A* and Informed Search

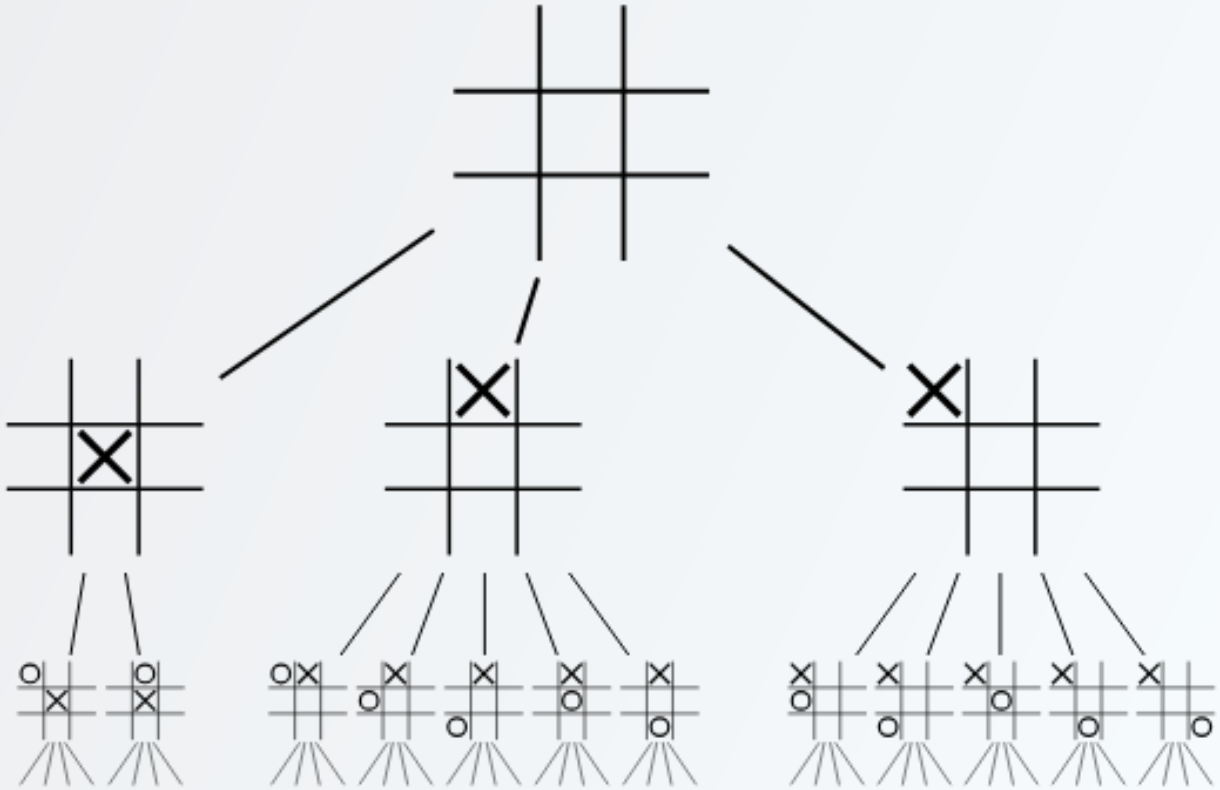
Search and Graphs

- Mostly, you've seen graphs as data structures that exist independently of the search
  - A network of social connections
  - A finite state machine
  - Points on a grid
- Sometimes we want to search for solutions to a problem
  - "Find a solution if it exists" instead of "find a path from s to t"

Generating The Graph As You Search

We can use graphs to represent the states of problems (nodes) accessible by sequences of actions (edges)

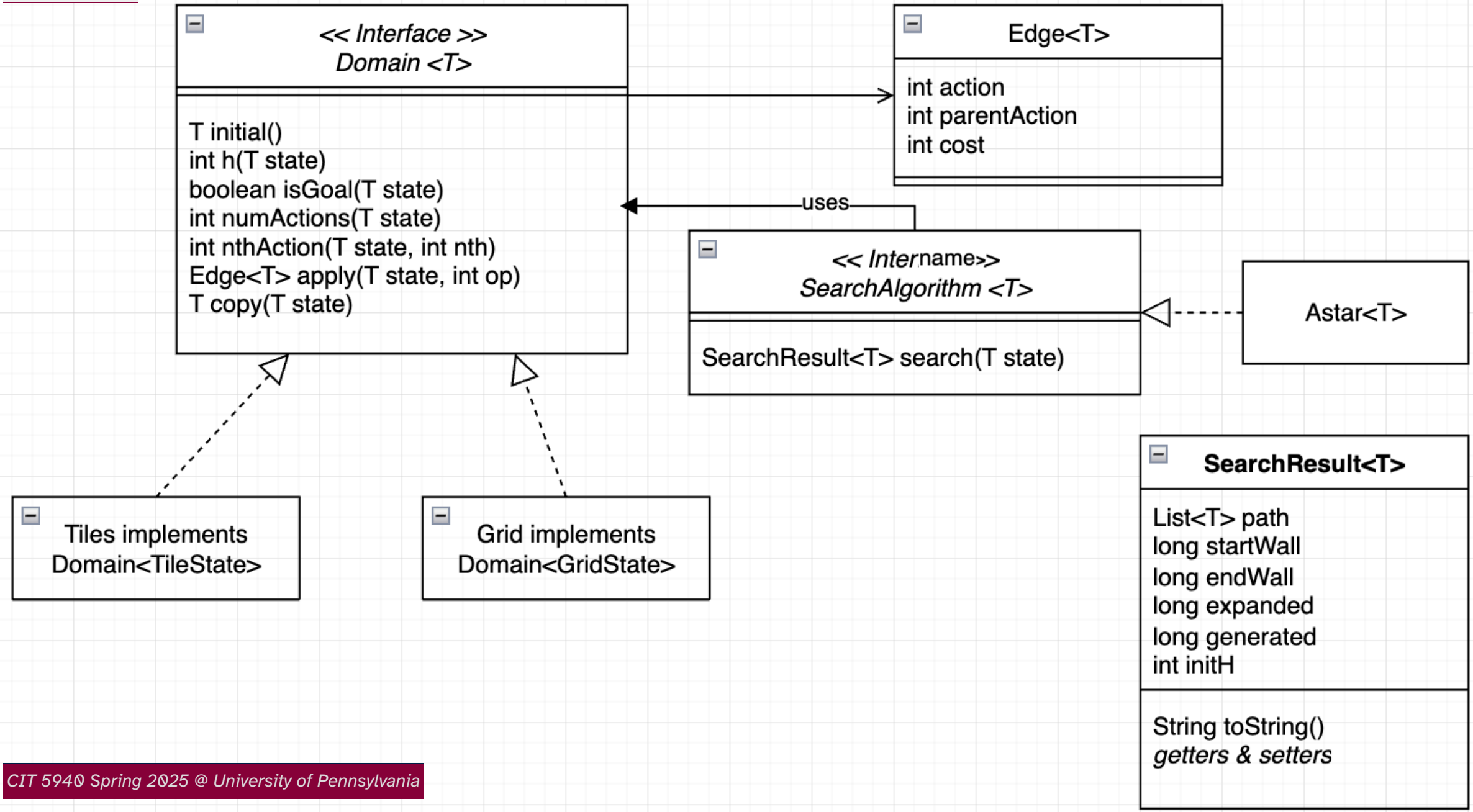
- Typically don't have the full picture of the graph from this POV
- can generate it as you go



