Pathfinding

In Clojure

What is pathfinding?

- Find the shortest[1] path between two points
- More generally known as the shortest path problem
- Typically applied to graphs
- Can be applied to abstract problems

eg, finding a “path” through a graph of states to reach a desired state

1. Shortest or “cheapest”, for example, preferring large roads over small

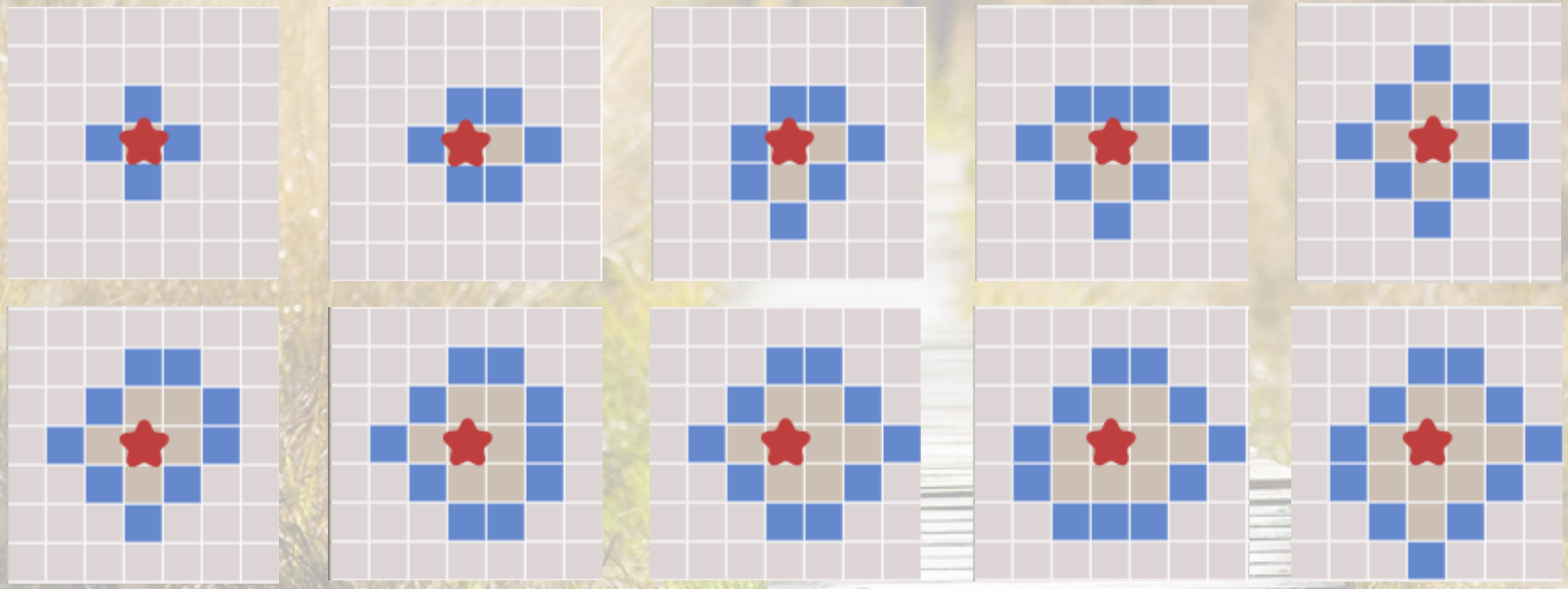
Algorithms

