Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

V T'kindt

tkindt@univ-tours.fr, Université Francois-Rabelais, CNRS, Tours, France

June 2019

- What is called an "exponential algorithm"?....
- For a NP-hard problem, an exact or heuristic algorithm for which *the worst-case (time/space) complexity can be computed*,
- An **Exact** Exponential-Time Algorithm (E-ETA) provides an optimal solution to the problem,
- A **Heuristic** Exponential-Time Algorithm (H-ETA) provides a solution which worst-case quality can be bounded (approximation algorithm).

- What is called an "exponential algorithm"?....
- For a NP-hard problem, an exact or heuristic algorithm for which *the worst-case (time/space) complexity can be computed*,
- An **Exact** Exponential-Time Algorithm (E-ETA) provides an optimal solution to the problem,
- A **Heuristic** Exponential-Time Algorithm (H-ETA) provides a solution which worst-case quality can be bounded (approximation algorithm).

- What is called an "exponential algorithm"?....
- For a NP-hard problem, an exact or heuristic algorithm for which *the worst-case (time/space) complexity can be computed*,
- An **Exact** Exponential-Time Algorithm (E-ETA) provides an optimal solution to the problem,
- A **Heuristic** Exponential-Time Algorithm (H-ETA) provides a solution which worst-case quality can be bounded (approximation algorithm).

- What is called an “exponential algorithm”?....
- For a NP-hard problem, an exact or heuristic algorithm for which *the worst-case (time/space) complexity can be computed*,
- An **Exact** Exponential-Time Algorithm (E-ETA) provides an optimal solution to the problem,
- A **Heuristic** Exponential-Time Algorithm (H-ETA) provides a solution which worst-case quality can be bounded (approximation algorithm).

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find “theoretical” algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find "theoretical" algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find "theoretical" algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find "theoretical" algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find "theoretical" algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About E-ETA :
  - An E-ETA is not intended to be good in practice (E-ETA vs Branch-and-Bound algorithms),
  - What happen in the worst-case is the matter of E-ETA,
  - Find "theoretical" algorithms with worst-case time/space upper bounds as low as possible...

    The MIS problem has been shown to be solvable $O^*(2^n)$ in 1977, $O^*(1.381^n)$ in 1999, $O^*(1.2201^n)$ in 2009, ...

    NB : $O^*(exp(n)) = O(poly(n)exp(n))$
  - In the future, E-ETA will start to beat in practice heuristics ?

    1.2201^n is smaller than n^4 for $n \leq 90$,

    1.1^n is faster than n^4 for $n \leq 230$,
  - Provide a quantitative information on the difficulty of a NP-hard problem,

- About H-ETA :
  - For a given heuristic H we compute a worst-case ratio ρ : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$,
  - H-ETA are relevant for problems that cannot be approximated (bounded ratio) in polynomial time,

    The MIS problem cannot be approximated in polynomial time within ratio $n^{\epsilon-1}, \forall \epsilon > 0$ (Zuckerman, 2006).

    The MIS problem can be approximated in $O^*(\gamma^{\rho n})$ time within ratio $\rho \leq 1$ by using an E-ETA running in $O^*(\gamma^n)$ time ([0]).
  - Pay for an exponential time to get a guarantee on the quality (but pay less than to solve to optimality),

[0] N. Bourgeois, B. Escoffier, V. Paschos (2011). Approximation of Max Independent Set, Min Vertex Cover and related problems by moderately exponential algorithms, Discrete Applied Mathematics, 159(17) : 1954-1970

- About H-ETA :
  - For a given heuristic H we compute a worst-case ratio ρ : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$,
  - H-ETA are relevant for problems that cannot be approximated (bounded ratio) in polynomial time,

    The MIS problem cannot be approximated in polynomial time within ratio $n^{\epsilon-1}, \forall \epsilon > 0$ (Zuckerman, 2006).

    The MIS problem can be approximated in $O^*(\gamma^{\rho n})$ time within ratio $\rho \leq 1$ by using an E-ETA running in $O^*(\gamma^n)$ time ([0]).
  - Pay for an exponential time to get a guarantee on the quality (but pay less than to solve to optimality),

[0] N. Bourgeois, B. Escoffier, V. Paschos (2011). Approximation of Max Independent Set, Min Vertex Cover and related problems by moderately exponential algorithms, Discrete Applied Mathematics, 159(17) : 1954-1970

- About H-ETA :
  - For a given heuristic H we compute a worst-case ratio ρ : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$,
  - H-ETA are relevant for problems that cannot be approximated (bounded ratio) in polynomial time,

    The MIS problem cannot be approximated in polynomial time within ratio $n^{\epsilon-1}, \forall \epsilon > 0$ (Zuckerman, 2006).

    The MIS problem can be approximated in $O^*(\gamma^{\rho n})$ time within ratio $\rho \leq 1$ by using an E-ETA running in $O^*(\gamma^n)$ time ([0]).
  - Pay for an exponential time to get a guarantee on the quality (but pay less than to solve to optimality),

[0] N. Bourgeois, B. Escoffier, V. Paschos (2011). Approximation of Max Independent Set, Min Vertex Cover and related problems by moderately exponential algorithms, Discrete Applied Mathematics, 159(17) : 1954-1970

- About H-ETA :
  - For a given heuristic H we compute a worst-case ratio ρ : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$,
  - H-ETA are relevant for problems that cannot be approximated (bounded ratio) in polynomial time,

    The MIS problem cannot be approximated in polynomial time within ratio $n^{\epsilon-1}, \forall \epsilon > 0$ (Zuckerman, 2006).

    The MIS problem can be approximated in $O^*(\gamma^{\rho n})$ time within ratio $\rho \leq 1$ by using an E-ETA running in $O^*(\gamma^n)$ time ([0]).
  - Pay for an exponential time to get a guarantee on the quality (but pay less than to solve to optimality),

[0] N. Bourgeois, B. Escoffier, V. Paschos (2011). Approximation of Max Independent Set, Min Vertex Cover and related problems by moderately exponential algorithms, Discrete Applied Mathematics, 159(17) : 1954-1970

- In this talk...
- We first tackle E-ETA providing several techniques that can be applied successfully applied to scheduling problems,
- Next, we tackle H-ETA and first applications to scheduling problems.

- In this talk...
- We first tackle E-ETA providing several techniques that can be applied successfully applied to scheduling problems,
- Next, we tackle H-ETA and first applications to scheduling problems.

1 Introduction

2 Exact Exponential-Time Algorithms
- Technique 1 : Dynamic Programming
- Technique 2 : Branch-and-Reduce
- Technique 3 : Sort&Search

3 Heuristic Exponential-Time Algorithms

4 Conclusions

- A lot of works on graph or decision problems (70's, 2000-),
  - 3-SAT : $O^*(1.3211^n)$ time (Iwama et al., 2010),
  - Hamiltonian circuit : $O^*(1.657^n)$ time (Bjorklund, 2010),
  - MIS : $O^*(1.2132^n)$ time (Kneis et al, 2009),
  - List coloring : $O^*(2^n)$ time (Bjorklund and Husfeldt, 2006) and (Koivisto, 2006),
  - ...
- A growing interest since $\approx$ 2005 in scheduling literature,

- A lot of works on graph or decision problems (70's, 2000-),
  - 3-SAT : $O^*(1.3211^n)$ time (Iwama et al., 2010),
  - Hamiltonian circuit : $O^*(1.657^n)$ time (Bjorklund, 2010),
  - MIS : $O^*(1.2132^n)$ time (Kneis et al, 2009),
  - List coloring : $O^*(2^n)$ time (Bjorklund and Husfeldt, 2006) and (Koivisto, 2006),
  - ...
- A growing interest since $\approx$ 2005 in scheduling literature,

- A lot of works on graph or decision problems (70's, 2000-),
  - 3-SAT : $O^*(1.3211^n)$ time (Iwama et al., 2010),
  - Hamiltonian circuit : $O^*(1.657^n)$ time (Bjorklund, 2010),
  - MIS : $O^*(1.2132^n)$ time (Kneis et al, 2009),
  - List coloring : $O^*(2^n)$ time (Bjorklund and Husfeldt, 2006) and (Koivisto, 2006),
  - ...
- A growing interest since $\approx$ 2005 in scheduling literature,

- What about scheduling problems (single machine) ?

Problem	brute force	wctc	wcsc	Reference
$1\|dec\|f_{max}$	$O^*(n!)$	$O^*(2^n)$	exp	[1]
$1\|dec\|\sum_i f_i$	$O^*(n!)$	$O^*(2^n)$	exp	[1]
$1\|prec\|\sum_i C_i$	$O^*(n!)$	$O^*((2-\epsilon)^n)$	exp	[2]
$1\|prec\|\sum_i w_i C_i$	$O^*(n!)$	$O^*(2^n)$	exp	[3]
$1\|d_i\|\sum_i w_i U_i$	$O^*(n!)$	$O^*(2^n)$ $O^*(1.4142^n)$	exp exp	[3] [4]
$1\|d_i\|\sum_i T_i$	$O^*(n!)$	$O^*(2^n)$	exp	[3] & [4]
$1\|d_i\|\sum_i w_i T_i$	$O^*(n!)$	$O^*(2^n)$	poly	[5]
$1\|r_i, prec\|\sum_i w_i C_i$	$O^*(n!)$	$O^*(3^n)$	exp	[3] & [4]

[1] F. Fomin, D. Kratsch (2010). Exact Exponential Algorithms, Springer.

[2] M. Cygan, M. Philipczuk, M. Philipczuk, J. Wojtaszczyk (2011). Scheduling partially ordered jobs faster than 2^n, Proceedings of 19th Annual European Symposium (ESA 2011), Lecture Notes in Computer Science, vol. 6942, pp. 299-310.

[3] G. Woeginger (2003). Exact algorithms for NP-hard problems : A survey, in M. Junger, G. Reinelt, G. Rinaldi (Eds) : Combinatorial Optimization – Eureka I shrink !, Springer, LNCS 2570.

[4] C. Lenté, M. Liedloff, A. Soukhal, V.T'kindt (2013). On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, 511, pp 13-22.

[5] M. Garraffa, L. Shang, F. Della Croce, V. T'Kindt (2018). An exact exponential branch-and-merge algorithm for the single machine tardiness problem. Theoretical Computer Science, 745, 133-149.

- What about scheduling problems (others) ?

Problem	brute force	wctc	wcsc	Reference
$P\|dec\|f_{max}$	$O^*(m^n n!)$	$O^*(3^n)$	exp	[4]
$P\|dec\|\sum_i f_i$	$O^*(m^n n!)$	$O^*(3^n)$	exp	[4]
$P4\|\|C_{max}$	$O^*(4^n)$	$O^*(2.4142^n)$	exp	[4]
$P3\|\|C_{max}$	$O^*(3^n)$	$O^*(1.7321^n)$	exp	[4]
$P2\|\|C_{max}$	$O^*(2^n)$	$O^*(1.4142^n)$	exp	[4]
$P2\|d_i\|\sum_i w_i U_i$	$O^*(3^n)$	$O^*(1.7321^n)$	exp	[4]
$F2\|\|C^k_{max}$	$O^*(2^n)$	$O^*(1.4142^n)$	exp	[4]
$F3\|\|C_{max}$	$O^*(n!)$	$O^*(3^n)$	exp	[6]
$F3\|\|f_{max}$	$O^*(n!)$	$O^*(5^n)$	exp	[6]
$F3\|\|\sum_i f_i$	$O^*(n!)$	$O^*(5^n)$	exp	[6]
$J2\|\|C^k_{max}$	$O^*(2^n)$	$O^*(1.4142^n)$	exp	[7]

[4] C. Lenté, M. Liedloff, A. Soukhal, V.T'kindt (2013). On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, 511, pp 13-22.

[6] L. Shang, C. Lenté, M. Liedloff, V.T'kindt (2018). An exponential dynamic programming algorithm for the 3-machine flowshop scheduling problem to minimize the makespan, Journal of Scheduling, 21(2) :227-233.

[7] F. Della Croce, C. Koulamas, V.T'kindt (2016). A constraint generation approach for two-machine shop problems with jobs selection, Eur. J. Oper. Research, submitted.

- We focus on three technics with application to scheduling :
  - Dynamic programming,
  - Branch-and-merge,
  - Sort&Search.

About permutation problems...

- Let us consider the $1||f_{max}$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i,
  - a non-decreasing cost function f_i depending on C_i
  -
- The worst-case complexity of ENUM...is in $O^*(n!)$.

About permutation problems...

- Let us consider the $1||f_{max}$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i,
  - a non decreasing cost function f_i depending on C_i,
  - Goal : Find the permutation which minimizes $f_{max} = \max_i f_i$.
- The worst-case complexity of ENUM...is in $O^*(n!)$.

About permutation problems...

- Let us consider the $1||f_{max}$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i,
  - a non decreasing cost function f_i depending on C_i,
  - Goal : Find the permutation which minimizes $f_{max} = \max_i f_i$.
- The worst-case complexity of ENUM...is in $O^*(n!)$.

About permutation problems...

- Let be $S \subseteq \{1, \ldots, n\}$,
- Let $Opt[S]$ be the recurrence function calculated on set S : $Opt[S]$ is equal to the minimal value of criterion $\max_i f_i$ for any permutation of the jobs in S.
- We have :
$$\begin{cases} Opt[\emptyset] = -\infty, \text{ if } f_t \text{ can be negative} \\ Opt[\emptyset] = 0, \text{ if } f_t \text{ cannot be negative} \\ Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\} \end{cases}$$
with $P(S) = \sum_{i \in S} p_i$.

About permutation problems...

- Let be $S \subseteq \{1, \ldots, n\}$,
- Let $Opt[S]$ be the recurrence function calculated on set S : $Opt[S]$ is equal to the minimal value of criterion $\max_i f_i$ for any permutation of the jobs in S.
- We have :

$$\begin{cases} Opt[\emptyset] = -\infty, \text{ if } f_t \text{ can be negative} \\ Opt[\emptyset] = 0, \text{ if } f_t \text{ cannot be negative} \\ Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\} \end{cases}$$

with $P(S) = \sum_{i \in S} p_i$.

About permutation problems...

- Let be $S \subseteq \{1, \ldots, n\}$,
- Let $Opt[S]$ be the recurrence function calculated on set S : $Opt[S]$ is equal to the minimal value of criterion $\max_i f_i$ for any permutation of the jobs in S.
- We have :

$$\begin{cases} Opt[\emptyset] = -\infty, \text{ if } f_t \text{ can be negative} \\ Opt[\emptyset] = 0, \text{ if } f_t \text{ cannot be negative} \\ Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\} \end{cases}$$

with $P(S) = \sum_{i \in S} p_i$.

About permutation problems...

Lost with that recurrence function ? Proceed with the exercice,

Exercice.

Apply the dynamic programming algorithm on the following instance :
$n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1$,

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1$,

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3$,
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1$,

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1$,

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3$,
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1,$

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1,$

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3,$
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1,$

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1,$

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3,$
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1$,

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1$,

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3$,
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- $n = 3$, $[p_i]_i = [3; 4; 5]$, $[d_i]_i = [4; 5; 8]$, $f_i(C_i) = C_i - d_i$,
- Enumerate all sets S with 1 element,

  $S = \{1\} : Opt[S] = \max(-\infty; 3 - 4) = -1$,

  $S = \{2\} : Opt[S] = \max(-\infty; 4 - 5) = -1$,

  $S = \{3\} : Opt[S] = \max(-\infty; 5 - 8) = -3$,
- Do on your own for all sets with 2 and 3 elements !

