Bio-Inspired Algorithms

EPIGEP-EP-4-S4-UTO-E309

Daniel Vaz

daniel.ramosvaz@esiee.fr

2026-02-05

ESIEE PARIS

Objectives and Operation

Goals of the Course

Discover classic bio-inspired algorithms

- Origin of artificial intelligence
- Used even today

Develop our algorithmic thinking

- Heuristic techniques
- Adapting algorithms to specific situations

Program

1. Introduction
2. Local Search
3. Simulated Annealing
4. Ant Colony Optimization
5. Evolutionary Algorithms
6. Genetic Programming

Teaching Outcomes

At the end of the course, you should be able to:

- Explain the principles of classic bio-inspired algorithms
- Compare and contrast different approaches
- Implement bio-inspired algorithms in Python
- Optimize algorithm parameters and mechanisms to a problem
- Test and validate different approaches on a benchmark problem

General Information

- **Theme:** Classic AI algorithms
- **Language:** English
- **Schedule:** 12h CM + 6h TD + 12h TP
  - Thursday mornings
  - Some TP on Monday afternoon (remote)

TD Sessions

- Exercise sheets on concepts and theory
- Direct preparation for the exam
- You are encouraged to:
  - Work in groups of 2 or 3
  - Write your own individual answers
- Don't forget paper and pen!
- You can submit your copies for feedback (at any time)

TP Sessions

TPs:

- Practical part of the course – Programming, testing, ...
  - Introductory exercises and projects
- Main place to get help on the projects
- Programming in Python (numpy, matplotlib)

Projects:

- 2 projects in groups of 3
- Exploration of the topics of the course
- Focused on adapting the approaches to a specific problem
- Progress checking in TPs

ChatGPT?

I strongly recommend to:

- not use AI to write code
- if you have any questions, ask
- use AI only as a research tool
    - understand an error, find a function, ...

Studies start to show that AI hinders skill development

- And leads to problems with retaining knowledge

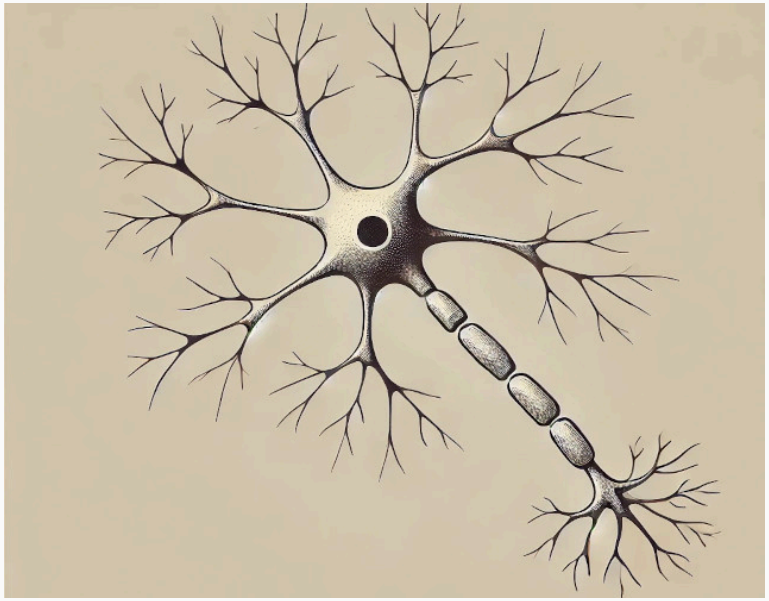
Evaluation

- 60% Exam + 20% Project I + 20% Project II
- Exam: 2h, theoretical aspects + problem solving
  - Open-ended questions, A4 cheat sheet
- Projects: groups of 3, code + report
  - Individual written evaluation during TP / exam

Outline

Bio-inspired Algorithms

Algorithms in Nature



Algorithms in Nature

Motivation: learn from nature!

- Adapt the ideas we see around us
- Trend started in the 80s with nice results
- We look at three natural phenomena:

  Evolution Swarm intelligence Physics

Goal: Take these ideas into our toolbox

- More so than just learning the algorithms

Bio-inspired Algorithms

Computational Problems

Type of problems

Type of problems: Optimization

- Input can be anything: graph, string, ...
  - Describes a set of feasible solutions
- Input also includes an objective function
  - Quantifies the quality of each solution
  - Often called the fitness of the solution
- Goal: find solution that optimizes the objective function
  - optimize = minimize or maximize

Usually NP-hard problems

Quick detour: NP-hardness

NP $\neq$ NP-hard!

P=NP?

A problem in NP: we can verify* that a solution is feasible

An NP-hard problem: at least as hard as any problem in NP.

- If we can solve* an NP-hard problem, we can solve any in NP.
- Cook: SAT is NP-hard
- Karp: A problem is NP-hard if it is at least as hard as SAT
  - SAT $\leq$ Clique $\leq$ Vertex Cover $\leq \ldots$

NP-Complete = NP + NP-hard

*in polynomial time

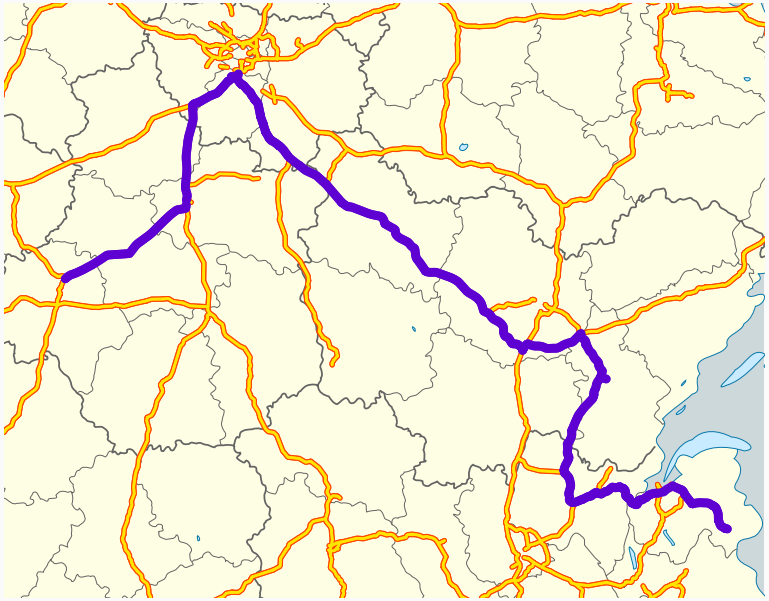
Type of problems

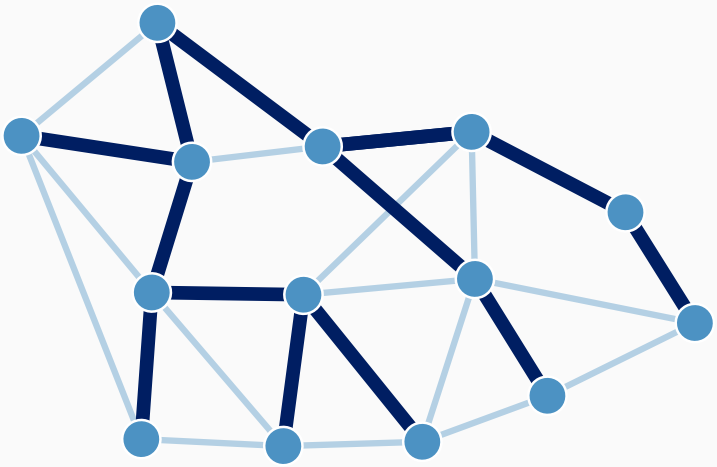
Type of problems: Optimization

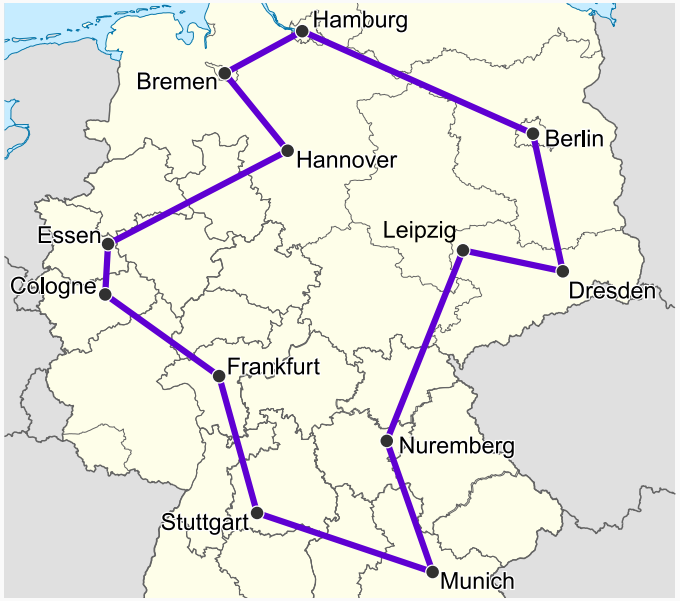
- Input can be anything: graph, string, ...
  - Describes a set of feasible solutions
- Input also includes an objective function
  - Quantifies the quality of each solution
  - Often called the fitness of the solution
- Goal: find solution that optimizes the objective function
  - optimize = minimize or maximize

Usually NP-hard problems

Type of problems – Examples

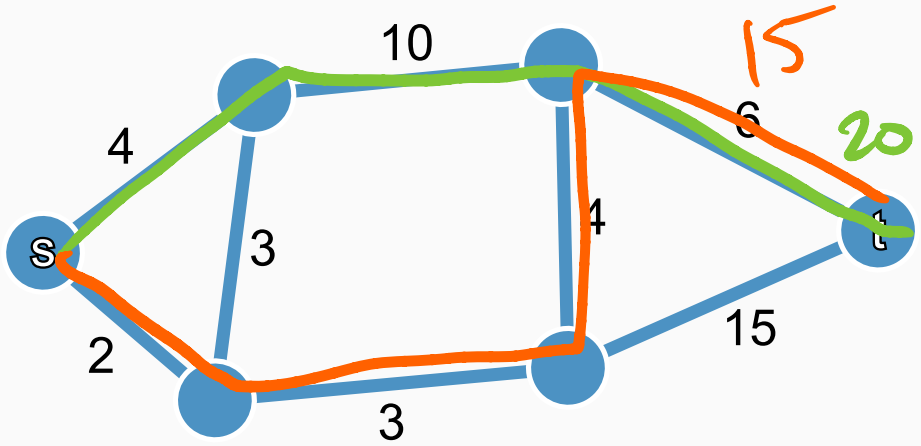






Example: Shortest path

- **Input:** a graph and two vertices s, t
  - A feasible solution is a path from s to t in the graph
- **Objective:** the sum of lengths of the edges on a path
- **Goal:** find the shortest s-t-path



Bio-inspired Algorithms

Approaches

Type of approaches

Type of approaches: heuristics

- Algorithms that solve problems in a practical way
  - Usually fast, compared to other approaches
- Use knowledge about the problem and data
  - Both theoretical and empirical
- Solutions are usually not optimal
  - Rather a solution that is considered good enough

Related approach: approximation algorithms

- which have theoretical guarantees

Type of approaches

All of the algorithms we study are meta-heuristics.

