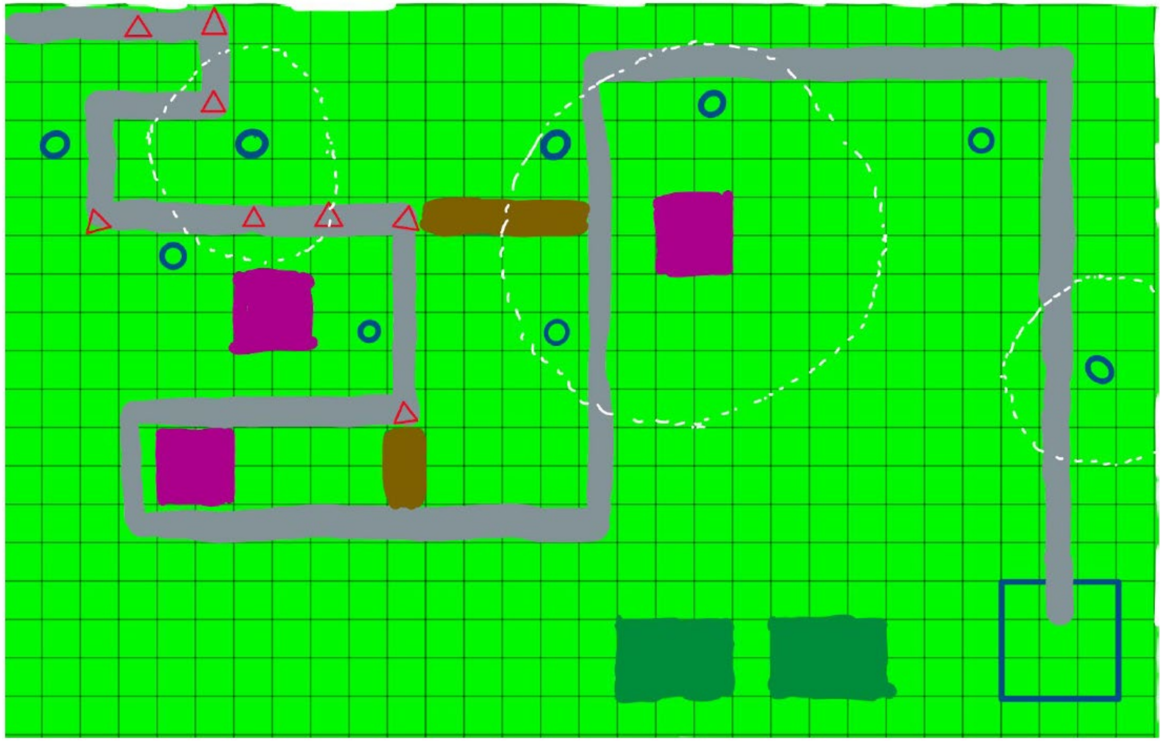
Example: Tower Defence

Normal unit motion cost:

- Street: cost 1
- Other: cost infinity

Boss unit: *which shortcuts will it take?*

- Street: cost 1
- Dirt road: cost 5
- Grass: cost 50
- Purple stuff: cost infinity



Lowest-Cost-First Search (LCFS)

- **Lowest-cost-first search** finds the path with the **lowest cost** to a goal node
- At each stage, it **selects** the path with the **lowest cost** on the frontier.
- The **frontier** is implemented as a priority queue ordered by path cost.

