Technische
Universität
Braunschweig

Algorithm Engineering 2026: Sheet 02

Rouven Kniep, April 23, 2026

Ex1: Setting

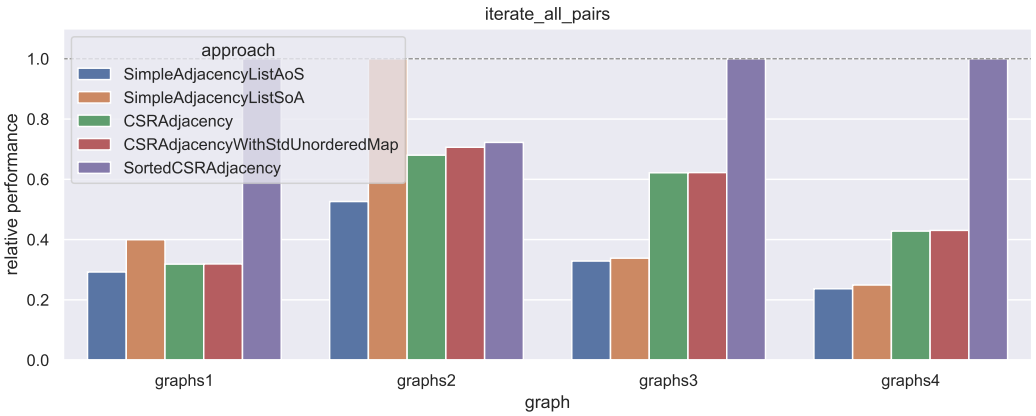
Data Structures

- Adjacency List
  - Array of Structs
  - Struct of Arrays
- Compressed Sparse Row
  - vector
  - sorted vector
  - `std::unordered_map`
    $\Longrightarrow$ hash as key to array of linked lists

Graphs

G	Vertices	Density	Notes
1	1000	medium	Erdős-Renyi
2	200	very dense	Erdős-Renyi
3	100000	sparse	
4	100000 100	sparser dense	usually sparse, high-degree

