TL;DC: Too Long; Didn't Count

Rafael Kiesel[1], Markus Hecher[2]

[1]TU Wien, [2]Massachusetts Institute of Technology

7th of September 2023

Length Limited Path Counting

Given: A graph $G = (V, E)$, with vertices V and undirected edges $E \subseteq V \times V$, and a maximum length $\ell \geq 0$.

Compute: The number of simple paths in G whose length is less or equal to ℓ.

Optionally, ("one pair") only those paths between terminals t_1, t_2, otherwise ("all pairs") between any two vertices.

TL;DC

Backtracking Search

Frontier-Based Search

Frontier-based Search

- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]

Frontier-based Search

- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]
- Compute a “good” edge ordering $e_1, \ldots, e_m$.

Frontier-based Search

- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]
- Compute a "good" edge ordering $e_1, \ldots, e_m$.
- Decide (in order) whether the current edge is included in the path.

Frontier-based Search

- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]
- Compute a “good” edge ordering $e_1, \ldots, e_m$.
- Decide (in order) whether the current edge is included in the path.
- Prune search based on infeasibility.

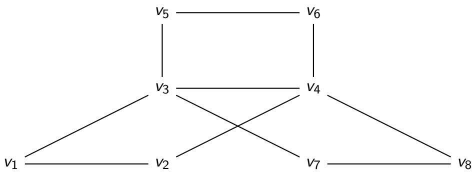
Frontier-based Search

- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]
- Compute a "good" edge ordering $e_1, \ldots, e_m$.
- Decide (in order) whether the current edge is included in the path.
- Prune search based on infeasibility.
- Keep only relevant state information.

Frontier-based Search

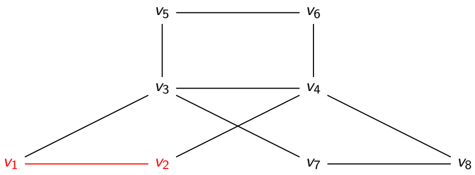
- Established approach for subgraph counting [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]
- Compute a "good" edge ordering $e_1, \ldots, e_m$.
- Decide (in order) whether the current edge is included in the path.
- Prune search based on infeasibility.
- Keep only relevant state information.
- Merge equal states to reduce the search space.

Edge Order



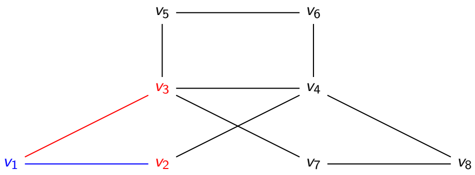
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



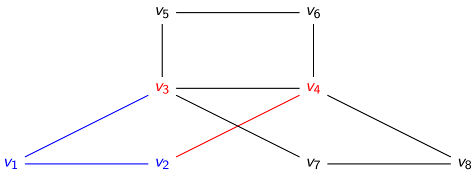
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

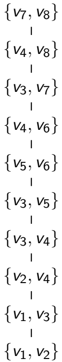
Edge Order



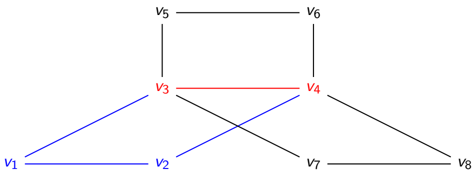
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order


v_5
v_6
v_3
v_4
v_1
v_2
v_7
v_8

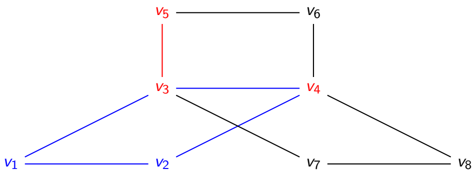

$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



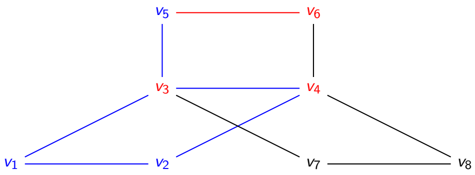
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



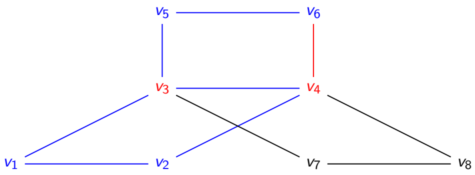
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



$\{v_7, v_8\}$

$\{v_4, v_8\}$

$\{v_3, v_7\}$

$\{v_4, v_6\}$

$\{v_5, v_6\}$

$\{v_3, v_5\}$

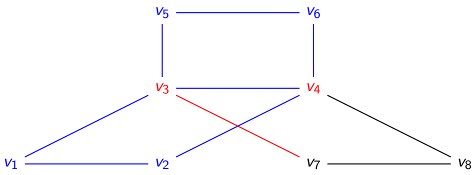
$\{v_3, v_4\}$

$\{v_2, v_4\}$

$\{v_1, v_3\}$

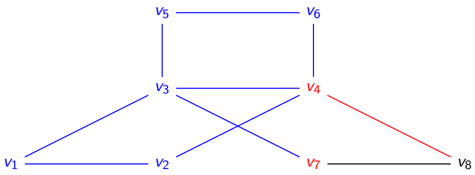
$\{v_1, v_2\}$

Edge Order



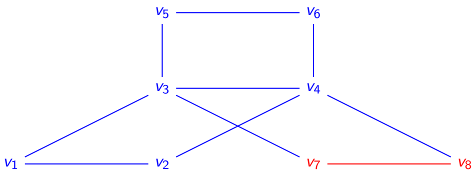
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



