HPS

https://github.com/lquenti/walky

Lars Quentin, Johann Carl Meyer, Dr. Artur Wachtel

Rusty Parallel Traveling Salesman Problem Solver

walky walky

Table of contents

Goals

1. Develop a CLI tool compatible with current state-of-the-art research

Goals

1. Develop a CLI tool compatible with current state-of-the-art research
2. Performance and Efficiency
   - Create a blazingly fast software package
   - Provide a 100% pure Rust alternative to classical solvers
   - Support both shared and distributed memory parallelization
   - Achieve full documentation coverage
   - Achieve high unit test coverage

Goals (cont.)

3. Exact Solving
   - Implement a simple, exact solver for the TSP
   - Offer several optimized versions
   - Create a shared memory parallelized verion
   - Develop a distributed memory, MPI-based parallelized solver

Goals (cont.)

3. Exact Solving
   - Implement a simple, exact solver for the TSP
   - Offer several optimized versions
   - Create a shared memory parallelized verion
   - Develop a distributed memory, MPI-based parallelized solver
4. Approximation Tactics
   - Include a trivial, easy to parallelize tactic and
   - A sophisticated, state of the art tactic
   - For both:
     - Provide a shared memory parallelized solver
     - Provide a distributed memory, MPI based parallelized solver

Goals (cont.)

3 Exact Solving
- Implement a simple, exact solver for the TSP
- Offer several optimized versions
- Create a shared memory parallelized verion
- Develop a distributed memory, MPI-based parallelized solver

4 Approximation Tactics
- Include a trivial, easy to parallelize tactic and
- A sophisticated, state of the art tactic
- For both:
  - Provide a shared memory parallelized solver
  - Provide a distributed memory, MPI based parallelized solver

5 Lower Bound Calculation for TSP
- Provide a sequential implementation
- Develop a parallelized implementation using MPI

Organizational Remark

Targeted Credits for this course:

- Lars: 9C
- Johann: 6C

See also https://hps.vi4io.org/_media/teaching/summer_term_2023/pchpc/pchpcassignment.pdf for expected work depending on the targeted credits.

Travelling Salesman Problem Definition

user "Kapitän Nemo" https://commons.wikimedia.org/w/index.php?curid=5584283

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" [**song_solving_2021**]

Travelling Salesman Problem Definition

- input graph
  - weighted, non-negative
  - undirected
  - complete (fully connected)

Travelling Salesman Problem Definition

- input graph
  - weighted, non-negative
  - undirected
  - complete (fully connected)
- output restrictions:
  - tour (cycle that visits every vertex)
  - use any edge *at most* one time

Travelling Salesman Problem Definition

- input graph
  - weighted, non-negative
  - undirected
  - complete (fully connected)
- output restrictions:
  - tour (cycle that visits every vertex)
  - use any edge *at most* one time
- problem: find a legal output that has minimal (cumulative) edge weight

Why is TSP interesting?

- well studied
- NP-complete $\rightarrow$ ressource intensive
- intuitive to understand

Why is TSP interesting?

- well studied
- NP-complete $\rightarrow$ ressource intensive
- intuitive to understand
- practical applications (see Concorde TSP Solver)

Our Implementation

- Publicly available on GitHub
- can be found on at `https://crates.io/crates/walky/`
- licensed under the MIT open source license

Naïve Approach

- Test out all possible paths
- Keep the shortest one
- Using Fast iterative enumeration algorithm [**nayuki_next_nodate**]
- First Optimization: Fixate the first element!
- Complexity: $\Theta(n!)$

Naïve Approach

- Test out all possible paths
- Keep the shortest one
- Using Fast iterative enumeration algorithm [**nayuki_next_nodate**]
- First Optimization: Fixate the first element!
- Complexity: $\Theta(n!)$

```
fn iterative_solver<T>(graph_matrix: &T) -> Solution
where
    T: AdjacencyMatrix,
{
    let n = graph_matrix.dim();
    let mut best_permutation: Path = (0..n).collect();
    let mut best_cost = f64::INFINITY;

    let mut curr = best_permutation.clone();
    while next_permutation(&mut curr[1..]) {
        let cost = graph_matrix.evaluate_circle(&curr);
        if cost < best_cost {
            best_cost = cost;
            best_permutation = curr.clone();
        }
    }
    (best_cost, best_permutation)
}
```

Cache Prefix Sums

- After every path, we compute the tour
- Reuse partial computations
- While enumerating, keep prefix as long as possible
  - **Recursive enumeration!**