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big),$
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big),$
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big),$
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big),$
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big),$
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big),$
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big),$
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big),$
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big),$
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big)$,
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big)$,
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big)$,
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big)$,
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big)$,
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big)$,
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1, 2\}$:
$$Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big),$$
$$\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$$
- $S = \{1, 3\}$:
$$Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big),$$
$$\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$$
- $S = \{2, 3\}$:
$$Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big),$$
$$\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$$

About permutation problems...

- Enumerate all sets S with 2 elements,
- $S = \{1,2\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{2\}]; f_1(7)\big); \max\big(Opt[\{1\}]; f_2(7)\big)\Big),$
  $\Rightarrow Opt[\{1,2\}] = \min\Big(\max(-1;3); \max(-1;2)\Big) = 2$
- $S = \{1,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_1(8)\big); \max\big(Opt[\{1\}]; f_3(8)\big)\Big),$
  $\Rightarrow Opt[\{1,3\}] = \min\Big(\max(-3;4); \max(-1;0)\Big) = 0$
- $S = \{2,3\}$:
  $Opt[S] = \min\Big(\max\big(Opt[\{3\}]; f_2(9)\big); \max\big(Opt[\{2\}]; f_3(9)\big)\Big),$
  $\Rightarrow Opt[\{2,3\}] = \min\Big(\max(-3;4); \max(-1;1)\Big) = 1$

About permutation problems...

- Enumerate all sets S with 3 elements,
- $S=\{1,2,3\}$: $Opt[S]=\min\Big(\max\big(Opt[\{2,3\}];f_1(12)\big);$
$$\max\big(Opt[\{1,3\}];f_2(12)\big);$$
$$\max\big(Opt[\{1,2\}];f_3(12)\big)\Big),$$

$$\Rightarrow Opt[\{1,2,3\}]=\min\Big(\max(1;8);\max(0;7);\max(2;4)\Big)=4$$

- This corresponds to the schedule $(1,2,3)$.

About permutation problems...

- Enumerate all sets S with 3 elements,
- $S=\{1,2,3\}$: $Opt[S]=\min\Big(\max\big(Opt[\{2,3\}];f_1(12)\big);$
$\max\big(Opt[\{1,3\}];f_2(12)\big);$
$\max\big(Opt[\{1,2\}];f_3(12)\big)\Big),$

$\Rightarrow Opt[\{1,2,3\}]=\min\Big(\max(1;8);\max(0;7);\max(2;4)\Big)=4$

- This corresponds to the schedule $(1,2,3)$.

About permutation problems...

- Enumerate all sets S with 3 elements,
- $S = \{1,2,3\}$: $Opt[S] = \min\Big(\max\big(Opt[\{2,3\}]; f_1(12)\big);$
$$\max\big(Opt[\{1,3\}]; f_2(12)\big);$$
$$\max\big(Opt[\{1,2\}]; f_3(12)\big)\Big),$$
$\Rightarrow Opt[\{1,2,3\}] = \min\Big(\max(1;8); \max(0;7); \max(2;4)\Big) = 4$
- This corresponds to the schedule $(1,2,3)$.

About permutation problems...

- Enumerate all sets S with 3 elements,
- $S = \{1,2,3\}$: $Opt[S] = \min\Big(\max\big(Opt[\{2,3\}]; f_1(12)\big); \max\big(Opt[\{1,3\}]; f_2(12)\big); \max\big(Opt[\{1,2\}]; f_3(12)\big)\Big),$

$\Rightarrow Opt[\{1,2,3\}] = \min\Big(\max(1;8); \max(0;7); \max(2;4)\Big) = 4$

- This corresponds to the schedule $(1,2,3)$.

About permutation problems...

- Analyse of the worst-case time complexity...

$$Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\}$$

- Usefull note : the computation of one $Opt[]$ can be done in $O(n)$ time.
- Fundamental question : how many computations of $Opt[]$ have to be done ?
- Generation of all subsets of a size $k \leq n$,

$$\sum_{k=0}^{n} \binom{n}{k},$$

which, by means of Newton's formula for sum of binomials :

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = (x + y)^n,$$

can be rewritten as : 2^n.

- The worst-case time (and space) complexity of DynPro is in $O^*(2^n)$,

About permutation problems...

- Analyse of the worst-case time complexity...

$$Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\}$$

- Usefull note : the computation of one $Opt[]$ can be done in $O(n)$ time.
- Fundamental question : how many computations of $Opt[]$ have to be done ?
- Generation of all subsets of a size $k \leq n$,

$$\sum_{k=0}^{n} \binom{n}{k},$$

which, by means of Newton's formula for sum of binomials :

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = (x+y)^n,$$

can be rewritten as : 2^n.
- The worst-case time (and space) complexity of DynPro is in $O^*(2^n)$.

About permutation problems...

- Analyse of the worst-case time complexity...

$$Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\}$$

- Usefull note : the computation of one $Opt[]$ can be done in $O(n)$ time.
- Fundamental question : how many computations of $Opt[]$ have to be done ?
- Generation of all subsets of a size $k \leq n$,

$$\sum_{k=0}^{n} \binom{n}{k},$$

which, by means of Newton's formula for sum of binomials :

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = (x+y)^n,$$

can be rewritten as : 2^n.

- The worst-case time (and space) complexity of DynPro is in $O^*(2^n)$.

About permutation problems...

- Analyse of the worst-case time complexity...

$$Opt[S] = \min_{t\in S}\{\max\big(Opt[S-\{t\}]; f_t(P(S))\big)\}$$

- Usefull note : the computation of one $Opt[]$ can be done in $O(n)$ time.
- Fundamental question : how many computations of $Opt[]$ have to be done ?
- Generation of all subsets of a size $k \leq n$,

$$\sum_{k=0}^{n} \binom{n}{k},$$

which, by means of Newton's formula for sum of binomials :

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = (x+y)^n,$$

can be rewritten as : 2^n.

- The worst-case time (and space) complexity of DynPro is in $O^*(2^n)$.

About permutation problems...

- Analyse of the worst-case time complexity...

$$Opt[S] = \min_{t \in S}\{\max\big(Opt[S - \{t\}]; f_t(P(S))\big)\}$$

- Usefull note : the computation of one $Opt[]$ can be done in $O(n)$ time.
- Fundamental question : how many computations of $Opt[]$ have to be done ?
- Generation of all subsets of a size $k \leq n$,

$$\sum_{k=0}^{n} \binom{n}{k},$$

which, by means of Newton's formula for sum of binomials :

$$\sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k} = (x+y)^n,$$

can be rewritten as : 2^n.
- The worst-case time (and space) complexity of `DynPro` is in $O^*(2^n)$,

About permutation problems...

- This improves upon the time complexity of ENUM for the permutation problem ($O^*(n!)$),
- This Dynamic Programming algorithm has been presented by Fomin and Kratsch [3]... this is *dynamic programming accross the subsets.*

[3] Fomin F, Kratsch D (2010) Exact Exponential Algorithms. Springer

About permutation problems...

- This improves upon the time complexity of ENUM for the permutation problem ($O^*(n!)$),
- This Dynamic Programming algorithm has been presented by Fomin and Kratsch [3]... this is *dynamic programming accross the subsets*.

[3] Fomin F, Kratsch D (2010) Exact Exponential Algorithms. Springer

Dynamic Programming AtS

- Applicable on *decomposable* scheduling problems ($C(S) = \sum_{i \in S} p_i$),
- Works on the following problems : $1|dec|f_{max}$, $1|dec|\sum_i f_i$, $1|prec|\sum_i w_i C_i$, $1|d_i|\sum_i w_i U_i$, $1|d_i|\sum_i w_i T_i$...
  ... $O^*(2^n)$ time and space.
- DPAtS can be extended ([6]) : a *Pareto Dynamic Programming* enables to derive :

Problem	wctc	wcsc
$F3\|\|C_{max}$	$O^*(3^n)$	$O^*(3^n)$
$F3\|\|f_{max}$	$O^*(5^n)$	$O^*(5^n)$
$F3\|\|\sum_i f_i$	$O^*(5^n)$	$O^*(5^n)$

[6] L. Shang, C. Lenté, M. Liedloff, V.T'kindt (2018). An exponential dynamic programming algorithm for the 3-machine flowshop scheduling problem to minimize the makespan, Journal of Scheduling, 21(2) :227-233.

Dynamic Programming AtS

- Applicable on *decomposable* scheduling problems ($C(S) = \sum_{i \in S} p_i$),
- Works on the following problems : $1|dec|f_{max}$, $1|dec|\sum_i f_i$, $1|prec|\sum_i w_i C_i$, $1|d_i|\sum_i w_i U_i$, $1|d_i|\sum_i w_i T_i$...
  ... $O^*(2^n)$ time and space.
- DPAtS can be extended ([6]) : a *Pareto Dynamic Programming* enables to derive :

Problem	wctc	wcsc
$F3\|\|C_{max}$	$O^*(3^n)$	$O^*(3^n)$
$F3\|\|f_{max}$	$O^*(5^n)$	$O^*(5^n)$
$F3\|\|\sum_i f_i$	$O^*(5^n)$	$O^*(5^n)$

[6] L. Shang, C. Lenté, M. Liedloff, V.T'kindt (2018). An exponential dynamic programming algorithm for the 3-machine flowshop scheduling problem to minimize the makespan, Journal of Scheduling, 21(2) :227-233.

Dynamic Programming AtS

- Applicable on *decomposable* scheduling problems ($C(S) = \sum_{i \in S} p_i$),
- Works on the following problems : $1|dec|f_{max}$, $1|dec|\sum_i f_i$, $1|prec|\sum_i w_i C_i$, $1|d_i|\sum_i w_i U_i$, $1|d_i|\sum_i w_i T_i$...

  ... $O^*(2^n)$ time and space.
- DPAtS can be extended ([6]) : a *Pareto Dynamic Programming* enables to derive :

Problem	wctc	wcsc
$F3\|\|C_{max}$	$O^*(3^n)$	$O^*(3^n)$
$F3\|\|f_{max}$	$O^*(5^n)$	$O^*(5^n)$
$F3\|\|\sum_i f_i$	$O^*(5^n)$	$O^*(5^n)$

[6] L. Shang, C. Lenté, M. Liedloff, V.T'kindt (2018). An exponential dynamic programming algorithm for the 3-machine flowshop scheduling problem to minimize the makespan, Journal of Scheduling, 21(2) :227-233.

Pareto Dynamic Programming

- Why a need for generalization ?
- The $1||f_{max}$ problem is *decomposable* but, for instance, the $F3||C_{max}$ is not,

  $F3||C_{max}$: Let n jobs to be scheduled on 3 machines (same routing from M_1 to M_3). Each job i is defined by processing times $p_{i,j}, 1 \leq j \leq 3$ and the goal is to find the permutation which minimizes $C_{max} = \max_i(C_{i,3})$.
- The intuition : when computing "$Opt[S] = \min_{t \in S}\{\max(Opt[S - \{t\}]; f_t(P(S)))\}$", many sequences $(S - \{t\})$ (at most 2^n) with different C^2_{max} and C^3_{max} must be kept in memory.

Pareto Dynamic Programming

- Why a need for generalization ?
- The $1||f_{max}$ problem is *decomposable* but, for instance, the $F3||C_{max}$ is not,

  $\underline{F3||C_{max}}$: Let n jobs to be scheduled on 3 machines (same routing from M_1 to M_3). Each job i is defined by processing times $p_{i,j}$, $1 \leq j \leq 3$ and the goal is to find the permutation which minimizes $C_{max} = \max_i(C_{i,3})$.

- The intuition : when computing "$Opt[S] = \min_{t \in S}\{\max(Opt[S-\{t\}]; f_t(P(S)))\}$", many sequences $(S-\{t\})$ (at most 2^n) with different C^2_{max} and C^3_{max} must be kept in memory.

Pareto Dynamic Programming

- Why a need for generalization ?
- The $1||f_{max}$ problem is *decomposable* but, for instance, the $F3||C_{max}$ is not,

  $F3||C_{max}$: Let n jobs to be scheduled on 3 machines (same routing from M_1 to M_3). Each job i is defined by processing times $p_{i,j}$, $1 \leq j \leq 3$ and the goal is to find the permutation which minimizes

  $$C_{max} = \max_i(C_{i,3}).$$

- The intuition : when computing "$Opt[S] = \min_{t \in S}\{\max\big(Opt[S-\{t\}]; f_t(P(S))\big)\}$", many sequences $(S-\{t\})$ (at most 2^n) with different C^2_{max} and C^3_{max} must be kept in memory.

Branch-and-... What ? !

- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the a **worst-case complexity**,
- A BaR algorithm implements three components :
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-... What ? !

- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the **a worst-case complexity**,
- A BaR algorithm implements three components :
  - [illegible]
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-... What ? !

- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the **a worst-case complexity**,
- A BaR algorithm implements three components :
  - A branching rule,
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-... What ? !

- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the a **worst-case complexity**,
- A BaR algorithm implements three components :
  - A branching rule,
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-... What ? !

- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the a **a worst-case complexity**,
- A BaR algorithm implements three components :
  - A branching rule,
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-... What ? !

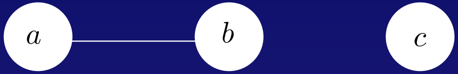
- Branch-and-Reduce (BaR) ressembles to known exact algorithms like Branch-and-Bound or Branch-and-Cut...
- BaR are tree-search based algorithms for which we try to reduce the **a worst-case complexity**,
- A BaR algorithm implements three components :
  - A branching rule,
  - A reduction rule at each node,
  - A stopping rule.

Branch-and-Reduce and the MIS

- Consider the *Maximum Independent Set* (MIS) problem :
  Let $G = (V, E)$ be an undirected graph,
  An independent set S is a set of vertices such that no two vertices from S are connected by an edge,
  The MIS problem consists in finding S with a maximum cardinality,

Branch-and-Reduce and the MIS

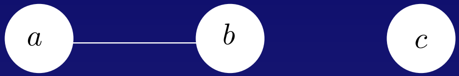
- First case : the degree $d(v) \leq 1, \forall v \in V$.



- Add any vertex v with $d(v) = 0$ into S,
- Add a vertex v with $d(v) = 1$ into S and remove the linked vertex (repeat),

Branch-and-Reduce and the MIS

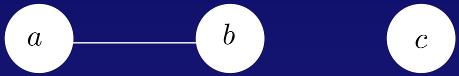
- First case : the degree $d(v) \leq 1, \forall v \in V$.



- Add any vertex v with $d(v) = 0$ into S,
- Add a vertex v with $d(v) = 1$ into S and remove the linked vertex (repeat),

Branch-and-Reduce and the MIS

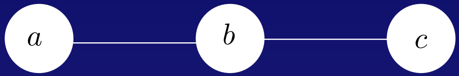
- First case : the degree $d(v) \leq 1, \forall v \in V$.



- Add any vertex v with $d(v) = 0$ into S,
- Add a vertex v with $d(v) = 1$ into S and remove the linked vertex (repeat),

Branch-and-Reduce and the MIS

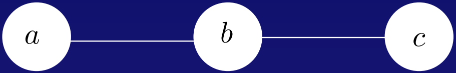
- Second case : the degree $d(v) \leq 2$, $\forall v \in V$.



- The graph is a set of chains,
- By testing, for each chain, if a vertex is in S, the problem can be solved in $O(|V|)$ time.

