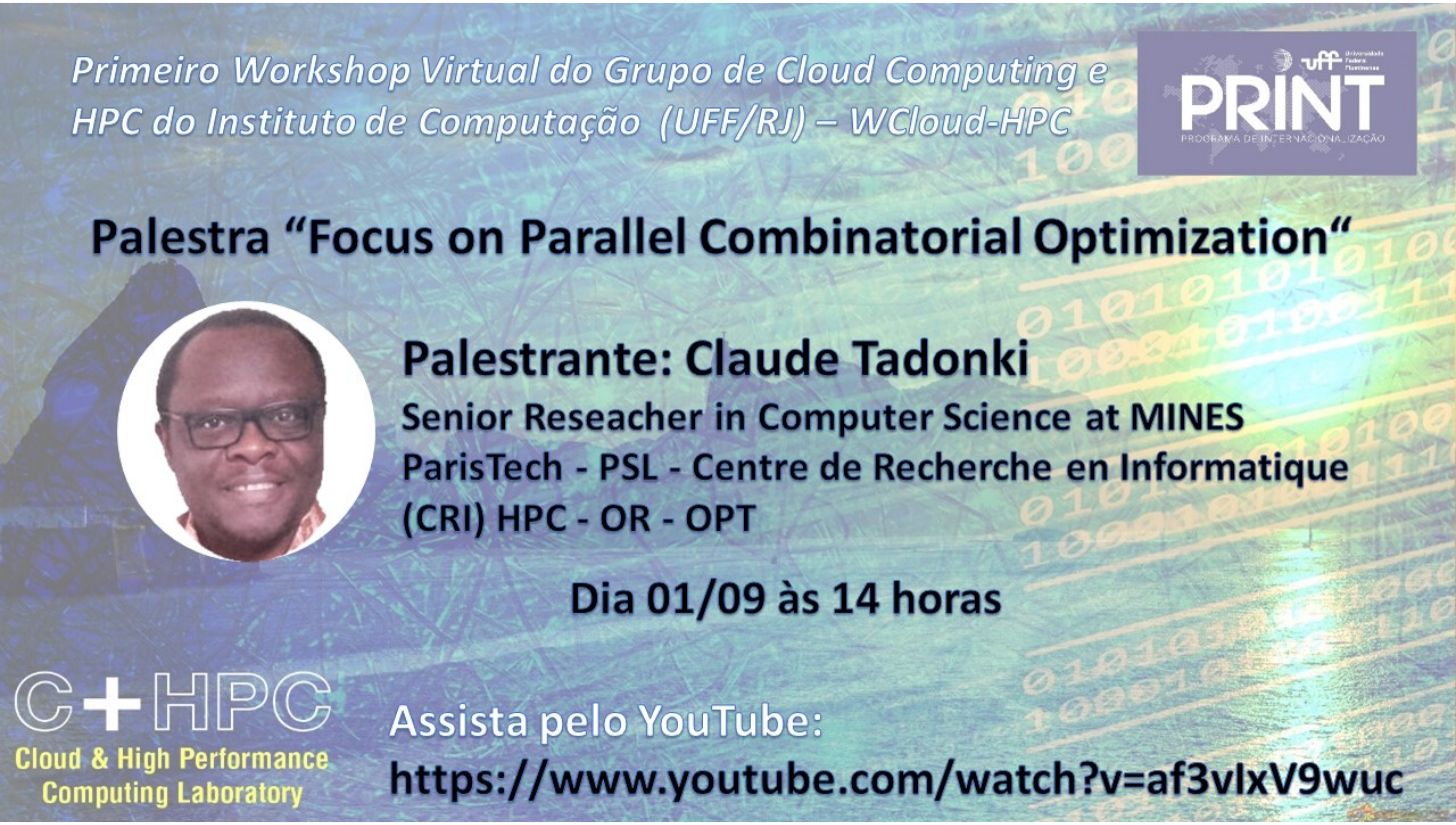


Primeiro Workshop Virtual do Grupo de Cloud Computing e HPC do Instituto de Computação (UFF/RJ) – WCloud-HPC

PRINT
PROGRAMA DE INTERNACIONALIZAÇÃO

Palestra "Focus on Parallel Combinatorial Optimization"

Palestrante: Claude Tadonki

Senior Reseacher in Computer Science at MINES ParisTech - PSL - Centre de Recherche en Informatique (CRI) HPC - OR - OPT

Dia 01/09 às 14 horas

C+HPC
Cloud & High Performance Computing Laboratory

Assista pelo YouTube:
https://www.youtube.com/watch?v=af3vlxV9wuc

Focus on Parallel Combinatorial Optimization

Workshop Cloud - HPC

September 1st, 2020, UFF - RJ (Brazil)