- Breadth-first Search
- Dijkstra's Shortest Path
- A*

- Very similar algorithms, build on each other

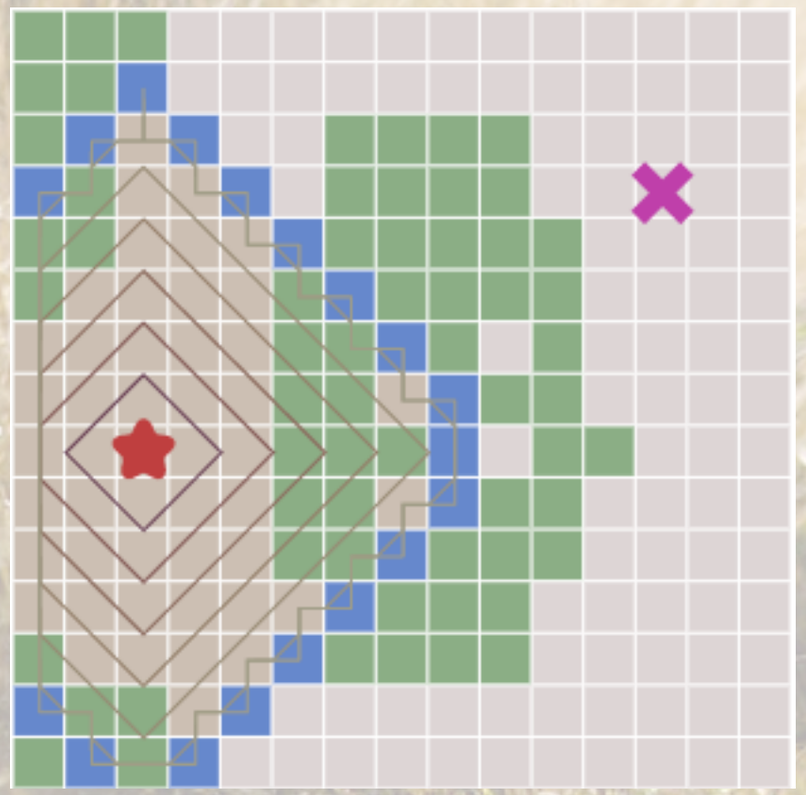
Breadth-first Search

- Explores each neighbouring “node” before exploring deeper

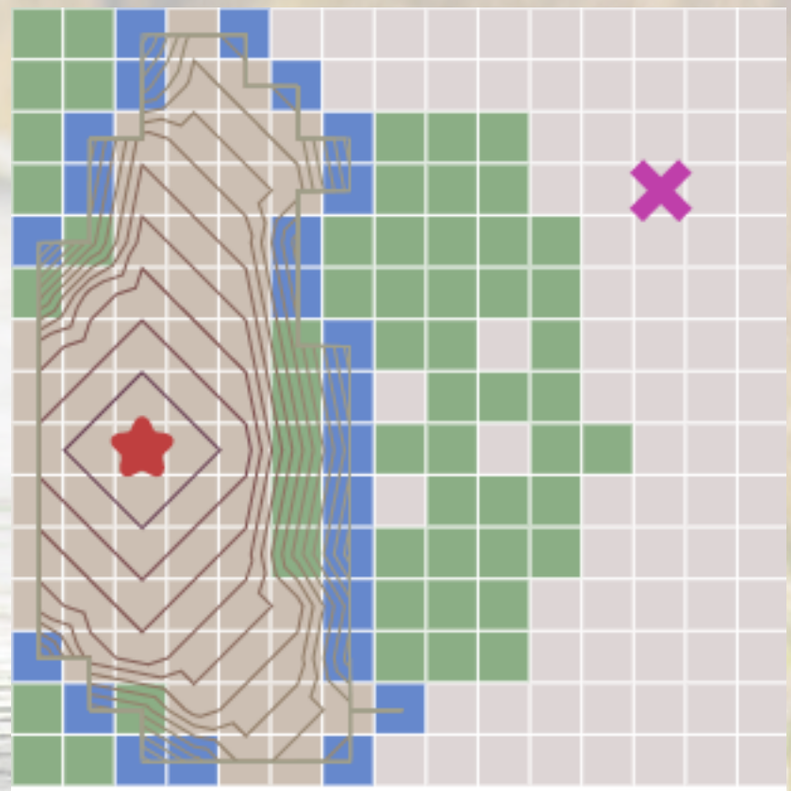


Dijkstra’s Shortest Path

- Similar to BFS, but explores “cheapest” neighbours first



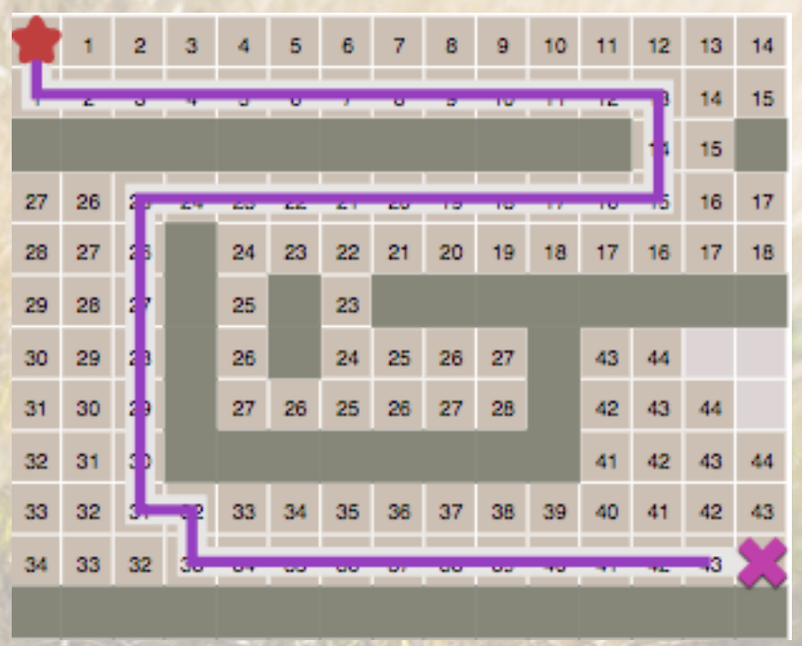