1a: Data



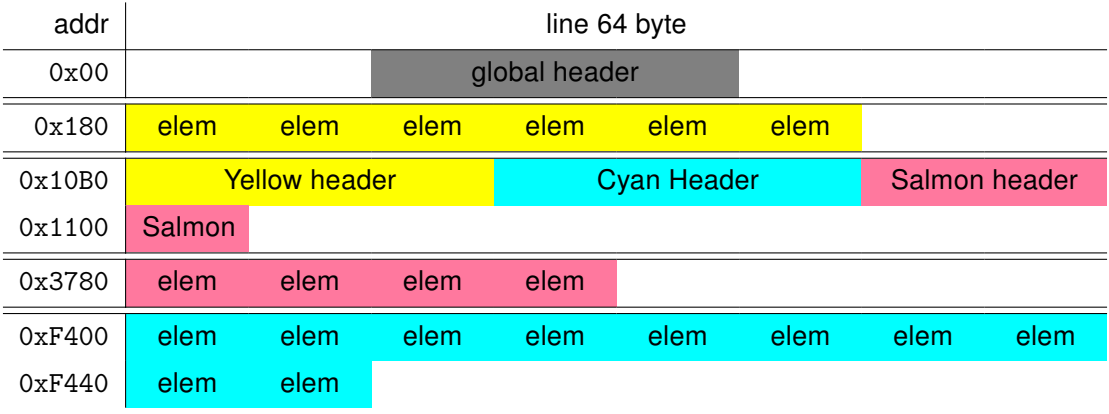
few vertices, medium dense

very few vertices, very dense

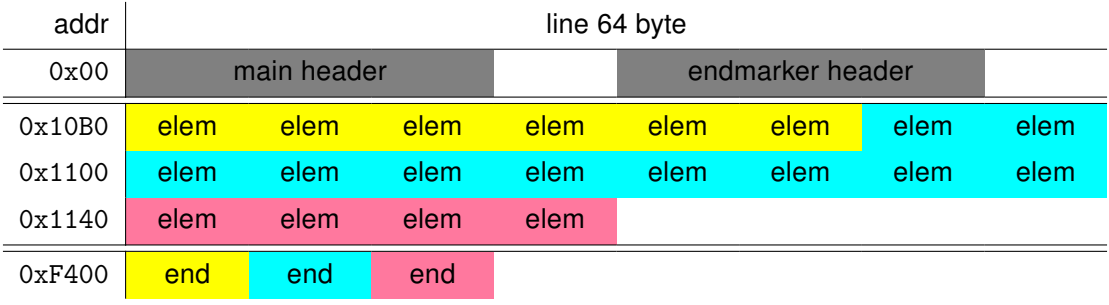
many vertices, sparse

many vertices, most very sparse

Nested Vector memory layout...



Compressed Sparse Row (CSR) memory layout...



$\Longrightarrow$ much better linearity and efficiency for low degrees!

CSR vs sorted CSR `for_each` performance

same code, relevant section:

```
1  for (Index i = start; i < end; ++i) {
2    Index partner = m_partners[i];
3    if (partner < u) { continue; }
4    Index pair_index = m_pair_indices[i];
5    if constexpr(result_is_boolean) {
6      if (!static_cast<bool>(
7      std::invoke(std::forward<Callable>(func),
8      u, partner, pair_index))
9      ) { return; }
10   } else {
11     std::invoke(std::forward<Callable>(func), u, partner, pair_index);
12   }
13 }
```

CSR vs sorted CSR `for_each` performance

`m_pair_indices`: data only, no program flow impact

```
1  for (Index i = start; i < end; ++i) {
2    Index partner = m_partners[i];
3    if (partner < u) { continue; }
4    Index pair_index = m_pair_indices[i];       // load from memory..
5    if constexpr(result_is_boolean) {
6      if (!static_cast<bool>(
7      std::invoke(std::forward<Callable>(func),
8      u, partner, pair_index))    // and feed into the invoked function
9      ) { return; }
10   } else {
11     std::invoke(std::forward<Callable>(func), u, partner, pair_index);
12   }
13 }
```

CSR vs sorted CSR `for_each` performance

`m_partners`: unsorted case

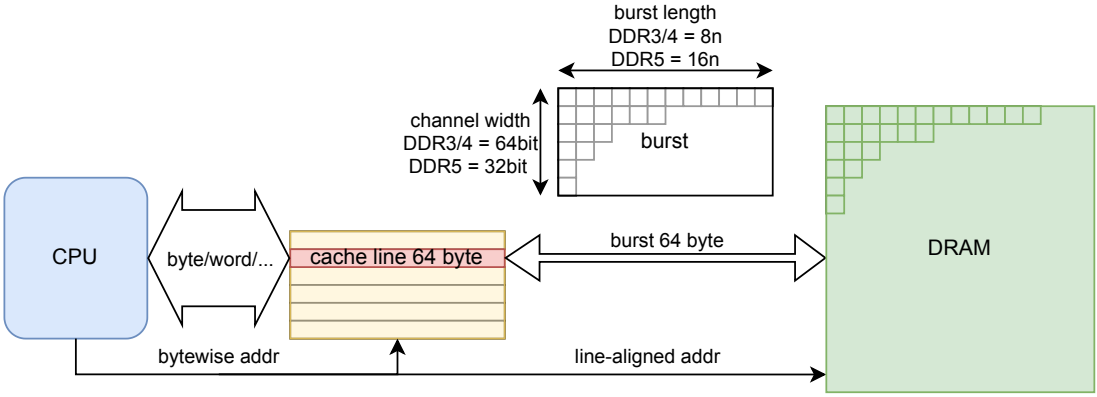
```
1  for (Index i = start; i < end; ++i) {
2    Index partner = m_partners[i];   // load from memory, data unsorted
3    if (partner < u) { continue; }   // branch -> cosly mispredictions
4    Index pair_index = m_pair_indices[i];
5    if constexpr(result_is_boolean) {
6      if (!static_cast<bool>(
7      std::invoke(std::forward<Callable>(func),
8      u, partner, pair_index))
9      ) { return; }
10   } else {
11     std::invoke(std::forward<Callable>(func), u, partner, pair_index);
12   }
13 }
```

