Optimization Techniques

Beniamin Bogosel

Aurel Vlaicu University, Arad

Organization

Labs: Anca Petcut-Lasc

Grading

- Labwork: presence (10%) + activity (40%)
- Final exam (50%)

Contact:

beniamin.bogosel@polytechnique.edu
beniamin.bogosel@uav.ro

What this course is about?

- recalling ideas about discrete optimization
- theoretical aspects in optimization
- algorithms for numerical optimization
- implementation of optimization algorithms

Objectives

After this course you should:

1. know the basic optimization algorithms: gradient descent, Newton, etc.
2. implement optimization algorithms for problems of reasonable size
3. translate the contents of a problem into an optimization algorithm
4. know how to use existing libraries in order to solve particular classes of optimization problems

What is optimization?

$\star$ given an objective function $x \mapsto f(x)$, find the value(s) of x which give the smallest value of f!
$\star$ x may be subjected to some constraints
$\star$ often the minimizer x^* cannot be found explicitly: numerical simulations are needed in this context

$\star$ numerical optimization algorithms produce a sequence (x_n) defined iteratively using the values of f and possibly its derivatives.
$\star$ various questions arise concerning the sequence of iterates:

- the convergence of the sequence (x_n) to a minimizer of f
- the speed of convergence

Basic examples

1. Minimize $\frac{1}{2}x^T Ax - b^T x$ where $A \in \mathcal{M}_{n\times n}$ is symmetric positive definite, $x \in \mathbb{R}^n, b \in \mathbb{R}^n$. Equivalently: solve $Ax = b$ (non-trivial for large A)

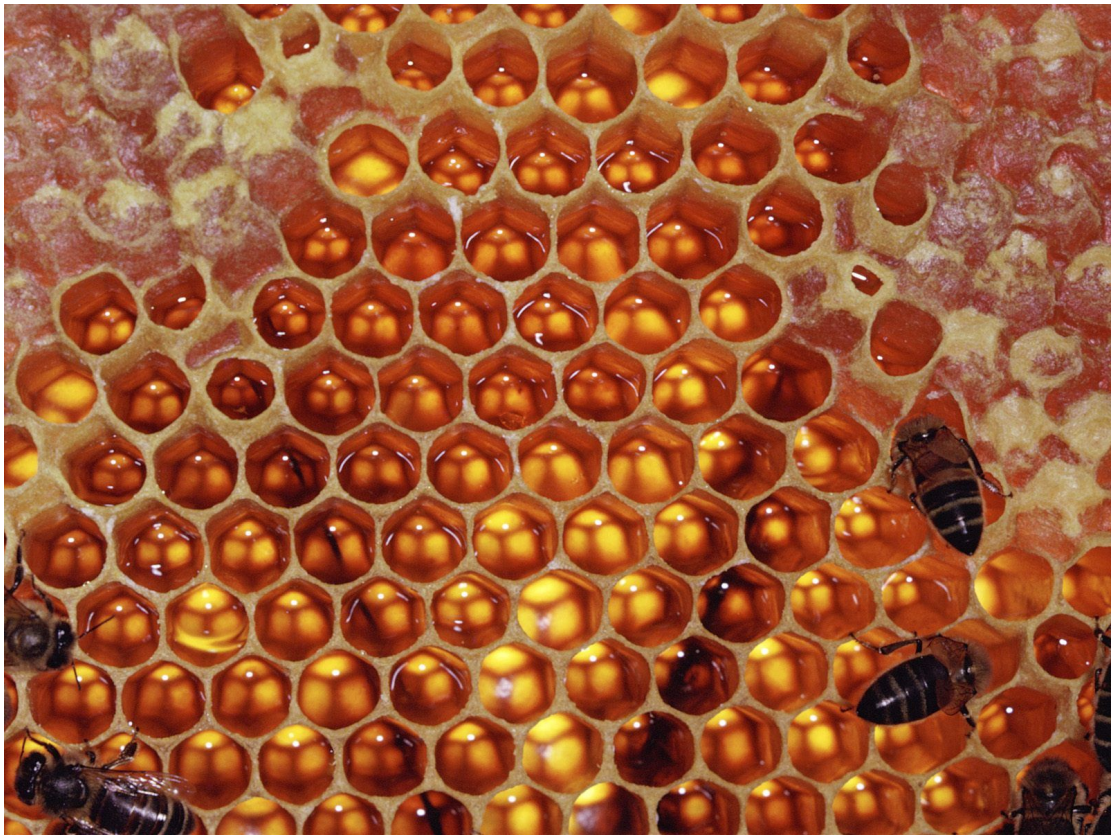
2. Minimize $c \cdot x$ where $c, x \in \mathbb{R}^n$, $x \geq 0$, $Ax \leq b$ (linear programming problem)

3. Model fitting: Given a set of data points (x_i, y_i), $1 \leq i \leq N$ find a function F such that $F(x_i) \approx y_i$.

4. Neural networks: tune machine learning algorithms in order to fit existing data in an optimal way. Use them to make reliable predictions.

Examples in Nature

- Honeycombs are optimal in terms of construction cost (mathematical understanding came only recently: Thomas C. Hales (1999))



$\star$ Laws of nature/physics have optimization built in: light pathways, river path, etc.

