UNIVERSITÄT DES SAARLANDES

Continuity and Robustness of Programs

Seminar: Robustness of Hardware and Software Systems

Prof. Dr.-Ing. Jan Reineke

Markus Schneider

Saarbrücken, December 12, 2013

Motivation

- For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.
  - in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy

Motivation

- For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.
  - in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
  - in **mobile devices**: they use slightly stale satellite data
  - in **randomized** and **approximate algorithms** for performance gains
  - in **differential privacy** to guarantee privacy in statistical databases

Motivation

- For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.
  - in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
  - in **mobile devices**: they use slightly stale satellite data
  - in **randomized** and **approximate algorithms** for performance gains
  - in **differential privacy** to guarantee privacy in statistical databases
- This uncertainty can be probabilistic or nondeterministic.

Motivation

- For many programs we cannot guarantee a certain behaviour due to **uncertain input data**, e.g.
  - in **embedded control software**: any sensor data to percept physical properties is uncertain and can be noisy
  - in **mobile devices**: they use slightly stale satellite data
  - in **randomized** and **approximate algorithms** for performance gains
  - in **differential privacy** to guarantee privacy in statistical databases
- This uncertainty can be probabilistic or nondeterministic.

$\longrightarrow$ We will introduce a concept of **continuity for programs**.

The Challenge: Handling the Control Flow

- **Conditional branching.**

  if $x > 2$ **then**
  $\quad y := \frac{1}{2} \cdot x$
  else
  $\quad y := -5x + 11$
  end if

The Challenge: Handling the Control Flow

- **Conditional branching.**

```
if x > 2 then
    y := 1/2 · x
else
    y := −5x + 11
end if
```

- **Loops.**

```
while W ≠ ∅ do
    choose edge (v, w) ∈ G such that d[w] is minimal
    remove (v, w) from W
    if d[w] + G[w, v] < d[v] then
        d[v] := d[w] + G[w, v]
    end if
end while
```

The Challenge: Handling the Control Flow

- **Conditional branching.**

```
if x > 2 then
    y := 1/2 · x
else
    y := −5x + 11
end if
```

- **Loops.**

```
while W ≠ ∅ do
    choose edge (v, w) ∈ G such that d[w] is minimal
    remove (v, w) from W
    if d[w] + G[w, v] < d[v] then
        d[v] := d[w] + G[w, v]
    end if
end while
```

$\longrightarrow$ **Control flow** makes an automated continuity analysis difficult.

A Necessary Tool: Metrics

- We consider a program as the mathematical function that it implements.
- To be able to talk about continuity we have to **define a metric for each datatype**.

A Necessary Tool: Metrics

- We consider a program as the mathematical function that it implements.
- To be able to talk about continuity we have to **define a metric for each datatype**.
- Examples of metrics:
  - *integer* and *real*, associated with the **Euclidean metric**

$$d(x, y) = |x - y|$$

A Necessary Tool: Metrics

- We consider a program as the mathematical function that it implements.
- To be able to talk about continuity we have to **define a metric for each datatype**.
- Examples of metrics:
  - *integer* and *real*, associated with the **Euclidean metric**

$$d(x, y) = |x - y|$$

  - *integer arrays* and *real arrays*, associated with the **maximum norm**

$$d(A_1, A_2) = L_\infty(A_1, A_2) = \max_i(|A_1[i] - A_2[i]|)$$

Closeness of Program States

Continuity analysis of programs requires a definition of a **"distance" between two program states**.

Closeness of Program States

Continuity analysis of programs requires a definition of a **"distance" between two program states**.

Given two states σ and $\sigma' \in \Sigma(P)$ and any $\epsilon > 0$, we define:

- σ and σ' are ϵ-**close** with respect to variable x_i and write

$$\sigma \approx_{\epsilon,i} \sigma' \;\; :\Leftrightarrow \;\; d(\sigma(i), \sigma'(i)) < \epsilon$$

Closeness of Program States

Continuity analysis of programs requires a definition of a **“distance” between two program states**.

