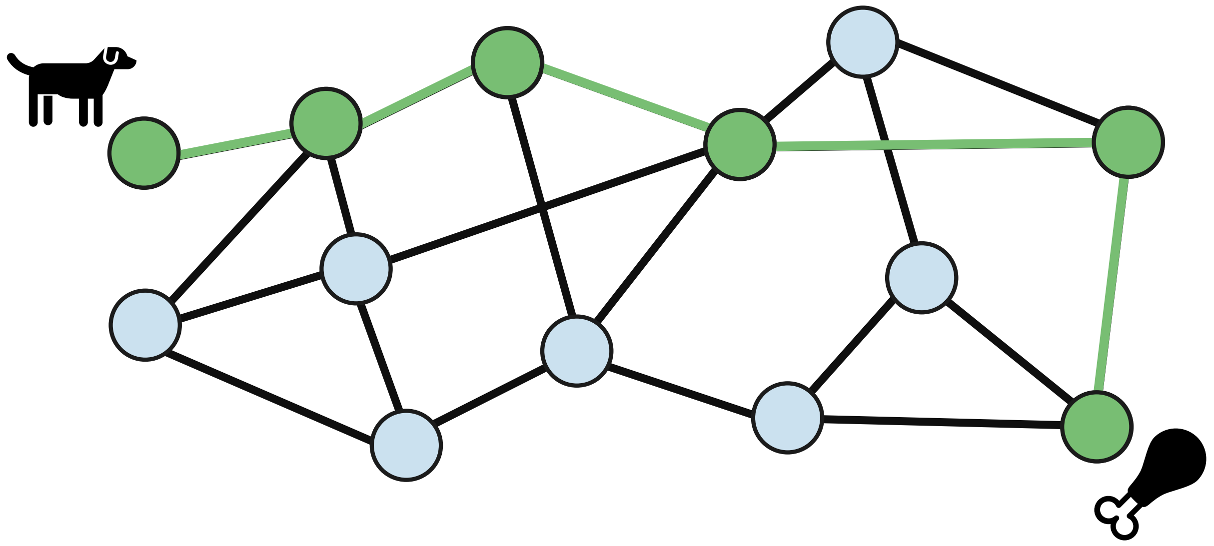


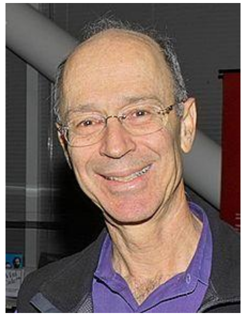
A*-Algorithm

Yannic Maus

Agenda

A*-algorithm was introduced in 1968:

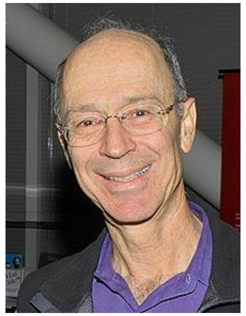
Peter E. Hart, Nils, J. Nilsson, Bertram Raphael –
*A **Formal** Basis for the **Heuristic** Determination of **Minimum Cost Paths***



Agenda

A*-algorithm was introduced in 1968:

Peter E. Hart, Nils, J. Nilsson, Bertram Raphael –
*A **Formal** Basis for the **Heuristic** Determination of **Minimum Cost Paths***

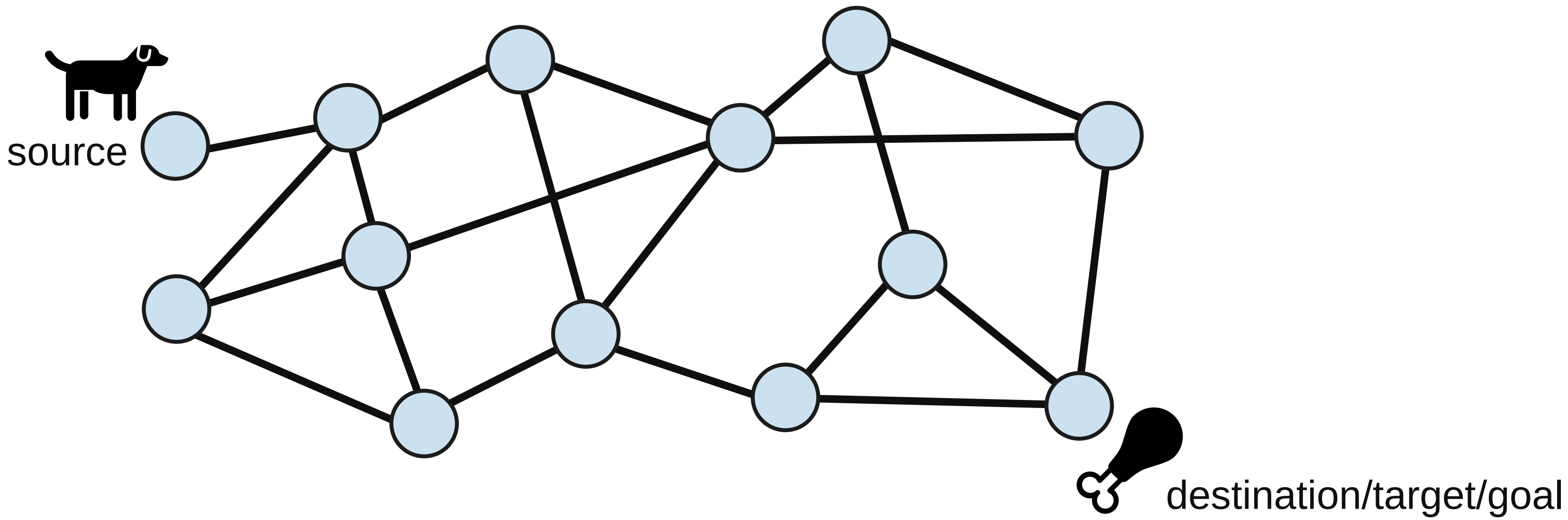


- Why do we need another shortest path algorithm?

  There are many shortest path algorithms: BFS, Dijkstra, Floyd Warshall, …
- The A*-Algorithm and its heuristic (proof of correctness)
- Advantages/Disadvantages

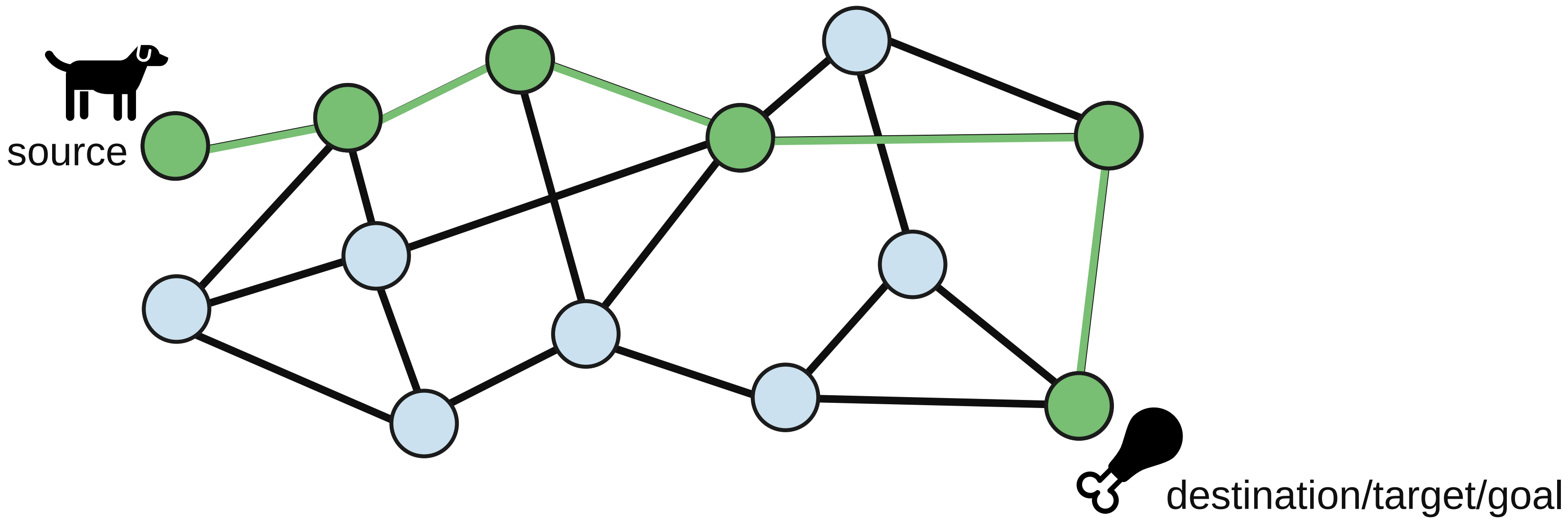
A*-Algorithm, yet another shortest path algorithm?

A* = A star is a **shortest path algorithm**



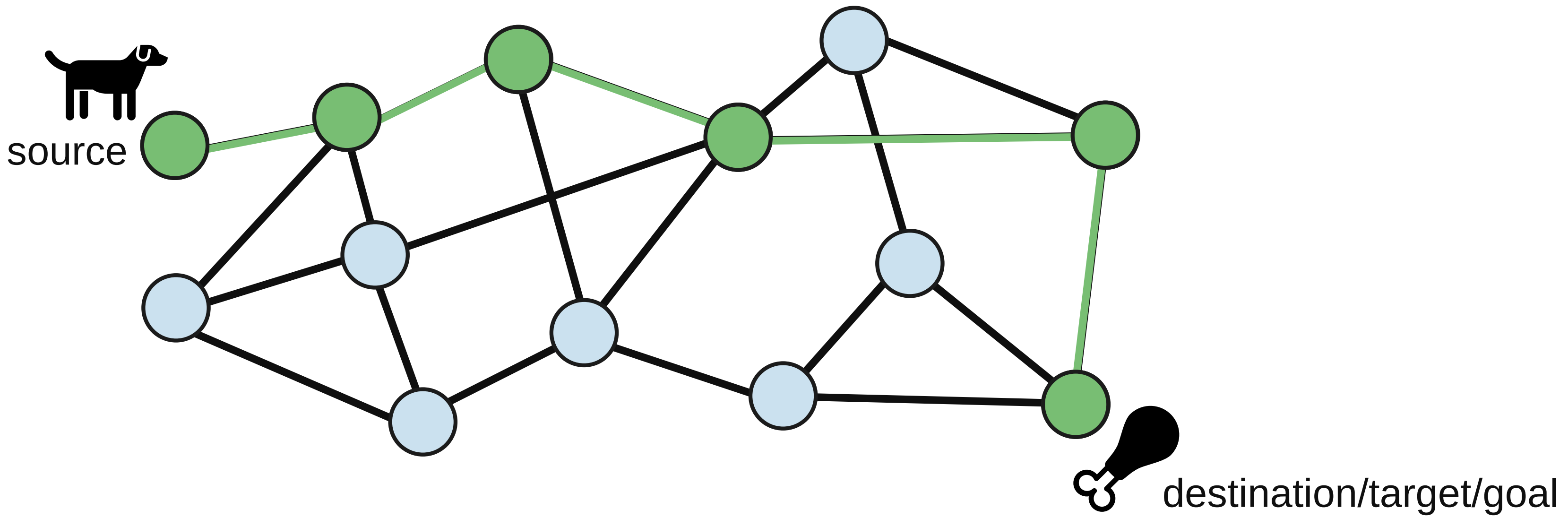
A*-Algorithm, yet another shortest path algorithm?

A* = A star is a **shortest path algorithm**



A*-Algorithm, yet another shortest path algorithm?

A* = A star is a **shortest path algorithm**

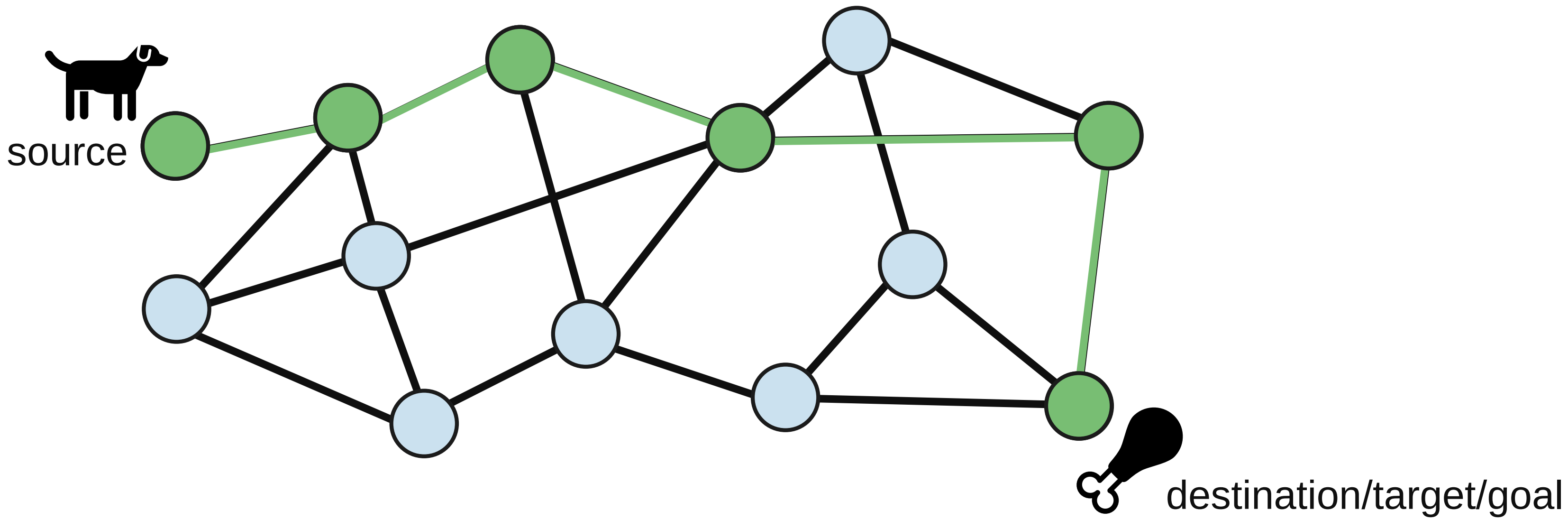


There are many shortest path algorithms

- **BFS:** unweighted edges
- **Dijkstra:** weighted edges
- **Floyd Warshall:** weighted edges
- **Bellman-Ford:** weighted edges, even negative weights

A*-Algorithm, yet another shortest path algorithm?

A* = A star is a **shortest path algorithm**



There are many shortest path algorithms

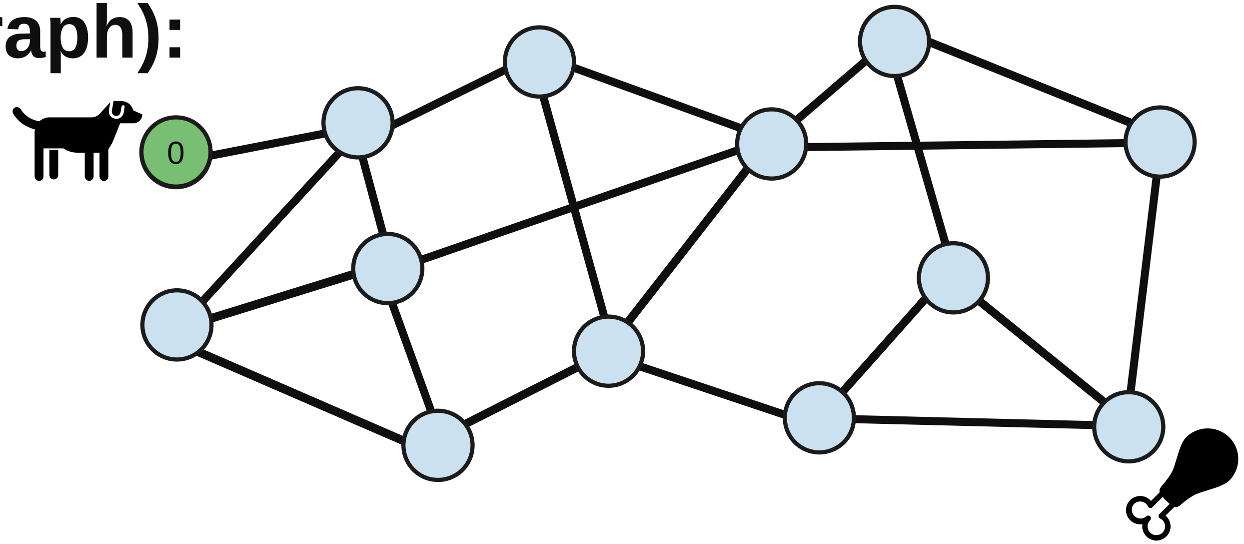
- **BFS:** unweighted edges
- **Dijkstra:** weighted edges

Recap (A* is more involved but similar)

- **Floyd Warshall:** weighted edges
- **Bellman-Ford:** weighted edges, even negative weights

Breadth First Search (unweighted graph):

Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:

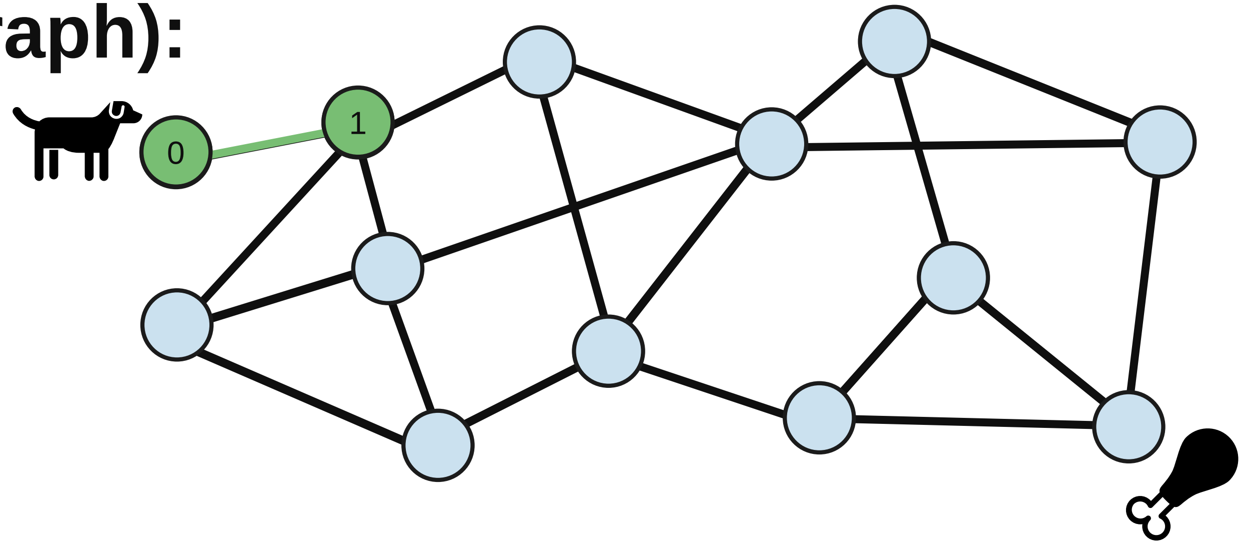


guaranteed shortest hop path.

Breadth First Search (unweighted graph):

Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:

guaranteed shortest hop path.

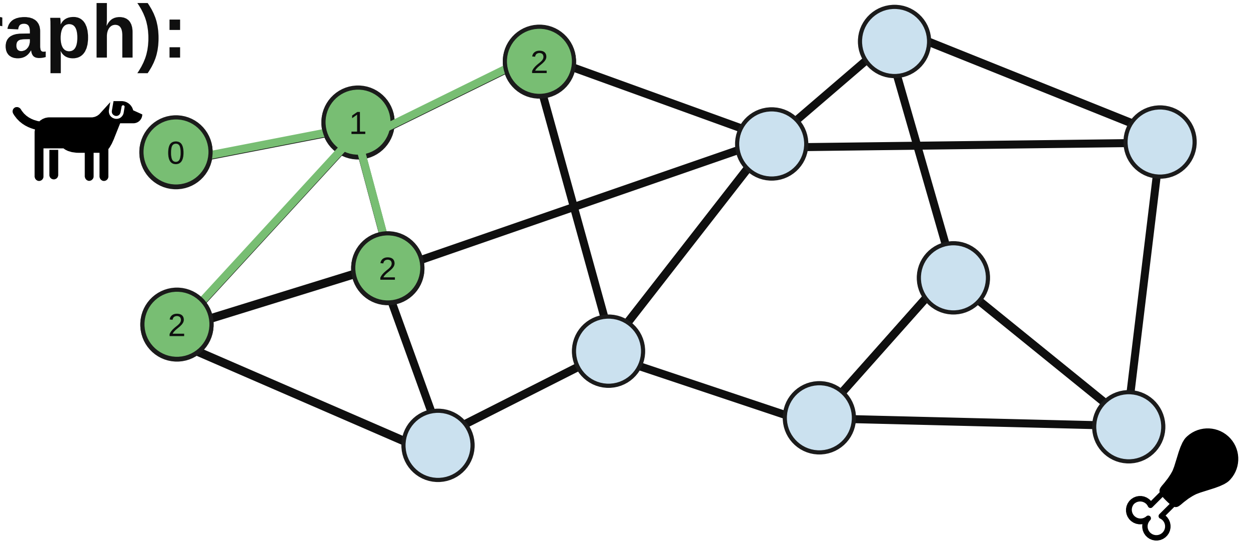


BFS

Breadth First Search (unweighted graph):

Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:

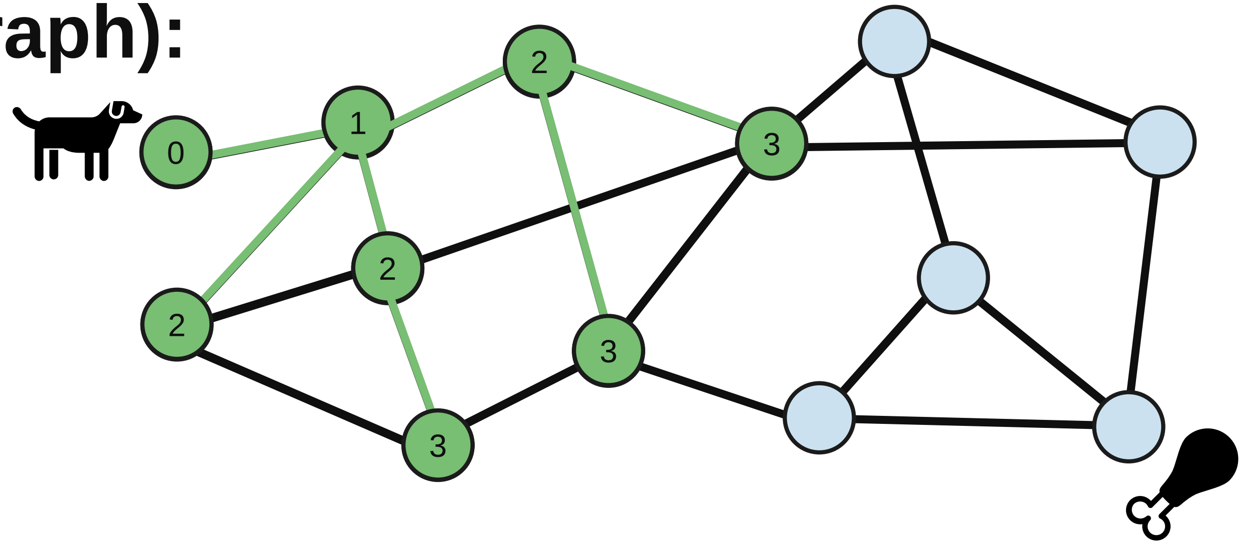
guaranteed shortest hop path.



Breadth First Search (unweighted graph):

Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:

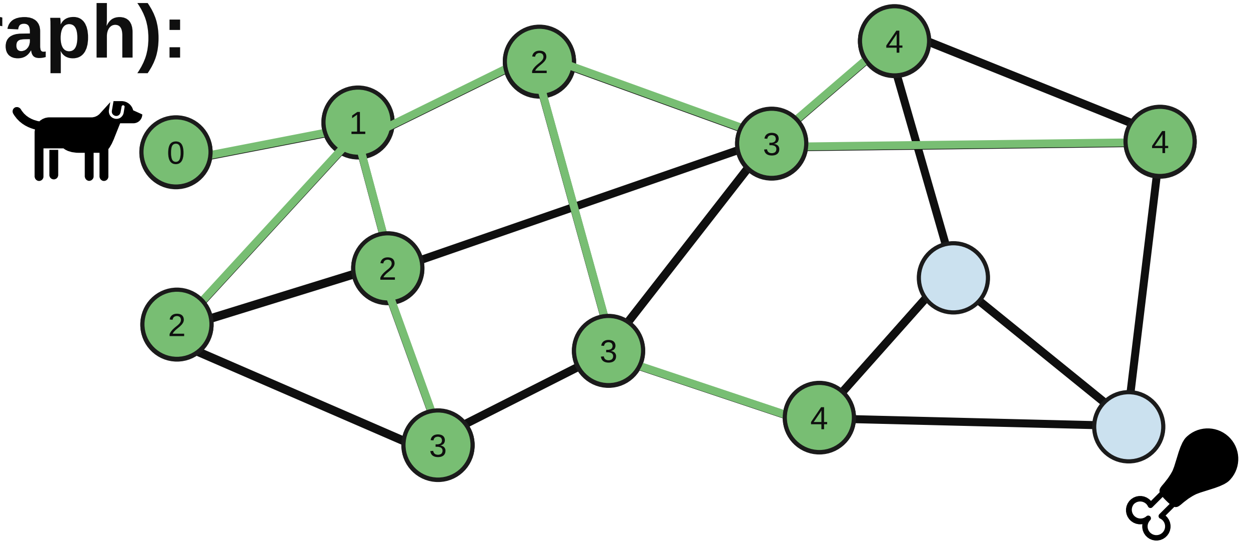
guaranteed shortest hop path.



BFS

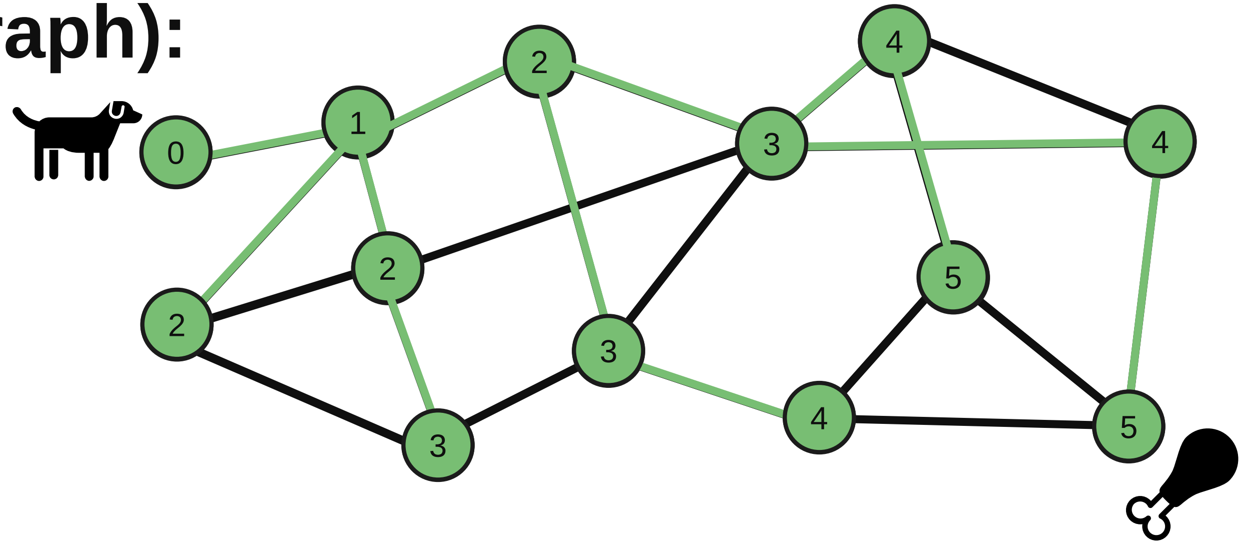
Breadth First Search (unweighted graph):
Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:

guaranteed shortest hop path.



Breadth First Search (unweighted graph):

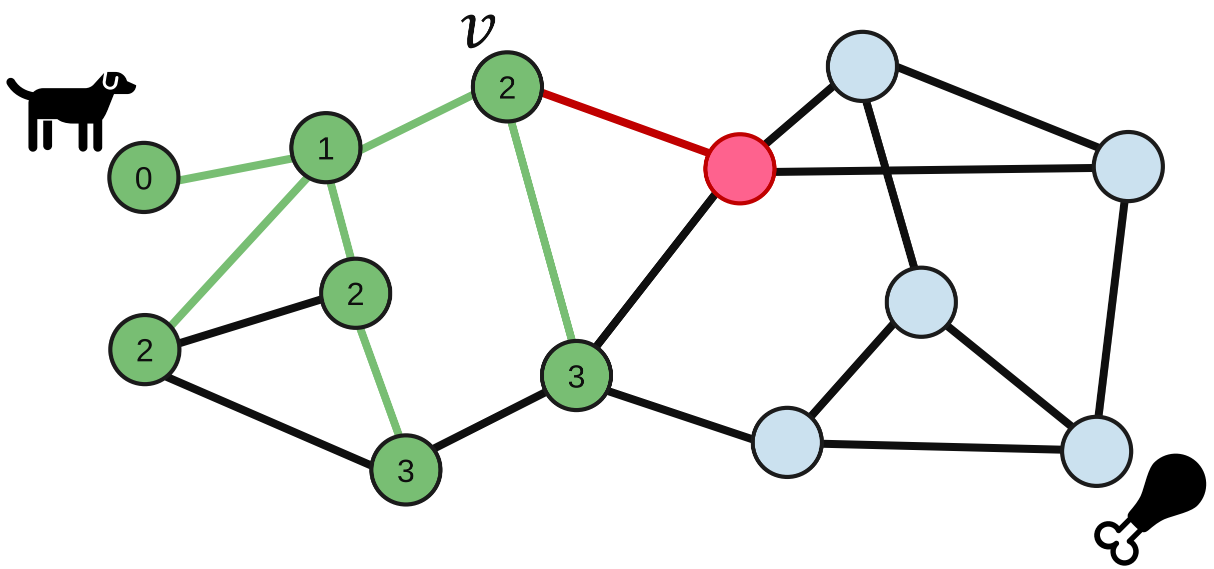
Look at paths of length 1,
then length 2,
then length 3,
etc.,
until a path is found:



guaranteed shortest hop path.

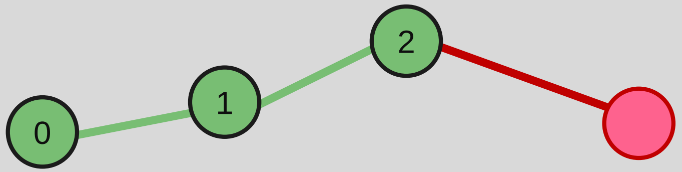
Zoom into BFS

- If we use BFS to find the path (disregarding weights), we would use **queue** to enqueue each path



BFS

```
Expand v and queue this path
```

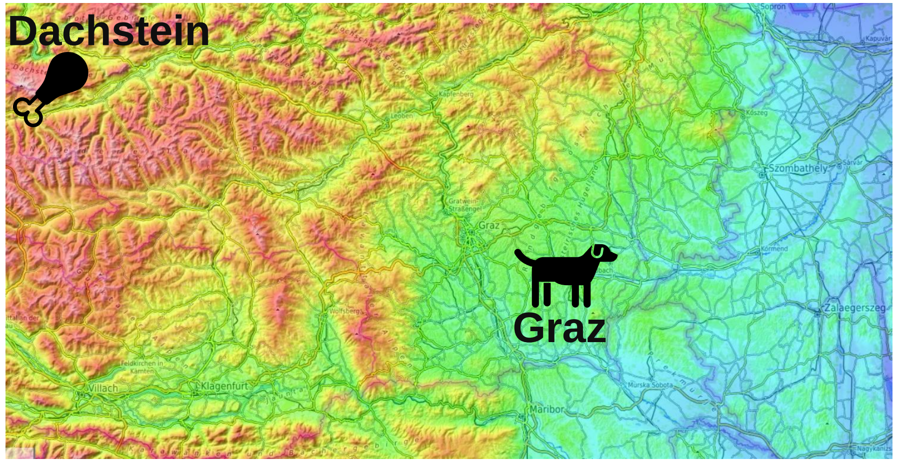


BFS: A Queue of paths

```
BFS from s to t:
create a queue of paths (a vector), q
q.enqueue(s (path))
while q is not empty and t is not yet visited:
    path = q.dequeue()
    v = last element in path
    mark v as visited
    if v==t then stop
    for each unvisited neighbor of v:
        make new path with v's neighbor as last element
        q.enqueue(new path)
```

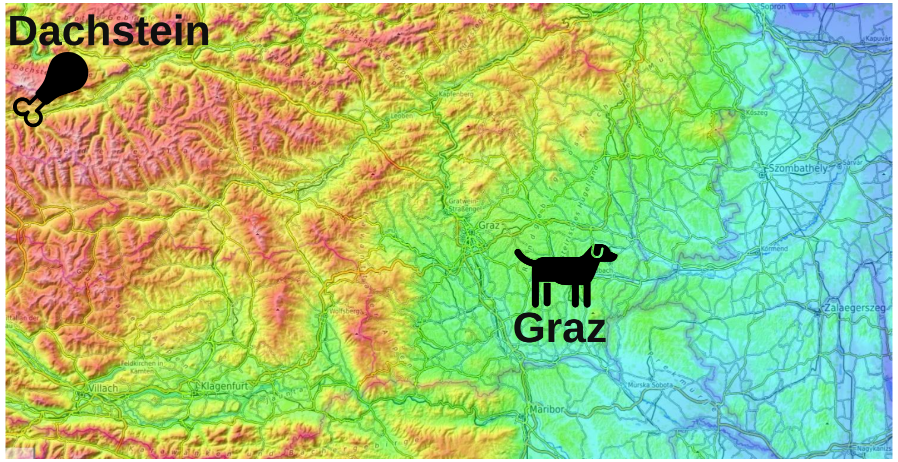
QUEUE = FIFO

BFS is often not enough …

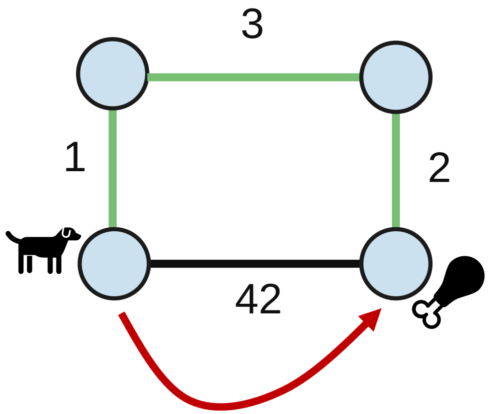


(we want weights)

BFS is often not enough …



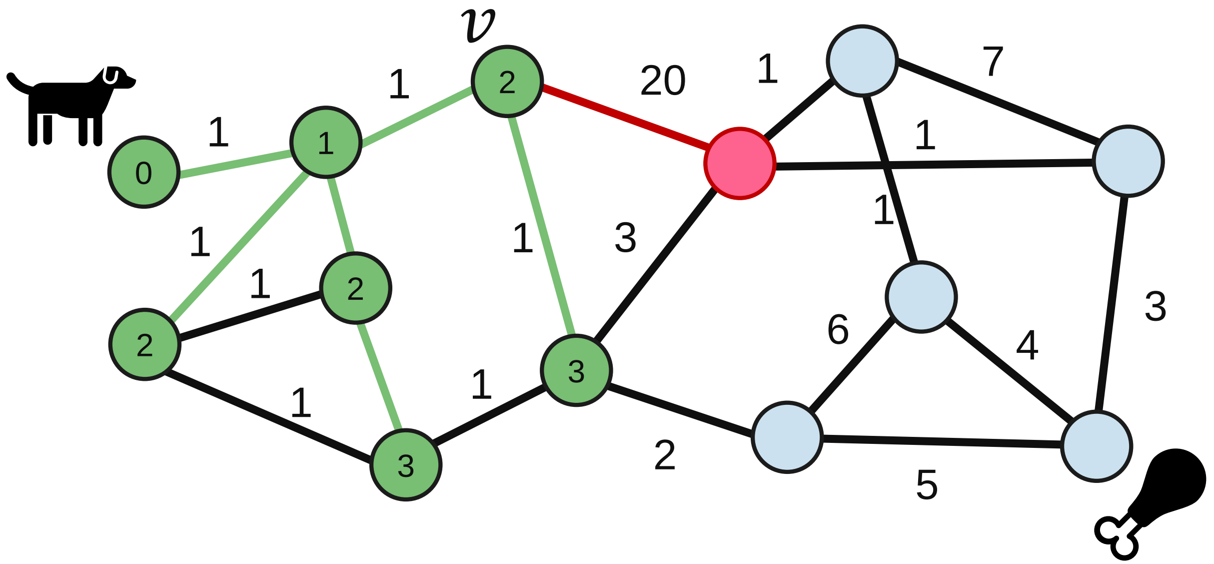
(we want weights)



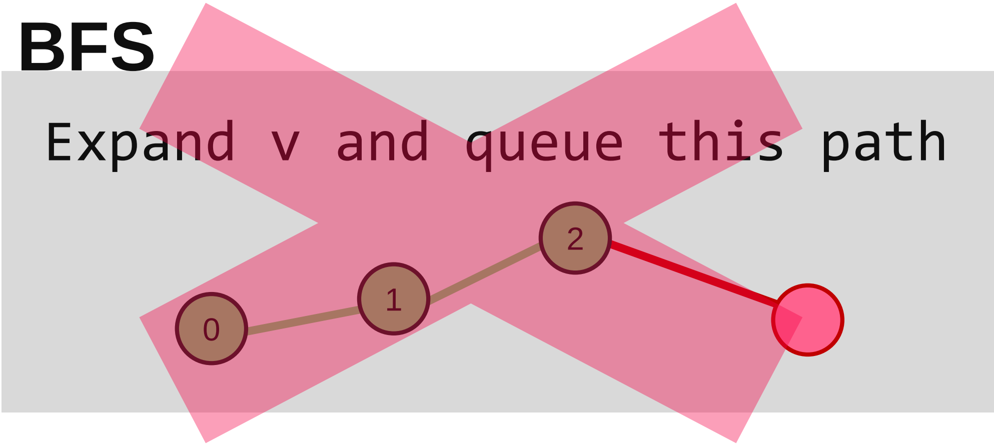
BFS does not work with weights

Dijkstra for weighted graphs

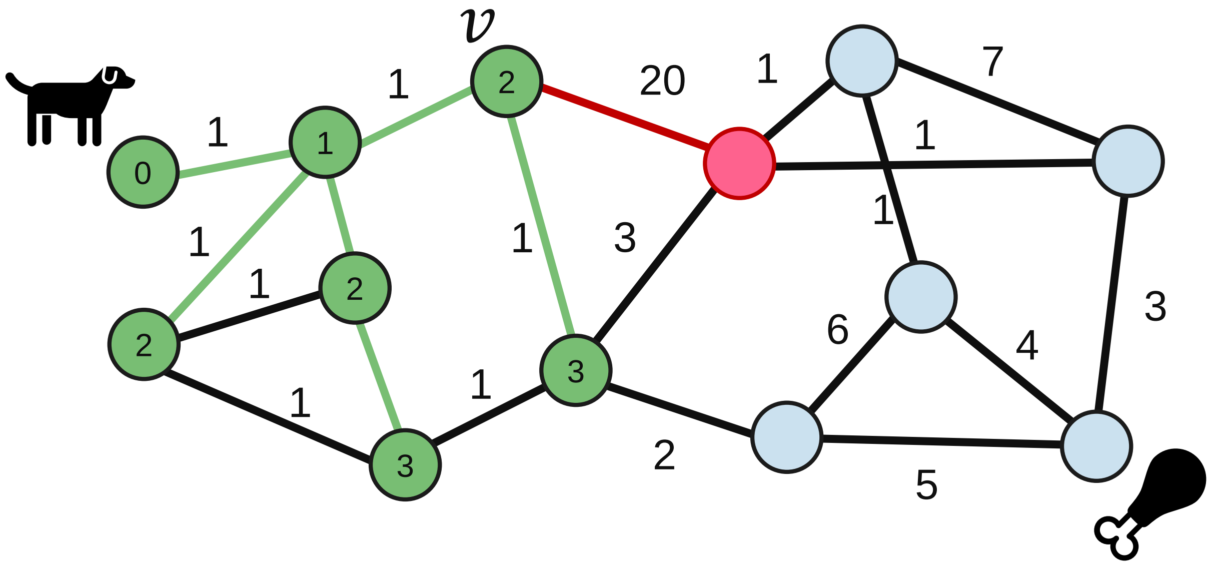
- If we use BFS to find the path (disregarding weights), we would use **queue** to enqueue each path



BFS

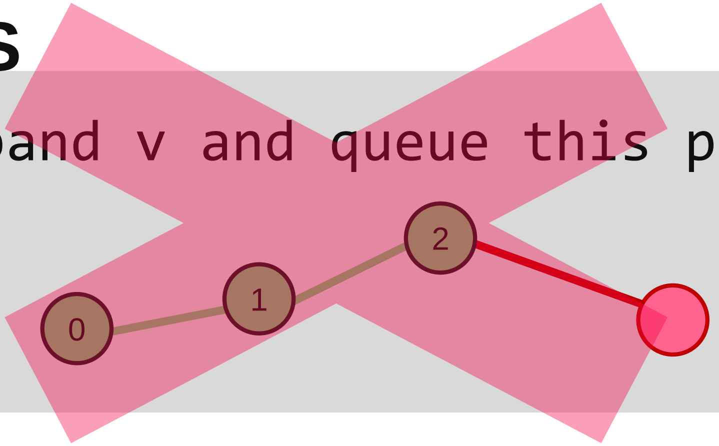


- If we use BFS to find the path (disregarding weights), we would use **queue** to enqueue each path



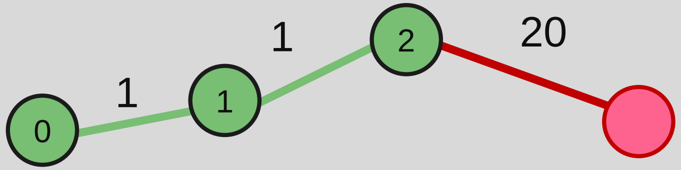
BFS

Expand v and queue this path



Dijkstra

Expand v and queue this path



Cost of path 22!

- "**Dijkstra's Algorithm**" uses a **priority queue** to enqueue each path.

Dijkstra: A **Priority Queue** of paths

```
Dijkstra from s to t:
create a priority queue of paths (a vector), q
q.enqueue(s (path))
while q is not empty and t is not yet visited:
    path = q.dequeue() //order determined by costs of path
    v = last element in path
    mark v as visited
    if v==t then stop
    for each unvisited neighbor of v:
        make new path with v's neighbor as last element //costs!
        q.enqueue(new path)
```

BFS: A Queue of paths

BFS from s to t:

create a ~~priority~~ queue of paths (a vector), q

q.enqueue(s (path))

while q is not empty and t is not yet visited:

    path = **q.dequeue()** ~~//order determined by costs of path~~

    v = last element in path

    mark v as visited

    if v==t then stop

    for each unvisited neighbor of v:

        make new path with v's neighbor as last element ~~//costs!~~

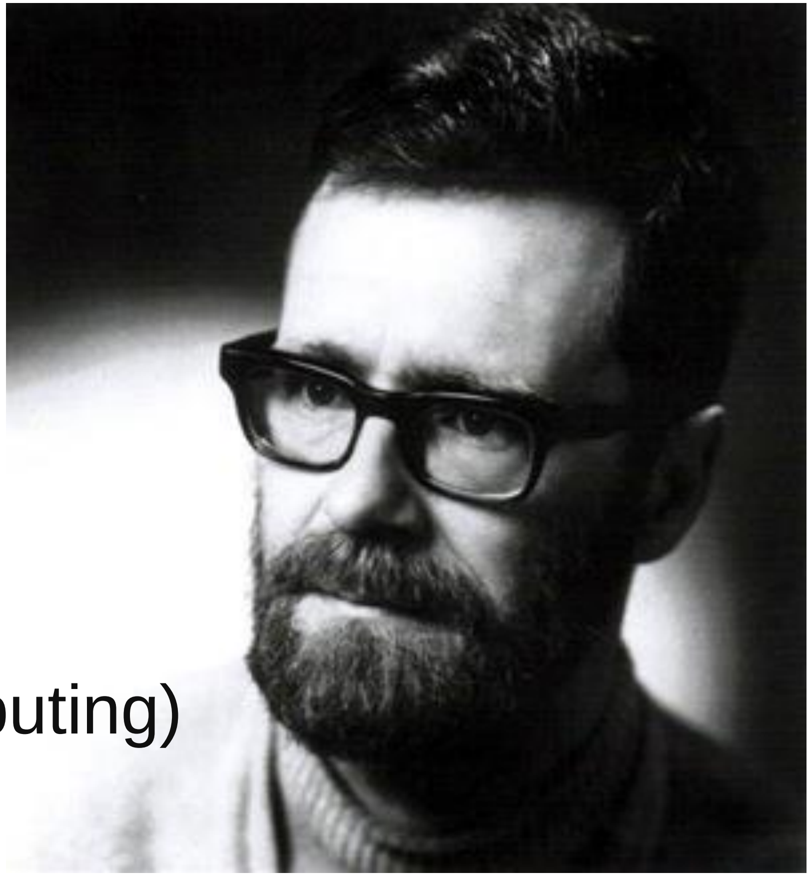
        q.enqueue(new path)

Dijkstra is Greedy

Dijkstra's algorithm is what we call a **"greedy"** algorithm.

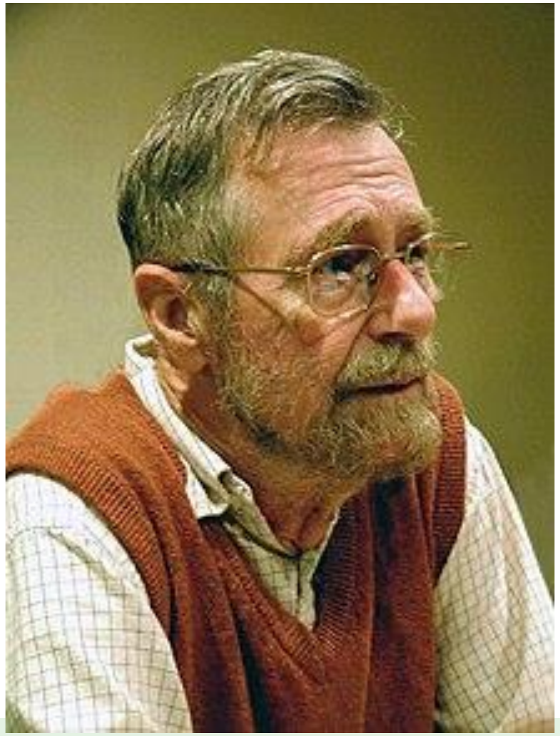
Take what’s **best at the given time,** and it finds an **optimal solution**!
(for many problems greedy algorithms don’t give optimal solutions)

Edsgar Dijkstra was very influential



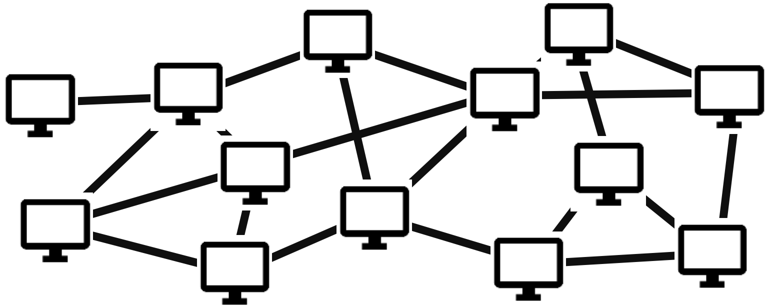
Countless contributions to computer science

1930-2002



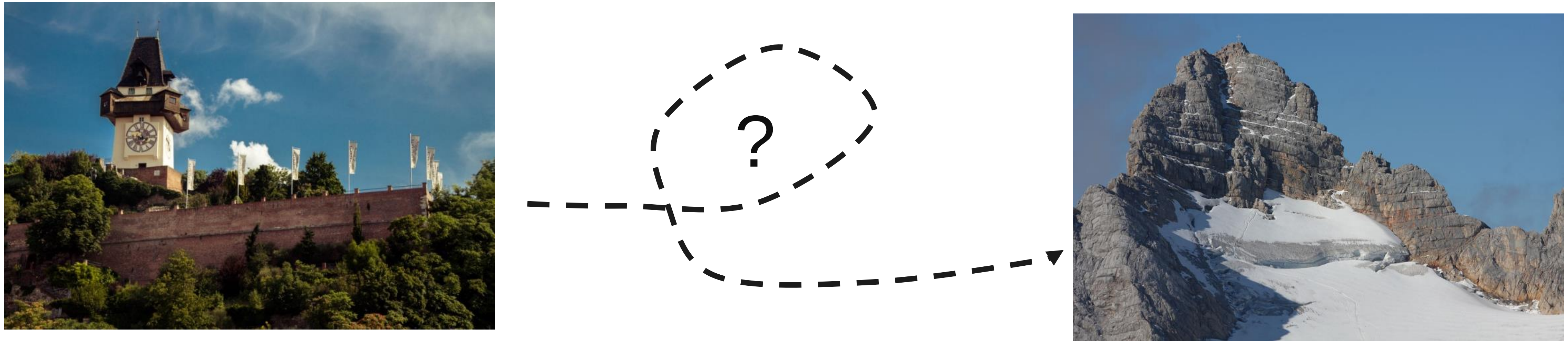
Dijkstra Award (in Distributed Computing)

- algorithms in large networks
- massively parallel computing (MapReduce by google, ..)
- blockchain, …
- …



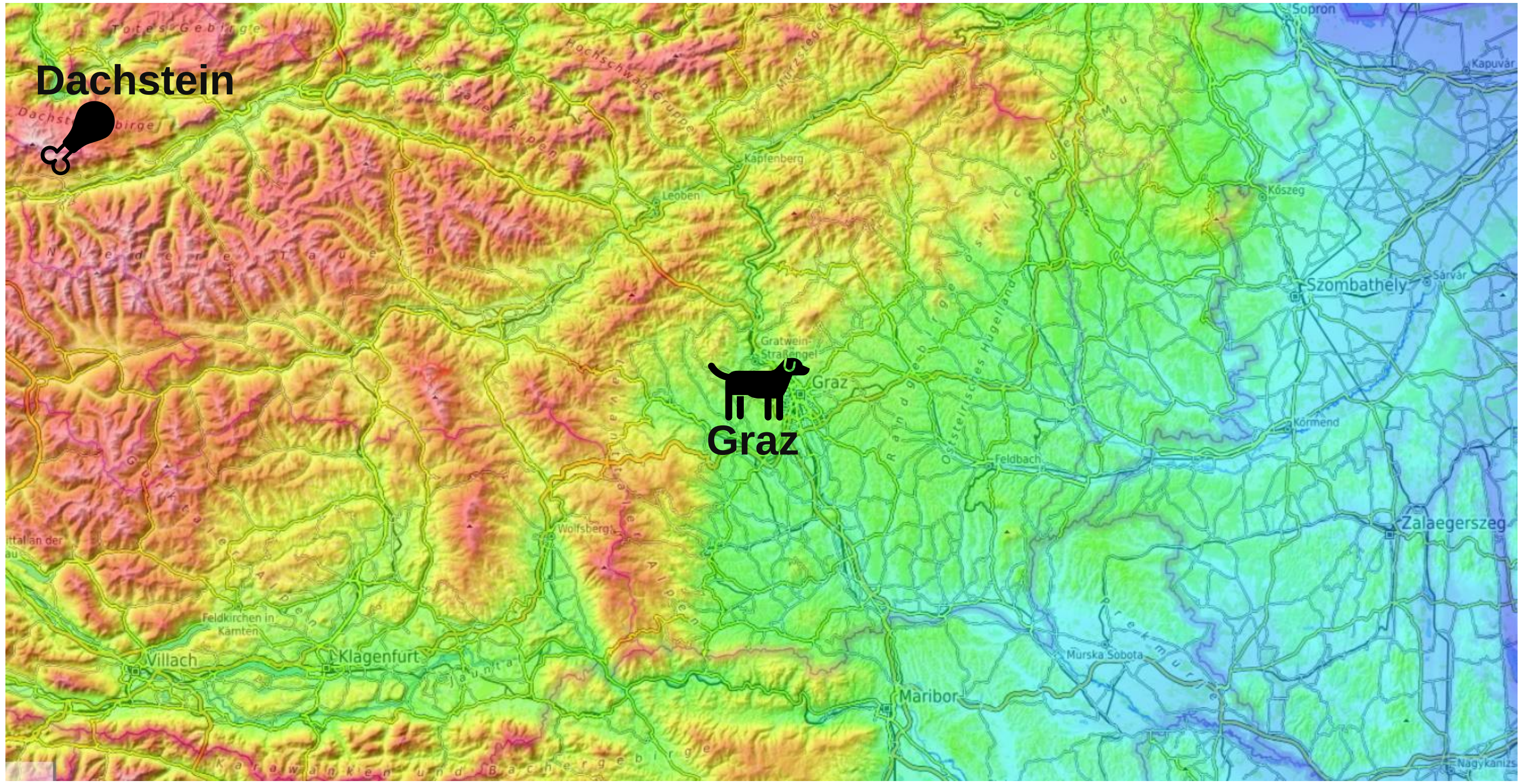
Dijkstra's is great

but we can do better?



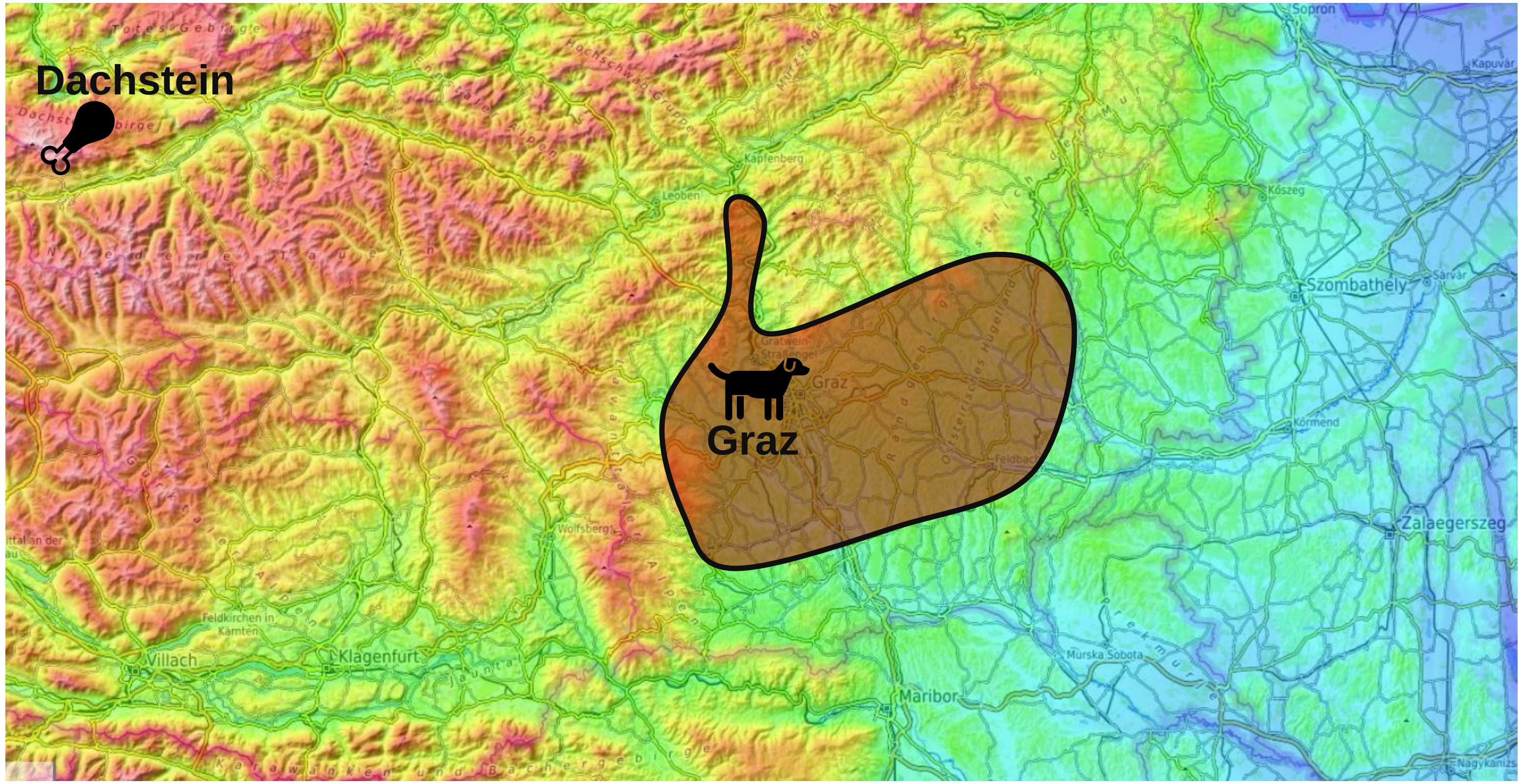
Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



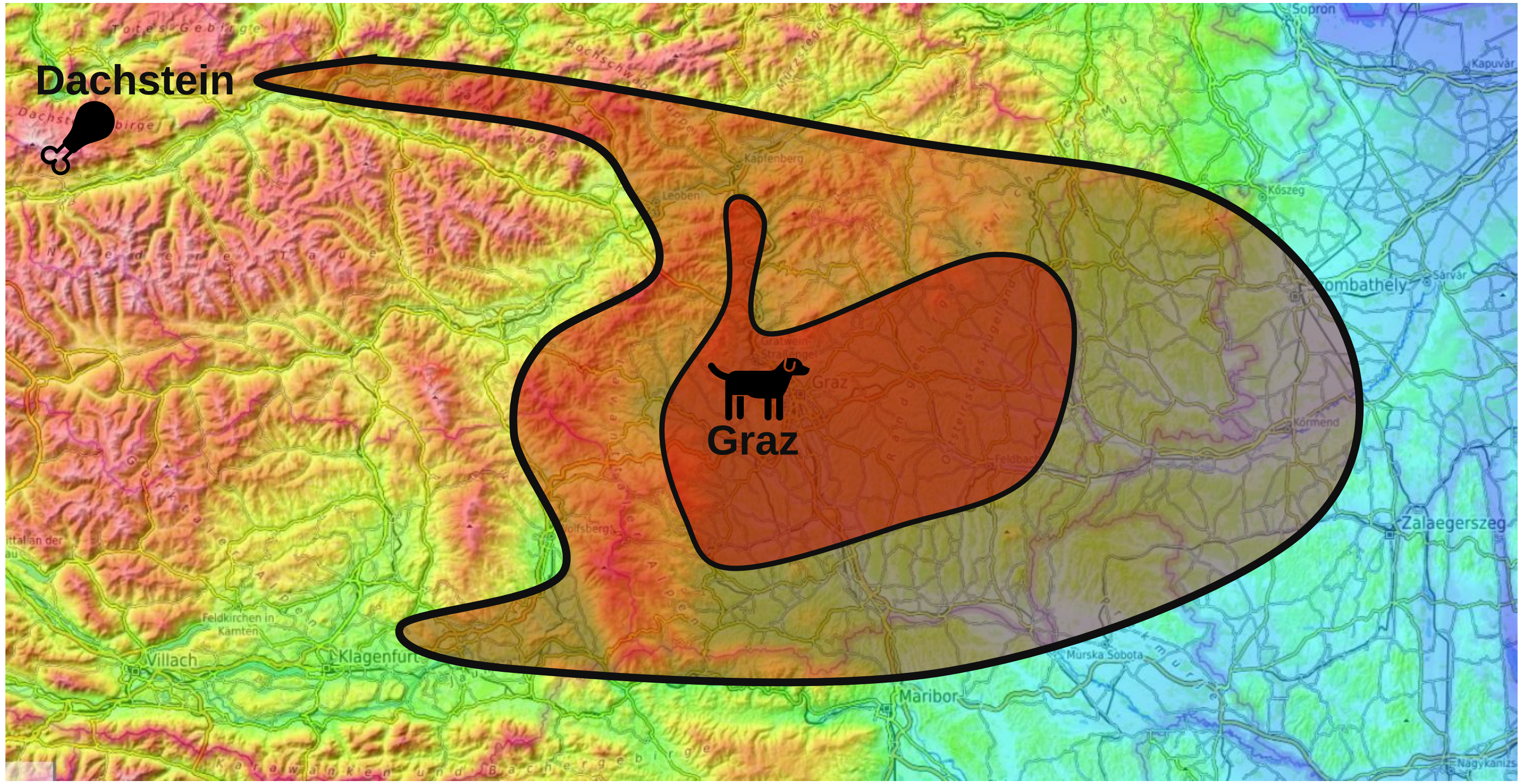
Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



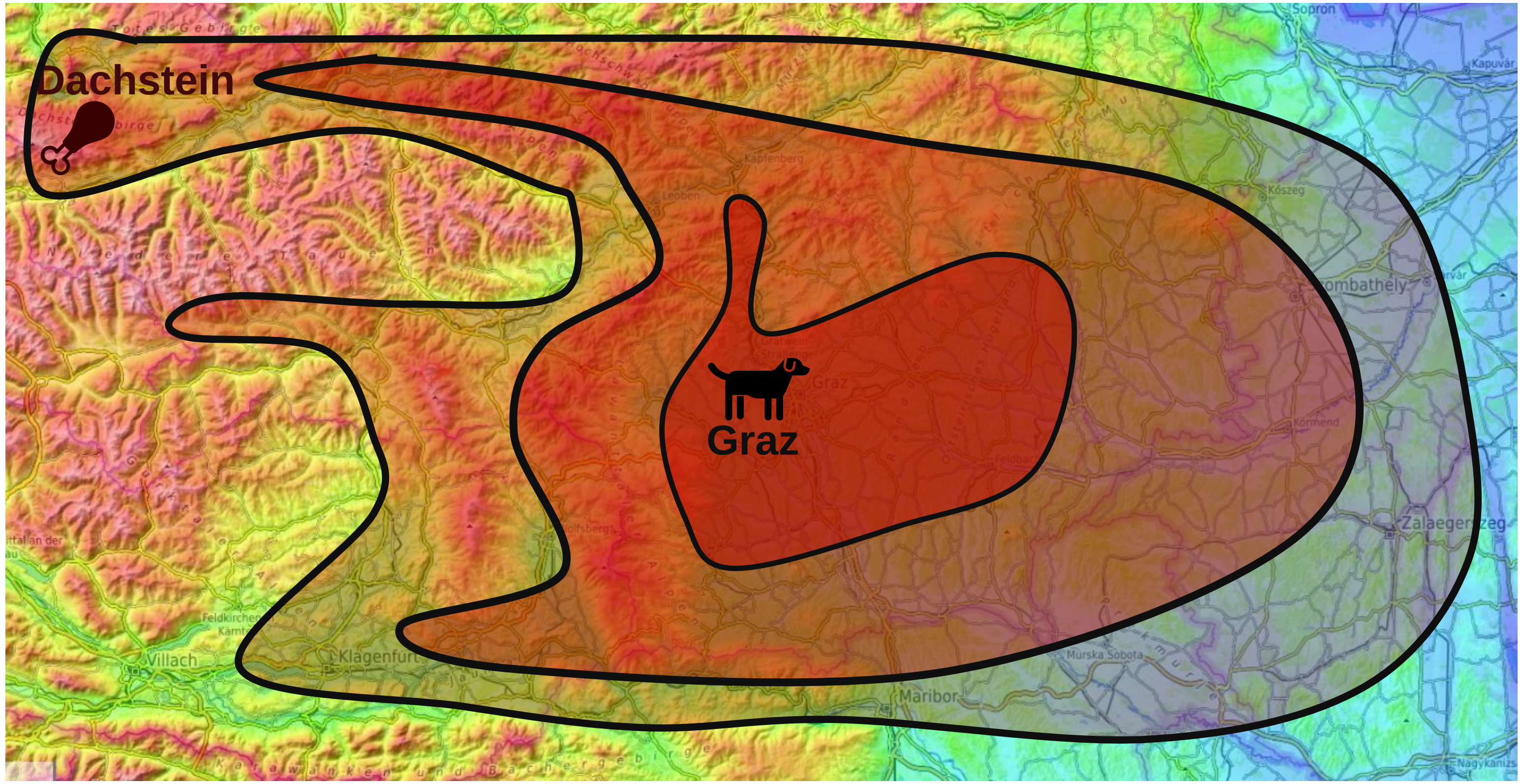
Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



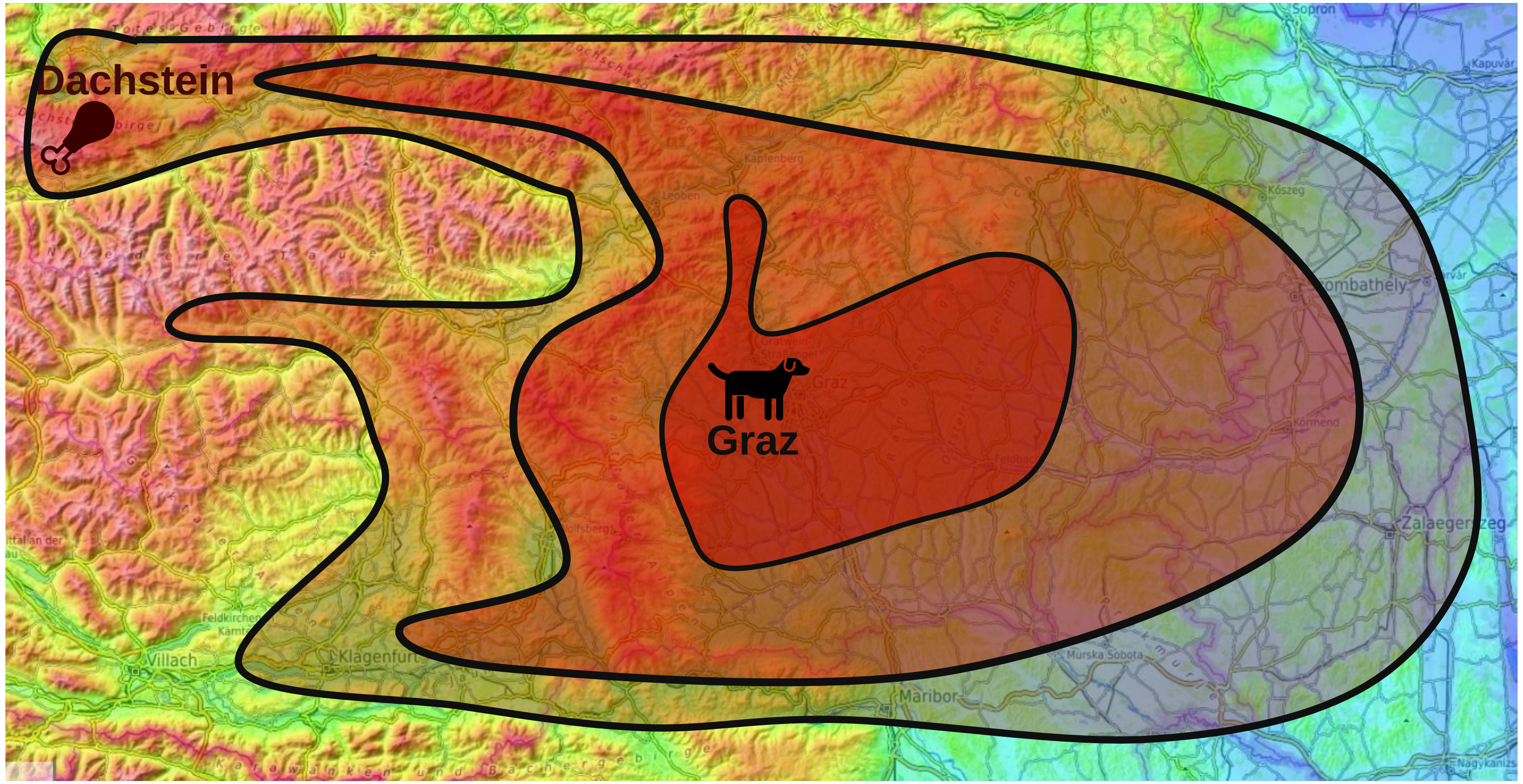
Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.

