A Formalised Lower Bound on Undirected Graph Reachability

Ulrich Schöpp
University of Munich

Computer Aided Formal Reasoning

Computer assistance in our everyday work?

- bookkeeping for complicated technical details
- revalidation after changes
- correctness

Computer Aided Formal Reasoning

Computer assistance in our everyday work?

- bookkeeping for complicated technical details
- revalidation after changes
- correctness

Theorem prover technology is getting to a point where it is useful in practice for proving theorems.

- programming language theory: POPLMark
- four colour theorem

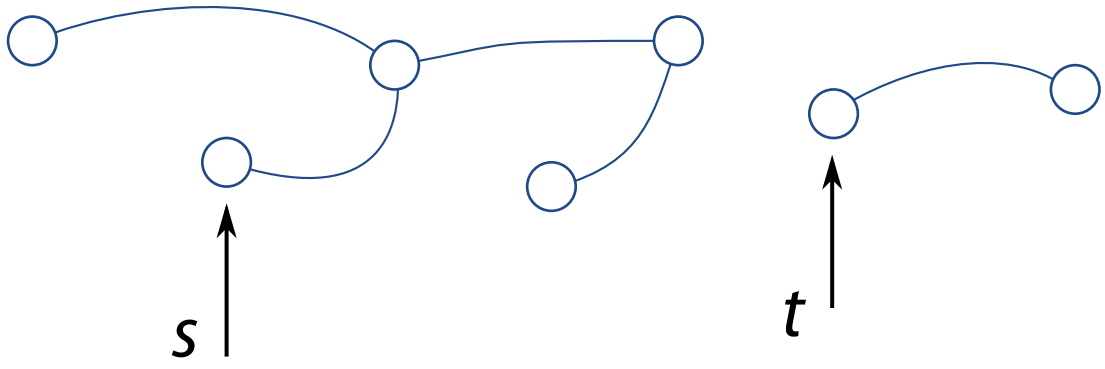
Today: case study from structural complexity theory

Context

Formalised proofs in my current work with Martin Hofmann

Expressivity of Pointer Programs on Graphs

- graph as a structured, read-only input
- while-language with boolean variables and pointer variables
- various constructs for pointer manipulation (operations of the input structure $succ_i$, iteration, etc.)



Pointer Programs and Undirected *s*-*t*-Reachability

pure pointer algoritms (only $succ_i$)

+ iteration over all nodes (~ Java Iterator)

+ counting registers (~ Java `int`)

+ pointer arithmetic (~ C `int`)

Pointer Programs and Undirected *s*-*t*-Reachability

---

pure pointer algoritms (only $succ_i$)

= Jumping Automata on Graphs (JAGs)

+ iteration over all nodes (~ Java Iterator)

> Deterministic Transitive Closure (DTC)

Logic

+ counting registers (~ Java `int`)

= RAM-JAG

+ pointer arithmetic (~ C `int`)

= LOGSPACE

Pointer Programs and Undirected *s*-*t*-Reachability

pure pointer algoritms (only $succ_i$) $\not\ni$ USTCON

= Jumping Automata on Graphs (JAGs) Cook & Rackoff 1980

+ iteration over all nodes (~ Java Iterator) $\not\ni$ USTCON

> Deterministic Transitive Closure (DTC) Sch. & Hofmann 2008

Logic

+ counting registers (~ Java `int`) $\ni$ USTCON

= RAM-JAG Reingold 2005

+ pointer arithmetic (~ C `int`) $\ni$ USTCON

= LOGSPACE

Pointer Programs and Undirected *s*-*t*-Reachability

pure pointer algoritms (only $succ_i$) $\not\ni$ USTCON

= Jumping Automata on Graphs (JAGs) Cook & Rackoff 1980

+ iteration over all nodes (~ Java Iterator) $\not\ni$ USTCON

> Deterministic Transitive Closure (DTC) Sch. & Hofmann 2008

Logic

+ counting $\ni$ USTCON

= RAM-J

Reingold 2005

+ pointer a $\ni$ USTCON

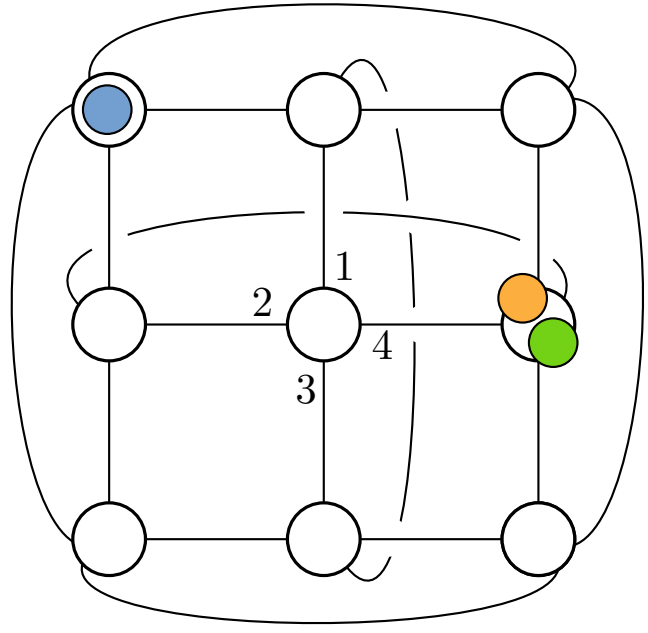
= LOGSPACE

Proof uses a generalisation of Cook & Rackoff's result which we have developed using Coq.

Improving the Result of Cook & Rackoff

Jumping Automata on Graphs

- finite set of states Q
- finite set of pebbles P
- automaton can see whether or not two pebbles are on the same graph node
- can move one pebble per step
  - move pebble along edge
  - jump pebble to another one

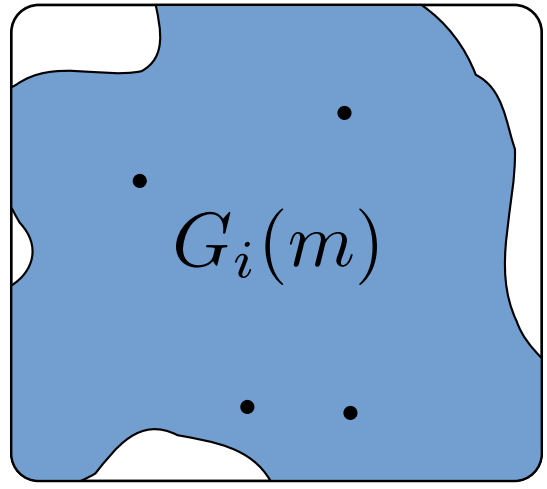


$P = \{$  ,  ,  $\}$

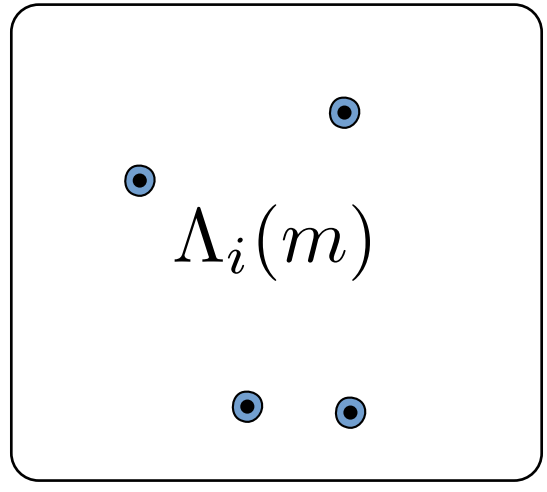
Improving Cook & Rackoff's Result

JAG with state set Q and pebble set P

Theorem (Cook & Rackoff 1980)
On the graph $G_i(m)$ the JAG can visit at most $(|Q| \cdot m)^{c^{|P|}}$ nodes from any start configuration.



Improvement
On the graph $\Lambda_i(m)$ the JAG can visit at most $(|Q| \cdot m \cdot 2^i)^{c^{|P|}}$ nodes from any start configuration.

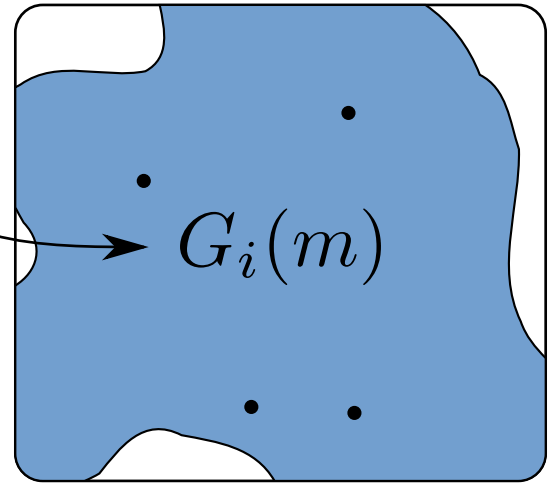


Improving Cook & Rackoff's Result

JAG with state set Q and pebble set P

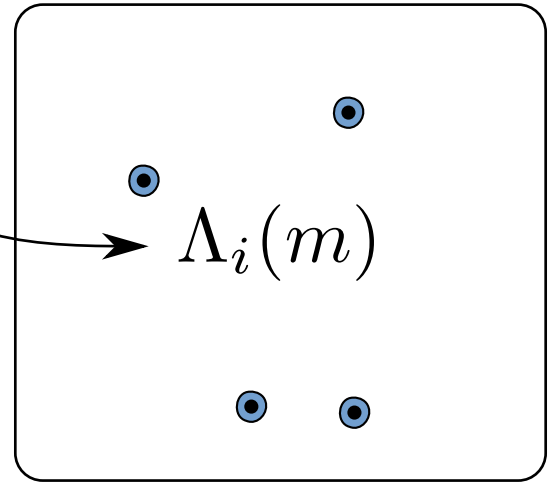
Degree: $2i$
Diameter: $m \cdot i$

Theorem (Cook &
On the graph $G_i(m)$ the JAG
can visit at most $(|Q| \cdot m)^{c^{|P|}}$ nodes
from any start configuration.



Degree: $2i + 2$
Diameter: $2^{2^{\cdot^{\cdot^{\cdot^{2^m}}}}}$

Improvemer
On the graph $\Lambda_i(m)$ the JAG can
visit at most $(|Q| \cdot m \cdot 2^i)^{c^{|P|}}$ nodes
from any start configuration.



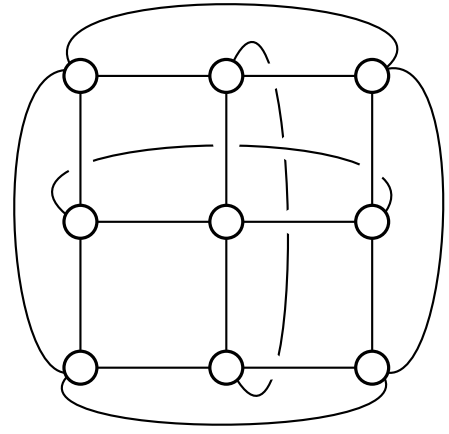
Outline of Cook & Rackoff's Proof

Find an upper bound on the number of nodes that a JAG can visit in $G_i(m)$.

Graph $G_i(m)$:

Nodes: $(\mathbb{Z}/m\mathbb{Z})^i$

Edges: between v and $v \pm (0, \ldots, 0, 1, 0, \ldots, 0)$

$G_2(3) =$ 

Use periodicity of $G_i(m)$:

If the actions of the JAG become periodic then a full configuration will be repeated not long after.

When are Actions Repeated?

Estimate the number of different behaviours of a JA

- behaviour = list of actions the JAG makes
- JAG repeats actions after at most #behaviours steps

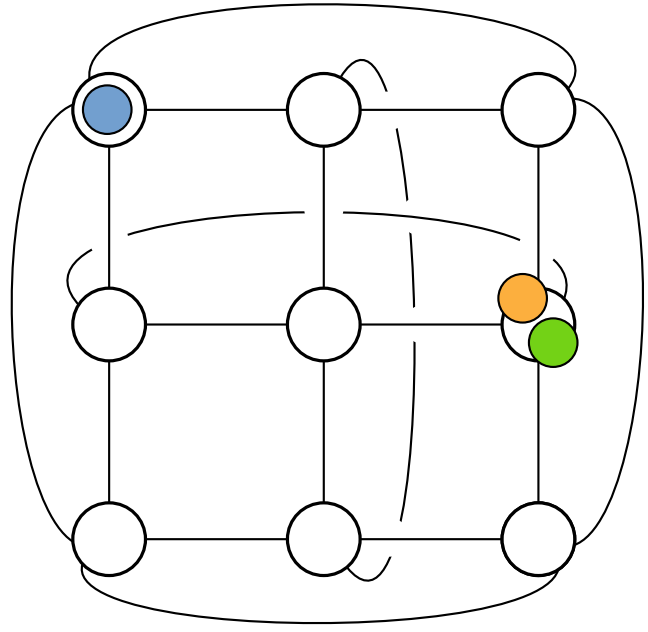
Analysis of Behaviour

- only relative pebble positions relevant
- some pebble collisions give JAG information

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

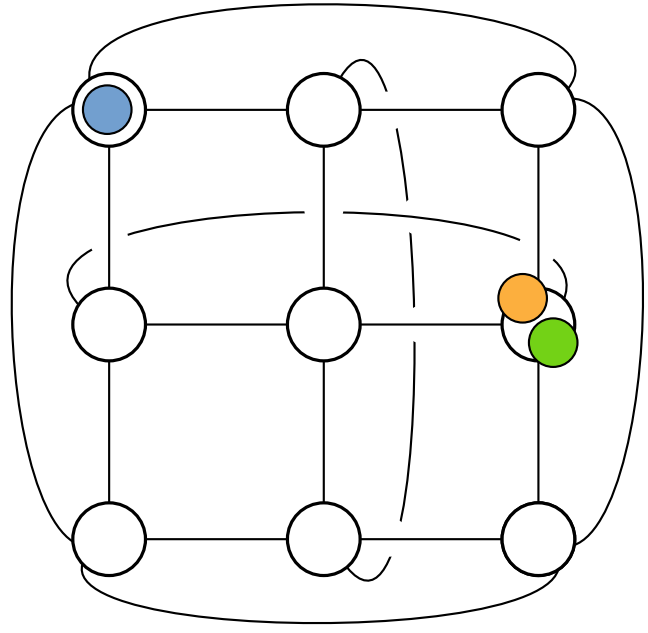
Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



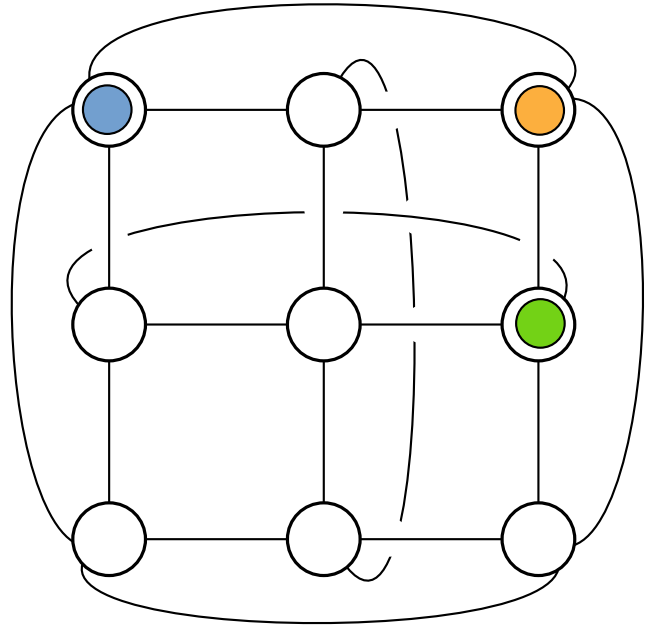
$d(\bullet, \bullet) = \infty$

$d(\bullet, \bullet) = (0, 0)$

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



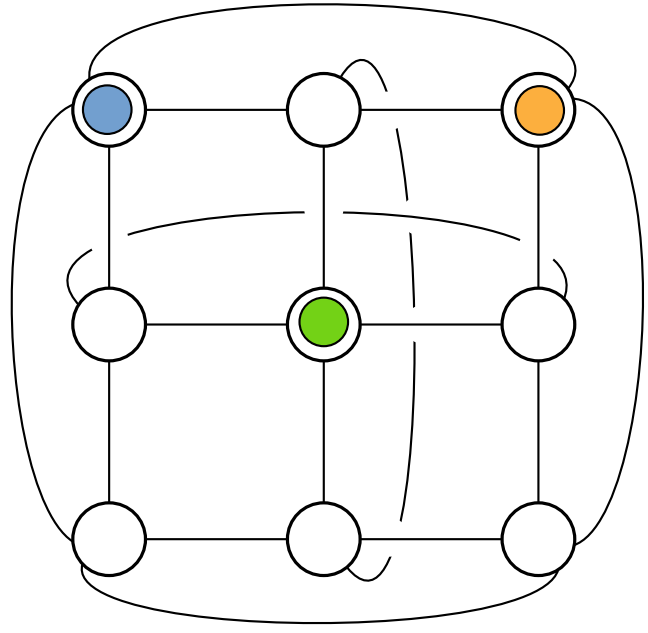
$d(\bullet, \bullet) = \infty$

$d(\bullet, \bullet) = (0, 1)$

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



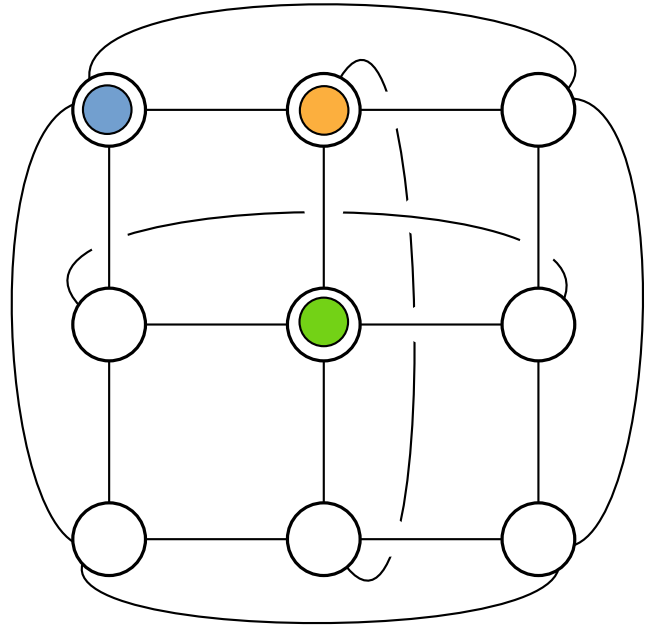
$d(\bullet, \bullet) = \infty$

$d(\bullet, \bullet) = (1, 1)$

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



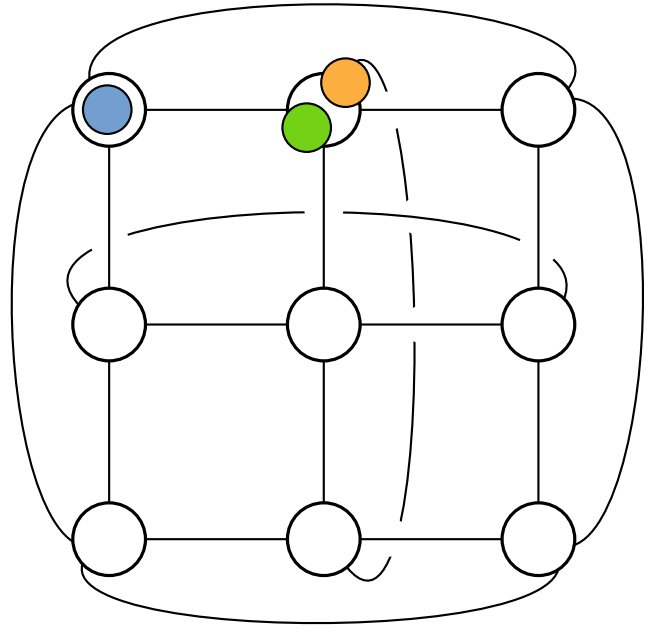
$d(\bullet, \bullet) = \infty$

$d(\bullet, \bullet) = (0, 1)$

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.



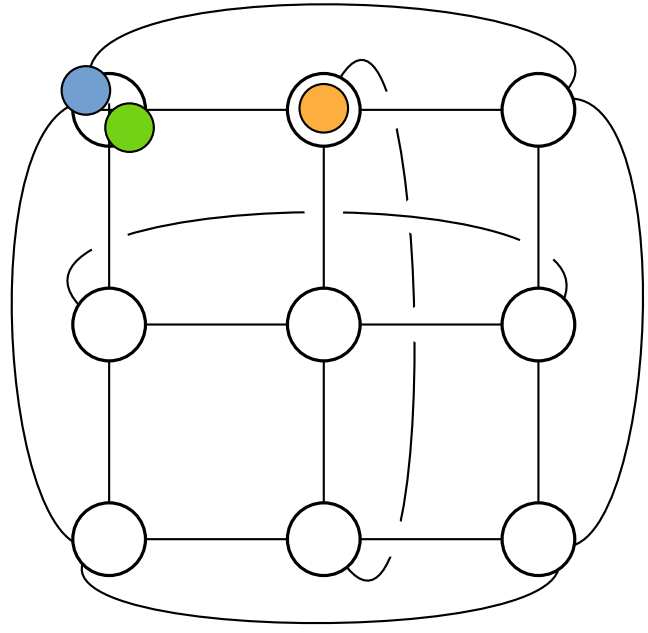
$d(\bullet, \bullet) = \infty$

$d(\bullet, \bullet) = (0, 0)$

Predicting Pebble Collisions

Some pebble collisions can be predicted from the computation history.

Keep track of known relative pebble distances (difference vectors throughout the course of the computation.

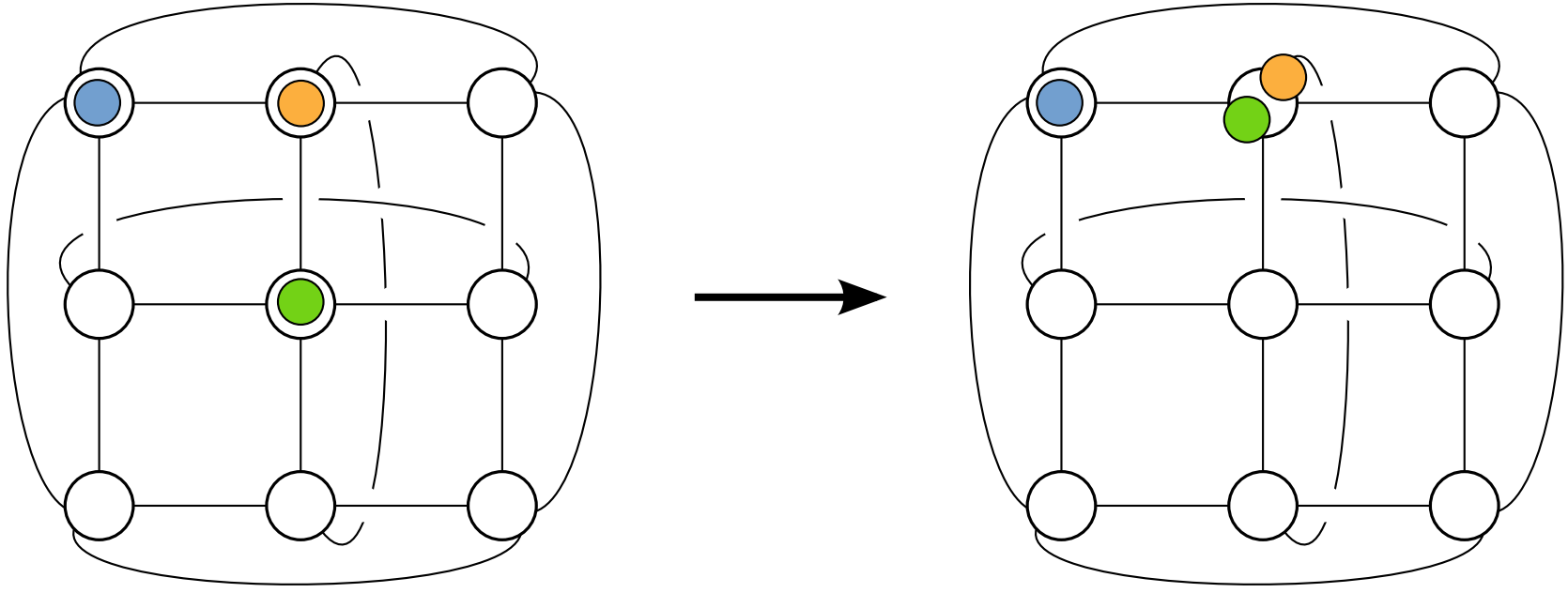


$d(\bullet, \bullet) = (0, 0)$

$d(\bullet, \bullet) = (1, 0)$

Predictable Steps

no collision of pebbles with unknown distance



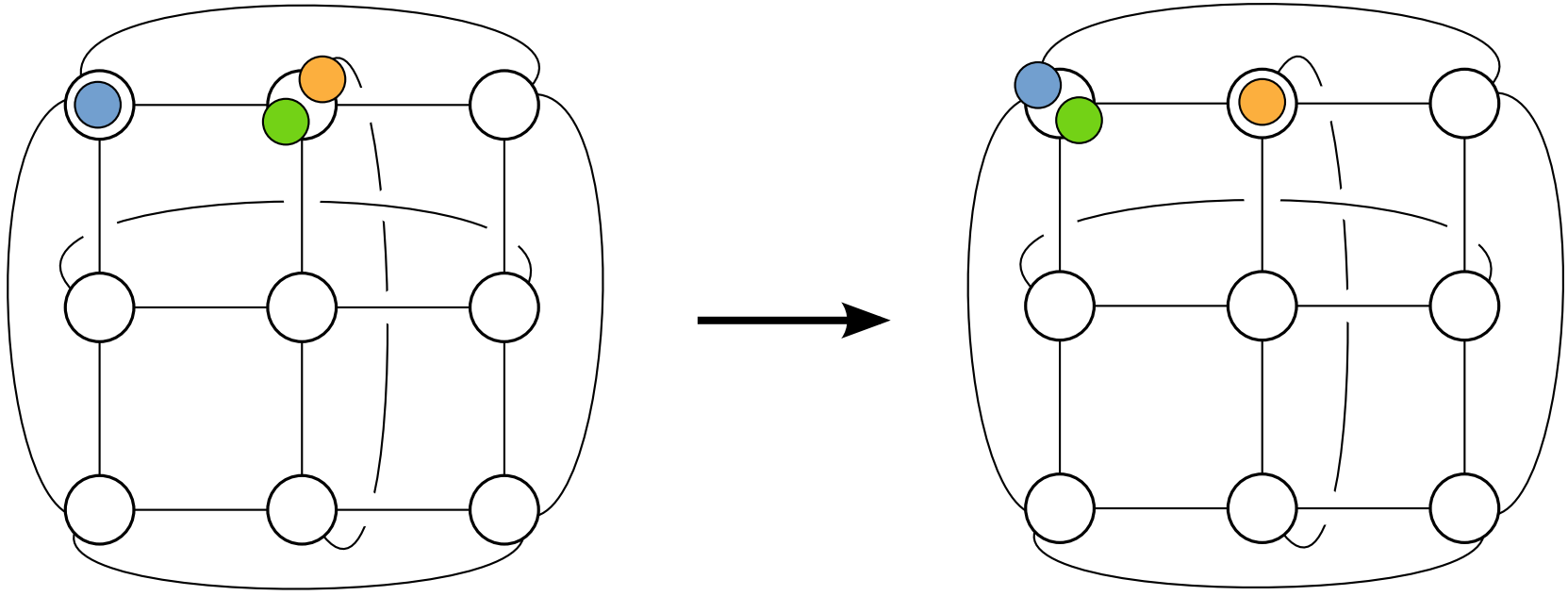
$d(\bullet,\bullet) = \infty$
$d(\bullet,\bullet) = (0,1)$

$d(\bullet,\bullet) = \infty$
$d(\bullet,\bullet) = (0,0)$

State and distance vectors after the step are uniquely determined from those before the step.

Non-Predictable Steps

collision of pebbles with unknown distance



$d(\bullet, \bullet) = \infty$
$d(\bullet, \bullet) = (0, 0)$

$d(\bullet, \bullet) = (0, 0)$
$d(\bullet, \bullet) = (1, 0)$

Any run contains at most $|P|$ non-predictable steps.

Extended State

Extended State:

State of JAG + knowledge about relative distances

Extended Behaviour:

List of extended states the JAG assumes
from a starting configuration

Establish a bound on the number of extended behaviours.

Analysis

Induction on the number of non-predictable steps

$C \equiv_n D \iff$ JAG has the same extended behaviour up to the $n+1$-th non-predicable step from both configurations C and D

$R_n =$ Number of equivalence classes of $\equiv_n$

$A_n =$ maximal number of distinct configurations in a run with $\leq n$ non-predictable steps

Analysis

$$A_n \leq R_n \cdot (1 + |P| + m|P|!)$$
$$R_0 \leq Q \cdot P^P$$
$$R_{n+1} \leq R_n \cdot (2 + A_n \cdot P^P) \cdot P^P$$

In the graph $G_i(m)$, a JAG with state set Q and pebble set P can reach at most $(|Q|m)^{c^{|P|}}$ nodes from any starting configuration.

Generalisation

In any *free action graph* with exponent m, a JAG with state set Q and pebble set P can reach at most $(|Q|m)^{c^{|P|}}$ nodes from any starting configuration.

Action graph $A = (V, D, \vec{g}, \odot)$

V — finite set of nodes

D — finite group

$\odot$ — free group action of D on V

$g_1, \ldots, g_d$ — group elements

Edges between v and $v \odot g_i$ for all $v \in V$ and $i = 1, \ldots, d$

Action Graphs

Graphs $G_i(m)$:

$$V = D = (\mathbb{Z}/m\mathbb{Z})^i$$

pointwise addition modulo m

Graphs $\Lambda_i(m)$:

i-fold lamplighter construction

$$V = D = (\mathbb{Z}/m\mathbb{Z}) \wr \underbrace{(\mathbb{Z}/2\mathbb{Z}) \wr \cdots \wr (\mathbb{Z}/2\mathbb{Z})}_{i \text{ times}}$$

The Formalisation

Formalisation in Coq

Coq is a very good tool to formalise this kind of proofs.

Coq

- dependent types, expressive programming language
- intuitionistic logic
- intensional equality

to formalise:

- proofs with classical logic
- counting arguments

Proof by Programming

How to to formalise the counting arguments?

- data types often finite with decidable equalit (here: graphs, configurations
- write program that enumerates all objects with a certain property (as a list)
- prove correctness of the program

→ proof by programming

Coq is perhaps the best currently available tool for combining programs and proofs.

Reflection

Combine logical inference with program evaluation

- `bool`-valued functions instead of predicates

```
f : A -> bool
```

- combine `f` with logical property `P`

```
f(x) = true -> P(x)
f(x) = false -> ~P(x)
```

- to show `P(M)` it suffices to show `f(M) = true`
- ideally: `f(M)` evaluates to `true`
- to show `P(M)` it then suffices to show `true = true`

Reflection

Reflection can be seen as a mechanism of writing tactic within the logic

- Example: `f` implements a decision procedure for `P`

  `f`(φ): decides validity of φ using by computing the truth table

  `P`(φ): provability of φ in the logic of Coq

- automatic simplification of expressions by computation

Small Scale Reflection in Coq

SSReflect in Co

Geoges Gonthier, proof of the four colour theorem

- Reflection is useful on a small scale
- Library for working with finite data type
- Concise tactic language
  - Implicit coercions (e.g. can use `f` like a logical predicate)
  - Views (switch between predicates and functions)
  - Rewriting

Very well suited for proving by programming.

Finite Data Types

Finite data types are easy to work with

- excellent existing library
- typical example:

  $|A \cup B| + |A \cap B| = |A| + |B|$ has 3-line proof

- complicated types easy to integrate

finite function

- represented by graph
- coercion to and from normal functions
- at most $|Y|^{|X|}$ functions from X to Y

finite equivalence relation

- represented by choice function
- coercion to and from normal relations
- quotients

Classical Logic, Extensionality

Intuitionistic logic, intensional equality not an issue.

- classical reasoning for `bool`-valued predicates

```
Lemma pigeon: forall (d1 d2:finType) (f:d1->d2)
   (card d2 < card d1) -> (exists x:d1, exists y:d1, (x != y) &&  (f x == f y)).
Proof.
move=>d1 d2 f I.
case: (pickP (fun xy => ((xy.1) != (xy.2)) &&  (f (xy.1) == f (xy.2))))=>[[x y] /=m|
 by exists x; exists y.
 have V: (injective f).
  move=>x y E.
  move: (e (x, y))=>/=; move/nandP=>[H1|H2].
  by apply/eqP; apply: negbE2.
  by rewrite E eq_refl in H2
 have W: (card d1) <= (card d2) by apply: (injective_card V).
 have C: (card d1) < (card d1).
  by rewrite -(leq_add2l 1) !add1n in W; apply: (leq_trans W I).
 by rewrite ltnn in C.
Qed.
```

- extensional, decidable equality for finite functions
- quotients of finite equivalence relations

Formalising the Analysis

$C \equiv_n D \iff$ JAG has the same extended behaviour up to the $n+1$-th non-predicable step from both configurations C and D

```
equiv (k n: nat): (conf ...) -> (conf ...) -> bool
```

$R_n =$ Number of equivalence classes of $\equiv_n$

```
Definition R (k n : nat): nat :
  card (quotient (equiv k n)).
```

$A_n =$ maximal number of distinct configurations in a run with $\leq n$ non-predictable steps

```
Definition A (k n : nat): nat :
   max (fun C=> card (evolution_to_coal k n C)).
```

Formalisation

- ~5000 lines of code in SSReflect tactic language
- quite close to the informal proof
- reasonable amount of additional work for defining finite functions, quotients, ...
- reflection essential
- SSReflect and libraries very well suited to task
- arithmetic calculations done by hand

Conclusion

Formal proof in structural complexity theory

- verification down to the last detail
- structuring proofs
- encapsulation of technical details in abstractions
- reflection works very well for this kind of proofs

Endliche Datentypen

Typen mit entscheidbarer Gleichheit

```
Record eqType : Type := EqType {
  sort :> Type;
  eq : sort -> sort -> bool;
  _ : forall x y:sort, reflect (x=y) (eq x y)
}.
```

Endliche Typen

```
Record finType : Type := FinType
  sort :> eqType;
  enum : seq sort;
  enumP : forall x, count (pred1 x) enum = 1
}.
```