- A heuristic that describes heuristics.
- General approaches that can be applied to many situations
  - Describes the main ideas of the approach
  - Not strictly an algorithm as it is not clear for every problem
- Made clear and unambiguous using problem knowledge

↳ algorithm (heuristic)

Types of approaches

Approaches can:

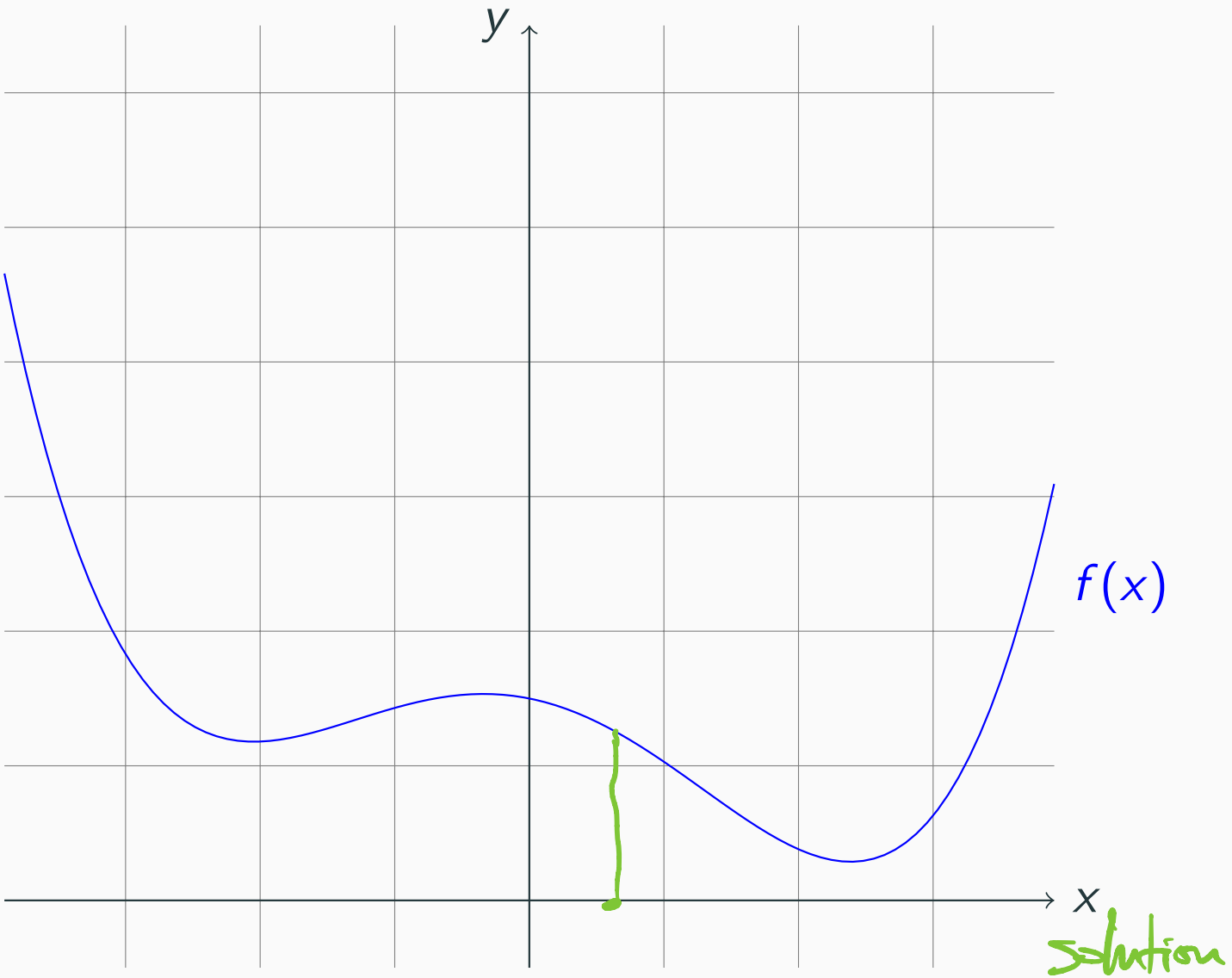
- be deterministic or stochastic (random)
- use a single solution or a population
- search locally or globally

Simple Heuristics

Simple Heuristics

Random Search

Example



Random Search

Algorithm *RandomSearchFunc*

Input: n, domain D, function $f : D \to \mathbb{R}$

Output: solution $x \in D$

```
1: x := RandomSolution(D)
2: for i = 1 to n − 1 do
3:     c ← RandomSolution(D)
4:     if f(c) < f(x) then
5:         x := c
6:     end if
7: end for
8: return x
```

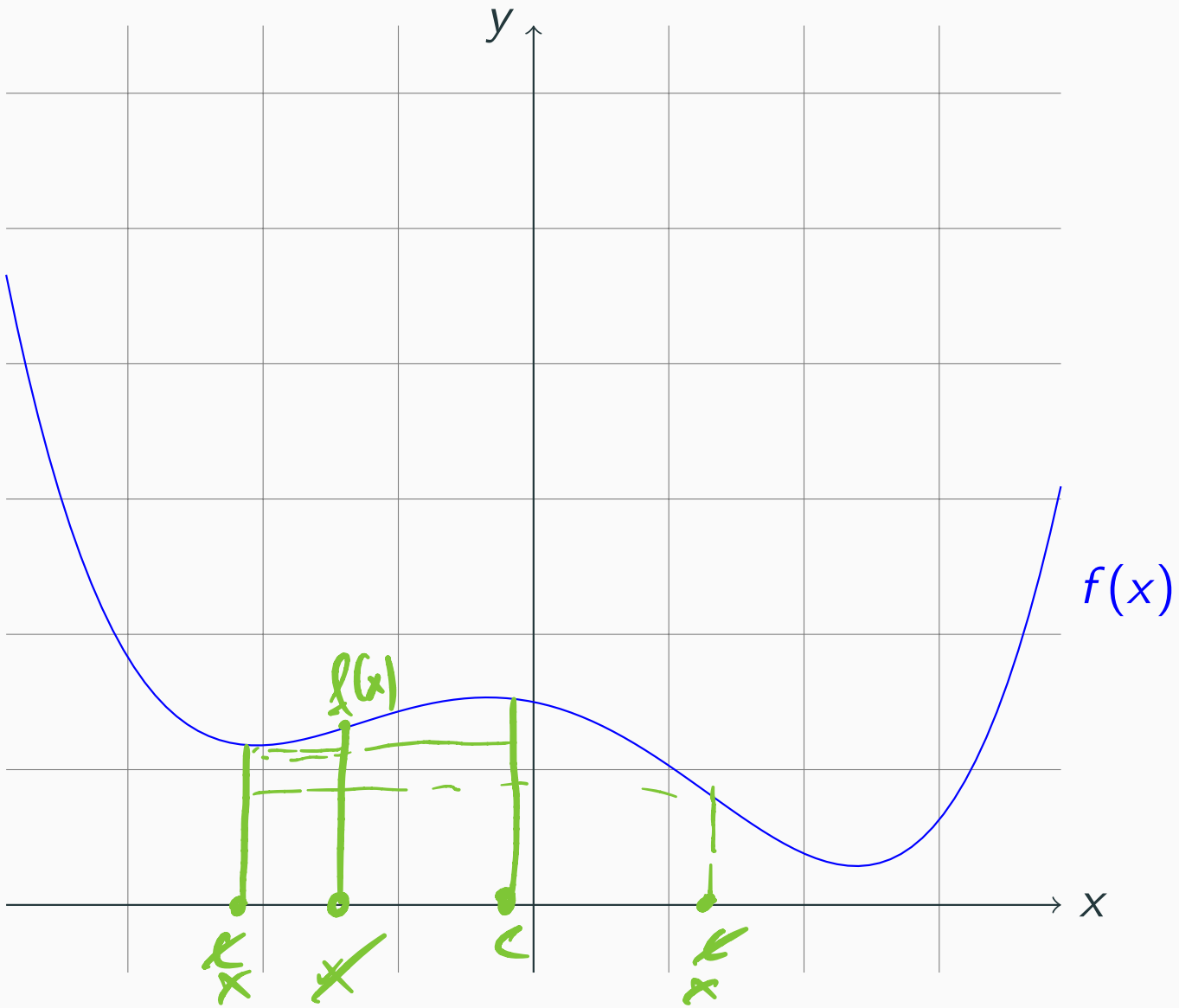
(take x unif. at random from D = [a, b])

Random Search

```
1 def random_search_func(n, D, f):
2     x = D.sample()
3     for i in range(1,n):
4         c = D.sample()
5         if f(c) < f(x):
6             x = c
7     return x
8
9 D = DomainR1(-3, 3)
10 f = lambda x: x**2 + x
11 x = random_search_func(10, D, f)
12 print(f"f({x:.3}) = {f(x):.3}")
13 #   f(-0.315) = -0.216
```

Links: lambda functions, f-strings

Example



Random Search

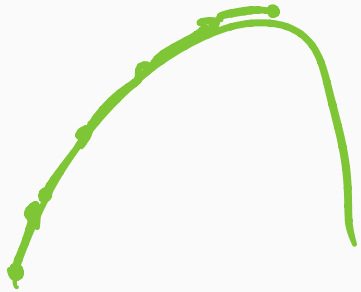
Characteristics:

- single, stochastic, global
- Not very efficient
    - Each step goes in a completely different direction

What if we stay around the previous solution?

Simple Heuristics

Hill Climbing



Hill Climbing

Algorithm HillClimbingFunc

Input: n, domain D, function $f : D \to \mathbb{R}$

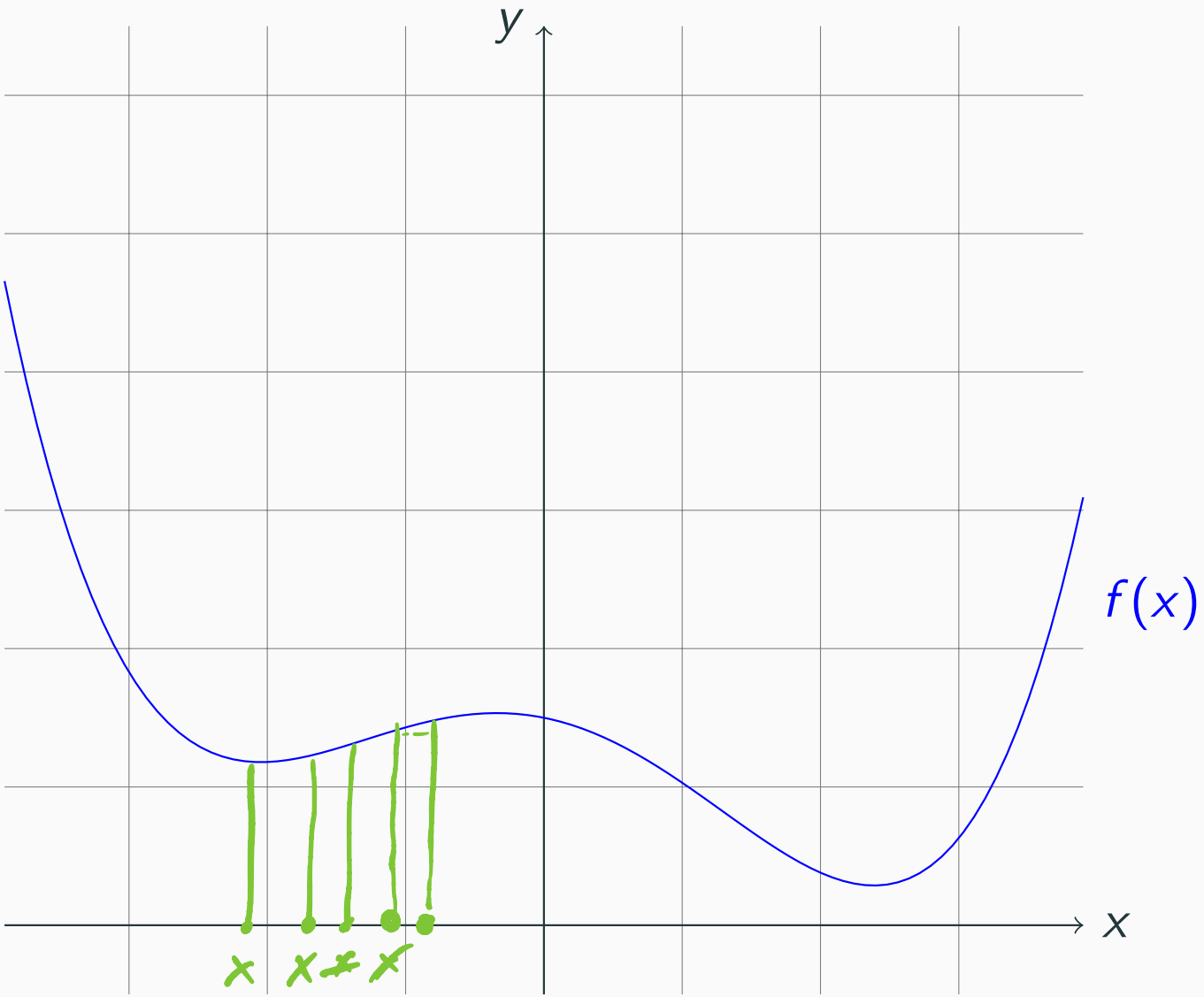
Output: solution $x \in D$

```
1: x := RandomSolution(D)
2: for i = 1 to n − 1 do
3:     c ← RandomNeighbor(x)
4:     if f(c) < f(x) then
5:         x := c
6:     end if
7: end for
8: return x
```

