The Thousand Faces of Constraint Propagation

Emmanuel Hebrard

LAAS-CNRS

Toulouse

Outline

Constraint Programming

- Solving hard combinatorial problems by Decomposition

Constraint Programming

- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ⋆ into known/easy/small subproblems

Constraint Programming

- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ⋆ into known/easy/small subproblems
  - Bad news: Solving each subproblem is not enough

Constraint Programming

- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ⋆ into known/easy/small subproblems
  - Bad news: Solving each subproblem is not enough
- Constraint Progamming:

Constraint Programming

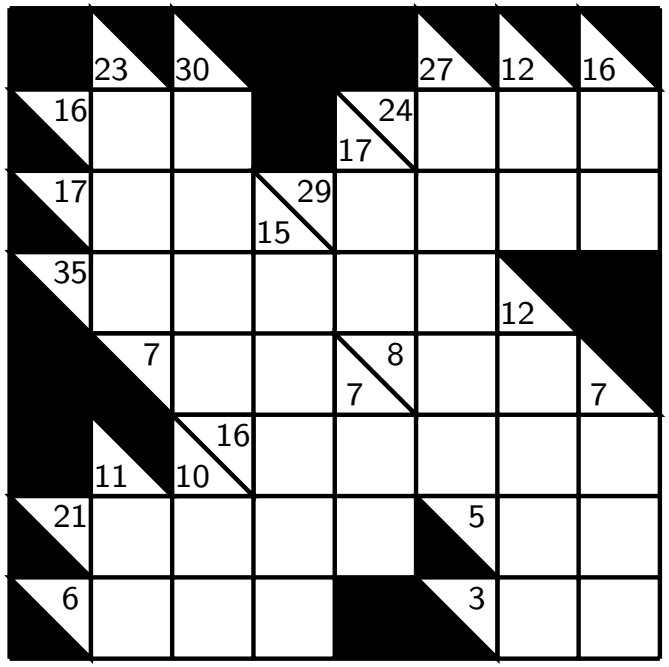
- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ⋆ into known/easy/small subproblems
  - Bad news: Solving each subproblem is not enough
- Constraint Progamming:
  - Constraint $\Leftrightarrow$ "easy" Subproblem

Constraint Programming

- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ⋆ into known/easy/small subproblems
  - Bad news: Solving each subproblem is not enough
- Constraint Progamming:
  - Constraint ⇔ "easy" Subproblem
  - Propagation to link the reasoning done on each constraint

Constraint Programming

- Solving hard combinatorial problems by Decomposition
  - Good news: We can always decompose
    - ★ into known/easy/small subproblems
  - Bad news: Solving each subproblem is not enough
- Constraint Progamming:
  - Constraint ⇔ "easy" Subproblem
  - Propagation to link the reasoning done on each constraint



Constraint Satisfaction Problem

Constraint Satisfaction Problem

- Variables: $x_1, \ldots, x_n$

Constraint Satisfaction Problem

- Variables: $x_1, \ldots, x_n$
- Constraints: $C_1, \ldots, C_k$
  - Relations = set of solutions

Constraint Satisfaction Problem

- Variables: $x_1, \ldots, x_n$
- Constraints: $C_1, \ldots, C_k$
  - Relations = set of solutions
- Domains: $D_1, \ldots, D_n$
  - Domain representation $\simeq$ relaxation of the problem's solutions
    - ★ $D_1 \times D_2 \times \ldots \times D_n$ is a super set of the solutions

Domain Representation (discrete domains)

- 4 Letters Palindrome City Problem
  - Con. 1: It is a city
  - Con. 2: It is a palindrome
  - Domain: any 2 letters (676 sol.)

Domain Representation (discrete domains)

- 4 Letters Palindrome City Problem
  - Con. 1: It is a city
  - Con. 2: It is a palindrome
  - Domain: any 2 letters (676 sol.)

L_1	L_2	L_2	L_1
A	K	K	A
A	N	N	A
I	L	L	I
M	A	A	M
O	T	T	O

Domain Representation (discrete domains)

- 4 Letters Palindrome City Problem
  - Con. 1: It is a city
  - Con. 2: It is a palindrome
  - Domain: any 2 letters (676 sol.)

L_1	L_2	L_2	L_1
A	K	K	A
A	N	N	A
I	L	L	I
M	A	A	M
O	T	T	O

{A,I,M,O} {A,K,L,N,T}

Domain Representation (discrete domains)

- 4 Letters Palindrome City Problem
  - Con. 1: It is a city
  - Con. 2: It is a palindrome
  - Domain: any 2 letters (676 sol.)

L_1	L_2	L_2	L_1
A	K	K	A
A	N	N	A
I	L	L	I
M	A	A	M
O	T	T	O

{A,I,M,O} {A,K,L,N,T}

- Minimal Relaxation (discrete domain)

A	A	A	A
A	K	K	A
A	L	L	A
A	N	N	A
A	T	T	A
I	A	A	I
I	K	K	I
I	L	L	I
I	N	N	I
I	T	T	I
M	A	A	M
M	K	K	M
M	L	L	M
M	N	N	M
M	T	T	M
O	A	A	O
O	K	K	O
O	L	L	O
O	N	N	O
O	T	T	O

Domain Representation (bounds)

- 4 Letters Palindrome City Problem
  - Con. 1: It is a city
  - Con. 2: It is a palindrome
  - Domain: any 2 letters (676 sol.)

L_1	L_2	L_2	L_1
A	K	K	A
A	N	N	A
I	L	L	I
M	A	A	M
O	T	T	O

[A,. . . ,O] [A,. . . ,T]

- Minimal Relaxation (discrete bounds)

A	A	A	A
A	B	B	A
A	C	C	A
A	D	D	A
A	E	E	A
A	F	F	A
A	G	G	A
A	H	H	A
A	I	I	A
A	J	J	A
A	K	K	A
A	L	L	A
A	M	M	A
A	N	N	A
A	O	O	A
A	P	P	A
A	Q	Q	A
A	R	R	A
A	S	S	A
.	.	.	.

Constraint Propagation

- Given a constraint C and a domain representation D

Constraint Propagation

- Given a constraint C and a domain representation D
  - Compute the intersection $C \cap D$

Constraint Propagation

- Given a constraint C and a domain representation D
  - Compute the intersection $C \cap D$
  - Find the minimal relaxation D' of $C \cap D$
    - ★ Find all solutions of $C \cap D$ (support)
    - ★ Project on the domain representation (filtering)

Constraint Propagation

- Given a constraint C and a domain representation D
  - Compute the intersection $C \cap D$
  - Find the minimal relaxation D' of $C \cap D$
    - ★ Find all solutions of $C \cap D$ (support)
    - ★ Project on the domain representation (filtering)
- Consistency: a domain is consistent iff it is a minimal relaxation

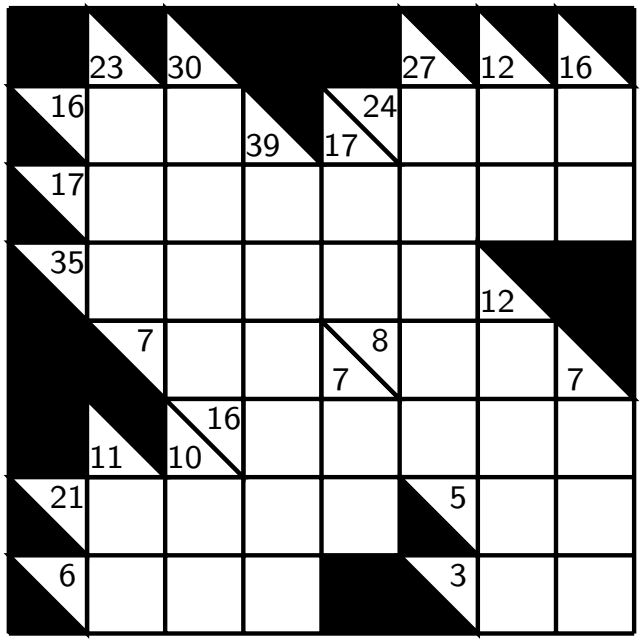
Constraint Propagation

- Given a constraint C and a domain representation D
  - Compute the intersection $C \cap D$
  - Find the minimal relaxation D' of $C \cap D$
    - ★ Find all solutions of $C \cap D$ (support)
    - ★ Project on the domain representation (filtering)
- Consistency: a domain is consistent iff it is a minimal relaxation
  - **Discrete domain: Arc Consistency (ac)**
  - **Bounds: Bounds Consistency (bc)**

