Computer science

— Theory vs Experimentation —

Part 3: Algorithm Engineering

Christine Solnon

INSA de Lyon - 5IF

2022 / 2023

In almost every computation, a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Ada Byron, 1843

3 possible levels of tuning:

- Algorithm $\rightsquigarrow$ Divide-and-conquer, Dynamic Programming, ...
- Code $\rightsquigarrow$ Loops, Memory management, ...
- Parameters $\rightsquigarrow$ Best setting for each instance / class of instances

Goal:

- Improve performance (time, memory consumption, ...)
- In most cases, theoretical complexities are not changed
  But empirical performance may be greatly improved!

Plan

Some General Rules to Improve Algorithms

Use memory to save time

- Memoize sub-problem solutions (dynamic programming)
- Incrementally maintain data instead of recomputing it from scratch
- etc...

Use relevant data structures

- Study operation frequencies to choose the best data structure
  $\rightsquigarrow$ Hash table, Tree, Heap, Disjoint-sets, Sparse-sets, Dancing links, ...

Exit from loops as soon as possible

Examples: Dijkstra, Bellman-Ford, ...

Prune branches of search trees

- Compute tight bounds on objective functions $\rightsquigarrow$ Branch & Bound
- Propagate constraints $\rightsquigarrow$ Branch & Propagate

Illustration on the TSP

Permut: Enumate all permutations $\rightsquigarrow \mathcal{O}(n!)$

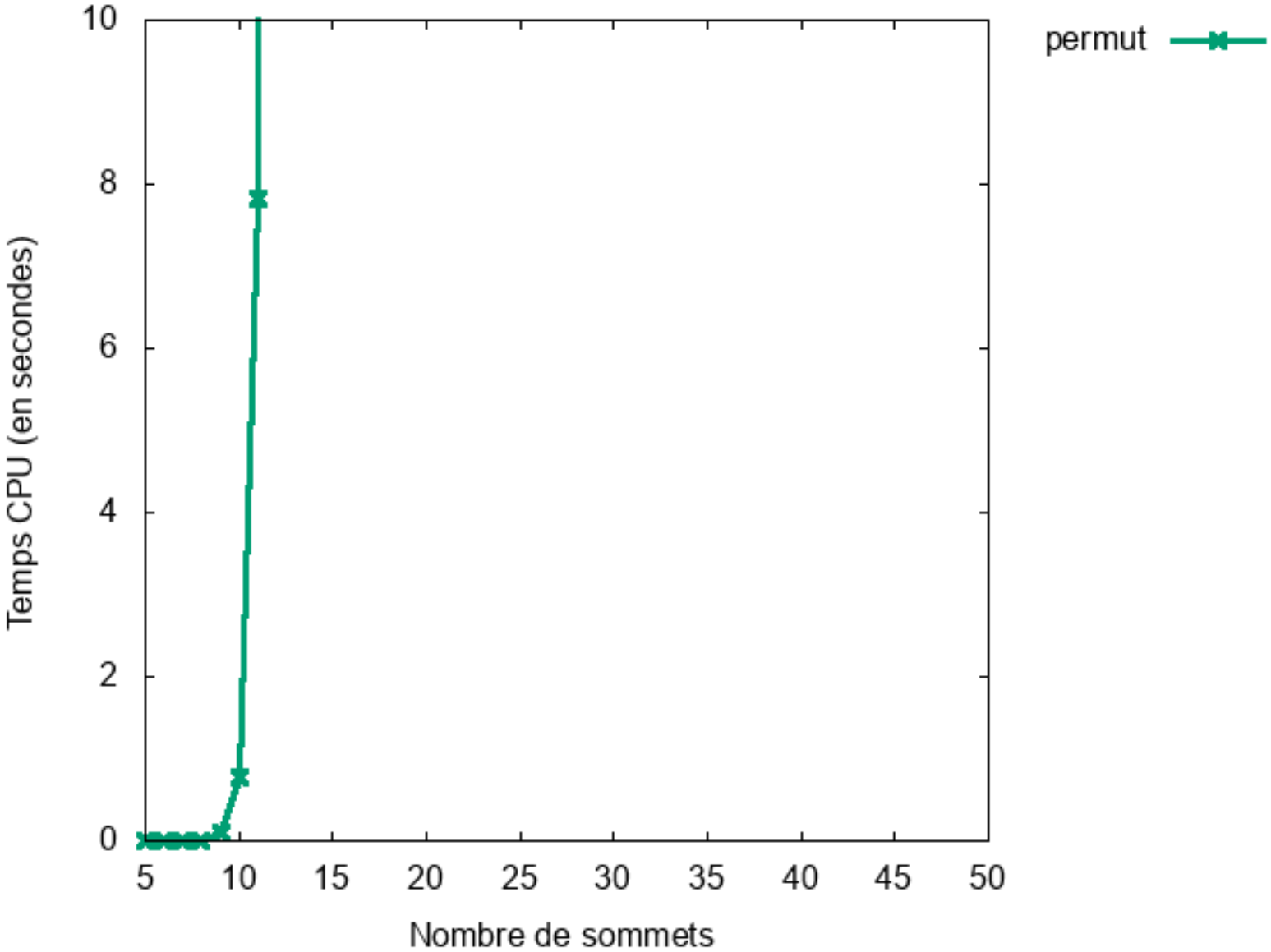


Illustration on the TSP

Stop: Prune the current branch if current length $\geq$ best

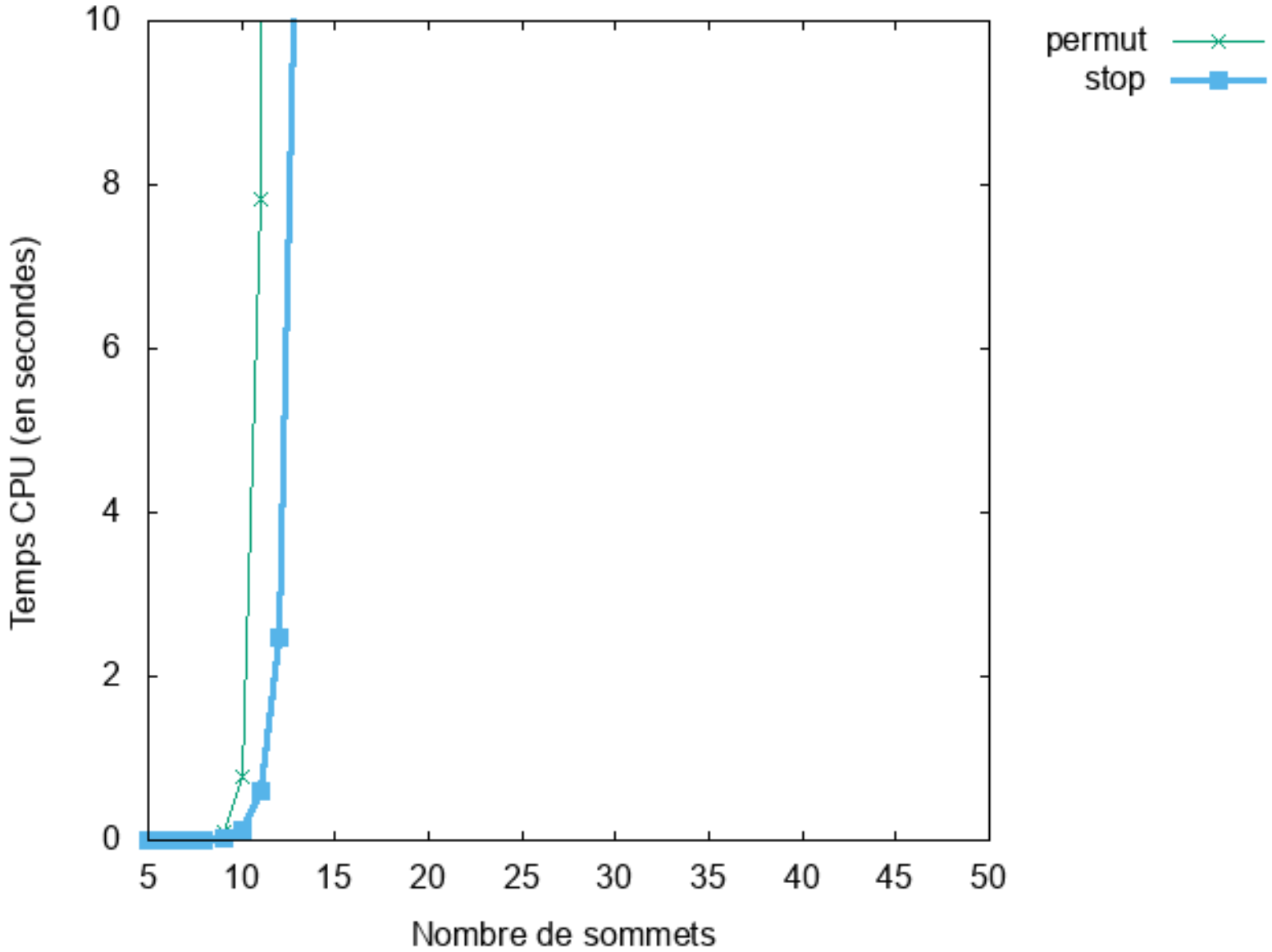


Illustration on the TSP

Branch & Bound 1: Bound = $d_{min} * nbNotVisited$

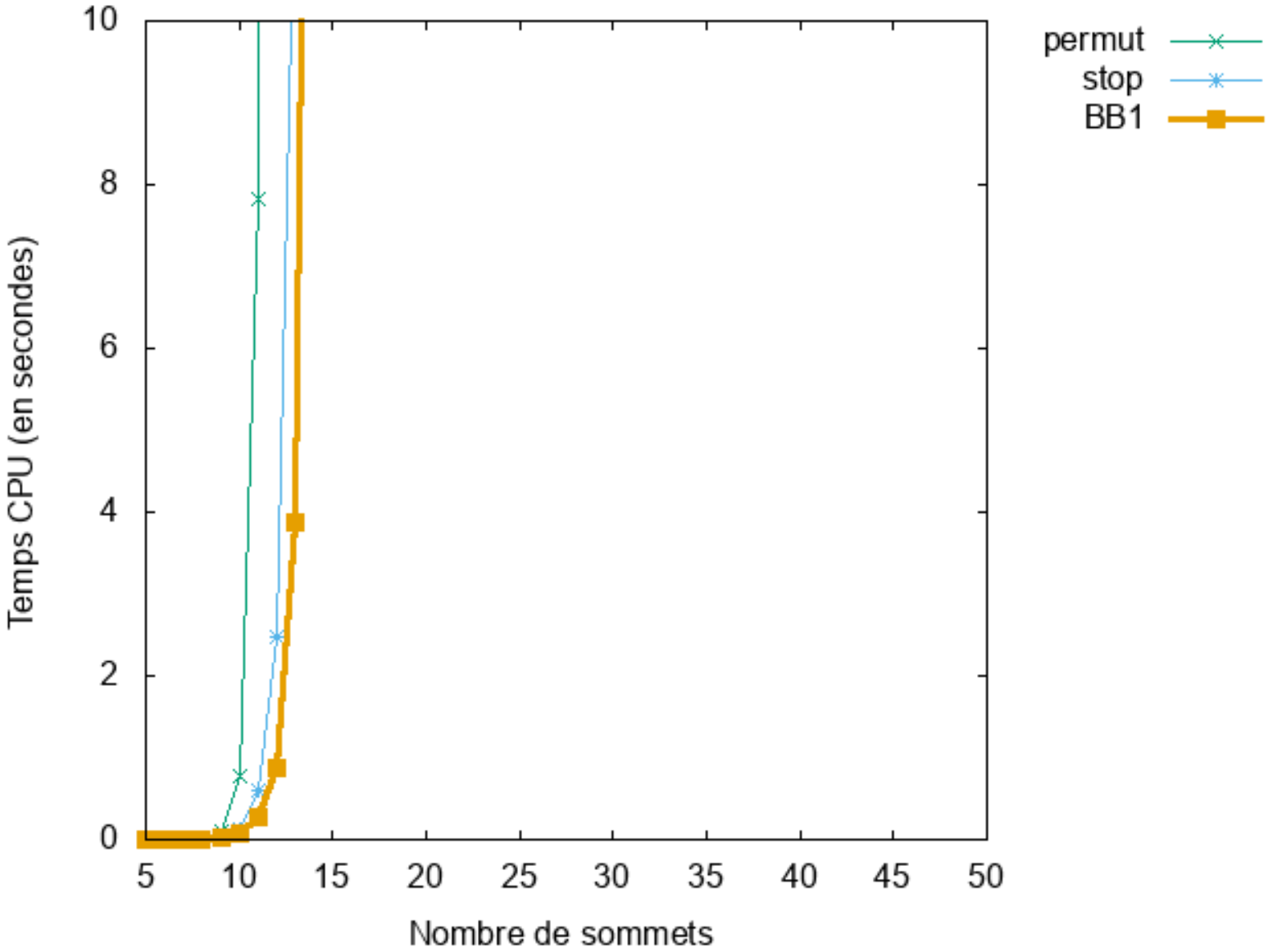


Illustration on the TSP

Branch & Bound 2: Bound = $\sum_{i \in notVisited} \min_{j \in notVisited, j \neq i} d_{ij}$

