Efficient Parallel Subgraph Enumeration on a Single Machine

Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo*

The Hong Kong University of Science and Technology
Outline

- Background
  - Basic Subgraph Enumeration Algorithm
  - Lazy Materialization Subgraph Enumeration
- Evaluation
- Conclusions
Subgraph Isomorphism

Given unlabeled graphs $g = (V, E)$ and $g' = (V', E')$, a subgraph isomorphism from $g$ to $g'$ is an injective function $\varphi: V \rightarrow V'$ such that $\forall e(u, u') \in E, e(\varphi(u), \varphi(u')) \in E'$. 
Problem Definition

Given a data graph $G$ and a pattern graph $P$, subgraph enumeration finds all subgraphs in $G$ that are isomorphic to $P$. 
Existing Algorithms on a Single Machine

- DUALSIM partitions data graphs that cannot fit in memory.
- EmptyHeaded utilizes the worst-case optimal join to enumerate subgraphs.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Environment</th>
<th>Year Published</th>
</tr>
</thead>
</table>
Existing Distributed Algorithms

Distributed algorithms adopt the parallel join method.
1. Decompose $P$ into a collection of small components.
2. Join the matches of the components in parallel.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Distributed Environment</th>
<th>Year Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afrati [1]</td>
<td>MapReduce</td>
<td>ICDE 2013</td>
</tr>
<tr>
<td>BiGJoin [6]</td>
<td>Timely Dataflow</td>
<td>VLDB 2018</td>
</tr>
</tbody>
</table>
Outline

- Background
- Basic Subgraph Enumeration Algorithm
- Lazy Materialization Subgraph Enumeration
- Evaluation
- Conclusions
Basic Subgraph Enumeration Algorithm

**Input:** a data graph $G$ and a pattern graph $P$.

**Output:** all subgraphs in $G$ that are isomorphic to $P$.

1. Generate an enumeration order $\pi$, which is a permutation of pattern vertices.
2. Enumerate all solutions by recursively extending partial results along $\pi$. 
Example of SE

Pattern Graph $P$.

Data Graph $G$.

Search Tree of SE.

Enumeration Order

Partial Result
Example of SE

Step 1.

Pattern Graph $P$.

Data Graph $G$.

Search Tree of SE.

Computation

$C_{\psi}(u_i) = N(v_0) \cap N(v_{101}) = \{v_{1-100}\}$

Materialization

Expand a Partial Result.

Step 2.
We find that there is a large amount of redundant computation in the enumeration.
Observation One

Pattern Graph $P$.  

Data Graph $G$.  

Search Tree of SE.  

The same set intersection $N(v_0) \cap N(v_{101})$ is repeated in the computation of partial results in the dashed rectangle for $u_3$.  

Observation Two

Pattern Graph $P$.

Data Graph $G$.

Search Path of SE.

Given partial results $\varphi_1$ and $\varphi_2$, the same set intersection $N(v_0) \cap N(v_{101})$ is repeated in the computation of candidates of $u_1$ and $u_3$. 
Outline

- Background
- Basic Subgraph Enumeration Algorithm
- Lazy Materialization Subgraph Enumeration
- Evaluation
- Conclusions
Lazy Materialization

We propose the lazy materialization subgraph enumeration algorithm, called **LIGHT**.

- Separate the computation and the materialization.
- Keep the order of the computation unchanged.
- Delay the materialization until some computation requires it.
Example of Lazy Materialization

Pattern Graph $P$.  

Data Graph $G$.  

Enumeration Order $\pi$.  

$u_0$  

$u_2$  

$u_1$  

$u_3$
Example of Lazy Materialization

Pattern Graph $P$. 

Data Graph $G$. 

Enumeration Order $\pi$. 

Operation Order of SE.

$u_0$, $u_1$, $u_2$, $u_3$, $v_0$, $v_1$, $v_2$, $v_9$, $v_{10}$, $v_{100}$, $v_{101}$, $v_{99}$, $v_{100}$.
Example of Lazy Materialization

Pattern Graph $P$.  