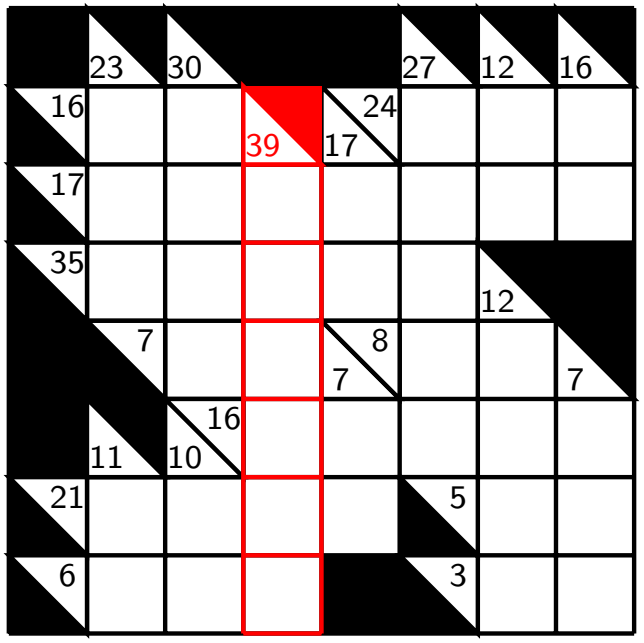
Constraint Propagation

- Given a constraint C and a domain representation D
  - Compute the intersection $C \cap D$
  - Find the minimal relaxation D' of $C \cap D$
    - ★ Find all solutions of $C \cap D$ (support)
    - ★ Project on the domain representation (filtering)
- Consistency: a domain is consistent iff it is a minimal relaxation
  - **Discrete domain: Arc Consistency (ac)**
  - **Bounds: Bounds Consistency (bc)**
  - Also:
    - ★ Multi-valued Decision Diagram MDD Consistency [Hooker, Hadzic, van Hoeve 2007]
    - ★ Set variables [Puget 1992, Gervet 1997], Length-Lex representation [Gervet and Van Hentenryck 2006]
    - ★ Graph variables, Function variables, ...

Example: Kakuro



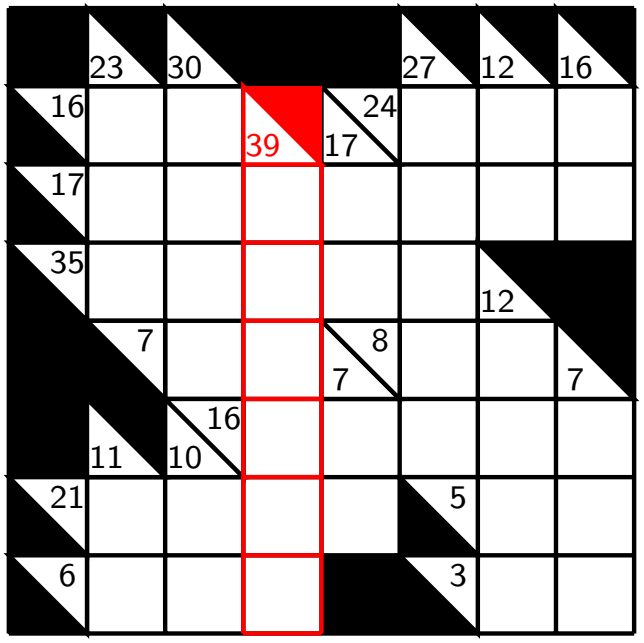
Example: Kakuro



- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	$\{$							8	9$\}$
x_2 :	$\{1$	2			6	7		8	9$\}$
x_3 :	$\{$							8	9$\}$
x_4 :	$\{1$			5	6			8	9$\}$
x_5 :	$\{1$	2			6	7		8	9$\}$
x_6 :	$\{$		4	5				8	9$\}$

Example: Kakuro



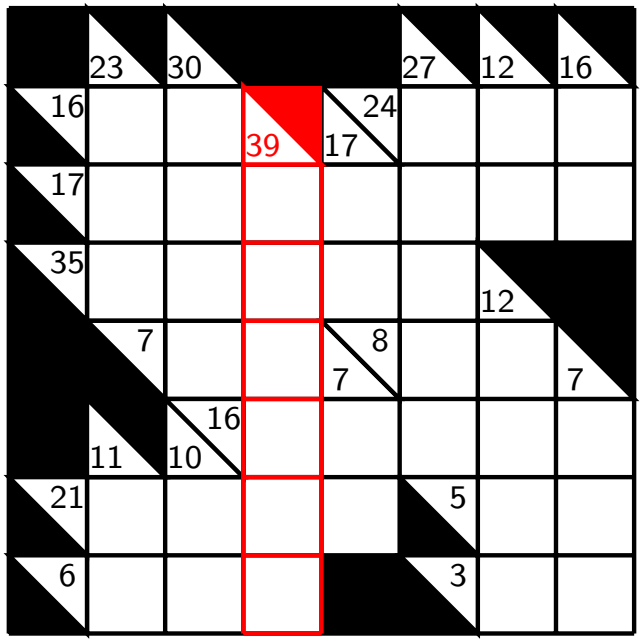
- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	$\{$							8	9$\}$
x_2 :	$\{$1	2			6	7		8	9$\}$
x_3 :	$\{$							8	9$\}$
x_4 :	$\{$1			5	6			8	9$\}$
x_5 :	$\{$1	2			6	7		8	9$\}$
x_6 :	$\{$		4	5				8	9$\}$

$\textsc{AllDifferent}(\{x_1, \ldots, x_6\}) \cap D$

```
888888888888888888888888888888888888999999999999999999999999999999999999
111111122222222222266666667777777777711111112222222222266666667777777777
999999999999999999999999999999999999988888888888888888888888888888888888
555666611115556666111155511115556666555666611115556666111155511115556666
267227766771671177227712722661261122267227766771671177227712722661261122
444454545454444545454544445454444545454444545454544445454545444454544444545
```

Example: Kakuro



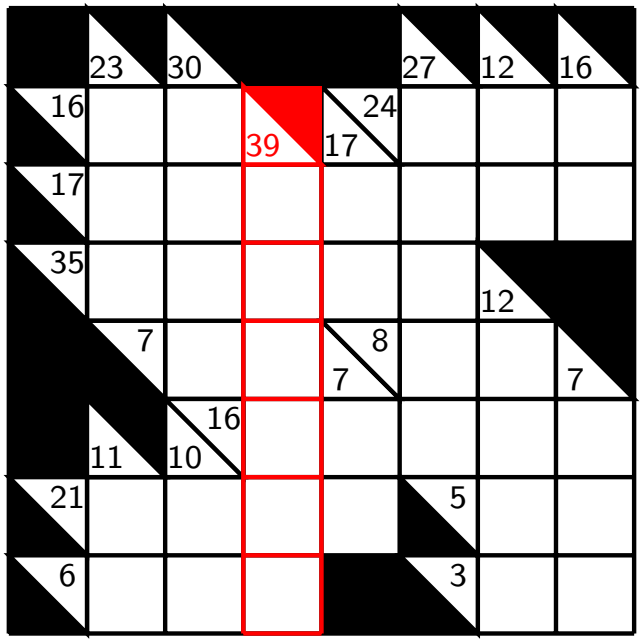
- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	{							8	9}
x_2 :	{1	2			6	7		8	}
x_3 :	{							8	9}
x_4 :	{1			5	6				}
x_5 :	{1	2			6	7			}
x_6 :	{		4	5					}

$\textsc{AllDifferent}(\{x_1, \ldots, x_6\}) \cap D$

```
888888888888888888888888888888888888999999999999999999999999999999999999999     {8, 9}
111111122222222222266666667777777777711111112222222222226666666777777777777 {1, 2, 6, 7}
999999999999999999999999999999999999988888888888888888888888888888888888888     {8, 9}
555666611115556666111155511115556666555666611115556666111155511115556666   {1, 5, 6}
267227766771671177227712722661261122267227766771671177227712722661261122 {1, 2, 6, 7}
444454545454444545454544445454444545444454545454444545454544445454544444545     {4, 5}
```