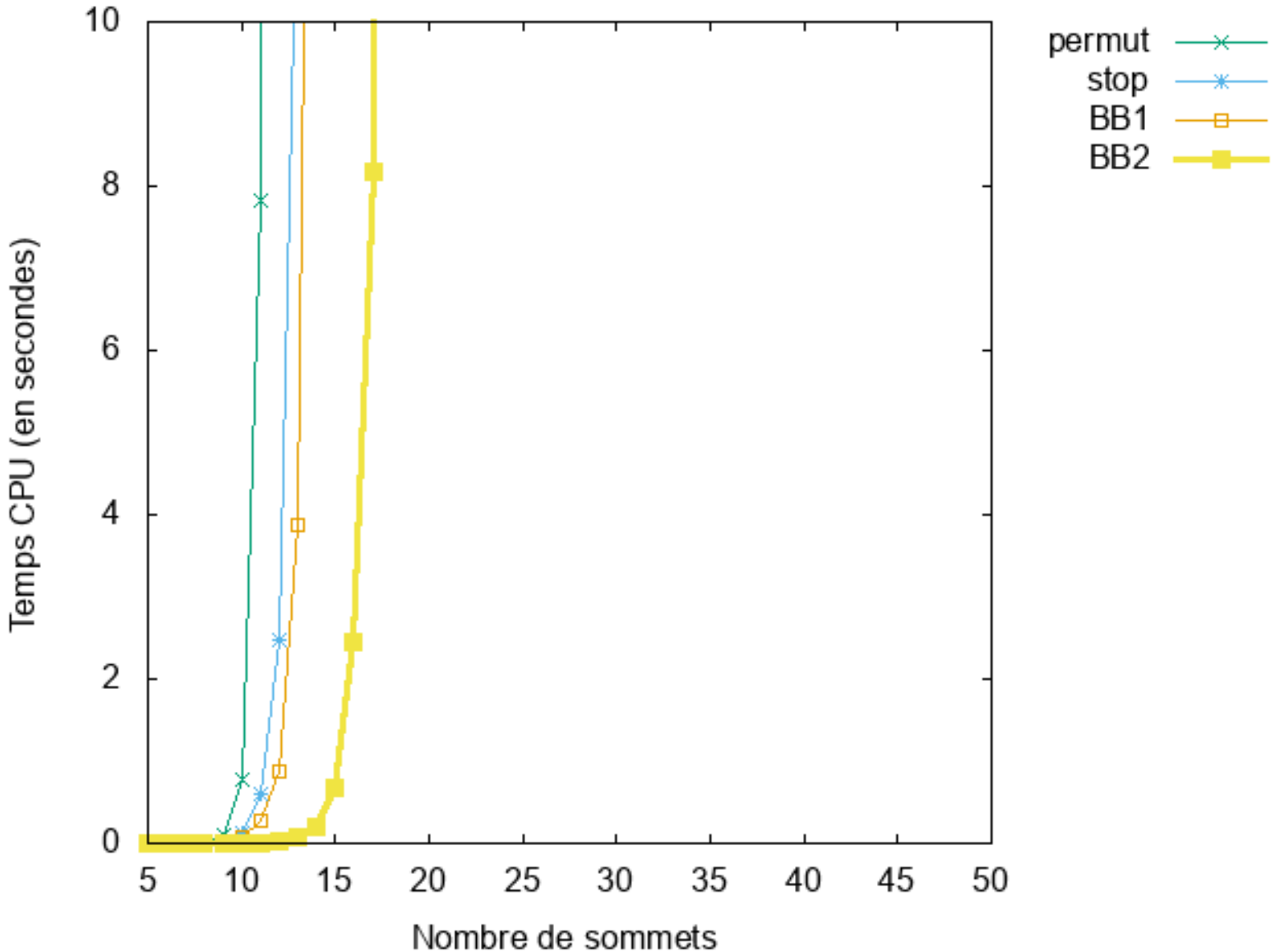


Illustration on the TSP

Branch & Bound 3: Bound = minimal 1-tree cost

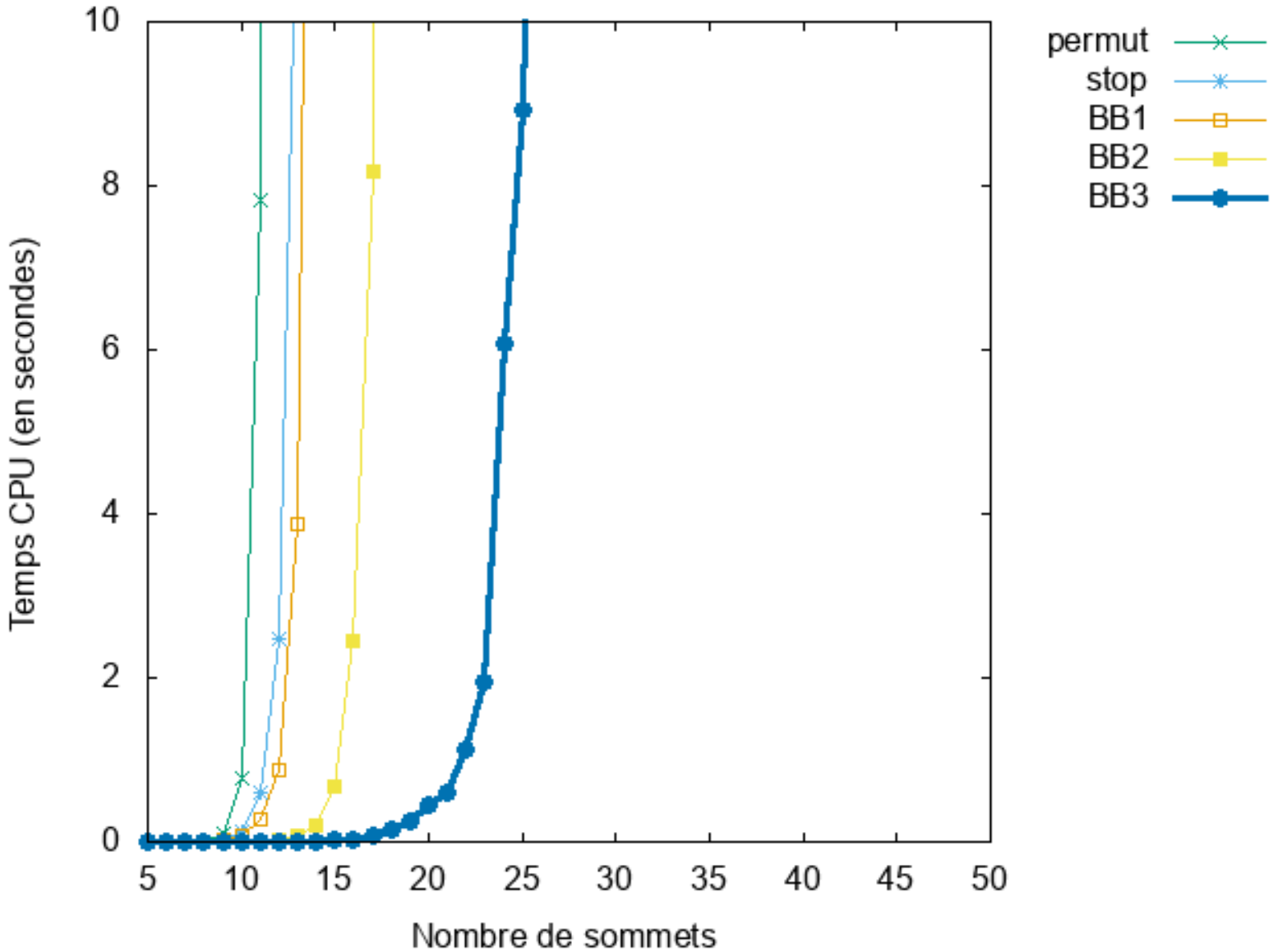


Illustration on the TSP

Branch & Bound 4: Bound = subgradian opt. of Held-Karp (iterated 1-tree)

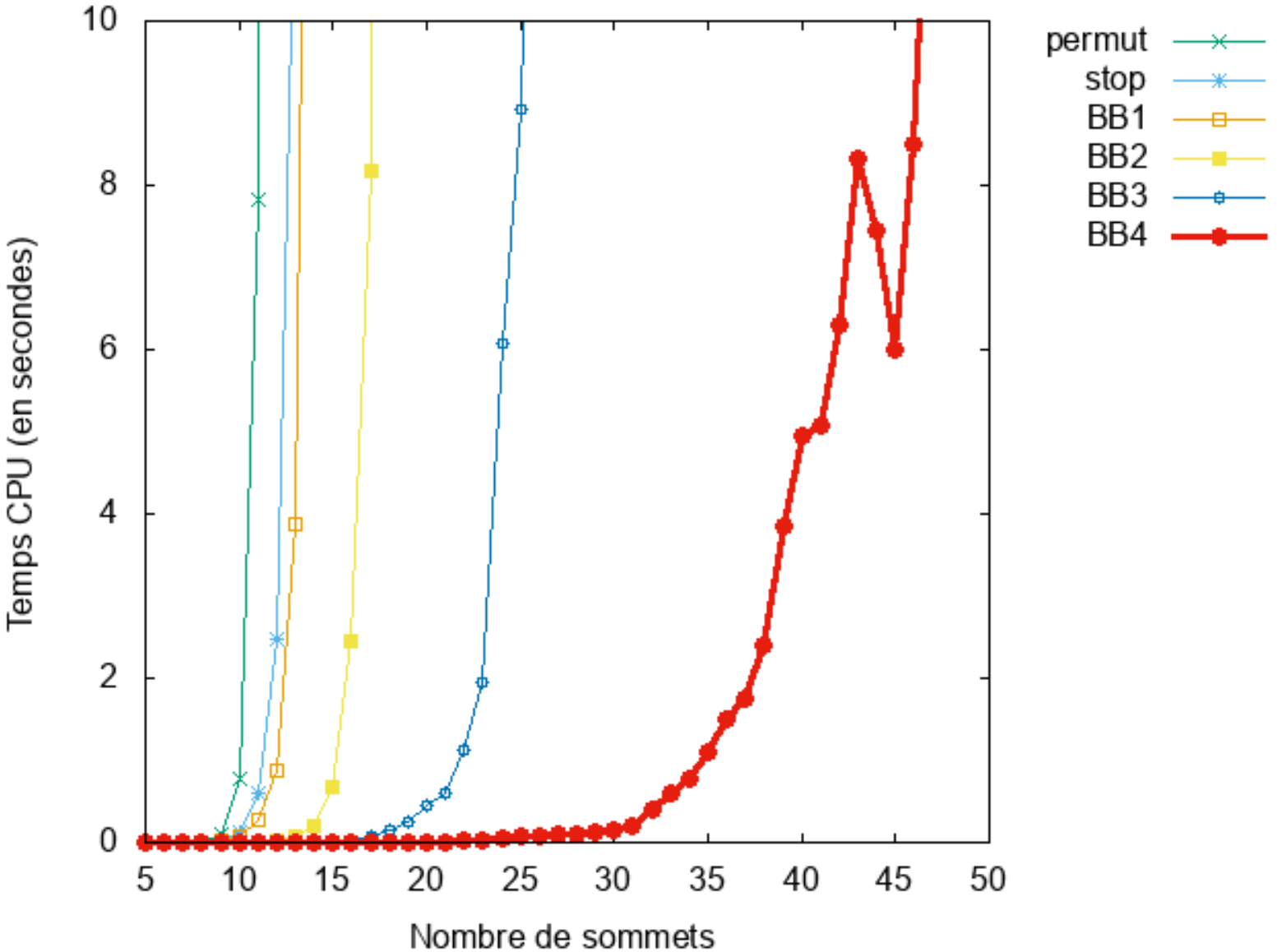


Illustration on the TSP

Branch & Bound 4 + h: Addition of an ordering heuristic

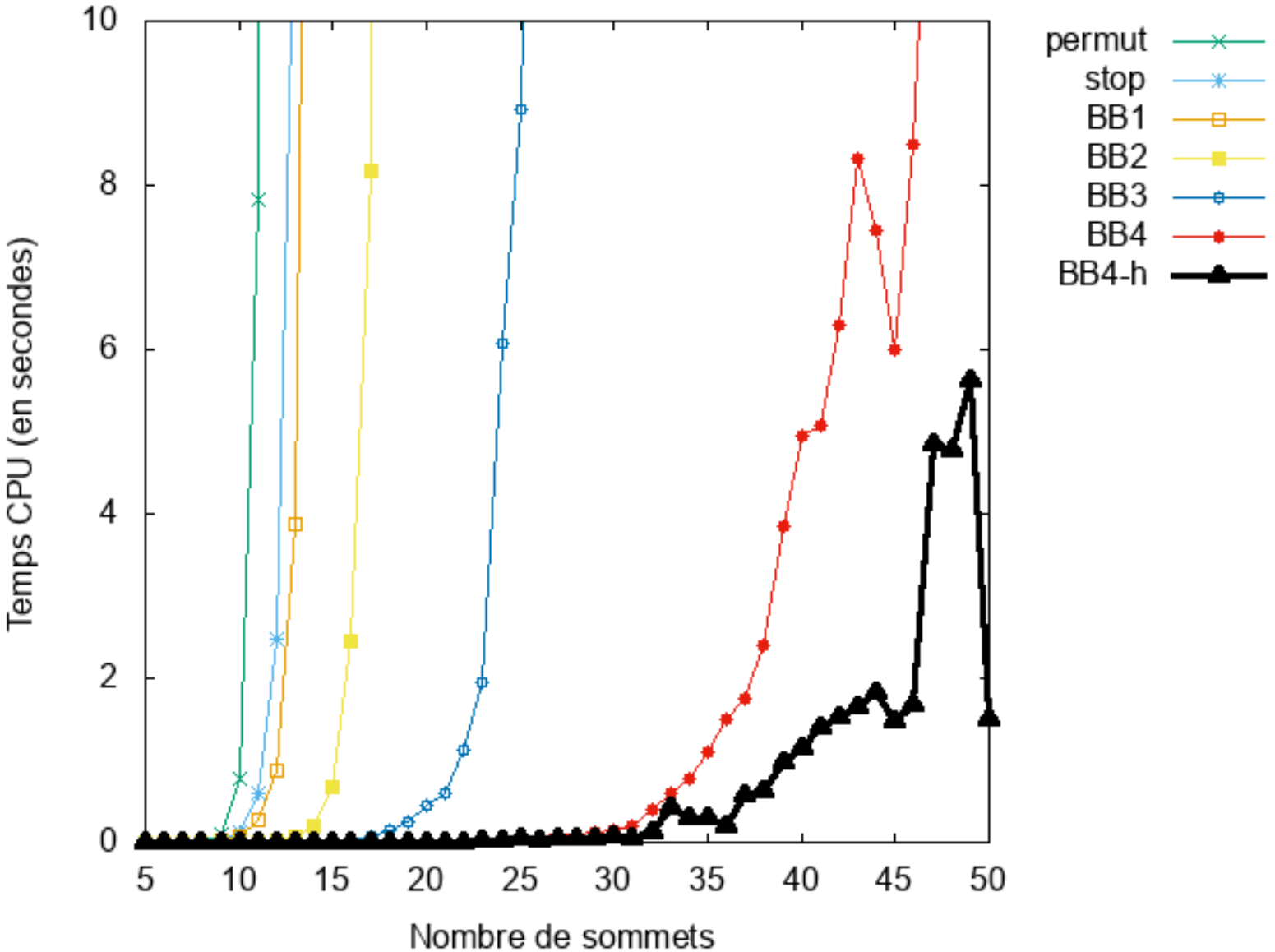


Illustration on the TSP

DPrec: Recursive Dynamic Programming $\rightsquigarrow \mathcal{O}(n^2 \cdot 2^n)$

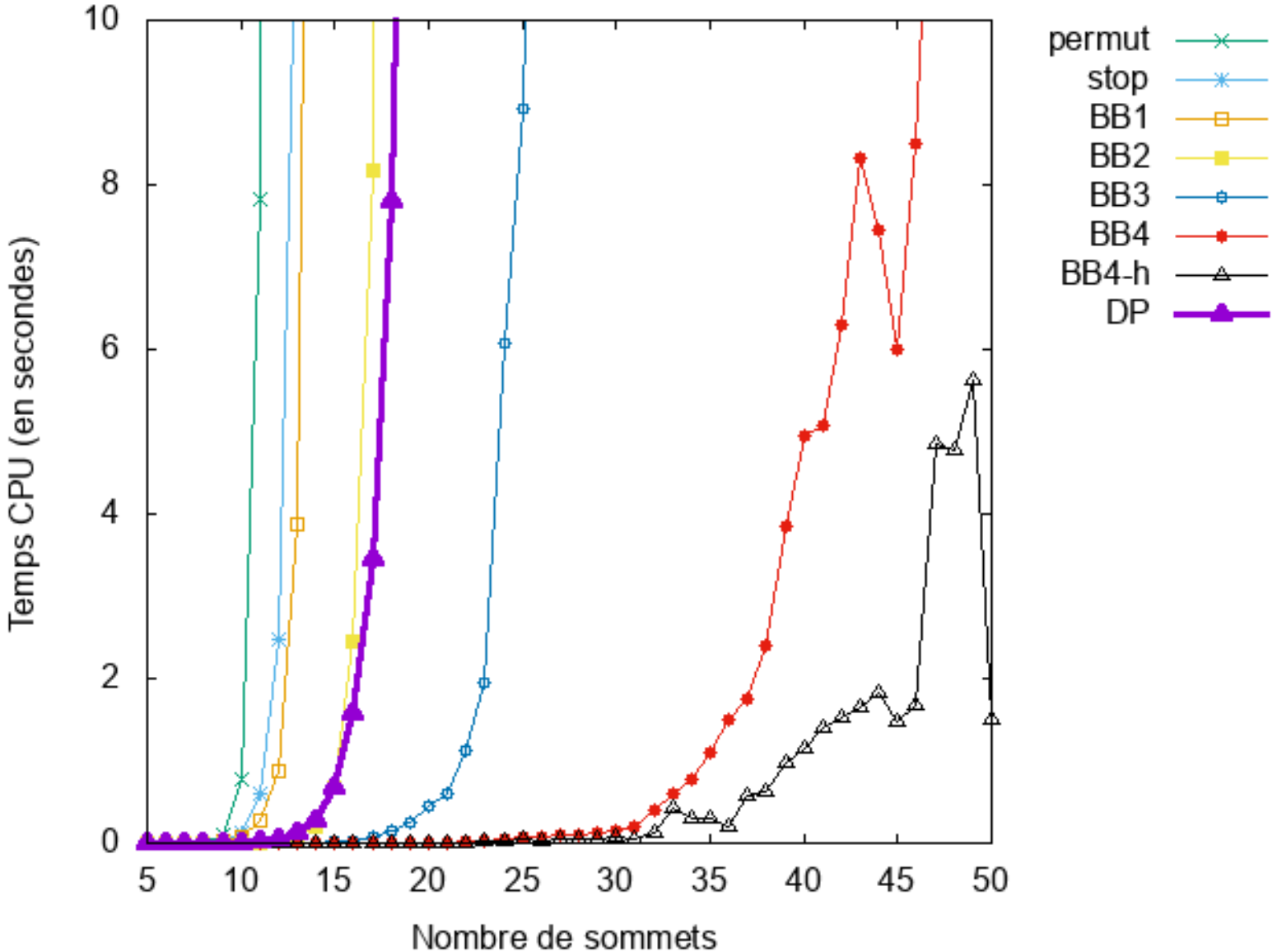


Illustration on the TSP

DPiter: Iterative Dynamic Programming $\rightsquigarrow \mathcal{O}(n^2 \cdot 2^n)$