Combinatorial Problems: Some Basic Cases

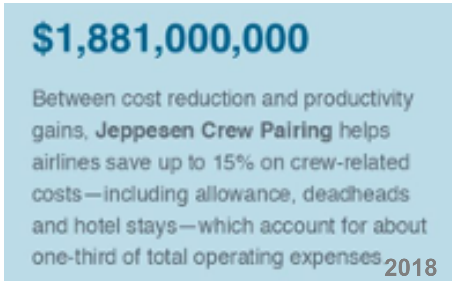
Airline Crew Pairing Problem

*In the **crew pairing problem**, each **crew is assigned** to a sequence of flight legs, so that **each flight** in the schedule **is covered** and the **total cost is minimized**.*

$$\min \sum_{p \in P} c_p y_p$$

$$\sum_{p : i \in p} y_p = 1$$

$$y_p \in \{0, 1\}$$



$1,881,000,000

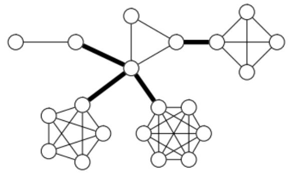
Between cost reduction and productivity gains, **Jeppesen Crew Pairing** helps airlines save up to 15% on crew-related costs—including allowance, deadheads and hotel stays—which account for about one-third of total operating expenses

2018

Genome Scaffolding

***Scaffolding** is an important step of the genome assembly and its function is to **order and orient the contigs** in the assembly of a draft genome **into larger scaffolds**.*

(σ_p, σ_c)-SCAFFOLDING (SCA)
Input: a scaffold graph (G^*, M^*, ω) and integers σ_p, σ_c.
Task: Find a collection S of σ_p alternating paths and σ_c alternating cycles maximizing $\sum_{e \in S \setminus M^*} \omega(e)$



Bin Packing Problem

*In the **bin packing problem**, items of different volumes must be packed into a finite number of bins each of a fixed volume in a way that minimizes the number of bins used.*

$$\min \sum_{j=1}^{m} y_j$$

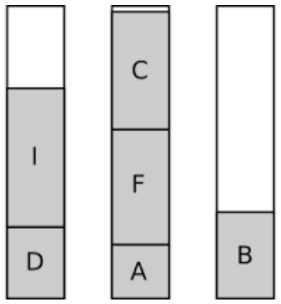
s.t

$$(1) \quad \sum_{j=1}^{m} x_{ij} = 1 \qquad \forall i = 1, \ldots, n$$

$$(2) \quad \sum_{i=1}^{n} s_i x_{ij} \le W y_j \qquad \forall j = 1, \ldots, m$$

$$x_{ij} \in \{0,1\} \qquad \forall i = 1, \ldots, n \quad \forall j = 1, \ldots, m$$

$$y_j \in \{0,1\} \qquad \forall j = 1, \ldots, m$$



Containers, buses, rooms, …

Stable Marriage Problem

*The stable marriage (or matching) problem is the problem of **finding a stable matching** between sets of elements **given an ordering of preferences** for each element.*

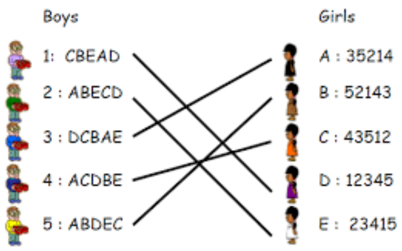
$$\max \sum_{i=1}^{n_1} \sum_{j \in F(i)} x_{ij}$$

$$\text{s.t.} \quad \sum_{j \in F(i)} x_{ij} \le 1, \qquad i = 1, \ldots, n_1,$$

$$\sum_{i \in C(j)} x_{ij} \le 1, \qquad j = 1, \ldots, n_2,$$

$$1 - \sum_{q \in F_j^{\le}(i)} x_{iq} \le \sum_{p \in C_i^{\le}(j)} x_{pj}, \qquad i = 1, \ldots, n_1,\ j \in F(i),$$

$$x_{ij} \in \{0,1\}, \qquad i = 1, \ldots, n_1,\ j \in F(i).$$

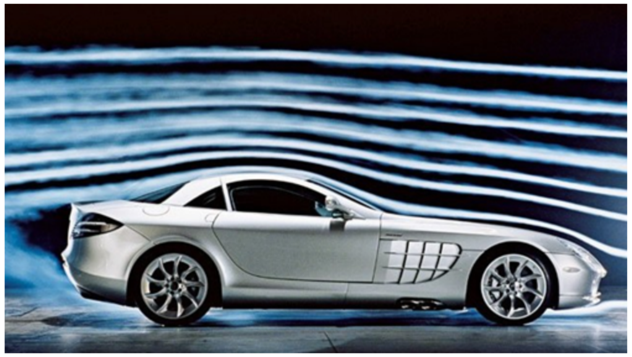


Compatible rooms assignment, …

Combinatorial Problems: The Need of Speed

- The instance to be solved might be a large-scale one
- The problem might be a subproblem that needs to be solve several times
- The configuration might change every time, so we have to run again the solver
- The need of real-time processing for real-life (combinatorial) problems
  Research; industry; gaming; simulations; transportation; design; decision; …

In the future, as our technology continues to improve and complexify, the ability to solve difficult problems of immense scale is likely to be in much higher demand, and will require breakthroughs in optimization algorithms.