Hill Climbing

```
1 def hill_climbing_func(n, D, f):
2     x = D.sample()
3     for i in range(1,n):
4         c = D.neighbor(x)
5         if f(c) < f(x):
6             x = c
7     return x
8
9 D = DomainR1(-3, 3)
10 f = lambda x: x**2 + x
11 x = hill_climbing_func(10, D, f)
12 print(f"f({x:.3}) = {f(x):.3}")
13 #   f(-0.909) = -0.083
```

Example



Hill Climbing

Characteristics:

- single, stochastic, local
- Can approach the solution faster
- Really dependent on starting point

Another Example: Guessing Game

Problem: secret binary vector

- There is a vector $\{0,1\}^n$ which we do not know
- Our goal is to guess it
- We have a function measuring similarity to the secret:
  - How many bits a solution has in common with the secret

How to use hill climbing:

- A solution is a vector $\{0,1\}^n$
- A neighbor of x is x with a flipped bit
- The fitness is the similarity

Secret Binary Vector

```
@dataclass
class DomainBinVec:
    n: int
    def sample(self):
        return tuple(random.randrange(2) \
                     for i in range(self.n))
    def neighbor(self, x):
        i = random.randrange(self.n)
        return x[:i] + (1-x[i],) + x[i+1:]

D = DomainBinVec(n)
f = lambda x: sum(secret[i] != x[i] \
                     for i in range(len(secret)) )
x = hill_climbing_func(20, D, f)
print(f"f(x) = {f(x)}")
```

Links: classes, dataclass, type hints

Analysing the Guessing Game

Random Search:

- How many steps can we expect? $\sim 2^n$
- Most problems have a large search space
  - With many interdependent dimensions
- Random search does not perform well

Hill Climbing:

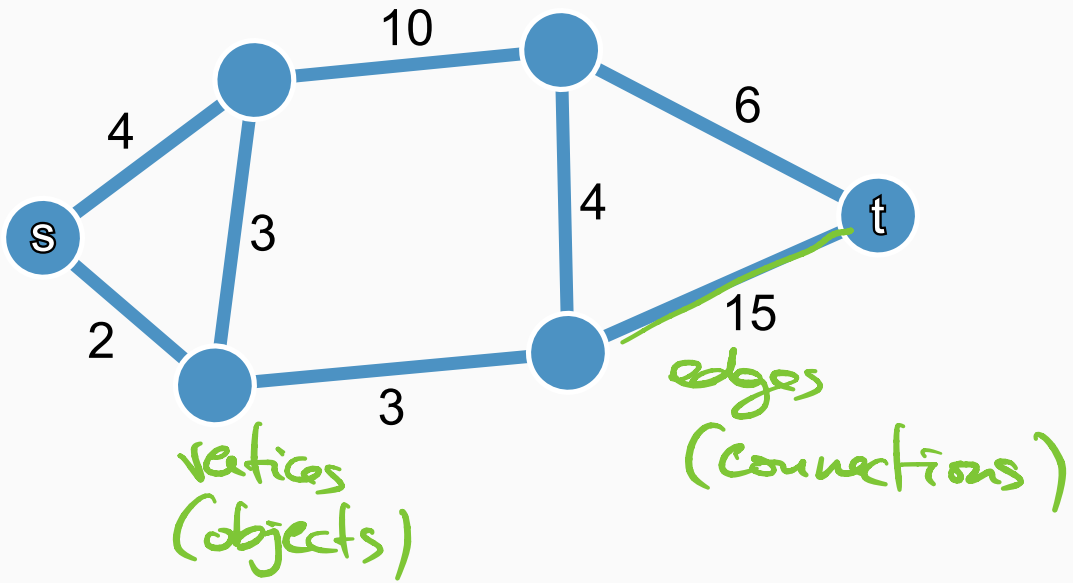
- How many steps can we expect? $\sim n \ln n$
- Operates independently on each dimension
- But bad with local minima

Simple Heuristics

Shortest Path and A*

Example: Shortest path

- **Input:** a graph and two vertices s, t
  - A feasible solution is a path from s to t in the graph
- **Objective:** the sum of lengths of the edges on a path
- **Goal:** find the shortest s-t-path



Dijkstra's Algorithm

Algorithm Dijkstra

Input: graph $G = (V, E)$; edge lengths $\ell\colon E \to \mathbb{R}_{\geq 0}$; vertices s, t

Output: length of the shortest s-t-path

```
1:  X := V                                              // unvisited vertices
2:  d_s(s) = 0; d_s(x) = +∞, for every x ∈ V \ {s}
3:  while t ∈ X do
4:      Let v ∈ X be the vertex with minimum d_s(v)
5:      Remove v from X
6:      for all edge vu ∈ E do
7:          if d_s(v) + ℓ(vu) < d_s(u) then // Update the distance to u
8:              d_s(u) := d_s(v) + ℓ(vu)
9:          end if
10:     end for
11: end while
12: return d_s(t)
```

Dijkstra's Algorithm

Algorithm Dijkstra

Input: graph $G = (V, E)$; edge lengths $\ell\colon E \to \mathbb{R}_{\geq 0}$; vertices s, t

Output: length of the shortest s-t-path

```
1:  X := V                                              // unvisited vertices
2:  d_s(s) = 0; d_s(x) = +∞, for every x ∈ V \ {s}
3:  while t ∈ X do
4:      Let v ∈ X be the vertex with minimum d_s(v)
5:      Remove v from X
6:      for all edge vu ∈ E do
7:          if d_s(v) + ℓ(vu) < d_s(u) then // Update the distance to u
8:              d_s(u) := d_s(v) + ℓ(vu)
9:          end if
10:     end for
11: end while
12: return d_s(t)
```

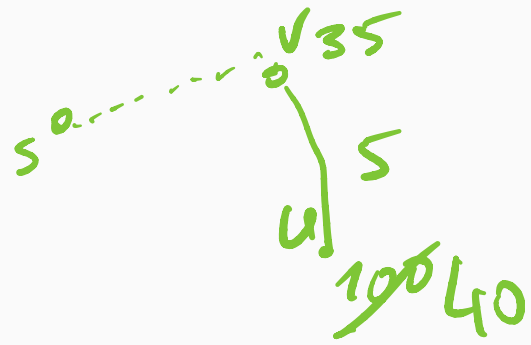
Dijkstra's Algorithm

Algorithm Dijkstra

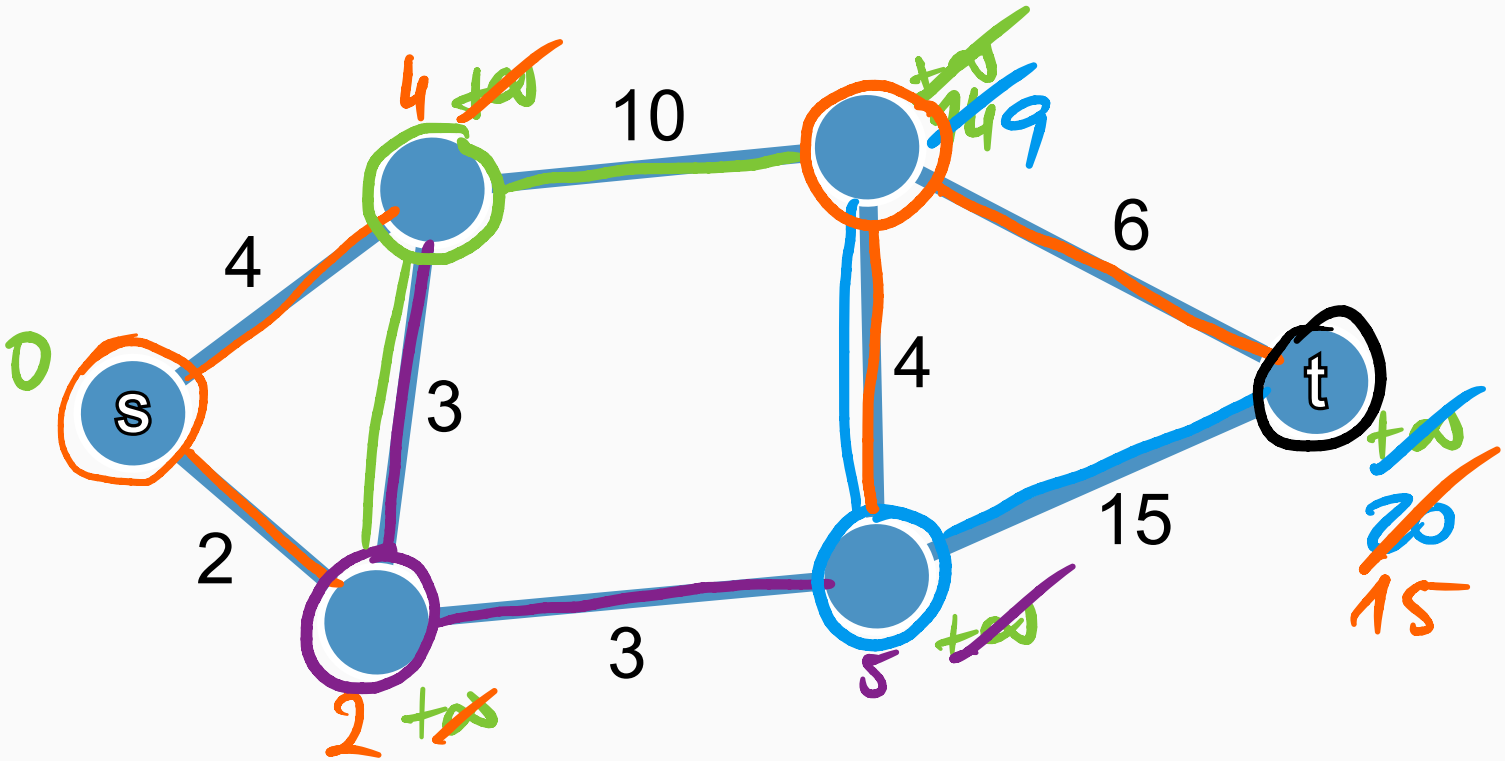
Input: graph $G = (V, E)$; edge lengths $\ell\colon E \to \mathbb{R}_{\geq 0}$; vertices s, t

Output: length of the shortest s-t-path

1: $X := V$ *// unvisited vertices*
2: $d_s(s) = 0$; $d_s(x) = +\infty$, for every $x \in V \setminus \{s\}$
3: **while** $t \in X$ **do**
4: Let $v \in X$ be the vertex with minimum $d_s(v)$
5: Remove v from X
6: **for all** edge $vu \in E$ **do**
7: **if** $d_s(v) + \ell(vu) < d_s(u)$ **then** *// Update the distance to u*
8: $d_s(u) := d_s(v) + \ell(vu)$
9: **end if**
10: **end for**
11: **end while**
12: **return** $d_s(t)$