Cache Prefix Sums

- After every path, we compute the tour
- Reuse partial computations
- While enumerating, keep prefix as long as possible
  - **Recursive enumeration!**

```python
def rec_enum(xs, n):
    """Recursively enumerate xs"""
    if len(xs) == n:
        print(xs)
    for i in range(n):
        if i not in xs:
            rec_enum(xs + [i], n)
```

Cache Prefix Sums

- After every path, we compute the tour
- Reuse partial computations
- While enumerating, keep prefix as long as possible
  - **Recursive enumeration!**

```python
def rec_enum(xs, n):
    """Recursively enumerate xs"""
    if len(xs) == n:
        print(xs)
    for i in range(n):
        if i not in xs:
            rec_enum(xs + [i], n)
```

But do we *actually* have to look at every solution?

Pruning

V1: Stop what doesn't work!

- Use the partial sum
- Lower bound: Previous best
- `if (partial_sum <= prev_best)`
  `rec_enum(...)`

Pruning

V1: Stop what doesn't work!

- Use the partial sum
- Lower bound: Previous best
- `if (partial_sum <= prev_best) rec_enum(...)`

V2: Nearest Neighbour (NN)

- prune if `(partial_sum + lower_bound)` of remaining vertices
- Lower bound:
  - Connect every vertex to the nearest one!

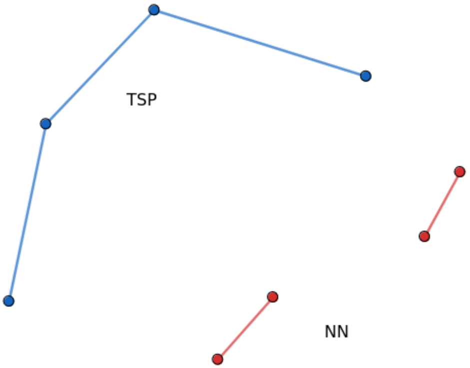
Pruning

V1: Stop what doesn't work!

- Use the partial sum
- Lower bound: Previous best
- `if (partial_sum <= prev_best) rec_enum(...)`

V2: Nearest Neighbour (NN)

- prune if `(partial_sum + lower_bound)` of remaining vertices
- Lower bound:
  - Connect every vertex to the nearest one!



Pruning (cont.)

V3: Minimal Spanning Tree (MST)

- Same idea
- Use Minimal Spanning Tree of remaining vertices
- $NN < MST < TSP$

Pruning (cont.)

V3: Minimal Spanning Tree (MST)

- Same idea
- Use Minimal Spanning Tree of remaining vertices
- $NN < MST < TSP$

V4: Caching

- Cache the MST in a HashMap
- Using a non-cryptographic HashMap

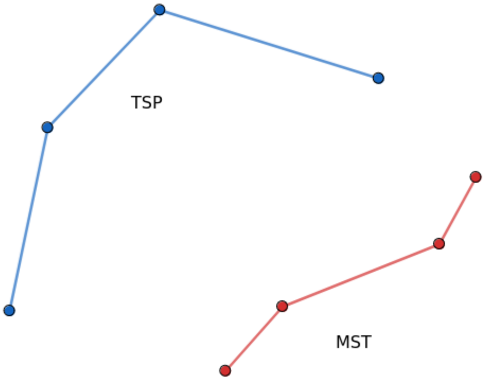
Pruning (cont.)

V3: Minimal Spanning Tree (MST)

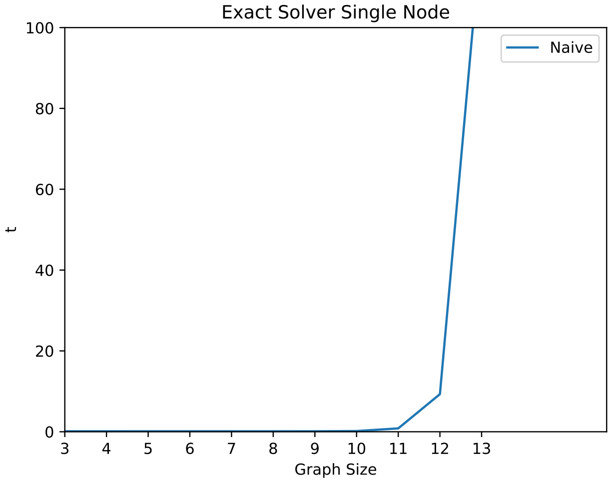
- Same idea
- Use Minimal Spanning Tree of remaining vertices
- $NN < MST < TSP$