Data Graph $G$.  

Enumeration Order $\pi$.  

Operation Order of SE.  

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.  

Data Graph $G$.  

Enumeration Order $\pi$.  

Operation Order of SE.  

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.

Data Graph $G$.

Enumeration Order $\pi$.

Operation Order of SE.

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.  

Data Graph $G$.  

Enumeration Order $\pi$.  

Operation Order of SE.  

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.

Data Graph $G$.

Enumeration Order $\pi$.

Operation Order of SE.

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.

Data Graph $G$.

Enumeration Order $\pi$.

Operation Order of SE.

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph \( P \).

Data Graph \( G \).

Enumeration Order \( \pi \).

Operation Order of SE.

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.  

Data Graph $G$.  

Enumeration Order $\pi$.  

Operation Order of SE.  

Operation Order of LIGHT.
Example of Lazy Materialization

Pattern Graph $P$.

Data Graph $G$.

Search Tree of SE.

Search Tree of LIGHT.
MSC based Candidate Sets Computation

Compute the candidate set of \( u \in \pi \) by utilizing candidate sets of \( u' \in M(u) \) in \( \pi \).

- Convert it to the minimum set cover (MSC) problem:

**Input:** \( U = N_+^\pi (u), S = \{u' \mid u' \in U\} \cup \{N_+^\pi (u') \mid N_+^\pi (u') \subseteq N_+^\pi (u) \land u' \in M(u)\} \).

**Output:** The smallest sub-collection \( S' \) of \( S \) whose union equals \( U \).

**Notation:**

1. The backward neighbors \( N_+^\pi (u) \) of \( u \) contains the neighbors of \( u \) positioned before \( u \) in \( \pi \).
2. \( M(u) \) contains all pattern vertices before \( u \) in \( \pi \).
Example of MSC

Pattern Graph $P$.

Data Graph $G$.

MSC Input:

$U = \{u_0, u_2\}$

$S = \{\{u_0\}, \{u_2\}, \{u_0, u_2\}\}$

MSC Output:

$S' = \{\{u_0, u_2\}\}$

$C_\varphi(u_3) = C_\varphi(u_1)$

Compute Candidate Set of $u_3$.

$N^\pi_+(u_3) = \{u_0, u_2\}$

$M(u_3) = \{u_0, u_1, u_2\}$

Enumeration Order $\pi$. 
Example of MSC

Pattern Graph $P$.

Data Graph $G$.

Search Path of SE.

Search Path of LIGHT.
Parallel Implementation

Utilize both vector registers and multiple cores in modern CPUs.

- Parallelize set intersections with SIMD (Single-Instruction-Multiple-Data) instructions.
- Parallelize the exploration of the search tree with multi-threading.
Outline

- Background
- Basic Subgraph Enumeration Algorithm
- Lazy Materialization Subgraph Enumeration
- Evaluation
- Conclusions
Datasets

Real-world Datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Name</th>
<th>$N$ (million)</th>
<th>$M$ (million)</th>
<th>Memory (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>youtube</td>
<td>yt</td>
<td>3.22</td>
<td>9.38</td>
<td>0.09</td>
</tr>
<tr>
<td>eu-2005</td>
<td>eu</td>
<td>0.86</td>
<td>19.24</td>
<td>0.15</td>
</tr>
<tr>
<td>live-journal</td>
<td>lj</td>
<td>4.85</td>
<td>68.48</td>
<td>0.53</td>
</tr>
<tr>
<td>com-orkut</td>
<td>ot</td>
<td>3.07</td>
<td>117.19</td>
<td>0.89</td>
</tr>
<tr>
<td>uk-2002</td>
<td>uk</td>
<td>18.52</td>
<td>298.11</td>
<td>2.30</td>
</tr>
<tr>
<td>friendster</td>
<td>fs</td>
<td>65.61</td>
<td>1,806.07</td>
<td>13.71</td>
</tr>
</tbody>
</table>

Pattern Graphs.