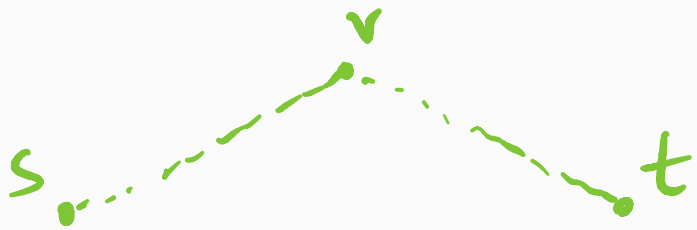
Dijkstra's Algorithm – Example



A* Algorithm

- Similar to Dijkstra
- Uses an estimate to "center" the search
- $h(v)$: estimate of the distance from v to t
  - E.g. straight-line distance in a map
  - Must be at most the actual distance
- Selects the vertex with smallest estimated distance:

$$f(v) = d_s(v) + h(v)$$



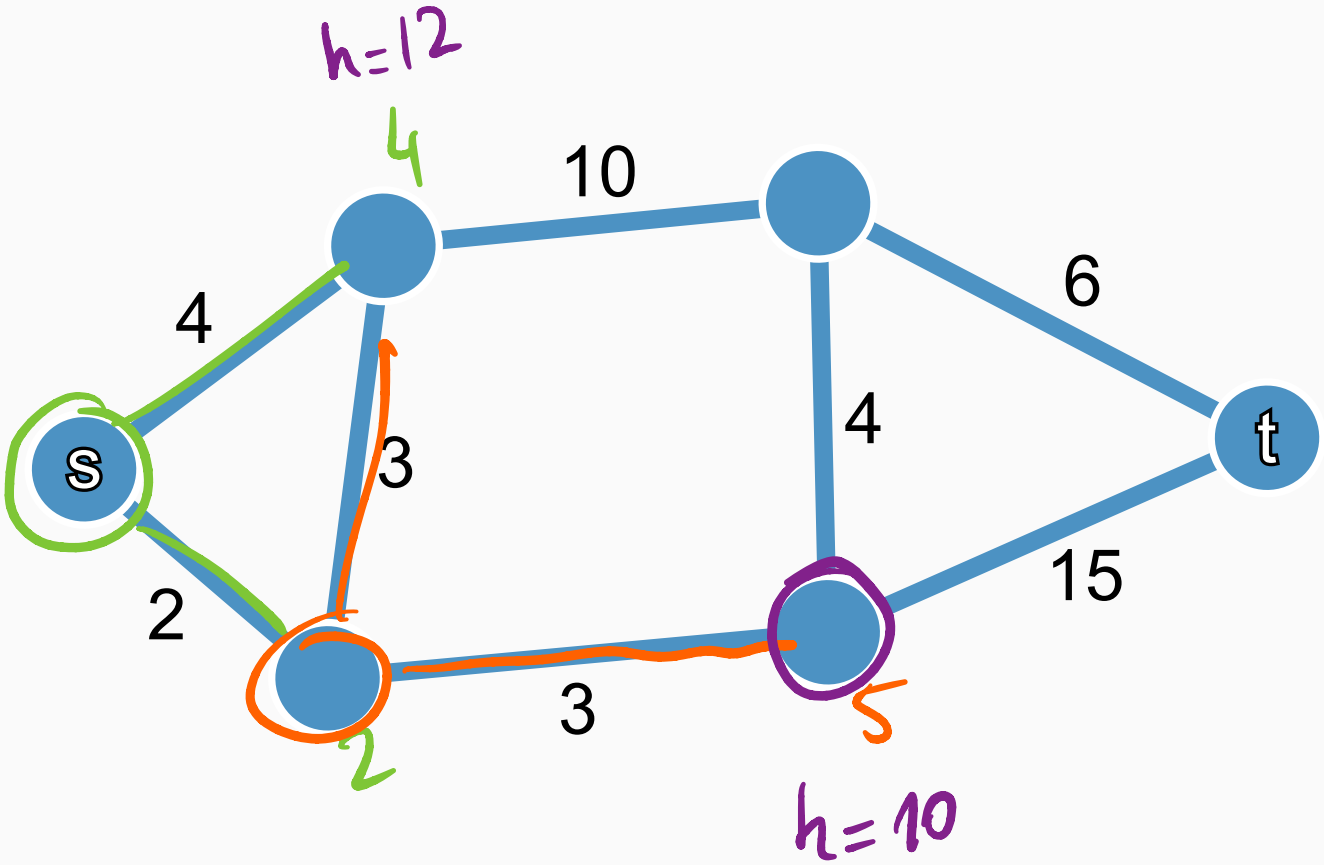
Algorithm AStar

Input: $G = (V, E)$; edge lengths ℓ; $s, t \in V$; $h\colon V \to \mathbb{R}_{\geq 0}$

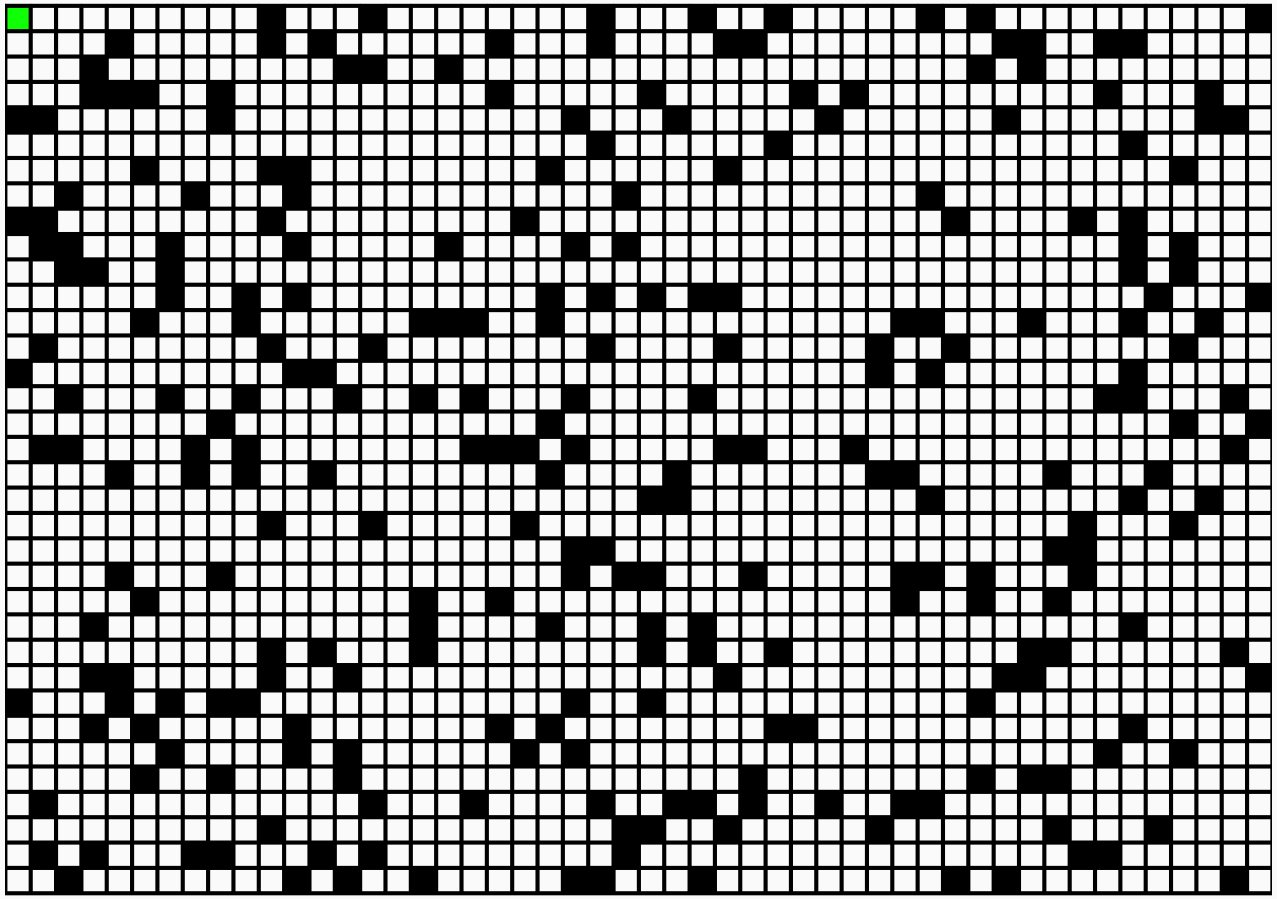
Output: length of the shortest s-t-path

```
1:  X := V                                                  // unvisited vertices
2:  d_s(s) = 0; d_s(x) = +∞, for every x ∈ V \ {s}
3:  while t ∈ X do
4:      Let v ∈ X be the vertex with minimum d_s(v) + h(v)
5:      Remove v from X
6:      for all edge vu ∈ E do
7:          if d_s(v) + ℓ(vu) < d_s(u) then
8:              d_s(u) := d_s(v) + ℓ(vu)
9:              Add u to X
10:         end if
11:     end for
12: end while
13: return d_s(t)
```