Breadth First



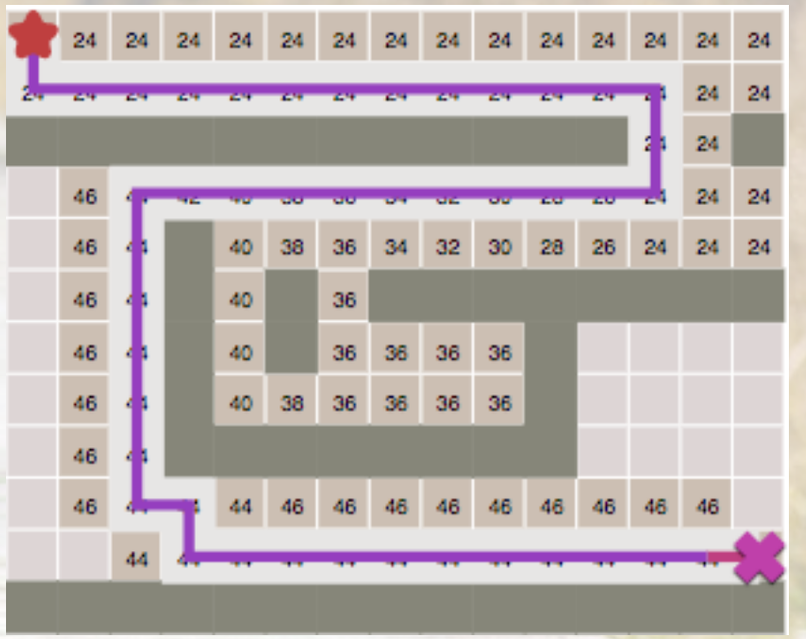
Dijkstra’s algorithm

A* Search

- Like Dijkstra's but prefers neighbours that get closer to goal



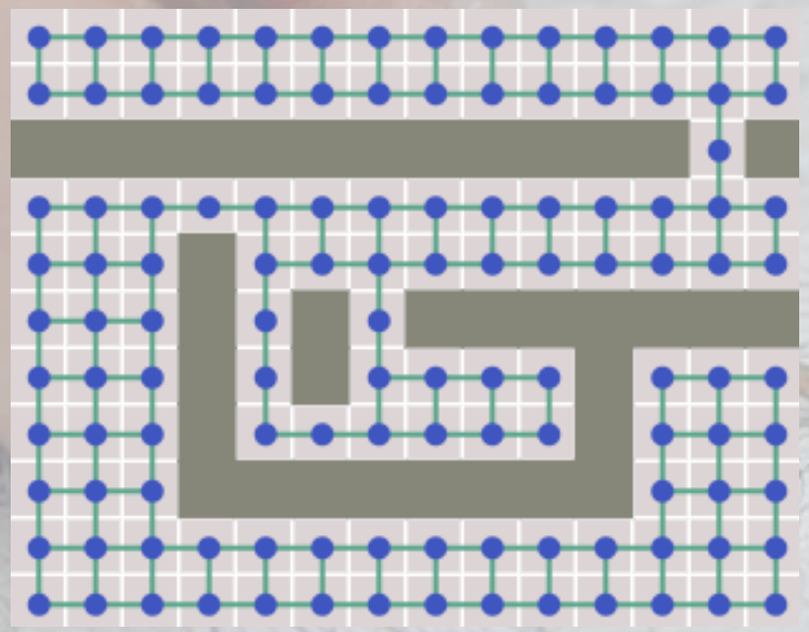
Dijkstra's algorithm



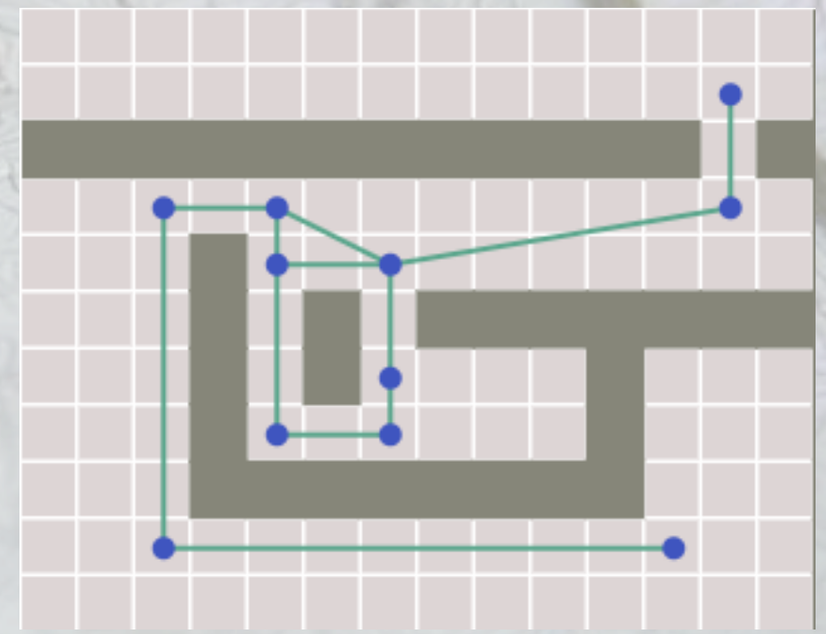
A* Search

Map Representation

- Map is a graph of connected nodes (the neighbours)
- Grid is converted into graph structure prior to running search