Branch-and-Reduce and the MIS

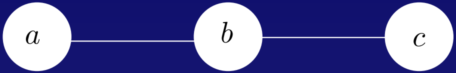
- Second case : the degree $d(v) \leq 2, \forall v \in V$.



- The graph is a set of chains,
- By testing, for each chain, if a vertex is in S, the problem can be solved in $O(|V|)$ time.

Branch-and-Reduce and the MIS

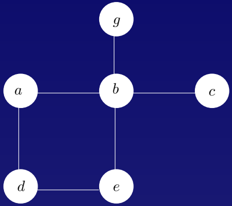
- Second case : the degree $d(v) \leq 2, \forall v \in V$.



- The graph is a set of chains,
- By testing, for each chain, if a vertex is in S, the problem can be solved in $O(|V|)$ time.

Branch-and-Reduce and the MIS

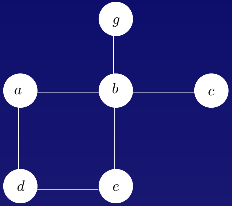
- General case : the maximum degree of vertices is at least 3.



- Let us consider a `BraRed` algorithm with the following branching rule :
  - Select the vertex v of maximum degree,
  - Create a child node with $v \in S$ and a child node with $v \notin S$.

Branch-and-Reduce and the MIS

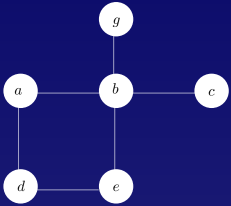
- General case : the maximum degree of vertices is at least 3.



- Let us consider a BraRed algorithm with the following branching rule :
  - Select the vertex v of maximum degree,
  - Create a child node with $v \in S$ and a child node with $v \notin S$.

Branch-and-Reduce and the MIS

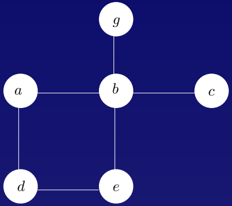
- General case : the maximum degree of vertices is at least 3.



- Let us consider a BraRed algorithm with the following branching rule :
  - Select the vertex v of maximum degree,
  - Create a child node with $v \in S$ and a child node with $v \notin S$.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.



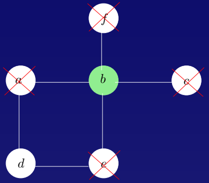
- Let us consider a BraRed algorithm with the following branching rule :
  - Select the vertex v of maximum degree,
  - Create a child node with $v \in S$ and a child node with $v \notin S$.

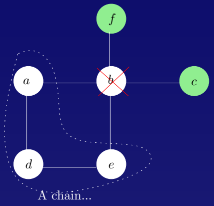
Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,
- Case 1 : $b \in S$, then a, c, e and f are removed. Vertex $d \in S$ by deduction.
- Case 2 : $b \notin S$, then c and f have degree 0 and are put in S. Vertices a, d, e form a graphe of max degree 2... solvable in polynomial time.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,

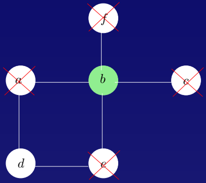


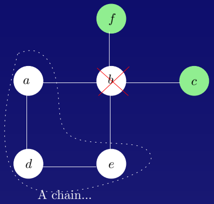


- Case 1 : $b \in S$, then a, c, e and f are removed. Vertex $d \in S$ by deduction.
- Case 2 : $b \notin S$, then c and f have degree 0 and are put in S. Vertices a, d, e form a graphe of max degree 2... solvable in polynomial time.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,

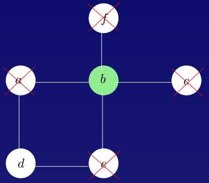


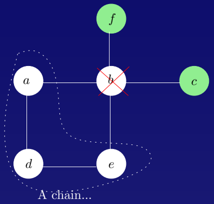


- Case 1 : $b \in S$, then a, c, e and f are removed. Vertex $d \in S$ by deduction.
- Case 2 : $b \notin S$, then c and f have degree 0 and are put in S. Vertices a, d, e form a graphe of max degree 2... solvable in polynomial time.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,

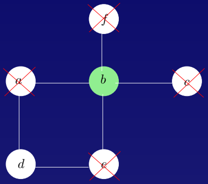


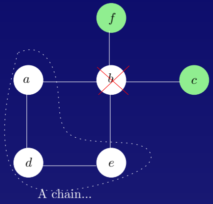


- Case 1 : $b \in S$, then a, c, e and f are removed. Vertex $d \in S$ by deduction.
- Case 2 : $b \notin S$, then c and f have degree 0 and are put in S. Vertices a, d, e form a graphe of max degree 2... solvable in polynomial time.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,

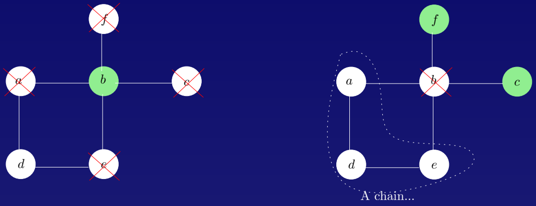




- In that case 2 nodes have been built.
- Reduction rule : when a decision is taken on a vertex v, decisions are taken for all its neighborhood,
- Stopping rule : for a node, stop branching as far as the maximum degree is 2.

Branch-and-Reduce and the MIS

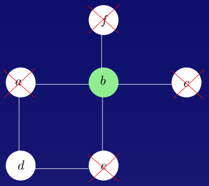
- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,

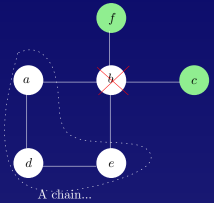


- In that case 2 nodes have been built.
- Reduction rule : when a decision is taken on a vertex v, decisions are taken for all its neighborhood,
- Stopping rule : for a node, stop branching as far as the maximum degree is 2.

Branch-and-Reduce and the MIS

- General case : the maximum degree of vertices is at least 3.
- Select vertex b of degree 4,





- In that case 2 nodes have been built.
- Reduction rule : when a decision is taken on a vertex v, decisions are taken for all its neighborhood,
- Stopping rule : for a node, stop branching as far as the maximum degree is 2.

Branch-and-Reduce and the MIS

- The BraRed algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) < 3$, solves the problem in polynomial time on the
- The above processing is applied on any unbranched node in BraRed.

Branch-and-Reduce and the MIS

- The `BraRed` algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) \leq 2$, solves the problem in polynomial time on the
- The above processing is applied on any unbranched node in `BraRed`.

Branch-and-Reduce and the MIS

- The BraRed algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) < 3$, solves the problem in polynomial time at the current node.
- The above processing is applied on any unbranched node in BraRed.

Branch-and-Reduce and the MIS

- The `BraRed` algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) < 3$, solves the problem in polynomial time at the current node.
- The above processing is applied on any unbranched node in `BraRed`.

Branch-and-Reduce and the MIS

- The BraRed algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) < 3$, solves the problem in polynomial time at the current node.
- The above processing is applied on any unbranched node in BraRed.

Branch-and-Reduce and the MIS

- The `BraRed` algorithm (main iterated loop) :
  - Put all vertices of degree 0 or 1 into S,
  - Let v be the vertex with maximum degree :
    - if $d(v) \geq 3$, create two child nodes : one with $v \in S$, another with $v \notin S$. Propagate to its neighborhood.
    - if $d(v) < 3$, solves the problem in polynomial time at the current node.
- The above processing is applied on any unbranched node in `BraRed`.

Branch-and-Reduce and the MIS

- What is the worst-case complexity of `BraRed` ?
- Let us observe the branching rule : $T(n)$ is the time required to solve a problem with n vertices,
- We can state that :

$$T(n) \leq T(n-1-d(v)) + T(n-1)$$

with v the vertex selected for branching.

- The worst case is obtained when $d(v)$ is minimal, *i.e.* $d(v) = 3$.
- So, in the worst case the time complexity for solving the problem is $T(n) = T(n-4) + T(n-1)$ with $n = |V|$.

Branch-and-Reduce and the MIS

- What is the worst-case complexity of `BraRed` ?
- Let us observe the branching rule : $T(n)$ is the time required to solve a problem with n vertices,
- We can state that :

$$T(n) \leq T(n-1-d(v)) + T(n-1)$$

with v the vertex selected for branching.
- The worst case is obtained when $d(v)$ is minimal, *i.e.* $d(v) = 3$.
- So, in the worst case the time complexity for solving the problem is $T(n) = T(n-4) + T(n-1)$ with $n = |V|$.

Branch-and-Reduce and the MIS

- What is the worst-case complexity of `BraRed` ?
- Let us observe the branching rule : $T(n)$ is the time required to solve a problem with n vertices,
- We can state that :

$$T(n) \leq T(n-1-d(v)) + T(n-1)$$

with v the vertex selected for branching.

- The worst case is obtained when $d(v)$ is minimal, *i.e.* $d(v) = 3$.
- So, in the worst case the time complexity for solving the problem is $T(n) = T(n-4) + T(n-1)$ with $n = |V|$.

Branch-and-Reduce and the MIS

- What is the worst-case complexity of `BraRed` ?
- Let us observe the branching rule : $T(n)$ is the time required to solve a problem with n vertices,
- We can state that :

$$T(n) \leq T(n-1-d(v)) + T(n-1)$$

with v the vertex selected for branching.

- The worst case is obtained when $d(v)$ is minimal, *i.e.* $d(v) = 3$.
- So, in the worst case the time complexity for solving the problem is $T(n) = T(n-4) + T(n-1)$ with $n = |V|$.

Branch-and-Reduce and the MIS

- What is the worst-case complexity of `BraRed` ?
- Let us observe the branching rule : $T(n)$ is the time required to solve a problem with n vertices,
- We can state that :

$$T(n) \leq T(n-1-d(v)) + T(n-1)$$

  with v the vertex selected for branching.
- The worst case is obtained when $d(v)$ is minimal, *i.e.* $d(v) = 3$.
- So, in the worst case the time complexity for solving the problem is $T(n) = T(n-4) + T(n-1)$ with $n = |V|$.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?

  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$

  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?
  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$
  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?

  By assuming that $T(n) = x^n$, we can write :

  $x^n = x^{n-4} + x^{n-1}$

  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for BraRed,
- $O^*(1.3803^n)$ is not bad. Also, BraRed has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?

  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$

  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?
  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$
  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?
  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$
  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce and the MIS

- How can we recursively evaluate $T(n) = T(n-4) + T(n-1)$?
  By assuming that $T(n) = x^n$, we can write :
  $x^n = x^{n-4} + x^{n-1}$
  $\Leftrightarrow 1 = x^{-4} + x^{-1}$
- Then, compute the largest zero of the above function,
- By using a solver like Matlab (for instance), we obtain $O^*(1.3803^n)$ as the worst-case time complexity for `BraRed`,
- $O^*(1.3803^n)$ is not bad. Also, `BraRed` has a polynomial space complexity,
- Notice that this is an upper bound (not tight at all), that could be refined.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Let us consider the $1|d_i|\sum_i T_i$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i, and a due date d_i,
  - $T_i = \max(0; C_i - d_i)$ is its tardiness,
  - Goal : Find the permutation which minimizes $\sum_i T_i$
- The worst-case complexity of `ENUM` is in $O^*(n!)$ time.
- The worst-case complexity of `DPAtS` is in $O^*(2^n)$ time and space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Let us consider the $1|d_i|\sum_i T_i$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i, and a due date d_i,
  - $T_i = \max(0; C_i - d_i)$ is its tardiness,
  - Goal : Find the permutation which minimizes $\sum_i T_i$.
- The worst-case complexity of ENUM is in $O^*(n!)$ time.
- The worst-case complexity of DPAtS is in $O^*(2^n)$ time and space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Let us consider the $1|d_i|\sum_i T_i$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i, and a due date d_i,
  - $T_i = \max(0; C_i - d_i)$ is its tardiness,
  - Goal : Find the permutation which minimizes $\sum_i T_i$.
- The worst-case complexity of ENUM is in $O^*(n!)$ time.
- The worst-case complexity of DPAtS is in $O^*(2^n)$ time and space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Let us consider the $1|d_i|\sum_i T_i$ scheduling problem,
- n jobs to be processed by a single machine. Each job i is defined by :
  - a processing time p_i, and a due date d_i,
  - $T_i = \max(0; C_i - d_i)$ is its tardiness,
  - Goal : Find the permutation which minimizes $\sum_i T_i$.
- The worst-case complexity of `ENUM` is in $O^*(n!)$ time.
- The worst-case complexity of `DPAtS` is in $O^*(2^n)$ time and space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume : $p_1 \geq p_2 \geq ... \geq p_n$ and $[k]$ is the job in position k in EDD,
- To define the branching scheme, we make use of ([8]) :

Property

Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as

$B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ *and* $A_1(h) = \{[h+1], \ldots, [n]\}$.

- Worst case : $d_1 \leq d_2 \leq ... \leq d_n$.

[8] E. L. Lawler (1977), "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* 1, 331–342.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume : $p_1 \geq p_2 \geq ... \geq p_n$ and $[k]$ is the job in position k in EDD,
- To define the branching scheme, we make use of ([8]) :

Property

Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as

$B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ *and* $A_1(h) = \{[h+1], \ldots, [n]\}$.

- Worst case : $d_1 \leq d_2 \leq ... \leq d_n$.

[8] E. L. Lawler (1977), "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* 1, 331–342.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume : $p_1 \geq p_2 \geq ... \geq p_n$ and $[k]$ is the job in position k in EDD,
- To define the branching scheme, we make use of ([8]) :

Property

Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as $B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ and $A_1(h) = \{[h+1], \ldots, [n]\}$.

- Worst case : $d_1 \leq d_2 \leq ... \leq d_n$.

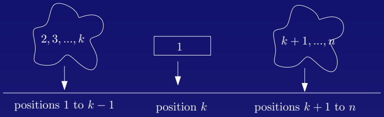
[8] E. L. Lawler (1977), "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* 1, 331–342.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume (wc) : $p_1 \geq p_2 \geq ... \geq p_n$ and $d_1 \leq d_2 \leq ... \leq d_n$,
- Branching scheme :
- Remark : When job 1 is branched on position k two subproblems of size $(k-1)$ and $(n-k)$ have to be solved.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

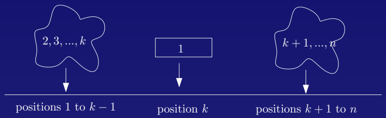
- We assume (wc) : $p_1 \geq p_2 \geq ... \geq p_n$ and $d_1 \leq d_2 \leq ... \leq d_n$,
- Branching scheme :

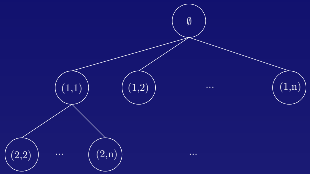


- Remark : When job 1 is branched on position k two subproblems of size $(k-1)$ and $(n-k)$ have to be solved.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume (wc) : $p_1 \geq p_2 \geq ... \geq p_n$ and $d_1 \leq d_2 \leq ... \leq d_n$,
- Branching scheme :

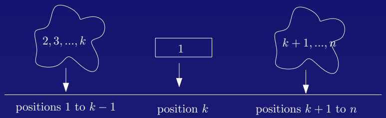


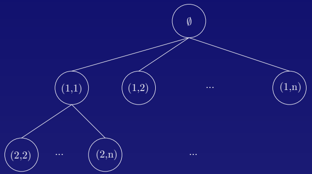


- Remark : When job 1 is branched on position k two subproblems of size $(k-1)$ and $(n-k)$ have to be solved.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We assume (wc) : $p_1 \geq p_2 \geq ... \geq p_n$ and $d_1 \leq d_2 \leq ... \leq d_n$,
- Branching scheme :





- Remark : When job 1 is branched on position k two subproblems of size $(k-1)$ and $(n-k)$ have to be solved.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

Exercice.

