A Comparison Between Two Matching Heuristics

Ludvig Sundström

What was implemented

Algorithm: Edmonds Cardinality Matching Algorithm (The blossom algorithm) implemented according to to Korte, Vygen 2002, pg. 224-225.

Runtime: $O(|V|^3)$

***Heuristics**:*

1. **Greedy Maximal Matching.**
   Runtime: $O(|E|)$
2. **Expand-Contract** after specification by Adrian (EC)
   Runtime: $O(|E||V|)$

Edmonds Blossom Algorithm For Unweighted Graphs

- Works by growing an alternating forest, and augmenting vertex-root paths.
- When odd cycles occur, vertices in the alternating tree might be both even and odd at the same time.
- To manage the odd cycles, add some additional logic to shrink cycles into pseudo-vertices.
- Proceed as usual.

Expand-Contract

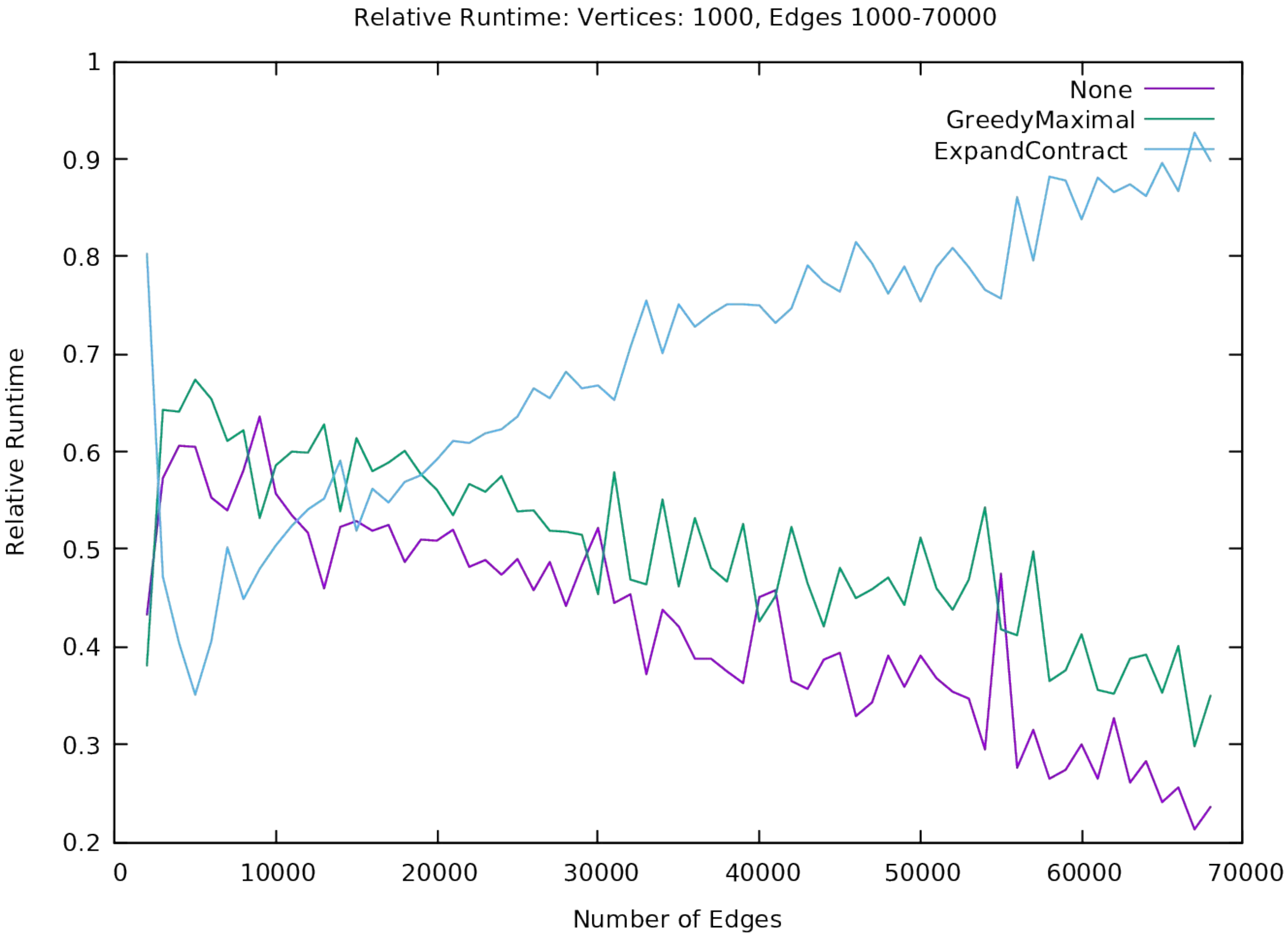
1. **Expand**: Turn the graph into a Bipartite graph
   a) For each vertex *v*, create *v1* and *v2*
   b) For edge *(v, w) create (v1, w2)* and *(w1, v2)*
2. **Find a maximum matching in the bipartite graph.**
3. **Contract:** Turn the graph into the original graph again.
   For all *(vx, wy)* in *M,* let all *(v, w)* be the found "matching" *M'*.
4. **Repair:** Look at the graph induced by *M'*
   a) Find all paths of even length and fix them.
   b) Find all even cycles of even length and fix them.
   c) Find all Odd cycles of even length and fix them.