Example: Kakuro



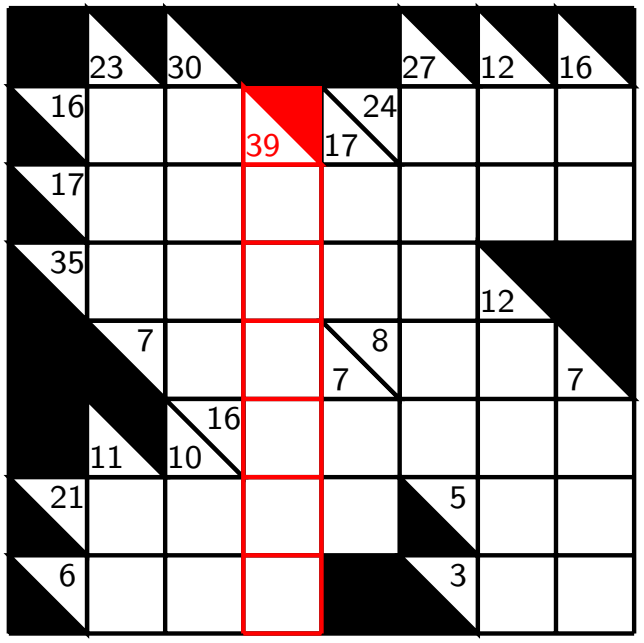
- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	{						8		9}
x_2 :	{1	2			6	7	8		}
x_3 :	{						8		9}
x_4 :	{1			5	6				}
x_5 :	{1	2			6	7			}
x_6 :	{		4	5					}

$\sum_{i=1}^{6} x_i = 39 \cap D$

8	8	8	8	8	8	8	8	8	8	9	9	9	9	9
6	6	6	6	6	6	7	7	7	7	6	6	6	6	7
8	8	8	9	9	9	8	8	8	9	8	8	8	9	8
5	6	6	5	5	6	5	5	6	5	5	5	6	5	5
7	6	7	6	7	6	6	7	6	6	6	7	6	6	6
5	5	4	5	4	4	5	4	4	4	5	4	4	4	4

Example: Kakuro



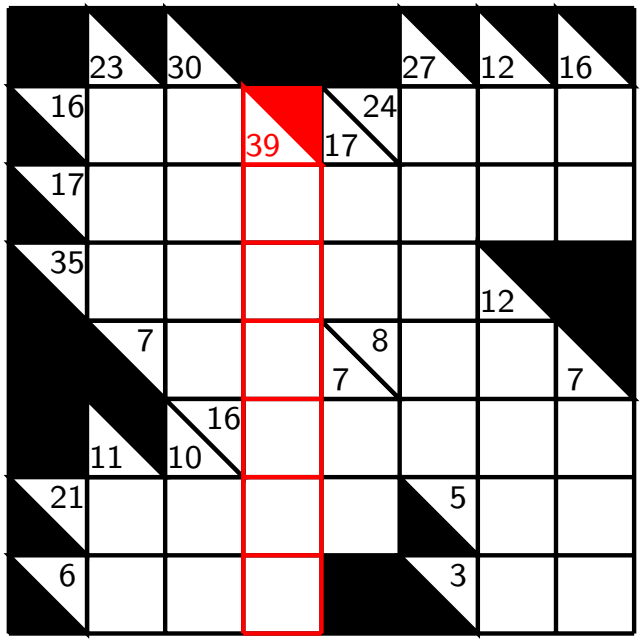
- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	{						8		9}
x_2 :	{				6	7	8		}
x_3 :	{						8		9}
x_4 :	{			5	6				}
x_5 :	{				6	7			}
x_6 :	{		4	5					}

$\sum_{i=1}^{6} x_i = 39 \cap D$

8	8	8	8	8	8	8	8	8	8	9	9	9	9	9	$\{8, 9\}$
6	6	6	6	6	6	7	7	7	7	6	6	6	6	7	$\{6, 7\}$
8	8	8	9	9	9	8	8	8	9	8	8	8	9	8	$\{8, 9\}$
5	6	6	5	5	6	5	5	6	5	5	5	6	5	5	$\{5, 6\}$
7	6	7	6	7	6	6	7	6	6	6	7	6	6	6	$\{6, 7\}$
5	5	4	5	4	4	5	4	4	4	5	4	4	4	4	$\{4, 5\}$

Example: Kakuro



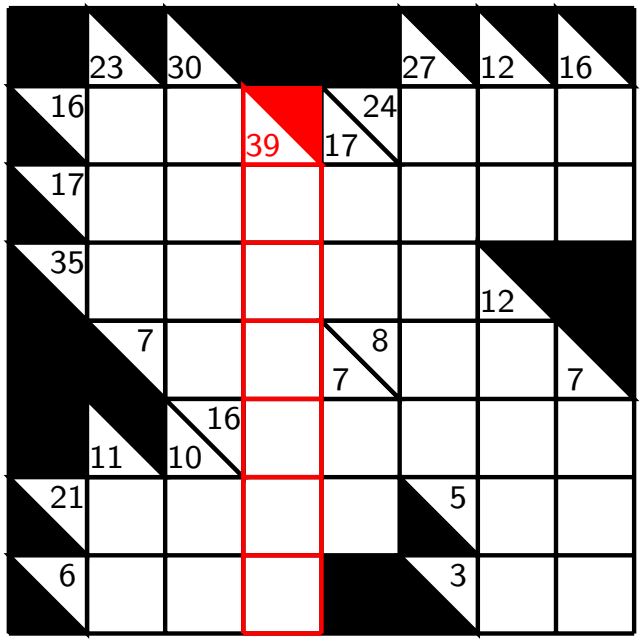
- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	{						8		9}
x_2 :	{				6	7	8		}
x_3 :	{						8		9}
x_4 :	{			5	6				}
x_5 :	{				6	7			}
x_6 :	{		4	5					}

$\textsc{AllDifferent}(\{x_1, \ldots, x_6\}) \cap D$

8	8	9	9	$\{8, 9\}$
6	7	6	7	$\{6, 7\}$
9	9	8	8	$\{8, 9\}$
5	5	5	5	$\{5\}$
7	6	7	6	$\{6, 7\}$
4	4	4	4	$\{4\}$

Example: Kakuro



- $\sum_{i=1}^{6} x_i = 39$
- $\textsc{AllDifferent}(\{x_1, \ldots, x_6\})$

x_1 :	{							8	9}
x_2 :	{				6	7		8	}
x_3 :	{							8	9}
x_4 :	{			5					}
x_5 :	{				6	7			}
x_6 :	{		4						}

$\textsc{AllDifferent}(\{x_1, \ldots, x_6\}) \cap D$

8	8	9	9	$\{8, 9\}$
6	7	6	7	$\{6, 7\}$
9	9	8	8	$\{8, 9\}$
5	5	5	5	$\{5\}$
7	6	7	6	$\{6, 7\}$
4	4	4	4	$\{4\}$

Constraint Propagation

- Of course we do not want to enumerate solutions!