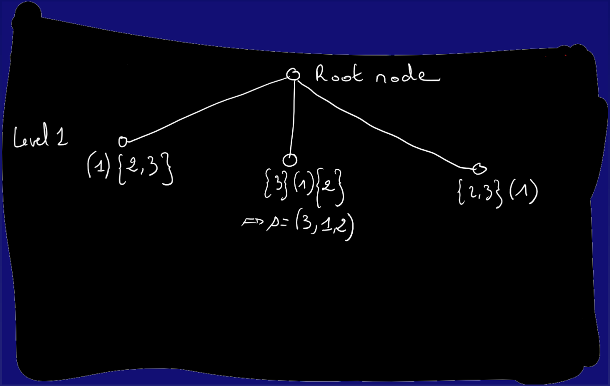
Build the search tree on the following instance :
$n = 3$, $[p_i]_i = [5; 4; 3]$, $[d_i]_i = [6; 8; 10]$,

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- First level, the longest job is job 1 : it can be scheduled in positions 1, 2 or 3 leading to the following nodes,

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- First level, the longest job is job 1 : it can be scheduled in positions 1, 2 or 3 leading to the following nodes,

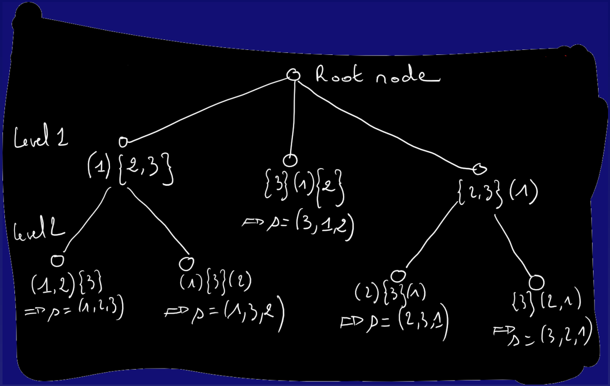


Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Second level, the longest job is job 2,

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- Second level, the longest job is job 2,



Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We get the following recursive relation :

$$T(n) = 2T(n-1)+2T(n-2)+...+2T(2)+2T(1)+O(p(n))$$

$$\Leftrightarrow T(n) = 3T(n-1) + O(p(n))$$

- This yields $O^*(3^n)$ worst-case time complexity, and polynomial space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We get the following recursive relation :

$$T(n) = 2T(n-1)+2T(n-2)+...+2T(2)+2T(1)+O(p(n))$$
$$\Leftrightarrow T(n) = 3T(n-1) + O(p(n))$$

- This yields $O^*(3^n)$ worst-case time complexity, and polynomial space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- We get the following recursive relation :

$$T(n) = 2T(n-1)+2T(n-2)+...+2T(2)+2T(1)+O(p(n))$$
$$\Leftrightarrow T(n) = 3T(n-1) + O(p(n))$$

- This yields $O^*(3^n)$ worst-case time complexity, and polynomial space.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- By making use of the following property ([9])...

Property

For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated.

... we can derive that :

$$T(n) = 2T(n-1)+2T(n-3)+...+2T(4)+2T(2)+O(p(n))$$
$$\Leftrightarrow T(n) = 2T(n-1) + T(n-2) + O(p(n))$$

- This yields $O^*(2.4143^n)$ worst-case time complexity, and polynomial space.

[9] W. Szwarc and S. Mukhopadhyay (1996), "Decomposition of the single machine total tardiness problem", *Operations Research Letters* 19, 243–250.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- By making use of the following property ([9])...

Property

For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated.

... we can derive that :

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + O(p(n))$$

$$\Leftrightarrow T(n) = 2T(n-1) + T(n-2) + O(p(n))$$

- This yields $O^*(2.4143^n)$ worst-case time complexity, and polynomial space.

[9] W. Szwarc and S. Mukhopadhyay (1996), “Decomposition of the single machine total tardiness problem”, *Operations Research Letters* 19, 243–250.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- By making use of the following property ([9])...

Property

For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated.

... we can derive that :

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + O(p(n))$$
$$\Leftrightarrow T(n) = 2T(n-1) + T(n-2) + O(p(n))$$

- This yields $O^*(2.4143^n)$ worst-case time complexity, and polynomial space.

[9] W. Szwarc and S. Mukhopadhyay (1996), "Decomposition of the single machine total tardiness problem", *Operations Research Letters* 19, 243–250.

Branch-and-Reduce : the $1|d_i|\sum_i T_i$

- By making use of the following property ([9])...

Property

For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated.

... we can derive that :

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + O(p(n))$$
$$\Leftrightarrow T(n) = 2T(n-1) + T(n-2) + O(p(n))$$

- This yields $O^*(2.4143^n)$ worst-case time complexity, and polynomial space.

[9] W. Szwarc and S. Mukhopadhyay (1996), "Decomposition of the single machine total tardiness problem", *Operations Research Letters* 19, 243–250.

Branch-and-Reduce : add-ons

- Changing the way to do the analysis : *Measure and Conquer*,
- Pruning nodes by use of an exponential memory : *Memo(r)ization*,
- Pruning nodes without the use of an exponential memory : *Merging*,

[5] M. Garraffa, L. Shang, F. Della Croce, V. T'Kindt (2018). An exact exponential branch-and-merge algorithm for the single machine tardiness problem. Theoretical Computer Science, 745, 133-149.

Branch-and-Reduce : add-ons

- Changing the way to do the analysis : *Measure and Conquer*,
- Pruning nodes by use of an exponential memory : *Memo(r)ization*,
- Pruning nodes without the use of an exponential memory : *Merging*,

[5] M. Garraffa, L. Shang, F. Della Croce, V. T'Kindt (2018). An exact exponential branch-and-merge algorithm for the single machine tardiness problem. Theoretical Computer Science, 745, 133-149.

Branch-and-Reduce : add-ons

- Changing the way to do the analysis : *Measure and Conquer*,
- Pruning nodes by use of an exponential memory : *Memo(r)ization*,
- Pruning nodes without the use of an exponential memory : *Merging*,

  $1|d_i|\sum_i T_i$: $O^*(2^n)$ time and poly space when DPAtS uses $O^*(2^n)$ space ([5]).

[5] M. Garraffa, L. Shang, F. Della Croce, V. T'Kindt (2018). An exact exponential branch-and-merge algorithm for the single machine tardiness problem. Theoretical Computer Science, 745, 133-149.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Branch-and-Reduce : to conclude

- BaR sounds like BaB (for instance), however there are different,
- Open question : what would be the practical efficiency of a BaR with all the materials of a BaB included ? ? ? ?
- The complexity analysis can be very complicated (*Measure and Conquer*, *merging*, ...),
- It is hard to get tight upper bounds on the worst-case complexity,
- Some researches focus on getting **lower bounds** on that complexity,
- Important point : leads to polynomial space ETA.

Sort & Search : the principles

- It is an old technique which consists in **sorting** “data” to make the **search** for an optimal solution more efficient,
- It has been proposed by Horowitz and Sahni ([6]) to solve the knapsack problem,
- Let us start with the `KNAPSACK` problem,
  - Let be $O = \{o_1, \ldots, o_n\}$ a set of n objects,
  - Each object o_i is defined by a value $v(o_i)$ and a weight $w(o_i)$, $1 \leq i \leq n$,
  -
  - Goal : Find $O' \subseteq O$ such that $\sum_{o \in O'} w(o) \leq W$ and $\sum_{o \in O'} v(o)$ is maximum.
- We can easily show that `ENUM` is in $O^*(2^n)$ time,

[6] E. Horowitz and G. Sahni. Computing partitions with applications to the knapsack problem, Journal of the ACM, vol 21, pp.277-292, 1974

Sort & Search : the principles

- It is an old technique which consists in **sorting** “data” to make the **search** for an optimal solution more efficient,
- It has been proposed by Horowitz and Sahni ([6]) to solve the knapsack problem,
- Let us start with the KNAPSACK problem,
  - Let be $O=\{o_1,\ldots,o_n\}$ a set of n objects,
  - Each object o_i is defined by a value $v(o_i)$ and a weight $w(o_i)$, $1\leq i\leq n$,
  - Goal : Find $O'\subseteq O$ such that $\sum_{o\in O'} w(o)\leq W$ and $\sum_{o\in O'} v(o)$ is maximum.
- We can easily show that $\mathtt{ENUM}$ is in $O^*(2^n)$ time,

[6] E. Horowitz and G. Sahni. Computing partitions with applications to the knapsack problem, Journal of the ACM, vol 21, pp.277-292, 1974

Sort & Search : the principles

- It is an old technique which consists in **sorting** “data” to make the **search** for an optimal solution more efficient,
- It has been proposed by Horowitz and Sahni ([6]) to solve the knapsack problem,
- Let us start with the KNAPSACK problem,
  - Let be $O = \{o_1, \ldots, o_n\}$ a set of n objects,
  - Each object o_i is defined by a value $v(o_i)$ and a weight $w(o_i)$, $1 \leq i \leq n$,
  - The, integer, capacity W of the knapsack.
  - Goal : Find $O' \subseteq O$ such that $\sum_{o \in O'} w(o) \leq W$ and $\sum_{o \in O'} v(o)$ is maximum.
- We can easily show that ENUM is in $O^*(2^n)$ time,

[6] E. Horowitz and G. Sahni. Computing partitions with applications to the knapsack problem, Journal of the ACM, vol 21, pp.277-292, 1974

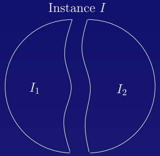
Sort & Search : the principles

- It is an old technique which consists in **sorting** "data" to make the **search** for an optimal solution more efficient,
- It has been proposed by Horowitz and Sahni ([6]) to solve the knapsack problem,
- Let us start with the `KNAPSACK` problem,
  - Let be $O = \{o_1, \ldots, o_n\}$ a set of n objects,
  - Each object o_i is defined by a value $v(o_i)$ and a weight $w(o_i)$, $1 \leq i \leq n$,
  - The, integer, capacity W of the knapsack.
  - Goal : Find $O' \subseteq O$ such that $\sum_{o \in O'} w(o) \leq W$ and $\sum_{o \in O'} v(o)$ is maximum.
- We can easily show that `ENUM` is in $O^*(2^n)$ time,

[6] E. Horowitz and G. Sahni. Computing partitions with applications to the knapsack problem, Journal of the ACM, vol 21, pp.277-292, 1974

Sort & Search : the principles

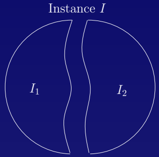
- The idea is the following : separate the instance into **2** sub-instances,



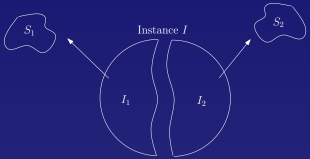
- Then, enumerate all partial solutions from I_1 and all partial solutions from I_2,

Sort & Search : the principles

- The idea is the following : separate the instance into **2** sub-instances,

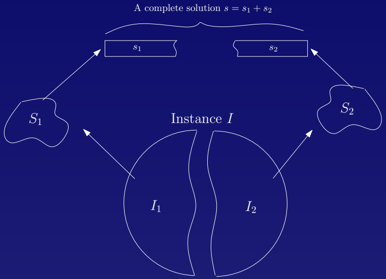


- Then, enumerate all partial solutions from I_1 and all partial solutions from I_2,



Sort & Search : the principles

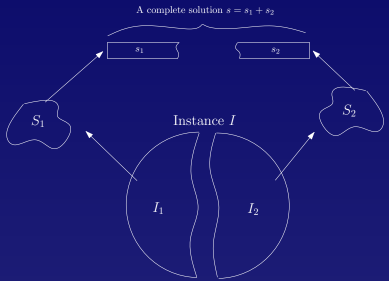
- By recombination of partial solutions, find the optimal solution of the initial problem



- The combinatoric appears when building S_1 and S_2 by enumeration (*sort* phase) and when finding in these sets the optimal solution (*search* phase).

Sort & Search : the principles

- By recombination of partial solutions, find the optimal solution of the initial problem



- The combinatoric appears when building S_1 and S_2 by enumeration (*sort* phase) and when finding in these sets the optimal solution (*search* phase).

Sort & Search : the principles

- The idea : cut the cake into two equal-size pieces and just pay for one (but take both !),
- Let us go back to the KNAPSACK and see how it works on an example,
- We have $n = 6$, $O = \{a, b, c, d, e, f\}$ and $W = 9$.

O	a	b	c	d	e	f
v	3	4	2	5	1	3
w	4	2	1	3	2	5

$O_1 = \{a, b, c\}$ $O_2 = \{d, e, f\}$

- Next, we enumerate the set of all possible assignments for O_1 (Table T_1),

T_1	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$\sum v$	0	3	4	2	7	5	6	9
$\sum w$	0	4	2	1	6	5	3	7

Sort & Search : the principles

- The idea : cut the cake into two equal-size pieces and just pay for one (but take both !),
- Let us go back to the `KNAPSACK` and see how it works on an example,
- We have $n = 6$, $O = \{a, b, c, d, e, f\}$ and $W = 9$.

O	a	b	c	d	e	f
v	3	4	2	5	1	3
w	4	2	1	3	2	5

$O_1 = \{a, b, c\}$ $\quad O_2 = \{d, e, f\}$

- Next, we enumerate the set of all possible assignments for O_1 (Table T_1),

T_1	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$\sum v$	0	3	4	2	7	5	6	9
$\sum w$	0	4	2	1	6	5	3	7

Sort & Search : the principles

- The idea : cut the cake into two equal-size pieces and just pay for one (but take both !),
- Let us go back to the KNAPSACK and see how it works on an example,
- We have $n = 6$, $O = \{a, b, c, d, e, f\}$ and $W = 9$.

O	a	b	c	d	e	f
v	3	4	2	5	1	3
w	4	2	1	3	2	5

$O_1 = \{a, b, c\}$ $\quad O_2 = \{d, e, f\}$

- Next, we enumerate the set of all possible assignments for O_1 (Table T_1),

T_1	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$\sum v$	0	3	4	2	7	5	6	9
$\sum w$	0	4	2	1	6	5	3	7

Sort & Search : the principles

- The idea : cut the cake into two equal-size pieces and just pay for one (but take both !),
- Let us go back to the KNAPSACK and see how it works on an example,
- We have $n = 6$, $O = \{a, b, c, d, e, f\}$ and $W = 9$.

O	a	b	c	d	e	f
v	3	4	2	5	1	3
w	4	2	1	3	2	5

$O_1 = \{a, b, c\}$ $\quad O_2 = \{d, e, f\}$

- Next, we enumerate the set of all possible assignments for O_1 (Table T_1),

T_1	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a, b\}$	$\{a, c\}$	$\{b, c\}$	$\{a, b, c\}$
$\sum v$	0	3	4	2	7	5	6	9
$\sum w$	0	4	2	1	6	5	3	7

Sort & Search : the principles

- Next, we do the same for O_2 (Table T_2),

T_2	$\emptyset$	$\{e\}$	$\{d\}$	$\{f\}$	$\{d,e\}$	$\{e,f\}$	$\{d,f\}$	$\{d,e,f\}$
$\sum v$	0	1	5	3	6	4	8	9
$\sum w$	0	2	3	5	5	7	8	10
ℓ_k	1	2	3	3	5	5	7	8

Note : In table T_2, columns are sorted by increasing order of $\sum w$.
Note : ℓ_k is the column number with maximum $\sum v$ "on the left" of the current column.

