Graph matching for BSP

Rob H. Bisseling

Mathematical Institute, Utrecht University
Joint work with Fredrik Manne (Bergen, Norway)

KU Leuven, October 31, 2014
Matching
  Introduction
  Greedy matching

BSP algorithm for edge-weighted matching
  Sequential approximation algorithm
  BSP approximation algorithm

Conclusion
Matchmaker, Matchmaker, Make me a match

From the film *Fiddler on the roof*

- Hodel: Well, somebody has to arrange the matches. Young people can’t decide these things themselves.
- Hodel: For Papa, make him a scholar.
- Chava: For Mama, make him rich as a king.
Matching can win you a Nobel prize

**Marriage as an Economic Problem**

Lloyd Shapley and Alvin Roth win the Nobel Prize for showing the best way to match people with what they really want.

By Matthew Yglesias | Posted Monday, Oct. 15, 2012, at 1:51 PM ET

Source: Slate magazine October 15, 2012
Motivation of graph matching

- **Graph matching** is a pairing of neighbouring vertices.
- It has applications in
  - medicine: finding suitable *organ donors* for patients
  - social networks: finding *partners*
  - scientific computing: finding *pivot elements* in matrix computations
  - graph coarsening: making the graph smaller by merging *similar vertices* before partitioning it for parallel computations
  - bioinformatics: finding similarity in *Protein-Protein Interaction* networks
Motivation of greedy/approximation graph matching

- Optimal solution is possible in polynomial time.
- Time for weighted matching in graph $G = (V, E)$ is $O(mn + n^2 \log n)$ with $n = |V|$ the number of vertices, and $m = |E|$ the number of edges (Gabow 1990).
- The aim is 10 billion vertices, $n = 10^{10}$, with 1000 edges per vertex, i.e. $m = 10^{13}$.
- Thus, a time of $O(10^{23}) = 100,000,000$ Petaflop units is far too long. Fastest supercomputer today, the Tianhe-2, performs 34 Petaflop/s and would need 34 days.
- We need linear-time greedy or approximation algorithms.
Formal definition of graph matching

- A graph is a pair $G = (V, E)$ with vertices $V$ and edges $E$.
- All edges $e \in E$ are of the form $e = (v, w)$ for vertices $v, w \in V$.
- A matching is a collection $M \subseteq E$ of disjoint edges.
- Here, the graph is undirected, so $(v, w) = (w, v)$. 
A matching is maximal if we cannot enlarge it further by adding another edge to it.
A matching is **maximum** if it possesses the largest possible number of edges, compared to all other matchings.
Edge-weighted matching

If the edges are provided with weights $\omega : E \rightarrow \mathbb{R}_{>0}$, finding a matching $M$ which maximises

$$\omega(M) = \sum_{e \in M} \omega(e),$$

is called edge-weighted matching.

Greedy matching provides us with maximal matchings, but not necessarily with maximum possible weight.
Sequential greedy matching

- In random order, vertices \( v \in V \) select and match neighbours one-by-one.
- Here, we can pick
  - the first available neighbour \( w \) of \( v \) (greedy random matching)
  - the neighbour \( w \) with maximum \( \omega(v, w) \) (greedy weighted matching)
- Or: we sort the edges by weight, and successively match the vertices \( v \) and \( w \) of the heaviest available edge \((v, w)\) (greedy matching)
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Sequential greedy matching
Greedy is a 1/2-approximation algorithm

- **Weight** $\omega(M) \geq \omega_{\text{optimal}}/2$
- **Cardinality** $|M| \geq |M_{\text{card-max}}|/2$, because $M$ is maximal.
- **Time complexity** is $O(m \log m)$, because all edges must be sorted.
Parallel greedy matching: trouble

Suppose we match vertices simultaneously.
Parallel greedy matching: trouble

Two vertices each find an unmatched neighbour...
Parallel greedy matching: trouble

...but generate an invalid matching.
Dominant-edge algorithm

\[
\text{while } E \neq \emptyset \text{ do} \\
\quad \text{pick dominant edge } (v, w) \in E \\
\quad M := M \cup \{(v, w)\} \\
\quad E := E \setminus \{(x, y) \in E : x = v \lor x = w\} \\
\quad V := V \setminus \{v, w\} \\
\text{return } M
\]

