Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

V T’kindt

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

June 2019
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
1 Introduction

2 Exact Exponential-Time Algorithms

3 Heuristic Exponential-Time Algorithms

4 Conclusions
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: $O^*(\exp(n)) = O(\text{poly}(n)\exp(n))$

- 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 $\rho$:
  \[ \frac{Z^H}{Z^{Opt}} \leq \rho, \]
  - 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),

About H-ETA:

- For a given heuristic $H$ we compute a worst-case ratio $\rho$:
  \[ \frac{Z_H}{Z_{0pt}} \leq \rho, \]

- 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),

About H-ETA :

- For a given heuristic $H$ we compute a worst-case ratio $\rho$ :
  $$\frac{Z^H}{Z^{\text{Opt}}} \leq \rho,$$
- 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),

About H-ETA :

- For a given heuristic $H$ we compute a worst-case ratio $\rho$:
  $$\frac{Z^H}{Z^{Opt}} \leq \rho,$$

- 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),

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 ≈ 2005 in scheduling literature,
What about scheduling problems (single machine)?

<table>
<thead>
<tr>
<th>Problem</th>
<th>brute force</th>
<th>wctc</th>
<th>wcsc</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dec</td>
<td>(f_{\text{max}})</td>
<td>(O^*(n!))</td>
<td>(O^*(2^n))</td>
</tr>
<tr>
<td>1</td>
<td>dec</td>
<td>(\sum_i f_i)</td>
<td>(O^*(n!))</td>
<td>(O^*(2^n))</td>
</tr>
<tr>
<td>1</td>
<td>prec</td>
<td>(\sum_i C_i)</td>
<td>(O^*(n!))</td>
<td>(O^*((2 - \epsilon)^n))</td>
</tr>
<tr>
<td>1</td>
<td>prec</td>
<td>(\sum_i w_i C_i)</td>
<td>(O^*(n!))</td>
<td>(O^*(2^n))</td>
</tr>
<tr>
<td>1</td>
<td>(d_i)</td>
<td>(\sum_i U_i)</td>
<td>(O^*(n!))</td>
<td>(O^*(2^n))</td>
</tr>
<tr>
<td>1</td>
<td>(d_i)</td>
<td>(\sum_i T_i)</td>
<td>(O^*(n!))</td>
<td>(O^*(2^n))</td>
</tr>
<tr>
<td>1</td>
<td>(r_i, \text{prec})</td>
<td>(\sum_i w_i C_i)</td>
<td>(O^*(n!))</td>
<td>(O^*(3^n))</td>
</tr>
</tbody>
</table>


What about scheduling problems (others)?

<table>
<thead>
<tr>
<th>Problem</th>
<th>brute force</th>
<th>wctc</th>
<th>wcsc</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P</td>
<td>\text{dec}</td>
<td>f_{\text{max}}$</td>
<td>$O^*(m^n n!)$</td>
<td>$O^*(3^n)$</td>
</tr>
<tr>
<td>$P</td>
<td>\text{dec}</td>
<td>\sum_i f_i$</td>
<td>$O^*(m^n n!)$</td>
<td>$O^*(3^n)$</td>
</tr>
<tr>
<td>$P4</td>
<td></td>
<td>C_{\text{max}}$</td>
<td>$O^*(4^n)$</td>
<td>$O^*(2.4142^n)$</td>
</tr>
<tr>
<td>$P3</td>
<td></td>
<td>C_{\text{max}}$</td>
<td>$O^*(3^n)$</td>
<td>$O^*(1.7321^n)$</td>
</tr>
<tr>
<td>$P2</td>
<td></td>
<td>C_{\text{max}}$</td>
<td>$O^*(2^n)$</td>
<td>$O^*(1.4142^n)$</td>
</tr>
<tr>
<td>$P2</td>
<td>d_i</td>
<td>\sum_i w_i U_i$</td>
<td>$O^*(3^n)$</td>
<td>$O^*(1.7321^n)$</td>
</tr>
<tr>
<td>$F2</td>
<td></td>
<td>C_{\text{max}}^k$</td>
<td>$O^*(2^n)$</td>
<td>$O^*(1.4142^n)$</td>
</tr>
<tr>
<td>$F3</td>
<td></td>
<td>C_{\text{max}}$</td>
<td>$O^*(n!)$</td>
<td>$O^*(3^n)$</td>
</tr>
<tr>
<td>$F3</td>
<td>f_{\text{max}}$</td>
<td>$O^*(n!)$</td>
<td>$O^*(5^n)$</td>
<td>exp</td>
</tr>
<tr>
<td>$F3</td>
<td>\sum_i f_i$</td>
<td>$O^*(n!)$</td>
<td>$O^*(5^n)$</td>
<td>exp</td>
</tr>
<tr>
<td>$J2</td>
<td></td>
<td>C_{\text{max}}^k$</td>
<td>$O^*(2^n)$</td>
<td>$O^*(1.4142^n)$</td>
</tr>
</tbody>
</table>


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 $1||f_{\text{max}}$ 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 $f_{\text{max}} = \max_i f_i$.

The worst-case complexity of ENUM... is in $O^*(n!)$.
Let us consider the $1\|f_{\text{max}}$ 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 $f_{\text{max}} = \max_i f_i$.

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:
  \[
  \begin{cases}
  Opt[\emptyset] = -\infty, & \text{if } f_t \text{ can be negative} \\
  Opt[\emptyset] = 0, & \text{if } f_t \text{ cannot be negative} \\
  Opt[S] = \min_{t \in S} \{ \max( Opt[S - \{t\}; f_t(P(S))) \} 
  \end{cases}
  \]
  with $P(S) = \sum_{i \in S} p_i$. 
About permutation problems...

Let be \( S \subseteq \{1, \ldots, n\} \),

Let \( \text{Opt}[S] \) be the recurrence function calculated on set \( S \) : \( \text{Opt}[S] \) is equal to the minimal value of criterion \( \max_i f_i \) for any permutation of the jobs in \( S \).

We have:
\[
\begin{cases}
    \text{Opt}[\emptyset] = -\infty, & \text{if } f_i \text{ can be negative} \\
    \text{Opt}[\emptyset] = 0, & \text{if } f_i \text{ cannot be negative} \\
    \text{Opt}[S] = \min_{t \in S} \{\max(\text{Opt}[S - \{t\]]; f_t(P(S)))\}
\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:
  
  \[
  \begin{cases} 
  Opt[\emptyset] = -\infty, & \text{if } f_t \text{ can be negative} \\
  Opt[\emptyset] = 0, & \text{if } f_t \text{ cannot be negative} \\
  Opt[S] = \min_{t \in S} \{\max( Opt[S - \{t\}; f_t(P(S))) \} \\
  \text{with } P(S) = \sum_{i \in S} p_i. 
  \end{cases}
  \]
About permutation problems...

Lost with that recurrence function? Proceed with the exercise,

**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,
  - \( S = \{1\} : \text{Opt}[S] = \max(-\infty; 3 - 4) = -1, \)
  - \( S = \{2\} : \text{Opt}[S] = \max(-\infty; 4 - 5) = -1, \)
  - \( S = \{3\} : \text{Opt}[S] = \max(-\infty; 5 - 8) = -3, \)
- 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,
  \[ S = \{1\} : \text{Opt}[S] = \max(-\infty; 3 - 4) = -1, \]
  \[ S = \{2\} : \text{Opt}[S] = \max(-\infty; 4 - 5) = -1, \]
  \[ S = \{3\} : \text{Opt}[S] = \max(-\infty; 5 - 8) = -3, \]
- 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,
  $S = \{1\} : \text{Opt}[S] = \max(-\infty; 3 - 4) = -1$,
  $S = \{2\} : \text{Opt}[S] = \max(-\infty; 4 - 5) = -1$,
  $S = \{3\} : \text{Opt}[S] = \max(-\infty; 5 - 8) = -3$,
- 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,
  
  $S = \{1\} : \text{Opt}[S] = \max(-\infty; 3 - 4) = -1$,
  
  $S = \{2\} : \text{Opt}[S] = \max(-\infty; 4 - 5) = -1$,
  
  $S = \{3\} : \text{Opt}[S] = \max(-\infty; 5 - 8) = -3$,

- 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,
  \( S = \{1\} : \text{Opt}[S] = \max(-\infty; 3 - 4) = -1 \),
  \( S = \{2\} : \text{Opt}[S] = \max(-\infty; 4 - 5) = -1 \),
  \( S = \{3\} : \text{Opt}[S] = \max(-\infty; 5 - 8) = -3 \),
- 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,
  