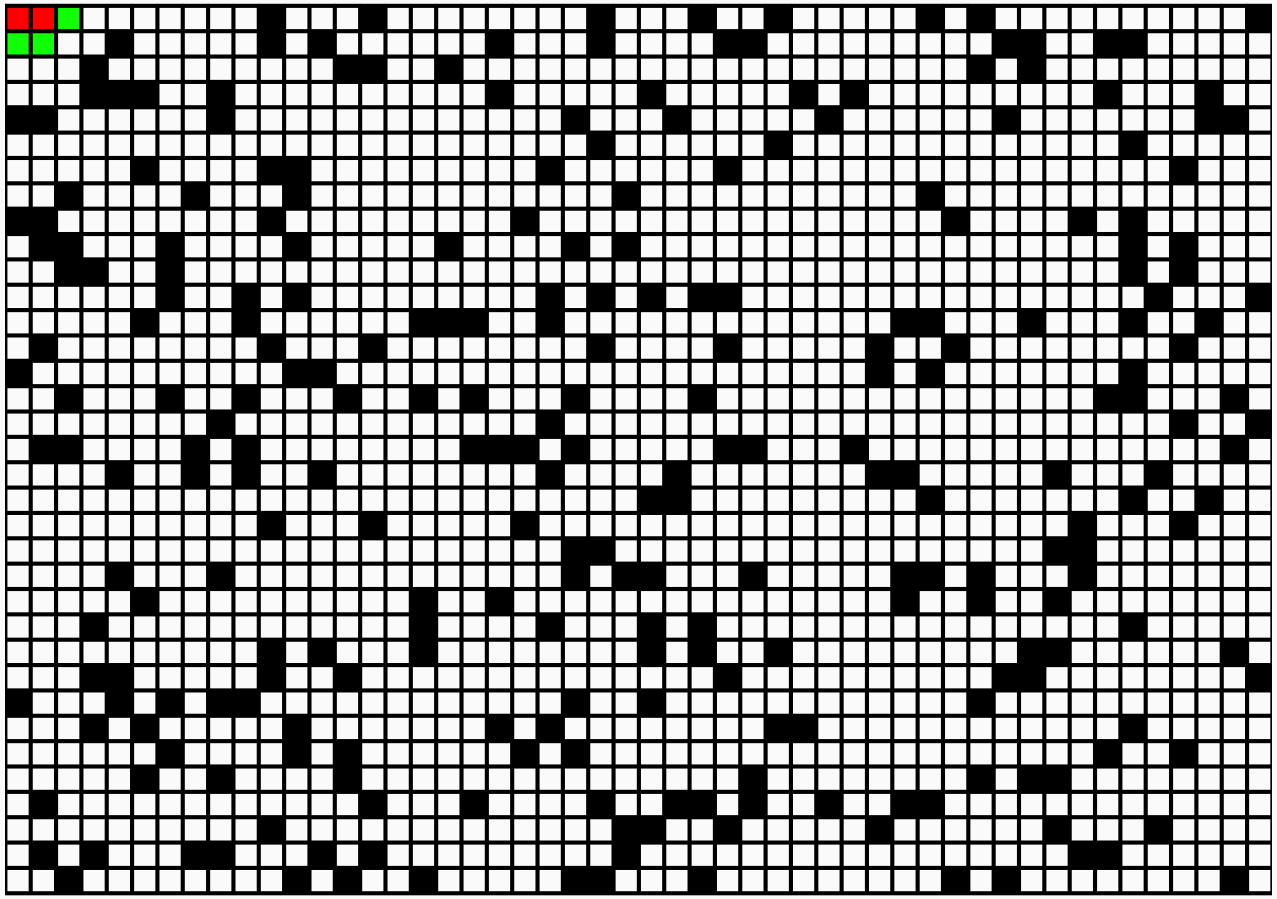
A* Algorithm – Example



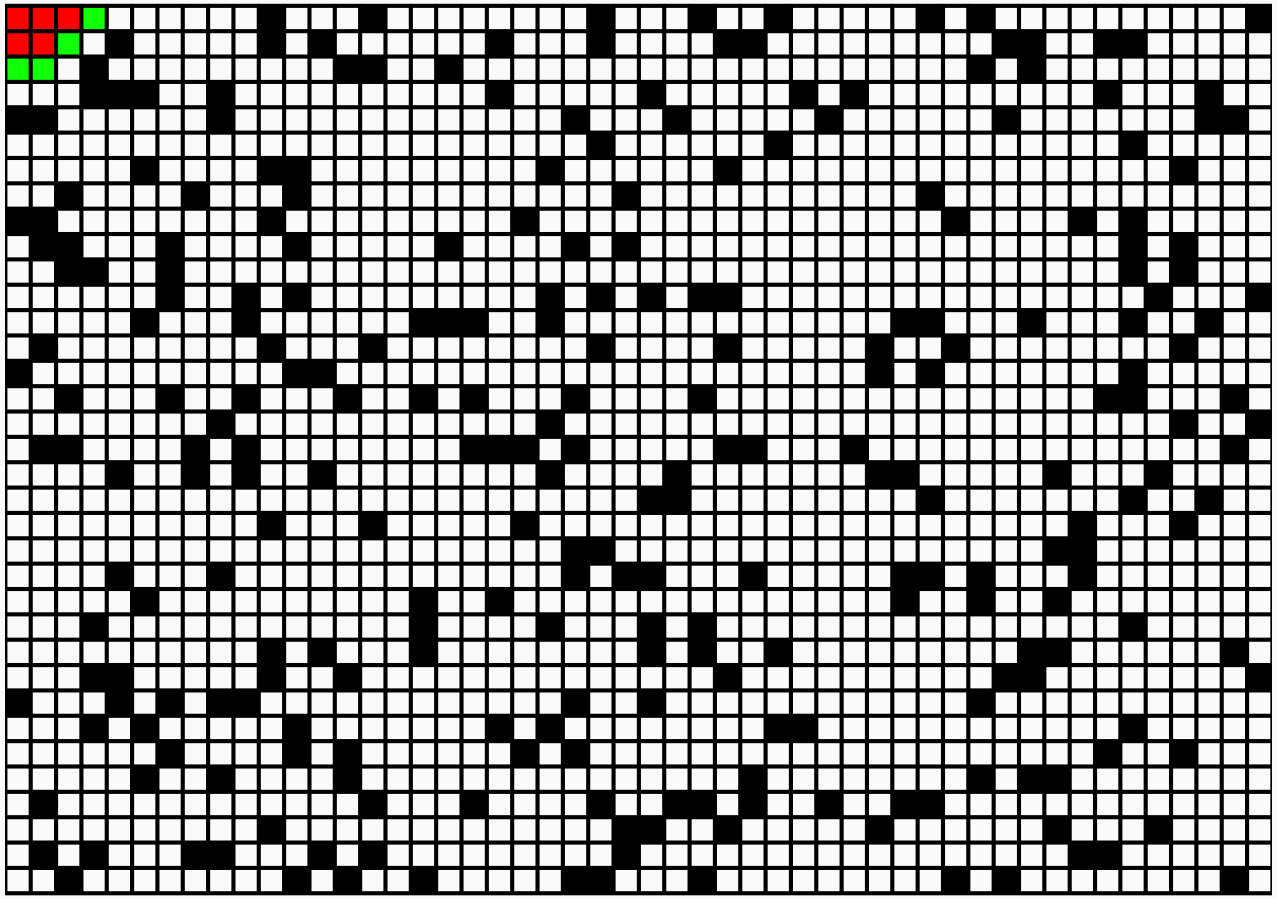
A* Algorithm



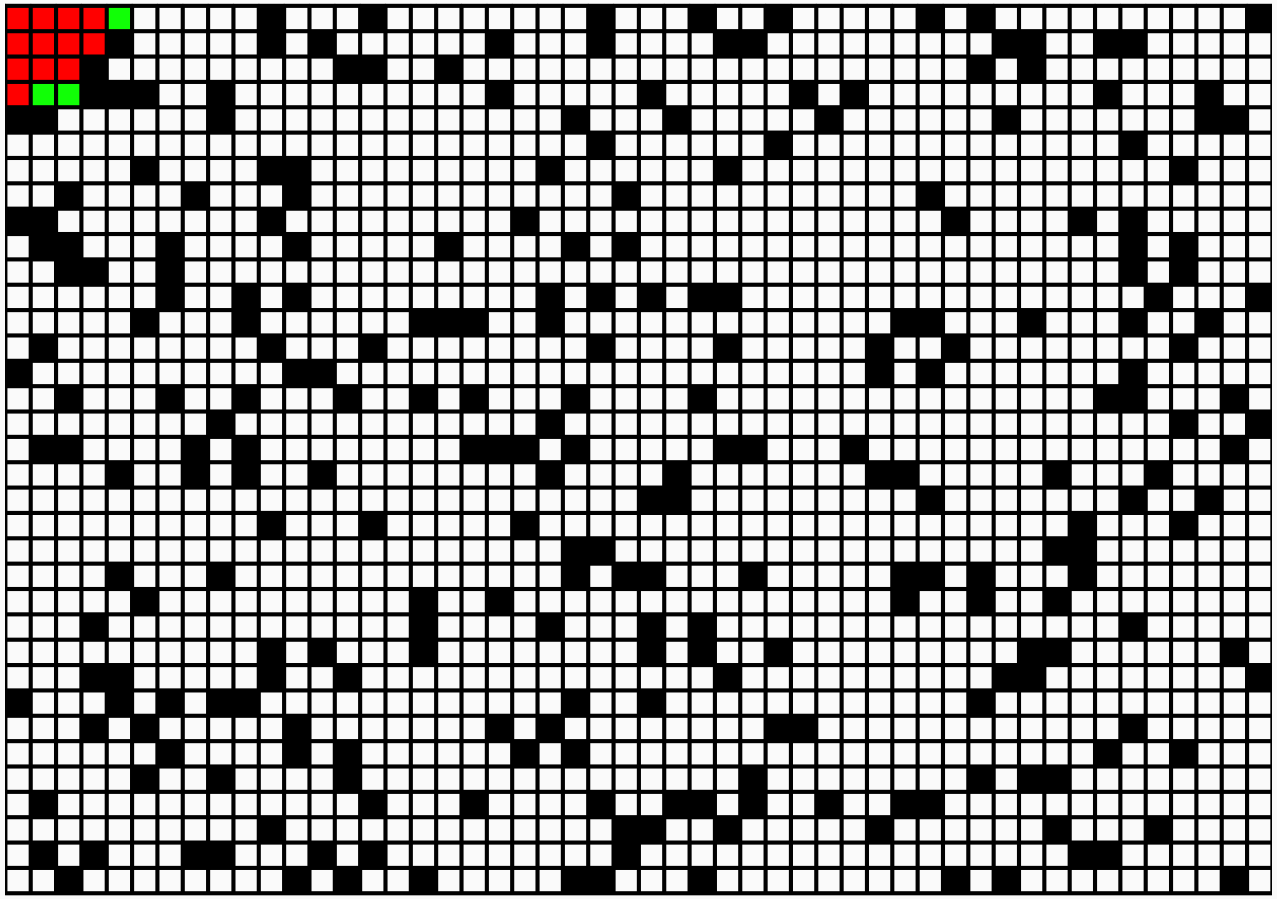
A* Algorithm



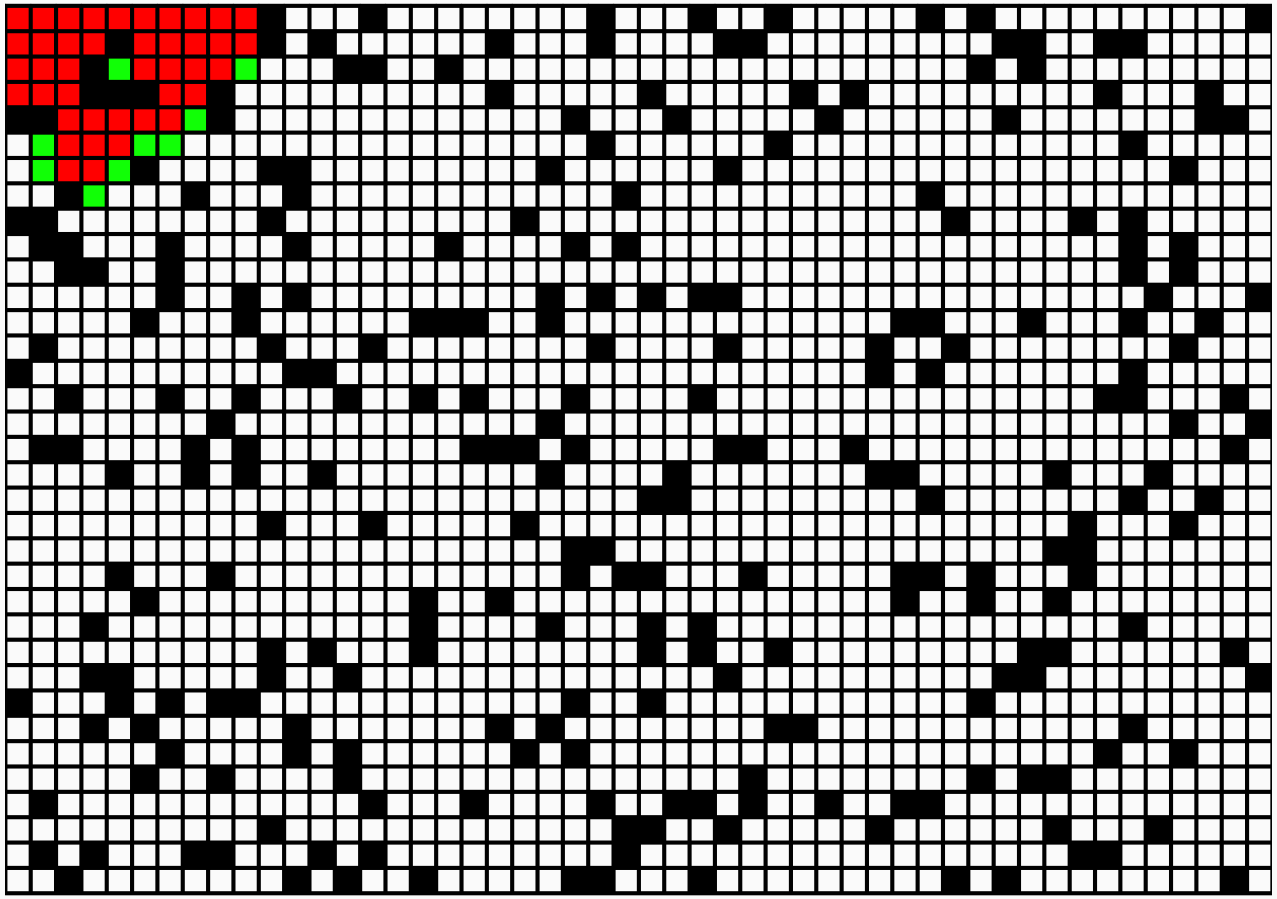
A* Algorithm



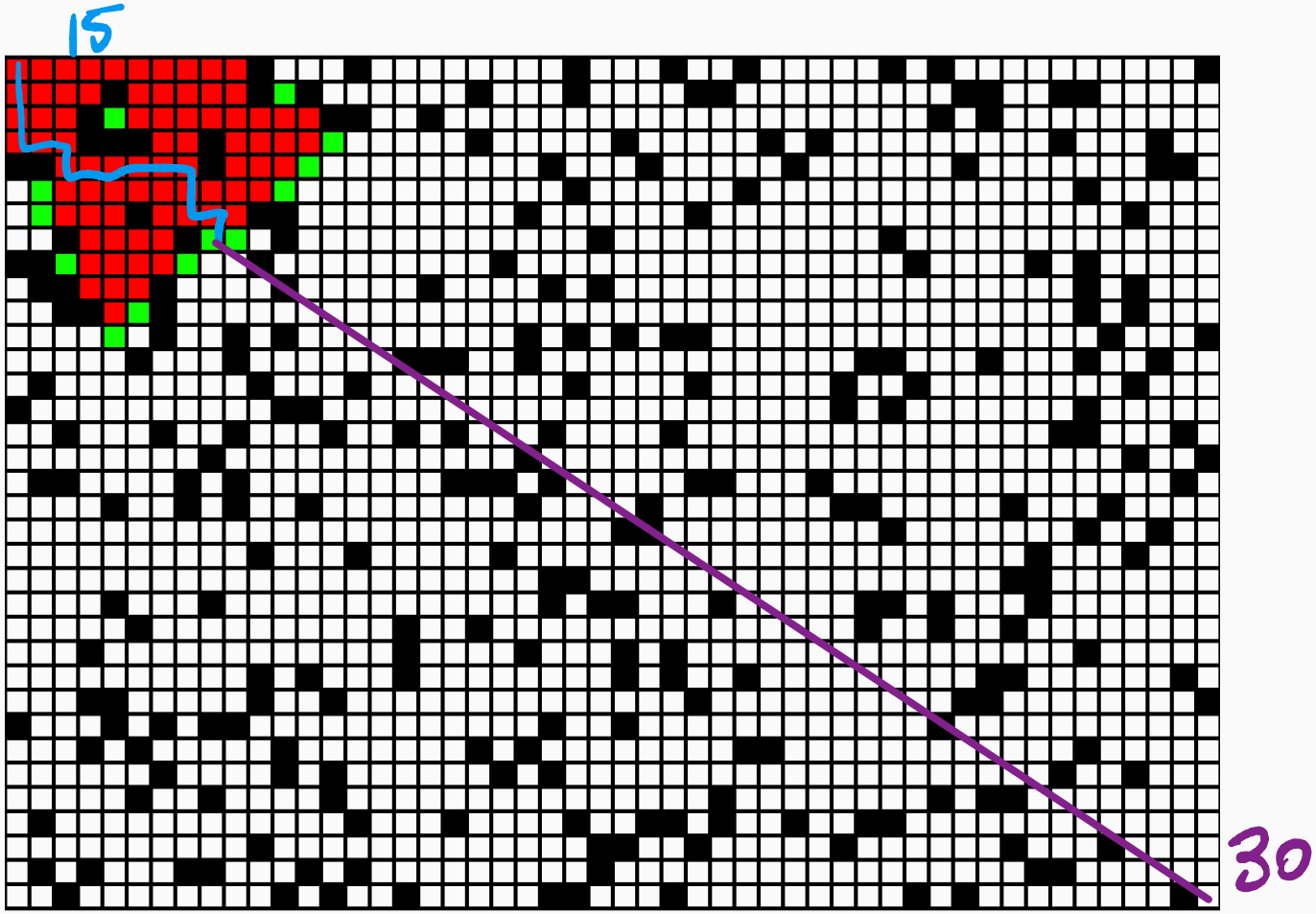
A* Algorithm



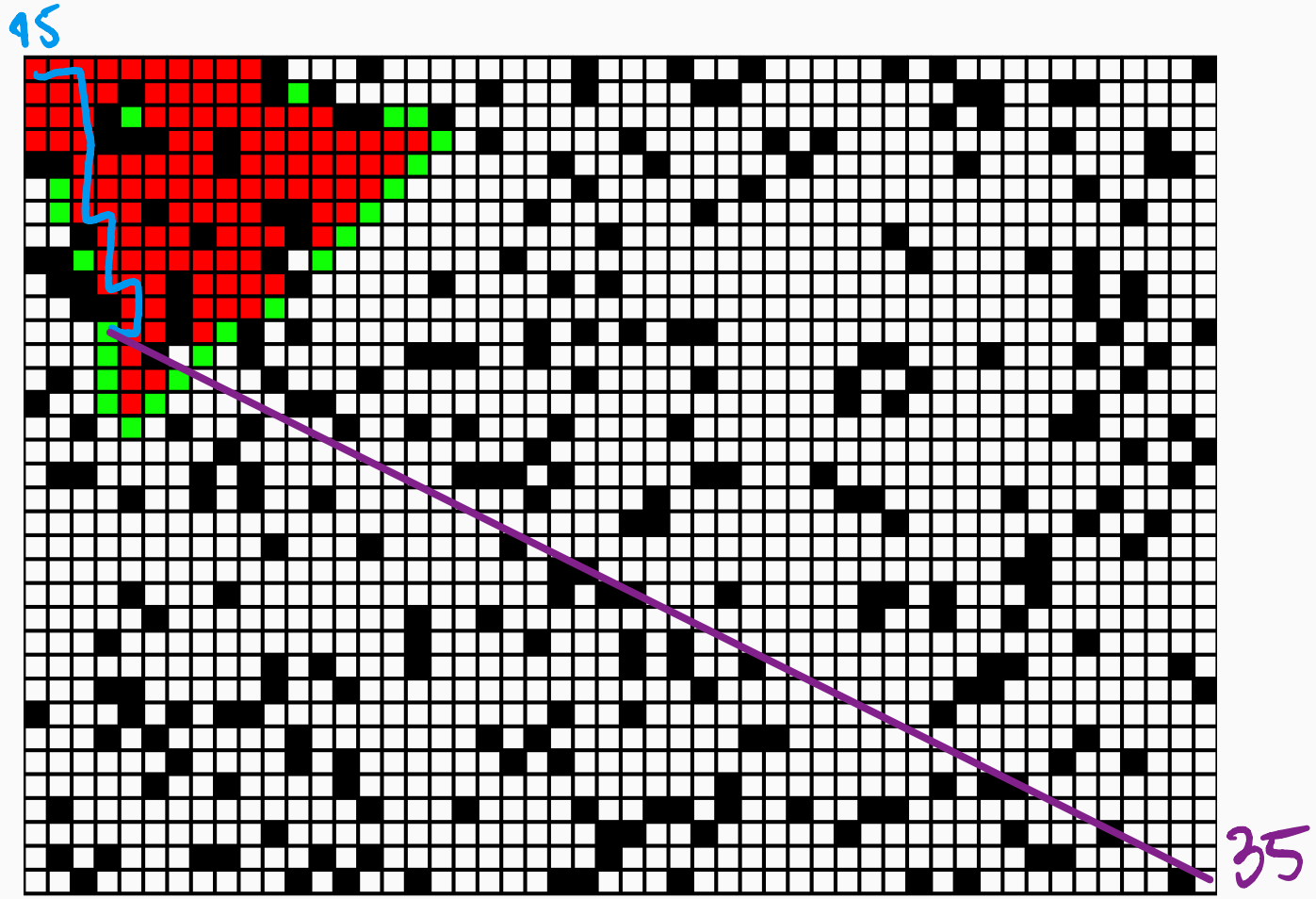
A* Algorithm



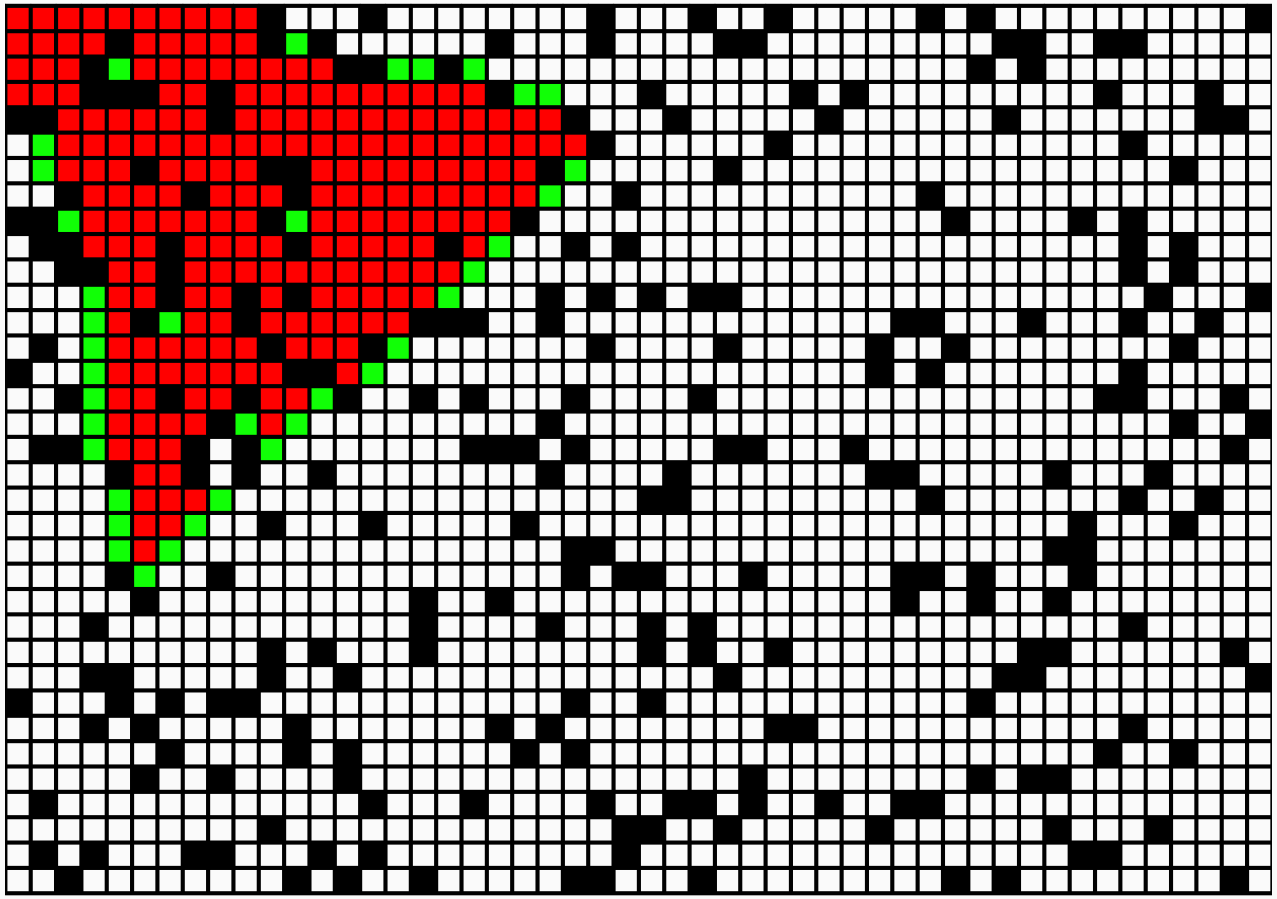
A* Algorithm