Constraint Propagation

- Of course we do not want to enumerate solutions!
  - Propagating is not necessarily harder than solving
    - ★ AC on AllDifferent in $O(n^{1.5}d)$, same as Matching

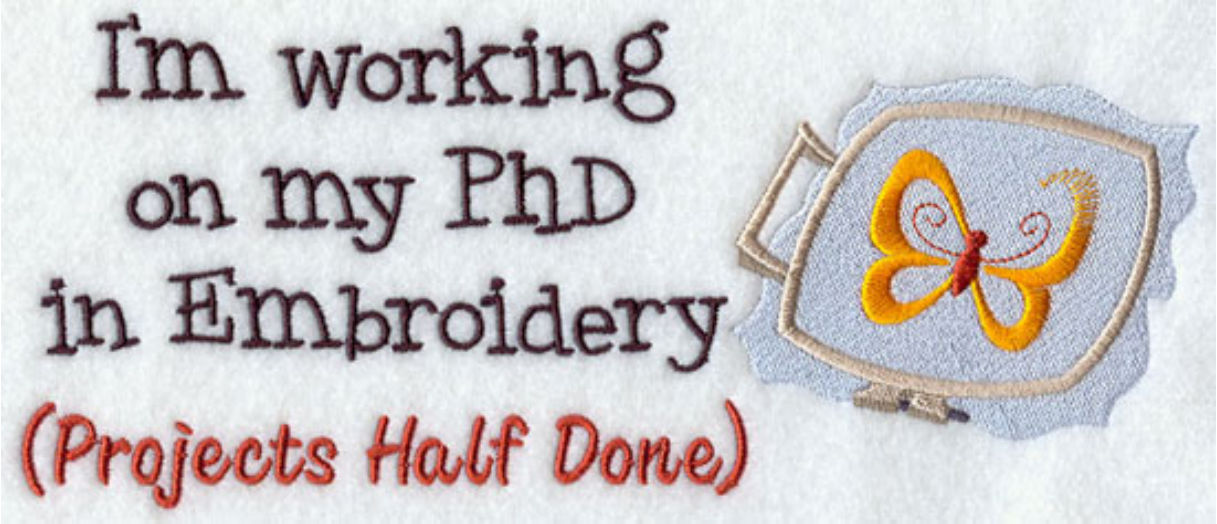
Constraint Propagation

- Of course we do not want to enumerate solutions!
  - Propagating is not necessarily harder than solving
    - ★ AC on AllDifferent in $O(n^{1.5}d)$, same as Matching
  - In general:
    - ★ Solving in $O(K)$, then propagating in $O(ndK)$
    - ★ Propagating in $O(K)$, then solving in $O(nK)$

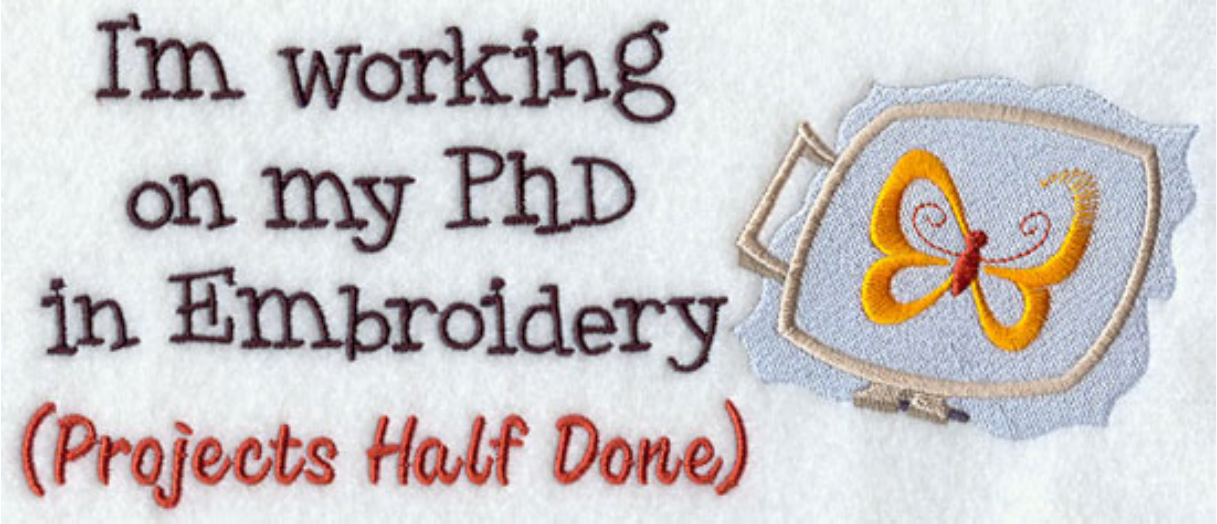
Constraint Propagation

- Of course we do not want to enumerate solutions!
  - Propagating is not necessarily harder than solving
    - ★ AC on AllDifferent in $O(n^{1.5}d)$, same as Matching
  - In general:
    - ★ Solving in $O(K)$, then propagating in $O(ndK)$
    - ★ Propagating in $O(K)$, then solving in $O(nK)$
  - In practice:
    - ★ Same complexity class, but different algorithms
    - ★ Propagating is harder

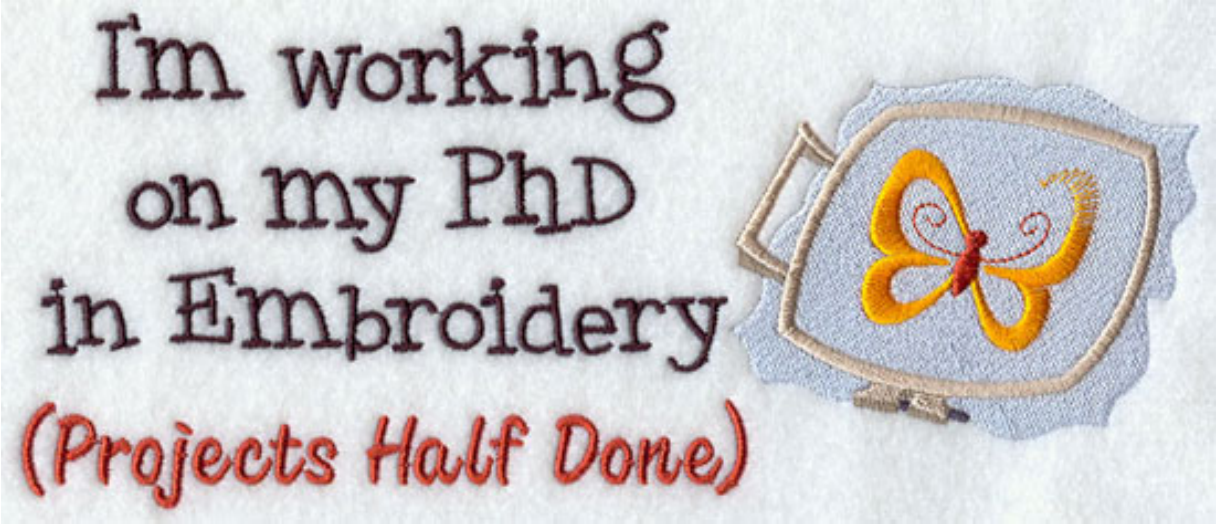
An Example of Propagator: the SWITCH Constraint



- A set of n garments to embroid
  - Each garment require a subset of m available colors
- A machine with k reels of thread

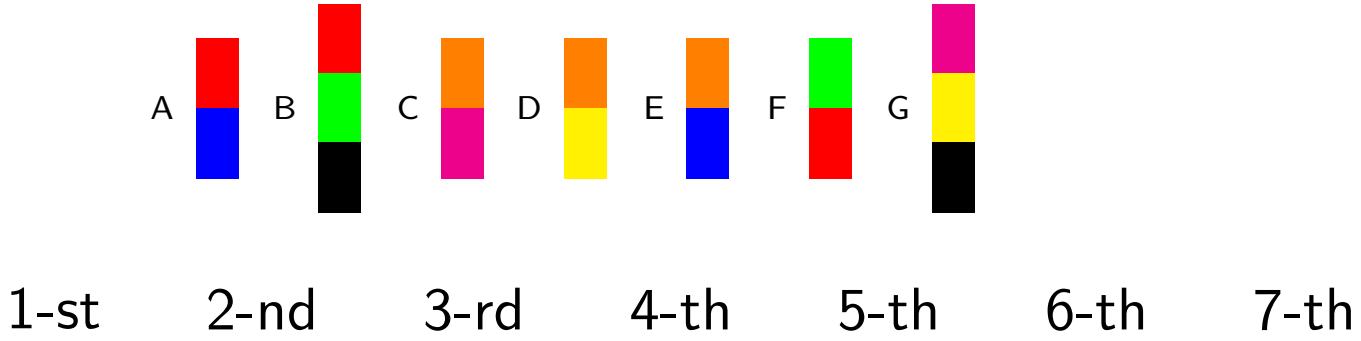


- A set of n garments to embroid
  - Each garment require a subset of m available colors
- A machine with k reels of thread
- Replacing a reel costs time



- A set of n garments to embroid
  - Each garment require a subset of m available colors
- A machine with k reels of thread
- Replacing a reel costs time
- What is the best sequence of garments?