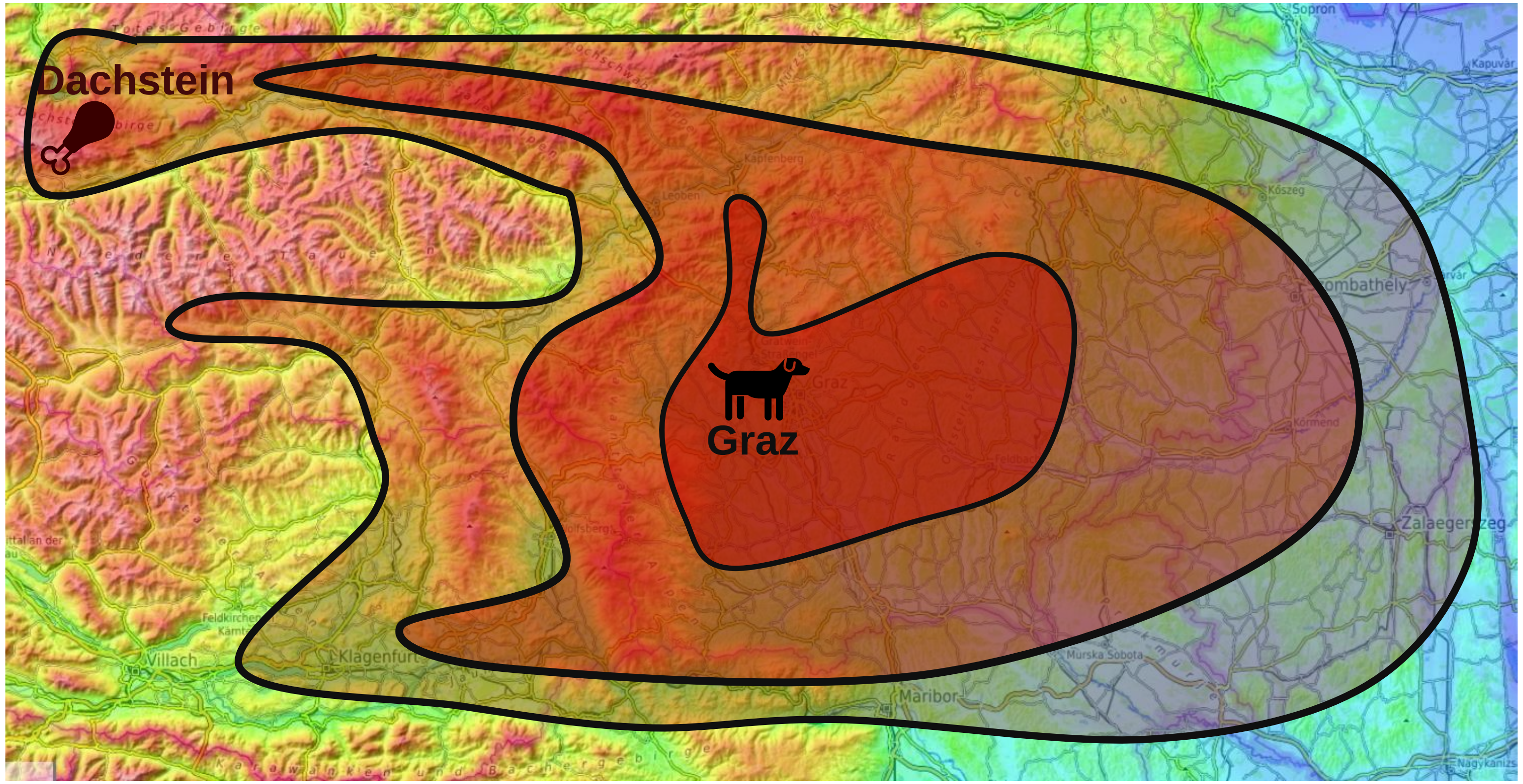


But, we know something about how to get to Dachstein, we need to go Northwest from Graz!

This is more information! *Let's not only prioritize by weights, but also give some priority to the* ***direction*** *we want to go.*

Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



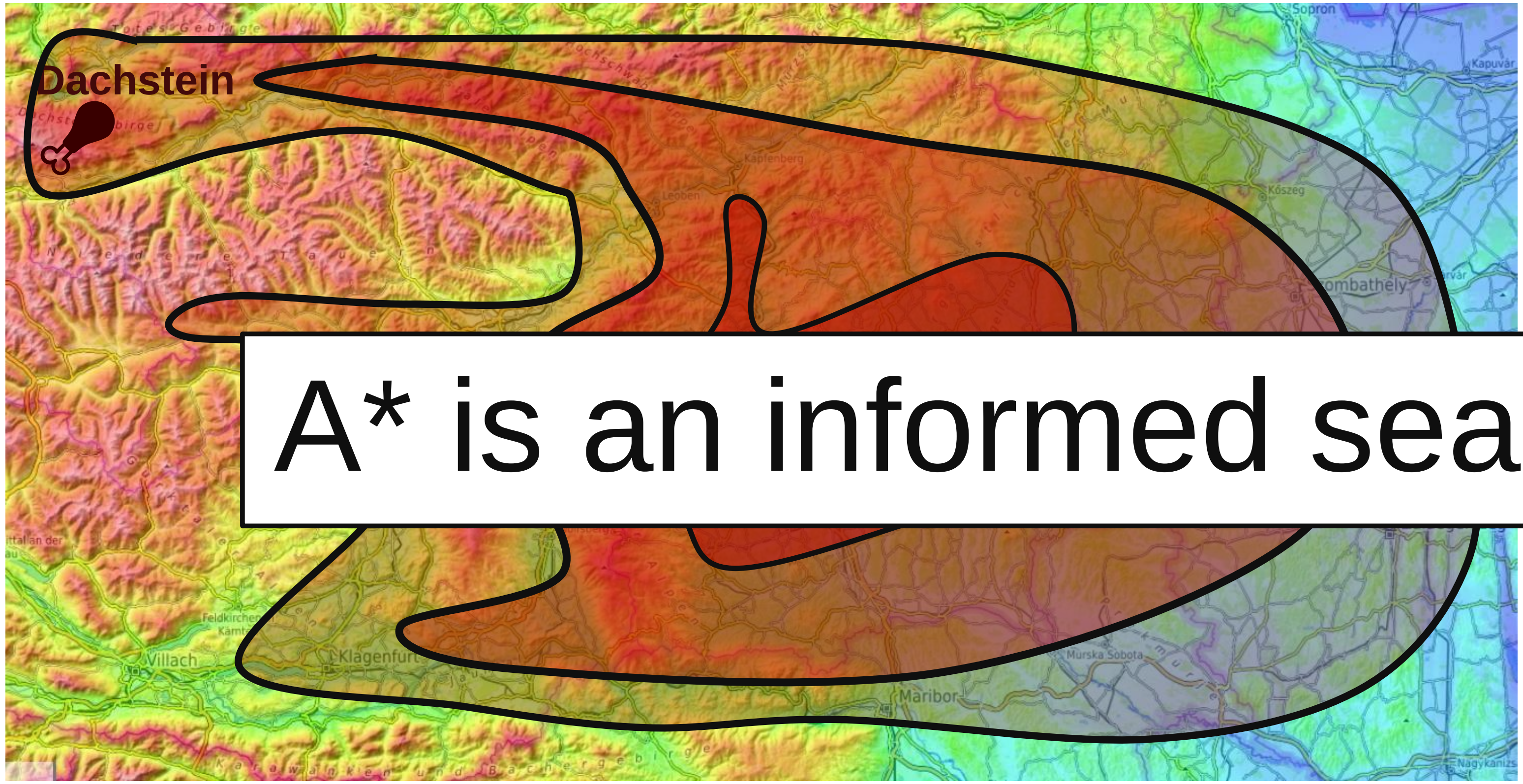
But, we know something about how to get to Dachstein, we need to go Northwest from Graz!

This is more information! *Let's not only prioritize by weights, but also give some priority to the* ***direction*** *we want to go.*

E.g., we will add more information based on a heuristic, (which could be direction in the case of a street map.)

Why do we search in all directions?

If we want to travel from Graz to the Dachstein, Dijkstra's algorithm will look at path distances around Graz.



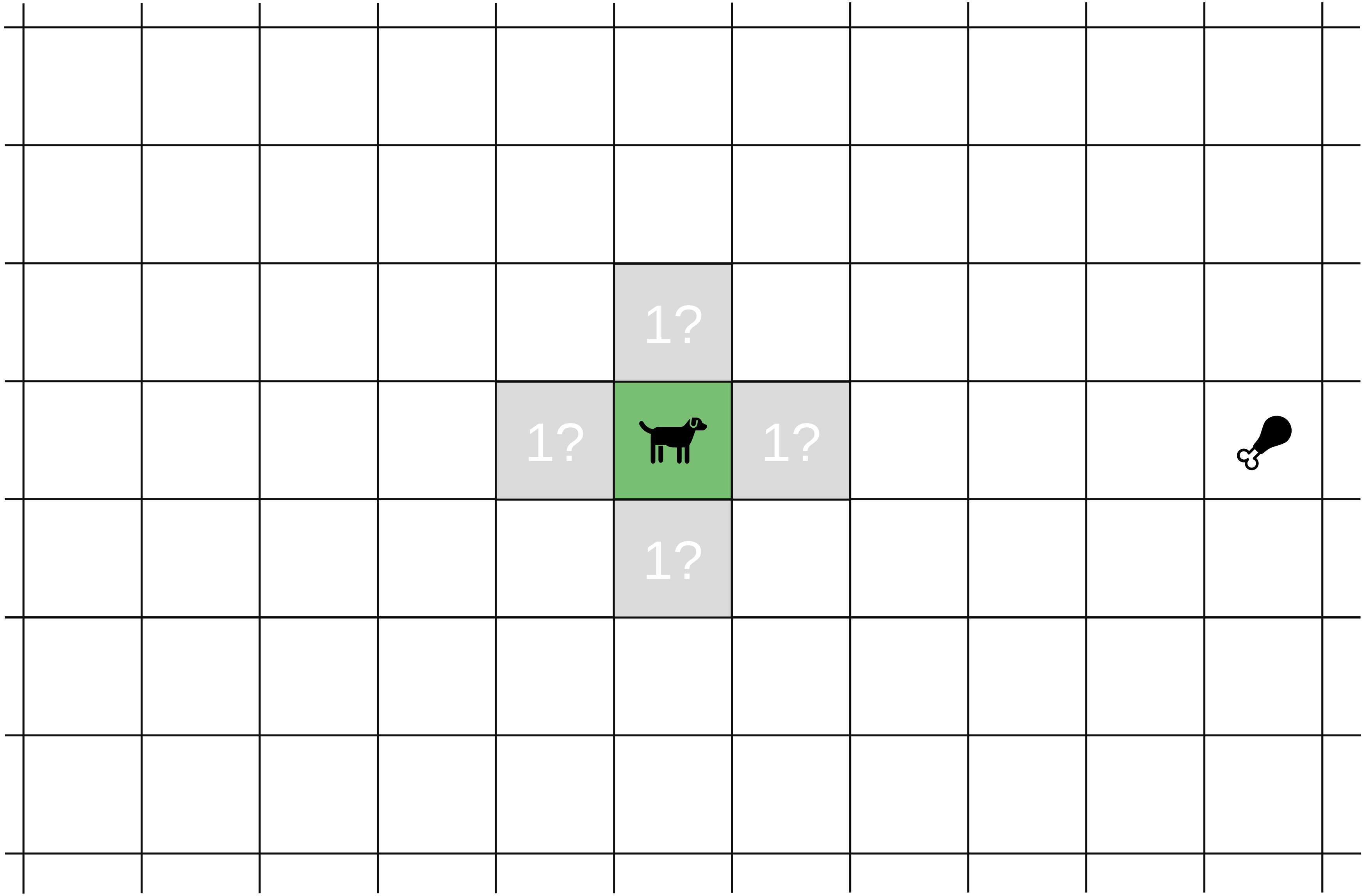
But, we know something about how to get to Dachstein, we need to go Northwest from Graz!

A* is an informed search algorithm

's not
t also
*give some priority to the **direction** we want to go.*

E.g., we will add more information based on a heuristic, (which could be direction in the case of a street map.)

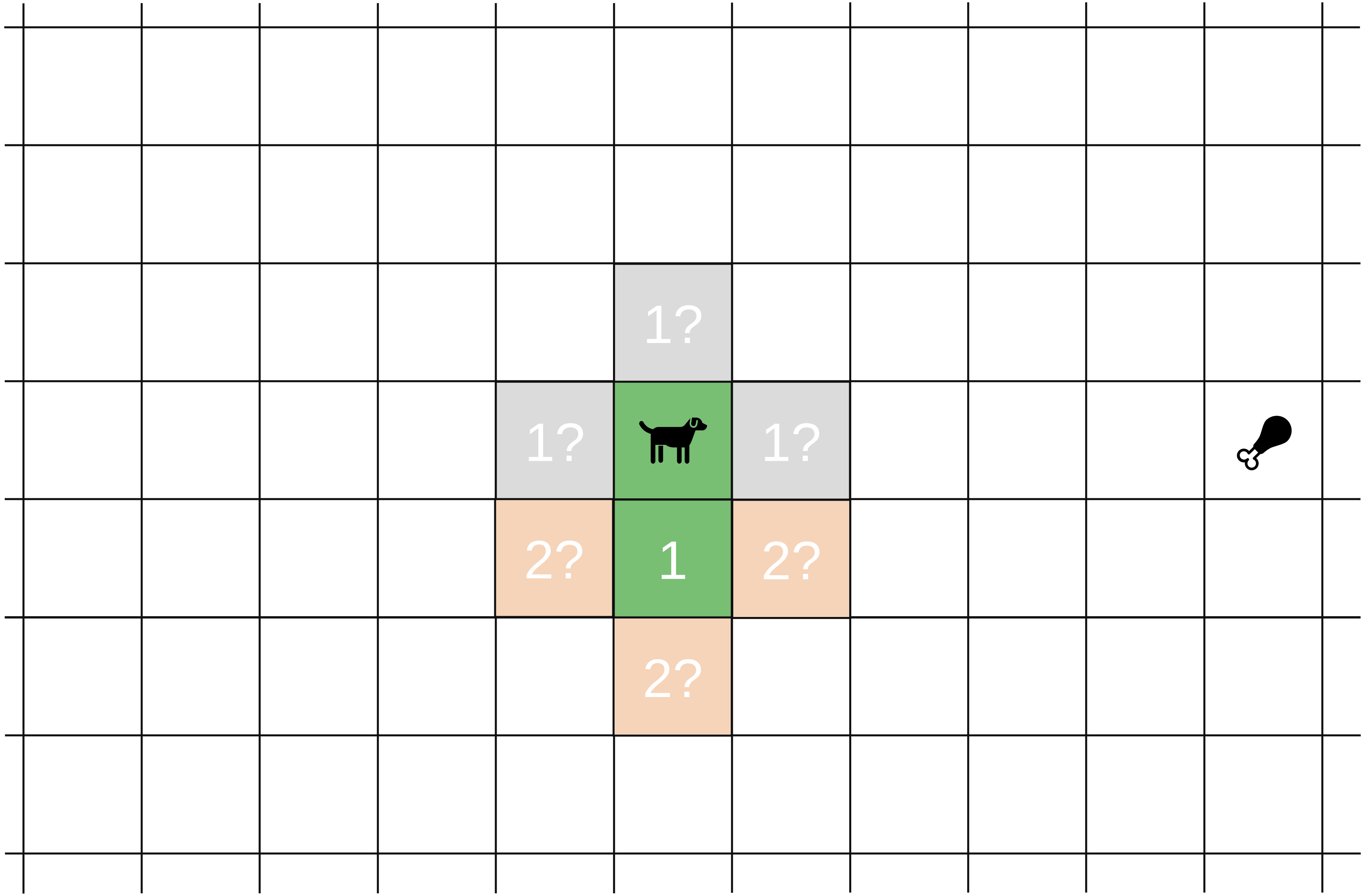
Dijkstra where each edge has cost 1



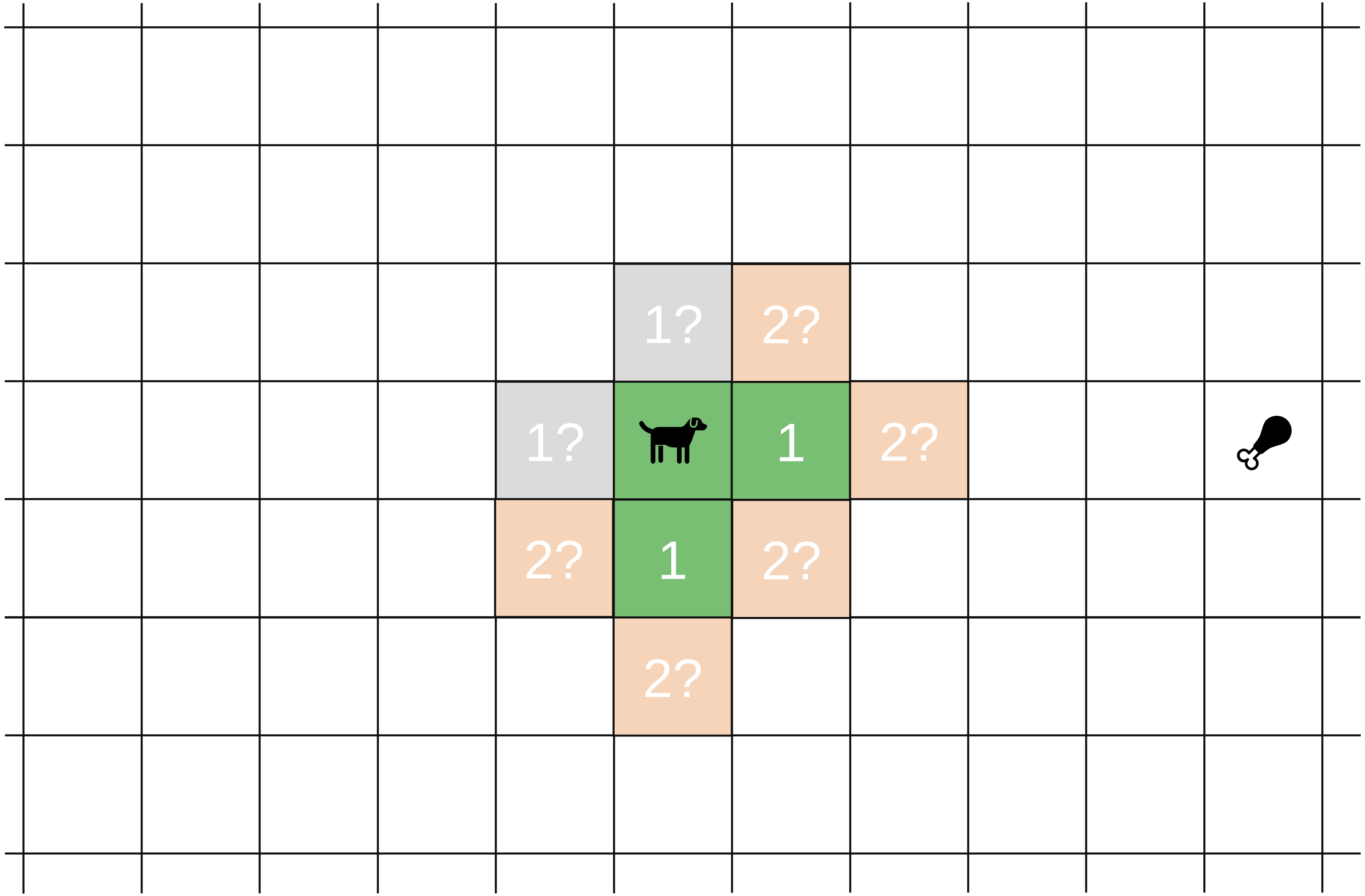
Greedily continue at smallest

distance(s, u)

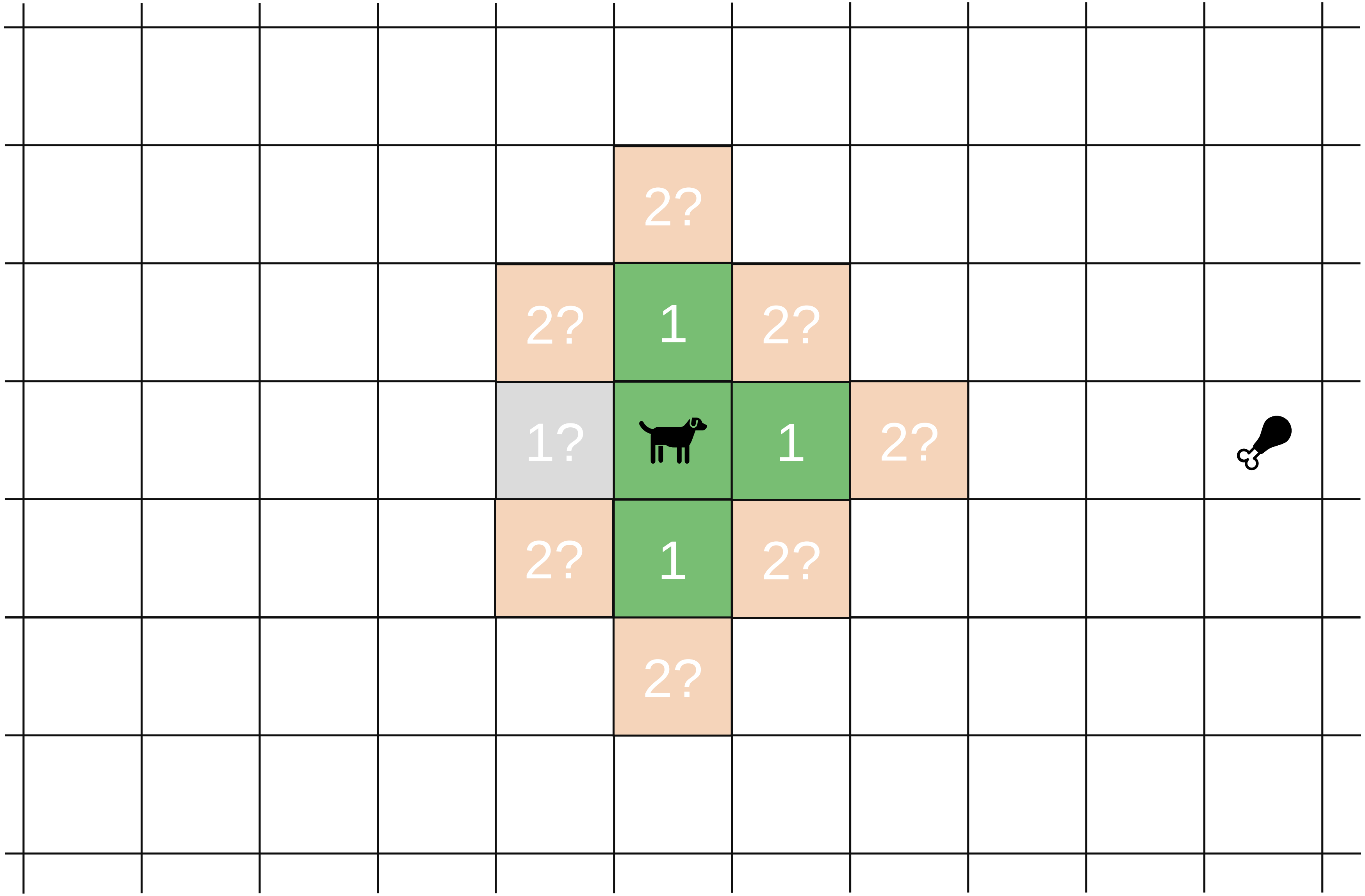
Dijkstra where each edge has cost 1



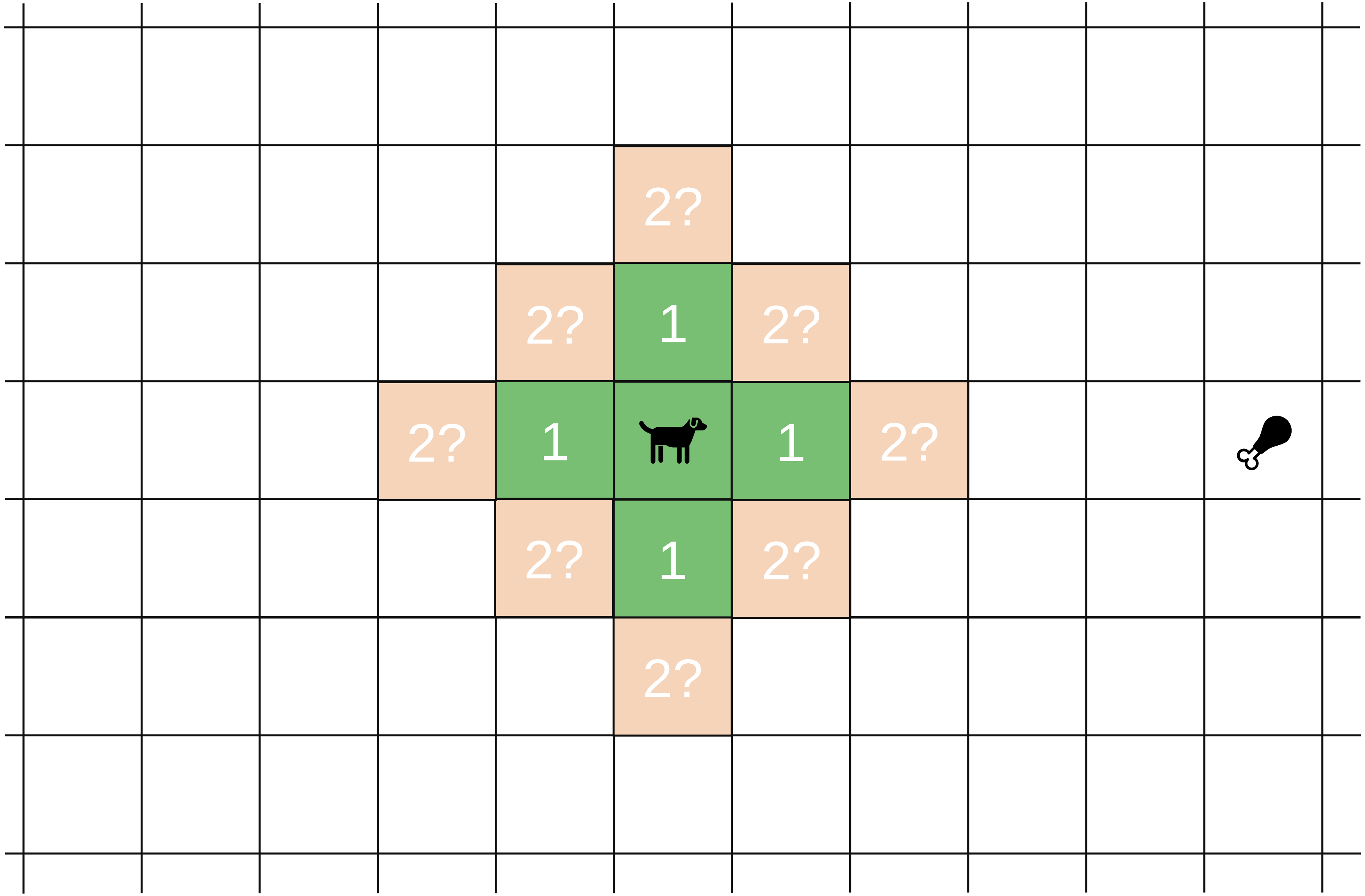
Dijkstra where each edge has cost 1



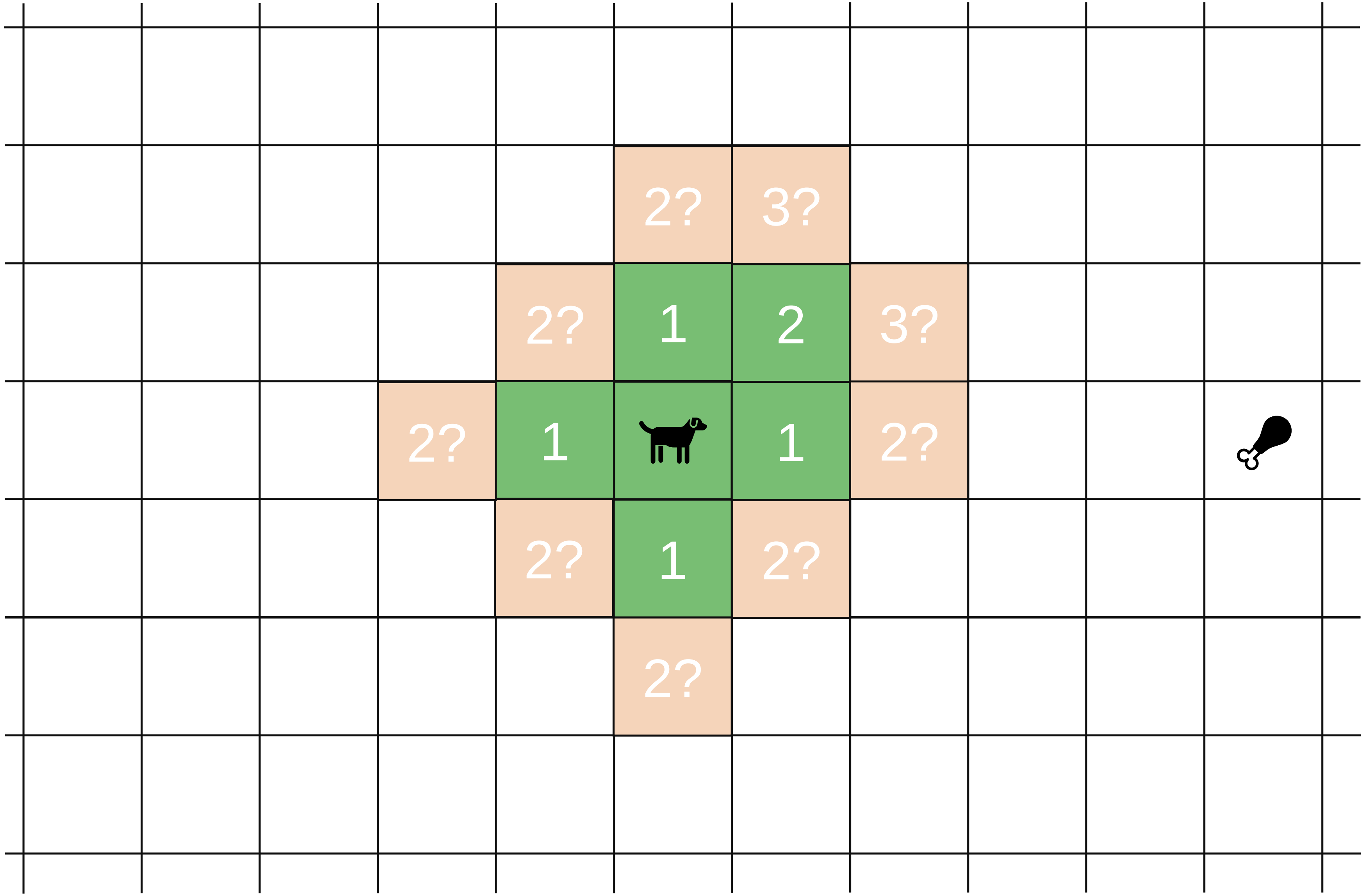
Dijkstra where each edge has cost 1



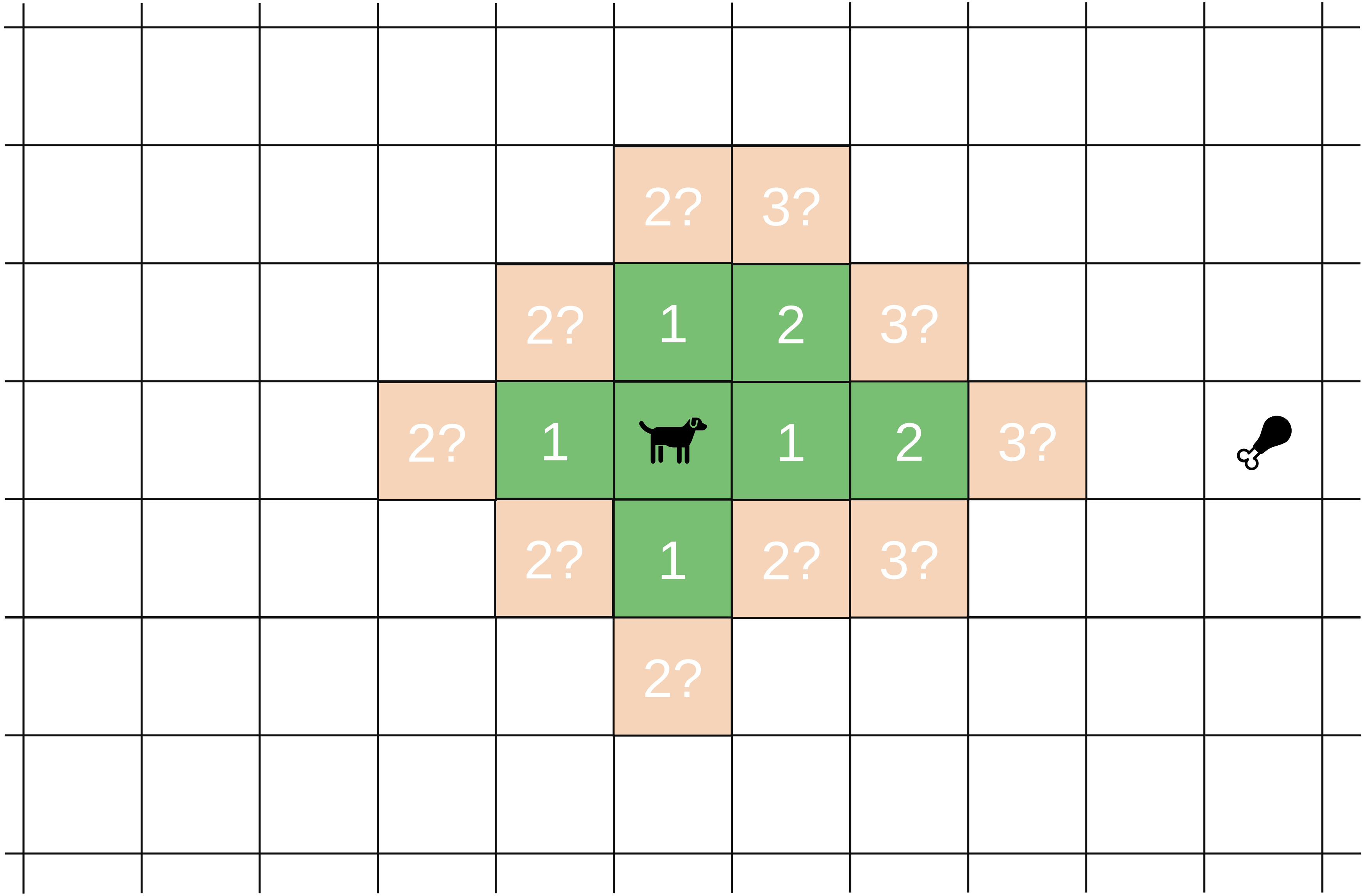
Dijkstra where each edge has cost 1



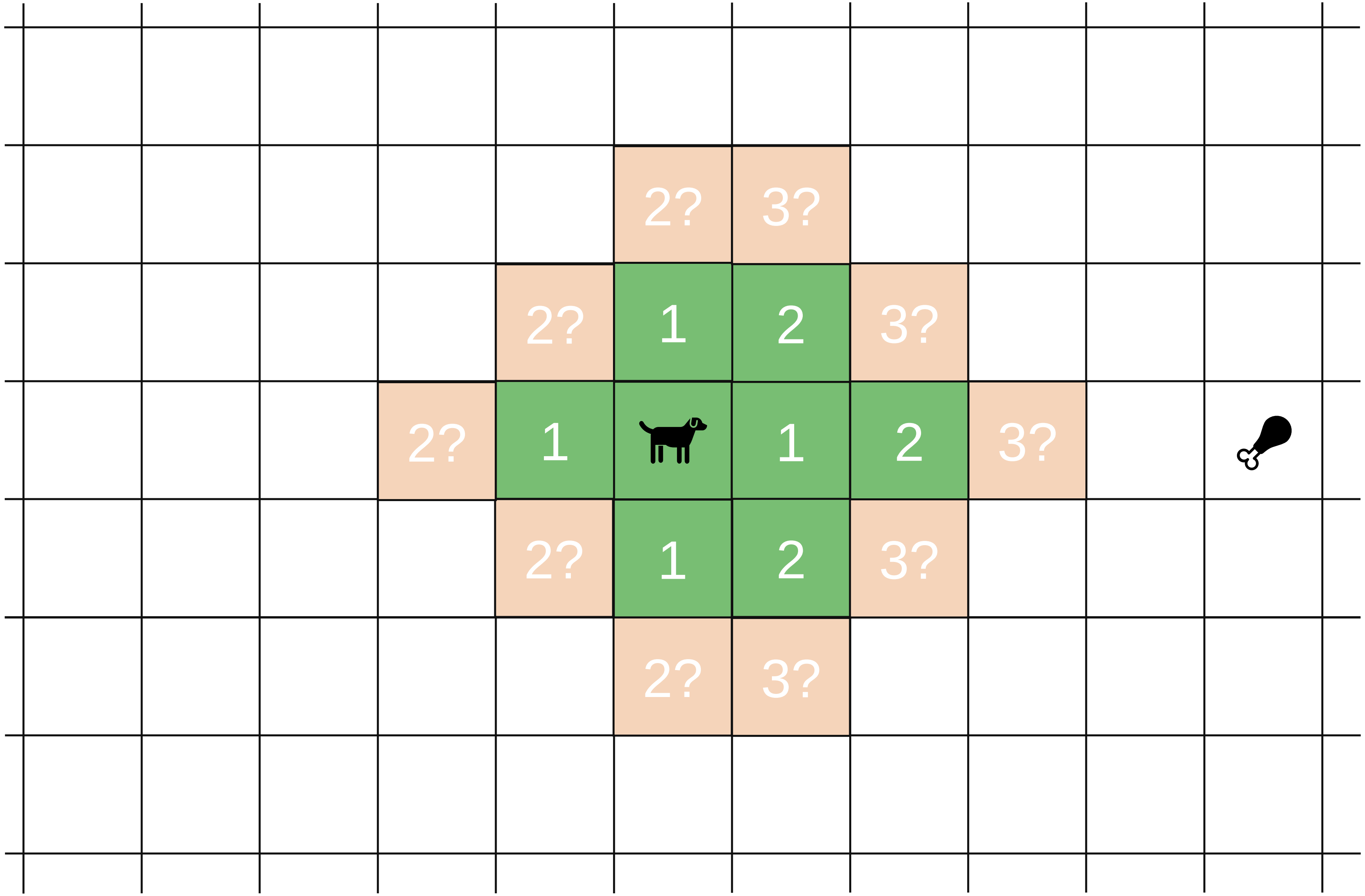
Dijkstra where each edge has cost 1



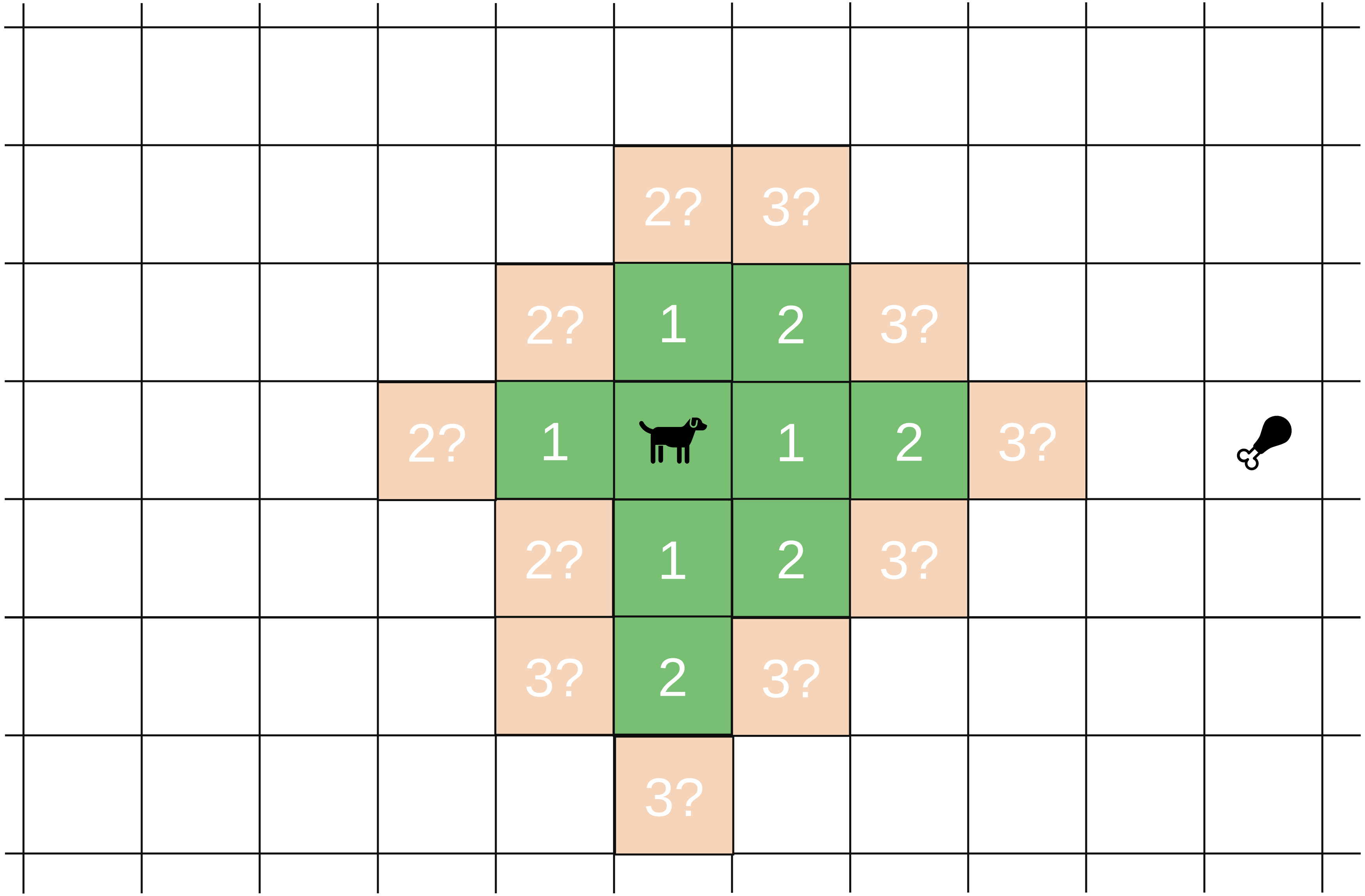
Dijkstra where each edge has cost 1



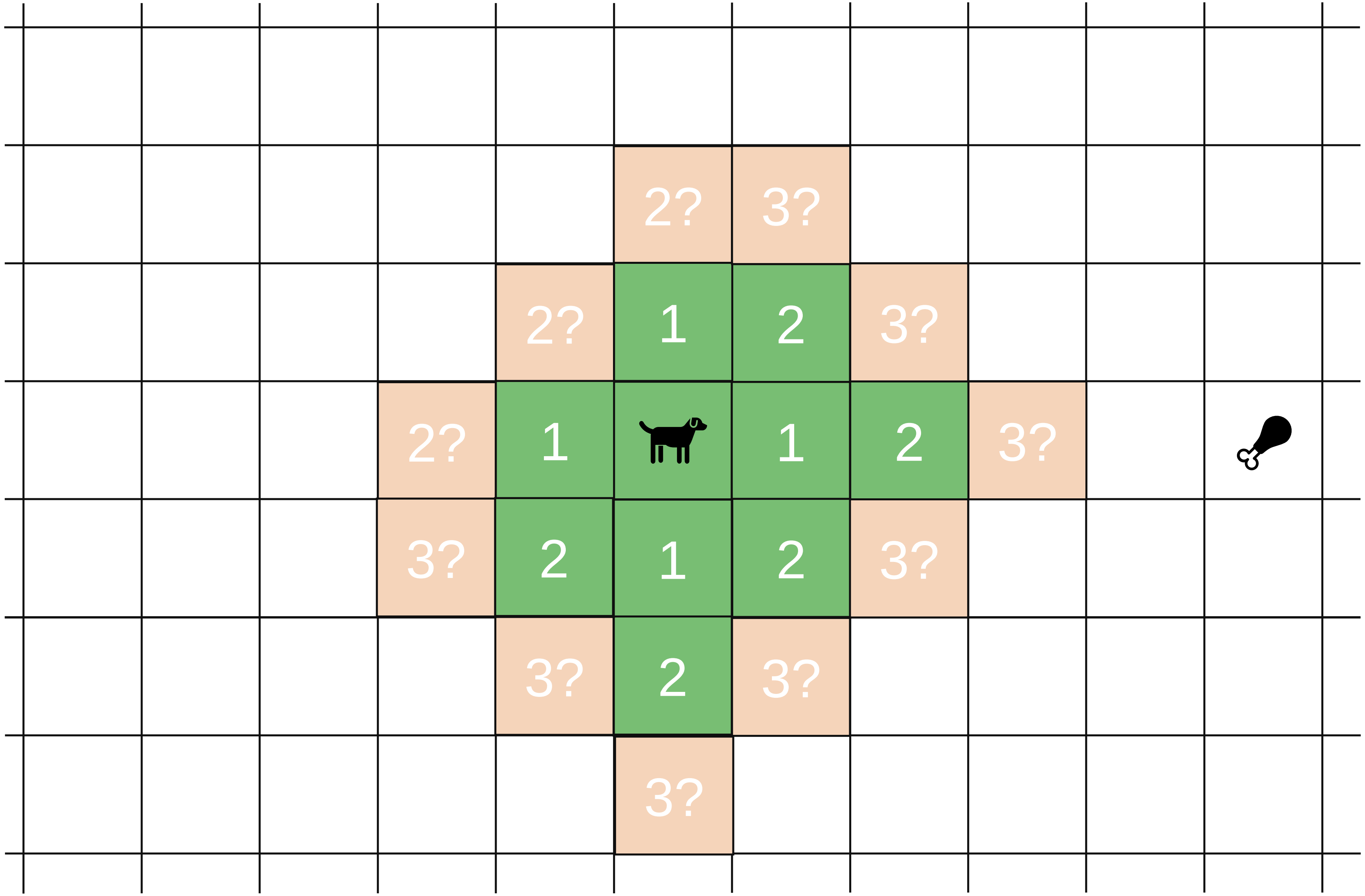
Dijkstra where each edge has cost 1



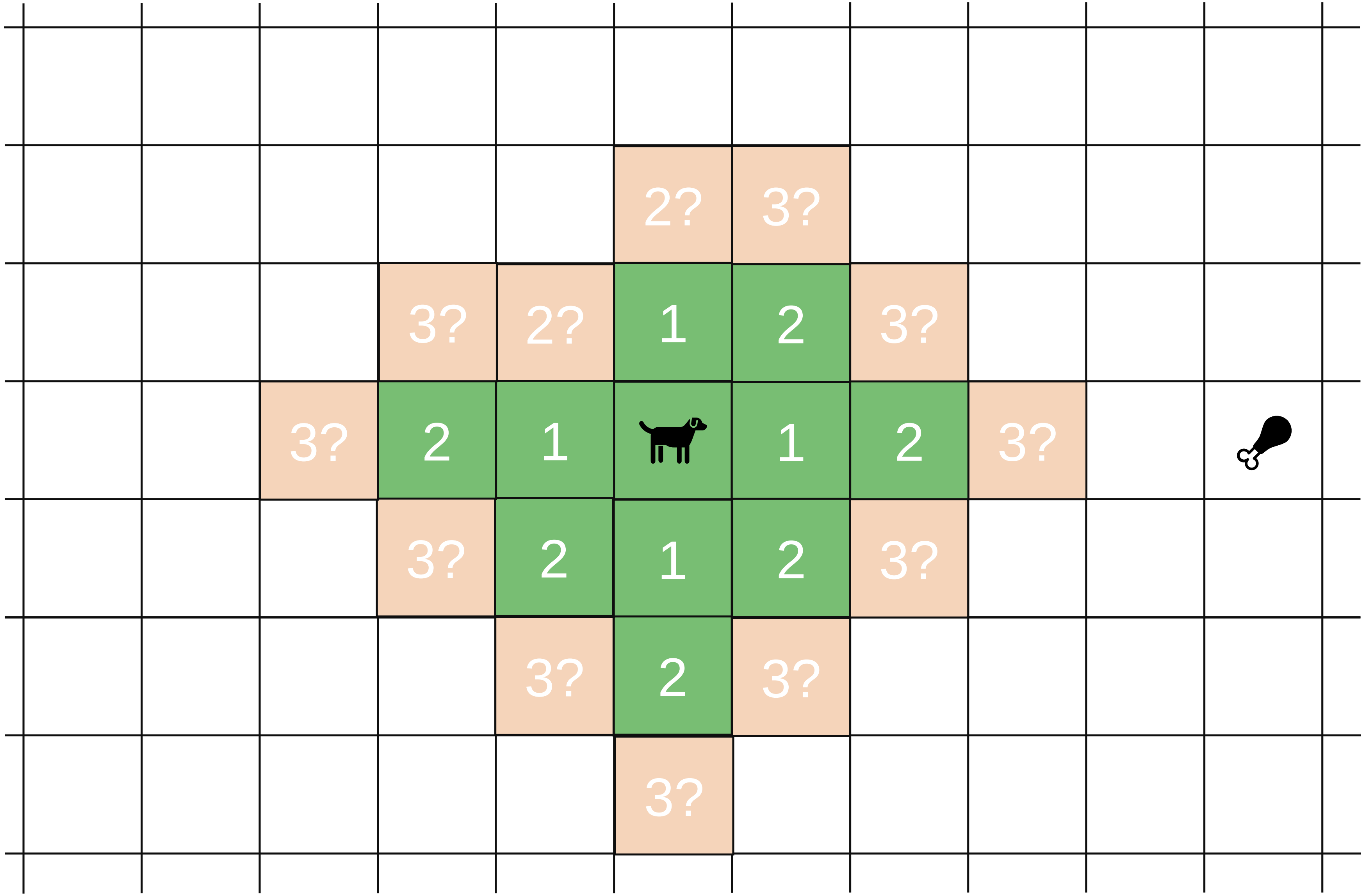
Dijkstra where each edge has cost 1



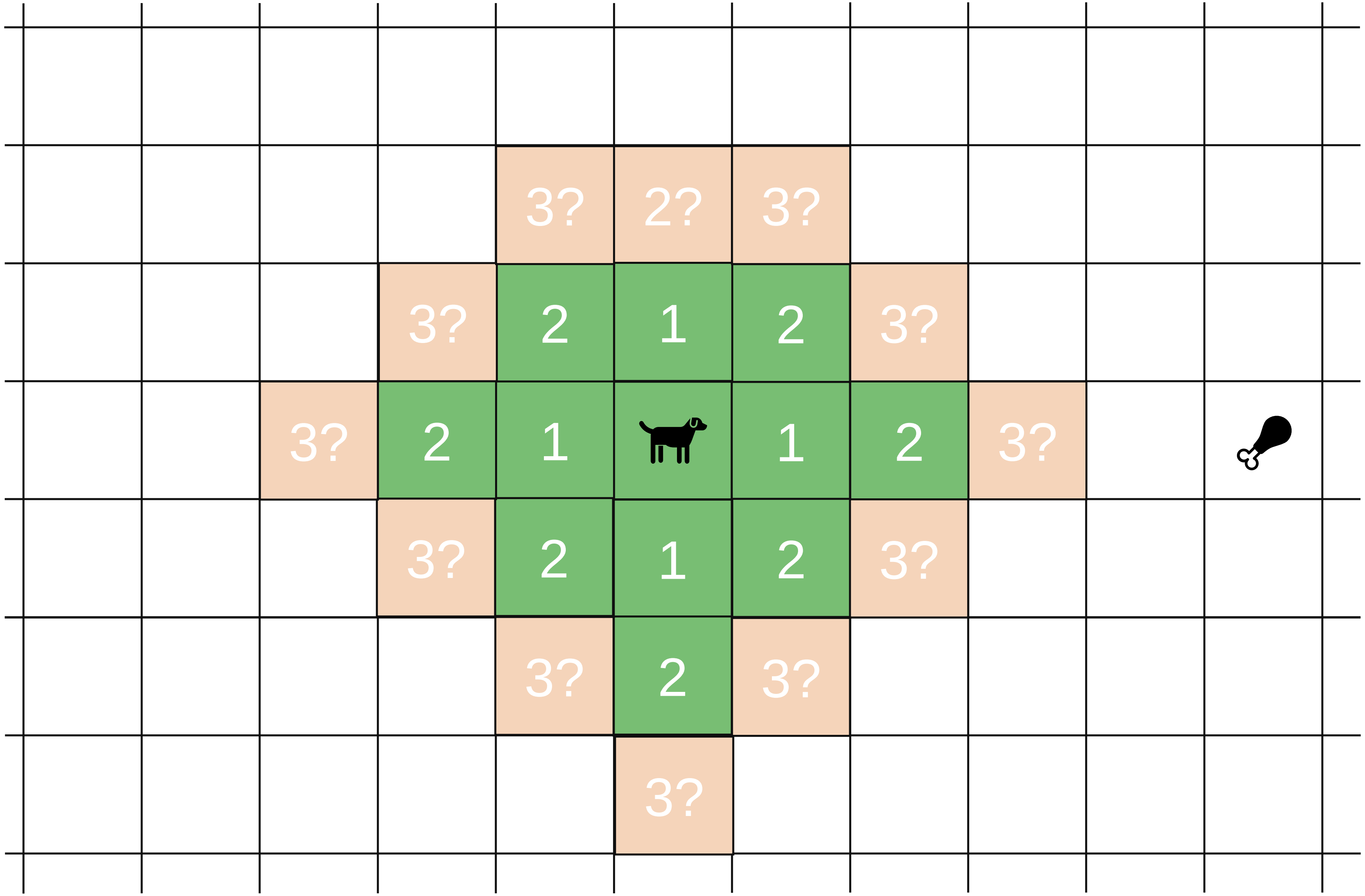
Dijkstra where each edge has cost 1



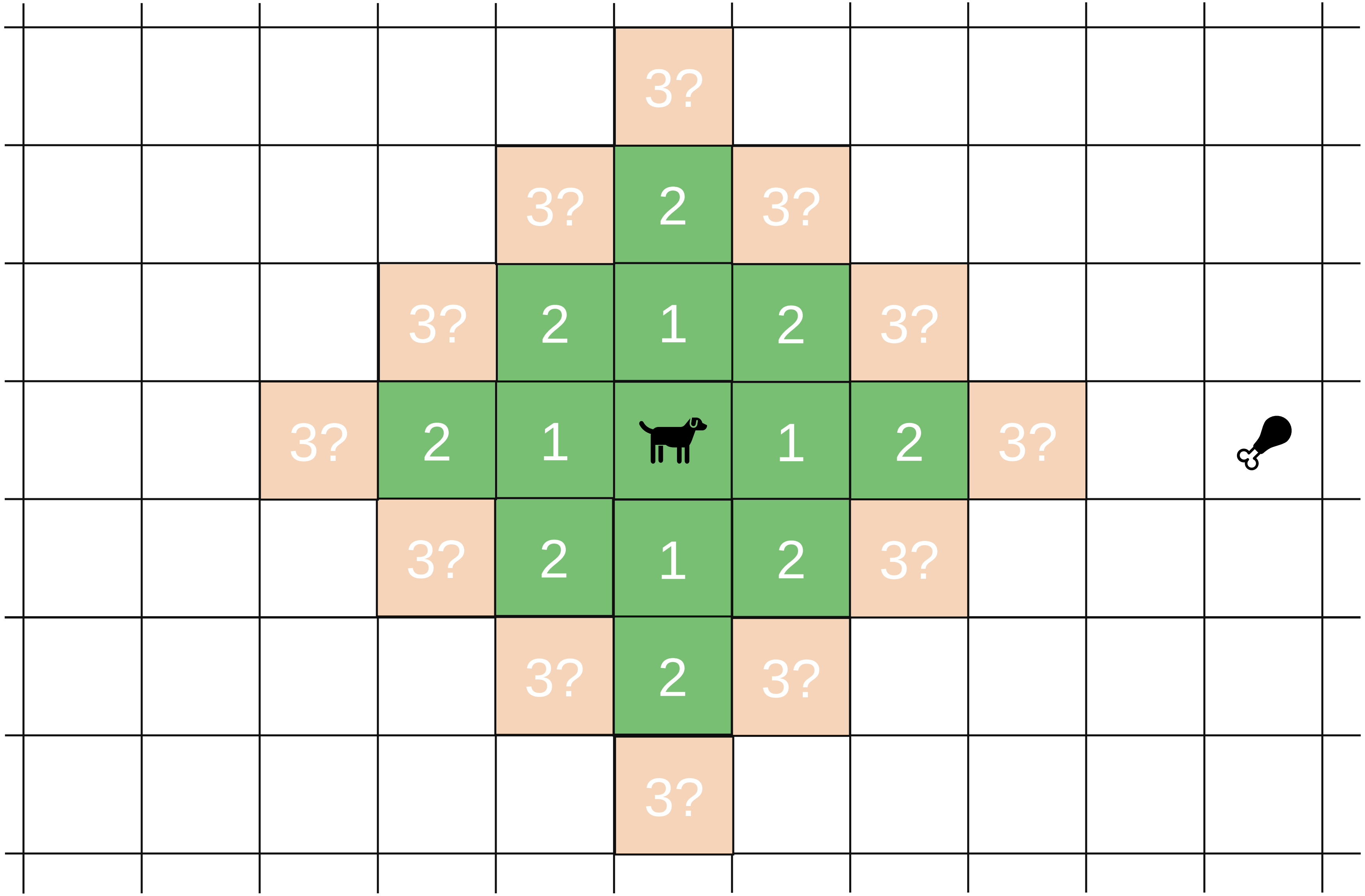
Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1