(a) $P_1$. (b) $P_2$. (c) $P_3$. (d) $P_4$. (e) $P_5$. (f) $P_6$. (g) $P_7$. 
Experimental Environment.

- Implemented in C++ and compiled with icpc 16.0.0.
- A machine equipped with 20 cores (2 Intel Xeon E5-2650 v3 @ 2.30GHz CPUs), 64GB RAM and 1TB HDD.
- Use the AVX2 (256-bit) instruction set and execute with 64 threads.
Comparison with SE

- $T_{SE}$ and $T_{LIGHT}$ are the serial execution time of SE and LIGHT respectively.
- $T_{SE+P}$ and $T_{LIGHT+P}$ are the parallel execution time of SE and LIGHT respectively.
- Overall Speedup $= \frac{T_{SE}}{T_{LIGHT+P}}$.

<table>
<thead>
<tr>
<th>Dataset Pattern</th>
<th>$yt$</th>
<th>$lj$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset Pattern</td>
<td>$P_2$</td>
<td>$P_4$</td>
</tr>
<tr>
<td>$T_{SE}$</td>
<td>645</td>
<td>176,181</td>
</tr>
<tr>
<td>$T_{SE+P}$</td>
<td>22</td>
<td>4,034</td>
</tr>
<tr>
<td>$T_{LIGHT}$</td>
<td>31</td>
<td>3,309</td>
</tr>
<tr>
<td>$T_{LIGHT+P}$</td>
<td>0.3</td>
<td>56</td>
</tr>
<tr>
<td>Speedup</td>
<td>2,150X</td>
<td>3,146X</td>
</tr>
</tbody>
</table>
Conclusions

We propose an efficient parallel subgraph enumeration algorithm LIGHT for a single machine.

- Reduce the redundant computation by the lazy materialization and the MSC based candidate sets computation.
- Parallelize LIGHT with both SIMD and multi-threading to fully utilize the parallel computation capabilities in modern CPUs.
Selected References

Thanks.

Q&A
Automorphism

An automorphism of $P$ is a match from $P$ to itself. Because of the automorphisms, a subgraph in $G$ isomorphic to $P$ can result in duplicate matches from $P$ to $G$. 
Automorphism

An automorphism of $P$ is a match from $P$ to itself. Because of the automorphisms, a subgraph in $G$ isomorphic to $P$ can result in duplicate matches from $P$ to $G$. 

\begin{itemize}
  \item Pattern Graph $P$. 
  \item Data Graph $G$. 
\end{itemize}
Automorphism

An automorphism of $P$ is a match from $P$ to itself. Because of the automorphisms, a subgraph in $G$ isomorphic to $P$ can result in duplicate matches from $P$ to $G$.

There is only 1 subgraph in $G$ isomorphic to $P$, while we can find 6 matches from $P$ to $G$. 
Symmetry Breaking

In order to eliminate the duplicate matches, symmetry breaking assigns order $<$ to pattern vertices, and requires the matches $\phi$ to satisfy that given $u, u' \in V(P)$, if $u < u'$, then $\phi(u) < \phi(u')$.

The orders of $P$ is $u_0 < u_1 < u_2$. There is only one match from $P$ to $G$ that satisfies the constraint of the symmetry breaking, which is $\{(u_0, v_0), (u_1, v_1), (u_2, v_2)\}$.
Problem Definition

Given a data graph $G$ and a pattern graph $P$, subgraph enumeration finds subgraphs in $G$ that are isomorphic to $P$.