Nodes and Edges, not a Graph

Representing the graph:

- We can't store the whole graph, but we can store nodes that we find
  - Nodes are objects storing `{parent, state, ...}` (??)
- The set of actions available at each state denote the edges from each node


<< Interface >>
Domain <T>
T initial()
int h(T state)
boolean isGoal(T state)
int numActions(T state)
int nthAction(T state, int nth)
Edge<T> apply(T state, int op)
T copy(T state)
Edge<T>
int action
int parentAction
int cost
uses
<< Internname>>
SearchAlgorithm <T>
SearchResult<T> search(T state)
Astar<T>
Tiles implements
Domain<TileState>
Grid implements
Domain<GridState>
SearchResult<T>
List<T> path
long startWall
long endWall
long expanded
long generated
int initH
String toString()
getters & setters

Nodes in Dijkstra's

Want a `PriorityQueue` of these things, so they should be *ordered by path cost*.

```
public class Node implements Comparable<Node>{
    private T state; // e.g. location in space or tic-tac-toe board
    private int g; // path cost, called "g" by convention
    private Action parentAction; // action taken to get here from parent
    private Node parent; // the parent itself

    public int compareTo(Node other) { return this.g - other.g; }
}
```

Nodes are objects storing `{state, g, parentOperation, parent}`

Dijkstra's

Setup...

```
start = new Node(startState, 0, null, -1)
frontier = new PriorityQueue<Node>();
frontier.add(start);
costSoFar = new HashMap<Node, Integer>();
costSoFar.put(start, 0);
```

Dijkstra's

Core Loop...

```
while (!frontier.isEmpty()) {
    current = frontier.pop();
    if (current.isGoal()) { return current; }
    for (int i = 0; i < current.numActions(); i++) {
        Edge<T> action = current.nthAction(i);
        Node next = action.resolve();
        int newCost = costSoFar.get(current) + action.cost;
        if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
            costSoFar.put(next, newCost);
            frontier.push(next, new_cost);
        }
    }
}
```

Nodes and Edges, not a Graph

- Need domain knowledge to help generate the graph as you traverse it
- Essential Qs to answer:
  - Where does the search start from?
  - What does a solution look like?
  - What actions are available from each state?
  - What happens if I perform this action on this state?

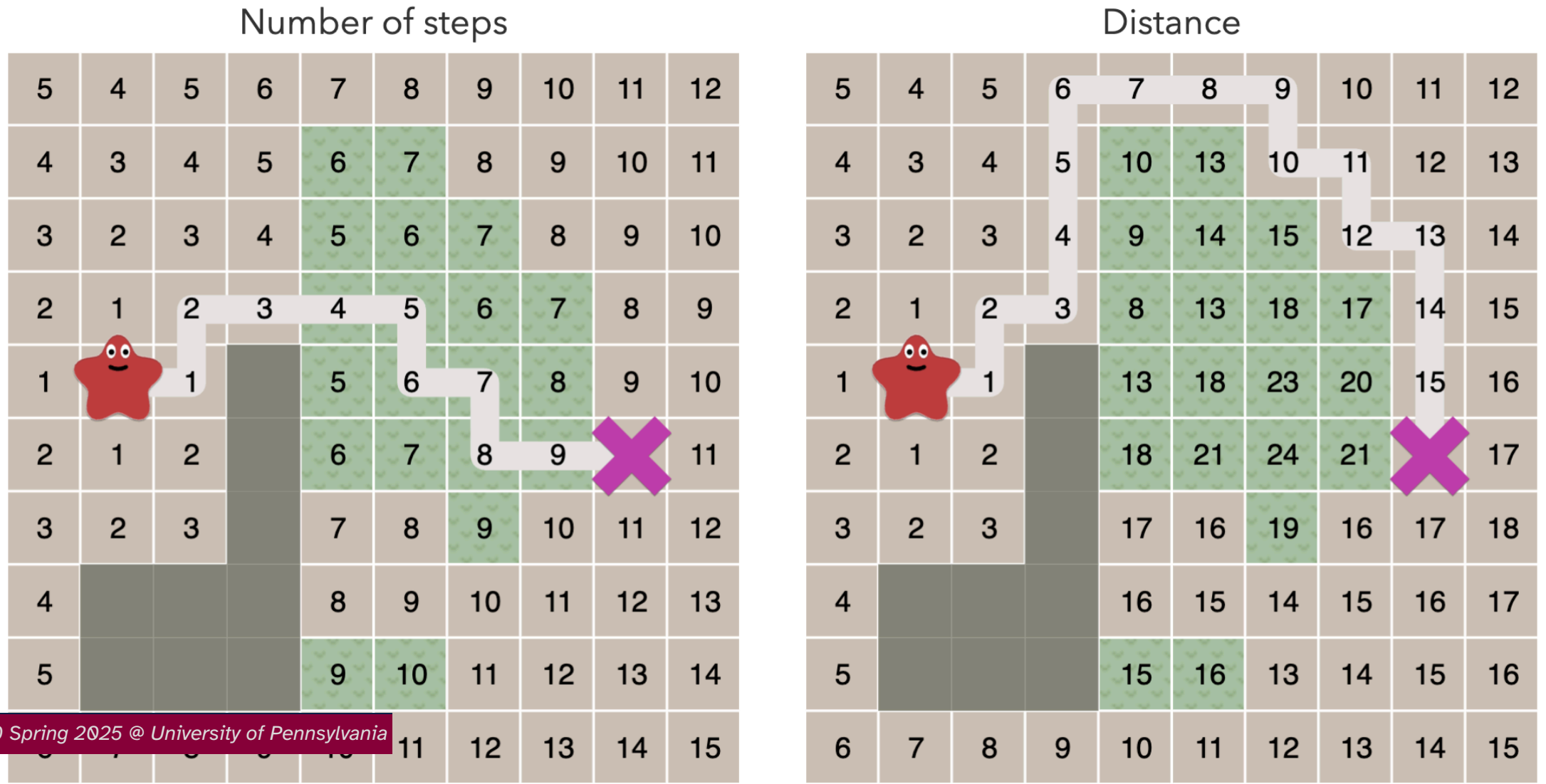
Nodes and Edges

```
private interface INode<T> {
  // how far do we seem to be from a goal?
  public int f();
  // how far have we come from the start?
  public int g();
  // how did we get here?
  public int parent();
  // where are we now?
  public Domain<T> state();
}
```