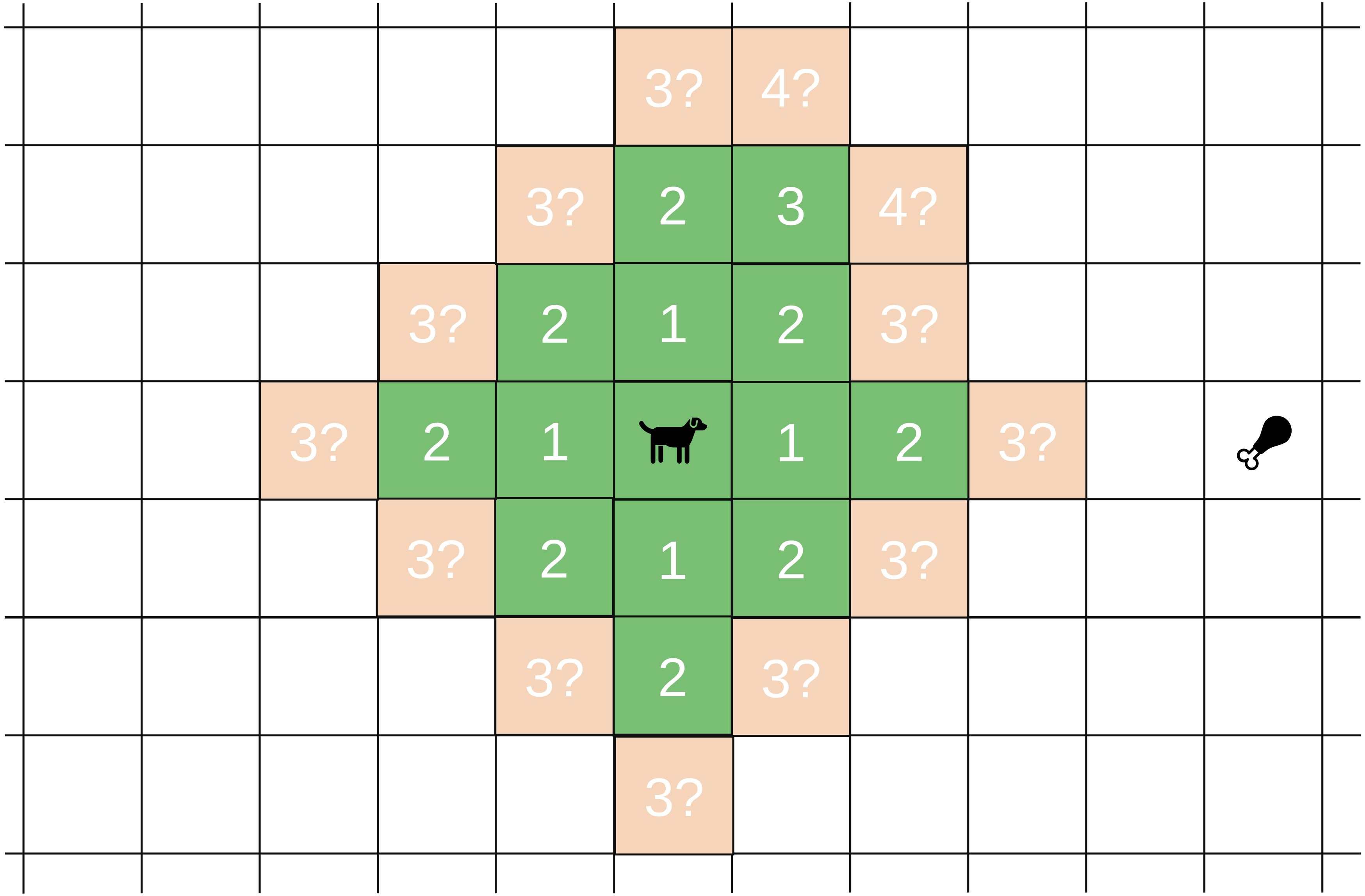


Dijkstra where each edge has cost 1

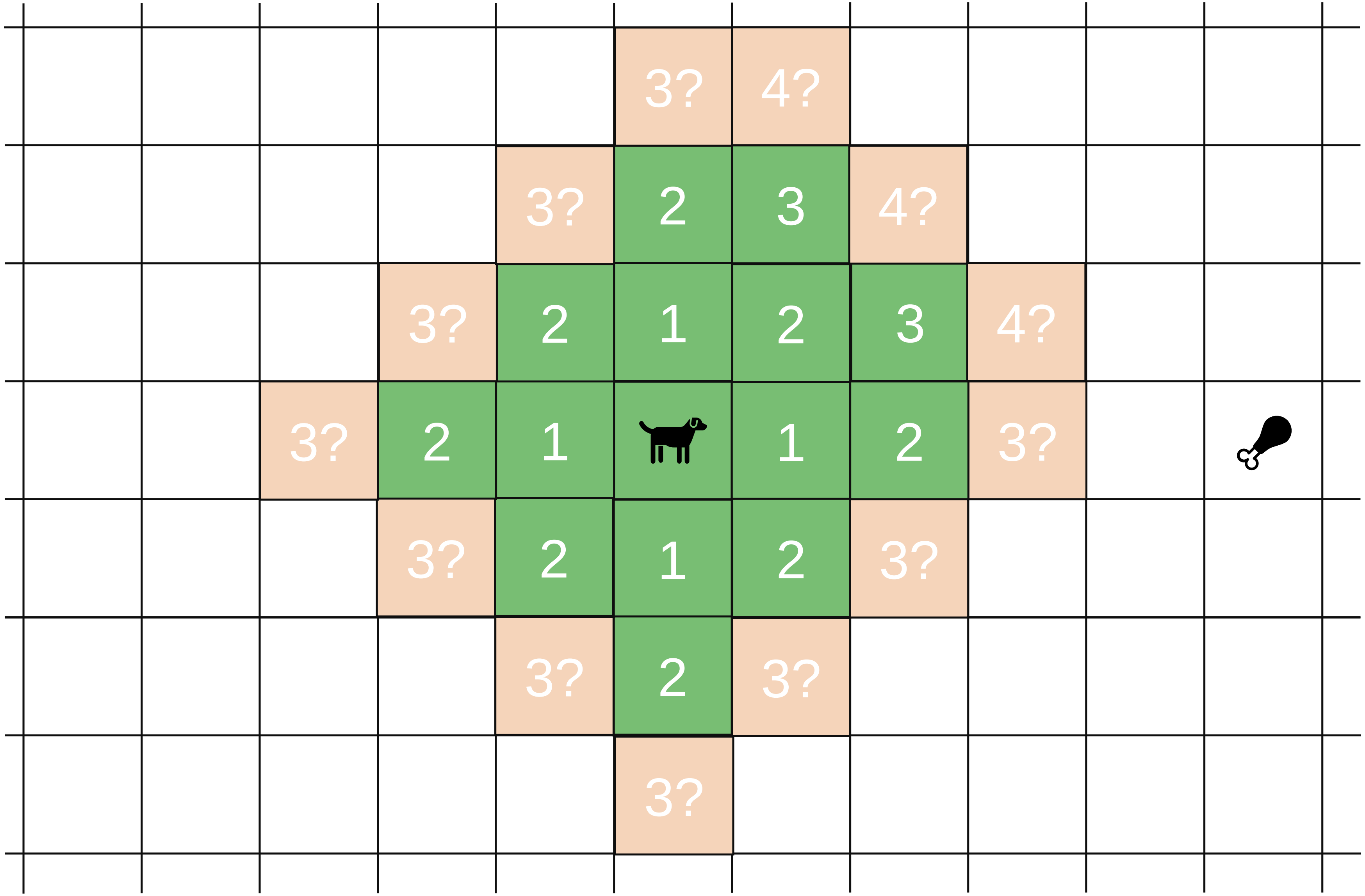


Yannic Maus

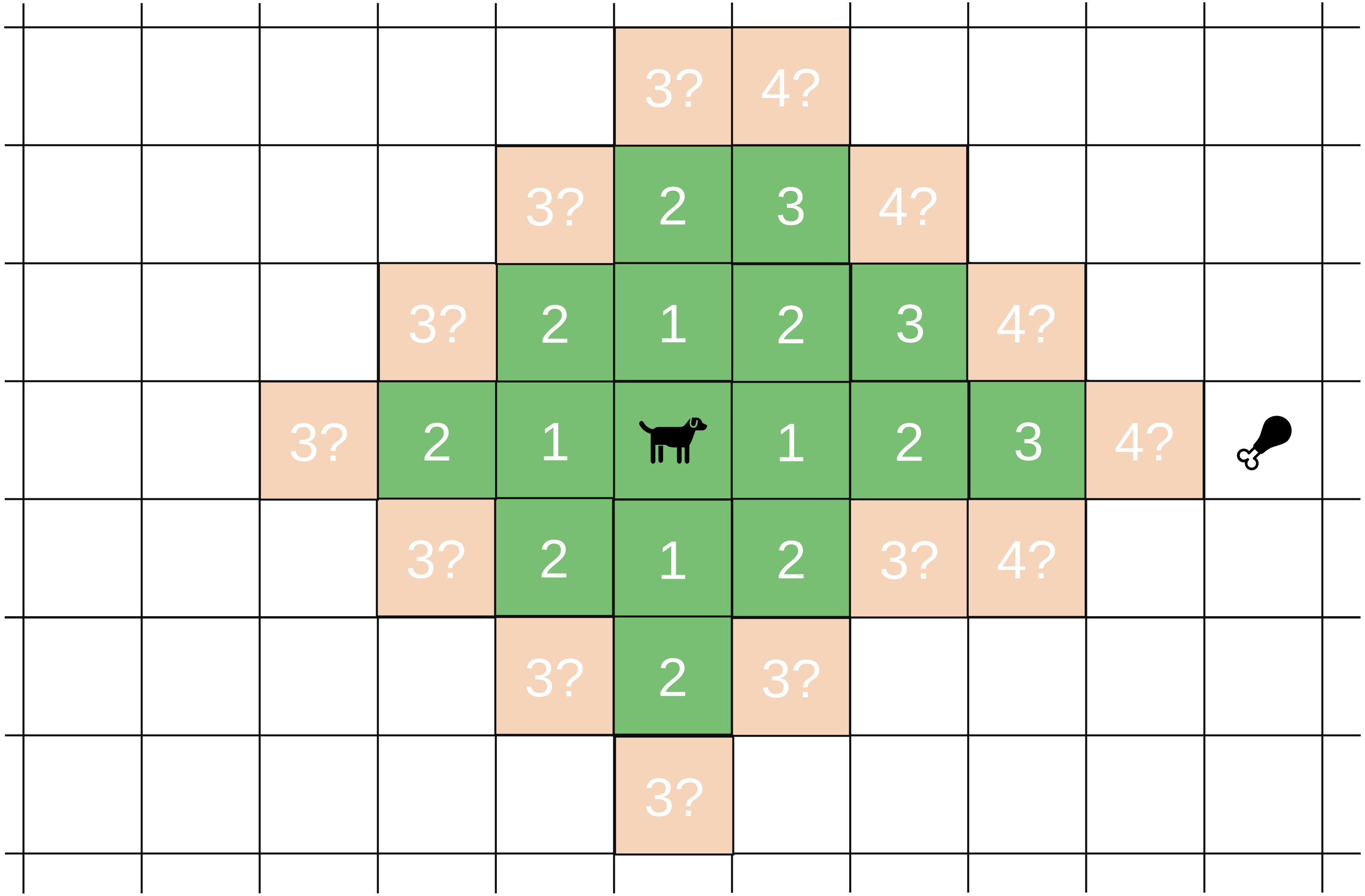
Dijkstra where each edge has cost 1



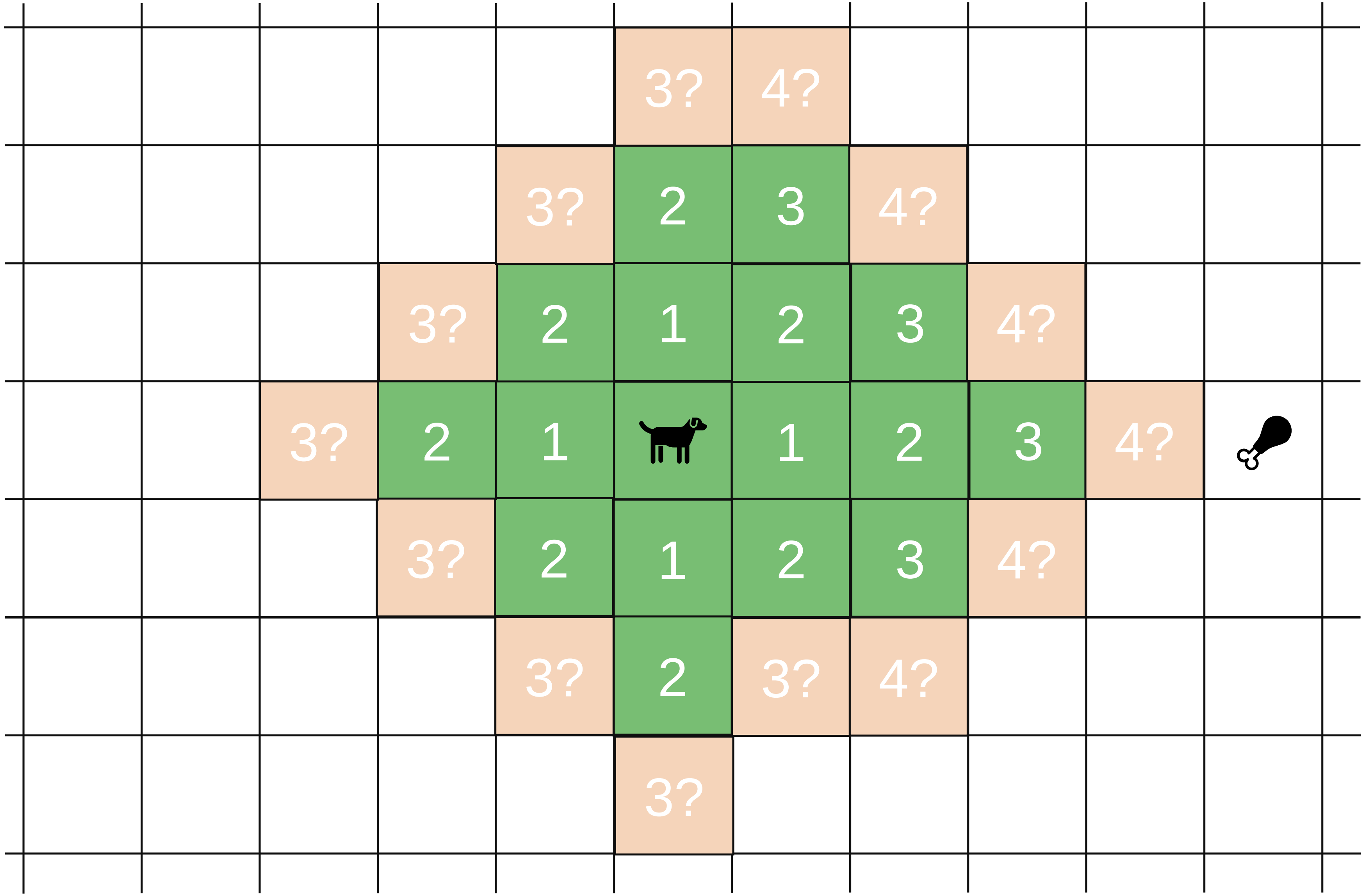
Dijkstra where each edge has cost 1



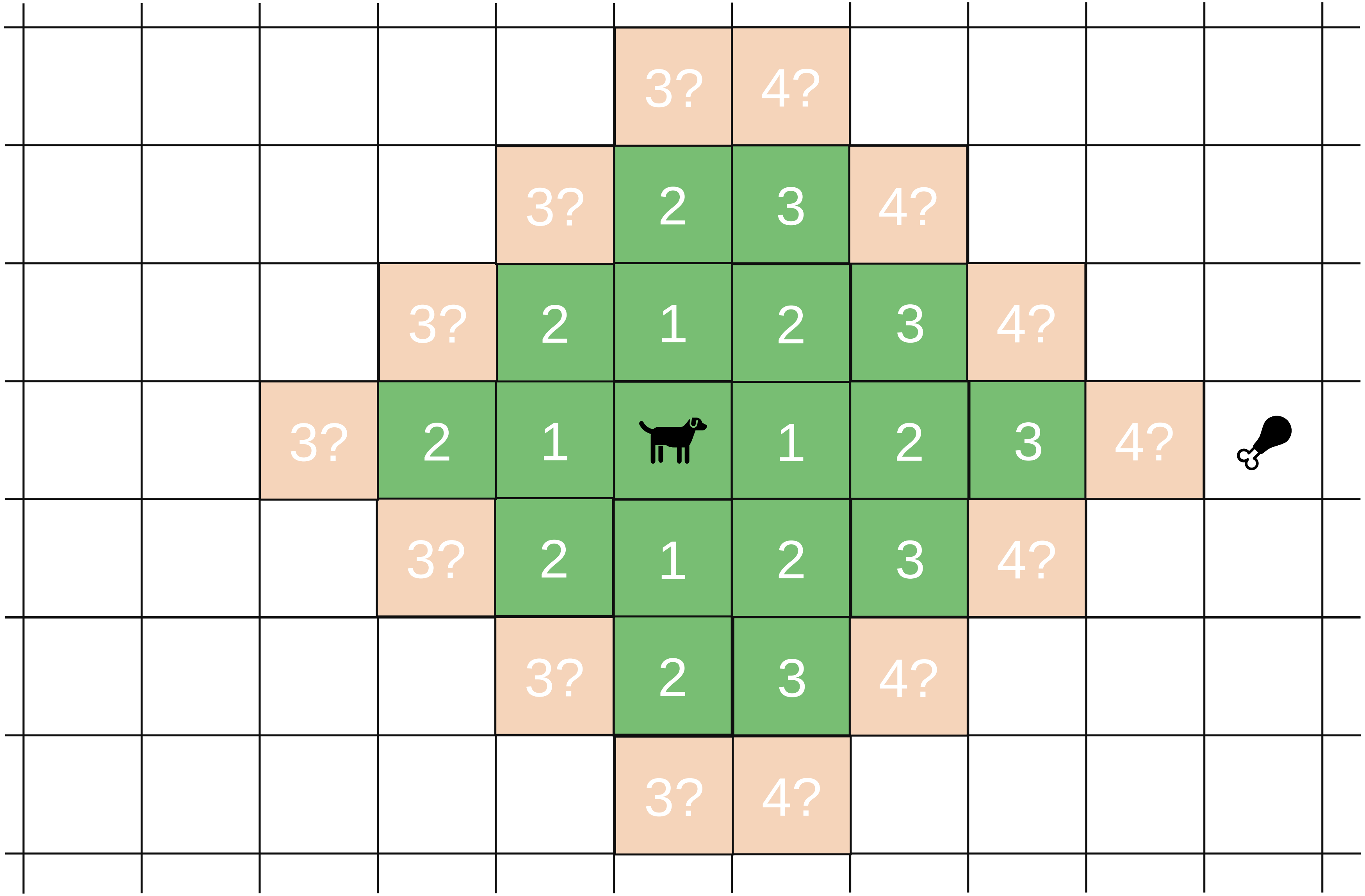
Dijkstra where each edge has cost 1



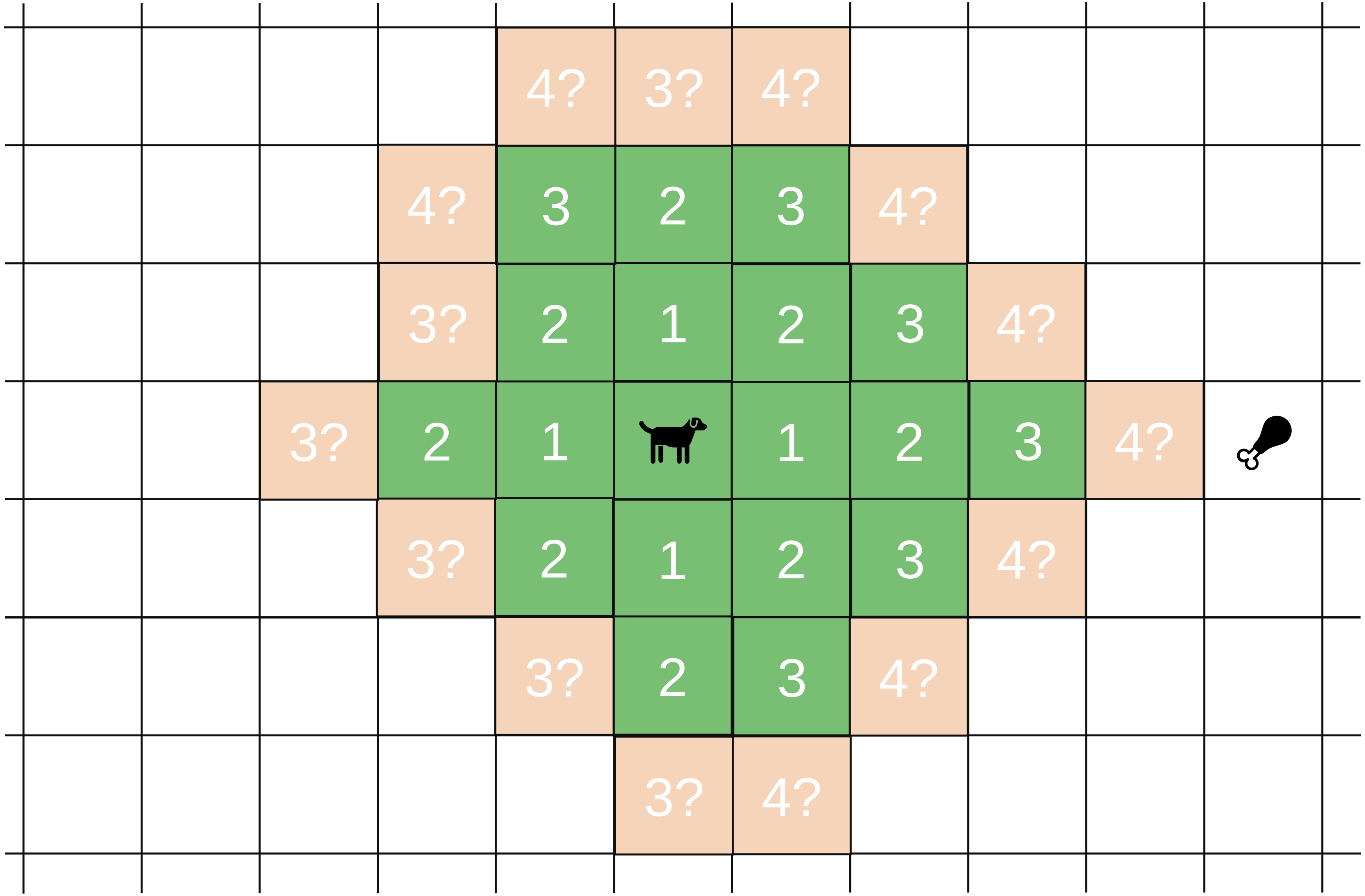
Dijkstra where each edge has cost 1



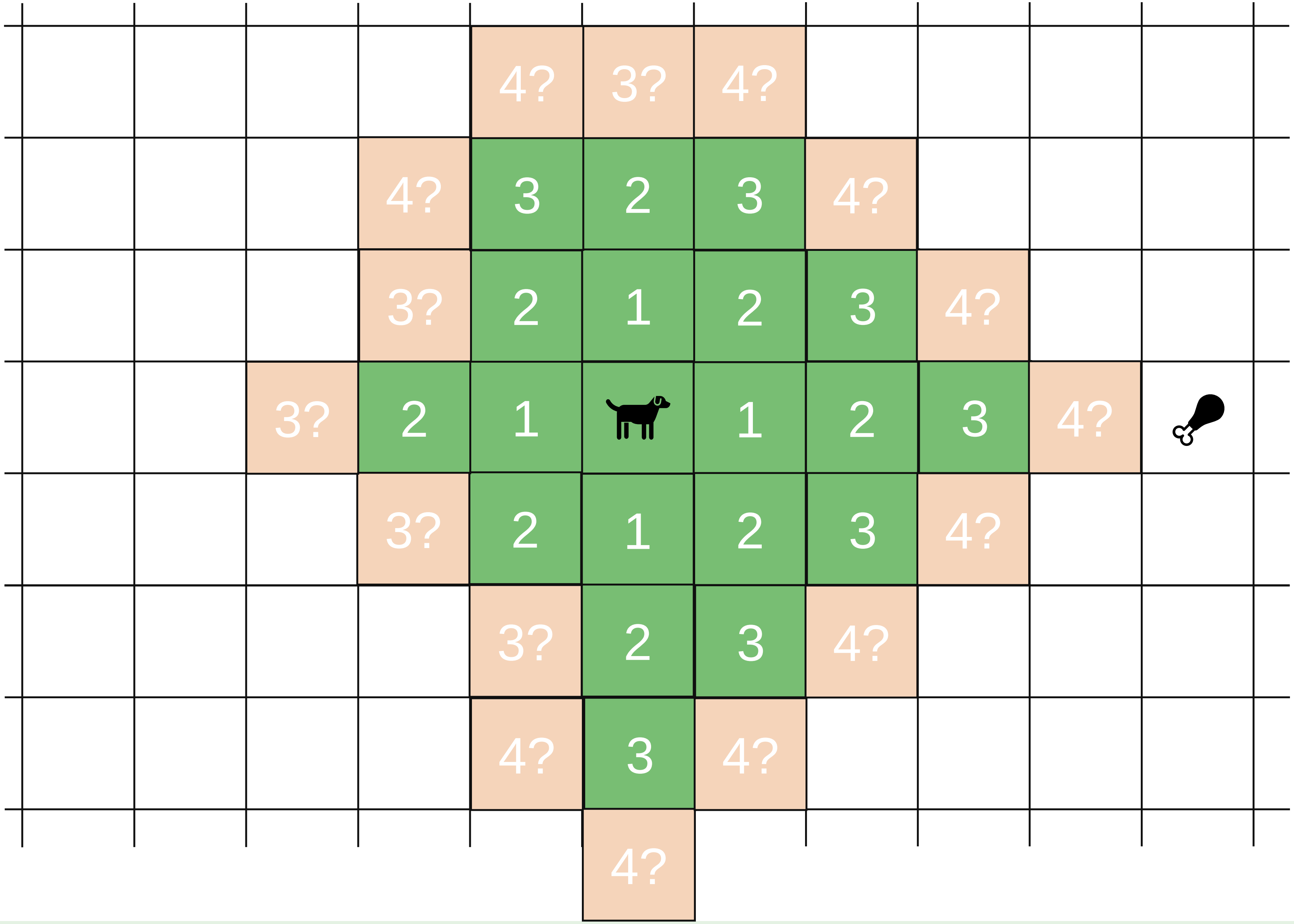
Dijkstra where each edge has cost 1



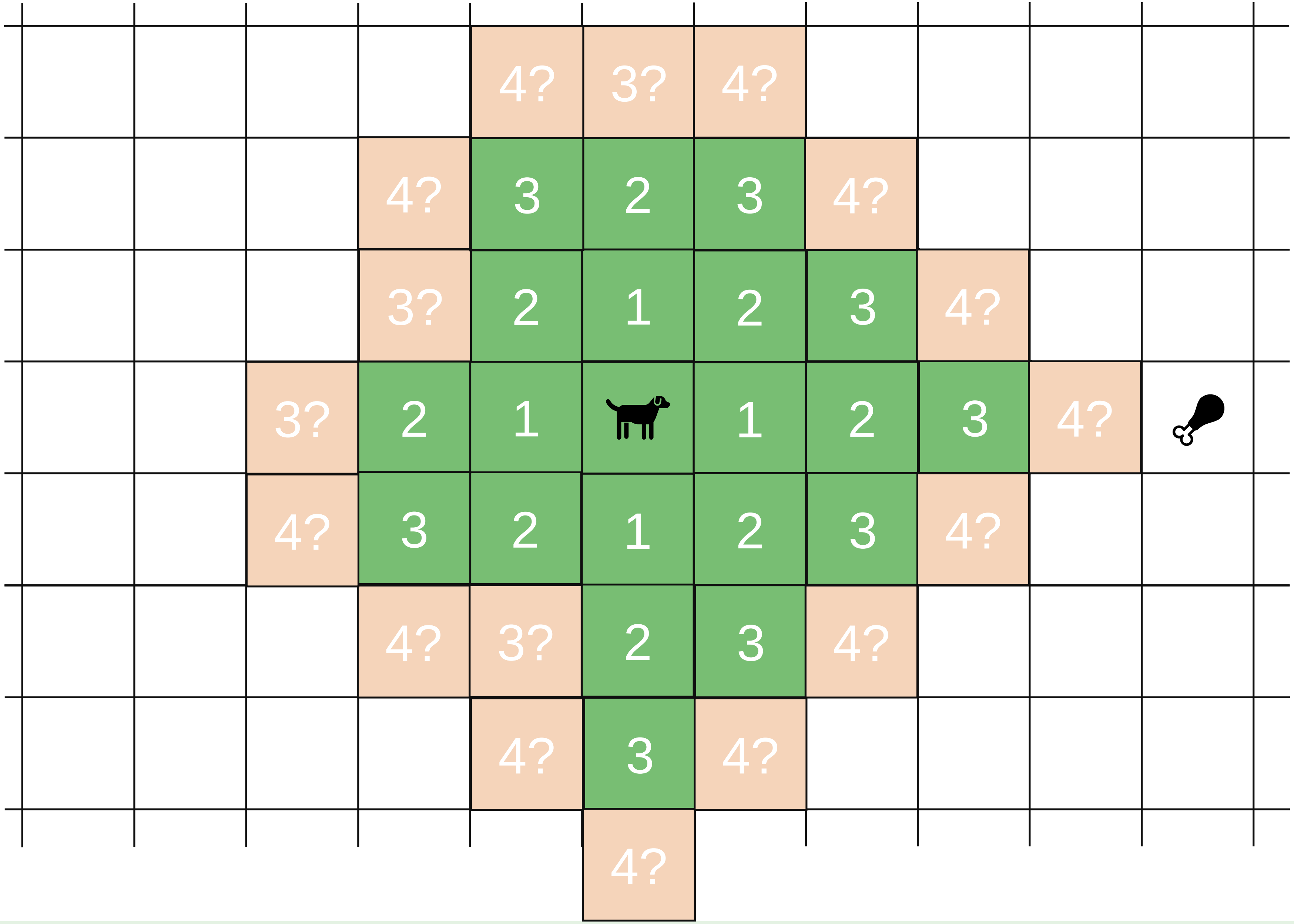
Dijkstra where each edge has cost 1



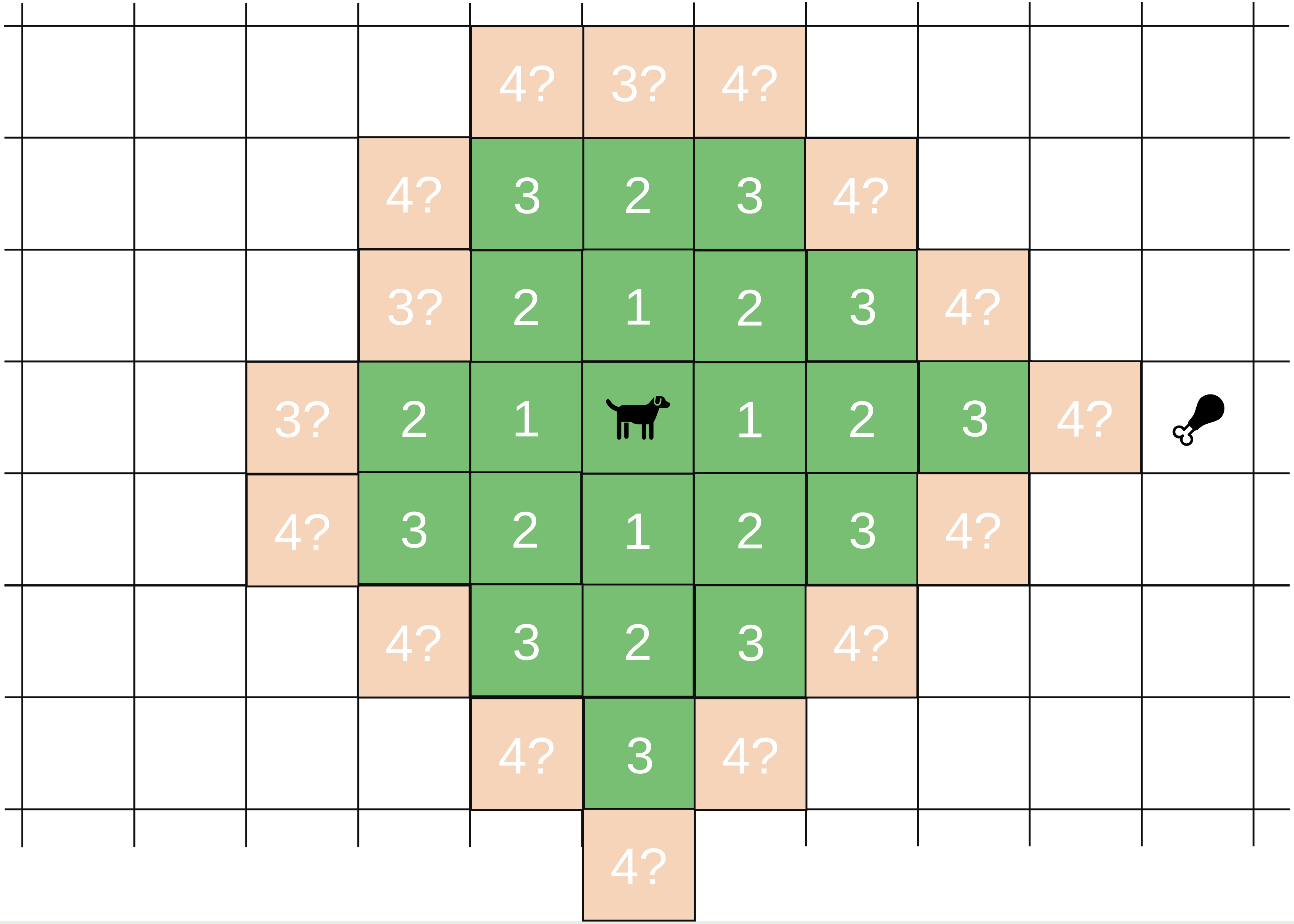
Dijkstra where each edge has cost 1



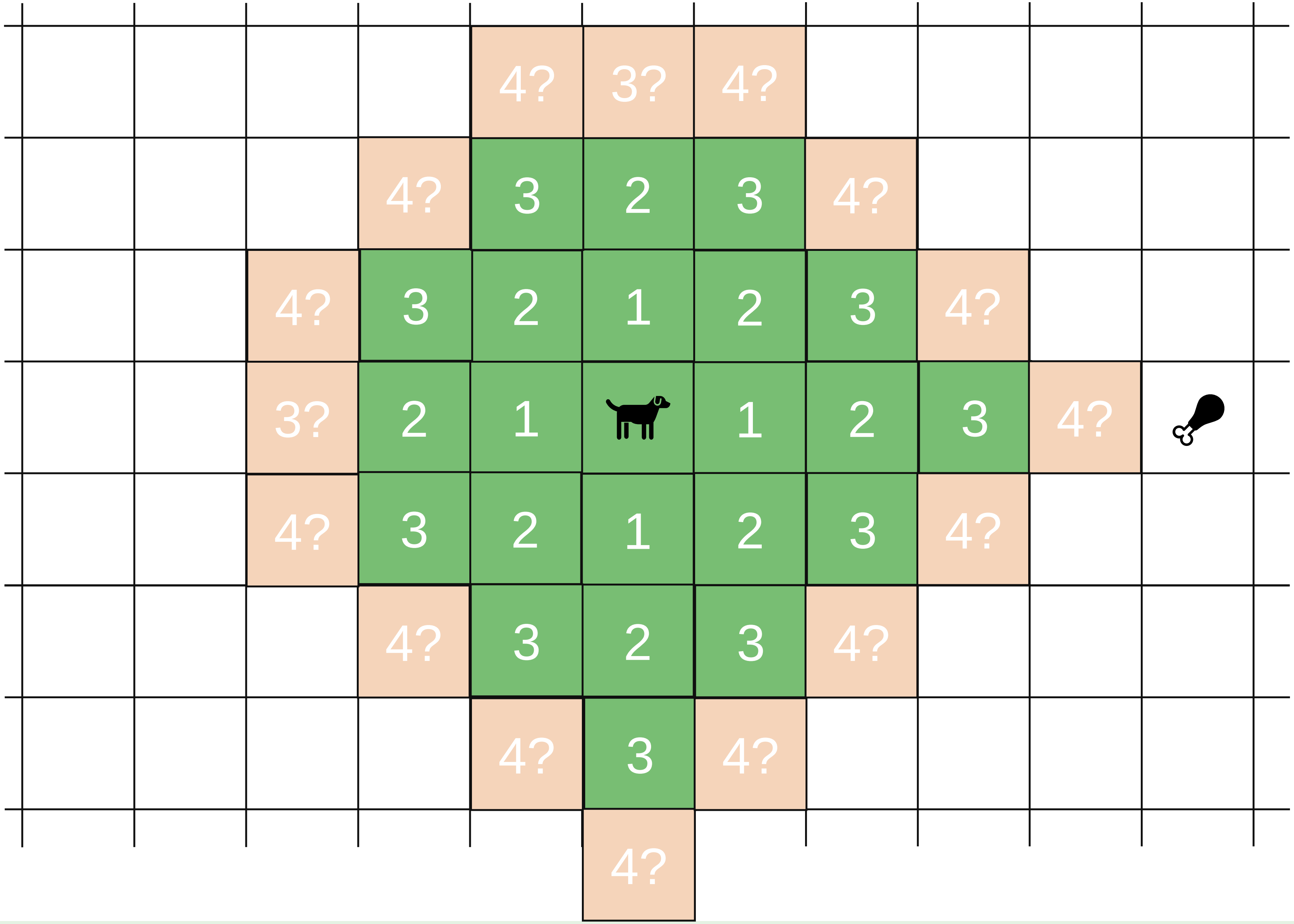
Dijkstra where each edge has cost 1



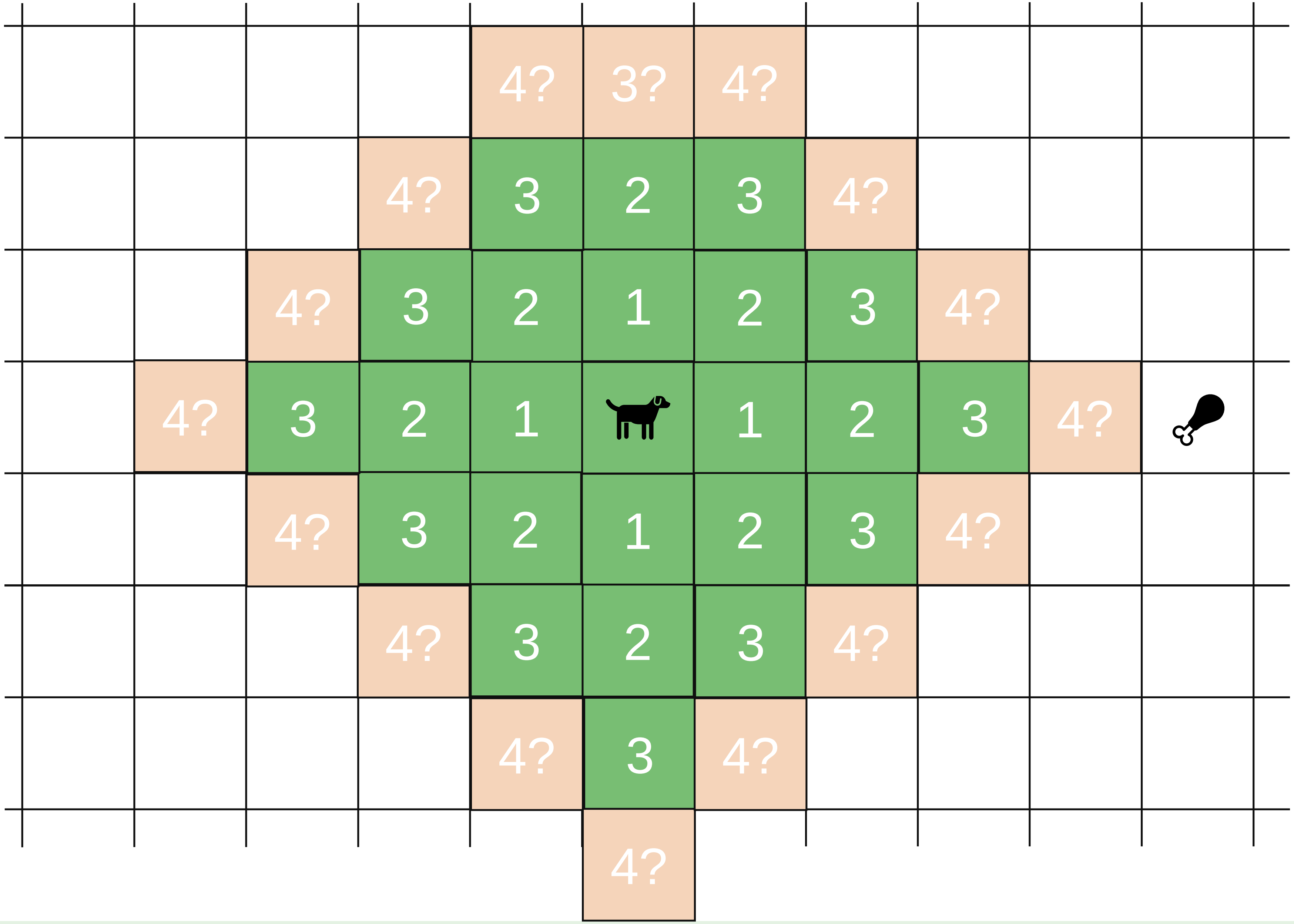
Dijkstra where each edge has cost 1



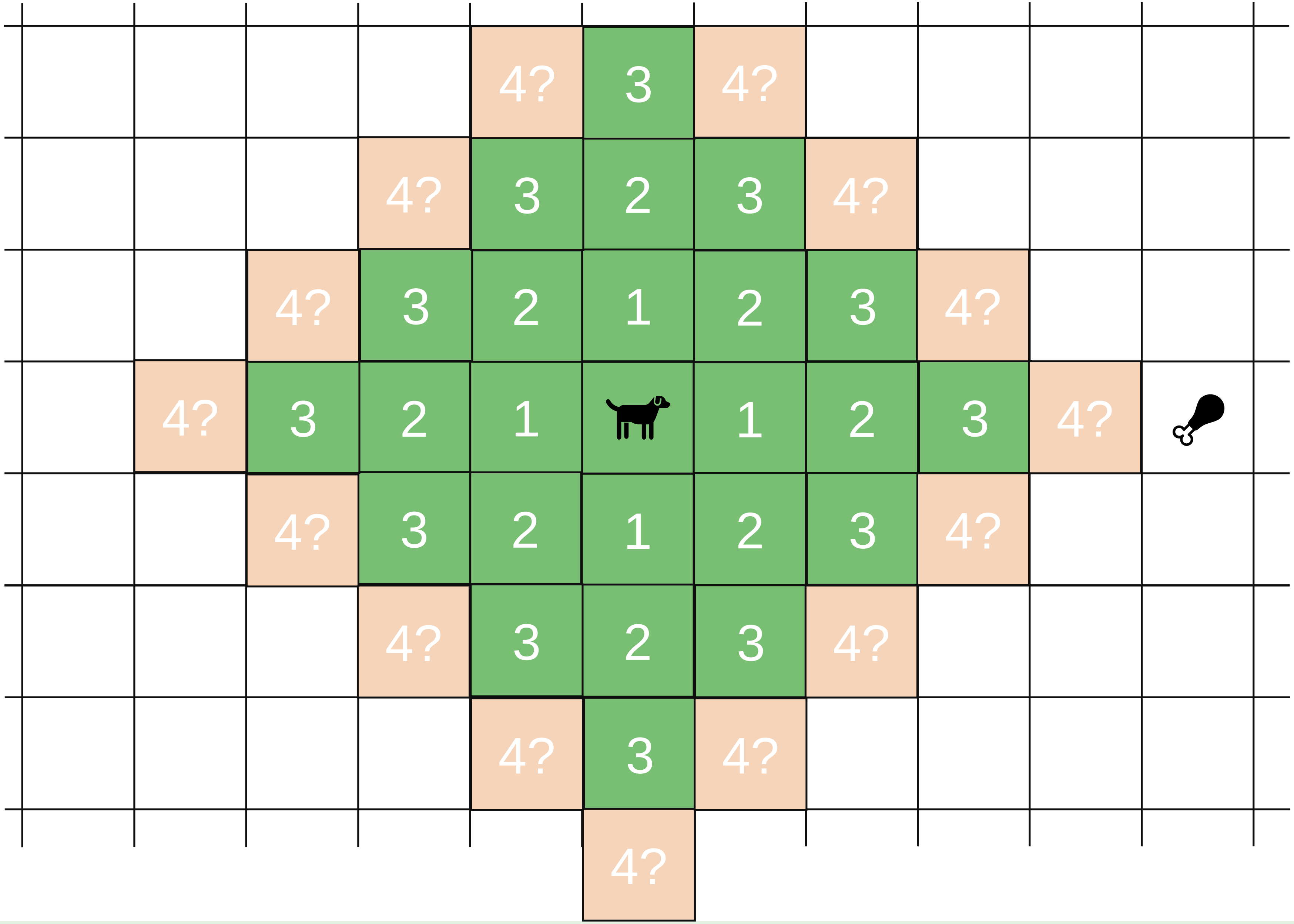
Dijkstra where each edge has cost 1



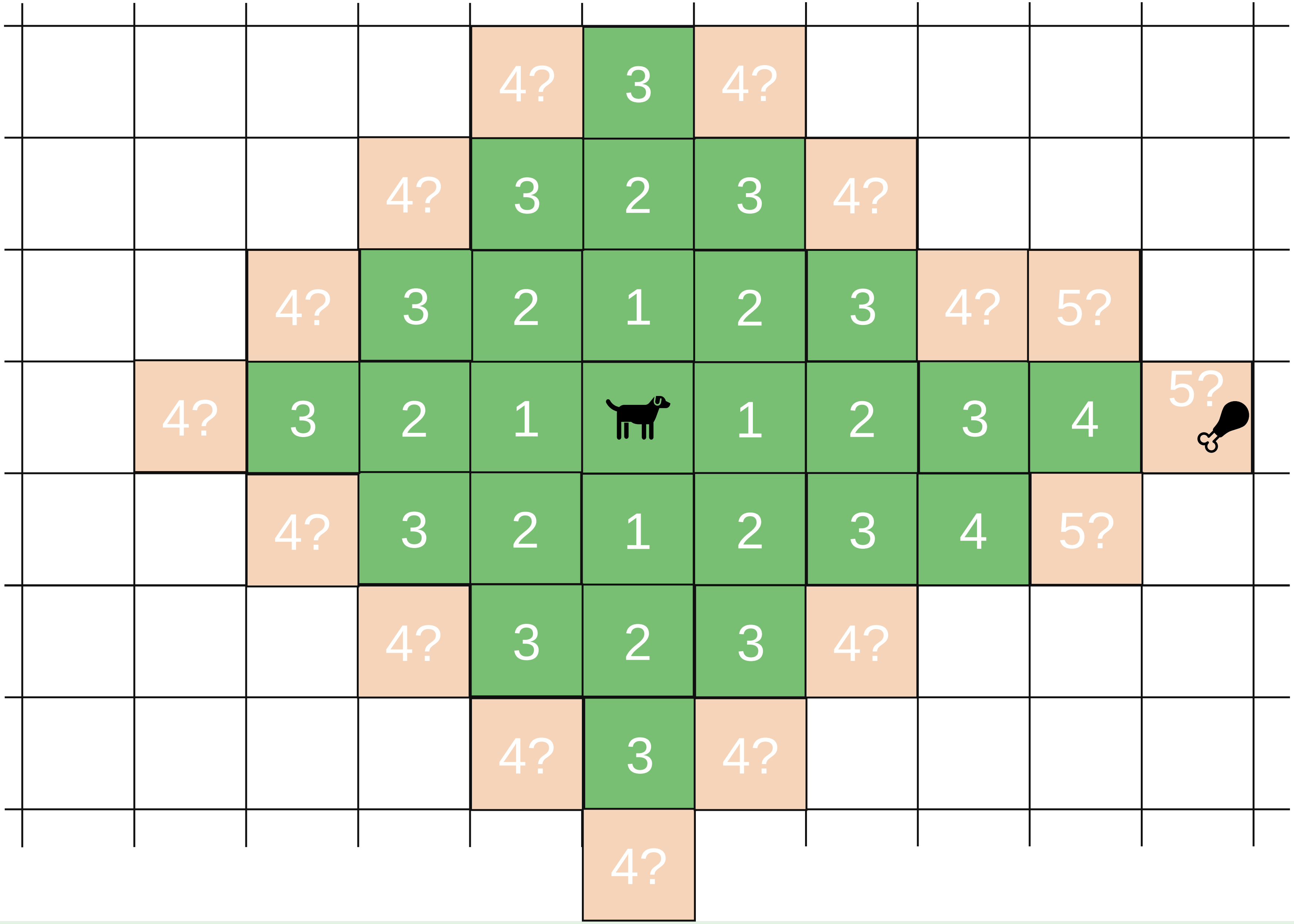
Dijkstra where each edge has cost 1



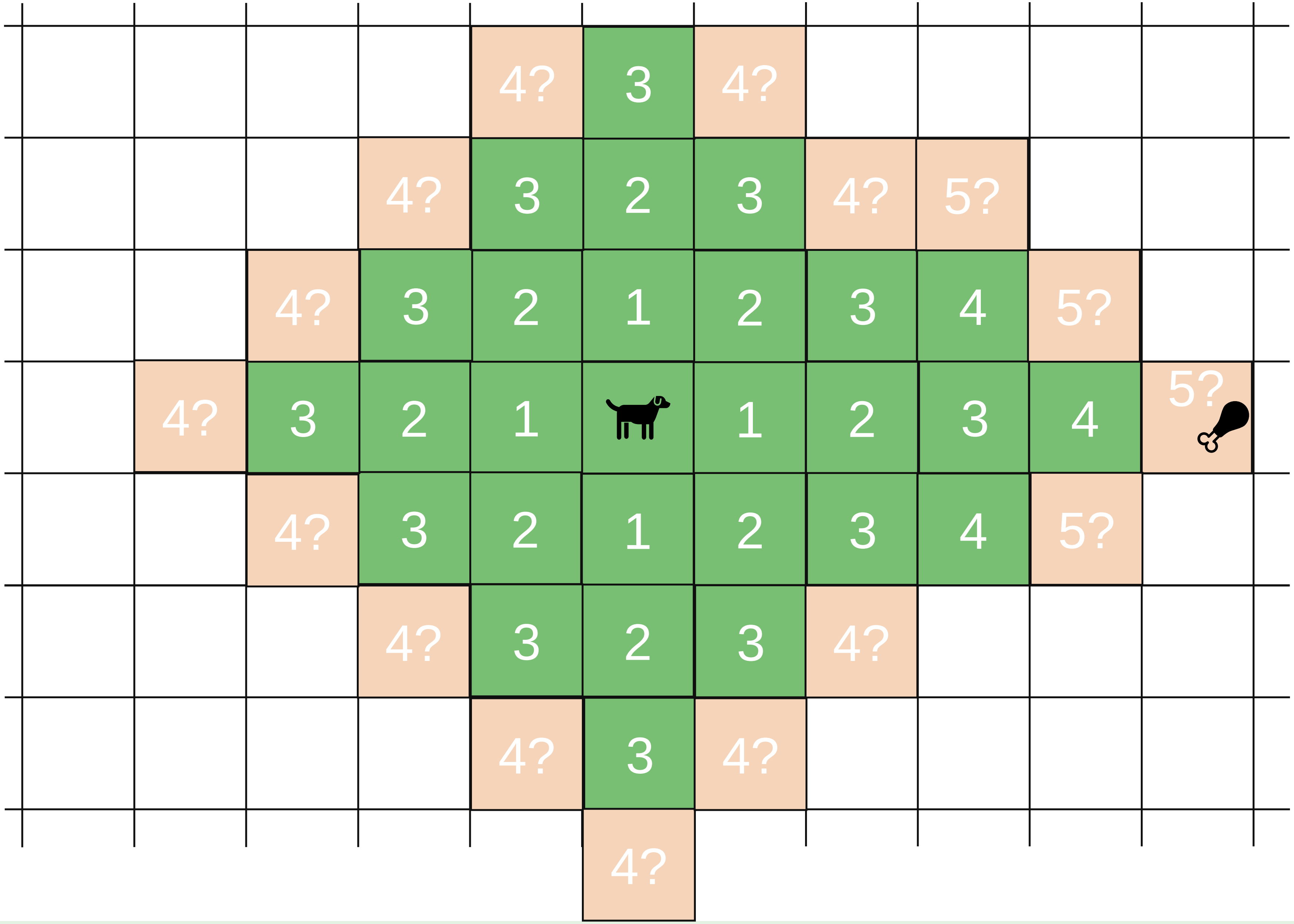
Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1