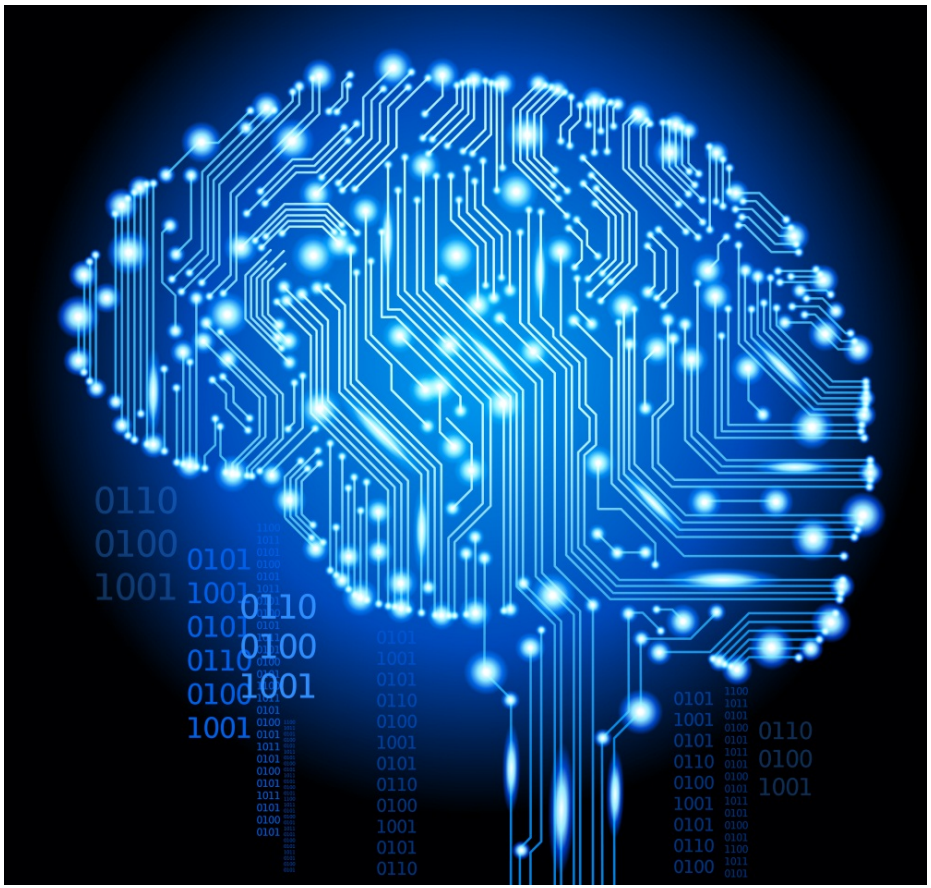
Examples in Nature

- Soap bubbles tend to minimize the surface area while keeping a fixed volume

$\star$ Laws of nature/physics have optimization built in: light pathways, river path, etc.

Applications

- finance, deep learning: process existing information in order to take the best decisions (photo `rostigrabench.ch`)



Applications in industry

- Optimal design of structures: reduce the weight while maintaining the desired mechanical properties

Dilemma: how much optimization is too much?

$\star$ ethics: optimizing your gains might increase dramatically someone else's losses

$\star$ climate change: some climate change models, based on the maximization of the global "welfare" in the world, state that it is "optimal" to have more than 3 degrees Celsius of rise in temperature.

$\star$ Ivar Ekeland: *Quand l'optimisation nuit* (when optimization is detrimental).
`http://math.univ-lyon1.fr/~gentil/interview-Ekeland.pdf`

Motivation for numerical optimization

- for practical applications, optimization algorithms are used: analytic solutions are not available for complex optimization problems
- the user should formulate an optimization problem starting from the given data or models
- once a function which associates a real value to a certain set of parameters is known, optimization algorithms can be used to search for the minimum
- the methods of optimization are vast
  - gradient-free vs gradient based methods
  - higher order methods (Newton)
- the choice of the method depends on the objective function: unimodal functions (nice), highly oscillating functions, non-smooth functions, etc.
- often some constraints need to be enforced, which complicate the theoretical and numerical aspects of optimization problems

Objective of the course

$\star$ present the theory and practice of the basic optimization algorithms
$\star$ underline possible advantages and pitfalls of the optimization algorithms studied: there is no universal algorithm!

Contents of the course

1. Discrete Optimization: particular algorithms
2. Optimization in dimension 1
   - Methods of order zero (without derivatives)
   - Methods of order one and two (using derivatives)
3. Optimization in higher dimensions
   - Gradient descent methods
   - Stochastic gradient
   - Newton methods
   - quasi-Newton methods
4. Constrained optimization
   - Lagrange multipliers
   - a quick glimpse of linear programming (emphasis on practical issues)
5. Genetic algorithms

Discrete Optimization

General optimization problem

In the following: $\mathcal{A}$ is a non-void set, J is a real function defined on $\mathcal{A}$.

Canonical formulation

Let $J : \mathcal{A} \to \mathbb{R}$ be a real function. We wish to solve the problem

$$\min_{x \in \mathcal{A}} J(x)$$

Question: what about maximization problems?

Remark: Note that maximization problems are also included in this framework

$$\max_{x \in \mathcal{A}} J(x) = - \left(\min_{x \in \mathcal{A}} -J(x) \right).$$

Remark 2: The rigorous way is to write inf instead of min when we don't know that a solution exists in $\mathcal{A}$.

Questions:

- how do we deal with optimization problems in terms of $\mathcal{A}$? (discrete vs continuous case)
- when do we have a solution? what are the conditions for $\mathcal{A}$ and J?

$\mathcal{A}$ is finite

$\mathcal{A} = \{x_1, x_2, ..., x_N\}$ so J takes the values

$$\{J(x_1), J(x_2), ..., J(x_N)\}.$$

Questions:

- what about existence of solutions?
- if a solution exists, how do you find it?
- if $\mathcal{A}$ is finite, we always have existence of solutions!
- the difficulty of finding the optimal value among $J(x_i)$ depends on multiple factors:
  - how big is N?
  - how fast can you compute $J(x_i)$?
  - is there some underlying structure which can help us get to the solution faster?

Introductory Examples

$\star$ find the minimum element in an array
Solution: loop through the array, complexity $O(n)$, where n is the size of the array

$\star$ given n points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ find the pair of distinct points at minimal distance.
Solution: brute force: double loop in $O(n^2)$ time.
a more refined solution using **divide and conquer** approach:

- sort points w.r.t x coordinate $O(n \log n)$
- recursively compute minimal distances in the first half and second half of vertically split set of points
- check segments crossing the half point

`https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/`

Exploiting the structure

⋆ Often the complexity of the brute-force solution makes it impossible for large problem sizes
⋆ Discrete problems often come with a **structure** (for example graphs) which can be exploited to find more efficient algorithms

Two examples in the following:

- The optimal assignment problem
- Dijkstra's algorithm for shortest paths

Example 1: Optimal assignment problem

Let's say we have the following situation:

	Person 1	Person 2	Person 3
Job 1	100€	120€	80€
Job 2	150€	110€	120€
Job 3	90€	80€	110€

Questions:

1. What is the optimal assignment: **Job** $i \longrightarrow$ **Person** j such that the cost is minimal for the employer?
2. What is the cost of the naïve implementation in terms of the number of persons?

Example 1: Optimal assignment problem

Let's say we have the following situation:

	Person 1	Person 2	Person 3
Job 1	100€	120€	80€
Job 2	150€	110€	120€
Job 3	90€	80€	110€

Questions:

1. What is the optimal assignment: **Job** $i \longrightarrow$ **Person** j such that the cost is minimal for the employer?
2. What is the cost of the naïve implementation in terms of the number of persons? Answer: $O(n!)$

Example 1: Optimal assignment problem

Let's say we have the following situation:

	Person 1	Person 2	Person 3
Job 1	100€	120€	80€
Job 2	150€	110€	120€
Job 3	90€	80€	110€

Questions:

1. What is the optimal assignment: **Job** $i \longrightarrow$ **Person** j such that the cost is minimal for the employer?
2. What is the cost of the naïve implementation in terms of the number of persons? Answer: $O(n!)$
3. Is there a better algorithm? Yes: Hungarian algorithm with complexity $O(n^3)$.