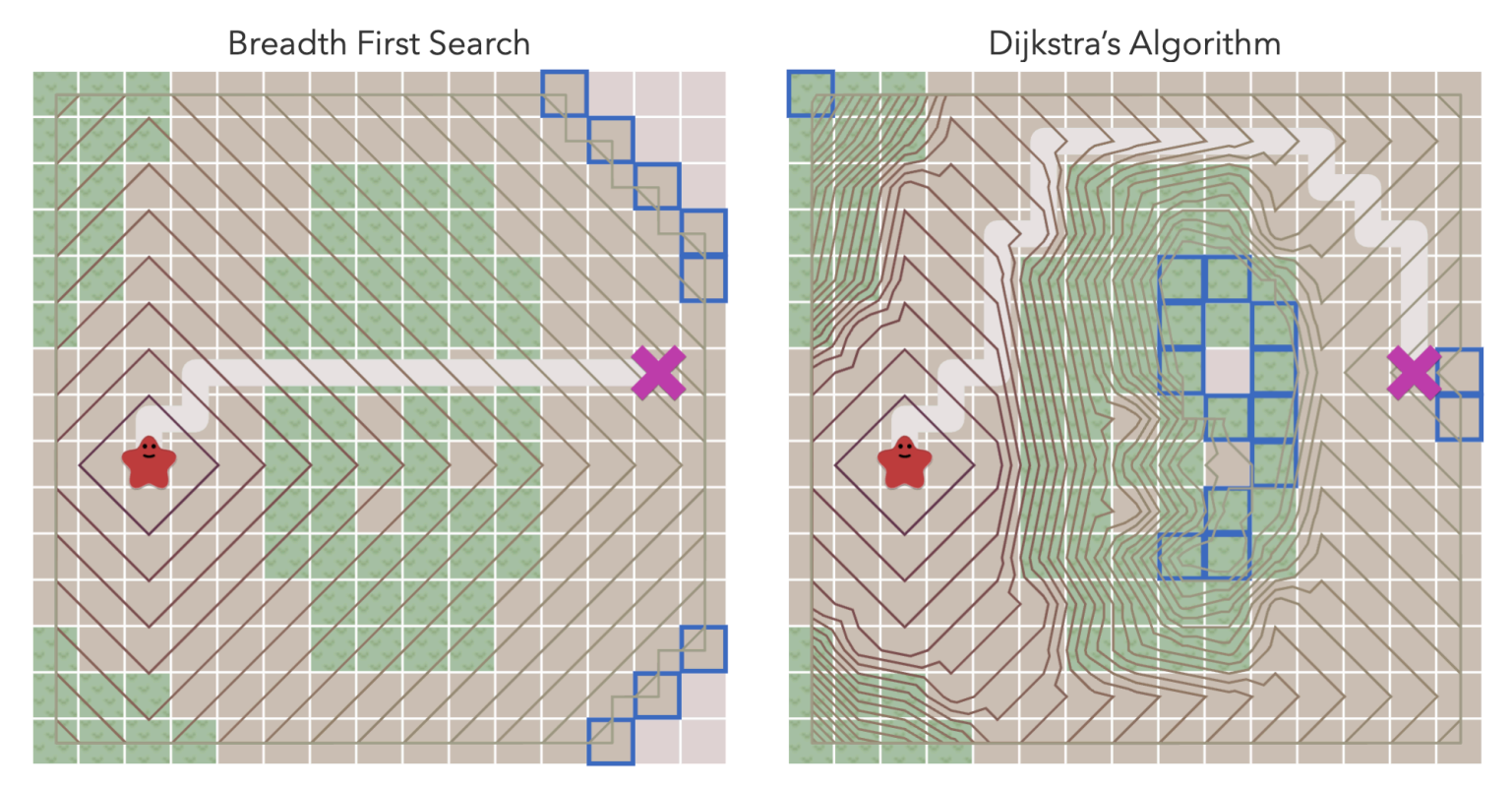
```
private interface IEdge<T> {
  // what is the cost of taking this action?
  public int getCost();
  // what is this action?
  public int getAction();
  // what is the action that we took to get here?
  public int getParentAction();
}
```

No Graph.java needed!