For the ease of analysis, we assume that there is only one automorphism. Then, the problem is equivalent to finding all matches from $P$ to $G$. 
Basic Subgraph Enumeration Algorithm

Algorithm 1: SE Algorithm

Input: a pattern graph $P$ and a data graph $G$
Output: all matches from $P$ to $G$

1 begin
2 \[ \pi \leftarrow \text{compute a connected enumeration order of } V(P); \]
3 \[ i \leftarrow 1, \varphi \leftarrow \{\}; \]
4 foreach $v \in V(G)$ do
5 \[ \text{Add } (\pi[i], v) \text{ to } \varphi; \]
6 \[ \text{Enumerate } (\pi, \varphi, i + 1); \]
7 \[ \text{Remove } (\pi[i], v) \text{ from } \varphi; \]
8 Procedure Enumerate $(\pi, \varphi, i)$
9 \[ \text{if } i = |\pi| + 1 \text{ then Output } \varphi, \text{ return;} \]
10 \[ /* \text{The computation phase.} */ \]
11 \[ C_{\varphi}(\pi[i]) \leftarrow \text{ComputeCandidates}(\pi[i], \varphi); \]
12 \[ /* \text{The materialization phase.} */ \]
13 foreach $v \in C_{\varphi}(\pi[i])$ do
14 \[ \text{if } v \notin \varphi.values \text{ then Same as Lines 5-7;} \]
15 Function ComputeCandidates $(u, \varphi)$
16 \[ C_{\varphi}(u) \leftarrow \bigcap_{u' \in N_{\overline{P}}(u)} N(\varphi(u')); \]
17 return $C_{\varphi}(u)$;
Basic Subgraph Enumeration Algorithm

Algorithm 1: SE Algorithm

Input: a pattern graph \( P \) and a data graph \( G \)
Output: all matches from \( P \) to \( G \)

begin
\begin{enumerate}
\item \( \pi \leftarrow \text{compute a connected enumeration order of} \ V(P) \); \\
\item \( i \leftarrow 1, \varphi \leftarrow \{\} \); \\
\item \( \text{foreach} \ v \in V(G) \text{ do} \)
\begin{enumerate}
\item Add \( (\pi[i], v) \text{ to} \varphi \); \\
\item \( \text{Enumerate}(\pi, \varphi, i + 1) \); \\
\item \( \text{Remove} \ (\pi[i], v) \text{ from} \varphi \); \\
\end{enumerate}
\end{enumerate}
Procedure \( \text{Enumerate}(\pi, \varphi, i) \)
\begin{enumerate}
\item if \( i = |\pi| + 1 \) then Output \( \varphi \), return; \\
\begin{enumerate}
\item \( C_\varphi(\pi[i]) \leftarrow \text{ComputeCandidates}(\pi[i], \varphi) \); \\
\end{enumerate}
\end{enumerate}
\begin{enumerate}
\item foreach \( v \in C_\varphi(\pi[i]) \) do \\
\item if \( v \not\in \varphi.\text{values} \) then Same as Lines 5-7; \\
\end{enumerate}
Function \( \text{ComputeCandidates}(u, \varphi) \)
\begin{enumerate}
\item \( C_\varphi(u) \leftarrow \bigcap_{u' \in N_\varphi(u)}^{} N(\varphi(u')) \); \\
\end{enumerate}
return \( C_\varphi(u) \);

Enumeration order \( \pi \) is a permutation of \( V(P) \). \( \pi[i] \) is the \( i \)th vertex in \( \pi \).
Basic Subgraph Enumeration Algorithm

Algorithm 1: SE Algorithm

Input: a pattern graph $P$ and a data graph $G$
Output: all matches from $P$ to $G$

begin
\[ \pi \leftarrow \text{compute a connected enumeration order of } V(P); \]
\[ i \leftarrow 1, \varphi \leftarrow \{\}; \]
foreach $v \in V(G)$ do
  Add $(\pi[i], v)$ to $\varphi$;
  Enumerate $(\pi, \varphi, i + 1)$;
  Remove $(\pi[i], v)$ from $\varphi$;
Procedure Enumerate $(\pi, \varphi, i)$
if $i = |\pi| + 1$ then Output $\varphi$, return;
/* The computation phase. */
$C_\varphi(\pi[i]) \leftarrow \text{ComputeCandidates}(\pi[i], \varphi)$;
/* The materialization phase. */
foreach $v \in C_\varphi(\pi[i])$ do
  if $v \notin \varphi.values$ then Same as Lines 5-7;
Function ComputeCandidates($u$, $\varphi$)
$C_\varphi(u) \leftarrow \bigcap_{u' \in N_\pi(u)} N(\varphi(u'))$;
return $C_\varphi(u)$;