Use of search

- Use search to determine next state (next state on shortest path to goal/best outcome)
- Measures:
  - *Evaluate goal/best outcome*
  - *Evaluate distance (shortest path in what metric?)*

Problems:

- Cost of full search (at every step) can be prohibitive
- Search in adversarial environment
  - *Player will try to outsmart you*

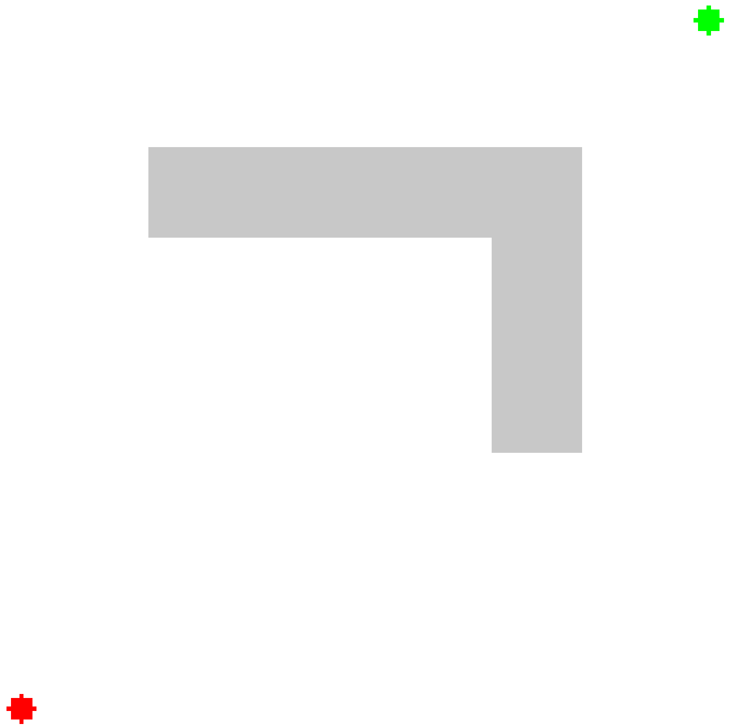
Heuristic Search

- Blind search algorithms do not take goal into account until they reach it
- We often have estimates of distance/cost from node n to a goal node
- **Estimate = search heuristic**
  - **a scoring function h(x)**

Best First Search (BestFS)

- Best First: always choose the path on the frontier with the smallest h value
  - *Frontier = priority queue ordered by h*
  - *Once reach goal can discard most unexplored paths…*
    - Why?
  - *Worst case: still explore all/most space*
  - *Best case: very efficient*
- **Greedy**: (only) expand path whose last node seems closest to the goal
  - *Get solution that is **locally** best*

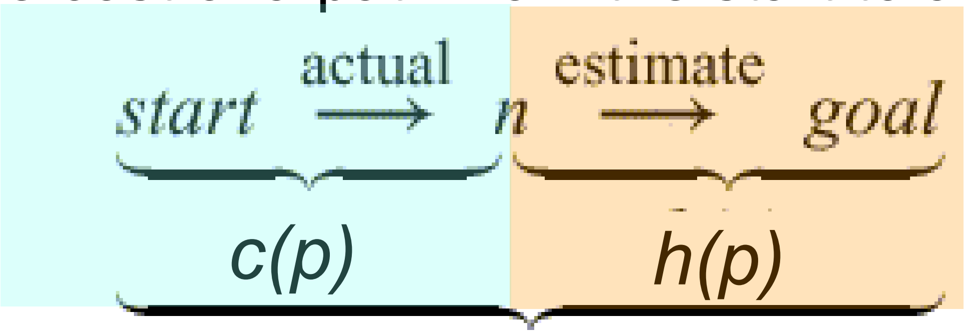
A* search



https://en.wikipedia.org/wiki/A*_search_algorithm

A* Search

- A* search takes into account both
  - *c(p)* = cost of path *p* to current node
  - *h(p)* = heuristic value at node *p (estimated "remaining" path cost)*
- Let $f(p) = c(p) + h(p)$.
  - *f(p)* is an estimate of the cost of a path from the start to a goal via *p*.



A* always chooses the path on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that path.