Grid Navigation Setting

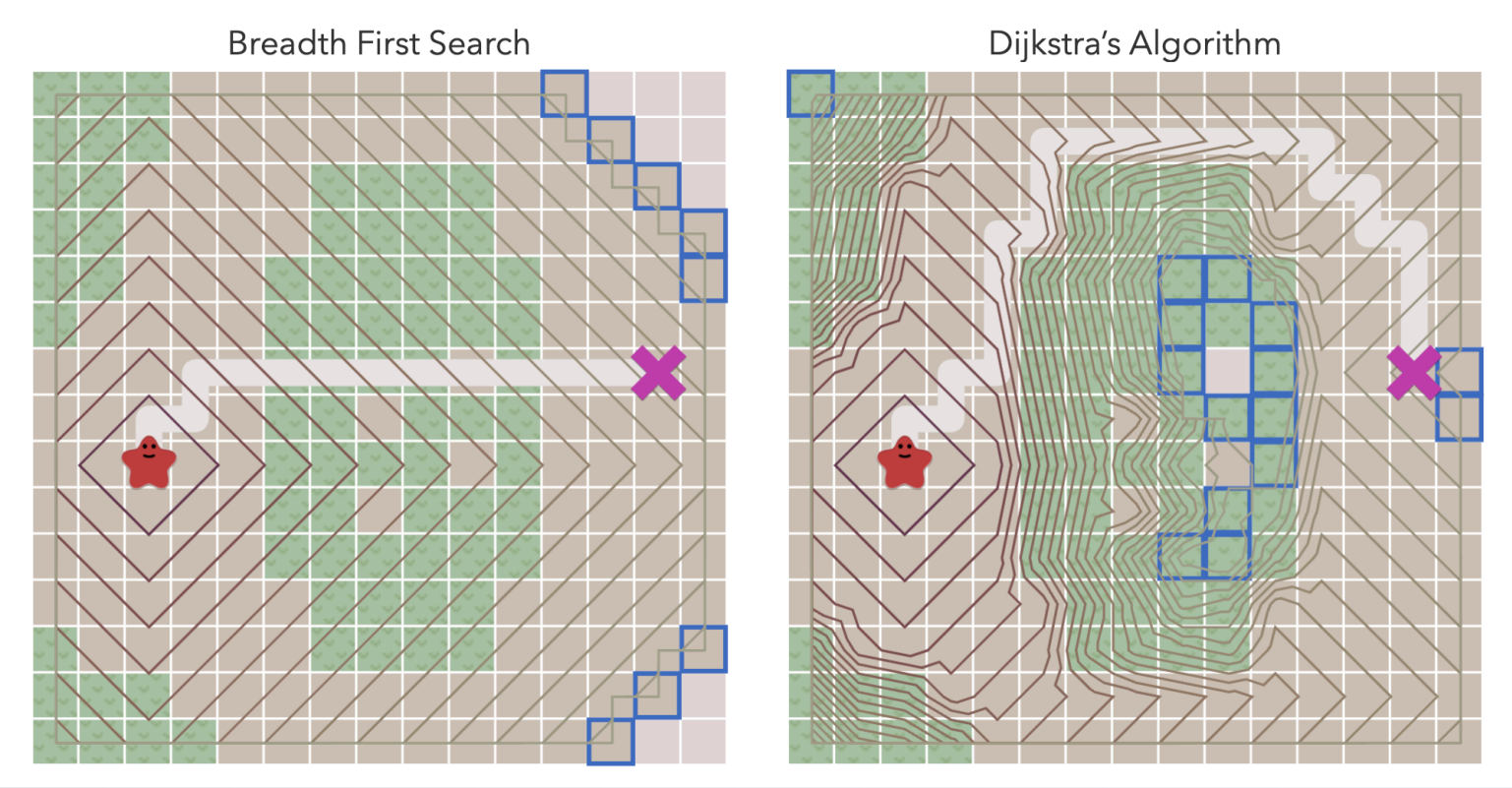


Grid Navigation Setting



Finds a better path than an unweighted BFS would...