V4: Caching

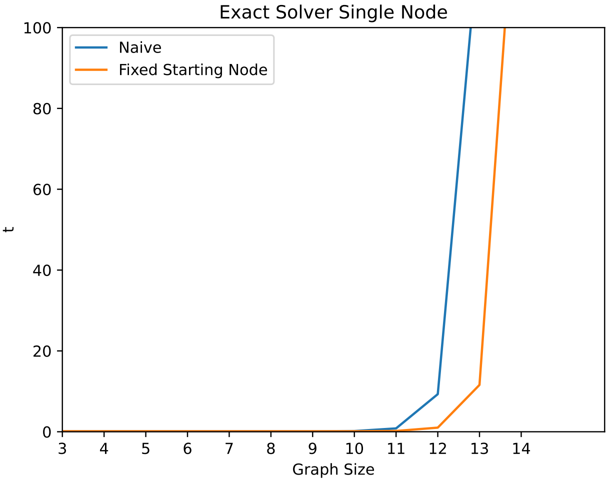
- Cache the MST in a HashMap
- Using a non-cryptographic HashMap



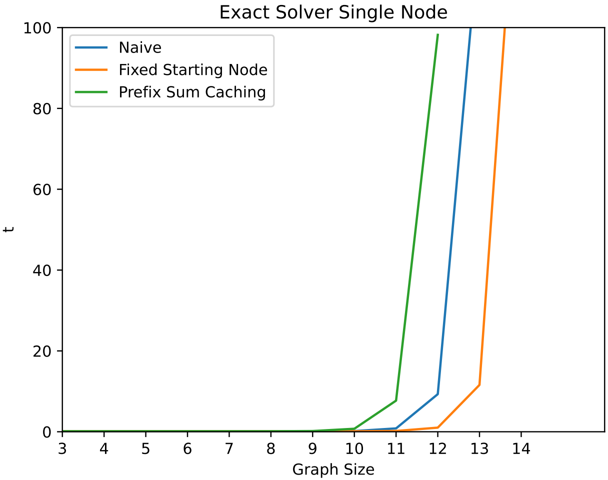
Benchmarking Results



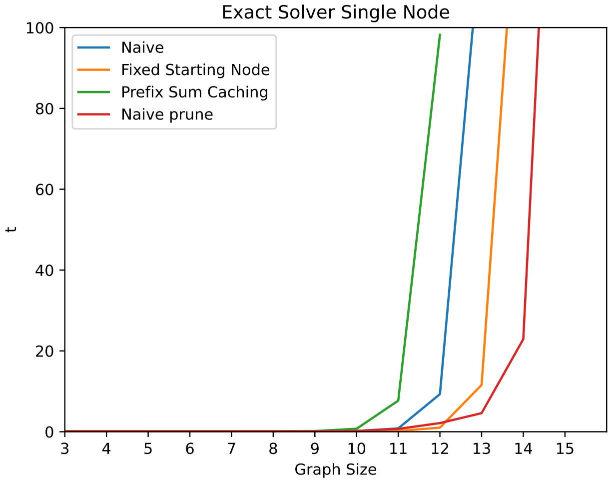
Benchmarking Results



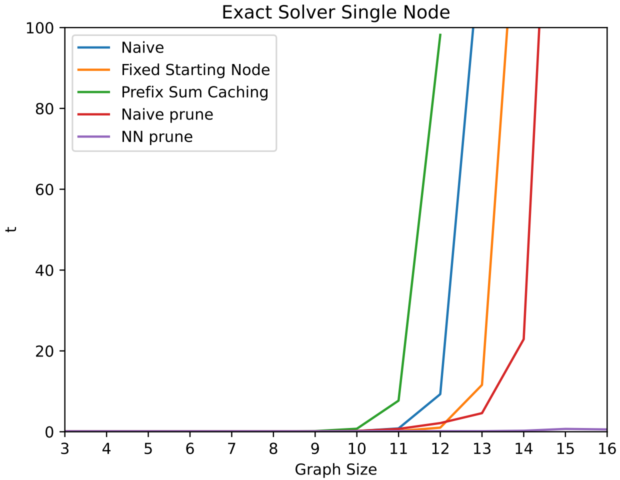
Benchmarking Results



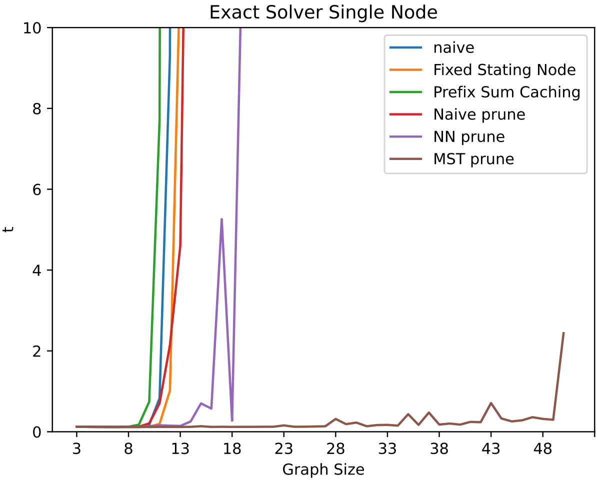
Benchmarking Results



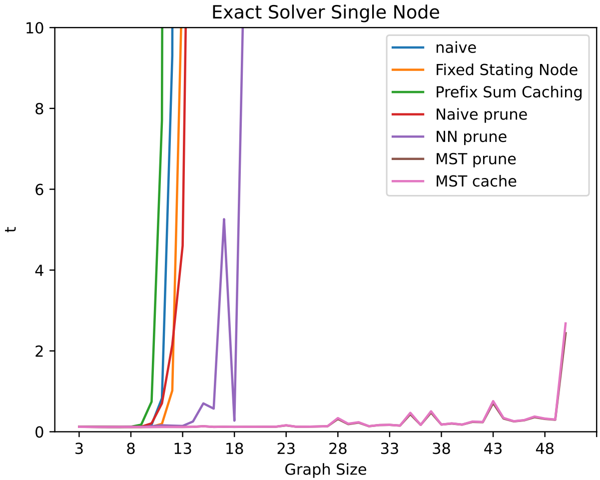
Benchmarking Results



Benchmarking Results



Benchmarking Results



Threading

Algorithm

Threading

Algorithm

1. Spawn n threads

Threading

Algorithm

1. Spawn n threads
2. Divide the prefix space locally, i-th thread gets i-th chunk

Threading

Algorithm

1. Spawn n threads
2. Divide the prefix space locally, i-th thread gets i-th chunk
3. Compute next prefix (with MST lower bound)

Threading

Algorithm

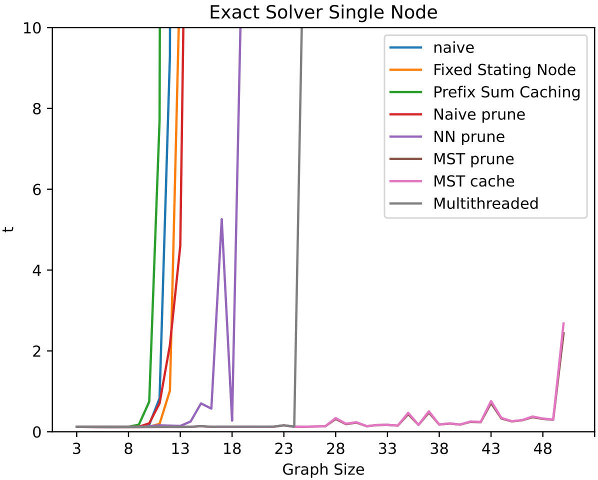
1. Spawn n threads
2. Divide the prefix space locally, i-th thread gets i-th chunk
3. Compute next prefix (with MST lower bound)
4. Update local optimum shared with all threads

Threading

Algorithm

1. Spawn n threads
2. Divide the prefix space locally, i-th thread gets i-th chunk
3. Compute next prefix (with MST lower bound)
4. Update local optimum shared with all threads
5. `GOTO` 3 until done with chunk

Benchmarking Results



MPI

Computation

- One coordinator, $n - 1$ worker

MPI

Computation

- One coordinator, $n-1$ worker
- Prefix chunk division like threaded

MPI

Computation

- One coordinator, $n - 1$ worker
- Prefix chunk division like threaded
- After each prefix, the worker reports current best **cost** to coordinator

MPI

Computation

- One coordinator, $n - 1$ worker
- Prefix chunk division like threaded
- After each prefix, the worker reports current best **cost** to coordinator
- Coordinator answers with global best **cost**
  - Tightest possible bound for pruning

MPI

Computation

- One coordinator, $n - 1$ worker
- Prefix chunk division like threaded
- After each prefix, the worker reports current best **cost** to coordinator
- Coordinator answers with global best **cost**
  - Tightest possible bound for pruning
- At the end, worker tells coordinator that its done and waits at barrier

MPI

Computation

- One coordinator, $n - 1$ worker
- Prefix chunk division like threaded
- After each prefix, the worker reports current best **cost** to coordinator
- Coordinator answers with global best **cost**
  - Tightest possible bound for pruning
- At the end, worker tells coordinator that its done and waits at barrier
- After all are done, coordinator joins the barrier

MPI (cont.)