Given two states σ and $\sigma' \in \Sigma(P)$ and any $\epsilon > 0$, we define:

- σ and σ' are ϵ-**close** with respect to variable x_i and write

$$\sigma \approx_{\epsilon,i} \sigma' \;\;:\Leftrightarrow\;\; d(\sigma(i), \sigma'(i)) < \epsilon$$

- σ' is an ϵ-**perturbation of** σ with respect to variable x_i and write

$$\sigma \equiv_{\epsilon,i} \sigma' \;\;:\Leftrightarrow\;\; \sigma \approx_{\epsilon,i} \sigma' \wedge \forall j \neq i : \sigma(j) = \sigma'(j)$$

Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program

Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program

Continuity of a Program

Well-known ϵ-δ-Definition of Continuous Functions:
A function $f : D \to \mathbb{R}$ is continuous at a point $x \in D$, if

$$\forall \epsilon > 0 \; \exists \delta > 0 \; \forall y \in D : \; |x - y| < \delta \Rightarrow |f(x) - f(y)| < \epsilon$$

Continuity of a Program

Well-known ϵ-δ-Definition of Continuous Functions:
A function $f : D \to \mathbb{R}$ is continuous at a point $x \in D$, if

$$\forall \epsilon > 0 \; \exists \delta > 0 \; \forall y \in D : \; |x - y| < \delta \Rightarrow |f(x) - f(y)| < \epsilon$$

Continuity of a Program:
A program P is **continuous** at a state σ with respect to an input variable x_i and an output variable x_j, if

$$\forall \epsilon > 0 \; \exists \delta > 0 \; \forall \sigma' \in \Sigma(P) : \; \sigma \equiv_{\delta,i} \sigma' \Rightarrow [\![P]\!](\sigma) \approx_{\epsilon,j} [\![P]\!](\sigma')$$

Verifying Continuity (1)

- **Goal:** establish an automated framework for proving a program to be continuous

Verifying Continuity (1)

- **Goal:** establish an automated framework for proving a program to be continuous

- The analysis is
  - **sound** (a program proven continuous is indeed continuous),
  - but **incomplete** (a program may be continuous even if the analysis is not able to derive this).

Verifying Continuity (1)

- **Goal:** establish an automated framework for proving a program to be continuous

- The analysis is
  - **sound** (a program proven continuous is indeed continuous),
  - but **incomplete** (a program may be continuous even if the analysis is not able to derive this).

- Breaking down a program into its syntactic substructures we get a set of **inference rules** of the style

$$\frac{P \text{ is } \textsc{skip} \text{ or } x := e}{b \vdash \mathrm{Cont}(P, \mathrm{In}, \mathrm{Out})}$$

to derive **continuity judgements**.

Verifying Continuity (2)

Disallowing divisions the critical statements are **conditional branches**.

- The branches have to be *output-equivalent* at the decision boundary of the branch.

if $x > 2$ **then**
    $y := \frac{1}{2} \cdot x$
else
    $y := -5x + 11$
end if

Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program

Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:

A function $f : D \to \mathbb{R}$ is Lipschitz continuous, if there is a constant K so that any $\pm\epsilon$-change to x can change $f(x)$ at most by $\pm K \cdot \epsilon$.

Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function $f : D \to \mathbb{R}$ is Lipschitz continuous, if there is a constant K so that any $\pm\epsilon$-change to x can change $f(x)$ at most by $\pm K \cdot \epsilon$.

Lipschitz Continuity of a Program:
Let $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function that takes the size of variable x_i as its input. A program P is K-**Lipschitz** with respect to an input variable x_i and an output variable x_j, if $\forall \sigma, \sigma' \in \Sigma(P)$ and $\forall \epsilon > 0$

$$\sigma \equiv_{\epsilon,i} \sigma' \Rightarrow [\![P]\!](\sigma) \approx_{K\cdot\epsilon,j} [\![P]\!](\sigma')$$

Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:
A function $f : D \rightarrow \mathbb{R}$ is Lipschitz continuous, if there is a constant K so that any $\pm\epsilon$-change to x can change $f(x)$ at most by $\pm K \cdot \epsilon$.