Grid Navigation Setting

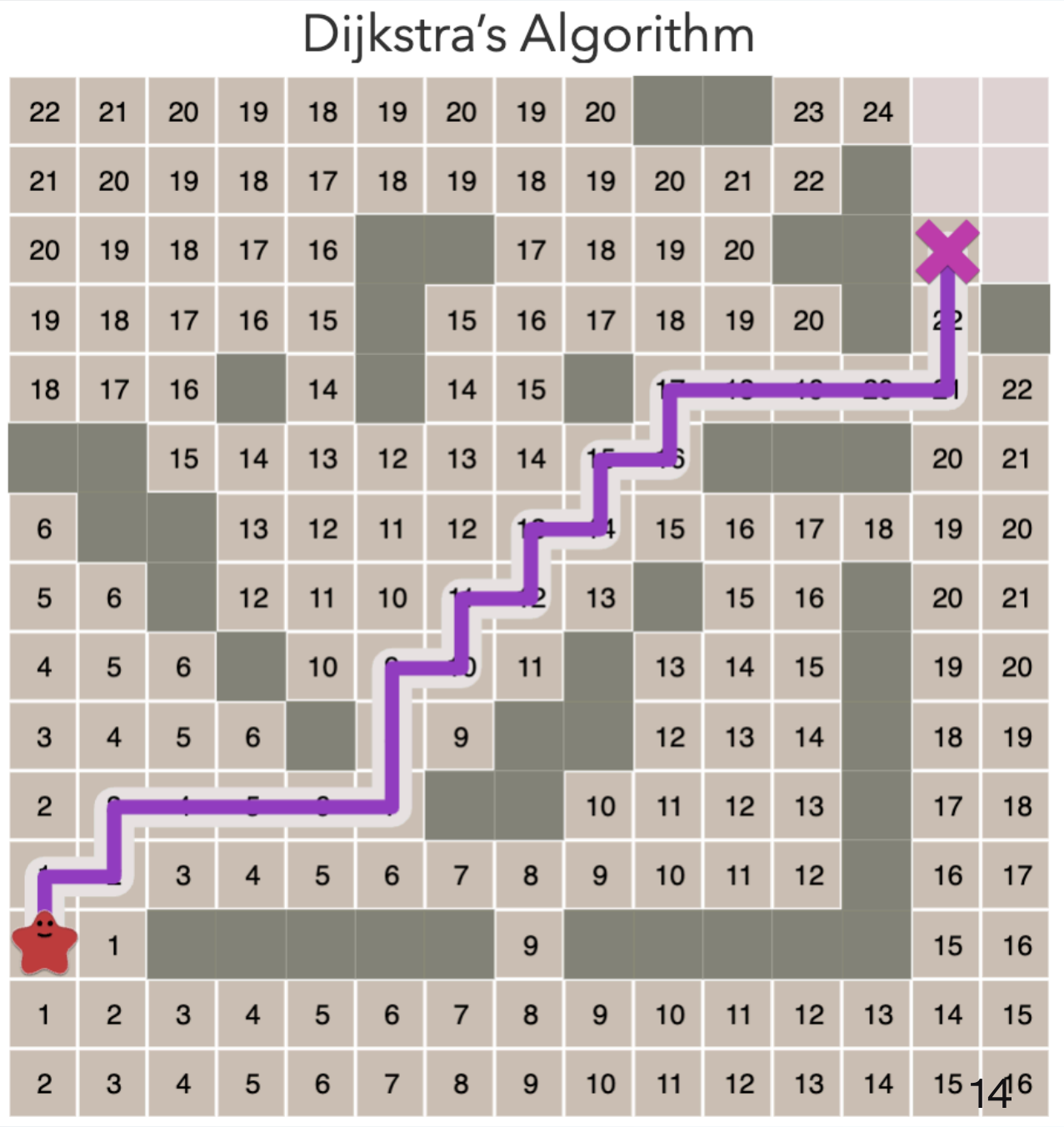


...but look at all that explored space that wasn't useful.

Dijkstra's: Slow and Steady

Dijkstra's definitionally explores all paths of cost c before exploring any paths of cost $c + 1$.

- Lots of "unviable" paths explored
- Doesn't incorporate any information about what would make a path promising.



Idea: Be Greedy

Greedy algorithms make locally optimal decisions in hopes that they will lead to globally optimal solutions.

What if we explored nodes in order of how close they seem to the goal, disregarding *actual board information* and *the cost of getting to that node in the first place*?

Nodes in Greedy Best First Search

Want a `PriorityQueue` of these things, so they should be *ordered by apparent distance to goal*.

```
public class Node implements Comparable<Node>{
    private T state; // e.g. location in space or tic-tac-toe board
    private int h; // estimate of cost from current location to goal
    private Action parentAction; // action taken to get here from parent