The Hungarian Algorithm

Key idea: if one adds or subtracts an element from a row or a column of the matrix, an optimal assignment for the resulting cost matrix is also optimal for the original cost matrix.

Consider a cost matrix A of size $n \times n$.

1. Subtract the smallest entry in each row from all other entries in that row.
2. Subtract the smallest entry in each column from all other entries in that column.
3. Draw the minimal number of vertical and horizontal lines which can cover all the zeros.
4. If there are n lines drawn, choose an assignment using exactly one zero per line/column. This will be optimal.
5. If not, find smallest entry not covered by any line. Subtract this from each **row that is not crossed out** and add it to each **column that is crossed out**.

Complexity: $O(n^3)$

Example of application

Step 1: $\begin{pmatrix} 100 & 120 & 80 \\ 150 & 110 & 120 \\ 90 & 80 & 110 \end{pmatrix} \longrightarrow \begin{pmatrix} 20 & 40 & 0 \\ 40 & 0 & 10 \\ 10 & 0 & 30 \end{pmatrix}$

Step 2: $\begin{pmatrix} 20 & 40 & 0 \\ 40 & 0 & 10 \\ 10 & 0 & 30 \end{pmatrix} \longrightarrow \begin{pmatrix} 10 & 40 & 0 \\ 30 & 0 & 10 \\ 0 & 0 & 30 \end{pmatrix}$

Step 3: $\begin{pmatrix} 10 & 40 & 0 \\ 30 & 0 & 10 \\ 0 & 0 & 30 \end{pmatrix}$: at least three lines are needed to cover all zeros

Step 4: The minimal number of lines/columns needed to cover the zeros is 3: **we have found an optimal assignment**!

Another example

Step 1: $\begin{pmatrix} 108 & 125 & 150 \\ 150 & 135 & 175 \\ 122 & 148 & 250 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 & 17 & 42 \\ 15 & 0 & 40 \\ 0 & 26 & 128 \end{pmatrix}$

Step 2: $\begin{pmatrix} 0 & 17 & 42 \\ 15 & 0 & 40 \\ 0 & 26 & 128 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 & 17 & 2 \\ 15 & 0 & 0 \\ 0 & 26 & 88 \end{pmatrix}$

Steps 3-4: $\begin{pmatrix} 0 & 17 & 2 \\ 15 & 0 & 0 \\ 0 & 26 & 88 \end{pmatrix}$: Column 1+Line 2: only two lines needed

Step 5: subtract minimal uncovered element from uncovered rows, add it to covered columns.

$$\begin{pmatrix} 0 & 17 & 2 \\ 15 & 0 & 0 \\ 0 & 26 & 88 \end{pmatrix} \longrightarrow \begin{pmatrix} -2 & 15 & 0 \\ 15 & 0 & 0 \\ -2 & 24 & 86 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 & 15 & 0 \\ 17 & 0 & 0 \\ 0 & 24 & 86 \end{pmatrix}$$

Equivalent: subtract minimum from all uncovered elements, add it to elements crossed 2 times

Re-test: the minimal number of lines needed to cross all zeros is 3. An optimal assignment is found!

Possible Applications

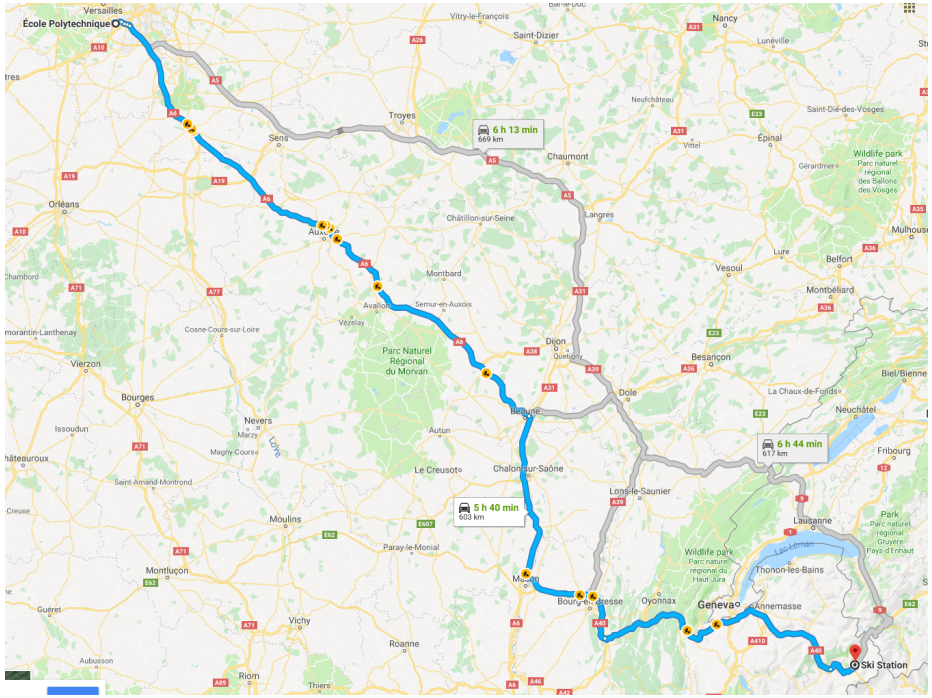
1. Assignment planning: minimize total cost, maximize personnel well being
2. Computer vision: assign right IDs to the right objects
3. Uber Algorithm: matching drivers and clients
   - take into account distances
   - for similar distances some drivers have different fares
   - take into account the driver rating
4. Extension to a rectangular cost matrix.

Lab work:

1. Brute force: use iterators in Python for permutations.
2. Explore an implemented solution in `scipy`.
3. Explore an explicit implementation.
4. Apply one of the solutions to other test cases.
5. Make your own implementation.

Example 2: Minimal path through a graph

Dijkstra's algorithm: intelligently find the optimal path going through the branches of your graph



Application: find fastest route on a map.

Undirected weighted graphs

A graph is defined by:

- a set of **vertices** V
- a set of **edges** E containing pairs of vertices (v_1, v_2) which are **connected**
- optionally, each edge may have a **weight** associated to it (Examples: distance on a map, cost of a connection, etc)

Dijkstra's algorithm computes distances from a node called **source node** to all other nodes in the graph. Optionally, the algorithm may be stopped once the distance to the **target node** is found.

Dijkstra's algorithm is a greedy algorithm: local solution leads to a global one.

