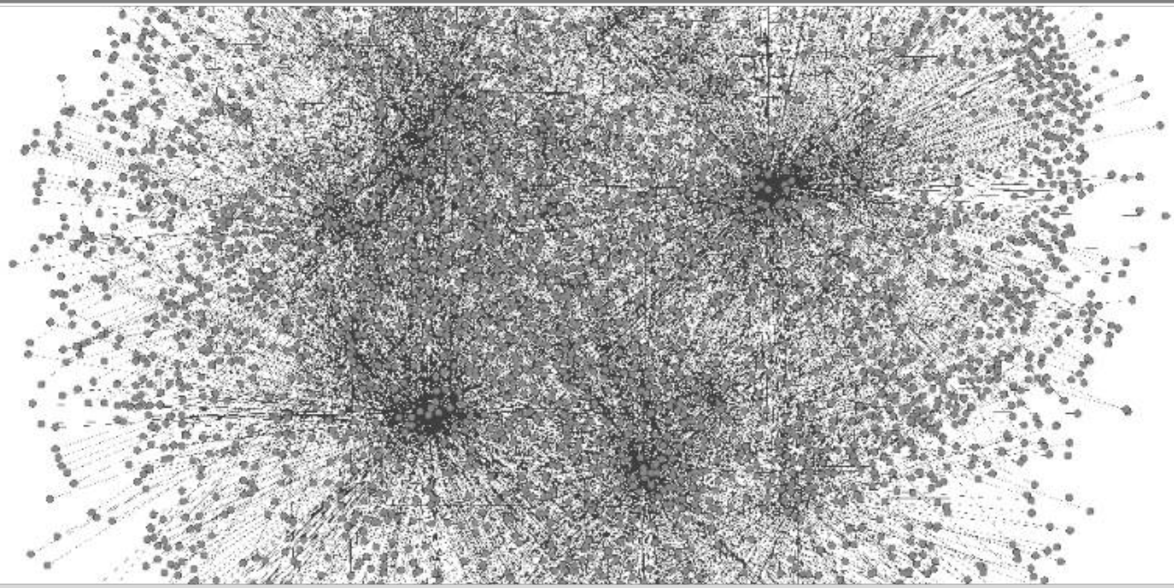
Karlsruhe Institute of Technology

Efficient-Parallel-C++

General Weighted Matching – Alan Mazankiewicz

Institute for Theoretical Informatics – Prof. Sanders



Overview

Matching Problem

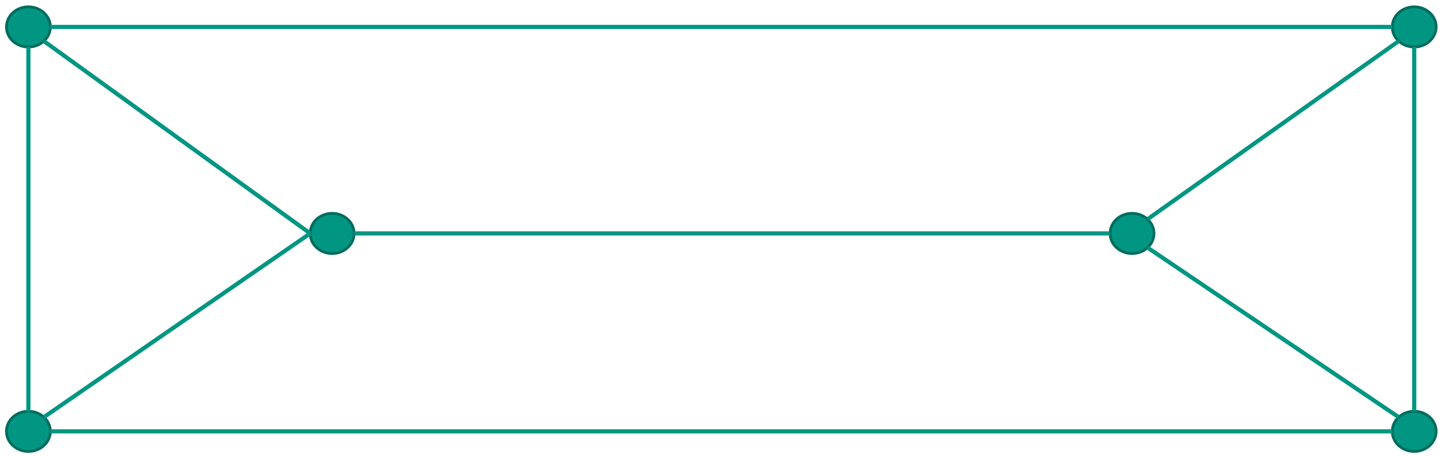
Algorithms

- Local Dominant
- Suitor
- Heavy Matching

Evaluation

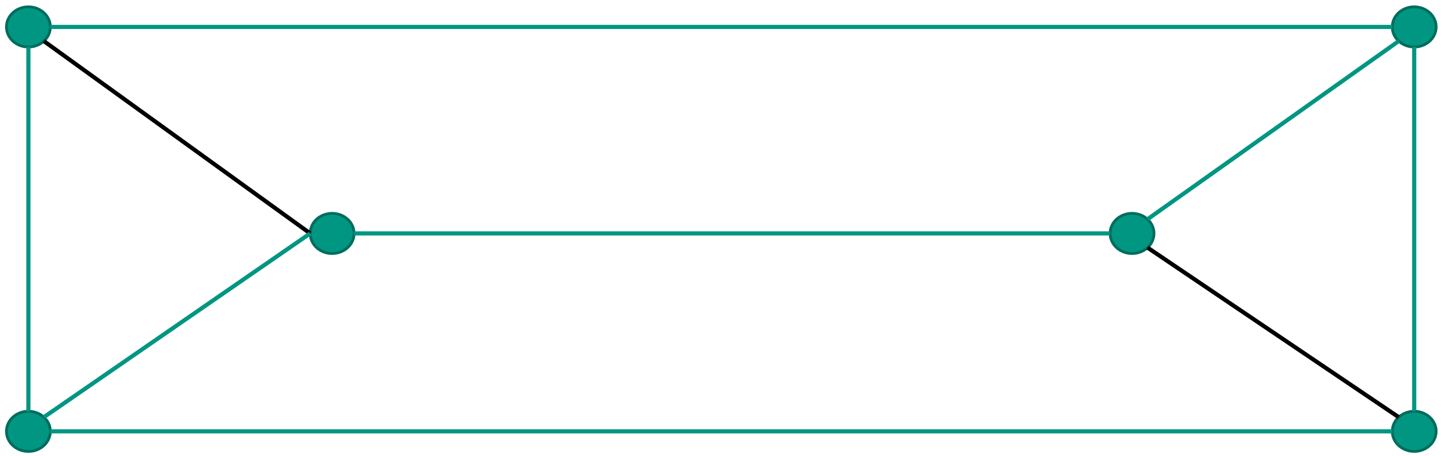
Definition - Matching

A **matching** in an undirected Graph $G = (V, E)$ is a subset of edges M such that no two elements of M have common endpoints



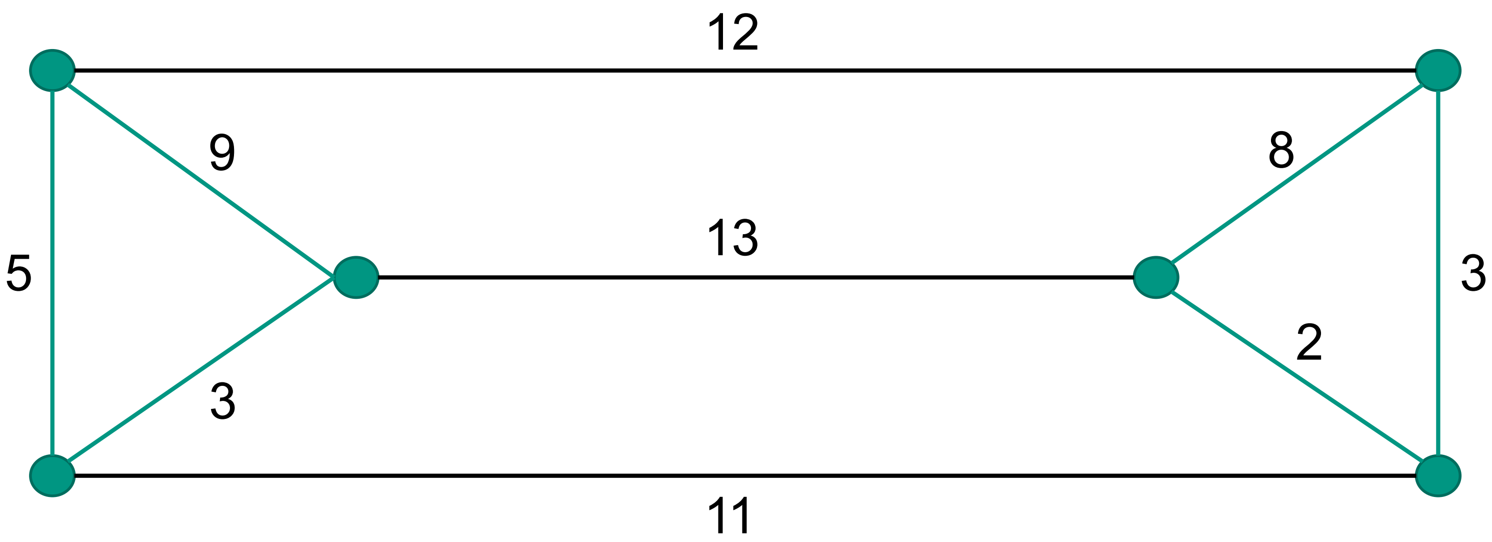
Definition - Matching

A **matching** in an undirected Graph $G = (V, E)$ is a subset of edges M that such no two elements of M have common endpoints



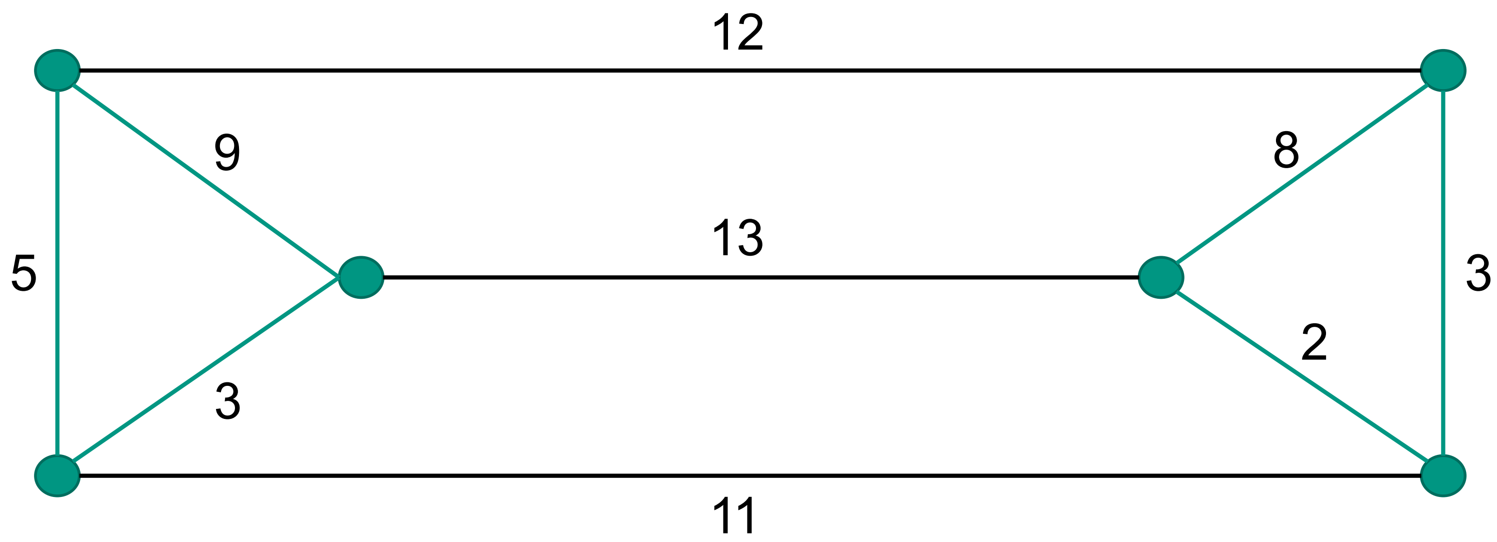
Definition – Maximum Weighted Matching

A **maximum weighted matching** of an undirected graph $G = (V, E, W)$ is a matching M with the largest possible sum* of weights



Definition – Maximum Weighted Matching

A **maximum weighted matching** of an undirected graph $G = (V, E, W)$ is a matching M with the largest possible sum* of weights



$O(nm + n^2 \log n)$ [Gabow 90]

Half-Approximation Algorithms

Sequential
Greedy – Avis
PGA' - Path Growing Algorithm - Drake and Hougardy
GPA - Global Paths Algorithm - Maue and Sanders
HEM – Heurisitc of Heavy Matching – Birn et al.

Sequential and Parallel
Locally Dominant Edges – Halappanavar et al.
Suitor – Manne and Halappanavar

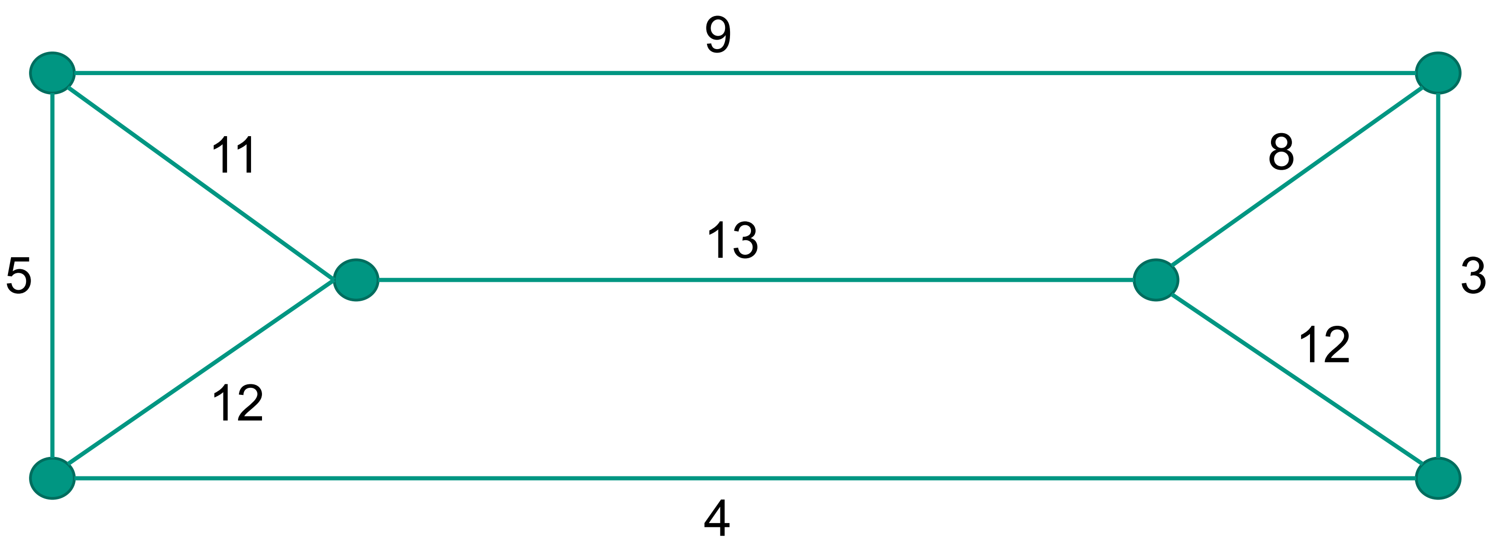
Half-Approximation Algorithms

Sequential
Greedy – Avis
PGA‘ - Path Growing Algorithm - Drake and Hougardy
GPA - Global Paths Algorithm - Maue and Sanders
HEM – Heurisitc of Heavy Matching – Birn et al.

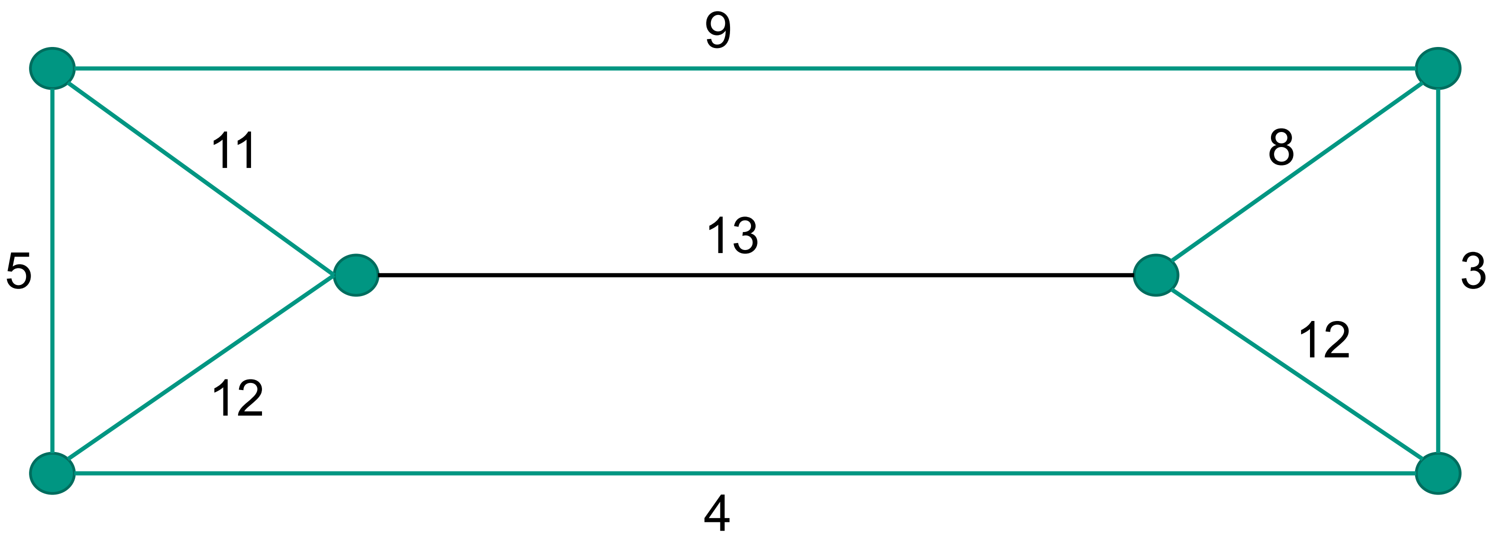
Sequential and Parallel
Locally Dominant Edges – Halappanavar et al.
Suitor – Manne and Halappanavar