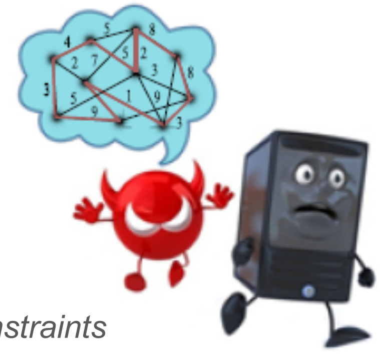
towardsdatascience.com/reinforcement-learning-for-combinatorial-optimization-d1402e396e91

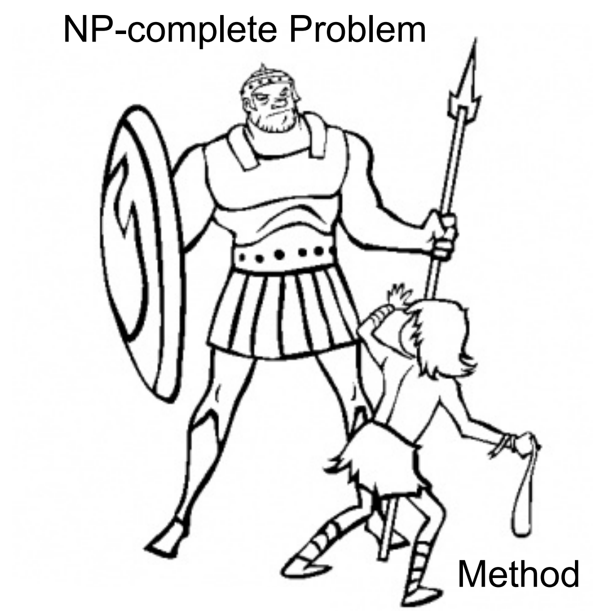
Combinatorial Optimization: Importance of the Method

- TRAVELING SALESMAN PROBLEM
  Given a list of cities and their pairwise distances, the task is to find a **shortest tour** that visits each city exactly once.

$$\begin{cases} \text{minimize} & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\ \text{subject to} & \\ & \sum_{j=1}^{n} x_{ij} = 1 \quad i=1,\cdots,n, i\neq j \qquad (2.4) \\ & \sum_{i=1}^{n} x_{ij} = 1 \quad j=1,\cdots,n, i\neq j \\ & x\in\{0,1\}^{n\times n} \end{cases}$$

+ *Subtour breaking constraints*





Applications:

Transportation, logistic, genome sequencing, benchmark for optimization methods, ...

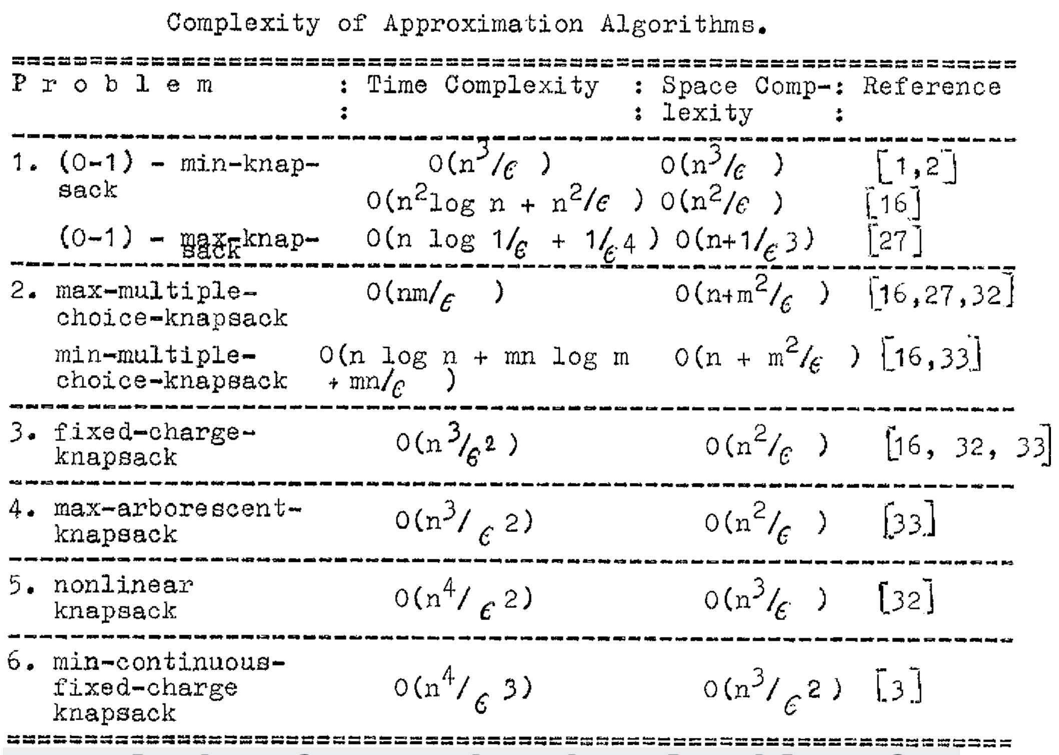
TSP heroes: Applegate, Bixby, Chvatal, and Cook

- As difficult as the HAMILTONIAN CYCLE PROBLEM, which is NP-COMPLETE (Karp)
- Please, forget about brute force approach! (**16 cities** $\Rightarrow$ **653 837 184 000 possibilities**)
- Modern optimization method have shown optimistic performances on practical instances
- Competitive solutions are **parallel implementation of powerful optimization methods**
- TSP (VLSI-Bell Labs) of size **85 900** solved in **1.5 year** (2004-2006) using a cluster of 96 2.8 GHz Intel Xeon and 32 2.4 GHz AMD Opteron connected with 100 MB ethernet.