Enumeration order $\pi$ is a permutation of $V(P). \pi[i]$ is the $i$th vertex in $\pi$.
Recursively expand the partial result $\varphi$ by mapping pattern vertices to data vertices along $\pi$ to find all matches from $P$ to $G$. 
Basic Subgraph Enumeration Algorithm

### Algorithm 1: SE Algorithm

**Input:** a pattern graph $P$ and a data graph $G$  
**Output:** all matches from $P$ to $G$

**begin**

1. $\pi \leftarrow$ compute a connected enumeration order of $V(P)$;  
2. $i \leftarrow 1, \varphi \leftarrow \{\}$;
3. **foreach** $v \in V(G)$ **do**
   4. Add $(\pi[i], v)$ to $\varphi$;
   5. Enumerate $(\pi, \varphi, i + 1)$;
   6. Remove $(\pi[i], v)$ from $\varphi$;
4. **if** $i = |\pi| + 1$ **then** Output $\varphi$, **return**;

**Procedure** Enumerate $(\pi, \varphi, i)$

  /* The computation phase. */
  $C_\varphi(\pi[i]) \leftarrow$ ComputeCandidates($\pi[i], \varphi$);

  /* The materialization phase. */
  **foreach** $v \in C_\varphi(\pi[i])$ **do**
  1. **if** $v \notin \varphi$values **then** Same as Lines 5-7;

**Function** ComputeCandidates($u, \varphi$)

1. $C_\varphi(u) \leftarrow \bigcap_{u' \in N^I_T(u)} N(\varphi(u'))$;
2. return $C_\varphi(u)$;

**Enumeration order** $\pi$ is a permutation of $V(P)$. $\pi[i]$ is the $i$th vertex in $\pi$.

Recursively expand the partial result $\varphi$ by mapping pattern vertices to data vertices along $\pi$ to find all matches from $P$ to $G$.

The computation phase is to obtain the candidate set $C_\varphi(\pi[i])$ of $\pi[i]$ given $\varphi$, and the materialization phase extends $\varphi$ by mapping $\pi[i]$ to $v \in C_\varphi(\pi[i])$. 

46
Basic Subgraph Enumeration Algorithm

**Algorithm 1: SE Algorithm**

```
Input: a pattern graph $P$ and a data graph $G$
Output: all matches from $P$ to $G$

begin
  $\pi \leftarrow$ compute a connected enumeration order of $V(P)$;
  $i \leftarrow 1$, $\varphi \leftarrow \{\}$;
  foreach $v \in V(G)$ do
    Add ($\pi[i], v$) to $\varphi$;
    Enumerate($\pi, \varphi, i + 1$);
    Remove ($\pi[i], v$) from $\varphi$;

Procedure Enumerate($\pi, \varphi, i$)
if $i = |\pi| + 1$ then Output $\varphi$, return;
/* The computation phase. */
$C_{\varphi}(\pi[i]) \leftarrow$ ComputeCandidates($\pi[i], \varphi$);
/* The materialization phase. */
foreach $v \in C_{\varphi}(\pi[i])$ do
  if $v \notin \varphi$ values then Same as Lines 5-7;

Function ComputeCandidates($u, \varphi$)
$C_{\varphi}(u) \leftarrow \bigcap_{u' \in N_{\pi}(u)} N(\varphi(u'))$;
return $C_{\varphi}(u)$;
```