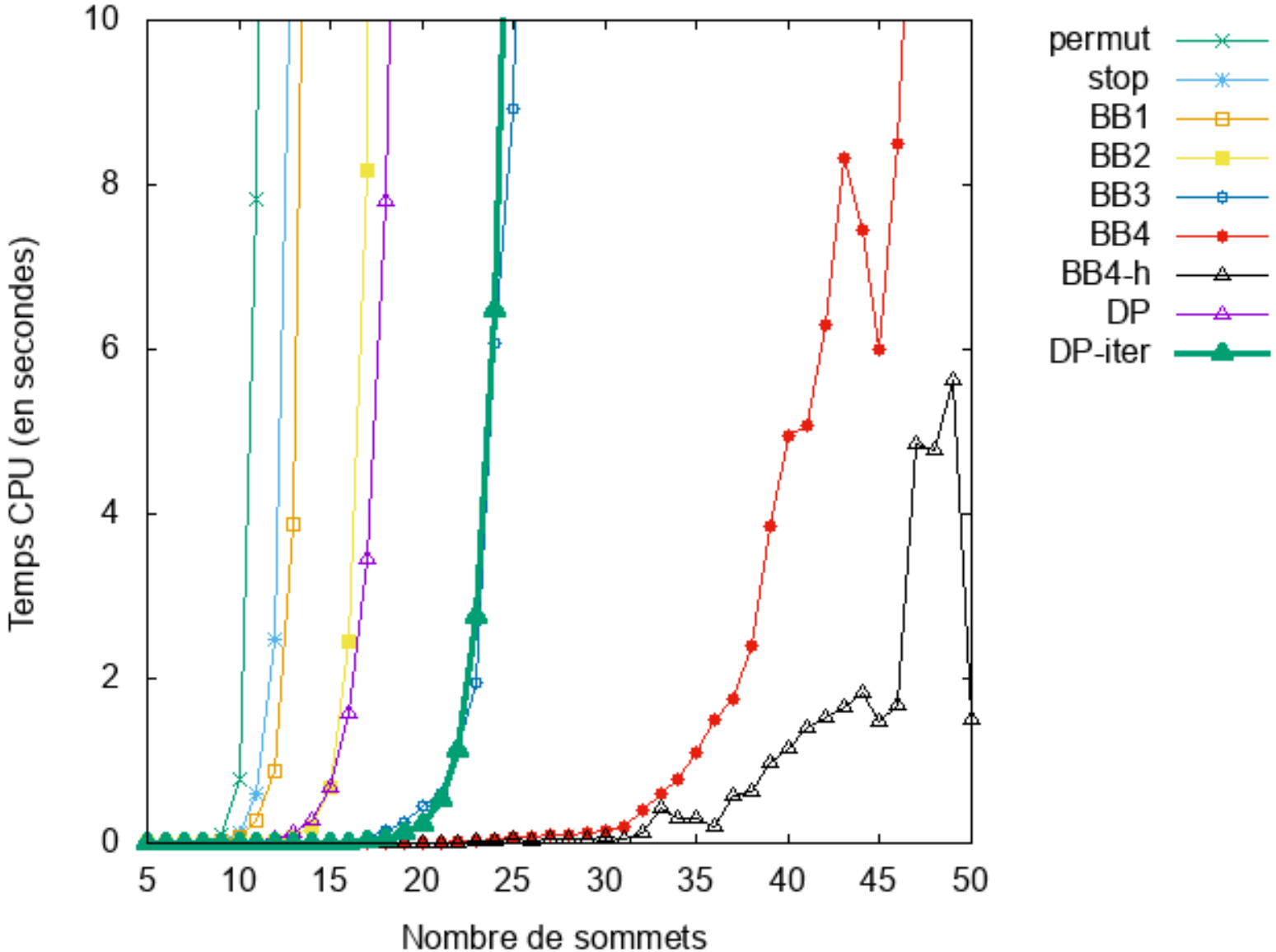


Illustration on the TSP

What if we change the benchmark?

Model used to generate graphs in the previous slide:

- Random generation of n coordinates $(x, y) \in [0, 1000]^2$
  $\rightsquigarrow$ Uniform distribution
- Edge cost = Euclidean distance (rounded to the closest integer value)

$\rightsquigarrow$ Experiments on 10 graphs (performance measure = average CPU time)

New model:

- For each edge: random generation of an integer cost $\in [10, 20]$
  $\rightsquigarrow$ Uniform distribution

$\rightsquigarrow$ Experiments on 10 graphs (performance measure = average CPU time)

Illustration on the TSP

What if we change the benchmark?

Model used to generate graphs in the previous slide:

- Random generation of n coordinates $(x, y) \in [0, 1000]^2$
  $\rightsquigarrow$ Uniform distribution
- Edge cost = Euclidean distance (rounded to the closest integer value)

$\rightsquigarrow$ Experiments on 10 graphs (performance measure = average CPU time)

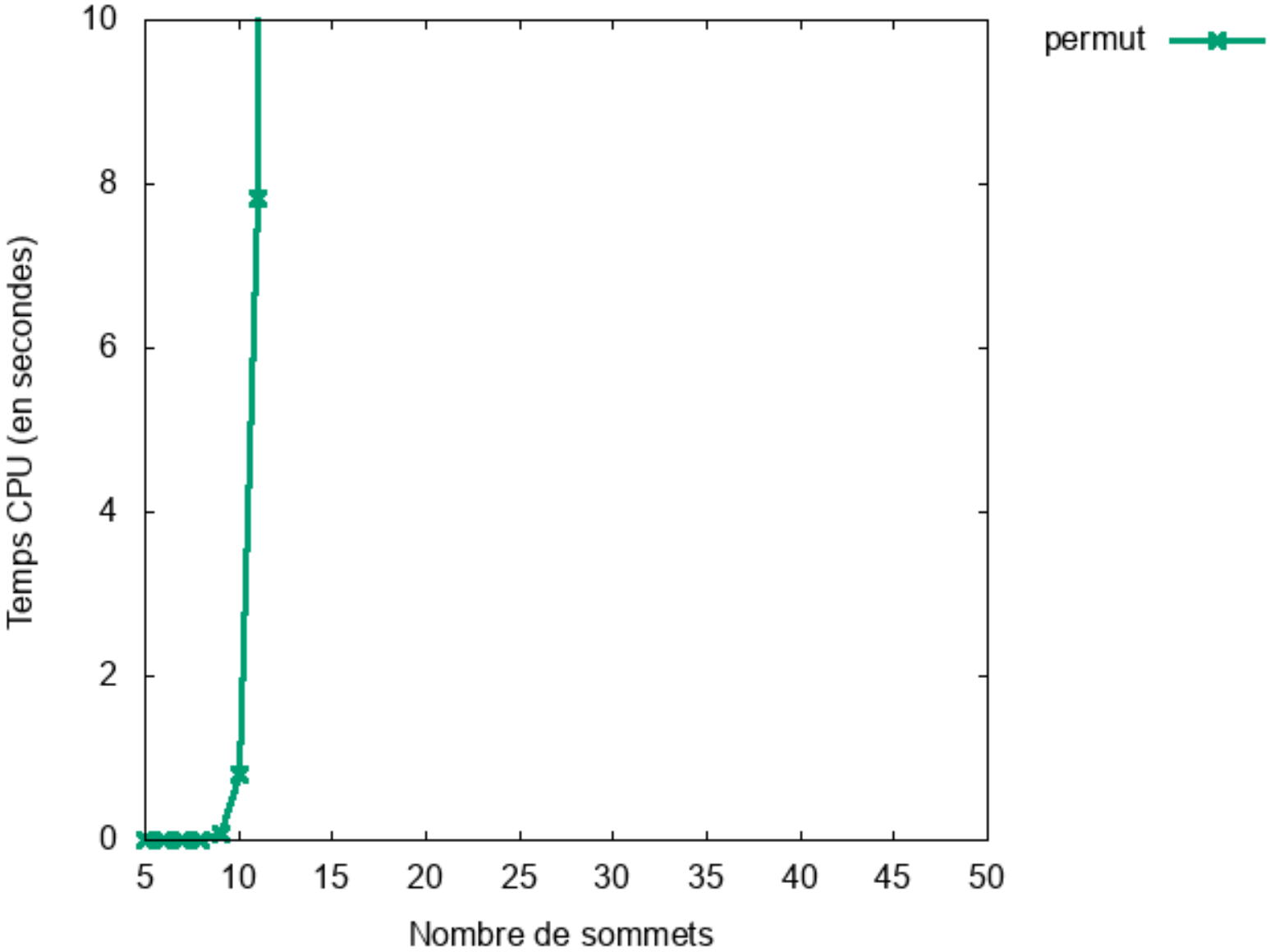
New model:

- For each edge: random generation of an integer cost $\in [10, 20]$
  $\rightsquigarrow$ Uniform distribution

$\rightsquigarrow$ Experiments on 10 graphs (performance measure = average CPU time)