    private Node parent; // the parent itself

    public int compareTo(Node other) { return this.h - other.h; }
}
```

Nodes are objects storing `{state, h, parentOperation, parent}`

h is for "Heuristic"

A **heuristic** (from the Greek "Eureka!") is an educated guess or an estimate generated through insight.

- In grid navigation with obstacles, a heuristic for distance to goal would be the euclidean distance to the goal
- For a Rubik's Cube, a heuristic for better states might be the number of mismatched color squares

Greedy Best First Search

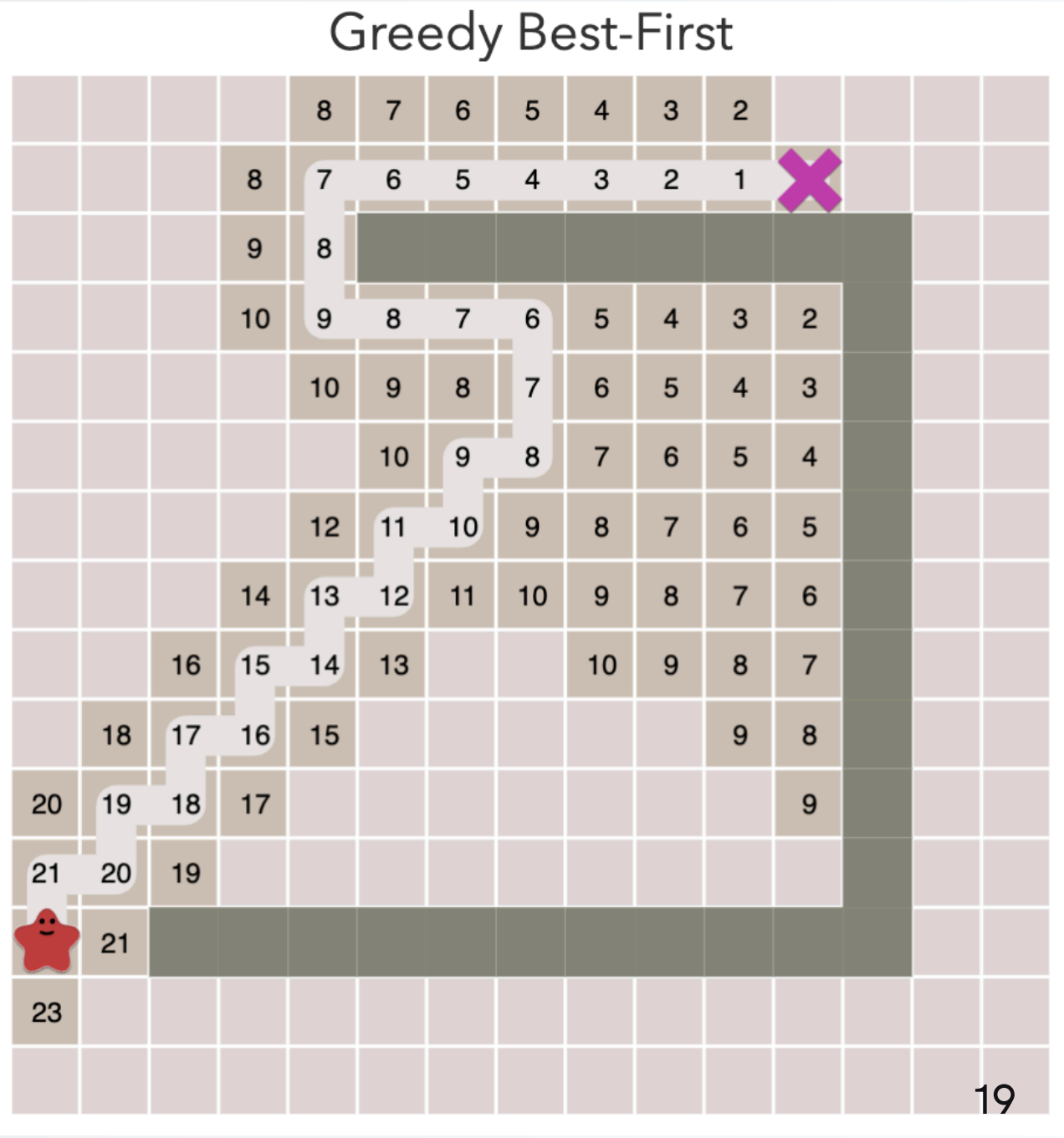
Core Loop...

```
while (!frontier.isEmpty()) {
    current = frontier.pop();
    if (current.isGoal()) { return current; }
    for (int i = 0; i < current.numActions(); i++) {
        Edge<T> action = current.nthAction(i);
        Node next = action.resolve();
        int newCost = distanceToGoal(next); // only change from Dijkstra!
        if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
            costSoFar.put(next, newCost);
            frontier.push(next, new_cost);
        }
    }
}
```