Greedy

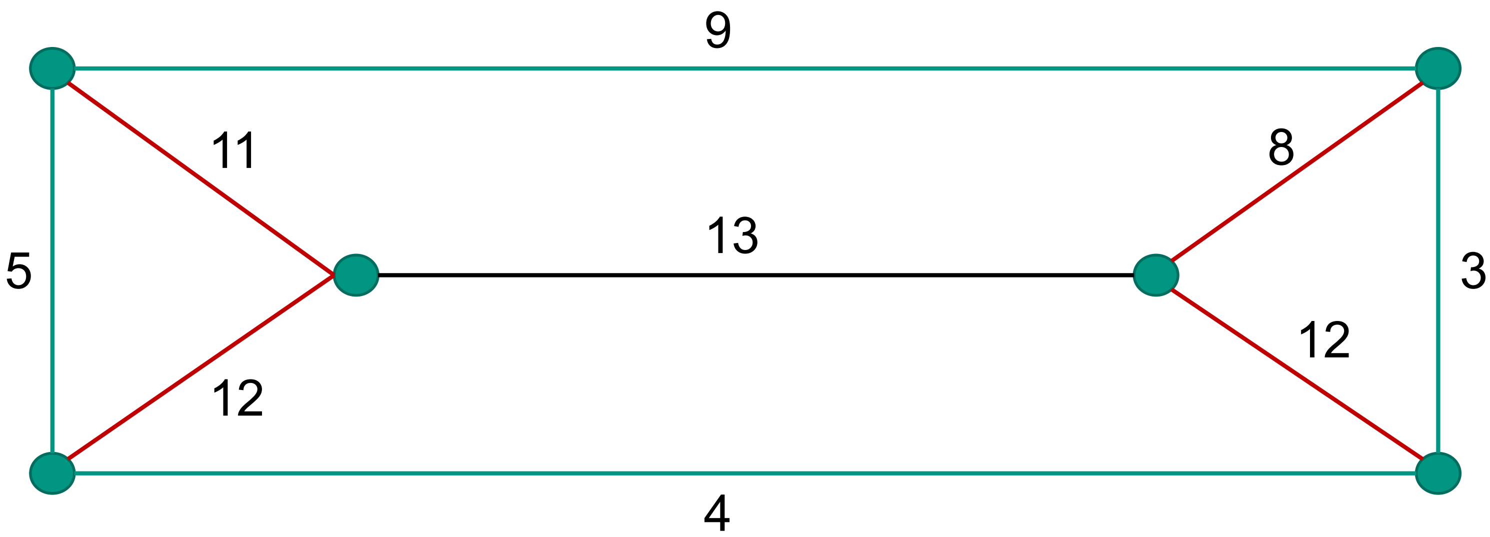
- Sort edges in descending order



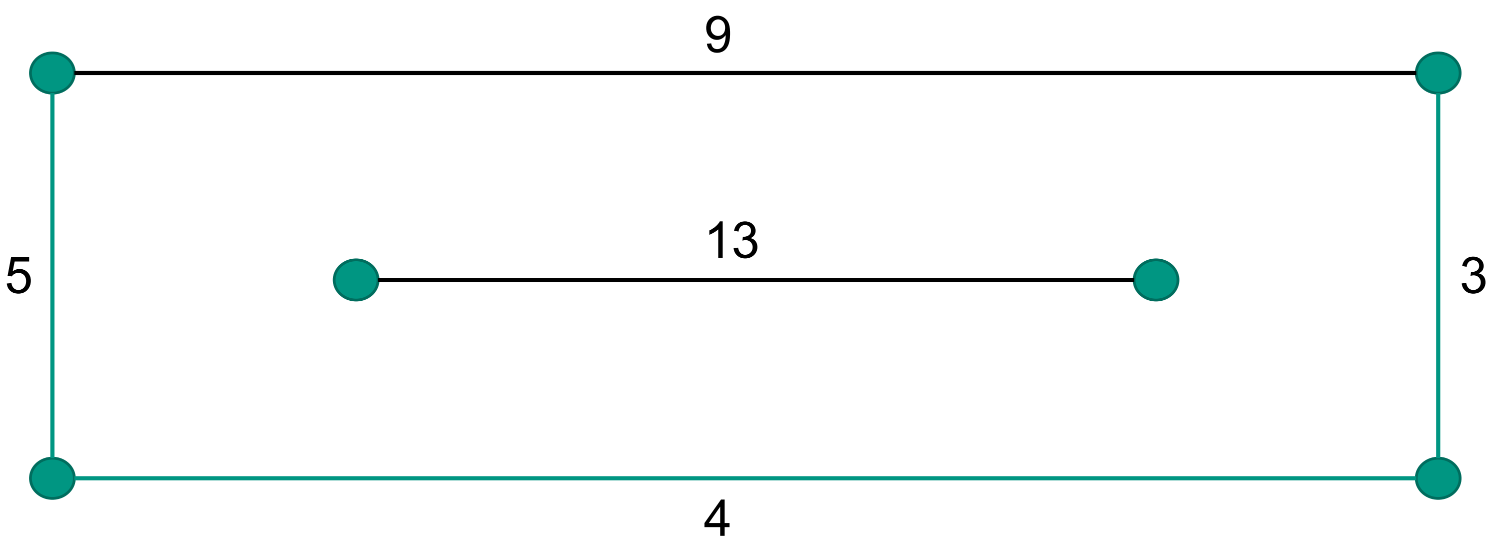
Greedy



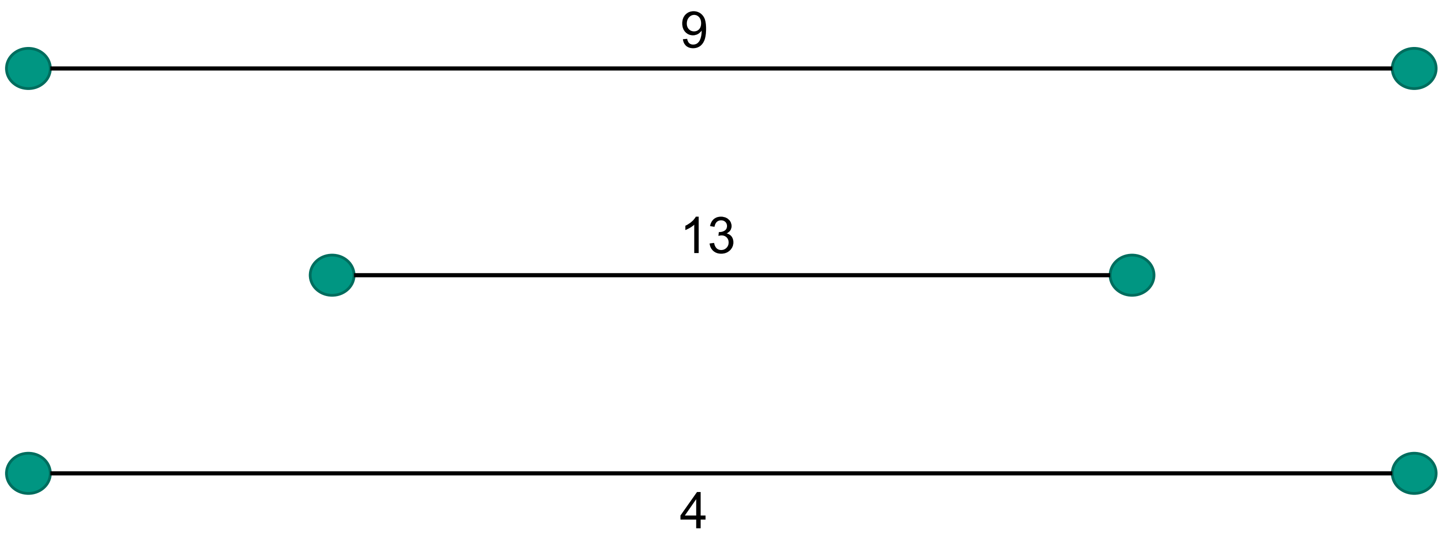
Greedy



Greedy

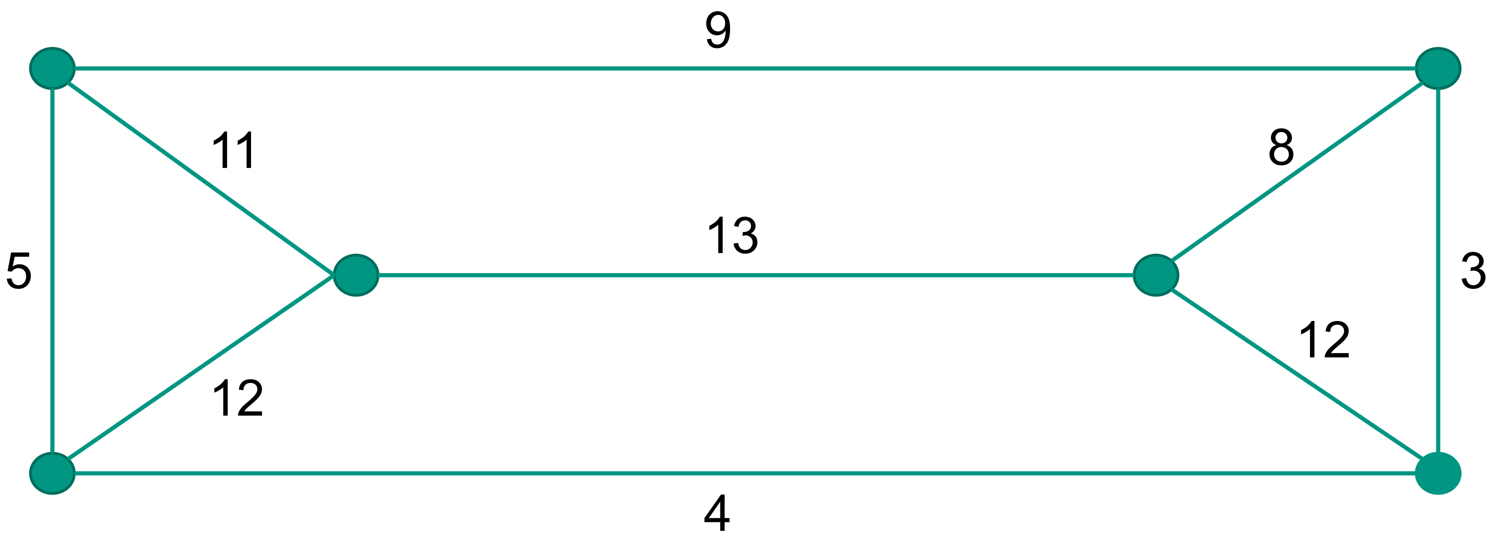


Greedy



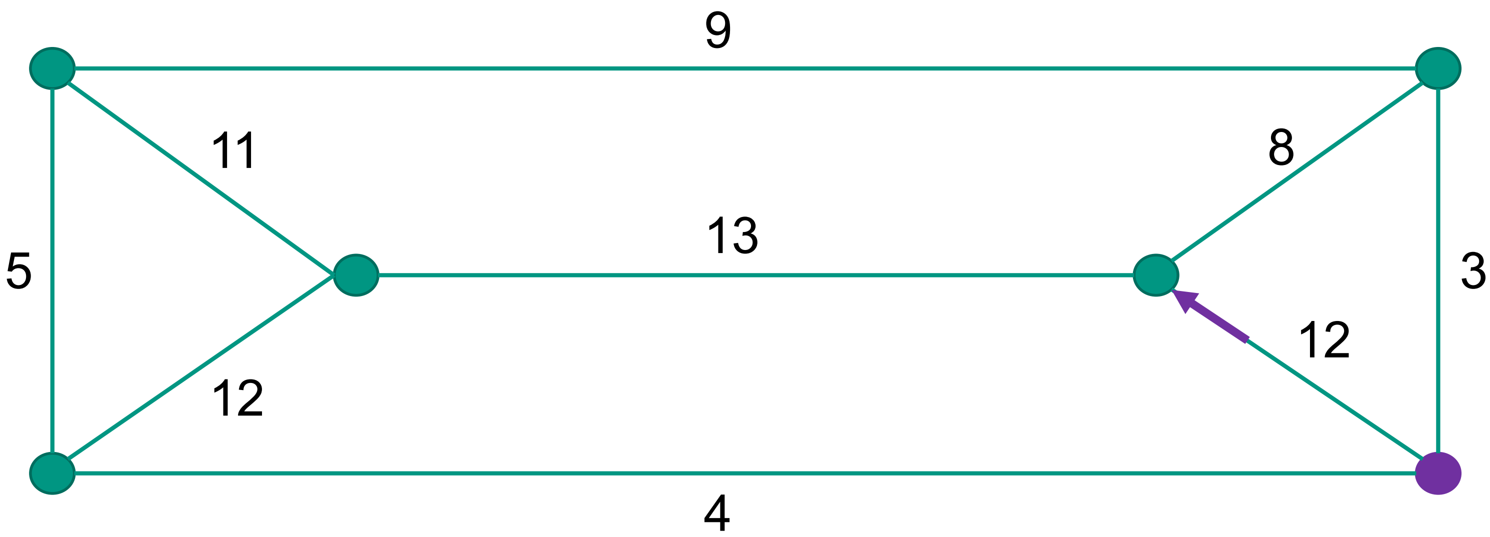
$$O(m + m \log n)$$

Local Dominant - Sequential



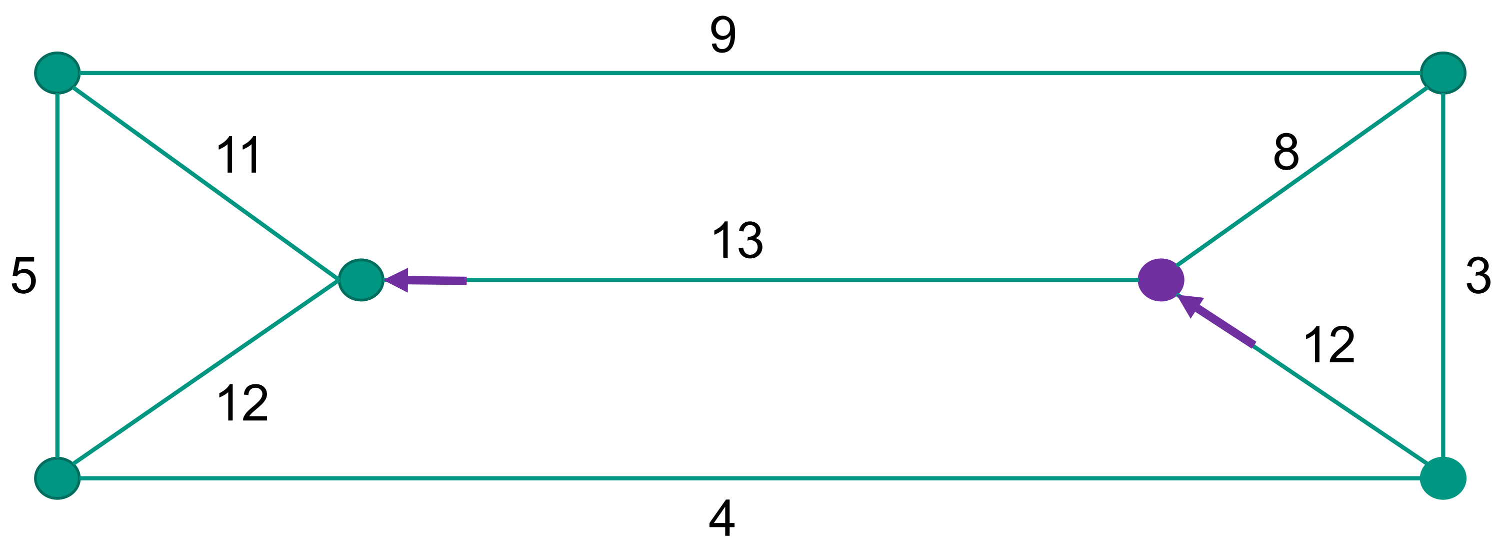
Local Dominant - Sequential

- Phase I
- Process each Vertex



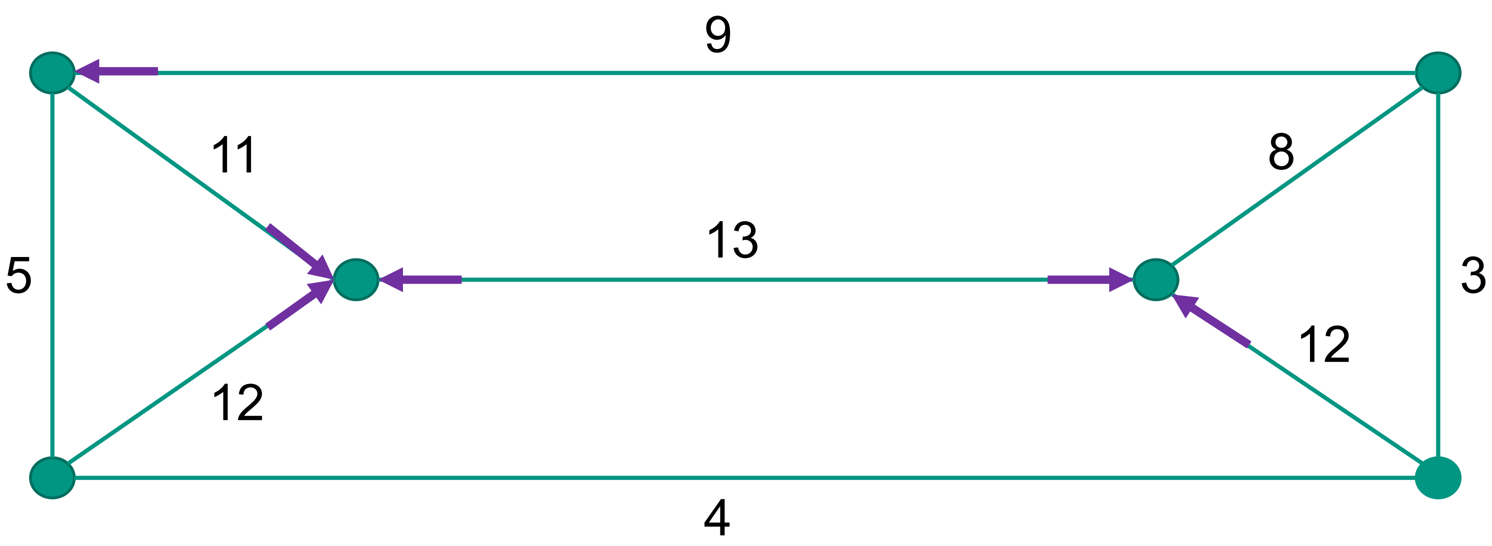
Local Dominant - Sequential

- Phase I
- Process each Vertex



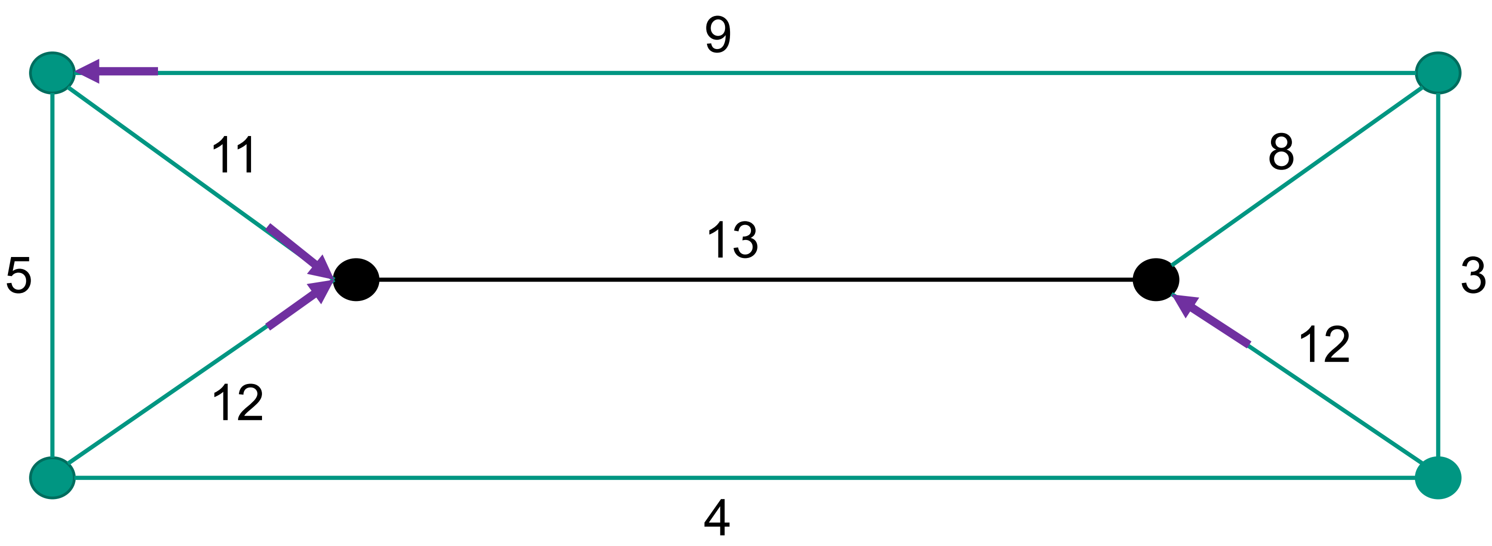
Local Dominant - Sequential

- Phase I
- Process each Vertex



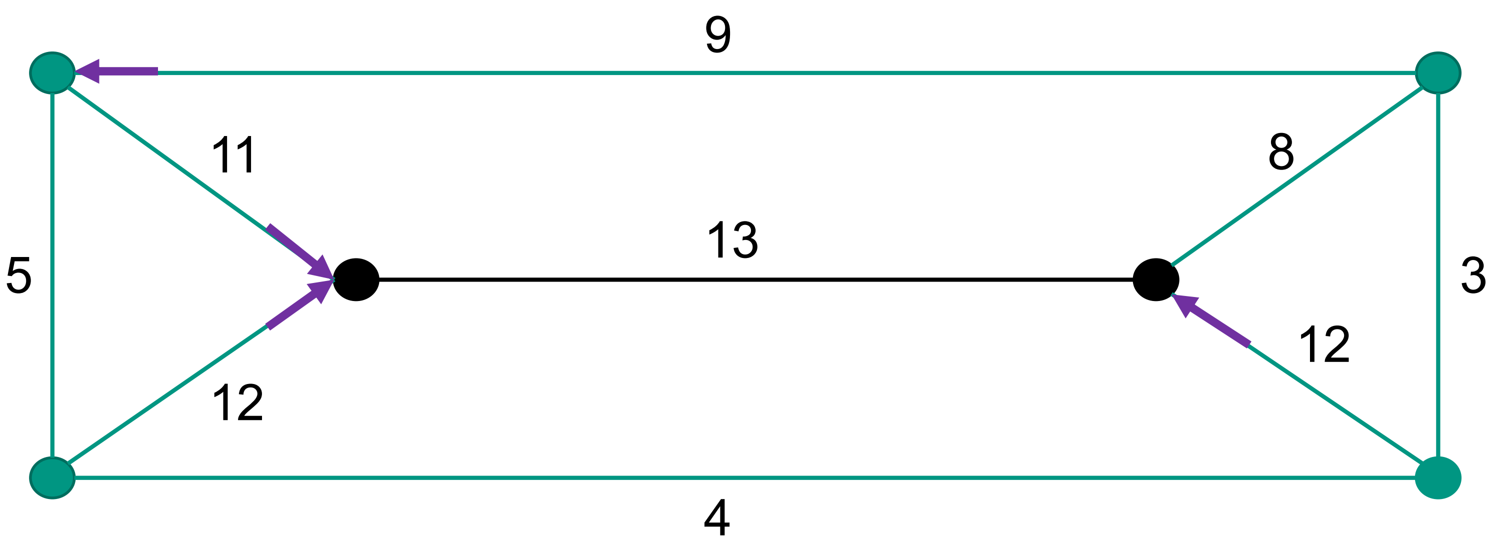
Local Dominant - Sequential

- Phase I
- Add Endpoints to Queue Q



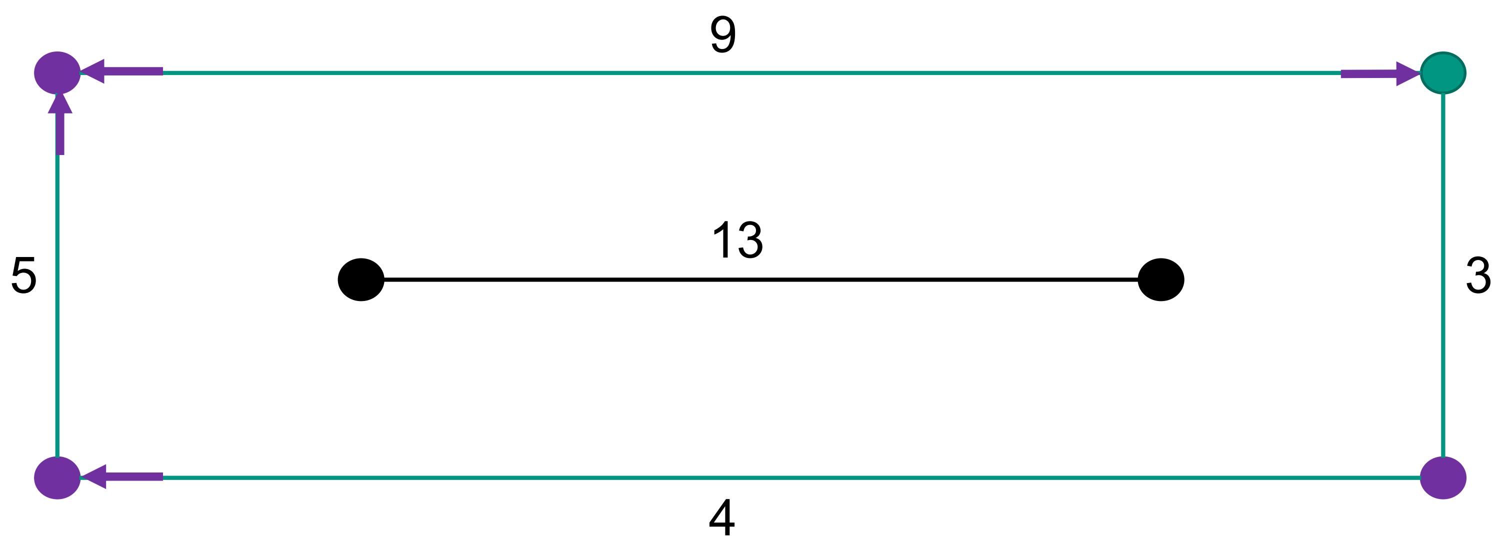
Local Dominant - Sequential

- Phase II - $while\ Q \neq \{\}$



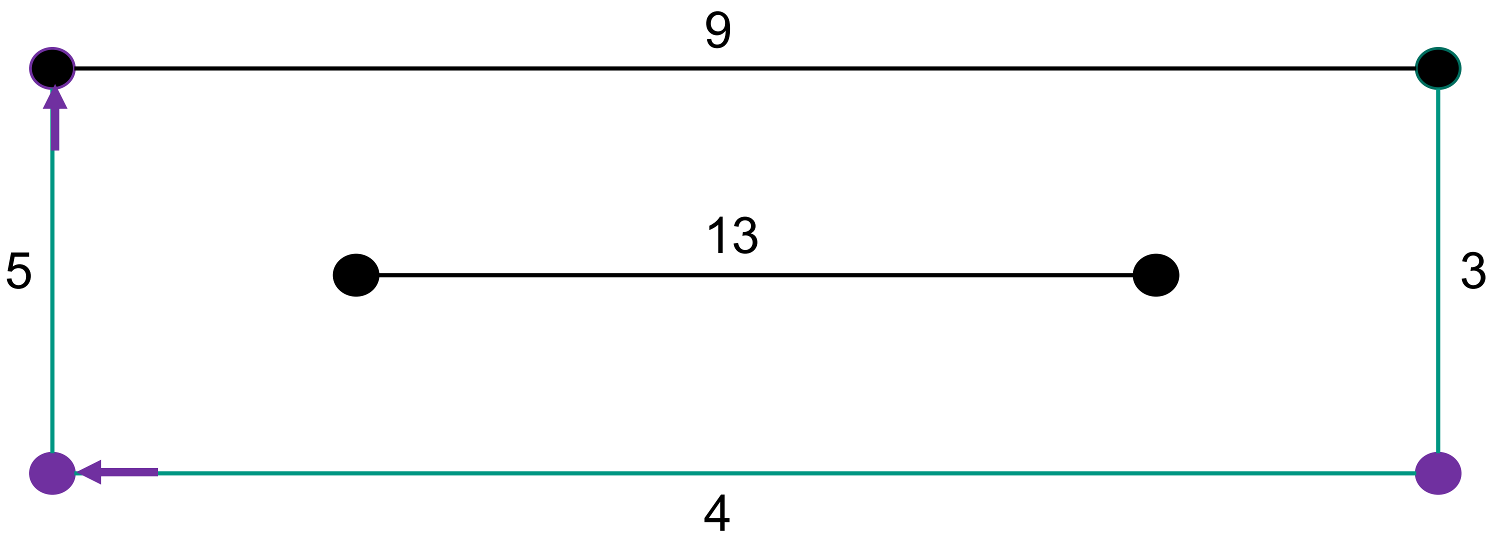
Local Dominant - Sequential

- Phase II - $while\ Q \neq \{\}$
- Process each vertex that endpoint is candidate of



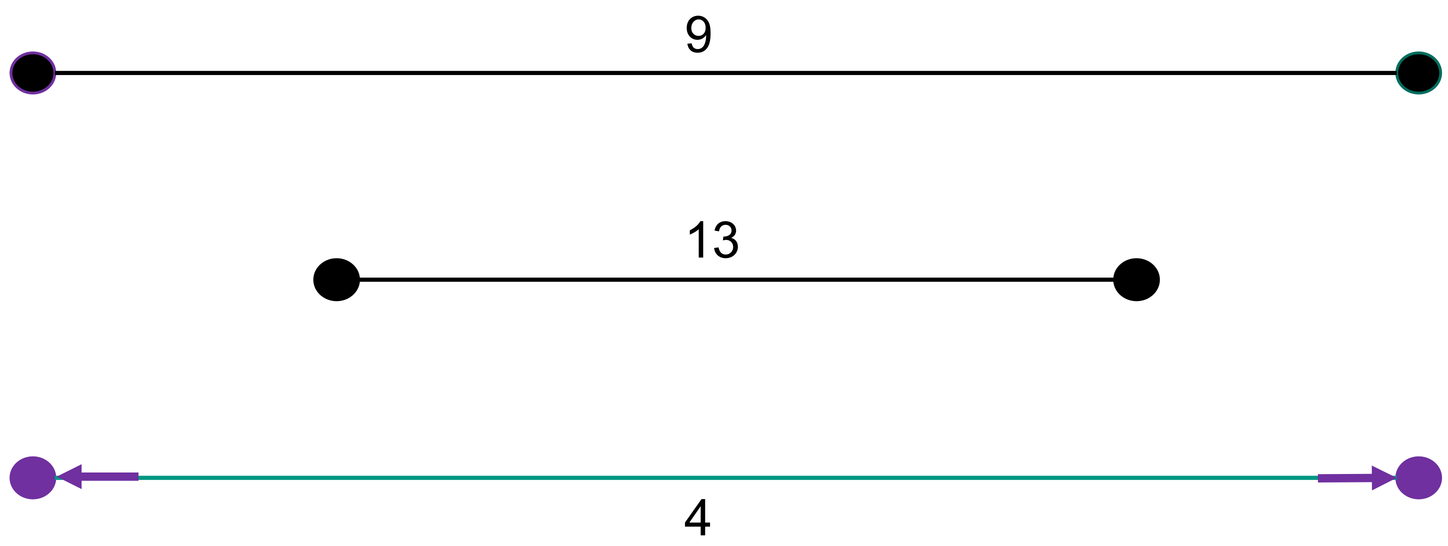
Local Dominant - Sequential

- Phase II - $while\ Q \neq \{\}$
- Process each vertex that endpoint is candidate of



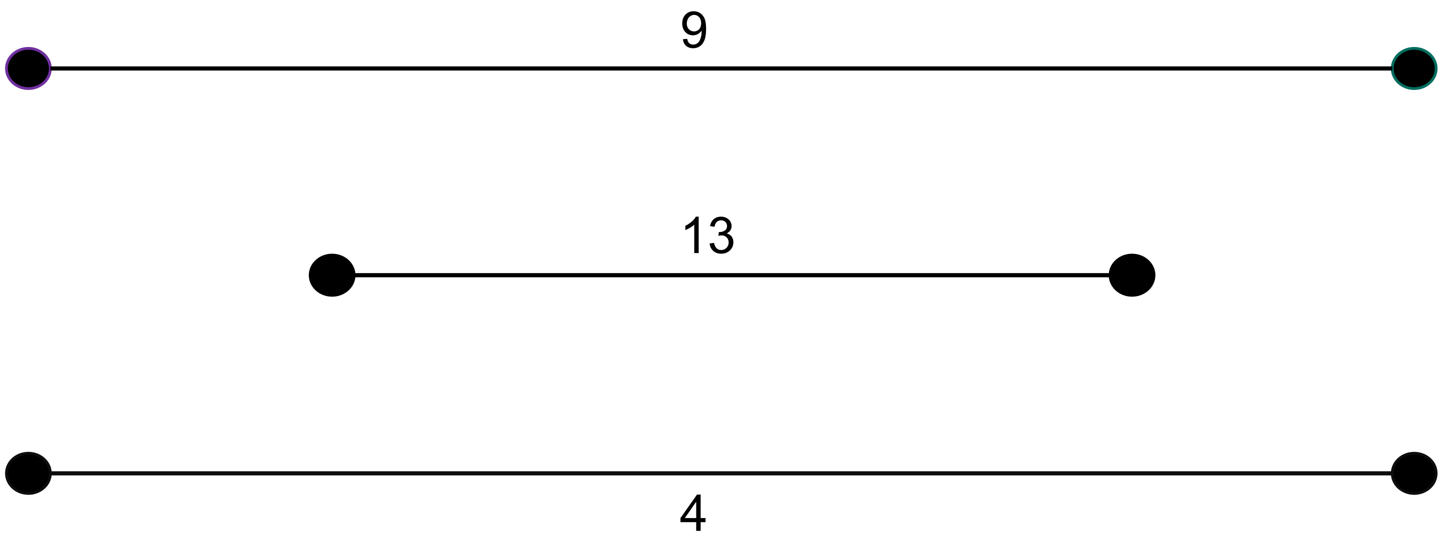
Local Dominant - Sequential

- Phase II - $while\ Q \neq \{\}$
- Process each vertex that endpoint is candidate of



Local Dominant - Sequential

- Same matching as Greedy



$$O(n + m * \Delta)$$

Local Dominant - Parallel

```
procedure Parallel-Queue(G(V, E), mate)
    Q_C ← ∅
    Q_N ← ∅
    —— Phase-1 ——
    for each u ∈ V in parallel do

        mate[u] ← ∅
        candidate[u] ← ∅
        max_wt ← −∞
        max_wt_id ← ∅
        for each v ∈ adj(u) do
            if (mate[v] = ∅) AND (max_wt < w(e_{u,v})) then
                max_wt ← w(e_{u,v})
                max_wt_id ← v
        candidate[u] ← max_wt_id
    for each u ∈ V in parallel do

        if candidate[candidate[u]] = u then
            mate[u] ← candidate[u]
            Q_C ← Q_C ∪ {u}

    —— Phase-2 ——
    while Q_C ≠ ∅ do
        for each u ∈ Q_C in parallel do
            for each v ∈ adj(u)\ do
                if candidate[v] = u then

                    ProcessVertex(v, Q_N)
        Q_C ← Q_N

        Q_N ← ∅
```

Source: Halappanavar et al.

Local Dominant - Parallel

```
void ParallelDominant::processVertex(NodeHandle u, moodycamel::ConcurrentQueue<NodeHandle> &Q, Mate &candidate)
{
    double max_wt = 0;
    NodeHandle max_wt_id = G.null_node();
    size_t u_id = G.nodeId(u);

    for (EdgeIterator t = G.beginEdges(u); t != G.endEdges(u); ++t)
    {
        if ((mate[*t] == G.null_node()) && (max_wt < G.edgeWeight(t)))
        {
            max_wt = G.edgeWeight(t);
            max_wt_id = G.node(*t);
        }
    }
    candidate[u_id] = max_wt_id;

    if (candidate[u_id] == G.null_node())
        return;

    if (G.nodeId(candidate[G.nodeId(candidate[u_id])]) == u_id)
    {

        mate[u_id] = candidate[u_id];
        mate[G.nodeId(candidate[u_id])] = u;
        Q.enqueue(u);
        Q.enqueue(candidate[u_id]);
    }
}
```

```
class ParallelDominant
{
public:
    using EdgeIterator = AdjacencyArray::EdgeIterator;
    using NodeHandle = AdjacencyArray::NodeHandle;
    using Mate = std::vector<NodeHandle>;
```

Suitor – Sequential

- $suitor(u)$ - Who points to u

```
for each u ∈ V do
    suitor(u) = NULL
    ws(u) = 0
for each u ∈ V do
    current = u
    done = False
    while not done do
        partner = suitor(current)
        heaviest = ws(current)
        for each v ∈ N(current) do
            if w(current, v) > heaviest and
            w(current, v) > ws(v) then
                partner = v
                heaviest = w(current, v)
        done = True
        if heaviest > 0 then
            y = suitor(partner)
            suitor(partner) = current
            ws(partner) = heaviest
            if y ≠ NULL then
                current = y
                done = False
```

Source: Manne and Halappanavar

Suitor – Sequential

- $O(m * \Delta)$

```
for each u ∈ V do
    suitor(u) = NULL
    ws(u) = 0
for each u ∈ V do
    current = u
    done = False
    while not done do
        partner = suitor(current)
        heaviest = ws(current)
        for each v ∈ N(current) do
            if w(current, v) > heaviest and
            w(current, v) > ws(v) then
                partner = v
                heaviest = w(current, v)
        done = True
        if heaviest > 0 then
            y = suitor(partner)
            suitor(partner) = current
            ws(partner) = heaviest
            if y ≠ NULL then
                current = y
                done = False
```

Source: Manne and Halappanavar

Suitor – Parallel with Lock

```