A* implementation

- 1. Initialize open and closed lists.
  - Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q’s “successors”, and set their parent nodes to q

A* implementation

- 1. Initialize open, closed lists. Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q's "successors", and set their parent nodes to q
  - For each successor u:
    - If successor is the goal, done!
    - $c(u) = c(q) + d(q,u)$
      $h(u) = D(goal, u)$
      $f(u) = c(u) + h(u)$
    - If successor u already exists in open list with lower f skip it
    - If successor already exists in closed list with lower f, skip it
    - Otherwise, add successor to open list

A* implementation

- 1. Initialize open, closed lists. Put starting node on open list.
- 2. While open list is not empty:
  - Find node with smallest f on the list, call it q
  - Pop q off of open list
  - Find q's "successors", and set their parent nodes to q
  - For each successor:
    - If successor is the goal, done!
    - g(successor) = g(q) + d(q,successor)
      h(successor) = d(goal, successor)
    - If successor already exists in open list with lower f, skip it
    - If successor already exists in closed list with lower f, skip it
    - Otherwise, add successor to open list
  - **Put q on closed list**

Variants

- ***Randomness***
- ***Make the AI dump/non-perfect***
  - *How?*
- ***Different terrain types?***

Overview

First half:

- ***Shortest paths cont.***
- ***Two-player games***

… all about traversing trees efficiently

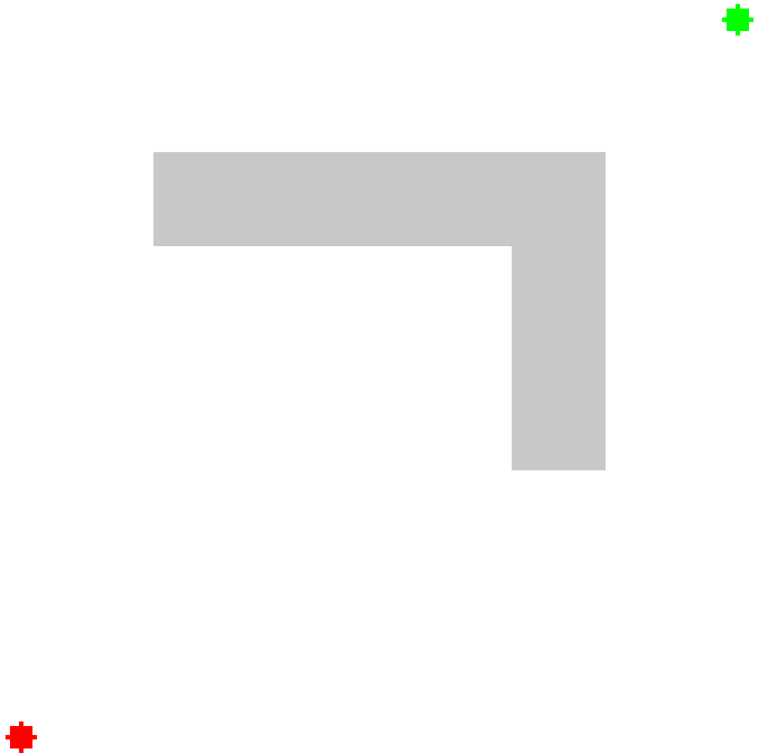
+ Some debugging tips

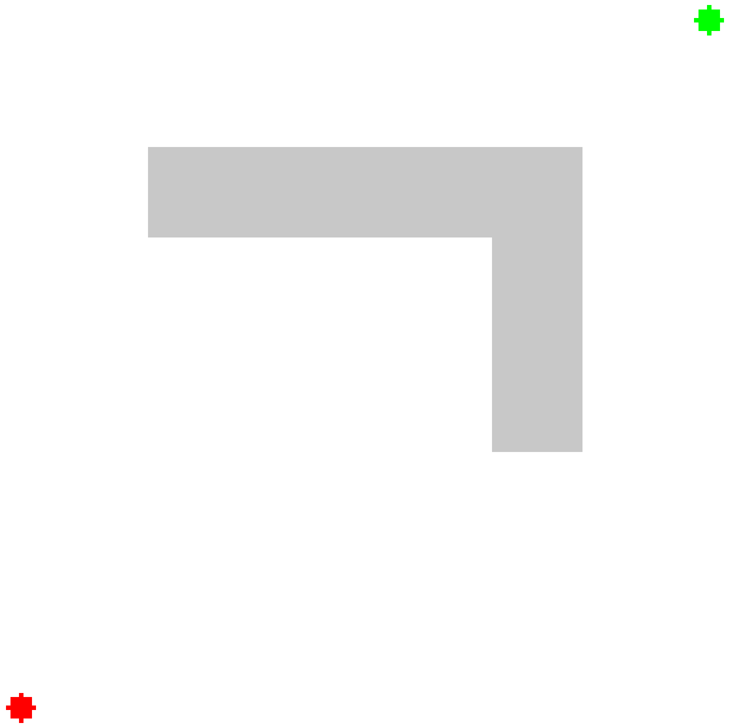
Second half:

- ***Physical simulation basics***
  - *setting and definitions*
- ***Efficient & precise simulation***
  - *today: what can go wrong?*

End of the day: ***be able to implement efficient shortest path, two-player AI, and to simulate flying pebbles (for A3!)***

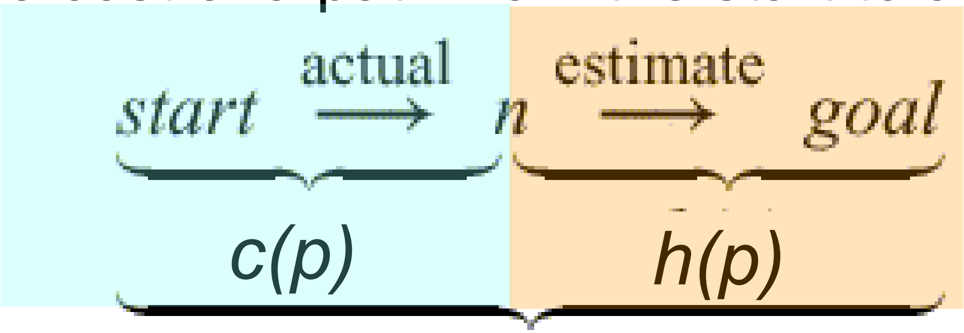
Breadth-first vs. A*





A* Search

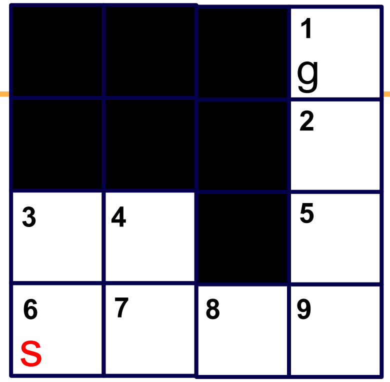
- A* search takes into account both
  - *c(p)* = cost of path *p* to current node
  - *h(p)* = heuristic value at node *p (estimated "remaining" path cost)*
- Let *f(p)* = *c(p)* + *h(p).*
  - *f(p)* is an estimate of the cost of a path from the start to a goal via *p*.



A* always chooses the path on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that path.

A* Example

Frontier (open list)



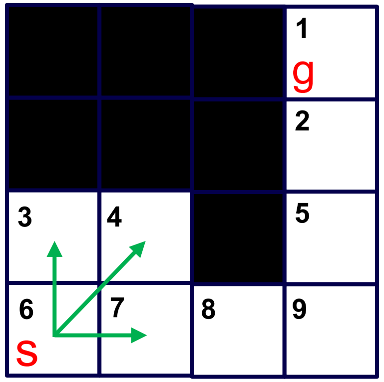
Init:

- ***Put starting node on open list:*** Lo = {6}
- ***Set its cost to 0:*** c[6] = 0
- ***Set closed list to empty list:*** Lc = {}