**Explanation:**
- **Enumeration order** $\pi$ is a permutation of $V(P)$. $\pi[i]$ is the $i$th vertex in $\pi$.
- Recursively expand the partial result $\varphi$ by mapping pattern vertices to data vertices along $\pi$ to find all matches from $P$ to $G$.
- The computation phase is to obtain the candidate set $C_{\varphi}(\pi[i])$ of $\pi[i]$ given $\varphi$, and the materialization phase extends $\varphi$ by mapping $\pi[i]$ to $v \in C_{\varphi}(\pi[i])$.
- Compute common neighbors of data vertices mapped to backward neighbors of $u$ where backward neighbors $N_{\pi}^{-}(u)$ of $u$ is the neighbors of $u$ positioned before $u$ in $\pi$. 
Parallelize Set Intersection

- Given two sets $S_1$ and $S_2$, which are stored as sorted arrays, we use SIMD to parallelize the set intersection between $S_1$ and $S_2$.

- We use a hybrid set intersection method to handle the size skewness of input sets:
  1. If the size of $S_1$ and $S_2$ is similar, use the merge-based set intersection.
  2. Otherwise, use the Galloping [1] algorithm.
Parallelize Search Tree Exploration

We take the partial results as parallel tasks, and each worker expands the assigned partial results in DFS independently.
Parallelize Search Tree Exploration

We adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
Parallelize Search Tree Exploration

We adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
Parallelize Search Tree Exploration

We adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
Parallelize Search Tree Exploration

We adopt a sender-initiated method with a global concurrent queue to deliver tasks among workers.
Optimize Enumeration Order

Utilize the ordering method proposed in SEED.

Experimental Setup

Algorithms Under Study.

- EH [8]: EmptyHeaded, a relational engine for graph processing that answers queries with WCOJ algorithms.
- CFL [9]: the state-of-the-art labeled subgraph enumeration algorithm.
- SE: Algorithm 1, which is the baseline algorithm.
- LM: LIGHT with the Lazy Materialization strategy only.
- MSC: LIGHT with the Minimum Set Cover based candidate set computation method only.
- LIGHT: LIGHT with both the lazy materialization and the minimum set cover based candidate set computation.
Enumeration Order

SE, LM, MSC and LIGHT adopt the same enumeration order.

- $\pi(P_2) = (u_0, u_2, u_1, u_3)$, $\pi(P_4) = (u_0, u_1, u_4, u_2, u_3)$, and $\pi(P_6) = (u_0, u_1, u_2, u_3, u_4)$.

The enumeration order of CFL is as follows.

- $\pi(P_2) = (u_0, u_2, u_1, u_3)$, $\pi(P_4) = (u_0, u_2, u_4, u_1, u_3)$, and $\pi(P_6) = (u_0, u_1, u_2, u_3, u_4)$.

The enumeration order of EH is as follows.

- $\pi(P_2) = (u_1, u_3, u_0, u_2)$
- $\pi(P_4') = (u_0, u_3, u_4, u_1)$, and $\pi(P_4'') = (u_0, u_3, u_2)$. Join the matches of $P_4'$ and $P_4''$.
- $\pi(P_6') = (u_0, u_1, u_2, u_3)$, and $\pi(P_6'') = (u_0, u_1, u_4)$. Join the matches of $P_6'$ and $P_6''$. 
Reducing Redundant Computation

- EH runs slower than other algorithms on $P_2$, and runs out of memory on $P_4$ and $P_6$. 

Comparison of Execution Time.

Comparison of Number of Set Intersections.
Reducing Redundant Computation

- EH runs slower than other algorithms on \(P_2\), and runs out of memory on \(P_4\) and \(P_6\).
- CFL cannot complete \(P_4\) within the time limit, and performs the same number of set intersections with SE.
Reducing Redundant Computation

- EH runs slower than other algorithms on $P_2$, and runs out of memory on $P_4$ and $P_6$.
- CFL cannot complete $P_4$ within the time limit, and performs the same number of set intersections with SE.
- LIGHT significantly reduces the number of set intersections compared with SE, and outperforms the other algorithms.

