KU LEUVEN kulak fwo Research Foundation Flanders Opening new horizons

On vertex-girth-regular graphs: (non-)existence, bounds and enumeration

Jorik Jooken
KU Leuven
Joint work: Robert Jajcay and István Porupsánszki

0 Outline

1 Outline

1 Motivating problem: the cage problem

Definition

For integers $v, k, g \geq 3$, a (v, k, g) graph is a graph with v vertices (*order* v) such that the length of the shortest cycle is g (the *girth* is g) and every vertex has degree k.

1 Motivating problem: the cage problem

Definition

For integers $v, k, g \geq 3$, a (v, k, g) graph is a graph with v vertices (*order* v) such that the length of the shortest cycle is g (the *girth* is g) and every vertex has degree k.

It is known that for all integers $k, g \geq 3$ a (v, k, g) graph exists for some v.

1 Motivating problem: the cage problem

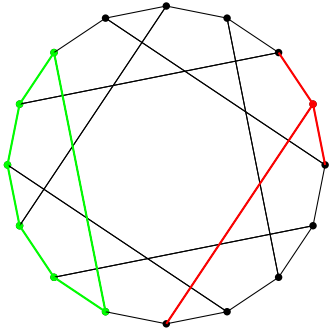
Definition

For integers $v, k, g \geq 3$, a (v, k, g) graph is a graph with v vertices (*order* v) such that the length of the shortest cycle is g (the *girth* is g) and every vertex has degree k.

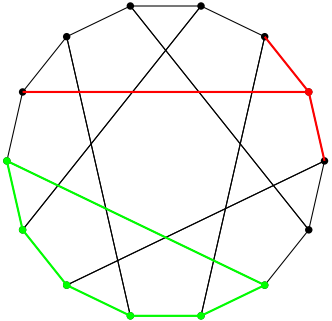
It is known that for all integers $k, g \geq 3$ a (v, k, g) graph exists for some v.

Cage problem: for given integers k and g, find the smallest v such that a (v, k, g) graph exists (denoted by $n(k, g)$) as well as the corresponding graphs, called (k, g)-*cages*.

1 Examples: Möbius-Kantor graph and Heawood graph



(a) A $(16, 3, 6)$ graph (Möbius-Kantor)



(b) The $(14, 3, 6)$ cage (Heawood).

We have $\mathbf{n(3, 6) = 14}$.

1 The cage problem

Important, but notoriously difficult problem: cages are only known for small k and g.

- Best upper bounds on $n(k, g)$: either constructions or graphs that were found by a computer.

1 The cage problem

Important, but notoriously difficult problem: cages are only known for small k and g.

- Best upper bounds on $n(k, g)$: either constructions or graphs that were found by a computer.
- Best lower bounds on $n(k, g)$: either mathematical formulas by reasoning about the structure of such graphs or based on algorithmic searches.

1 The cage problem

Important, but notoriously difficult problem: cages are only known for small k and g.

- Best upper bounds on $n(k, g)$: either constructions or graphs that were found by a computer.
- Best lower bounds on $n(k, g)$: either mathematical formulas by reasoning about the structure of such graphs or based on algorithmic searches.

E.g., it is known for decades that $202 \leq n(3, 13) \leq 272$.

1 The cage problem

Important, but notoriously difficult problem: cages are only known for small k and g.

- Best upper bounds on $n(k, g)$: either constructions or graphs that were found by a computer.
- Best lower bounds on $n(k, g)$: either mathematical formulas by reasoning about the structure of such graphs or based on algorithmic searches.

E.g., it is known for decades that $202 \leq n(3, 13) \leq 272$.

- **How to progress?** Improve fundamental understanding by studying strongly related problems.

1 Heuristics

- Most (k, g)-cages and smallest known (k, g)-graphs 'behave nicely'.

1 Heuristics

- Most (k, g)-cages and smallest known (k, g)-graphs 'behave nicely'.
- Therefore, a heuristic is to only consider classes of (k, g)-graphs that 'behave nicely'$\rightarrow$ the heuristic then makes precise what 'behave nicely' means

1 Heuristics

- Most (k, g)-cages and smallest known (k, g)-graphs 'behave nicely'.
- Therefore, a heuristic is to only consider classes of (k, g)-graphs that 'behave nicely'$\rightarrow$ the heuristic then makes precise what 'behave nicely' means

k	g	$n(k, g)$	vertex-transitive	Cayley
3	6	**14**	**14**	**14**
3	7	**24**	26	30
3	8	**30**	**30**	42
3	9	**58**	60	60
3	10	**70**	80	96

1 Heuristics

- The bias introduced by the heuristic determines how far one can go

1 Heuristics

- The bias introduced by the heuristic determines how far one can go
- For example, all cubic graphs have been generated until ≈ 32 vertices, whereas this is ≈ 5000 for cubic Cayley graphs and ≈ 1280 for cubic vertex-transitive graphs

1 Heuristics

- The bias introduced by the heuristic determines how far one can go
- For example, all cubic graphs have been generated until ≈ 32 vertices, whereas this is ≈ 5000 for cubic Cayley graphs and ≈ 1280 for cubic vertex-transitive graphs
- The stronger the bias, the further we can go, but it also makes it less likely to have a very strong connection with the original cage problem

1 Heuristics

- The bias introduced by the heuristic determines how far one can go
- For example, all cubic graphs have been generated until ≈ 32 vertices, whereas this is ≈ 5000 for cubic Cayley graphs and ≈ 1280 for cubic vertex-transitive graphs
- The stronger the bias, the further we can go, but it also makes it less likely to have a very strong connection with the original cage problem
- We aim to study classes of graphs of 'just the right level of bias'

1 Vertex-girth-regular graphs

Observation: many (k, g)-cages have the property that each vertex is contained in the same number of cycles of length g (*girth cycles*).

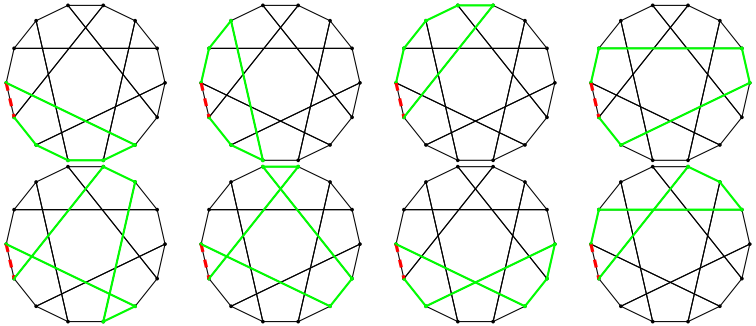


Figure: The $(3, 6)$-cage: each edge is contained in 8 girth cycles and thus each vertex is contained in 12 girth cycles.

1 Vertex-girth-regular graphs

Def.: a *vertex-girth-regular* (v, k, g, λ) graph (abbreviated as $\mathrm{vgr}(v, k, g, \lambda)$) is a (v, k, g) graph in which every vertex is contained in λ girth cycles.

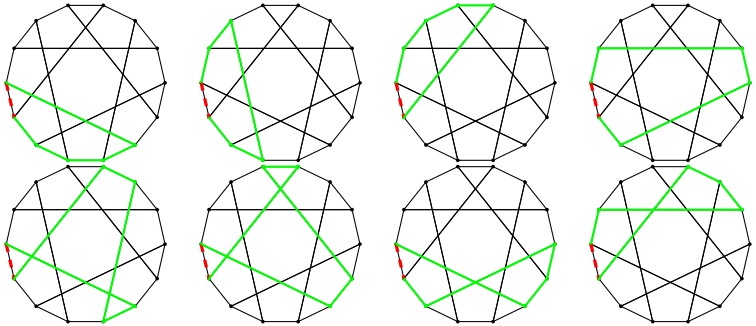


Figure: The $(3, 6)$-cage: each edge is contained in 8 girth cycles and thus each vertex is contained in 12 girth cycles.

1 Vertex-girth-regular graphs

Goal: this motivates us to study $n(k, g, \lambda)$, which is the smallest order v such that a $\mathrm{vgr}(v, k, g, \lambda)$ graph exists (e.g. $n(3, 6, 12) = 14$).

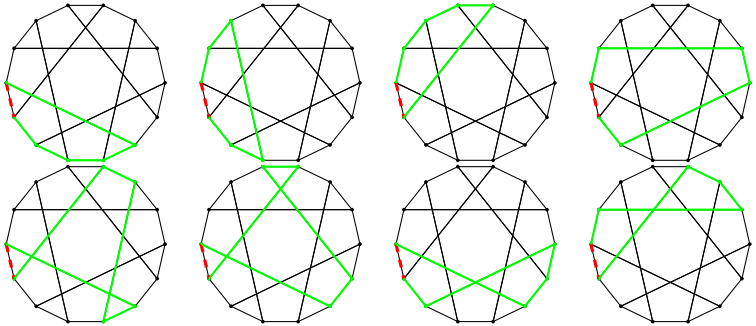


Figure: The $(3, 6)$-cage: each edge is contained in 8 girth cycles and thus each vertex is contained in 12 girth cycles.

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs
- For a vertex u of degree k, let $\mathbf{a}(u) = \{a_1, a_2, \ldots, a_k\}$ be the number of times the k edges adjacent to u are contained in a cycle of length g

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs
- For a vertex u of degree k, let $\mathbf{a}(u) = \{a_1, a_2, \ldots, a_k\}$ be the number of times the k edges adjacent to u are contained in a cycle of length g
- Edge-girth-regular graphs: $a_1 = a_2 = \ldots = a_k$ for all vertices u

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs
- For a vertex u of degree k, let $\mathbf{a}(u) = \{a_1, a_2, \ldots, a_k\}$ be the number of times the k edges adjacent to u are contained in a cycle of length g
- Edge-girth-regular graphs: $a_1 = a_2 = \ldots = a_k$ for all vertices u
- Girth-regular graphs: the multisets $\mathbf{a}(u)$ and $\mathbf{a}(v)$ are the same for all vertices u, v

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs
- For a vertex u of degree k, let $\mathbf{a}(u) = \{a_1, a_2, \ldots, a_k\}$ be the number of times the k edges adjacent to u are contained in a cycle of length g
- Edge-girth-regular graphs: $a_1 = a_2 = \ldots = a_k$ for all vertices u
- Girth-regular graphs: the multisets $\mathbf{a}(u)$ and $\mathbf{a}(v)$ are the same for all vertices u, v
- Vertex-girth-regular graphs: the sum of entries in $\mathbf{a}(u)$ equals the sum of entries in $\mathbf{a}(v)$ for all vertices u, v

1 Relationship with other classes of graphs

Other classes of regular graphs received earlier attention in literature:

- Cayley graphs and vertex-transitive graphs
- For a vertex u of degree k, let $\mathbf{a}(u) = \{a_1, a_2, \ldots, a_k\}$ be the number of times the k edges adjacent to u are contained in a cycle of length g
- Edge-girth-regular graphs: $a_1 = a_2 = \ldots = a_k$ for all vertices u
- Girth-regular graphs: the multisets $\mathbf{a}(u)$ and $\mathbf{a}(v)$ are the same for all vertices u, v
- Vertex-girth-regular graphs: the sum of entries in $\mathbf{a}(u)$ equals the sum of entries in $\mathbf{a}(v)$ for all vertices u, v
- **Vertex-girth-regular graphs are the most general of all these**

2 Outline

2 λ is bounded for fixed k, g

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists

2 λ is bounded for fixed k, g

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists
- One may therefore wonder whether for all k, g, λ a $vgr(v, k, g, \lambda)$ graph exists

2 λ is bounded for fixed k, g

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists
- One may therefore wonder whether for all k, g, λ a $vgr(v, k, g, \lambda)$ graph exists
- This is not immediately clear and we will focus on existence and non-existence now

2 λ is bounded for fixed k, g

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists
- One may therefore wonder whether for all k, g, λ a $vgr(v, k, g, \lambda)$ graph exists
- This is not immediately clear and we will focus on existence and non-existence now

Proposition (Jajcay, J., Porupsánszki)

Let G be a $\mathrm{vgr}(v, k, g, \lambda)$-graph. Then $\lambda \leq nc(k, g) := \frac{k(k-1)^{\lfloor \frac{g}{2} \rfloor}}{2}$ with equality if and only if G is a Moore graph.

2 Proof upper bound on λ

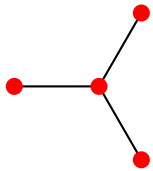
Assume g is odd, take for example $k = 3$ and $g = 5$.

2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.

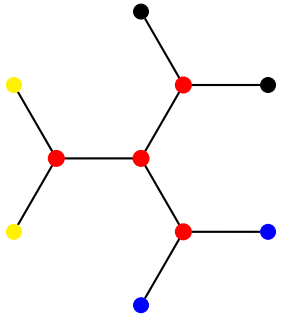
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



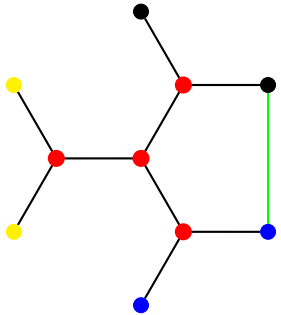
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



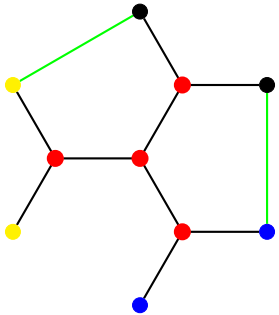
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



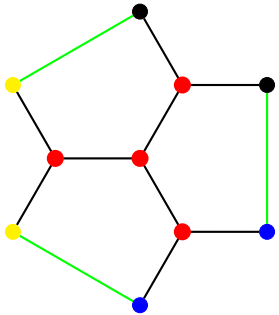
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



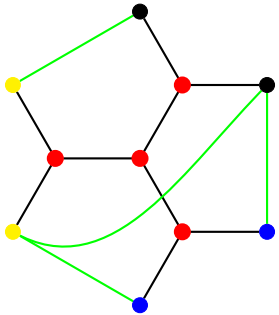
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



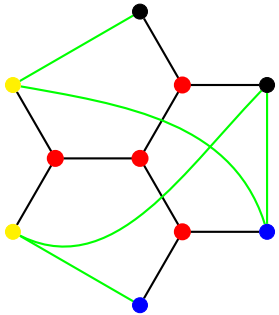
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



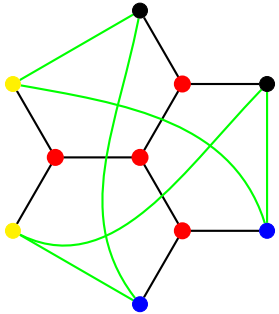
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



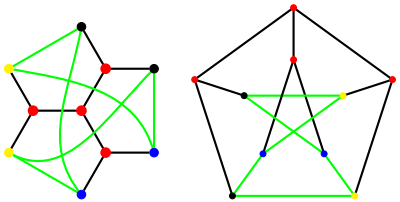
2 Proof upper bound on λ

Assume g is odd, take for example $k = 3$ and $g = 5$.



2 Proof upper bound on λ

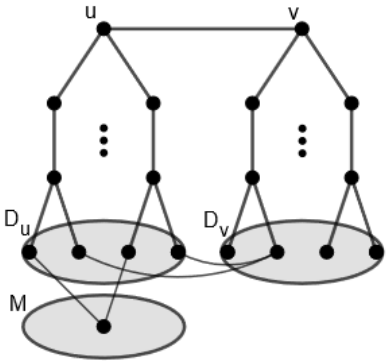
This graph is in fact isomorphic with the Petersen graph ($k = 3$, $g = 5$, $\lambda = 6$).



2 Proof upper bound on λ

Assume g is even. Girth cycles containing u can be partitioned into 3 sets.

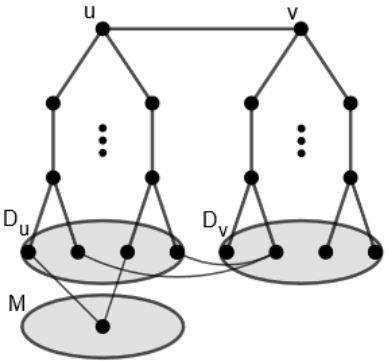
- $\mathcal{A}_u$: girth cycle contains exactly one edge with one endpoint in D_u and other endpoint in D_v



2 Proof upper bound on λ

Assume g is even. Girth cycles containing u can be partitioned into 3 sets.

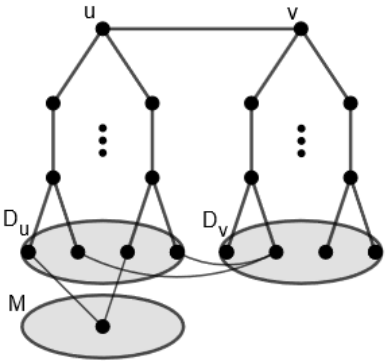
- $\mathcal{A}_u$: girth cycle contains exactly one edge with one endpoint in D_u and other endpoint in D_v
- $\mathcal{B}_u$: girth cycle contains exactly two edges with one endpoint in D_u and a shared endpoint in D_v



2 Proof upper bound on λ

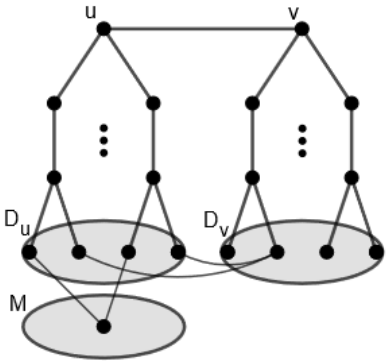
Assume g is even. Girth cycles containing u can be partitioned into 3 sets.

- $\mathcal{A}_u$: girth cycle contains exactly one edge with one endpoint in D_u and other endpoint in D_v
- $\mathcal{B}_u$: girth cycle contains exactly two edges with one endpoint in D_u and a shared endpoint in D_v
- $\mathcal{C}_u$: girth cycle contains exactly two edges with one endpoint in D_u and a shared endpoint in M



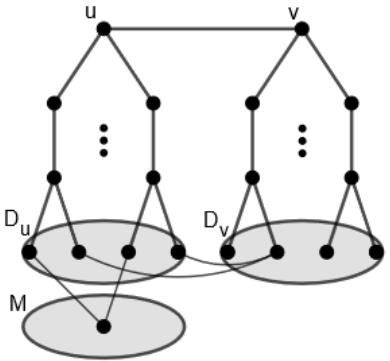
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u



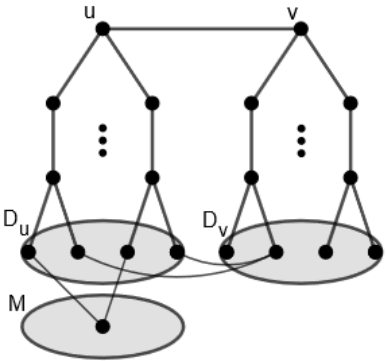
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u
- For each vertex $w \in D_v$, let y_w be the number of neighbors of w in the set D_u



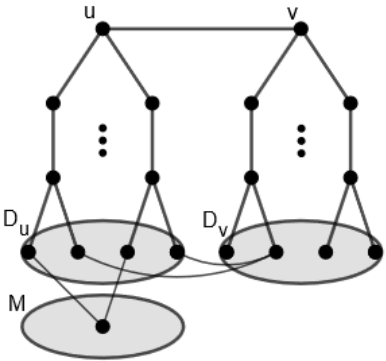
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u
- For each vertex $w \in D_v$, let y_w be the number of neighbors of w in the set D_u
- $|D_u| = |D_v| = (k-1)^{\frac{g-2}{2}}$



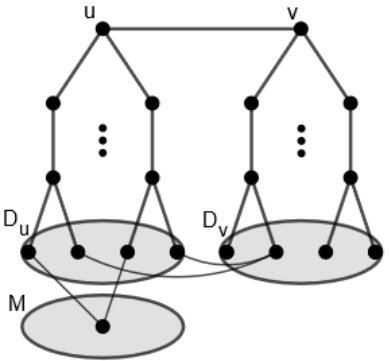
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u
- For each vertex $w \in D_v$, let y_w be the number of neighbors of w in the set D_u
- $|D_u| = |D_v| = (k-1)^{\frac{g-2}{2}}$
- Every vertex has degree k, so $|\mathcal{A}_u| = \sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}}$



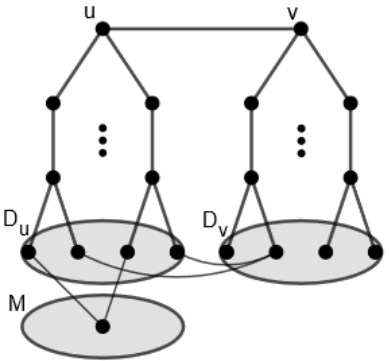
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u
- For each vertex $w \in D_v$, let y_w be the number of neighbors of w in the set D_u
- $|D_u| = |D_v| = (k-1)^{\frac{g-2}{2}}$
- Every vertex has degree k, so $|\mathcal{A}_u| = \sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}}$
- $|\mathcal{B}_u| = \sum_{w \in D_v} \binom{y_w}{2}$



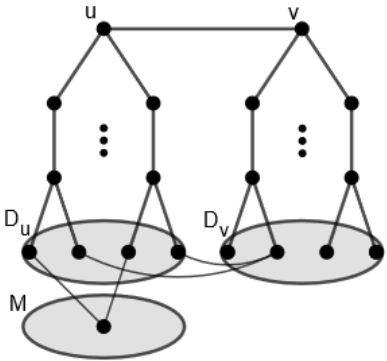
2 Proof upper bound on λ

- For each vertex $w \in M$, let x_w be the number of neighbors of w in the set D_u
- For each vertex $w \in D_v$, let y_w be the number of neighbors of w in the set D_u
- $|D_u| = |D_v| = (k-1)^{\frac{g-2}{2}}$
- Every vertex has degree k, so $|\mathcal{A}_u| = \sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}}$
- $|\mathcal{B}_u| = \sum_{w \in D_v} \binom{y_w}{2}$
- $|\mathcal{C}_u| = \sum_{w \in M} \binom{x_w}{2}$



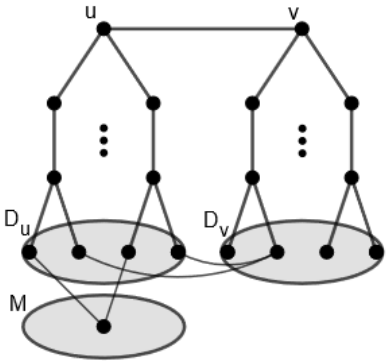
2 Proof upper bound on λ

- Every vertex is incident with at most $k-1$ vertices in D_u, since otherwise one would obtain too short cycles



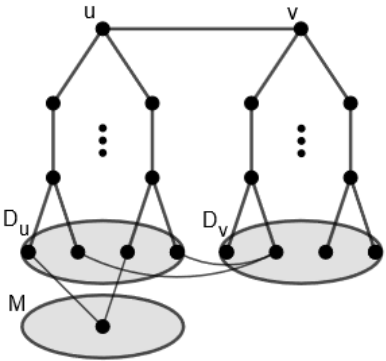
2 Proof upper bound on λ

- Every vertex is incident with at most $k-1$ vertices in D_u, since otherwise one would obtain too short cycles
- Therefore, $|\mathcal{B}_u| + |\mathcal{C}_u| \leq (k-1)^{\frac{g-2}{2}} \frac{(k-1)(k-2)}{2}$



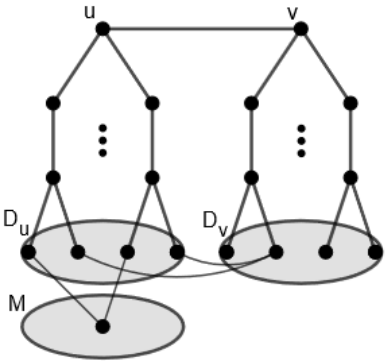
2 Proof upper bound on λ

- Every vertex is incident with at most $k-1$ vertices in D_u, since otherwise one would obtain too short cycles
- Therefore, $|\mathcal{B}_u| + |\mathcal{C}_u| \leq (k-1)^{\frac{g-2}{2}} \frac{(k-1)(k-2)}{2}$
- Summing everything yields the upper bound: $\lambda \leq nc(k,g)$



2 Proof upper bound on λ

- Every vertex is incident with at most $k-1$ vertices in D_u, since otherwise one would obtain too short cycles
- Therefore, $|\mathcal{B}_u| + |\mathcal{C}_u| \leq (k-1)^{\frac{g-2}{2}} \frac{(k-1)(k-2)}{2}$
- Summing everything yields the upper bound: $\lambda \leq nc(k,g)$
- Equality can only occur when $|\mathcal{A}_u|$ and $|\mathcal{B}_u|$ are maximal, in which case the graph must be a Moore graph



2 Non-existence

- A similar set-up can be used to show that a $vgr(v, k, g, \lambda)$ graph does not exist when λ is too close (but not equal) to its maximal value (i.e., $nc(k, g) := \frac{k(k-1)^{\lfloor \frac{g}{2} \rfloor}}{2}$)

2 Non-existence

- A similar set-up can be used to show that a $vgr(v, k, g, \lambda)$ graph does not exist when λ is too close (but not equal) to its maximal value (i.e., $nc(k, g) := \frac{k(k-1)^{\lfloor \frac{g}{2} \rfloor}}{2}$)
- We demonstrate the case where g is even

2 Non-existence

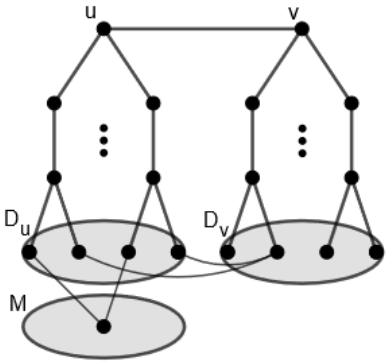
- A similar set-up can be used to show that a $vgr(v, k, g, \lambda)$ graph does not exist when λ is too close (but not equal) to its maximal value (i.e., $nc(k, g) := \frac{k(k-1)^{\lfloor \frac{g}{2} \rfloor}}{2}$)
- We demonstrate the case where g is even

Theorem (Jajcay, J., Porupsánszki)

Let $k \geq 3$, $g = 2s \geq 4$, and $0 < \epsilon < k - 1$ be integers. Then there is no $\mathrm{vgr}(n, k, g, nc(k, g) - \epsilon)$-graph.

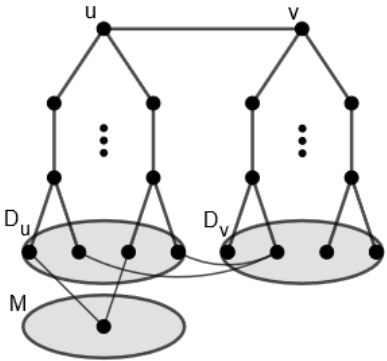
2 Non-existence proof

- Since $\epsilon > 0$, at least one horizontal edge is missing



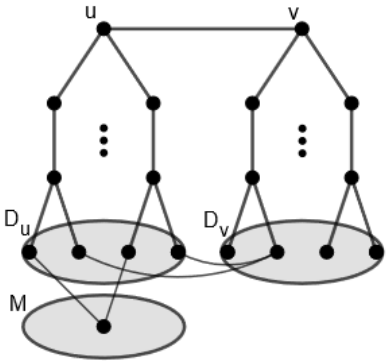
2 Non-existence proof

- Since $\epsilon > 0$, at least one horizontal edge is missing
- How large can the number of girth cycles be now?



2 Non-existence proof

Maximize:
$\sum_{w \in D_v} y + \sum_{w \in D_v} \binom{y_w}{2} + \sum_{w \in M} \binom{x_w}{2}$
Subject to:



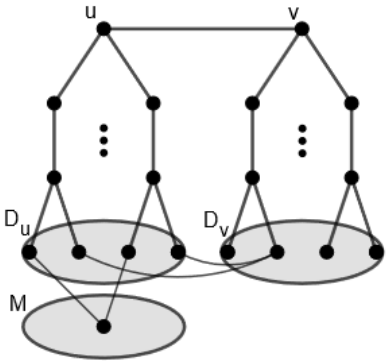
2 Non-existence proof

Maximize:

$\sum_{w \in D_v} y + \sum_{w \in D_v} \binom{y_w}{2} + \sum_{w \in M} \binom{x_w}{2}$

Subject to:

- $\sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}} - 1$



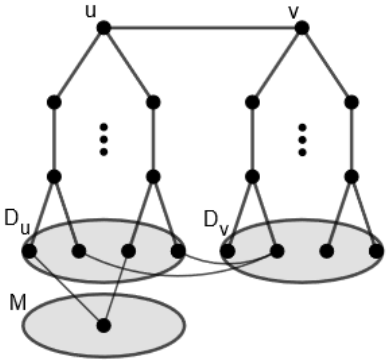
2 Non-existence proof

Maximize:

$\sum_{w \in D_v} y + \sum_{w \in D_v} \binom{y_w}{2} + \sum_{w \in M} \binom{x_w}{2}$

Subject to:

- $\sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}} - 1$
- $0 \leq x_w \leq k-1$



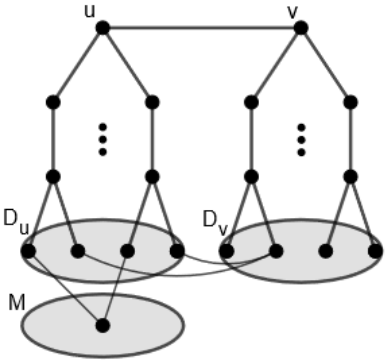
2 Non-existence proof

Maximize:

$\sum_{w \in D_v} y + \sum_{w \in D_v} \binom{y_w}{2} + \sum_{w \in M} \binom{x_w}{2}$

Subject to:

- $\sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}} - 1$
- $0 \leq x_w \leq k-1$
- $0 \leq y_w \leq k-1$



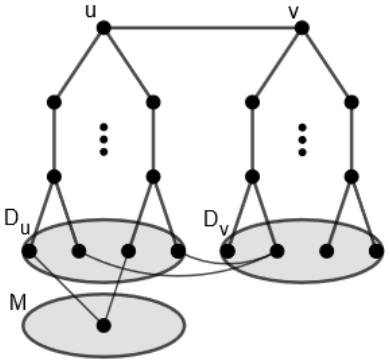
2 Non-existence proof

Maximize:

$\sum_{w \in D_v} y + \sum_{w \in D_v} \binom{y_w}{2} + \sum_{w \in M} \binom{x_w}{2}$

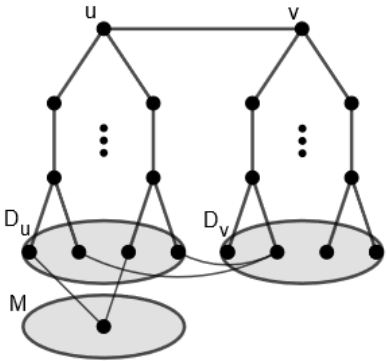
Subject to:

- $\sum_{w \in D_v} y_w \leq (k-1)^{\frac{g}{2}} - 1$
- $0 \leq x_w \leq k-1$
- $0 \leq y_w \leq k-1$
- $\sum_{w \in D_v} y_w + \sum_{w \in M} x_w \leq (k-1)^{\frac{g}{2}}$



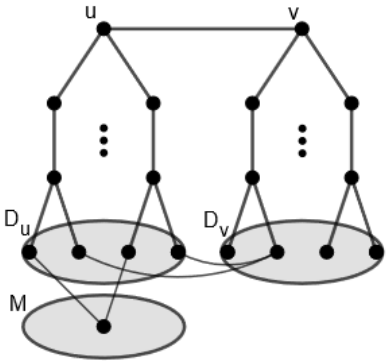
2 Non-existence proof

- This expression is maximized when $y_w = k - 1$ for all except for one $w \in D_v$, in which case $y_w = k - 2$.



2 Non-existence proof

- This expression is maximized when $y_w = k - 1$ for all except for one $w \in D_v$, in which case $y_w = k - 2$.
- Filling in the numbers then completes the proof



2 Lower bounds on $n(k, g, \lambda)$

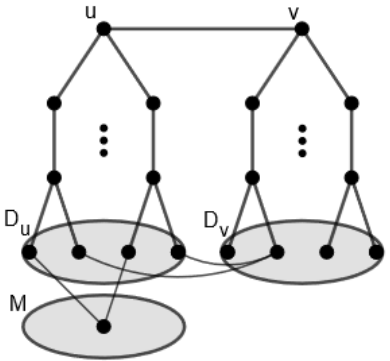
- We also prove lower bounds on $n(k, g, \lambda)$: some combinatorial ones and a spectral one

2 Lower bounds on $n(k, g, \lambda)$

- We also prove lower bounds on $n(k, g, \lambda)$: some combinatorial ones and a spectral one
- I will only briefly comment on one of the combinatorial bounds (without going into detail) for the even girth case

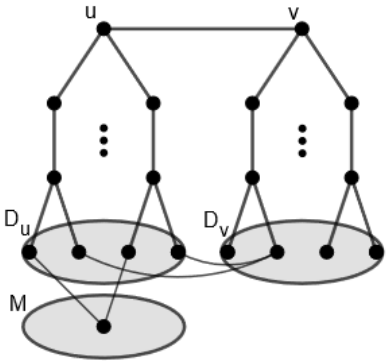
2 Lower bounds on $n(k, g, \lambda)$

- In any $\mathrm{vgr}(v, k, g, \lambda)$ graph, $v \geq M(k, g) + |M|$



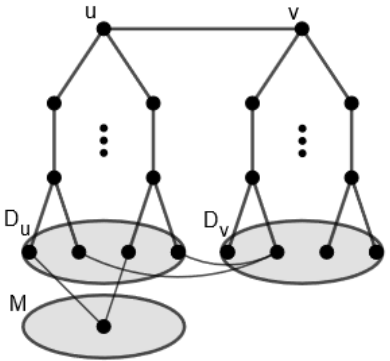
2 Lower bounds on $n(k, g, \lambda)$

- In any $\mathrm{vgr}(v, k, g, \lambda)$ graph, $v \geq M(k, g) + |M|$
- $nc(k, g) - \lambda$ tells us something about how many horizontal edges must at least be missing



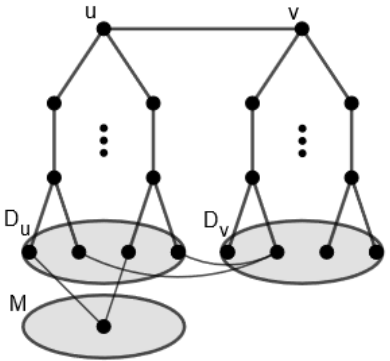
2 Lower bounds on $n(k, g, \lambda)$

- In any $\mathrm{vgr}(v, k, g, \lambda)$ graph, $v \geq M(k, g) + |M|$
- $nc(k, g) - \lambda$ tells us something about how many horizontal edges must at least be missing
- If λ is very small, many edges must exist between D_u and M



2 Lower bounds on $n(k, g, \lambda)$

- In any $\mathrm{vgr}(v, k, g, \lambda)$ graph, $v \geq M(k, g) + |M|$
- $nc(k, g) - \lambda$ tells us something about how many horizontal edges must at least be missing
- If λ is very small, many edges must exist between D_u and M
- However, it is impossible that many of these edges go to the same vertex in M, because this would yield many girth cycles



2 Existence results

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists

2 Existence results

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists
- Idea is based on generalized truncation: replace each vertex of degree d with a suitable graph of order d

2 Existence results

- Erdős and Sachs showed that for all $k, g \geq 3$, a (v, k, g) graph exists
- Idea is based on generalized truncation: replace each vertex of degree d with a suitable graph of order d
- Same underlying idea can be used to prove:

Proposition (Jajcay, J., Porupsánszki)

If a $\mathrm{vgr}(v, k, g, \lambda)$*-graph exists, then there exist infinitely many integers* v' *such that a* $\mathrm{vgr}(v', k+1, g, \lambda)$*-graph also exists.*

2 Edge-girth-regular graphs yield vertex-girth-regular graphs

Recall that an $\mathrm{egr}(v, k, g, \lambda)$ graph is a k-regular graph on v vertices with girth g such that every edge is contained in λ girth cycles

2 Edge-girth-regular graphs yield vertex-girth-regular graphs

Recall that an $\mathrm{egr}(v, k, g, \lambda)$ graph is a k-regular graph on v vertices with girth g such that every edge is contained in λ girth cycles

Observation. *If G is an $\mathrm{egr}(v, k, g, \lambda)$-graph, then G is a $\mathrm{vgr}(v, k, g, \frac{k\lambda}{2})$-graph.*

2 Edge-girth-regular graphs yield vertex-girth-regular graphs

Recall that an $\mathrm{egr}(v, k, g, \lambda)$ graph is a k-regular graph on v vertices with girth g such that every edge is contained in λ girth cycles

Observation. *If G is an $\mathrm{egr}(v, k, g, \lambda)$-graph, then G is a $\mathrm{vgr}(v, k, g, \frac{k\lambda}{2})$-graph.*

Several families of edge-girth-regular graphs are known, as well as operations yielding new edge-girth-regular graphs

2 Existence results

Combining generalized truncation with already existing ideas from edge-girth-regular graphs, one can obtain that vertex-girth-regular graphs must exist for many (k, g, λ):

Theorem (Jajcay, J., Porupsánszki)

There are infinitely many integers v such that a $\mathrm{vgr}(v, k, g, \lambda)$-graph exists:

(i) *for $\lambda = 1$ and all integers $k, g \geq 3$;*

(ii) *for $\lambda = 2$ and all integers $k \geq 4$, $g \geq 3$;*

(iii) *for all integers $\lambda \geq 3$, $k \geq \lambda$, $g \geq 3$, $g \notin \{4, 5\}$.*

3 Outline

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.
The smallest graph is clearly connected, hence we only focus on connected graphs in this talk.

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.
The smallest graph is clearly connected, hence we only focus on connected graphs in this talk.
Studying $n(k, g, \lambda)$ becomes easier if we have an algorithm for solving the following problem:

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.
The smallest graph is clearly connected, hence we only focus on connected graphs in this talk.
Studying $n(k, g, \lambda)$ becomes easier if we have an algorithm for solving the following problem:

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.
The smallest graph is clearly connected, hence we only focus on connected graphs in this talk.
Studying $n(k, g, \lambda)$ becomes easier if we have an algorithm for solving the following problem:

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

- If the algorithm found a $\mathrm{vgr}(v, k, g, \lambda)$ graph, then clearly $n(k, g, \lambda) \leq v$.

3 From $n(k, g, \lambda)$ to exhaustive generation

We are interested in studying $n(k, g, \lambda)$.
The smallest graph is clearly connected, hence we only focus on connected graphs in this talk.
Studying $n(k, g, \lambda)$ becomes easier if we have an algorithm for solving the following problem:

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

- If the algorithm found a $\mathrm{vgr}(v, k, g, \lambda)$ graph, then clearly $n(k, g, \lambda) \leq v$.
- If the algorithm did not find any $\mathrm{vgr}(v', k, g, \lambda)$ graph for all $v' < v$, then $n(k, g, \lambda) \geq v$, because the algorithm is exhaustive.

3 A naive approach

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

There exist efficient generation algorithms for exhaustively enumerating all k-regular graphs with girth g of order v.

3 A naive approach

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

There exist efficient generation algorithms for exhaustively enumerating all k-regular graphs with girth g of order v.
Naive idea: use such an algorithm and filter out those graphs which are vertex-girth-regular.

3 A naive approach

Problem

Given integers v, k, g and λ, enumerate all $\mathrm{vgr}(v, k, g, \lambda)$ graphs.

There exist efficient generation algorithms for exhaustively enumerating all k-regular graphs with girth g of order v.
Naive idea: use such an algorithm and filter out those graphs which are vertex-girth-regular.

Order	Nb. 3-regular	Nb. 3-regular and vgr
$n = 18$	$41,301$	10
$n = 20$	$510,489$	44
$n = 22$	$7,319,447$	3

3 Exhaustive enumeration algorithm

Algorithm for enumerating all $\mathrm{vgr}(v, k, g, \lambda)$ graphs:

- Start with a (k, g) Moore tree and add isolated vertices until there are v vertices (this graph must occur as a subgraph)

3 Exhaustive enumeration algorithm

Algorithm for enumerating all $\mathrm{vgr}(v, k, g, \lambda)$ graphs:

- Start with a (k, g) Moore tree and add isolated vertices until there are v vertices (this graph must occur as a subgraph)
- Recursively add edges in all possible ways between other vertices

3 Exhaustive enumeration algorithm

Algorithm for enumerating all $\mathrm{vgr}(v, k, g, \lambda)$ graphs:

- Start with a (k, g) Moore tree and add isolated vertices until there are v vertices (this graph must occur as a subgraph)
- Recursively add edges in all possible ways between other vertices

This basic strategy is extended with pruning rules and heuristics that are very important to obtain a fast algorithm.

3 Pruning rules and heuristics

Pruning rules:

- If the graph has girth smaller than g or a vertex contained in more than λ girth cycles: **prune**

3 Pruning rules and heuristics

Pruning rules:

- If the graph has girth smaller than g or a vertex contained in more than λ girth cycles: **prune**
- If there is a vertex that does not have enough edges that can be added to obtain degree k: **prune**

3 Pruning rules and heuristics

Pruning rules:

- If the graph has girth smaller than g or a vertex contained in more than λ girth cycles: **prune**
- If there is a vertex that does not have enough edges that can be added to obtain degree k: **prune**
- If an isomorphic graph was encountered before: **prune**

3 Pruning rules and heuristics

Pruning rules:

- If the graph has girth smaller than g or a vertex contained in more than λ girth cycles: **prune**
- If there is a vertex that does not have enough edges that can be added to obtain degree k: **prune**
- If an isomorphic graph was encountered before: **prune**

Heuristics:

- Order in which edges are added does not matter

3 Pruning rules and heuristics

Pruning rules:

- If the graph has girth smaller than g or a vertex contained in more than λ girth cycles: **prune**
- If there is a vertex that does not have enough edges that can be added to obtain degree k: **prune**
- If an isomorphic graph was encountered before: **prune**

Heuristics:

- Order in which edges are added does not matter
- Add next edge adjacent to vertex with fewest options (fail-first principle)

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

Naive idea: enumerate all girth cycles containing e using a backtracking algorithm and count.

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

Naive idea: enumerate all girth cycles containing e using a backtracking algorithm and count.
Disadvantage: not clear if and how to obtain polynomial running time using backtracking.

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

Naive idea: enumerate all girth cycles containing e using a backtracking algorithm and count.
Disadvantage: not clear if and how to obtain polynomial running time using backtracking.
Instead: linear time algorithm based on dynamic programming

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

Definition

Let $ngc(G, uv)$, $nsp(G, u, v)$ and $d(G, u, v)$ be respectively the number of girth cycles containing uv, number of shortest paths between u and v and distance between u and v in graph G.

3 Efficient algorithms for subproblems: girth cycles

Problem

For a graph G of girth g and edge $e \in E(G)$, determine the number of cycles of length g that contain e.

Definition

Let $ngc(G, uv)$, $nsp(G, u, v)$ and $d(G, u, v)$ be respectively the number of girth cycles containing uv, number of shortest paths between u and v and distance between u and v in graph G.

Observation. *Let u_1u_2 be an edge of the graph $G = (V, E)$ with girth g. Now $ngc(G, u_1u_2) = 0$ if $d(G - u_1u_2, u_1, u_2) > g - 1$ and $ngc(G, u_1u_2) = nsp(G - u_1u_2, u_1, u_2)$ otherwise.*

3 Efficient algorithms for subproblems: girth cycles

Observation. *Let* $D_i = \{u \in V(G) \mid d(G - u_1u_2, u, u_1) = i\}$ *be the set of vertices at distance* i *from* u_1 *in the graph* $G - u_1u_2$*. We now have the following base case (for* $u = u_1$*):*

$$nsp(G - u_1u_2, u_1, u_1) = 1$$

For a vertex $u \neq u_1$*, let* $d(G - u_1u_2, u, u_1) = i > 0$*. Since every shortest path of length* i *can be obtained by adding an edge to a shortest path of length* $i - 1$*, we have the following recursive case:*

$$nsp(G - u_1u_2, u, u_1) = \sum_{u' \in N_{G-u_1u_2}(u) \cap D_{i-1}} nsp(G - u_1u_2, u', u_1)$$

3 Efficient algorithms for subproblems: distance update

Problem

For a graph G, where $d(G, u, v)$ is known for each pair of vertices u, v, efficiently determine $d(G + ab, u, v)$ for each pair of vertices u, v when an edge ab is added.

3 Efficient algorithms for subproblems: distance update

Problem

For a graph G, where $d(G, u, v)$ is known for each pair of vertices u, v, efficiently determine $d(G + ab, u, v)$ for each pair of vertices u, v when an edge ab is added.

Idea 1: recalculate everything from scratch and use a standard all-pairs shortest path algorithm such as Floyd-Warshall ($O(|V|^3)$ time)

3 Efficient algorithms for subproblems: distance update

Problem

For a graph G, where $d(G, u, v)$ is known for each pair of vertices u, v, efficiently determine $d(G + ab, u, v)$ for each pair of vertices u, v when an edge ab is added.

Idea 1: recalculate everything from scratch and use a standard all-pairs shortest path algorithm such as Floyd-Warshall ($O(|V|^3)$ time)
Idea 2: $d(G + ab, u, v) =$
$\min(d(G, u, v), d(G, u, a) + 1 + d(G, b, v), d(G, u, b) + 1 + d(G, a, v))$.
Distance update in $O(1)$ time per pair u, v.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Can we just compare the adjacency matrices?

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

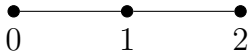
Can we just compare the adjacency matrices?

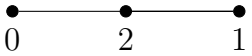
3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Can we just compare the adjacency matrices?





$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Idea: canonical form of graph, label graph in such a way that the adjacency matrix is lexicographically minimal.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Idea: canonical form of graph, label graph in such a way that the adjacency matrix is lexicographically minimal.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Idea: canonical form of graph, label graph in such a way that the adjacency matrix is lexicographically minimal.

$$\begin{bmatrix}0&1&0\\1&0&1\\0&1&0\end{bmatrix} \quad \begin{bmatrix}0&0&1\\0&0&1\\1&1&0\end{bmatrix} \quad \begin{bmatrix}0&1&1\\1&0&0\\1&0&0\end{bmatrix} \quad \begin{bmatrix}0&0&1\\0&0&1\\1&1&0\end{bmatrix} \quad \begin{bmatrix}0&1&1\\1&0&0\\1&0&0\end{bmatrix} \quad \begin{bmatrix}0&1&0\\1&0&1\\0&1&0\end{bmatrix}$$

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Use *nauty* for computing canonical form of graphs and store in appropriate data structure.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Use *nauty* for computing canonical form of graphs and store in appropriate data structure.
For example, array containing 10^6 canonical forms would require 10^6 comparisons in the worst case to solve the above problem.

3 Efficient algorithms for subproblems: isomorphism

Problem

For a graph G, determine if the recursive function was already called with a graph G' isomorphic with G.

Use *nauty* for computing canonical form of graphs and store in appropriate data structure.
For example, array containing 10^6 canonical forms would require 10^6 comparisons in the worst case to solve the above problem.
A standard data structure such as a splay tree only needs around 20 comparisons (in general: insertion and lookup in $O(\log(n))$ time).

3 Computational results

- We implemented the previous algorithm and investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) (complete table in the paper)

3 Computational results

- We implemented the previous algorithm and investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) (complete table in the paper)
- The algorithm was parallelized and ran on a supercomputer for about 2 CPU-years

3 Computational results

- We implemented the previous algorithm and investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) (complete table in the paper)
- The algorithm was parallelized and ran on a supercomputer for about 2 CPU-years
- We also searched for vertex-girth-regular graphs in known lists of graphs (e.g. vertex-transitive graphs) to obtain upper bounds for $n(k, g, \lambda)$

3 Computational results

- We implemented the previous algorithm and investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) (complete table in the paper)
- The algorithm was parallelized and ran on a supercomputer for about 2 CPU-years
- We also searched for vertex-girth-regular graphs in known lists of graphs (e.g. vertex-transitive graphs) to obtain upper bounds for $n(k, g, \lambda)$
- We now discuss a few interesting observations based on these calculations

3 Influence of λ

- For fixed k, g how does $n(k, g, \lambda)$ behave in terms of λ?

3 Influence of λ

- For fixed k, g how does $n(k, g, \lambda)$ behave in terms of λ?
- All our lower bounds decrease for increasing λ

3 Influence of λ

- For fixed k, g how does $n(k, g, \lambda)$ behave in terms of λ?
- All our lower bounds decrease for increasing λ
- Moore graphs occur precisely when λ attains it maximal value

3 Influence of λ

- For fixed k, g how does $n(k, g, \lambda)$ behave in terms of λ?
- All our lower bounds decrease for increasing λ
- Moore graphs occur precisely when λ attains it maximal value
- Perhaps $n(k, g, \lambda) \geq n(k, g, \lambda + 1)$?

3 The inequality can fail: $n(3,8,8) = 42 < n(3,8,9) = 48$

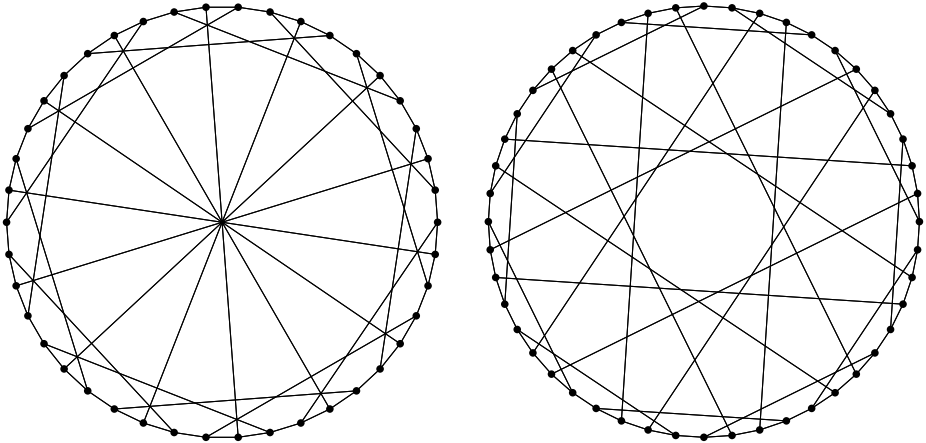


Figure: An extremal $\mathrm{vgr}(42,3,8,8)$-graph (left) and an extremal $\mathrm{vgr}(48,3,8,9)$-graph

3 Regularity versus symmetry

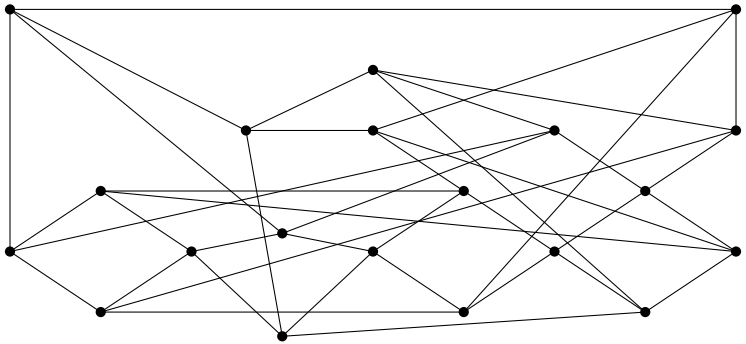


Figure: A $\mathrm{vgr}(20, 4, 4, 2)$-graph which is asymmetric (it only has the trivial automorphism).

3 Non-bipartite minimum order vgr graphs with even girth

Question, open for more than 40 years: **Is every even girth minimum order (k, g) graph bipartite?**

3 Non-bipartite minimum order vgr graphs with even girth

Question, open for more than 40 years: **Is every even girth minimum order (k, g) graph bipartite?**

The analogous question for vertex-girth-regular graphs has a negative answer:

3 Non-bipartite minimum order vgr graphs with even girth

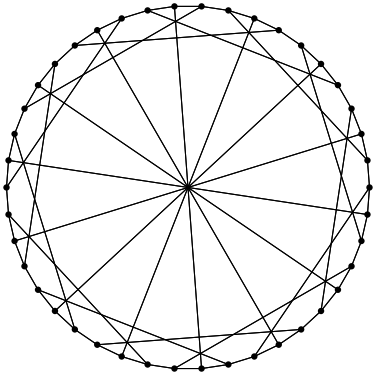


Figure: An extremal $\mathrm{vgr}(42, 3, 8, 8)$-graph

4 Outline

4 Conclusions

- In this paper, we initiated the study of vertex-girth-regular graphs

4 Conclusions

- In this paper, we initiated the study of vertex-girth-regular graphs
- We focused on existence, non-existence, lower bounds and enumeration

4 Conclusions

- In this paper, we initiated the study of vertex-girth-regular graphs
- We focused on existence, non-existence, lower bounds and enumeration
- **Future work**: analyze the obtained graphs and detect patterns leading to infinite families of graphs

4 Conclusions

- In this paper, we initiated the study of vertex-girth-regular graphs
- We focused on existence, non-existence, lower bounds and enumeration
- **Future work**: analyze the obtained graphs and detect patterns leading to infinite families of graphs
- **Future work**: improve upper bounds on $n(k, g, \lambda)$ based on existing constructions for cages

4 Conclusions

- In this paper, we initiated the study of vertex-girth-regular graphs
- We focused on existence, non-existence, lower bounds and enumeration
- **Future work**: analyze the obtained graphs and detect patterns leading to infinite families of graphs
- **Future work**: improve upper bounds on $n(k, g, \lambda)$ based on existing constructions for cages
- **Future work**: use similar algorithimic ideas for other variants of the cage problem

Thank you!

Full paper:

Contact: jorik.jooken@kuleuven.be

4 Implementation details

Two important parts: algorithm design and algorithm implementation

4 Implementation details

Two important parts: algorithm design and algorithm implementation
Typically, most important speed-ups come from algorithmic insights.
However, for a fixed algorithm the speed can differ an order of magnitude depending on how carefully it is implemented.

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):
Advantages:

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

Advantages:

- Very fast program execution

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

Advantages:

- Very fast program execution
- Leverage research in compiler design: compilers can optimize parts of your code automatically (e.g. "-O3")

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

Advantages:

- Very fast program execution
- Leverage research in compiler design: compilers can optimize parts of your code automatically (e.g. "-O3")
- Easy interaction with *nauty* package (e.g. computing canonical forms)

Disadvantages:

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

Advantages:

- Very fast program execution
- Leverage research in compiler design: compilers can optimize parts of your code automatically (e.g. "-O3")
- Easy interaction with *nauty* package (e.g. computing canonical forms)

Disadvantages:

- Not very user-friendly, many things written from scratch

4 Implementation details

This motivates us to use a programming language close to the hardware (e.g. C/C++):

Advantages:

- Very fast program execution
- Leverage research in compiler design: compilers can optimize parts of your code automatically (e.g. "-O3")
- Easy interaction with *nauty* package (e.g. computing canonical forms)

Disadvantages:

- Not very user-friendly, many things written from scratch
- It might cost your sanity :-)

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.
Example:

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.
Example:

- The set $\{0, 1, 3\}$ can be encoded by the integer $2^0 + 2^1 + 2^3 = 11$.

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.
Example:

- The set $\{0, 1, 3\}$ can be encoded by the integer $2^0 + 2^1 + 2^3 = 11$.
- Gain: instead of storing 3 integers (0, 1 and 3), we only store 1 integer (11).

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.
Example:

- The set $\{0, 1, 3\}$ can be encoded by the integer $2^0 + 2^1 + 2^3 = 11$.
- Gain: instead of storing 3 integers (0, 1 and 3), we only store 1 integer (11).
- Hardware supports many standard operations on bitsets (set intersection, set union, maximum, minimum, ...) using only $\frac{n}{64}$ elementary operations as opposed to n.

4 Bitwise programming

A data structure that almost always comes back in my research: the **bitset**
Idea: compactly represent small sets in a way that is nice for hardware to work with.
Example:

- The set $\{0, 1, 3\}$ can be encoded by the integer $2^0 + 2^1 + 2^3 = 11$.
- Gain: instead of storing 3 integers (0, 1 and 3), we only store 1 integer (11).
- Hardware supports many standard operations on bitsets (set intersection, set union, maximum, minimum, ...) using only $\frac{n}{64}$ elementary operations as opposed to n.
- Many "graph theory operations" (e.g. do two vertices have a common neighbor?) can be sped up by factor of 64.

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle
- Gunnar Brinkmann

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle
- Gunnar Brinkmann
- Jan Goedgebeur

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle
- Gunnar Brinkmann
- Jan Goedgebeur
- ... and many others

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle
- Gunnar Brinkmann
- Jan Goedgebeur
- ... and many others

Code to learn from:

- https://kulak.kuleuven.be/algo/software

4 Shameless (self-)advertising

For people interested in knowing more about computer-aided graph theory, some relevant people to look up:

- Brendan D. McKay
- Gordon F. Royle
- Gunnar Brinkmann
- Jan Goedgebeur
- ... and many others

Code to learn from:

- https://kulak.kuleuven.be/algo/software
- https://github.com/JorikJooken

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

- Each run was performed on a different core with up to 28 GB RAM per core (Total CPU time: around 6 CPU-years)

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

- Each run was performed on a different core with up to 28 GB RAM per core (Total CPU time: around 6 CPU-years)
- Hardware used: Flemish Supercomputer Centre

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

- Each run was performed on a different core with up to 28 GB RAM per core (Total CPU time: around 6 CPU-years)
- Hardware used: Flemish Supercomputer Centre
- Only accessible for researchers at Flemish university

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

- Each run was performed on a different core with up to 28 GB RAM per core (Total CPU time: around 6 CPU-years)
- Hardware used: Flemish Supercomputer Centre
- Only accessible for researchers at Flemish university
- Cost for KU Leuven researchers: around 10 euros per CPU-year (one of the reasons why people like to collaborate with our group)

4 Hardware infrastructure

We investigated $n(k, g, \lambda)$ for 159 tuples (k, g, λ) and many different orders v, leading to thousands of algorithm runs.

- Each run was performed on a different core with up to 28 GB RAM per core (Total CPU time: around 6 CPU-years)
- Hardware used: Flemish Supercomputer Centre
- Only accessible for researchers at Flemish university
- Cost for KU Leuven researchers: around 10 euros per CPU-year (one of the reasons why people like to collaborate with our group)
- Google, Amazon, ...: > 1000 euros per CPU-year (I think, never used it myself)