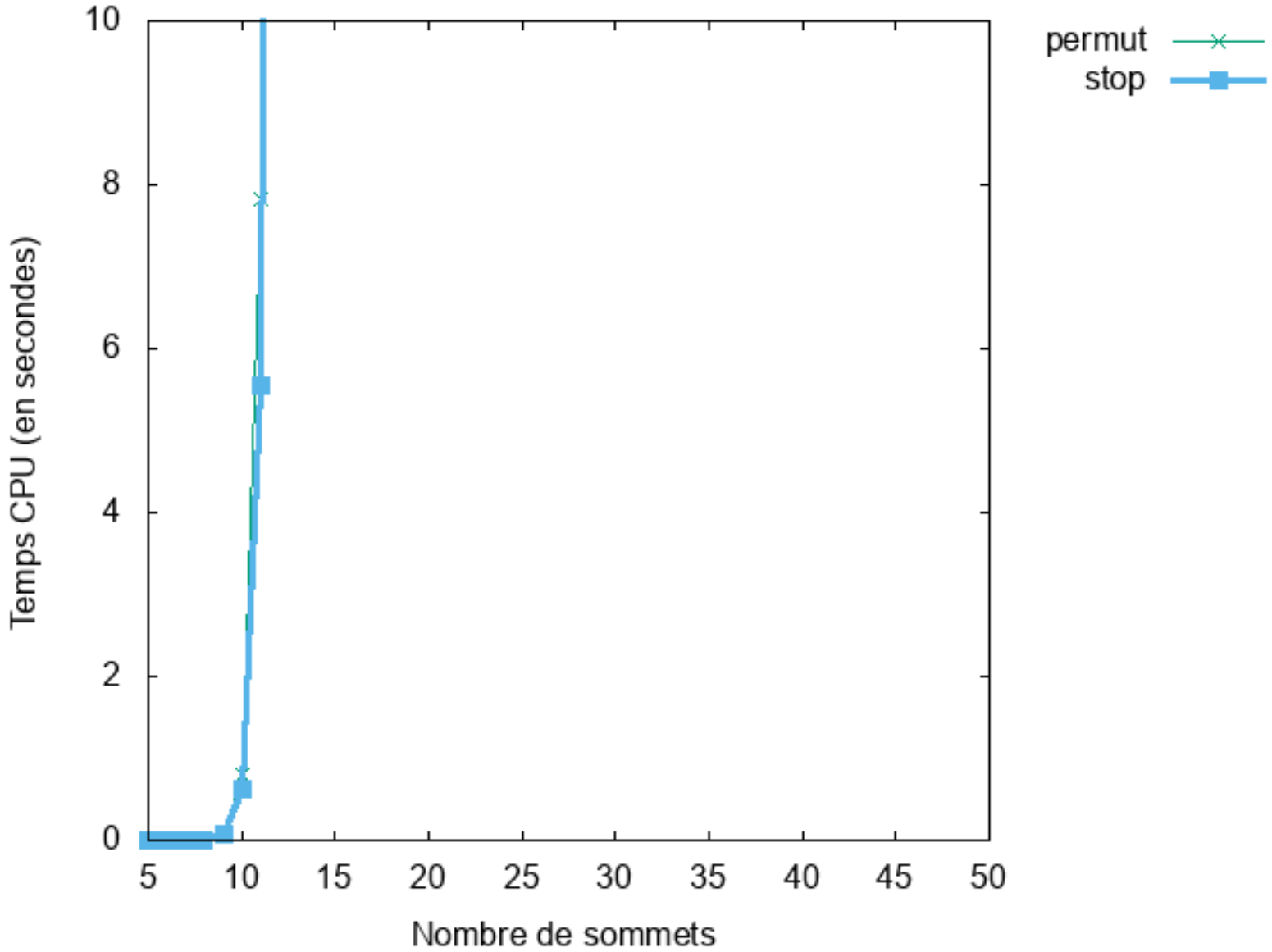
Results on Benchmark 2

Permut: Enumate all permutations $\rightsquigarrow \mathcal{O}(n!)$



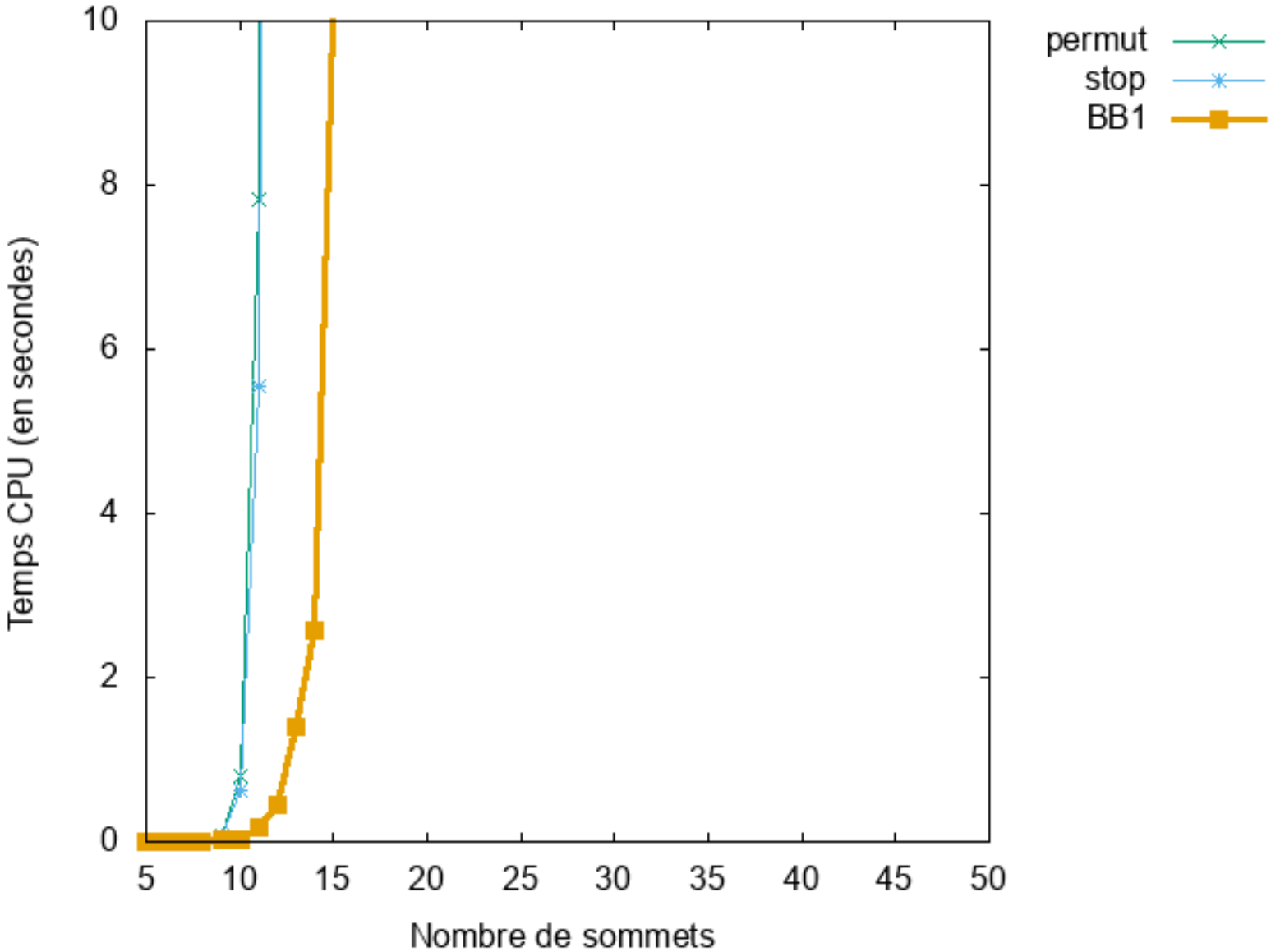
Results on Benchmark 2

Stop: Prune the current branch if current length $\geq$ best



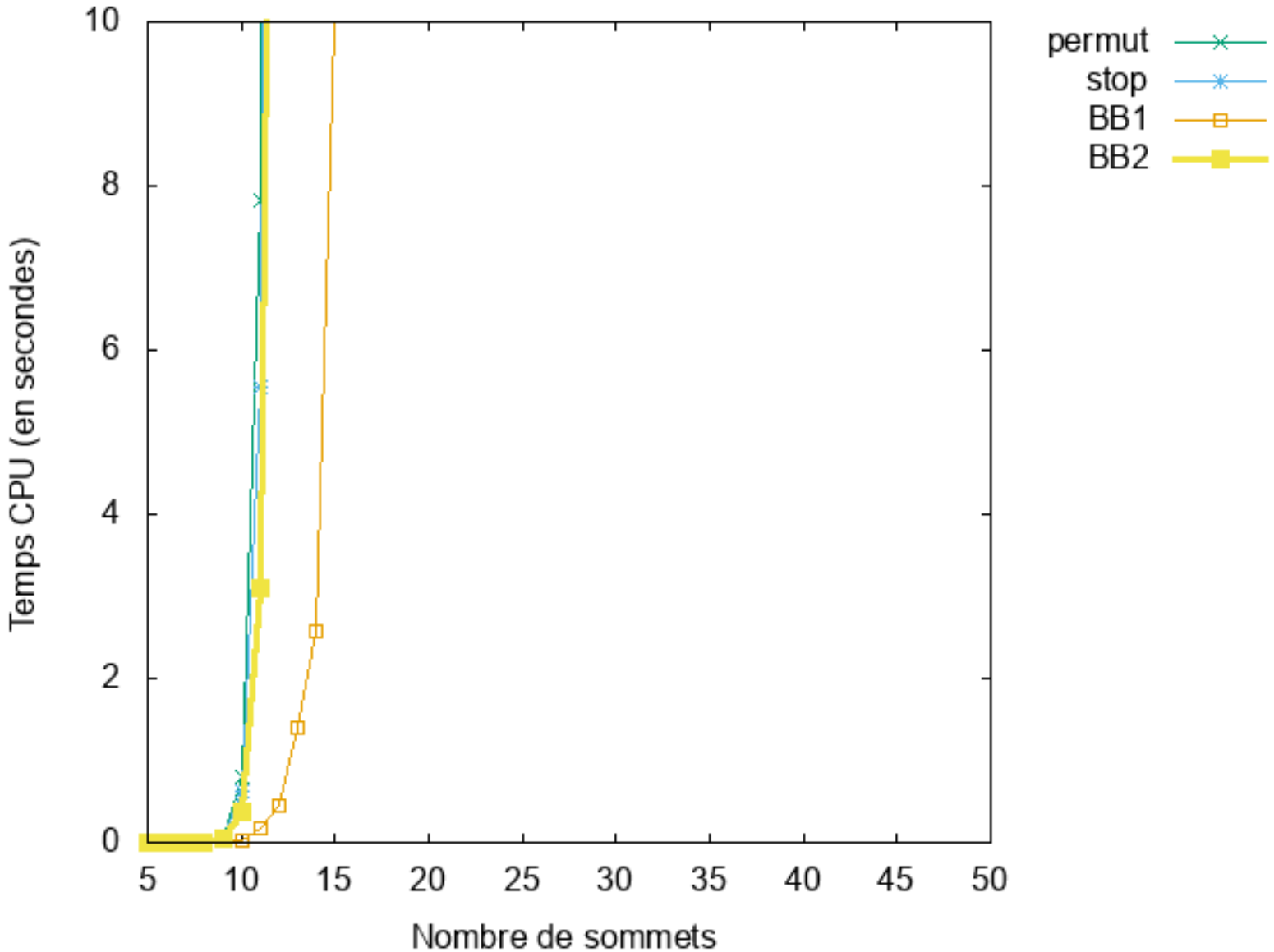
Results on Benchmark 2

Branch & Bound 1: Bound = $d_{min} * nbNotVisited$



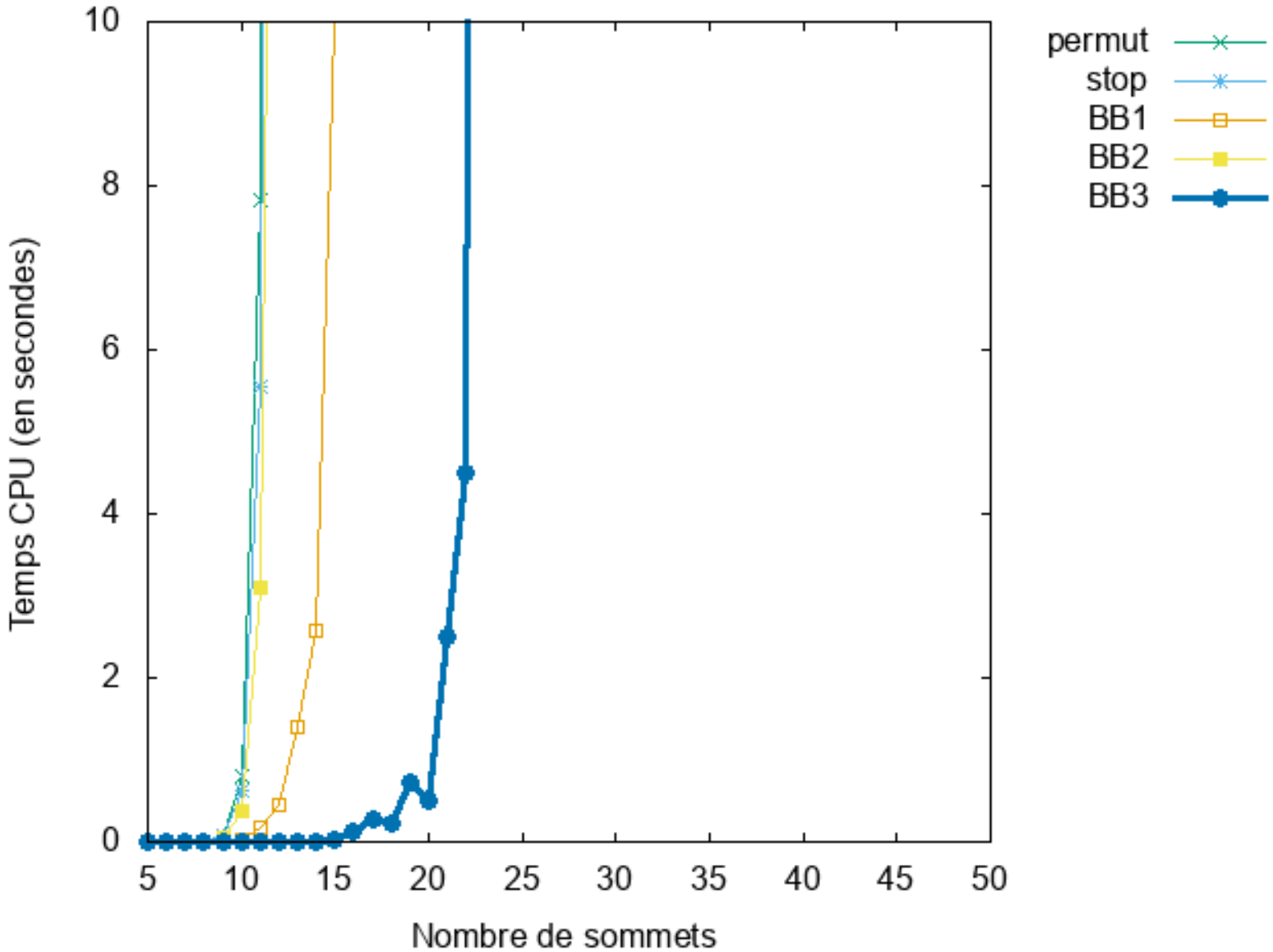
Results on Benchmark 2

Branch & Bound 2: Bound = $\sum_{i \in notVisited} \min_{j \in notVisited, j \neq i} d_{ij}$



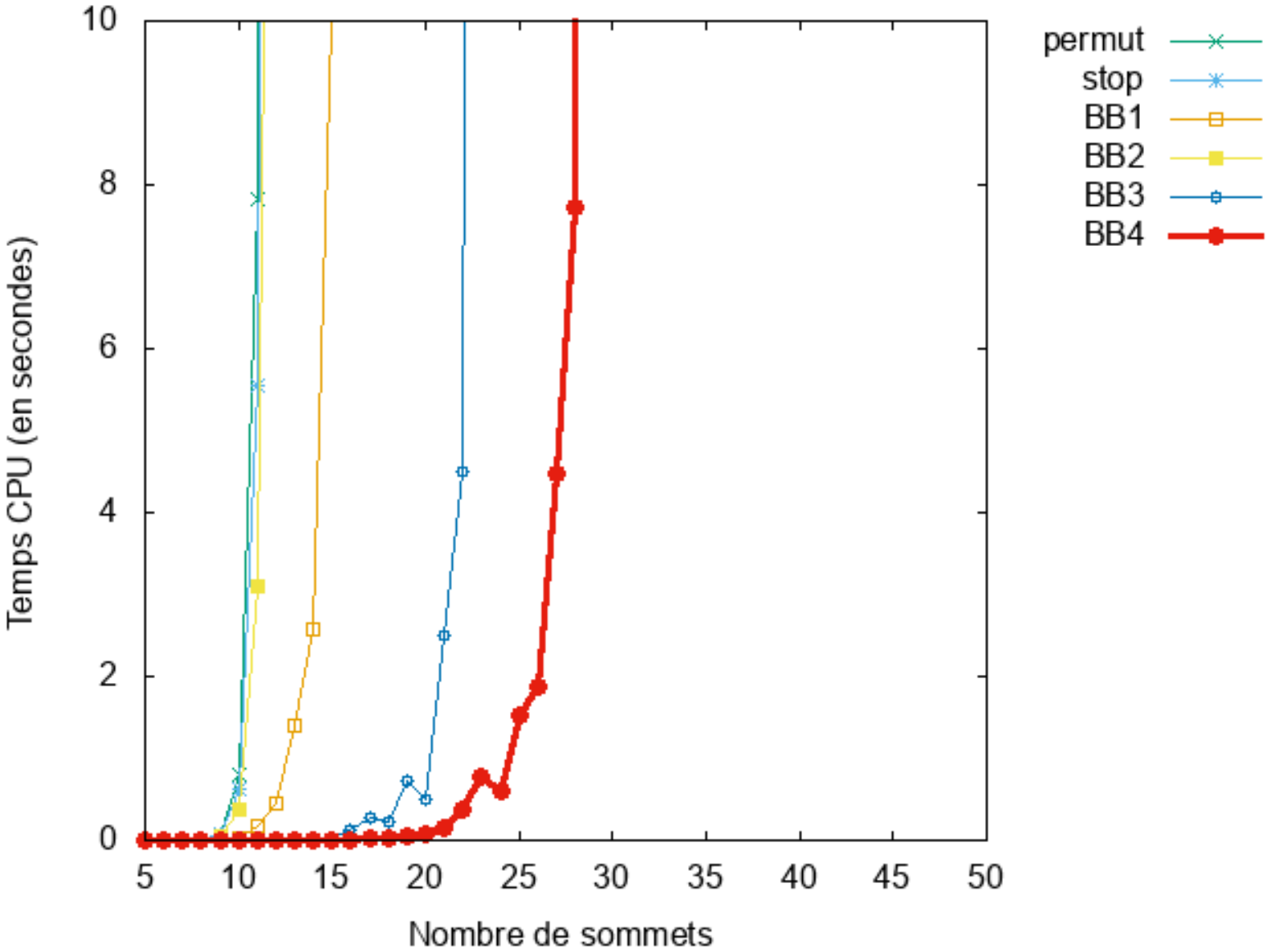
Results on Benchmark 2

Branch & Bound 3: Bound = minimal 1-tree cost



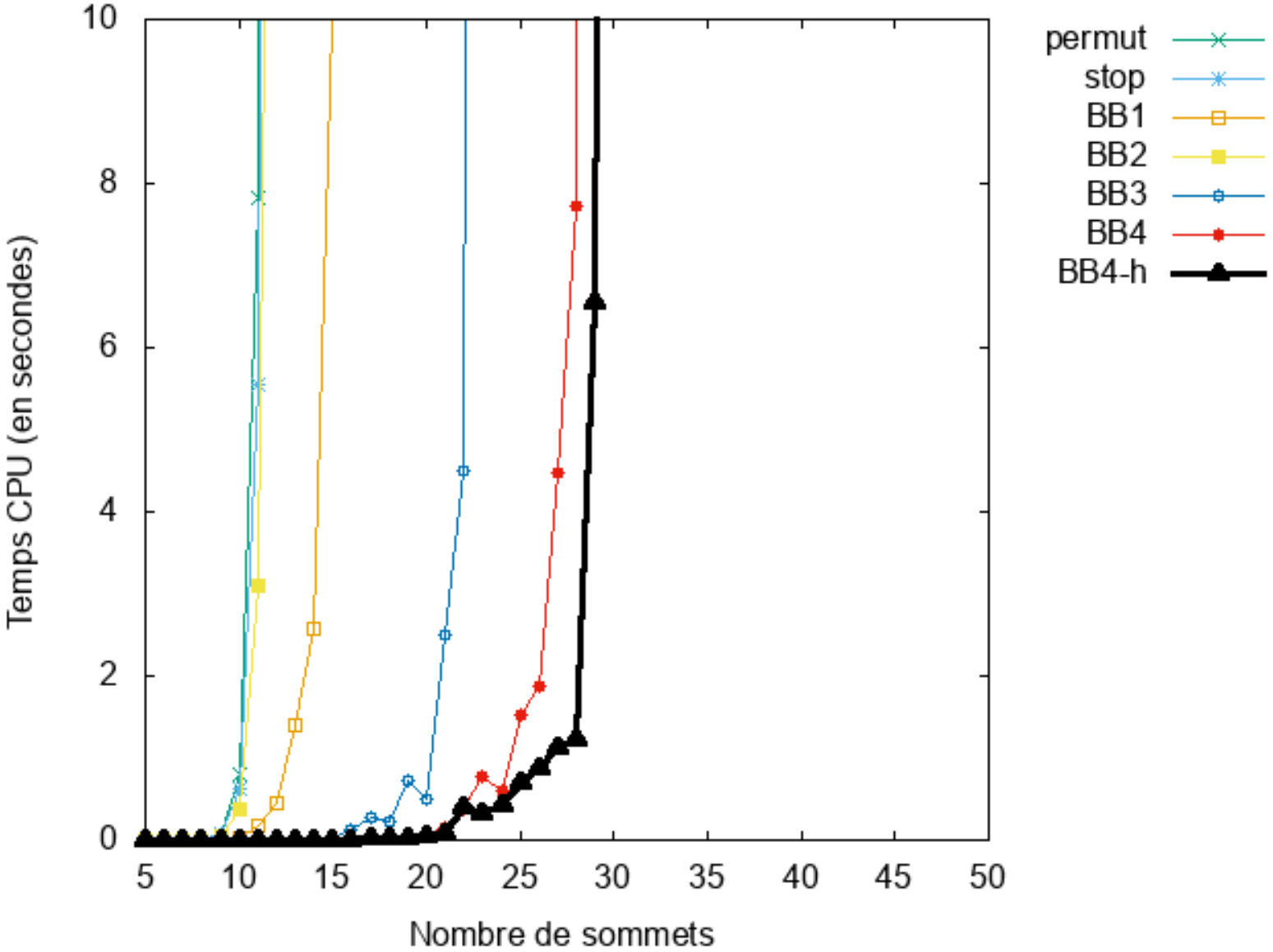
Results on Benchmark 2

Branch & Bound 4: Bound = subgradian opt. of Held-Karp (iterated 1-tree)