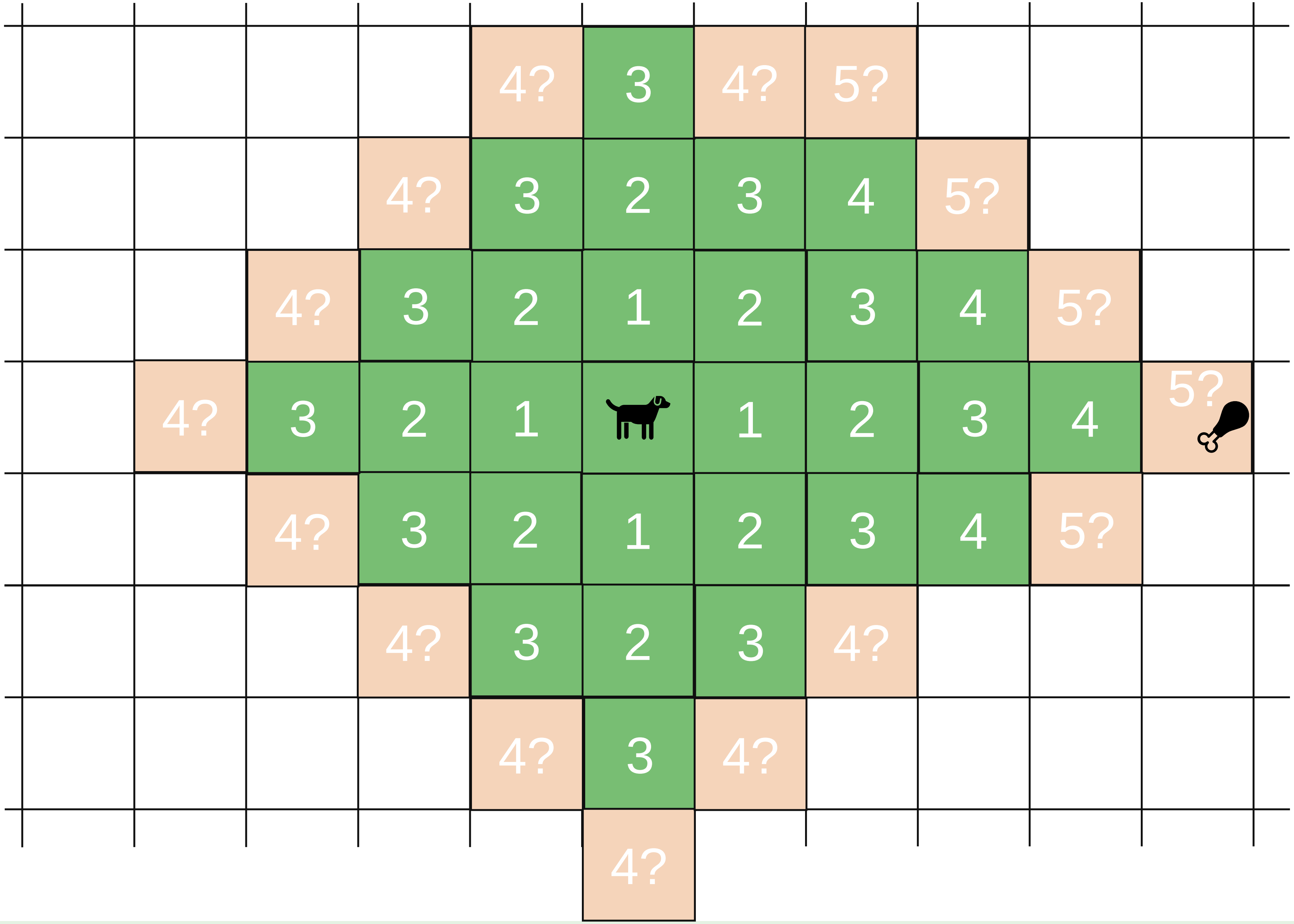


Dijkstra where each edge has cost 1

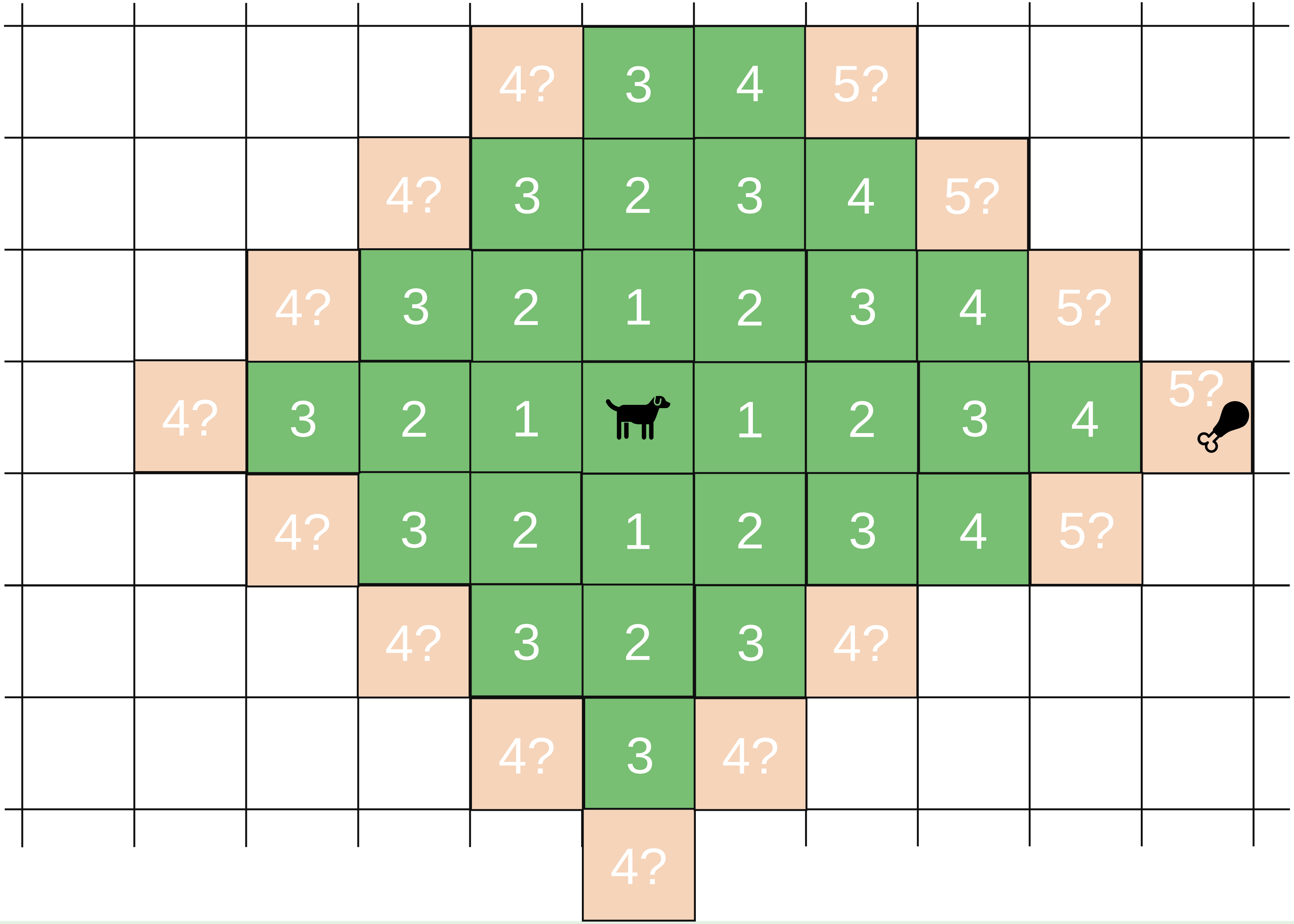


Yannic Maus

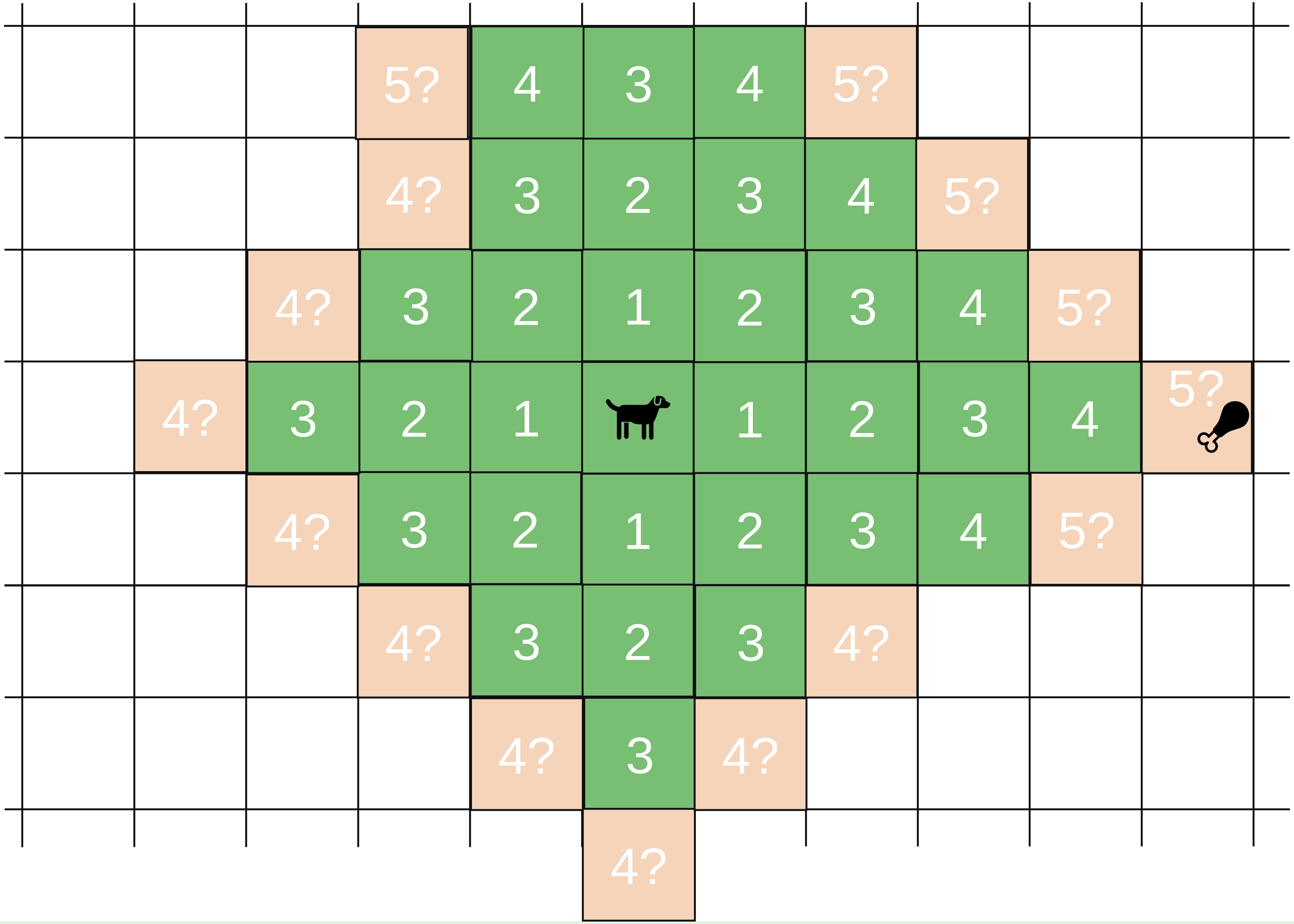
Dijkstra where each edge has cost 1



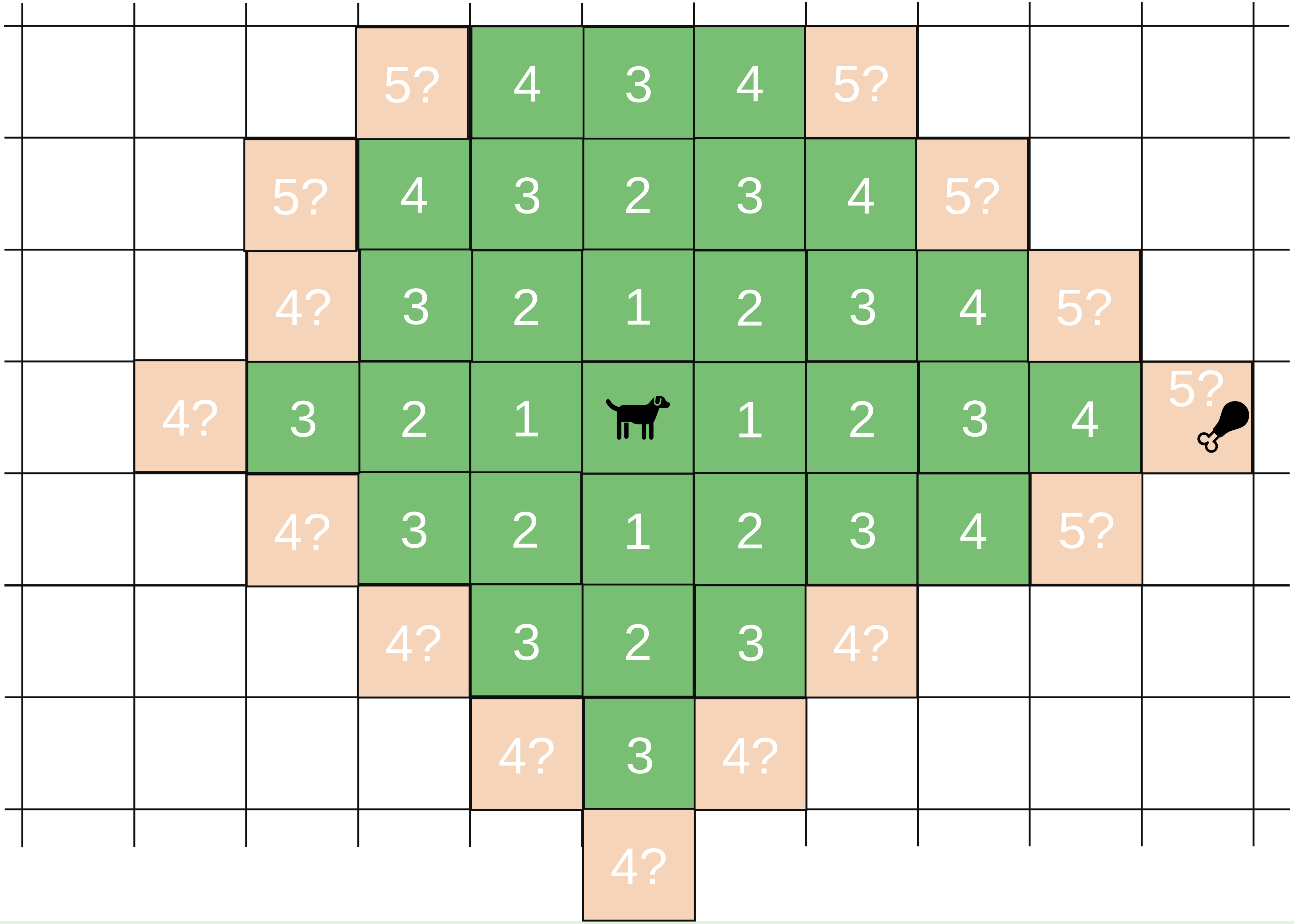
Dijkstra where each edge has cost 1



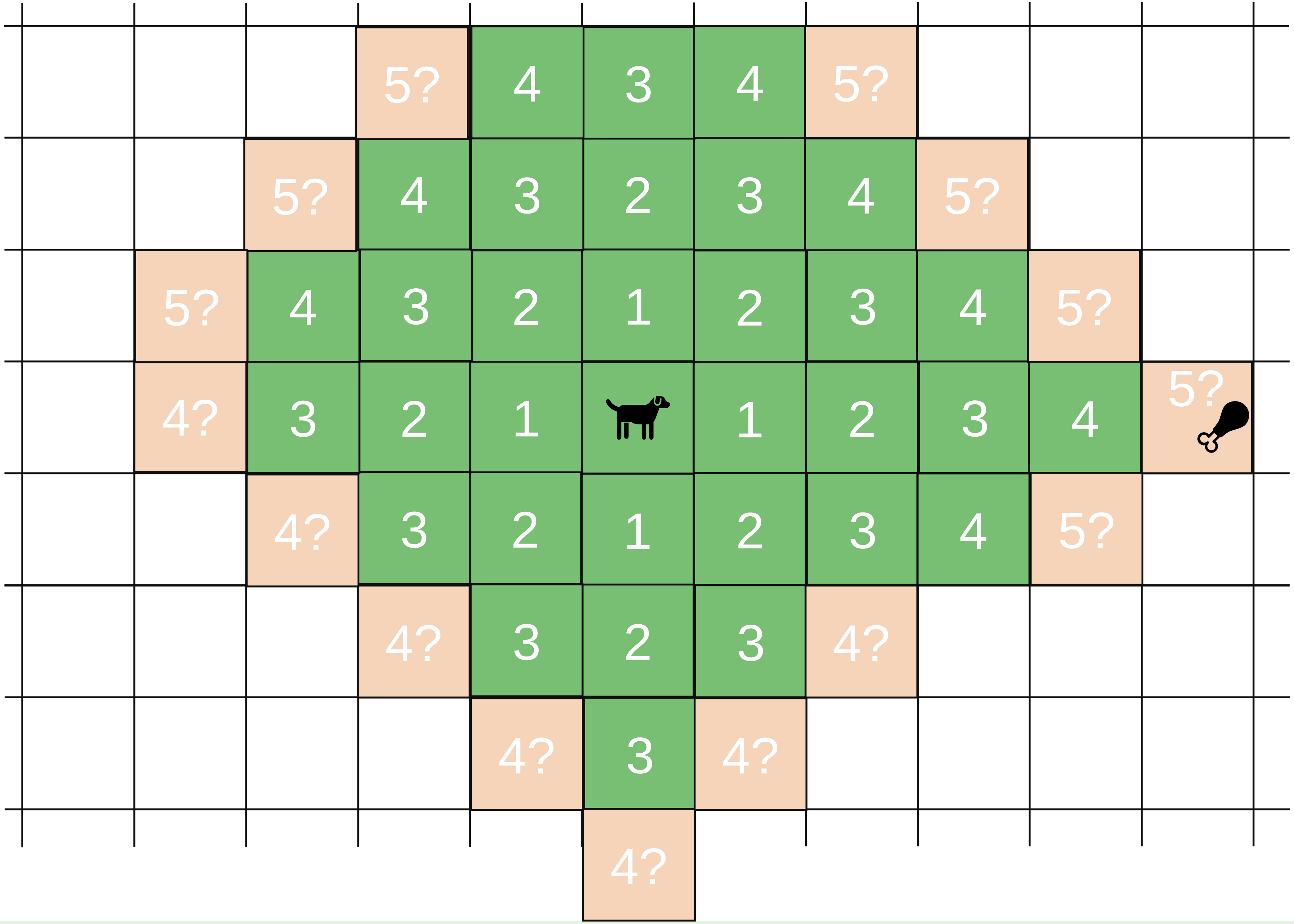
Dijkstra where each edge has cost 1



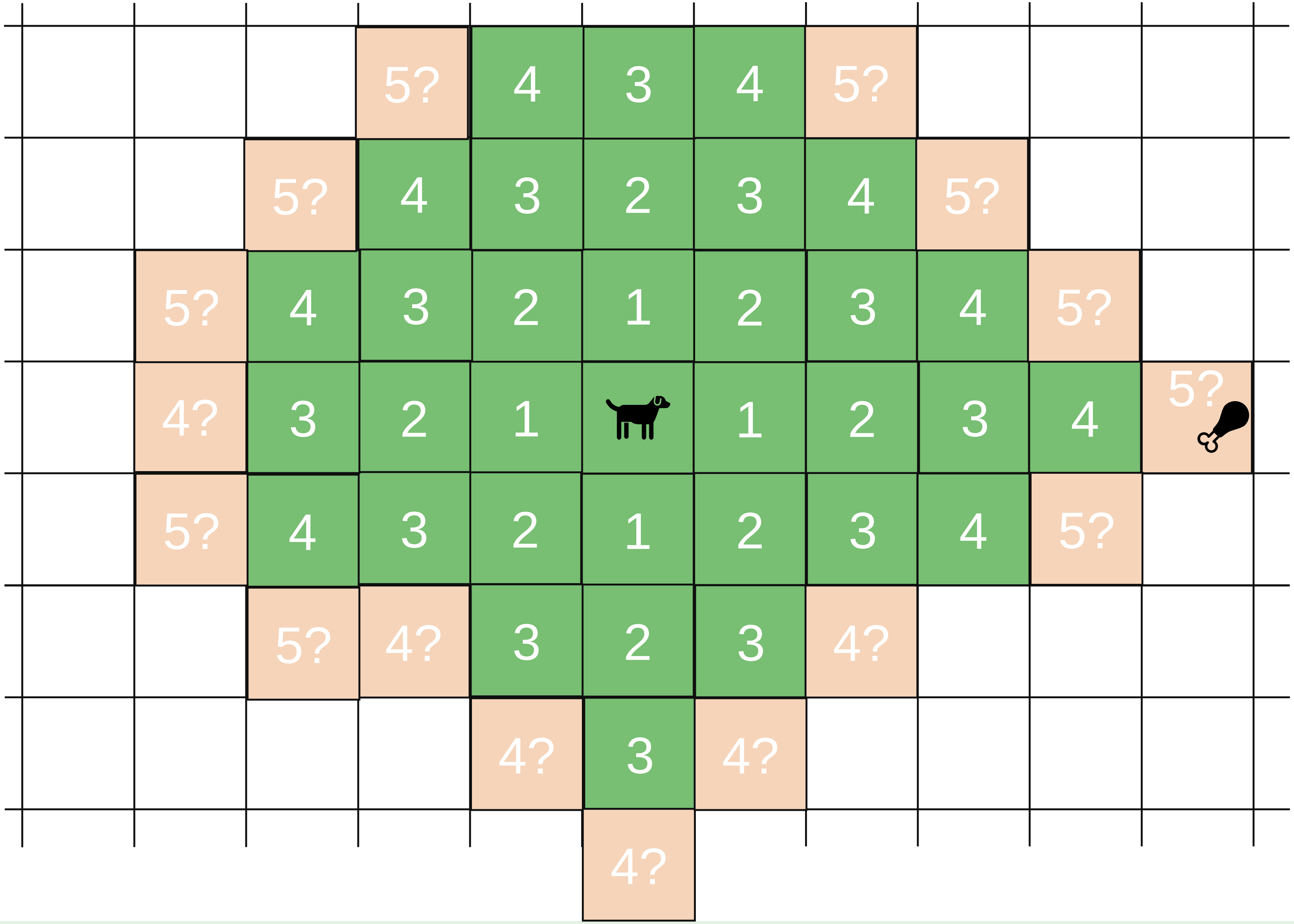
Dijkstra where each edge has cost 1



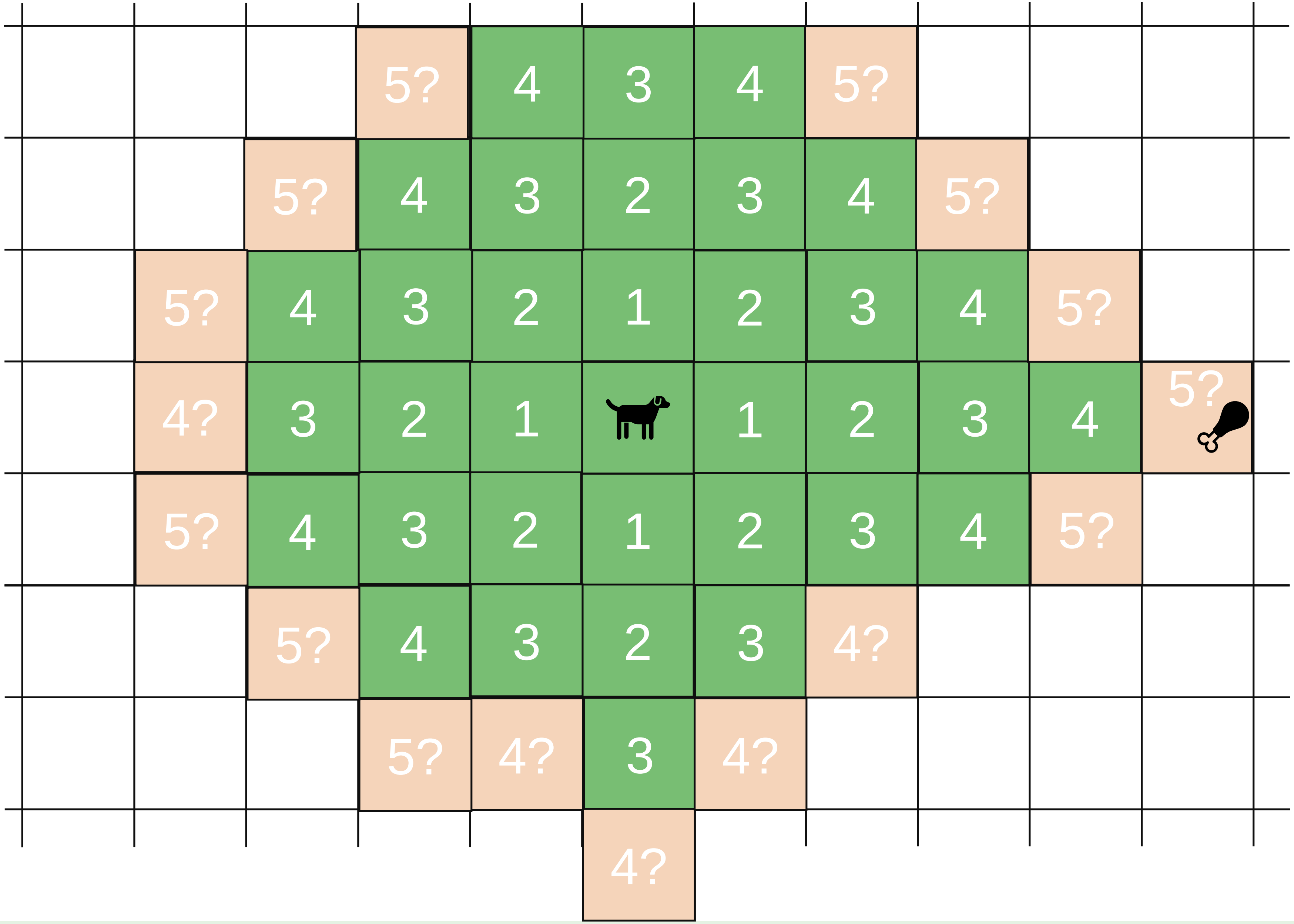
Dijkstra where each edge has cost 1



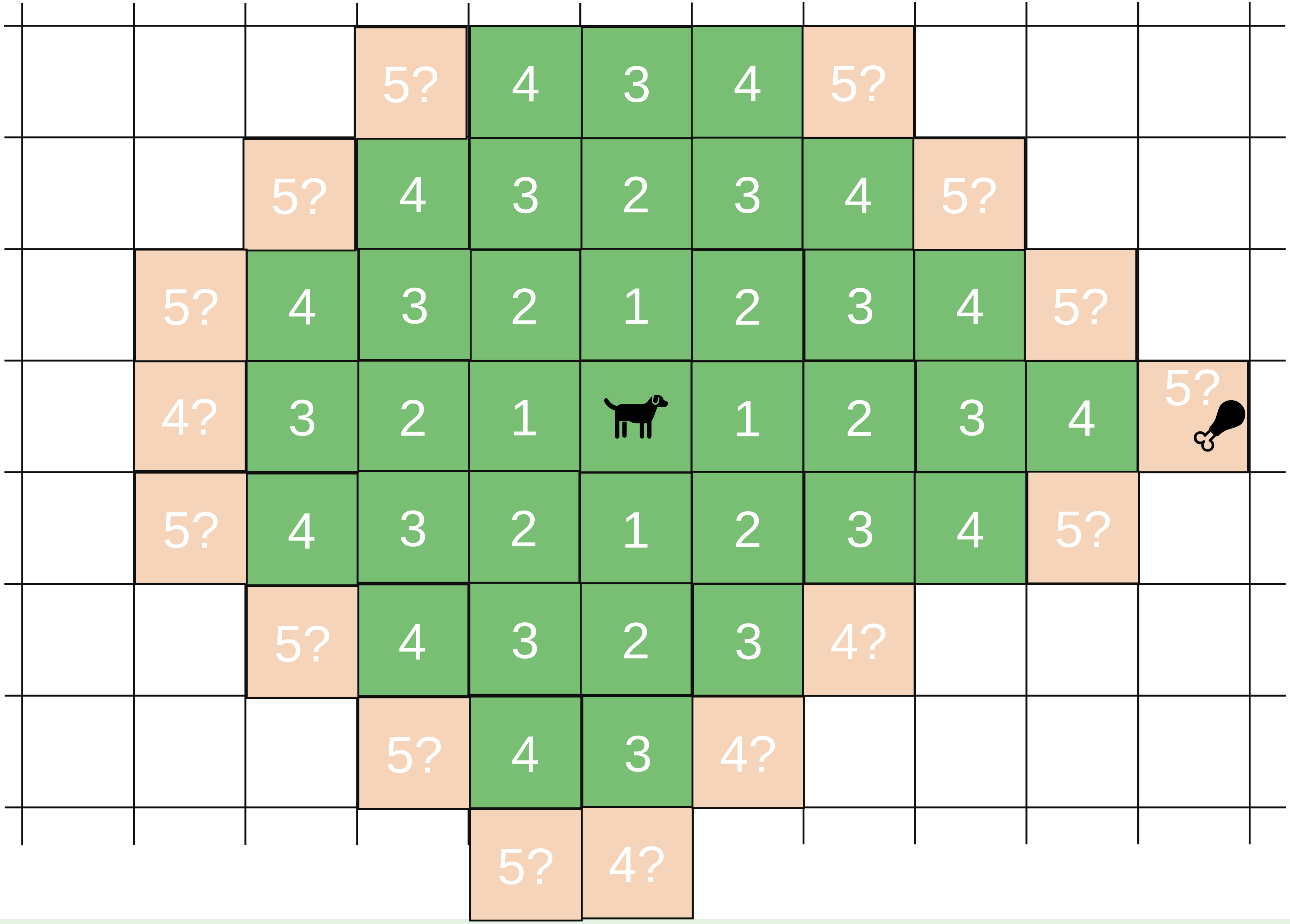
Dijkstra where each edge has cost 1



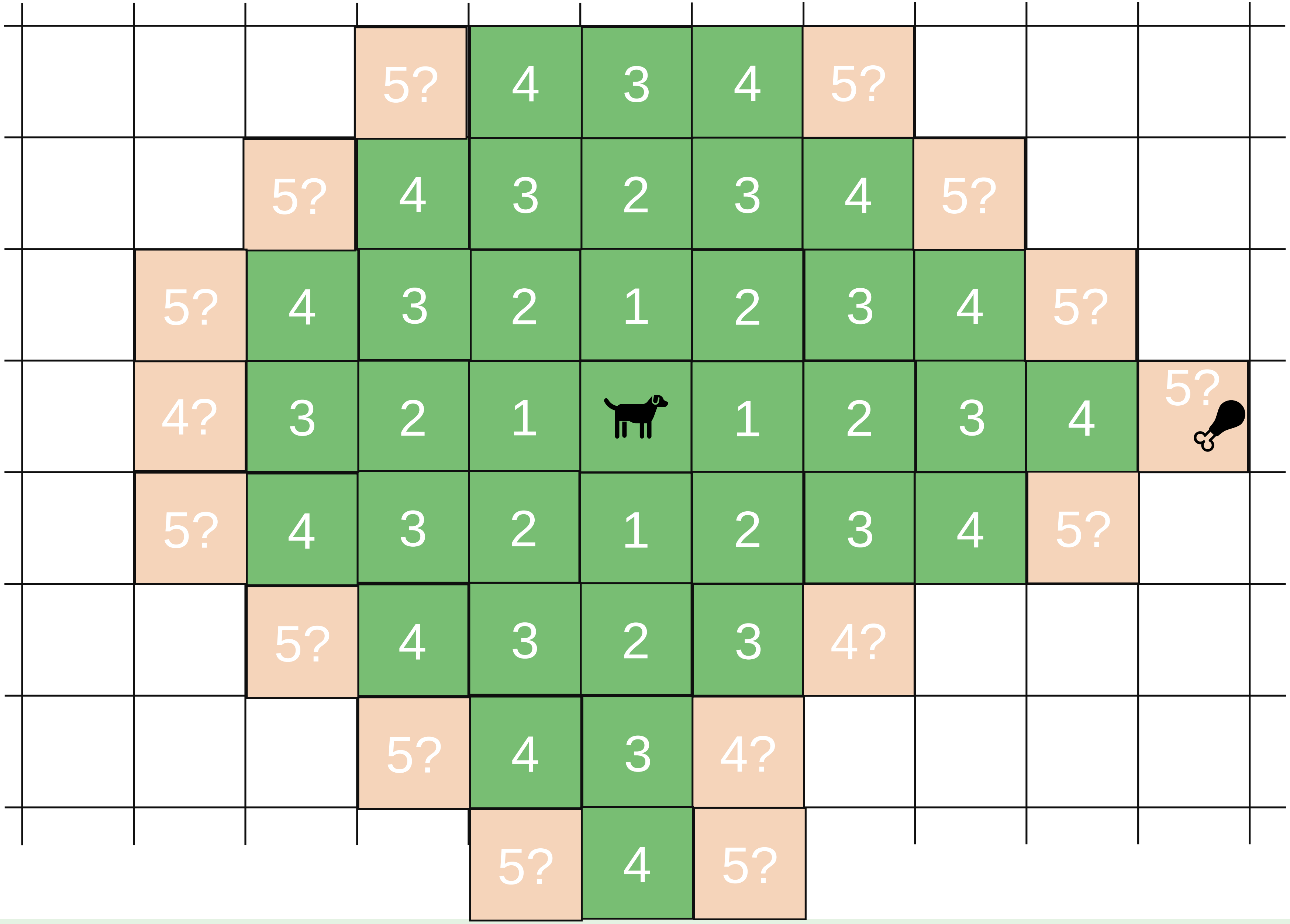
Dijkstra where each edge has cost 1



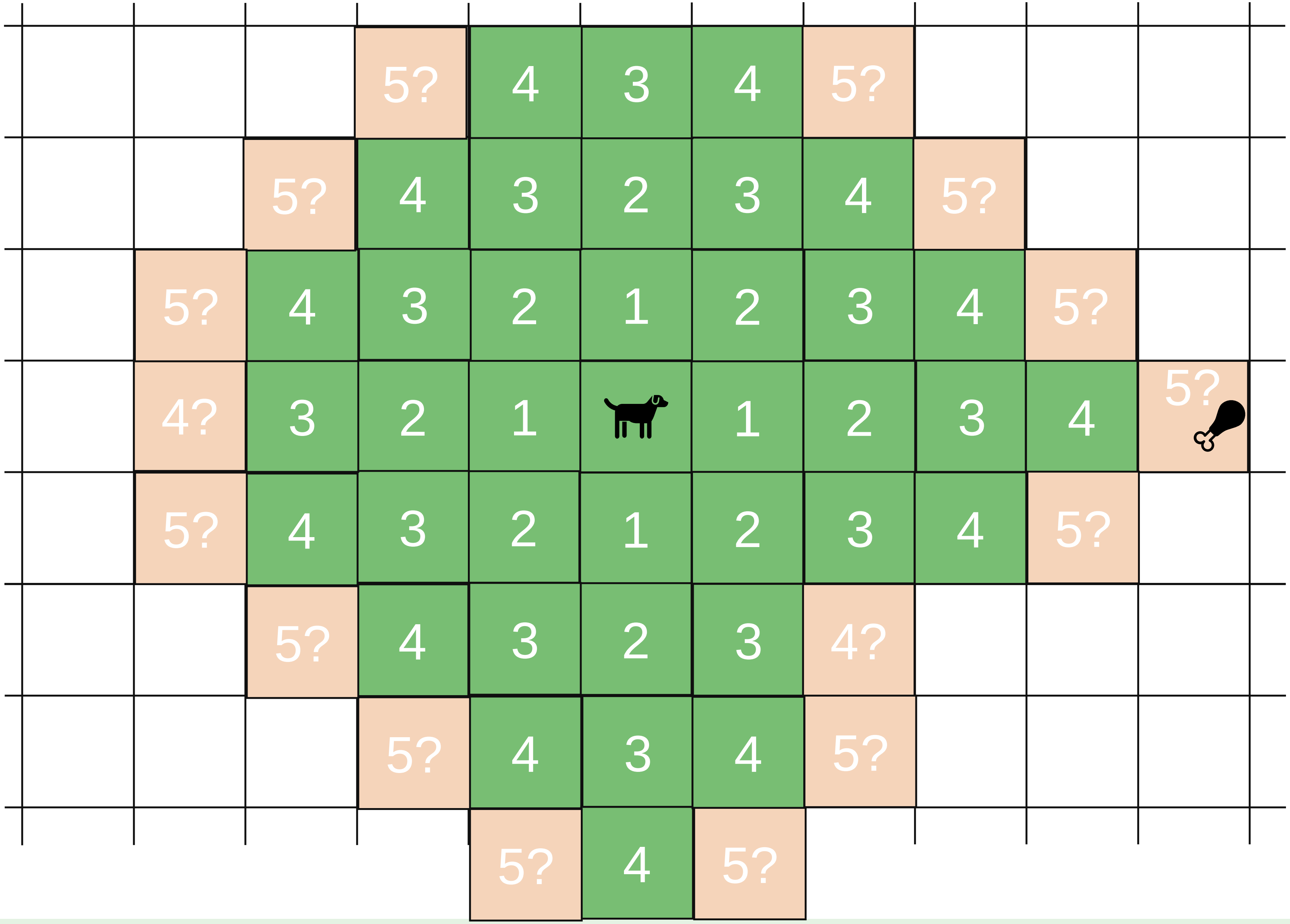
Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1



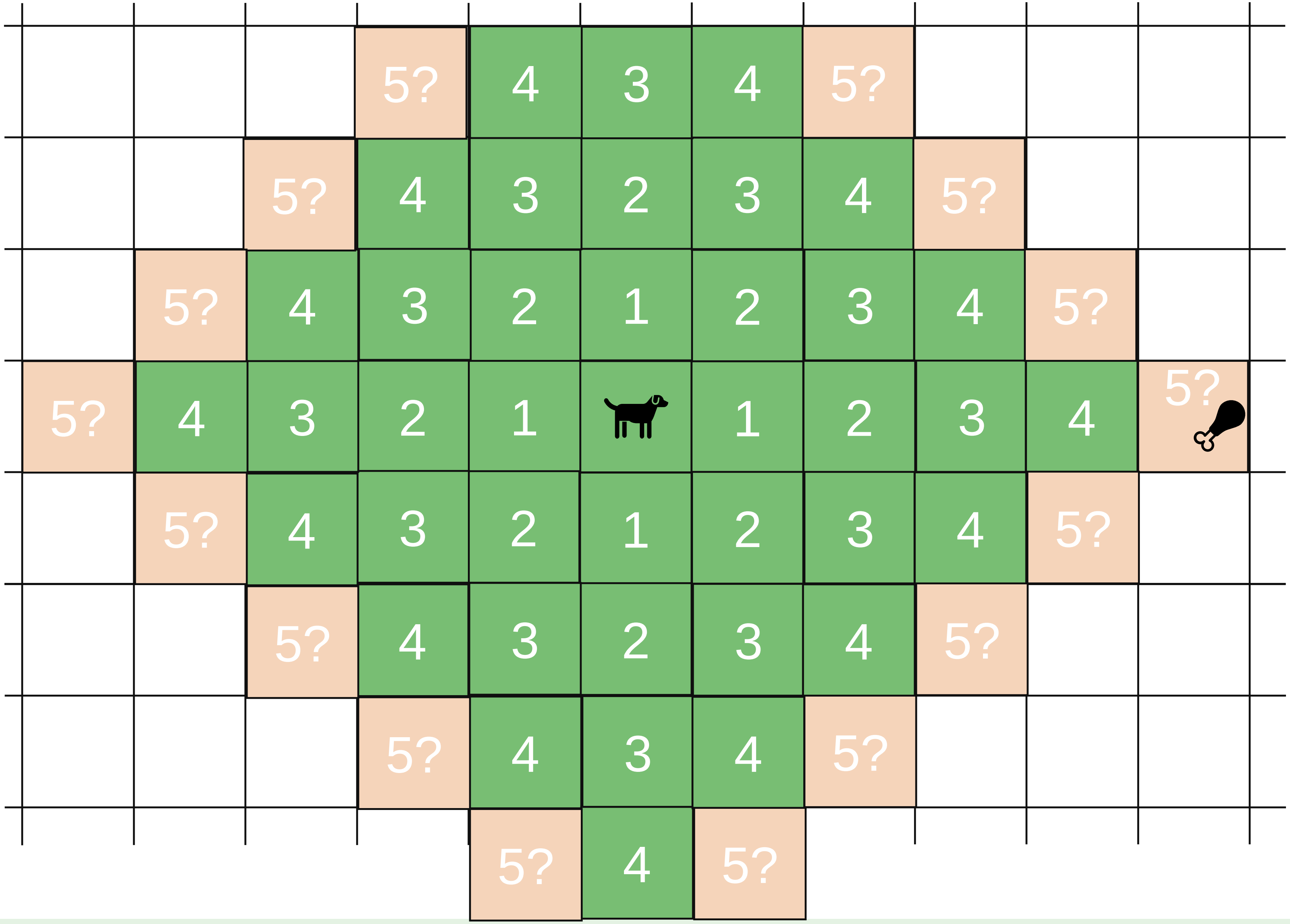
Dijkstra where each edge has cost 1



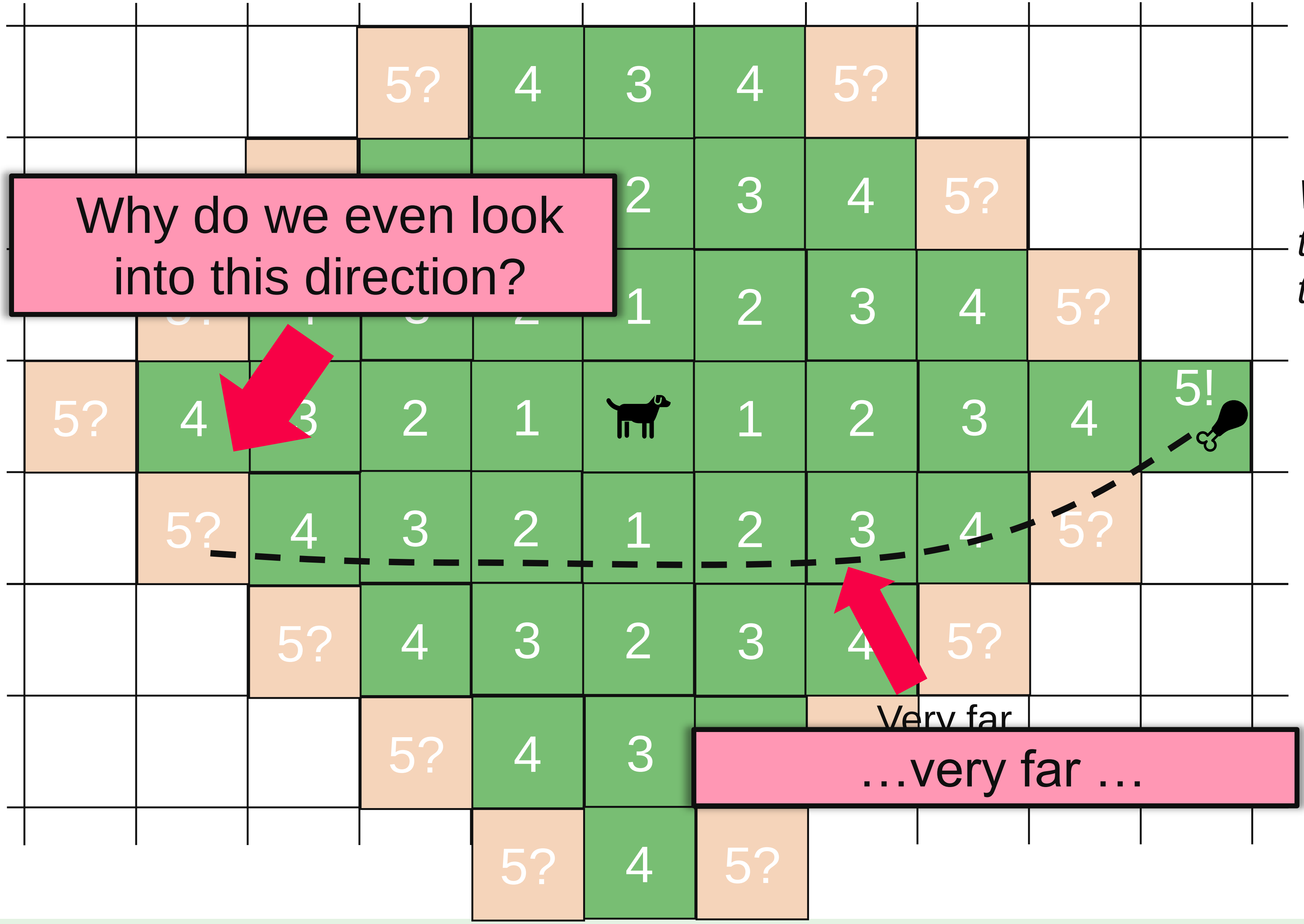
Dijkstra where each edge has cost 1

			5?	4	3	4	5?			
		5?	4	3	2	3	4	5?		
	5?	4	3	2	1	2	3	4	5?	
	4?	3	2	1		1	2	3	4	5?
	5?	4	3	2	1	2	3	4	5?	
		5?	4	3	2	3	4	5?		
			5?	4	3	4	5?			
				5?	4	5?				

Dijkstra where each edge has cost 1



Dijkstra where each edge has cost 1

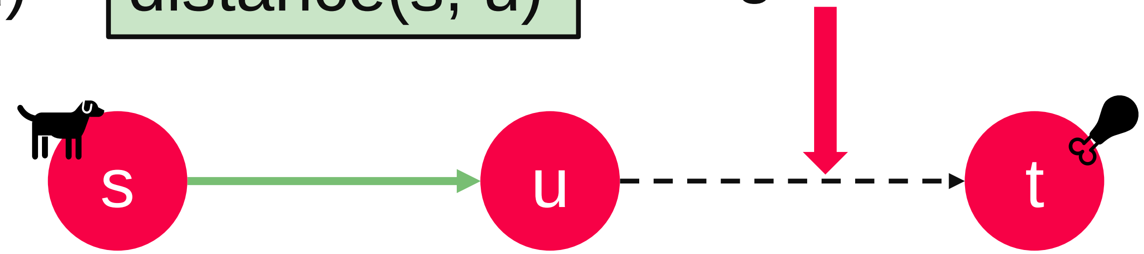


We haven't taken into account the cost from that node to the target

Dijkstra Priority

priority(u) = $\boxed{\text{distance(s, u)}}$

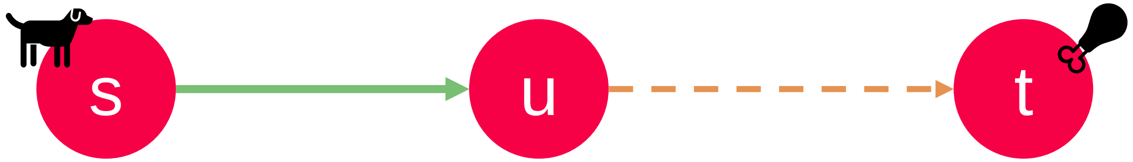
ignored



priority of the path that ends in u

Ideal Priority

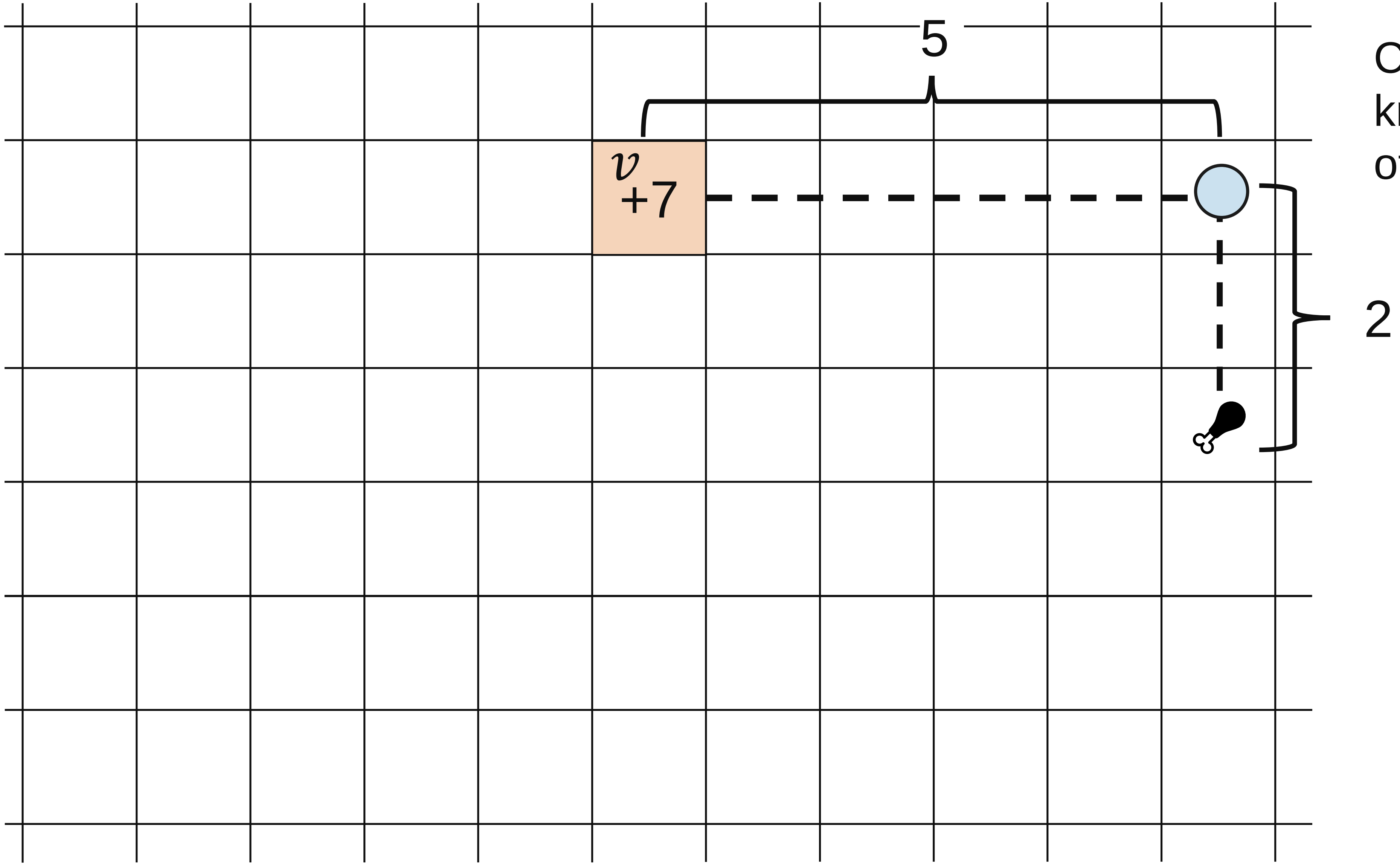
$$\text{priority}(u) = \boxed{\text{distance}(s, u)} + \boxed{\text{futureCost}(u,t)}^{*}$$



priority of the path that ends in u

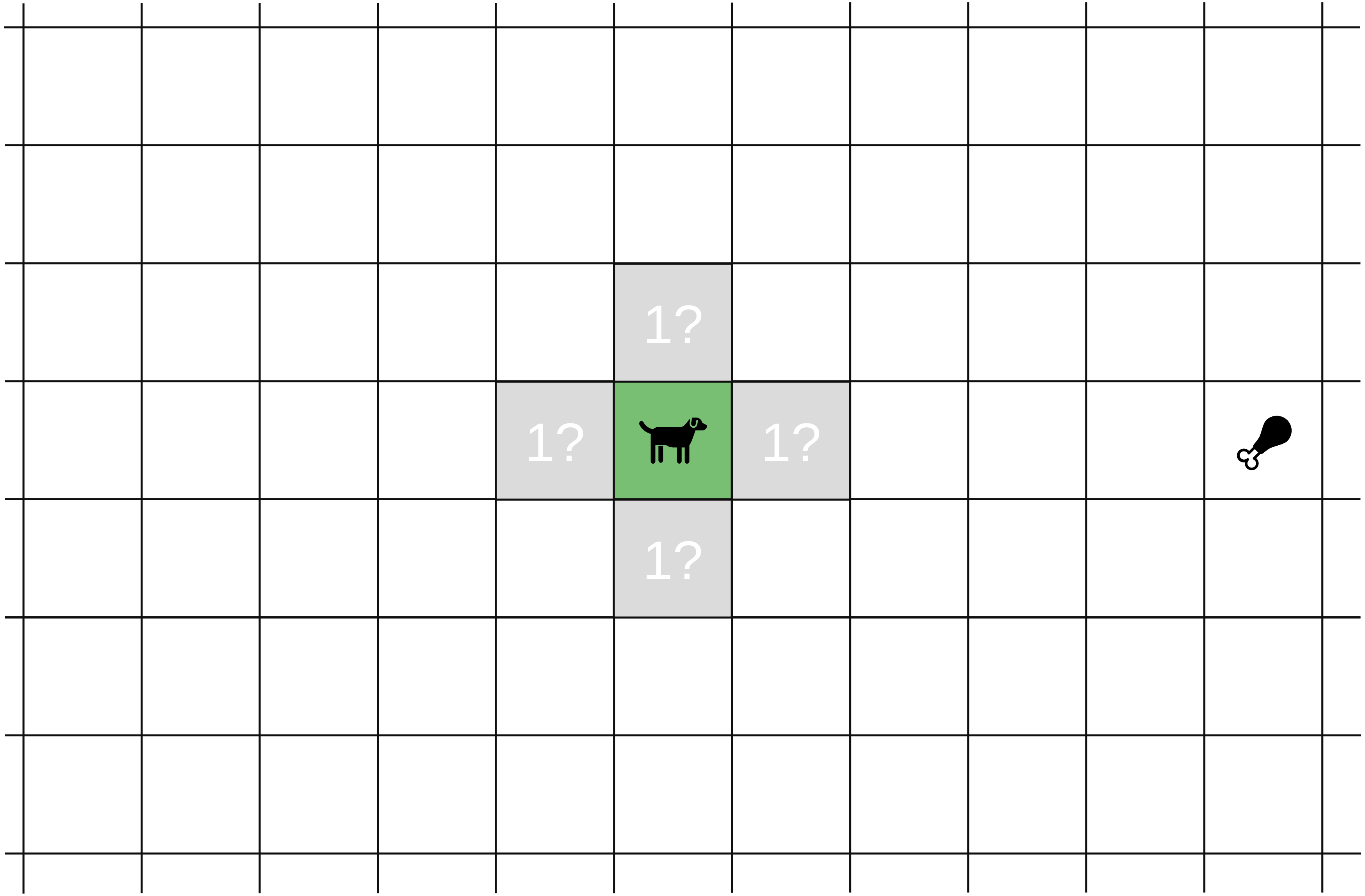
*needs a revision, often we do not know $\boxed{\text{futureCost}(u,t)}$

A*: Future cost = row + column distance



Can be computed if we know row and column of the target t and of v

A*: Future cost = row + column distance



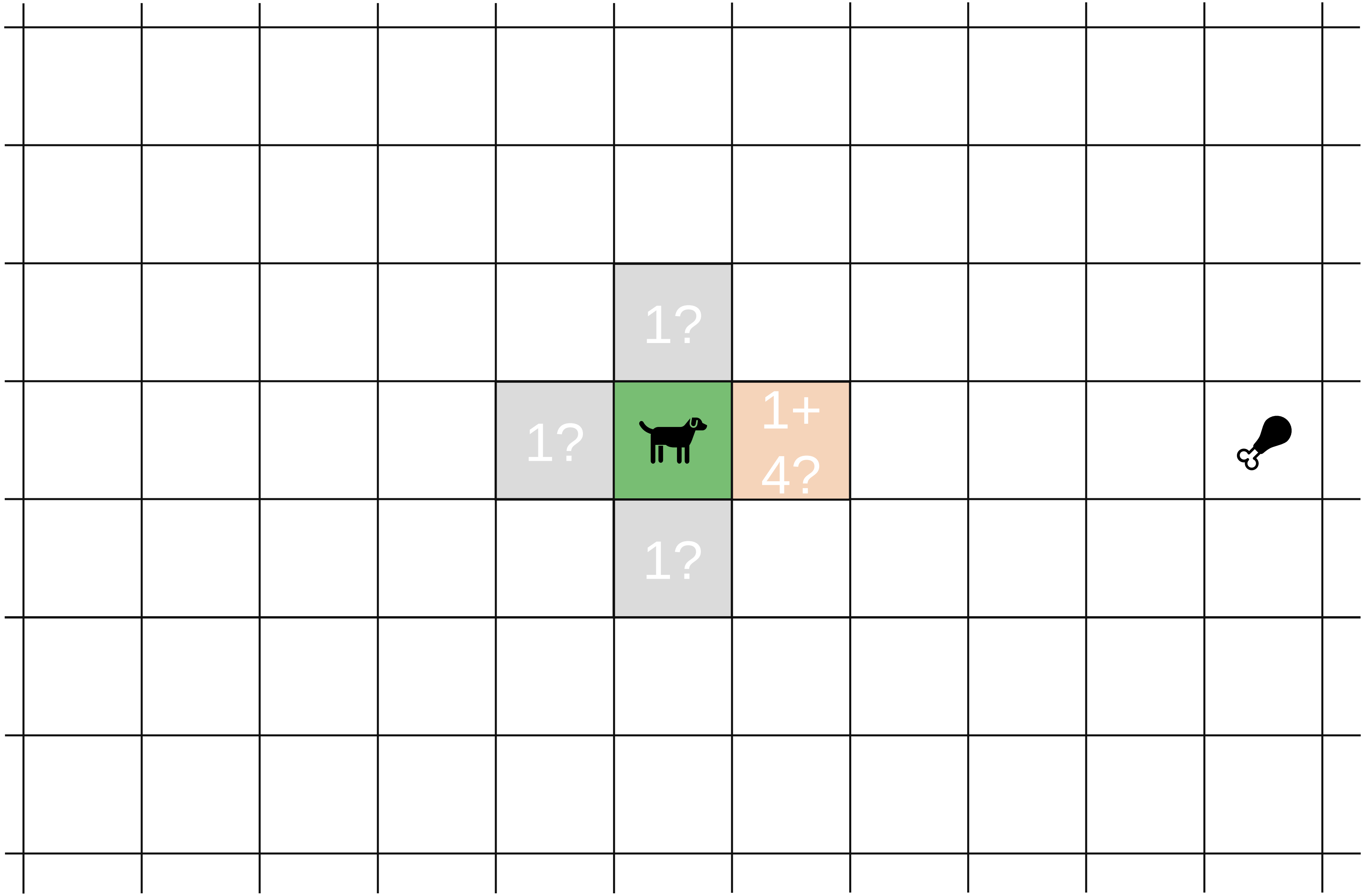
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



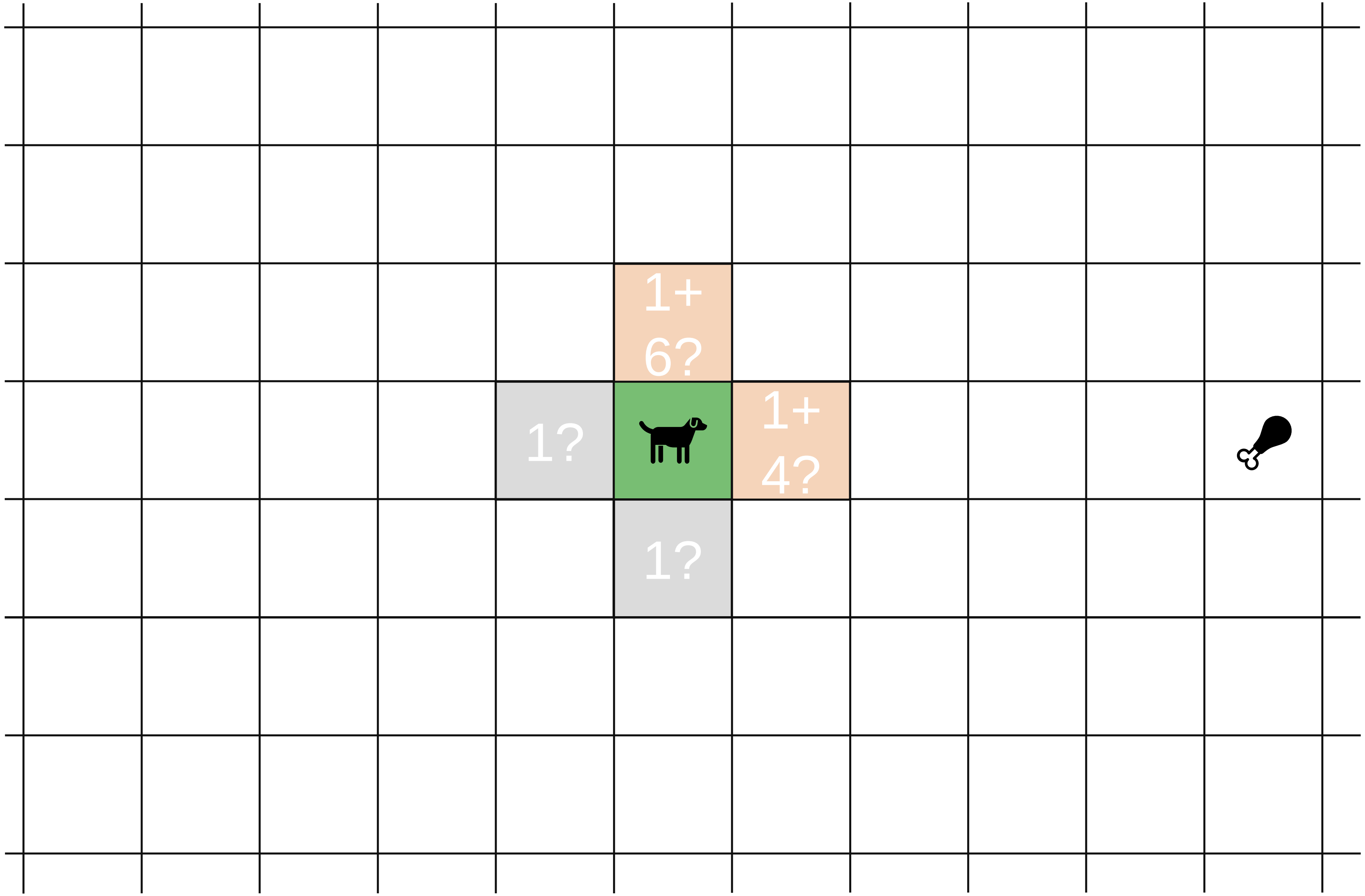
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



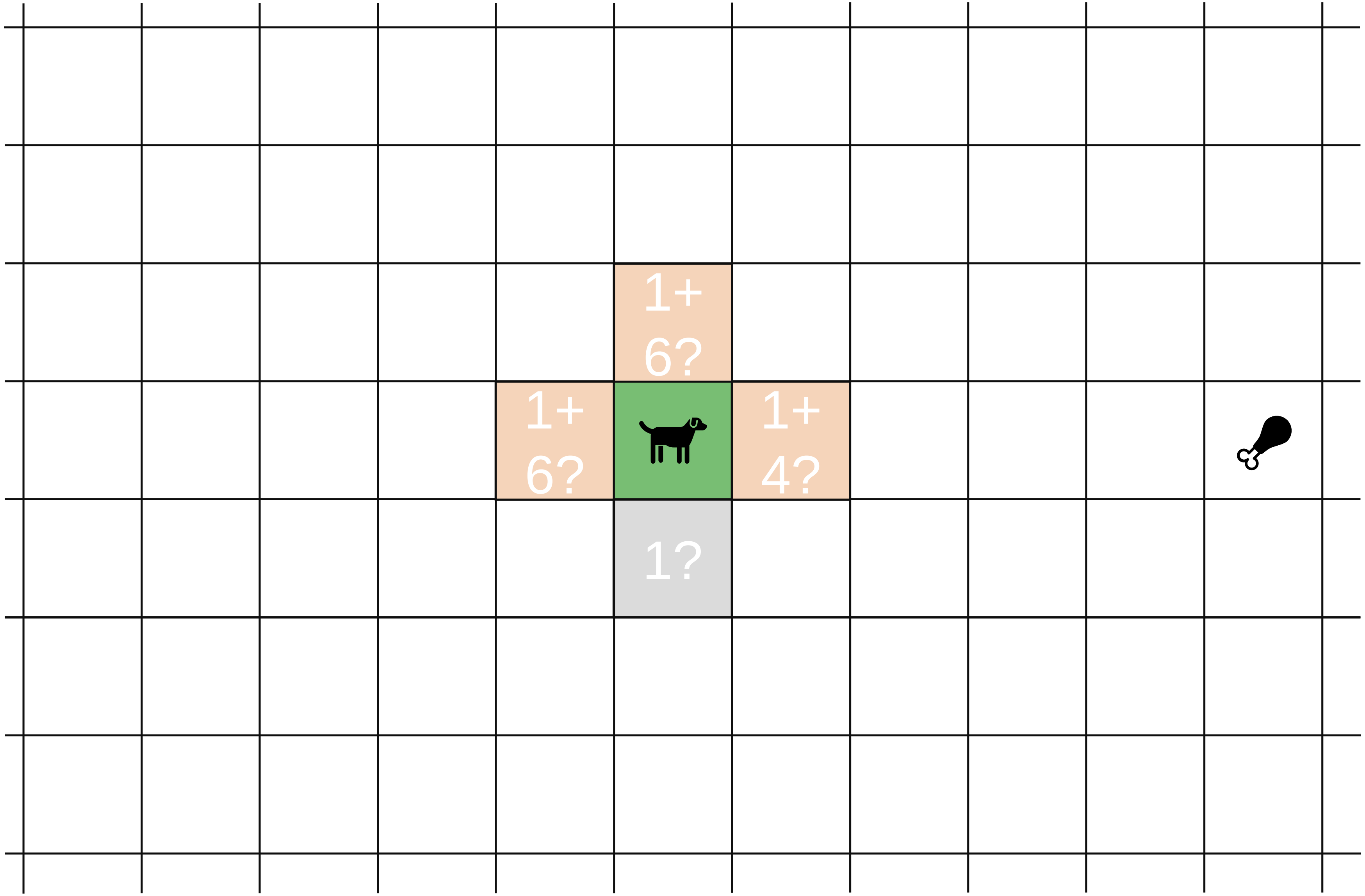
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