- an optimal path from a to b must pass through a neighbor of b.
- if c is a neighbor of b: an optimal path from a to b containing c must be **optimal** from a to c
- if optimal paths are known for all neighbors of b, they can be used to find an optimal path to b

Dijkstra's algorithm

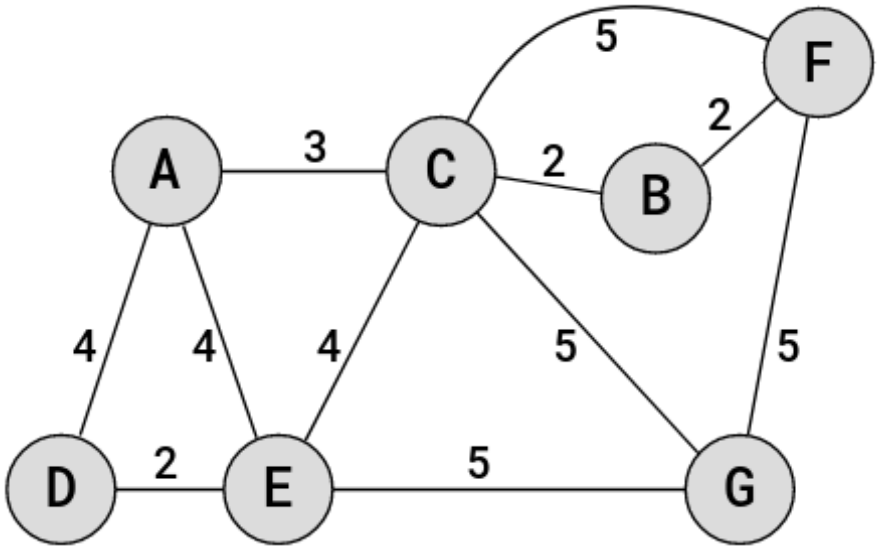
Consider $G = (V, E)$ a graph with weights W. Consider a source node a. **Initialization:** distances to all other vertices $= \infty$, keep a list of visited nodes, consider the source as the **current vertex**

1. Choose as current vertex the one which is unvisited and has shortest distence from the source.
2. For each of the current vertex **unvisited neighbors** compute the distance from the source and **update the distance if the new one is lower**
3. Mark the current vertex as visited. Visited vertices are not re-checked.
4. Go back to the first step.

After the algorithm finishes, all vertices will contain the **minimal distance to the source**.

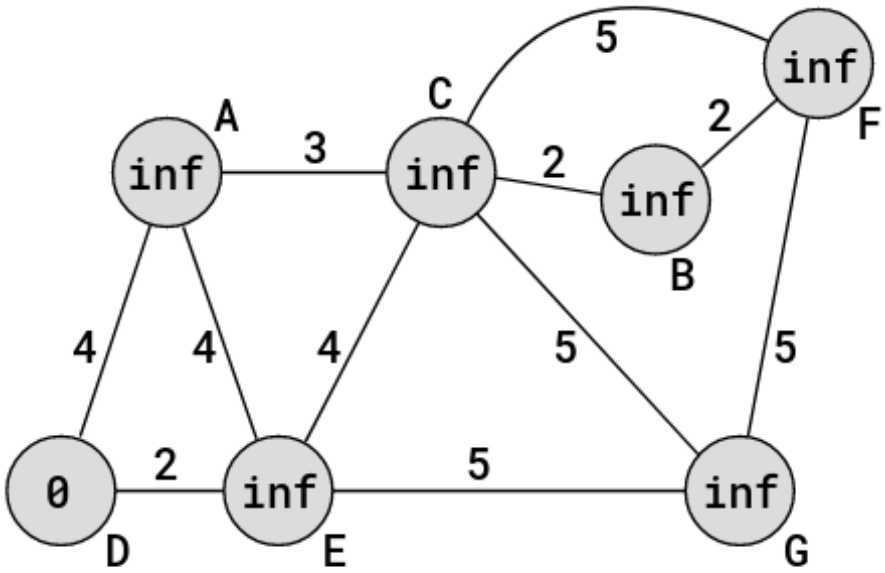
Example

Source node D.