  $S = \{1, 2\}$:
  $Opt[S] = \min \left( \max (Opt[\{2\}]; f_1(7)); \max (Opt[\{1\}]; f_2(7)) \right)$,
  
  $\Rightarrow Opt[\{1, 2\}] = \min \left( \max (-1; 3); \max (-1; 2) \right) = 2$

- $S = \{1, 3\}$:
  $Opt[S] = \min \left( \max (Opt[\{3\}]; f_1(8)); \max (Opt[\{1\}]; f_3(8)) \right)$,
  
  $\Rightarrow Opt[\{1, 3\}] = \min \left( \max (-3; 4); \max (-1; 0) \right) = 0$

- $S = \{2, 3\}$:
  $Opt[S] = \min \left( \max (Opt[\{3\}]; f_2(9)); \max (Opt[\{2\}]; f_3(9)) \right)$,
  
  $\Rightarrow Opt[\{2, 3\}] = \min \left( \max (-3; 4); \max (-1; 1) \right) = 1$
About permutation problems...

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

- Enumerate all sets \( S \) with 2 elements,
  - \( S = \{1, 2\} : \)
    \[
    Opt[S] = \min \left( \max(Opt[\{2\}; f_1(7)) ; \max(Opt[\{1\}; f_2(7)) \right),
    \]
    \[\Rightarrow Opt[\{1, 2\}] = \min \left( \max(-1; 3) ; \max(-1; 2) \right) = 2
    \]
  - \( S = \{1, 3\} : \)
    \[
    Opt[S] = \min \left( \max(Opt[\{3\}; f_1(8)) ; \max(Opt[\{1\}; f_3(8)) \right),
    \]
    \[\Rightarrow Opt[\{1, 3\}] = \min \left( \max(-3; 4) ; \max(-1; 0) \right) = 0
    \]
  - \( S = \{2, 3\} : \)
    \[
    Opt[S] = \min \left( \max(Opt[\{3\}; f_2(9)) ; \max(Opt[\{2\}; f_3(9)) \right),
    \]
    \[\Rightarrow Opt[\{2, 3\}] = \min \left( \max(-3; 4) ; \max(-1; 1) \right) = 1
    \]
About permutation problems...

- Enumerate all sets $S$ with 2 elements,

  $S = \{1, 2\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{2\}; f_1(7)); \max (Opt[\{1\}; f_2(7)) \right)$,

  $\Rightarrow Opt[\{1, 2\}] = \min (\max (-1; 3); \max (-1; 2)) = 2$

  $S = \{1, 3\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{3\]; f_1(8)); \max (Opt[\{1\]; f_3(8)) \right)$,

  $\Rightarrow Opt[\{1, 3\}] = \min (\max (-3; 4); \max (-1; 0)) = 0$

  $S = \{2, 3\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{3\]; f_2(9)); \max (Opt[\{2\]; f_3(9)) \right)$,

  $\Rightarrow Opt[\{2, 3\}] = \min (\max (-3; 4); \max (-1; 1)) = 1$
About permutation problems...

- Enumerate all sets $S$ with 2 elements,
  
  $S = \{1, 2\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{2\}; f_1(7)); \max (Opt[\{1\}; f_2(7)) \right)$,
  
  $\Rightarrow Opt[\{1, 2\}] = \min \left( \max (-1; 3); \max (-1; 2) \right) = 2$

  $S = \{1, 3\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{3\}; f_1(8)); \max (Opt[\{1\}; f_3(8)) \right)$,
  
  $\Rightarrow Opt[\{1, 3\}] = \min \left( \max (-3; 4); \max (-1; 0) \right) = 0$

  $S = \{2, 3\}$:
  
  $Opt[S] = \min \left( \max (Opt[\{3\}; f_2(9)); \max (Opt[\{2\}; f_3(9)) \right)$,
  
  $\Rightarrow Opt[\{2, 3\}] = \min \left( \max (-3; 4); \max (-1; 1) \right) = 1$
About permutation problems...

- Enumerate all sets $S$ with 2 elements,
  
  $S = \{1, 2\}$ :
  
  $Opt[S] = \min \left( \max (Opt[\{2\}; f_1(7)); \max (Opt[\{1\}; f_2(7)) \right),$
  
  $\Rightarrow Opt[\{1, 2\}] = \min \left( \max (-1; 3); \max (-1; 2) \right) = 2$

- $S = \{1, 3\}$ :
  
  $Opt[S] = \min \left( \max (Opt[\{3\}; f_1(8)); \max (Opt[\{1\}; f_3(8)) \right),$
  
  $\Rightarrow Opt[\{1, 3\}] = \min \left( \max (-3; 4); \max (-1; 0) \right) = 0$

- $S = \{2, 3\}$ :
  
  $Opt[S] = \min \left( \max (Opt[\{3\}; f_2(9)); \max (Opt[\{2\}; f_3(9)) \right),$
  
  $\Rightarrow Opt[\{2, 3\}] = \min \left( \max (-3; 4); \max (-1; 1) \right) = 1$
About permutation problems...

- Enumerate all sets $S$ with 2 elements,
  - $S = \{1, 2\}$ :
    \[
    \text{Opt}[S] = \min \left( \max(\text{Opt}[\{2\}]; f_1(7)); \max(\text{Opt}[\{1\}]; f_2(7)) \right),
    \]
    $\Rightarrow \text{Opt}[\{1, 2\}] = \min \left( \max(-1; 3); \max(-1; 2) \right) = 2$
  - $S = \{1, 3\}$ :
    \[
    \text{Opt}[S] = \min \left( \max(\text{Opt}[\{3\}]; f_1(8)); \max(\text{Opt}[\{1\}]; f_3(8)) \right),
    \]
    $\Rightarrow \text{Opt}[\{1, 3\}] = \min \left( \max(-3; 4); \max(-1; 0) \right) = 0$
  - $S = \{2, 3\}$ :
    \[
    \text{Opt}[S] = \min \left( \max(\text{Opt}[\{3\}]; f_2(9)); \max(\text{Opt}[\{2\}]; f_3(9)) \right),
    \]
    $\Rightarrow \text{Opt}[\{2, 3\}] = \min \left( \max(-3; 4); \max(-1; 1) \right) = 1$
About permutation problems...

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

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

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

About permutation problems...

Enumerate all sets $S$ with 3 elements,

$S = \{1, 2, 3\} : \text{Opt}[S] = \min \left( \max (\text{Opt}[\{2, 3\}; f_1(12)));
\max (\text{Opt}[\{1, 3\}; f_2(12)));
\max (\text{Opt}[\{1, 2\}; f_3(12)) \right),$

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

This corresponds to the schedule $(1, 2, 3)$. 
About permutation problems...

- Enumerate all sets $S$ with 3 elements,

$S = \{1, 2, 3\} : \text{Opt}[S] = \min \left( \max (\text{Opt}[\{2, 3\]; f_1(12)));
\right.

\left. \quad \max (\text{Opt}[\{1, 3\]; f_2(12)));
\right.

\left. \quad \max (\text{Opt}[\{1, 2\]; f_3(12))).
\right)

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

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

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

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

- This corresponds to the schedule \((1, 2, 3)\).
About permutation problems...

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

  \[ \text{Opt}[S] = \min_{t \in S} \{ \max(\text{Opt}[S - \{t\}], f_t(P(S))) \} \]

- Usefull note: the computation of one \( \text{Opt}[] \) can be done in \( O(n) \) time.

- Fundamental question: how many computations of \( \text{Opt}[] \) 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...

\[ \text{Opt}[S] = \min_{t \in S} \{ \max \left( \text{Opt}[S - \{t\}], f_t(P(S)) \right) \} \]

- Usefull note : the computation of one \( \text{Opt}[] \) can be done in \( O(n) \) time.

- Fundamental question : how many computations of \( \text{Opt}[] \) 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...

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

- Usefull note : the computation of one \( \text{Opt}[\cdot] \) can be done in \( O(n) \) time.