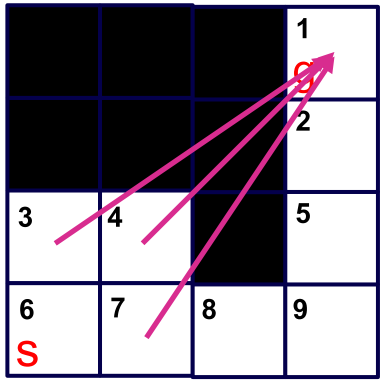
Step 1:

- ***Find node with smallest f on the list, call it q:*** q = 6
- ***Find q's "successors":*** sucs = {3,4,7}
- ***For each successor u:*** for u in sucs …
  - c(u) = c(q) + d(q,u)   c[3] = c[6] + 1 = 1
c[4] = c[6] + 1.4 = 1.4
c[7] = c[6] + 1 = 1
  - h(u) = d(g, u)
f(u) = c(u) + h(u)   h[3] = 3.6
h[4] = 2.8
h[7] = 3.6   f[3] = c[3] + h[3] = 4.6
f[4] = c[4] + h[4] = 4.2
f[7] = c[7] + h[7] = 4.6
- ***add successors to open list and move q to closed:***
Lo = {3,4,7}; Lc = {6}

Step cost c



Heuristic dist. h

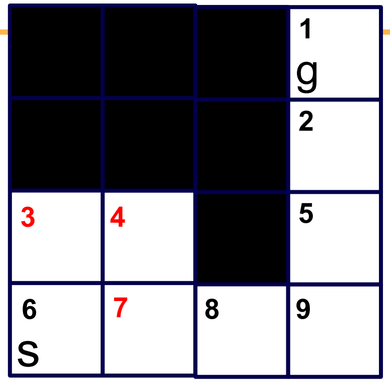


A* Example

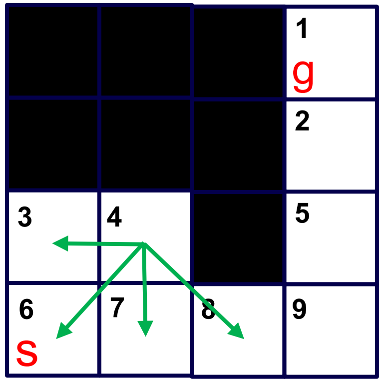
Step 2: Lo = {3,4,7}; Lc = {6}

- ***Find node with smallest f on Lo, call it q:***
  - f[3] = 4.6
    f[4] = 4.2 -> q = 4
    f[7] = 4.6
- ***Find q's "successors":*** sucs = {3,6,7,8}
- for u in sucs…
  - c_tmp[3] = c[4] + 1 = 2.4 > c[3] = 1, skip
    c_tmp[6] = c[4] + 1.4 = 2.8 > c[6] = 0, skip
    c_tmp[7] = c[4] + 1 = 2.4 > c[7] = 1, skip
    c_tmp[8] = c[4] + 1.4 = 2.4 not in Lo or Lc, select c[8] = c_tmp[8]
  - Update heuristic and estimated cost f:
    h[8] = 3.2
    f[8] = c[8] + h[8] = 5.6
- ***add successors to open list and move q to closed list:***
  Lo = {3,7,8}; Lc = {6,4}

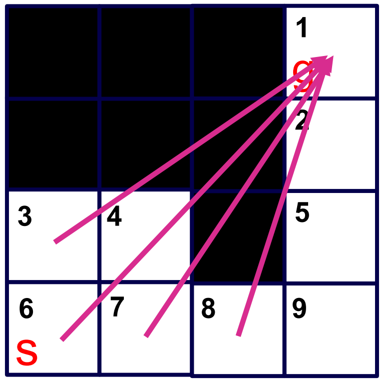
Frontier (open list)