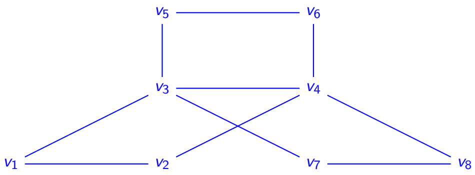
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



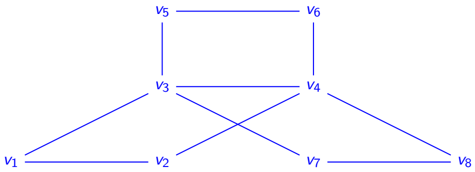
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

Edge Order



$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

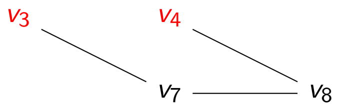
Edge Order



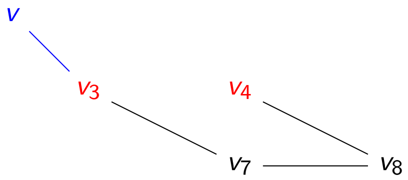
$\{v_7, v_8\}$
$\{v_4, v_8\}$
$\{v_3, v_7\}$
$\{v_4, v_6\}$
$\{v_5, v_6\}$
$\{v_3, v_5\}$
$\{v_3, v_4\}$
$\{v_2, v_4\}$
$\{v_1, v_3\}$
$\{v_1, v_2\}$

"Width" of 3, since there are at most three "active" vertices at the same time.

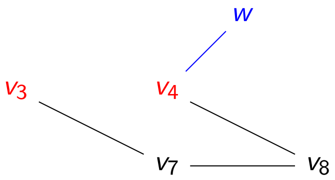
States



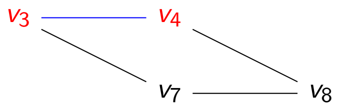
States



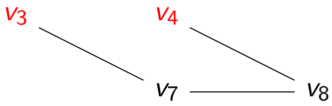
States



States



States

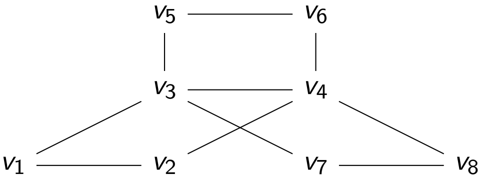


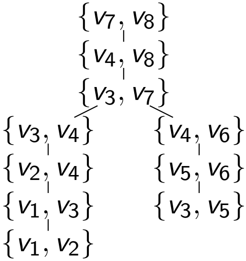
$\mathcal{O}(k^k \cdot \text{poly}(G))$ states for width k.

Using Disconnected Components

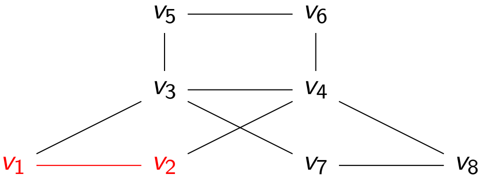
- Classical [Kawahara *et al.*, 2017],[Korf *et al.*, 2005]: "Width along a path."
- [Yasuda *et al.*, 2017]: "Width along a tree."

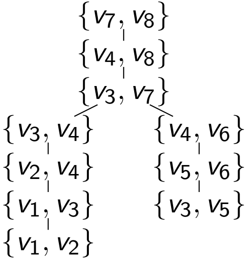
Edge Tree (vtree)



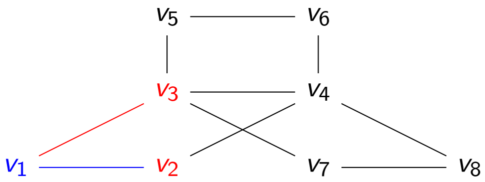


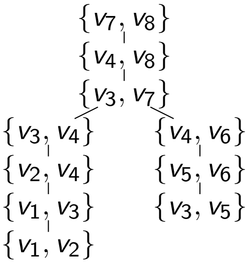
Edge Tree (vtree)



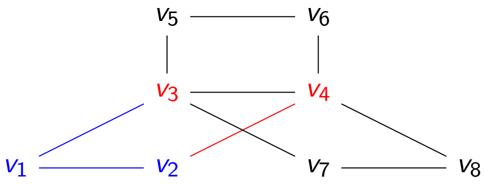


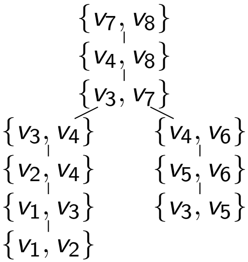
Edge Tree (vtree)



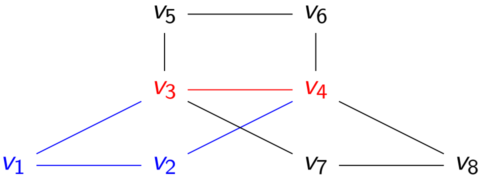


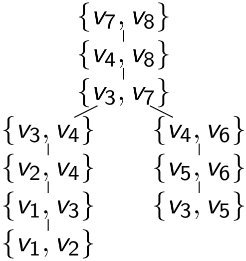
Edge Tree (vtree)