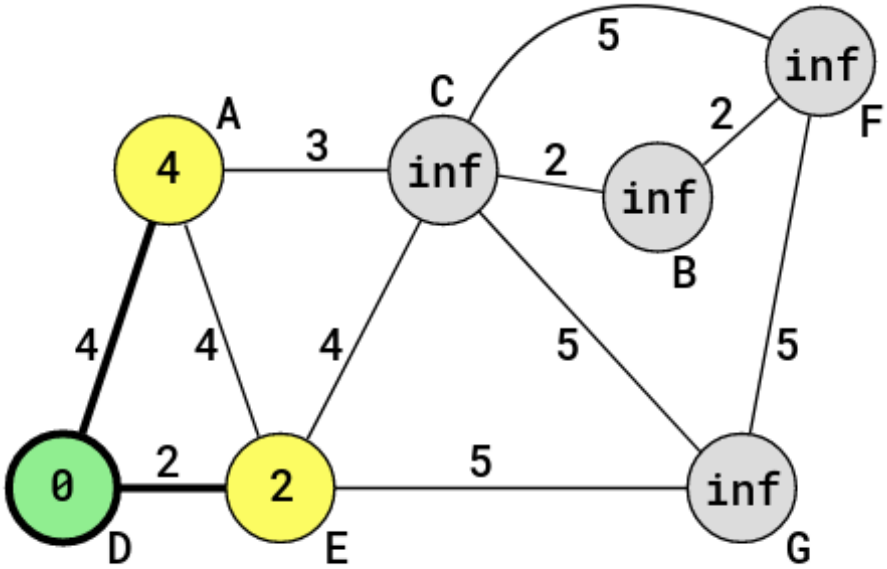
Example

Source node D.
Initialization: distance to all other nodes $= \infty$



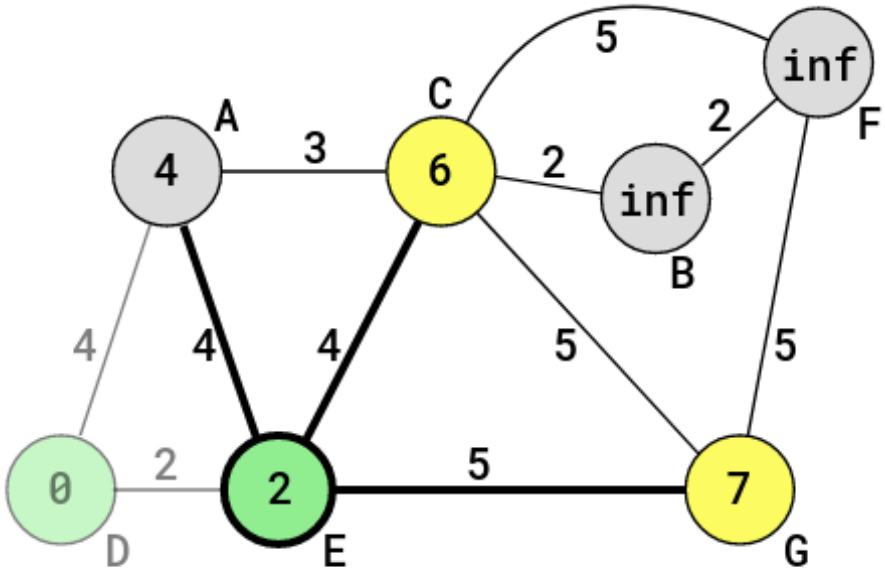
Example

Source node D.
Current node: D. Compute distance to the non-visited neighbors. Update if smaller. Mark D as visited.



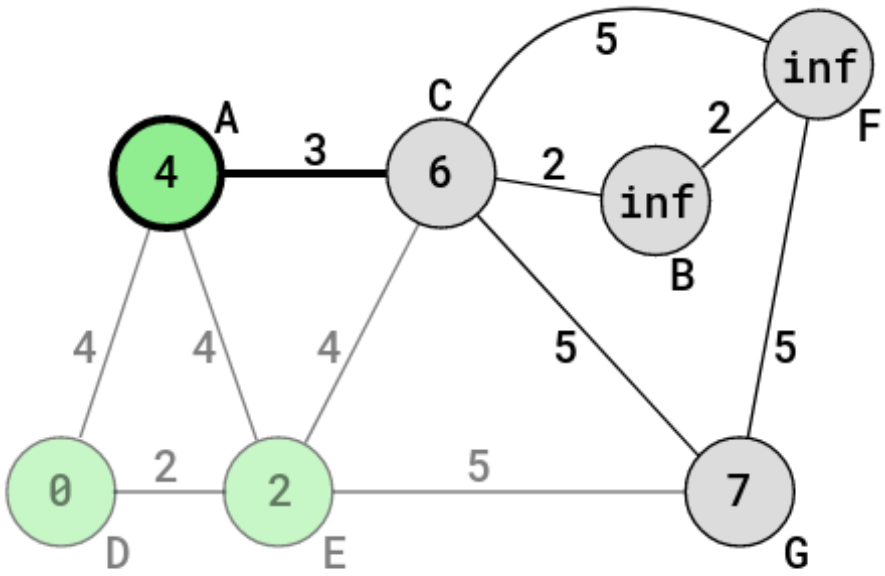
Example

Source node D.
Current node: E (smallest non-visited value). Compute distance to the non-visited neighbors. Update if smaller. Mark E as visited.



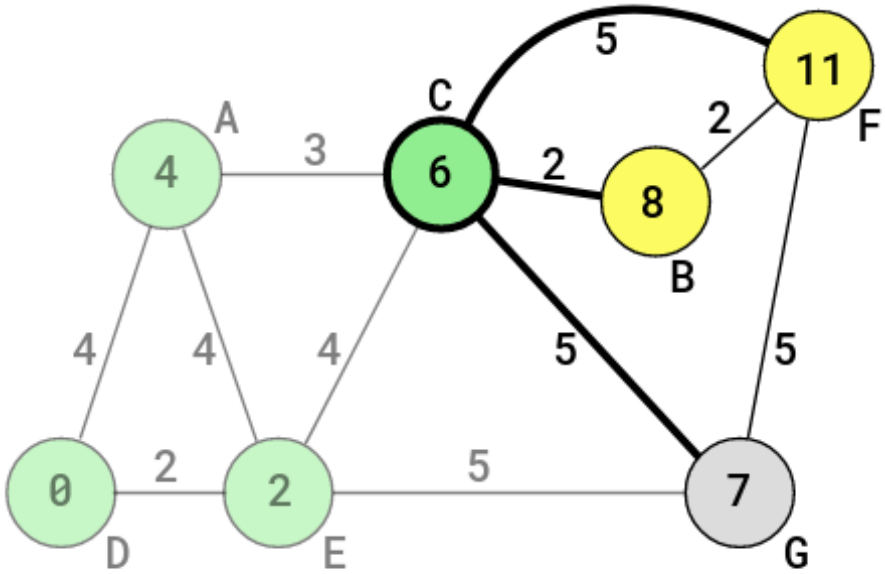
Example

Source node D.
Current node: A (smallest non-visited value). Compute distance to the non-visited neighbors. Update if smaller. Mark A as visited.



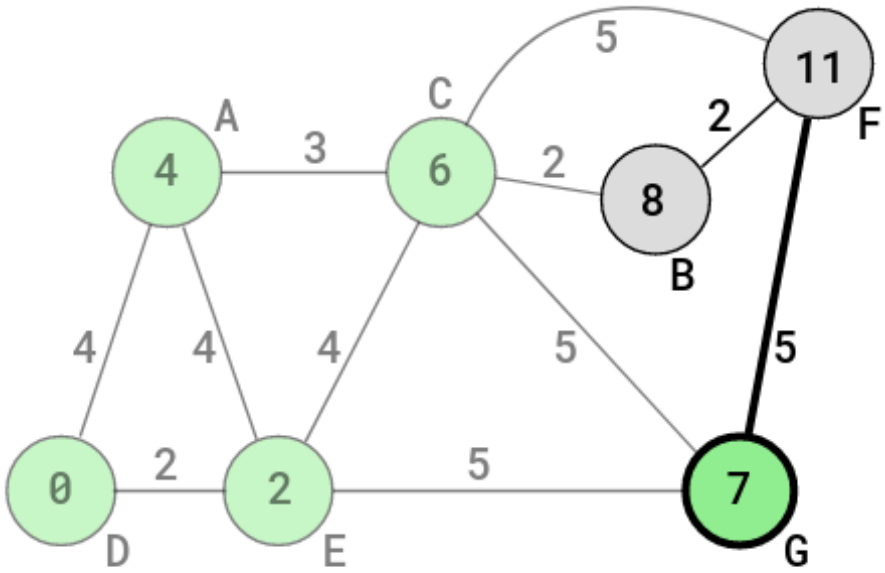
Example

Source node D.
Current node: C (smallest non-visited value). Compute distance to the non-visited neighbors. Update if smaller. Mark C as visited.