Waypoints would be more efficient…



Implementation

- Keep a queue of “frontier” nodes (unexplored neighbours)
- For each frontier node in turn, explore
- Explore = if goal then done else add all neighbours to frontier
- Keep a “came-from” map of current-node → previous-node
- “came-from” map used to reconstruct path at the end

This is essentially BFS! Dijkstra & A* simply modify the order in which the frontier nodes are explored

Challenges in Clojure

- Clojure likes things to be immutable
- Frontier queue is modified as its being consumed
- Cannot simply map/reduce over a frontier node sequence

Solutions

- Use loop/recur
    - Manually loop through frontier, allowing us to add extra nodes when we recur, if needed
- Use iterate
    - Allows us to generate a new state (frontier queue etc) given the previous state

      f(previous) = successor

Loop/recur

```
(loop [frontier [start]
       came-from {start :start}]
  (let [current (first frontier)]
    (if (not= current goal)
      (let [neighbours (get-neighbours current)]
        (recur (concat (next frontier) neighbours)
               (into came-from
                 (for [neighbour neighbours]
                   [neighbour current]))))
      came-from)))
```

get-neighbours

- Gets the current node from the global graph
- Gets all neighbour node keys
- Removes neighbours already contained in comes-from map

iterate

```
(take 5 (iterate (fn [x] (* x 2)) 1))
```

```
((fn [x] (* x 2)) 1))        => 1
((fn [x] (* x 2)) 2))        => 2
((fn [x] (* x 2)) 4))        => 4
((fn [x] (* x 2)) 8))        => 8
((fn [x] (* x 2)) 16))       => 16
```

=

```
(1 2 4 8 16)
```

iterate

```
(->> {:frontier [start]
      :came-from {start :start}}
  (iterate
    (fn [{:keys [frontier came-from]}]
      (let [current (first frontier)]
        (when (not= current goal)
          (let [neighbours (get-neighbours current)]
            {:frontier  (concat (next frontier)
                                neighbours)
             :came-from (into came-from
                              (for [neighbour neighbours]
                                [neighbour current]))}))))))
  (take-while identity)
  last
  :came-from)
```

Dijkstra's Algorithm

- Takes "cost" of a neighbour into account and explores lowest cost first

```
(loop [frontier (priority-queue start)
       came-from {start :start}
       cost-so-far {start 0}]
  (let [current (first (peek frontier))]
    (if (not= current goal)
      (let [neighbours (get-neighbours current)
            costs (get-costs current neighbours
                             cost-so-far)]
        (recur (into (pop frontier)
                     (for [[node-id cost] costs]
                       [node-id cost]))
               …
               (into cost-so-far costs)
```

get-costs

```
(defn get-costs
  [current neighbours cost-so-far]
  (->>
    (for [neighbour neighbours]
      (let [new-cost (+ (get cost-so-far current)
                        (get-cost current neighbour))]
        (when (or (not (contains? cost-so-far
                                  neighbour))
                  (< new-cost (get cost-so-far
                                   neighbour)))
          [neighbour new-cost])))))
```

A* Search

- One-line change to Dijkstra's algorithm! (…plus a heuristic fn)

```
…
(recur
  (into (pop frontier)
        (for [[node-id cost] costs]
          [node-id (+ cost (heuristic goal node-id))])))
…

(defn heuristic
  "Manhattan distance on a square grid"
  [goal current]
  (let [[ax ay] goal
        [bx by] current]
    (+ (Math/abs (- ax bx))
       (Math/abs (- ay by)))))
```

Why does it work?

- Dijkstra calculates the total cost from start to neighbours
- Frontier queue is a priority queue
- Calculated cost used as priority in frontier queue
- Lowest cost items are taken from priority queue first
- A* adds the heuristic (in this case, manhattan distance) to the cost, forcing the queue to prefer “closer” neighbours

Resources

- Amit Patel/Red Blob Games

  https://www.redblobgames.com/pathfinding/a-star/introduction.html

- Sample code for this talk (uses “iterate” not “loop/recur”)

  https://github.com/danielytics/pathfinding-talk

Questions?