- An edge \((v, w) \in E\) is dominant if

\[
\omega(v, w) = \max\{\omega(x, y) : (x, y) \in E \land (x = v \lor x = w)\}
\]
Dominant edge
Proof: algorithm is 1/2-approximation

- Let $M$ be the matching produced by the dominant-edge algorithm.
- Let $M^*$ be a maximum matching with weight $\omega_{\text{optimal}}$.
- Let $M^* = \{e_0^*, \ldots, e_{k-1}^*\}$. For each edge $e_i^* \in M^*$, if $e_i^* \in M$, then let $e_i = e_i^*$, otherwise let $e_i$ be the edge that removes $e_i^*$ from $E$ in the algorithm.
- It may happen that $e_i = e_j$ for $i \neq j$.
- $\omega(e_i) \geq \omega(e_i^*)$ for all $i$, since $e_i$ is locally dominant in the algorithm and removes $e_i^*$, or $e_i = e_i^*$. 
Proof (cont’d)

Every edge $e \in M$ can occur at most twice in the list of $e_i$'s, since it can remove at most 2 edges in $M^*$ from $E$.

\[
2\omega(M) \geq \sum_{i=0}^{k-1} \omega(e_i) \geq \sum_{i=0}^{k-1} \omega(e_i^*) = \omega_{\text{optimal}}
\]

Hence $\omega(M) \geq \omega_{\text{optimal}}/2$. 
Sequential approximation algorithm: initialisation

function \texttt{SeqMatching}(V, E)

\textbf{for all} \ v \ \in \ V \ \textbf{do}

\hspace{1cm} \text{pref}(v) = \text{null}

\hspace{1cm} D := \emptyset

\hspace{1cm} M := \emptyset

\{ \text{Find dominant edges} \}

\textbf{for all} \ v \ \in \ V \ \textbf{do}

\hspace{1cm} Adj_v := \{ w \in V : (v, w) \in E \}

\hspace{1cm} \text{pref}(v) := \arg\max\{ \omega(v, w) : w \in Adj_v \}

\hspace{1cm} \textbf{if} \ \text{pref}(\text{pref}(v)) = v \ \textbf{then}

\hspace{2cm} D := D \cup \{ v, \text{pref}(v) \}

\hspace{2cm} M := M \cup \{ (v, \text{pref}(v)) \}
Mutual preferences

Outline
Matching
Introduction
Greedy matching
BSP matching
Approximation
BSP algorithm
Conclusion
Non-mutual preferences
Sequential approximation algorithm: main loop

while $D \neq \emptyset$ do
    pick $v \in D$
    $D := D \setminus \{v\}$
    for all $x \in Adj_v \setminus \{pref(v)\} : (x, pref(x)) \notin M$ do
        $Adj_x := Adj_x \setminus \{v\}$
        $pref(x) := \text{argmax}\{\omega(x, w) : w \in Adj_x\}$
        if $pref(pref(x)) = x$ then
            $D := D \cup \{x, pref(x)\}$
            $M := M \cup \{(x, pref(x))\}$
    return $M$
Properties of the dominant-edge algorithm

- Dominant-edge algorithm is a 1/2-approximation:
  \[ \omega(M) \geq \omega_{\text{optimal}}/2 \]

- Dominant edge means mutual preference:
  \[ v = \text{pref}(w) \text{ and } w = \text{pref}(v). \]

- Dominance is a local property: easy to parallelise.
- Algorithm keeps going until set of dominant vertices \( D \) is empty and matching \( M \) is maximal.
- Assumption without loss of generality: weights are unique. Otherwise, use vertex numbering to break ties.
Time complexity

- Linear time complexity $O(|E|)$ if edges of each vertex are sorted by weight.
- Sorting costs are

$$\sum_{v} \deg(v) \log \deg(v) \leq \sum_{v} \deg(v) \log \Delta = 2|E| \log \Delta,$$

where $\Delta$ is the maximum vertex degree.
- This algorithm is based on a dominant-edge algorithm by Preis (1999), called LAM, which is linear-time $O(|E|)$, does not need sorting, and also is a $1/2$-approximation, but is hard to parallelise.
Parallel algorithm (Manne & Bisseling, 2007)

- Processor $P(s)$ has vertex set $V_s$, with

\[\bigcup_{s=0}^{p-1} V_s = V\]

and $V_s \cap V_t = \emptyset$ if $s \neq t$.