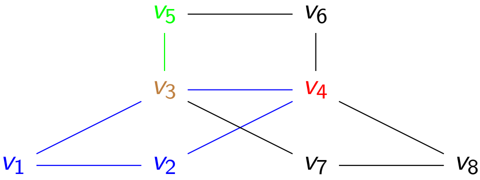


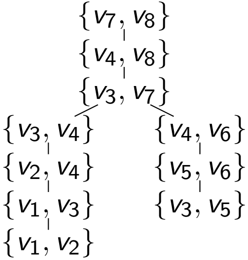
Edge Tree (vtree)



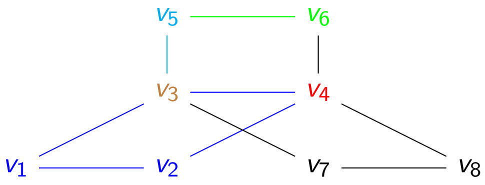


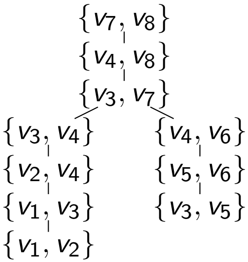
Edge Tree (vtree)



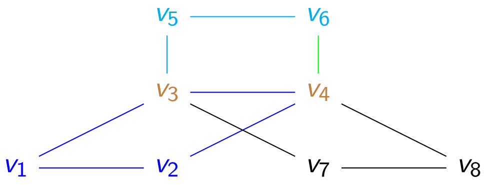


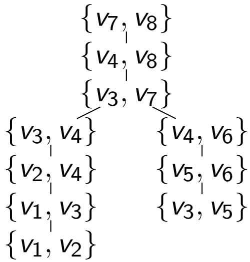
Edge Tree (vtree)



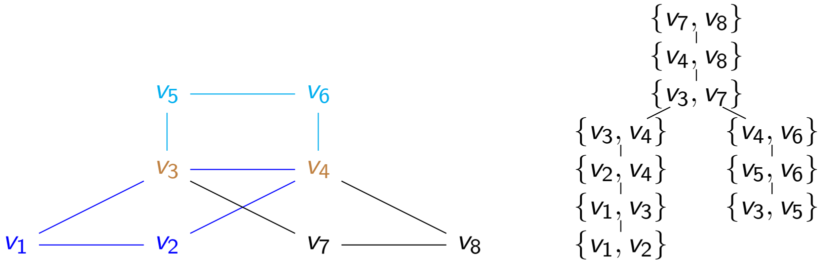


Edge Tree (vtree)

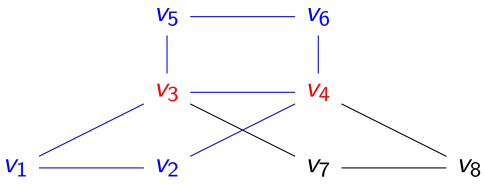


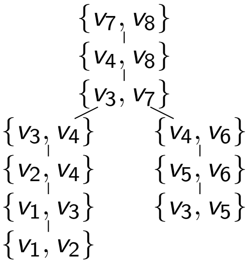


Edge Tree (vtree)

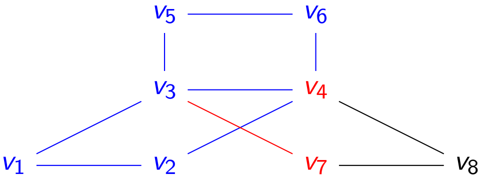


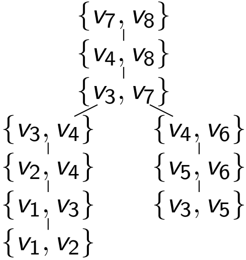
Edge Tree (vtree)



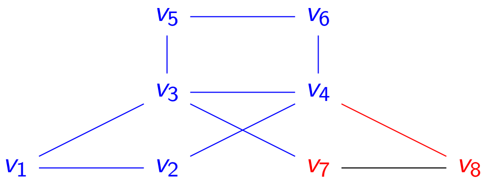


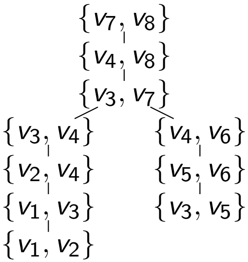
Edge Tree (vtree)



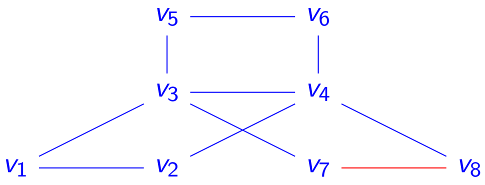


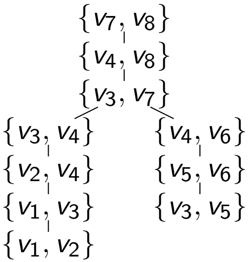
Edge Tree (vtree)



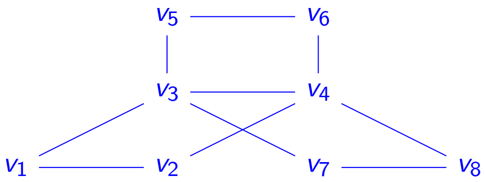


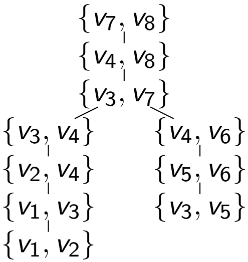
Edge Tree (vtree)