Joining the local optima

MPI (cont.)

Joining the local optima

- After all are done, the coordinator broadcasts
  - which rank won
  - and the minimal cost

MPI (cont.)

Joining the local optima

- After all are done, the coordinator broadcasts
  - which rank won
  - and the minimal cost
- That rank then broadcasts **the full path**
  - This is an traffic efficiency optimization!

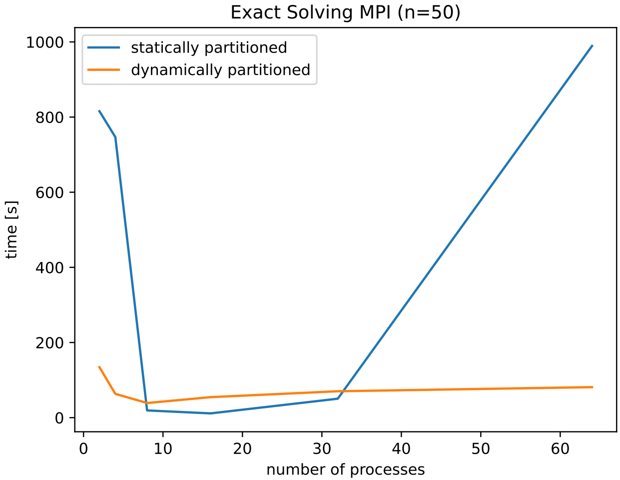
MPI (cont.)

Joining the local optima

- After all are done, the coordinator broadcasts
  - which rank won
  - and the minimal cost
- That rank then broadcasts **the full path**
  - This is an traffic efficiency optimization!

Now every process knows the best cost and path.

Benchmarks


Exact Solving MPI (n=50)
statically partitioned
dynamically partitioned
time [s]
0
200
400
600
800
1000
0
10
20
30
40
50
60
number of processes

Nearest Neighbour

Single Nearest Neighbour

1. Start at a random node
2. Check distances to all unvisited nodes
3. Go to the one with the shortest distance
4. GOTO 2 until all nodes are visited

Nearest Neighbour

Single Nearest Neighbour

1. Start at a random node
2. Check distances to all unvisited nodes
3. Go to the one with the shortest distance
4. GOTO 2 until all nodes are visited

Nearest Neighbour

- Do Single NN for every starting node
- Choose the best

Nearest Neighbour (cont.)

Single Threaded

```
n_random_numbers(0, graph_matrix.dim(), n)
    .into_iter()
    .map(|k| single_nearest_neighbour(graph_matrix, k))
    .min_by_key(|&(distance, _)| OrderedFloat(distance))
    .unwrap()

```

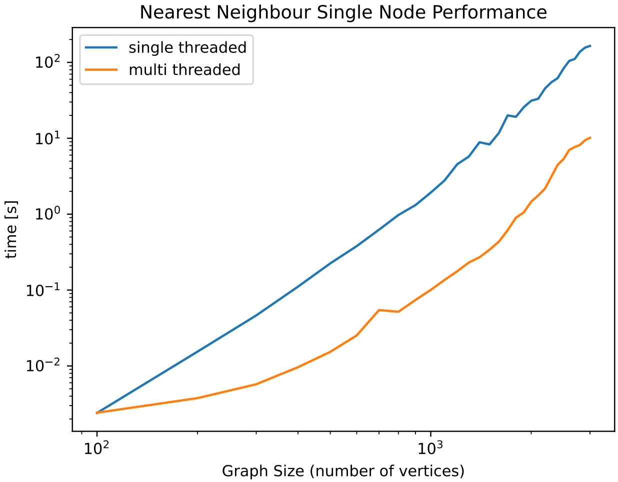
Nearest Neighbour (cont.)