Lipschitz Continuity of a Program:
Let $K : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ be a function that takes the size of variable x_i as its input. A program P is K-**Lipschitz** with respect to an input variable x_i and an output variable x_j, if $\forall \sigma, \sigma' \in \Sigma(P)$ and $\forall \epsilon > 0$

$$\sigma \equiv_{\epsilon,i} \sigma' \Rightarrow [\![P]\!](\sigma) \approx_{K\cdot\epsilon,j} [\![P]\!](\sigma')$$

where K only depends on the size of $\sigma(i)$. The size of a variable v is defined as

- $||v|| := 1$, if v is an integer or a real,
- $||v|| := N$, if v is an array of size N.

Lipschitz Continuity of a Program

Definition of Lipschitz continuous Functions:

A function $f : D \to \mathbb{R}$ is Lipschitz continuous, if there is a constant K so that any $\pm\epsilon$-change to x can change $f(x)$ at most by $\pm K \cdot \epsilon$.

Lipschitz Continuity of a Program:

Let $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function that takes the size of variable x_i as its input. A program P is K-**Lipschitz** with respect to an input variable x_i and an output variable x_j, if $\forall \sigma, \sigma' \in \Sigma(P)$ and $\forall \epsilon > 0$

$$\sigma \equiv_{\epsilon,i} \sigma' \wedge \left(||\sigma(i)|| = ||\sigma'(i)||\right) \Rightarrow [\![P]\!](\sigma) \approx_{K\cdot\epsilon,j} [\![P]\!](\sigma')$$

where K only depends on the size of $\sigma(i)$. The size of a variable v is defined as

- $||v|| := 1$, if v is an integer or a real,
- $||v|| := N$, if v is an array of size N.

Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  Example:

$$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
$$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  Example:

$$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
$$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

  Perturbing each item of an array at most by $\pm\epsilon$ changes each item of the output array at most by $\pm\epsilon$.

Example (1): Sorting Algorithms

- *Sort*$_1$ maps an array to its sorted permutation.
  Example:

$$Sort_1(6,3,3,1) = (1,3,3,6)$$
$$Sort_1(6,3+\epsilon,3,1) = (1,3,3+\epsilon,6)$$

  Perturbing each item of an array at most by $\pm\epsilon$ changes each item of the output array at most by $\pm\epsilon$.

- *Sort*$_2$ maps an array to the list of indices giving the order.
  Example:

$$Sort_2(6,3,3,1) = (4,2,3,1)$$
$$Sort_2(6,3+\epsilon,3,1) = (4,3,2,1)$$

Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  Example:

  $$Sort_1(6,3,3,1) = (1,3,3,6)$$
  $$Sort_1(6,3+\epsilon,3,1) = (1,3,3+\epsilon,6)$$

  Perturbing each item of an array at most by $\pm\epsilon$ changes each item of the output array at most by $\pm\epsilon$.

- $Sort_2$ maps an array to the list of indices giving the order.
  Example:

  $$Sort_2(6,3,3,1) = (4,2,3,1)$$
  $$Sort_2(6,3+\epsilon,3,1) = (4,3,2,1)$$

  Perturbing one item by $\pm\epsilon$ can already lead to unbounded changes in the corresponding outputs.

Example (1): Sorting Algorithms

- $Sort_1$ maps an array to its sorted permutation.
  Example:

$$Sort_1(6, 3, 3, 1) = (1, 3, 3, 6)$$
$$Sort_1(6, 3 + \epsilon, 3, 1) = (1, 3, 3 + \epsilon, 6)$$

  Perturbing each item of an array at most by $\pm\epsilon$ changes each item of the output array at most by $\pm\epsilon$.

- $Sort_2$ maps an array to the list of indices giving the order.
  Example:

$$Sort_2(6, 3, 3, 1) = (4, 2, 3, 1)$$
$$Sort_2(6, 3 + \epsilon, 3, 1) = (4, 3, 2, 1)$$

  Perturbing one item by $\pm\epsilon$ can already lead to unbounded changes in the corresponding outputs.

$\rightarrow$ $Sort_1$ is Lipschitz continuous, $Sort_2$ is not even continuous.

Example (2): Shortest Path Algorithms

- SP_1 maps a graph to its minimal distance array d.
- SP_2 maps a graph to an array containing the shortest paths.

→ SP_1 is continuous, SP_2 is not.

Example (2): Shortest Path Algorithms

- SP_1 maps a graph to its minimal distance array d.
- SP_2 maps a graph to an array containing the shortest paths.

$\rightarrow$ SP_1 is continuous, SP_2 is not.

We have to define the **output** of our program exactly!

Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.

Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.
- The upper bound $K \cdot \epsilon$ on the output changes does not depend on the values of the input variables.

Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.
- The upper bound $K \cdot \epsilon$ on the output changes does not depend on the values of the input variables.

$\longrightarrow$ The program behaves predictably on uncertain inputs.

Robustness of Programs

For Lipschitz continuous programs we can state:

- The output changes proportionally to any change on the inputs.
- The upper bound $K \cdot \epsilon$ on the output changes does not depend on the values of the input variables.

$\longrightarrow$ The program behaves predictably on uncertain inputs.

A program is called robust, if it is K-Lipschitz for some Lipschitz constant K.

Overview

Continuity of Programs and Continuity Judgements

Lipschitz Continuity of Programs

Verifying the Robustness of a Program

Our Two Step Procedure

The sequence of assignment or SKIP-statements that P executes on some input is called a **control flow path** of P.

Our Two Step Procedure

The sequence of assignment or SKIP-statements that P executes on some input is called a **control flow path** of P.

Let x_j be the input and x_i be the output variable of our program.

Our Two Step Procedure

The sequence of assignment or SKIP-statements that P executes on some input is called a **control flow path** of P.

Let x_j be the input and x_i be the output variable of our program.

Lipschitz continuity of a program is proven by establishing that

1. P is continuous in all states w.r.t. input x_j and output x_i.
2. Each control flow path of P is K-Lipschitz w.r.t. input x_j and output x_i.

The Idea for Finding Lipschitz Constants

The remaining task is to find out the Lipschitz constants for each control flow path (if there exists one).

The Idea for Finding Lipschitz Constants

The remaining task is to find out the Lipschitz constants for each control flow path (if there exists one).

Our approach:

- Compute **Lipschitz matrices** containing upper bounds on the slope of any computation that can be carried out in a control flow path of P.

Lipschitz Matrices

Let program P have n variables $x_1, .., x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ as its matrix elements.

Lipschitz Matrices

Let program P have n variables $x_1, .., x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ as its matrix elements.
- We will derive a set $\mathcal{J}$ of Lipschitz matrices.
- A judgement $P : \mathcal{J}$ means:
  For each control flow path C in P and each x_i, x_j there is a $J \in \mathcal{J}$ such that C is J_{ij}-Lipschitz in input x_j and output x_i.

Lipschitz Matrices

Let program P have n variables $x_1, .., x_n$.

- A **Lipschitz matrix** is a $n \times n$-matrix with functions $K : \mathbb{N} \to \mathbb{R}_{\geq 0}$ as its matrix elements.
- We will derive a set $\mathcal{J}$ of Lipschitz matrices.
- A judgement $P : \mathcal{J}$ means:
  For each control flow path C in P and each x_i, x_j there is a $J \in \mathcal{J}$ such that C is J_{ij}-Lipschitz in input x_j and output x_i.

Note the similarity to the *Jacobian*:

- If the program represents a differentiable function, J_{ij} is an upper bound on $|\frac{\partial x_i}{\partial x_j}|$.

Merging of Lipschitz Matrices

- Given any judgement $P : \mathcal{J}$, we can merge two arbitrary Lipschitz matrices A and $B \in \mathcal{J}$. Formally, we can infer

  $$P : (\mathcal{J} \setminus \{A, B\}) \cup \{A \sqcup B\}$$

  where the **merge operation** $\sqcup$ is defined as

  $$(A \sqcup B)_{ij} = \max(A_{ij}, B_{ij}) \qquad \forall i, j \in \{1, .., n\}$$

Rules for Deriving Lipschitz Matrices (1)

$$\text{skip}\ \frac{}{\textsc{skip} : \{\mathbf{I}\}}$$

Rules for Deriving Lipschitz Matrices (1)

$$\text{skip}\ \frac{}{\text{SKIP} : \{\mathbf{I}\}}$$

$$\text{weaken}\ \frac{P : \mathcal{J} \qquad J_1, J_2 \in \mathcal{J}}{P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}}$$

Rules for Deriving Lipschitz Matrices (1)

$$\text{skip}\ \frac{}{\textsc{skip} : \{\mathbf{I}\}}$$

$$\text{weaken}\ \frac{P : \mathcal{J} \qquad J_1, J_2 \in \mathcal{J}}{P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}}$$

$$\text{ITE}\ \frac{P_1 : \mathcal{J}_1 \qquad P_2 : \mathcal{J}_2}{(\textsc{if}\ B\ \textsc{then}\ P_1\ \textsc{else}\ P_2) : \mathcal{J}_1 \cup \mathcal{J}_2}$$

Rules for Deriving Lipschitz Matrices (1)

$$\text{skip}\ \frac{}{\textsc{skip} : \{\mathbf{I}\}}$$

$$\text{weaken}\ \frac{P : \mathcal{J} \quad J_1, J_2 \in \mathcal{J}}{P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}}$$

$$\text{ITE}\ \frac{P_1 : \mathcal{J}_1 \quad P_2 : \mathcal{J}_2}{(\textsc{if}\ B\ \textsc{then}\ P_1\ \textsc{else}\ P_2) : \mathcal{J}_1 \cup \mathcal{J}_2}$$

$$\text{sequence}\ \frac{P_1 : \mathcal{J}_1 \quad P_2 : \mathcal{J}_2}{(P_1; P_2) : \{J_2 \cdot J_1 \mid J_1 \in \mathcal{J}_1, J_2 \in \mathcal{J}_2\}}$$

Rules for Deriving Lipschitz Matrices (1)

skip $\overline{\text{SKIP} : \{\mathbf{I}\}}$

weaken $$\frac{P : \mathcal{J} \qquad J_1, J_2 \in \mathcal{J}}{P : (\mathcal{J} \setminus \{J_1, J_2\}) \cup \{J_1 \sqcup J_2\}}$$

ITE $$\frac{P_1 : \mathcal{J}_1 \qquad P_2 : \mathcal{J}_2}{(\text{IF } B \text{ THEN } P_1 \text{ ELSE } P_2) : \mathcal{J}_1 \cup \mathcal{J}_2}$$

sequence $$\frac{P_1 : \mathcal{J}_1 \qquad P_2 : \mathcal{J}_2}{(P_1; P_2) : \{J_2 \cdot J_1 \mid J_1 \in \mathcal{J}_1, J_2 \in \mathcal{J}_2\}}$$

while $$\frac{\begin{array}{c} P = \text{WHILE } b \text{ DO } R \qquad R : \mathcal{J} \qquad \mathit{Bound}^{+}(P, M) \\ \forall J \in \mathcal{J}\ \forall i, j : \ J_{ij} \geq 1 \lor J_{ij} = 0 \end{array}}{P : \{J_1 \cdot J_2 \cdot \ldots \cdot J_M \mid J_i \in \mathcal{J}\}}$$

Rules for Deriving Lipschitz Matrices (2)

For assignments we first define a vector ∇_e whose j-th element is an upper bound on $|\frac{\partial [\![e]\!]}{\partial x_j}|$:

$$\nabla_e(j) = \begin{cases} 0, & \text{if } e \text{ is a constant} \\ 1, & \text{if } e \text{ is } x_j \text{ or } x_j[k] \text{ for some } k \\ 0, & \text{if } e \text{ is } x_l \text{ or } x_l[k] \text{ for some } k \text{ and } l \neq j \\ \nabla_a(j) + \nabla_b(j), & \text{if } e \text{ is } (a+b) \\ \nabla_a(j)|b| + \nabla_b(j)|a|, & \text{if } e \text{ is } (a \cdot b) \text{ and } a \text{ or } b \text{ is a constant} \\ \infty, & \text{otherwise} \end{cases}$$

Rules for Deriving Lipschitz Matrices (2)

For assignments we first define a vector ∇_e whose j-th element is an upper bound on $|\frac{\partial [\![e]\!]}{\partial x_j}|$:

$$\nabla_e(j) = \begin{cases} 0, & \text{if } e \text{ is a constant} \\ 1, & \text{if } e \text{ is } x_j \text{ or } x_j[k] \text{ for some } k \\ 0, & \text{if } e \text{ is } x_l \text{ or } x_l[k] \text{ for some } k \text{ and } l \neq j \\ \nabla_a(j) + \nabla_b(j), & \text{if } e \text{ is } (a+b) \\ \nabla_a(j)|b| + \nabla_b(j)|a|, & \text{if } e \text{ is } (a \cdot b) \text{ and } a \text{ or } b \text{ is a constant} \\ \infty, & \text{otherwise} \end{cases}$$

$$\textbf{assign} \quad \frac{}{(x_i := e) : \{J\}} \quad \text{where } J_{kj} := \begin{cases} \nabla_e(j), & \text{if } k = i \\ 1, & \text{if } k = j \neq i \\ 0, & \text{otherwise} \end{cases}$$

Rules for Deriving Lipschitz Matrices (3)

$$\text{array-assign} \quad \frac{}{(x_i[m] := e) : \{J, \mathbf{I}\}}$$

with the same matrix J: $J_{kj} := \begin{cases} \nabla_e(j), & \text{if } k = i \\ 1, & \text{if } k = j \neq i \\ 0, & \text{otherwise} \end{cases}$

Example: Dijkstra's-Algorithm

```
DIJKSTRA(G: real array, src: int)
...
while W ≠ ∅ do
    choose edge (v, w) ∈ G such that d[w] is minimal
    remove (v, w) from W
    if d[w] + G[w, v] < d[v] then
        d[v] := d[w] + G[w, v]
    end if
end while
```

Example: Dijkstra's-Algorithm

DIJKSTRA(G: real array, src: int)

```
...
while W ≠ ∅ do
    choose edge (v, w) ∈ G such that d[w] is minimal
    remove (v, w) from W
    if d[w] + G[w, v] < d[v] then
        d[v] := d[w] + G[w, v]
    end if
end while
```

DIJKSTRA is continuous and we can infer the Lipschitz matrix

$$\begin{pmatrix} 1 & 0 \\ N & 1 \end{pmatrix}$$

so that DIJKSTRA is N-Lipschitz in input $G =: x_0$ and output $d =: x_1$, where N denotes the number of edges in G.

Conclusion

- We asked for a theory about **robustness** of programs to uncertainty.
- **Lipschitz continuity** is an adequate answer to this question. It is a strong property.

- Developing an **automated** continuity analysis is demanding.
- The analysis is proven to be sound, but incomplete.

Conclusion

- We asked for a theory about **robustness** of programs to uncertainty.
- **Lipschitz continuity** is an adequate answer to this question. It is a strong property.
- Developing an **automated** continuity analysis is demanding.
- The analysis is proven to be sound, but incomplete.
- **Arising questions:**
  - Is it satisfactory to live without divisions?
  - The degree of automation remains unclear.

Literature

- Swarat Chaudhuri, Sumit Gulwani & Roberto Lublinerman (2010). *Continuity Analysis of Programs.* POPL, 57-70.
- Swarat Chaudhuri, Sumit Gulwani, Sara Navidpour & Roberto Lublinerman (2011). *Proving Programs Robust.* FSE, 102-112.
- Swarat Chaudhuri, Sumit Gulwani & Roberto Lublinerman (2012). *Continuity and Robustness of Programs.* CACM, 107-115.