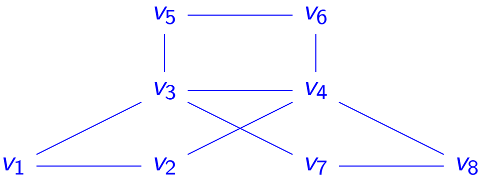


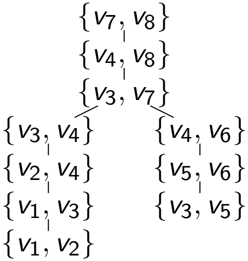
Edge Tree (vtree)





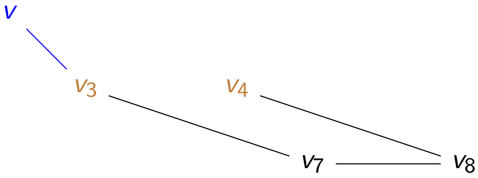
Edge Tree (vtree)



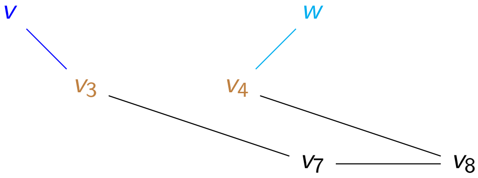


"Width" of 2, since there are at most two "active" vertices at the same time.

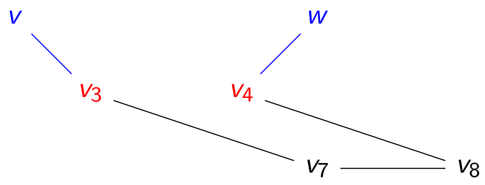
Merging States



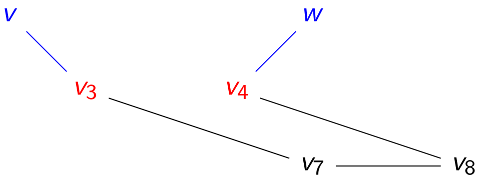
Merging States



Merging States



Merging States



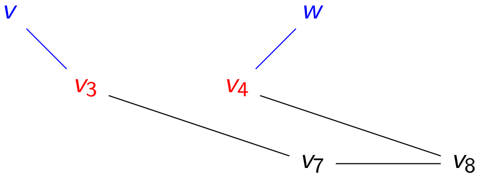
$\mathcal{O}(k^{2k} \cdot \mathrm{poly}(G))$ merges for width k.

TL;DC

Backtracking Search

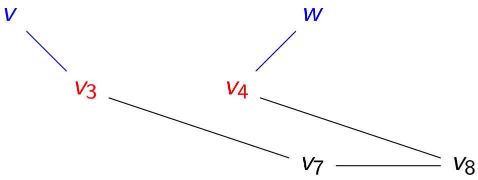
$$
\begin{array}{ccc}
 & \text{Frontier-based Search} & \\
\swarrow & & \searrow \\
\text{Pathwidth} & & \text{Treewidth}
\end{array}
$$

Length Limit



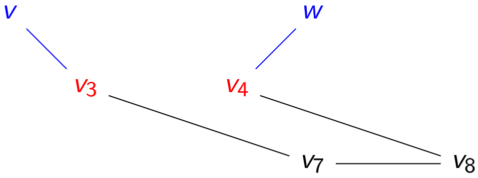
- Length limit $\ell = 5$, used edges $u = 3$.

Length Limit



- Length limit $\ell = 5$, used edges $u = 3$.
- We need all three remaining edges to complete the path $\lightning$.

Length Limit



- Length limit $\ell = 5$, used edges $u = 3$.
- We need all three remaining edges to complete the path $\lightning$.
- Prune based on length.

Length-based Pruning

- Given length limit ℓ, used edges u, and state m, check whether m can lead to a path in at most $\ell - u$ edges.

Length-based Pruning

- Given length limit ℓ, used edges u, and state m, check whether m can lead to a path in at most $\ell - u$ edges.
- Checking this is NP-hard in general $\lightning$.

Length-based Pruning

- Given length limit ℓ, used edges u, and state m, check whether m can lead to a path in at most $\ell - u$ edges.
- Checking this is NP-hard in general $\lightning$.
- Instead, compute a lower-bound on the edges.

Length-based Pruning

- Given length limit ℓ, used edges u, and state m, check whether m can lead to a path in at most $\ell - u$ edges.
- Checking this is NP-hard in general $\lightning$.
- Instead, compute a lower-bound on the edges.
- TL;DC: Based on sum of minimum distance between partial path endpoints.

Unit Propagation

1. Current state m and current edge e_i.

Unit Propagation

1. Current state m and current edge e_i.
2. Let $m_{with}/m_{without}$ be the states obtained by taking/skipping e_i.

Unit Propagation

1. Current state m and current edge e_i.
2. Let $m_{with}/m_{without}$ be the states obtained by taking/skipping e_i.
3. If we prune m_{with}, then set m to $m_{without}$ and goto 1.

Unit Propagation

1. Current state m and current edge e_i.
2. Let $m_{with}/m_{without}$ be the states obtained by taking/skipping e_i.
3. If we prune m_{with}, then set m to $m_{without}$ and goto 1.
4. If we prune $m_{without}$, then set m to m_{with} and goto 1.