Combinatorial Problems: Existing Methods & Approaches

- Ah-hoc Algorithms (brute force,intuition, common sense, …)
- Greedy Algorithms
- Dynamic Programming
- Approximation Algorithms
- Genetic Algorithms
- Branch and Bound
- Mathematical Programming
- Artificial Intelligence

AI approaches are emerging



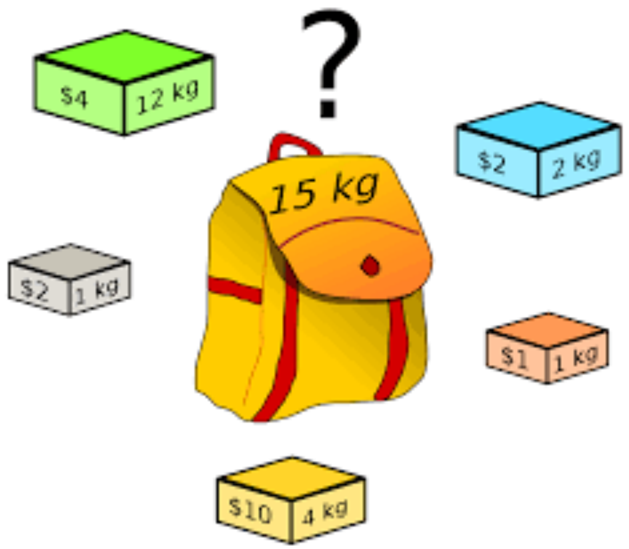
Complexity of Approximation Algorithms.

Problem	Time Complexity	Space Complexity	Reference
1. (0-1) - min-knapsack	$O(n^3/\epsilon)$	$O(n^3/\epsilon)$	[1,2]
	$O(n^2 \log n + n^2/\epsilon)$	$O(n^2/\epsilon)$	[16]
(0-1) - max-knapsack	$O(n \log 1/\epsilon + 1/\epsilon^4)$	$O(n+1/\epsilon^3)$	[27]
2. max-multiple-choice-knapsack	$O(nm/\epsilon)$	$O(n+m^2/\epsilon)$	[16,27,32]
min-multiple-choice-knapsack	$O(n \log n + mn \log m + mn/\epsilon)$	$O(n + m^2/\epsilon)$	[16,33]
3. fixed-charge-knapsack	$O(n^3/\epsilon^2)$	$O(n^2/\epsilon)$	[16, 32, 33]
4. max-arborescent-knapsack	$O(n^3/\epsilon^2)$	$O(n^2/\epsilon)$	[33]
5. nonlinear knapsack	$O(n^4/\epsilon^2)$	$O(n^3/\epsilon)$	[32]
6. min-continuous-fixed-charge knapsack	$O(n^4/\epsilon^3)$	$O(n^3/\epsilon^2)$	[3]