Greedy Best-First Search

Explores far fewer nodes
than Dijkstra's!

...but also gives a suboptimal path...



A: The Big Idea*

Dijkstra's is helpful because it always returns a shortest path.

- Get this guarantee by exploring in increasing order of path cost

Greedy Best First Search is helpful because it spends less time exploring useless paths.

- Gets this speedup by preferring nodes that appear closer to the goal

A* could improve on both by doing both!

Nodes in A*

Want a `PriorityQueue` of these things, so they should be *ordered by apparent distance to goal*.

```
public class Node implements Comparable<Node>{
    private T state; // e.g. location in space or tic-tac-toe board
    private int h; // estimate of cost from current location to goal
    private int g; // path cost to reach this node
    private int f; // g + h
    private Action parentAction; // action taken to get here from parent
    private Node parent; // the parent itself

    public int compareTo(Node other) { return this.f - other.f; }
}
```

Nodes are objects storing `{state, f, parentOperation, parent}`

Key Idea:

As long as a heuristic **never overestimates the distance to the goal**, a `PriorityQueue` ordered by `f = g + h` will allow us to find an optimal path.

In practice, we will explore fewer fruitless paths.

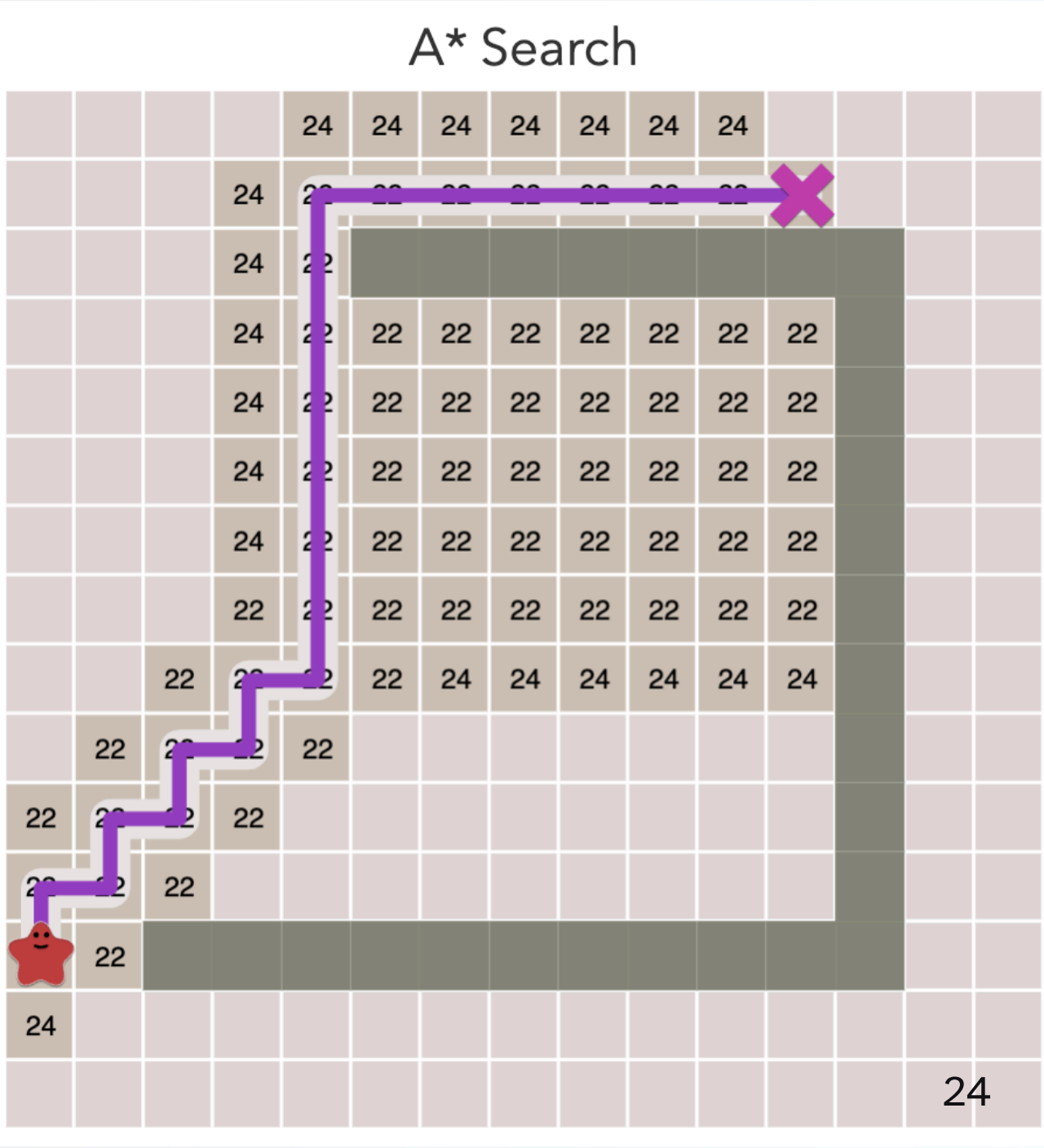
A*

Core Loop...

```
while (!frontier.isEmpty()) {
    current = frontier.pop();
    if (current.isGoal()) { return current; }
    for (int i = 0; i < current.numActions(); i++) {
        Edge<T> action = current.nthAction(i);
        Node next = action.resolve();
        int newCost = costSoFar.get(current) + action.cost + distanceToGoal(next);
        if (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
            costSoFar.put(next, newCost);
            frontier.push(next, new_cost);
        }
    }
}
```

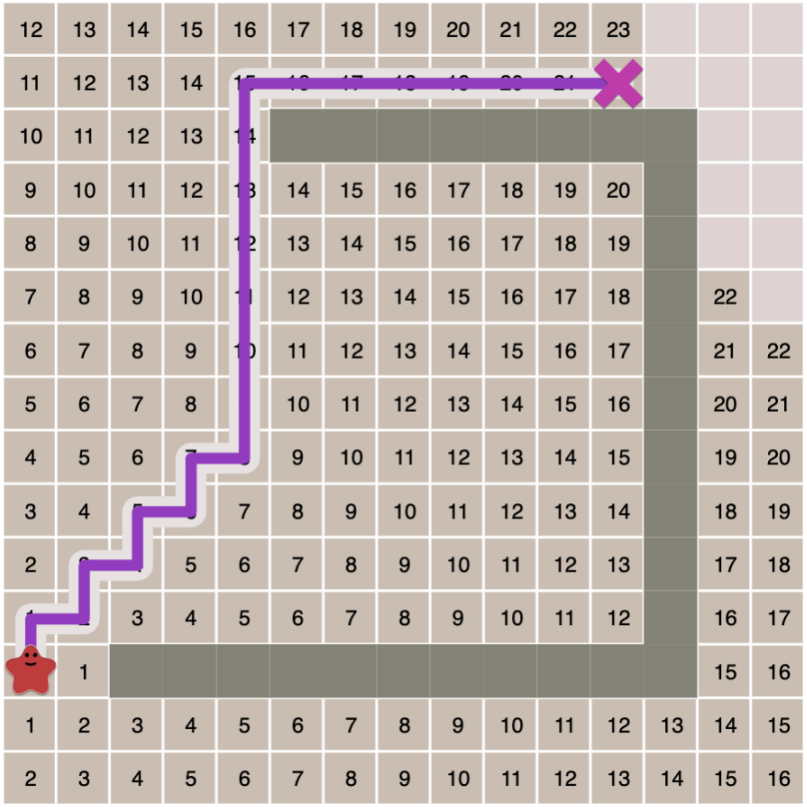
*A**

Explore fewer nodes than Dijkstra's and still find an optimal path.