- Fundamental question : how many computations of \( \text{Opt}[\cdot] \) 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...
  \[ \text{Opt}[S] = \min_{t \in S} \{ \max(\text{Opt}[S - \{t]\}; f_t(P(S))) \} \]

- Usefull note: the computation of one \text{Opt}[\cdot] can be done in \(O(n)\) time.

- Fundamental question: how many computations of \text{Opt}[\cdot] 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...
  \[ \text{Opt}[S] = \min_{t \in S} \{ \max(\text{Opt}[S - \{t\}; f_t(P(S))] \} \]

- 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 \(O^*(2^n)\),
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 across the subsets.

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 \textit{dynamic programming across the subsets}.

Dynamic Programming AtS

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>wctc</th>
<th>wcsc</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_3 \parallel C_{\text{max}} )</td>
<td>( O^*(3^n) )</td>
<td>( O^*(3^n) )</td>
</tr>
<tr>
<td>( F_3 \parallel f_{\text{max}} )</td>
<td>( O^*(5^n) )</td>
<td>( O^*(5^n) )</td>
</tr>
<tr>
<td>( F_3 \parallel \sum_i f_i )</td>
<td>( O^*(5^n) )</td>
<td>( O^*(5^n) )</td>
</tr>
</tbody>
</table>

Dynamic Programming AtS

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>wctc</th>
<th>wcsc</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F3</td>
<td></td>
<td>C_{\text{max}} )</td>
</tr>
<tr>
<td>( F3</td>
<td></td>
<td>f_{\text{max}} )</td>
</tr>
<tr>
<td>( F3</td>
<td></td>
<td>\sum_i f_i )</td>
</tr>
</tbody>
</table>

Dynamic Programming AtS

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

<table>
<thead>
<tr>
<th>Problem</th>
<th>wctc</th>
<th>wcsc</th>
</tr>
</thead>
<tbody>
<tr>
<td>F3(</td>
<td></td>
<td>C_{\text{max}})</td>
</tr>
<tr>
<td>F3(</td>
<td></td>
<td>f_{\text{max}})</td>
</tr>
<tr>
<td>F3(</td>
<td></td>
<td>\sum_i f_i)</td>
</tr>
</tbody>
</table>

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

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

  many sequences $(S - \{t\})$ (at most $2^n$) with different $C_{max}^2$ and $C_{max}^3$ must be kept in memory.
Pareto Dynamic Programming

Why a need for generalization?

The $1\|f_{\text{max}}$ problem is decomposable but, for instance, the $F3\|C_{\text{max}}$ is not.

$F3\|C_{\text{max}}$: 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 $C_{\text{max}} = \max_i(C_{i,3})$.

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 $C_{\text{max}}^2$ and $C_{\text{max}}^3$ 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 

"$Opt[S] = \min_{t \in S} \{ \max(\text{Opt}[S - \{t\}], f_t(P(S))) \}$", many sequences ($S - \{t\}$) (at most $2^n$) with different $C_{\max}^2$ and $C_{\max}^3$ must be kept in memory.
Branch-and-... What ?!

- Branch-and-Reduce (BaR) resembles 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 worst-case complexity.
- A BaR algorithm implements three components:
  - A branching rule,
  - A reduction rule at each node,
  - A stopping rule.
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 **worst-case complexity**, A BaR algorithm implements three components:

- A branching rule,
- A reduction rule at each node,
- A stopping rule.
Branch-and-Reduce (BaR) resembles 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 **worst-case complexity**, 

A BaR algorithm implements three components:

- A branching rule,
- A reduction rule at each node,
- A stopping rule.
Branch-and-Reduce (BaR) resembles 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 \textit{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 (BaR) resembles 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 **worst-case complexity**,

A BaR algorithm implements three components:

- A branching rule,
- A reduction rule at each node,
- A stopping rule.
Branch-and-Reduce (BaR) resembles 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 **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,
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).

\begin{tikzpicture}
  
  % Define vertices
  
  \node[vertex] (a) at (0,0) {$a$};
  \node[vertex] (b) at (2,0) {$b$};
  \node[vertex] (c) at (4,0) {$c$};

  % Draw edges
  
  \draw (a) -- (b);

\end{tikzpicture}
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),
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 Branch-and-Reduce 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 $v \notin S$. 
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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 graph 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,

```
   a  b  c
d  e  f
   a  b  c
d  e  f
```

- 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 graph 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,

![Graph](image)

- 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 graph 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 graph 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,

![Graph image]

- 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,

![Diagram showing a graph with vertices $a$, $b$, $c$, $d$, $e$, and $f$. $b$ is 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,

![Diagram showing a graph with vertices and edges]

- 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.
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.
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.
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.
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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.
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 $n = |V|$. 

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 $n = |V|$.
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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 $n = |V|$.
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.
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.
How can we recursively evaluate
\[ T(n) = T(n - 4) + T(n - 1) \, ? \]

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

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.
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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.
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.
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 DPA+tS is in $O^*(2^n)$ time and space.
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 $1|d_i|\sum T_i$

Let us consider the $1|d_i|\sum T_i$ 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 $\sum T_i$.

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.
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: $d_1 \leq d_2 \leq ... \leq d_n$.

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: $d_1 \leq d_2 \leq \ldots \leq d_n$.

We assume: \( p_1 \geq p_2 \geq \ldots \geq p_n \) 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

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

Worst case: \(d_1 \leq d_2 \leq \ldots \leq d_n\).

Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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

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

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

Branching scheme :

- Sphere with positions 1 to $k-1$
- Position $k$
- Sphere with positions $k+1$ to $n$

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) : $p_1 \geq p_2 \geq \ldots \geq p_n$ and $d_1 \leq d_2 \leq \ldots \leq d_n$,

- Branching scheme:

```
2, 3, ..., k
\downarrow
positions 1 to k - 1
1
\downarrow
position k
k + 1, ..., n
\downarrow
positions k + 1 to n
```

- 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): $p_1 \geq p_2 \geq \ldots \geq p_n$ and $d_1 \leq d_2 \leq \ldots \leq d_n$.
- 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 :
$n = 3$, $[p_i] = [5; 4; 3]$, $[d_i] = [6; 8; 10]$, 
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,
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,
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:

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

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

- This yields $O^*(3^n)$ worst-case time complexity, and polynomial space.
We get the following recursive relation:
\[ T(n) = 2T(n-1) + 2T(n-2) + \ldots + 2T(2) + 2T(1) + O(p(n)) \]
\[ \iff T(n) = 3T(n-1) + O(p(n)) \]
This yields \( O^*(3^n) \) worst-case time complexity, and polynomial space.
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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

- We get the following recursive relation:

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

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

- This yields $O^*(3^n)$ worst-case time complexity, and polynomial space.
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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:

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

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

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

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:

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

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

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

Exact or Heuristic Exponential-Time Algorithms
with applications to scheduling

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

- 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:

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

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

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


Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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:

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

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

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

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*,

Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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*,

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*,

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

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 $w(o_i)$,
    \[ 1 \leq i \leq n, \]
  - 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,

---

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,

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,

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,

---

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.

A complete solution \( s = s_1 + s_2 \)

- The combinatorics 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

\[ s = s_1 + s_2 \]

- The combinatorics 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 $W = 9$.

<table>
<thead>
<tr>
<th>$O$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$w$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\emptyset$</th>
<th>${a}$</th>
<th>${b}$</th>
<th>${c}$</th>
<th>${a, b}$</th>
<th>${a, c}$</th>
<th>${b, c}$</th>
<th>${a, b, c}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
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 $W = 9$.

<table>
<thead>
<tr>
<th>$O$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$w$</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

$O_1 = \{a, b, c\}$ \hspace{1cm} $O_2 = \{d, e, f\}$

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

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\emptyset$</th>
<th>{a}</th>
<th>{b}</th>
<th>{c}</th>
<th>{a, b}</th>
<th>{a, c}</th>
<th>{b, c}</th>
<th>{a, b, c}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
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 \( W = 9 \).