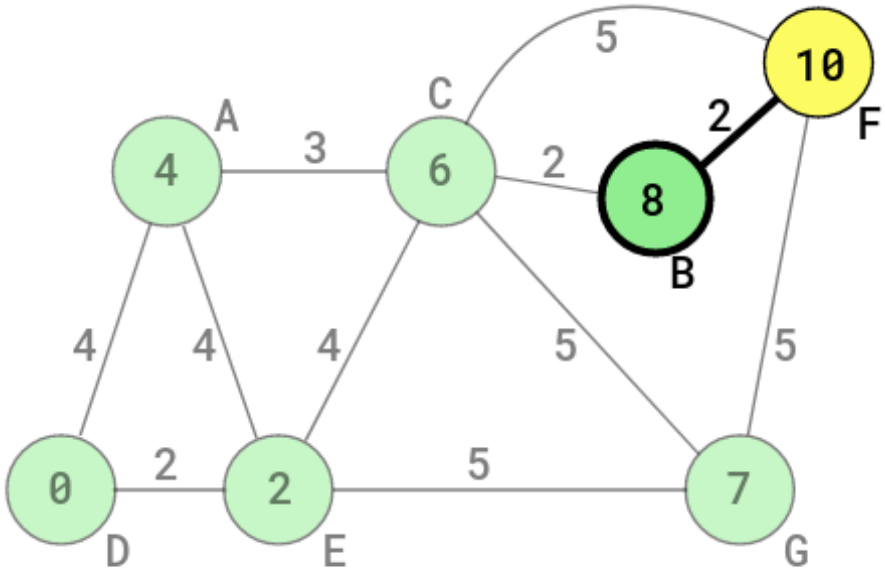
Example

Source node D.
Current node: G (smallest non-visited value). Compute distance to the non-visited neighbors. Update if smaller. Mark G as visited.



Example

Source node D.
Current node: B (smallest non-visited value). Compute distance to the non-visited neighbors. Update if smaller. Mark B as visited.



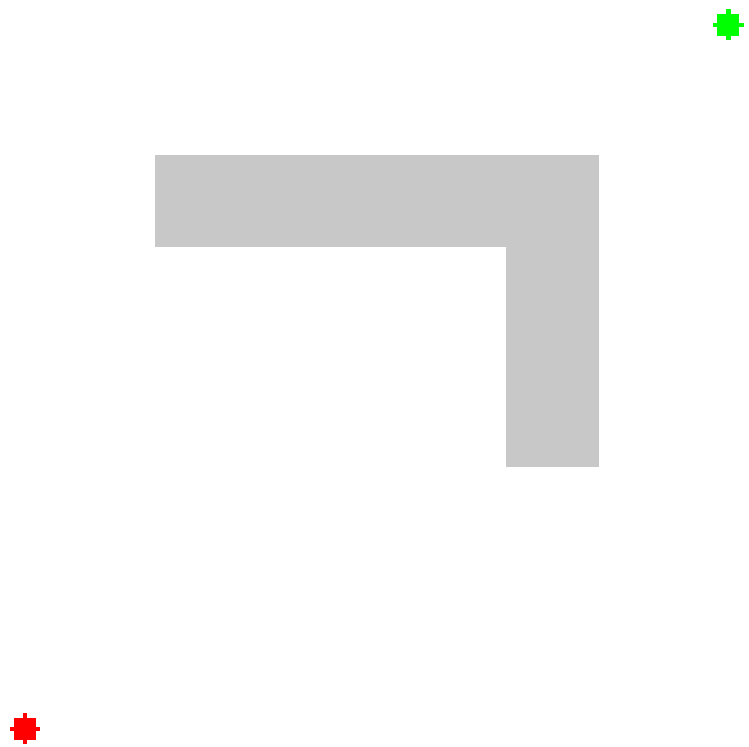
Re-constructing the minimal path

⋆ consider a vector `prev` having the same size as the number of vertices n in G
⋆ when the value at node j is updated at node i (a new minimal path ending in j is found), change: `prev[j]=i`
⋆ when the target node t is reached do the following:

$$t_1 := prev[t]; t_2 := prev[t_1] \ldots \text{ until } source := prev[t_k].$$

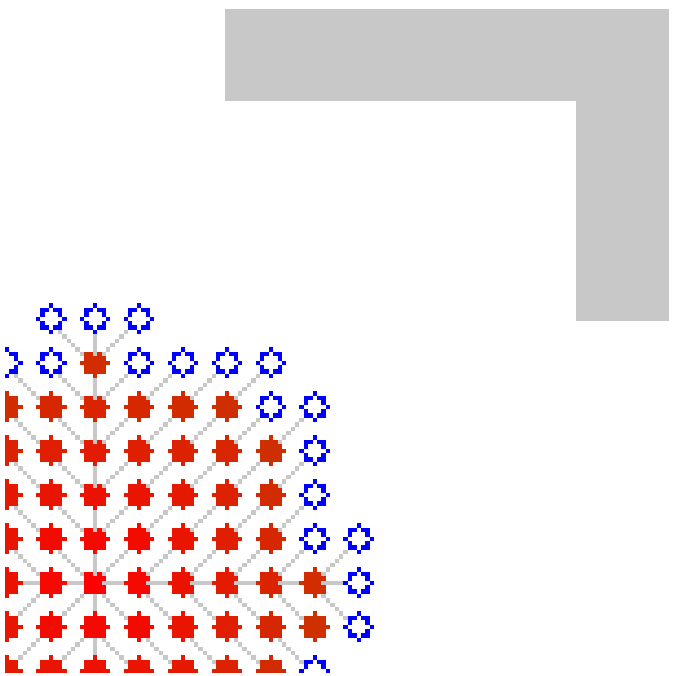
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



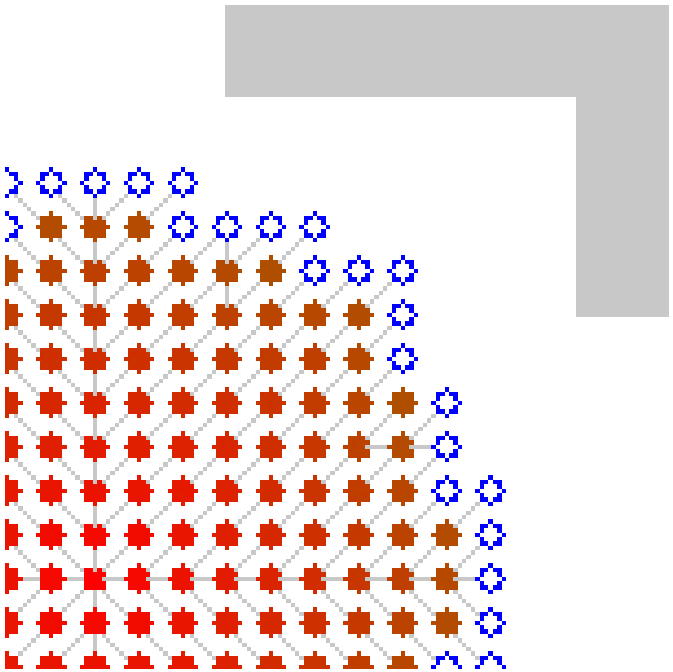
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