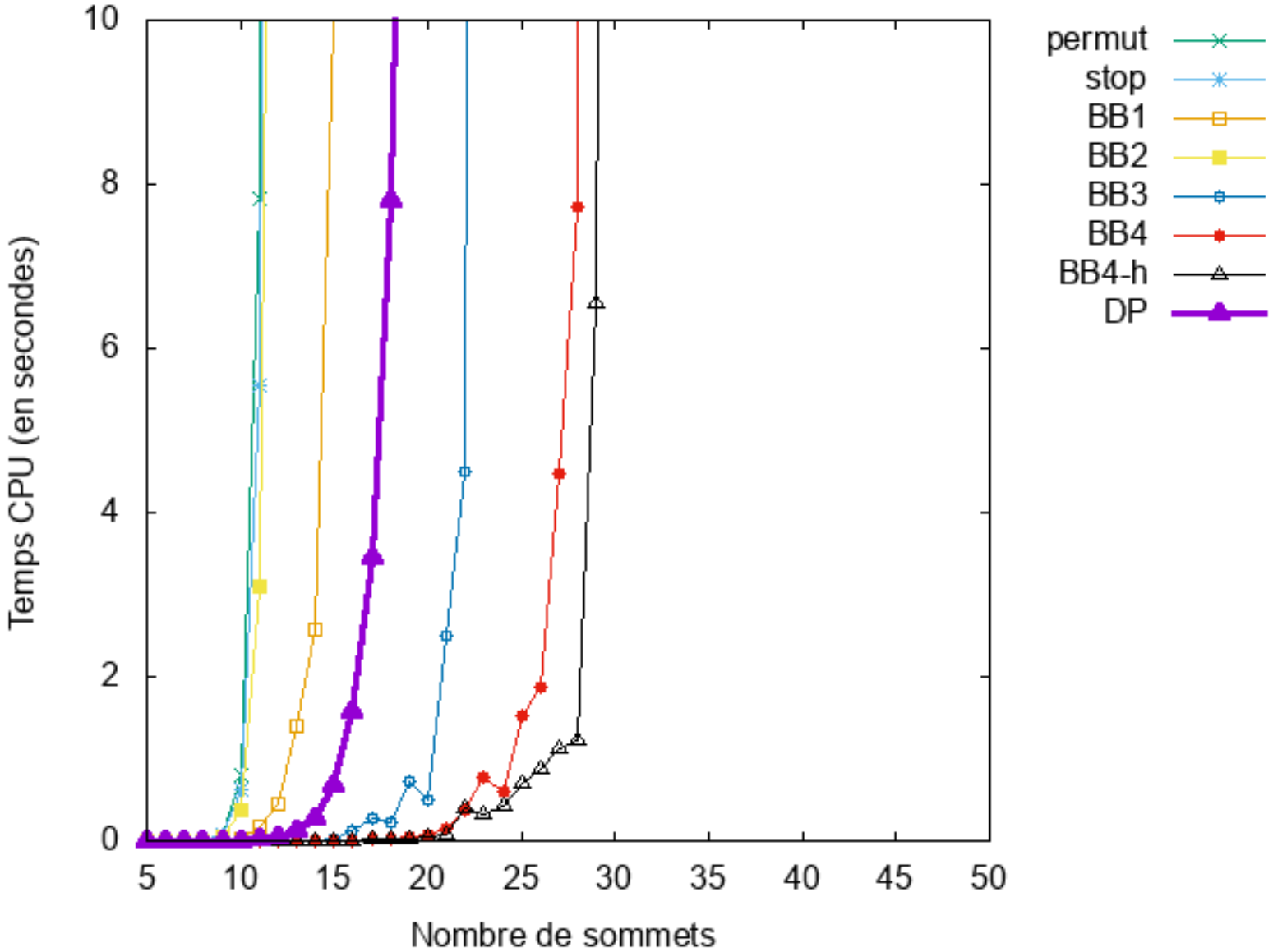
Results on Benchmark 2

Branch & Bound 4 + h: Addition of an ordering heuristic



Results on Benchmark 2

DPrec: Recursive Dynamic Programming $\rightsquigarrow \mathcal{O}(n^2 \cdot 2^n)$



Results on Benchmark 2

DPiter: Iterative Dynamic Programming $\rightsquigarrow \mathcal{O}(n^2 \cdot 2^n)$

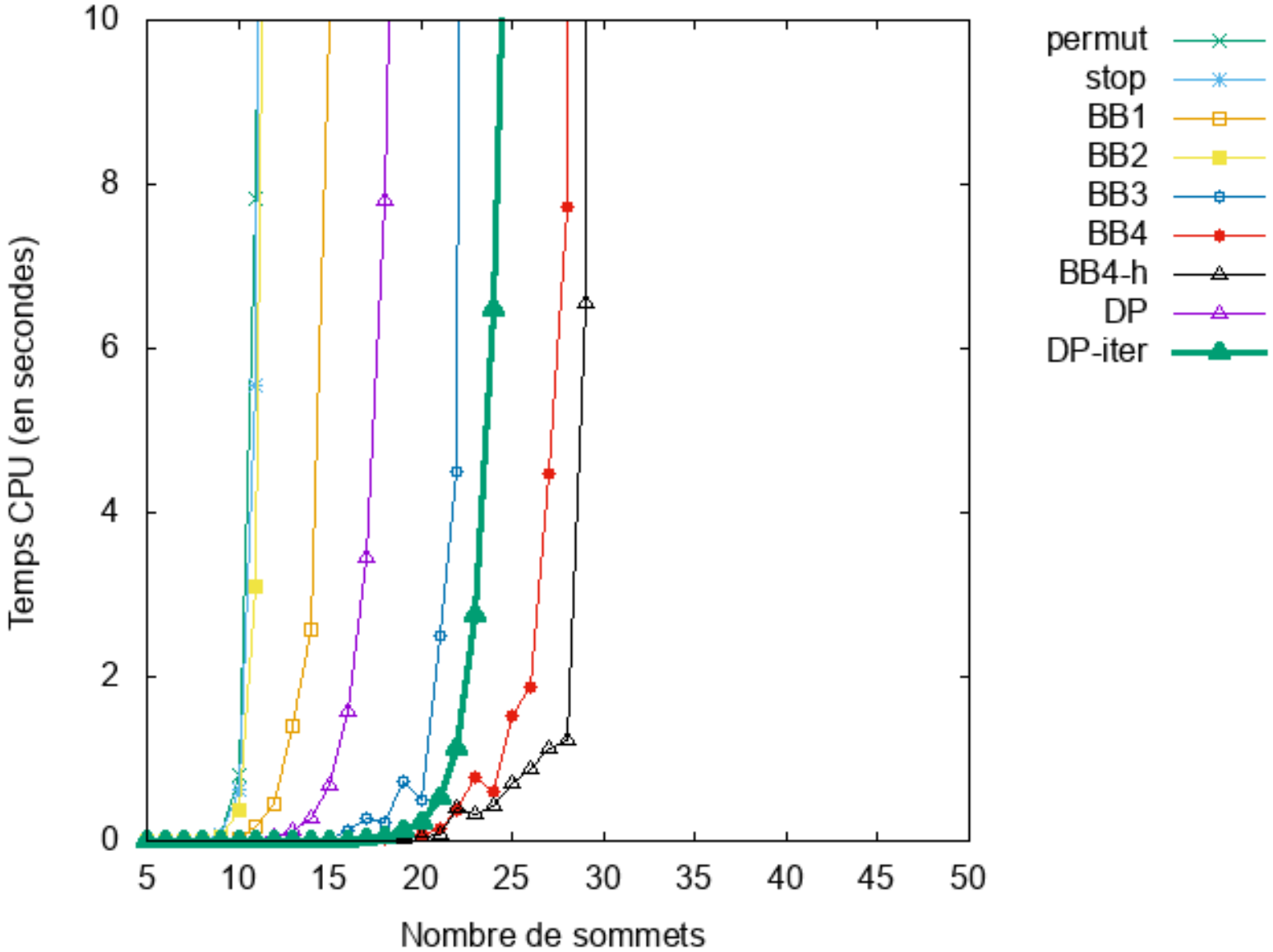
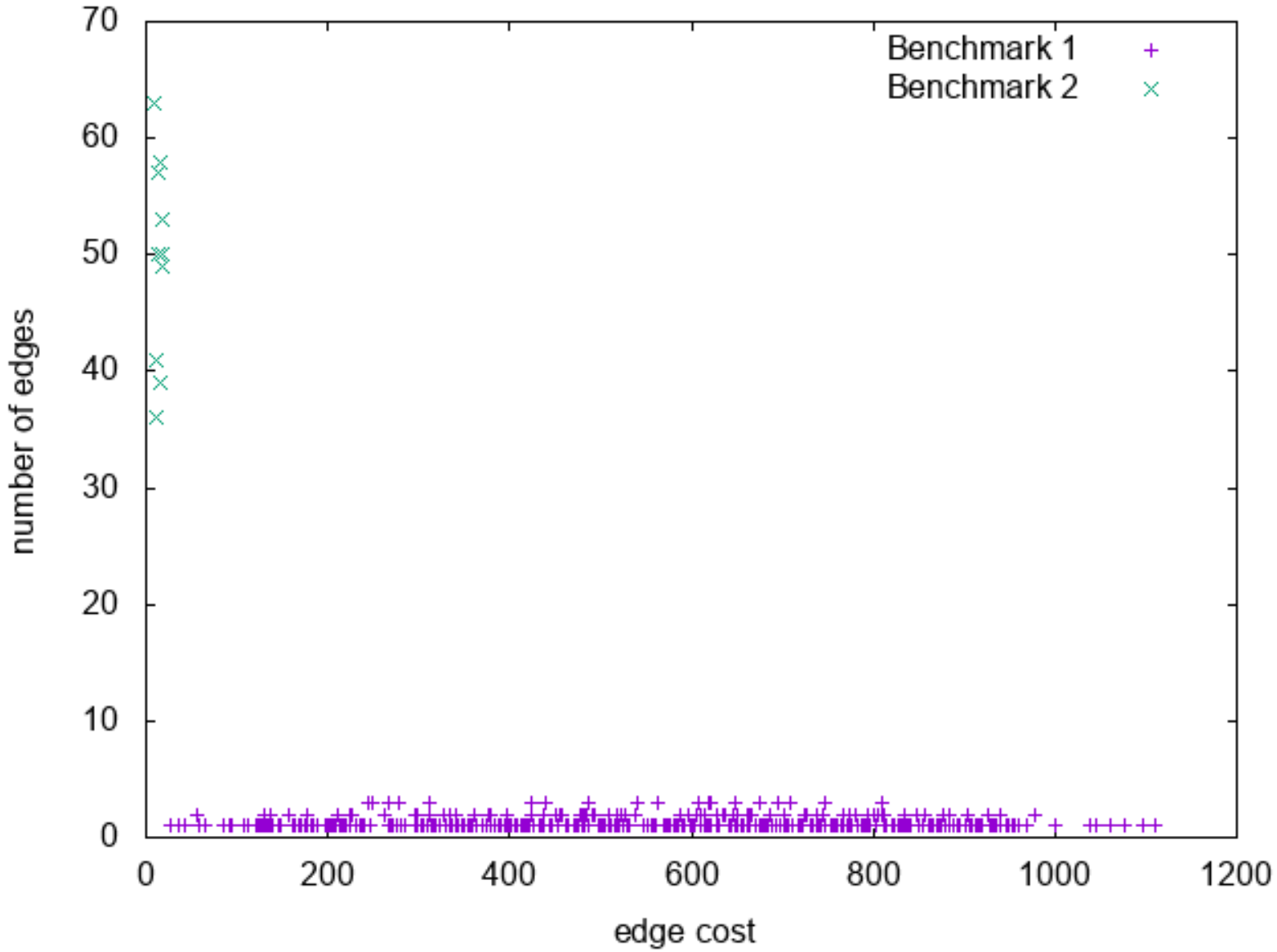


Illustration on the TSP

Comparison of Edge Cost Distributions on the 2 benchmarks



Plan

Code Tuning

Finer grain optimisation:

- Loops and procedures rather than algorithm paradigms
- Memory management rather than data structures

$\rightsquigarrow$ Small improvements... and loss of readability and generality!
$\rightsquigarrow$ Many of these optimisations are done by compilers (-O3 option of gcc)

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

D. Knuth

Example 1: Insertion Sort

```
for i ranging from 1 to n − 1 do
    /* Invariant:  tab[0..i − 1] is sorted                    */
    j ← i
    while j > 0 and tab[j] > tab[j − 1] do
        exchange(tab[j], tab[j − 1])
        j ← j − 1
```

Possible optimisations?

	n=40000	n=80000	n=160000
Initial code	0.65	2.66	10.59

Example 1: Insertion Sort

```
for i ranging from 1 to n − 1 do
    /* Invariant:  tab[0..i − 1] is sorted                    */
    j ← i; x ← tab[i]
    while j > 0 and tab[j] > tab[j − 1] do
        tab[j] ← tab[j − 1]
        j ← j − 1
    tab[j] ← x
```

Possible optimisations?

- Opt1: Memorise $tab[i]$ before entering the loop

	n=40000		n=80000		n=160000	
Initial code	0.65		2.66		10.59	
Opt1	0.40	-38%	1.61	-40%	6.43	-39%

Example 1: Insertion Sort

```
tab[0] ← −∞
for i ranging from 1 to n − 1 do
    /* Invariant:  tab[0..i − 1] is sorted                    */
    j ← i; x ← tab[i]
    while ~~j > 0 and~~ tab[j] > tab[j − 1] do
        tab[j] ← tab[j − 1]
        j ← j − 1
    tab[j] ← x
```

Possible optimisations?