- That was the *Sort* phase !
- Running time (and space) should be "about" $2^{n/2}$,

Sort & Search : the principles

- Next, we do the same for O_2 (Table T_2),

T_2	$\emptyset$	$\{e\}$	$\{d\}$	$\{f\}$	$\{d,e\}$	$\{e,f\}$	$\{d,f\}$	$\{d,e,f\}$
$\sum v$	0	1	5	3	6	4	8	9
$\sum w$	0	2	3	5	5	7	8	10
ℓ_k	1	2	3	3	5	5	7	8

  Note : In table T_2, columns are sorted by increasing order of $\sum w$.
  Note : ℓ_k is the column number with maximum $\sum v$ "on the left" of the current column.

- That was the *Sort* phase !
- Running time (and space) should be "about" $2^{n/2}$,

Sort & Search : the principles

- Next, we do the same for O_2 (Table T_2),

T_2	$\emptyset$	$\{e\}$	$\{d\}$	$\{f\}$	$\{d,e\}$	$\{e,f\}$	$\{d,f\}$	$\{d,e,f\}$
$\sum v$	0	1	5	3	6	4	8	9
$\sum w$	0	2	3	5	5	7	8	10
ℓ_k	1	2	3	3	5	5	7	8

  Note : In table T_2, columns are sorted by increasing order of $\sum w$.
  Note : ℓ_k is the column number with maximum $\sum v$ "on the left" of the current column.
- That was the *Sort* phase !
- Running time (and space) should be "about" $2^{n/2}$,

Sort & Search : the principles

- Search phase can start,
- For any column $j \in T_1$, find the "best" complementing column $k \in T_2$,
- Best : column k which maximizes $\sum w$... then column ℓ_k will be the one which maximizes $\sum v$,

Sort & Search : the principles

- Search phase can start,
- For any column $j \in T_1$, find the "best" complementing column $k \in T_2$,
- Best : column k which maximizes $\sum w$... then column ℓ_k will be the one which maximizes $\sum v$,

Sort & Search : the principles

- Search phase can start,
- For any column $j \in T_1$, find the "best" complementing column $k \in T_2$,
- Best : column k which maximizes $\sum w$... then column ℓ_k will be the one which maximizes $\sum v$,

Sort & Search : the principles

Table 1

T_1	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
$\sum v$	0	3	4	2	7	5	6	9
$\sum w$	0	4	2	1	6	5	3	7

Table 2

T_2	$\emptyset$	$\{e\}$	$\{d\}$	$\{f\}$	$\{d,e\}$	$\{e,f\}$	$\{d,f\}$	$\{d,e,f\}$
$\sum v$	0	1	5	3	6	4	8	9
$\sum w$	0	2	3	5	5	7	8	10
ℓ_k	1	2	3	3	5	5	7	8

Search phase ($W = 9$)

j	$\emptyset$	$\{a\}$	$\{b\}$	$\{c\}$	$\{a,b\}$	$\{a,c\}$	$\{b,c\}$	$\{a,b,c\}$
k	$\{d,f\}$	$\{d,e\}$	$\{d,e\}$	$\{d,f\}$	$\{d\}$	$\{d\}$	$\{d,e\}$	$\{e\}$
$w(O'_j)+w(O'_k)$	8	9	7	9	9	8	8	9
$v(O'_j)+v(O'_{\ell_k})$	8	9	10	10	12	10	12	10

Consequently, the optimal solution has value 12 and is achieved with $\{a,b,d\}$ or $\{b,c,d,e\}$.

Sort & Search : formalization

- *Sort & Search* is a powerfull technique that can be applied to a lot of problems,
- Intuitively, to be applicable efficiently, problems must satisfy two properties :
  - Two partial solutions can be combined in polynomial time to get a feasible solution,
  - The Sort phase must enable to lead to a Search phase which ... tables.
- *Sort & Search*, as introduced by Horowitz and Sahni, can be applied to a class of problems called *Single Constraint Problems* (SCP),

Sort & Search : formalization

- *Sort & Search* is a powerfull technique that can be applied to a lot of problems,
- Intuitively, to be applicable efficiently, problems must satisfy two properties :
    1. Two partial solutions can be combined in polynomial time to get a feasible solution,
    2. The Sort phase must enable to lead to a Search phase which complexity does not exceed the one required to build the tables.
- *Sort & Search*, as introduced by Horowitz and Sahni, can be applied to a class of problems called *Single Constraint Problems* (SCP),

Sort & Search : formalization

- *Sort & Search* is a powerfull technique that can be applied to a lot of problems,
- Intuitively, to be applicable efficiently, problems must satisfy two properties :
  1. Two partial solutions can be combined in polynomial time to get a feasible solution,
  2. The Sort phase must enable to lead to a Search phase which complexity does not exceed the one required to build the tables.
- *Sort & Search*, as introduced by Horowitz and Sahni, can be applied to a class of problems called *Single Constraint Problems* (SCP),

Sort & Search : formalization

- *Sort & Search* is a powerfull technique that can be applied to a lot of problems,
- Intuitively, to be applicable efficiently, problems must satisfy two properties :
  1. Two partial solutions can be combined in polynomial time to get a feasible solution,
  2. The Sort phase must enable to lead to a Search phase which complexity does not exceed the one required to build the tables.
- *Sort & Search*, as introduced by Horowitz and Sahni, can be applied to a class of problems called *Single Constraint Problems* (SCP),

Sort & Search : formalization

- *Sort & Search* is a powerfull technique that can be applied to a lot of problems,
- Intuitively, to be applicable efficiently, problems must satisfy two properties :
  1. Two partial solutions can be combined in polynomial time to get a feasible solution,
  2. The Sort phase must enable to lead to a Search phase which complexity does not exceed the one required to build the tables.
- *Sort & Search*, as introduced by Horowitz and Sahni, can be applied to a class of problems called *Single Constraint Problems* (SCP),

Sort & Search : formalization

- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = ((b_1, b'_1), (b_2, b'_2) \ldots\ (b_{n_B}, b'_{n_B}))$ a table of n_B couples,
- Let f and g' be two functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$, increasing with respect to their last variable,
- The (SCP) :

$$\begin{aligned}&\text{Minimize } f(\vec{a}_j, b_k)\\ &\text{s.t.}\\ &\quad g'(\vec{a}_j, b'_k) \geq 0\\ &\quad \vec{a}_j \in A,\ (b_k, b'_k) \in B.\end{aligned}$$

- **There exists a *Sort & Search* algorithm in $O(n_B \log_2(n_B) + n_A \log_2(n_B))$ time and $O(n_A + n_B)$ space.**
- KNAPSACK : $n_A = n_B = 2^{\frac{n}{2}} \Rightarrow O^*(2^{\frac{n}{2}})$ time and space.

Sort & Search : formalization

- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = ((b_1, b'_1),\ (b_2, b'_2) \ldots\ (b_{n_B}, b'_{n_B}))$ a table of n_B couples,
- Let f and g' be two functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$, increasing with respect to their last variable,
- The (SCP) :

$$\begin{aligned}
&\text{Minimize } f(\vec{a}_j, b_k) \\
&\text{s.t.} \\
&\quad g'(\vec{a}_j, b'_k) \geq 0 \\
&\quad \vec{a}_j \in A,\ (b_k, b'_k) \in B.
\end{aligned}$$

- **There exists a** ***Sort & Search*** **algorithm in** $O(n_B \log_2(n_B) + n_A \log_2(n_B))$ **time and** $O(n_A + n_B)$ **space.**
- KNAPSACK : $n_A = n_B = 2^{\frac{n}{2}} \Rightarrow O^*(2^{\frac{n}{2}})$ time and space.

Sort & Search : formalization

- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = ((b_1, b'_1), (b_2, b'_2) \ldots\ (b_{n_B}, b'_{n_B}))$ a table of n_B couples,
- Let f and g' be two functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$, increasing with respect to their last variable,
- The (SCP) :

$$\begin{aligned}
&\text{Minimize } f(\vec{a_j}, b_k) \\
&\text{s.t.} \\
&\quad g'(\vec{a_j}, b'_k) \geq 0 \\
&\quad \vec{a_j} \in A,\ (b_k, b'_k) \in B.
\end{aligned}$$

- **There exists a *Sort & Search* algorithm in $O(n_B \log_2(n_B) + n_A \log_2(n_B))$ time and $O(n_A + n_B)$ space.**
- KNAPSACK : $n_A = n_B = 2^{\frac{n}{2}} \Rightarrow O^*(2^{\frac{n}{2}})$ time and space.

Sort & Search : generalization

- We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),
- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = (\vec{b}_1,\ \vec{b}_2, \ldots \vec{b}_{n_B})$ a table of n_B vectors $\vec{b}_k = (b_k^0, b_k^1, \ldots, b_k^{d_B})$ of dimension $d_B + 1$,
- Let f and g_ℓ $(1 \leq \ell \leq d_B)$ be $d_B + 1$ functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$ (increasing with respect to their last variable),
- The (MCP) is defined by :

$$\begin{aligned} &\text{Minimize } f(\vec{a}_j, b_k^0) \\ &\text{s.t.} \\ &\quad g_\ell(\vec{a}_j, b_k^\ell) \geq 0, \quad (1 \leq \ell \leq d_B) \\ &\quad \vec{a}_j \in A,\ \vec{b}_k \in B. \end{aligned}$$

Sort & Search : generalization

- We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),
- Let be $A = (\vec{a}_1,\ \vec{a}_2, \dots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = (\vec{b}_1,\ \vec{b}_2, \dots \vec{b}_{n_B})$ a table of n_B vectors $\vec{b}_k = (b_k^0, b_k^1, \dots, b_k^{d_B})$ of dimension $d_B + 1$,
- Let f and g_ℓ $(1 \leq \ell \leq d_B)$ be $d_B + 1$ functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$ (increasing with respect to their last variable),
- The (MCP) is defined by :

$$
\begin{aligned}
&\text{Minimize } f(\vec{a}_j, b_k^0) \\
&\text{s.t.} \\
&\qquad g_\ell(\vec{a}_j, b_k^\ell) \geq 0, \quad (1 \leq \ell \leq d_B) \\
&\qquad \vec{a}_j \in A,\ \vec{b}_k \in B.
\end{aligned}
$$

Sort & Search : generalization

- We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),
- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = (\vec{b}_1,\ \vec{b}_2, \ldots \vec{b}_{n_B})$ a table of n_B vectors $\vec{b}_k = (b_k^0, b_k^1, \ldots, b_k^{d_B})$ of dimension $d_B + 1$,
- Let f and g_ℓ $(1 \leq \ell \leq d_B)$ be $d_B + 1$ functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$ (increasing with respect to their last variable),
- The (MCP) is defined by :

$$\text{Minimize } f(\vec{a_j}, b_k^0)$$

s.t.

$$g_\ell(\vec{a_j}, b_k^\ell) \geq 0, \quad (1 \leq \ell \leq d_B)$$
$$\vec{a_j} \in A,\ \vec{b_k} \in B.$$

Sort & Search : generalization

- We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),
- Let be $A = (\vec{a}_1,\ \vec{a}_2, \dots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = (\vec{b}_1,\ \vec{b}_2, \dots \vec{b}_{n_B})$ a table of n_B vectors $\vec{b}_k = (b_k^0, b_k^1, \dots, b_k^{d_B})$ of dimension $d_B + 1$,
- Let f and g_ℓ $(1 \leq \ell \leq d_B)$ be $d_B + 1$ functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$ (increasing with respect to their last variable),
- The (MCP) is defined by :

$$\begin{aligned} &\text{Minimize } f(\vec{a}_j, b_k^0) \\ &\text{s.t.} \\ &\qquad g_\ell(\vec{a}_j, b_k^\ell) \geq 0, \quad (1 \leq \ell \leq d_B) \\ &\qquad \vec{a}_j \in A,\ \vec{b}_k \in B. \end{aligned}$$

Sort & Search : generalization

- We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),
- Let be $A = (\vec{a}_1,\ \vec{a}_2, \ldots \vec{a}_{n_A})$ a table of n_A vectors of dimension d_A,
- Let be $B = (\vec{b}_1,\ \vec{b}_2, \ldots \vec{b}_{n_B})$ a table of n_B vectors $\vec{b}_k = (b_k^0, b_k^1, \ldots, b_k^{d_B})$ of dimension $d_B + 1$,
- Let f and g_ℓ $(1 \leq \ell \leq d_B)$ be $d_B + 1$ functions from $\mathbb{R}^{d_A+1}$ to $\mathbb{R}$ (increasing with respect to their last variable),
- The (MCP) is defined by :

$$\begin{aligned}
&\text{Minimize } f(\vec{a}_j, b_k^0) \\
&\text{s.t.} \\
&\quad g_\ell(\vec{a}_j, b_k^\ell) \geq 0, \quad (1 \leq \ell \leq d_B) \\
&\quad \vec{a}_j \in A,\ \vec{b}_k \in B.
\end{aligned}$$

Sort & Search : generalization

- By means of appropriate data structures (*range trees*) and properties on *rectangular range queries*...
- ... we can establish a *Sort & Search* algorithm in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time and $O(n_B \log_2^{d_B-1}(n_B))$ space ([4]).

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

Sort & Search : generalization

- By means of appropriate data structures (*range trees*) and properties on *rectangular range queries*...
- ... we can establish a *Sort & Search* algorithm in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time and $O(n_B \log_2^{d_B-1}(n_B))$ space ([4]).