Step cost c



Heuristic dist. h

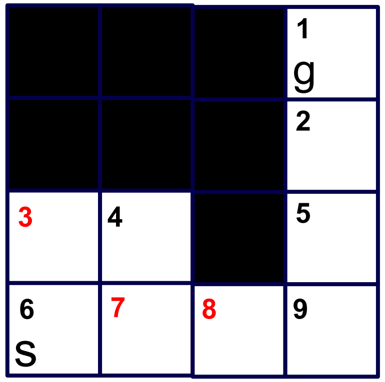


A* Example

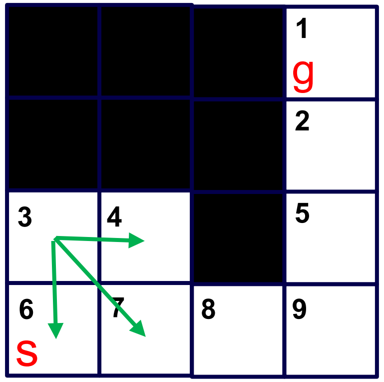
Step 3: Lo = {3,7,8}; Lc = {6,4}

- ***Find node with smallest f on Lo, call it q:***
  - f[3] = 4.6 -> q = 3
    f[7] = 4.6
    f[8] = 5.6
- ***Find q's "successors":*** sucs = {4,6,7}
- for u in sucs…
  - c_tmp[4] = c[3] + 1 = 2 > c[4] = 1.4, skip
    c_tmp[6] = c[3] + 1.4 = 2.4 > c[6] = 0, skip
    c_tmp[7] = c[3] + 1 = 2 > c[7] = 1, skip
- ***add successors to open list? no successors!***
- ***move q to closed list:***
  Lo = {7,8};
  Lc = {6,4,3}

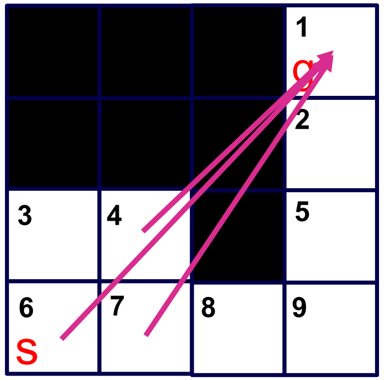
Frontier (open list)



Step cost c



Heuristic dist. h

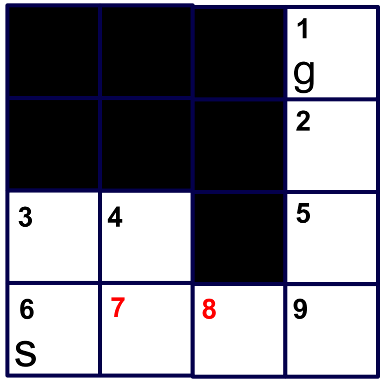


A* Example

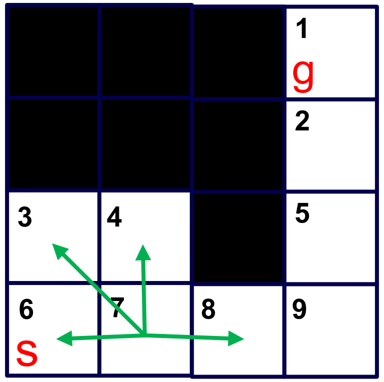
Step 4: Lo = {7,8}; Lc = {6,4,3}

- ***Find node with smallest f on Lo, call it q:***
  - f[7] = 4.6   ->   q = 7
    f[8] = 5.6
- ***Find q's "successors":*** sucs = {3,4,6,8}
- **for u in sucs…**
  - c_tmp[3] = c[7] + 1.4 = 2.4   >   c[3] = 1, skip
    c_tmp[4] = c[7] + 1 = 2   >   c[4] = 1, skip
    c_tmp[6] = c[7] + 1 = 2   >   c[6] = 0, skip
    c_tmp[8] = c[7] + 1 = 2   >   c[8] = 2.4, select new c[8] = 2
- ***add successors to open list? Already there!***
- ***move q to closed list:***
  Lo = {8};
  Lc = {6,4,3,7}