Embroidery Scheduling

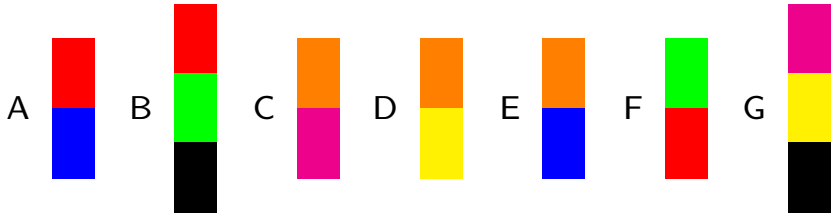


Reel 1

Reel 2

Reel 3

Embroidery Scheduling



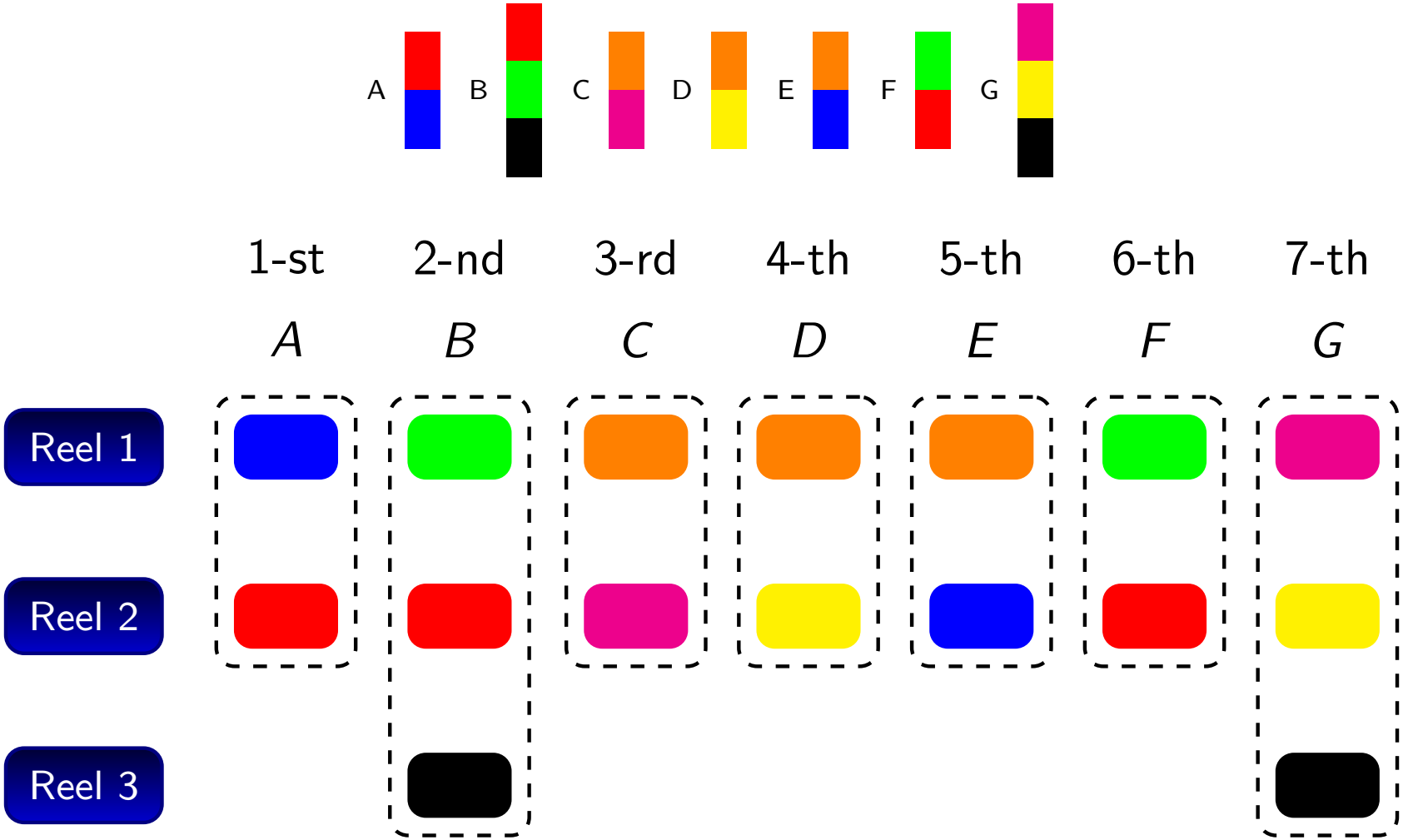
1-st	2-nd	3-rd	4-th	5-th	6-th	7-th
A	*B*	*C*	*D*	*E*	*F*	*G*