Multi Threaded

```
n_random_numbers(0, graph_matrix.dim(), n)
    .into_par_iter()
    .map(|k| single_nearest_neighbour(graph_matrix, k))
    .min_by_key(|&(distance, _)| OrderedFloat(distance))
    .unwrap()

```

Benchmarks


Nearest Neighbour Single Node Performance
single threaded
multi threaded
time [s]
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{2}
10^{3}
Graph Size (number of vertices)

Nearest Neighbour: MPI

- Divide number of nodes into equal chunks

Nearest Neighbour: MPI

- Divide number of nodes into equal chunks
- Every process computes their chunks

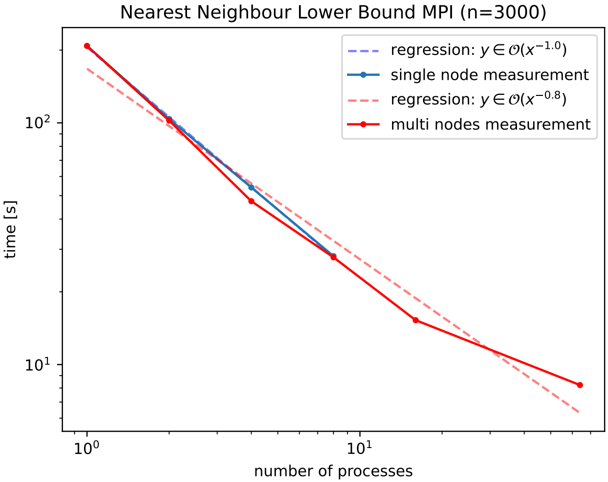
Nearest Neighbour: MPI

- Divide number of nodes into equal chunks
- Every process computes their chunks
- `MPI_Allreduce` the **cost** (and keep rank)

Nearest Neighbour: MPI

- Divide number of nodes into equal chunks
- Every process computes their chunks
- `MPI_Allreduce` the **cost** (and keep rank)
- Winner rank `MPI_Bcast` the solution path.

Benchmarks


Nearest Neighbour Lower Bound MPI (n=3000)
regression: $y \in \mathcal{O}(x^{-1.0})$
single node measurement
regression: $y \in \mathcal{O}(x^{-0.8})$
multi nodes measurement
time [s]
10^2
10^1
10^0
10^1
number of processes

Christofides Algorithm

- Assumption: the input graph is metric, i.e. the triangle inequality holds

Christofides Algorithm

- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST

Christofides Algorithm

- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST
  2. calculate a matching in the complete graph of minimum weight, over all vertices, that have odd degree in the MST
     - see also next slides

Christofides Algorithm

- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST
  2. calculate a matching in the complete graph of minimum weight, over all vertices, that have odd degree in the MST
     - see also next slides
     - parallelization: mostly in this step

Christofides Algorithm

- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST
  2. calculate a matching in the complete graph of minimum weight, over all vertices, that have odd degree in the MST
     - see also next slides
     - parallelization: mostly in this step
  3. combine the MST and the matching into one multigraph

Christofides Algorithm

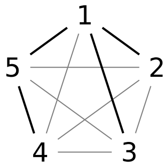
- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST
  2. calculate a matching in the complete graph of minimum weight, over all vertices, that have odd degree in the MST
     - see also next slides
     - parallelization: mostly in this step
  3. combine the MST and the matching into one multigraph
  4. find an eulerian cycle through the multigraph

Christofides Algorithm

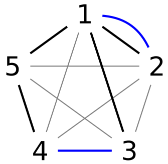
- Assumption: the input graph is metric, i.e. the triangle inequality holds
- the algorithm goes as following [**christofides_worst-case_1976**]:
  1. calculate the MST
  2. calculate a matching in the complete graph of minimum weight, over all vertices, that have odd degree in the MST
     - see also next slides
     - parallelization: mostly in this step
  3. combine the MST and the matching into one multigraph
  4. find an eulerian cycle through the multigraph
  5. make the eulerian cycle hamiltonian

Christofides Algorithm: Where To Find A Matching?