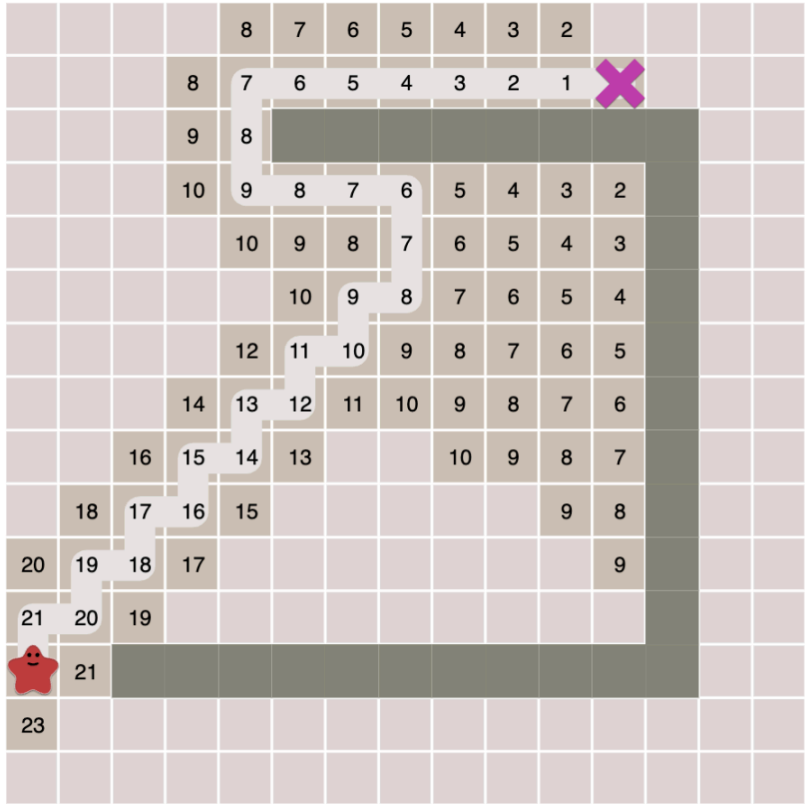


A* SEARCH

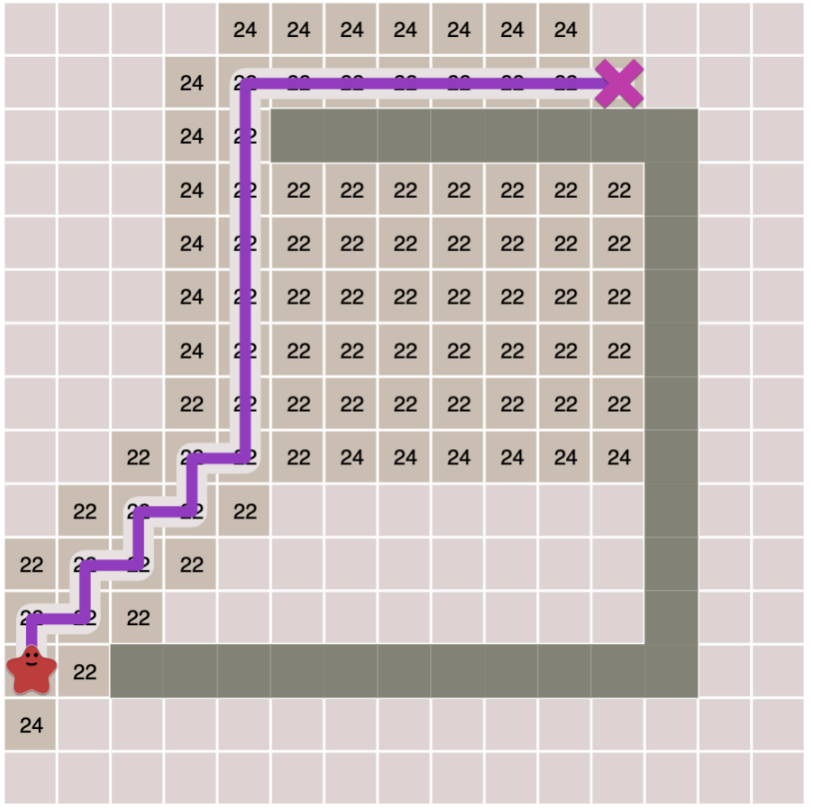
Dijkstra's Algorithm



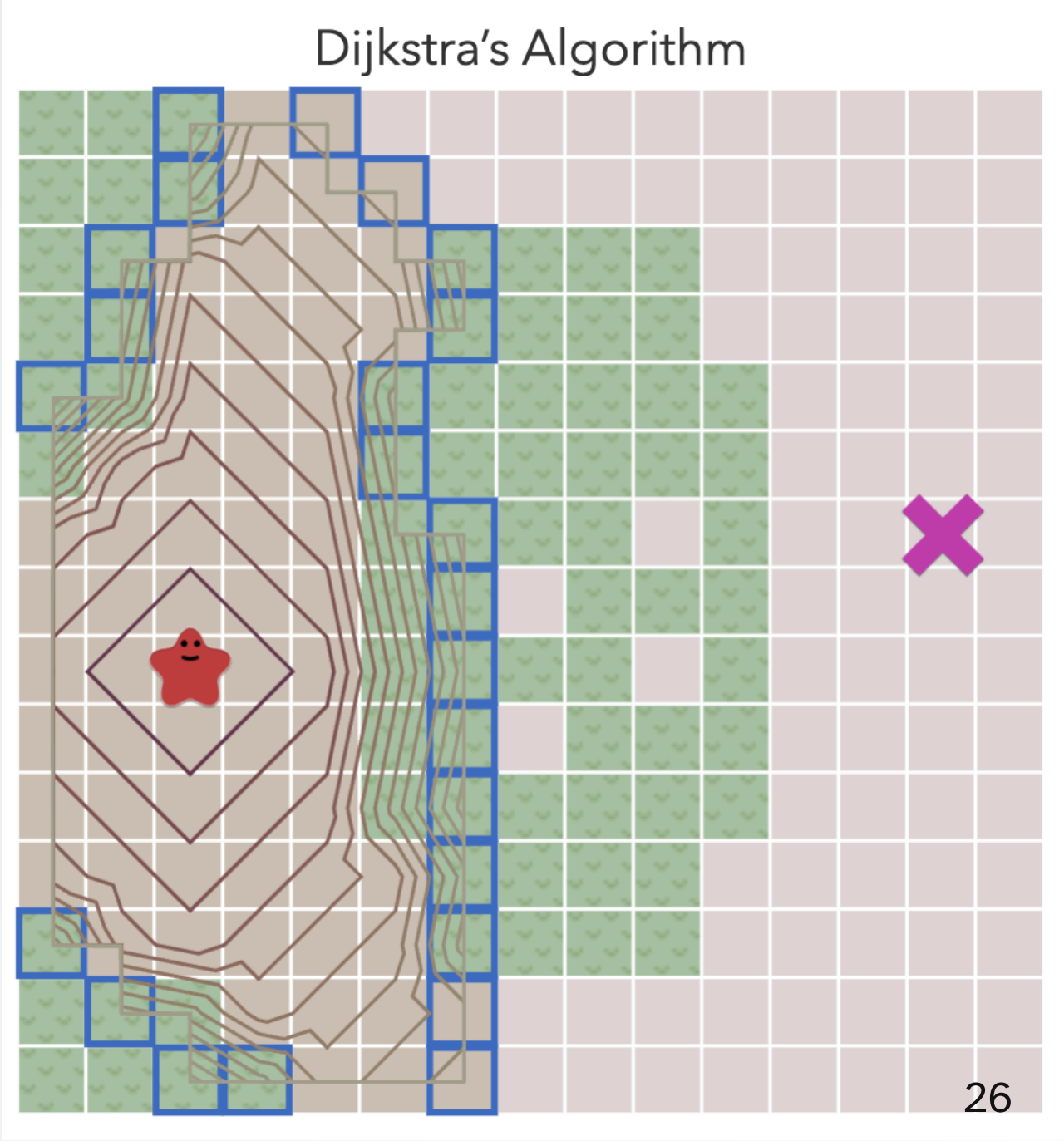
Greedy Best-First



A* Search



Prefer to explore along contours of slowly increasing path costs that are still oriented towards the goal.



IBM published an amazing overview of A in Java here.*

[Note: This slide contained a screenshot of an IBM article titled "Faster problem solving in Java with heuristic search" by Matthew Hatem, Ethan Burns, Wheeler Ruml, published July 16, 2013.]

Link: IBM A* in Java article