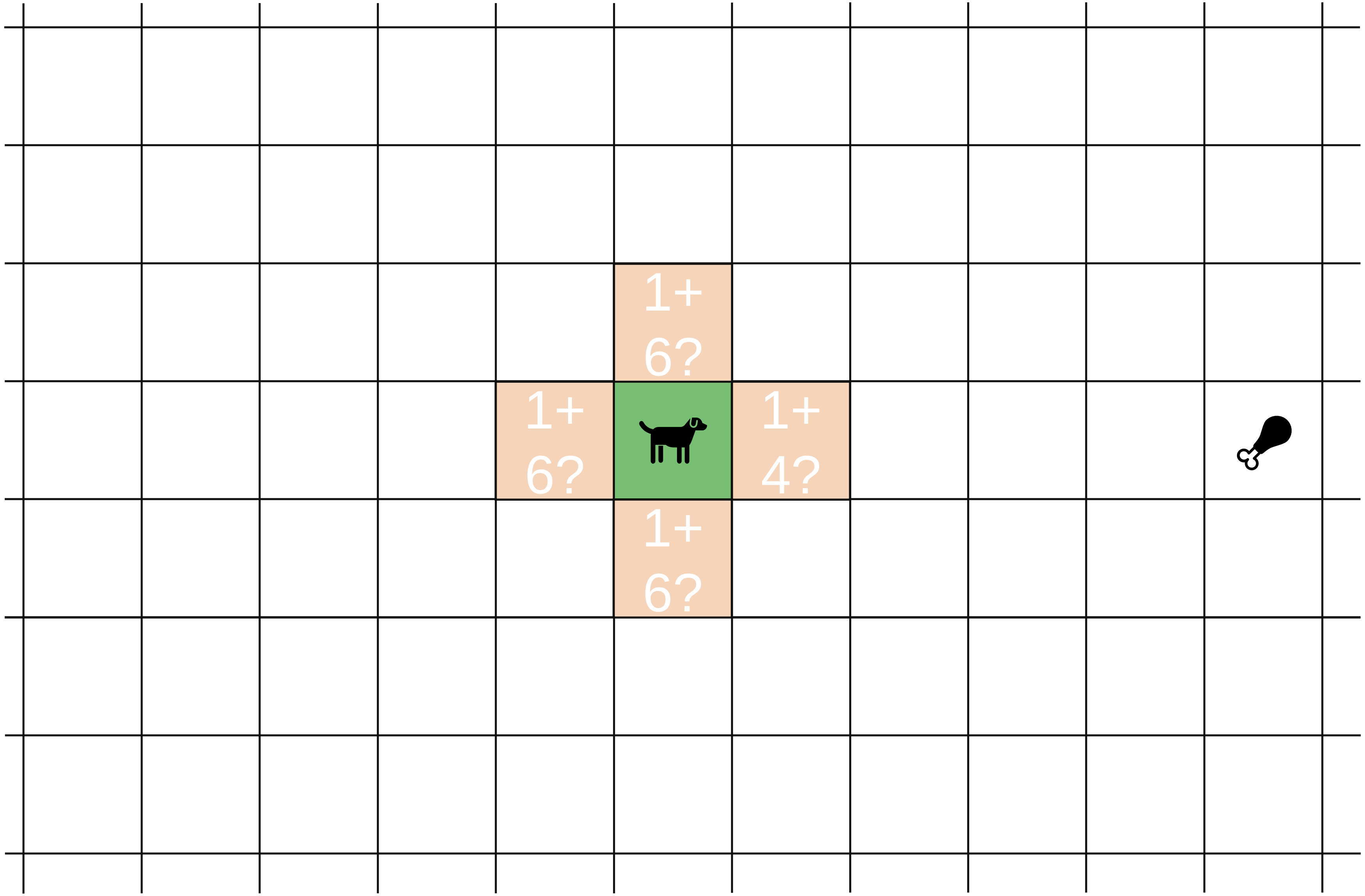
Greedily continue at smallest

$\boxed{\text{distance(s, u)}}$

+

$\boxed{\text{futureCost(u,t)}}$

A*: Future cost = row + column distance



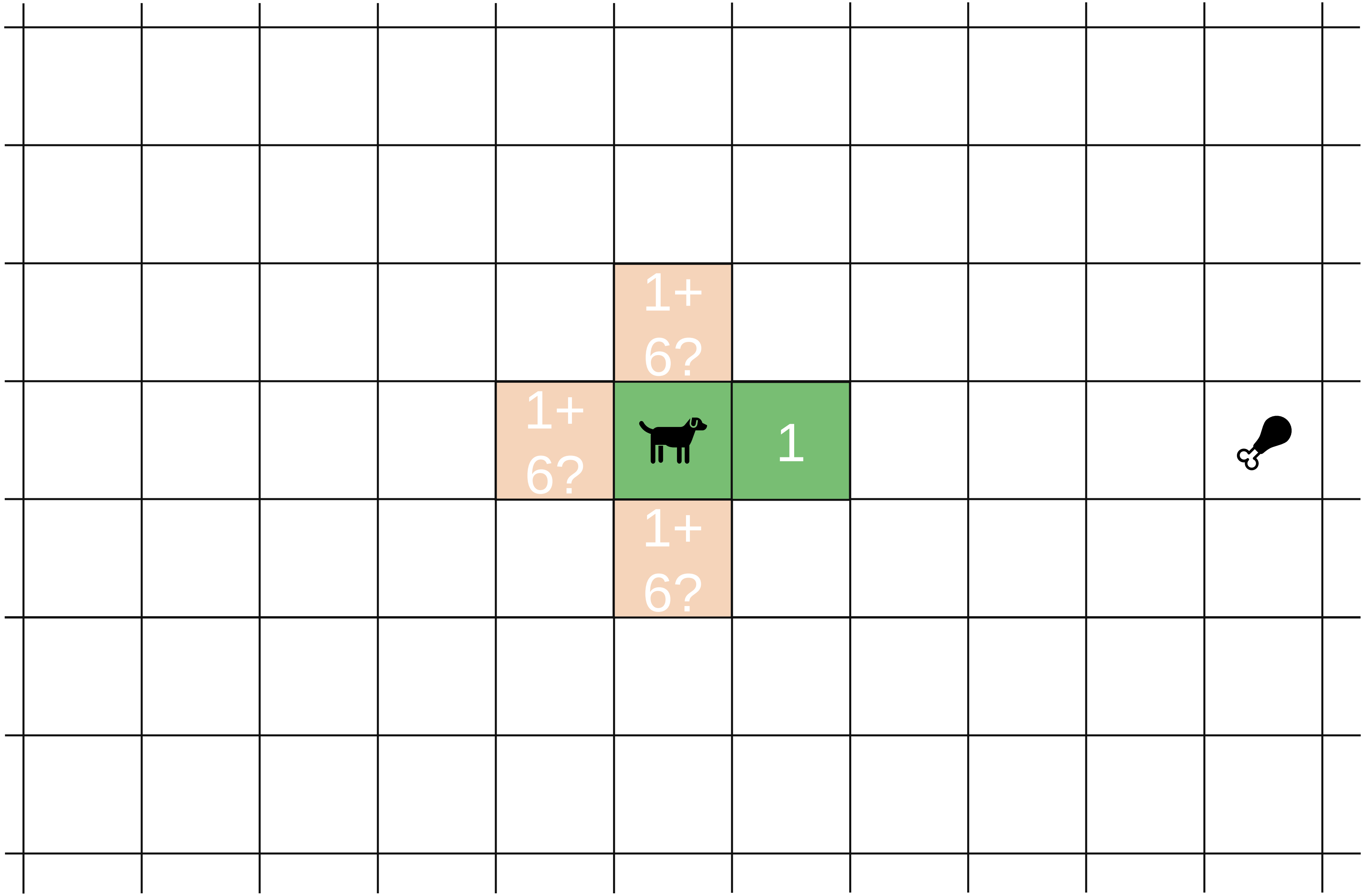
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



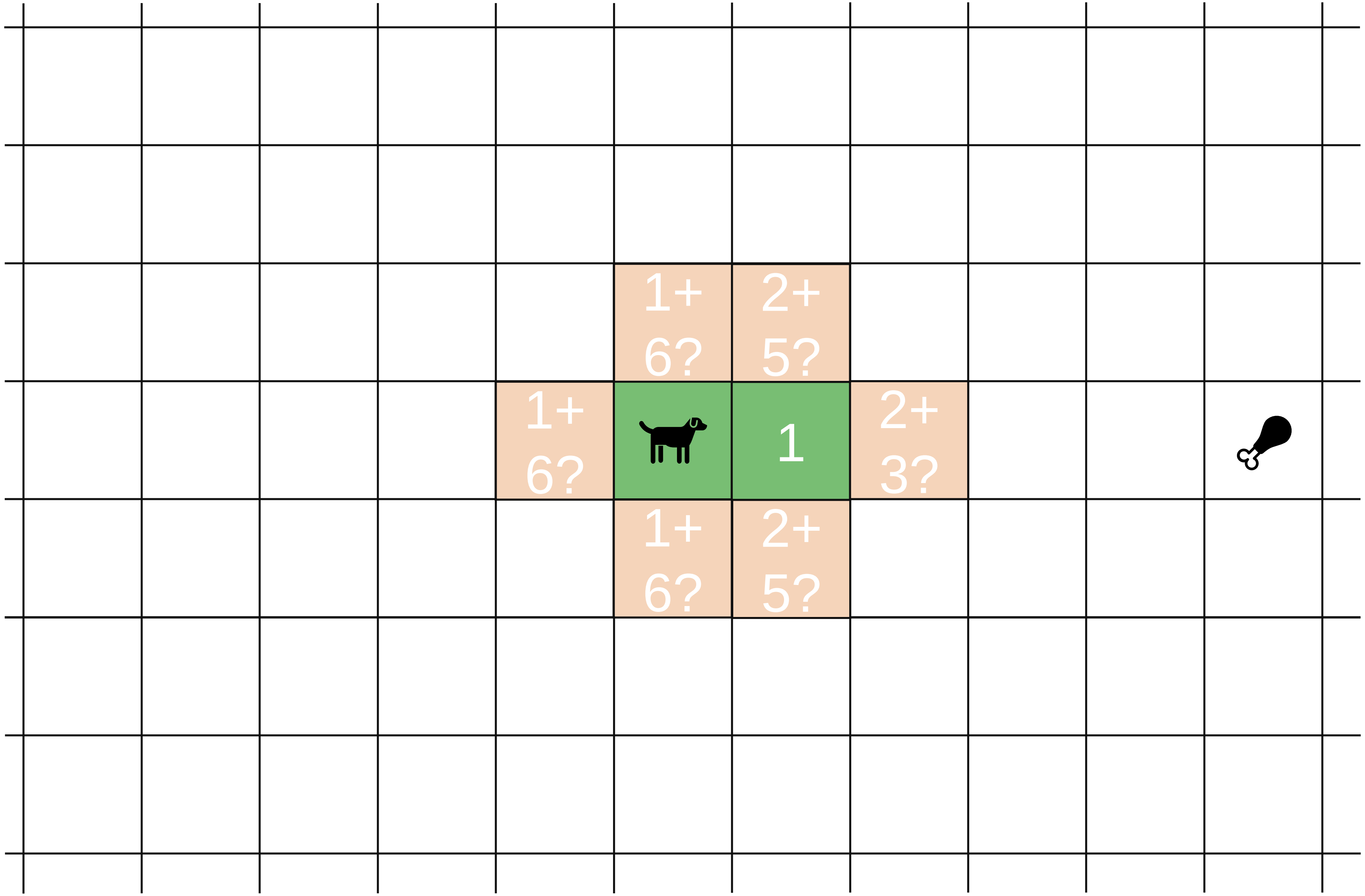
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



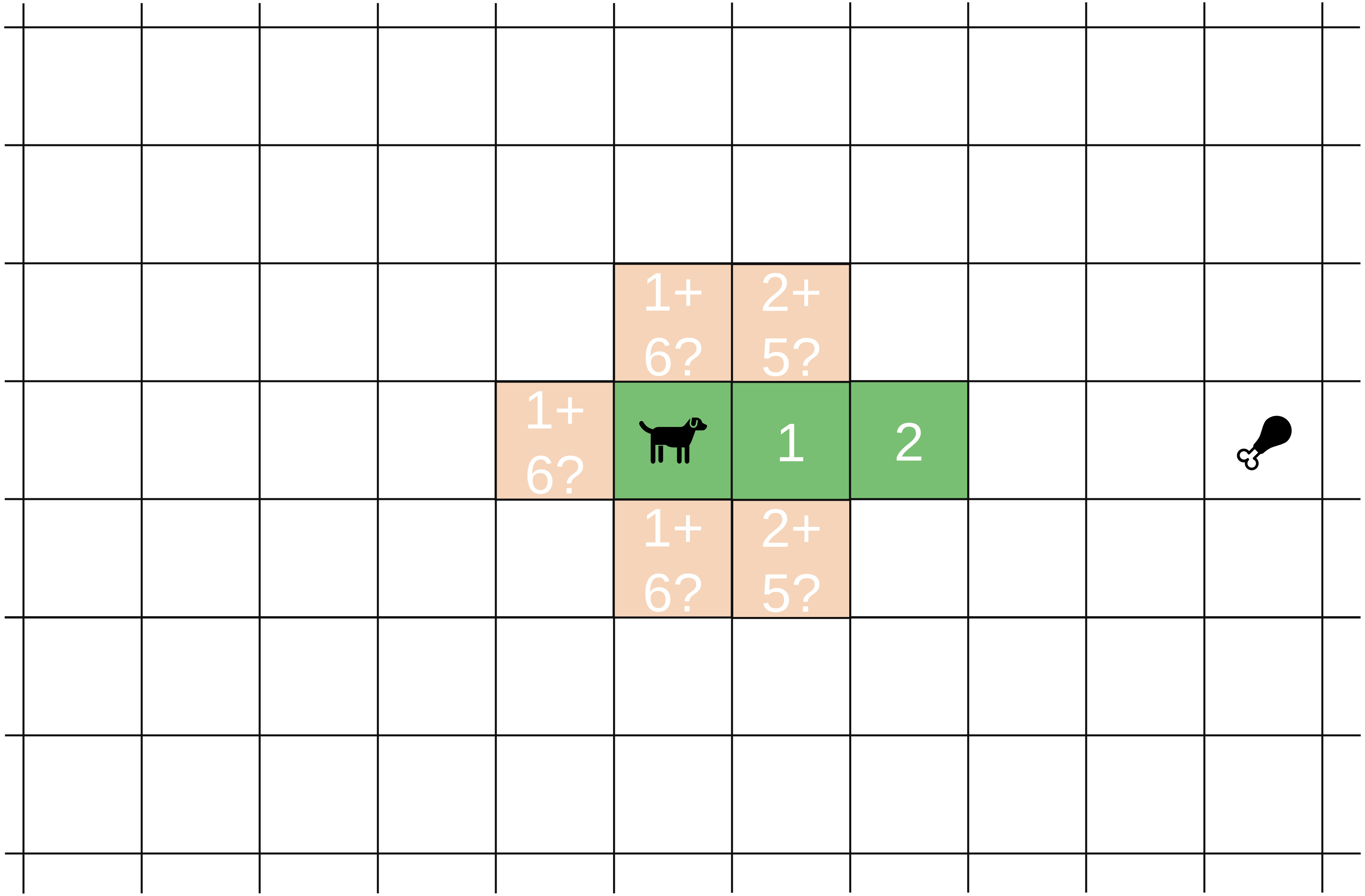
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



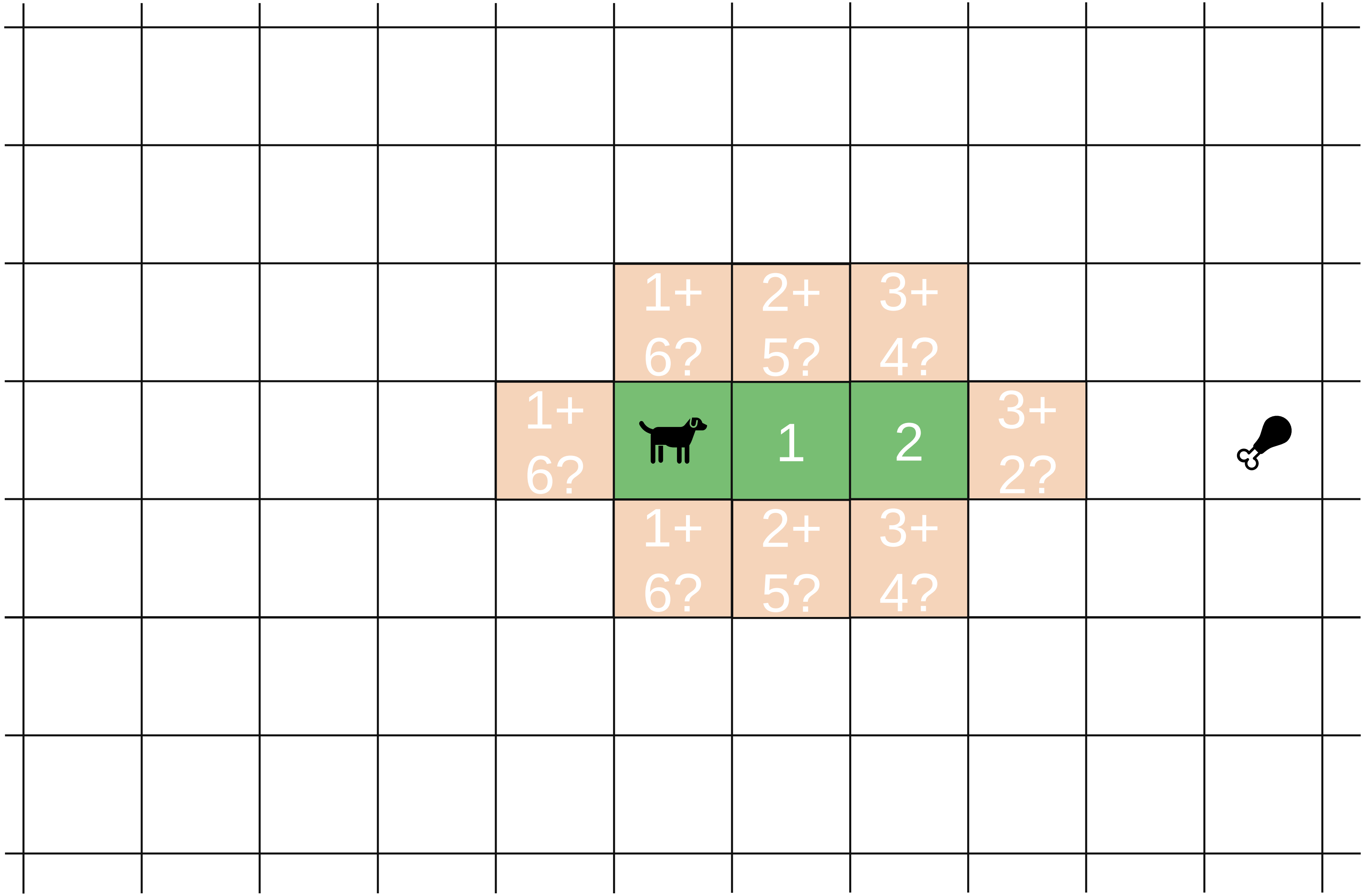
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



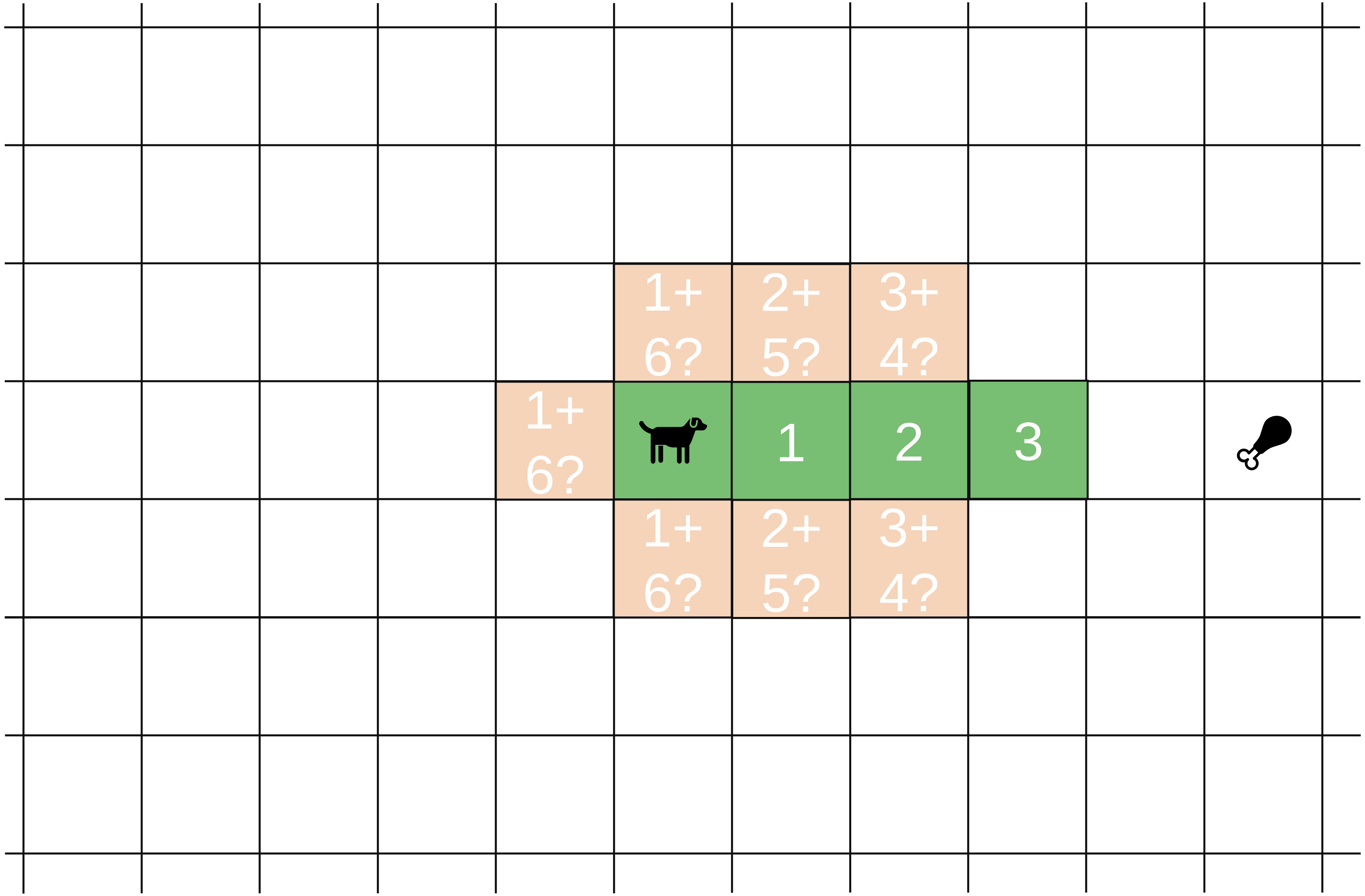
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



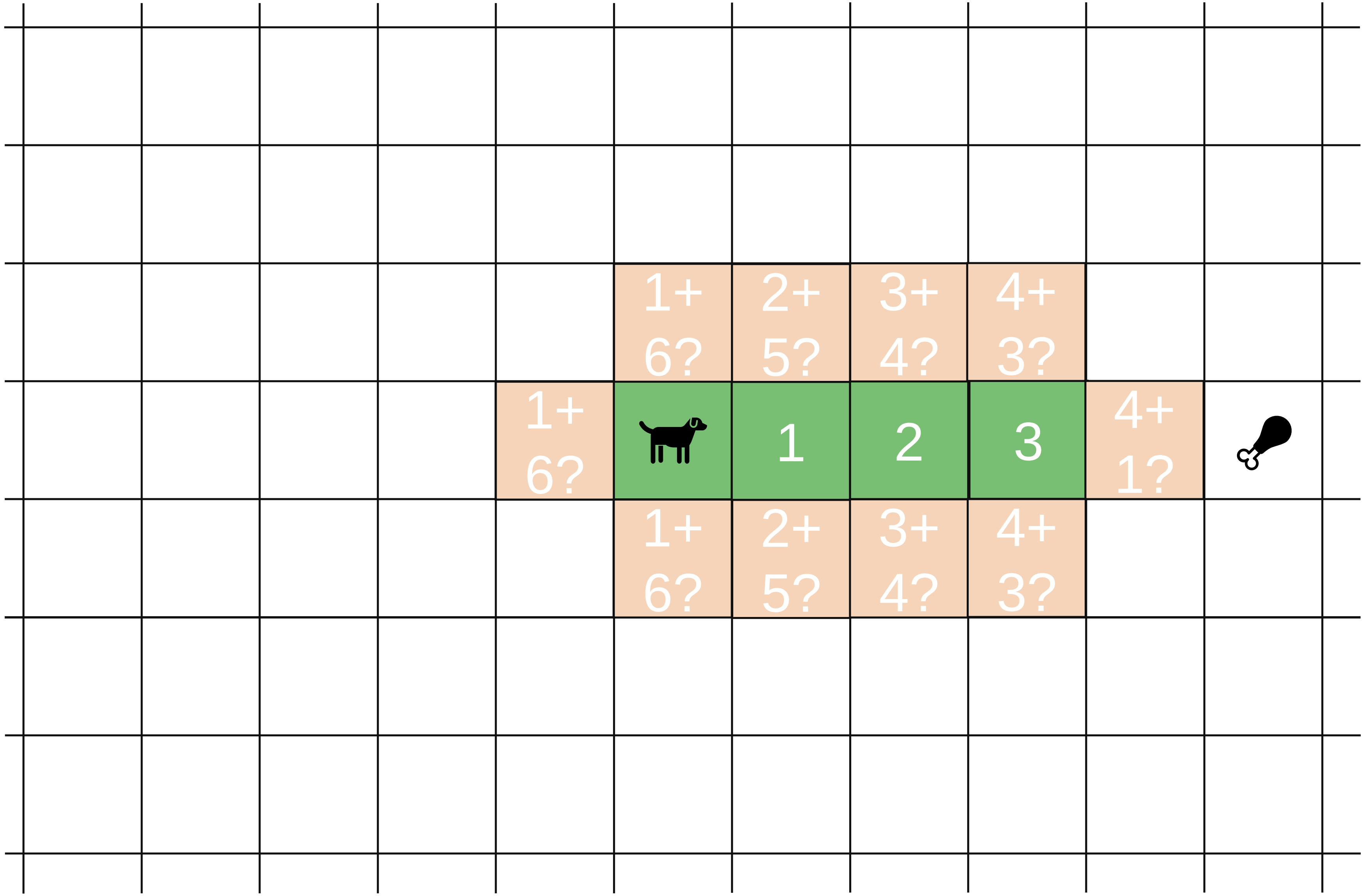
Greedily continue at smallest

$\boxed{\text{distance(s, u)}}$

+

$\boxed{\text{futureCost(u,t)}}$

A*: Future cost = row + column distance



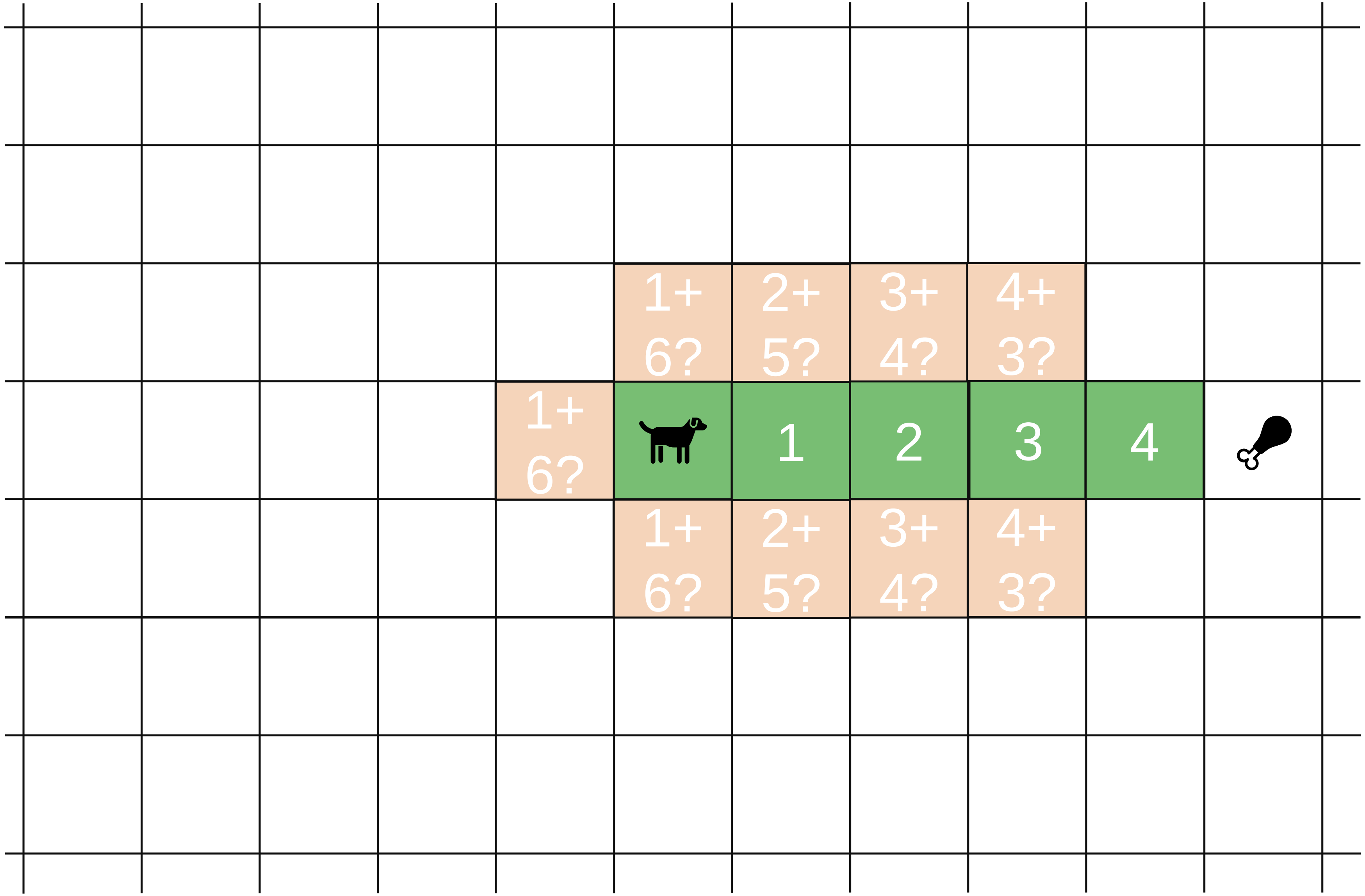
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



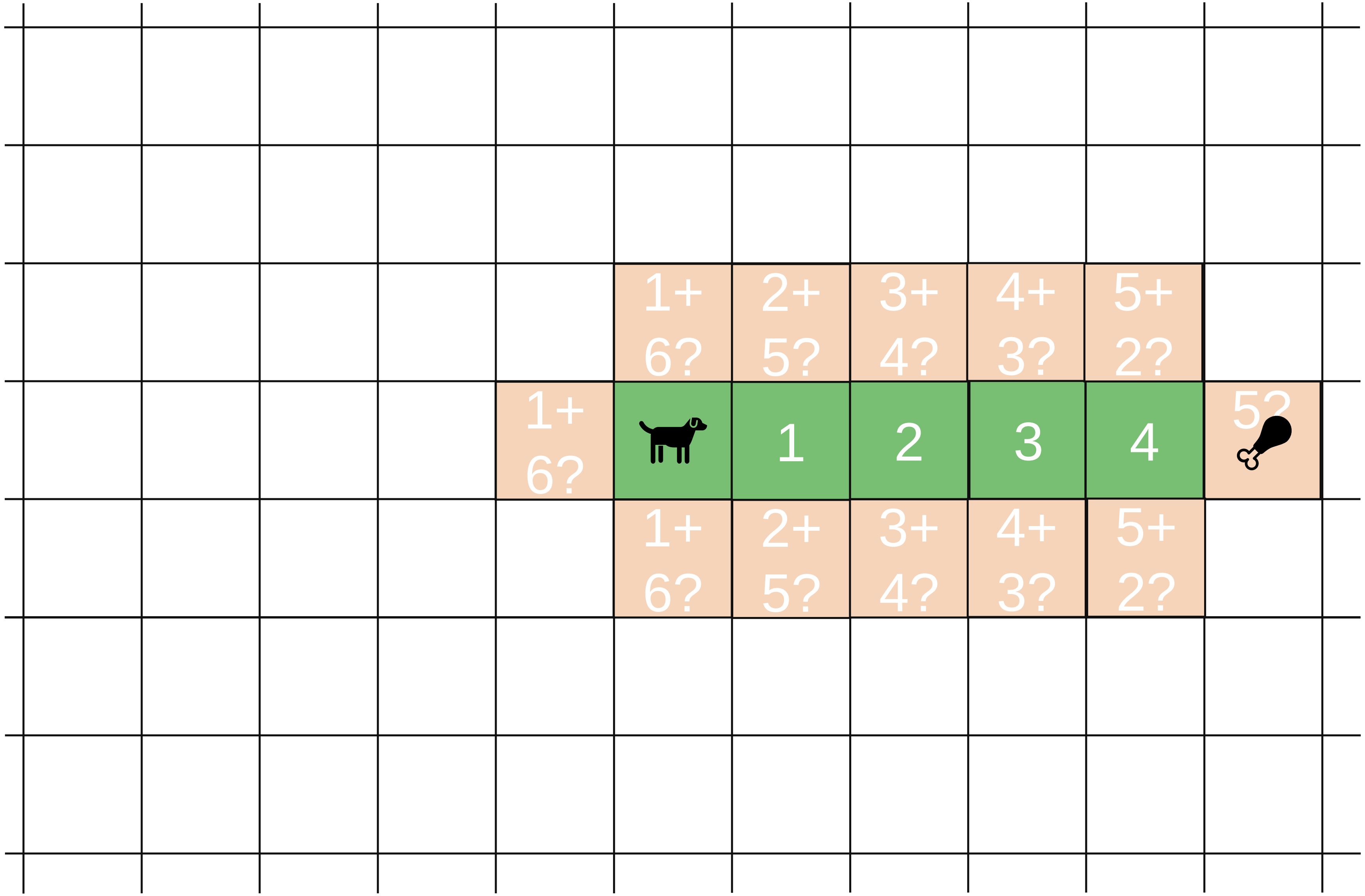
Greedily continue at smallest

distance(s, u)

+

futureCost(u,t)

A*: Future cost = row + column distance



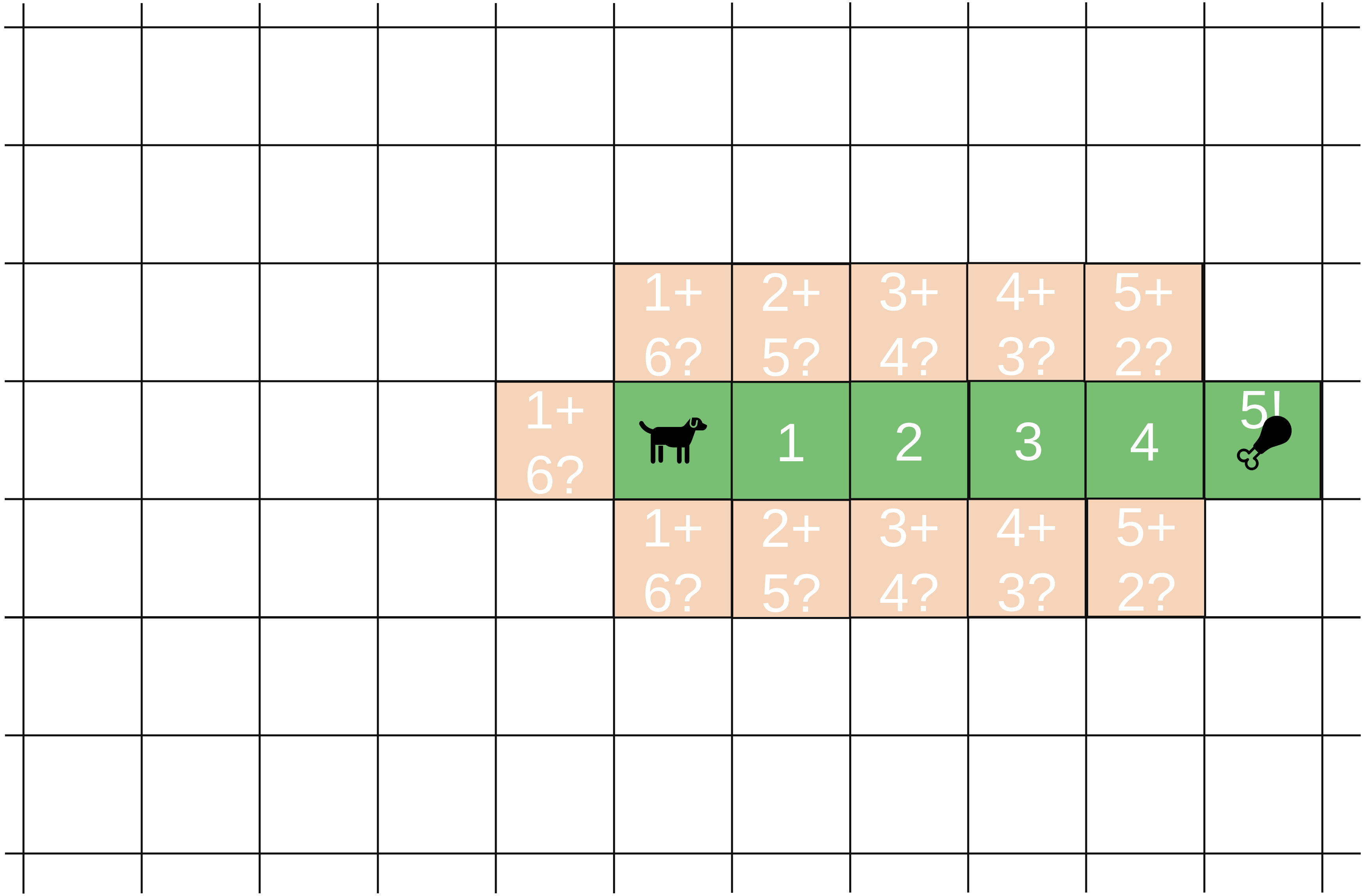
Greedily continue at smallest

$\boxed{\text{distance(s, u)}}$

+

$\boxed{\text{futureCost(u,t)}}$

A*: Future cost = row + column distance



Greedily continue at smallest

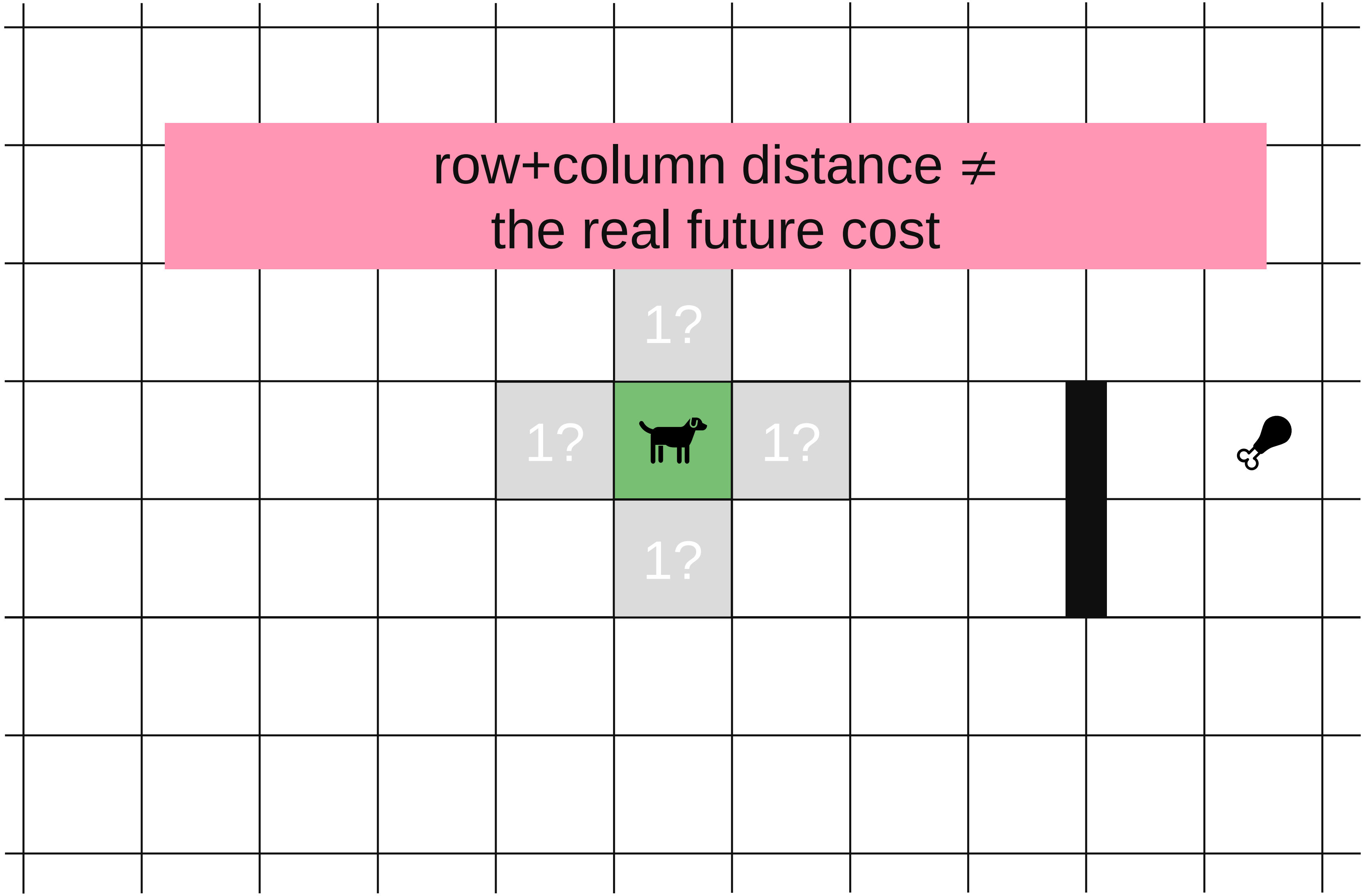
distance(s, u)

+

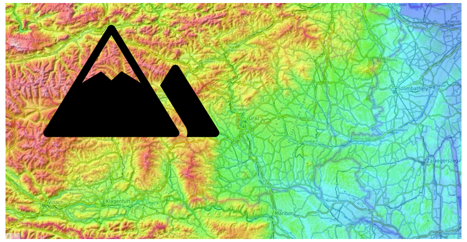
futureCost(u,t)

A bit easy

A*: Use row + column distance



row+column distance ≠
the real future cost



Greedily continue at smallest

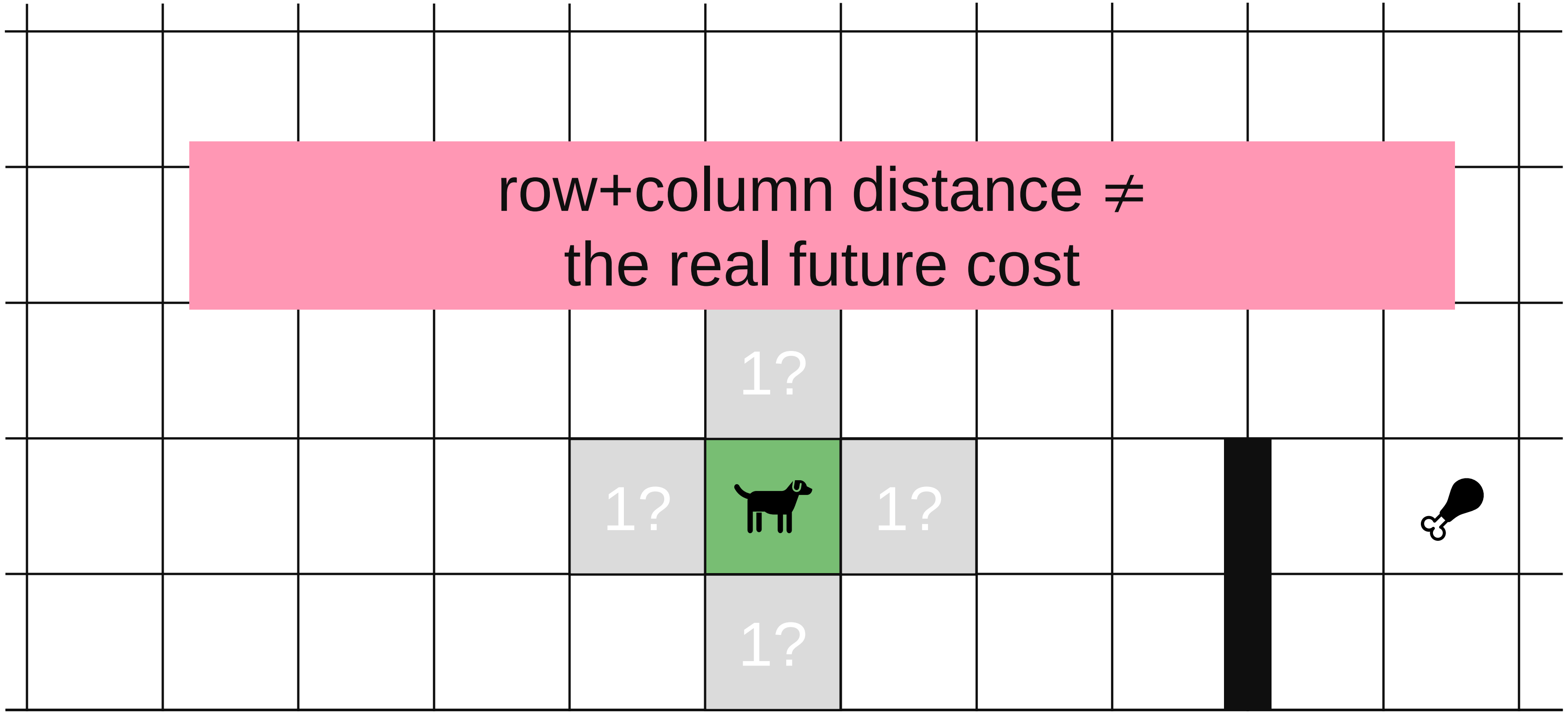
distance(s, u)

+

~~future~~ ~~(u,t)~~

wanna be futureCost(u,t)

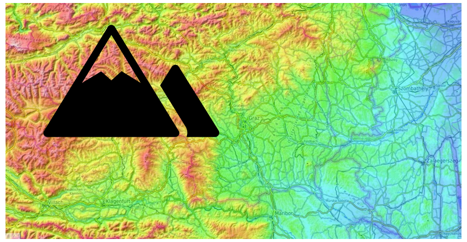
A*: Use row + column distance



row+column distance ≠
the real future cost

This is the A* Algorithm!

(Dijkstra with different path priorities)



Greedily continue at smallest

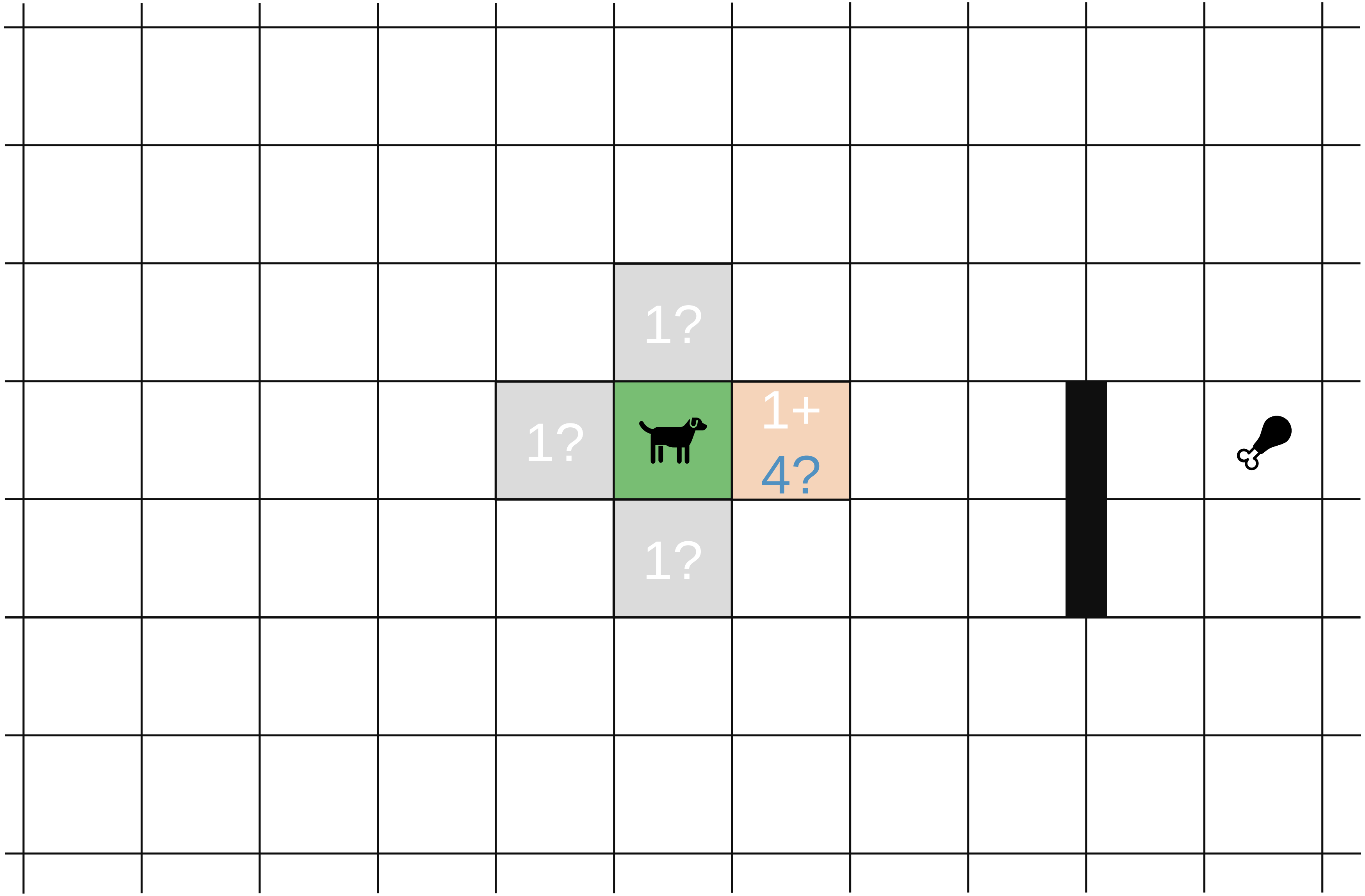
distance(s, u)

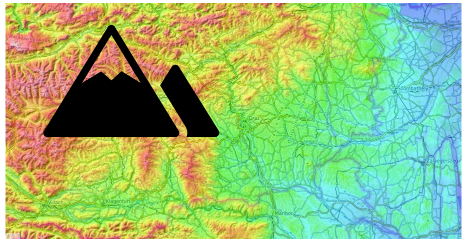
+

~~futureCost(u,t)~~

wanna be futureCost(u,t)

A*: Use row + column distance





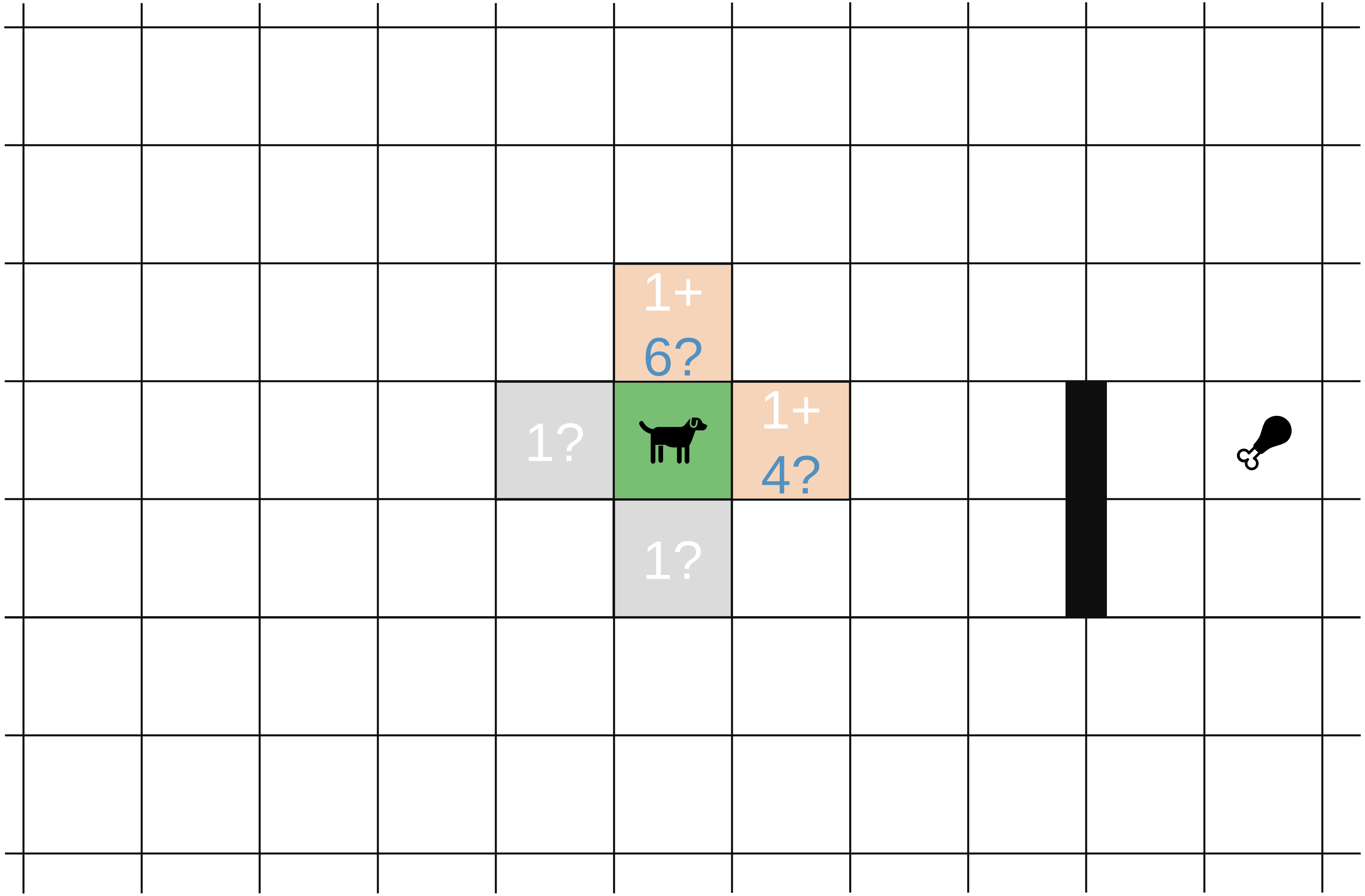
Greedily continue at smallest

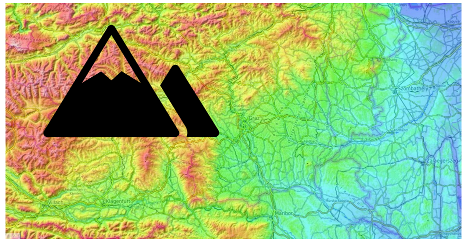
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





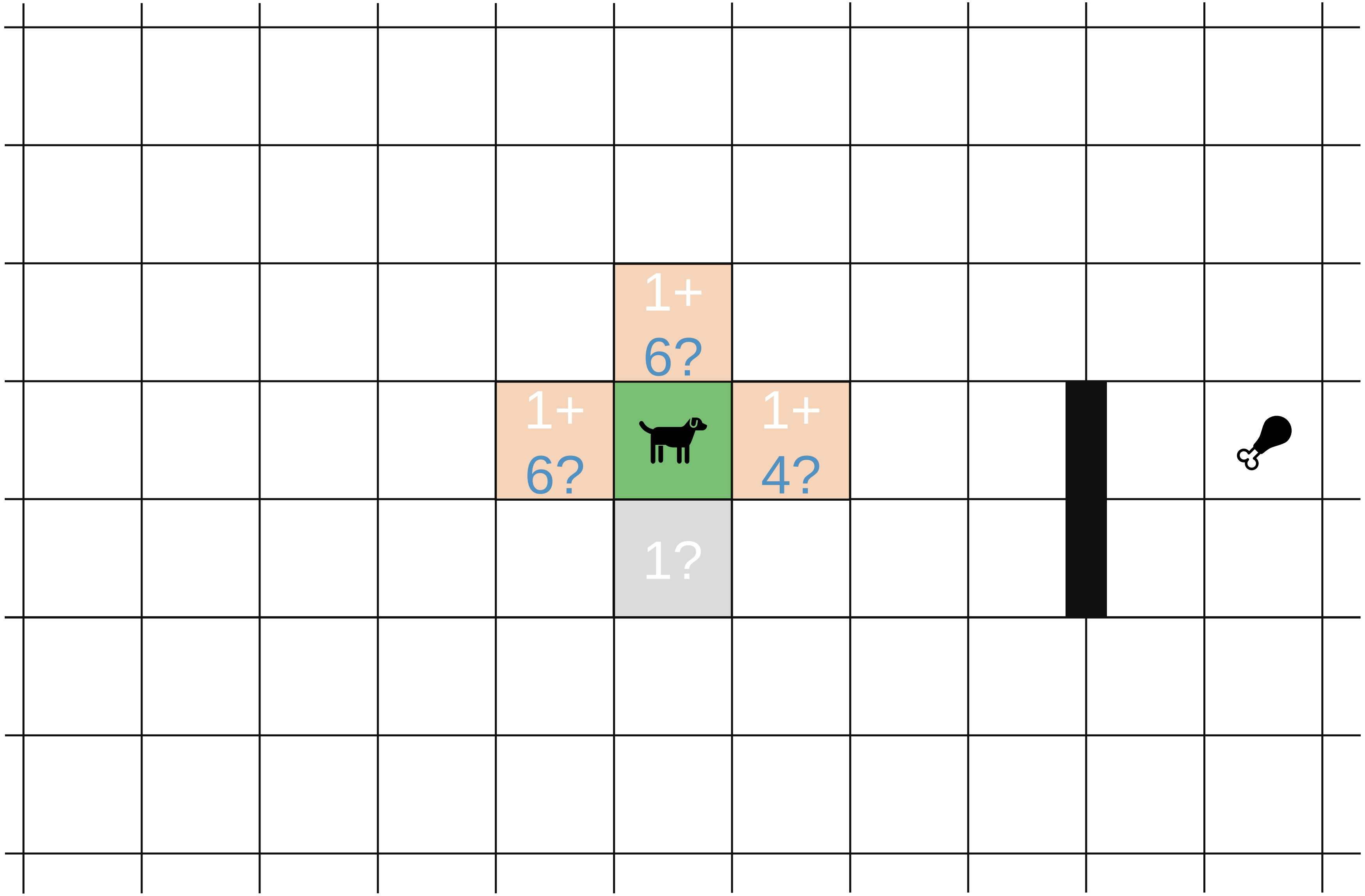
Greedily continue at smallest

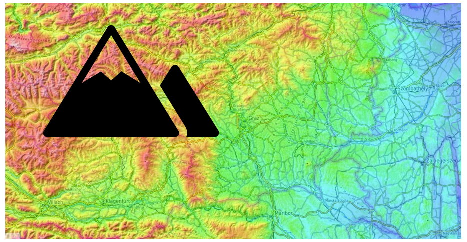
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





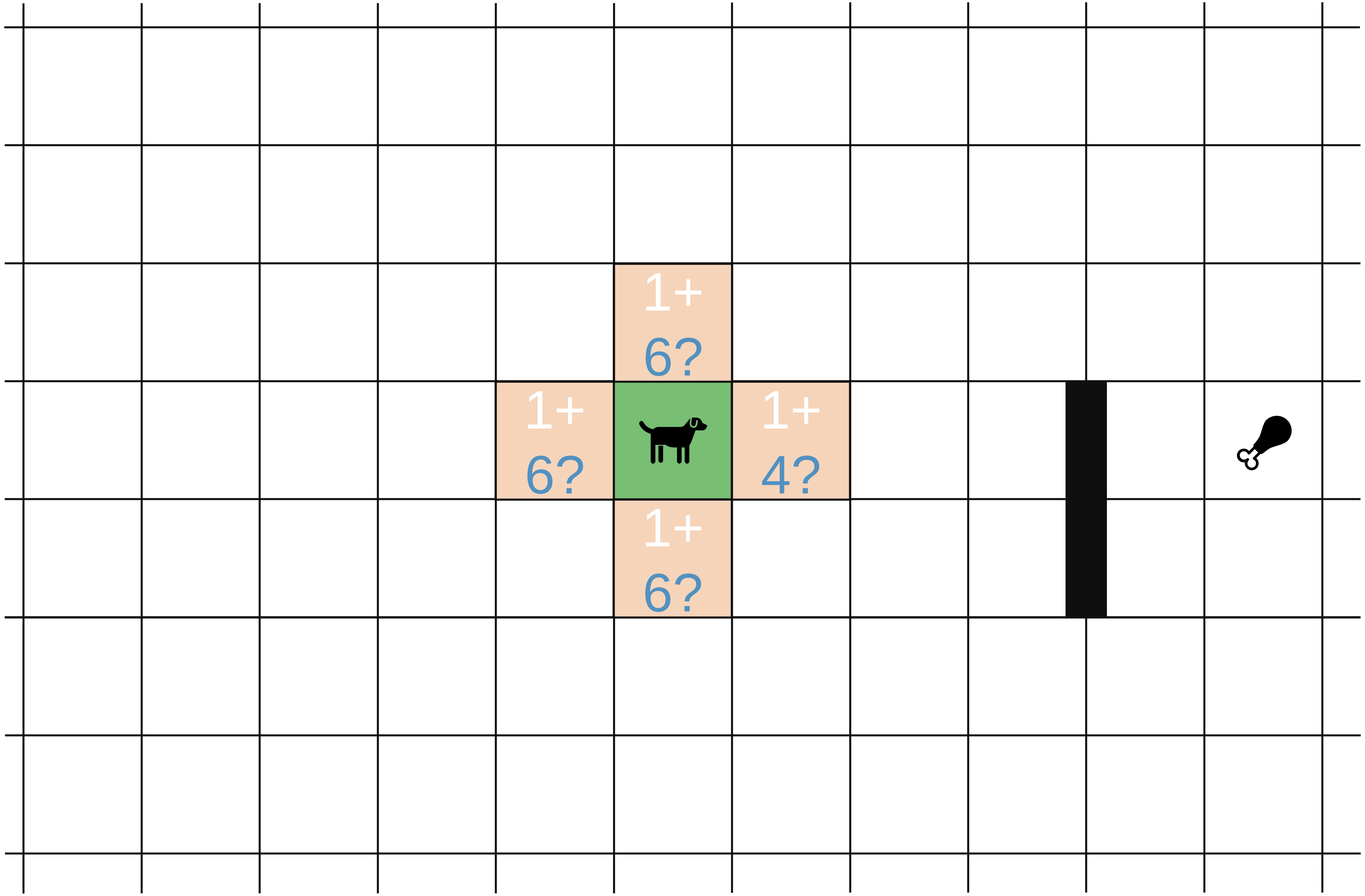
Greedily continue at smallest

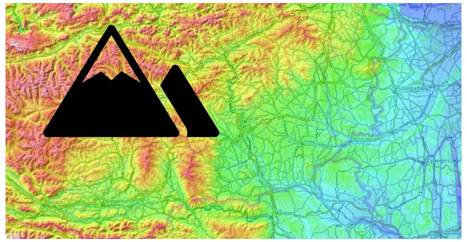
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