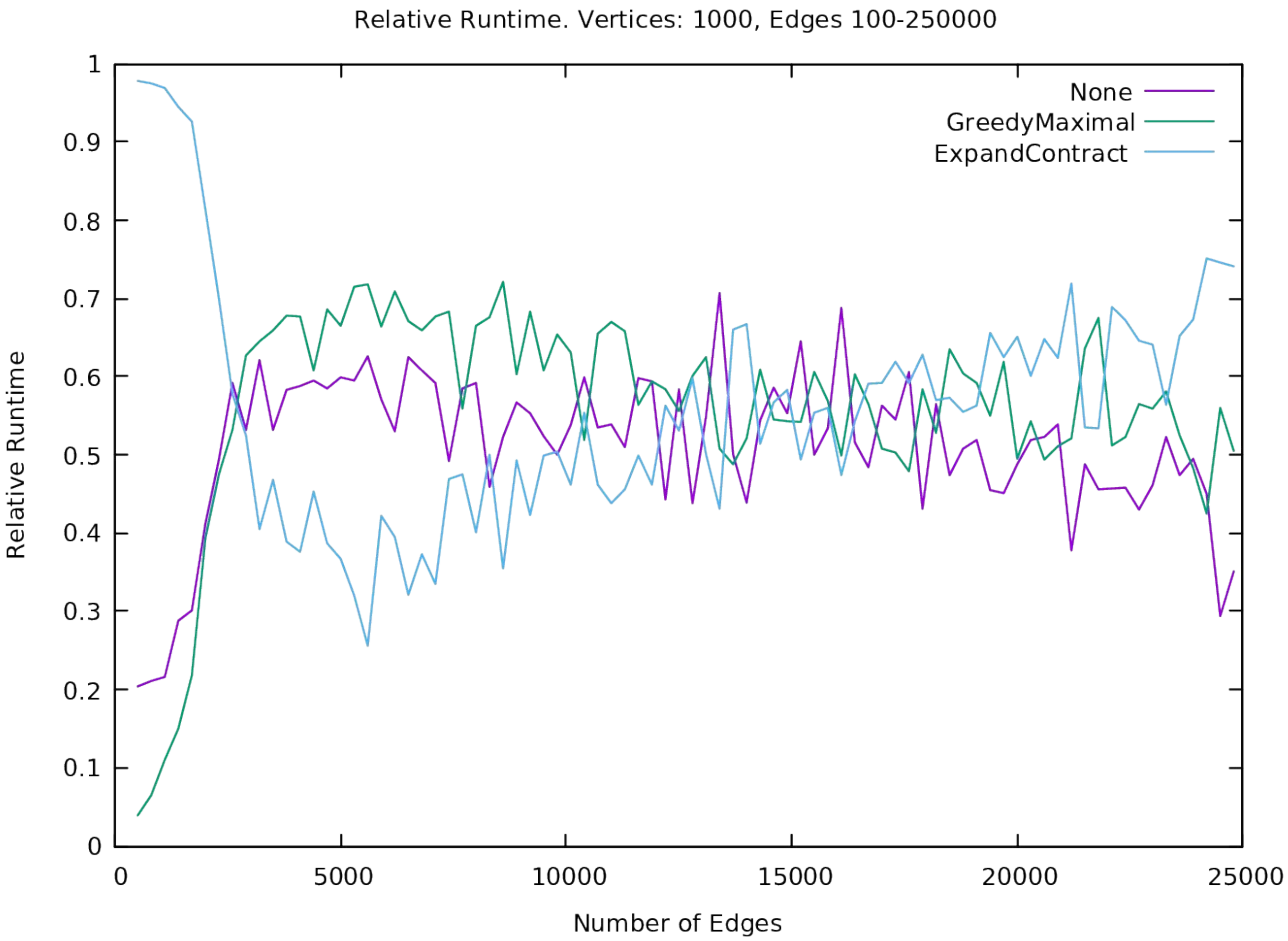
We fix a component by removing every second edge.