- This is a $p$-way partitioning of the vertex set.
Halo vertices

- The adjacency set $\text{Adj}_v$ of a vertex $v$ may contain vertices $w$ from another processor.
- We define the set of halo vertices

$$H_s = \bigcup_{v \in V_s} \text{Adj}_v \setminus V_s$$

- The weights $\omega(v, w)$ are stored with the edges, for all $v \in V_s$ and $w \in V_s \cup H_s$.
- $E_s = \{(v, w) \in E : v \in V_s\}$ is the subset of all the edges connected to $V_s$. 
Parallel algorithm for $P(s)$: initialisation

function \textsc{ParMatching}(V_s, H_s, E_s, \text{distribution } \phi) 
    \textbf{for all } v \in V_s \textbf{ do} 
    \hspace{1cm} \text{pref}(v) = \text{null}
    \hspace{1cm} D_s := \emptyset
    \hspace{1cm} M_s := \emptyset

\{ \text{Find dominant edges} \}
\textbf{for all } v \in V_s \textbf{ do} 
    \hspace{1cm} \text{Adj}_v := \{w \in V_s \cup H_s : (v, w) \in E_s\}
    \hspace{1cm} \text{SetNewPreference}(v, \text{Adj}_v, \text{pref}, V_s, D_s, M_s, \phi)
\text{Sync}
Setting a vertex preference

\begin{align*}
\text{function } & \text{SetNewPreference}(v, \text{Adj}, V, D, M, \phi) \\
\text{pref}(v) & := \arg\max \{\omega(v, w) : w \in \text{Adj}\} \\
\text{if } & \text{pref}(v) \in V \text{ then} \\
\text{if } & \text{pref}(\text{pref}(v)) = v \text{ then} \\
D & := D \cup \{v, \text{pref}(v)\} \\
M & := M \cup \{(v, \text{pref}(v))\} \\
\text{else} \\
\text{put proposal}(v, \text{pref}(v)) \text{ in } P(\phi(\text{pref}(v)))
\end{align*}
How to propose

Source: www.theguardian.com

\[ \text{proposal}(v, w): v \text{ proposes to } w \]
Parallel algorithm for \( P(s) \): main loop

\[
\text{while } D_s \neq \emptyset \text{ do}
\]

\[
\quad \text{pick } v \in D_s
\]

\[
\quad D_s := D_s \setminus \{v\}
\]

\[
\quad \text{for all } x \in \text{Adj}_v \setminus \{\text{pref}(v)\} : (x, \text{pref}(x)) \notin M_s \text{ do}
\]

\[
\quad \quad \text{if } x \in V_s \text{ then}
\]

\[
\quad \quad \quad \text{Adj}_x := \text{Adj}_x \setminus \{v\}
\]

\[
\quad \quad \quad \text{SetNewPreference}(x, \text{Adj}_x, \text{pref}, V_s, D_s, M_s, \phi)
\]

\[
\quad \quad \text{else } \{x \in H_s\}
\]

\[
\quad \quad \quad \text{put unavailable}(v, x) \text{ in } P(\phi(x))
\]

\[
\text{Sync}
\]
Parallel algorithm for $P(s)$: communication

for all messages $m$ received do
  if $m = proposal(x, y)$ then
    if $\text{pref}(y) = x$ then
      $D_s := D_s \cup \{y\}$
      $M_s := M_s \cup \{(x, y)\}$
      put $\text{accepted}(x, y)$ in $P(\phi(x))$
  if $m = accepted(x, y)$ then
    $D_s := D_s \cup \{x\}$
    $M_s := M_s \cup \{(x, y)\}$
  if $m = unavailable(v, x)$ then
    if $(x, \text{pref}(x)) \notin M_s$ then
      $\text{Adj}_x := Adj_x \setminus \{v\}$
      $\text{SetNewPreference}(x, \text{Adj}_x, \text{pref}, V_s, D_s, M_s, \phi)$
Termination