Complexity of approximation algorithms for combinatorial problems: a survey

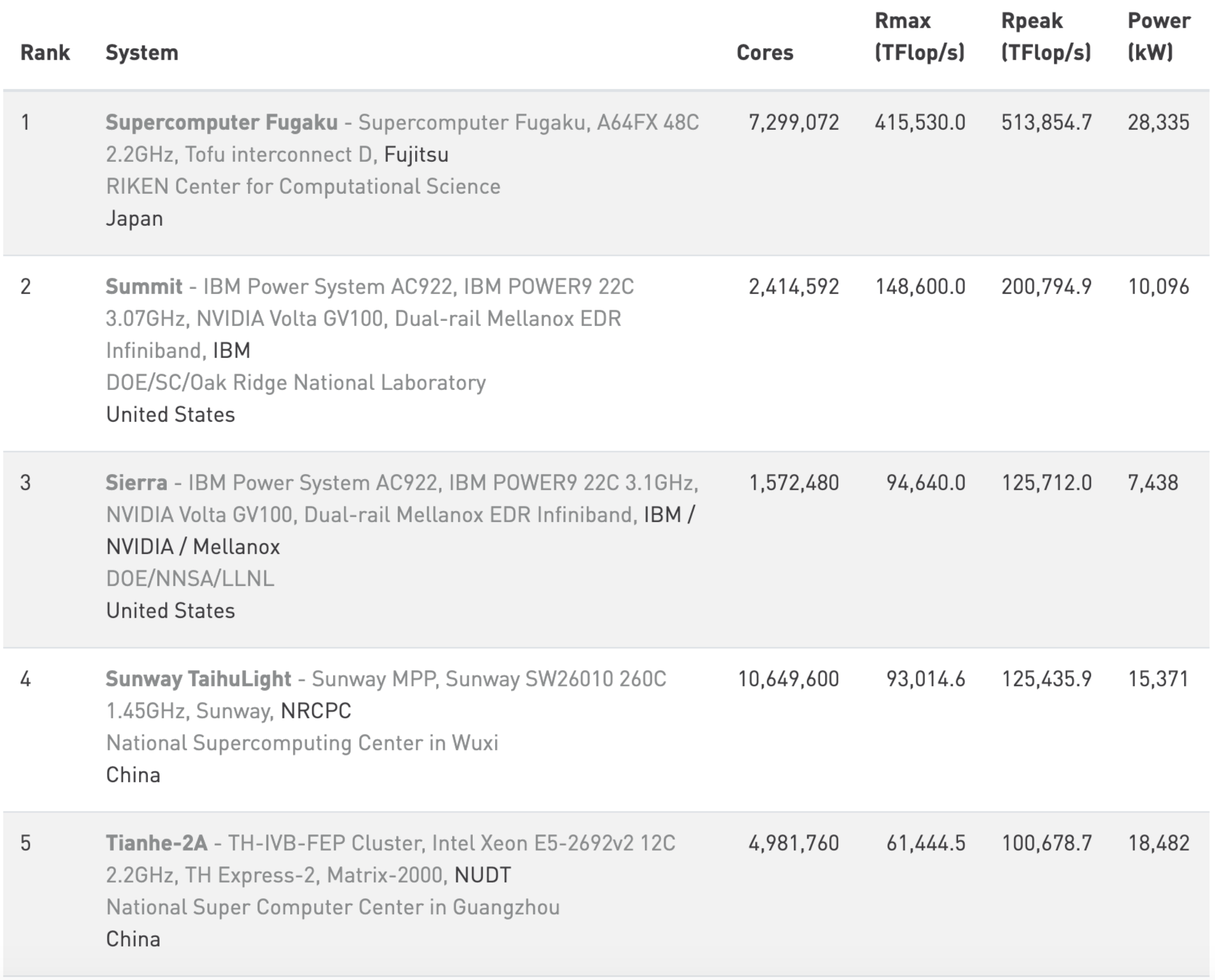
G V Gens, Evgenii Levner · Published 1980 · Computer Science · Sigact News

Knapsack Problem



The **time complexity** of an approximation algorithm **increases with its quality**!!!

June 2020 Top500 Ranking



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335	80%
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096	74%
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438	75%
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371	74%
5	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482	61%

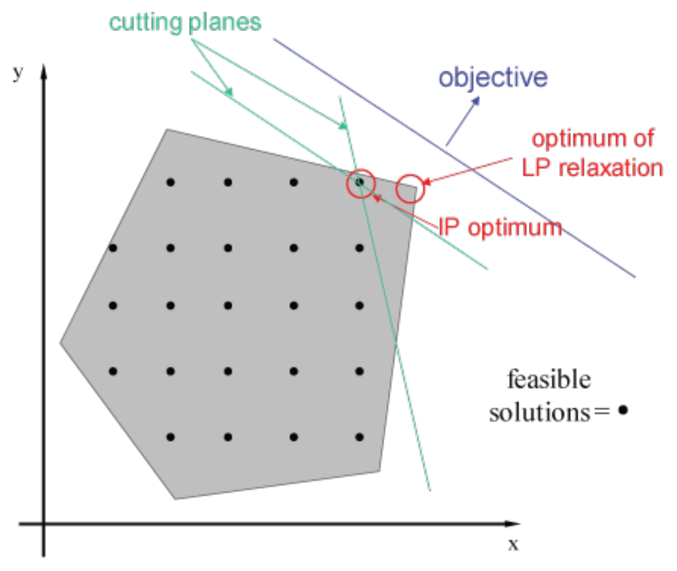
- **Considering the parallelism inherent to the paradigm itself**
  The method might indicate to consider several directions *(e.g. Branch and Bound)*

- **Parallelize the main components of the algorithm**
  If the method is made of (or requires) several computational items, parallelize each of these items *(derivatives, relaxations, associated problems, …)*.

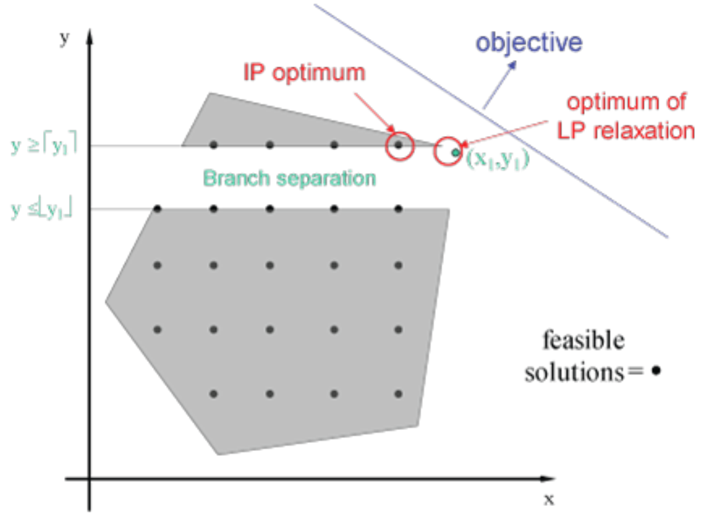
- **Parallel execution of the main components of the algorithm**
  With this approach, each component of the method has its own implementation and the whole system is executed in parallel (with necessary synchronization and sharing) .