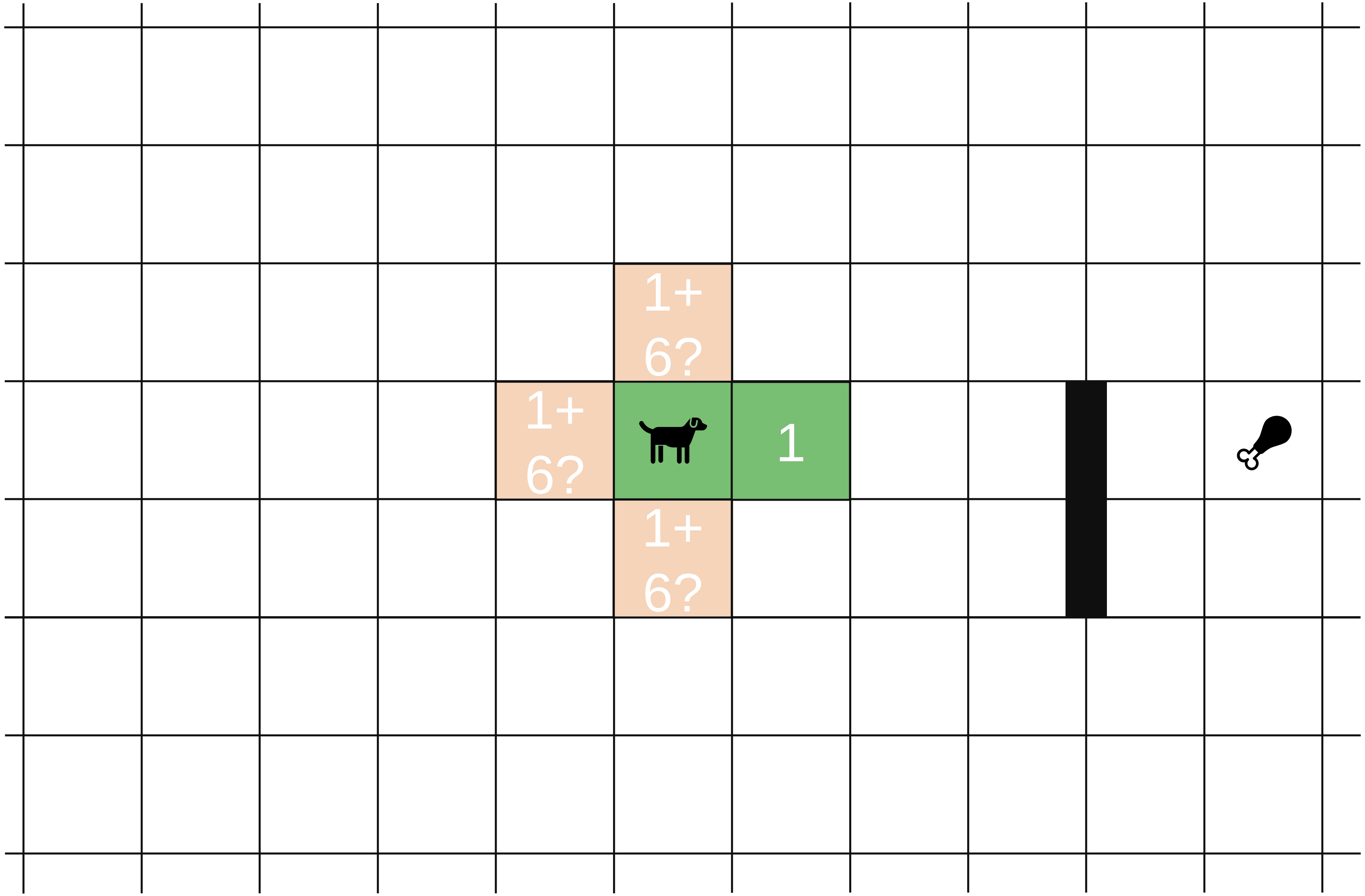
Greedily continue at smallest

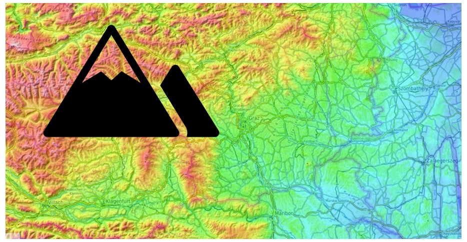
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





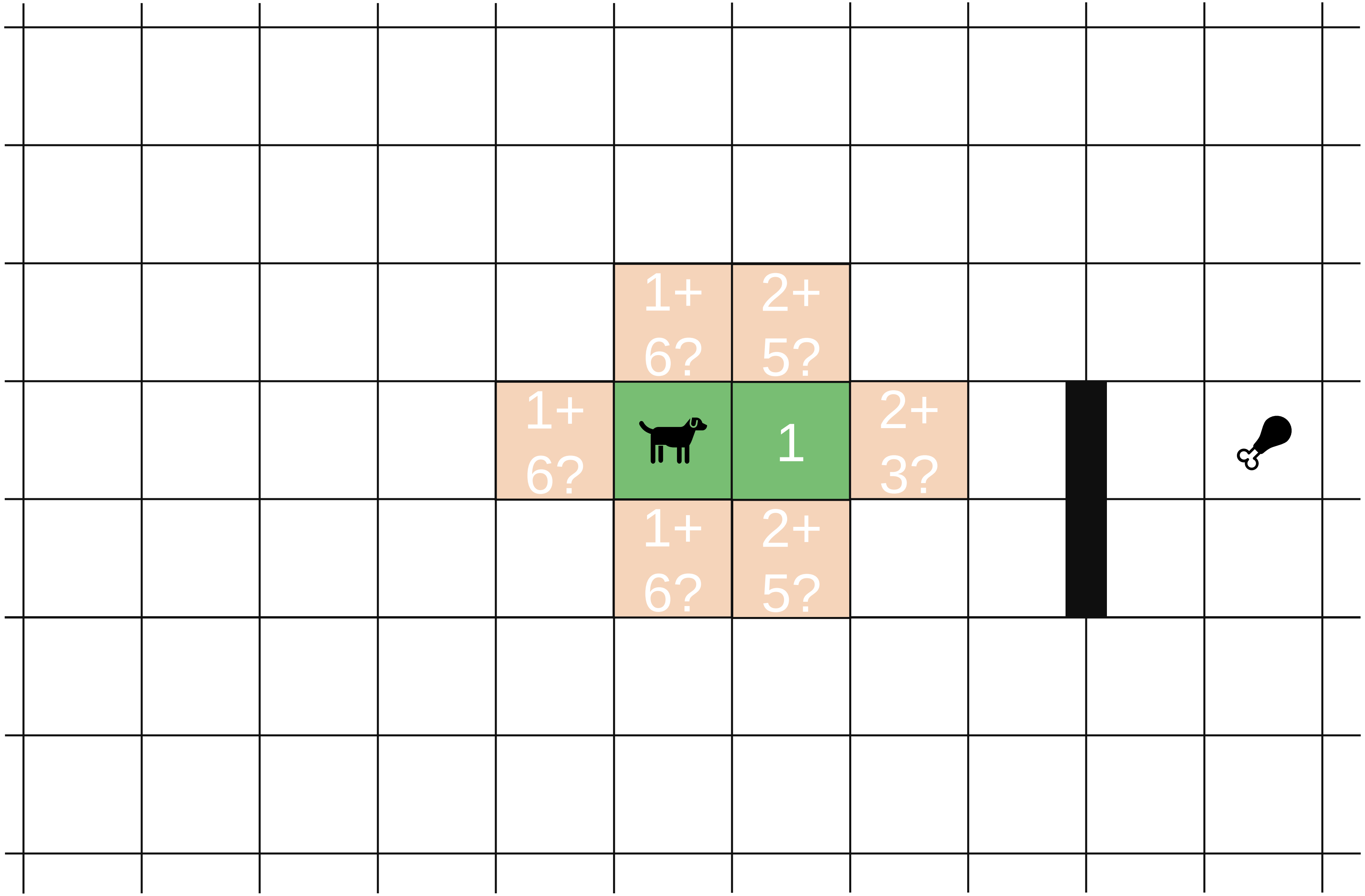
Greedily continue at smallest

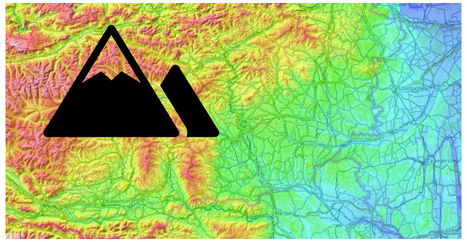
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





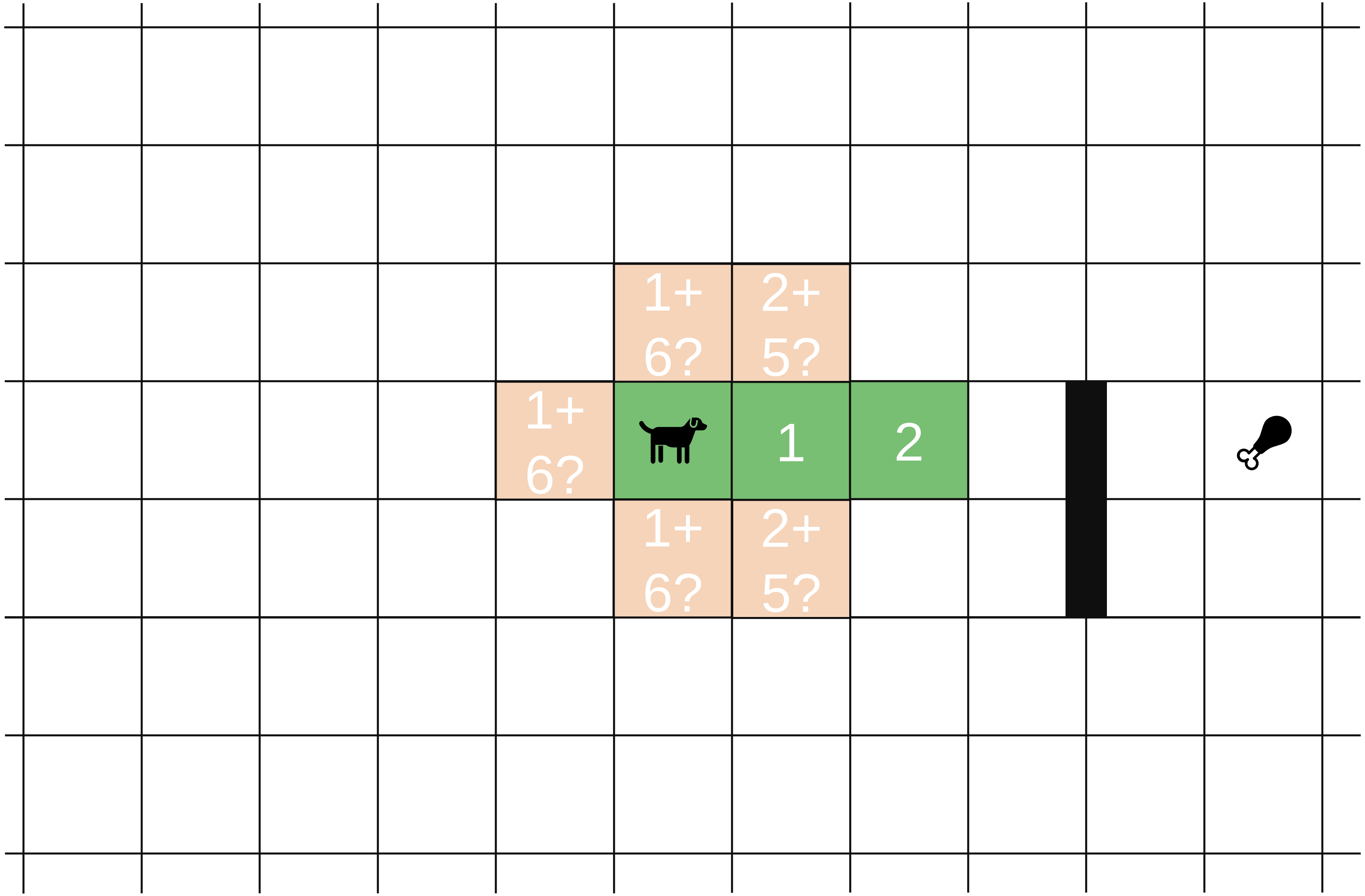
Greedily continue at smallest

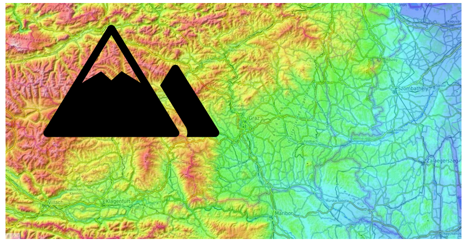
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





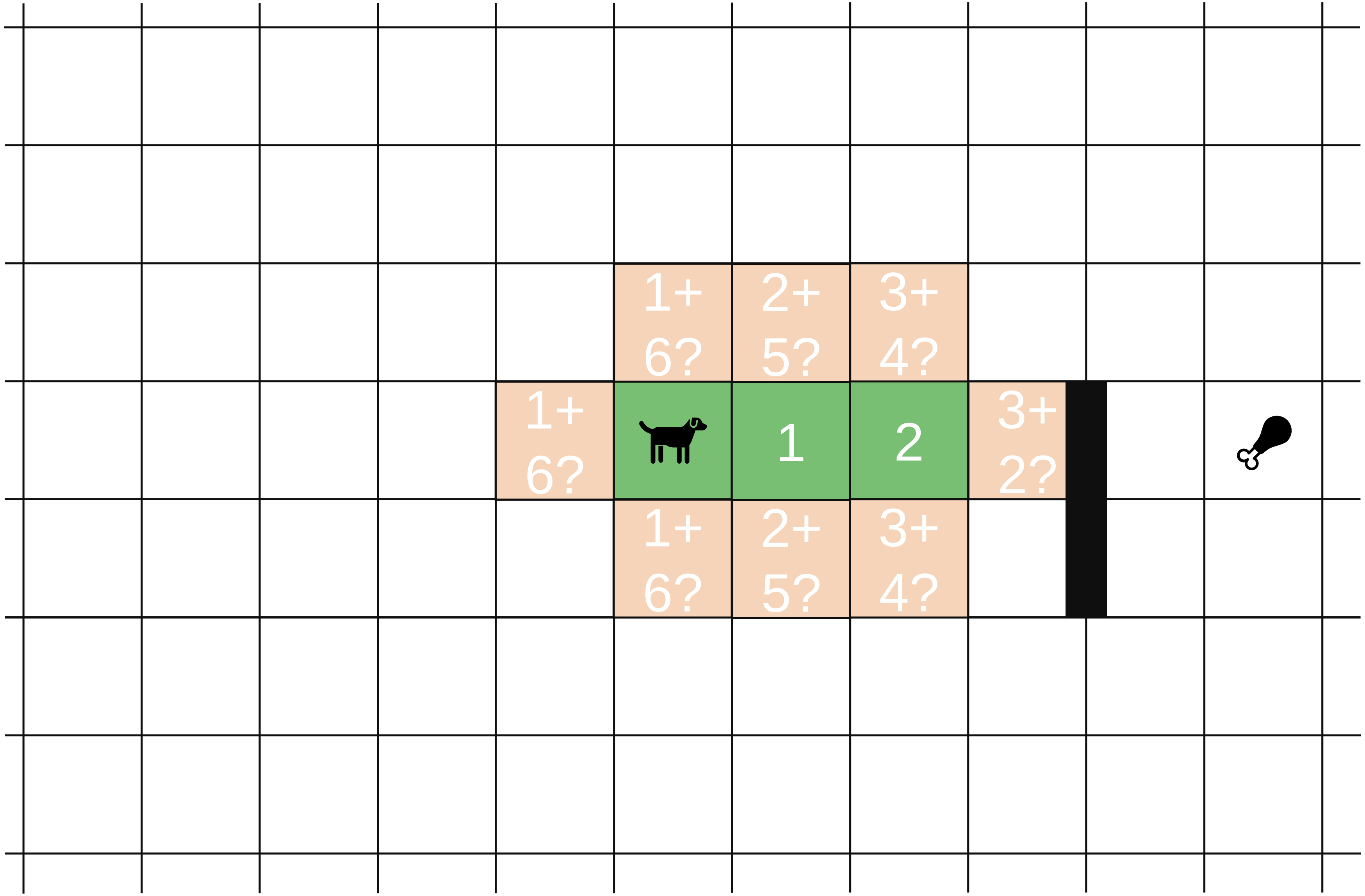
Greedily continue at smallest

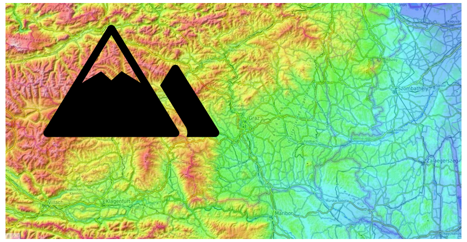
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





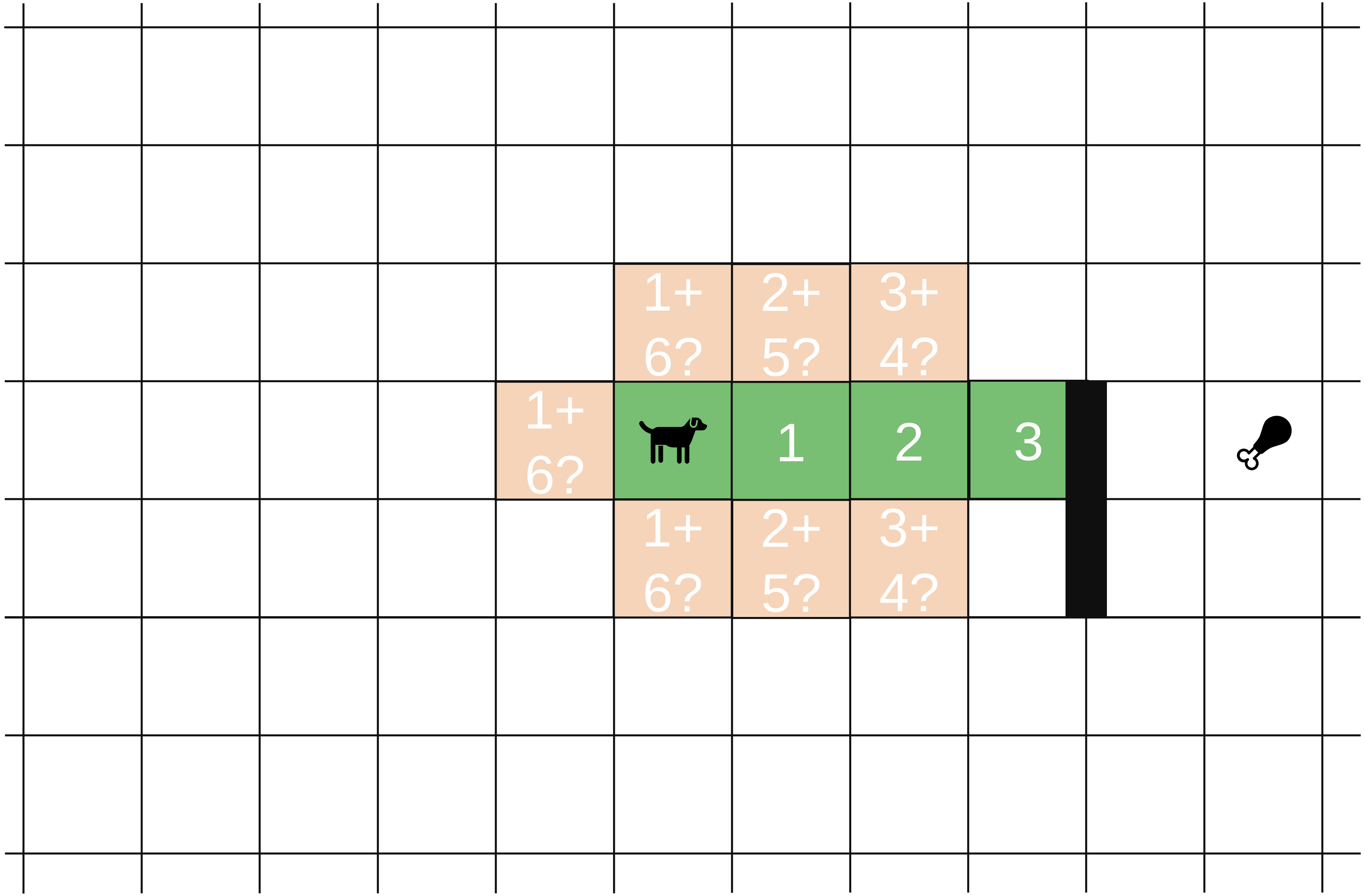
Greedily continue at smallest

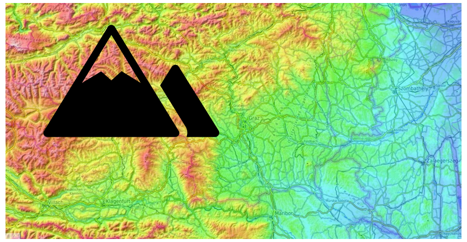
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





Greedily continue at smallest

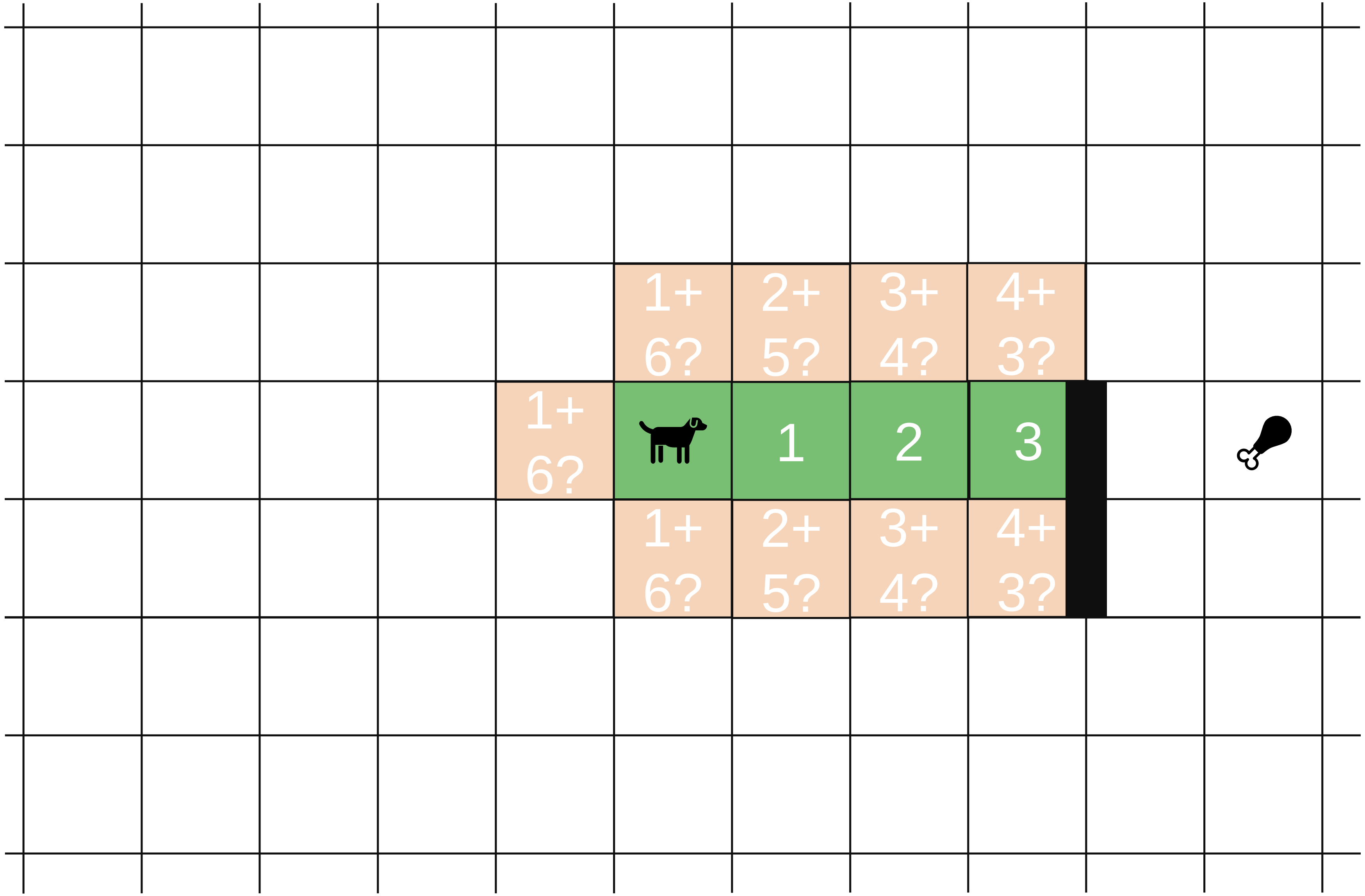
distance(s, u)

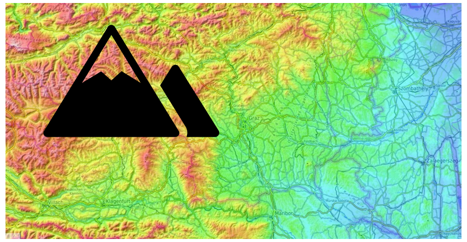
+

wanna be futureCost(u,t)

Yannic Maus

A*: Use row + column distance





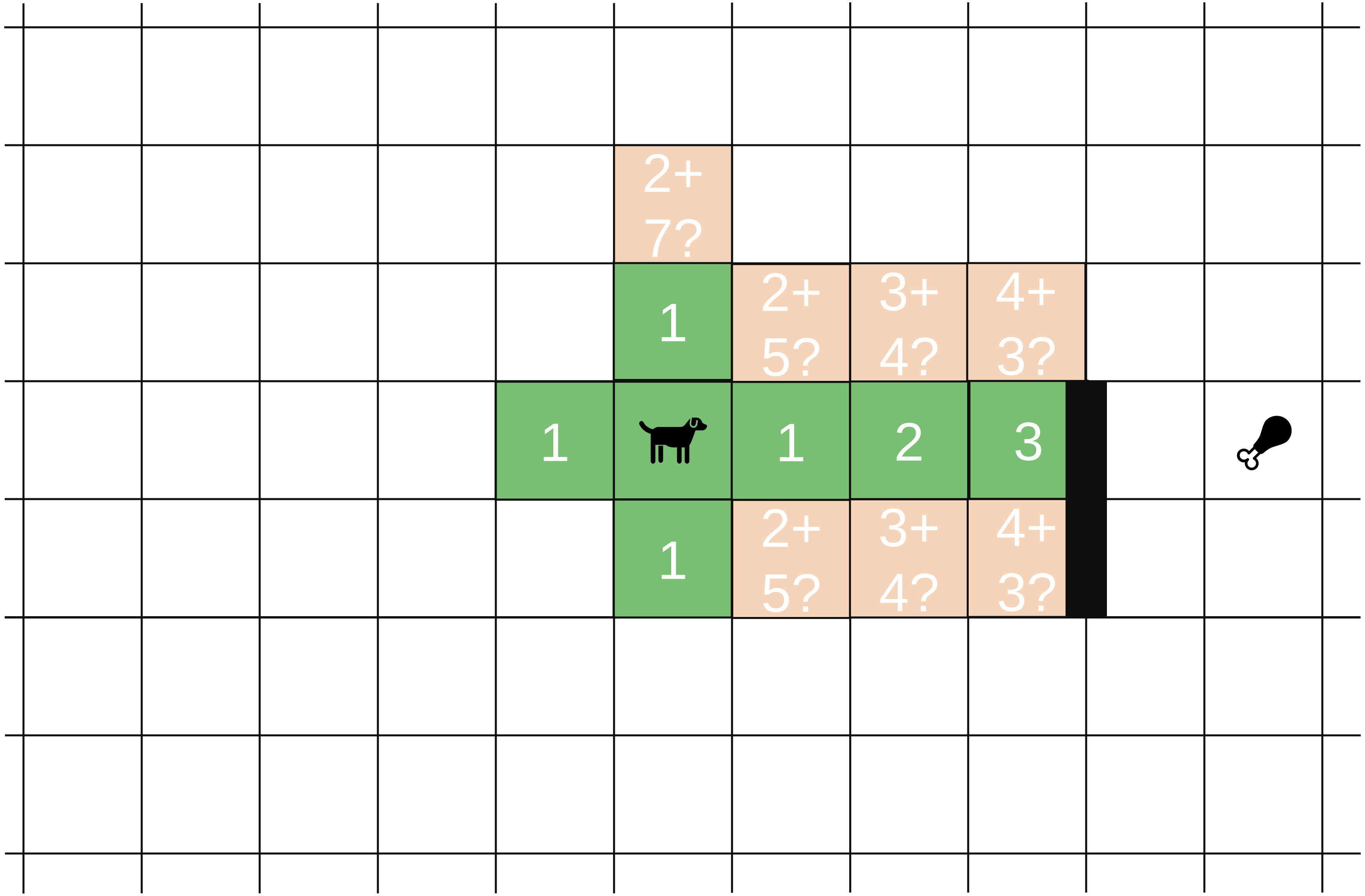
Greedily continue at smallest

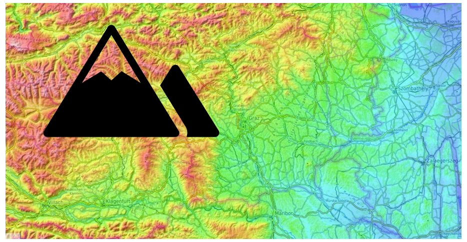
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





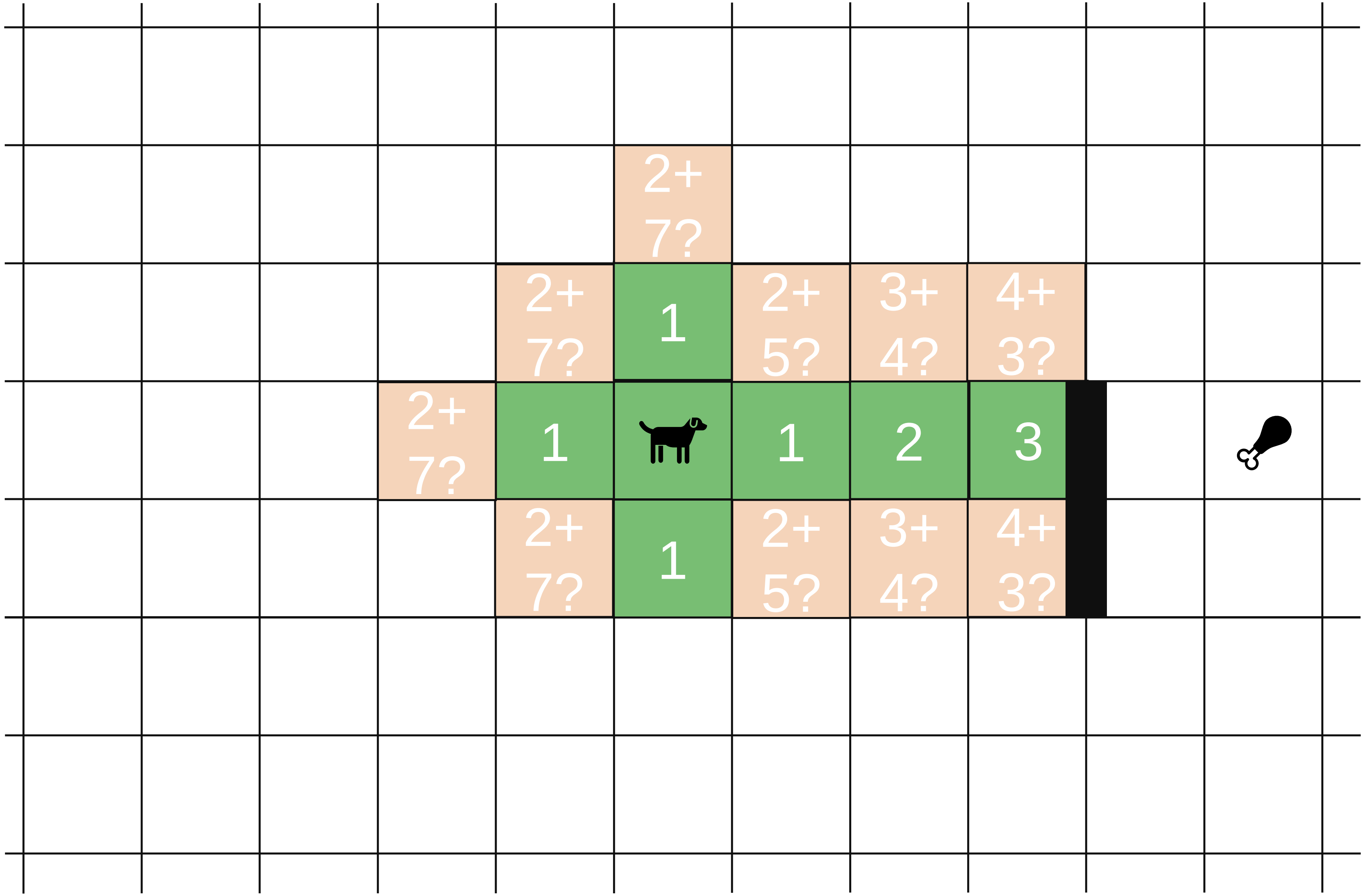
Greedily continue at smallest

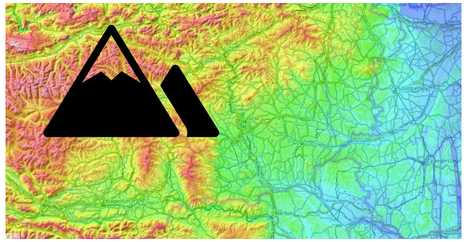
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





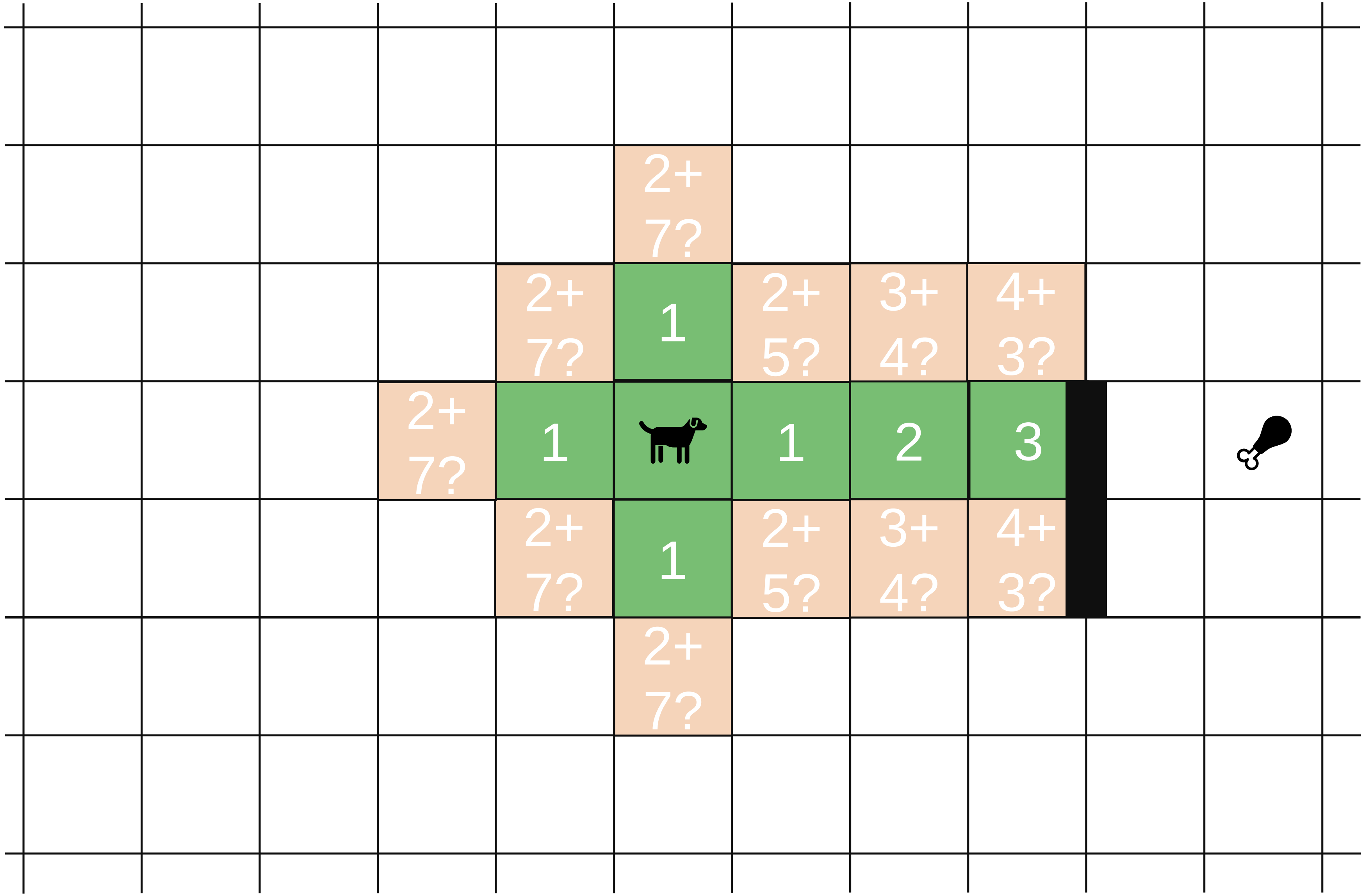
Greedily continue at smallest

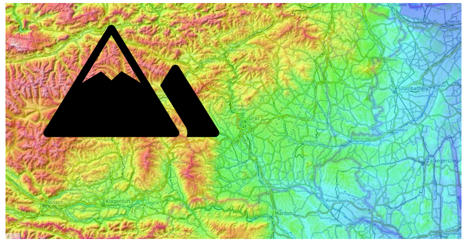
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





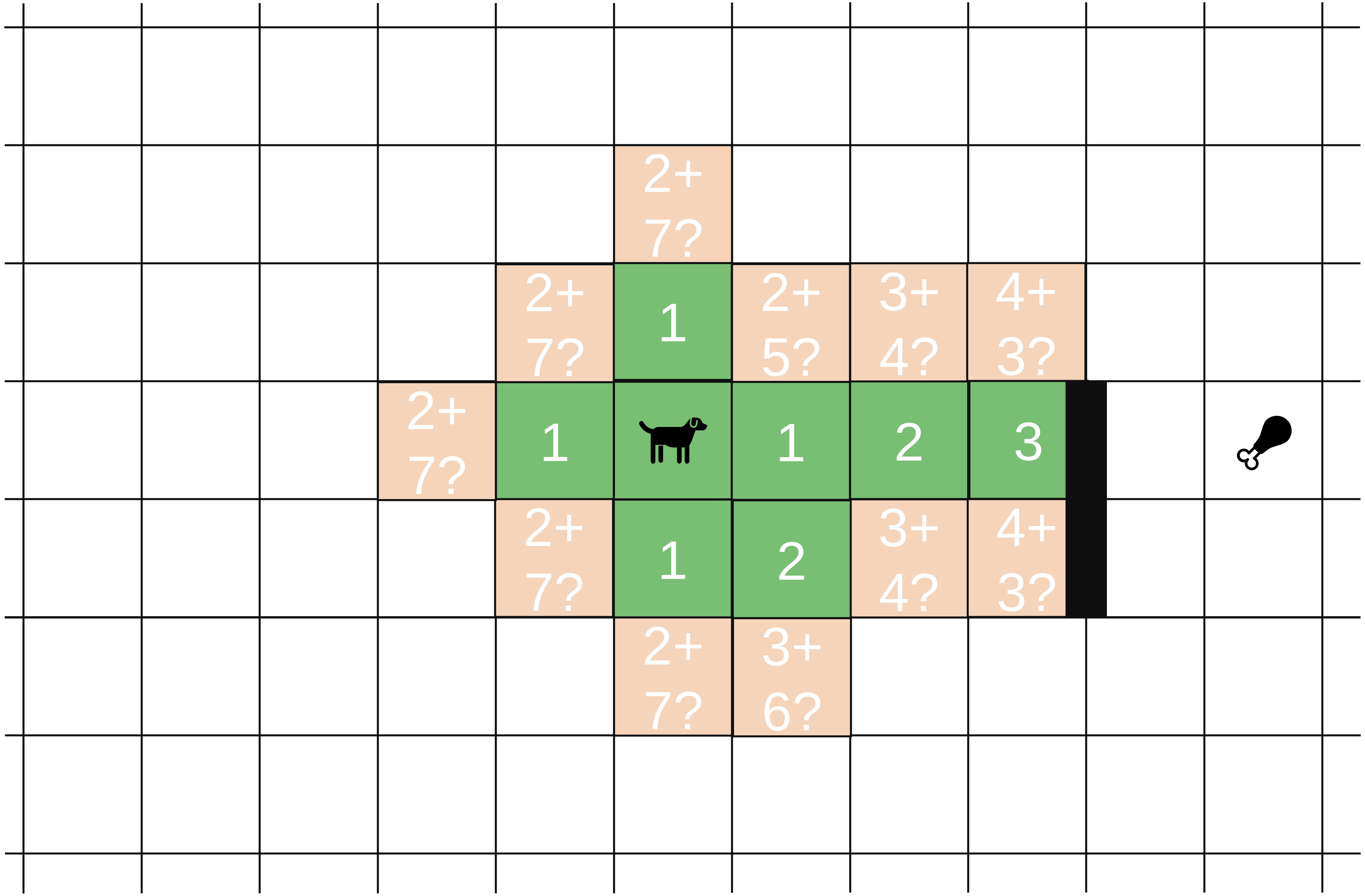
Greedily continue at smallest

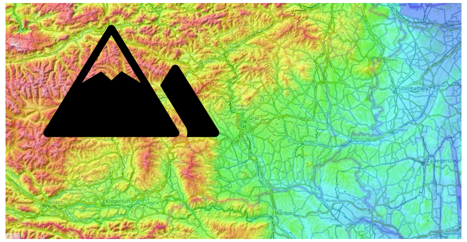
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





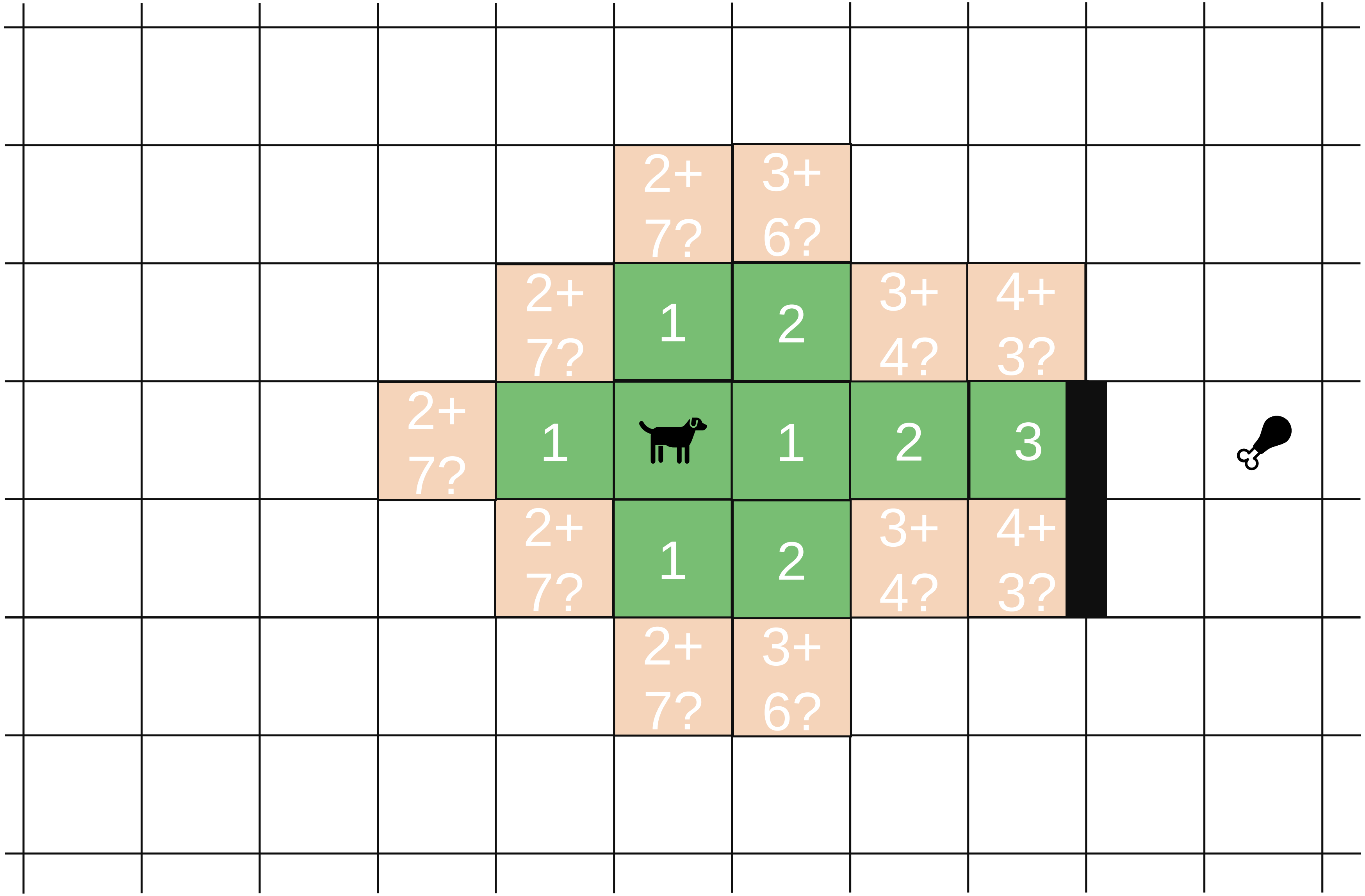
Greedily continue at smallest

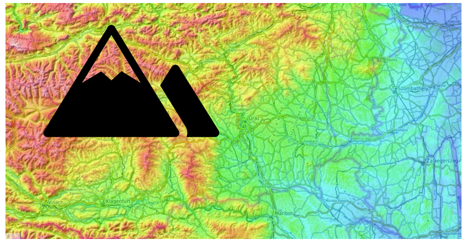
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





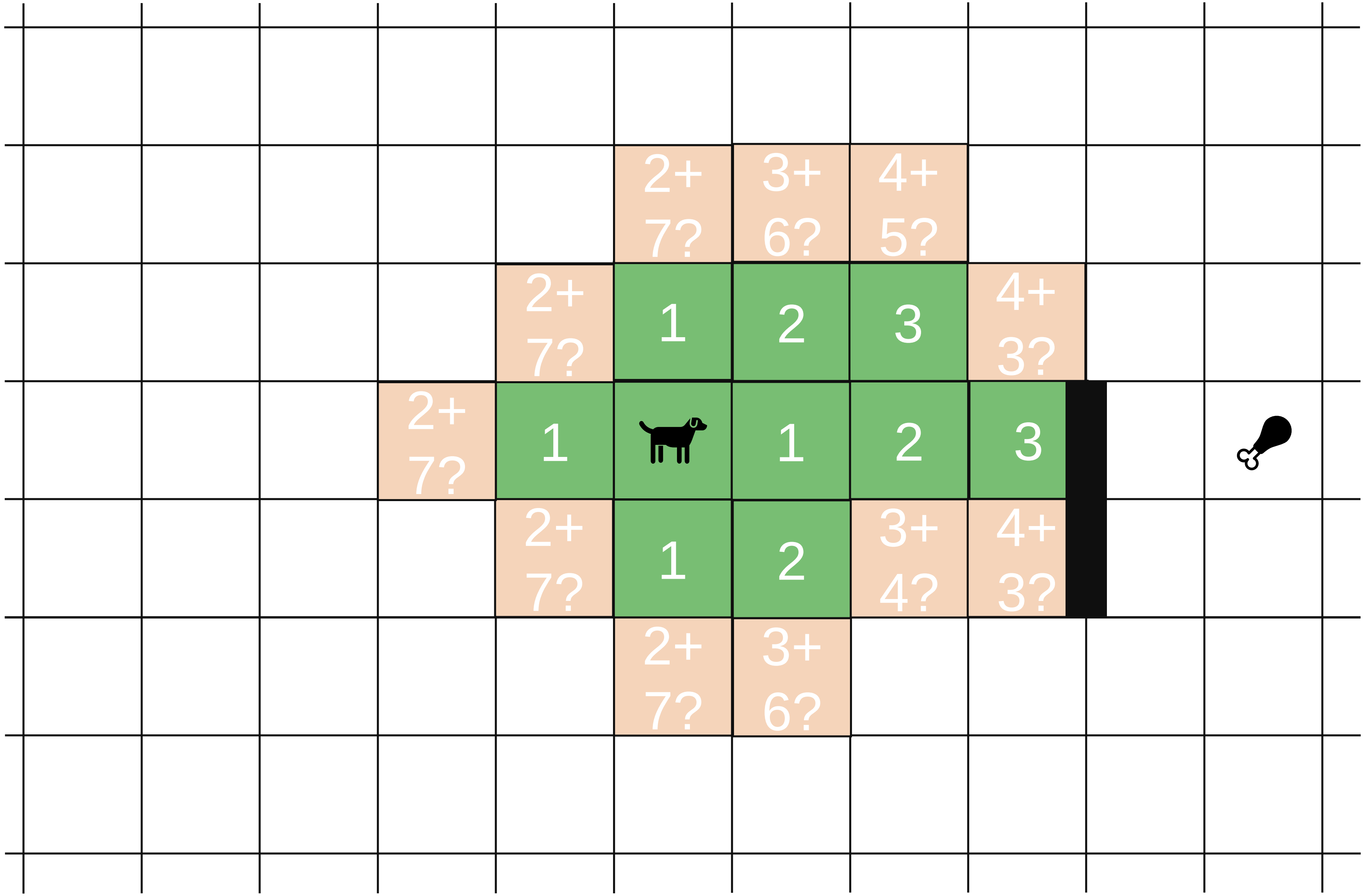
Greedily continue at smallest

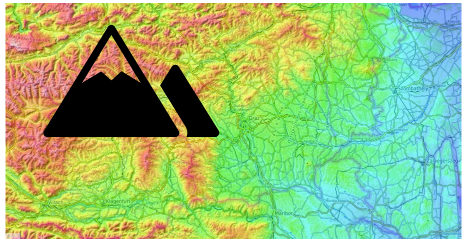
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





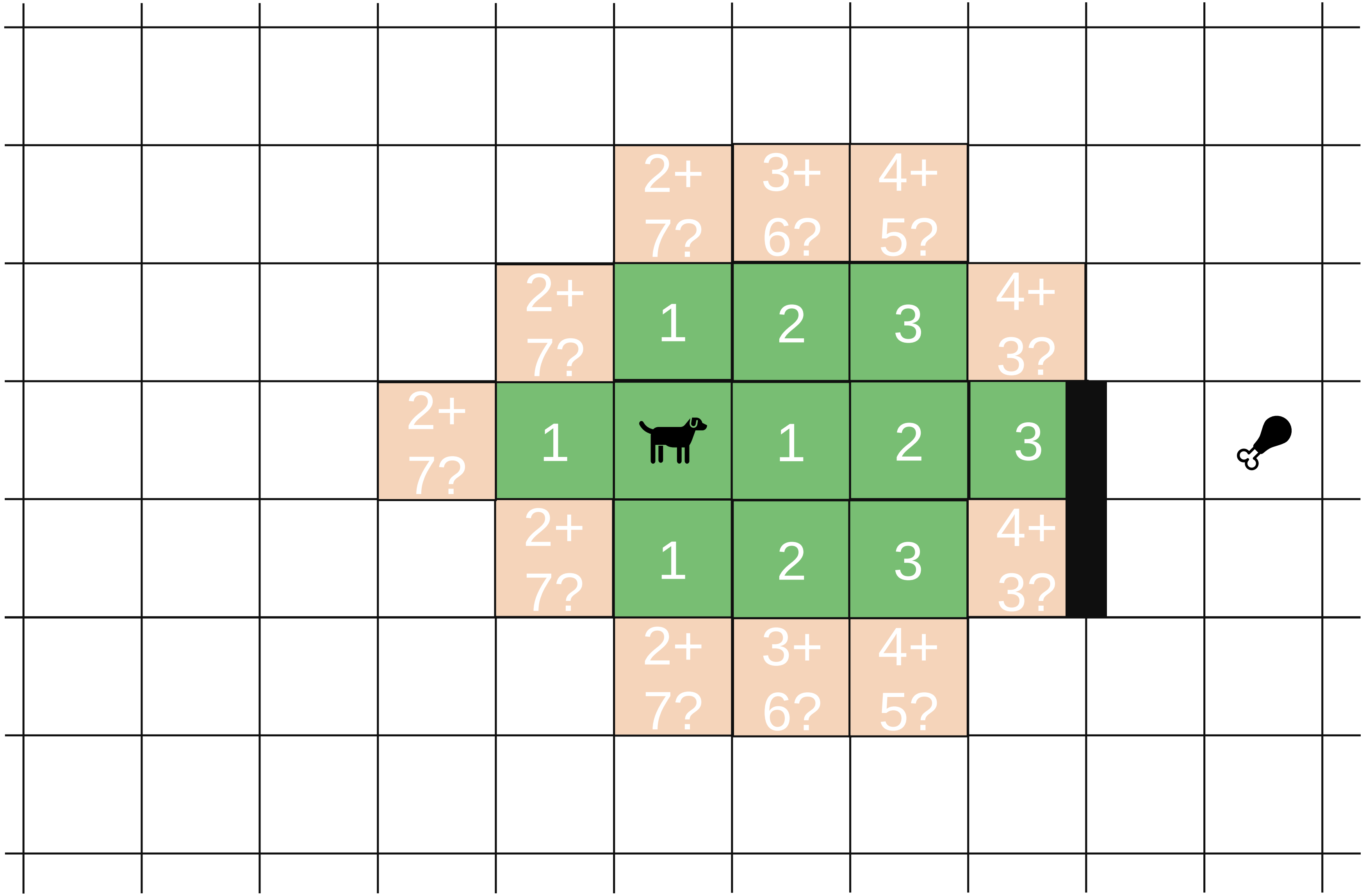
Greedily continue at smallest

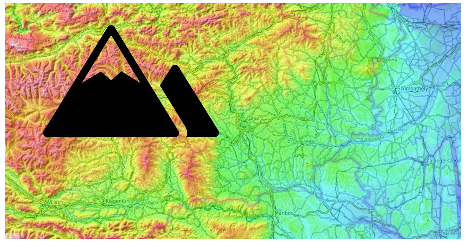
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





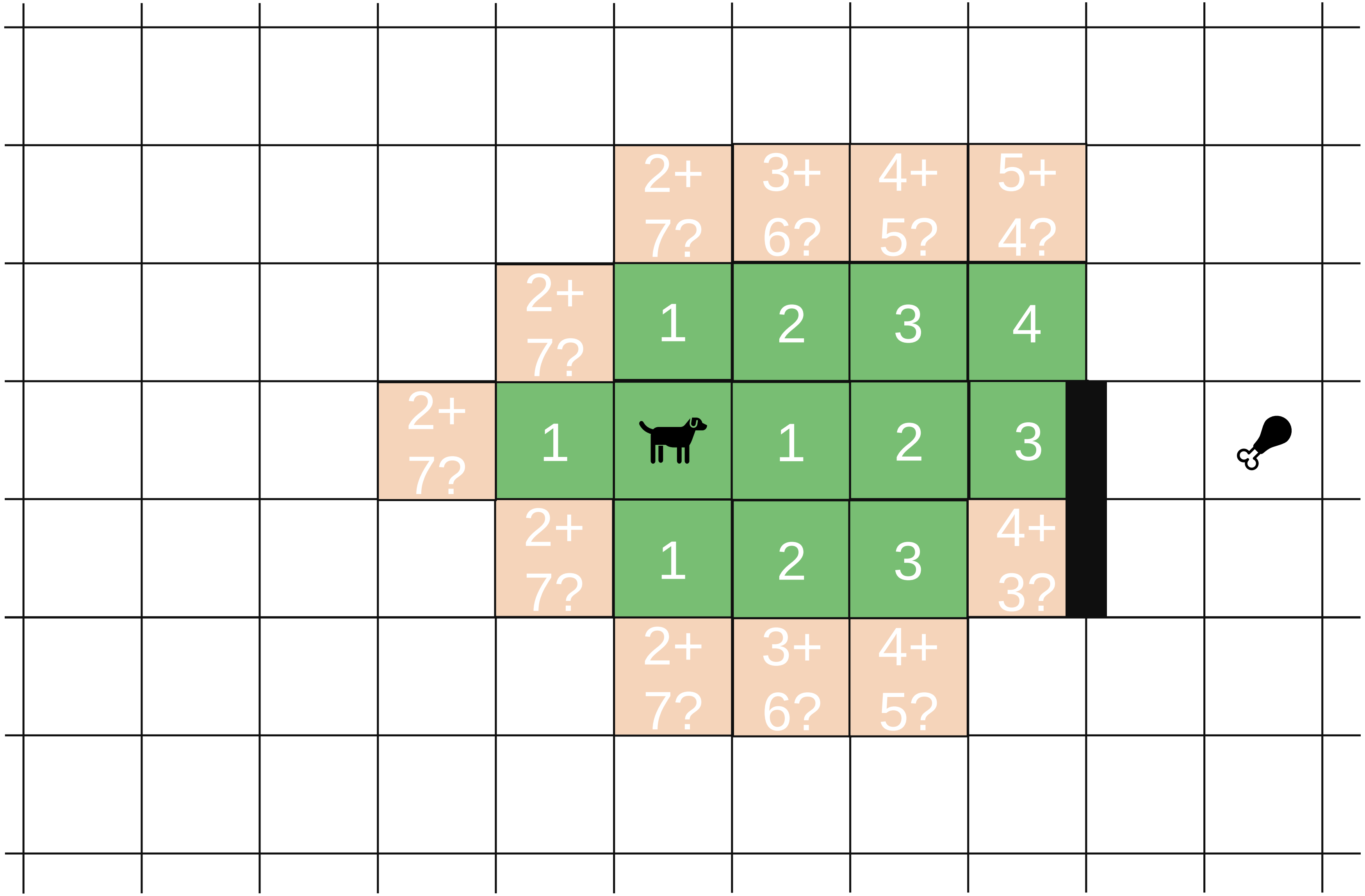
Greedily continue at smallest

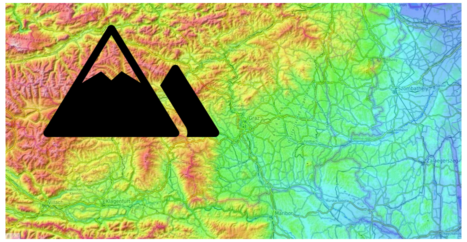
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





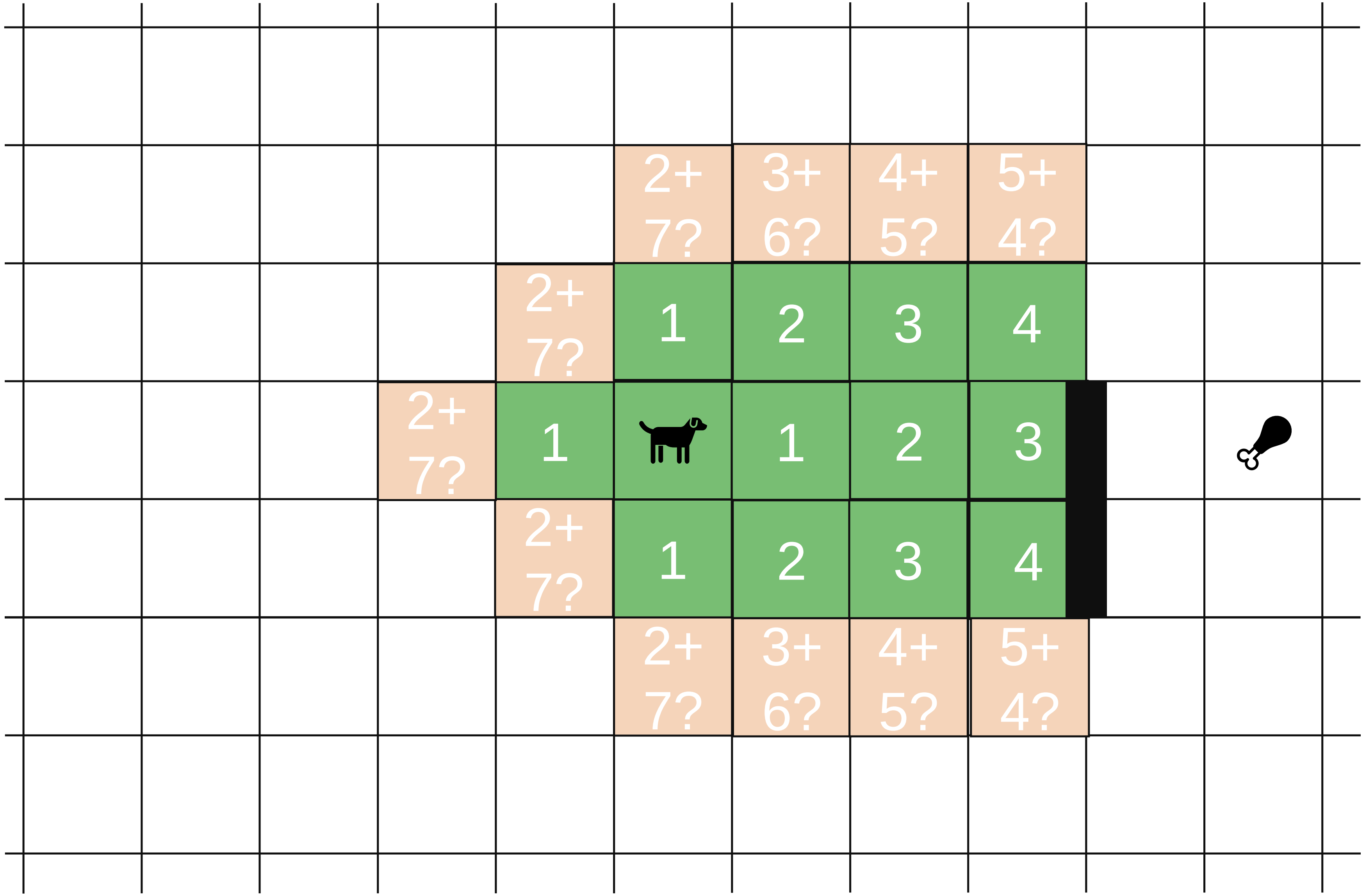
Greedily continue at smallest

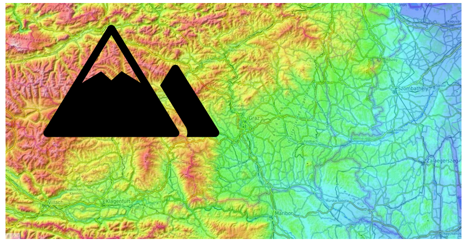
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