CSR vs sorted CSR `for_each` performance

`m_partners`: sorted case

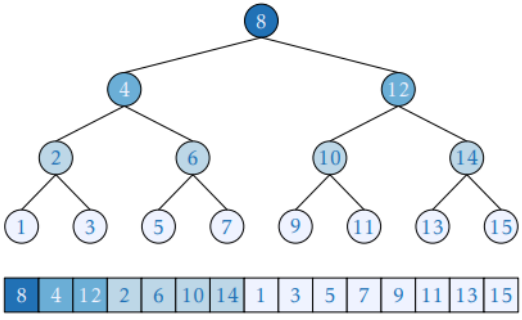
```
1  for (Index i = start; i < end; ++i) {
2    Index partner = m_partners[i];   // load from memory, data sorted
3    if (partner < u) { continue; }   // branch -> less mispredictions
4    Index pair_index = m_pair_indices[i];
5    if constexpr(result_is_boolean) {
6      if (!static_cast<bool>(
7      std::invoke(std::forward<Callable>(func),
8      u, partner, pair_index))
9      ) { return; }
10   } else {
11     std::invoke(std::forward<Callable>(func), u, partner, pair_index);
12   }
13 }
```

Cache Lines


burst length
DDR3/4 = 8n
DDR5 = 16n
channel width
DDR3/4 = 64bit
DDR5 = 32bit
burst
CPU
byte/word/...
cache line 64 byte
burst 64 byte
DRAM
bytewise addr
line-aligned addr

Eytzinger-Layout

from: https://algorithmica.org/en/eytzinger



- smaller child: $2k$
- bigger child: $2k + 1$
- children in d steps: in $[2^d * k, 2^{d+1} * k)$

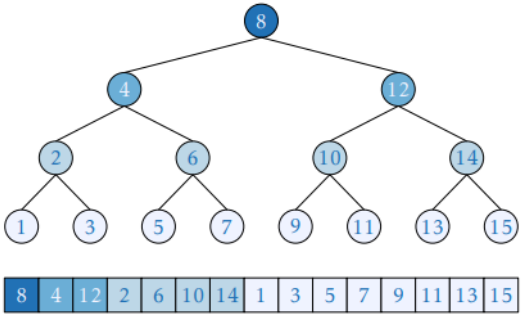
Eytzinger in memory

line 64 byte, elements 8 byte							
empty	1 (root)	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

- can prefetch an aligned line *without cost* d steps in advance, with $d = log_2(\frac{|l|}{|e|}) = log_2(\frac{64\,\text{B}}{64\,\text{bit}}) = 3$
- $\implies$ more efficient for small data types
- requires software prefetching:
  `__builtin_prefetch(base + k * 64/sizeof(node_type_t);`
- potentially even prefetching multiple lines
- requires alignment: `alignas(64) node_type_t base[n+1]`