Runtime Comparison Experiment

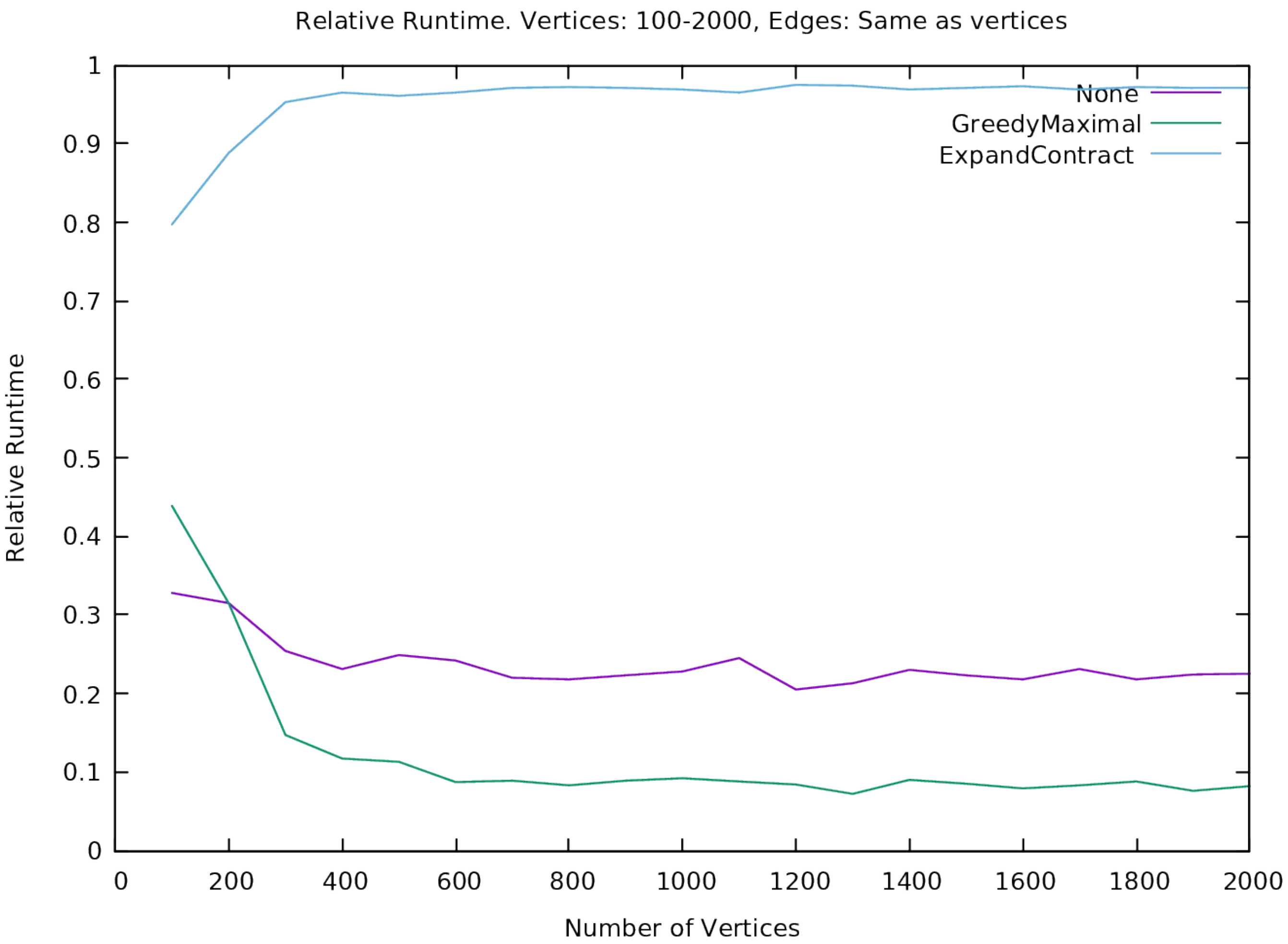
Procedure:

1. Generate a random graph of a specific size.
2. Start timer.
3. Find a maximal matching with a specified Heuristic.
4. Run Edmonds Blossom algorithm with the maximal matching.
5. Stop timer.
6. Compare relative runtimes.


Relative Runtime: Vertices: 1000, Edges 1000-70000
None
GreedyMaximal
ExpandContract
Relative Runtime
Number of Edges
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
10000
20000
30000
40000
50000
60000
70000



Relative Runtime. Vertices: 1000, Edges 100-250000
None
GreedyMaximal
ExpandContract
Relative Runtime
Number of Edges
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5000
10000
15000
20000
25000


Relative Runtime. Vertices: 100-2000, Edges: Same as vertices
Relative Runtime
Number of Vertices
None
GreedyMaximal
ExpandContract
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
200
400
600
800
1000
1200
1400
1600
1800
2000

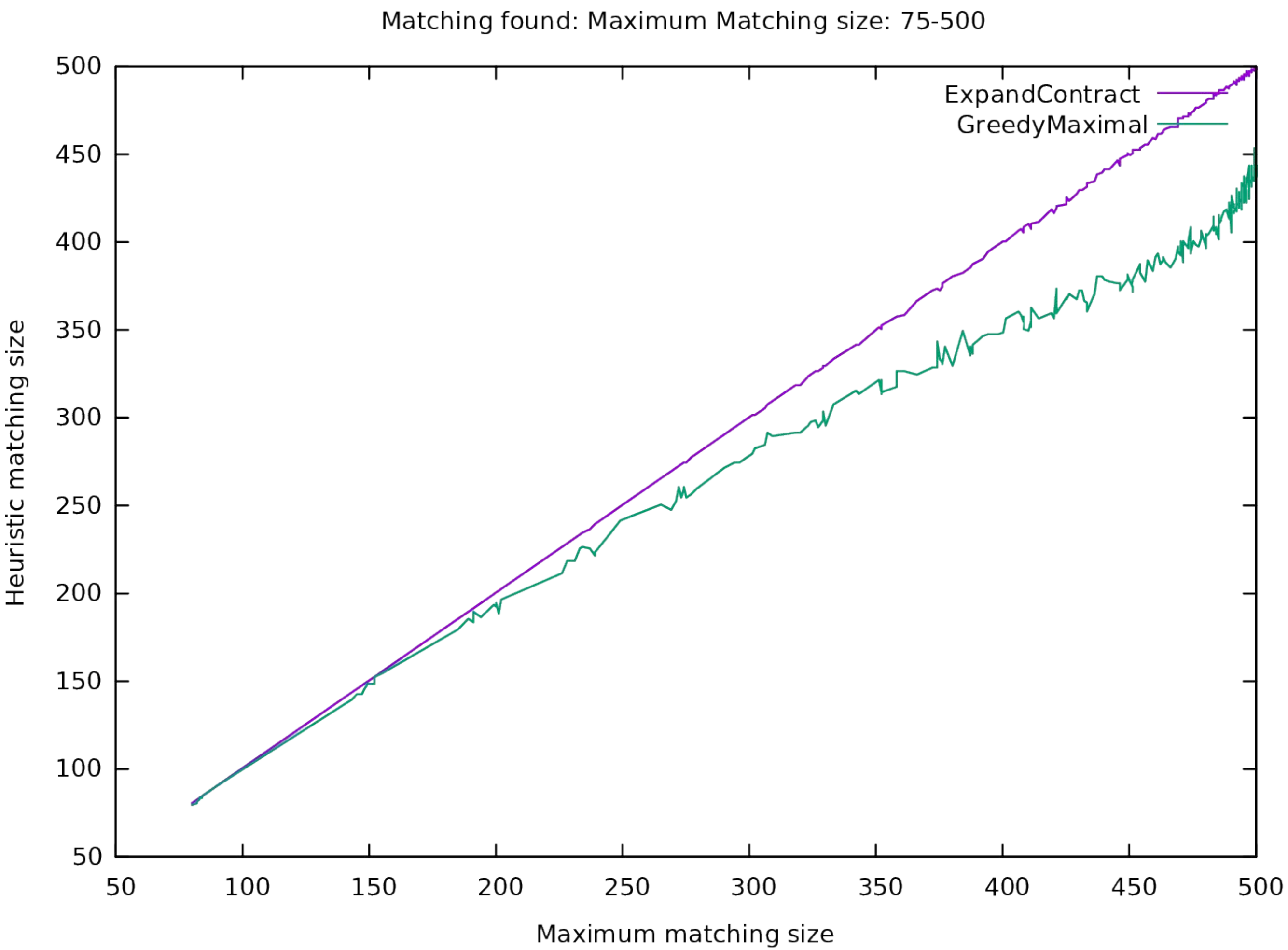
Experiment Conclusions: Runtime

1. Finding a Greedy Maximal matching first did not influence the runtime significantly.
2. Expand-Contract is fast for graphs of a specific vertex-edge ratio.
3. Expand-Contract is very slow for very dense graphs.
4. Changing the graph size did not change the relative runtimes of the Heuristics when keeping |vertex| : |edge| ratio fixed.

Matching Size Experiment

Procedure:

1. Generate a random graph.
2. Find a matching with Greedy Maximal.
3. Find a matching with Expand-Contract.
4. Compare the two matching sizes. with Maximum matching.