void ParallelSuitor::process(NodeHandle start, NodeHandle finish)
{
    for (NodeHandle u = start; u != finish; ++u)
    {
        NodeHandle current = u;
        bool done = false;

        while (!done)
        {
            NodeHandle partner = G.null_node();
            double heaviest = 0;

            for (EdgeIterator v = G.beginEdges(current); v != G.endEdges(current); ++v)
            {
                double weight_v = G.edgeWeight(v);
                if (weight_v > heaviest && weight_v > ws[*v])
                {
                    partner = G.node(*v);
                    heaviest = weight_v;
                }
            }
            done = true;
            if (heaviest > 0)
            {
                std::unique_lock<std::mutex> lock(mtx[G.nodeId(partner)]);

                if (ws[G.nodeId(partner)] < heaviest)
                {
                    NodeHandle y = suitor[G.nodeId(partner)];
                    suitor[G.nodeId(partner)] = current;
                    ws[G.nodeId(partner)] = heaviest;
                    lock.unlock();

                    if (y != G.null_node())
                    {
                        current = y;
                        done = false;
                    }
                } else
                    done = false;
            }
        }
    }
}
```

Graph Data Structure

- Nodes: [0, 3, …] → NodeHandle
- Edges: [1,2,3, 4,5,6 …] → EdgeIterator
- Weights: [11,12,13, 14, 15, 16, …] → Index of EdgeIterator

Suitor – Parallel Lockfree

```
void ParallelLocklessSuitor::process(NodeHandle start, NodeHandle finish)
{
    for (NodeHandle u = start; u != finish; ++u)
    {
        NodeHandle current = u;
        bool done = false;

        while (!done)
        {
            EdgeIterator partner = G.null_edge();
            double heaviest = 0;

            for (EdgeIterator v = G.beginEdges(current); v != G.endEdges(current); ++v)
            {
                double weight_v = v->second;
                if (weight_v > heaviest && weight_v > suitor[v->first].load()->second)
                {
                    partner = v;
                    heaviest = weight_v;
                }
            }
            done = true;
            if (heaviest > 0)
            {
                size_t partnerId = partner->first;
                EdgeIterator current_suitor = suitor[partnerId];
                if (current_suitor->second < heaviest)
                {
                    if (suitor[partnerId].compare_exchange_strong(current_suitor, partner))
                    {
                        NodeHandle y = G.edgeHead(current_suitor);
                        if (y != G.null_node())
                        {
                            current = y;
                            done = false;
                        }
                    } else
                        done = false;
                } else
                    done = false;
            }
        }
    }
}
```

Graph Data Structure

- Nodes: [0, 3, …] → NodeHandle
- Edges: [1,2,3, 4,5,6 …] → EdgeIterator
- Weights: [11,12,13, 14, 15, 16, …] → Index of EdgeIterator

- Nodes: [0, 3, …] → NodeHandle
- Edges: [(1,11),(2, 12),(3,13), (4, 14) …] → EdgeIterator

Suitor – Parallel Lockfree

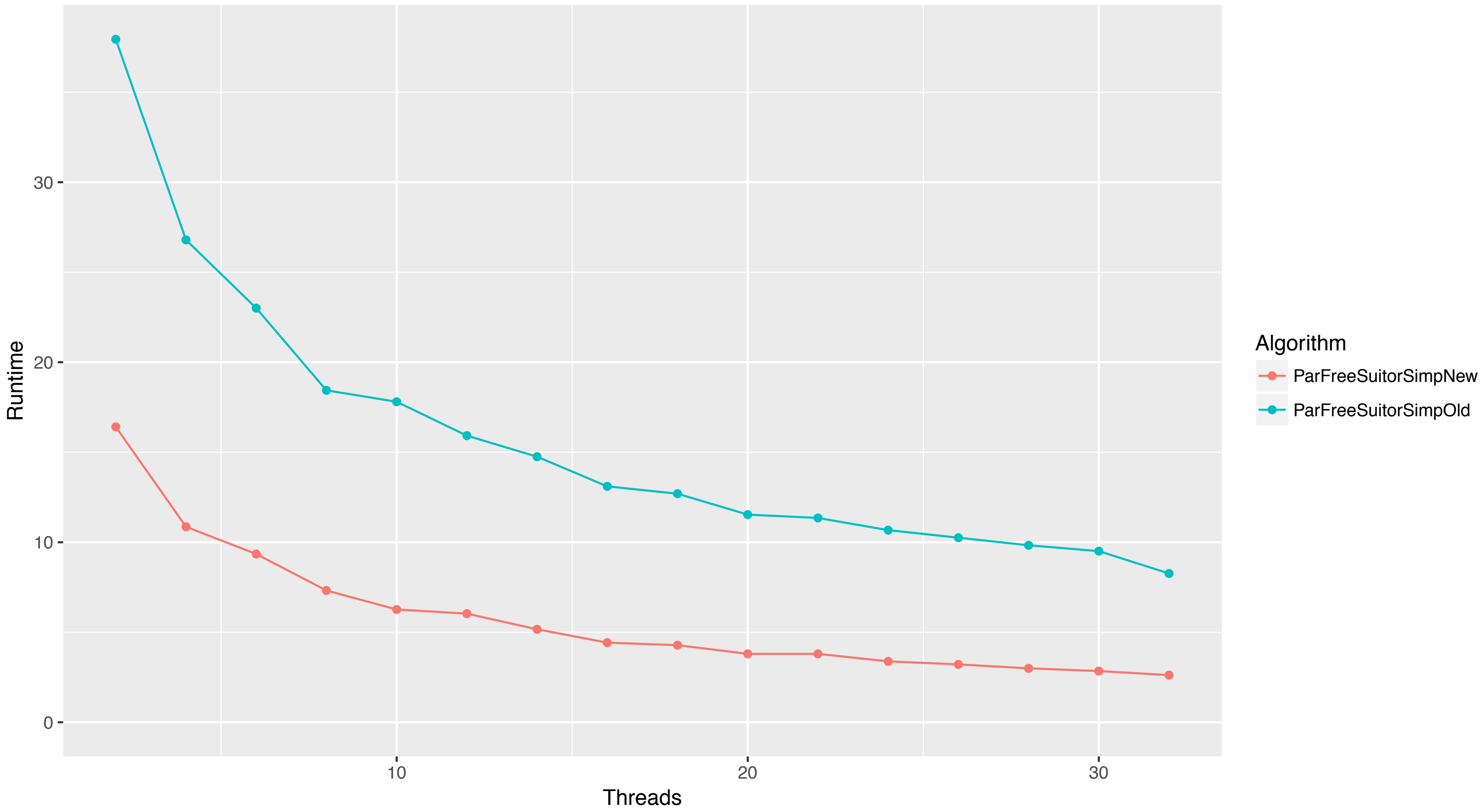
```
void ParallelLocklessSuitor::process(NodeHandle start, NodeHandle finish)
{
    for (NodeHandle u = start; u != finish; ++u)
    {
        NodeHandle current = u;
        bool done = false;

        while (!done)
        {
            EdgeIterator partner = G.null_edge();
            double heaviest = 0;

            for (EdgeIterator v = G.beginEdges(current); v != G.endEdges(current); ++v)
            {
                double weight_v = v->second;
                if (weight_v > heaviest && weight_v > suitor[v->first].load()->second)
                {
                    partner = v;
                    heaviest = weight_v;
                }
            }
            done = true;
            if (heaviest > 0)
            {
                size_t partnerId = partner->first;
                EdgeIterator current_suitor = suitor[partnerId];
                if (current_suitor->second < heaviest)
                {
                    if (suitor[partnerId].compare_exchange_strong(current_suitor, partner))
                    {
                        NodeHandle y = G.edgeHead(current_suitor);
                        if (y != G.null_node())
                        {
                            current = y;
                            done = false;
                        }
                    } else
                        done = false;
                } else
                    done = false;
            }
        }
    }
}
```