Eytzinger-Layout: Caveats

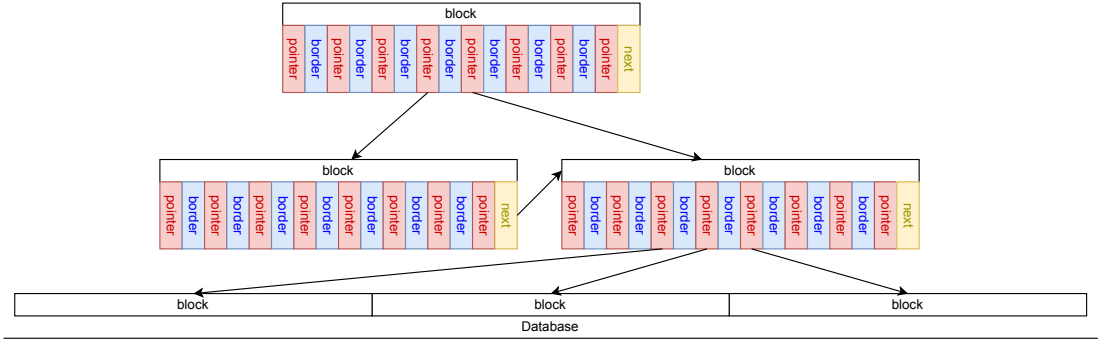
graphic from: https://algorithmica.org/en/eytzinger



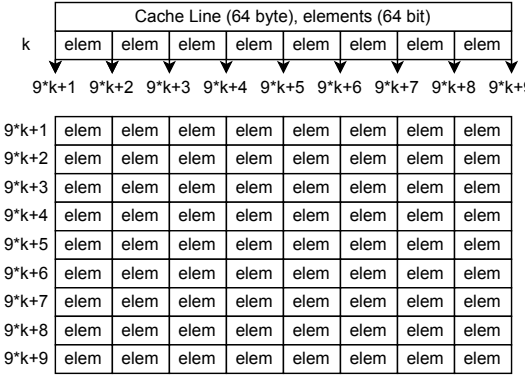
- performance gain up to $d = 3\times$ by pipelining away memory latency
- but low memory utilization: only $\frac{|e|}{|l|} = \frac{64\,\text{bit}}{64\,\text{B}} = \frac{1}{8}$ bytes actually used
- $\Longrightarrow$ bad in memory-constrained environments
  (which is basically anywhere)

B+-Trees ("Block"-Trees? b+1-ary Trees?)

note: this implementation is used in data base indices



implicit B-Trees

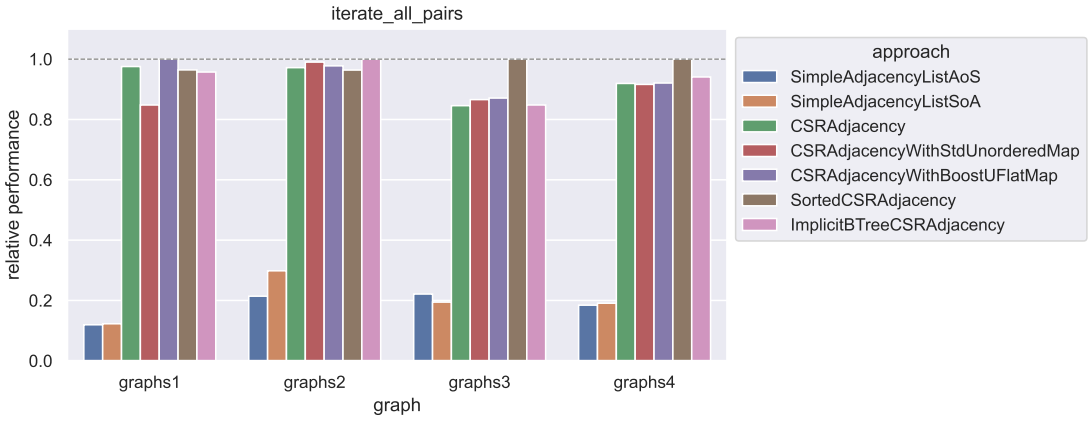


- $b = \frac{|e|}{|I|} = 8$ elements per node
- $b + 1$ children ("b+1-ary tree")
- when linearized: children of k in $[(b+1) * k + 1, (b+1) * k + b + 1]$
- shallow tree: depth $log_{b+1}(n)$
- memory-bandwidth friendlier
- but needs more compute