- **Domain decomposition based parallelism**
  If possible, process in parallel with the subdomains, especially if they have reasonable interactions between them.

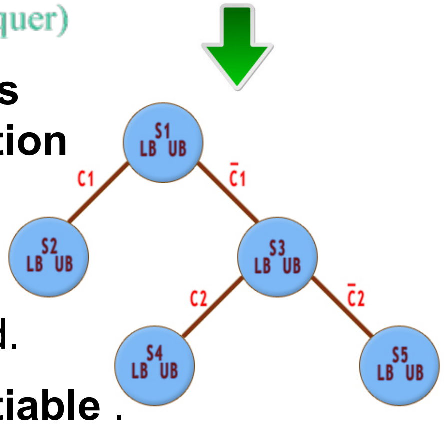
- **Systematic parallelization of a sequential implementation**
  Starting with a sequential implementation of a given method, use a systematic (source-to-source) parallelization approach.



Integer Program: Cutting Planes



Integer Program: Branch and Bound (or Divide and Conquer)



- **Practical instances** of **discrete** (pure or mixed) **optimization problems** are better solved though a skillfull combination of **continuous optimization** techniques and **branch&bound-like mechanisms**.
- For a pure discrete problem, a **relaxation** is used.
  For a mixed formulation, a **decomposition approach** can be considered.
- In number of cases, the objective function is (or becomes) **non differentiable** .
- We then need a good non differentiable optimization method and solver.

Load imbalance
The real volume of a branch is know after its exploration and cannot be predicted

Synchronization
There is a strong need of synchronization for the management of the common pool of working items like the gradients and lower/upper bounds.

Concurrent and irregular memory accesses
From the structural nature of the branch-and-bound, we should expect a significant memory penalty, which will be exacerbated on NUMA architectures.

Heavy data exchanges
Lot of occurrences of data exchange are expected for the coordination of the process beside ordinary reasons.

Resources sharing and weak scalability of node problems
Solving the node problems might suffer from weak scalability depending on the Implementation and the required resources are taken from the whole machine.

We now focus on directive-based parallelization
of
Dynamic Programming and **Greedy Algorithms**

From a given input S, dynamic programming works iteratively in a finite number of computing steps of the form

$$S_{k+1} = f(k, S_k)$$

where f is the generic iteration function and k the iterator parameter.

- It is common to consider in-place computation
- thus the procedure works by means of iterative updates

Table I provides a selection of well-known dynamic programming cases.

N°	Problem	Algorithm	Generic Update
1	Shortest Paths	Floyd-Warshall	$m_{i,j} = \min(m_{i,j}, m_{i,k} + m_{k,j})$
2	Dominated Graph Flooding	Berge	$\tau_i = \min(\tau_i, max(v_{i,j}, \tau_j))$
3	0-1 Knapsack Problem	Standard DP	$t_{i,w} = \max(t_{i-1,w}, v_{i-1} + t_{i-1,w-w_i})$
4	Longest Common Subsequence	Standard DP	$c_{i,j} = \begin{cases} c_{i-1,j-1} + 1 & \text{if } (s_i = t_i), \\ \max(c_{i-1,j}, c_{i,j-1}) & \text{otherwise} \end{cases}$
5	Longest Increasing Subsequence	Standard DP	$l_i = \max(l_i, l_j + 1)$ if $(a_i > a_j)$

```
for(k=0;k<n;k++)
 #pragma omp parallel for private(j)
 for(i=0;i<n;i++)
  for(j=0;j<n;j++)
   M[i,j] = min(M[i,j],M[i,k]+M[k,j]);
```

- A direct parallelization of a the update is valid because
  - The only dependencies are with the pivot (row and column)
  - The pivot (row and column) is invariant at the corresponding step
- Pivots sharing is a good point especially with a efficient SM cache protocol

Dominated Graph Flooding

Given weighted undirected graph $G = (X, E, v)$ and a ceiling function $\omega : X \to \mathbb{R}$.