Unit Propagation

1. Current state m and current edge e_i.
2. Let $m_{with}/m_{without}$ be the states obtained by taking/skipping e_i.
3. If we prune m_{with}, then set m to $m_{without}$ and goto 1.
4. If we prune $m_{without}$, then set m to m_{with} and goto 1.

- Enforces the invariant that cached entries can skip *and* take the next edge.

Unit Propagation

1. Current state m and current edge e_i.
2. Let $m_{with}/m_{without}$ be the states obtained by taking/skipping e_i.
3. If we prune m_{with}, then set m to $m_{without}$ and goto 1.
4. If we prune $m_{without}$, then set m to m_{with} and goto 1.

- Enforces the invariant that cached entries can skip *and* take the next edge.
- Reduces the number of cache accesses, thus, improves cache hit rate.

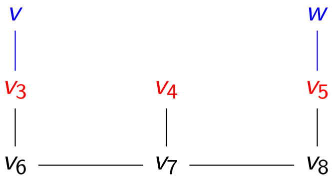
W.l.o.g. Caching

- Idea: Modify states to get more cache hits.

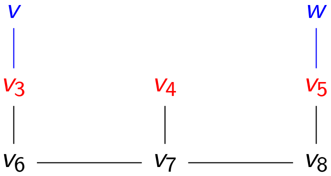
W.l.o.g. Caching

- Idea: Modify states to get more cache hits.
- "Without loss of generality all states..."

W.l.o.g. I

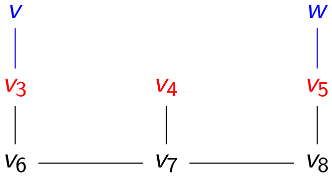


W.l.o.g. I



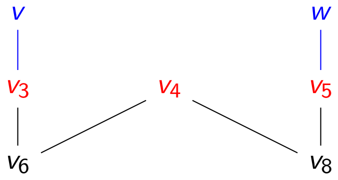
It does not matter whether v_4 appears in partial paths zero or two times. In either case, we cannot use it.

W.l.o.g. I



It does not matter whether v_4 appears in partial paths zero or two times. In either case, we cannot use it.
Assume w.l.o.g. that it appeared two times.

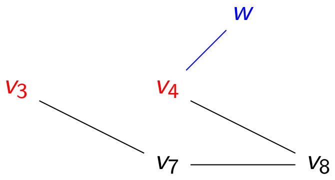
W.l.o.g. I

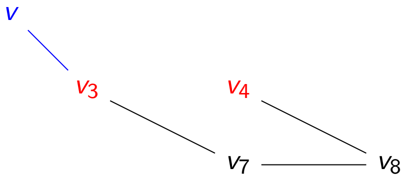


Only works if there is exactly one edge remaining, and there are two ends “outside”.

W.l.o.g. II

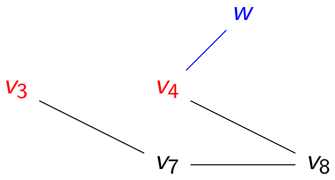
What is the difference between these two pictures?





W.l.o.g. II

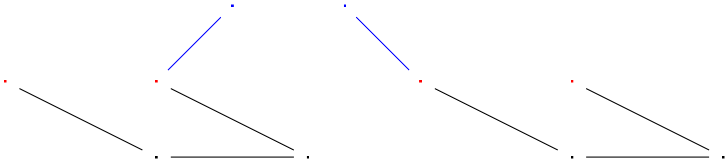
What is the difference between these two pictures?



They are the same! That is, they are equivalent modulo renaming a.k.a. *isomorph*.

W.l.o.g. II

What is the difference between these two pictures?



They are the same! That is, they are equivalent modulo renaming a.k.a. *isomorph*.

W.l.o.g. II (Canonical Representation)

- Checking for each cache entry whether the current remaining state + graph is isomorphic is costly.

W.l.o.g. II (Canonical Representation)

- Checking for each cache entry whether the current remaining state + graph is isomorphic is costly.
- Use Nauty [McKay and Piperno, 2014] to compute a *canonical representation*.

W.l.o.g. II (Canonical Representation)

- Checking for each cache entry whether the current remaining state + graph is isomorphic is costly.
- Use Nauty [McKay and Piperno, 2014] to compute a *canonical representation*.
- Two state + graph combinations are isomorphic iff they have the same canonical representation.

W.l.o.g. II (Canonical Representation)

- Checking for each cache entry whether the current remaining state + graph is isomorphic is costly.
- Use Nauty [McKay and Piperno, 2014] to compute a *canonical representation*.
- Two state + graph combinations are isomorphic iff they have the same canonical representation.
- Use the canonical representation as the cache key.

W.l.o.g. II (Canonical Representation)

- Checking for each cache entry whether the current remaining state + graph is isomorphic is costly.
- Use Nauty [McKay and Piperno, 2014] to compute a *canonical representation*.
- Two state + graph combinations are isomorphic iff they have the same canonical representation.
- Use the canonical representation as the cache key.
- Careful: This only works for pathwidth.

Other

- Do not compile everything first and prune the compiled circuit but prune as soon as possible.