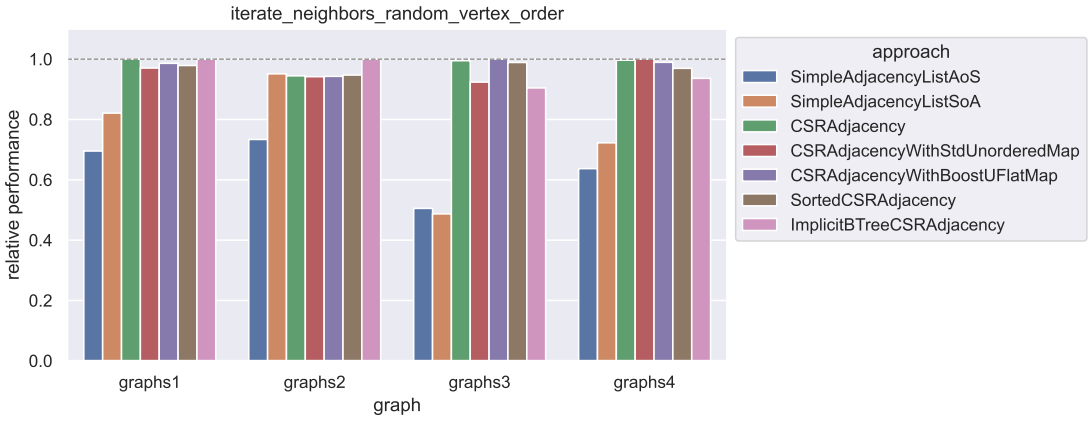
ImplicitBTreeCSRAdjacency::p_get_pair_index **main loop**

```
1  __m256i block1_vec = _mm256_loadu_si256((const __m256i*)(&b[k*BLOCK_SIZE]))
2  block1_vec = _mm256_xor_si256(block1_vec, range_shift);
3  __m256i block2_vec = _mm256_loadu_si256((const __m256i*)(&b[k*BLOCK_SIZE+4]]
4  block2_vec = _mm256_xor_si256(block2_vec, range_shift);
5  __m256i mask1 = _mm256_cmpgt_epi64(search_vec, block1_vec);
6  __m256i mask2 = _mm256_cmpgt_epi64(search_vec, block2_vec);
7  unsigned m1 = _mm256_movemask_pd(_mm256_castsi256_pd(mask1));
8  unsigned m2 = _mm256_movemask_pd(_mm256_castsi256_pd(mask2));
9  unsigned mask = ~(m1 | (m2 << 4));
10 Index i = std::countr_zero(mask);
11 if(i < BLOCK_SIZE) { res = k * BLOCK_SIZE + i; }
12 k = go(k, i);
```

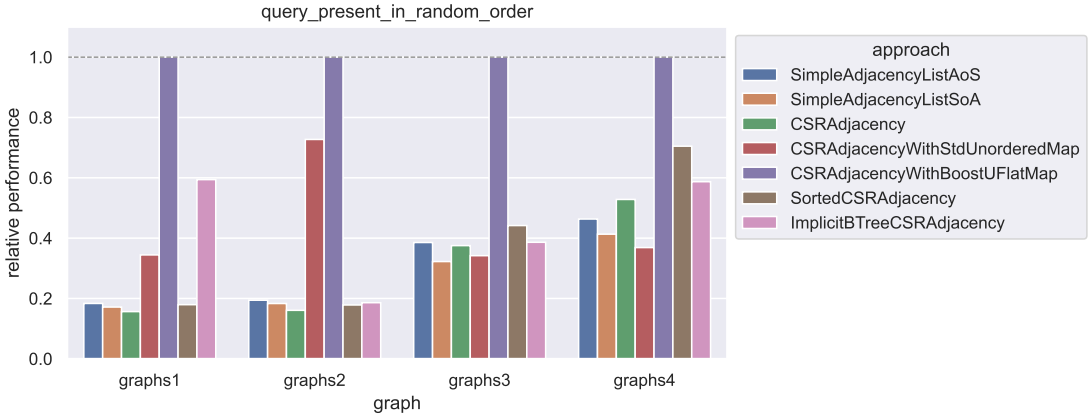
Performance: `iterate_all_pairs`



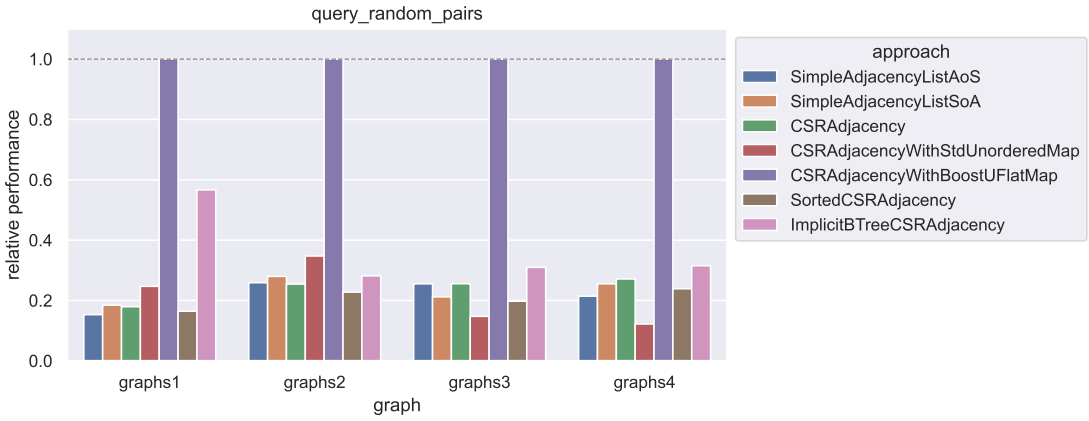
Performance: `iterate_neighbors_random_vertex_order`



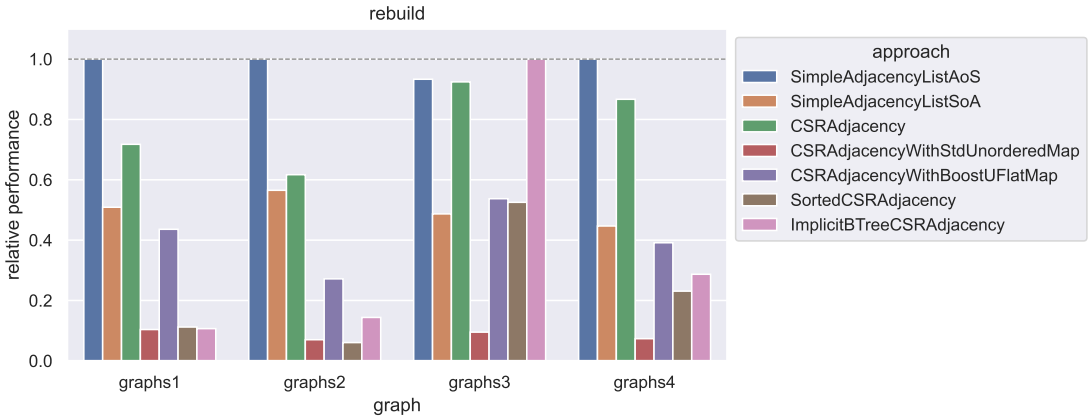
Performance: query_present_in_random_order



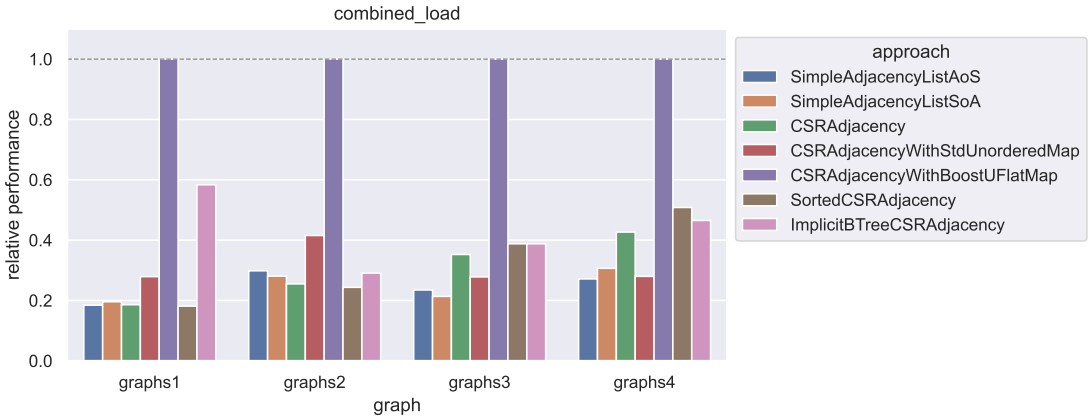
Performance: `query_random_pairs`



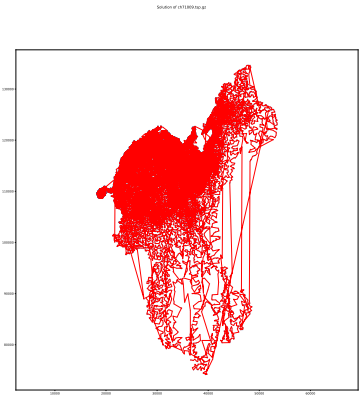
Performance: `rebuild`



Performance: `combined_load`



failed Idea 1: Divide+Conquer



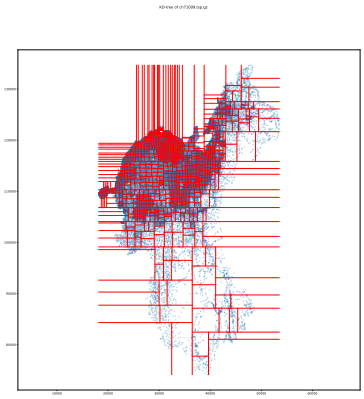
Idea:

- Griddify graph
- stitch grid cells in snake order
- run 2opt on grid cells

But:

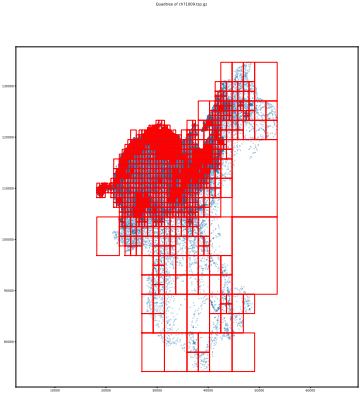
- locally nice..
- .. but merge step too bad
- bad solution quality

KD-Tree (limited depth)



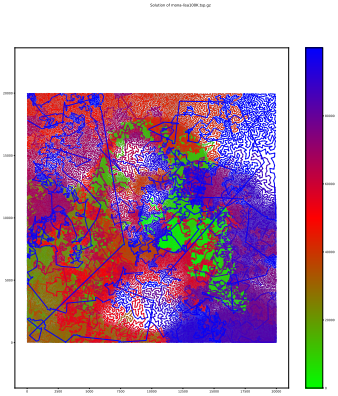
- querys decide based on median
- very good balance
- but search requires a lot of memory accesses (median reads)

Quad-Tree



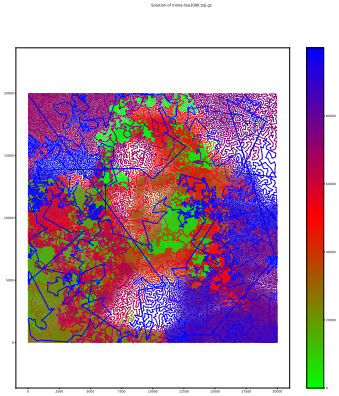
- querys decide based on BBOX
- balance not ideal
- search can require very few memory reads:
  - branch not data-dependent
  - implicit layout -> no pointers
  - bitvector for node presence/status

Greedy: Nearest Neighbour



- very fast
- solution quality much better
- but.. not quite good enough.
- especially later parts of the tour often cross

Greedy: Nearest Neighbour (multishot)

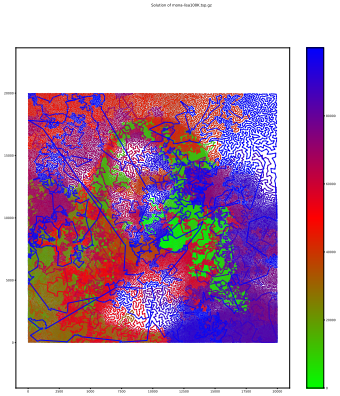


multiple attempts, similar to Sheet 1 Ex 03c

instance	tourlen	time	gap
sw24978	1,057,479	0.05 s	23.6 %
ch71009	5,629,180	0.14 s	23.3 %
mona-lisa100K	6,868,893	0.22 s	19.3 %
lra498378	2,690,495	1.00 s	24.1 %

PASSED

Greedy: Nearest Neighbour + twoopt



add twoopt-passes

- windowed 2opt (200 interval) $\Longrightarrow \mathcal{O}(n)$
- section twoopt (last 2000) $\Longrightarrow \mathcal{O}(1)$

instance	tourlen	time	gap
sw24978	994,382	0.27 s	16.2 %
ch71009	5,355,160	0.52 s	17.3 %
mona-lisa100K	6,554,454	0.66 s	13.8 %
lra498378	2,563,243	3.92 s	18.2 %

PASSED

Other Heuristics...

https://hal.science/hal-04909391v2/file/TSP_2024_V1.11.pdf

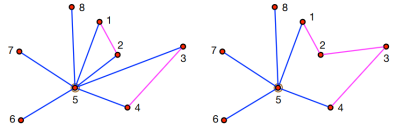


Figure 22: Merging of two pseudo-tours by linking vertices 2 and 3

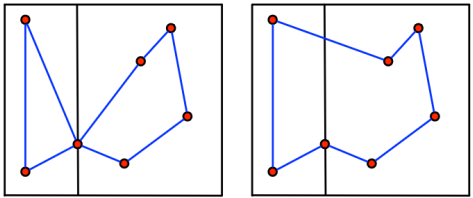


Figure 29: Patching subtours

Other Heuristics...

https://hal.science/hal-04909391v2/file/TSP_2024_V1.11.pdf

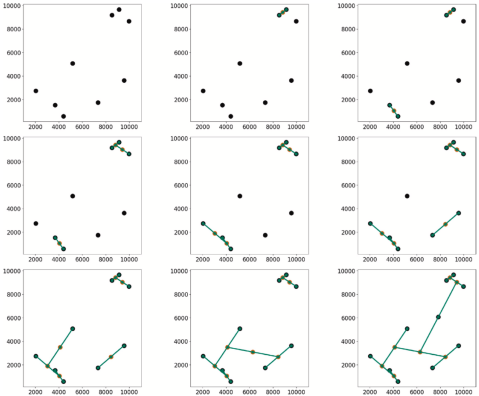


Figure 31: Visualization of the first phase – the clustering phase – of the pair-center algorithm (from A. Formella [12])

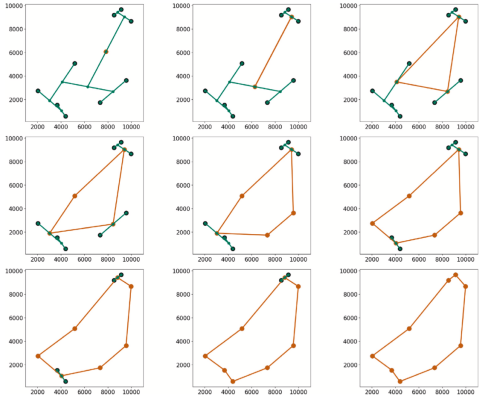


Figure 32: Visualization of the second phase – the construction phase – of the pair-center algorithm (from A. Formella [12])