<table>
<thead>
<tr>
<th>( O )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v )</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( w )</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( \emptyset )</th>
<th>( {a} )</th>
<th>( {b} )</th>
<th>( {c} )</th>
<th>( {a, b} )</th>
<th>( {a, c} )</th>
<th>( {b, c} )</th>
<th>( {a, b, c} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum v )</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>( \sum w )</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
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 $W = 9$.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>w</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\emptyset$</th>
<th>${a}$</th>
<th>${b}$</th>
<th>${c}$</th>
<th>${a, b}$</th>
<th>${a, c}$</th>
<th>${b, c}$</th>
<th>${a, b, c}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
Sort & Search: the principles

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

<table>
<thead>
<tr>
<th></th>
<th>(\emptyset)</th>
<th>{e}</th>
<th>{d}</th>
<th>{f}</th>
<th>{d, e}</th>
<th>{e, f}</th>
<th>{d, f}</th>
<th>{d, e, f}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\sum v)</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>(\sum w)</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>(\ell_k)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: In table $T_2$, columns are sorted by increasing order of $\sum w$.

Note: $\ell_k$ 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 \( T_2 \)),

<table>
<thead>
<tr>
<th>( T_2 )</th>
<th>( \emptyset )</th>
<th>{e}</th>
<th>{d}</th>
<th>{f}</th>
<th>{d, e}</th>
<th>{e, f}</th>
<th>{d, f}</th>
<th>{d, e, f}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sum v )</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>( \sum w )</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>( \ell_k )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Note : In table \( T_2 \), columns are sorted by increasing order of \( \sum w \).

Note : \( \ell_k \) 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 $T_2$),

<table>
<thead>
<tr>
<th>$T_2$</th>
<th>$\emptyset$</th>
<th>${e}$</th>
<th>${d}$</th>
<th>${f}$</th>
<th>${d, e}$</th>
<th>${e, f}$</th>
<th>${d, f}$</th>
<th>${d, e, f}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>$\ell_k$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: In table $T_2$, columns are sorted by increasing order of $\sum w$.

Note: $\ell_k$ 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 $\ell_k$ 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 $\ell_k$ will be the one which maximizes $\sum v$, 

Exact or Heuristic Exponential-Time Algorithms with applications to scheduling
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 \( \ell_k \) will be the one which maximizes \( \sum v \),
Sort & Search: the principles

Table 1

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$\emptyset$</th>
<th>${a}$</th>
<th>${b}$</th>
<th>${c}$</th>
<th>${a, b}$</th>
<th>${a, c}$</th>
<th>${b, c}$</th>
<th>${a, b, c}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2

<table>
<thead>
<tr>
<th>$T_2$</th>
<th>$\emptyset$</th>
<th>${e}$</th>
<th>${d}$</th>
<th>${f}$</th>
<th>${d, e}$</th>
<th>${e, f}$</th>
<th>${d, f}$</th>
<th>${d, e, f}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum v$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$\sum w$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>$\ell_{k_k}$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Search phase ($W = 9$)

<table>
<thead>
<tr>
<th>$j$</th>
<th>$k$</th>
<th>$\emptyset$</th>
<th>${a}$</th>
<th>${b}$</th>
<th>${c}$</th>
<th>${a, b}$</th>
<th>${a, c}$</th>
<th>${b, c}$</th>
<th>${a, b, c}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w(O'<em>j) + w(O'</em>{k_k})$</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>$v(O'<em>j) + v(O'</em>{k_k})$</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

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 powerful 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 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 powerful 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 powerful 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 powerful 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 powerful 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):

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

- 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: $n_A = n_B = 2^n \Rightarrow O^*(2^{n/2})$ 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) :

  Minimize \( f(\vec{a}_j, b_k) \)
  s.t.
  \[
  g'(\vec{a}_j, b'_k) \geq 0 \\
  \vec{a}_j \in A, \ (b_k, b'_k) \in B.
  \]

- 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 : \( n_A = n_B = 2^n/2 \Rightarrow O^*(2^{n/2}) \) 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) : 

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

- 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.
- \textbf{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 $\vec{b}_k = (b^0_k, b^1_k, \ldots, b^{d_B}_k)$ of dimension $d_B + 1$,
- Let $f$ and $g_\ell$ ($1 \leq \ell \leq d_B$) 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:

$$\begin{align*}
\text{Minimize } & f(\vec{a}_j, b^0_k) \\
\text{s.t.} & g_\ell(\vec{a}_j, b^\ell_k) \geq 0, \ (1 \leq \ell \leq d_B) \\
& \vec{a}_j \in A, \ b^\ell_k \in B.
\end{align*}$$
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 $\vec{b}_k = (b_0^k, b_1^k, \ldots, b_{d_B}^k)$ of dimension $d_B + 1$,

Let $f$ and $g_\ell (1 \leq \ell \leq d_B)$ 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 :

\[
\text{Minimize } f(\vec{a}_j, b_0^k) \\
\text{s.t. } g_\ell(\vec{a}_j, b_\ell^k) \geq 0, \quad (1 \leq \ell \leq d_B) \\
\vec{a}_j \in A, \quad \vec{b}_k \in B.
\]
Sort & Search : generalization

We can extend the original *Sort & Search* approach to *Multiple Constraint Problems* (MCP),

Let be \( A = (\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_{n_A}) \) 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 \( \vec{b}_k = (b^0_k, b^1_k, \ldots, b^{d_B}_k) \) of dimension \( d_B + 1 \),

Let \( f \) and \( g_\ell \) (\( 1 \leq \ell \leq d_B \)) 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:

\[
\text{Minimize } f(\vec{a}_j, b^0_k) \\
\text{s.t.} \\
g_\ell(\vec{a}_j, b^\ell_k) \geq 0, \quad (1 \leq \ell \leq d_B) \\
\vec{a}_j \in A, \quad \vec{b}_k \in B.
\]
Sort & Search: generalization

We can extend the original Sort & Search approach to Multiple Constraint Problems (MCP),

Let be $A = (\vec{a}_1, \vec{a}_2, \ldots , \vec{a}_{n_A})$ 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

$\vec{b}_k = (b^0_k, b^1_k, \ldots , b^{d_B}_k)$ of dimension $d_B + 1$,

Let $f$ and $g_\ell$ ($1 \leq \ell \leq d_B$) 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 :

Minimize $f(\vec{a}_j, b^0_k)$

s.t.

$g_\ell(\vec{a}_j, b^\ell_k) \geq 0, \quad (1 \leq \ell \leq d_B)$

$\vec{a}_j \in A, \vec{b}_k \in B$. 
Sort & Search : generalization

We can extend the original Sort & Search approach to Multiple Constraint Problems (MCP),

Let be $A = (\vec{a}_1, \vec{a}_2, \ldots \vec{a}_{n_A})$ 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

$\vec{b}_k = (b^0_k, b^1_k, \ldots, b^{d_B}_k)$ of dimension $d_B + 1$,

Let $f$ and $g_\ell (1 \leq \ell \leq d_B)$ 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 :

Minimize $f(\vec{a}_j, b^0_k)$

s.t.

$g_\ell(\vec{a}_j, b^\ell_k) \geq 0, \quad (1 \leq \ell \leq d_B)$

$\vec{a}_j \in A, \; \vec{b}_k \in B$. 
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]).

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 $O(n_B \log_{2}^{dB}(n_B) + n_A \log_{2}^{dB+2}(n_B))$ time and $O(n_B \log_{2}^{dB-1}(n_B))$ space ([4]).

Consider the \( P3||C_{\text{max}} \) 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
  \[ C_{\text{max}} = \max_i (C_i) \]
  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 $P3||\text{C}_{\text{max}}$ 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 $C_{\text{max}} = \max_i (C_i)$ 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 $J$,
- Let $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\}$ be the subset of the first job of $J$,
- Let $I_2 = \left\{ \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \right\}$ be the subset of the last job of $J$,
- Let be $E^j_1 = (E^j_{1,1}, E^j_{1,2}, E^j_{1,3})$ a 3-partition of $I_1$ $(1 \leq j \leq 3|I_1|)$,
- We associate to it a schedule $s^j_1$ containing the sequence of jobs on machines,
- Similarly, let be $E^k_2$ a 3-partition of $I_2$ $(1 \leq j \leq 3|I_2|)$,
- We associate to it a schedule $s^k_2$ 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 $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \left\{ \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \right\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ 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 $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \left\{ \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \right\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ last job of $\mathcal{J}$,
- Let be $E_j^1 = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ 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 $E_k^2$ 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 $J$.
- Let $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $J$.
- Let $I_2 = \left\{ \left\lceil \frac{n}{2} \right\rceil + 1, \ldots, n \right\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ last job of $J$.
- Let be $E_1^j = (E_1^j, E_1^j, E_1^j)$ 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 $E_2^k$ 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 $J$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor \}$ be the subset of the first $\lfloor \frac{n}{2} \rfloor$ first job of $J$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $J$,
- Let be $E^j_1 = (E^j_{1,1}, E^j_{1,2}, E^j_{1,3})$ a 3-partition of $I_1$ ($1 \leq j \leq 3|I_1|$),
- We associate to it a schedule $s^j_1$ containing the sequence of jobs on machines,
- Similarly, let be $E^k_2$ a 3-partition of $I_2$ ($1 \leq j \leq 3|I_2|$),
- We associate to it a schedule $s^k_2$ 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 $J$,
- Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor \}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $J$,
- Let $I_2 = \{\lfloor \frac{n}{2} \rfloor + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $J$,
- Let be $E^j_1 = (E^j_{1,1}, E^j_{1,2}, E^j_{1,3})$ a 3-partition of $I_1$ ($1 \leq j \leq 3|I_1|$),
- We associate to it a schedule $s^j_1$ containing the sequence of jobs on machines,
- Similarly, let be $E^k_2$ a 3-partition of $I_2$ ($1 \leq j \leq 3|I_2|$),
- We associate to it a schedule $s^k_2$ 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 $J$,