Reel 1

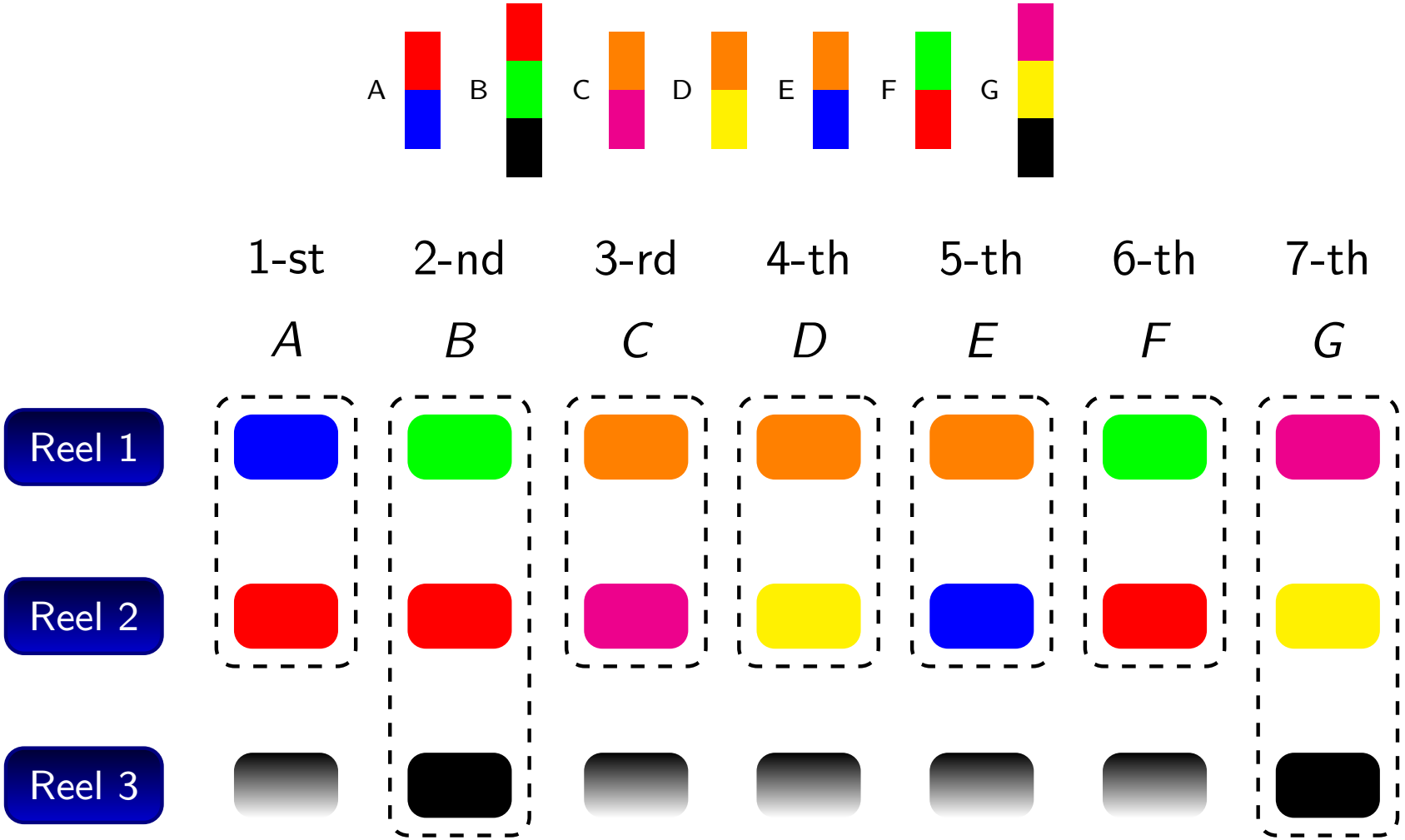
Reel 2

Reel 3

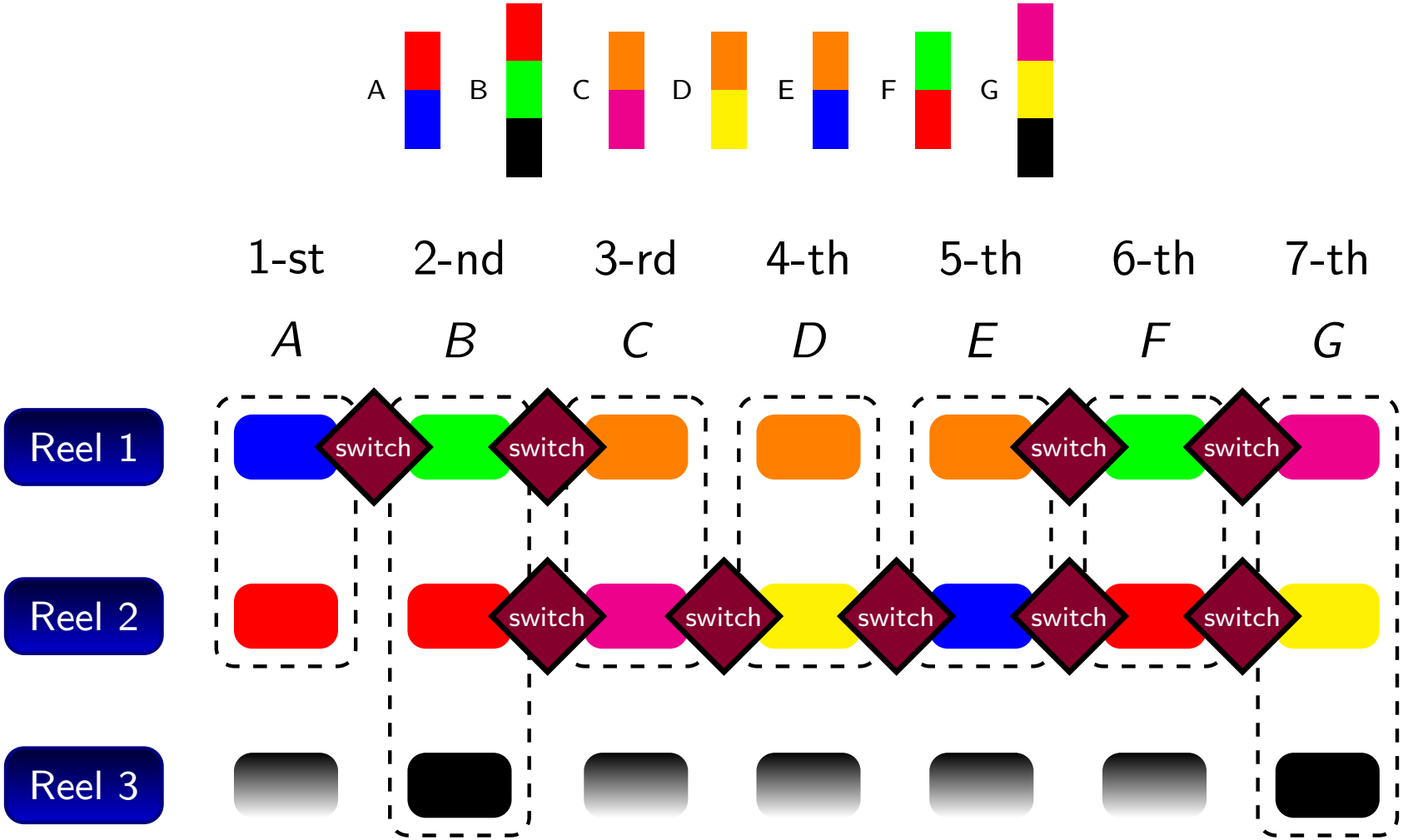
Embroidery Scheduling



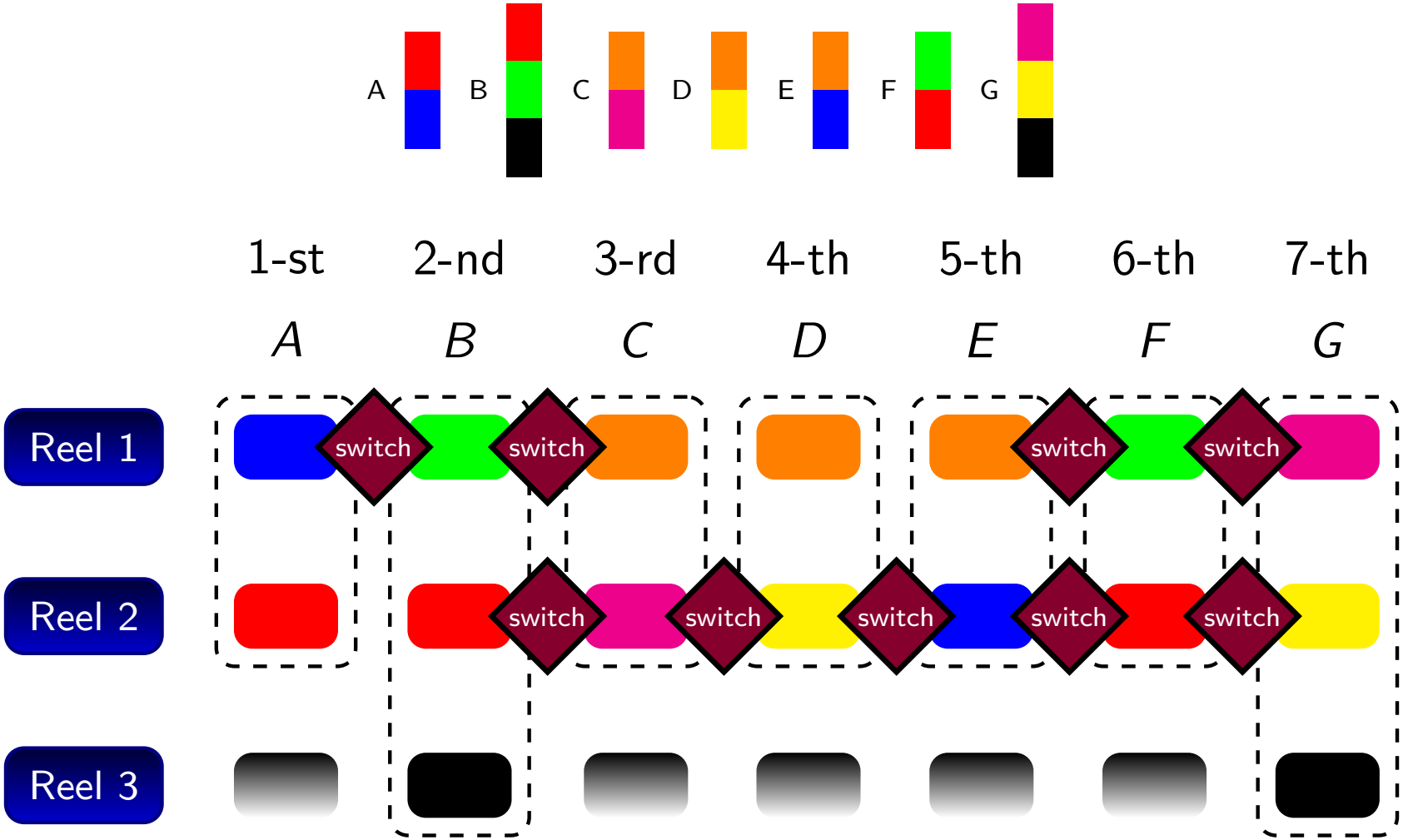
Embroidery Scheduling



Embroidery Scheduling

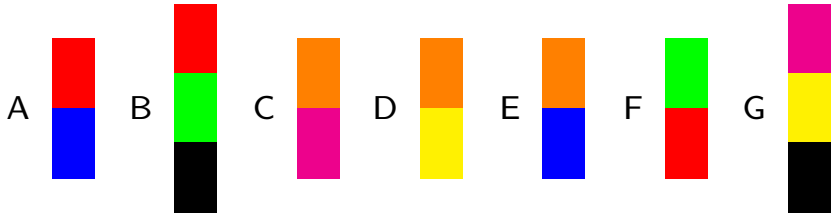


Embroidery Scheduling



- A,B,C,D,E,F,G: 9 switches

Embroidery Scheduling



1-st	2-nd	3-rd	4-th	5-th	6-th	7-th
A	*B*	*C*	*D*	*E*	*F*	*G*

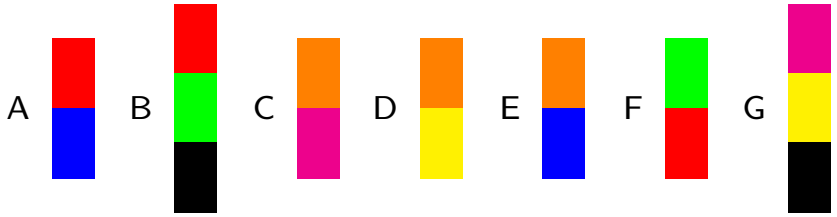
Reel 1

Reel 2

Reel 3

- A,B,C,D,E,F,G: 9 switches