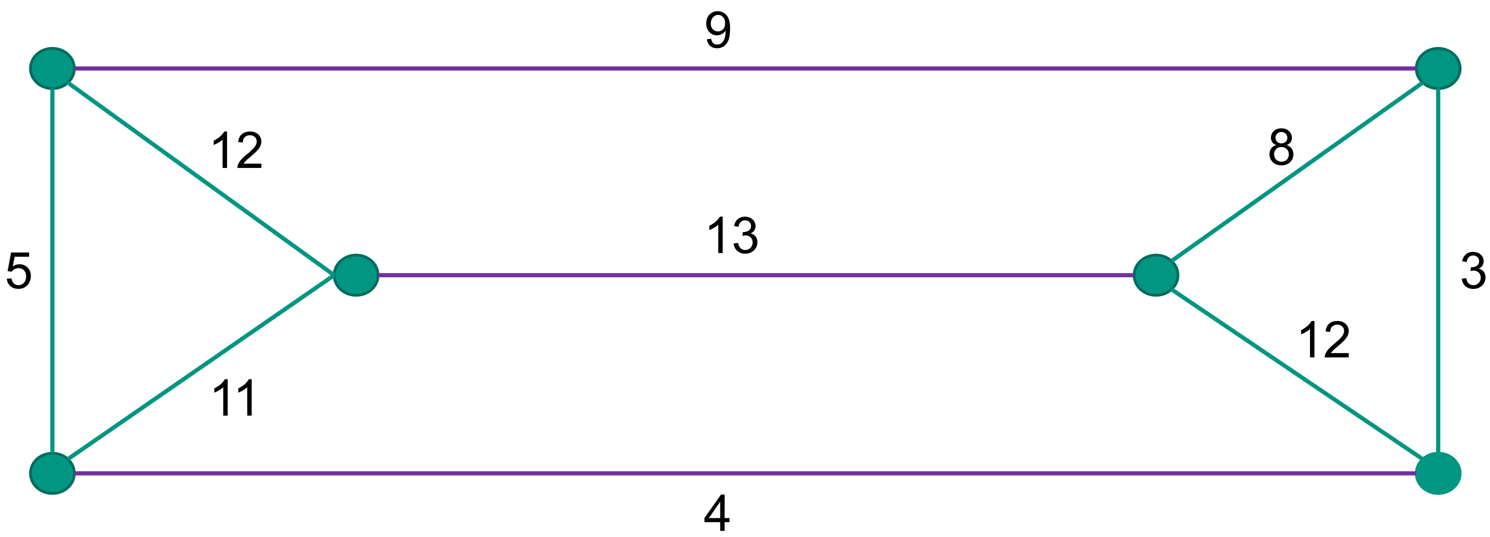
Suitor - Parallel

Runtime optimized vs unoptimized



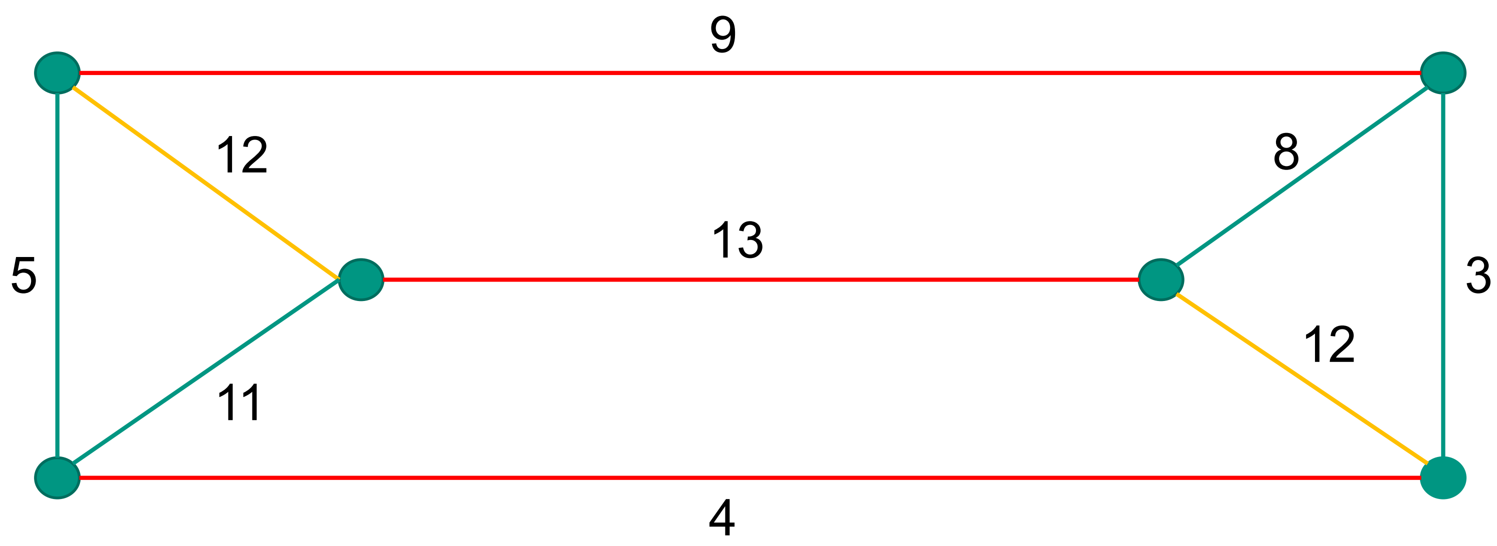
Quality Improvement – Heavy Matching

- 1st Matching



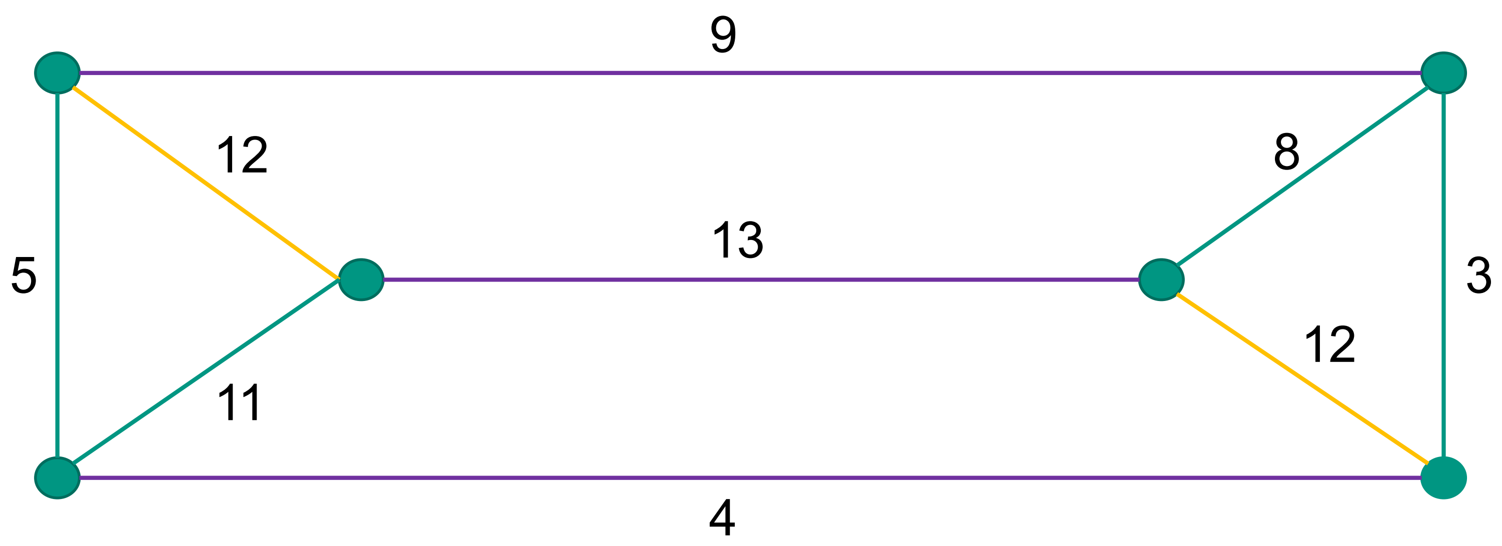
Quality Improvement – Heavy Matching

- 2nd Matching



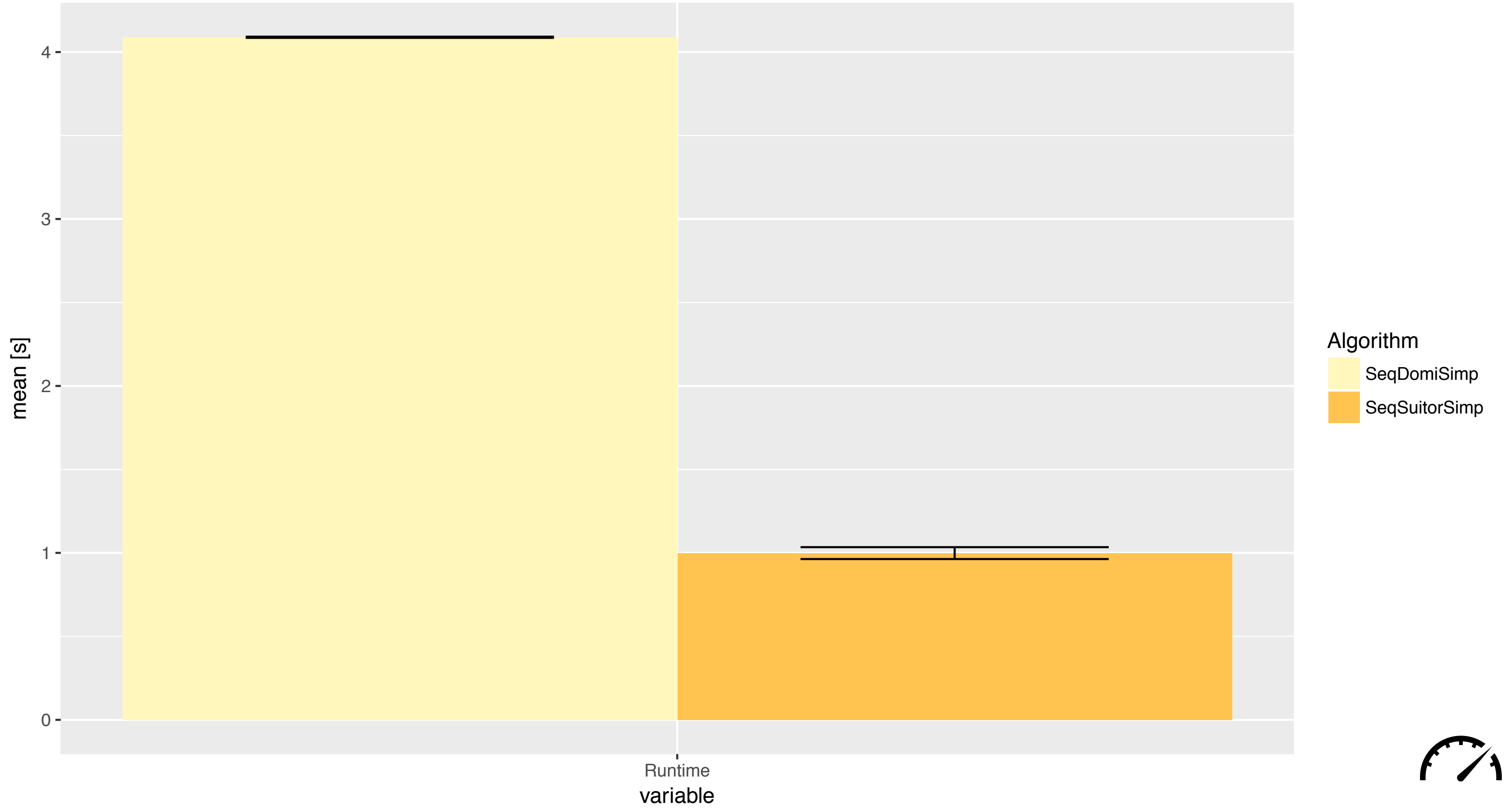
Quality Improvement – Heavy Matching

- Path or Cycle
- Can be paralleized



Evaluation

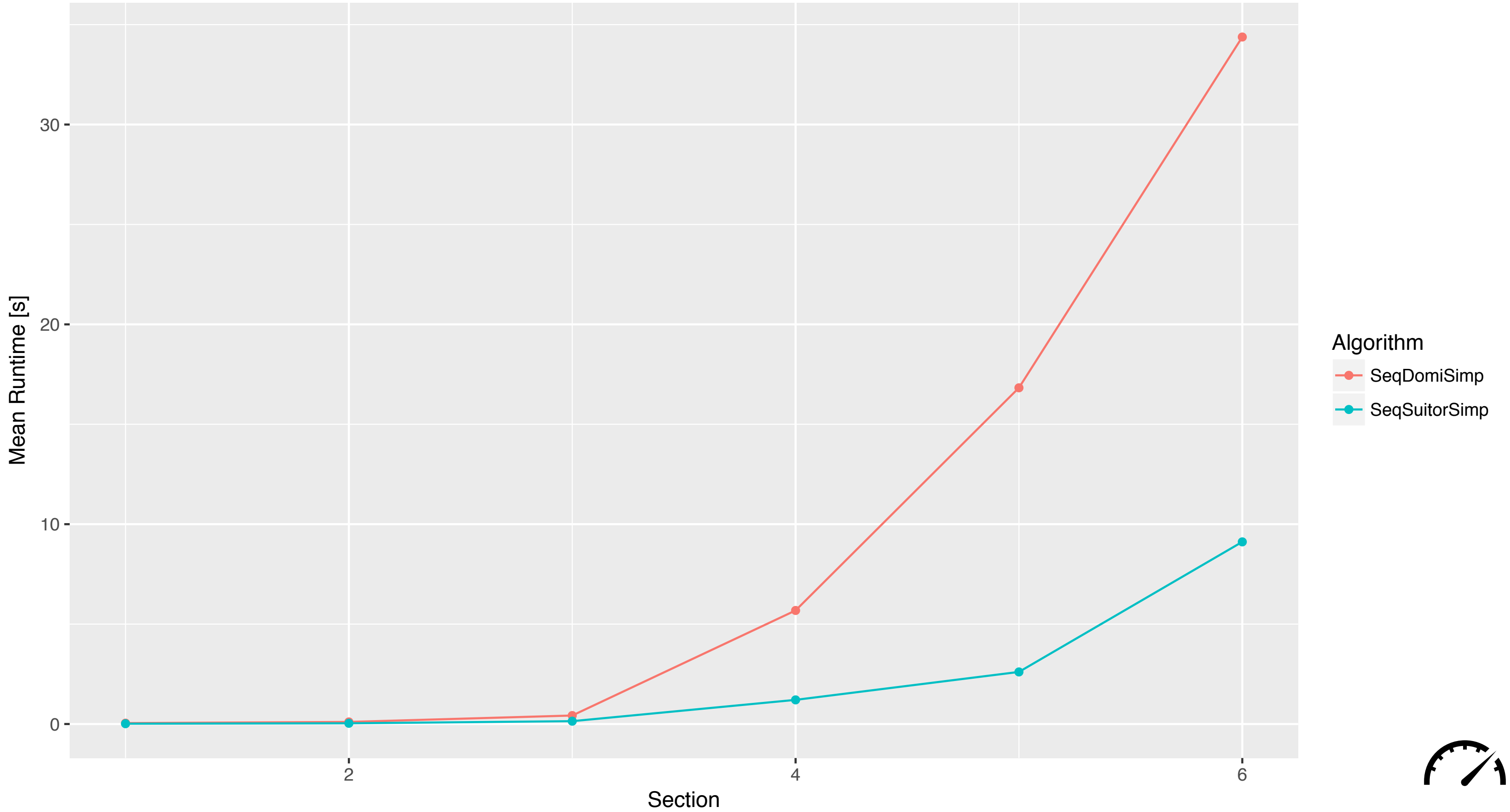
Runtime Sequential Implementation



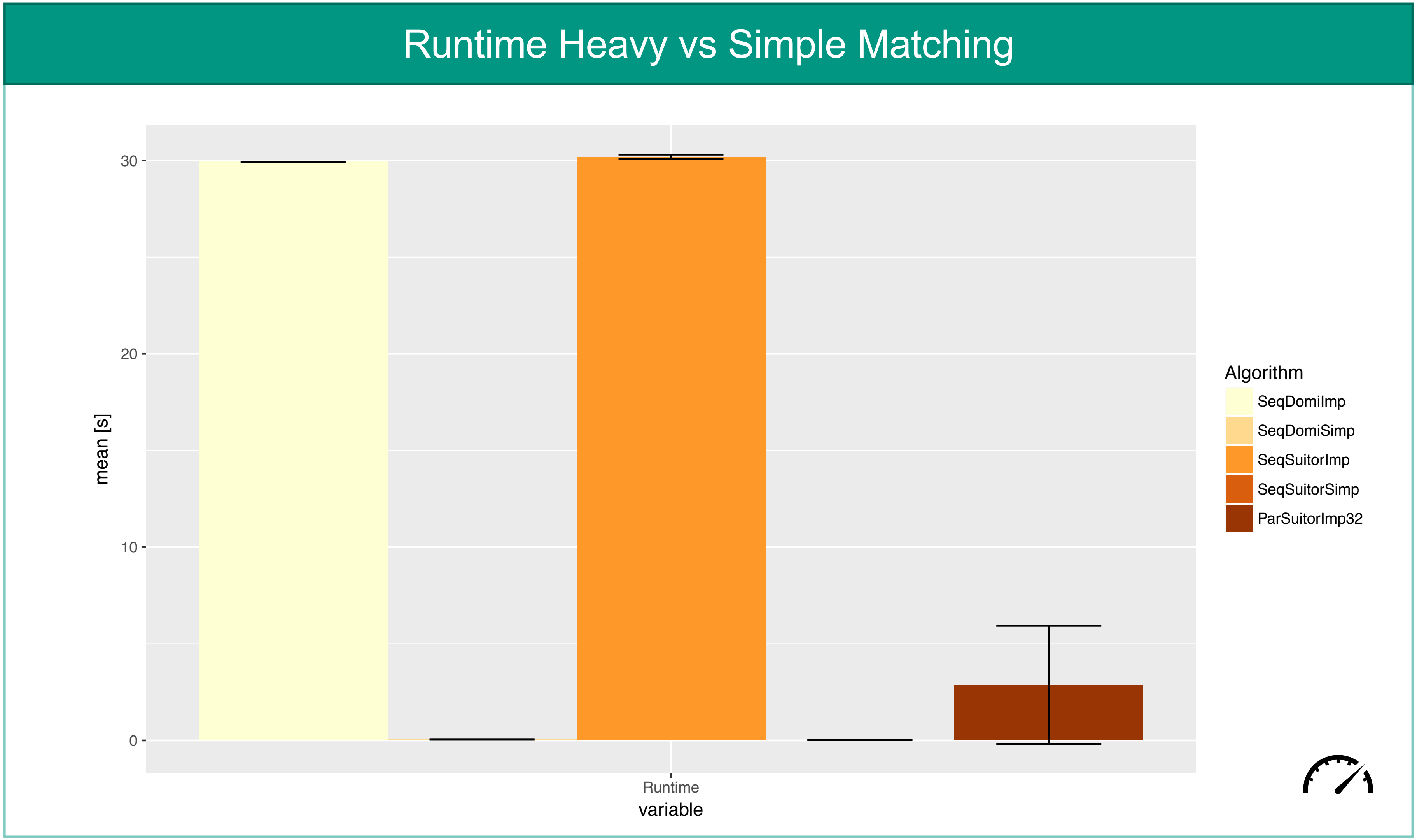
Evaluation

Karlsruhe Institute of Technology

Scalability Sequential Implementation

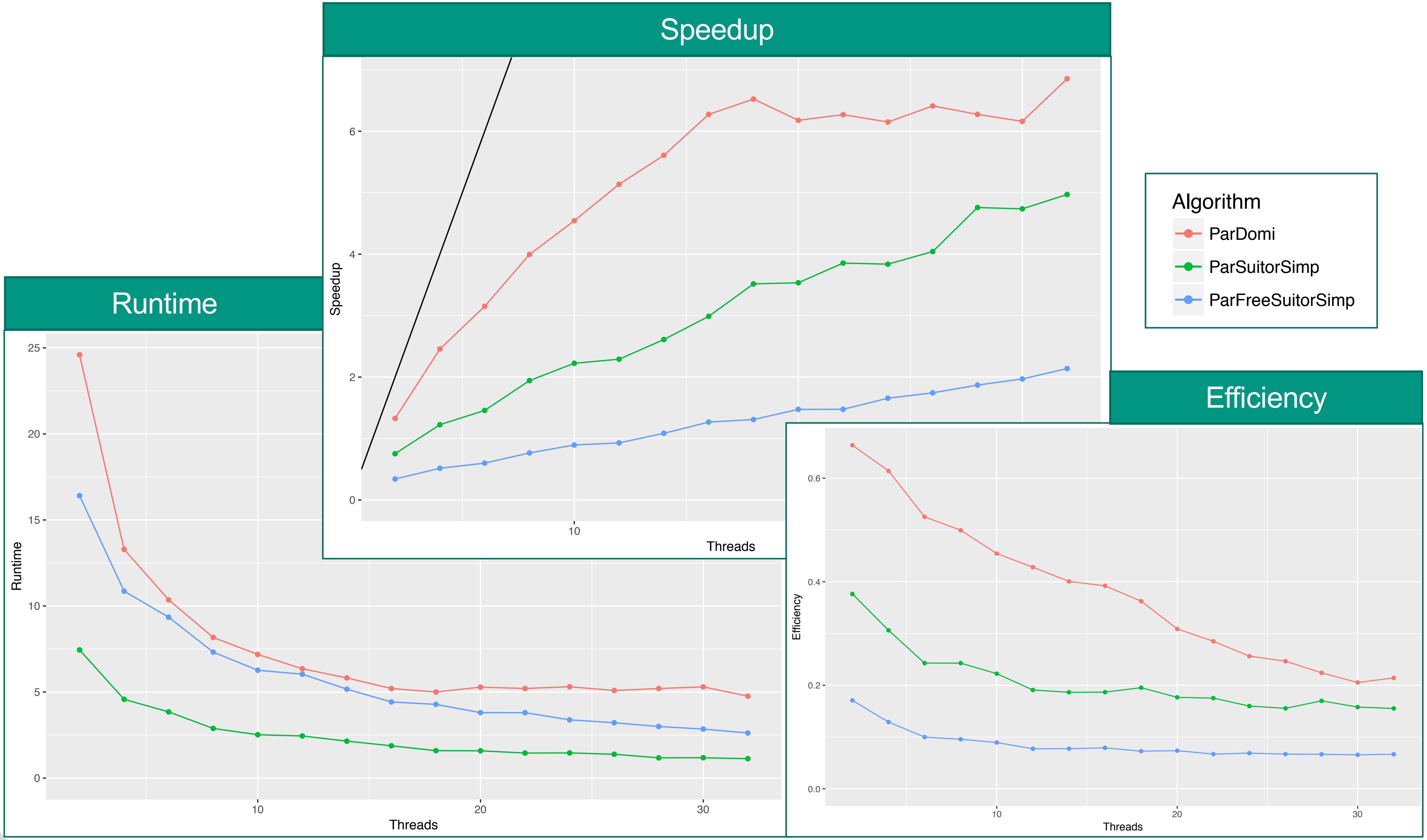


Evaluation



Runtime Heavy vs Simple Matching
30
20
10
0
mean [s]
Runtime
variable
Algorithm
SeqDomiImp
SeqDomiSimp
SeqSuitorImp
SeqSuitorSimp
ParSuitorImp32

Evaluation



Speedup
Runtime
Efficiency
Algorithm
ParDomi
ParSuitorSimp
ParFreeSuitorSimp
Speedup
Runtime
Efficiency
Threads
0
2
4
6
0
5
10
15
20
25
0.0
0.2
0.4
0.6
10
20
30

Evaluation


Quality Comparison
2e+06
1e+06
0e+00
Quality
Quality
variable
Algorithm
SeqDomiImp
SeqDomiSimp
SeqSuitorImp
SeqSuitorSimp
ParDomi
ParSuitorImp
ParSuitorSimp
ParFreeSuitorSimp

Summary

Karlsruhe Institute of Technology

Matching Problem

Algorithms

- Local Dominant
- Suitor
- Heavy Matching

Evaluation

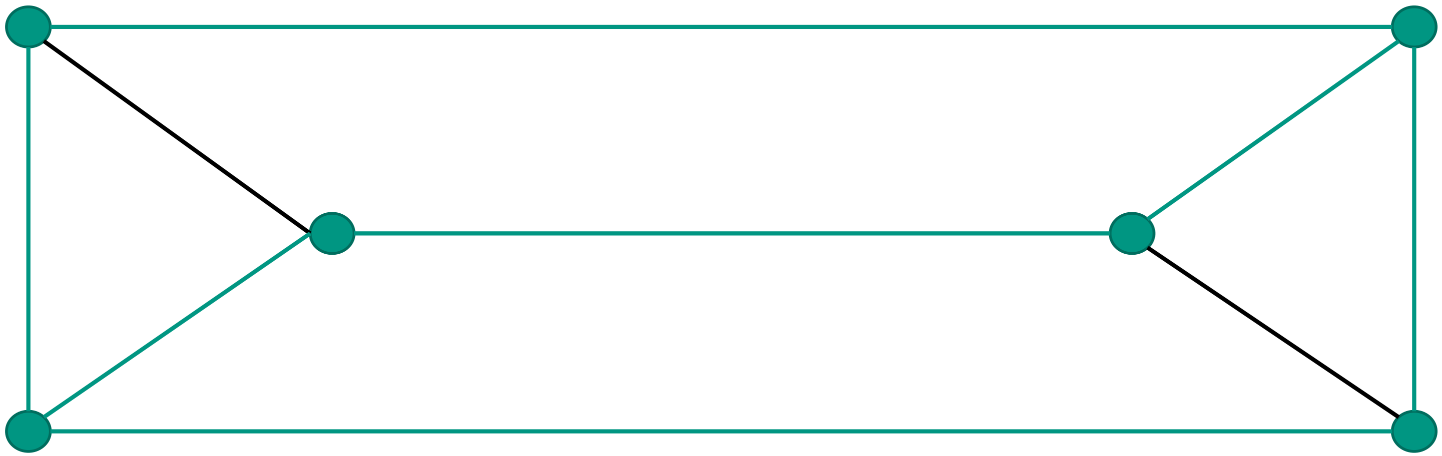
References

- J. Maue and P. Sanders, “Engineering algorithms for approximate weighted matching,” in *WEA’07*. LNCS, Springer, 2007, pp. 242–255.
- M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, “Approximate weighted matching on emerging manycore and multithreaded architectures,” *Int. J. High Perf. Comput. App.*, vol. 26, no. 4, pp. 413– 430, 2012.
- Manne, F., & Halappanavar, M. (2014, May). New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*(pp. 519-528). IEEE.

Backup

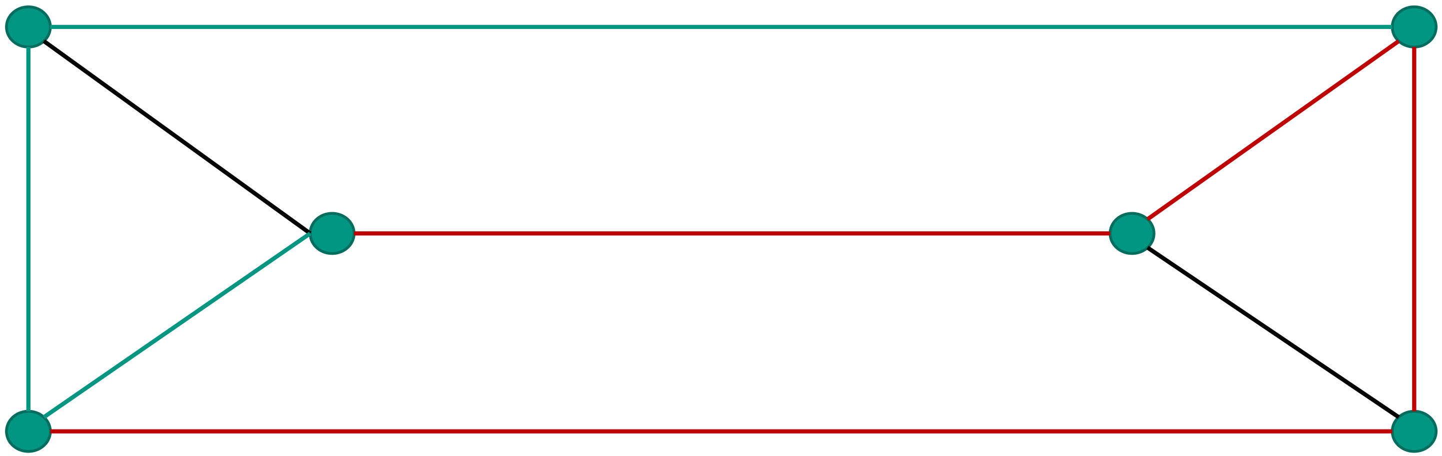
Definition – Maximal Matching

A **maximal matching** is a matching M of an undirected graph $G = (V, E)$ with the property that if any edge not in M is added to M, it is no longer a matching



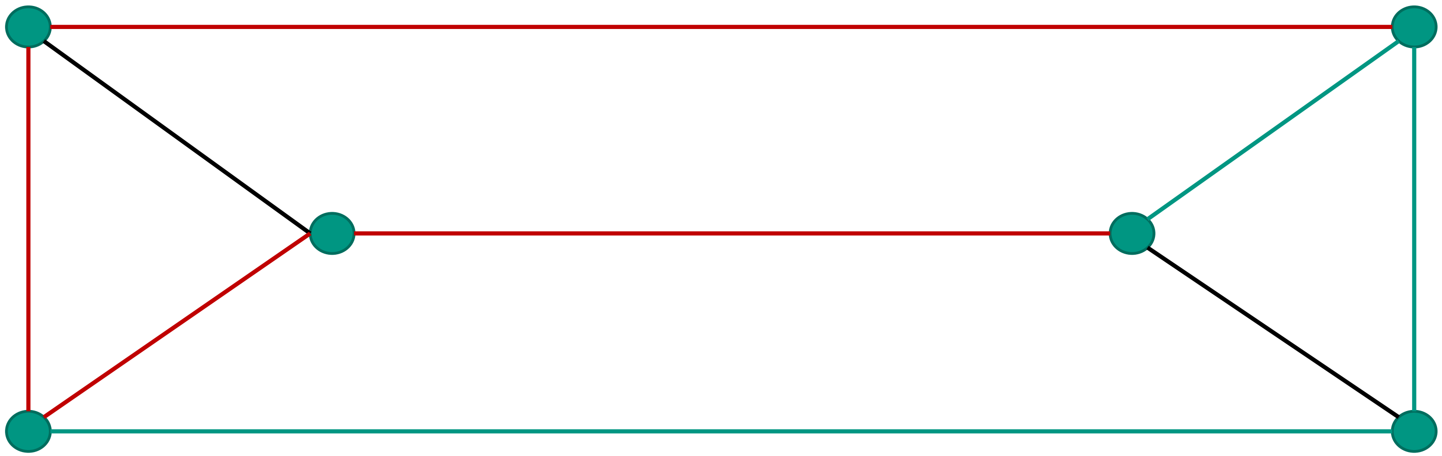
Definition – Maximal Matching