Let $I_1 = \{1, \ldots, \lfloor \frac{n}{2} \rfloor \}$ be the subset of the $\lfloor \frac{n}{2} \rfloor$ first job of $J$,

Let $I_2 = \{\lceil \frac{n}{2} \rceil + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $J$,

Let be $E_{1,j} = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ 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 $E_{2,k}$ 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 $J$,
- Let $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $J$,
- Let $I_2 = \{\left\lceil \frac{n}{2} \right\rceil + 1, \ldots, n\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ last job of $J$,
- Let be $E^j_1 = (E^j_{1,1}, E^j_{1,2}, E^j_{1,3})$ a 3-partition of $I_1$ ($1 \leq j \leq 3|I_1|$),
- We associate to it a schedule $s^j_1$ containing the sequence of jobs on machines,
- Similarly, let be $E^k_2$ a 3-partition of $I_2$ ($1 \leq j \leq 3|I_2|$),
- We associate to it a schedule $s^k_2$ 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 $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ 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 $J$, 
Let $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $J$, 
Let $I_2 = \left\lceil \frac{n}{2} \right\rceil + 1, \ldots, n\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ last job of $J$, 
Let be $E_{j_1}^j = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ 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 $E_{j_2}^k$ 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 $J$,
- Let $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $J$,
- Let $I_2 = \left\lceil \frac{n}{2} \right\rceil + 1, \ldots, n \}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ last job of $J$,
- Let be $E^j_1 = (E^{j,1}_1, E^{j,2}_1, E^{j,3}_1)$ a 3-partition of $I_1$ $(1 \leq j \leq 3^{|I_1|})$,
- We associate to it a schedule $s^{j}_1$ containing the sequence of jobs on machines,
- Similarly, let be $E^k_2$ a 3-partition of $I_2$ $(1 \leq j \leq 3^{|I_2|})$,
- We associate to it a schedule $s^{k}_2$ 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 $I_1 = \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \}$ be the subset of the $\left\lfloor \frac{n}{2} \right\rfloor$ first job of $\mathcal{J}$,
- Let $I_2 = \left\{ \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \right\}$ be the subset of the $\left\lceil \frac{n}{2} \right\rceil$ 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 $I_2 = \{\lceil \frac{n}{2} \rceil + 1, \ldots, n\}$ be the subset of the $\lceil \frac{n}{2} \rceil$ last job of $\mathcal{J}$,
- Let be $E_j^1 = (E_{1,1}^j, E_{1,2}^j, E_{1,3}^j)$ a 3-partition of $I_1$ ($1 \leq j \leq 3|I_1|$),
- We associate to it a schedule $s_j^1$ containing the sequence of jobs on machines,
- Similarly, let be $E_j^2$ a 3-partition of $I_2$ ($1 \leq j \leq 3|I_2|$),
- We associate to it a schedule $s_j^2$ 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 $\ell$ 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 $\ell$ 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 $\ell$ 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 $\ell$ in $s$, $\ell \in [1, 3]$,
- Let $P(s)$ be the sum of processing times of all jobs of $s$, 

\[
\begin{array}{c}
\text{Machine 1} \\
\hline
P_1(s_1) & P_1(s_2) \\
\text{Machine 2} \\
\hline
P_2(s_1) & P_2(s_2) \\
\text{Machine 3} \\
\hline
P_3(s_1) & P_3(s_2) \\
\hline
\delta_1(s_1) & \delta_2(s_1) \\
\end{array}
\]

\[C_{\text{max}}\]
Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine $\ell$.

We have:

$$
\sum_{\ell=1}^{2} \delta_\ell(s) = \sum_{\ell=1}^{2} \delta_\ell(s) = 3P_3(s) - P(s).
$$

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 $\sigma$,

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 $\ell$.

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 $C_{\text{max}}$ value.

Now, let us concentrate on the concatenation of two partial schedules $s$ and $\sigma$.

We have: $C_{\text{max}}(s\sigma) = \max_{1 \leq \ell \leq 3} (P_\ell(s) + P_\ell(\sigma))$. 

- Without loss of generality, we can restrict the schedule where the last machine gives the $C_{\text{max}}$ value.
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 $\ell$,

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 $C_{\text{max}}$ value,

Now, let us concentrate on the concatenation of two partial schedules $s$ and $\sigma$,

We have: $C_{\text{max}}(s\sigma) = \max_{1 \leq \ell \leq 3} (P_\ell(s) + P_\ell(\sigma))$. 
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 $\ell$.

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 $C_{\text{max}}$ value.

Now, let us concentrate on the concatenation of two partial schedules $s$ and $\sigma$,

We have: $C_{\text{max}}(s\sigma) = \max_{1 \leq \ell \leq 3} (P_\ell(s) + P_\ell(\sigma))$. 

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 \( \ell \),

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 \( C_{\text{max}} \) value,

Now, let us concentrate on the concatenation of two partial schedules \( s \) and \( \sigma \),

We have: \( C_{\text{max}}(s\sigma) = \max_{1 \leq \ell \leq 3} (P_\ell(s) + P_\ell(\sigma)) \).
Let us define $\delta_\ell(s) = P_3(s) - P_\ell(s)$ as the difference between the load of the last machine and machine $\ell$.

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 $\sigma$,

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 $\ell$.

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 $\sigma$,

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):

$$\forall \ell \in \llbracket 1, 2 \rrbracket, \delta_\ell(s) + \delta_\ell(\sigma) \geq 0$$

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

Reformulation

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

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

s.t. $\forall \ell \in [1, 2], \delta_\ell(s) + \delta_\ell(\sigma) \geq 0$
Sort & Search : an application (main lines)

\[
\begin{align*}
\vec{a}_j &= (\delta_1(s_1^j), \delta_2(s_1^j)) \\
(b_k^0, b_k^1, b_k^2) &= (\delta_1(s_2^k) + \delta_2(s_2^k), \delta_1(s_2^k), \delta_2(s_2^k)) \\
f(\vec{a}_j, b_k^0) &= (P + \delta_1(s_1^j) + \delta_2(s_1^j) + \delta_1(s_2^k) + \delta_2(s_2^k))/3 \\
g_1(\vec{a}_j, b_k^1) &= \delta_1(s_1^j) + \delta_1(s_2^k) \\
g_2(\vec{a}_j, b_k^2) &= \delta_2(s_1^j) + \delta_2(s_2^k)
\end{align*}
\]

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 \( n_A = n_B = 3^n \) columns,
- Besides, \( d_A = 2 \), and \( d_B = 2 \)
- Then, the worst-case time complexity is in \( O(3^n \log_2^2 (3^n) + 3^n \log_2^4 (3^n)) = O^*(3^n) \approx O^*(1.7321^n) \).
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 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 caused 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.
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.
1. Introduction

2. Exact Exponential-Time Algorithms

3. Heuristic Exponential-Time Algorithms

4. Conclusions
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 $P|d_i|\sum_i U_i$.
- $NP$-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: $N = \{1, \ldots, n\}$, $m$ 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 $\rho$.

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 $\rho$.