Embroidery Scheduling



1-st	2-nd	3-rd	4-th	5-th	6-th	7-th
B	*F*	*A*	*E*	*D*	*C*	*G*

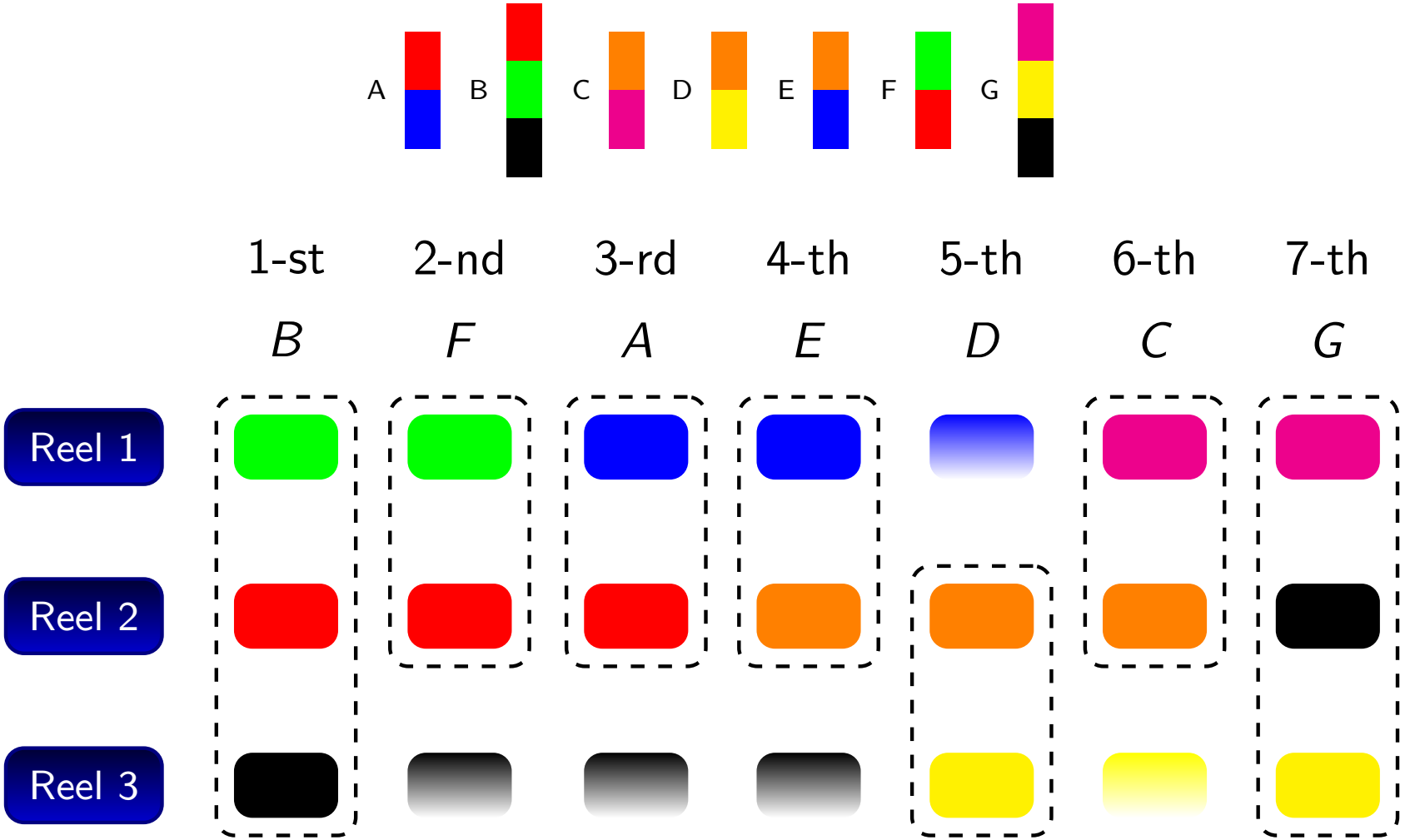
Reel 1

Reel 2

Reel 3

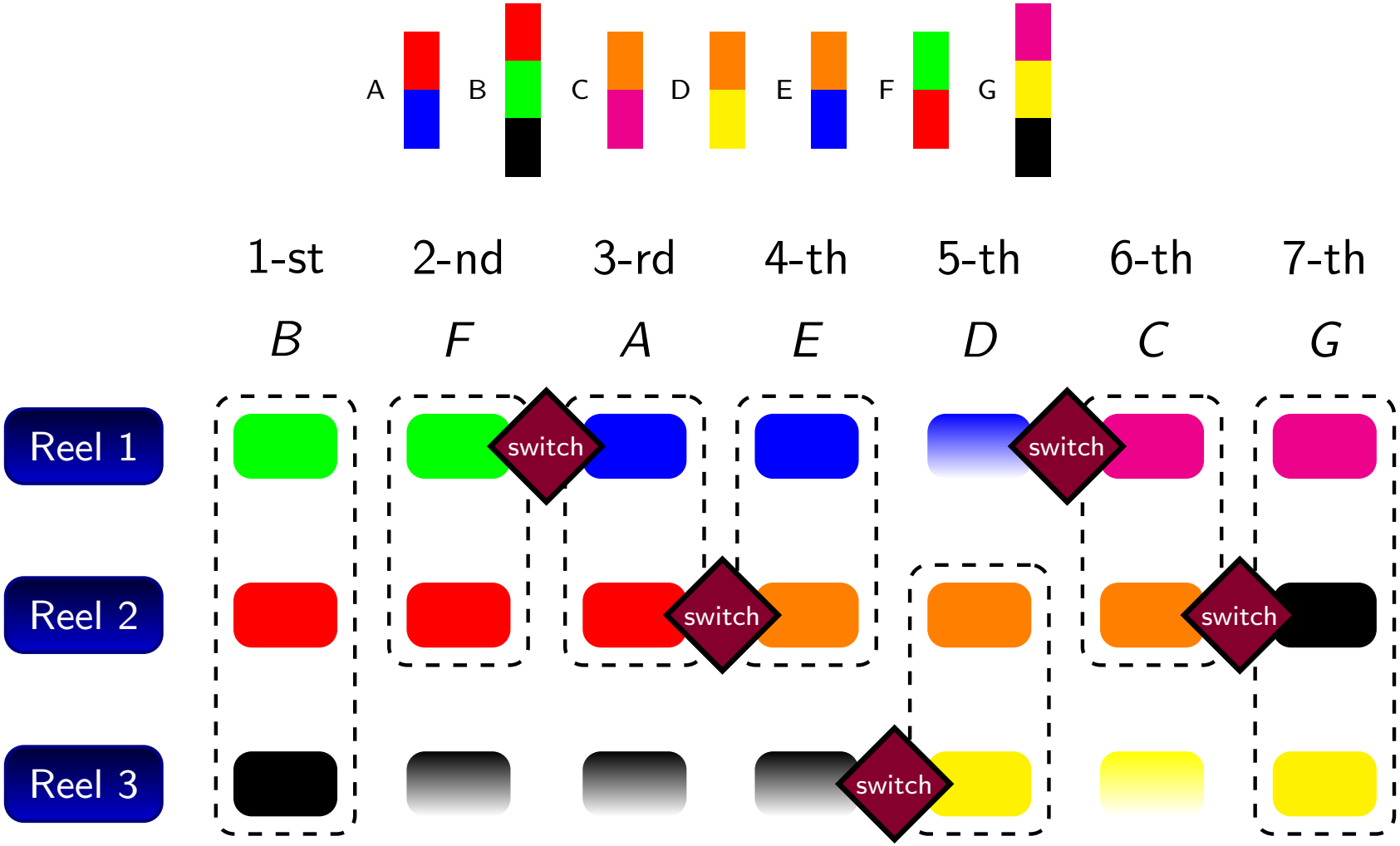
- A,B,C,D,E,F,G: 9 switches

Embroidery Scheduling



- A,B,C,D,E,F,G: 9 switches

Embroidery Scheduling



- A,B,C,D,E,F,G: 9 switches
- B,F,A,E,D,C,G: 5 switches

The Switch Constraint

The Switch Constraint

- One Boolean variable per color j and position i
  - $x_i^j = 1$ iff the j-th color is on the buffer at time i