**Comparison of Execution Time.**

**Comparison of Number of Set Intersections.**
Parallelization

- HybridAVX2 runs 1.2-6.5X times faster than Merge.

![Execution Time with Different Set Intersection Methods.](image)

- Execution Time with the Number of Threads Varied.

![Execution Time with the Number of Threads Varied.](image)
Parallelization

- HybridAVX2 runs 1.2-6.5X times faster than Merge.
- LIGHT achieves almost linear speedup, when #threads varies from 1 to 16.
Comparison with Existing Algorithms

Execution Time of LIGHT, DUALSIM, SEED and CRYSTAL on the Real-world Datasets.
Backup

<table>
<thead>
<tr>
<th>Dataset</th>
<th>yt</th>
<th>eu</th>
<th>lj</th>
<th>ot</th>
<th>uk</th>
<th>fs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory (GB)</td>
<td>0.123</td>
<td>0.090</td>
<td>0.022</td>
<td>0.048</td>
<td>0.239</td>
<td>0.008</td>
</tr>
</tbody>
</table>

Memory consumption of candidate sets on $P_5$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$P_2$</th>
<th>$P_4$</th>
<th>$P_6$</th>
<th>$P_2$</th>
<th>$P_4$</th>
<th>$P_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>yt</td>
<td></td>
<td></td>
<td>lj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage</td>
<td>34.8%</td>
<td>35.9%</td>
<td>8.1%</td>
<td>1.1%</td>
<td>2.1%</td>
<td>0.7%</td>
</tr>
</tbody>
</table>

Percentage of the Galloping search.
Backup

The Number of Matches ($P_0$ represents the triangle).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$lj$</th>
<th>$ot$</th>
<th>$uk$</th>
<th>$fs$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>$1.78 \times 10^8$</td>
<td>$6.28 \times 10^8$</td>
<td>$2.22 \times 10^9$</td>
<td>$4.17 \times 10^9$</td>
</tr>
<tr>
<td>$p_1$</td>
<td>$2.64 \times 10^{10}$</td>
<td>$1.28 \times 10^{11}$</td>
<td>$9.15 \times 10^{11}$</td>
<td>$4.66 \times 10^{11}$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$3.95 \times 10^{10}$</td>
<td>$6.71 \times 10^{10}$</td>
<td>$1.11 \times 10^{12}$</td>
<td>$1.85 \times 10^{11}$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$5.22 \times 10^9$</td>
<td>$3.22 \times 10^9$</td>
<td>$1.07 \times 10^{11}$</td>
<td>$8.96 \times 10^9$</td>
</tr>
<tr>
<td>$p_4$</td>
<td>$2.62 \times 10^{13}$</td>
<td>$4.97 \times 10^{13}$</td>
<td>$9.42 \times 10^{14}$</td>
<td>$5.47 \times 10^{13}$</td>
</tr>
<tr>
<td>$p_5$</td>
<td>$7.38 \times 10^{15}$</td>
<td>$4.01 \times 10^{15}$</td>
<td>$6.13 \times 10^{17}$</td>
<td>$1.34 \times 10^{15}$</td>
</tr>
<tr>
<td>$p_6$</td>
<td>$9.56 \times 10^{12}$</td>
<td>$2.60 \times 10^{12}$</td>
<td>$4.01 \times 10^{14}$</td>
<td>$3.18 \times 10^{12}$</td>
</tr>
<tr>
<td>$p_7$</td>
<td>$2.46 \times 10^{11}$</td>
<td>$1.58 \times 10^{10}$</td>
<td>$1.16 \times 10^{13}$</td>
<td>$2.17 \times 10^{10}$</td>
</tr>
</tbody>
</table>