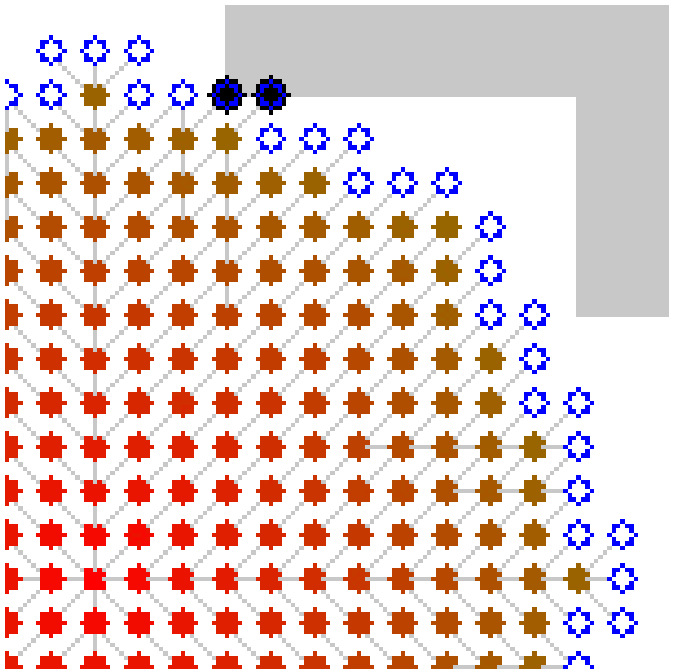
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



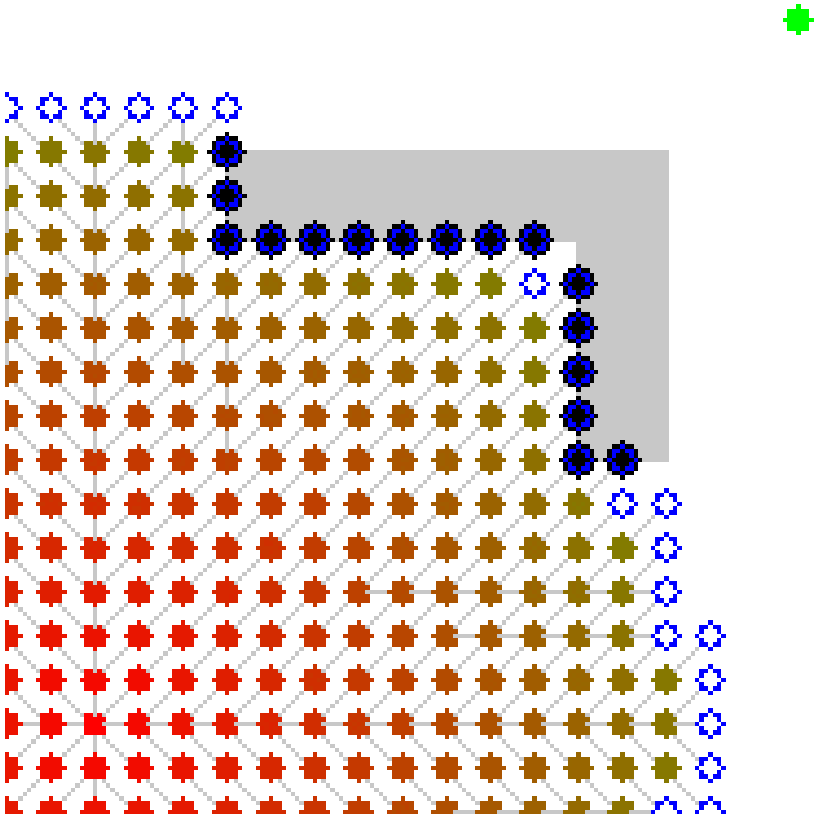
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



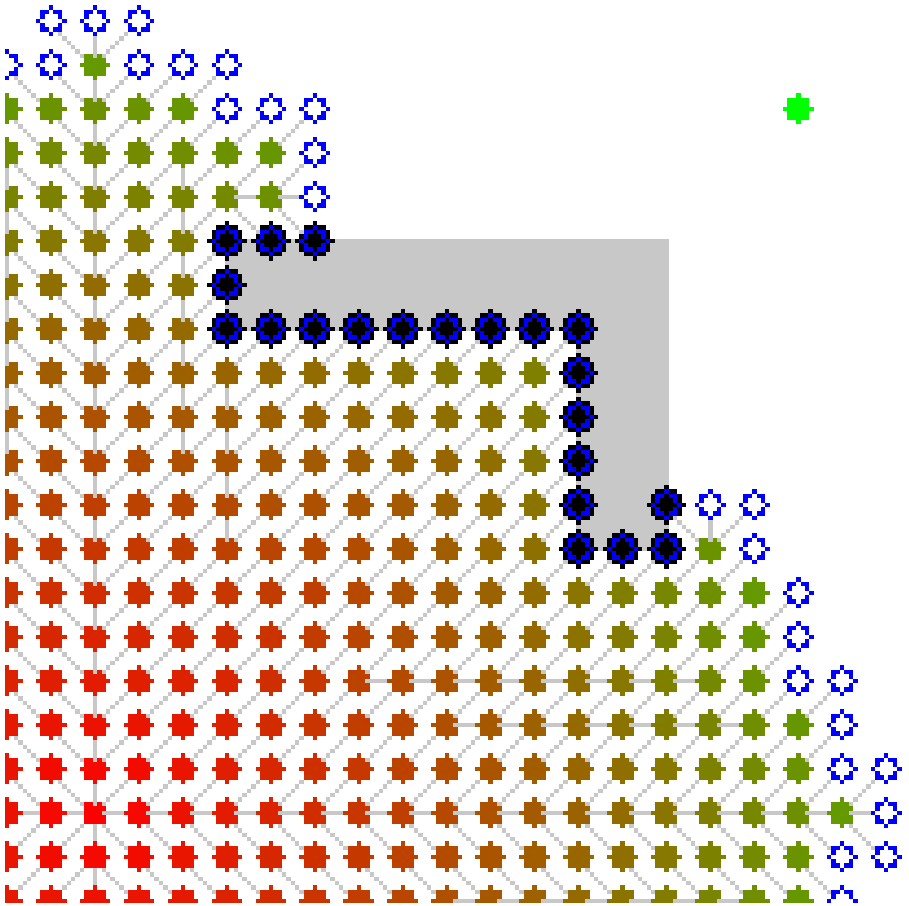
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