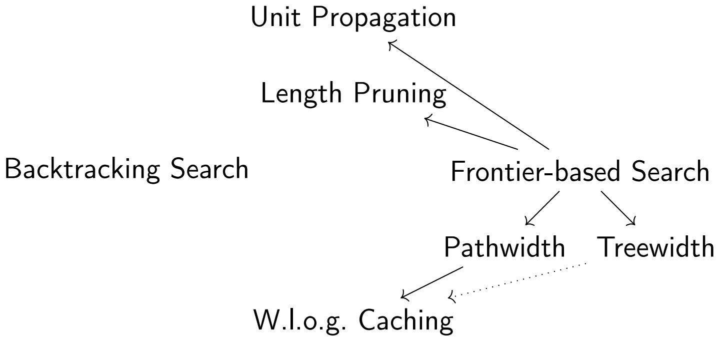
Other

- Do not compile everything first and prune the compiled circuit but prune as soon as possible.
- Do not compile first, count immediately.

Other

- Do not compile everything first and prune the compiled circuit but prune as soon as possible.
- Do not compile first, count immediately.
- These probably do not offer much of a runtime improvement but only some memory reduction. (Not tested.)

TL;DC


Unit Propagation
Length Pruning
Backtracking Search
Frontier-based Search
Pathwidth
Treewidth
W.l.o.g. Caching

Naïve Backtracking

- Perform Depth-first Search (DFS) on G starting at t_1.

Naïve Backtracking

- Perform Depth-first Search (DFS) on G starting at t_1.
- When the number of used edges exceeds ℓ bracktrack.

Naïve Backtracking

- Perform Depth-first Search (DFS) on G starting at t_1.
- When the number of used edges exceeds ℓ bracktrack.
- When t_2 is reached backtrack.

Naïve Backtracking

- Perform Depth-first Search (DFS) on G starting at t_1.
- When the number of used edges exceeds ℓ bracktrack.
- When t_2 is reached backtrack.
- Simple, but takes at least as many steps as there are paths $\lightning$.

Length Limit I

- At each step, compute distances d_1, d_2 to (current) t_1 and t_2 for each vertex v.

Length Limit I

- At each step, compute distances d_1, d_2 to (current) t_1 and t_2 for each vertex v.
- If $d_1 + d_2 + u > \ell$, where u is the number of used edges, mark v as visited.

Length Limit I

- At each step, compute distances d_1, d_2 to (current) t_1 and t_2 for each vertex v.
- If $d_1 + d_2 + u > \ell$, where u is the number of used edges, mark v as visited.
- Similar to w.l.o.g. caching: Can lead to collapse of cache entries.

Length Limit II

- Recall: Counting *shortest paths* from source to goal is polynomial time.

Length Limit II

- Recall: Counting *shortest paths* from source to goal is polynomial time.
- At each step, compute distances d_1, d_2 to (current) t_1 and t_2 for each vertex v.

Length Limit II

- Recall: Counting *shortest paths* from source to goal is polynomial time.
- At each step, compute distances d_1, d_2 to (current) t_1 and t_2 for each vertex v.
- If $d_1 + d_2 + u = \ell$, for a *neighbor of* t_1 compute the number of paths using v in polynomial time.
- Also, mark v as visited.

Unit Propagation

- Current start t_1 and remaining graph G.

Unit Propagation

- Current start t_1 and remaining graph G.
- If there is exactly one neighbor v of t_1 such that:
  - $v \neq t_2$,
  - $d_1 + d_2 + u < \ell$ (previous slide), and
  - v is not visited

  set u to $u + 1$, mark t_1 as visited, and set t_1 to v.

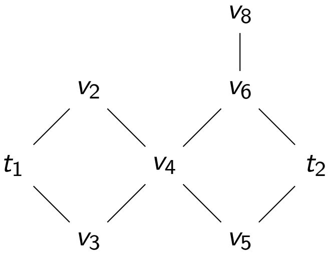
Unit Propagation

- Current start t_1 and remaining graph G.
- If there is exactly one neighbor v of t_1 such that:
  - $v \neq t_2$,
  - $d_1 + d_2 + u < \ell$ (previous slide), and
  - v is not visited

  set u to $u + 1$, mark t_1 as visited, and set t_1 to v.
- And of course count the paths for the remaining neighbors in polynomial time.

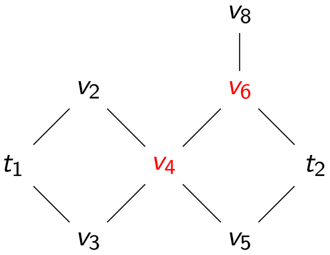
Articulation Points I

A vertex v is an *articulation point*, if its removal introduces new disconnected components.



Articulation Points I

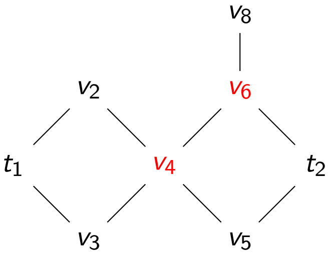
A vertex v is an *articulation point*, if its removal introduces new disconnected components.



v_4 and v_6 are articulation points.

Articulation Points I

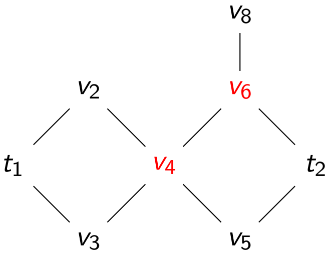
A vertex v is an *articulation point*, if its removal introduces new disconnected components.



v_4 and v_6 are articulation points.
Every path from t_1 to t_2 goes through v_4.

Articulation Points I

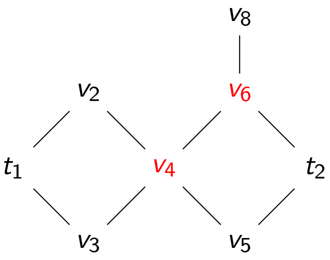
A vertex v is an *articulation point*, if its removal introduces new disconnected components.



v_4 and v_6 are articulation points.
Every path from t_1 to t_2 goes through v_4.
No path from t_1 to t_2 can go through v_8.

Articulation Points I

A vertex v is an *articulation point*, if its removal introduces new disconnected components.



v_4 and v_6 are articulation points.
Every path from t_1 to t_2 goes through v_4.
No path from t_1 to t_2 can go through v_8.
v_8 is *hidden* behind an articulation point.

Articulation Points II

- If v is hidden behind an articulation point, mark v as visited.

Articulation Points II

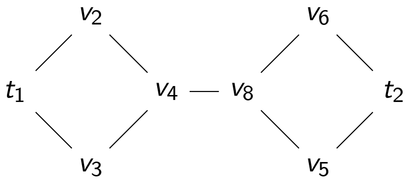
- If v is hidden behind an articulation point, mark v as visited.
- If v is an articulation point that occurs on every path from t_1 to t_2, we could compute the number of paths from t_1 to v and v to t_2 and combine the results.

Articulation Points II

- If v is hidden behind an articulation point, mark v as visited.
- If v is an articulation point that occurs on every path from t_1 to t_2, we could compute the number of paths from t_1 to v and v to t_2 and combine the results.
- Not done in TL;DC - would work well with compilation first approaches.

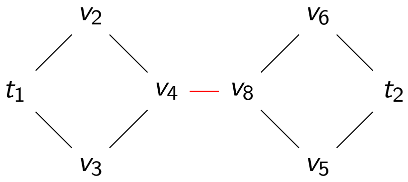
Bridges

An edge $\{v, w\}$ is a *bridge*, if its removal introduces new disconnected components.



Bridges

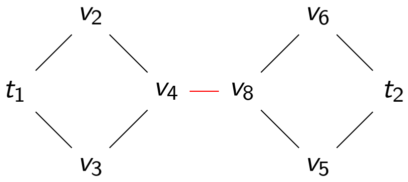
An edge $\{v, w\}$ is a *bridge*, if its removal introduces new disconnected components.



Every path from t_1 to t_2 has to use $\{v_4, v_8\}$.

Bridges

An edge $\{v, w\}$ is a *bridge*, if its removal introduces new disconnected components.

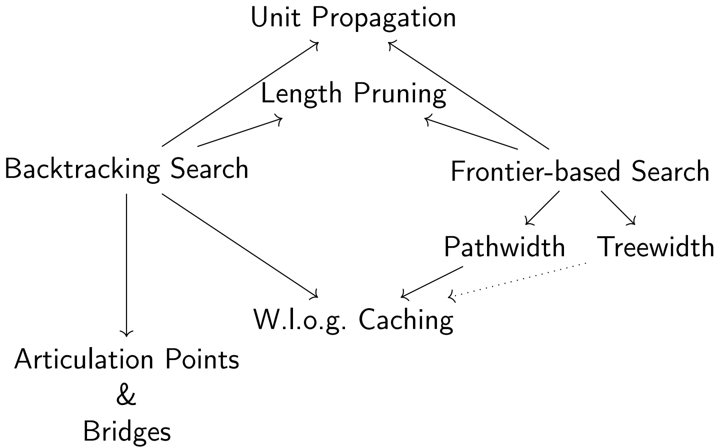


Every path from t_1 to t_2 has to use $\{v_4, v_8\}$.
Contract $\{v_4, v_8\}$ and set u to $u + 1$.

W.l.o.g. Caching

- As for FBS, use not the terminal + graph combination but its canonical representation.

TL;DC


Unit Propagation
Length Pruning
Backtracking Search
Frontier-based Search
Pathwidth
Treewidth
W.l.o.g. Caching
Articulation Points
&
Bridges

Cache:

Key	Value			
$t_1, t_2, G, u+1$	Length	5	6	7
	Count	10	20	40

- What if we need t_1, t_2, G, u?

Cache:

Key	Value			
$t_1, t_2, G, u+1$	Length	5	6	7
	Count	10	20	40

- What if we need t_1, t_2, G, u?
- Same terminals and graph, but smaller number u of used edges.

Cache:

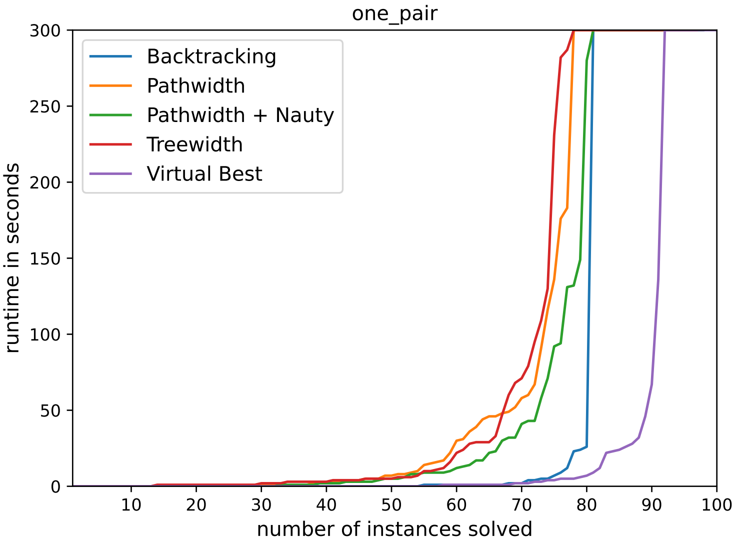
Key	Value			
$t_1, t_2, G, u+1$	Length	5	6	7
	Count	10	20	40