A **maximal matching** is a matching M of an undirected graph $G = (V, E)$ with the property that if any edge not in M is added to M, it is no longer a matching



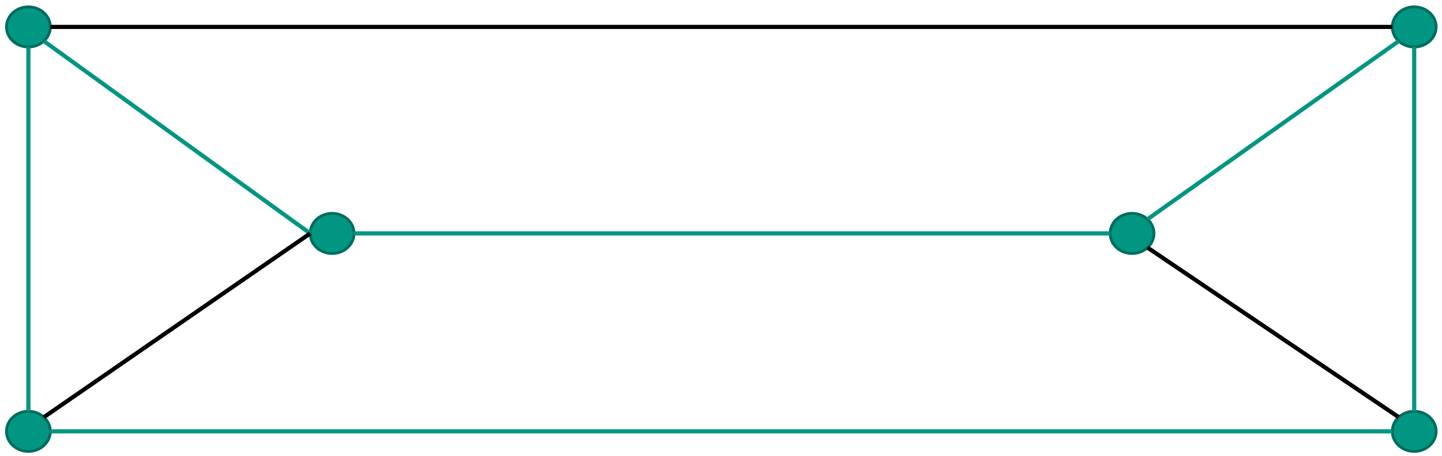
Definition – Maximal Matching

A **maximal matching** is a matching M of an undirected graph $G = (V, E)$ with the property that if any edge not in M is added to M, it is no longer a matching



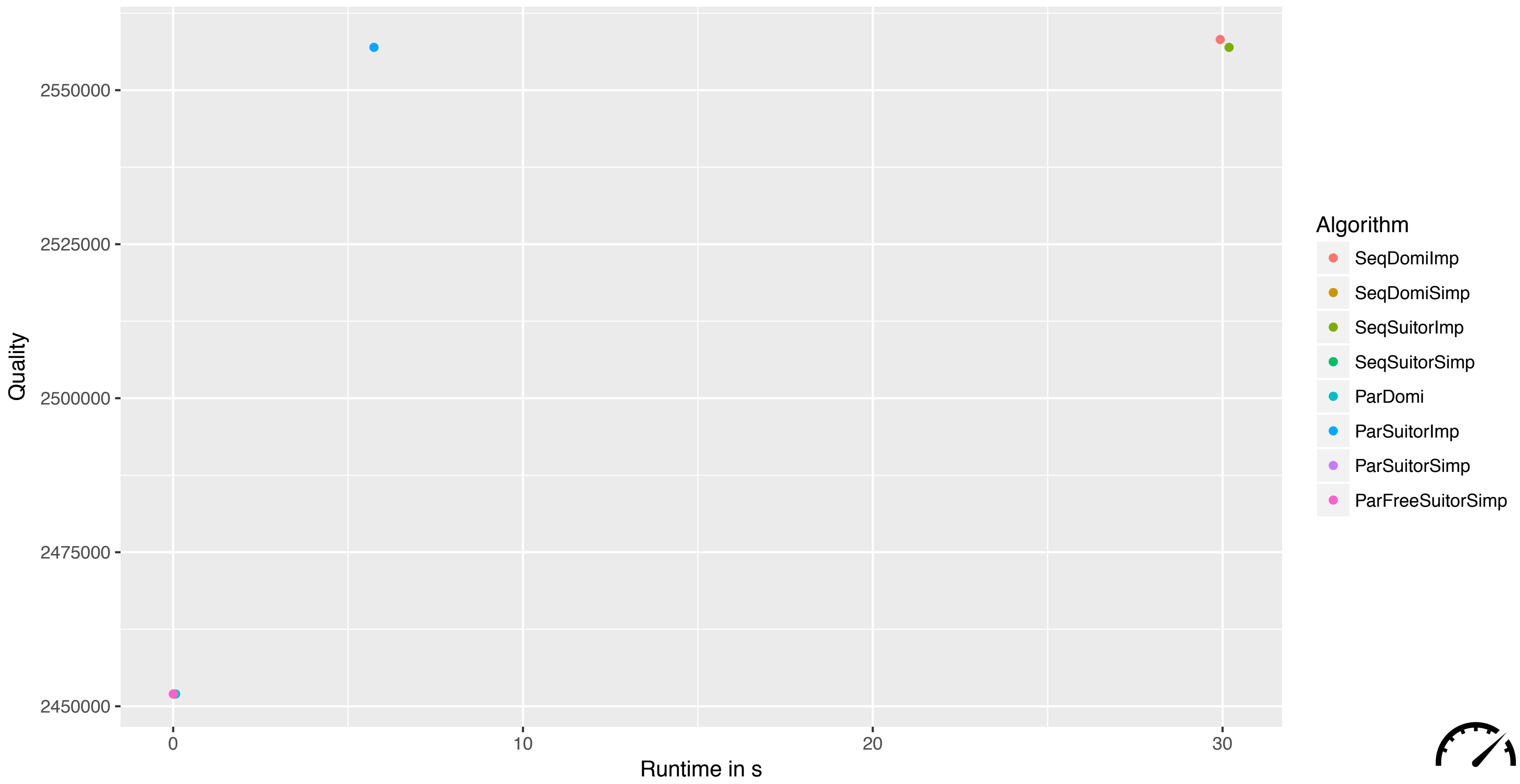
Definition – Maximum Cardinality Matching

A **maximum cardinality matching** of an undirected graph $G = (V, E)$ is a matching M that contains the largest possible number of elements



Backup - Evaluation

Quality vs Time



*Parallel with 32 threads

Backup – Complexities

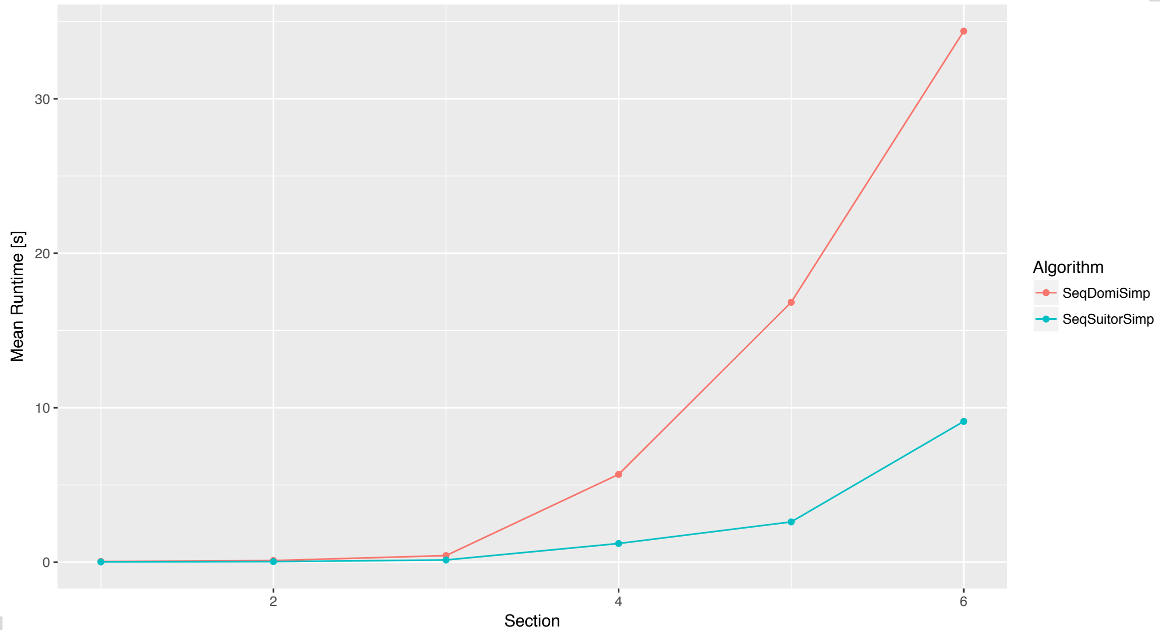
- Greedy: $O(m + m \log n)$
- PGA': $O(m)$
- Local Dominant: $O(n + m * \Delta)$
- Suitor: $O(m * \Delta)$
- Gabow: $O(nm + n^2 \log n)$

Backup – Graphs

- Runtime Sequential Implementation
  - nVertices 1000000
  - nEdges 100000000
- Runtime Heavy vs Simple Matching & Quality Comparison
  - nVertices 100000
  - nEdges 1000000
- Parallel Running Time
  - nVertices 6000000
  - nEdges 400000000

Backup - Graphs

- Scalability Sequential Implementation
  - Sec. 1: nVertices 100.000 nEdges 1.000.000
  - Sec. 2: nVertices 200000 nEdges 2.000.000
  - Sec. 3: nVertices 600.000 nEdges 6.000.000
  - Sec. 4: nVertices 1.000.000 nEdges 1.000.000.000
  - Sec. 5: nVertices 2.000.000 nEdges 2.000.000.000
  - Sec. 6: nVertices 4.000.000 nEdges 4.000.000.000



Backup – Sequential Local Dominant

```
procedure SERIAL-QUEUE (G(V, E), mate)
    for each u ∈ V do
        mate[u] ← ∅
        candidate[u] ← ∅
    Q ← ∅
    for each u ∈ V do
        PROCESS VERTEX(u, Q)
    while Q ≠ ∅ do
        u ← FRONT(Q)
        Q ← Q\{u}
        for each v ∈ adj(u) \ mate(u) do
            if candidate[v] = u then

                PROCESSVERTEX(v, Q)
```

```
procedure PROCESSVERTEX (s, Q)
    max_wt ← −∞
    max_wt_id ← ∅
    for each t ∈ adj(s) do
        if (mate[t] = ∅) AND
            (max_wt < w(e_{s,t})) then
            max_wt ← w(e_{s,t})
            max_wt_id ← t
    candidate[s] ← max_wt_id
    if candidate[candidate[s]] = s
        then
        mate[s] ← candidate[s]
        mate[candidate[s]] ← s
        Q ← Q ∪ {s,candidate[s]}
```

Source: Halappanavar et al.