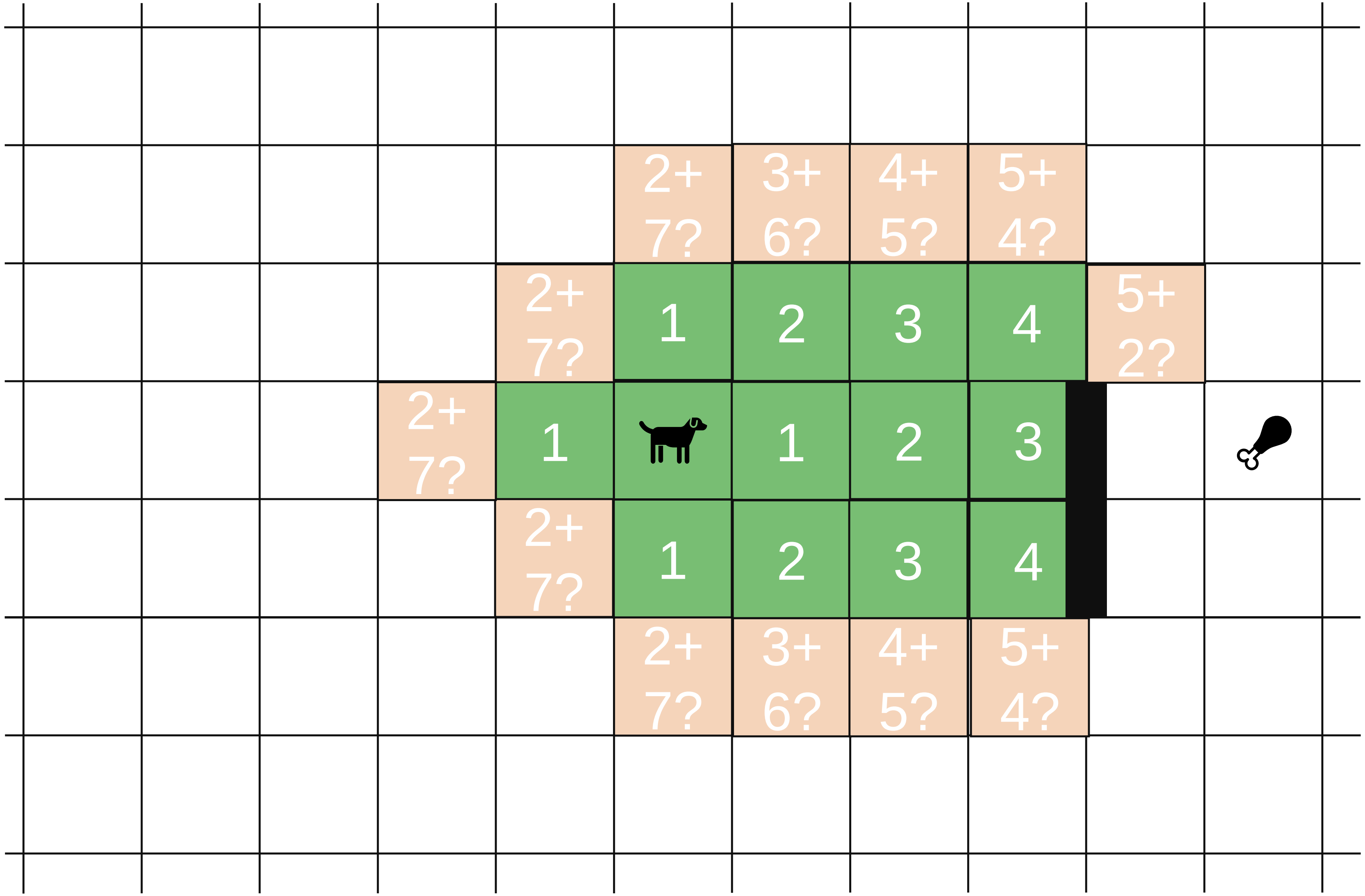
Greedily continue at smallest

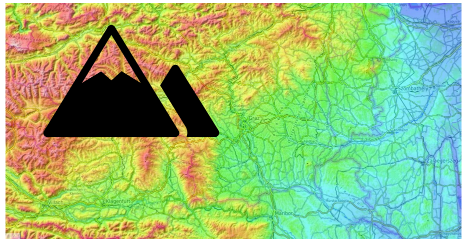
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





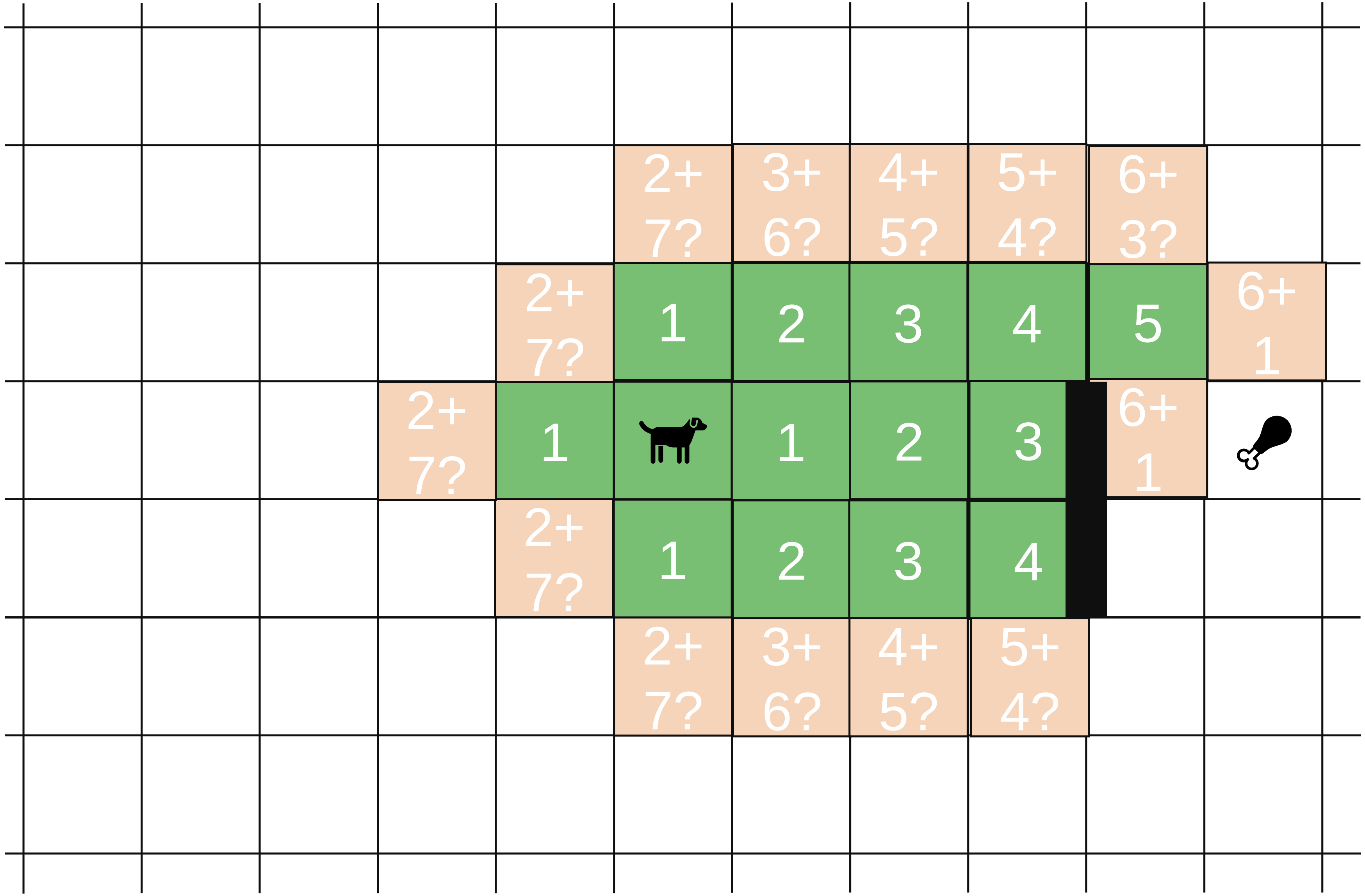
Greedily continue at smallest

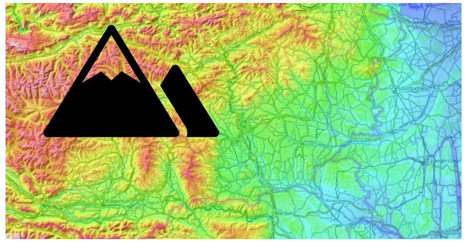
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





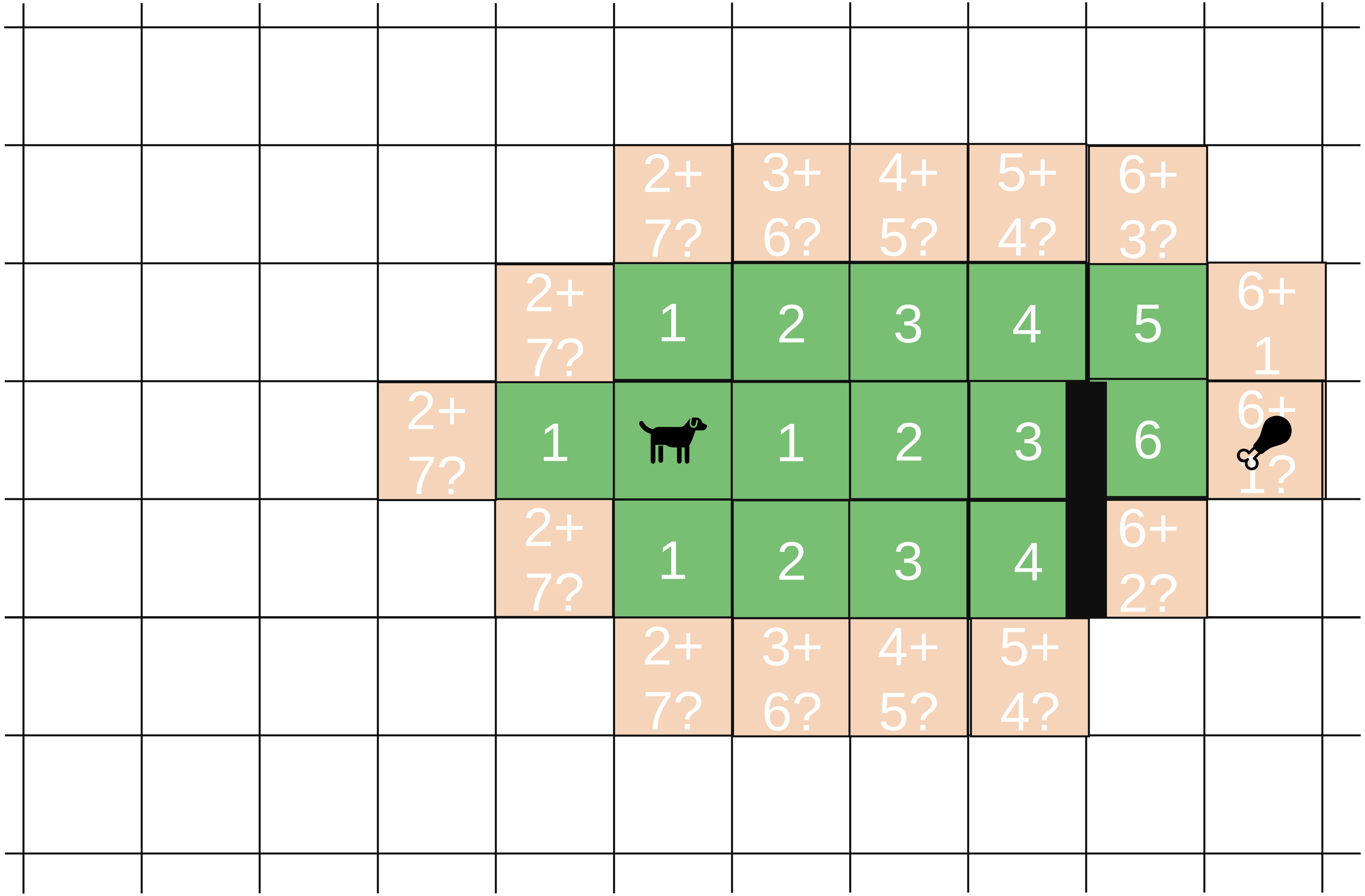
Greedily continue at smallest

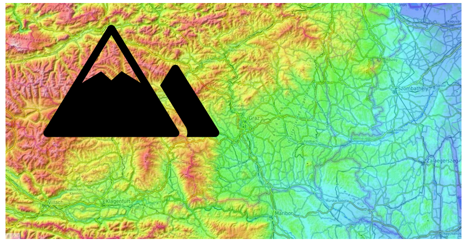
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





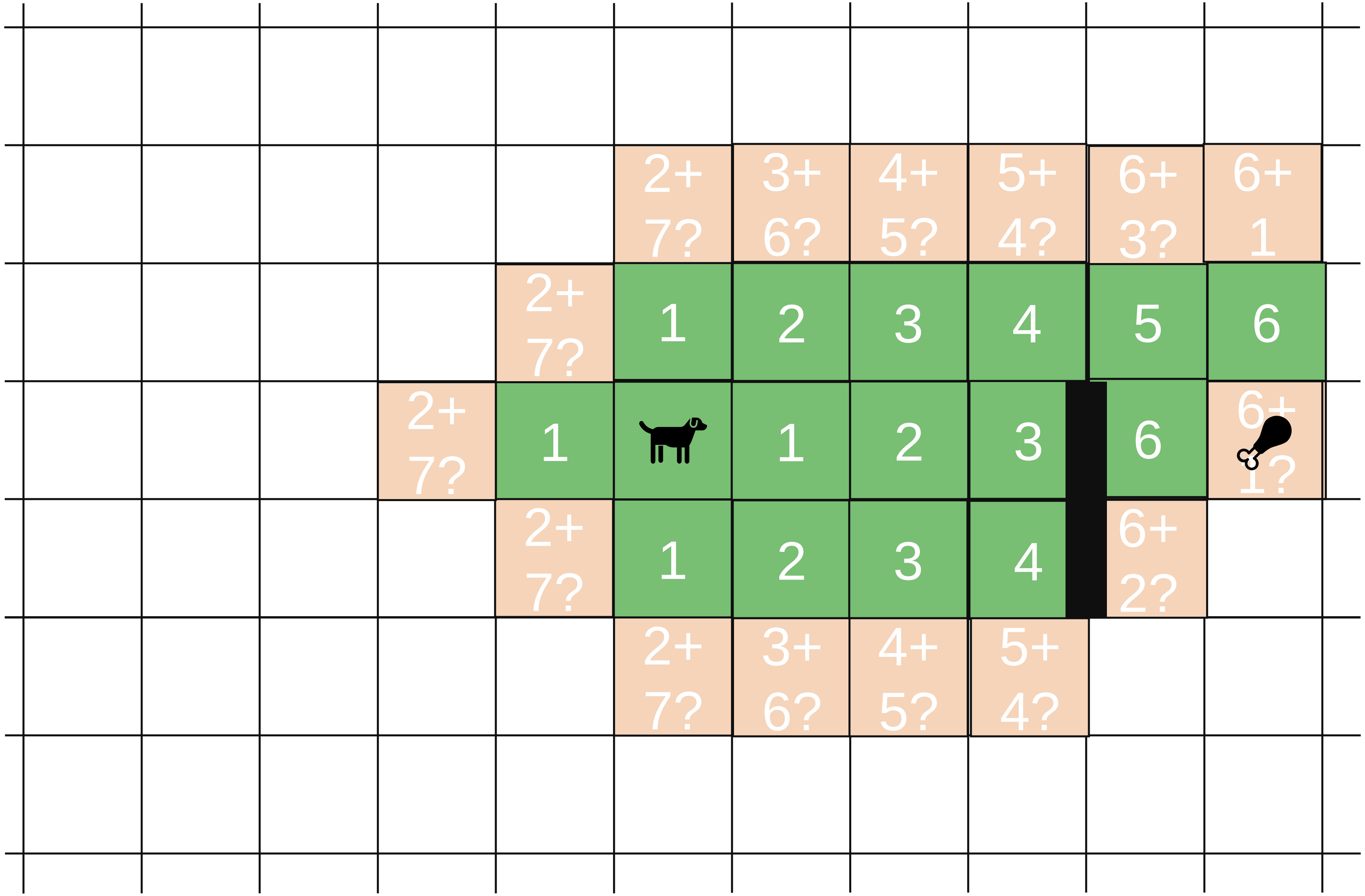
Greedily continue at smallest

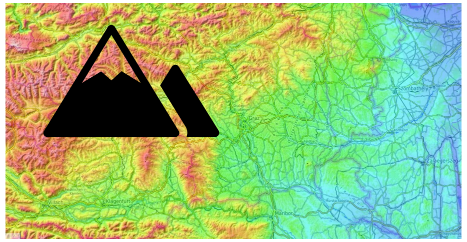
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





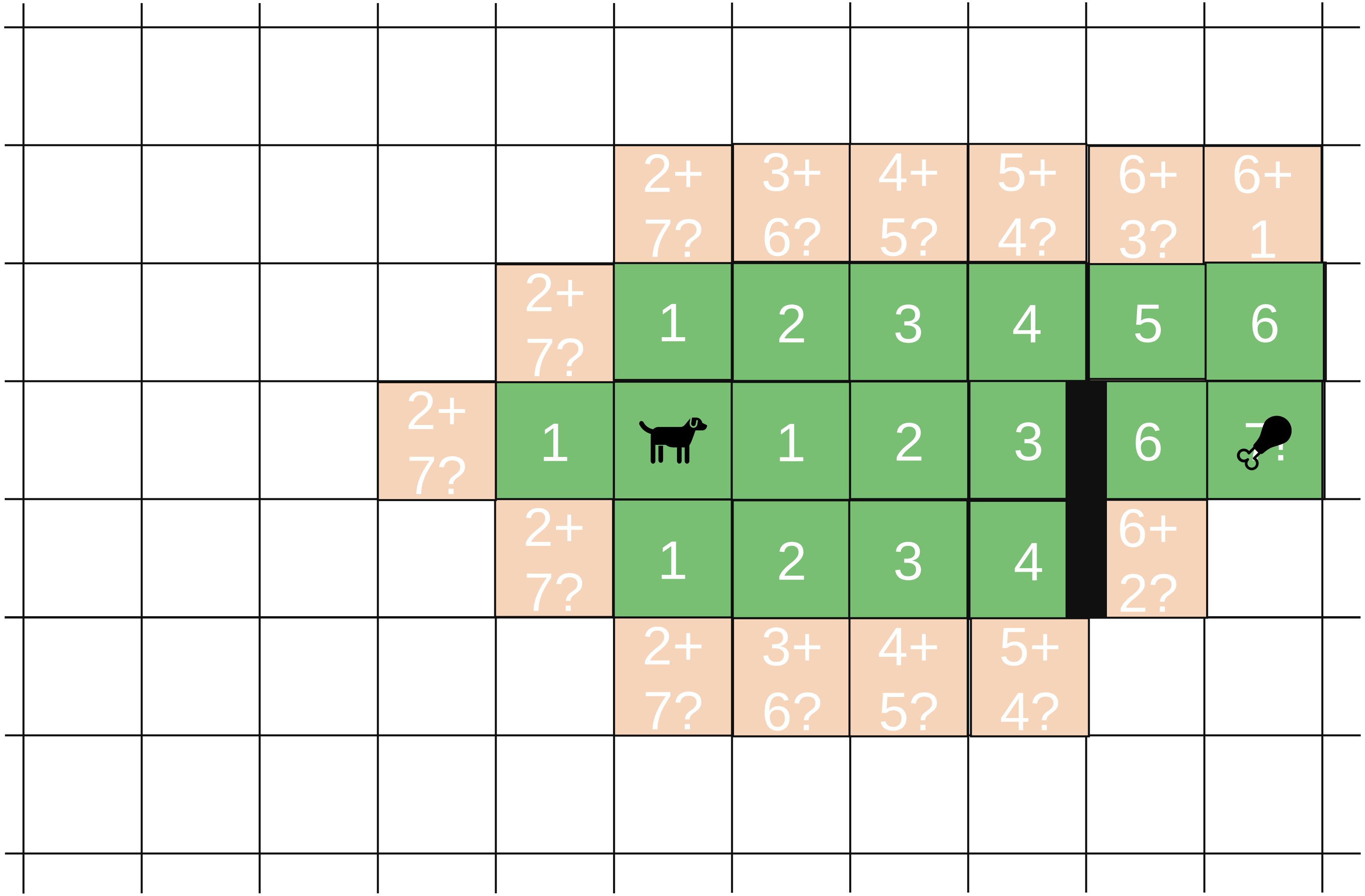
Greedily continue at smallest

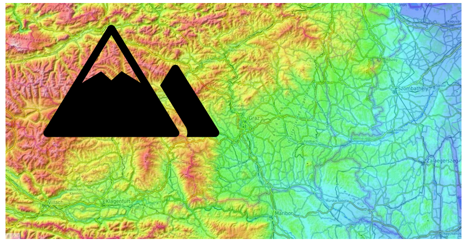
distance(s, u)

+

wanna be futureCost(u,t)

A*: Use row + column distance





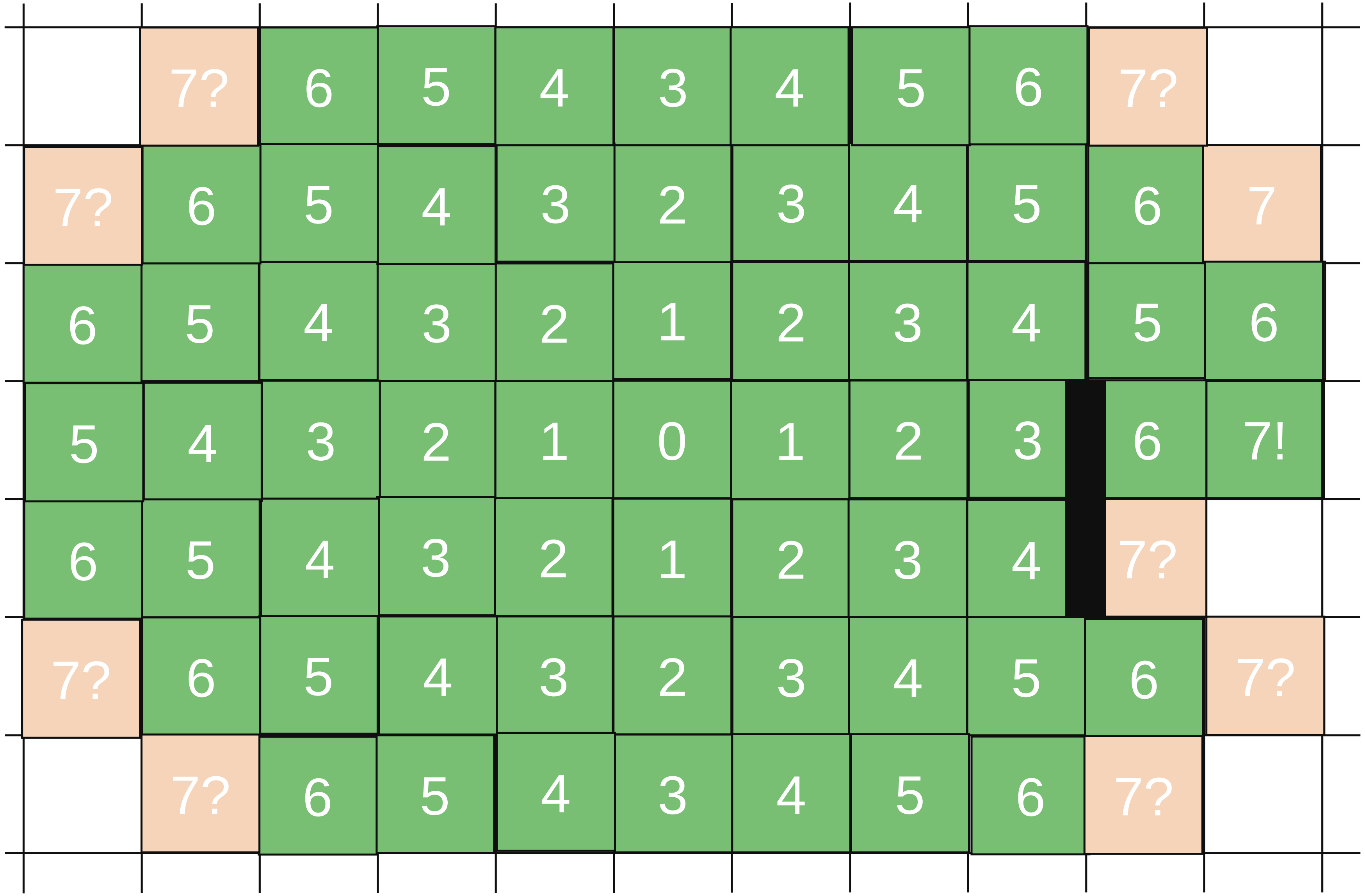
Greedily continue at smallest

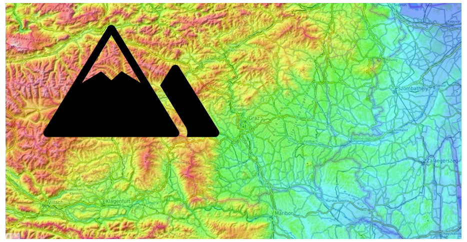
distance(s, u)

+

wanna be futureCost(u,t)

What Dijkstra would have done





Greedily continue at smallest

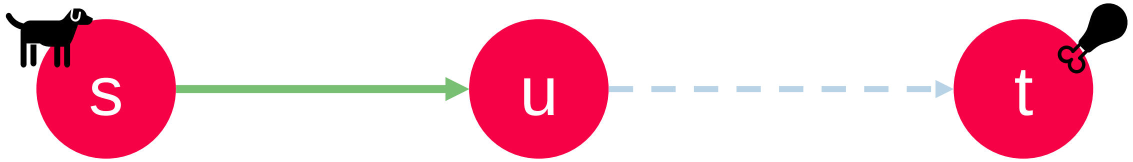
distance(s, u)

+

~~wa h fut (u,t)~~

A* Path Priorities

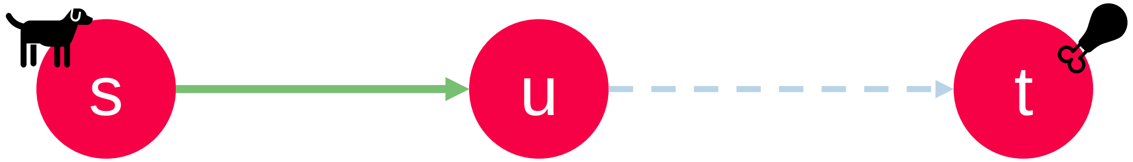
priority(u) = $\boxed{\text{distance(s, u)}}$ + $\boxed{\text{wanna be futureCost(u,t)}}$ *



priority of the path that ends in u

A* Path Priorities, based on Heuristic

priority(u) = $\boxed{\text{distance(s, u)}}$ + $\boxed{\text{heuristic(u,t)}}$



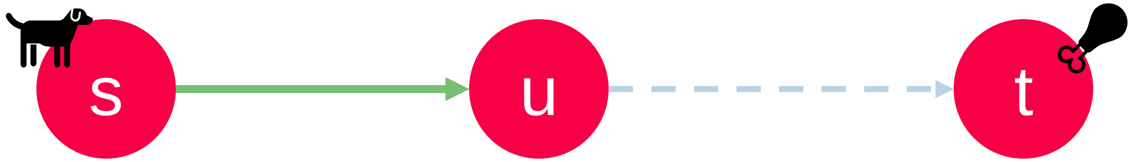
depends on the problem

priority of the path that ends in u

A heuristic is like a good educated guess.

A* Path Priorities, based on Heuristic

priority(u) = $\boxed{\text{distance(s, u)}}$ + $\boxed{\text{heuristic(u,t)}}$



depends on the problem

priority of the path that ends in u

A heuristic is like a good educated guess.

A* only works if your heuristic is good!

good, if it underestimates the future cost

Admissible Heuristic

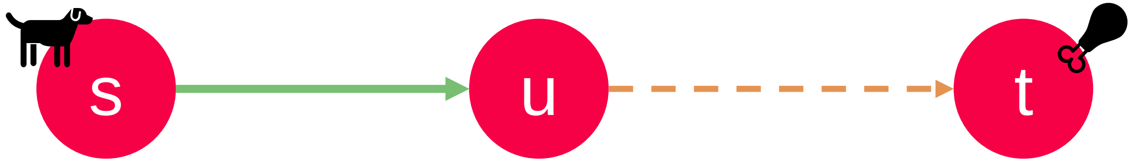
Definition: A heuristic is **admissible** if it underestimates the future cost , that is,

$h(u) = h(u,t) \leq d(u,t)$ holds.

heuristic(u,t)

$\leq$

priority(u) = distance(s, u) + futureCost(u,t) *



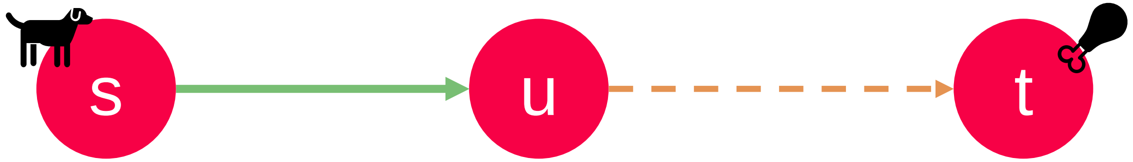
Admissible Heuristic

Definition: A heuristic is **admissible** if it underestimates the future cost , that is,
$h(u) = h(u, t) \leq d(u, t)$ holds.

heuristic(u,t)

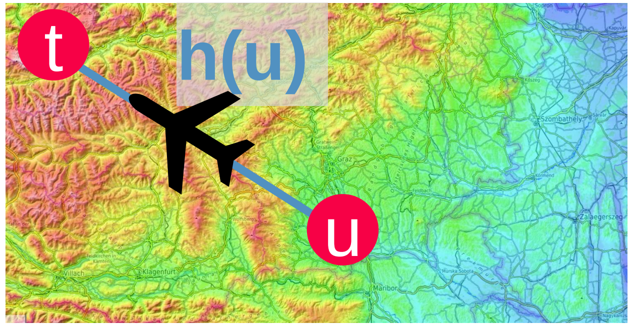
$\leq$

priority(u) = distance(s, u) + futureCost(u,t) *



Example for an admissible heuristic:

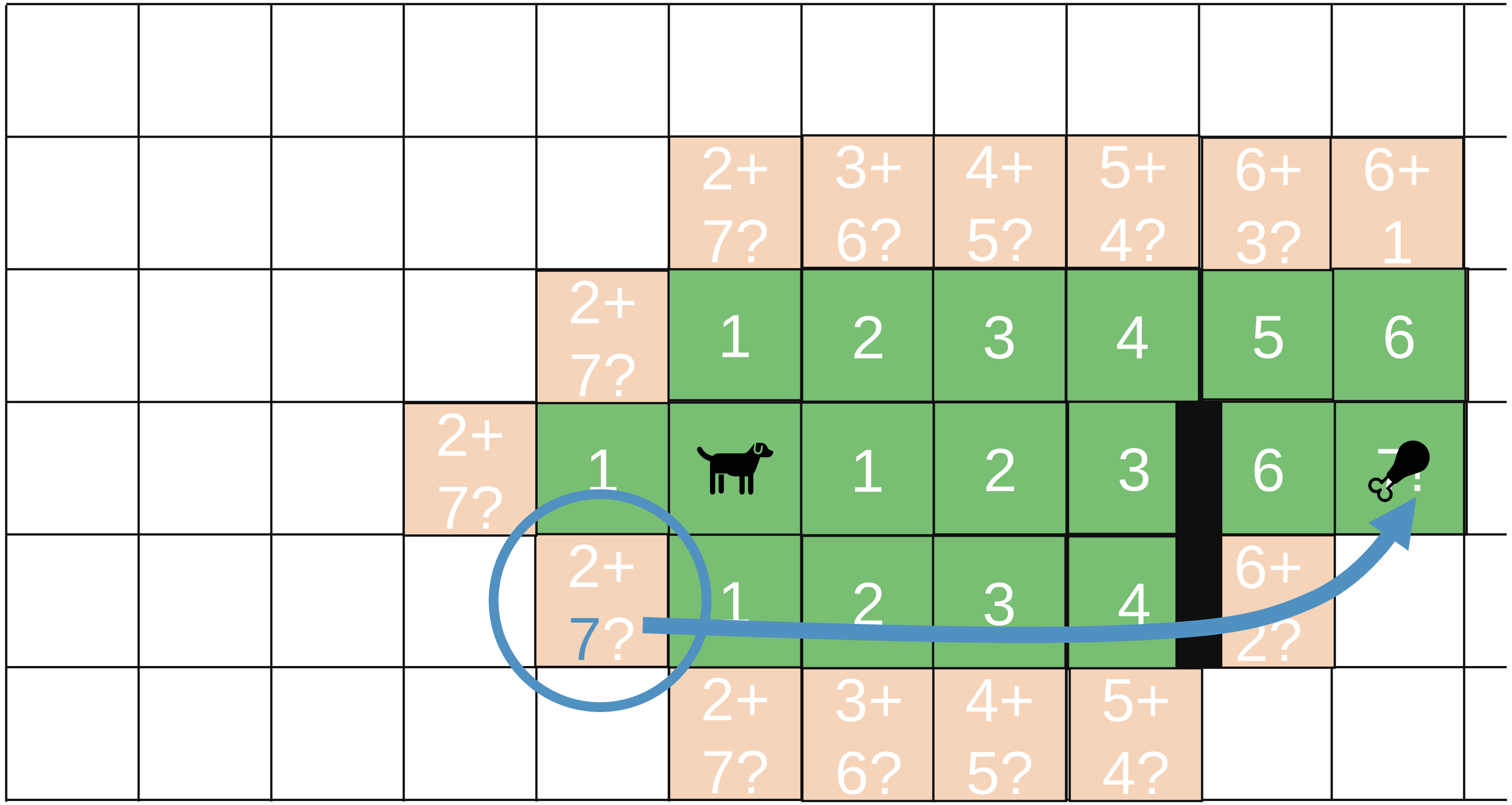
As the crow flies



Intuition (not a proof!): Admissible heuristic

An **admissible heuristic** always underestimates the true cost.

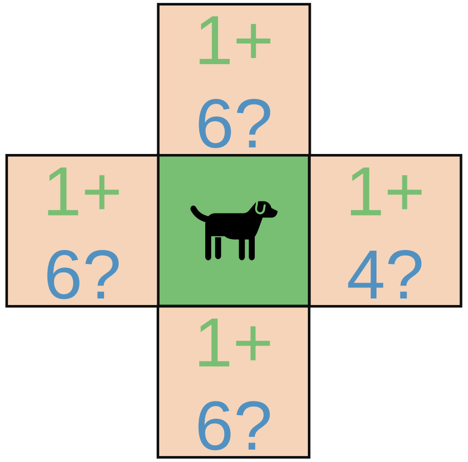
“Ignore path, as even if everything goes perfect (from here onwards), this path is still terrible…”



A*-Algorithm

Input: G(V,E,W) start point: s, end point: t

Initialize S={s}, g(s)=0, g(V\N(s))=∞, g(v)=w(s,v), v∈ $N(s)$



S: expanded/closed vertices
$V \setminus S$: the open vertices

s

g(v)+
h(u)?

$g(v)$ is the length of the best **known (!)** path from s to v

A*-Algorithm

Input: G(V,E,W) start point: s, end point: t

Initialize S={s}, g(s)=0, g(V\N(s))=∞, g(v)=w(s,v), v∈ $N(s)$

While t ∉ S **do**

u = $\operatorname{argmin}_{u \in V \setminus S}\{g(u) + h(u,t)\}$

For v s.t. {u,v}∈ E **do**

temp=min{g(v),g(u)+w(u,v)}

If temp<g(v) then:

g(v)=temp

S=S\{v} //does nothing if $v \notin S$

S= S ∪ $\{\boldsymbol{u}\}$

g(v)+ h(u)?

$g(v)$ is the length of the best **known (!)** path from s to v

$h(u,t)$ is a heuristic guess for the path from u to t.

S might decrease!

A*-Algorithm

Input: G(V,E,W) start point: s, end point: t

Initialize S={s}, g(s)=0, g(V\N(s))=∞, g(v)=w(s,v), v∈ $N(s)$

While t ∉ S **do**

u = $\text{argmin}_{u \in V \setminus S}\{g(u) + h(u,t)\}$

For v s.t. {u,v}∈ E **do**

temp=min{g(v),g(u)+w(u,v)}

If temp<g(v) then:

g(v)=temp

S=S\{v} //does nothing if $v \notin S$

S= **S** ∪ $\{\boldsymbol{u}\}$

g(v)+ h(u)?

$g(v)$ is the length of the best **known (!)** path from s to v

$h(u,t)$ is a heuristic guess for the path from u to t.

S might decrease!

Yannic Maus

Analysis of A*

Does it always terminate?

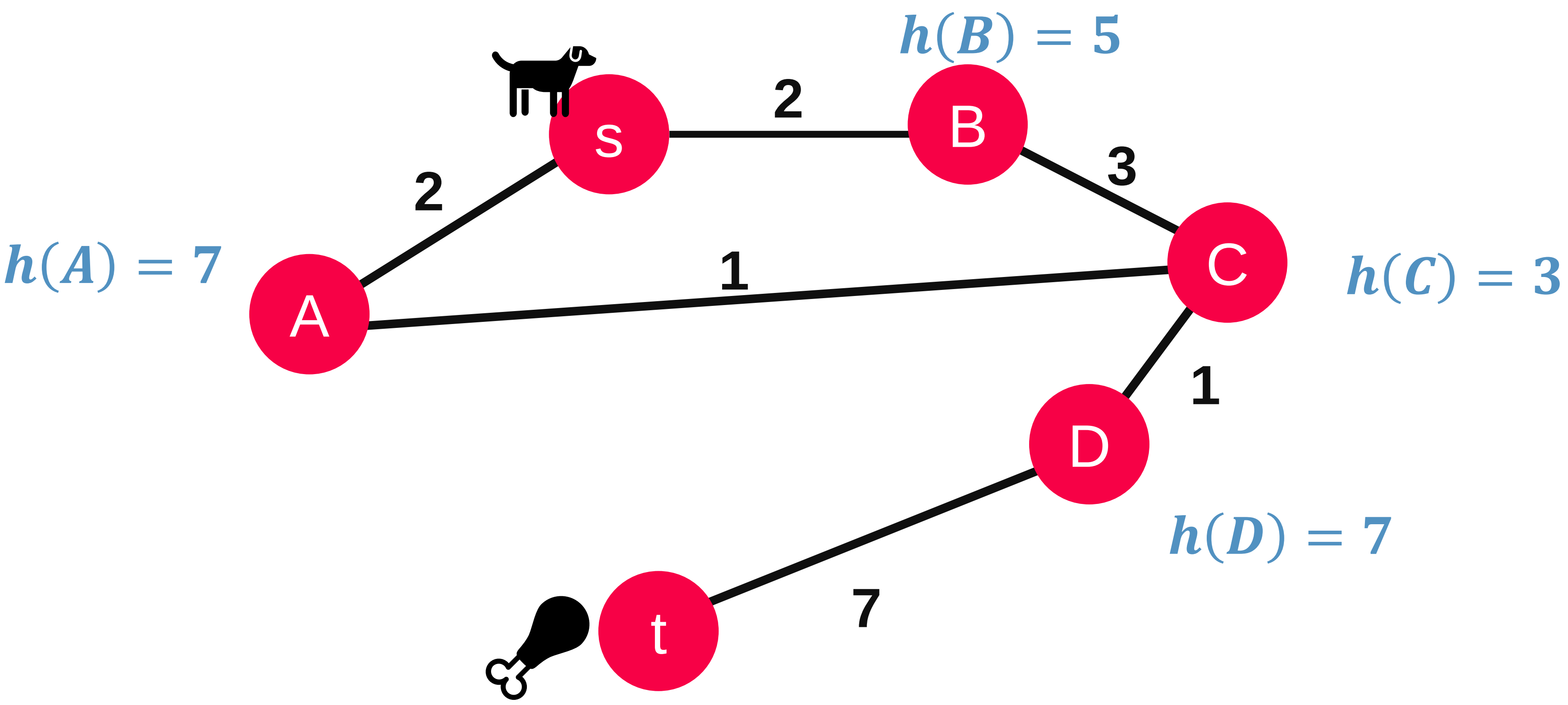
Does it always find an optimal path?

Runtime? Space consumption?

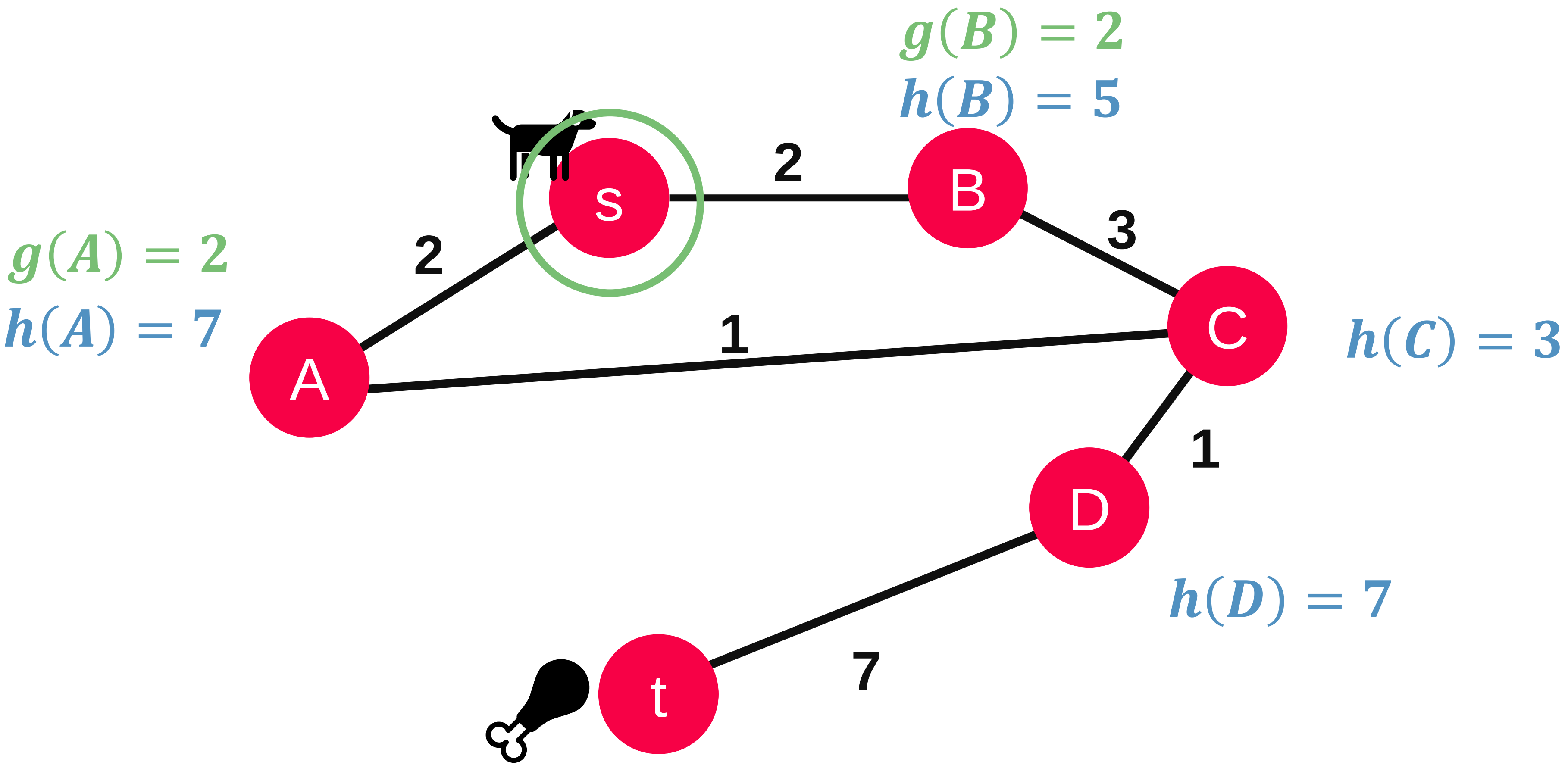
path finding with a heuristic

?

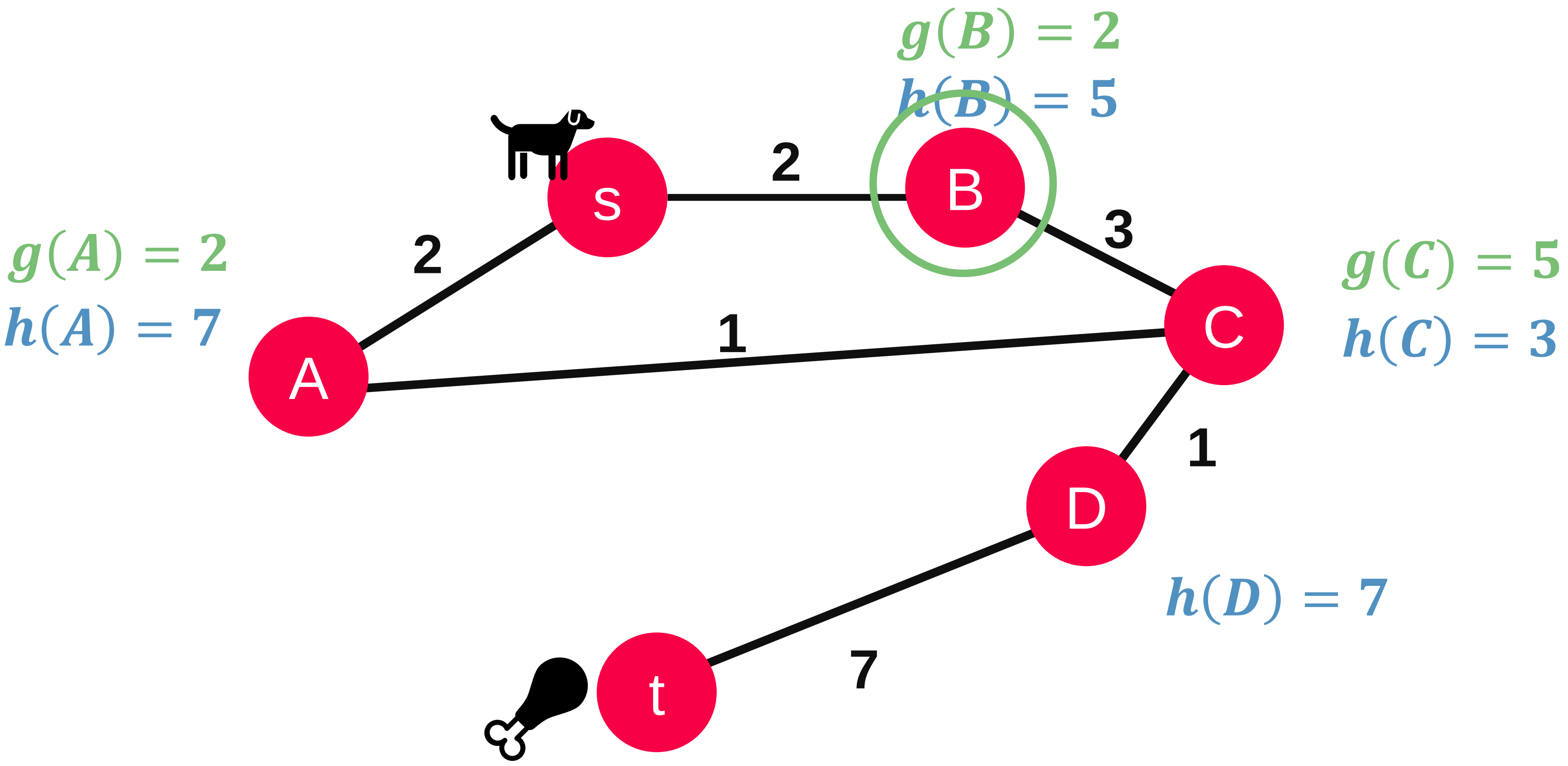
Downside: A node might be expanded multiple times



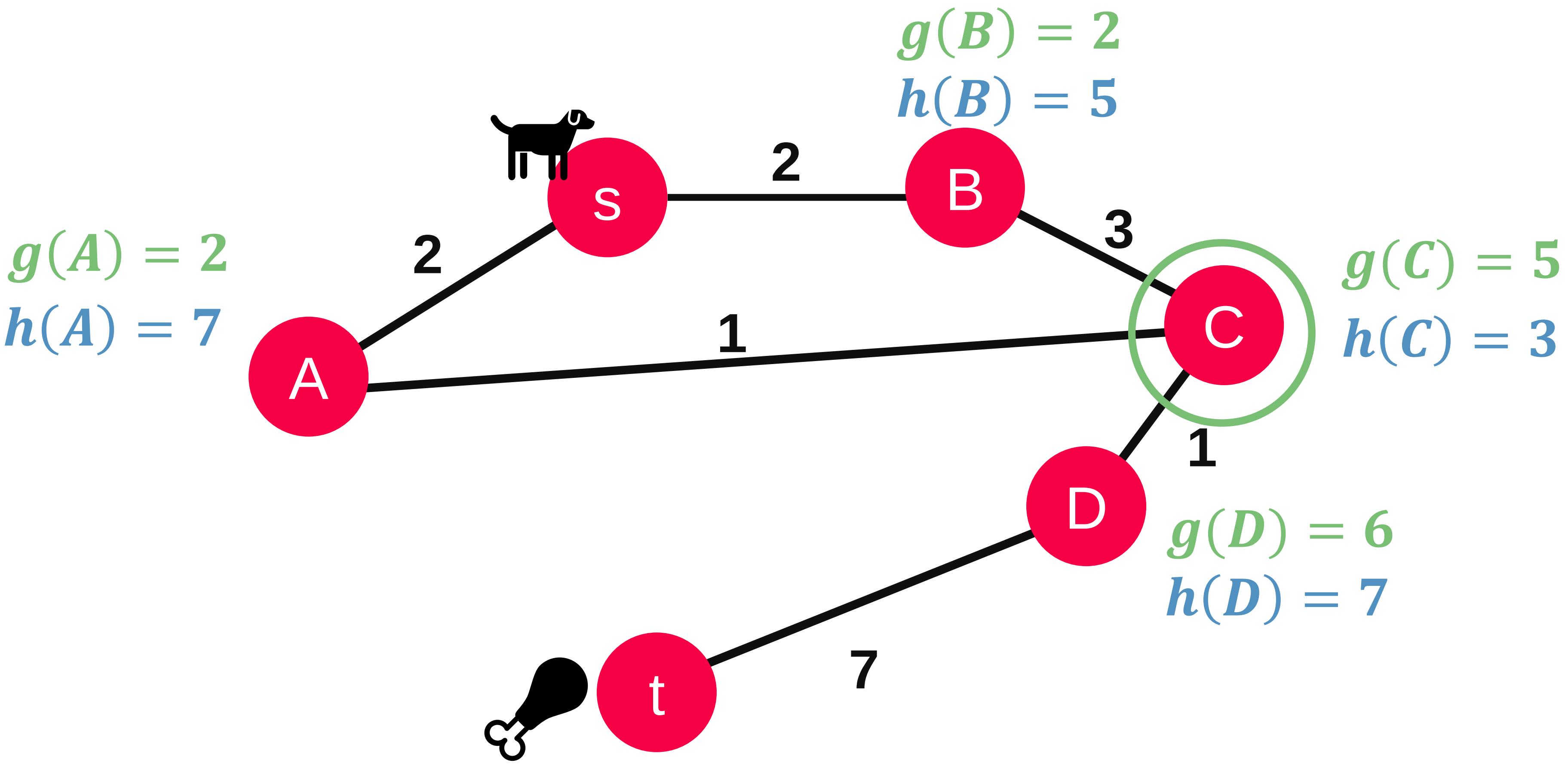
Downside: A node might be expanded multiple times



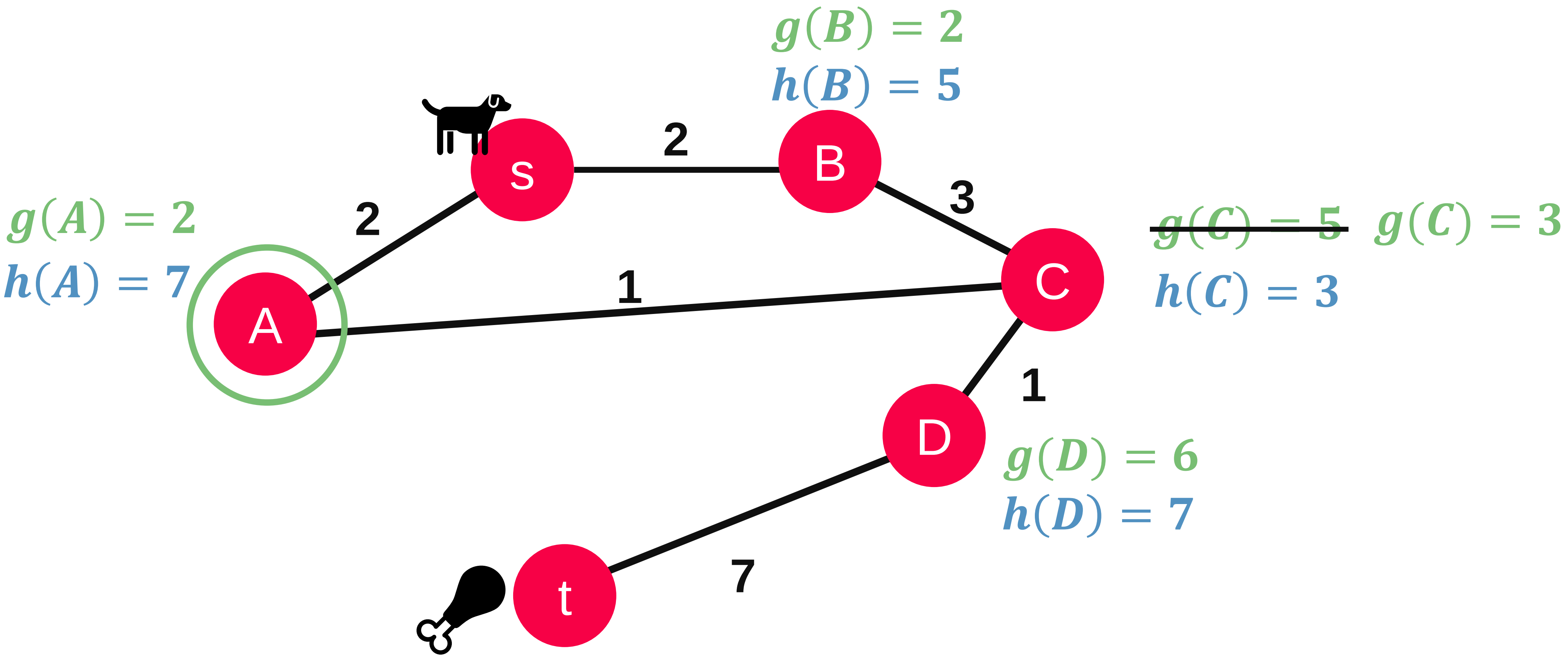
Downside: A node might be expanded multiple times



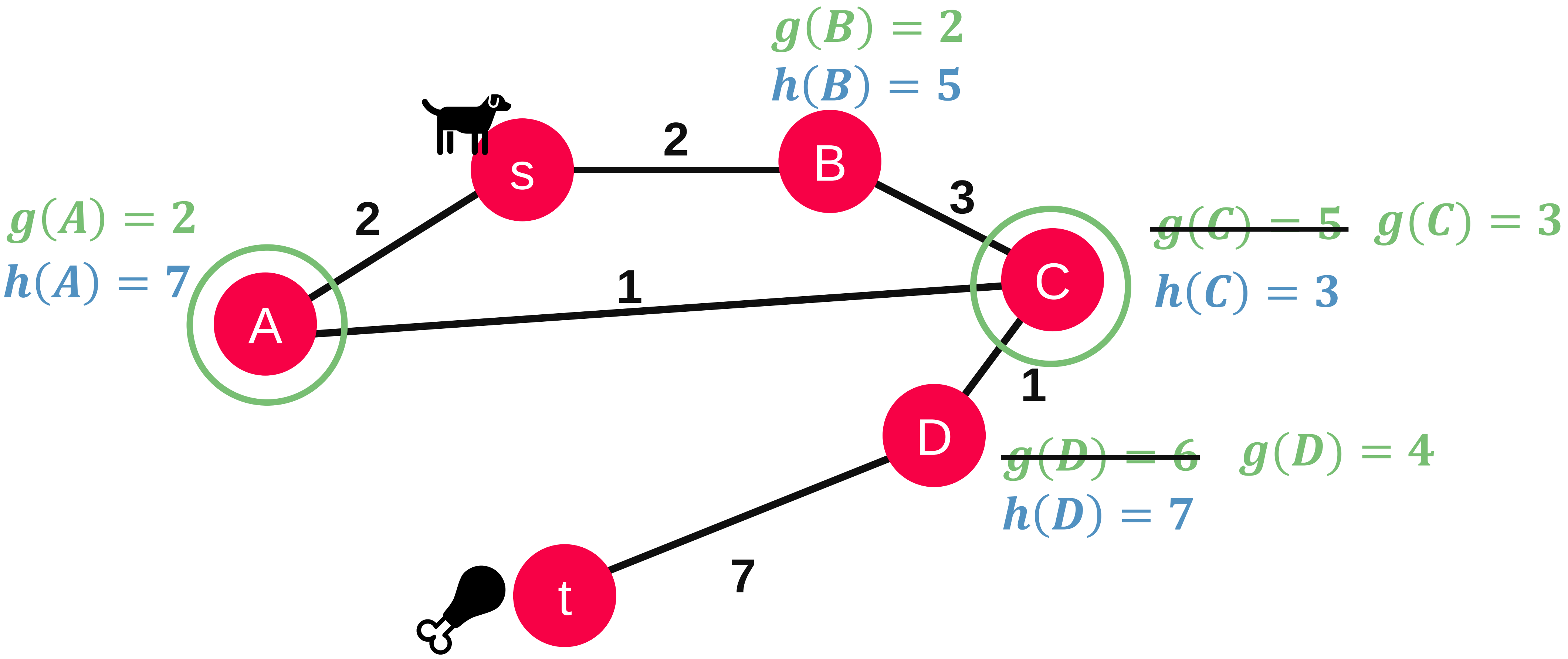
Downside: A node might be expanded multiple times



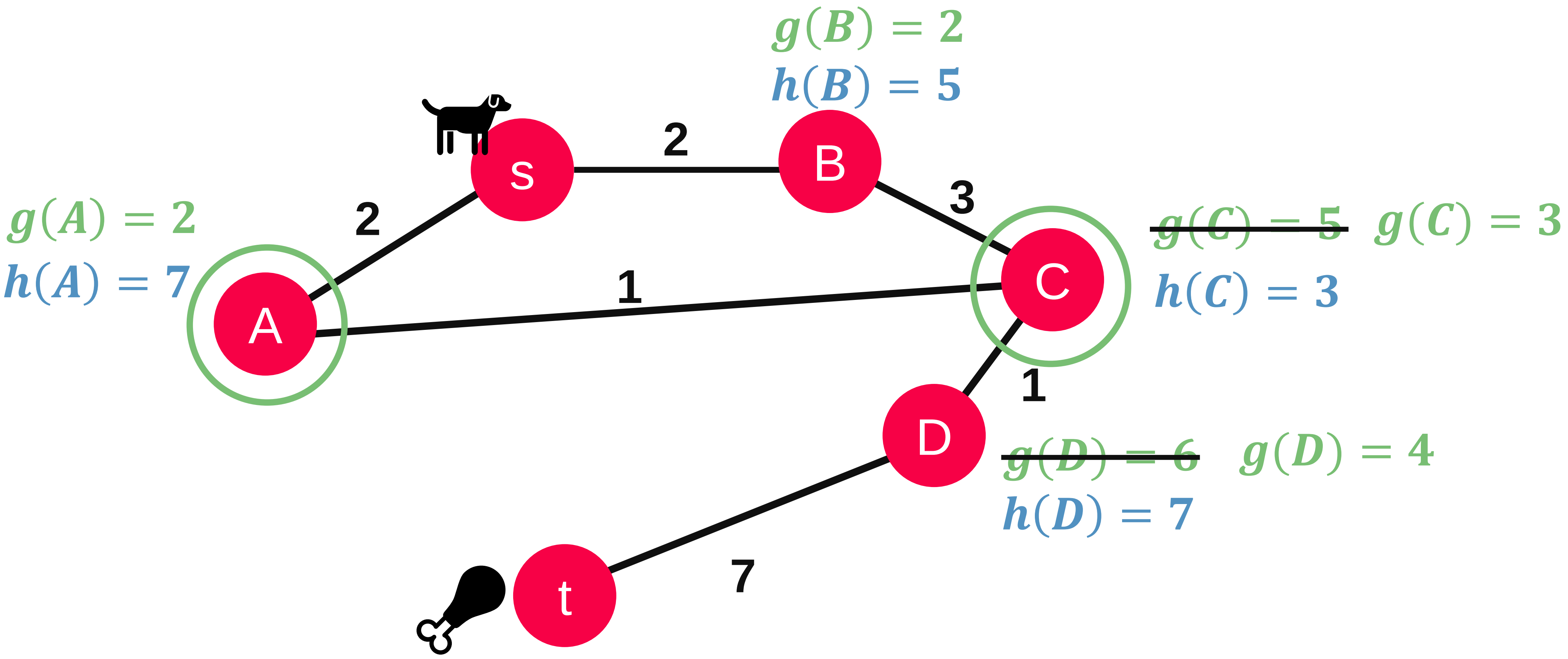
Downside: A node might be expanded multiple times



Downside: A node might be expanded multiple times



Downside: A node might be expanded multiple times



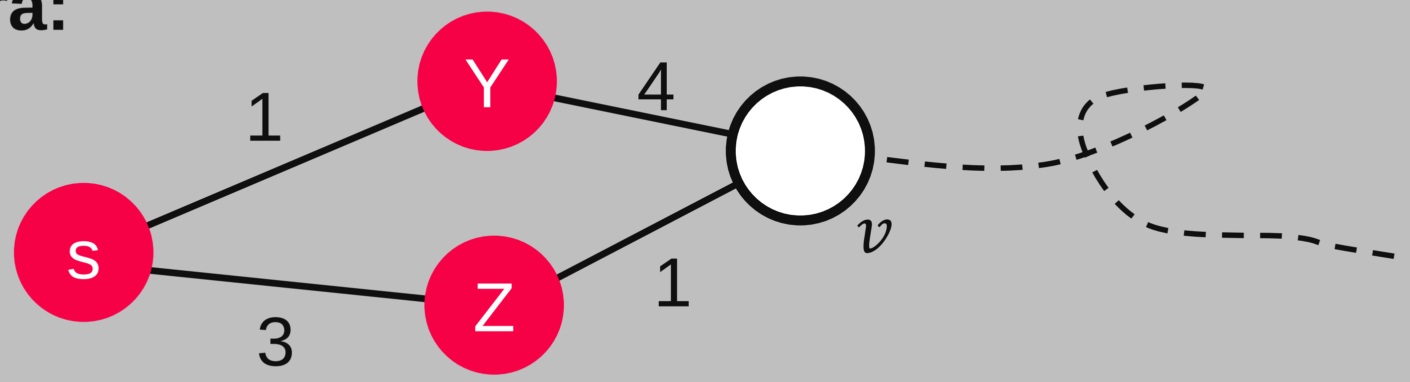
Order of expansion: S, B, C, A, C, …

Why did this happen?

Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

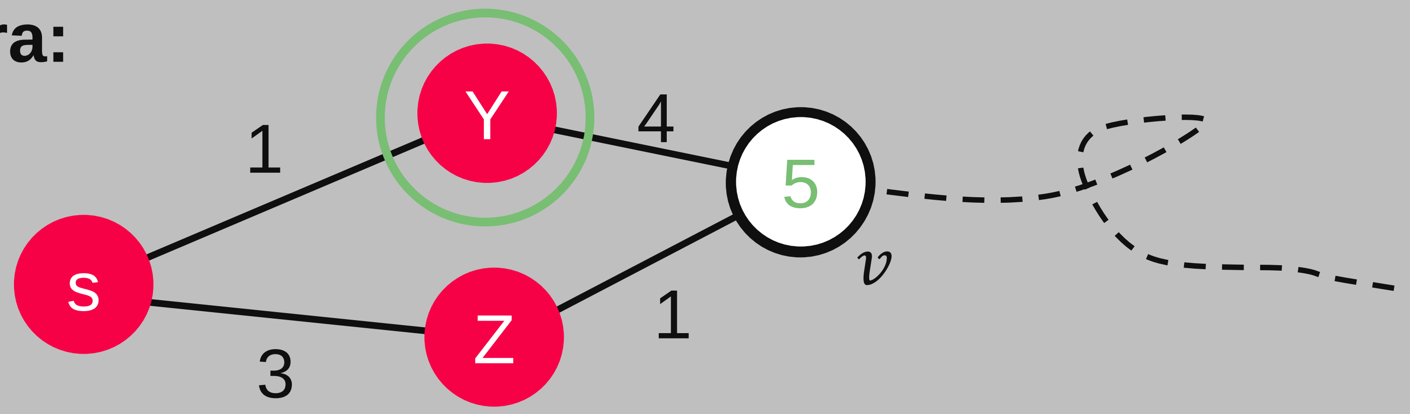
Example of Dijkstra:



Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

Example of Dijkstra:

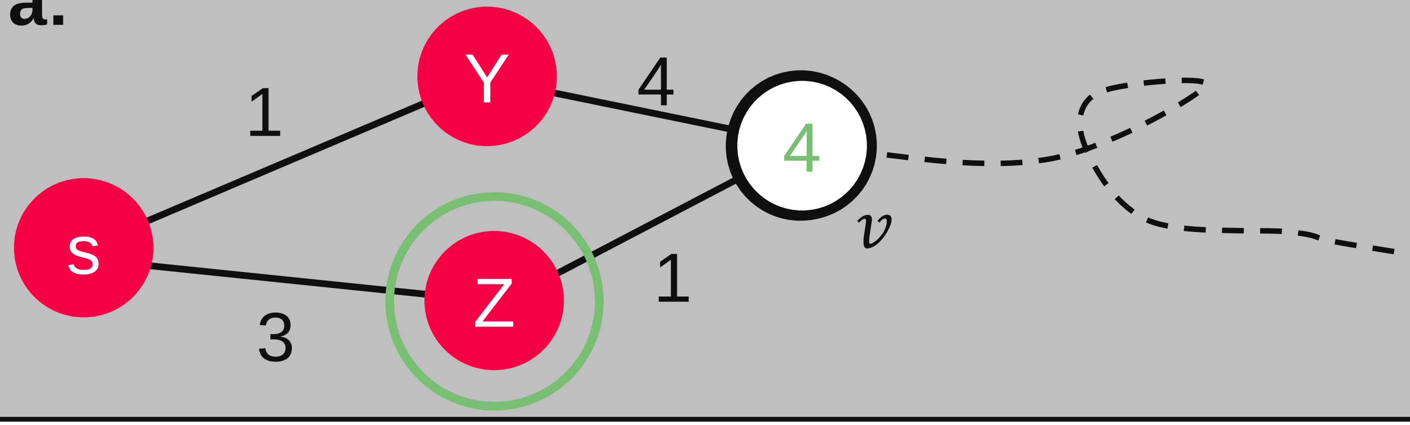


Why did this happen?

Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

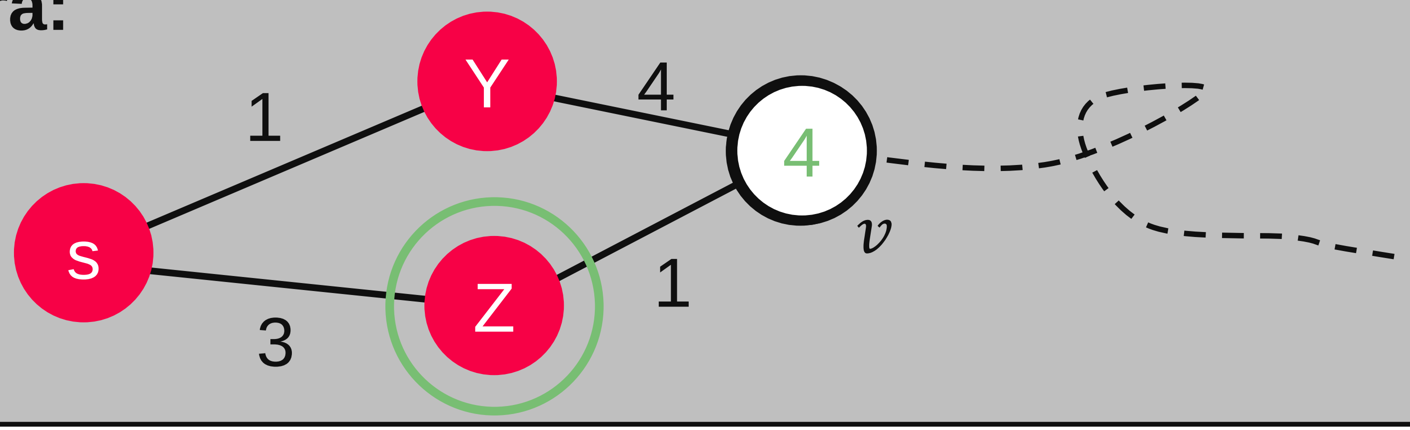
Example of Dijkstra:



Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

Example of Dijkstra:

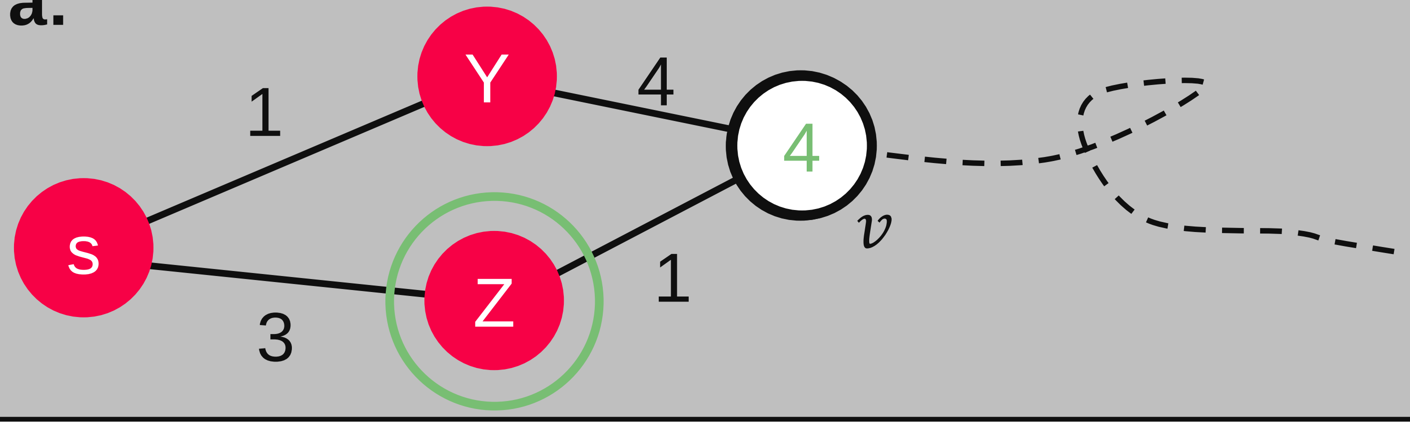


(in Dijkstra a node is only expanded once)

Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

Example of Dijkstra:



(in Dijkstra a node is only expanded once)

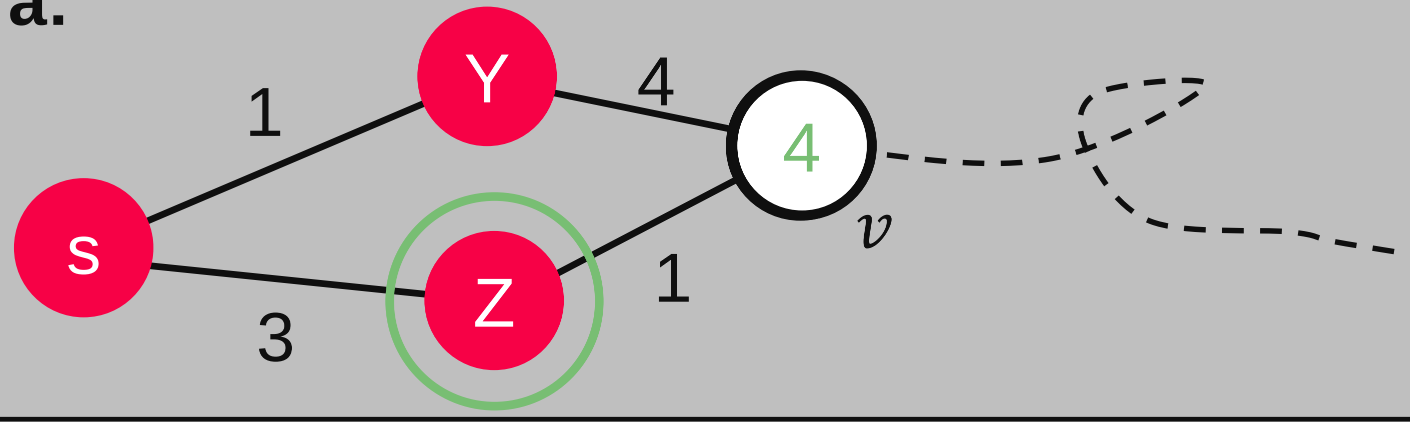
- in Dijkstra v will not be expanded before g(v) is finalized,

Why did this happen?

Why can a node be expanded more than once in A*?

In A*, similar to Dijkstra the g(v) value can change,

Example of Dijkstra:



(in Dijkstra a node is only expanded once)

- in Dijkstra v will not be expanded before g(v) is finalized,
- in A* a node might be expanded **before** its g(v) value is **finalized**!

Is A* Guaranteed to Terminate? Yes!

Proof:

- In each iteration of A* a new *acyclic path* is generated because:
  - Node added the first time: new path
  - node is “promoted” (`If temp<g(v)`): path is new because it is shorter

Is A* Guaranteed to Terminate? Yes!

Proof:

- In each iteration of A* a new *acyclic path* is generated because:
  - Node added the first time: new path
  - node is "promoted" (`If temp<g(v)`): path is new because it is shorter
- $\#steps \leq \#acyclicPaths$

Is A* Guaranteed to Terminate? Yes!

Proof:

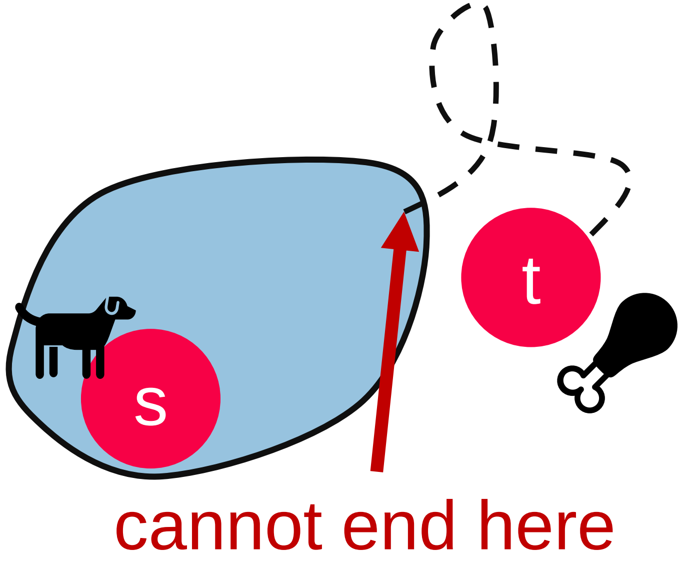
- In each iteration of A* a new *acyclic path* is generated because:
  - Node added the first time: new path
  - node is “promoted” (`If temp<g(v)`): path is new because it is shorter
- $\#steps \leq \#acyclicPaths$
- $\#acyclicPaths$ starting from s is finite

Is A* Guaranteed to Terminate? Yes!

Proof:

- In each iteration of A* a new *acyclic path* is generated because:
  - Node added the first time: new path
  - node is “promoted” (`If temp<g(v)`): path is new because it is shorter
- $\#steps \leq \#acyclicPaths$
- $\#acyclicPaths$ starting from s is finite

This also means we find t



Main A*-Theorem

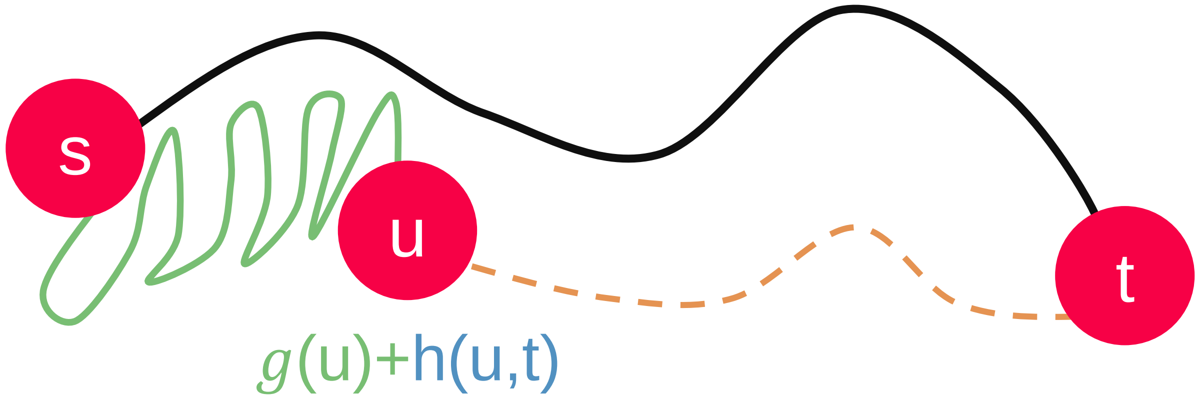
The A*-Algorithm with an **admissible** heuristic is:

- Complete: if a path to t exists, it finds one ✓
- Optimal: if a path to t exists, it always finds an optimal path

Intuitive Proof of Optimality

Wikipedia: *When A* terminates its search, it has found a path from start to goal whose actual cost is lower than the* ***estimated cost*** *of any path from start to goal through any open node (nodes in* $V \setminus S$*). When the heuristic is admissible, those* ***estimates are optimistic****, so A* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has.*

Wikipedia: *When A* terminates its search, it has found a path from start to goal whose actual cost is lower than the* ***estimated cost*** *of any path from start to goal through any open node (nodes in $V \setminus S$). When the heuristic is admissible, those* ***estimates are optimistic****, so A* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has.*



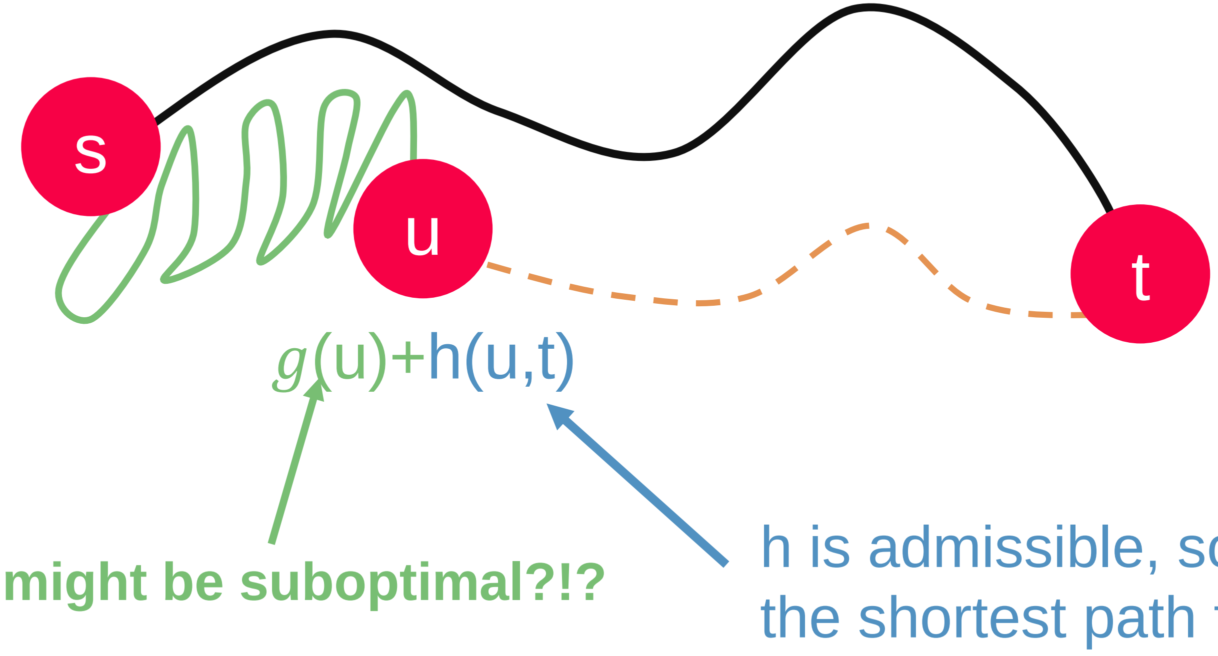
h is admissible, so better than the shortest path from u to t

Don’t expand u because large

g(u) + heuristic(u,t)

Intuitive Proof of Optimality

Wikipedia: *When A* terminates its search, it has found a path from start to goal whose actual cost is lower than the* ***estimated cost*** *of any path from start to goal through any open node (nodes in $V \setminus S$). When the heuristic is admissible, those* ***estimates are optimistic****, so A* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has.*

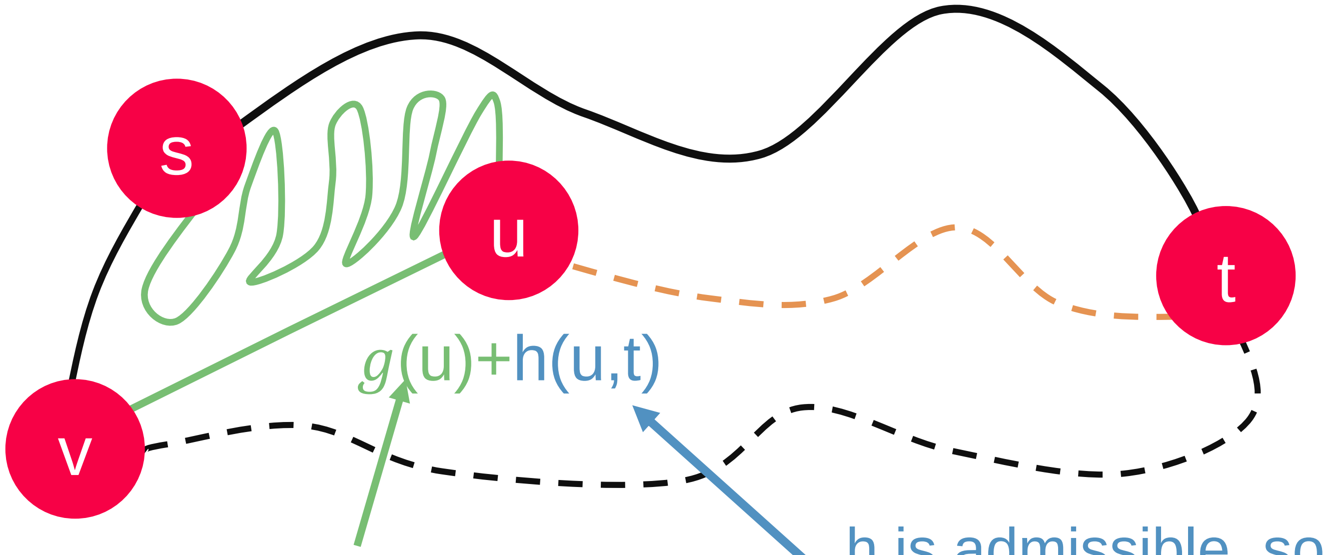


Don't expand u because large

g(u)	+	heuristic(u,t)

Intuitive Proof of Optimality

Wikipedia: *When A* terminates its search, it has found a path from start to goal whose actual cost is lower than the* ***estimated cost*** *of any path from start to goal through any open node (nodes in $V \setminus S$). When the* *heuristic is admissible*, *those* ***estimates are optimistic***, *so A* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has.*



Don't expand u because large

$$\boxed{g(\mathrm{u})} + \boxed{\mathrm{heuristic(u,t)}}$$

$$\neq$$

$$\boxed{\mathrm{distance(s,\ u)}} + \boxed{\mathrm{heuristic(u,t)}}$$

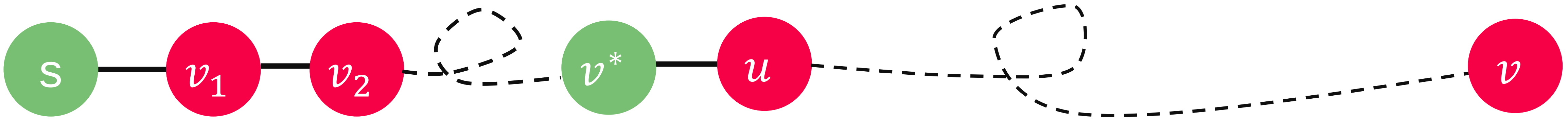
Optimality: Lemma 1

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(\text{u})}$ **=** $\boxed{\text{distance(s, u)}}$

Optimality: Lemma 1

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(\text{u})}$ **=** $\boxed{\text{distance(s, u)}}$

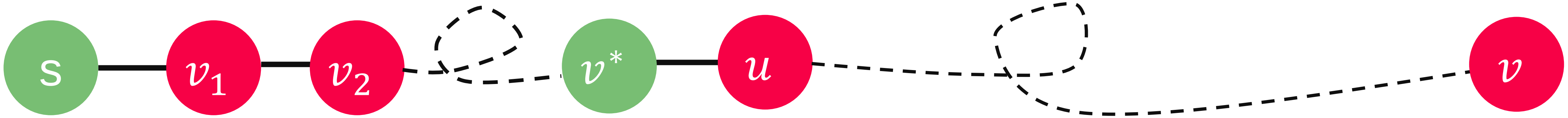
Proof: Let $P = (s = v_0, v_1, v_2, \dots, v = v_k)$.



Optimality: Lemma 1

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(u)} = \boxed{\text{distance}(s, u)}$

Proof: Let $P = (s = v_0, v_1, v_2, \ldots, v = v_k)$.

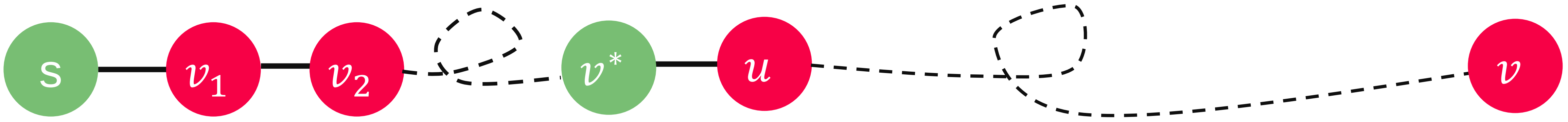


$C = \{v_i \in P \mid v_i \textbf{ closed}, \mathrm{g}(v_i) = d(s, v_i)\} \neq \emptyset,$

Optimality: Lemma 1

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(u)} = \boxed{\text{distance}(s, u)}$

Proof: Let $P = (s = v_0, v_1, v_2, \dots, v = v_k)$.

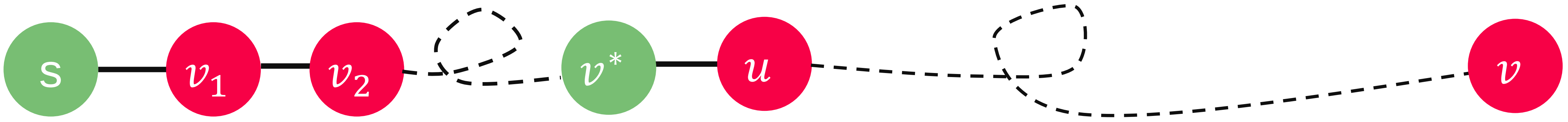


$C = \{v_i \in P \mid v_i \textbf{ closed}, g(v_i) = d(s, v_i)\} \neq \emptyset,$

- Let v^*, the vertex in C with highest index. $v^* \neq v$.
- Let u be the successor of v^* in P (possibly $u = v$).

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(\mathrm{u})} = \boxed{\text{distance(s, u)}}$

Proof: Let $P = (s = v_0, v_1, v_2, \dots, v = v_k)$.



$C = \{v_i \in P \mid v_i \ \mathbf{closed}, \mathrm{g}(v_i) = d(s, v_i)\} \neq \emptyset,$

- Let v^*, the vertex in C with highest index. $v^* \neq v$.
- Let u be the successor of v^* in P (possibly $u = v$).

$$\boldsymbol{g(u) \leq g(v^*) + w(v^*, u) = d(s, v^*) + w(v^*, u) = d(s, u) \leq g(u)}$$

v^* expanded

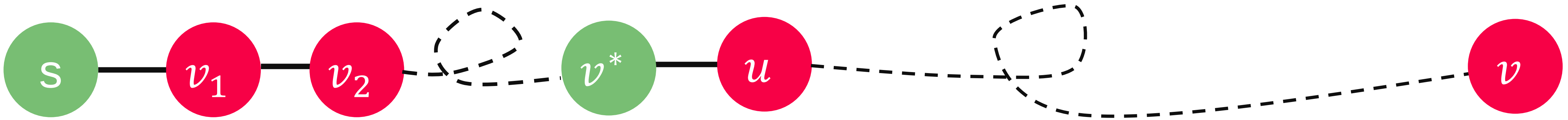
definition of v^*

P is optimal path

always

Lemma 1: Always, for every *open* node v and every optimal path P from s to v, there exists an ***open*** *node* u on P with $\boxed{g(\mathrm{u})}$ **=** $\boxed{\mathrm{distance(s, u)}}$

Proof: Let $P = (s = v_0, v_1, v_2, \dots, v = v_k)$.



$$C = \{v_i \in P \mid v_i \ \mathbf{closed}, \mathrm{g}(v_i) = d(s, v_i)\} \neq \emptyset,$$

- Let v^*, the vertex in C with highest index. $v^* \neq v$.
- Let u be the successor of v^* in P (possibly $u = v$).

P is optimal path

$$g(\boldsymbol{u}) \leq \boldsymbol{g}(v^*) + \boldsymbol{w}(v^*, \boldsymbol{u}) = \boldsymbol{d}(\boldsymbol{s}, v^*) + \boldsymbol{w}(v^*, \boldsymbol{u}) = \boldsymbol{d}(\boldsymbol{s}, \boldsymbol{u}) \leq \boldsymbol{g}(\boldsymbol{u})$$

v^* expanded

definition of v^*

always

u has to be open by definition of C

Optimality: Corollary 1

Corollary: Suppose h is admissible and A* has not terminated. Then, for any optimal path P from s to t, there is an open node u with

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} \leq \boxed{\text{distance(s,t)}}$$

Optimality: Corollary 1

Corollary: Suppose h is admissible and A* has not terminated. Then, for any optimal path P from s to t, there is an open node u with

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} \leq \boxed{\text{distance(s,t)}}$$

Proof: By Lemma 1, we have open node $u \in P$ *with* $\boxed{g(\text{u})} = \boxed{\text{distance(s, u)}}$

Optimality: Corollary 1

Corollary: Suppose h is admissible and A* has not terminated. Then, for any optimal path P from s to t, there is an open node u with

$$\boxed{g(\mathrm{u})} + \boxed{\mathrm{heuristic(u,t)}} \leq \boxed{\mathrm{distance(s,t)}}$$

Proof: By Lemma 1, we have open node $u \in P$ *with* $\boxed{g(\mathrm{u})} = \boxed{\mathrm{distance(s,\ u)}}$

$$\boxed{g(\mathrm{u})} + \boxed{\mathrm{heuristic(u,t)}} = \boxed{\mathrm{distance(s,\ u)}} + \boxed{\mathrm{heuristic(u,t)}}$$

Corollary: Suppose h is admissible and A* has not terminated. Then, for any optimal path P from s to t, there is an open node u with

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} \leq \boxed{\text{distance(s,t)}}$$

Proof: By Lemma 1, we have open node $u \in P$ *with* $\boxed{g(\text{u})} = \boxed{\text{distance(s, u)}}$

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} = \boxed{\text{distance(s, u)}} + \boxed{\text{heuristic(u,t)}}$$

$$\leq \boxed{\text{distance(s, u)}} + \boxed{\text{futureCost(u,t)}}$$

h admissible

Optimality: Corollary 1

Corollary: Suppose h is admissible and A* has not terminated. Then, for any optimal path P from s to t, there is an open node u with

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} \leq \boxed{\text{distance(s,t)}}$$

Proof: By Lemma 1, we have open node $u \in P$ *with* $\boxed{g(\text{u})} = \boxed{\text{distance(s, u)}}$

$$\boxed{g(\text{u})} + \boxed{\text{heuristic(u,t)}} = \boxed{\text{distance(s, u)}} + \boxed{\text{heuristic(u,t)}}$$

$$\leq \boxed{\text{distance(s, u)}} + \boxed{\text{futureCost(u,t)}}$$

$$= \boxed{\text{distance(s,t)}}$$

h admissible

u on optimal path P

Theorem 1

The A*-Algorithm with an **admissible** heuristic is:

- Complete: if a path to t exists, it finds one ✔
- **Optimal: if a path to t exists, it always finds an optimal path**

The A*-Algorithm with an **admissible** heuristic is:

- Complete: if a path to t exists, it finds one ✓
- **Optimal: if a path to t exists, it always finds an optimal path**

Proof (Optimality):

Suppose A* terminates at t with a suboptimal path, i.e., in the last step we expanded t with

$$\boxed{g(\mathrm{t})} + \overbrace{\boxed{\text{heuristic(t,t)}}}^{0} > \boxed{\text{distance(s, t)}}$$

Theorem 1

The A*-Algorithm with an **admissible** heuristic is:

- Complete: if a path to t exists, it finds one ✔
- **Optimal: if a path to t exists, it always finds an optimal path**

Proof (Optimality):

Suppose A* terminates at t with a suboptimal path, i.e., in the last step we expanded t with

$$g(t) + \overbrace{\text{heuristic}(t,t)}^{0} > \text{distance}(s, t)$$

But, by the corollary, there existed just before "expanding t", there is an open node u on an optimal path with

$$g(u) + \text{heuristic}(u,t) \leq \text{distance}(s, t)$$

Contradiction!

A* is Optimally Efficient

Informal Remark: *A* with an admissible heuristic is also*
Optimally efficient: There is no other algorithm that finds the solution faster with the same heuristic (*A* expands a minimal number of vertices)*

The remark is not about the number of expansions!

A*-Theorem

A*-Theorem

A*-Theorem

We have seen: A node might be expanded several times with an admissible heuristic

Def: A heuristic is **consistent** if for every edge $\{u, v\} \in E$ we have

$$h(u) \leq w(u, v) + h(v)$$

With an **admissible** and **consistent** heuristic, A* is guaranteed to find an optimal path without expanding any node more than once.

A*-Theorem

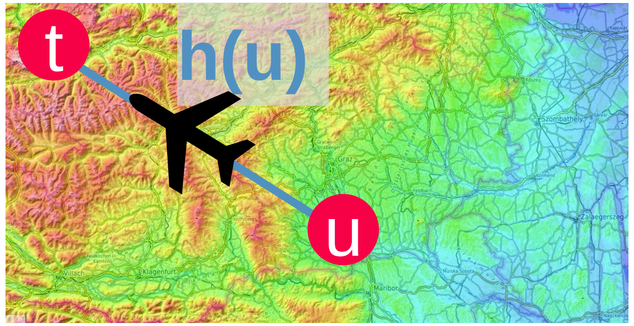
We have seen: A node might be expanded several times with an admissible heuristic

Def: A heuristic is **consistent** if for every edge $\{u, v\} \in E$ we have

$$h(u) \leq w(u, v) + h(v)$$

With an **admissible** and **consistent** heuristic, A* is guaranteed to find an optimal path without expanding any node more than once.

“as the crow flies”-heuristic is consistent



A*-Theorem

We have seen: A node might be expanded several times with an admissible heuristic

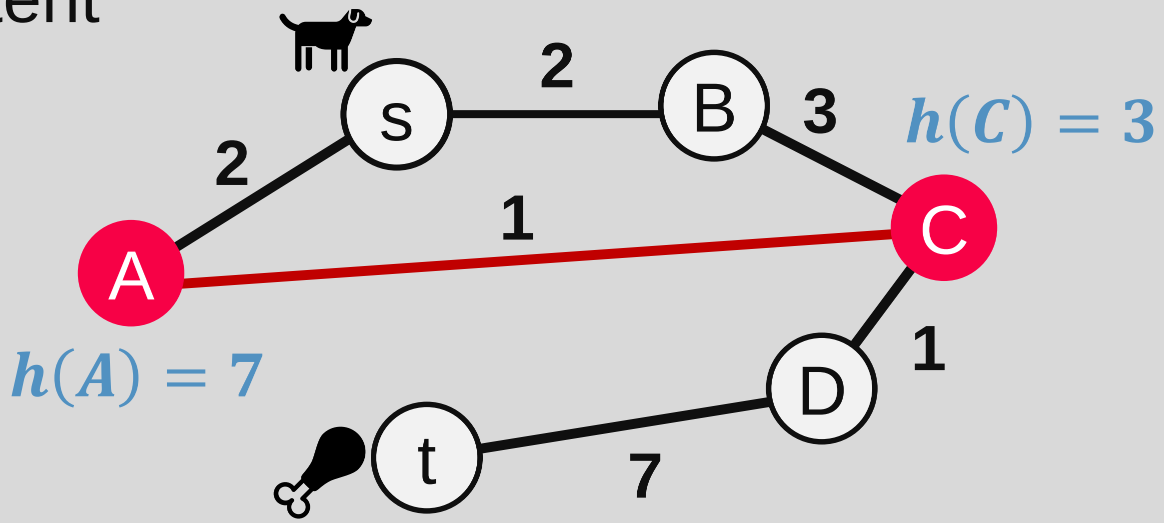
Def: A heuristic is **consistent** if for every edge $\{u, v\} \in E$ we have

$$h(u) \leq w(u, v) + h(v)$$

With an **admissible** and **consistent** heuristic, A* is guaranteed to find an optimal path without expanding any node more than once.

Heuristic in previous example was not consistent

$$\boldsymbol{h(A) = 7 \not\leq w(A, C) + h(C) = 4}$$



More on the choice of the heuristic

heuristic(u,t) = 0

heuristic(u,t) = underestimate

heuristic(u,t) = perfect distance

heuristic(u,t) = overestimate

Dijkstra

More on the choice of the heuristic

heuristic(u,t) = 0

heuristic(u,t) = underestimate

heuristic(u,t) = perfect distance

heuristic(u,t) = overestimate

Only possible heuristic,
might be faster than Dijkstra

More on the choice of the heuristic

heuristic(u,t) = 0

heuristic(u,t) = underestimate

heuristic(u,t) = perfect distance

heuristic(u,t) = overestimate

Only finds best paths, super fast!

Requires perfect knowledge!

More on the choice of the heuristic

heuristic(u,t) = 0

heuristic(u,t) = underestimate

heuristic(u,t) = perfect distance

heuristic(u,t) = overestimate

Also fast (not admissible),
but might not find the paths! 💀

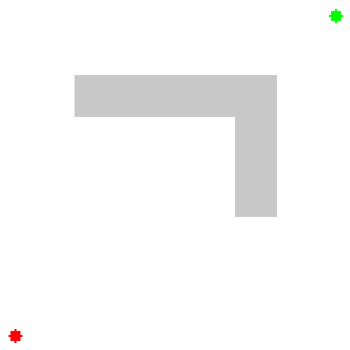
Path construction

Remarks:

- path construction similar to Dijkstra (we will not detail on this):
  - always remember a pointer to the best (known) predecessor
  - → sufficient to reconstruct the path to the start node s.
  - A* works in directed graphs

Example (Wikipedia)

(no comment)
(Wikipedia)



Runtime/Space

- It depends on…
  - the heuristic
  - Implementation of `argmin` (e.g. binary heap)
  - Implementation of set S (e.g., as array)

worst case

Runtime/Space

- It depends on…
  - the heuristic
  - Implementation of `argmin` (e.g. binary heap)
  - Implementation of set S (e.g., as array)

worst case

Bottom line:

- worst case runtime is not interesting for A*
- Often much better than Dijkstra
- In applications **space** is the bottleneck ($g(v) + h(v)$ is stored for each visited v)

Runtime/Space

- It depends on…
  - the heuristic
  - Implementation of `argmin` (e.g. binary heap)
  - Implementation of set S (e.g., as array)

worst case

Bottom line:

- worst case runtime is not interesting for A*
- Often much better than Dijkstra
- In applications **space** is the bottleneck ($g(v) + h(v)$ is stored for each visited v)

15-Puzzle: Search space has a node for each configuration: 16!=20.922.789.888.000 vertices!

Runtime/Space

- It depends on…
  - the heuristic
  - Implementation of `argmin` (e.g. binary heap)
  - Implementation of set S (e.g., as array)

worst case

Bottom line:

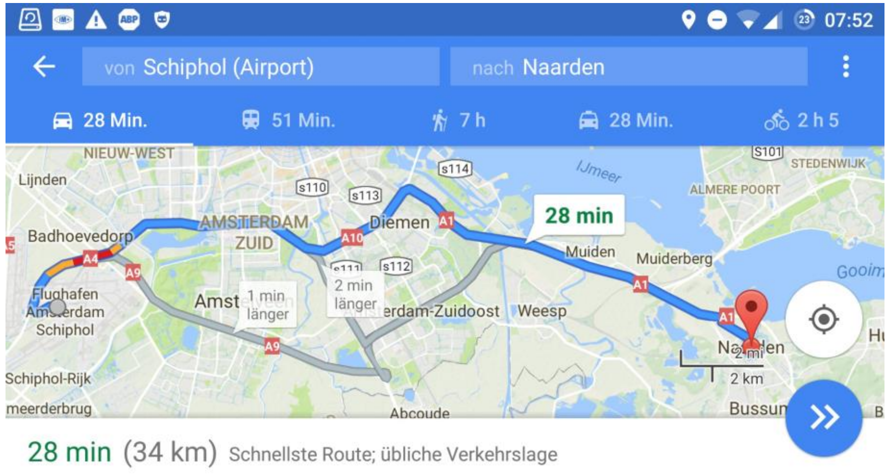
- worst case runtime is not interesting for A*
- Often much better than Dijkstra
- In applications **space** is the bottleneck ($g(v) + h(v)$ is stored for each visited v)

15-Puzzle: Search space has a node for each configuration: 16!=20.922.789.888.000 vertices!

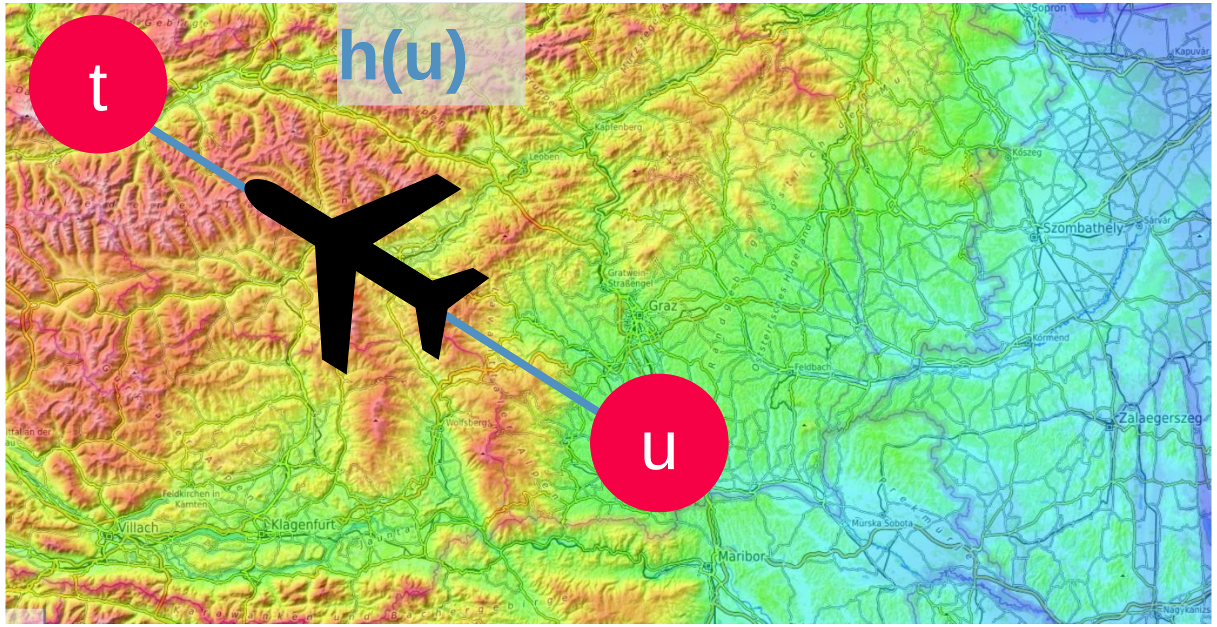
→ **Memory bounded heuristic search: Iterative deepening A*, …**

Why doesn't Google Maps Pre-Compute Paths?

- How many nodes are in the Google Maps graph?
  Answer: About $N = 75$ million
- How many sets of paths would they need to generate?
  Answer: (roughly) N^2
- How long would that take?
  Answer: $6 \times\ 10^{15}$ seconds,
  or... 190 million years
  (numbers might be outdated)



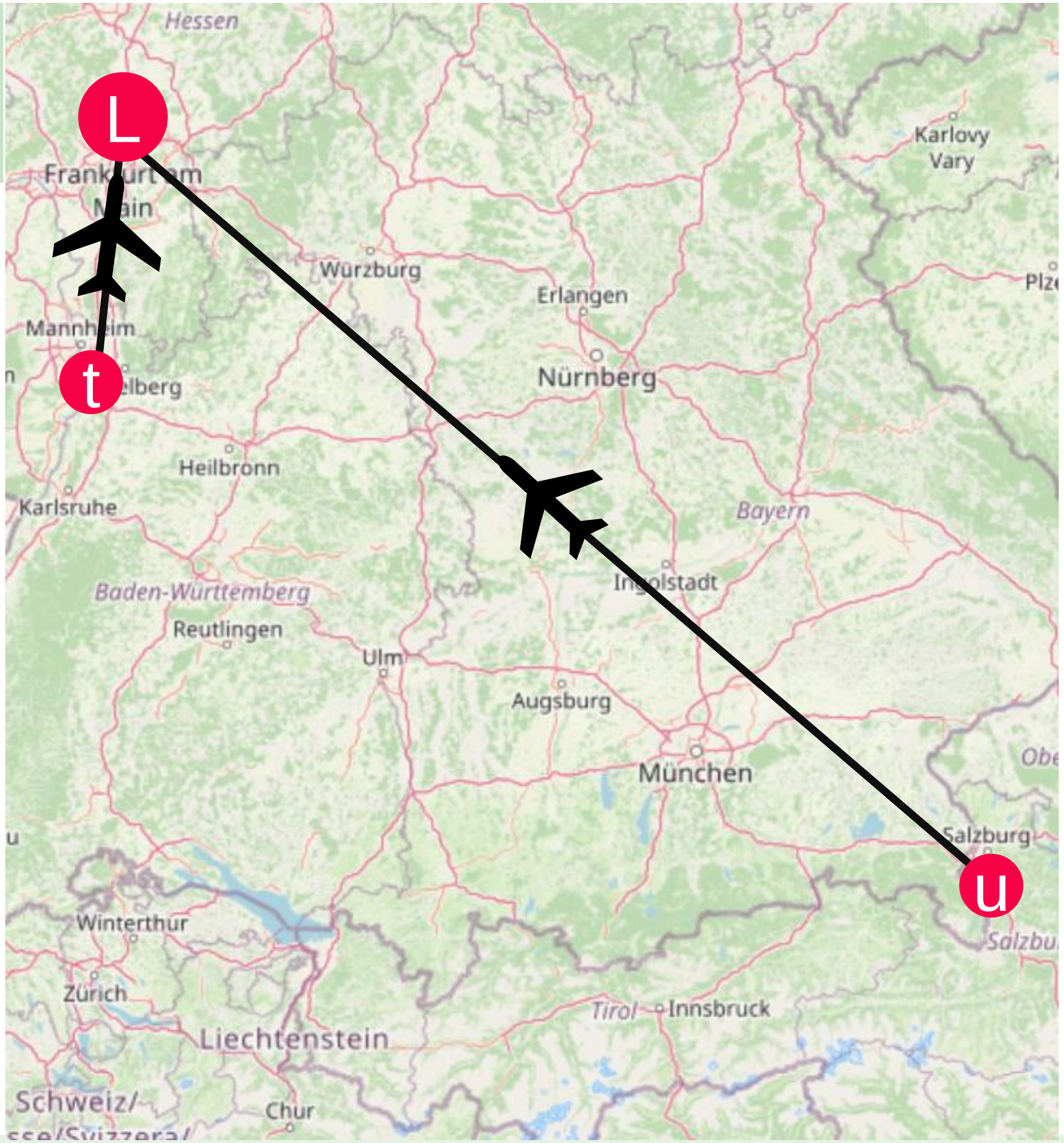
- as the crow flies (straight-line distance)
  - calculate the straight-line distance from u to t, and divide by the speed on the fastest highway



Landmark heuristic

Landmark heuristic with landmark L:

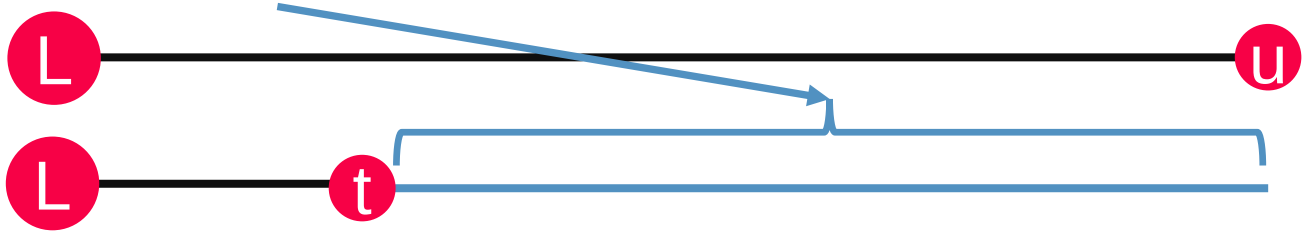
$$h\,(u,t) = |d(u,L) - d(t,L)| \le \; d(u,t)$$

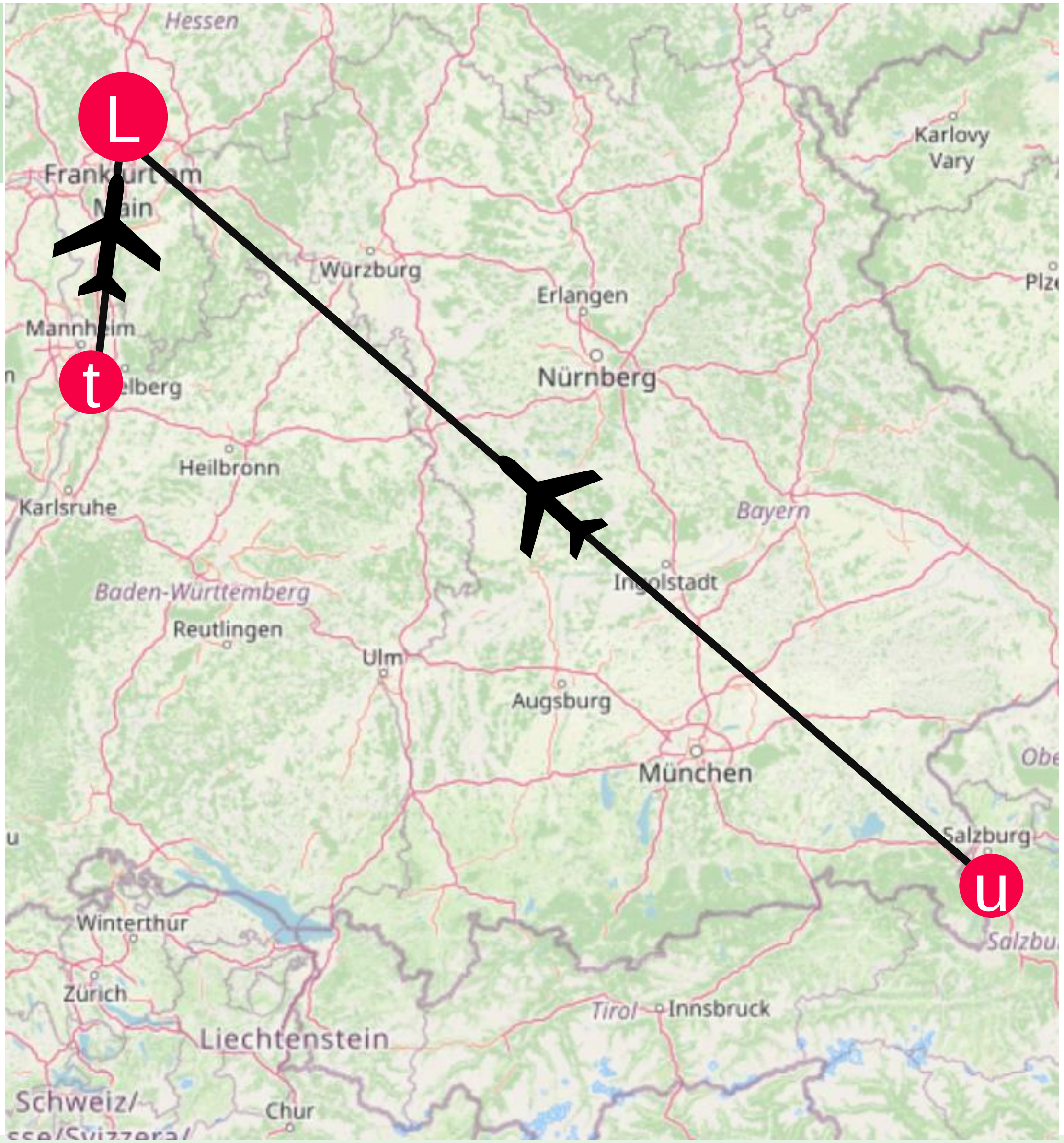


Landmark heuristic

Landmark heuristic with landmark L:

$$h\,(u,t) = |d(u,L) - d(t,L)| \le \; d(u,t)$$

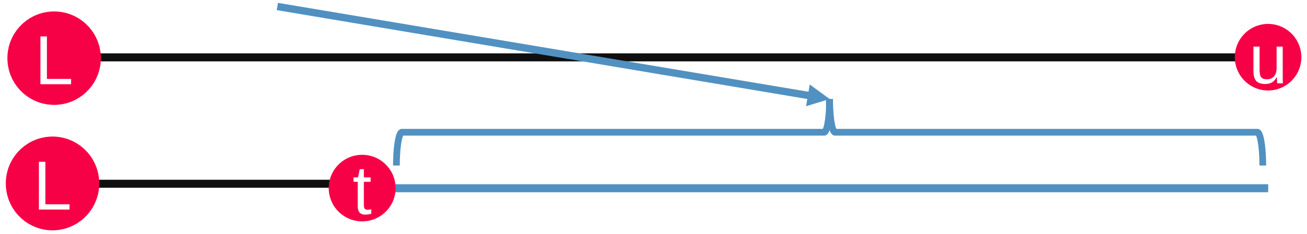


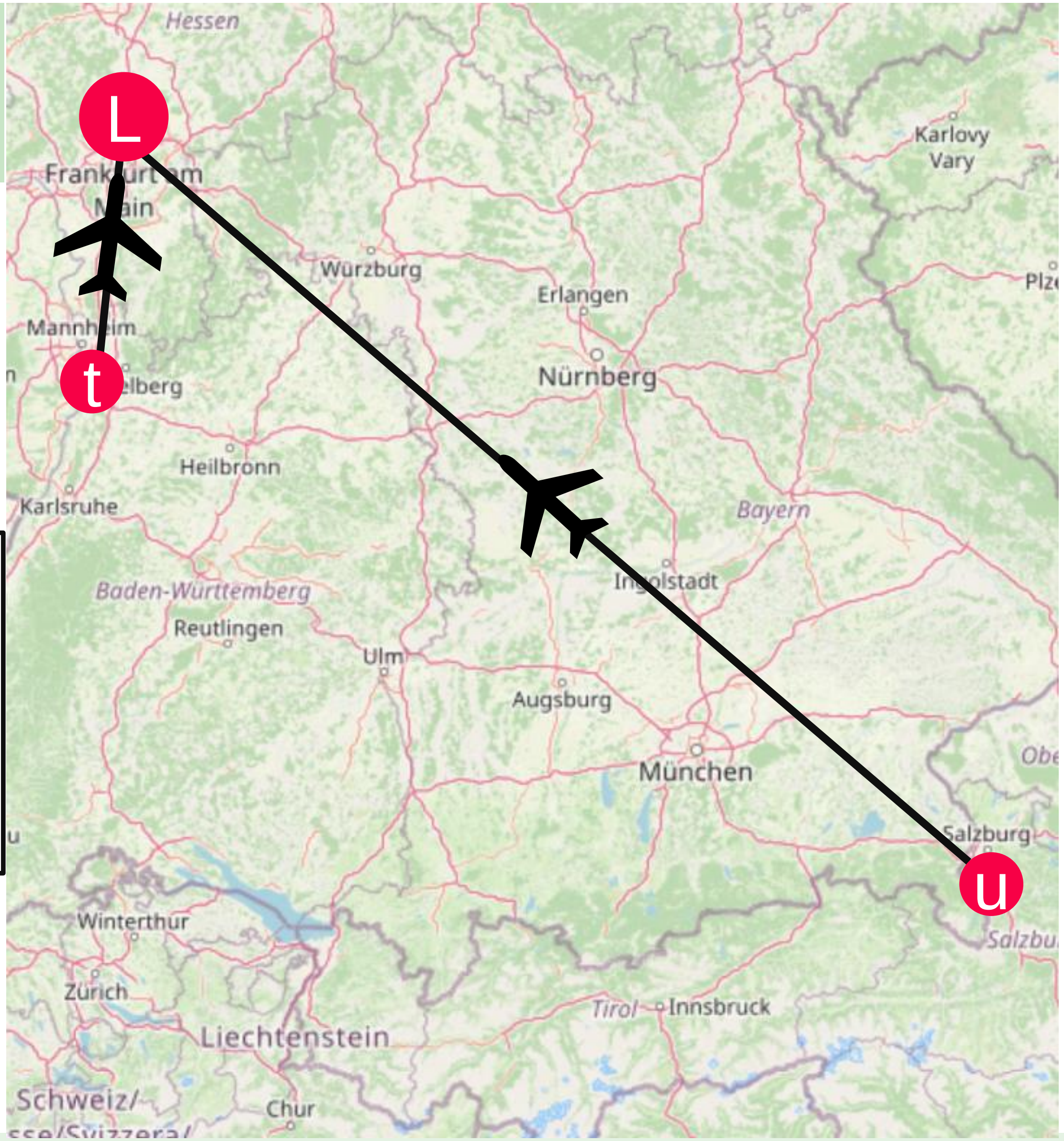


Landmark heuristic

Landmark heuristic with landmark L:

$$h\,(u,t) = |d(u,L) - d(t,L)| \leq \; d(u,t)$$





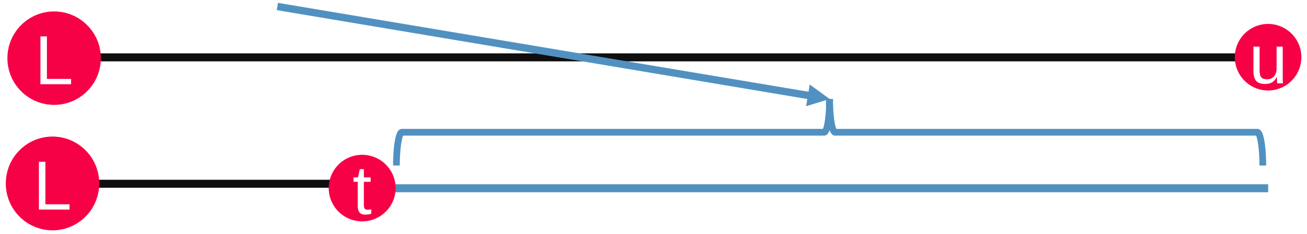
What does Google probably do?

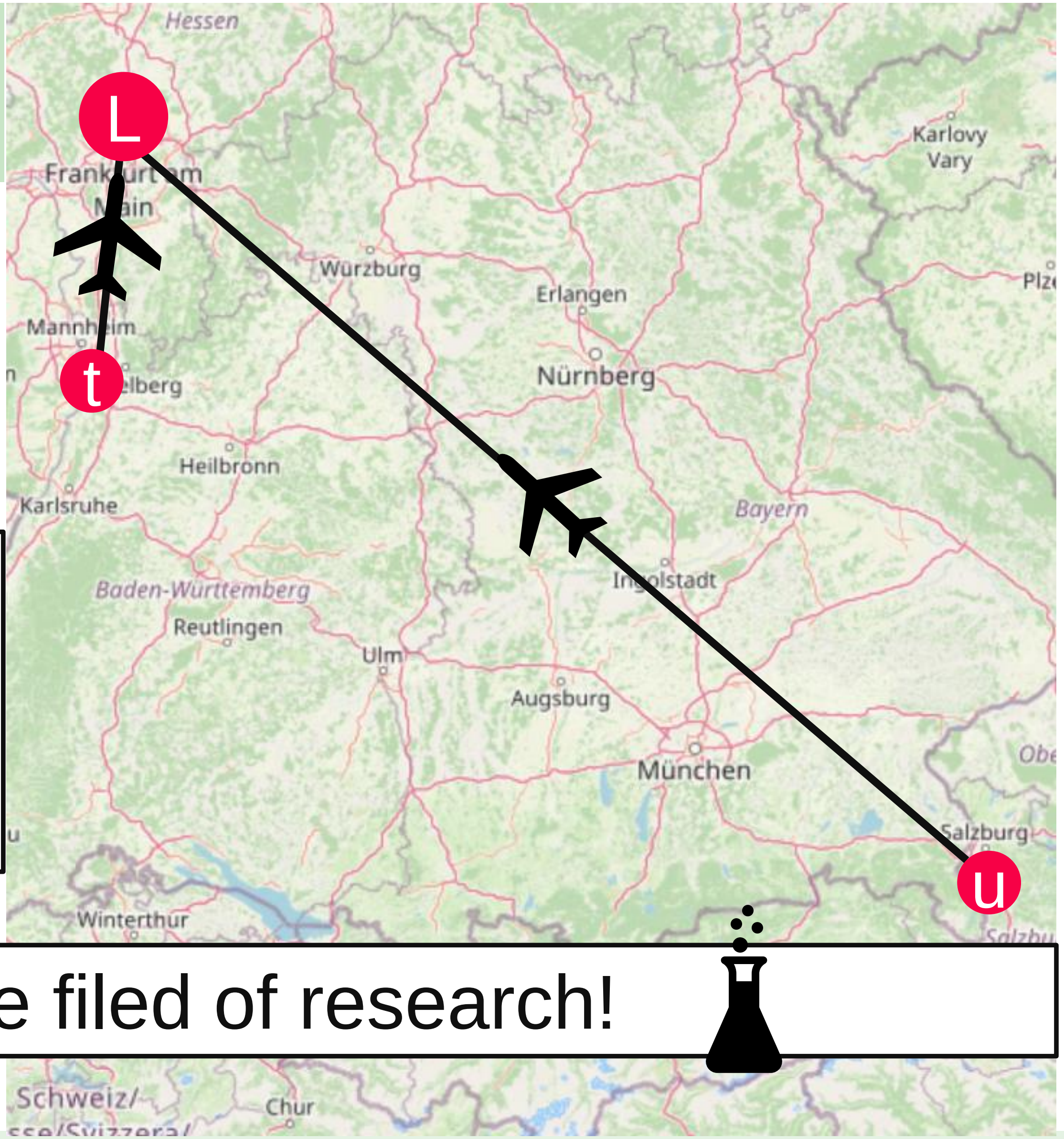
- all of these heuristics and more?
- One can use multiple heuristics and choose the max

Landmark heuristic

Landmark heuristic with landmark L:

$$h\,(u,t) = |d(u,L) - d(t,L)| \leq \; d(u,t)$$





What does Google probably do?

- all of these heuristics and more?
- One can use multiple heuristics and choose the max

Finding good heuristics is an active filed of research!

When to use which algorithm?

One source and One Destination

- Use A* Search Algorithm (For Unweighted as well as Weighted Graphs)

The Heuristic matters!

When to use which algorithm?

One source and One Destination

- Use A* Search Algorithm (For Unweighted as well as Weighted Graphs)

The Heuristic matters!

One Source, All Destination

- BFS (For Unweighted Graphs)
- Use Dijkstra (For Weighted Graphs without negative weights)
- Use Bellman Ford (For Weighted Graphs with negative weights)

When to use which algorithm?

One source and One Destination

- Use A* Search Algorithm (For Unweighted as well as Weighted Graphs)

The Heuristic matters!

One Source, All Destination

- BFS (For Unweighted Graphs)
- Use Dijkstra (For Weighted Graphs without negative weights)
- Use Bellman Ford (For Weighted Graphs with negative weights)

Between every pair of nodes

- Floyd-Warshall
- Johnson’s Algorithm (not discussed)

What else?

Link to animated version of A*: https://algorithms.discrete.ma.tum.de/

Thank you