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 $P^2|d_i|\sum_i U_i$ does not admit a polynomial-time approximation algorithm with a bounded ratio $\rho$.
- Deciding the existence of a schedule with $\sum_i U_i^* = 0$ is $NP$-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 $H$ with a worst-case guarantee:

- Fixed ratio: $\frac{Z^H}{Z^{0_{\text{opt}}}} \leq \rho$ and polynomial time in input length,
- PTAS: $\frac{Z^H}{Z^{0_{\text{opt}}}} \leq (1 + \epsilon)$ and polynomial time in input length when $\epsilon$ is fixed,
- FPTAS: $\frac{Z^H}{Z^{0_{\text{opt}}}} \leq (1 + \epsilon)$ 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 \( H \) with a worst-case guarantee:

- Fixed ratio: \( \frac{Z^H}{Z^{0pt}} \leq \rho \) and polynomial time in input length,
- PTAS: \( \frac{Z^H}{Z^{0pt}} \leq (1 + \epsilon) \) and polynomial time in input length when \( \epsilon \) is fixed,
- FPTAS: \( \frac{Z^H}{Z^{0pt}} \leq (1 + \epsilon) \) 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 \(NP\)-hard minimization problems, polynomial-time heuristic algorithms \(H\) with a worst-case guarantee:

- Fixed ratio: \(\frac{Z^H}{Z^{0pt}} \leq \rho\) and polynomial time in input length,
- PTAS: \(\frac{Z^H}{Z^{0pt}} \leq (1 + \epsilon)\) and polynomial time in input length when \(\epsilon\) is fixed,
- FPTAS: \(\frac{Z^H}{Z^{0pt}} \leq (1 + \epsilon)\) 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 $O^*(c^n)$, 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 worst-case time complexity in $\mathcal{O}^*(c^n)$.

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 $O^*(m^{n/2})$ 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 $O^*((m + 1)^{n/2})$ time and space in the worst case.

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 $O^*(m^{n/2})$ 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 $O^*((m + 1)^{n/2})$ time and space in the worst case.

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 $O^*(m^{\frac{n}{2}})$ 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 $O^*((m + 1)^{\frac{n}{2}})$ time and space in the worst case.

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 \( P|\tilde{d}_i|\sum_i U_i \) problem can be decided in \( O^*(m^{n/2}) \) time and space.

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

We propose a first approximation algorithm, referred to as B\text{approx},

Let $k \in \mathbb{N}^*$ be a given parameter,

Wlog, we assume $d_1 \leq d_2 \leq \ldots \leq d_n$,

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 $B_\ell$ contains jobs $\{(\ell - 1) \ast k + 1, \ldots, \ell k\}$, $1 \leq \ell \leq \lfloor \frac{n}{k} \rfloor$. 
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 \( d_1 \leq d_2 \leq \ldots \leq d_n \),
- 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 \( \left\lceil \frac{n}{k} \right\rceil \) batches.
  - Each batch \( B_\ell \) contains jobs \( \{(\ell - 1) \times k + 1, \ldots, \ell k\} \), \( 1 \leq \ell \leq \left\lfloor \frac{n}{k} \right\rfloor \).
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 $d_1 \leq d_2 \leq \ldots \leq d_n$,
- 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 $\left\lceil \frac{n}{k} \right\rceil$ batches. Each batch $B_\ell$ contains jobs $\{(\ell - 1) \times k + 1, \ldots, \ell k\}$, $1 \leq \ell \leq \left\lfloor \frac{n}{k} \right\rfloor$. 
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 $\left\lceil \frac{n}{k} \right\rceil$ batches.
  Each batch $B_\ell$ contains jobs $\{(\ell - 1)\cdot k + 1, ..., \ell k\}$, $1 \leq \ell \leq \left\lfloor \frac{n}{k} \right\rfloor$. 
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 $d_1 \leq d_2 \leq \ldots \leq d_n$,
- 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 $\left\lceil \frac{n}{k} \right\rceil$ batches. Each batch $B_\ell$ contains jobs $\{(\ell - 1) \cdot k + 1, \ldots, \ell k\}$, $1 \leq \ell \leq \left\lfloor \frac{n}{k} \right\rfloor$.
Algorithm outline

- Algorithm $B_{\text{approx}}$ builds a binary search tree by branching at each level $\ell$ on batch $B_{\ell}$ and scheduling all its jobs either early or 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 $\forall i \in N \setminus E_s, \tilde{d}_i = +\infty$
Algorithm outline

- Algorithm \texttt{Bapprox} builds a binary search tree by branching at each level \( \ell \) on batch \( B_{\ell} \) and scheduling all its jobs either early or 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.

  \[ \Rightarrow \forall i \in E_s, \tilde{d}_i = d_i, \text{ and } \forall i \in N \setminus E_s, \tilde{d}_i = +\infty \]
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 $\frac{2}{1} = \ldots$?
Exact or Heuristic Exponential-Time Algorithms with applications to scheduling

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 $\frac{2}{1} = ...$?
Algorithm outline

- Branch and evaluate all leaf nodes,

![Diagram showing a tree structure with branch and evaluation]

- 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 $O^*((1 + m^{\frac{k}{2}})^{\frac{n}{k}})$ time and $O^*(m^{\frac{n}{2}})$ space.

Proof (sketch) : Ratio.
- Assume $\alpha$ 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, $\rho \leq \sum_{u=1}^{\alpha} \frac{\alpha k}{\ell_u}$.
- The ratio is maximum when $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$. 
Analysis

Theorem 2

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

Proof (sketch) : Ratio.

- Assume \( \alpha \) batches are scheduled tardy by \( B_{\text{approx}} \),
- But what is the situation in an optimal schedule?
- If \( B_{\text{approx}} \) 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, \( \rho \leq \sum_{u=1}^{\alpha} \frac{\alpha k}{\ell_u} \),
- The ratio is maximum when \( \sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k \).
Theorem 2

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

Proof (sketch) : Ratio.

- Assume \( \alpha \) batches are scheduled tardy by \( B_{\text{approx}} \),
- But what is the situation in an optimal schedule?
- If \( B_{\text{approx}} \) 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 $B_{approx}$ admits a worst-case ratio $\rho \leq k$ (tight).
Algorithm $B_{approx}$ requires $O^*((1 + m^{k/2})^{n/k})$ time and $O^*(m^{n/2})$ space.

Proof (sketch) : Ratio.

- Assume $\alpha$ batches are scheduled tardy by $B_{approx}$,
- But what is the situation in an optimal schedule?
- If $B_{approx}$ 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 \( \text{Bapprox} \) admits a worst-case ratio \( \rho \leq k \) (tight).
Algorithm \( \text{Bapprox} \) requires \( O^*((1 + m \frac{k}{2}) \frac{n}{k}) \) time and \( O^*(m \frac{n}{2}) \) space.

Proof (sketch) : Ratio.

- Assume \( \alpha \) batches are scheduled tardy by \( \text{Bapprox} \),
- But what is the situation in an optimal schedule?
- If \( \text{Bapprox} \) 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 \( \sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k. \)
Theorem 2

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

Proof (sketch) : Ratio.

- Assume $\alpha$ batches are scheduled tardy by \text{Bapprox},
- But what is the situation in an optimal schedule?
- If \text{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 $O^{\ast}\left((1 + m^{k/2})\frac{n}{k}\right)$ time and $O^{\ast}\left(m^{n/2}\right)$ space.

Proof (sketch) : Ratio.

- Assume $\alpha$ 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 $\sum_{u=1}^{\alpha} \ell_u = \alpha \Rightarrow \rho \leq k$. 
Algorithm \( B_{\text{approx}} \) admits a worst-case ratio \( \rho \leq k \) (tight).
Algorithm \( B_{\text{approx}} \) requires \( O^*((1 + m^{\frac{k}{2}})^{\frac{n}{k}}) \) time and \( O^*(m^{\frac{n}{2}}) \) space.

Proof (sketch) : Ratio.

- Assume \( \alpha \) batches are scheduled tardy by \( B_{\text{approx}} \).
- But what is the situation in an optimal schedule?
- If \( B_{\text{approx}} \) 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 $O^*((1 + m^{k/2})^{n/k})$ time and $O^*(m^{n/2})$ space.