Backup – Evaluation Suitor

	Fl. Small	*Fl. Large*	*Rand*	*RMAT*	*Geo.*	*Comp. Geo*	*Comp. Rand*
GREEDY	56.1	56.7	31.4	39.2	43.8	108.7	105.8
GREEDY-	1.0	1.5	0.8	1.3	1.8	1.7	1.3
PGA′	3.2	4.5	4.2	5.5	4.9	2.7	2.7
LOCALMAX	8.2	5.5	5.6	9.9	4.0	1166.0	5.8
SUITOR	2.5	2.8	3.1	4.1	3.1	195.6	2.4
SORTSUITOR	42.7	31.7	15.9	16.8	16.9	118.1	116.1
SORTSUITOR-	1.3	1.5	1.7	2.4	1.9	0.9	0.01

TABLE I
TIME TAKEN RELATIVE TO THE HEM ALGORITHM.

	Fl. Small	*Fl. Large*	*Rand*	*RMAT*	*Geo*	*Comp. Geo*	*Comp. Rand*
GPA	22.8	21.9	12.3	11.3	15.7	55.7	54.2
GPA-	0.9	2.3	1.4	1.8	3.5	1.1	0.8
2RPGA’	3.2	3.5	3.1	3.2	3.5	3.1	3.1
2RSUITOR	2.1	2.0	2.6	2.6	2.2	199.2	2.5
2RSORTSUITOR	18.0	13.0	6.6	5.4	5.9	60.7	59.3
2RSORTSUITOR-	1.2	1.6	1.4	1.8	1.5	0.9	0.01

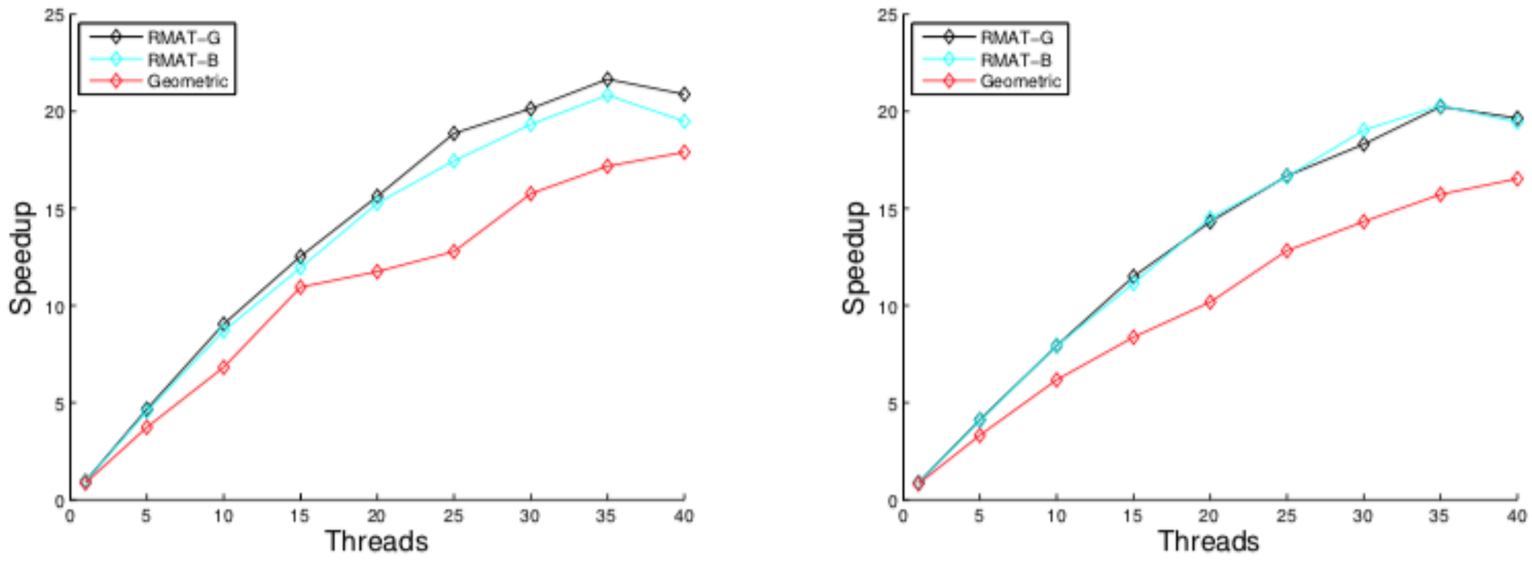
TABLE II
TIME TAKEN RELATIVE TO THE 2RHEM ALGORITHM.

	Fl. Small	*Fl. Large*	*Rand*	*RMAT*	*Geo*	*Comp. Geo*	*Comp. Rand*
2RSUITOR	0.9986	0.9977	0.9983	1.0027	0.9989	1.0000	0.9995
2RPGA’	0.9972	0.9966	0.9960	0.9983	0.9969	0.9994	0.9997
2RHEM	0.8172	0.7795	0.9762	0.9675	1.0007	0.9959	0.9995
GREEDY	0.9921	0.9790	0.9761	0.9626	0.9749	0.9999	0.9992

TABLE III
RELATIVE WEIGHT OF TWO-ROUND ALGORITHMS COMPARED TO GPA.

Source: Manne and Halappanavar

Backup – Evaluation Suitor



(c) SUITOR on *RMAT* and *Geometric* graphs

(d) 2RSUITOR on *RMAT* and *Geometric* graphs

Source: Manne and Halappanavar

Suitor – Parallel Lockfree

Lock

```
double weight_v = G.edgeWeight(v);
if (weight_v > heaviest && weight_v > ws[*v])
```

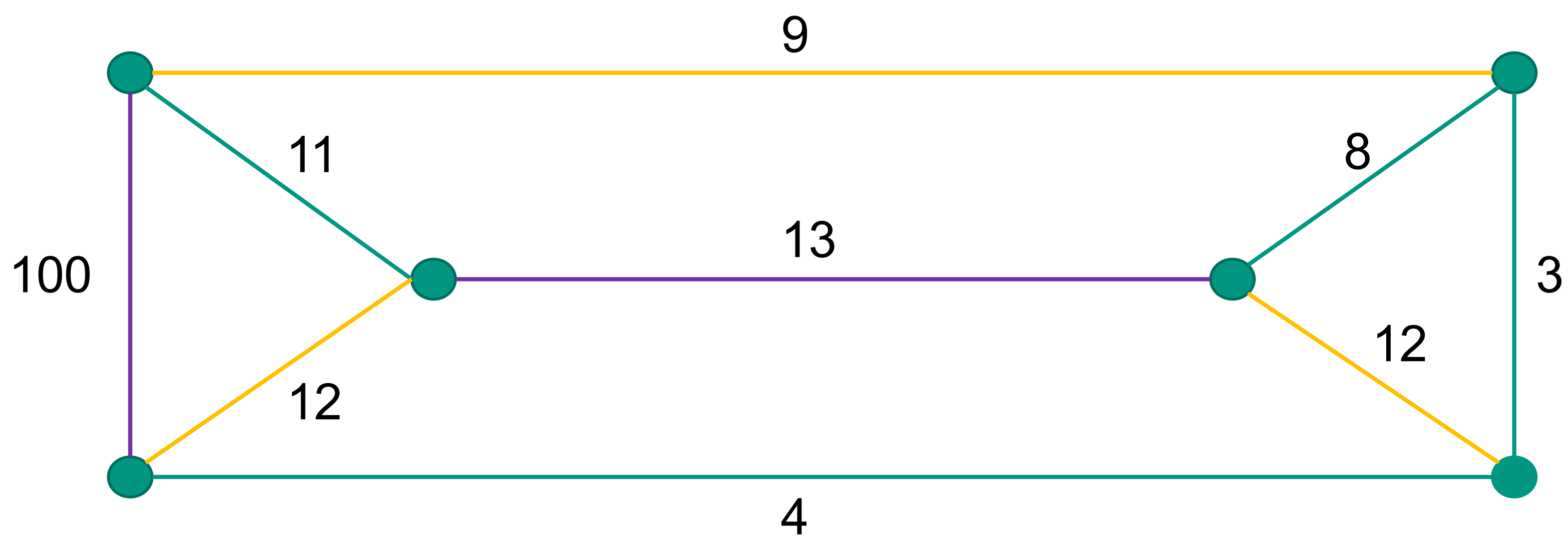
Lockfree - unoptimized

```
double weight_v = G.edgeWeight(v);
if (weight_v > heaviest && weight_v > G.edgeWeight(suitor[*v]))
```

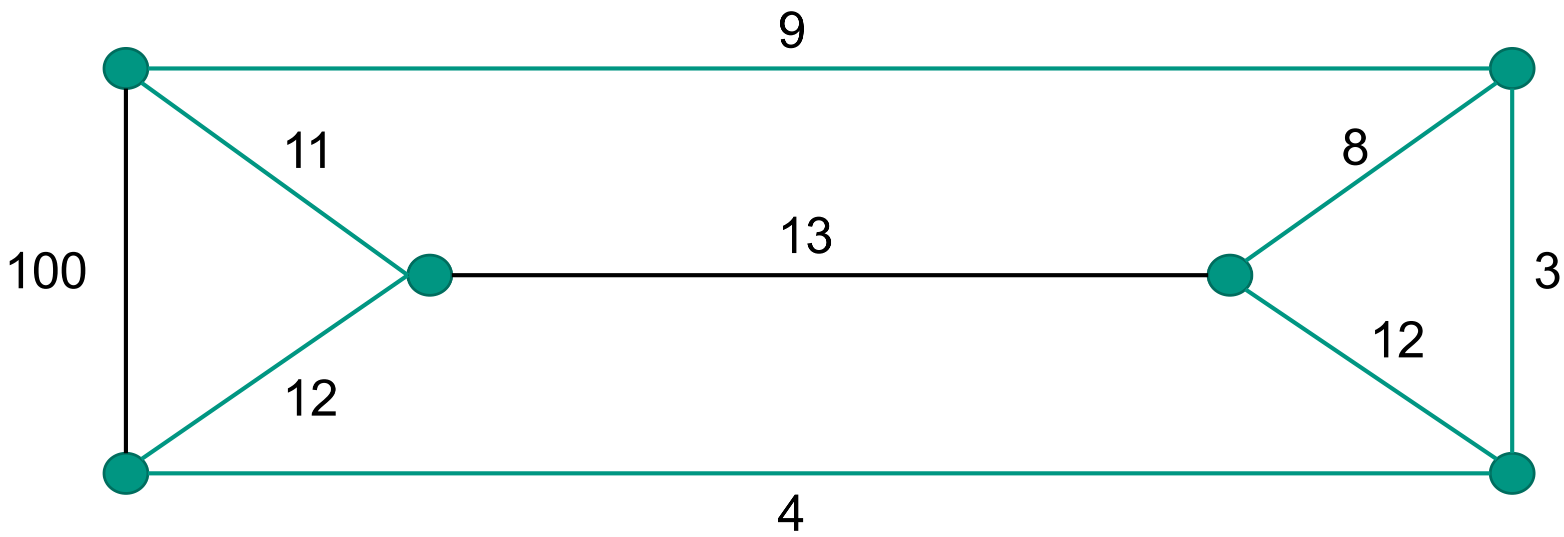
Lockfree - optimized

```
double weight_v = v->second;
if (weight_v > heaviest && weight_v > suitor[v->first].load()->second)
```