Complete input graph with highlighted MST:



Vertices with odd degree: $1, 2, 3, 4$. $\hookrightarrow$ Find a matching over these vertices (blue):



Note: edge weights are left out for simplicity

Christofides Algorithm: Finding A Matching

Finding a minimum cost matching:

- exact solution:
  - uses a sophisticated algorithm (the blossom algorithm)
  - hard to parallelize
  - slow (uses a lot of HashSets)

Christofides Algorithm: Finding A Matching

Finding a minimum cost matching:

- exact solution:
  - uses a sophisticated algorithm (the blossom algorithm)
  - hard to parallelize
  - slow (uses a lot of HashSets)
- randomized approximate solution:
  - idea: guess a matching and do some randomized improvements. Repeat this and take the best matching
  - easy to implement
  - easy to parallelize

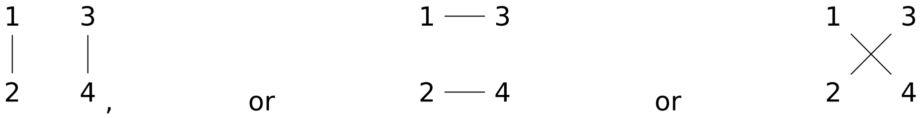
Christofides Algorithm: Randomly Finding A Matching

Finding a matching: the graph is complete & has even amount of vertices (trivial)

1. Given the list of all vertices $[0, 1, 2, 3, 4, 5, 6, 7]$
2. randomly scramble the list: $[2, 1, 0, 3, 7, 5, 6, 4]$
3. interpret the list as a matching: $[[2, 1], [0, 3], [7, 5], [6, 4]]$

Christofides Algorithm: Improving A Matching Of 4 Vertices

Improving a matching on 4 vertices: easy: only 3 cases to consider:



Chose the matching with the lowest cost.

Christofides Algorithm: Randomly Improving A Matching

Improving a matching:
improve pairs of edges:

1. Given a matching $[[2,1],[0,3],[7,5],[6,4]]$
2. randomly scramble the list: $[[7,5],[0,3],[2,1],[6,4]]$
3. consider consecutive blocks of two edges: $[7,5],[0,3]$ and $[2,1],[6,4]$
4. for a block of two edges, consider the other two possible matchings among the four vertices, are they better? Given: $[7,5],[0,3]$ consider $[7,0],[5,3]$ and $[7,3],[5,0]$
5. repeat with step 2

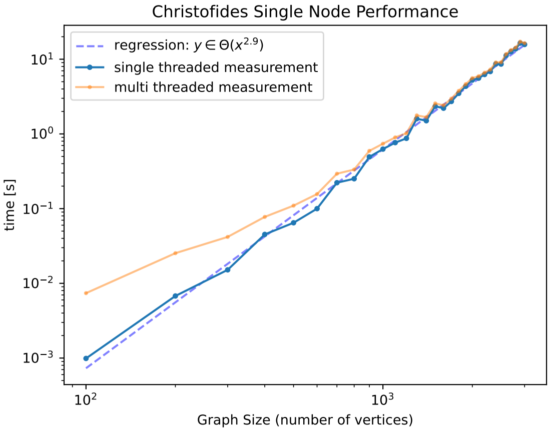
Christofides Algorithm: Randomly Improving A Matching In Parallel

Parallelize the randomized algorithm: do the same thing many times in parallel

1. each process: generates a random matching, and randomly improves it
2. then: pick the best result and return it

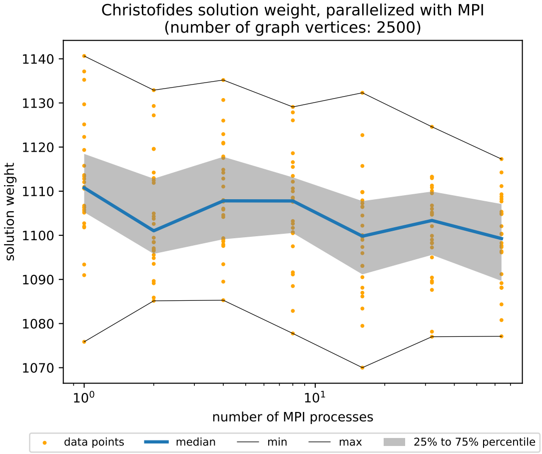
Christofides Algorithm: Benchmarking

Christofides algorithm does not benefit from parallelization w.r.t. execution time:



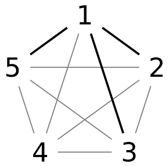
Christofides Algorithm: Benchmarking

Christofides algorithm does slightly benefit w.r.t. solution weight:

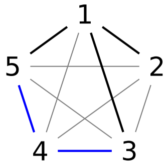


How To Get A 1-tree Lower Bound?