- Opt1: Memorise $tab[i]$ before entering the loop
- Opt2: Add a sentinel value (assuming $tab[0]$ is not used)

	n=40000		n=80000		n=160000	
Initial code	0.65		2.66		10.59	
Opt1	0.40	-38%	1.61	-40%	6.43	-39%
Opt2	0.67		2.72		10.91	

Example 1: Insertion Sort

```
tab[0] ← −∞
for i ranging from 1 to n − 1 do
    /* Invariant:  tab[0..i − 1] is sorted                    */
    j ← i; x ← tab[i]
    while tab[j] > tab[j − 1] do
        tab[j] ← tab[j − 1]
        j ← j − 1
    tab[j] ← x
```

Possible optimisations?

- Opt1: Memorise $tab[i]$ before entering the loop
- Opt2: Add a sentinel value (assuming $tab[0]$ is not used)

	n=40000		n=80000		n=160000	
Initial code	0.65		2.66		10.59	
Opt1	0.40	-38%	1.61	-40%	6.43	-39%
Opt2	0.67		2.72		10.91	
Opt1+Opt2	0.30	-54%	1.22	-54%	4.88	-54%

Example 2: Enumerate all permutations of an array

```
permut(int* tab, int k, int n)
begin
    if k = n − 1 then display(tab, n);
    else
        for i ranging from k to n − 1 do
            exchange(tab[k], tab[i])
            permut(tab, k + 1, n)
            exchange(tab[k], tab[i])
```

Possible optimisations?

n	Initial code	Opt1	Opt2	Opt1+Opt2
11	0.89			
13	143.64			

Example 2: Enumerate all permutations of an array

```
permut(int* tab, int k, int n)
begin
    if k = n − 1 then display(tab, n);
    else
        for i ranging from k to n − 1 do
            exchange(tab[k], tab[i])
            permut(tab, k + 1, n)
            exchange(tab[k], tab[i])
```

Possible optimisations?

- Opt1: Unfold *exchange* (inlining procedure call)

n	Initial code	Opt1	Opt2	Opt1+Opt2
11	0.89	0.63		
13	143.64	97.60		

Example 2: Enumerate all permutations of an array

```
permut(int* tab, int k, int n)
begin
    if k = n − 1 then display(tab, n);
    else
        for i ranging from k to n − 1 do
            exchange(tab[k], tab[i])
            permut(tab, k + 1, n)
            exchange(tab[k], tab[i])
```

```
if k = n − 1 then display(tab, n);
else
    for i ranging from k to n − 1 do
        exchange(tab[k], tab[i])
        for j from k + 1 to n − 1 do
            exchange(tab[k + 1], tab[i])
            permut2(tab, k + 2, n)
            exchange(tab[k + 1], tab[i])
        exchange(tab[k], tab[i])
```

Possible optimisations?

- Opt1: Unfold *exchange* (inlining procedure call)
- Opt2: Divide by 2 the number of recursive calls (Assumption: n odd)

n	Initial code	Opt1	Opt2	Opt1+Opt2
11	0.89	0.63	0.82	
13	143.64	97.60	126.21	

Example 2: Enumerate all permutations of an array

```
permut(int* tab, int k, int n)
begin
    if k = n − 1 then display(tab, n);
    else
        for i ranging from k to n − 1 do
            exchange(tab[k], tab[i])
            permut(tab, k + 1, n)
            exchange(tab[k], tab[i])
```

```
if k = n − 1 then display(tab, n);
else
    for i ranging from k to n − 1 do
        exchange(tab[k], tab[i])
        for j from k + 1 to n − 1 do
            exchange(tab[k + 1], tab[i])
            permut2(tab, k + 2, n)
            exchange(tab[k + 1], tab[i])
        exchange(tab[k], tab[i])
```

Possible optimisations?

- Opt1: Unfold *exchange* (inlining procedure call)
- Opt2: Divide by 2 the number of recursive calls (Assumption: n odd)

n	Initial code	Opt1	Opt2	Opt1+Opt2
11	0.89	0.63	0.82	0.56
13	143.64	97.60	126.21	85.20

Example 2: Enumerate all permutations of an array

```
permut(int* tab, int k, int n)
begin
    if k = n − 1 then display(tab, n);
    else
        for i ranging from k to n − 1 do
            exchange(tab[k], tab[i])
            permut(tab, k + 1, n)
            exchange(tab[k], tab[i])
```

```
if k = n − 1 then display(tab, n);
else
    for i ranging from k to n − 1 do
        exchange(tab[k], tab[i])
        for j from k + 1 to n − 1 do
            exchange(tab[k + 1], tab[i])
            permut2(tab, k + 2, n)
            exchange(tab[k + 1], tab[i])
        exchange(tab[k], tab[i])
```

Possible optimisations?

- Opt1: Unfold *exchange* (inlining procedure call)
- Opt2: Divide by 2 the number of recursive calls (Assumption: n odd)

n	Initial code		Opt1		Opt2		Opt1+Opt2	
11	0.89	0.35	0.63	0.34	0.82	0.28	0.56	0.28
13	143.64	52.92	97.60	52.87	126.21	44.28	85.20	44.27

Results with the -O3 option of gcc

$\rightsquigarrow$ In many cases, we'd better let the compiler do optimisations!

Tools for Algorithm and Code Tuning

Profilers:

- gprof (gcc)
- Cachegrind and Callgrind (Valgrind)
- Instruments (Xcode)
- ...

Time spent in each function (percentage and absolute value)
$\rightsquigarrow$ Not always compatible with compiler optimisations!

Tools for the experimental evaluation and data analysis

Experimentally check that your optimisations actually optimise the program!

Illustration: Optimisation of *AntClique*

MaxClique Problem (recall)

- Input: a graph $G = (V, E)$
- Output: Largest subset $C \subseteq V$ such that $\forall \{i, j\} \subseteq C, \{i, j\} \in E$

AntClique:

- Incomplete algorithm: May find a sub-optimal solution
- Based on the Ant Colony Optimization (ACO) meta-heuristic
  $\leadsto$ Particular kind of reinforcement learning

Reference:

C. Solnon & S. Fenet: *A study of ACO capabilities for solving the Maximum Clique Problem*, Journal of Heuristics, 12(3):155-180, Springer, 2006

Basic Idea of AntClique

- initialize pheromone trails
- repeat
  1. each ant builds a clique
  2. update pheromone trails
- until optimal clique found or stagnation

Basic Idea of AntClique

- **initialize pheromone trails**
- repeat
  1. each ant builds a clique
  2. update pheromone trails
- until optimal clique found or stagnation

Pheromone is laid on edges:

$\rightsquigarrow \tau(i,j) =$ learned desirability of selecting both i and j in a same clique

Initialize $\tau(i,j)$ to τ_{max}, for each edge $\{i,j\} \in E$

$\rightsquigarrow \tau_{max}$ = parameter

Basic Idea of AntClique

- initialize pheromone trails
- repeat
  1. **each ant builds a clique**
  2. update pheromone trails
- until optimal clique found or stagnation

Greedy randomized construction of a clique $\mathcal{C}$

- Randomly choose $i \in V$ and initialize $\mathcal{C}$ to $\{i\}$
- While $cand = \{j \in V \setminus \mathcal{C} : \forall i \in \mathcal{C}, \{i,j\} \in E\} \neq \emptyset$:
  - Select randomly a vertex $v_j \in cand$ wrt probability

$$p(v_j) = \frac{[\sum_{i \in \mathcal{C}} \tau(i,j)]^\alpha}{\sum_{k \in cand} [\sum_{i \in \mathcal{C}} \tau(i,k)]^\alpha}$$

    where α = pheromone weight (parameter)
  - Add v_j to $\mathcal{C}$
- Return $\mathcal{C}$

Basic Idea of AntClique

- initialize pheromone trails
- repeat
  1. each ant builds a clique
  2. **update pheromone trails**
- until optimal clique found or stagnation

Pheromone updating step

- Evaporation: multiply pheromone trails by $(1 - \rho)$
  $\rightsquigarrow$ ρ = evaporation rate ($0 \leq \rho \leq 1$)
- Reward: add pheromone on all edges of the best clique
- Bound all pheromone trails to prevent early stagnation:
  - If $\tau(i,j) < \tau_{min}$ then $\tau(i,j) \leftarrow \tau_{min}$
  - If $\tau(i,j) > \tau_{max}$ then $\tau(i,j) \leftarrow \tau_{max}$

- Profiling of `main`

60650.0ms	99.9%	1,0	▼main essai
58457.0ms	96.3%	22,0	▶buildClique essai
2122.0ms	3.4%	2122,0	updatePheromoneTrails essai
69.0ms	0.1%	6,0	▶createGraph essai
1.0ms	0.0%	1,0	initPhero essai

- 96.3% of the time spent in `buildClique`
  ⇝ Zoom on `buildClique`

```
clique[0] = getNextRand(G->nbVertices);
cliqueSize = 1;
nbCandidates = selectCandidates(cliqueSize, clique, candidates, G);
while (nbCandidates>0){
    computeProba(nbCandidates, candidates, cliqueSize, clique, alpha, G, p);
    clique[cliqueSize++] = candidates[chooseNextVertex(p, nbCandidates)];
    nbCandidates = selectCandidates(cliqueSize, clique, candidates, G);
}
```

- Profiling of `buildClique`

58780.0ms	96.3%	12,0	▼buildClique essai
45996.0ms	75.4%	17453,0	▶selectCandidates essai
12311.0ms	20.1%	5984,0	▶computeProba essai
218.0ms	0.3%	198,0	▶chooseNextVertex essai

- 75.4% of the time spent in `selectCandidates`
  ⇝ Opt1: Incrementally maintain the candidate list

- Code of `buildClique2`:

```
clique[0] = getNextRand(G->nbVertices);
cliqueSize = 1;
nbCandidates = G->degree[clique[0]];
for (i=0; i<nbCandidates; i++) candidates[i] = G->succ[clique[0]][i];
while (nbCandidates>0){
    computeProba(nbCandidates, candidates, cliqueSize, clique, alpha, G, p);
    v = candidates[chooseNextVertex(p, nbCandidates)];
    clique[cliqueSize++] = v;
    for (i=0; i<nbCandidates; i++){
        if (!isEdge(v, candidates[i], G)){
            candidates[i] = candidates[--nbCandidates];
            i--;
        }
    }
}
```

- Profiling of `buildClique2`

13407.0ms	85.7%	463,0		▼buildClique2 essai
12080.0ms	77.2%	5893,0		▶computeProba essai
604.0ms	3.8%	604,0		isEdge essai
221.0ms	1.4%	211,0		▶chooseNextVertex essai

- 77.2% of the time spent in `computeProba`
  $\rightsquigarrow$ Opt2: Incrementally maintain the pheromone factor

- Code of `buildClique3`:

```
clique[0] = getNextRand(G->nbVertices);
cliqueSize = 1;
nbCandidates = G->degree[clique[0]];
for (i=0; i<nbCandidates; i++){
    candidates[i] = G->succ[clique[0]][i];
    tauClique[candidates[i]] = (G->tauE)[clique[0]][candidates[i]];
}
while (nbCandidates>0){
    computeProba3(tauClique, nbCandidates, candidates, cliqueSize, clique, alpha, G, p);
    v = candidates[chooseNextVertex(p, nbCandidates)];
    clique[cliqueSize++] = v;
    for (i=0; i<nbCandidates; i++){
        if (!isEdge(v, candidates[i], G)){
            candidates[i] = candidates[--nbCandidates];
            i--;
        }
        else tauClique[candidates[i]] += (G->tauE)[v][candidates[i]];
    }
}
```

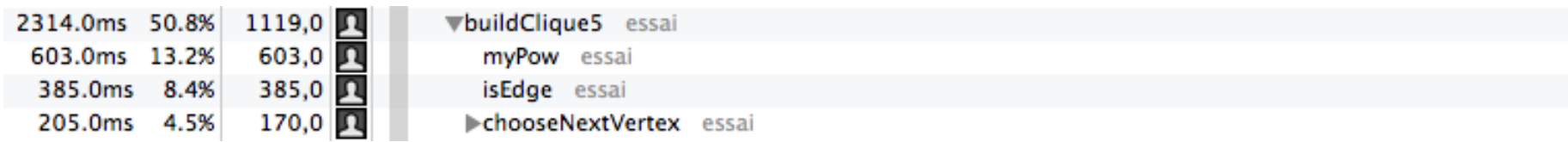
- Profiling of `buildClique3`

8614.0ms	79.3%	642,0	▼buildClique3 essai
7080.0ms	65.1%	810,0	▼computeProba3 essai
6270.0ms	57.7%	6270,0	0x7fff88363e00 libsystem_m.dylib
617.0ms	5.6%	617,0	isEdge essai
237.0ms	2.1%	215,0	▶chooseNextVertex essai

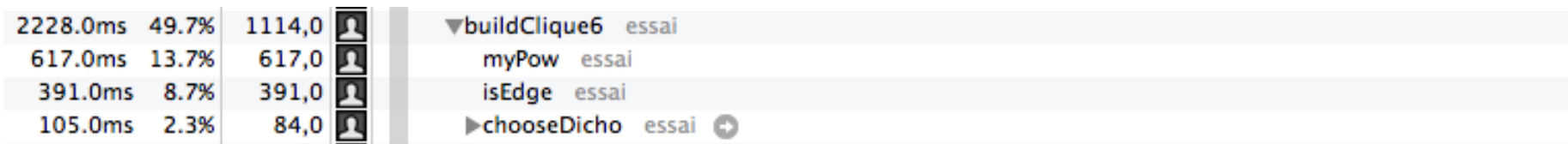
- 57.7% of the time spent in `pow` (from `math.h`)
  $\rightsquigarrow$ Opt3: replace `pow` with `myPow` !

- Profiling of `buildClique4`

- 20.2% of the time spent in `computeProba4`
  $\rightsquigarrow$ Opt4: merge the loop that computes proba. with candidate filtering
- Profiling of `buildClique5`



- A last optimisation?
  $\rightsquigarrow$ Opt5: replace the sequential search of `chooseNextVertex` with a dichotomous search
- Profiling of `buildClique6`



Gains are getting smaller and smaller . . . Is it still useful?