Proof: worst-case time complexity.
- Initial (feasibility) step requires $O^*(m^{n/2})$ 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: $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil n/k \rceil} \binom{n/k}{\ell}$. 
Analysis

Theorem 2

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

- Proof: worst-case time complexity.
  - Initial (feasibility) step requires \( \mathcal{O}^*(m^{n/2}) \) time.
  - Let \( \mathcal{L}\mathcal{N} \) 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: \( |\mathcal{L}\mathcal{N}| = \sum_{\ell=0}^{\lceil n/k \rceil} (\lceil n/k \rceil) \).
Theorem 2

Algorithm Bapprox admits a worst-case ratio $\rho \leq k$ (tight).

Algorithm Bapprox requires $O^*((1 + m^{k/2})^{n/k})$ time and $O^*(m^{n/2})$ space.

Proof: worst-case time complexity.

- Initial (feasibility) step requires $O^*(m^{n/2})$ 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: $|\mathcal{LN}| = \sum_{\ell=0}^{\lceil n/k \rceil} \binom{\lceil n/k \rceil}{\ell}$. 
Analysis

Theorem 2

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

Proof: worst-case time complexity.

- Initial (feasibility) step requires $\mathcal{O}^*(m^{n/2})$ time.
- Let $\mathcal{L}_N$ 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: $|\mathcal{L}_N| = \sum_{\ell=0}^{\lfloor n/k \rfloor} \left( \begin{array}{c} n \vdash k \end{array} \right)$. 

---

Exact or Heuristic Exponential-Time Algorithms with applications to scheduling
Analysis

Theorem 2

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

**Proof:** worst-case time complexity.

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

$$O^*(\sum_{\ell=0}^{\lfloor n/k \rfloor} \left( \frac{n}{k} \right)^\ell (m^{k/2})^\ell)$$

$\Leftrightarrow O^*((1 + m^{k/2})^{n/k}),$

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 \( \mathcal{O}^*\left((1 + m^{k/2})^{n/k}\right) \) time and \( \mathcal{O}^*(m^{n/2}) \) space.

Proof: worst-case time complexity.

\[ \forall (E; T) \in \mathcal{L}\mathcal{N}, \text{ deciding of the feasibility requires } \mathcal{O}^*(m^{|E|/2}) \text{ time, with } |E| = k\ell \text{ and } \ell \text{ the number of early batches in } E. \]

It follows that to build and test all leaf nodes the worst-case running time is in:

\[ \mathcal{O}^*\left(\sum_{\ell=0}^{\left\lceil n/k \right\rceil} \left\lceil n/k \right\rceil \left( m^{k/2} \right) \ell\right) \]

\[ \Leftrightarrow \mathcal{O}^*\left((1 + m^{k/2})^{n/k}\right), \]

by making use of the Newton’s binomial formula.
Analysis

Theorem 2

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

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

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\rho$</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$O(2.4142^n)$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$O(1.7320^n)$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$O(1.5643^n)$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$O(1.4953^n)$</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$O(1.4610^n)$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>$O(1.4186^n)$</td>
</tr>
</tbody>
</table>

Noteworthy, by comparison with the EETA running in $O(1.7320^n)$ time, algorithm $B_{\text{approx}}$ is relevant for $k \geq 3$. 
How to decrease the ratio $\rho$ 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 $\rho$ of algorithm $B_{\text{approx}}$?
- We add a preprocessing step (algorithm $P_{\text{Bapprox}}$).
- 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 $B_{\text{approx}}$ is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).
Branching and preprocessing

- How to decrease the ratio $\rho$ of algorithm $B_{\text{approx}}$?
- We add a preprocessing step (algorithm $PB_{\text{approx}}$).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\left\lfloor \frac{n}{c} \right\rfloor$ 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 $B_{\text{approx}}$ is used (on each subset of size $(n - \left\lfloor \frac{n}{c} \right\rfloor)$).
Branching and preprocessing

- How to decrease the ratio $\rho$ of algorithm $B_{\text{approx}}$?
- We add a preprocessing step (algorithm $P_{\text{Bapprox}}$).
- 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 $B_{\text{approx}}$ is used (on each subset of size $(n - \lfloor \frac{n}{c} \rfloor)$).
Branching and preprocessing

- How to decrease the ratio $\rho$ of algorithm $B_{\text{approx}}$?
- We add a preprocessing step (algorithm $PB_{\text{approx}}$).
- Let us introduce a parameter $c \in \mathbb{N}^*$.
- The preprocessing generates all possible subsets of at most $\left\lfloor \frac{n}{c} \right\rfloor$ 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 $B_{\text{approx}}$ is used (on each subset of size $(n - \left\lfloor \frac{n}{c} \right\rfloor)$).
Algorithm PB\text{approx} admits a worst-case ratio $\rho \leq \frac{k^2 + k(c-1) + 1}{k + c}$. 
Analysis

Theorem 4

Algorithm PBapprox requires
\[ O^* \left( \max \left( 2^H(c)n^m \frac{n(c-1)}{2c} ; (ce) \frac{n}{c} \left( 1 + m^k \right) \frac{n(c-1)}{ck} \right) \right) \] 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 \( O^*(m \frac{n(c-1)}{2c}) \).

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

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

- This is a partial sum of binomials!
Analysis

Theorem 4

Algorithm $\text{PBapprox}$ requires

$$\mathcal{O}^*\left(\max\left(2^{H(c)}n^m \frac{n(c-1)}{2c}; (ce)\frac{n}{c} \left(1 + m\frac{k}{2}\right) \frac{n(c-1)}{ck}\right)\right)$$

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 $\left\lfloor \frac{n}{c} \right\rfloor$ tardy jobs and solution of feasibility problems on the early jobs. Worst-case running time:

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

This is a partial sum of binomials!
Theorem 4

Algorithm PBapprox requires
\[ \mathcal{O}^* \left( \max \left( 2^H(c)n^\frac{\frac{(c-1)}{2c}}{m}; \left( ce \right)^\frac{n}{c} \left( 1 + m^\frac{k}{2} \right)^\frac{\frac{(c-1)}{ck}}{n} \right) \right) \] time,
where \( H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \) and \( e \) is Euler’s number.

The worst-case space requirement is in \( \mathcal{O}^* \left( m^\frac{\frac{(c-1)}{2c}}{n} \right) \).

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

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

- This is a partial sum of binomials!
Theorem 4

Algorithm PBapprox requires
\( O^* \left( \max(2^{H(c)n} m^{\frac{n(c-1)}{2c}} ; (ce)^{\frac{n}{c}} (1 + m^2) \frac{n(c-1)}{ck}) \right) \) time,
where \( H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \) and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^*\left( m^{\frac{n(c-1)}{2c}} \right) \).

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
\[ O^* \left( \max \left( 2^H(c)n m^{n(c-1)/2c} ; (ce) \frac{n}{c} (1 + m^{k/2c})^{n(c-1)/ck} \right) \right) \text{ time}, \]
where \( H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \) and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^*(m^{n(c-1)/2c}) \).

- Proof: worst-case time complexity.
- We obtain the following reformulation:
  \[
  \sum_{i=0}^{\left\lfloor \frac{n}{c} \right\rfloor} \binom{n}{i} m^{\frac{n-i}{2}} \leq \sum_{i=0}^{\left\lfloor \frac{n}{c} \right\rfloor} \binom{n}{i} \times m^{\frac{n(c-1)}{2c}} \leq 2^H(c)n m^{\frac{n(c-1)}{2c}}.
  \]
- The preprocessing phase has a worst-case time complexity in \( O^*(2^H(c)n m^{n(c-1)/2c}) \).
Analysis

Theorem 4

Algorithm PBapprox requires
\[ O^* \left( \max \left( 2^{H(c)n} m \frac{n(c-1)}{2c}; \left( ce \right) \frac{n}{c} \left( 1 + m \frac{k}{ck} \right) \frac{n(c-1)}{ck} \right) \right) \] 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 \( O^* \left( m \frac{n(c-1)}{2c} \right) \).

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

\[
\sum_{i=0}^{\left\lfloor \frac{n}{c} \right\rfloor} \binom{n}{i} m \frac{n-i}{2} \leq \sum_{i=0}^{\left\lfloor \frac{n}{c} \right\rfloor} \binom{n}{i} \times m \frac{n(c-1)}{2c} \leq 2^{H(c)n} m \frac{n(c-1)}{2c}.
\]

- The preprocessing phase has a worst-case time complexity in \( O^* \left( 2^{H(c)n} m \frac{n(c-1)}{2c} \right) \).
Theorem 4

Algorithm PBapprox requires
\[ O^* \left( \max \left( 2^H(c) n m \frac{n(c-1)}{2c} ; \ (ce) \frac{n}{c} (1 + m \frac{k}{2}) \frac{n(c-1)}{ck} \right) \right) \] time,
where
\[ H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \] and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^* \left( m \frac{n(c-1)}{2c} \right) \).

- **Proof**: worst-case time complexity.

- The branching phase: algorithm Bapprox requires
\[ O^* \left( (1 + m \frac{k}{2}) \frac{n - \lfloor \frac{n}{c} \rfloor}{k} \right) \] time.

- The branching phase has a worst-case running time in:
\[ O^* \left( \left( \left( \frac{n}{c} \right) \right) (1 + m \frac{k}{2}) \frac{n - \lfloor \frac{n}{c} \rfloor}{k} \right) \].
Theorem 4

Algorithm PBapprox requires
\[ O^* \left( \max(2^{H(c)n}m^{n(c-1)/2c}; (ce)^{n/c}(1 + m^{k/2})^{n(c-1)/ck}) \right) \text{ time,} \]

\[ H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \] and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^*(m^{n(c-1)/2c}) \).

- Proof: worst-case time complexity.
- The branching phase: algorithm Bapprox requires
\[ O^* \left( (1 + m^{k/2})^{n - \left\lfloor \frac{n}{c} \right\rfloor} \right) \text{ time.} \]
- The branching phase has a worst-case running time in:
\[ O^* \left( \left( \left\lfloor \frac{n}{c} \right\rfloor \right) \left(1 + m^{k/2} \right)^{n - \left\lfloor \frac{n}{c} \right\rfloor} \right). \]
Algorithm PBapprox requires
\[ O^*(\max\left(2^H(c)n m^{n(c-1)/2c}; (ce)^{n/c}\left(1 + m^{k/2}\right)^{n(c-1)/ck}\right)) \] time,
where \( H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \) and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^*(m^{n(c-1)/2c}) \).

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

Theorem 4

Algorithm PBapprox requires

\[ \mathcal{O}^* \left( \max \left( 2^H(c)n m^{\frac{n(c-1)}{2c}}; (ce) \frac{n}{c} \left(1 + m^{\frac{k}{2}} \frac{n(c-1)}{ck}\right) \right) \right) \]

time,

\[ H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \text{ and } e \text{ is Euler’s number}. \]

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:

\[
\left( \frac{n}{\lceil \frac{n}{c} \rceil} \right) \left(1 + m^{\frac{k}{2}} \right)^{\frac{n}{c}} < \left( \frac{ne}{\lceil \frac{n}{c} \rceil} \right)^{\frac{n}{c}} \left(1 + m^{\frac{k}{2}} \right)^{\frac{n(c-1)}{ck}}
\]

\[
< (ce) \frac{n}{c} \left(1 + m^{\frac{k}{2}} \right)^{\frac{n(c-1)}{ck}}.
\]

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

Theorem 4

Algorithm PBapprox requires

\[ O^* \left( \max \left( 2^{H(c)} n^m \frac{n(c-1)}{2c}; \left( ce \right) \frac{n}{c} \left( 1 + m \frac{k}{c} \right) \frac{n(c-1)}{ck} \right) \right) \] 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 \( O^* \left( m \frac{n(c-1)}{2c} \right) \).

- Proof: worst-case time complexity.
- By noting that \( \left( \frac{N}{K} \right) < \frac{N K e^K}{K^K} \) with \( e \) being Euler’s number, we obtain:

\[
\left( \frac{n}{\lfloor n/c \rfloor} \right) \left( 1 + m \frac{k}{2} \right) \frac{n - \lfloor n/c \rfloor}{k} < \left( \frac{ne}{\lfloor n/c \rfloor} \right) \left( 1 + m \frac{k}{2} \right) \frac{n(c-1)}{ck}
\]

\[
< \left( ce \right) \frac{n}{c} \left( 1 + m \frac{k}{2} \right) \frac{n(c-1)}{ck}.
\]

Finally, we have the worst-case time complexity stated in the theorem.
Exact or Heuristic Exponential-Time Algorithms
with applications to scheduling

Analysis

Theorem 4

Algorithm PBapprox requires
\[ O^*(\max(2^{H(c)}n^\frac{n(c-1)}{2c}; (ce)^\frac{n}{c}(1 + m^\frac{k}{2})^\frac{n(c-1)}{ck})) \] time,

where \( H(c) = -c \log_2(c) - (1 - c) \log_2(1 - c) \) and \( e \) is Euler’s number.

The worst-case space requirement is in \( O^*(m^\frac{n(c-1)}{2c}) \).

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:

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

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

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

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\rho$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$O(2.4142^n)$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$O(1.7320^n)$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$O(1.5643^n)$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$O(1.4953^n)$</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$O(1.4610^n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Rightarrow$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>$O(1.4186^n)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c$</th>
<th>$\rho$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1000</td>
<td>2.99</td>
<td>$O(1.5760^n)$</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.98</td>
<td>$O(1.6471^n)$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.84</td>
<td>$O(2.0813^n)$</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>3.99</td>
<td>$O(1.5066^n)$</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>3.97</td>
<td>$O(1.5752^n)$</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3.78</td>
<td>$O(1.9984^n)$</td>
</tr>
</tbody>
</table>

...
Generalizations

- **Weighted case**: $P|d_i| \sum w_i U_i$.
- Algorithm $B_{\text{approx}}$ can be generalized by changing the branching scheme (and making the wct analysis more complicated).
- **Ratio**: $\rho = k$,
- **Worst-case time complexity**: $O^*(\gamma^n)$ time and $O^*(m^{\frac{n}{2}})$ 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 $B\text{approx}$ can be generalized by changing the branching scheme (and making the wct analysis more complicated),

- Ratio: $\rho = k$,

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

\[ \text{Diagram:} \]
Generalizations

- **Weighted case**: $P|d_i|\sum_i w_i U_i$,
- Algorithm B\textit{appro}x can be generalized by changing the branching scheme (and making the wct analysis more complicated),

\begin{center}
\begin{tikzpicture}[level distance=15mm, level 1/.style={sibling distance=30mm}, level 2/.style={sibling distance=15mm}]

[level 1] (root) {root node [fill=white]}

[level 2, anchor=center] (root) {job 1 is early}

[level 2, anchor=center] (root) {jobs \{1, ..., k\} are tardy}

[level 3] (root) {job 2 is early}

[level 3] (root) {jobs in \{2, ..., k + 1\} are tardy}

[level 4] (root) {job k + 1 is early}

[level 4] (root) {jobs in \{k + 1, ..., 2k\} are tardy}

\end{tikzpicture}
\end{center}

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

- Weighted case: $P|d_i| \sum_i w_i U_i$,
- Algorithm $B_{approx}$ 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 $\mathcal{O}^*(m^{n/2})$ space, with $\gamma = m^{1/2\delta}$ and $\gamma^{-k} + \gamma^{-1+\delta} = 1$. 
1 Introduction

2 Exact Exponential-Time Algorithms

3 Heuristic Exponential-Time Algorithms

4 Conclusions
Conclusions

- The $P|d_i| \sum_i U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm $B_{approx}$: a branching-based heuristic,
- Algorithm $PB_{approx}$: improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of $PB_{approx}$,
- 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 $B_{approx}$: a branching-based heuristic,
- Algorithm $PB_{approx}$: improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of $PB_{approx}$,
- 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 $B_{\text{approx}}$ : a branching-based heuristic,
- Algorithm $PB_{\text{approx}}$ : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of $PB_{\text{approx}},$
- 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 $P|d_i| \sum U_i$ problem can be approximated by moderately exponential-time algorithms,
- Algorithm $B_{\text{approx}}$: a branching-based heuristic,
- Algorithm $P_{\text{Bapprox}}$: improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of $P_{\text{Bapprox}},$
- 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 $B_{approx}$: a branching-based heuristic,
- Algorithm $PB_{approx}$: improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of $PB_{approx}$,
- 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,
- Algorithm PBAapprox : improvement by adding a preprocessing phase,
- Need for improving the analysis of the worst-case time complexity of PBAapprox,
- 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!