Quality Improvement – Heavy Matching

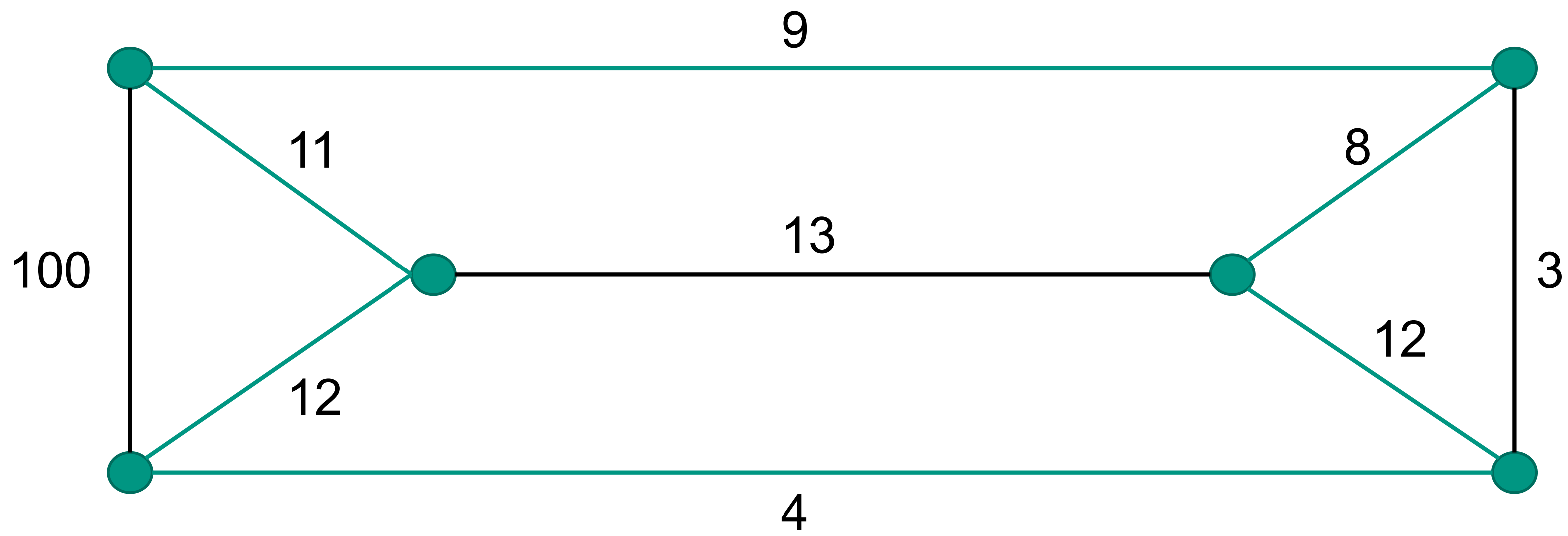


Quality Improvement – Heavy Matching

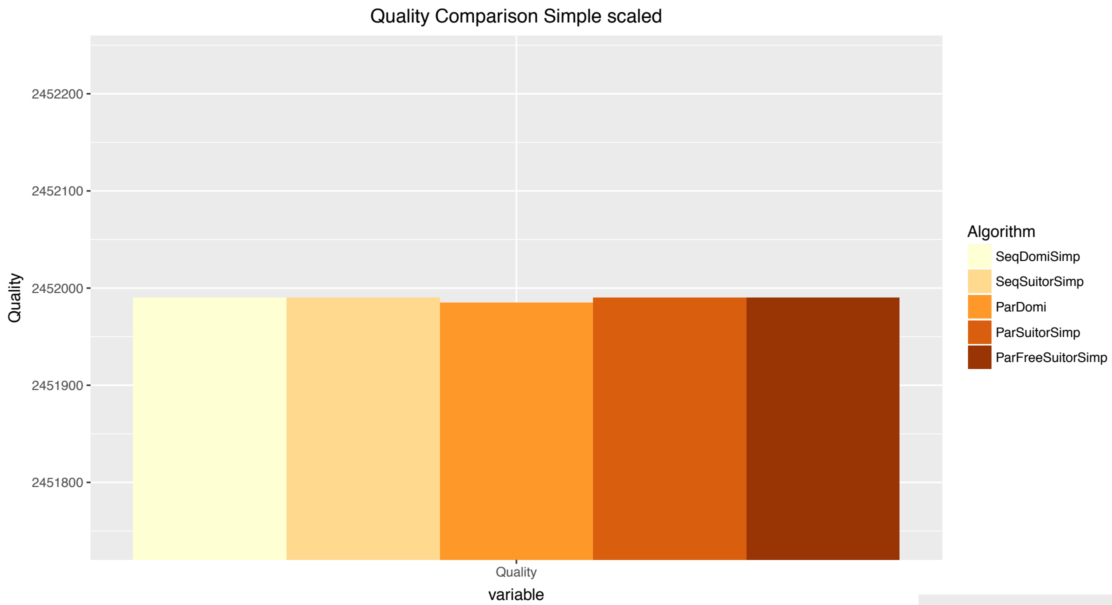


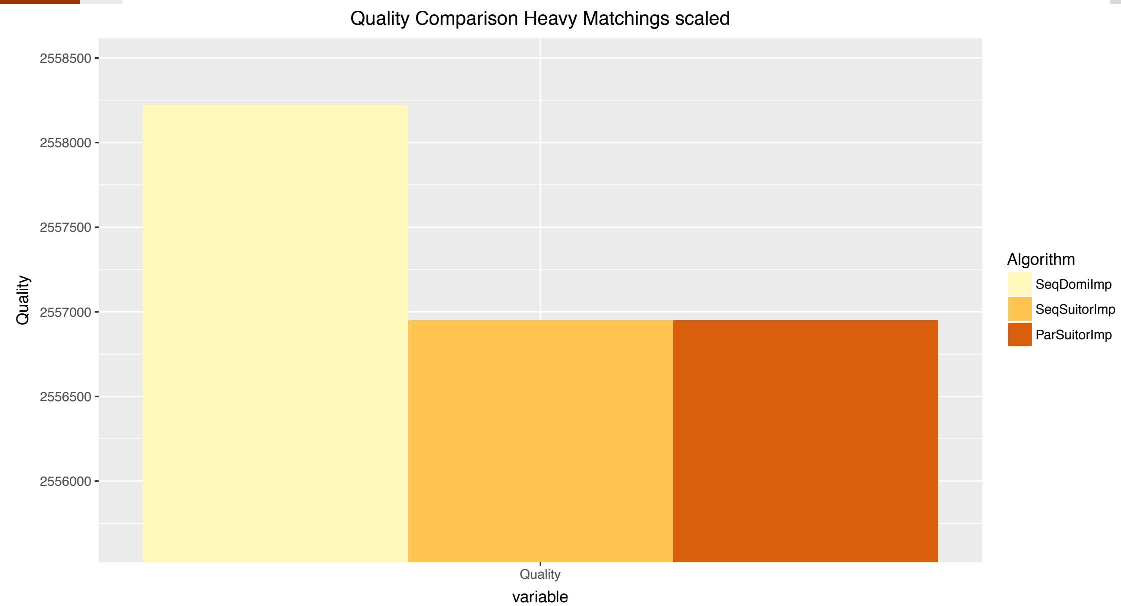
Quality Improvement – Heavy Matching

- Extend to maximal matching

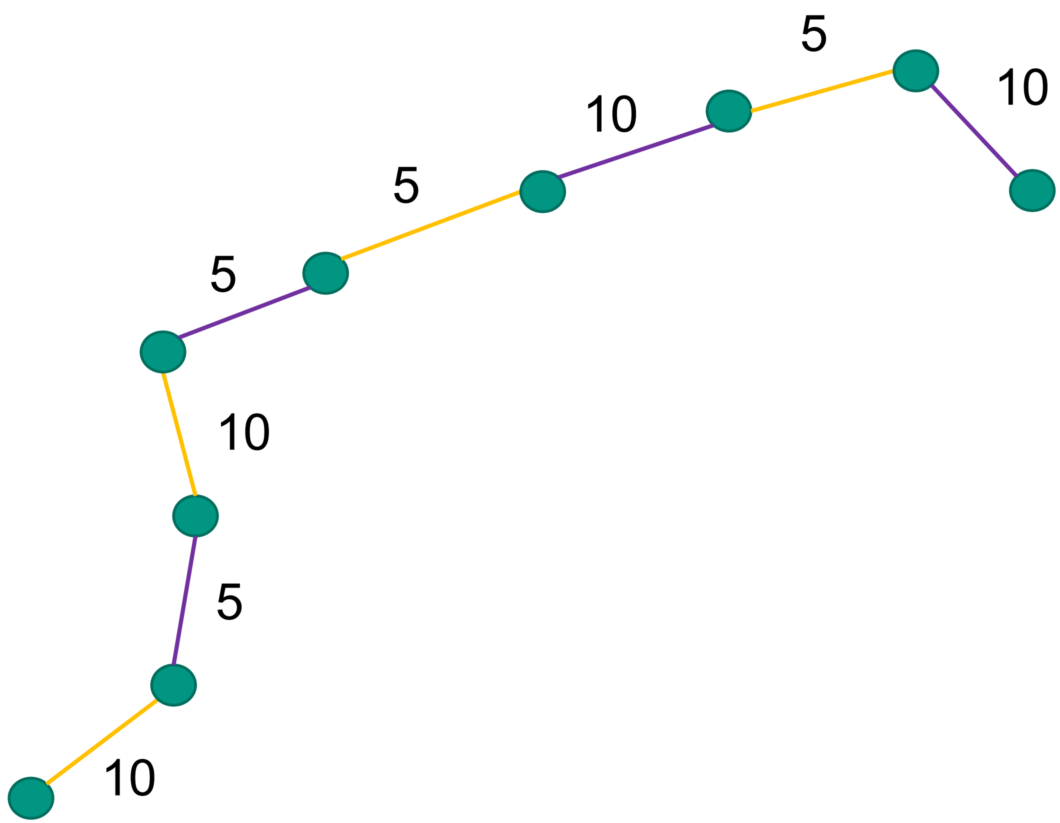


Backup – Quality Scaled



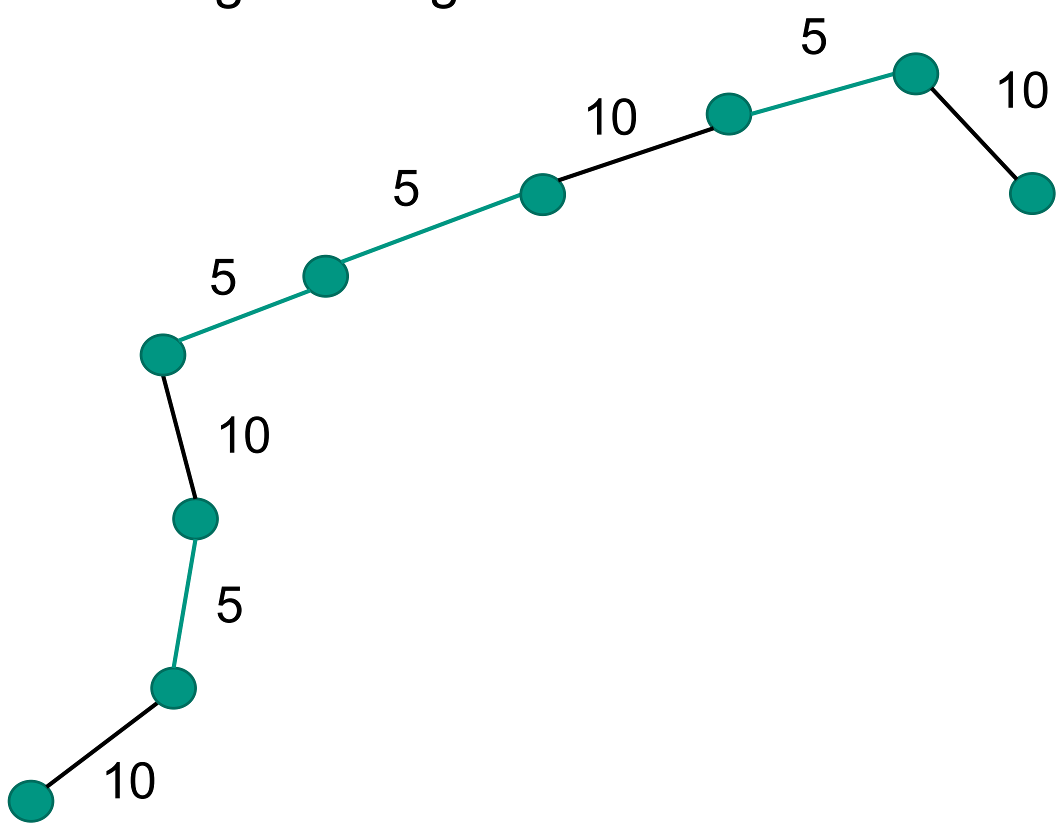


Quality Improvement – Heavy Matching



Quality Improvement – Heavy Matching

- Dynamic Programming



Backup – Greedy Algorithm

GRDY$(\boldsymbol{G = (V, E), w: E \to \mathbb{R}^{\geq 0}})$

$M := \emptyset$
while $E \neq \emptyset$ **do**
  let e be the edge with biggest weight in E
  add e to M
  remove e and all edges adjacent to its endpoints from E
return M

Fig. 1. The greedy algorithm for approximate weighted matchings.

Source: Maue and Sanders

Backup – PGA’ Algorithm

PGA′$(G = (V, E),\ w : E \to \mathbb{R}^{\geq 0})$

```
M := ∅
while E ≠ ∅ do
    P := ⟨⟩
    arbitrarily choose v ∈ V with deg(v) > 0
    while deg(v) > 0 do
        let e = (v, u) be the heaviest edge adjacent to v
        append e to P
        remove v and its adjacent edges from G
        v := u
    M := M ∪ MaxWeightMatching(P)
extend M to a maximal matching
return M
```

Fig. 2. The improved Path Growing Algorithm PGA′.

Source: Maue and Sanders

Backup – Dynamic Programming

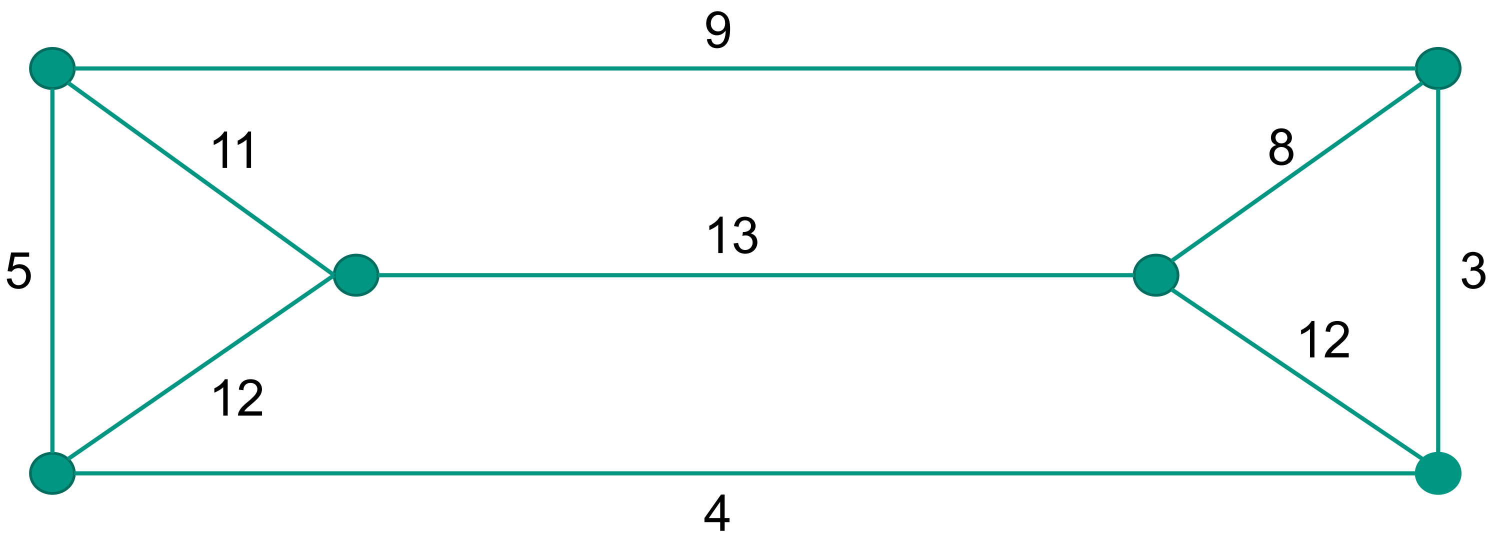
MaxWeightMatching$(P = \langle e_1, \ldots, e_k \rangle)$

$W[0] := 0; \quad W[1] := w(e_1)$
$M[0] := \emptyset; \quad M[1] := \{e_1\}$
for $i := 2$ **to** k **do**
  if $w(e_i) + W[i-2] > W[i-1]$ **then**
    $W[i] := w(e_i) + W[i-2]$
    $M[i] := M[i-2] \cup \{e_i\}$
  else
    $W[i] := W[i-1]$
    $M[i] := M[i-1]$
return $M[k]$

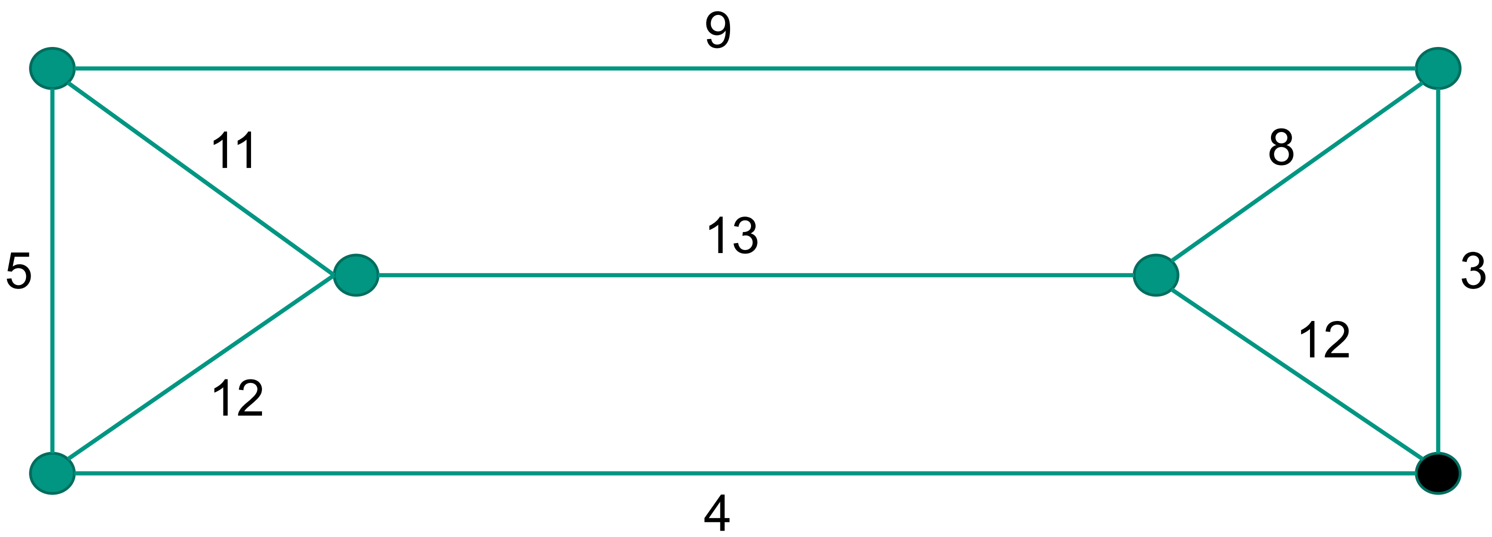
Fig. 7. Obtaining a maximum weight matching for a path by dynamic programming.