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	
C125.9	3.61	
C250.9	8.34	
C500.9	23.53	
C1000.9	111.30	
C2000.9	347.15	

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	Opt1	
C125.9	3.61	1.79	
C250.9	8.34	3.65	
C500.9	23.53	8.49	
C1000.9	111.30	28.94	
C2000.9	347.15	88.18	

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	Opt1	Opt2	
C125.9	3.61	1.79	1.29	
C250.9	8.34	3.65	2.78	
C500.9	23.53	8.49	5.84	
C1000.9	111.30	28.94	13.52	
C2000.9	347.15	88.18	27.32	

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	Opt1	Opt2	Opt3	
C125.9	3.61	1.79	1.29	0.32	
C250.9	8.34	3.65	2.78	0.71	
C500.9	23.53	8.49	5.84	1.59	
C1000.9	111.30	28.94	13.52	4.06	
C2000.9	347.15	88.18	27.32	10.64	

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	Opt1	Opt2	Opt3	Opt4	
C125.9	3.61	1.79	1.29	0.32	0.24	
C250.9	8.34	3.65	2.78	0.71	0.51	
C500.9	23.53	8.49	5.84	1.59	1.21	
C1000.9	111.30	28.94	13.52	4.06	3.32	
C2000.9	347.15	88.18	27.32	10.64	9.14	

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

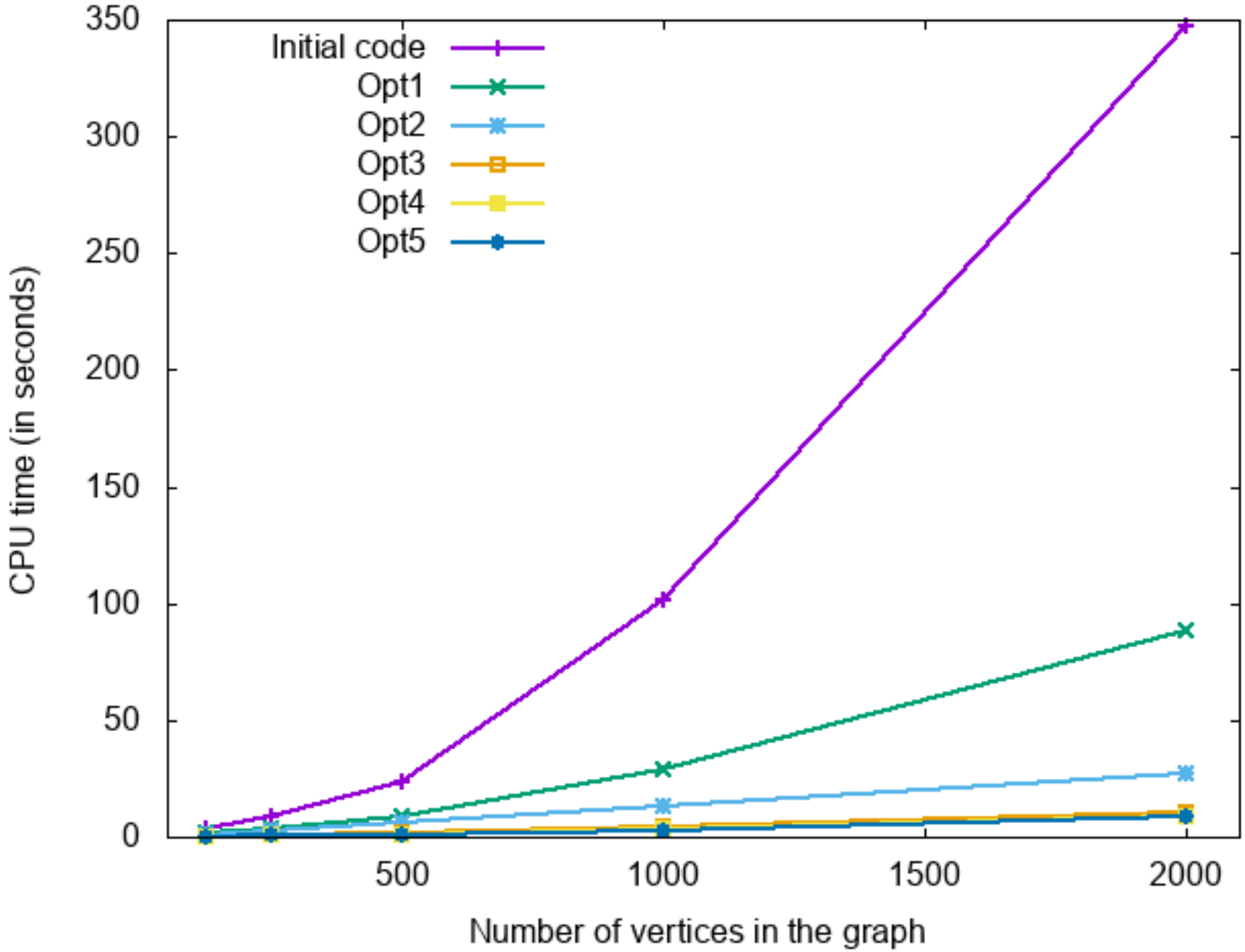
	Init	Opt1	Opt2	Opt3	Opt4	Opt5	$\frac{\text{init}}{\text{Opt5}}$
C125.9	3.61	1.79	1.29	0.32	0.24	0.24	15
C250.9	8.34	3.65	2.78	0.71	0.51	0.51	16
C500.9	23.53	8.49	5.84	1.59	1.21	1.11	21
C1000.9	111.30	28.94	13.52	4.06	3.32	2.98	37
C2000.9	347.15	88.18	27.32	10.64	9.14	8.81	39

Experimental Comparison (with -O3!)

- Initial code
- Opt1: Incrementally maintain candidates
- Opt2: Incrementally maintain pheromone factor
- Opt3: Replace `pow` of `math.h` with an *ad-hoc* function
- Opt4: Merge loops
- Opt5: Dichotomous search of the selected vertex

	Init	Opt1	Opt2	Opt3	Opt4	Opt5	$\frac{\text{init}}{\text{Opt5}}$
C125.9	3.61	1.79	1.29	0.32	0.24	0.24	15
C250.9	8.34	3.65	2.78	0.71	0.51	0.51	16
C500.9	23.53	8.49	5.84	1.59	1.21	1.11	21
C1000.9	111.30	28.94	13.52	4.06	3.32	2.98	37
C2000.9	347.15	88.18	27.32	10.64	9.14	8.81	39
C2000.5	33.64	11.37	6.23	4.32	4.15	4.03	8
C4000.5	85.53	30.50	18.73	14.75	14.17	14.02	6

Scale-up Properties of the 6 Variants



Plan

Parameters and Hyper-Parameters

Parameters = Variables that define thresholds, weights, frequencies, . . .

- Parameters change the algorithm performance
- Examples:
  - Simulated Annealing: Initial temperature, Cooling rate
  - Tabu Search: Tabu list length
  - GA: Population size, Cross-over rate, Mutation rate

Hyper-parameters = Variables that correspond to design choices

- Hyper-parameters change the algorithm
- Examples:
  - Branch & Bound: Bound function
  - Local Search: Neighborhood function
  - Constraint Programming: Filtering algorithm

Both param. and hyper-param. are called "Parameters" in what follows

The Vocabulary of Experimentation (recalls)

Factors = Parameters that are studied in the experiment

$\rightsquigarrow$ Identify "important" parameters, and fix the other parameters

Levels = Set of possible values for a factor

- Symbolic factor: 1 level per value
- Numeric factor: Identify intervals of relevant values by sampling
  $\rightsquigarrow$ Use a geometric serie to sample: 1, 2, 4, 8, ... or 1, 10, 100, ...

Configuration = An assignment of one level to each factor

Design Point = Configuration that must be experimentally evaluated

- Full factorial design = All Factor/Level combinations (grid search)
  - Pros: Identify all factor effects, including interaction effects due to inter-dependency of factors
  - Cons: Exponential number of combinations wrt number of factors
- Fractional factorial design = Selection of a subset of configurations
  $\rightsquigarrow$ **How to select configurations that must be evaluated?**

Manual Tuning vs Automatic Configuration

Main drawbacks of manual parameter tuning:

- It is time consuming
- Intuitions may be misleading
- It may be unfair
  $\leadsto$ Are we spending the same time for tuning all approaches?
- The tuning step is not reproducible

Programming by Optimisation [Hoos 2012]:

Developers specify a potentially large design space of programs that accomplish a given task, from which versions of the program optimised for various use contexts are generated automatically.

Automatic Configuration

Definition of the problem:

Given:

- A set of configurations Θ of an algorithm A
- A distribution $\mathcal{D}$ over the set of instances $\mathcal{I}$ of the problem solved by A
- A performance measure $m : \Theta \times \mathcal{I} \to \mathbb{R}$

Search for $\theta^* \in \Theta$ which optimises the expectation of $m(\theta^*, i)$ when $i \sim \mathcal{D}$

How to define the distribution $\mathcal{D}$?

- $\mathcal{D}$ should be representative of the actual instances that must be solved
  $\rightsquigarrow$ Gather a set $\mathcal{S}$ of representative instances

How to obtain training instances from $\mathcal{S}$?

- Solution 1: Design a model for randomly generating instances that have the same distribution as $\mathcal{S}$
- Solution 2: Use $\mathcal{S}$ as a finite support definition of $\mathcal{D}$
  $\rightsquigarrow$ Split $\mathcal{S}$ into training and test sets for cross-validation

Example of Automatic Configuration Tool

$\rightsquigarrow$ Sequential Model-based Algorithm Configuration (SMAC)

Basic Idea:

- Perform an initial set $\mathcal{R}$ of runs and select a first configuration θ^*
- Iterate the following steps:
  - Use $\mathcal{R}$ to build a model for predicting configuration performances
  - Use that model to select promising configurations
  - For each selected configuration θ:
    - Compare θ with θ^* using Random Online Agressive Racing (ROAR)
    - Update θ^* if θ wins the race, and update the set $\mathcal{R}$ of runs

Reference:

F. Hutter, H. Hoos, K. Leyton-Brown (2011): *Sequential Model-Based Optimization for General Algorithm Configuration*. LION

Source code available at
`http://www.cs.ubc.ca/labs/beta/Projects/SMAC/`

Some other Automatic Configuration Tools

ParamILS: Greedy Local Search with Restarts

F. Hutter, H. Hoos, K. Leyton-Brown, T. Stützle (2009): *ParamILS: An Automatic Algorithm Configuration Framework*. JAIR

Source code available at
`http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`

Iterated F-race: Iteratively sample configurations to race

M. Lopez-Ibanez, J. Dubois-Lacoste, L. Perez Caceres, M. Birattari, T. Stützle (2016): *The irace package: Iterated racing for automatic algorithm configuration*. Operations Research Perspectives

Available as a R package

Plan

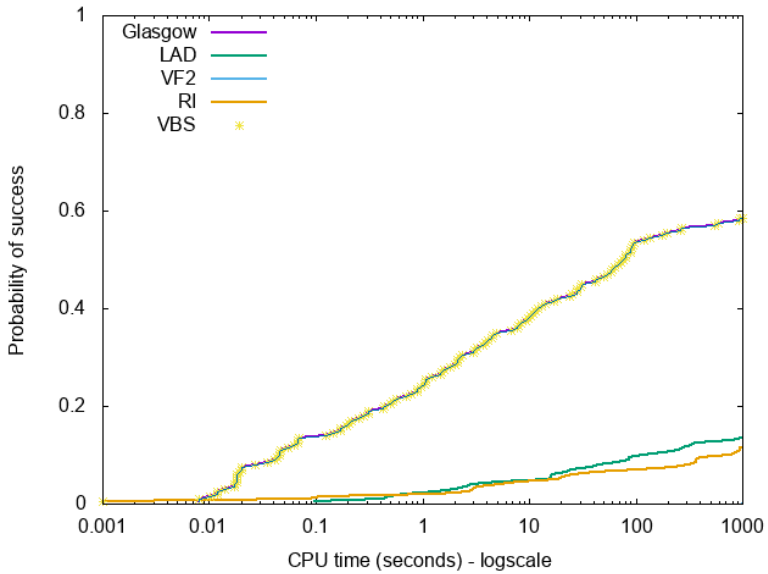
From Configuration to Selection

Automatic configuration finds the Single Best Solver (SBS)...

...But SBS may be far from VBS when instances are heterogeneous

Illustration on the subgraph isomorphism problem

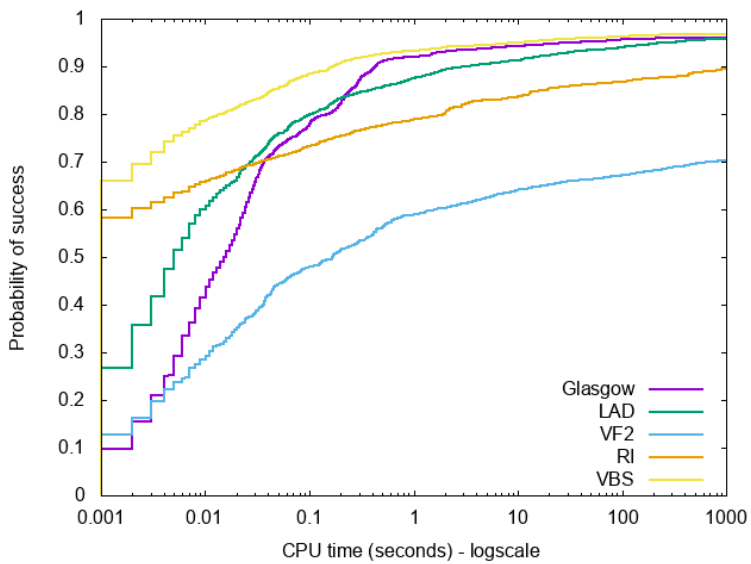
CDF for RandPhase instances:



- SBS = VBS = Glasgow

$\rightsquigarrow$ No need for per-instance selection

CDF for LV instances:



- SBS depends on time limit
- VBS outperforms SBSs

$\rightsquigarrow$ Use per-instance selection!

Per Instance Algorithm Selection

Definition of the problem:

Given a portfolio $\mathcal{P}$ of algorithms (or of algorithm configurations) and an instance i, select an algorithm $A \in \mathcal{P}$ expected to perform best on i

Offline training:

Given:

- A distribution $\mathcal{D}$ over the set of instances $\mathcal{I}$
- A performance measure $m : \mathcal{P} \times \mathcal{I} \to \mathbb{R}$
- An embedding function $f : \mathcal{I} \to \mathcal{F}$ where $\mathcal{F} \subseteq \mathbb{R}^m$ is the feature space
  $\leadsto$ Each instance $i \in \mathcal{I}$ is described by $f(i) \in \mathcal{F}$

Build a selector $S : \mathcal{F} \to \mathcal{P}$ which optimises $m(S(f(i)), i)$ when $i \sim \mathcal{D}$

Online selection of an algorithm for an instance $i \in \mathcal{I}$

Return $S(f(i))$

Examples of existing Automatic Selection Approaches

SATzilla:

- Offline: Learn a model for each algorithm
  $\rightsquigarrow$ Prediction of performance given instance features
- Online selection of an algorithm to solve a new instance *i*:
  $\rightsquigarrow$ Predict performance for each algorithm
  $\rightsquigarrow$ Select the algorithm with the best predicted performance

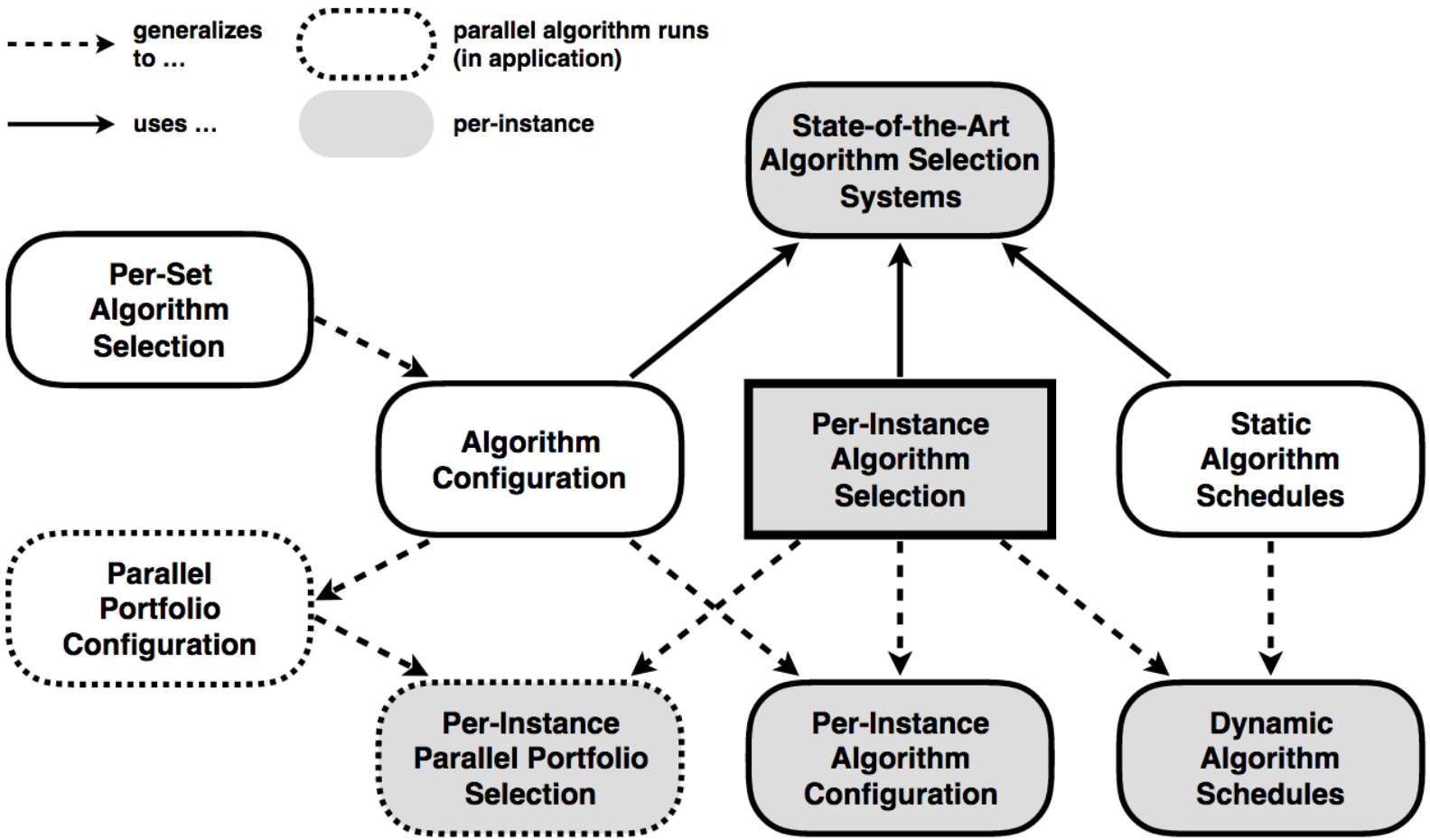
See L. Xu, F. Hutter, H. H. Hoos, K. Leyton-Brown (2009): SATzilla2009: an Automatic Algorithm Portfolio for SAT . SAT Competition 2009