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

Sort & Search : an application

- Consider the $P3||C_{max}$ scheduling problem :
  - 3 identical machines are available to process n jobs,
  - Each job i is defined by a processing time p_i and can be processed by any of the 3 machines,
  - Find a schedule which minimizes the makespan $C_{max} = \max_i(C_i)$ with C_i the completion time of job i.
- This problem is $\mathcal{NP}$-hard.
- The worst-case time complexity of $\texttt{ENUM}$ is in $O^*(3^n)$,

Sort & Search : an application

- Consider the $P3||C_{max}$ scheduling problem :
  - 3 identical machines are available to process n jobs,
  - Each job i is defined by a processing time p_i and can be processed by any of the 3 machines,
  - Find a schedule which minimizes the makespan $C_{max} = \max_i(C_i)$ with C_i the completion time of job i.
- This problem is $\mathcal{NP}$-hard.
- The worst-case time complexity of ENUM is in $O^*(3^n)$,

Sort & Search : an application

- Consider the $P3||C_{max}$ scheduling problem :
  - 3 identical machines are available to process n jobs,
  - Each job i is defined by a processing time p_i and can be processed by any of the 3 machines,
  - Find a schedule which minimizes the makespan $C_{max} = \max_i(C_i)$ with C_i the completion time of job i.
- This problem is $\mathcal{NP}$-hard.
- The worst-case time complexity of ENUM is in $O^*(3^n)$,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \dots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \dots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \le j \le 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \le j \le 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

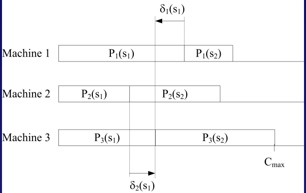
- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application (main lines)

- Let I be an instance with n jobs given in a set $\mathcal{J}$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $\mathcal{E}_1^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of I_1 $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule s_1^j containing the sequence of jobs on machines,
- Similarly, let be $\mathcal{E}_2^k$ a 3-partition of I_2 $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule s_2^k containing the sequence of jobs on machines,

Sort & Search : an application

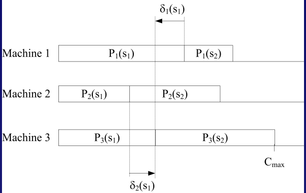
- The situation is pictured below (s_1 comes from I_1, s_2 comes from I_2),



- Let us state some necessary properties,
- Let $P_\ell(s)$ be the sum of processing times of jobs assigned to machine ℓ in s, $\ell \in [\![1, 3]\!]$,
- Let $P(s)$ be the sum of processing times of all jobs of s,

Sort & Search : an application

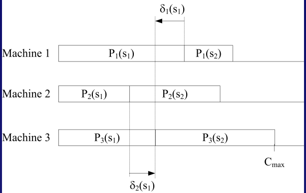
- The situation is pictured below (s_1 comes from I_1, s_2 comes from I_2),



- Let us state some necessary properties,
- Let $P_\ell(s)$ be the sum of processing times of jobs assigned to machine ℓ in s, $\ell \in [\![1, 3]\!]$,
- Let $P(s)$ be the sum of processing times of all jobs of s,

Sort & Search : an application

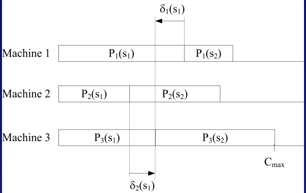
- The situation is pictured below (s_1 comes from I_1, s_2 comes from I_2),



- Let us state some necessary properties,
- Let $P_\ell(s)$ be the sum of processing times of jobs assigned to machine ℓ in s, $\ell \in [\![1, 3]\!]$,
- Let $P(s)$ be the sum of processing times of all jobs of s,

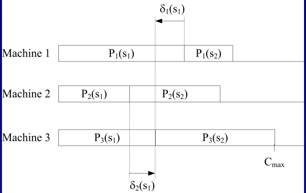
Sort & Search : an application

- The situation is pictured below (s_1 comes from I_1, s_2 comes from I_2),



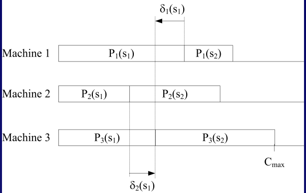
- Let us state some necessary properties,
- Let $P_\ell(s)$ be the sum of processing times of jobs assigned to machine ℓ in s, $\ell \in [\![1, 3]\!]$,
- Let $P(s)$ be the sum of processing times of all jobs of s,

Sort & Search : an application



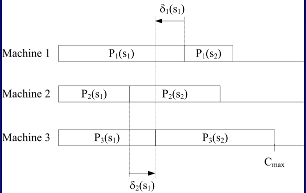
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $\sum_{\ell=1}^{3} \delta_\ell(s) = \sum_{\ell=1}^{2} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1 \leq \ell \leq 3}\,(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



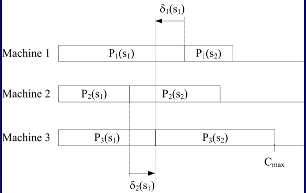
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1\leq\ell\leq3}(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



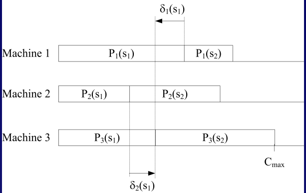
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1\leq\ell\leq3}(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



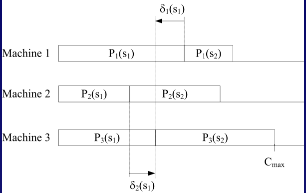
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1\leq\ell\leq 3}(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



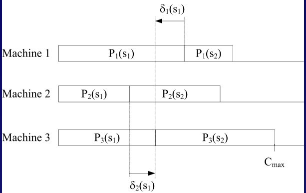
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1\leq\ell\leq 3}(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



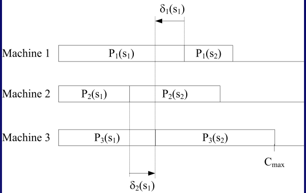
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1\leq\ell\leq3}(P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



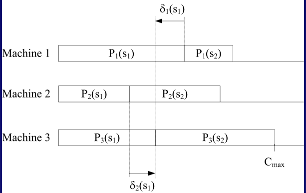
- Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine ℓ,
- We have :
  - $P(s) = \sum_{\ell=1}^{3} P_\ell(s)$,
  - $\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{3} \delta_\ell(s) = 3P_3(s) - P(s)$.
- Without loss of generality we can restrict to schedule where the last machine gives the C_{max} value,
- Now, let us concentrate on the concatenation of two partial schedules s and σ,
- We have : $C_{max}(s\sigma) = max_{1 \leq \ell \leq 3} (P_\ell(s) + P_\ell(\sigma))$,

Sort & Search : an application



- We can show that the makespan of $s\sigma$ is given by the last machine iff (constraint) :
  - $\forall \ell \in [\![1, 2]\!]$, $\delta_\ell(s) + \delta_\ell(\sigma) \geq 0$
- Then, we have $C_{max}(s\sigma) = P_3(s) + P_3(\sigma)$ which can be rewritten as (objective) :
  - $C_{max}(s\sigma) = \frac{1}{3}\left(P(s) + P(\sigma) + \sum_{\ell=1}^{2}(\delta_\ell(s) + \delta_\ell(\sigma))\right)$.

Sort & Search : an application



- We can show that the makespan of $s\sigma$ is given by the last machine iff (constraint) :
  - $\forall \ell \in [\![1, 2]\!]$, $\delta_\ell(s) + \delta_\ell(\sigma) \geq 0$
- Then, we have $C_{max}(s\sigma) = P_3(s) + P_3(\sigma)$ which can be rewritten as (objective) :
  - $C_{max}(s\sigma) = \frac{1}{3}\Big(P(s) + P(\sigma) + \sum_{\ell=1}^{2}(\delta_\ell(s) + \delta_\ell(\sigma))\Big)$.

Sort & Search : an application

Reformulation

A schedule $s\sigma$ is optimal for the $P3||C_{max}$ problem, iff the couple (s,σ) is an optimal solution of the following problem :

$$\text{Minimise} \sum_{\ell=1}^{2} \delta_\ell(s) + \delta_\ell(\sigma)$$

$$\text{s.t.} \quad \forall \ell \in [\![1,2]\!],\ \delta_\ell(s) + \delta_\ell(\sigma) \geq 0$$

Sort & Search : an application (main lines)

$$\left\{\begin{array}{ll} \vec{a}_j = & (\delta_1(s_1^j), \delta_2(s_1^j)) \\ (b_k^0, b_k^1, b_k^2) = & (\delta_1(s_2^k) + \delta_2(s_2^k), \delta_1(s_2^k), \delta_2(s_2^k)) \\ f(\vec{a}_j, b_k^0) = & (P + \delta_1(s_1^j) + \delta_2(s_1^j) + \delta_1(s_2^k) + \delta_2(s_2^k))/3 \\ g_1(\vec{a}_j, b_k^1) = & \delta_1(s_1^j) + \delta_1(s_2^k) \\ g_2(\vec{a}_j, b_k^2) = & \delta_2(s_1^j) + \delta_2(s_2^k) \end{array}\right. \quad (1)$$

Besides f, g_1 are g_2 increasing function with respect to their last variable.

Sort & Search : an application

- The complexity of *Sort & Search* is in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time,
- Starting from I_1 and I_2, tables A and B have respectively $n_A = n_B = 3^{\frac{n}{2}}$ columns,
- Besides, $d_A = 2$, and $d_B = 2$
- Then, the worst-case time complexity is in $O(3^{\frac{n}{2}} \log_2^2(3^{\frac{n}{2}}) + 3^{\frac{n}{2}} \log_2^4(3^{\frac{n}{2}})) = O^*(3^{\frac{n}{2}}) \approx O^*(1.7321^n)$.

Sort & Search : an application

- The complexity of *Sort & Search* is in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time,
- Starting from I_1 and I_2, tables A and B have respectively $n_A = n_B = 3^{\frac{n}{2}}$ columns,
- Besides, $d_A = 2$, and $d_B = 2$
- Then, the worst-case time complexity is in $O(3^{\frac{n}{2}} \log_2^2(3^{\frac{n}{2}}) + 3^{\frac{n}{2}} \log_2^4(3^{\frac{n}{2}})) = O^*(3^{\frac{n}{2}}) \approx O^*(1.7321^n)$.

Sort & Search : an application

- The complexity of *Sort & Search* is in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time,
- Starting from I_1 and I_2, tables A and B have respectively $n_A = n_B = 3^{\frac{n}{2}}$ columns,
- Besides, $d_A = 2$, and $d_B = 2$
- Then, the worst-case time complexity is in $O(3^{\frac{n}{2}} \log_2^2(3^{\frac{n}{2}}) + 3^{\frac{n}{2}} \log_2^4(3^{\frac{n}{2}})) = O^*(3^{\frac{n}{2}}) \approx O^*(1.7321^n)$.

Sort & Search : an application

- The complexity of *Sort & Search* is in $O(n_B \log_2^{d_B}(n_B) + n_A \log_2^{d_B+2}(n_B))$ time,
- Starting from I_1 and I_2, tables A and B have respectively $n_A = n_B = 3^{\frac{n}{2}}$ columns,
- Besides, $d_A = 2$, and $d_B = 2$
- Then, the worst-case time complexity is in $O(3^{\frac{n}{2}} \log_2^2(3^{\frac{n}{2}}) + 3^{\frac{n}{2}} \log_2^4(3^{\frac{n}{2}})) = O^*(3^{\frac{n}{2}}) \approx O^*(1.7321^n)$.

Sort & Search : to conclude

- *Sort & Search* is an interesting technique for deriving "quickly" E-ETA,
- Requires exponential space,
- In scheduling, it is usable for parallel machine scheduling problems.

Sort & Search : to conclude

- *Sort & Search* is an interesting technique for deriving "quickly" E-ETA,
- Requires exponential space,
- In scheduling, it is usable for parallel machine scheduling problems.

Sort & Search : to conclude

- *Sort & Search* is an interesting technique for deriving “quickly” E-ETA,
- Requires exponential space,
- In scheduling, it is usable for parallel machine scheduling problems.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics [illegible]
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
    - *Sort & Search* seems to be applicable when the problem [illegible]
    - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
    - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics is only induced by "assignment decisions",
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics is only induced by "assignment decisions",
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics is only induced by "assignment decisions",
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

Time to conclude on E-ETA

- A theoretical and nice research area,
- Helps in understanding what makes a problem hard to be solved,
- It's emerging in scheduling literature (problems are difficult),
- With respect to the three techniques introduced in this talk :
  - *Sort & Search* seems to be applicable when the combinatorics is only induced by "assignment decisions",
  - *Branch & Reduce* and *Dynamic Programming* are more suitable for permutation problems,
  - It is challenging to design reduction rules in *Branch & Reduce* algorithms.

About the problem

Definition

Given n jobs : $N = \{1, \ldots, n\}$, m parallel identical machines, each job i has a processing time p_i and a due date d_i. Determine the job sequence on each machine which minimizes $\sum_i U_i$, with $U_i = 1$ if job i is tardy ; 0 otherwise.

- Problem denoted by $P|d_i|\sum_i U_i$.
- $\mathcal{NP}$-hard (Garey and Johnson, 1979), even in the case $m = 2$.
- The question we had : can we approximate optimal solutions for this problem ?
- We focus on the approximation ratio of an heuristic H :

$$\rho = \frac{\sum_i U_i^H}{\sum_i U_i^*}$$

About the problem

Definition

Given n jobs : $N = \{1, \ldots, n\}$, m parallel identical machines, each job i has a processing time p_i and a due date d_i. Determine the job sequence on each machine which minimizes $\sum_i U_i$, with $U_i = 1$ if job i is tardy ; 0 otherwise.

- Problem denoted by $P|d_i|\sum_i U_i$.
- $\mathcal{NP}$-hard (Garey and Johnson, 1979), even in the case $m = 2$.
- The question we had : can we approximate optimal solutions for this problem ?
- We focus on the approximation ratio of an heuristic H :

$$\rho = \frac{\sum_i U_i^H}{\sum_i U_i^*}$$

About the problem

Definition

Given n jobs : $N = \{1, \dots, n\}$, m parallel identical machines, each job i has a processing time p_i and a due date d_i. Determine the job sequence on each machine which minimizes $\sum_i U_i$, with $U_i = 1$ if job i is tardy ; 0 otherwise.

- Problem denoted by $P|d_i|\sum_i U_i$.
- $\mathcal{NP}$-hard (Garey and Johnson, 1979), even in the case $m = 2$.
- The question we had : can we approximate optimal solutions for this problem ?
- We focus on the approximation ratio of an heuristic H :

$$\rho = \frac{\sum_i U_i^H}{\sum_i U_i^*}$$

About the problem

Definition

Given n jobs : $N = \{1, \ldots, n\}$, m parallel identical machines, each job i has a processing time p_i and a due date d_i. Determine the job sequence on each machine which minimizes $\sum_i U_i$, with $U_i = 1$ if job i is tardy ; 0 otherwise.

- Problem denoted by $P|d_i|\sum_i U_i$.
- $\mathcal{NP}$-hard (Garey and Johnson, 1979), even in the case $m = 2$.
- The question we had : can we approximate optimal solutions for this problem ?
- We focus on the approximation ratio of an heuristic H :

$$\rho = \frac{\sum_i U_i^H}{\sum_i U_i^*}$$

About the problem

- First result : Problem $P2|d_i|\sum_i U_i$ does not admit a polynomial-time approximation algorithm with a bounded ratio ρ.
- Deciding the existence of a schedule with $\sum_i U_i^* = 0$ is $\mathcal{NP}$-hard.
- What can we do if we pay for exponential computation time : can we approximate in a moderately exponential time the $P|d_i|\sum_i U_i$ problem ?

About the problem

- First result : Problem $P2|d_i|\sum_i U_i$ does not admit a polynomial-time approximation algorithm with a bounded ratio ρ.
- Deciding the existence of a schedule with $\sum_i U_i^* = 0$ is $\mathcal{NP}$-hard.
- What can we do if we pay for exponential computation time : can we approximate in a moderately exponential time the $P|d_i|\sum_i U_i$ problem ?

About the problem

- First result : Problem $P2|d_i|\sum_i U_i$ does not admit a polynomial-time approximation algorithm with a bounded ratio ρ.
- Deciding the existence of a schedule with $\sum_i U_i^* = 0$ is $\mathcal{NP}$-hard.
- What can we do if we pay for exponential computation time : can we approximate in a moderately exponential time the $P|d_i|\sum_i U_i$ problem ?

Approximation Algorithms

Generality

For $\mathcal{NP}$-hard minimization problems, polynomial-time heuristic algorithms $\mathtt{H}$ with a worst-case guarantee :

- Fixed ratio : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$ and polynomial time in input length,
- PTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time in input length when ϵ is fixed,
- FPTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time both in input length and $\frac{1}{\epsilon}$.

- A large part of the scheduling literature...
- Few works on approximation with moderately exponential computation time (Sevastianov and Woeginger (1998), Hall (1998), Jansen (2003))... complexities in $f(\epsilon, m) + O(p(n))$.