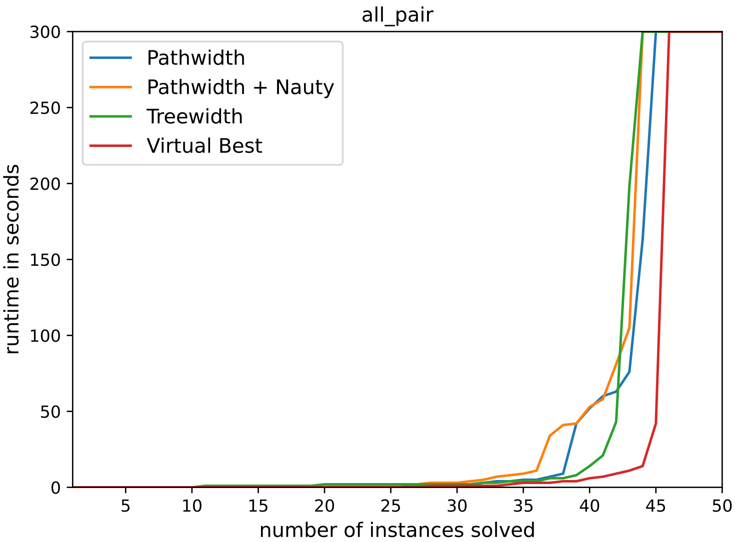
- What if we need t_1, t_2, G, u?
- Same terminals and graph, but smaller number u of used edges.
- If we know t_1, t_2, G, u, we also know $t_1, t_2, G, u+1$ but not the other way around!

Cache:

Key	Value			
$t_1, t_2, G, u+1$	Length	5	6	7
	Count	10	20	40

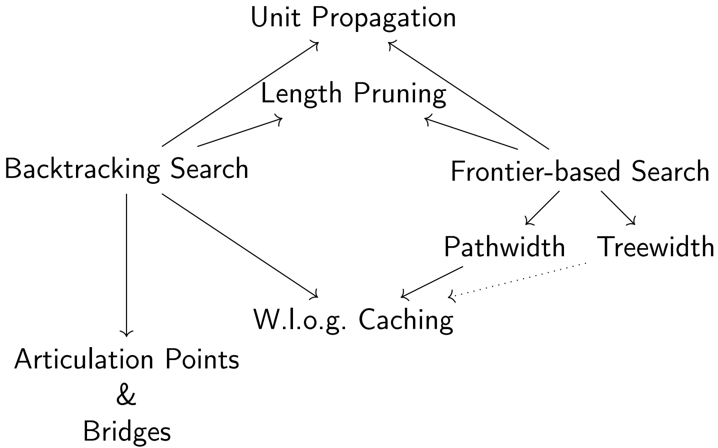
- What if we need t_1, t_2, G, u?
- Same terminals and graph, but smaller number u of used edges.
- If we know t_1, t_2, G, u, we also know $t_1, t_2, G, u+1$ but not the other way around!
- ↯ Top-down DP: compute and cache *results* for t_1, t_2, G, u
- ✓ Bottom-up DP: process all cache entries with graphs of size n, and cache *how many ways* there are to reach t_1, t_2, G. Here, $|G| < n$. Now process graph size $n-1$.


one_pair
Backtracking
Pathwidth
Pathwidth + Nauty
Treewidth
Virtual Best
runtime in seconds
0
50
100
150
200
250
300
10
20
30
40
50
60
70
80
90
100
number of instances solved


all_pair
Pathwidth
Pathwidth + Nauty
Treewidth
Virtual Best
runtime in seconds
number of instances solved
0
50
100
150
200
250
300
5
10
15
20
25
30
35
40
45
50

- Various new insights into limited length path counting.
- Especially, "w.l.o.g." caching helps.
- Different approaches work well in different settings:
  - Low Pathwidth
    $\hookrightarrow$ FBS + Pathwidth
  - Low Treewidth
    $\hookrightarrow$ FBS + Treewidth
  - Many Automorphisms + Medium Pathwidth
    $\hookrightarrow$ FBS + Pathwidth + Nauty
  - Low Length Limit
    $\hookrightarrow$ BT (+ Nauty)

TL;DC: https://github.com/raki123/TL-DC



Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato.
Frontier-based search for enumerating all constrained subgraphs with compressed representation.
IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 100(9):1773–1784, 2017.

Richard E Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald.
Frontier search.
Journal of the ACM (JACM), 52(5):715–748, 2005.

Brendan D McKay and Adolfo Piperno.
Practical graph isomorphism, ii.
Journal of symbolic computation, 60:94–112, 2014.

Norihito Yasuda, Teruji Sugaya, and Shin-ichi Minato.
Fast compilation of s-t paths on a graph for counting and enumeration.
In Antti Hyttinen, Joe Suzuki, and Brandon M. Malone, editors, *Proceedings of the 3rd Workshop on Advanced Methodologies for Bayesian Networks, AMBN 2017, Kyoto, Japan, September 20-22, 2017*, volume 73 of *Proceedings of Machine Learning Research*, pages 129–140. PMLR, 2017.