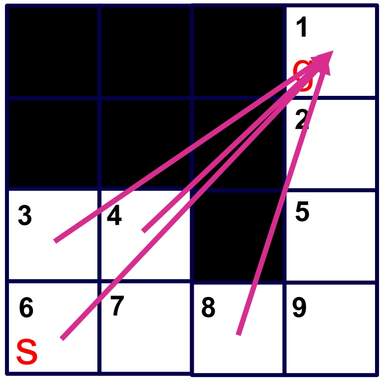
Frontier (open list)



Step cost c



Heuristic dist. h

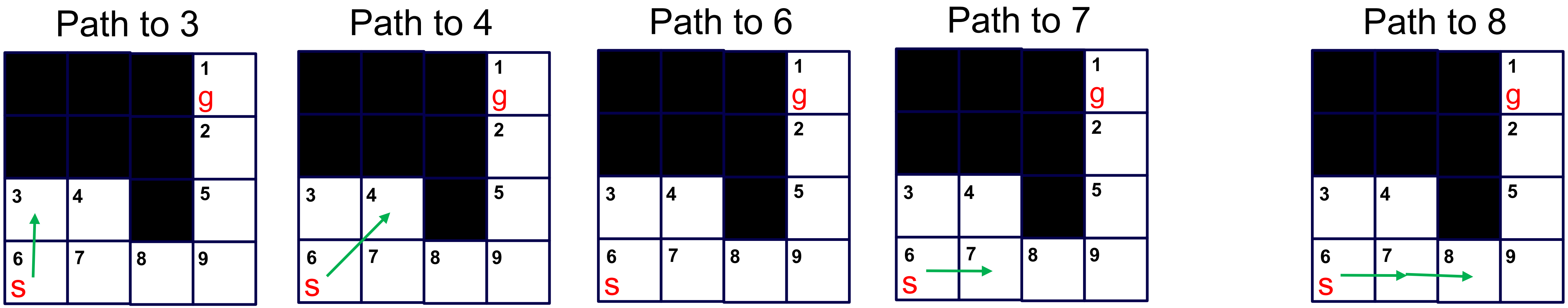


Keep track of your parents

- ***We neglected parent-child relation in previous slides…***

Lc = {6,4,3}

Lo = {8};



- ***Note, closed paths have no 'free' neighbors***
  - impassable or already visited from a shorter path

A* search

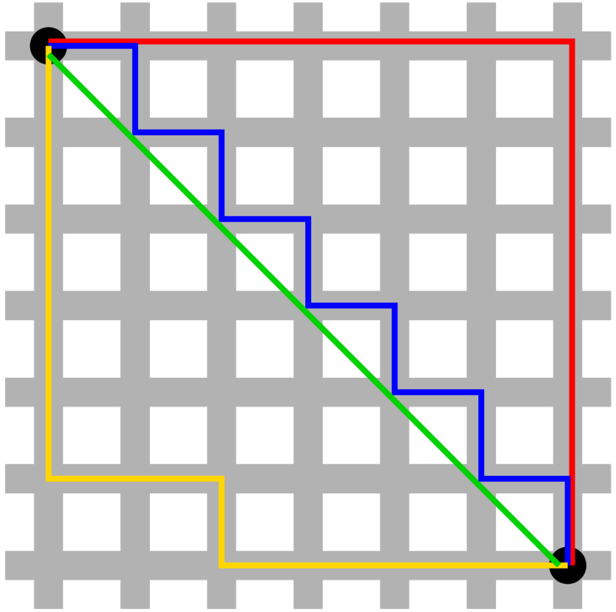
Key idea: H is a heuristic, and not the real distance:

h(p,q) = |(p.x – q.x)| + |(p.y – q.y)|

- Manhattan distance

h(p,q) = sqrt((p.x – q.x)^2 + (p.y – q.y)^2)

- Euclidean distance



https://en.wikipedia.org/wiki/Taxicab_geometry

Conditions:

- a heuristic function is **admissible** if it never overestimates the cost of reaching the goal
- a heuristic function is said to be **consistent**, or **monotone**, if its estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour