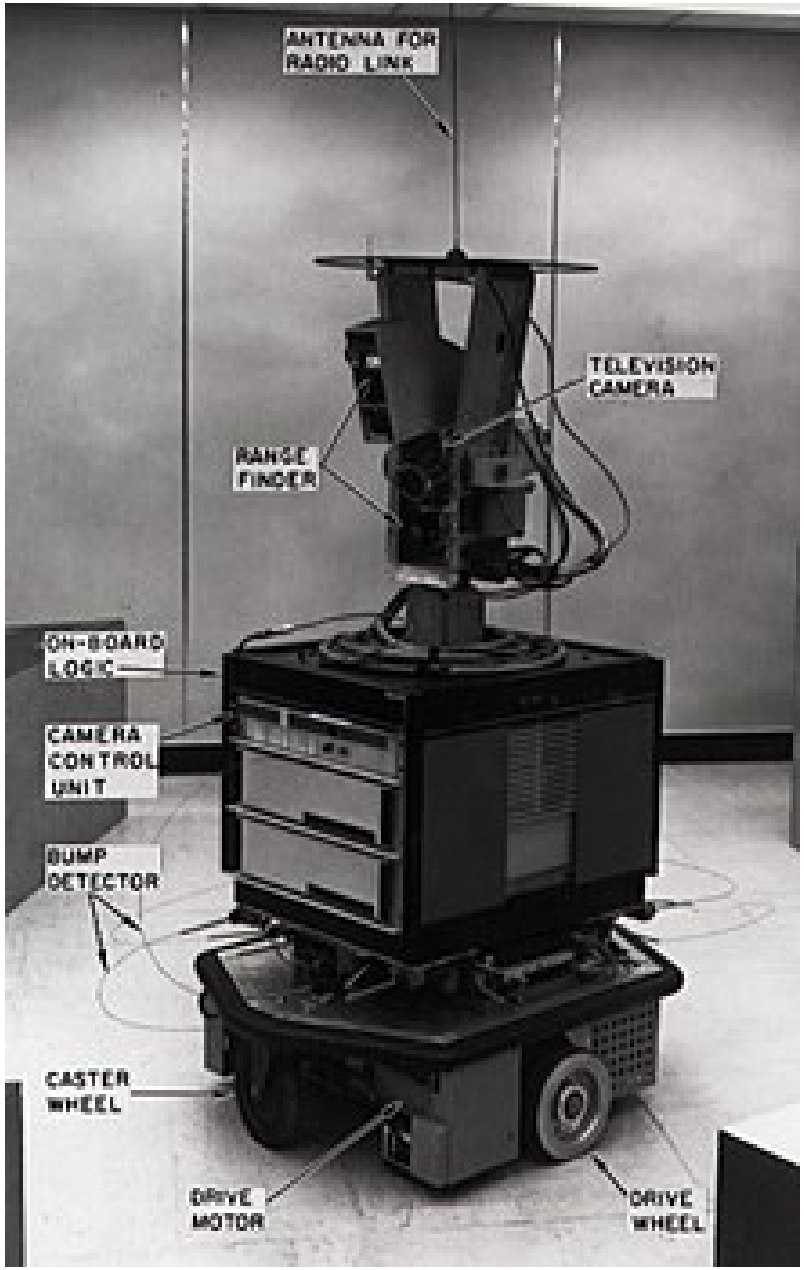
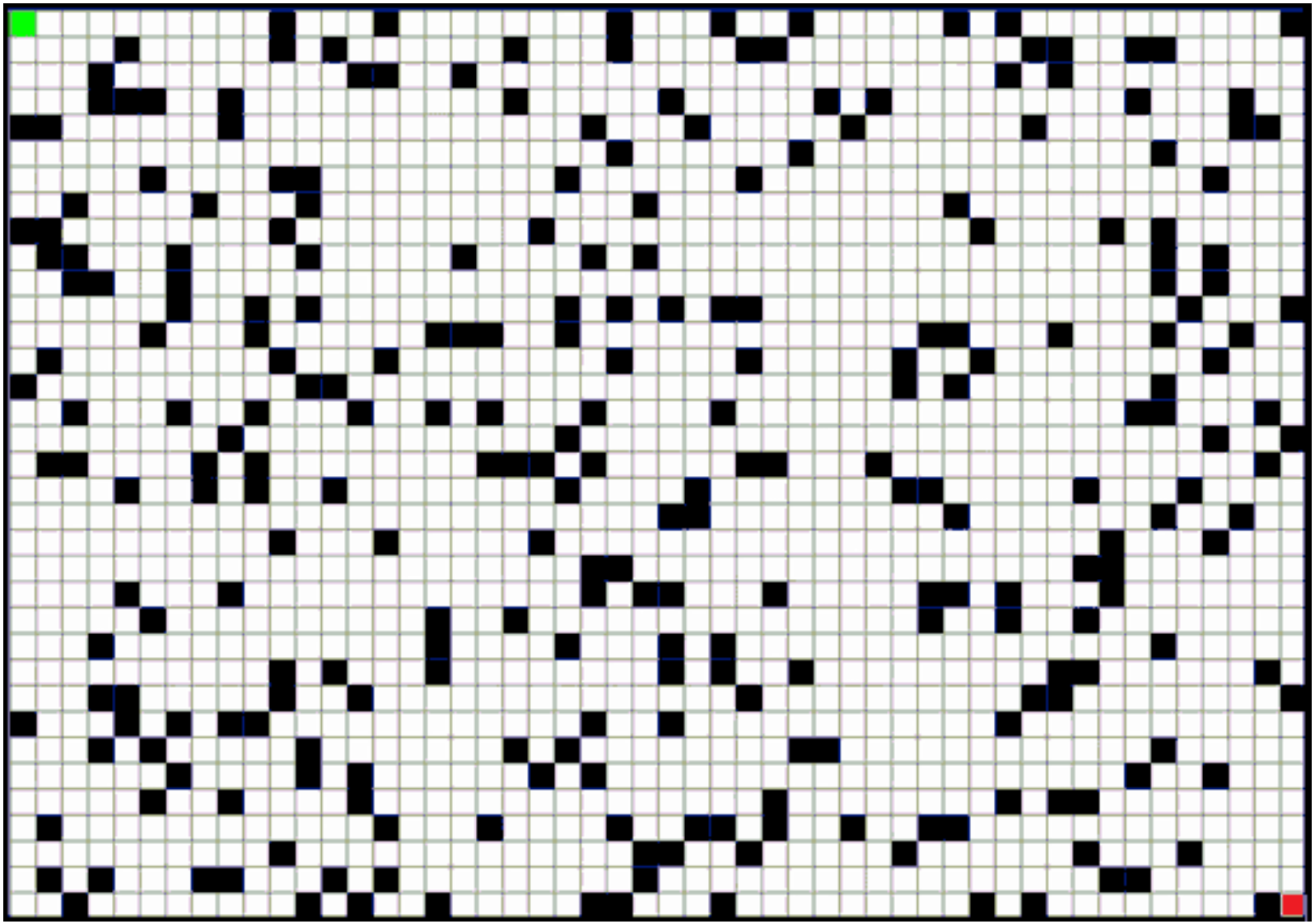
Введение в алгоритм A*

Виктор Васильевич Лепин

Робот


ANTENNA FOR RADIO LINK
TELEVISION CAMERA
RANGE FINDER
ON-BOARD LOGIC
CAMERA CONTROL UNIT
BUMP DETECTOR
CASTER WHEEL
DRIVE MOTOR
DRIVE WHEEL

Есть припядствия

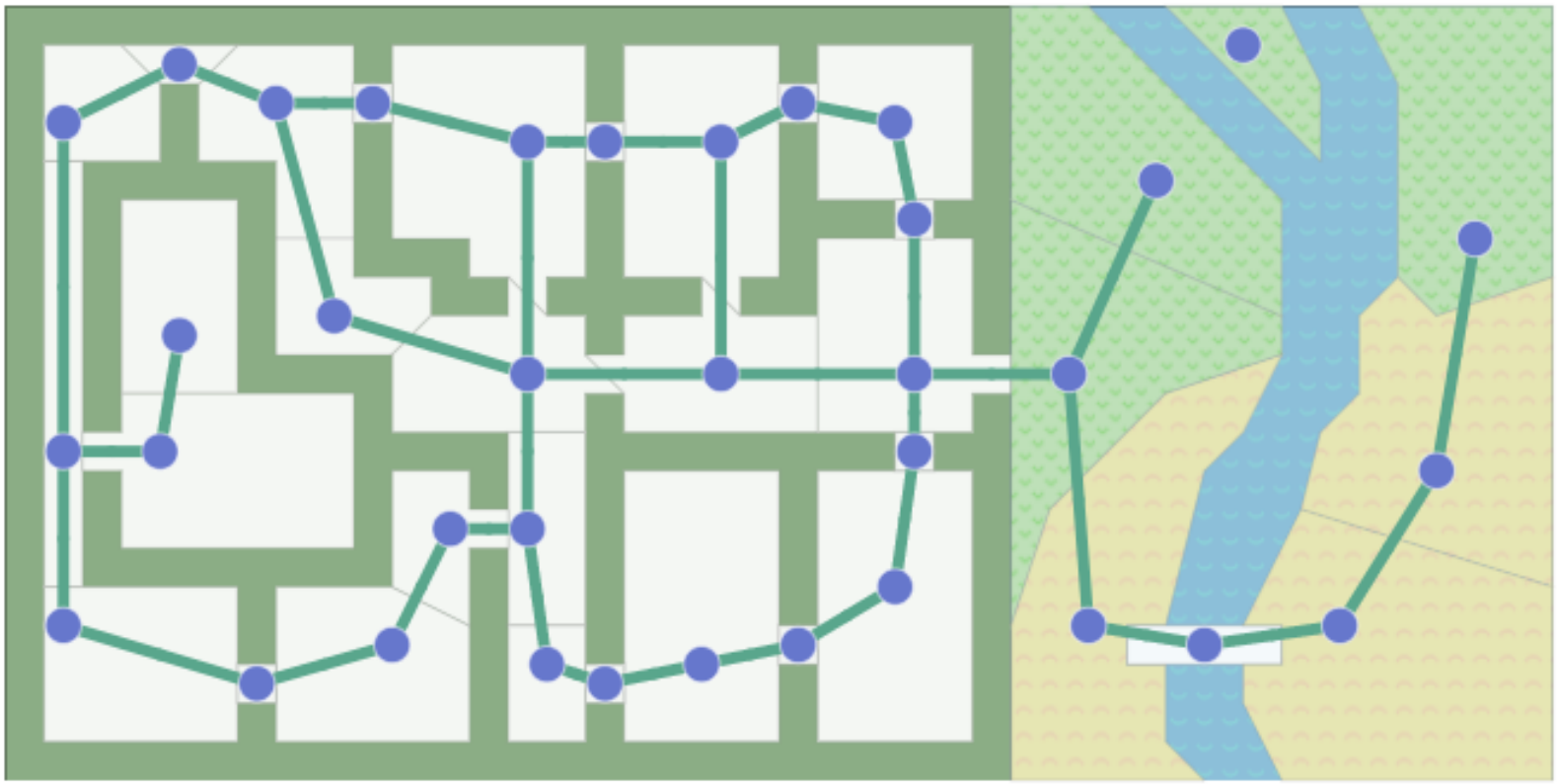


Поиск по графу

- Чтобы найти этот путь, мы можем использовать алгоритм поиска по графу, который работает, когда карта представлена в виде графа. Алгоритм A^* – популярный выбор для поиска по графу.
- Поиск в ширину – простейший из алгоритмов поиска по графу, так что давайте начнем с него и будем продвигаться к A^*.

Представление карты

Граф представляет собой набор местоположений («узлов») и связей («ребер») между ними. Вот граф, который можно дать на вход алгоритму A*:



Представление карты

A* больше ничего не видит. Видит только граф.
Он не знает, находится ли что-то в помещении или на улице, комната это или дверной проем, или насколько велика площадь.
Он видит только граф!

Представление карты

Алгоритм A* найдет путь в графе. Т.е. укажет вам последовательность перемещений из одного места в другое, но не скажет, как это сделать.
Помните, что он ничего не знает о комнатах или дверях; все, что он видит, это граф.
Вам нужно будет решить, означает ли ребро графа, возвращаемое A*, движение от плитки к плитке, или движение по прямой линии, или открытие двери.

Представление карты. Компромиссы

Для любой заданной карты существует множество различных способов построения графа поиска пути, который можно передать A*.
Приведенном графе, дверным проемам соответствуют узлы.

Представление карты

Граф поиска пути не обязательно должен совпадать с тем, что использует робот. Из карта робота с сеткой можно построить граф поиска пути без сетки или наоборот.
A* работает быстрее всего с наименьшим количеством узлов графа; с сетками часто легче работать, но они приводят к большому количеству узлов.
На этой странице рассматривается алгоритм A*, но не дизайн графа. Для пояснений на оставшейся далее будут использоваться сетки, потому что так проще визуализировать концепции.

Алгоритмы

- Поиск в ширину одинаково исследует во всех направлениях. Это невероятно полезный алгоритм не только для обычного поиска пути, но и для процедурной генерации карт, поиска пути в потоковых алгоритмах, карт расстояний и других типов анализа карт.
- Алгоритм Дейкстры (также называемый унифицированным поиском по стоимости) позволяет нам расставить приоритеты, какие пути исследовать. Вместо того, чтобы одинаково исследовать все возможные пути, он отдает предпочтение более дешевым путям. Мы можем назначить более низкие затраты на поощрение движения по дорогам, более высокие затраты на избегание лесов, более высокие затраты на предотвращение приближения к врагам и многое другое. Когда затраты на перемещение различаются, мы используем его вместо поиска в ширину.

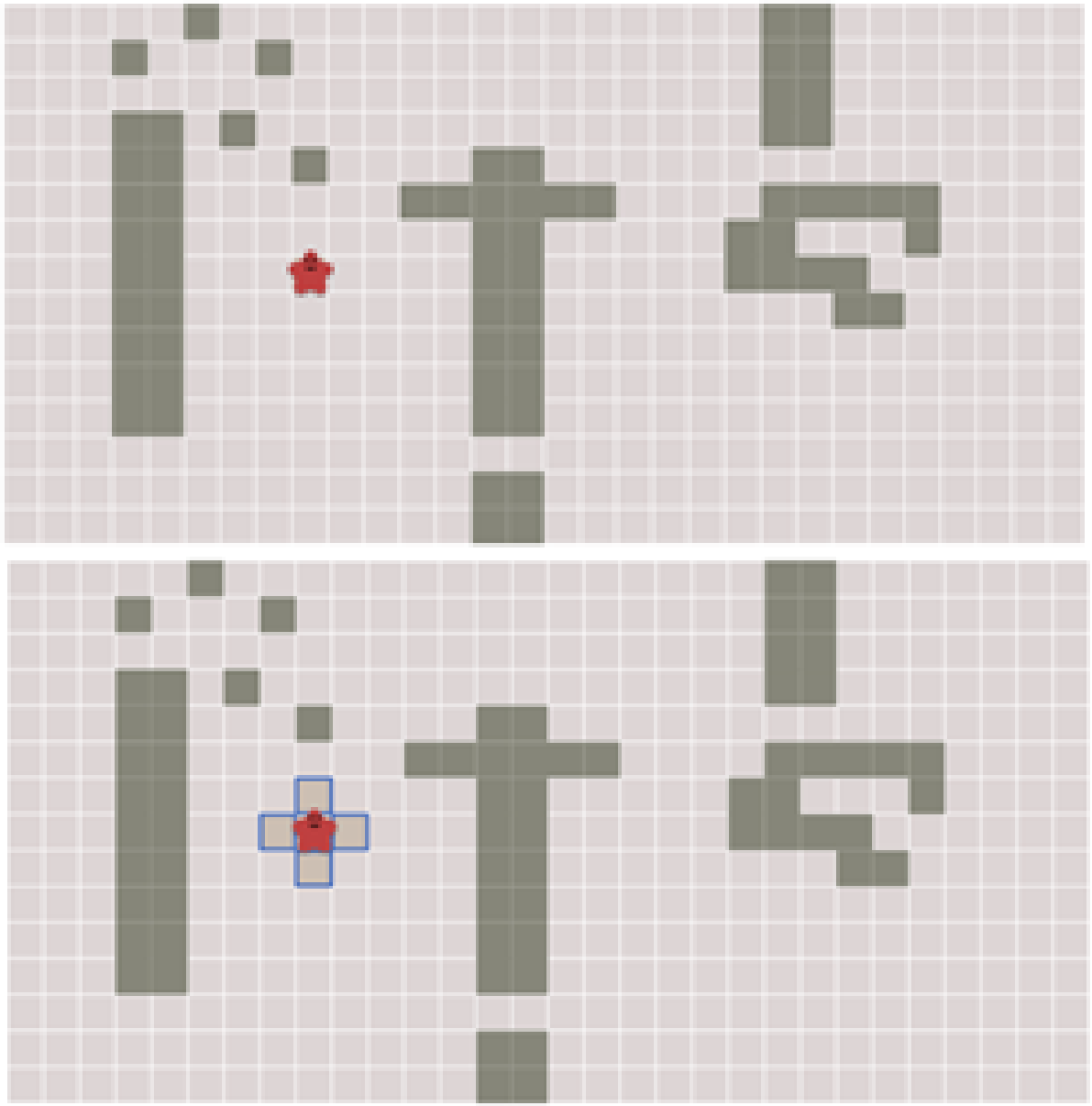
Алгоритмы

- A* — это модификация алгоритма Дейкстры, оптимизированная для одного пункта назначения. Алгоритм Дейкстры может найти пути ко всем местам; A* находит пути к одному местоположению или ближайшему из нескольких местоположений. Он отдает приоритет путям, которые, кажется, ведут ближе к цели.

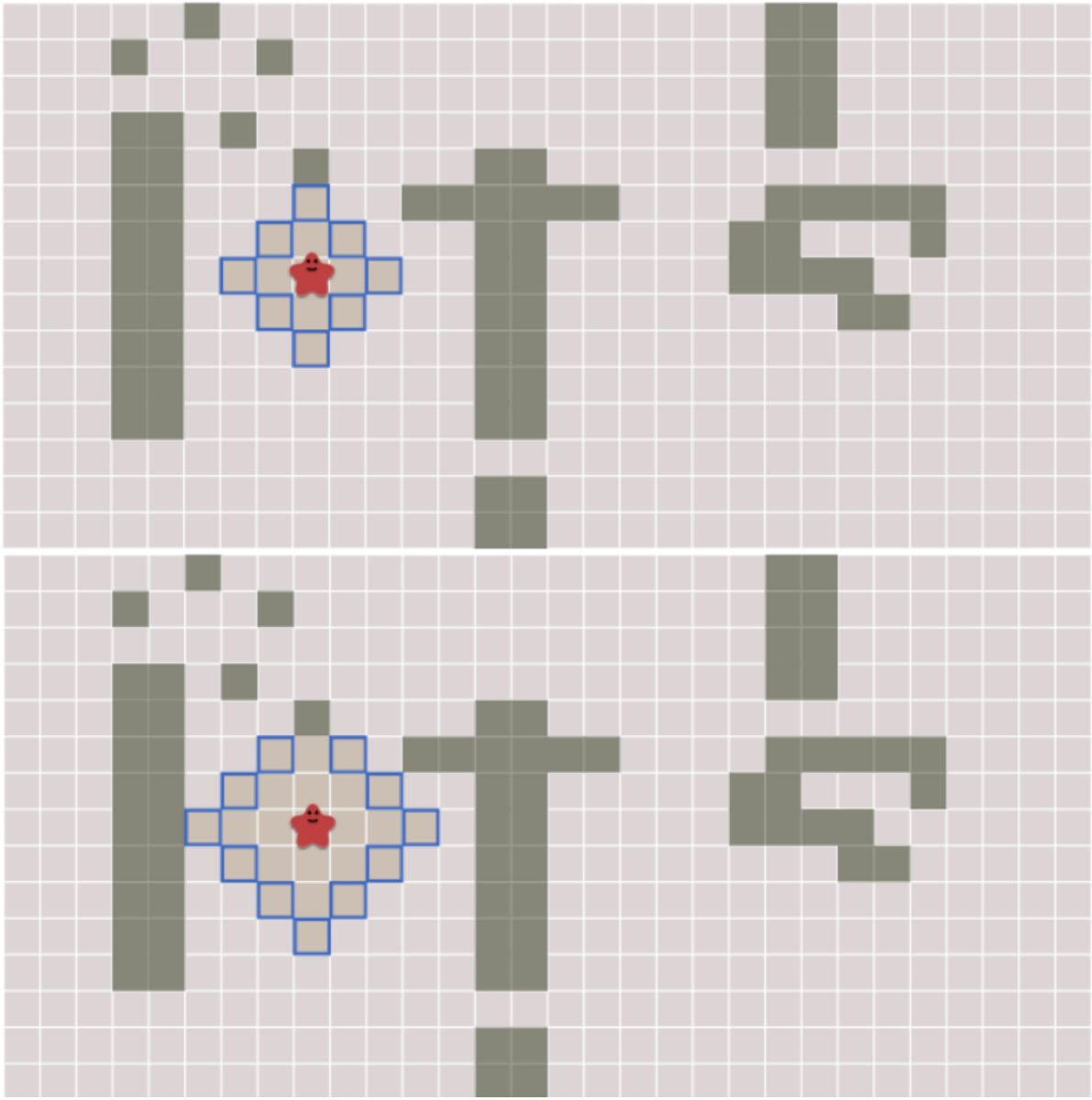
Поиск в ширину

- Ключевая идея всех этих алгоритмов заключается в том, что мы отслеживаем расширяющееся кольцо, называемое границей. В сетке этот процесс иногда называют «заливкой», но тот же метод работает и без сетки.

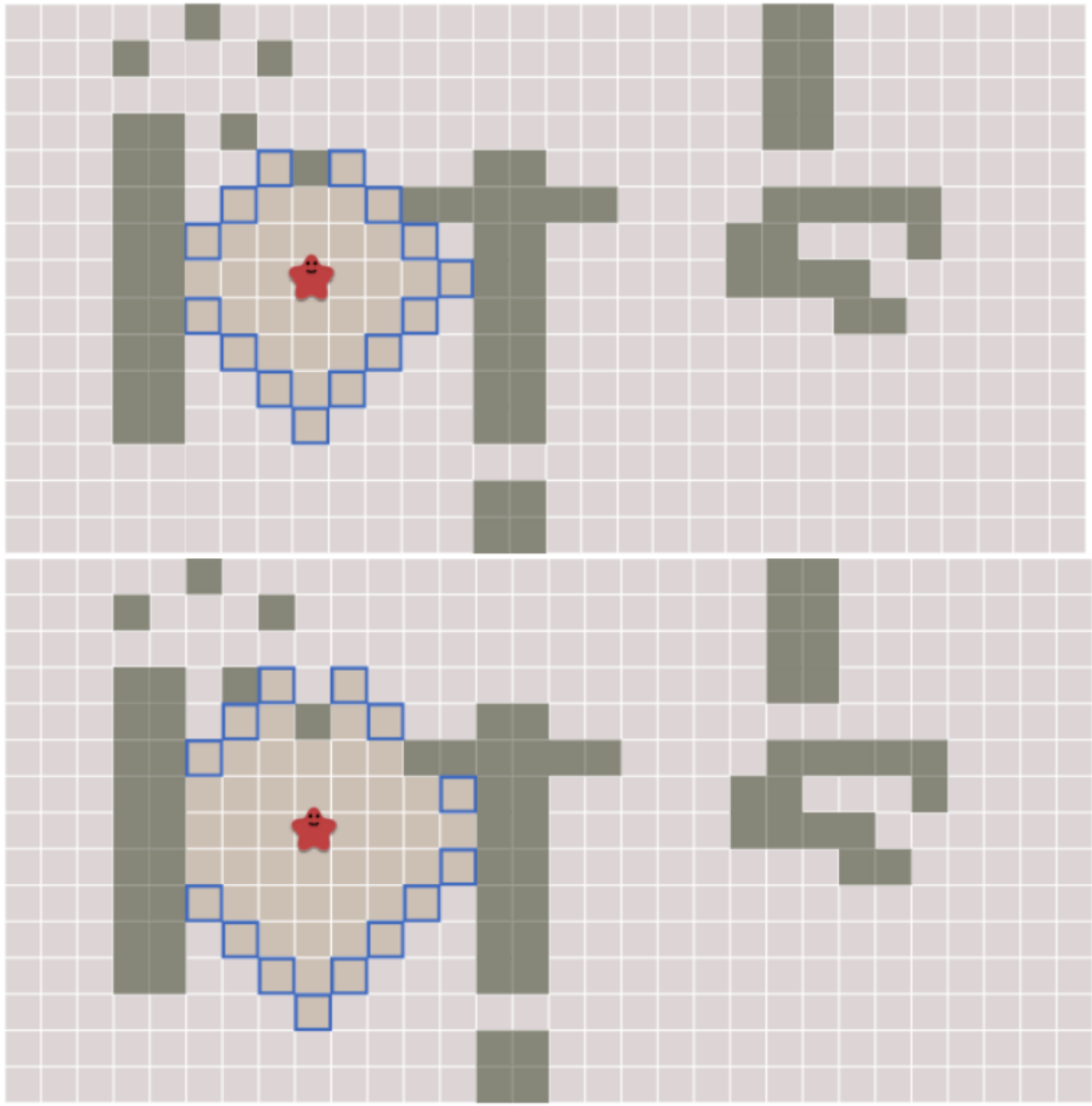
Поиск в ширину



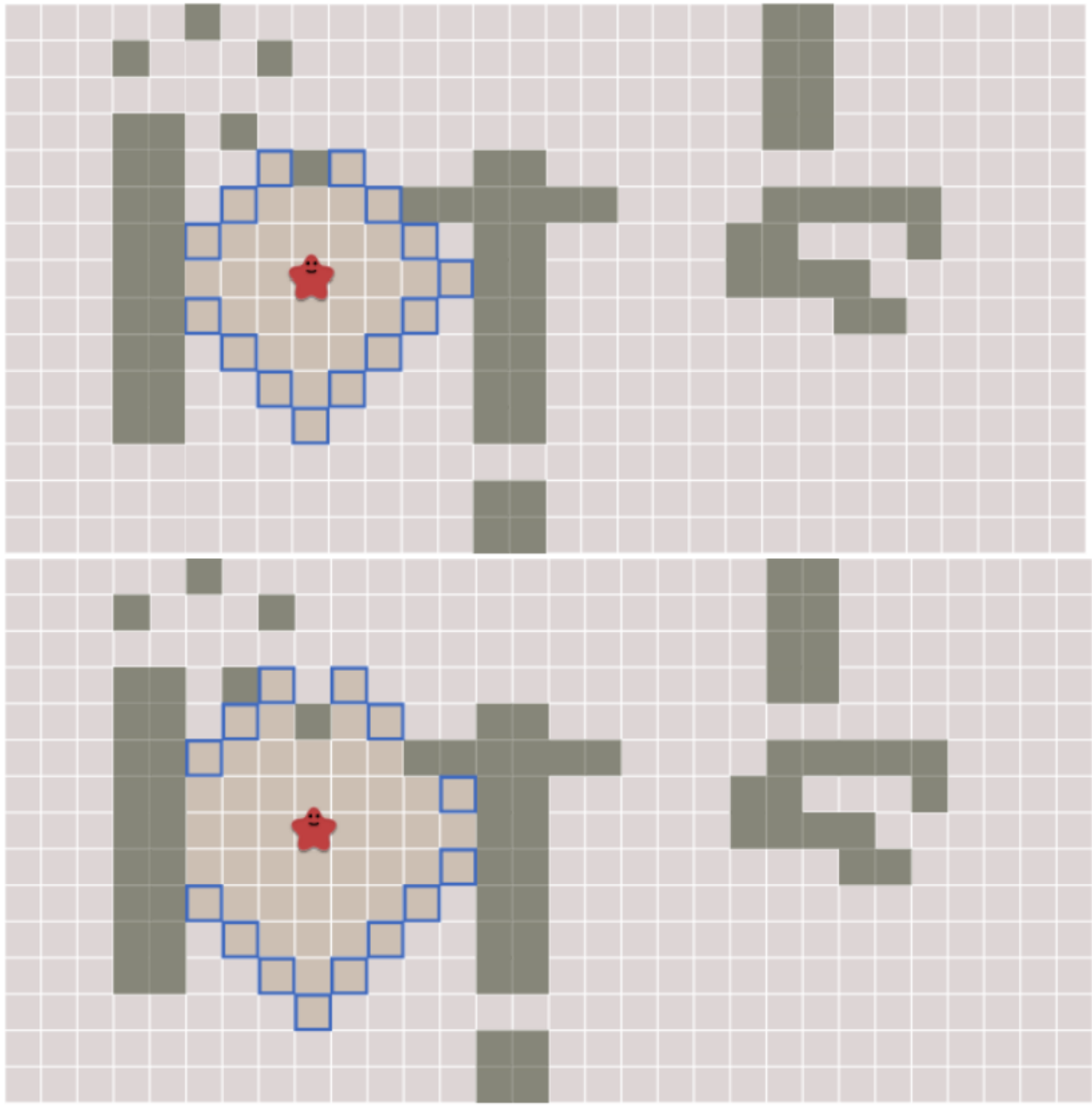
Поиск в ширину



Поиск в ширину



Поиск в ширину



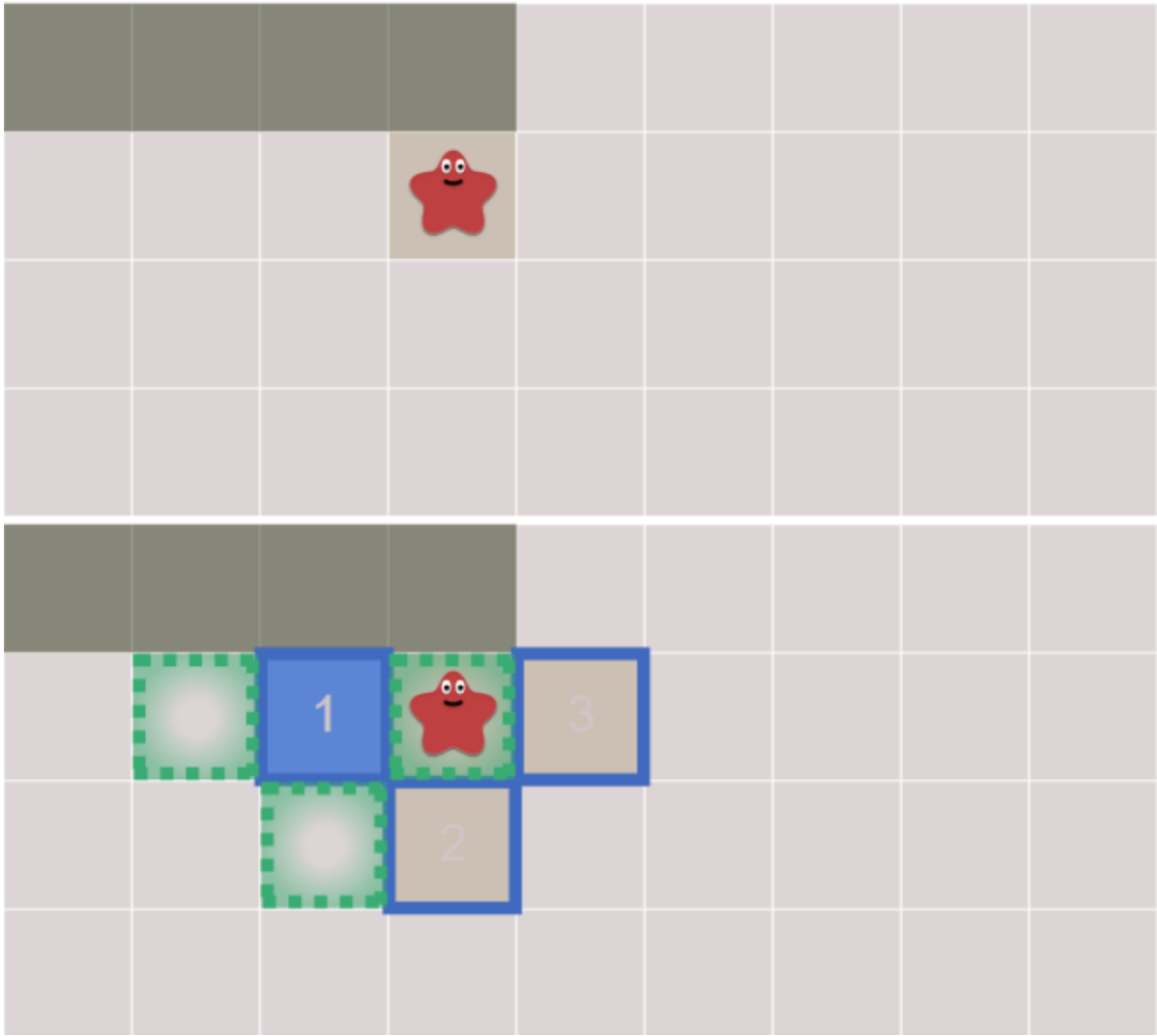
Поиск в ширину

Это всего десять строк кода (Python):

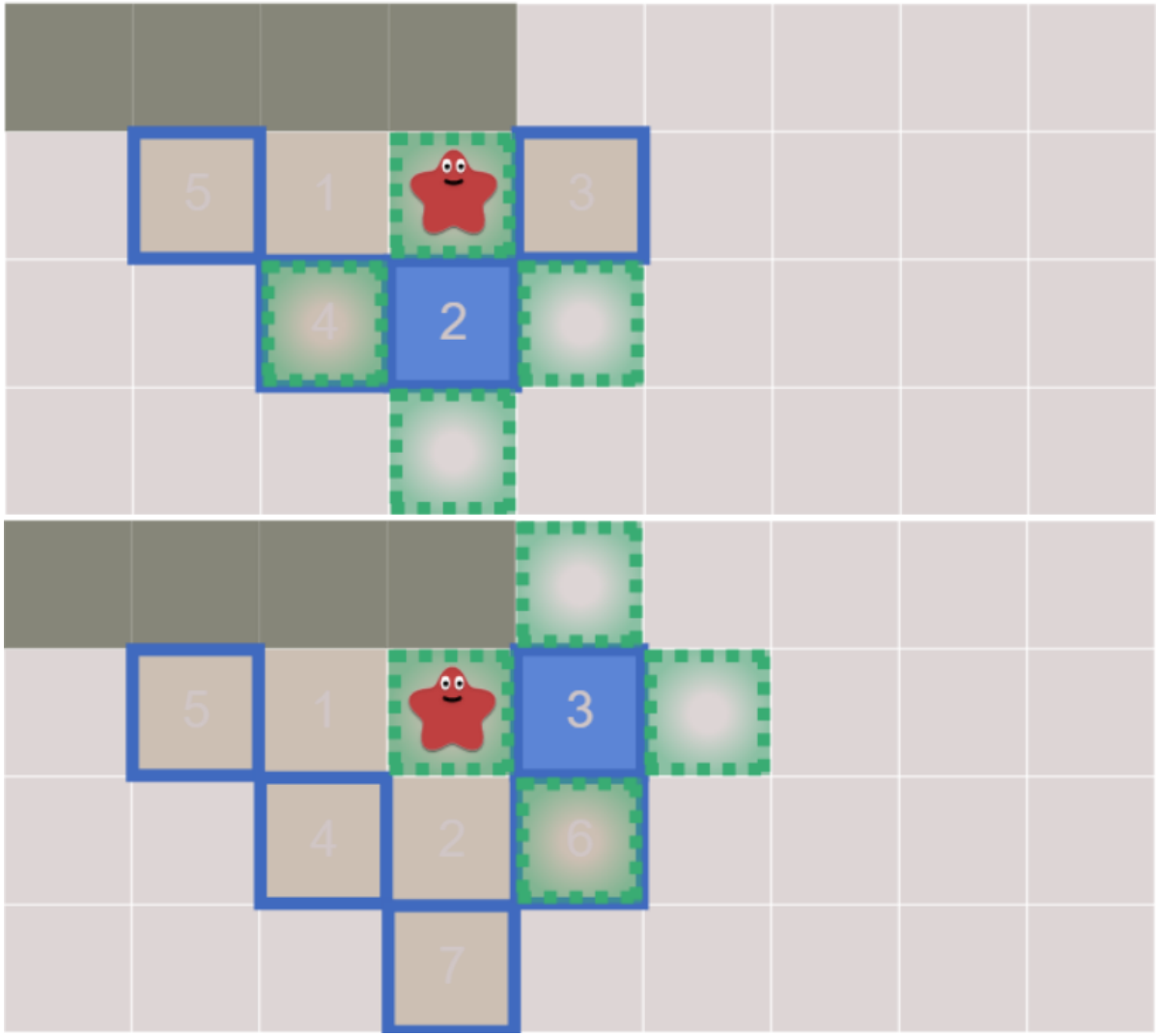
```
frontier = Queue()
frontier.put(start )
reached = set()
reached.add(start)
while not
frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
    if next not in reached:
      frontier.put(next)
      reached.add(next)
```

Поиск в ширину

Давайте посмотрим на это вблизи. Плитки нумеруются в порядке их посещения.



Поиск в ширину



Поиск в ширину

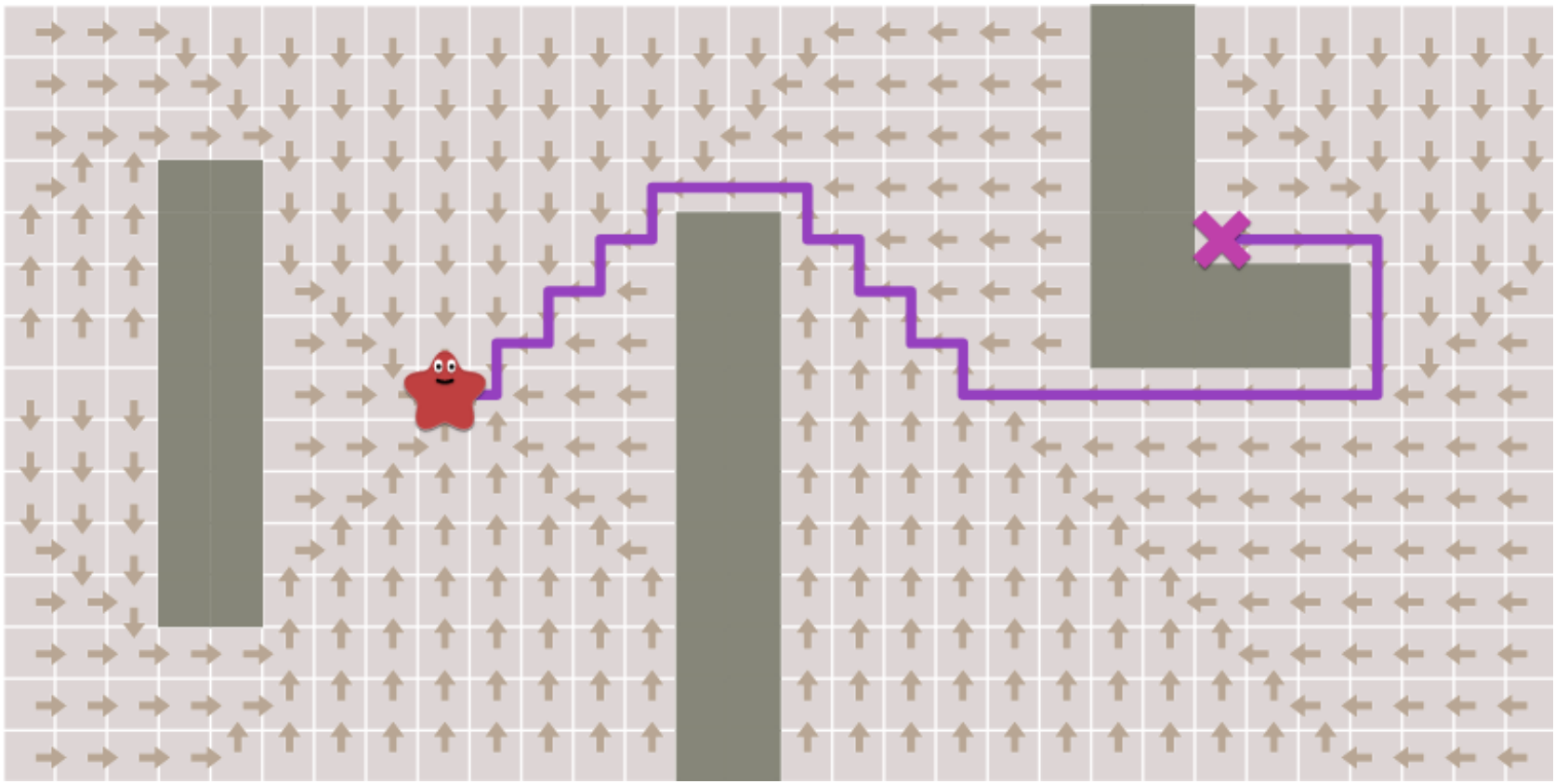
- Этот цикл является сущностью алгоритмов поиска по графу, включая A*. Но как найти кратчайший путь?
- Цикл на самом деле не создает пути; он только говорит нам, как посетить все доступное на карте.
- Это связано с тем, что поиск в ширину можно использовать не только для поиска путей, но его также можно использовать для карт расстояний и многих других вещей.
- Однако здесь мы хотим использовать его для поиска путей, поэтому давайте изменим цикл, чтобы отслеживать, откуда мы пришли для каждого достигнутого местоположения

Поиск в ширину

```
frontier = Queue()
frontier.put(start * )
CameFrom = dict() # path A->B is stored as CameFrom[B] == A
CameFrom[start] = None
while not frontier.empty():
   current = frontier.get()
   for next in graph.neighbors(current):
    if next not in CameFrom:
     frontier.put(next)
     CameFrom[next] = current
```

Теперь CameFrom для каждого местоположения указывает место, откуда мы пришли. Это как «хлебные крошки». Их достаточно, чтобы восстановить весь путь.

Поиск в ширину

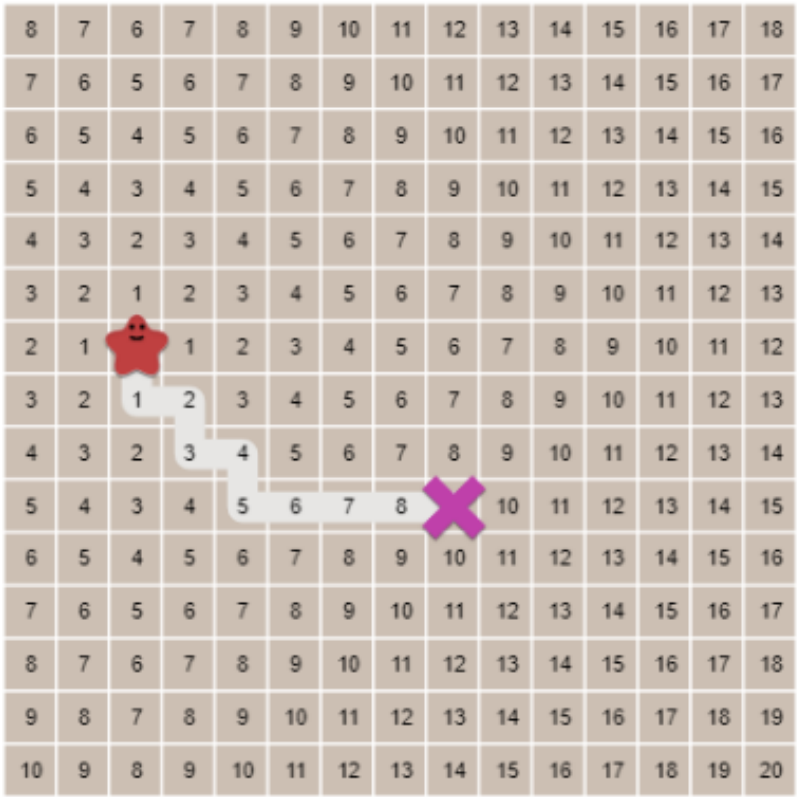


Код для восстановления путей прост: следуйте по стрелкам в обратном направлении от цели к началу.

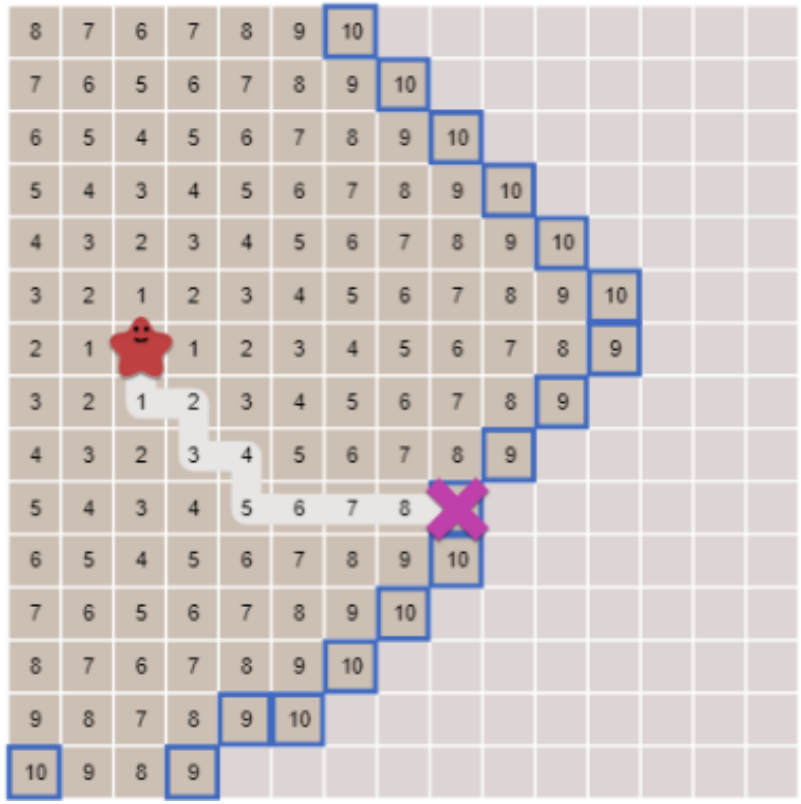
Ранний выход

Мы нашли пути из одного места во все остальные места. Часто нам не нужны все пути; нам нужен только путь из одного места в другое место. Мы можем перестать расширять границы, как только найдем нашу цель.

Без раннего выхода



С досрочным выходом



Ранний выход

```
frontier = Queue()
frontier.put(start * )
CameFrom = dict()
CameFrom[start] = None
while not frontier.empty():
  current = frontier.get()
  if current == goal: ×
   break
  for next in graph.neighbors(current):
   if next not in CameFrom:
    frontier.put(next)
    CameFrom[next] = current
```

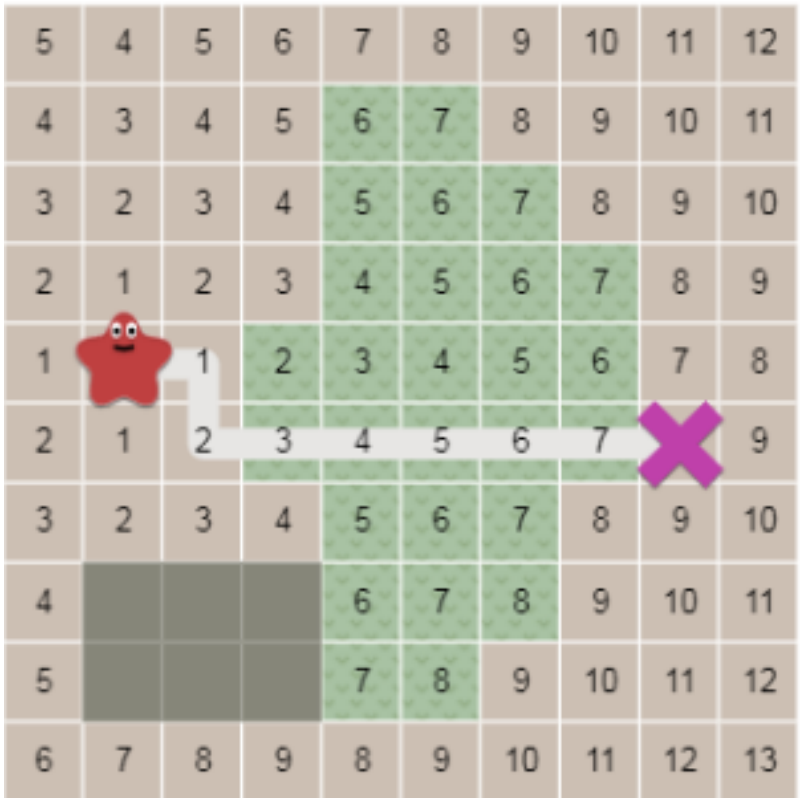
Расходы на передвижение

- Пока что сделанный нами шаг имеет одинаковую «стоимость».
- В некоторых сценариях поиска пути существуют разные затраты на разные типы движения.
- Например, в движение по равнине или пустыне может стоить 1 очко движения, а движение через лес или холмы может стоить 5 очков движения.
- Другим примером является диагональное движение по сетке, которое стоит больше, чем осевое движение.
- Мы хотели бы, чтобы робот учитывал эти затраты.

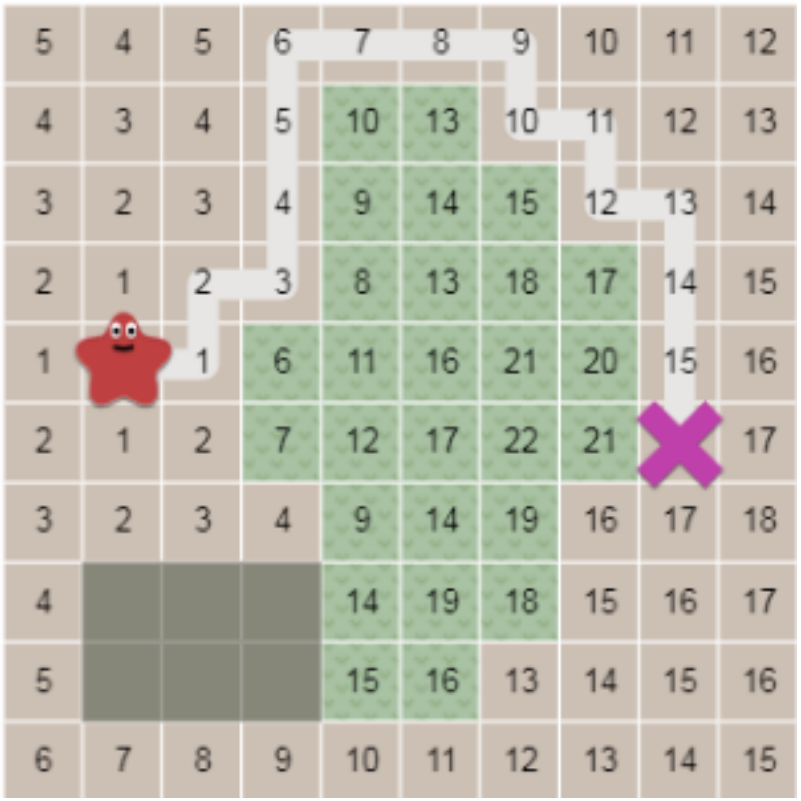
Расходы на передвижение

Сравним количество шагов от старта с расстоянием от старта:

Количество шагов



Расстояние



Расходы на передвижение

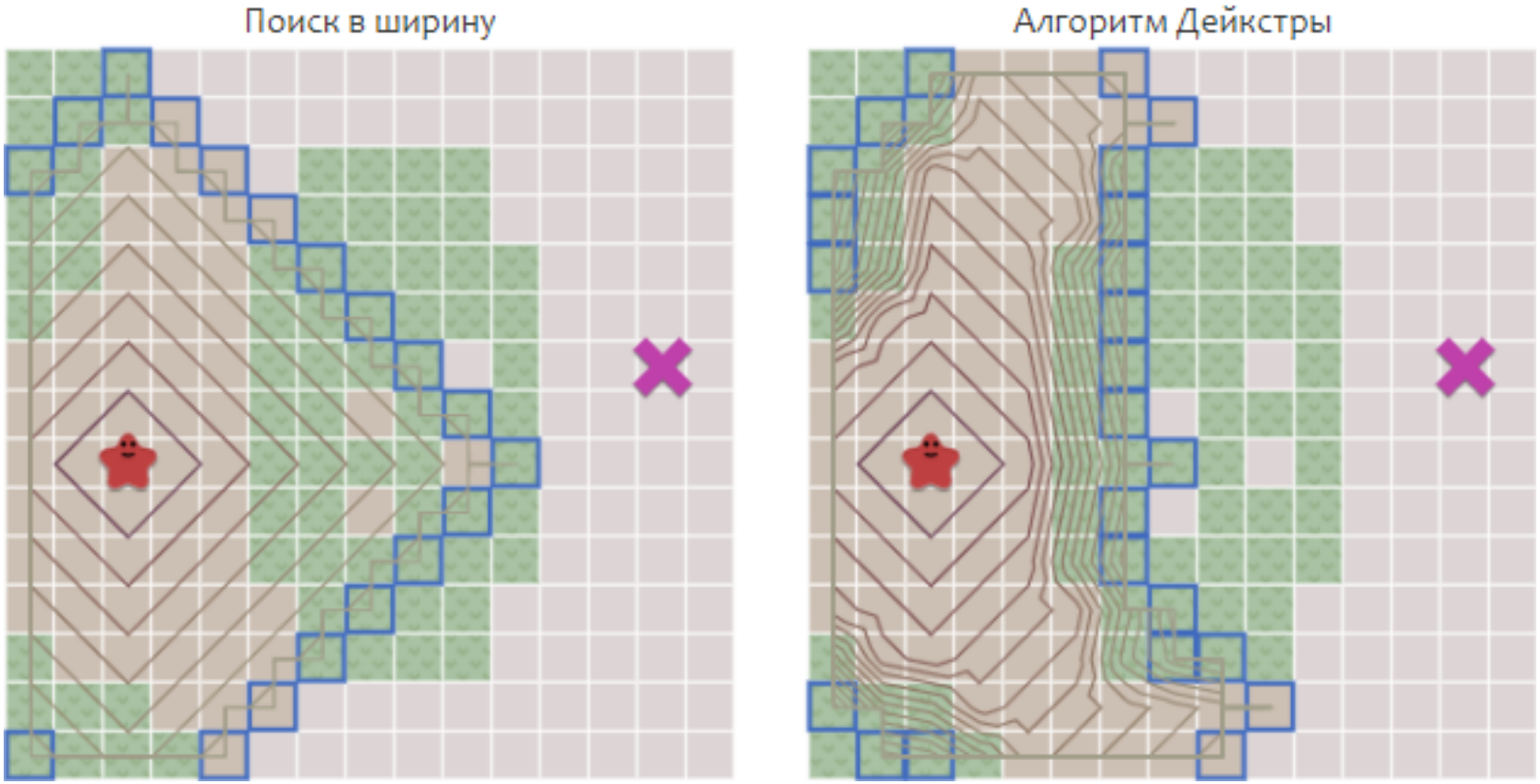
- Для этого нам нужен алгоритм Дейкстры (или поиск по единой стоимости).
- Чем он отличается от поиска в ширину?
- Нам нужно отслеживать стоимость перемещения, поэтому давайте добавим новую переменную, CostSoFar чтобы отслеживать общую стоимость перемещения из начального местоположения.
- Мы хотим учитывать затраты на передвижение при принятии решения о том, как оценивать местоположения; давайте превратим нашу очередь в очередь с приоритетом.
- Менее очевидно, что мы можем в конечном итоге посетить место несколько раз с разной стоимостью, поэтому нам нужно немного изменить логику.
- Вместо добавления местоположения к границе, если местоположение никогда не было достигнуто, мы добавим его, если новый путь к местоположению лучше, чем лучший предыдущий путь.

Алгоритм Дейкстры

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next]:
            cost_so_far[next] = new_cost
            priority = new_cost
            frontier.put(next, priority)
            came_from[next] = current
```

Алгоритм Дейкстры

Использование очереди с приоритетом вместо обычной очереди меняет способ расширения границы. Контурные линии – один из способов увидеть это.



Расходы на передвижение

- Стоимость движения также может быть использована для того, чтобы избегать или предпочитать районы в зависимости от близости к врагам или союзникам.
- Версия алгоритма Дейкстры и A*, которую я представляю на этих слайдах, отличается от того, что есть в учебниках по алгоритмам.
- Это намного ближе к тому, что называется поиском по единой стоимости.

Эвристический поиск

- С поиском в ширину и алгоритмом Дейкстры граница расширяется во всех направлениях.
- Это разумный выбор, если вы пытаетесь найти путь ко всем локациям или ко многим локациям.
- Однако распространенный случай – найти путь только к одному местоположению.
- Сделаем так, чтобы граница расширялась к цели больше, чем в другие стороны.
- Во-первых, мы определим эвристическую функцию, которая сообщает нам, насколько мы близки к цели:

```
def heuristic(a, b):
    # Manhattan distance on a square grid
    return abs(a.x - b.x) + abs(a.y - b.y)
```

Эвристический поиск

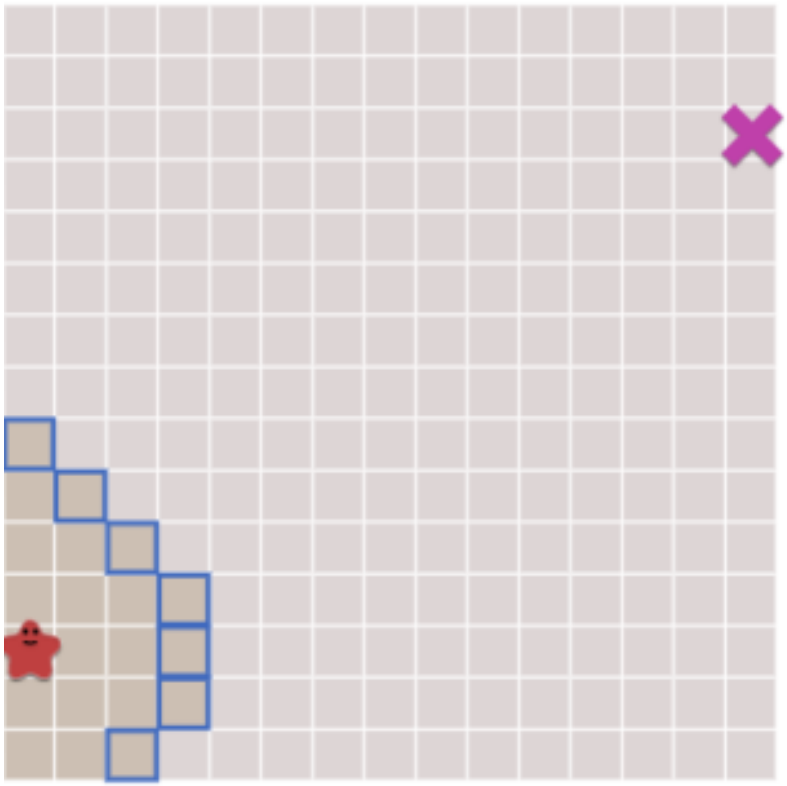
- В алгоритме Дейкстры мы использовали фактическое расстояние от начала для упорядочения приоритетной очереди.
- Вместо этого в Greedy Best First Search мы будем использовать расчетное расстояние до цели для упорядочения приоритетной очереди.
- Ближайшее к цели место будет исследовано первым.
- Код использует приоритетную очередь из алгоритма Дейкстры, но без CostSoFar:

Эвристический поиск

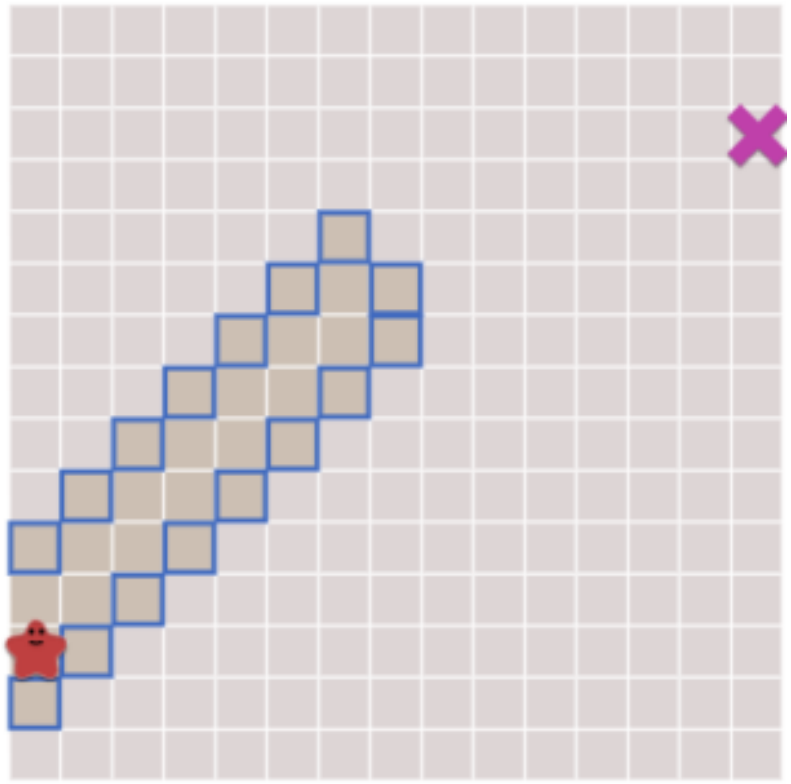
```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in came_from:
            priority = heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

Посмотрим, насколько хорошо это работает:

Алгоритм Дейкстры

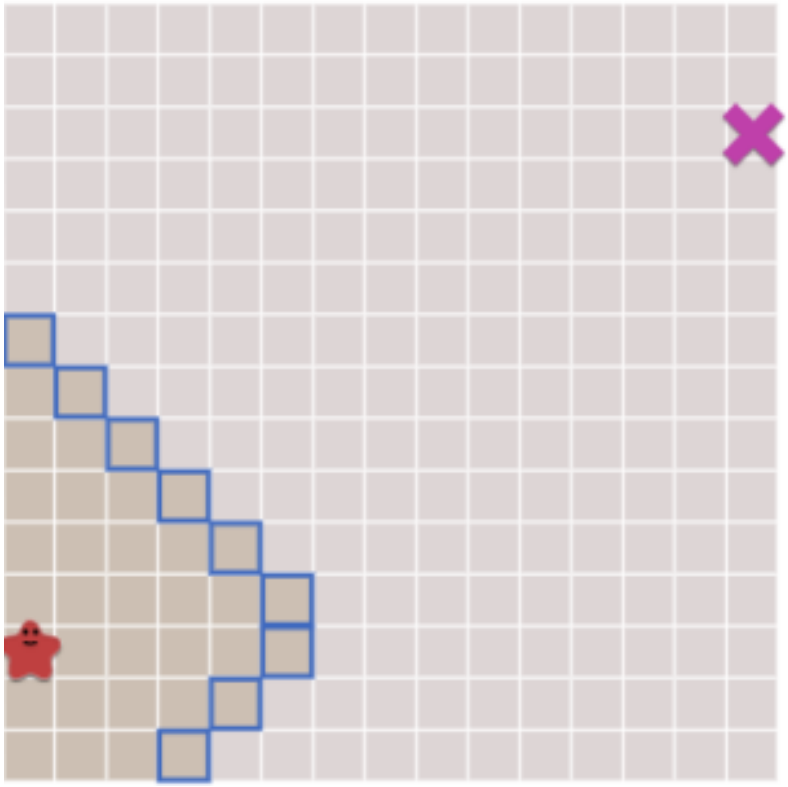


Жадный поиск по первому наилучшему

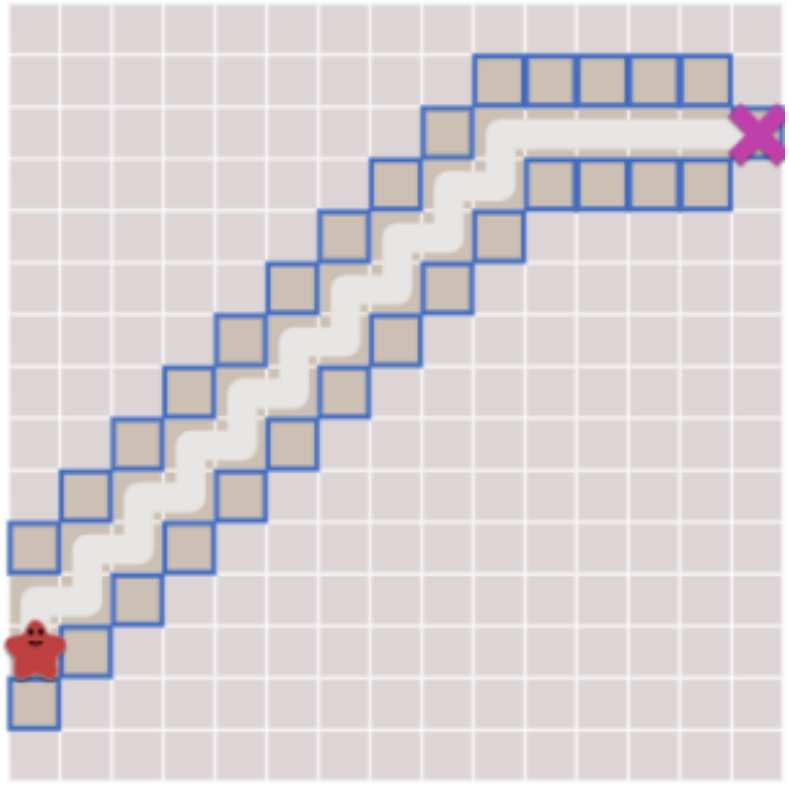


Эвристический поиск

Алгоритм Дейкстры

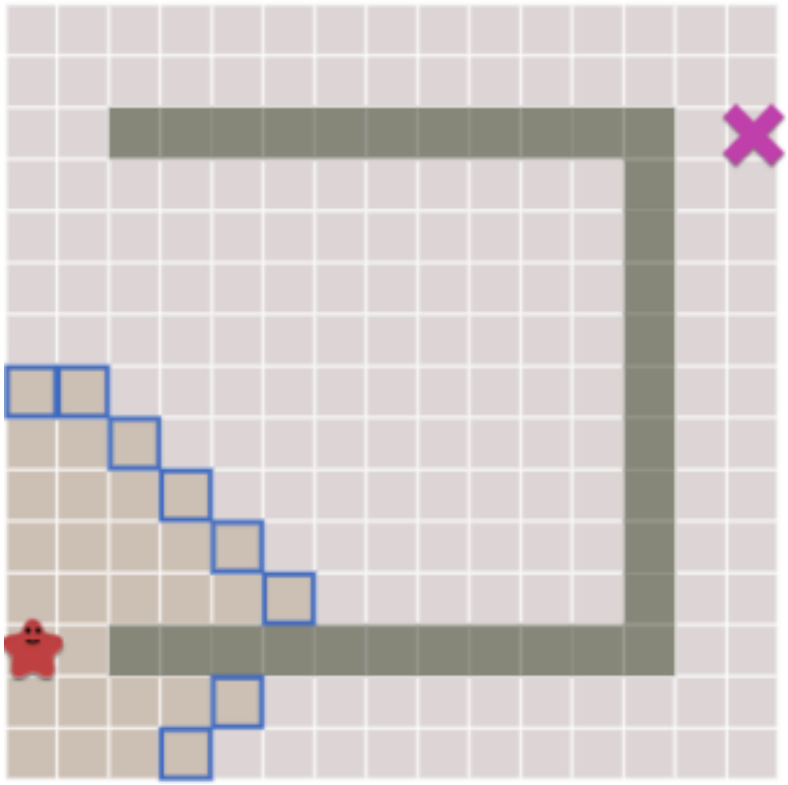


Жадный поиск по первому наилучшему

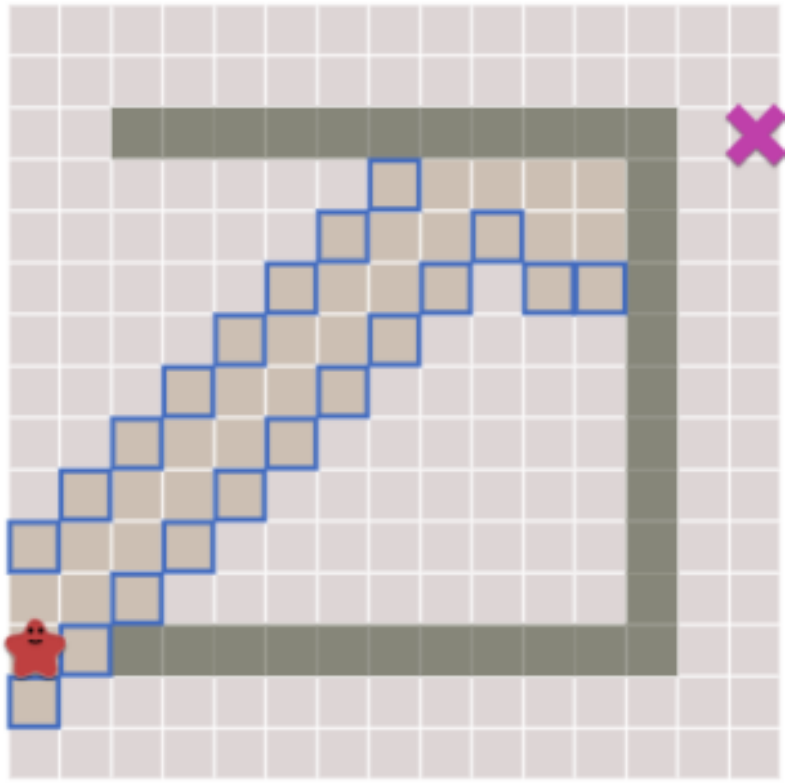


Пусть имеется стена – препядствие.

Алгоритм Дейкстры

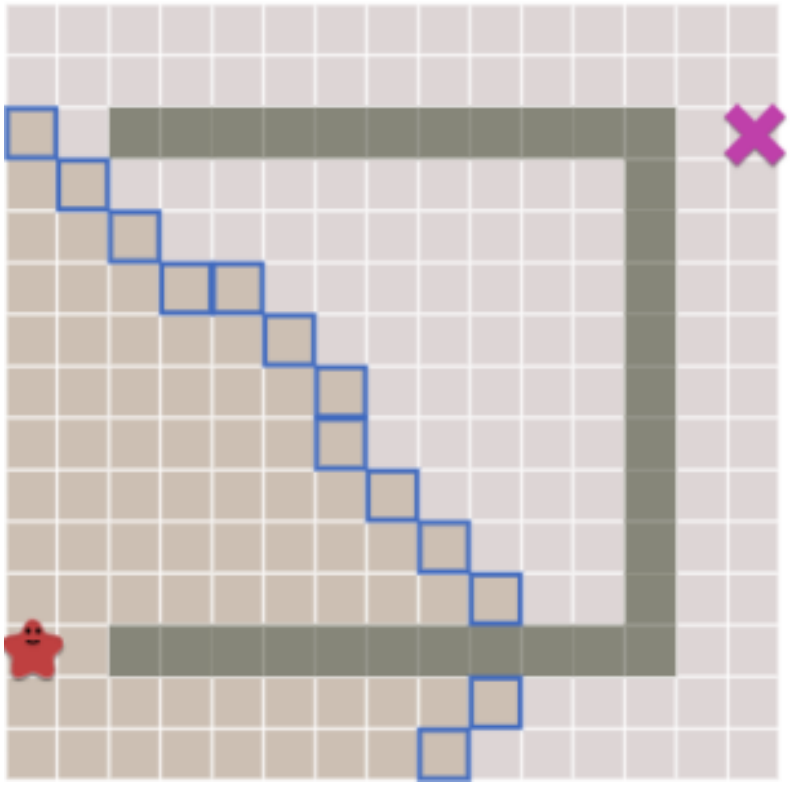


Жадный поиск по первому наилучшему

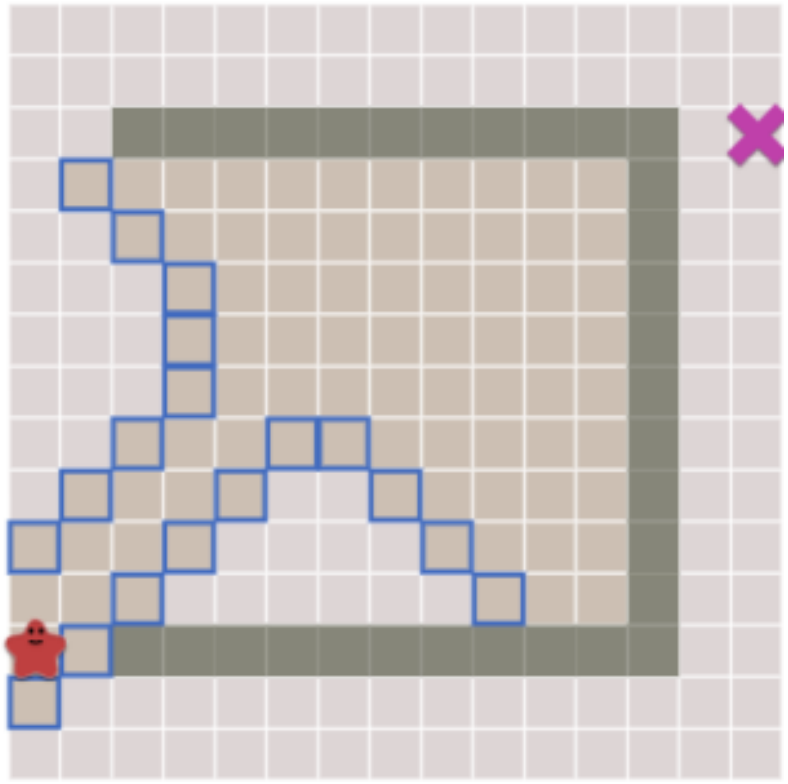


Эвристический поиск

Алгоритм Дейкстры

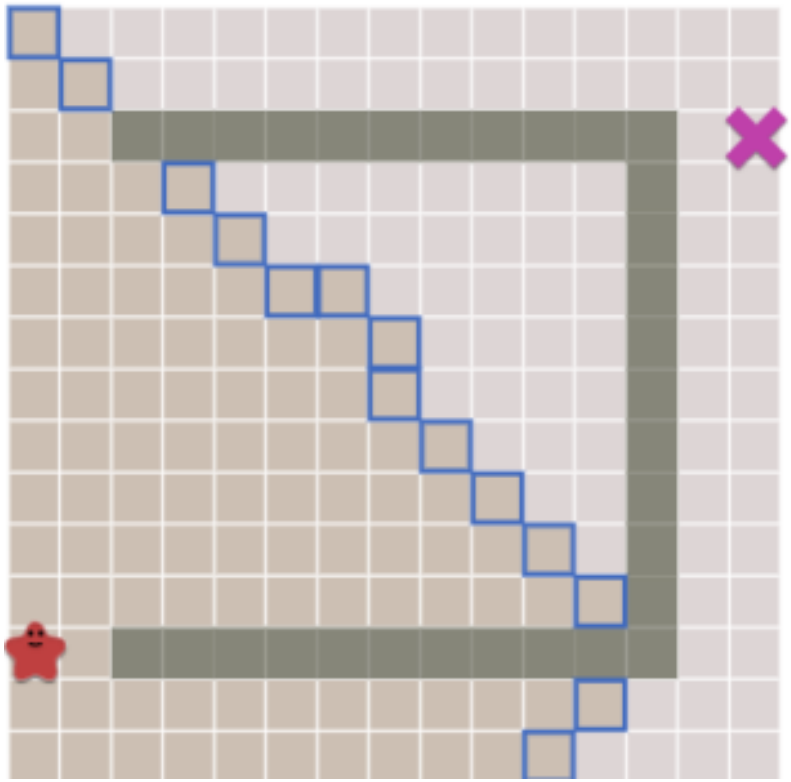


Жадный поиск по первому наилучшему

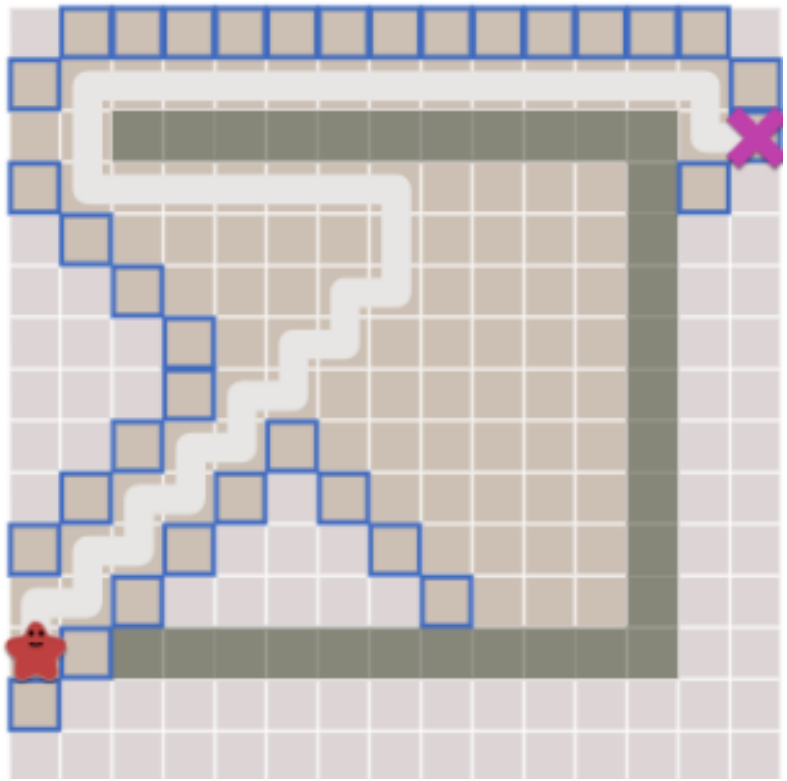


Эвристический поиск

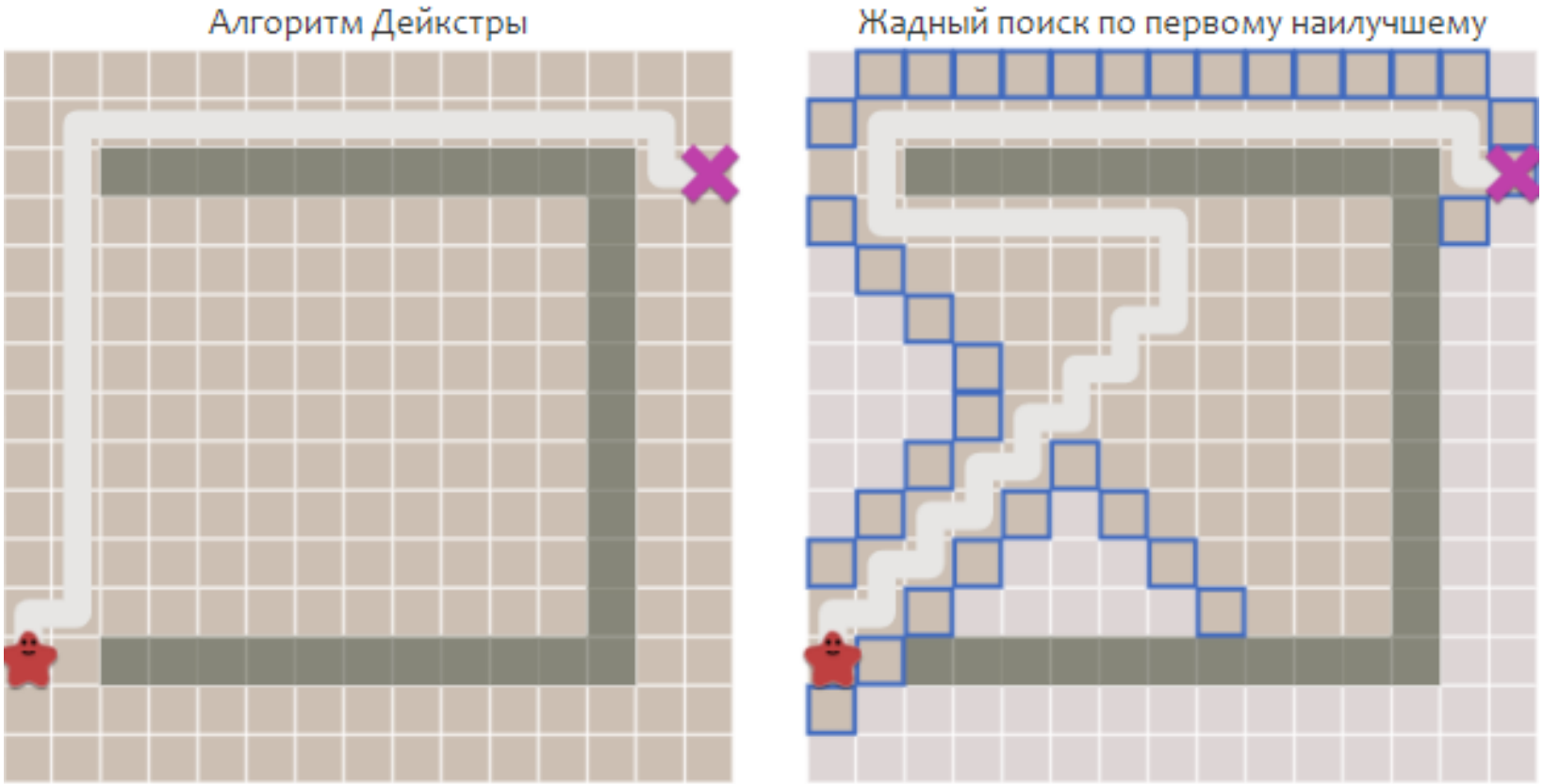
Алгоритм Дейкстры



Жадный поиск по первому наилучшему



Эвристический поиск



Эти пути не самые короткие. Таким образом, этот алгоритм работает быстрее, когда препятствий не так много, но пути не так хороши. Можем ли мы это исправить? Да!

Алгоритм A*

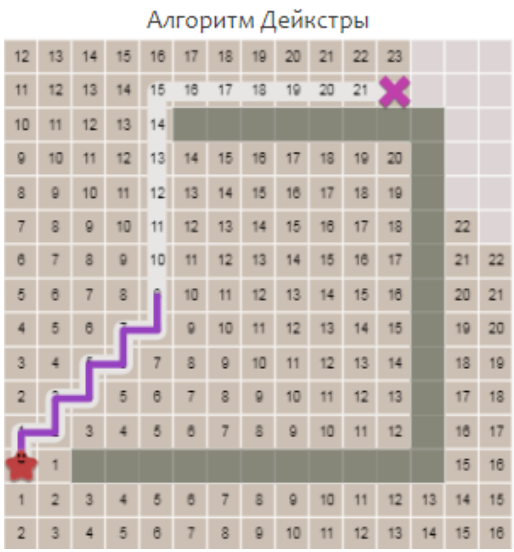
- Алгоритм Дейкстры хорошо работает, чтобы найти кратчайший путь, но он тратит время на изучение неперспективных направлений.
- Жадный поиск по первому наилучшему поиску исследует многообещающие направления, но может не найти кратчайшего пути.
- Алгоритм A* использует как фактическое расстояние от начала, так и расчетное расстояние до цели.
- Код очень похож на алгоритм Дейкстры:

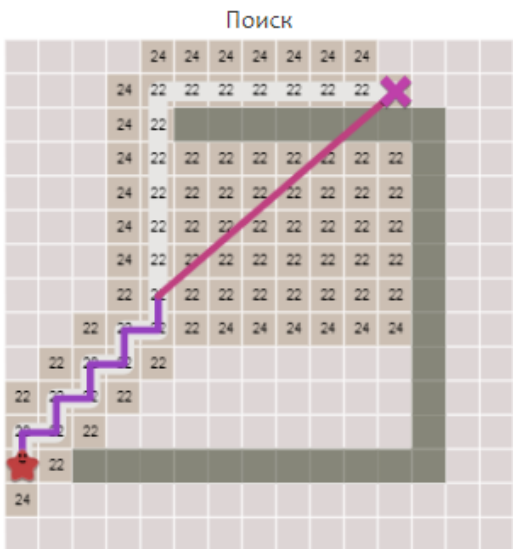
Алгоритм A*

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Алгоритм A*

Сравните алгоритмы: Алгоритм Дейкстры вычисляет расстояние от начальной точки. Жадный поиск по первому наилучшему оценивает расстояние до целевой точки. A* использует сумму этих двух расстояний.





Алгоритм A*

- A* — лучшее из обоих подходов.
- Пока эвристика не переоценивает расстояния, A* находит оптимальный путь, как это делает алгоритм Дейкстры.
- A* использует эвристику для переупорядочивания узлов, чтобы повысить вероятность того, что целевой узел будет обнаружен раньше.

Какой алгоритм следует использовать для поиска путей

- Если вы хотите найти пути из всех местоположений или ко всем местам, используйте поиск в ширину или алгоритм Дейкстры. Используйте поиск в ширину, если стоимость перемещения одинакова; используйте алгоритм Дейкстры, если стоимость движения варьируется.
- Если вы хотите найти пути к одному местоположению или к ближайшей из нескольких целей, используйте жадный поиск наилучшего первого или A*. В большинстве случаев предпочтительнее A*. Если вы испытываете искушение использовать жадный поиск наилучшего первого, рассмотрите возможность использования A* с «недопустимой» эвристикой .

Что можно сказать об оптимальных путях?

- Поиск в ширину и алгоритм Дейкстры гарантированно найдут кратчайший путь для входного графа.
- Жадный поиск наилучшего первого не является таким.
- A* гарантированно найдет кратчайший путь, если эвристика никогда не превышает истинное расстояние.
- По мере того как эвристика становится меньше, A* превращается в алгоритм Дейкстры.
- По мере того, как эвристика становится больше, A* превращается в жадный поиск наилучшего первого.

Как насчет производительности?

- Лучше всего удалить ненужные места при построении графа.
- Уменьшение размера графа помогает всем алгоритмам поиска по графу.
- После этого используйте самый простой алгоритм, какой только сможете; простые очереди работают быстрее.
- Жадный поиск по первому наилучшему обычно работает быстрее, чем алгоритм Дейкстры, но не дает оптимальных путей.
- A* – хороший выбор для большинства нужд поиска пути.

А как же без карт?

- Я показываю карты здесь, потому что считаю, что с помощью карты легче понять, как работают алгоритмы.
- Однако эти алгоритмы поиска по графу можно использовать на любом виде графа, а не только на картах.
- Затраты на перемещение на картах становятся произвольными весами на ребрах графа.
- Эвристики не так легко переносятся на произвольные графы; вы должны разработать эвристику для каждого типа графа.
- Для плоских карт расстояния – хороший выбор.

Что еще может возникнуть в задаче для робота?

- Имейте в виду, что поиск по графу – это только часть того, что вам нужно.
- Сам по себе A* не занимается такими вещами, как совместное движение, перемещение препятствий, изменение карты, оценка опасных зон, радиус поворота, размеры объектов, анимация, сглаживание пути или множество других тем.

Оценочная функция

- Порядок обхода вершин определяется эвристической оценочной функцией «расстояние + стоимость» (обычно обозначаемой как $f(x)$).
- Эта функция – сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как $g(x)$ и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как $h(x)$).

$$f(x) = g(x) + h(x)$$

Функция эвристической оценки расстояния от рассматриваемой вершины к конечной

- Функция $h(x)$ должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине.
- Например, для задачи маршрутизации $h(x)$ может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.
- Допустимая эвристика $h(x)$ может быть получена из упрощенной версии задачи, или информации из баз данных шаблонов, в которых хранятся точные решения подзадач задачи, или с помощью индуктивных методов обучения.

Свойства

- A* – оптимально эффективен для заданной допустимой эвристики h.
- Это значит, что любой другой алгоритм исследует не меньше узлов, чем A* (за исключением случаев, когда существует несколько частных решений с одинаковой эвристикой, точно соответствующей стоимости оптимального пути).

Свойства

- Поиск в глубину и поиск в ширину являются двумя частными случаями алгоритма A*.
- Если положить $h(x) = 0$ для всех вершин, то мы получим ещё один специальный случай – алгоритм Дейкстры.

Оценка сложности

- Временная сложность алгоритма A* зависит от эвристики.
- В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x));$$

где $h*$ – оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка $h(x)$ не должна расти быстрее, чем логарифм от оптимальной эвристики.

Оценка сложности

- Но еще большую проблему, чем временная сложность, представляют собой потребляемые алгоритмом ресурсы памяти.
- В худшем случае ему приходится помнить экспоненциальное количество узлов.
- Для борьбы с этим было предложено несколько вариаций алгоритма, таких как
  - алгоритм A* с итеративным углублением (iterative deeping A*, IDA*),
  - A* с ограничением памяти (memory-bounded A*, MA*),
  - упрощённый MA* (simplified MA*, SMA*) и
  - рекурсивный поиск по первому наилучшему совпадению (recursive best-first search, RBFS).