Approximation Algorithms

Generality

For $\mathcal{NP}$-hard minimization problems, polynomial-time heuristic algorithms $\mathtt{H}$ with a worst-case guarantee :

- Fixed ratio : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$ and polynomial time in input length,
- PTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time in input length when ϵ is fixed,
- FPTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time both in input length and $\frac{1}{\epsilon}$.

- A large part of the scheduling literature...
- Few works on approximation with moderately exponential computation time (Sevastianov and Woeginger (1998), Hall (1998), Jansen (2003))... complexities in $f(\epsilon, m) + O(p(n))$.

Approximation Algorithms

Generality

For $\mathcal{NP}$-hard minimization problems, polynomial-time heuristic algorithms $\mathtt{H}$ with a worst-case guarantee :

- Fixed ratio : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq \rho$ and polynomial time in input length,
- PTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time in input length when ϵ is fixed,
- FPTAS : $\frac{Z^{\mathtt{H}}}{Z^{\mathtt{Opt}}} \leq (1+\epsilon)$ and polynomial time both in input length and $\frac{1}{\epsilon}$.

- A large part of the scheduling literature...
- Few works on approximation with moderately exponential computation time (Sevastianov and Woeginger (1998), Hall (1998), Jansen (2003))... complexities in $f(\epsilon, m) + O(p(n))$.

Exact Exponential-Time Algorithms

General objectives

For $\mathcal{NP}$-hard problems, design exact algorithms with worst-case running time guarantee.

- Complexity $\mathcal{O}^*(c^n)$, with c a constant as small as possible

- In the remainder, we rely in the framework presented by Paschos (2015) : find approximation algorithms with wc time complexity in $O^*(c^n)$.

Paschos, V. (2015). *When polynomial approximation meets exact computation*. 4'OR, 13(3) :227-245

Exact Exponential-Time Algorithms

General objectives

For $\mathcal{NP}$-hard problems, design exact algorithms with worst-case running time guarantee.

- Complexity $\mathcal{O}^*(c^n)$, with c a constant as small as possible

- In the remainder, we rely in the framework presented by Paschos (2015) : find approximation algorithms with wc time complexity in $O^*(c^n)$.

Paschos, V. (2015). *When polynomial approximation meets exact computation*. 4'OR, 13(3) :227-245

Initial results

Theorem 1

Let $\tilde{d}_i$ be a deadline associated with job i, so that in a feasible schedule job i must complete before $\tilde{d}_i$. The existence of a feasible schedule for the $P|\tilde{d}_i|-$ problem can be decided in $\mathcal{O}^*(m^{\frac{n}{2}})$ time and space.

- This result is shown by reformulating the $P|\tilde{d}_i|-$ problem as a (MCP),
- We denote by A^f the algorithm solving the $P|\tilde{d}_i|-$ problem.
- Lente et al. ([4]) proposed an E-ETA for solving the $P|d_i|\sum_i U_i$ problem, which requires $\mathcal{O}^*((m+1)^{\frac{n}{2}})$ time and space in the worst case.

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

Initial results

Theorem 1

Let $\tilde{d}_i$ be a deadline associated with job i, so that in a feasible schedule job i must complete before $\tilde{d}_i$. The existence of a feasible schedule for the $P|\tilde{d}_i|-$ problem can be decided in $\mathcal{O}^*(m^{\frac{n}{2}})$ time and space.

- This result is shown by reformulating the $P|\tilde{d}_i|-$ problem as a (MCP),
- We denote by A^f the algorithm solving the $P|\tilde{d}_i|-$ problem.
- Lente et al. ([4]) proposed an E-ETA for solving the $P|d_i|\sum_i U_i$ problem, which requires $\mathcal{O}^*((m+1)^{\frac{n}{2}})$ time and space in the worst case.

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

Initial results

Theorem 1

Let $\tilde{d}_i$ be a deadline associated with job i, so that in a feasible schedule job i must complete before $\tilde{d}_i$. The existence of a feasible schedule for the $P|\tilde{d}_i|-$ problem can be decided in $\mathcal{O}^*(m^{\frac{n}{2}})$ time and space.

- This result is shown by reformulating the $P|\tilde{d}_i|-$ problem as a (MCP),
- We denote by A^f the algorithm solving the $P|\tilde{d}_i|-$ problem.
- Lente et al. ([4]) proposed an E-ETA for solving the $P|d_i|\sum_i U_i$ problem, which requires $\mathcal{O}^*((m+1)^{\frac{n}{2}})$ time and space in the worst case.

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

Initial results

Theorem 1

Let $\tilde{d}_i$ be a deadline associated with job i, so that in a feasible schedule job i must complete before $\tilde{d}_i$. The existence of a feasible schedule for the $P|\tilde{d}_i|-$ problem can be decided in $\mathcal{O}^*(m^{\frac{n}{2}})$ time and space.

- This result is shown by reformulating the $P|\tilde{d}_i|-$ problem as a (MCP),
- We denote by A^f the algorithm solving the $P|\tilde{d}_i|-$ problem.
- Lente et al. ([4]) proposed an E-ETA for solving the $P|d_i|\sum_i U_i$ problem, which requires $\mathcal{O}^*((m+1)^{\frac{n}{2}})$ time and space in the worst case.

[4] C. Lente, M. Liedloff, A. Soukhal and V. T'kindt. On an extension of the Sort & Search method with application to scheduling theory, Theoretical Computer Science, vol 511, pp. 13-22, 2013.

A branching heuristic

- We propose a first approximation algorithm, referred to as `Bapprox`,
- Let $k \in \mathbb{N}^*$ be a given parameter,
- Wlog, we assume $d_1 \leq d_2 \leq ... \leq d_n$,
- First, A^f is run with $\tilde{d}_j = d_j$ to check if a solution with $\sum_j U_j^* = 0$ exists,
- If not, the n jobs are grouped into $\lceil \frac{n}{k} \rceil$ batches. Each batch B_ℓ contains jobs $\{(\ell - 1) * k + 1, ..., \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$

A branching heuristic

- We propose a first approximation algorithm, referred to as `Bapprox`,
- Let $k \in \mathbb{N}^*$ be a given parameter,
- Wlog, we assume $d_1 \leq d_2 \leq ... \leq d_n$,
- First, A^f is run with $\tilde{d}_j = d_j$ to check if a solution with $\sum_j U_j^* = 0$ exists,
- If not, the n jobs are grouped into $\lceil \frac{n}{k} \rceil$ batches.
  Each batch B_ℓ contains jobs $\{(\ell - 1) * k + 1, ..., \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$

A branching heuristic

- We propose a first approximation algorithm, referred to as `Bapprox`,
- Let $k \in \mathbb{N}^*$ be a given parameter,
- Wlog, we assume $d_1 \leq d_2 \leq ... \leq d_n$,
- First, A^f is run with $\tilde{d}_j = d_j$ to check if a solution with $\sum_j U_j^* = 0$ exists,
- If not, the n jobs are grouped into $\lceil \frac{n}{k} \rceil$ batches. Each batch B_ℓ contains jobs $\{(\ell - 1) * k + 1, ..., \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$

A branching heuristic

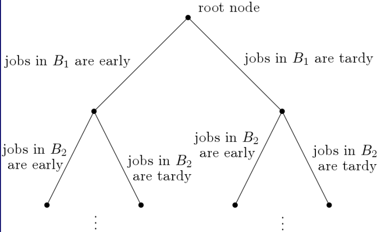
- We propose a first approximation algorithm, referred to as `Bapprox`,
- Let $k \in \mathbb{N}^*$ be a given parameter,
- Wlog, we assume $d_1 \leq d_2 \leq ... \leq d_n$,
- First, A^f is run with $\tilde{d}_j = d_j$ to check if a solution with $\sum_j U_j^* = 0$ exists,
- If not, the n jobs are grouped into $\lceil \frac{n}{k} \rceil$ batches. Each batch B_ℓ contains jobs $\{(\ell-1)*k+1, ..., \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$

A branching heuristic

- We propose a first approximation algorithm, referred to as `Bapprox`,
- Let $k \in \mathbb{N}^*$ be a given parameter,
- Wlog, we assume $d_1 \leq d_2 \leq ... \leq d_n$,
- First, A^f is run with $\tilde{d}_j = d_j$ to check if a solution with $\sum_j U_j^* = 0$ exists,
- If not, the n jobs are grouped into $\lceil \frac{n}{k} \rceil$ batches.
  Each batch B_ℓ contains jobs $\{(\ell - 1) * k + 1, ..., \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$

Algorithm outline

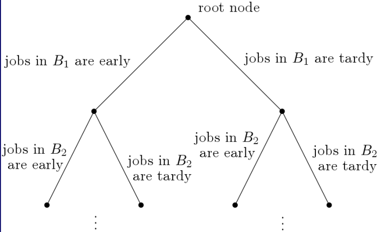
- Algorithm Bapprox builds a binary search tree by branching at each level ℓ on batch B_ℓ and scheduling all its jobs either early of tardy.



- Each leaf node s defines a set of possible early jobs E_s, the remaining jobs being tardy.
  Algorithm A^f is run to check if there exists a feasible schedule with jobs in E_s all early.
  $\Rightarrow \forall i \in E_s, \tilde{d}_i = d_i$, and $\forall i \in N \backslash E_s, \tilde{d}_i = +\infty$

Algorithm outline

- Algorithm Bapprox builds a binary search tree by branching at each level ℓ on batch B_ℓ and scheduling all its jobs either early of tardy.



- Each leaf node s defines a set of possible early jobs E_s, the remaining jobs being tardy.
  Algorithm A^f is run to check if there exists a feasible schedule with jobs in E_s all early.
  $\Rightarrow \forall i \in E_s, \tilde{d}_i = d_i$, and $\forall i \in N \backslash E_s, \tilde{d}_i = +\infty$

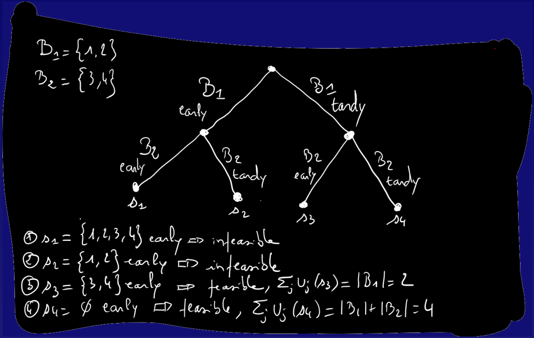
Algorithm outline

Exercice.

Apply `Bapprox` on the following instance :
$n = 4,\ m = 2,\ [p_i]_i = [5; 4; 3; 6],\ [d_i]_i = [4; 8; 9; 10].$
Find the optimal solution and provide the ratio on this example.

Algorithm outline

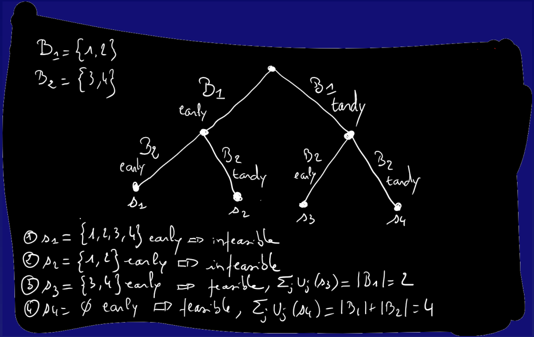
- Branch and evaluate all leaf nodes,



- The solution returned is s_3 with $\{3;4\}$ early and $\{1;2\}$ tardy, and $\sum_i U_i(s_3) = 2$,
- The optimal solution is s^* with $\{2;3;4\}$ early and $\{1\}$ tardy, and $\sum_i U_i(s^*) = 1$,
- Here, the ratio is $\frac{2}{1} = \ldots$?

Algorithm outline

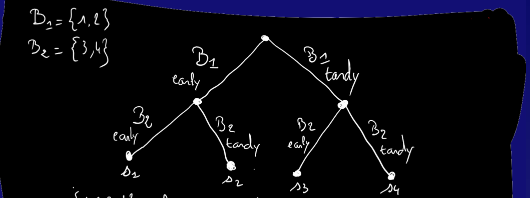
- Branch and evaluate all leaf nodes,



- The solution returned is s_3 with $\{3;4\}$ early and $\{1;2\}$ tardy, and $\sum_i U_i(s_3) = 2$,
- The optimal solution is s^* with $\{2;3;4\}$ early and $\{1\}$ tardy, and $\sum_i U_i(s^*) = 1$,
- Here, the ratio is $\frac{2}{1} = \ldots$?

Algorithm outline

- Branch and evaluate all leaf nodes,

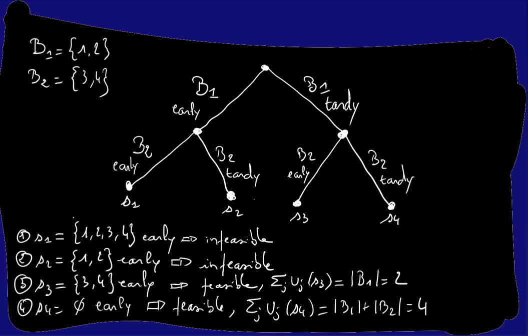


① $s_1 = \{1,2,3,4\}$ early $\Rightarrow$ infeasible
② $s_2 = \{1,2\}$ early $\Rightarrow$ infeasible
③ $s_3 = \{3,4\}$ early $\Rightarrow$ feasible, $\sum_j U_j(s_3) = |B_1| = 2$
④ $s_4 = \emptyset$ early $\Rightarrow$ feasible, $\sum_j U_j(s_4) = |B_1| + |B_2| = 4$

- The solution returned is s_3 with $\{3;4\}$ early and $\{1;2\}$ tardy, and $\sum_i U_i(s_3) = 2$,
- The optimal solution is s^* with $\{2;3;4\}$ early and $\{1\}$ tardy, and $\sum_i U_i(s^*) = 1$,
- Here, the ratio is $\frac{2}{1} = \ldots$?

Algorithm outline

- Branch and evaluate all leaf nodes,



- The solution returned is s_3 with $\{3;4\}$ early and $\{1;2\}$ tardy, and $\sum_i U_i(s_3) = 2$,
- The optimal solution is s^* with $\{2;3;4\}$ early and $\{1\}$ tardy, and $\sum_i U_i(s^*) = 1$,
- Here, the ratio is $\frac{2}{1} = \ldots$?

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by Bapprox.
  -
  - If Bapprox is not optimal then some tardy jobs are early in the optimal solution.
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy.
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$.
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{k}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by `Bapprox`,
  - But what is the situation in an optimal schedule ?
  - If `Bapprox` is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by `Bapprox`,
  - But what is the situation in an optimal schedule ?
  - If `Bapprox` is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by Bapprox,
  - But what is the situation in an optimal schedule ?
  - If Bapprox is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by Bapprox,
  - But what is the situation in an optimal schedule ?
  - If Bapprox is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by `Bapprox`,
  - But what is the situation in an optimal schedule ?
  - If `Bapprox` is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by `Bapprox`,
  - But what is the situation in an optimal schedule ?
  - If `Bapprox` is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof (sketch) : Ratio.
  - Assume α batches are scheduled tardy by Bapprox,
  - But what is the situation in an optimal schedule ?
  - If Bapprox is not optimal then some tardy jobs are early in the optimal solution,
  - For each of these batches u, in the optimal solution, only $\ell_u \geq 1$ jobs are tardy,
  - Then, $\rho \leq \frac{\alpha k}{\sum_{u=1}^{\alpha} \ell_u}$,
  - The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - Initial (feasibility) step requires $\mathcal{O}^*(m^{\frac{n}{2}})$ time.
  - Let $\mathcal{LN}$ be the list of all leaf nodes generated. A leaf node can be represented by $(E;T)$ two sets of early and tardy jobs.
  - We have : $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell}$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - Initial (feasibility) step requires $\mathcal{O}^*(m^{\frac{n}{2}})$ time.
  - Let $\mathcal{LN}$ be the list of all leaf nodes generated. A leaf node can be represented by $(E;T)$ two sets of early and tardy jobs.
  - We have : $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell}$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - Initial (feasibility) step requires $\mathcal{O}^*(m^{\frac{n}{2}})$ time.
  - Let $\mathcal{LN}$ be the list of all leaf nodes generated. A leaf node can be represented by $(E;T)$ two sets of early and tardy jobs.
  - We have : $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell}$.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - Initial (feasibility) step requires $\mathcal{O}^*(m^{\frac{n}{2}})$ time.
  - Let $\mathcal{LN}$ be the list of all leaf nodes generated. A leaf node can be represented by $(E;T)$ two sets of early and tardy jobs.
  - We have : $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell}$.

Analysis

Theorem 2

Algorithm `Bapprox` admits a worst-case ratio $\rho \leq k$.
Algorithm `Bapprox` requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - $\forall (E;T) \in \mathcal{LN}$, deciding of the feasibility requires $\mathcal{O}^*(m^{\frac{|E|}{2}})$ time, with $|E| = k\ell$ and ℓ the number of early batches in E.
  - It follows that to build and test all leaf nodes the worst-case running time is in :

$$\mathcal{O}^*\left(\sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell} (m^{\frac{k}{2}})^{\ell}\right)$$
$$\Leftrightarrow \mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}}),$$

    by making use of the Newton's binomial formula.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$.
Algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Proof : worst-case time complexity.
  - $\forall (E;T) \in \mathcal{LN}$, deciding of the feasibility requires $\mathcal{O}^*(m^{\frac{|E|}{2}})$ time, with $|E| = k\ell$ and ℓ the number of early batches in E.
  - It follows that to build and test all leaf nodes the worst-case running time is in :

$$\mathcal{O}^*\left(\sum_{\ell=0}^{\lceil \frac{n}{k} \rceil} \binom{\lceil \frac{n}{k} \rceil}{\ell} (m^{\frac{k}{2}})^{\ell}\right)$$
$$\Leftrightarrow \mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n}{k}}),$$

by making use of the Newton's binomial formula.

Analysis

Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$.
Algorithm Bapprox requires $\mathcal{O}^*((1 + m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space.

- Illustration (ratios and complexities) in the case $m = 2$:

k	ρ	time
1	1	$O(2.4142^n)$
2	2	$O(1.7320^n)$
3	3	$O(1.5643^n)$
4	4	$O(1.4953^n)$
5	5	$O(1.4610^n)$
		...
10	10	$O(1.4186^n)$

Noteworthy, by comparison with the EETA running in $O(1.7320^n)$ time, algorithm Bapprox is relevant for $k \geq 3$.

Branching and preprocessing

- How to decrease the ratio ρ of algorithm `Bapprox` ?
- We add a preprocessing step (algorithm `PBapprox`).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs among n and then, for each, solve the feasibility problem on the remaining jobs.
- If at least one of these subsets lead to a feasible schedule, then the optimal solution of the $P|d_i|\sum_i U_i$ problem is found. Otherwise, algorithm `Bapprox` is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).

Branching and preprocessing

- How to decrease the ratio ρ of algorithm `Bapprox` ?
- We add a preprocessing step (algorithm `PBapprox`).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs among n and then, for each, solve the feasibility problem on the remaining jobs.
- If at least one of these subsets lead to a feasible schedule, then the optimal solution of the $P|d_i|\sum_i U_i$ problem is found. Otherwise, algorithm `Bapprox` is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).

Branching and preprocessing

- How to decrease the ratio ρ of algorithm `Bapprox` ?
- We add a preprocessing step (algorithm `PBapprox`).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs among n and then, for each, solve the feasibility problem on the remaining jobs.
- If at least one of these subsets lead to a feasible schedule, then the optimal solution of the $P|d_i|\sum_i U_i$ problem is found. Otherwise, algorithm `Bapprox` is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).

Branching and preprocessing

- How to decrease the ratio ρ of algorithm `Bapprox` ?
- We add a preprocessing step (algorithm `PBapprox`).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs among n and then, for each, solve the feasibility problem on the remaining jobs.
- If at least one of these subsets lead to a feasible schedule, then the optimal solution of the $P|d_i|\sum_i U_i$ problem is found. Otherwise, algorithm `Bapprox` is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).

Branching and preprocessing

- How to decrease the ratio ρ of algorithm `Bapprox` ?
- We add a preprocessing step (algorithm `PBapprox`).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs among n and then, for each, solve the feasibility problem on the remaining jobs.
- If at least one of these subsets lead to a feasible schedule, then the optimal solution of the $P|d_i|\sum_i U_i$ problem is found. Otherwise, algorithm `Bapprox` is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).

Analysis

Theorem 3

Algorithm `PBapprox` admits a worst-case ratio $\rho \leq \frac{k^2+k(c-1)+1}{k+c}$.

Analysis

Theorem 4

Algorithm PBapprox requires $\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time, $H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The preprocessing phase : generation of subsets of size at most $\lfloor\frac{n}{c}\rfloor$ tardy jobs and solution of feasibility problems on the early jobs. Worst-case running time :

$$\mathcal{O}^*(\sum_{i=0}^{\lfloor\frac{n}{c}\rfloor}\binom{n}{i} m^{\frac{n-i}{2}}).$$

- This is a partial sum of binomials !

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}} (1 + m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The preprocessing phase : generation of subsets of size at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs and solution of feasibility problems on the early jobs. Worst-case running time :

$$\mathcal{O}^*(\sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} m^{\frac{n-i}{2}}).$$

- This is a partial sum of binomials !

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max\left(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\right)\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The preprocessing phase : generation of subsets of size at most $\lfloor \frac{n}{c} \rfloor$ tardy jobs and solution of feasibility problems on the early jobs. Worst-case running time :

$$\mathcal{O}^*\left(\sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} m^{\frac{n-i}{2}}\right).$$

- This is a partial sum of binomials !

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- No close formula, use of an upper bound :

$$\sum_{i=0}^{\ell}\binom{n}{i} \leq 2^{H(\frac{\ell}{n})n},$$

with $H(\frac{\ell}{n}) = -\frac{\ell}{n}\log_2(\frac{\ell}{n}) - (1-\frac{\ell}{n})\log_2(1-\frac{\ell}{n})$, $0 < \frac{\ell}{n} < 1$, the binary entropy of $\frac{\ell}{n}$.

Analysis

Theorem 4

Algorithm `PBapprox` requires
$\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- We obtain the following reformulation :

$$\begin{aligned}\sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} m^{\frac{n-i}{2}} &\leq \sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} \times m^{\frac{n(c-1)}{2c}} \\ &\leq 2^{H(c)n} m^{\frac{n(c-1)}{2c}}.\end{aligned}$$

- The preprocessing phase has a worst-case time complexity in $\mathcal{O}^*(2^{H(c)n} m^{\frac{n(c-1)}{2c}})$.

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}} (1 + m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- We obtain the following reformulation :

$$\begin{aligned}\sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} m^{\frac{n-i}{2}} &\leq \sum_{i=0}^{\lfloor \frac{n}{c} \rfloor} \binom{n}{i} \times m^{\frac{n(c-1)}{2c}} \\ &\leq 2^{H(c)n} m^{\frac{n(c-1)}{2c}}.\end{aligned}$$

- The preprocessing phase has a worst-case time complexity in $\mathcal{O}^*(2^{H(c)n} m^{\frac{n(c-1)}{2c}})$.

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The branching phase : algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}})$ time.
- The branching phase has a worst-case running time in :

$$\mathcal{O}^*\left(\binom{n}{\lfloor \frac{n}{c} \rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}}\right).$$

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max\left(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\right)\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The branching phase : algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}})$ time.
- The branching phase has a worst-case running time in :

$$\mathcal{O}^*\left(\binom{n}{\lfloor \frac{n}{c} \rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}}\right).$$

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max\left(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}} (1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\right)\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- The branching phase : algorithm Bapprox requires $\mathcal{O}^*((1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}})$ time.
- The branching phase has a worst-case running time in :

$$\mathcal{O}^*\left(\binom{n}{\lfloor \frac{n}{c} \rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}}\right).$$

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\left(\max\left(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\right)\right)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- By noting that $\binom{N}{K} < \frac{N^K e^K}{K^K}$ with e being Euler's number, we obtain :

$$\binom{n}{\lfloor\frac{n}{c}\rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor\frac{n}{c}\rfloor}{k}} < \left(\frac{ne}{\lfloor\frac{n}{c}\rfloor}\right)^{\lfloor\frac{n}{c}\rfloor}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}$$
$$< (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}.$$

- Finally, we have the worst-case time complexity stated in the theorem.

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\Big(\max(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}})\Big)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- By noting that $\binom{N}{K} < \frac{N^K e^K}{K^K}$ with e being Euler's number, we obtain :

$$\binom{n}{\lfloor \frac{n}{c} \rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor \frac{n}{c} \rfloor}{k}} < \left(\frac{ne}{\lfloor \frac{n}{c} \rfloor}\right)^{\lfloor \frac{n}{c} \rfloor}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}$$
$$< (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}.$$

- Finally, we have the worst-case time complexity stated in the theorem.

Analysis

Theorem 4

Algorithm PBapprox requires
$\mathcal{O}^*\Big(\max\big(2^{H(c)n} m^{\frac{n(c-1)}{2c}}; (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\big)\Big)$ time,
$H(c) = -c\log_2(c) - (1-c)\log_2(1-c)$ and e is Euler's number.
The worst-case space requirement is in $\mathcal{O}^*(m^{\frac{n(c-1)}{2c}})$.

- Proof : worst-case time complexity.
- By noting that $\binom{N}{K} < \frac{N^K e^K}{K^K}$ with e being Euler's number, we obtain :

$$\begin{aligned}\binom{n}{\lfloor\frac{n}{c}\rfloor}(1+m^{\frac{k}{2}})^{\frac{n-\lfloor\frac{n}{c}\rfloor}{k}} &< \Big(\frac{ne}{\lfloor\frac{n}{c}\rfloor}\Big)^{\lfloor\frac{n}{c}\rfloor}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}\\ &< (ce)^{\frac{n}{c}}(1+m^{\frac{k}{2}})^{\frac{n(c-1)}{ck}}.\end{aligned}$$

- Finally, we have the worst-case time complexity stated in the theorem.

Analysis

- Illustration (ratios and complexities) in the case $m = 2$:

k	ρ	time
1	1	$O(2.4142^n)$
2	2	$O(1.7320^n)$
3	3	$O(1.5643^n)$
4	4	$O(1.4953^n)$
5	5	$O(1.4610^n)$
	...	
10	10	$O(1.4186^n)$

$\Rightarrow$

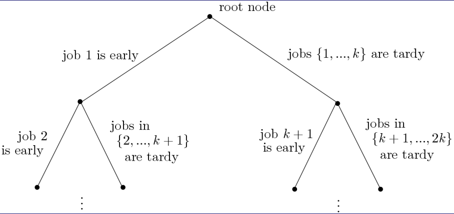
k	c	ρ	time
3	1000	2.99	$O(1.5760^n)$
	100	2.98	$O(1.6471^n)$
	10	2.84	$O(2.0813^n)$
4	1000	3.99	$O(1.5066^n)$
	100	3.97	$O(1.5752^n)$
	10	3.78	$O(1.9984^n)$
	...		

Generalizations

- Weighted case : $P|d_i|\sum_i w_i U_i$,
- Algorithm Bapprox can be generalized by changing the branching scheme (and making the wct analysis more complicated),
- Ratio : $\rho = k$,
- Worst-case time complexity : $\mathcal{O}^*(\gamma^n)$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space, with $\gamma = m^{\frac{1}{2\delta}}$ and $\gamma^{-k} + \gamma^{-1+\delta} = 1$.

Generalizations

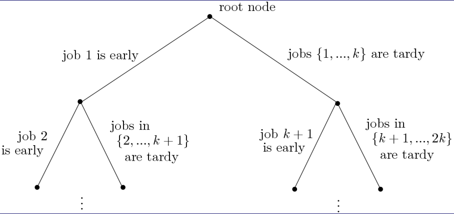
- Weighted case : $P|d_i|\sum_i w_i U_i$,
- Algorithm `Bapprox` can be generalized by changing the branching scheme (and making the wct analysis more complicated),



- Ratio : $\rho = k$,
- Worst-case time complexity : $\mathcal{O}^*(\gamma^n)$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space, with $\gamma = m^{\frac{1}{2\delta}}$ and $\gamma^{-k} + \gamma^{-1+\delta} = 1$.

Generalizations

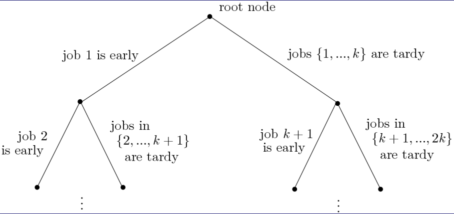
- Weighted case : $P|d_i|\sum_i w_i U_i$,
- Algorithm `Bapprox` can be generalized by changing the branching scheme (and making the wct analysis more complicated),



- Ratio : $\rho = k$,
- Worst-case time complexity : $\mathcal{O}^*(\gamma^n)$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space, with $\gamma = m^{\frac{1}{2\delta}}$ and $\gamma^{-k} + \gamma^{-1+\delta} = 1$.

Generalizations

- Weighted case : $P|d_i|\sum_i w_i U_i$,
- Algorithm Bapprox can be generalized by changing the branching scheme (and making the wct analysis more complicated),



- Ratio : $\rho = k$,
- Worst-case time complexity : $\mathcal{O}^*(\gamma^n)$ time and $\mathcal{O}^*(m^{\frac{n}{2}})$ space, with $\gamma = m^{\frac{1}{2\delta}}$ and $\gamma^{-k} + \gamma^{-1+\delta} = 1$.

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm Bapprox : a branching-based heuristic,
- Algorithme PBapprox : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of PBapprox,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm `Bapprox` : a branching-based heuristic,
- Algorithme `PBapprox` : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of `PBapprox`,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm `Bapprox` : a branching-based heuristic,
- Algorithme `PBapprox` : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of `PBapprox`,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm `Bapprox` : a branching-based heuristic,
- Algorithme `PBapprox` : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of `PBapprox`,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm `Bapprox` : a branching-based heuristic,
- Algorithme `PBapprox` : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of `PBapprox`,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- The $P|d_i|\sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm `Bapprox` : a branching-based heuristic,
- Algorithme `PBapprox` : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of `PBapprox`,
- Can we handle a reduction of ratio $\rho = k$ directly in the branching scheme ?
- Can we generalize this approach to other scheduling problems with number of tardy jobs ?

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !

Conclusions

- Exponential Time Algorithms provide us with worst-case information,
- Ok, finding ETA with reduced worst-case complexities is a challenging (theoretical) issue,
- Apparently, there is also a room for strong computational impacts,
- The idea : take advantage of decomposition approaches of ETA, embed problem-dependent knowledge,
- You may get efficient exact algorithms...
- ... even more with polynomial space ETA !