<u>maximal</u> function $\tau : X \to \mathbb{R}$ satisfying

$$\forall x, y \in X : \quad \tau(x) \leq \min(\max(v(x,y), \tau(y)), \, \omega(x))$$

$$\boxed{\forall x, y \in X : \quad \tau(x) = \min(\max(v(x,y), \tau(y)), \, \omega(x))}$$

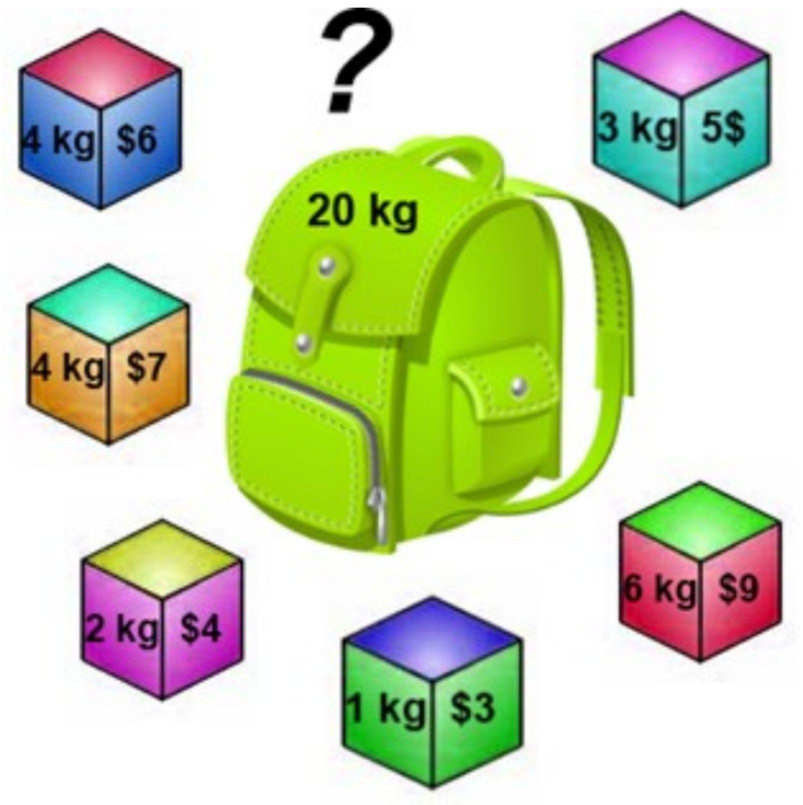
Berge algorithms computes the flooding $\tau = (\tau_i)$ as follows:
(i) $\tau^{(0)} \leftarrow \omega$
(ii) repeat update (4) until $(\tau_i^{(k)} = \tau_i^{(k-1)})$

$$\tau_i^{(k)} = \min(\tau_i^{(k)}, max(v_{i,j}, \tau_j^{(k-1)}), i = 1, 2, \cdots, n$$

- A direct parallelization of a the update is valid because
  - There is no dependence inside a step
  - The only dependencies are from one step to the next (k axis) & no in-place

```
while(doIt==1){
   #pragma omp parallel for private(j)
   for(i=0;i<n;i++){
      h[k%2,i] = h[(k+1)%2,i];
      for(j=0;j<n;j++)
         h[k%2,i] = min(h[k%2,i],
                        max(G[i,j],h[(k+1)%2,j]));
   }
   doIt=0;
   for(i=0;i<n;i++)
      if(h[1,i] != h[0,i]) {doIt=1; break;}
   k++;
}
```

The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items I_0, I_1, … I_{n-1}.



- Each item has 2 attributes:
  1) Value – let this be v_i for item I_i
  2) Weight – let this be w_i for item I_i

0-1 Knapsack Problem

```
**Knapsack**$(v, w, n, W)\{$
  **for**$(i = 1; i \leq n; j{+}{+})$
    **for**$(j = 1; j \leq W; j{+}{+})$
      **if**$(w[i] \leq j)$
        $V[i, j] = \max\{\ V[i-1, j], v[i] + V[i-1, j-w[i]]\ \};$
      **else**
        $V[i, j] = V[i-1, j];$
  **return** $V[n, W];$
$\}$
```

- All dependencies are of the form

$$(i, j) \leftarrow (i-1, j-\lambda)$$

- The computation along *i*-axis can be freely parallelized
- The one-step lifetime of V(*i*, :) suggests a V(*i* mod2, :) array compression

```
**Knapsack**$(v, w, n, W)\{$
  **for**$(i = 1; i \leq n; j$++)
    #pragma omp parallel for
    **for**$(j = 1; j \leq W; j$++)
      **if**$(w[i] \leq j)$
      $V[i\%2, j]=$ max$\{ V[(i-1)\%2, j], v[i]+V[i-1, j-w[i]] \};$
     **else**
        $V[\%2i, j]=V[(i-1)\%2, j];$
  **return** $V[n, W];$
$\}$
```

The *Longest Common Subsequence* (LCS) problem is to find the (length of the) longest common contiguous subsequence given two finite sequences.