Matching found: Maximum Matching size: 75-500
ExpandContract
GreedyMaximal
Heuristic matching size
Maximum matching size
50
100
150
200
250
300
350
400
450
500

Experiment Conclusions: Matching Size

1. For very sparse graphs, Greedy-Maximal and Expand-Contract was fairly identical.
2. As denseness increases, Greedy-Maximal fell off but Expand-Contract almost always finds a near-maximum matching.
3. The average matching found by Greedy-Maximal was a ~ 0.88-approximation of M^* while Expand-contract found a ~ 0.98-approximation of M^*.

Programming language

Haskell Pros:

- Good set of standard data-structures to work with.
- Very easy to define new data structures.
- Pure functional programming forces a cleaner approach, and to think in terms of Types and Data instead of memory manipulation.

Haskell Cons:

- Random access requires non-idiomatic data-structures that makes programming difficult. You can not easily define a mutable array. Most Algorithm descriptions in textbooks assume imperative programming with mutable state.

Difficulties and Shortcomings

Difficulties:

- Understanding the specification of the algorithm.
- Translating the low level specification into a higher-level view suitable for implementation in haskell.

Known improvements to the algorithm:

- Use constant time data-structures for accessing state. Current implementation is based on Integer Sets which has a *log(n)* lookup but at most the number of bits of an integer.