- The algorithm alternates supersteps of computation running the main loop and communication handling the received messages.
- The whole algorithm can terminate when no messages have been received by processor $P(s)$ and the local set $D_s$ is empty, for all $s$.
- This can be checked at every synchronisation point.
Load balance

- Processors can have different amounts of work, even if they have the same number of vertices or edges.
- Use can be made of a global clock based on ticks, the unit of time needed to handle a vertex $x$ (in $O(1)$).
- After every $k$ ticks, everybody synchronises.
Synchronisation frequency

- Guidance for the choice of $k$ is provided by the BSP parameter $l$, the cost of a global synchronisation.
- Choosing $k \geq l$ guarantees that at most 50% of the total time is spent in synchronisation.
- Choosing $k$ sufficiently small will cause all processors to be busy during most supersteps.
- Good choice: $k = 2l$?
Sending messages

- The BSP system takes care that messages are sent automatically, in bulk. A useful BSPlib primitive for doing this is `bsp_send`.
- In the next superstep, all received messages are read (using `bsp_move`) and processed.
- Google’s Pregel system (Malewicz 2010) follows this BSP style.
Further improvement: edge-based (2D) distribution

<table>
<thead>
<tr>
<th>Name</th>
<th>SpMV</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1D</td>
<td>2D</td>
</tr>
<tr>
<td>rw9 (af_shell10)</td>
<td>113</td>
<td>105</td>
</tr>
<tr>
<td>rw10 (boneS10)</td>
<td>150</td>
<td>145</td>
</tr>
<tr>
<td>rw11 (Stanford)</td>
<td>340</td>
<td>141</td>
</tr>
<tr>
<td>rw12 (gupta3)</td>
<td>710</td>
<td>44</td>
</tr>
<tr>
<td>rw13 (St.Berk.)</td>
<td>716</td>
<td>448</td>
</tr>
<tr>
<td>rw14 (F1)</td>
<td>139</td>
<td>130</td>
</tr>
<tr>
<td>sw1 (small world)</td>
<td>1,007</td>
<td>417</td>
</tr>
<tr>
<td>sw2</td>
<td>1,957</td>
<td>829</td>
</tr>
<tr>
<td>sw3</td>
<td>2,017</td>
<td>832</td>
</tr>
<tr>
<td>er1 (random)</td>
<td>1,856</td>
<td>1,133</td>
</tr>
<tr>
<td>er2</td>
<td>3,451</td>
<td>1,841</td>
</tr>
<tr>
<td>er3</td>
<td>5,476</td>
<td>2,569</td>
</tr>
</tbody>
</table>


Source: Patwary, Bisseling, Manne (2010).
MulticoreBSP enables shared-memory BSP

Introduction

MulticoreBSP brings Bulk Synchronous Parallel (BSP) programming to modern multicore processors. BSP programming leads to high-performance codes:

![Speed of SpMV multiplication on a 64-core machine](image)

Matching with MulticoreBSP

- BSP program can remain the same, giving portability.
- To exploit the ease of reading data in shared memory, the `bsp_direct_get` is available in MulticoreBSP.
- This performs the communication immediately and blocks until the communication has been carried out.
- Possible use: replace the set $M_s$ of matched edges by a boolean array $matched_s$ marking the local matched vertices.
- This array can be read by all processors using `bsp_direct_get`, to replace the check $(x, \text{pref}(x)) \notin M_s$. 
Conclusions and outlook

- BSP is extremely suitable for parallel graph computations:
  - no need to worry about communication because we buffer messages until the next synchronisation;
  - no need for send-receive pairs;
  - BSP cost model gives synchronisation frequency;
  - correctness proof of algorithm becomes simpler;
  - no deadlock possible.