Given two sequences $(u_i)_{i=1,\cdots,n}$ and $(v_i)_{i=1,\cdots,m}$, we define c_{ij} as the length of the LCS in $(u_1,\cdots,u_i)$ and $(v_1,\cdots,v_j)$. We have

$$c_{ij} = \begin{cases} c_{i-1,j-1}+1 & if\ u_i = v_j \\ \max(c_{i-1,j}, c_{i,j-1}) & otherwise \end{cases}$$

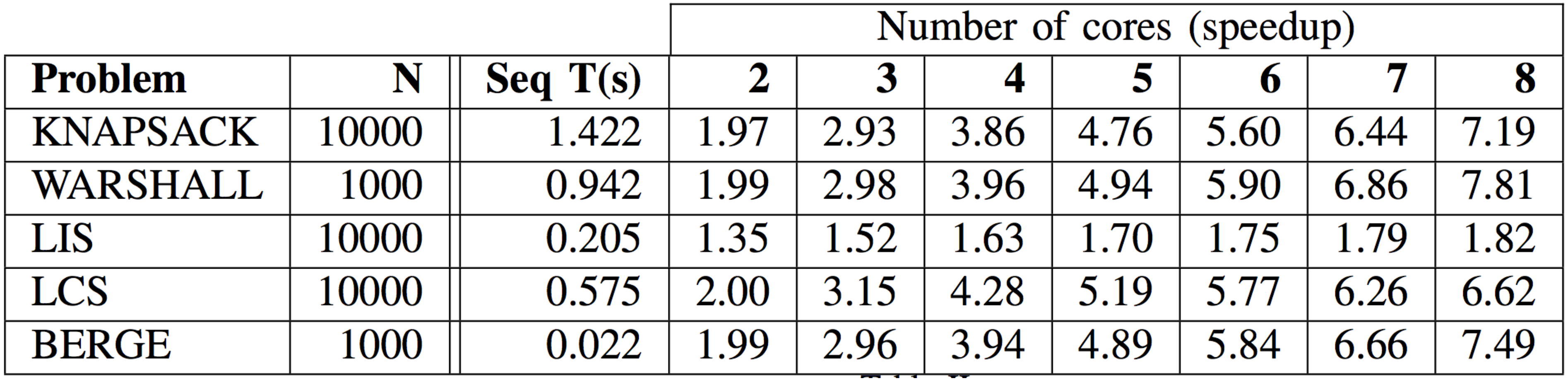
```
for(i=1;i<n;i++)
 for(j=1;j<n;j++)
  if(S[i] == T[j]) c[i,j] = c[i-1,j-1]+1;
  else c[i,j] = max(c[i,j-1],c[i-1,j]);
```

- The dependence $(i,j) \leftarrow (i-1,j-1)$ does not allow a direct parallelization
- A loop skewing transformation where the computation is done along the hyperplane $i+j=k,\ k=2,\cdots,2(n-1)$ makes a direct parallelization possible

Longest Common Subsequence (LCS)

```
for(k=2;k<=n;k++)
  #pragma omp parallel for
  for(i=1;i<k;i++){
   if(S[i] == T[(k-i)])
     c[w(i,k-i)] = c[w(i-1,(k-i)-1)]+1;
   else
     c[w(i,(k-i))] = max(c[w(i,(k-i)-1)],
                         c[w(i-1,(k-i))]);
  }
for(k=n+1;k<=2*(n-1);k++)
  #pragma omp parallel for
  for(i=(k-n)+1;i<n;i++){
   if(S[i] == T[k-i])
      c[i,k-i] = c[i-1,(k-i)-1]+1;
   else
      c[i,(k-i)] = max(c[i,(k-i)-1],
                         c[i-1,(k-i)]);
  }
```

Intel® Xeon® Processor E5-2699 v4
Released in April 2016



			Number of cores (speedup)						
Problem	**N**	**Seq T(s)**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
KNAPSACK	10000	1.422	1.97	2.93	3.86	4.76	5.60	6.44	7.19
WARSHALL	1000	0.942	1.99	2.98	3.96	4.94	5.90	6.86	7.81
LIS	10000	0.205	1.35	1.52	1.63	1.70	1.75	1.79	1.82
LCS	10000	0.575	2.00	3.15	4.28	5.19	5.77	6.26	6.62
BERGE	1000	0.022	1.99	2.96	3.94	4.89	5.84	6.66	7.49

From a given input set E, the generic step of a greedy algorithm is of the form

$$\boxed{S_{k+1} = S_k \cup f(E - S_k),}$$

where f is the generic selection function.

N°	Problem	Algorithm	Generic Selection
1	Shortest Paths (from a source node s)	Dijkstra	$i_{k+1} = \min\limits_{i \in E - S_k} dist(s,i)$
2	Minimum Spanning Tree	Prim	$a_{k+1} = \min\limits_{i \in S_k, j \in E - S_k} m_{i,j}$
3	Dominated Graph Flooding	Moore-Dijkstra	$i_{k+1} = \min\limits_{i \in E - S_k} \tau_i$

				Number of cores (speedup)						
N°	**N**	**Degrees**	**Seq T(s)**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	10^5	[20:100]	4.152	1.90	2.72	3.46	4.13	4.53	5.00	5.46
2	10^5	[10:20]	4.107	1.93	2.79	3.56	4.24	4.79	5.29	5.53
3	2×10^5	[10:20]	16.283	1.96	2.88	3.77	4.58	5.35	6.02	6.63
4	4×10^5	[10:20]	64.689	1.97	2.93	3.85	4.74	5.59	6.39	7.21

END & QUESTIONS