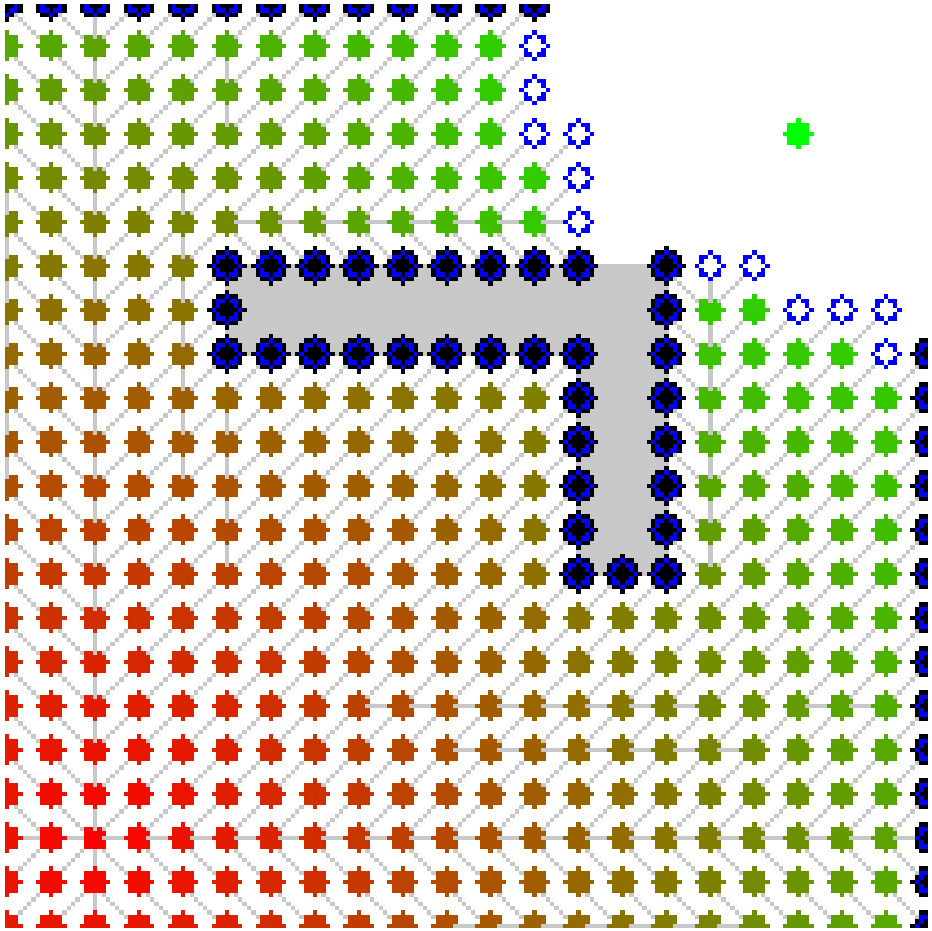
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



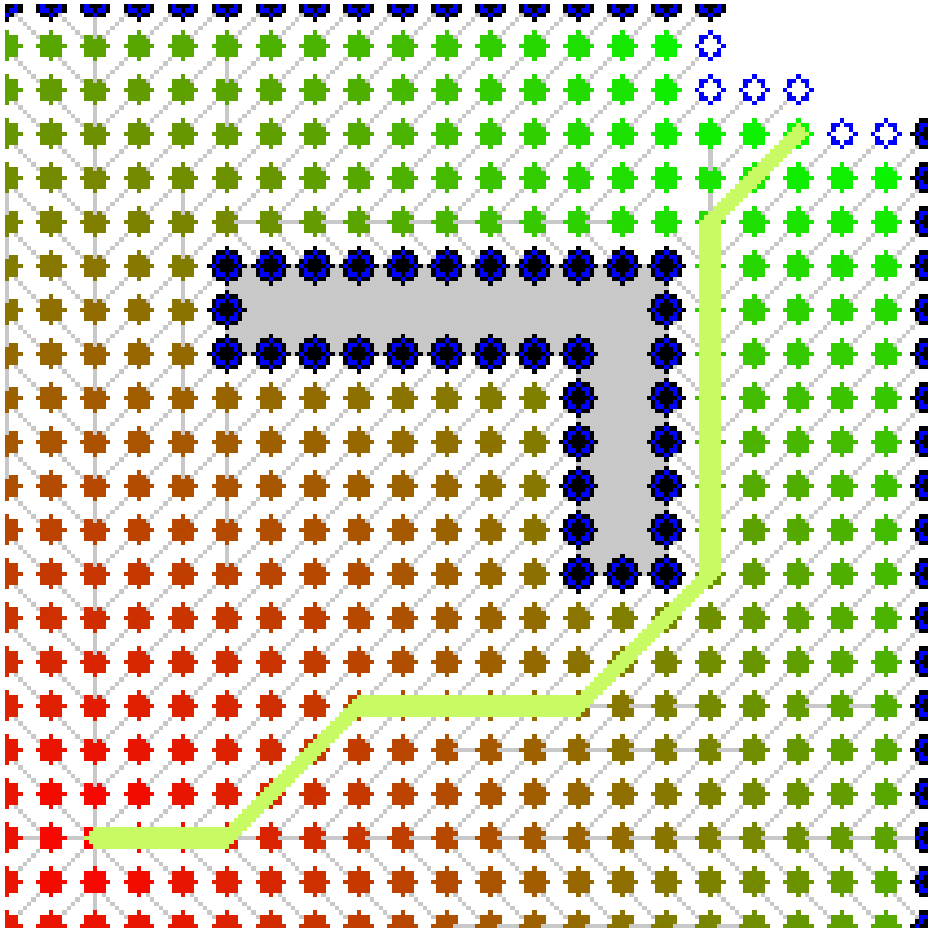
Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



Applications

1. Social networking: suggest list of people you may know
2. Telephone network: find best routing to transmit information
3. IP routing
4. Robotic path



Conclusion on the discrete part

- Discrete optimization problem: finite number of configurations $\longrightarrow$ existence of solutions
- That does not mean that we can always find the optimal solution in reasonable computation time
- Exploit the underlying structure of the problem (graphs, combinatorics)
- Many algorithmic challenges are solved: see Computer science courses
- [*The Art of Computer Programming* by Donald Knuth]
- We will not talk about discrete optimization in the rest of the course.