Source: Maue and Sanders

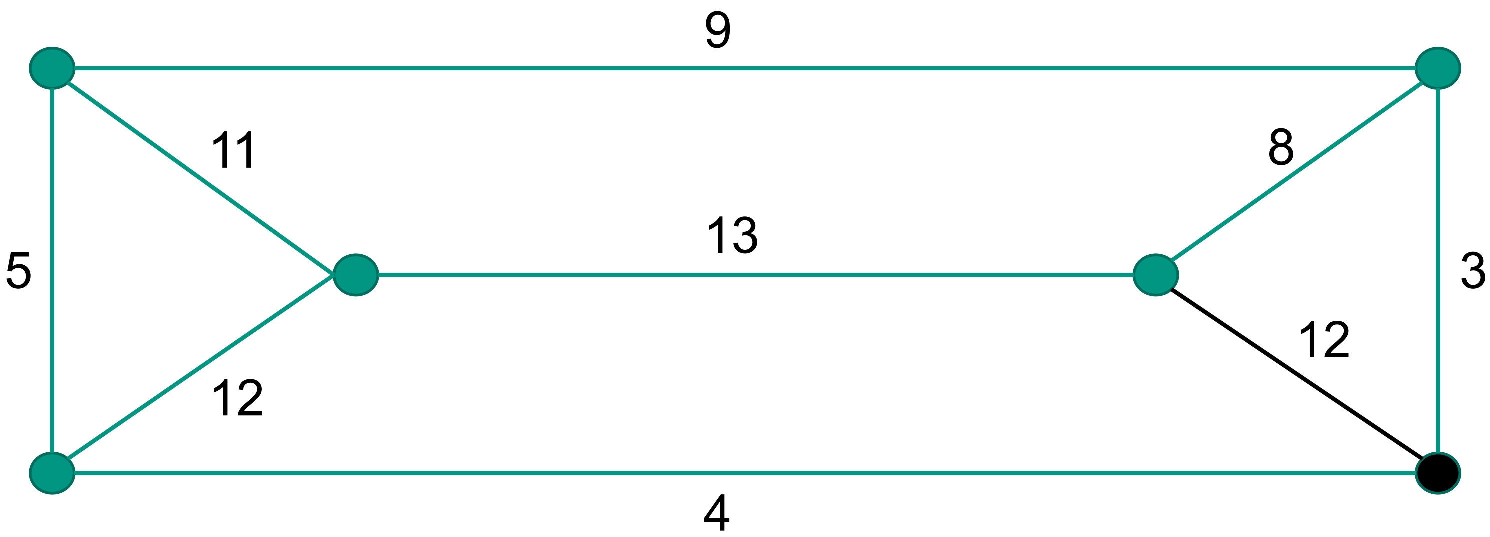
PGA



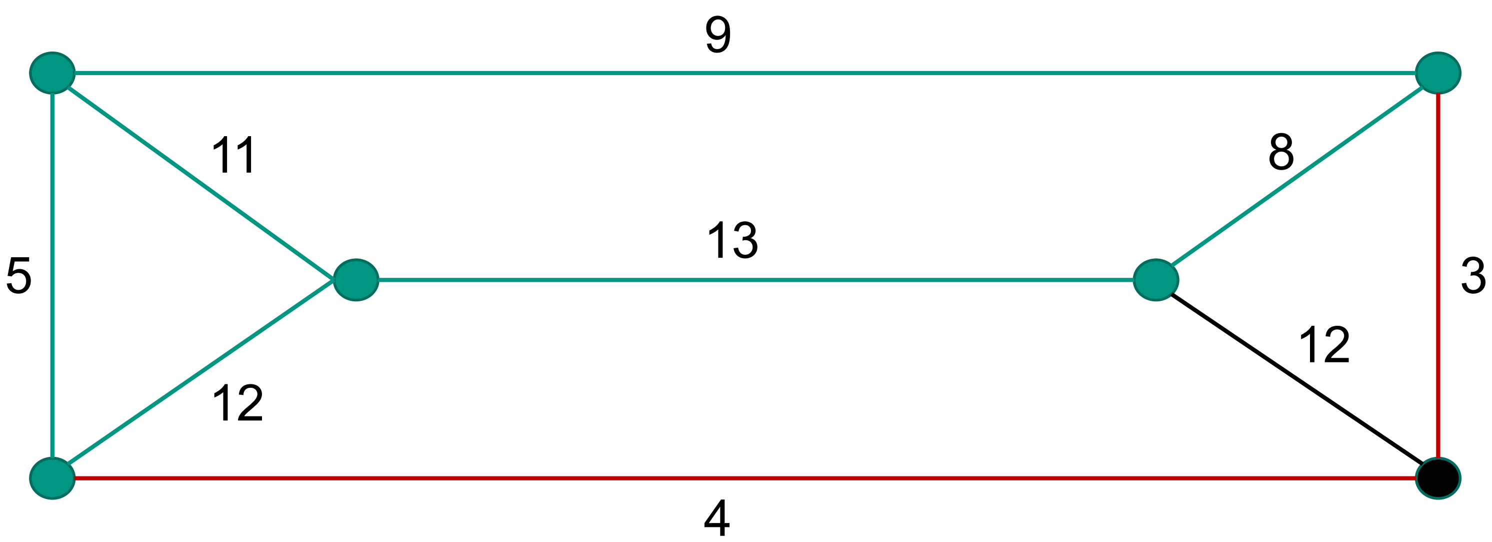
PGA



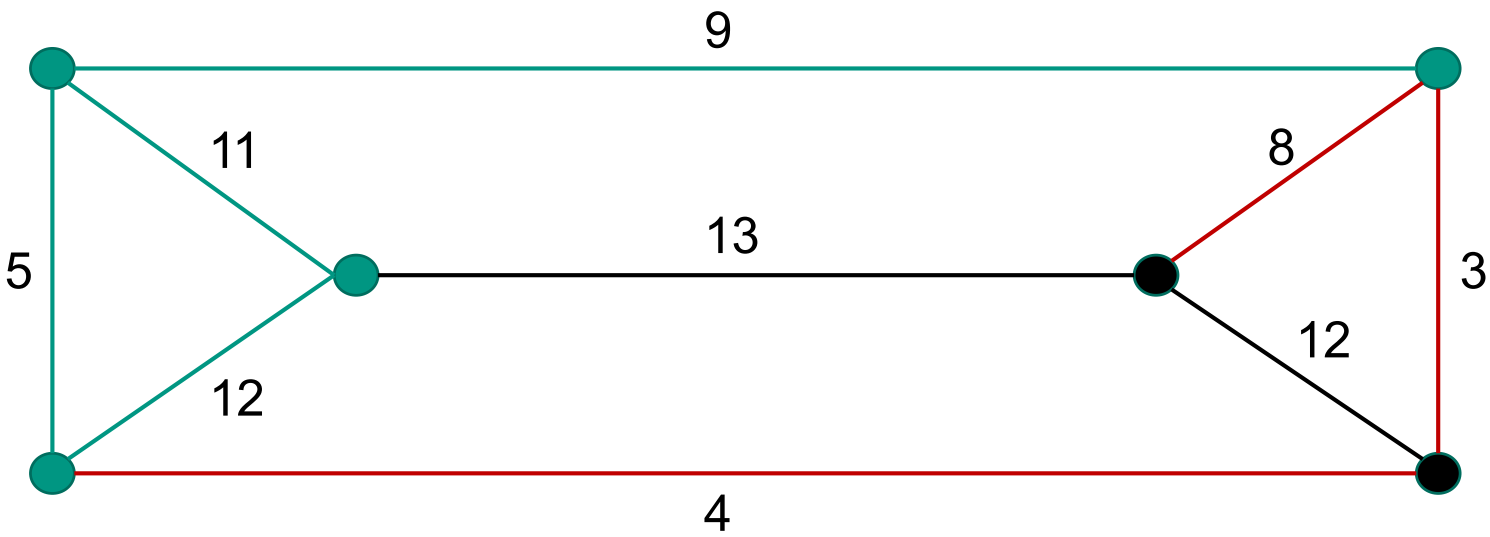
PGA



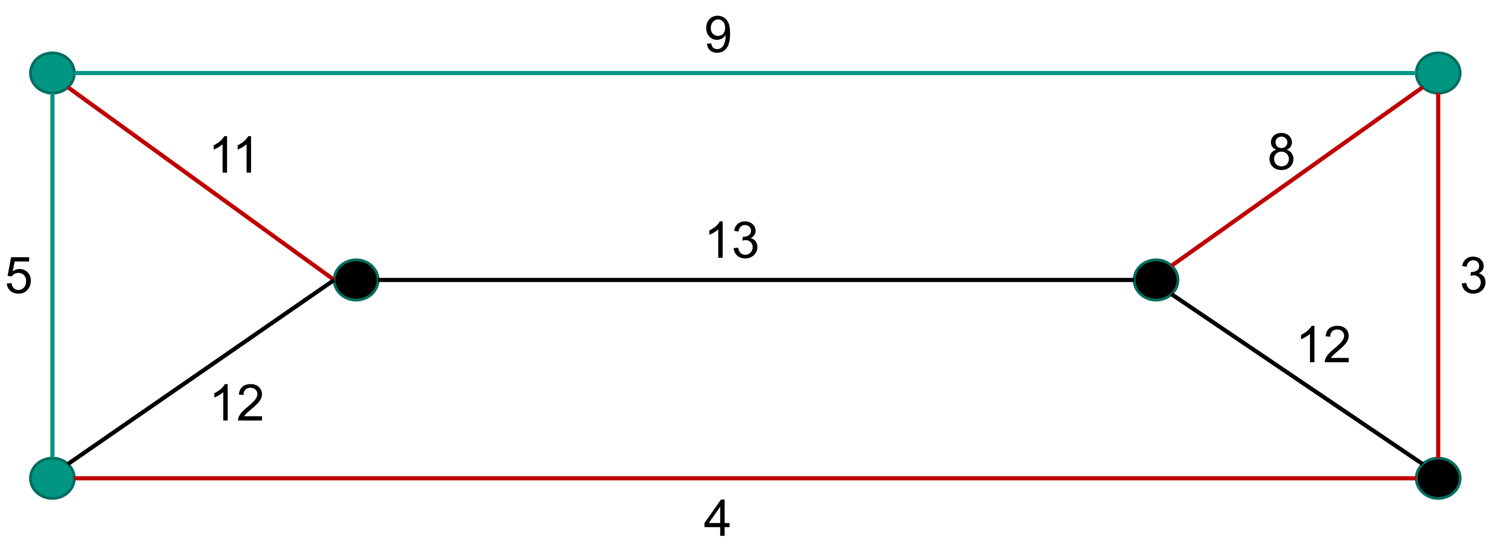
PGA



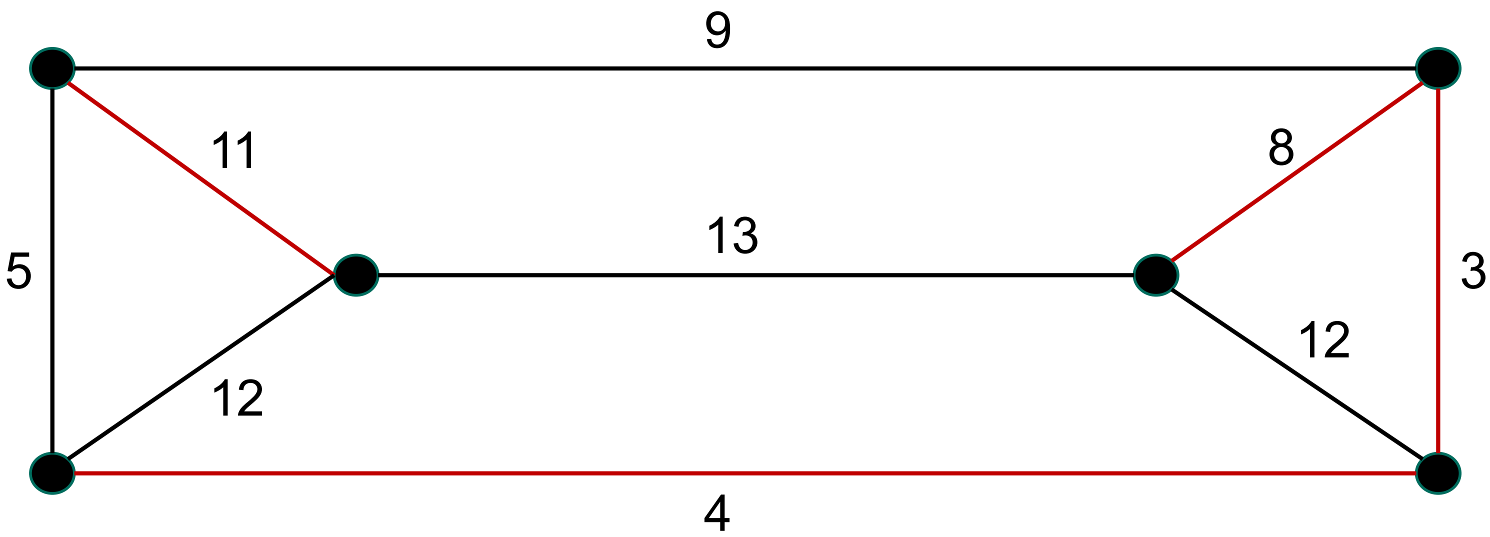
PGA



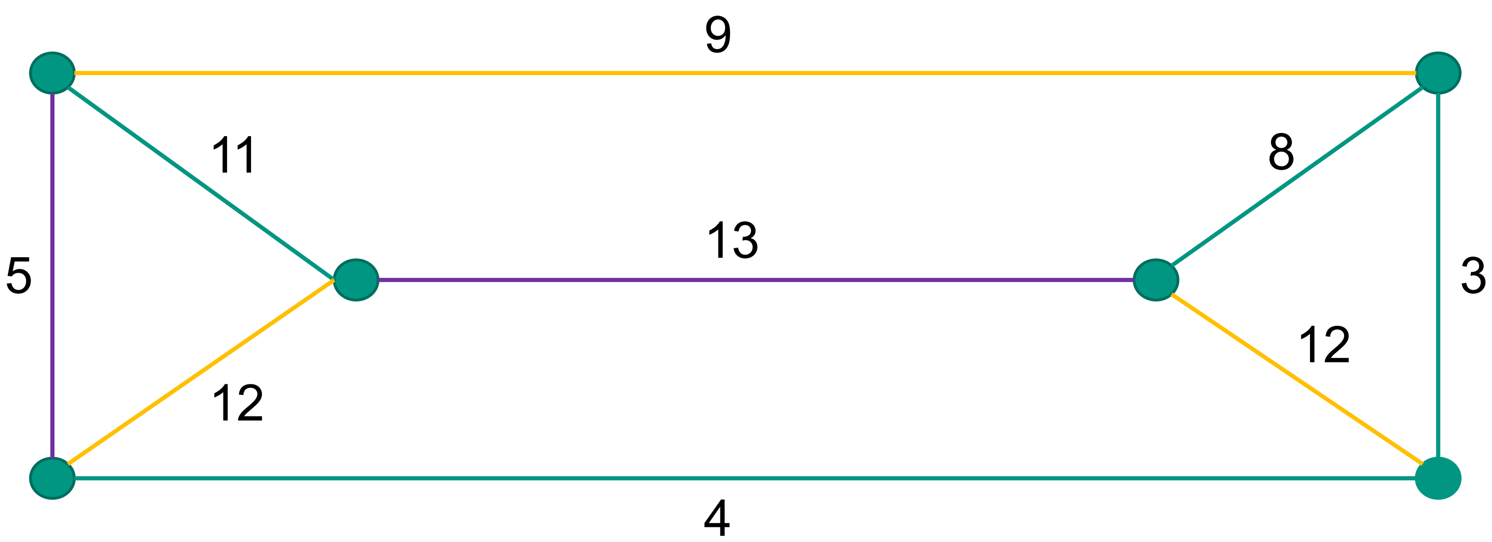
PGA



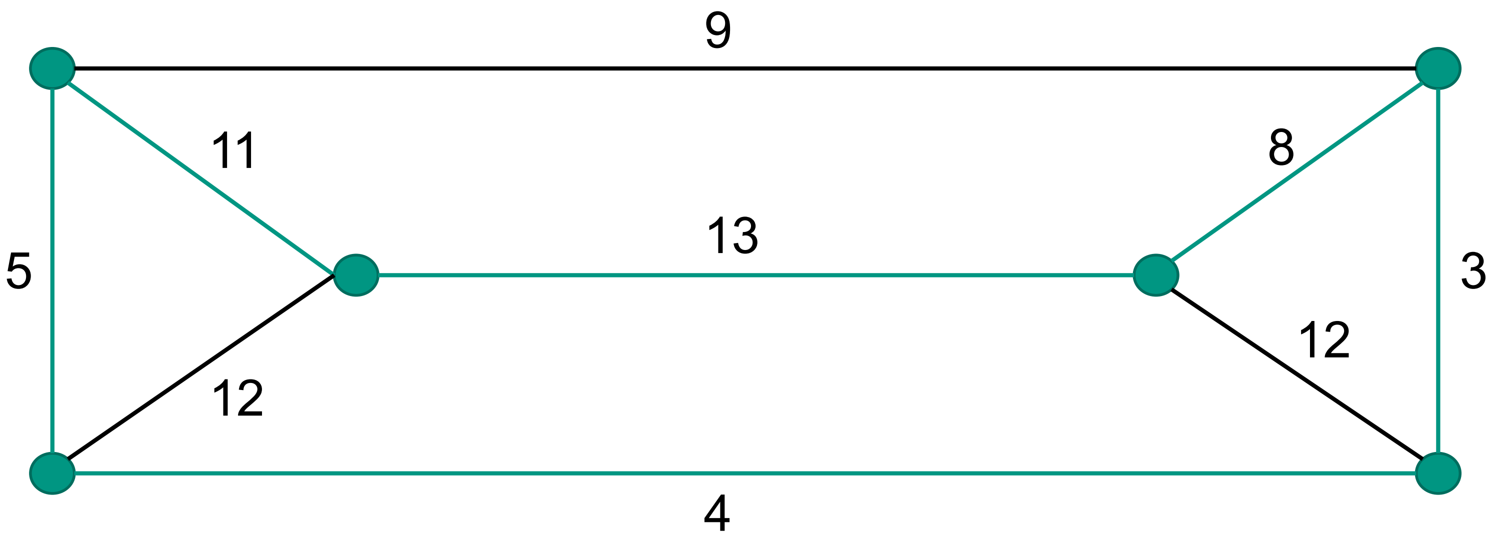
PGA



PGA

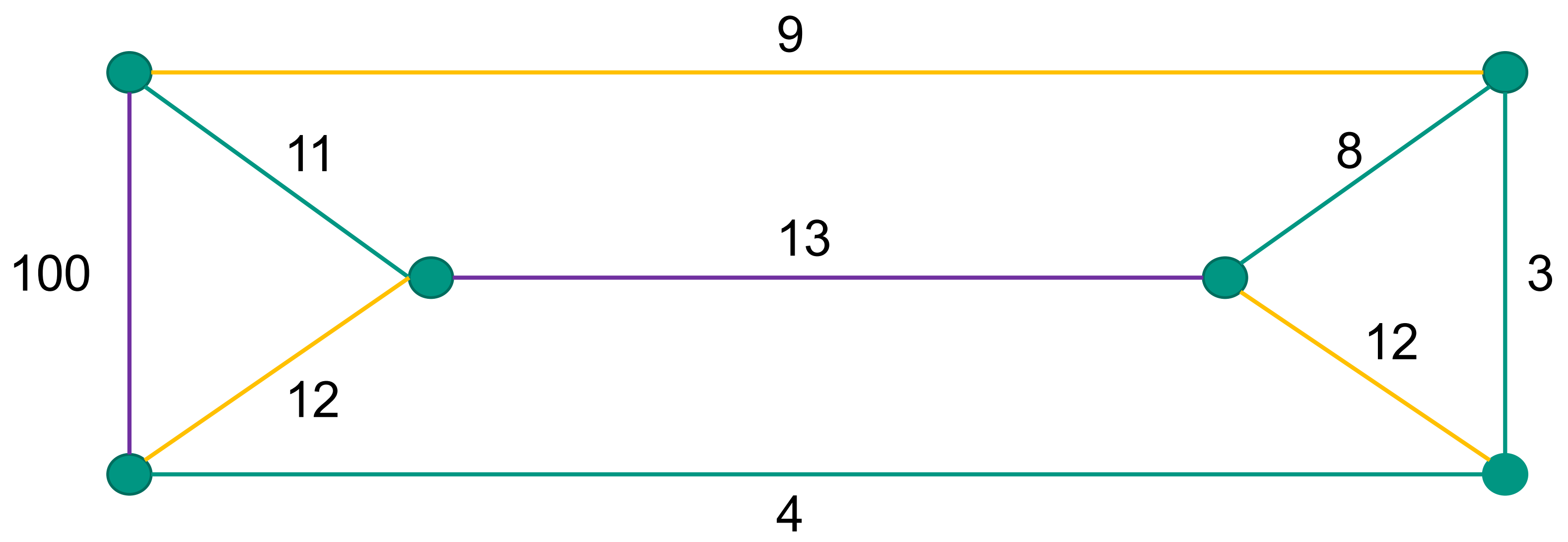


PGA

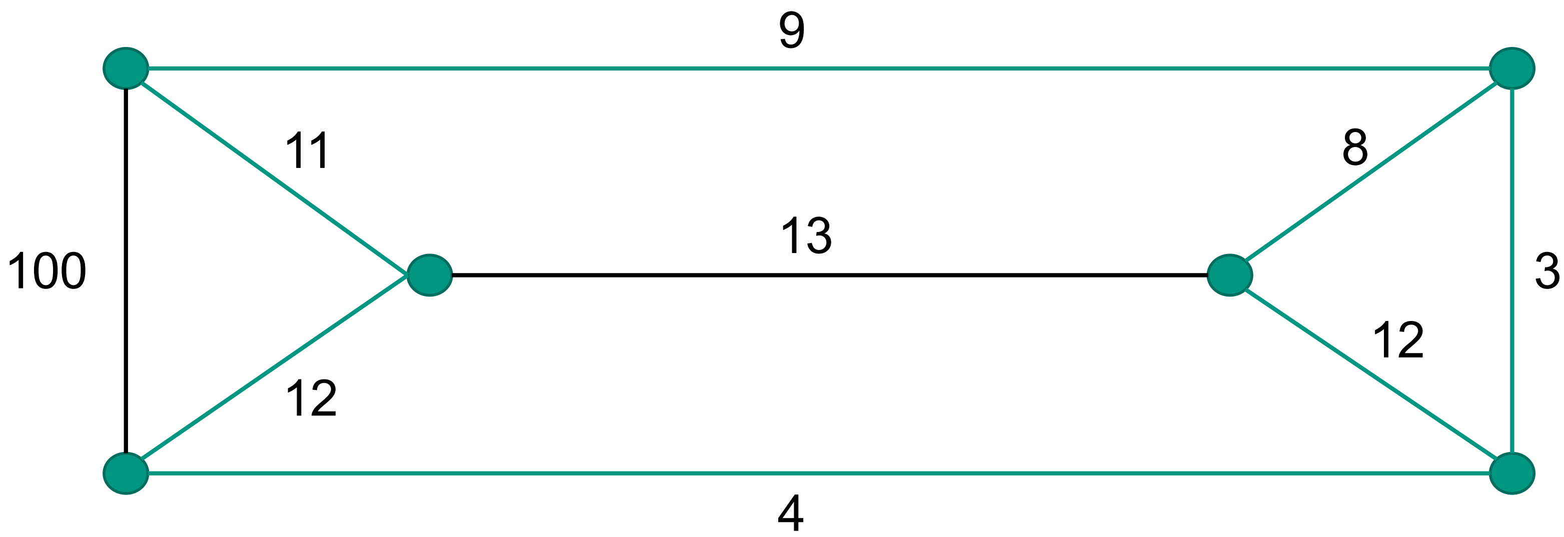


$$O(m)$$

PGA'

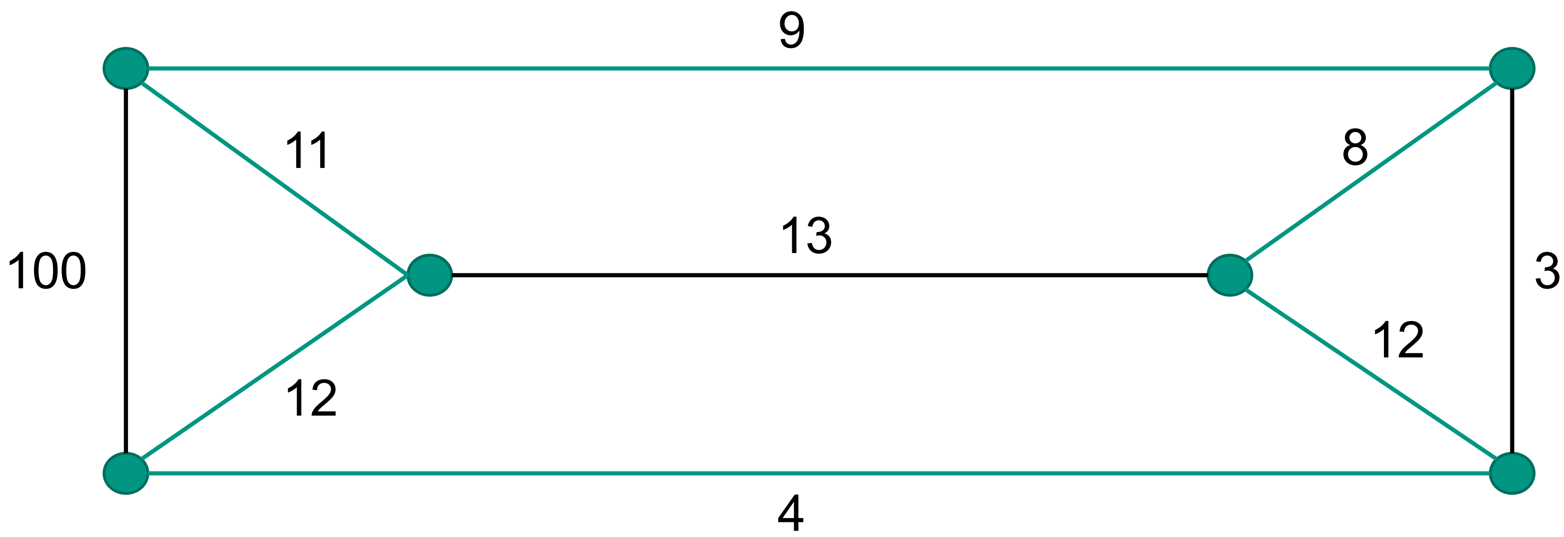


PGA‘

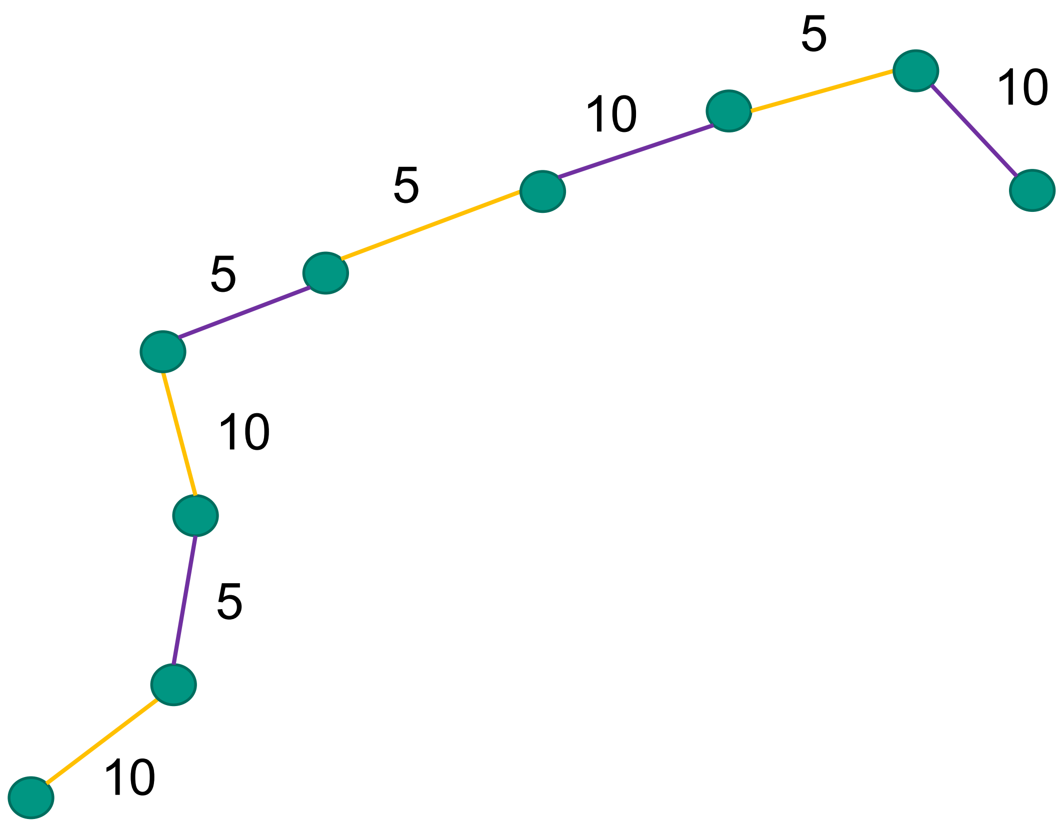


PGA‘

- Extend to maximal matching



PGA‘



PGA‘

- Dynamic Programming