ISAC:

- Offline: Partition instances into homogeneous clusters and use automatic configuration to determine the best algorithm for each cluster
- Online selection of an algorithm to solve a new instance *i*:
  $\rightsquigarrow$ Search for the cluster of *i* and select the corresponding algorithm

See Y. Malitsky: Instance-Specific Algorithm Configuration, PhD thesis, Brown University, 2012

Related Problems

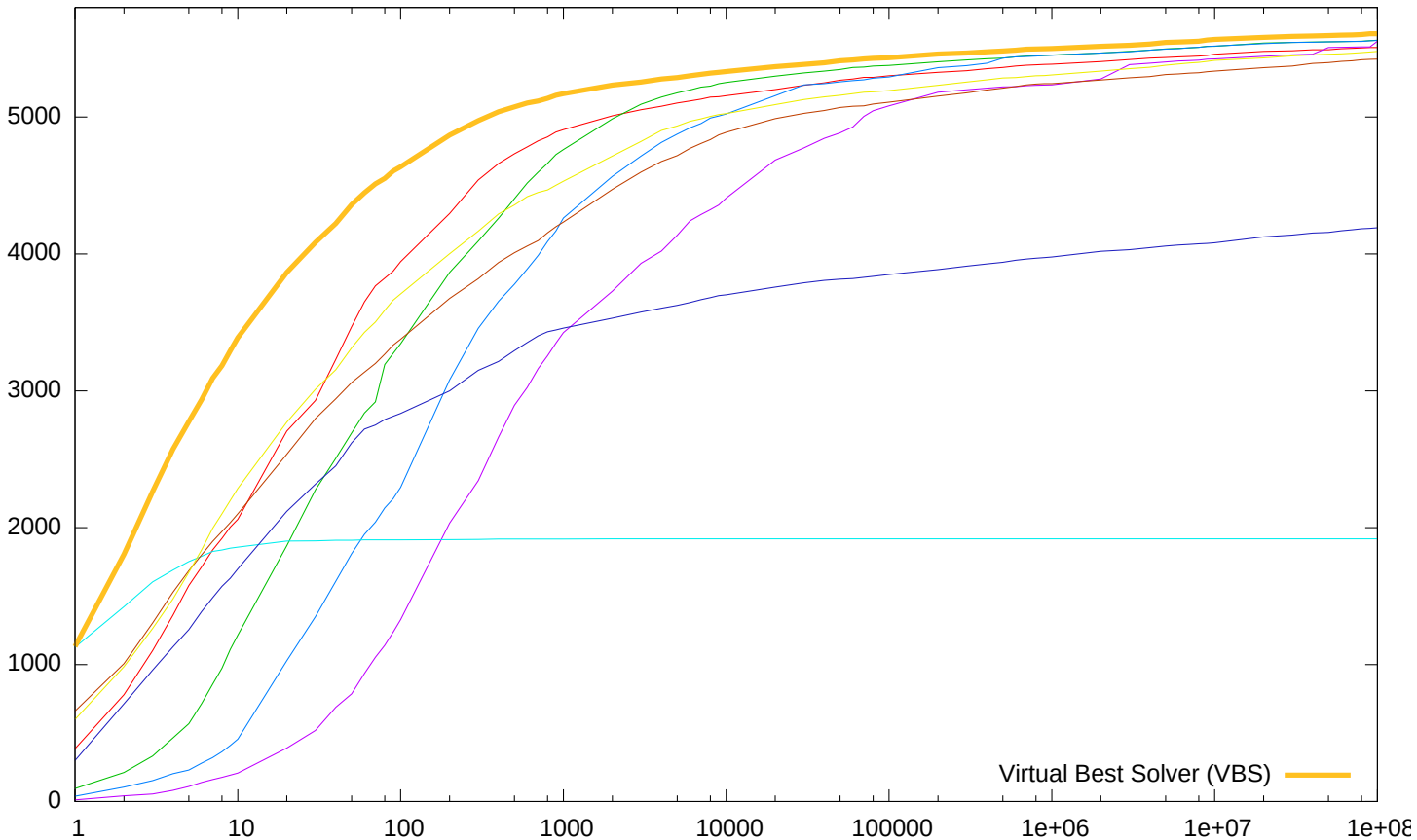


Reference:

P. Kerschke, H. Hoos, F. Neumann, H. Trautmann (2019): *Automated Algorithm Selection: Survey and Perspectives*. ECJ

Illustration: Algorithm Selection for Subgraph Isomorphism

⇝ CDF of 8 algorithms + VBS



Reference:

L. Kotthoff, C. McCreesh, C. Solnon: *Portfolios of Subgraph Isomorphism Algorithms*, in 10th International Conference on Learning and Intelligent OptimizatioN Conference (LION), 2016

Overview of the process

Offline:

- Describe each training instance by a feature vector
- Train a model that predicts the best algorithm for each training instance

Online: Solve a new instance $i \in \mathcal{I}$

- Sequentially run 2 very fast and complementary algorithms
  $\leadsto$ Solve very easy instances
  $\leadsto$ Collect dynamic features for instances that are not solved
- If instance not solved:
  - Extract features from i
  - Ask the model to select an algorithm given the features
  - Run the algorithm

Feature extraction

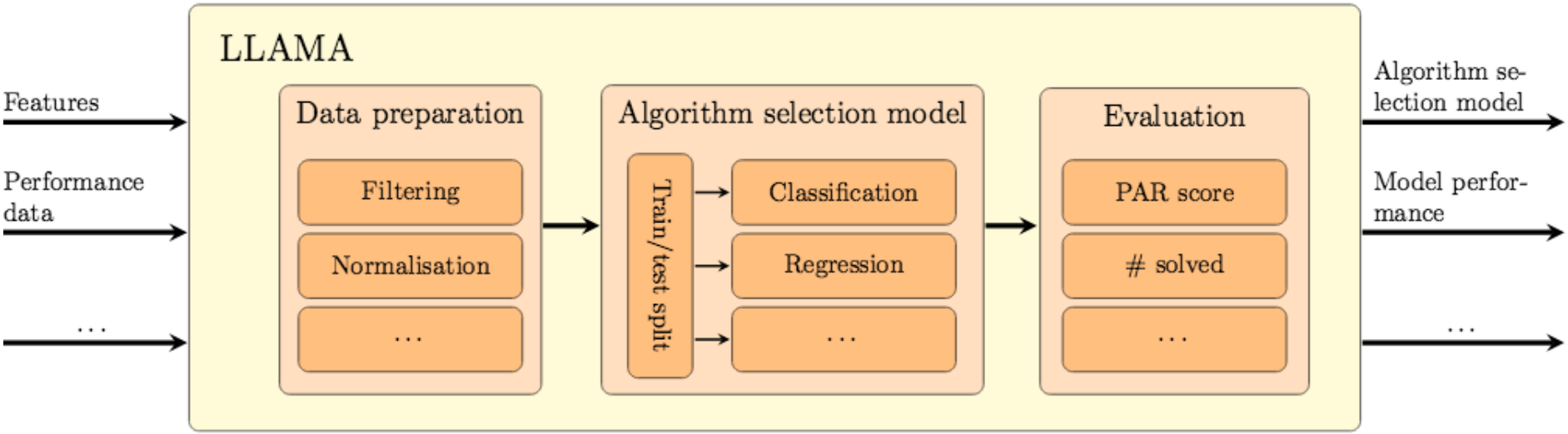
Static features extracted from the graphs

- Number of vertices and edges
- Density
- Number of loops
- Mean and max. degrees
- Mean and max. distance between all pairs of vertices
- Proportion of vertex pairs which are at least 2, 3 and 4 apart
- Binary features: Regular? Connected?

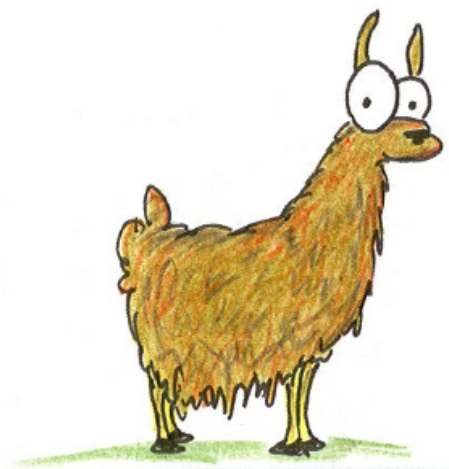
Dynamic features collected when running the 2 algorithms

- Number of value removals
- Percentage (average, min and max) of removed values per variable
- Algorithm solving time

Selection model: LLAMA



- R package for designing algorithm selectors
  https://bitbucket.org/lkotthoff/llama
- Includes different models
  ⇝ Best results: Pairwise regression approach with random forest regression
  - For each pair of algorithm, train a model to predict performance difference
  - Choose algorithm with highest cumulative performance difference



Experimental evaluation (1/2)

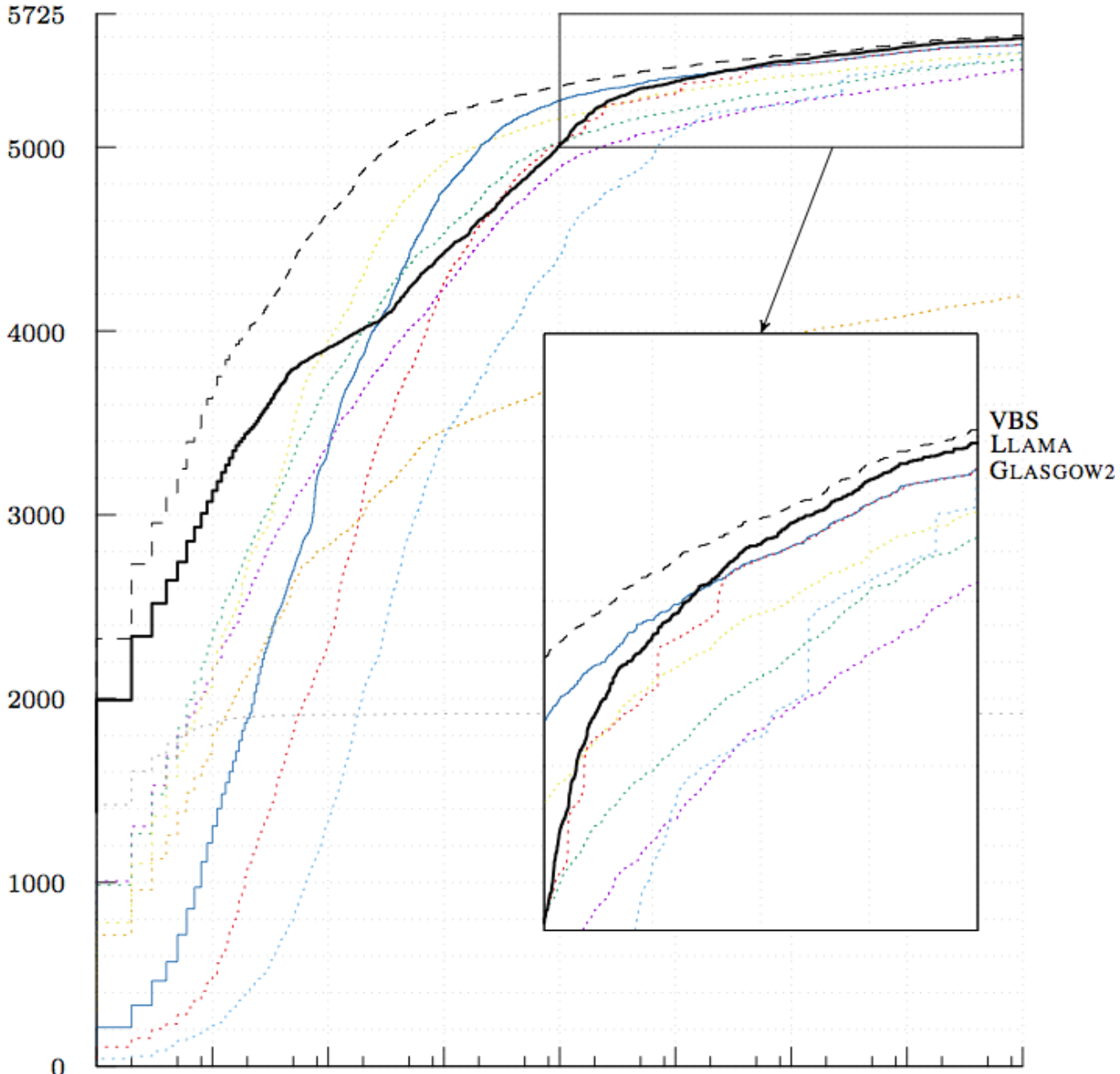
Experimental setup:

- 10-fold cross-validation
- Performance measures:
  - MCP: MisClassification Penalty
    $\rightsquigarrow$ Additional time required to solve an instance wrt VBS
  - # solved = number of instances that are solved
  - Time: time required to solve the instance, or 10^8 if not solved
    $\rightsquigarrow$ Lower bound of the actual time

Results:

Model	Mean MCP	# solved	Mean time
VBS	0	5,608	2,375,913
LLAMA	287,704	5,592	2,664,293
SBS	798,660	5,562	3,174,573

Experimental evaluation (2/2)



Plan du cours

Take away message?

Computer science is a science...

... where theory and practice should be combined!

- Theoretical analysis of algorithms
  - Study problem complexities before designing algorithms
  - Study the theoretical complexity of your algorithms
  - Prove properties of algorithms and codes
- Experimental analysis of algorithms
  - Choose benchmarks, factors, design points, and performance measures
  - Analyse results
  - Make it reproducible
- Algorithm engineering
  - Algorithm tuning vs code tuning
    $\leadsto$ Find the right compromise between efficiency and readability
  - Parameter tuning
    $\leadsto$ Use tools to automate parameter setting

Advertisement

You'd like to do a PhD thesis?

We have fundings in the context of the "ANR MAMUT"
$\rightsquigarrow$ Machine learning And Matheuristics algorithms for Urban Transportation