Start with an MST over $n-1$ edges (here vertex 4 is left out):



Then add the remaining vertex, and the two edges with lowest cost adjacent to that vertex:

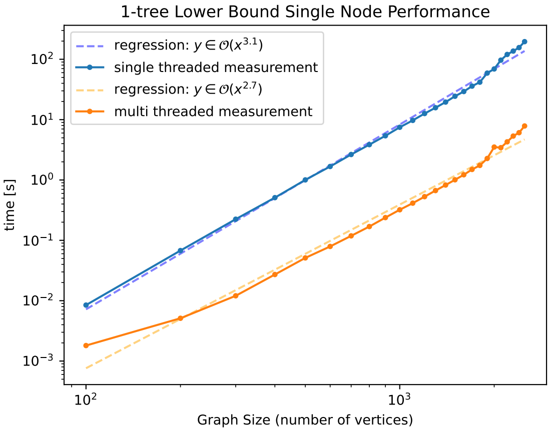


Lower Bound With 1-tree on TSP

- any 1-tree weight is a lower bound on the TSP solution [**held_traveling-salesman_1970**]
- $|V|$ 1-trees to check independently
- very easy to parallelize

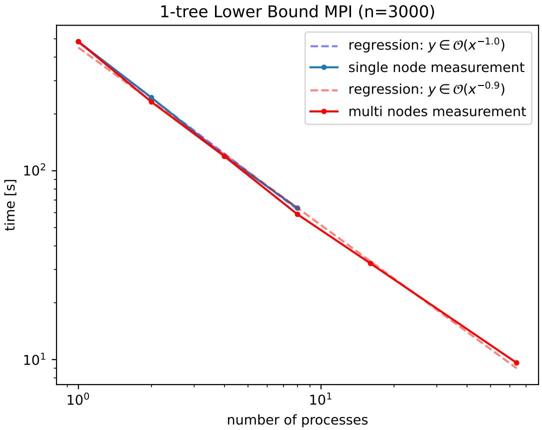
1-tree Lower Bound Benchmarking

The 1-tree lower bound benefits from parallelization:



1-tree Lower Bound Benchmarking

The 1-tree lower bound benefits from parallelization:



Future Work

Exact Solver: Dynamic Load Distribution

- Pruning makes the actual work load unpredictable
- Instead of dividing chunks, the coordinator gives out work dynamically
- Pro: More equal work distribution
- Contra: More communication

Future Work

Exact Solver: Dynamic Load Distribution

- Pruning makes the actual work load unpredictable
- Instead of dividing chunks, the coordinator gives out work dynamically
- Pro: More equal work distribution
- Contra: More communication

More MPI analysis and performance tuning

- Especially using Vampir

Contribution

1. Developed a CLI tool compatible with TSPLIB

Contribution

1. Developed a CLI tool compatible with TSPLIB
2. Provided a software that is
   - Blazingly fast
   - Pure Rust (compatible with C-based MPI flavours)
   - Supports shared- and distributed memory parallelization
   - Well documented and thoroughly tested

Contribution

1. Developed a CLI tool compatible with TSPLIB
2. Provided a software that is
   - Blazingly fast
   - Pure Rust (compatible with C-based MPI flavours)
   - Supports shared- and distributed memory parallelization
   - Well documented and thoroughly tested
3. Including an exact solver
   - With several pruning-based optimizations
   - Both shared- and distributed memory parallelized

Contribution

1. Developed a CLI tool compatible with TSPLIB
2. Provided a software that is
   - Blazingly fast
   - Pure Rust (compatible with C-based MPI flavours)
   - Supports shared- and distributed memory parallelization
   - Well documented and thoroughly tested
3. Including an exact solver
   - With several pruning-based optimizations
   - Both shared- and distributed memory parallelized
4. And multiple approximate solvers
   - Including the easy to parallelize "Nearest Neighbour" method
   - Supporting the sophisticated "Christofides" algorithm
   - Both shared- and distributed memory parallelized

Contribution

1. Developed a CLI tool compatible with TSPLIB
2. Provided a software that is
   - Blazingly fast
   - Pure Rust (compatible with C-based MPI flavours)
   - Supports shared- and distributed memory parallelization
   - Well documented and thoroughly tested
3. Including an exact solver
   - With several pruning-based optimizations
   - Both shared- and distributed memory parallelized
4. And multiple approximate solvers
   - Including the easy to parallelize "Nearest Neighbour" method
   - Supporting the sophisticated "Christofides" algorithm
   - Both shared- and distributed memory parallelized
5. Implemented the 1-tree lower bound
   - Utilized shared- and distributed memory parallelization

References