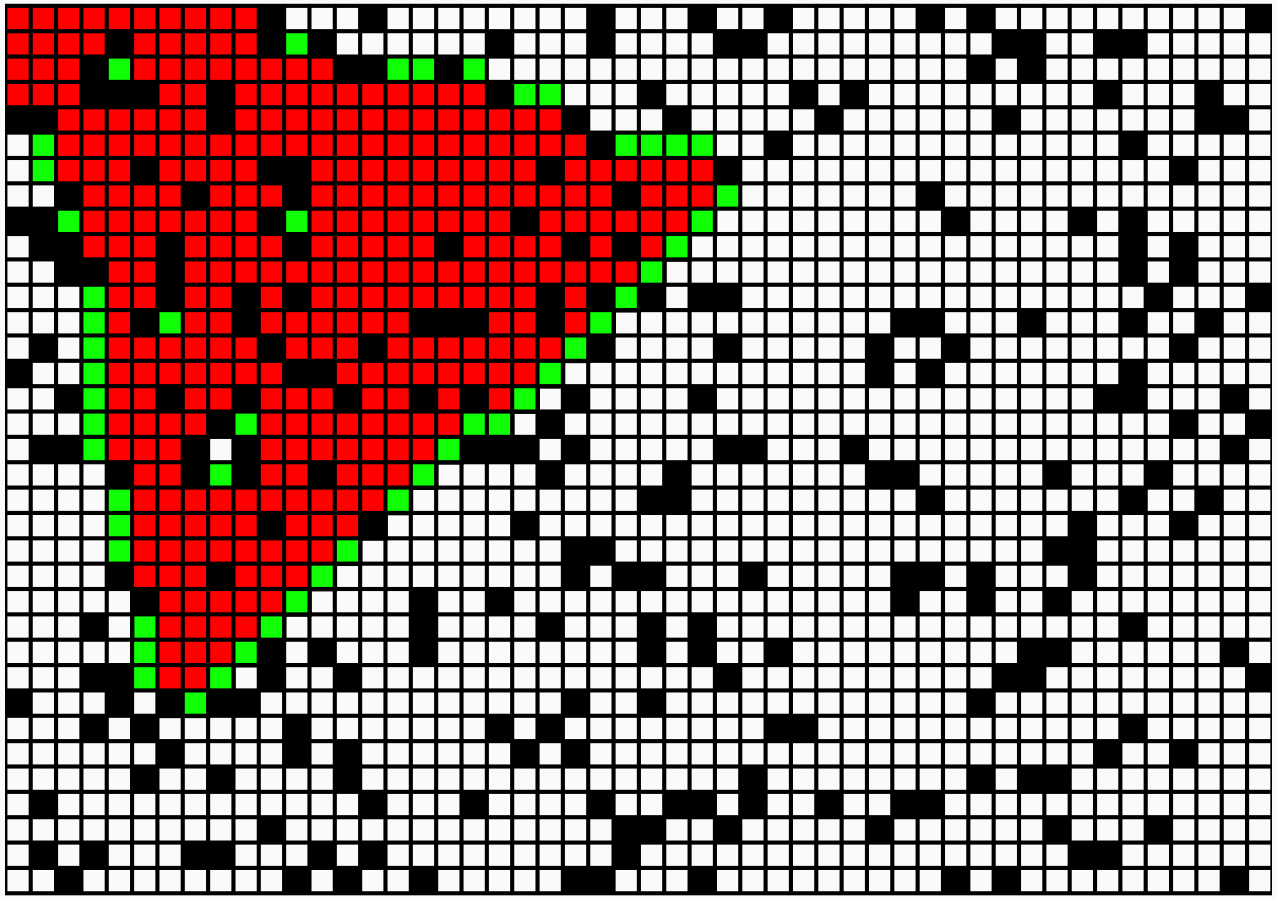
A* Algorithm



A* Algorithm

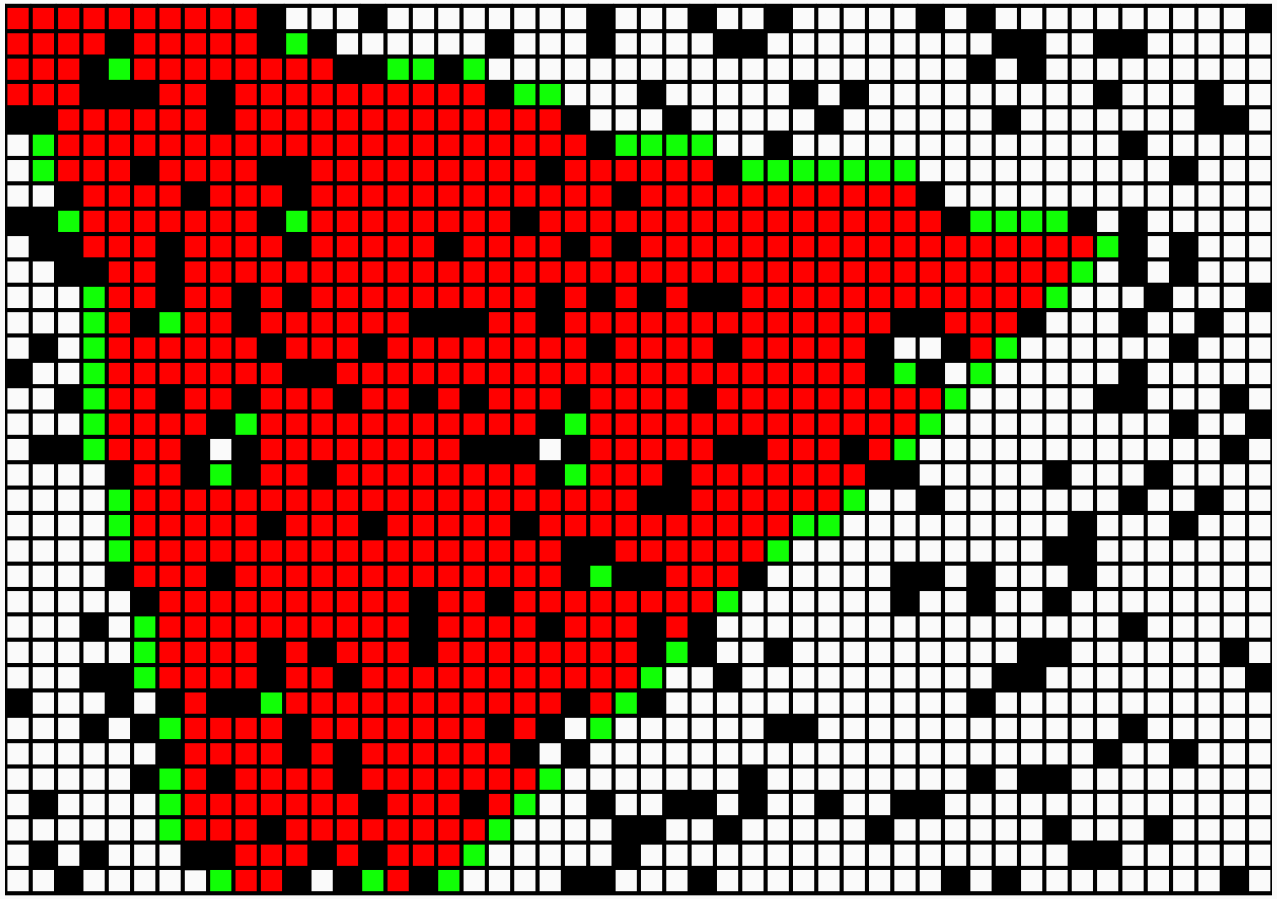


A* Algorithm

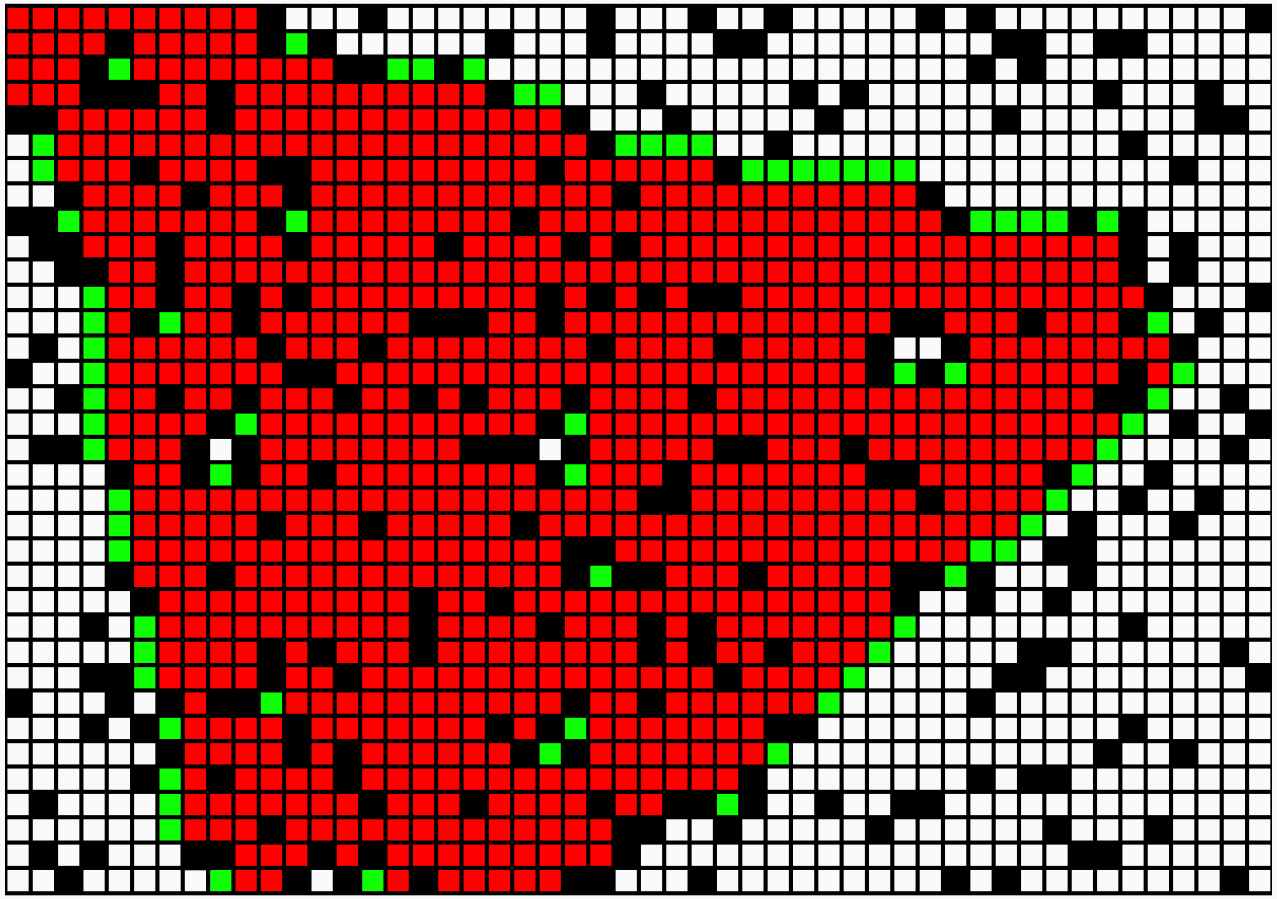


A* Algorithm

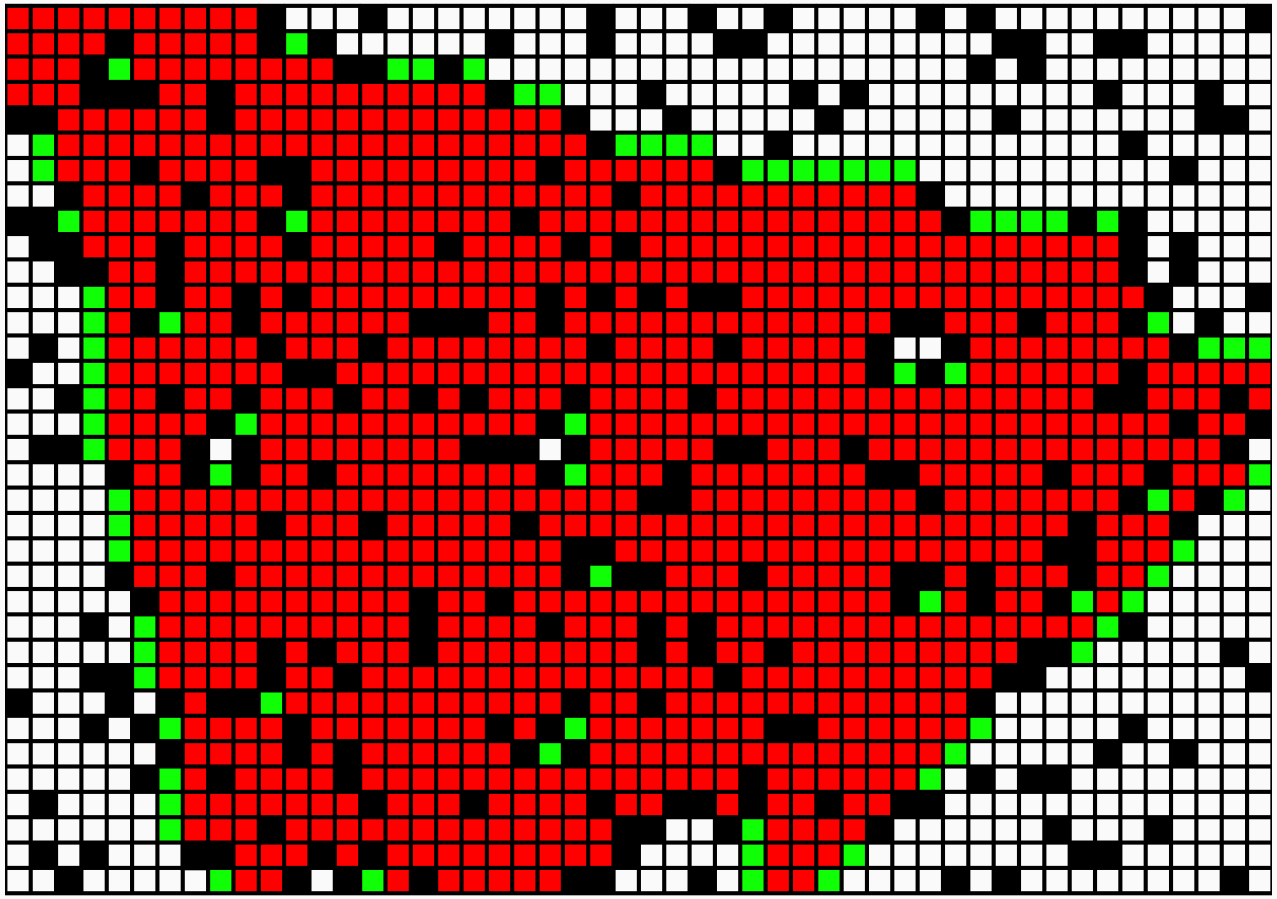
A* Algorithm



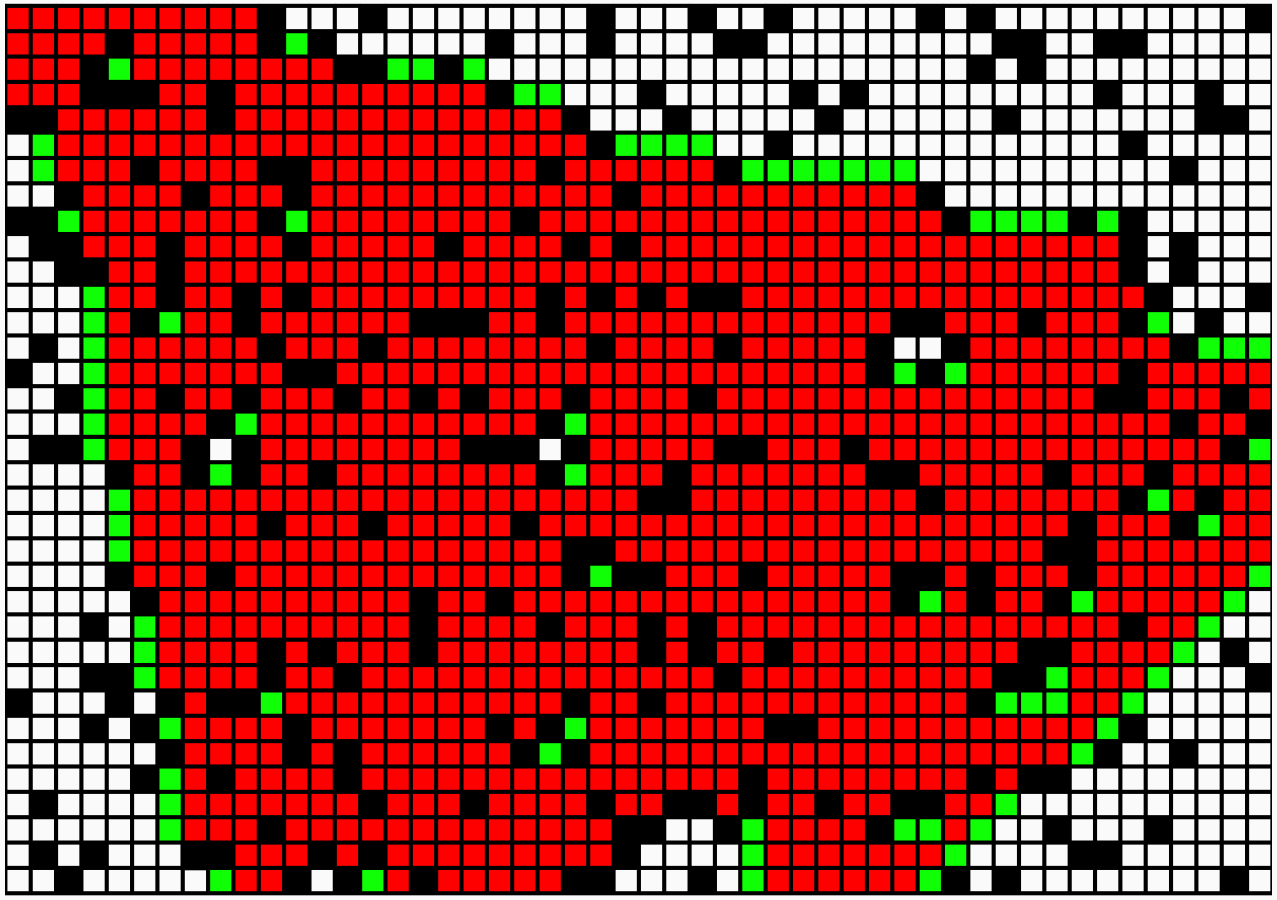
A* Algorithm



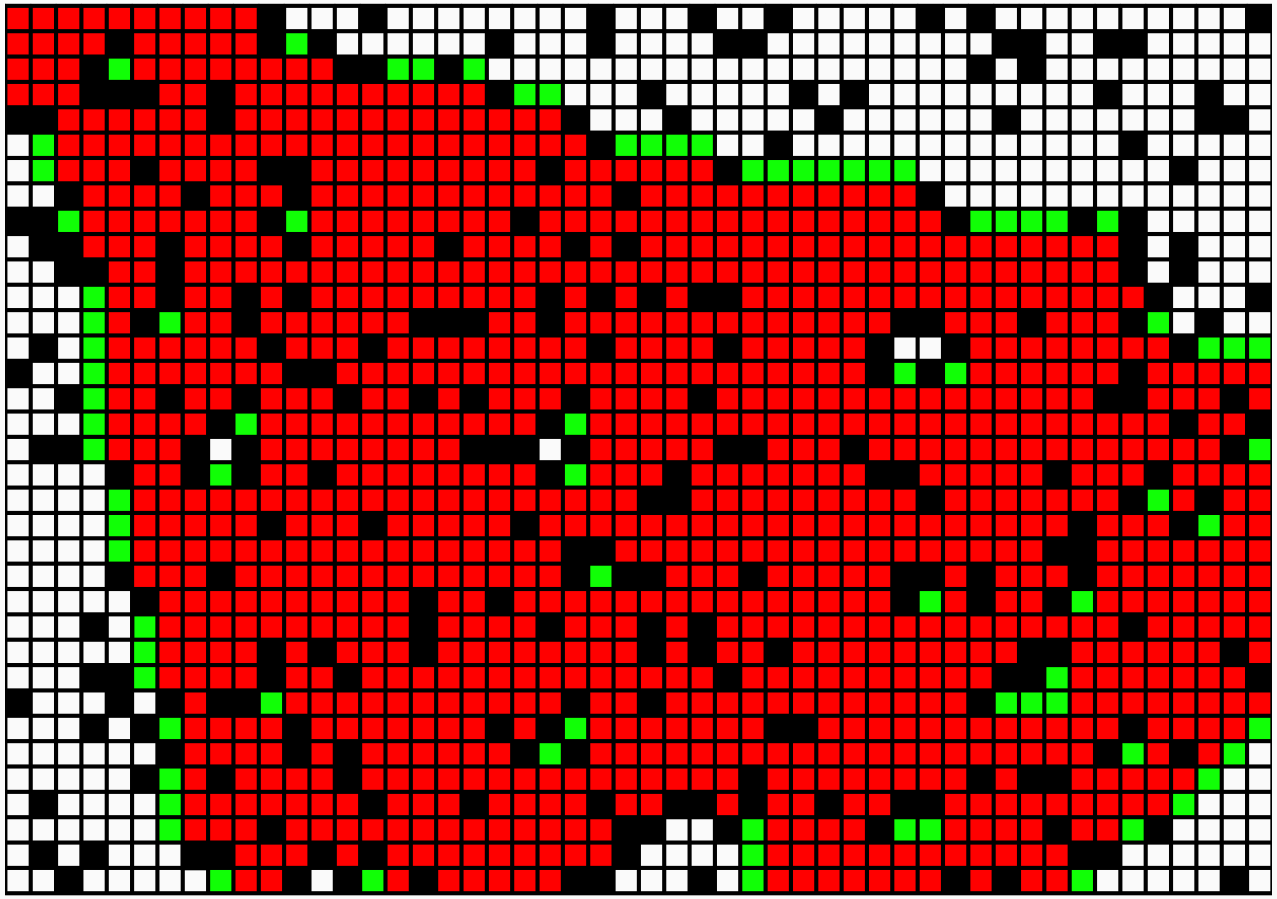
A* Algorithm



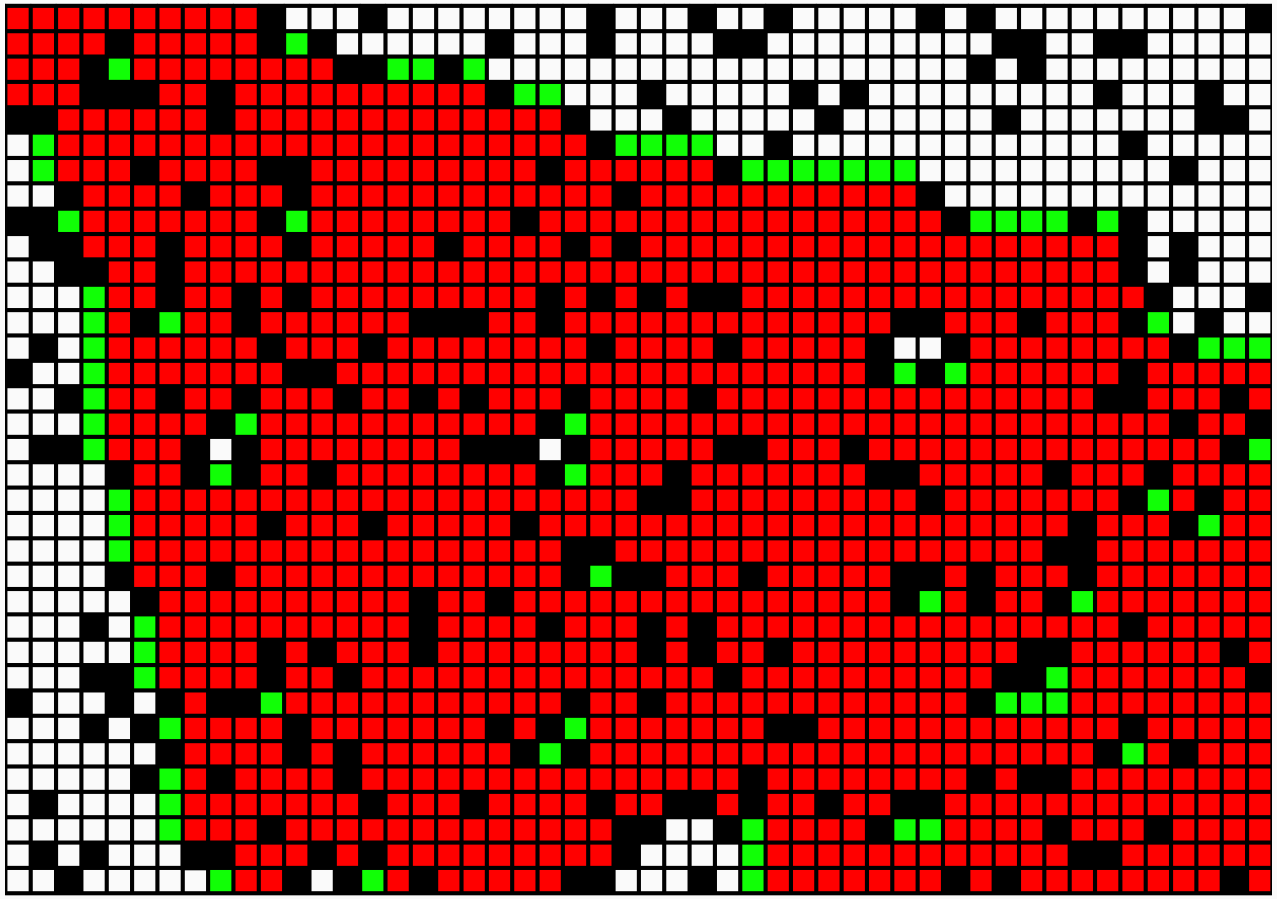
A* Algorithm



A* Algorithm



A* Algorithm



A* Algorithm – Analysis

- Is it optimal?
  - Yes! But why?
- Optimal because we cannot visit t before its predecessors
  - Which is true since $h(v) \leq \text{dist}(v, t)$
  - And so $d_s(v) + h(v) \leq \text{dist}(s, v) + \text{dist}(v, t) = \text{dist}(s, t)$
- However, the running time may be much worse
  - Because we keep re-adding vertices to X
- It is an optimistic algorithm
  - It assumes that the worst cases do not happen

Summary

Bio-inspired algorithms:

- Inspired in nature
- Mostly for optimization problems
  - Find the solution with the best fitness
- NP-hard problems: as hard as any in NP
- We will use heuristic approaches
  - which use practical knowledge

Some simple heuristics:

- Random search: tests random sampled solutions
- Hill climbing: moves to better neighbors
- A*: uses heuristic to center shortest path search