The Switch Constraint

- One Boolean variable per color j and position i
  - $x_i^j = 1$ iff the j-th color is on the buffer at time i
- A variable M equal to the number of changes $x_i^j < x_{i+1}^j$

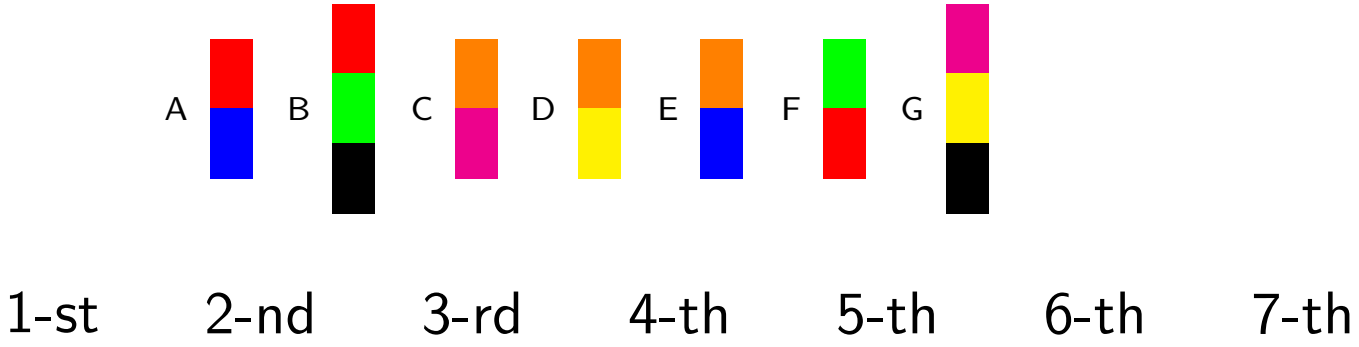
The Switch Constraint

- One Boolean variable per color j and position i
  - $x_i^j = 1$ iff the j-th color is on the buffer at time i
- A variable M equal to the number of changes $x_i^j < x_{i+1}^j$
- Not only useful for embroidery
  - Instruction scheduling (compilation)
  - Test sequencing
  - File transfer (observation satellites)

The Switch Constraint

- One Boolean variable per color j and position i
  - $x_i^j = 1$ iff the j-th color is on the buffer at time i
- A variable M equal to the number of changes $x_i^j < x_{i+1}^j$
- Not only useful for embroidery
  - Instruction scheduling (compilation)
  - Test sequencing
  - File transfer (observation satellites)
- Switch turned up in 3/4 of the industrial projects I was involved in!

Propagating the SWITCH Constraint

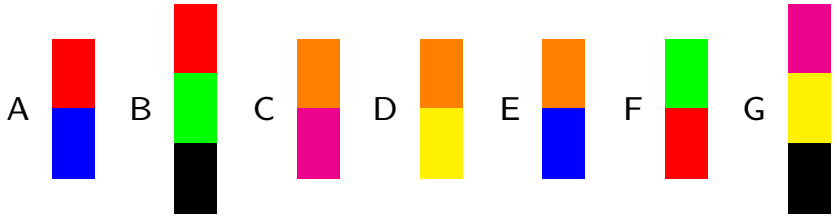


Reel 1

Reel 2

Reel 3

Propagating the SWITCH Constraint



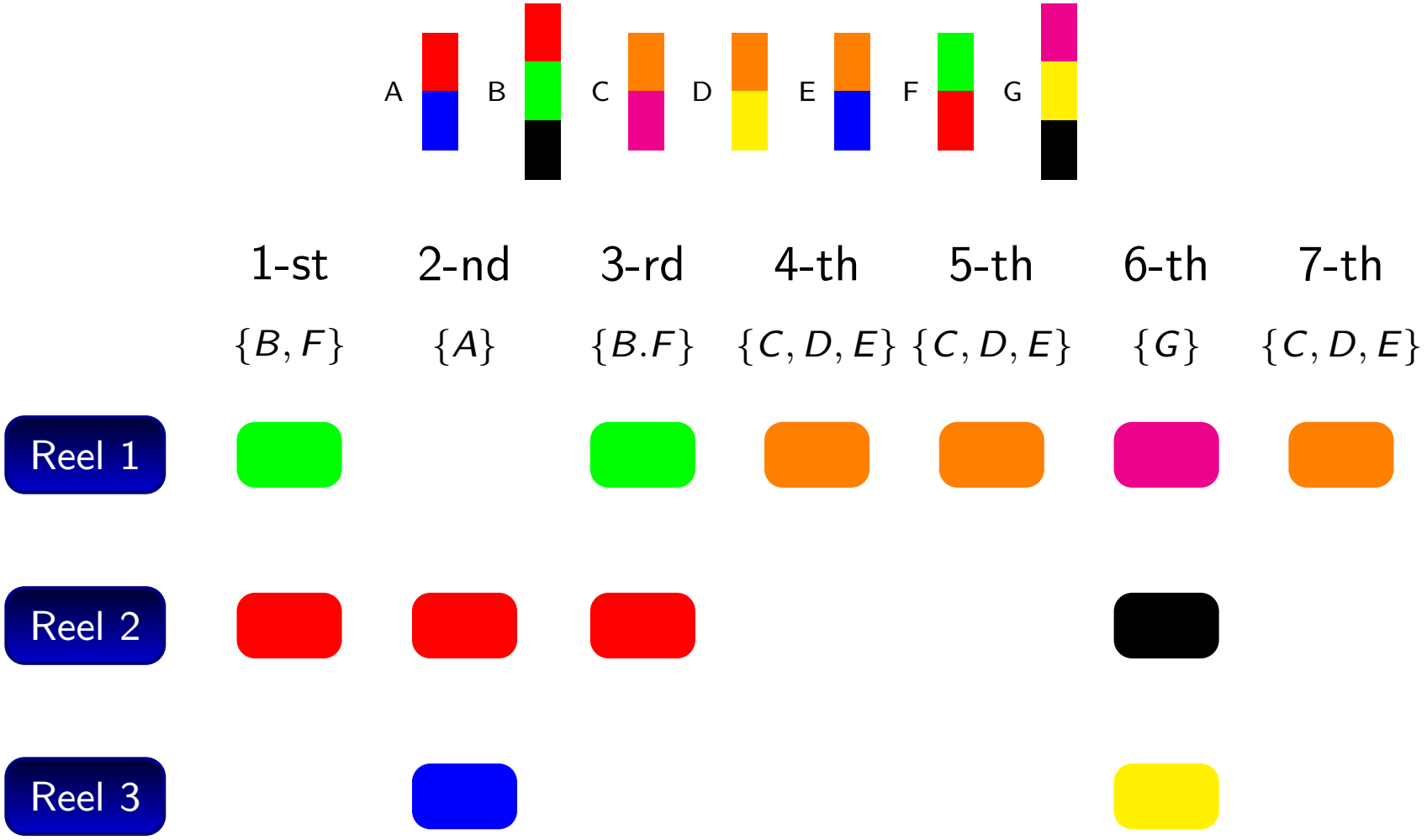
1-st	2-nd	3-rd	4-th	5-th	6-th	7-th
$\{B, F\}$	$\{A\}$	$\{B.F\}$	$\{C, D, E\}$	$\{C, D, E\}$	$\{G\}$	$\{C, D, E\}$

Reel 1

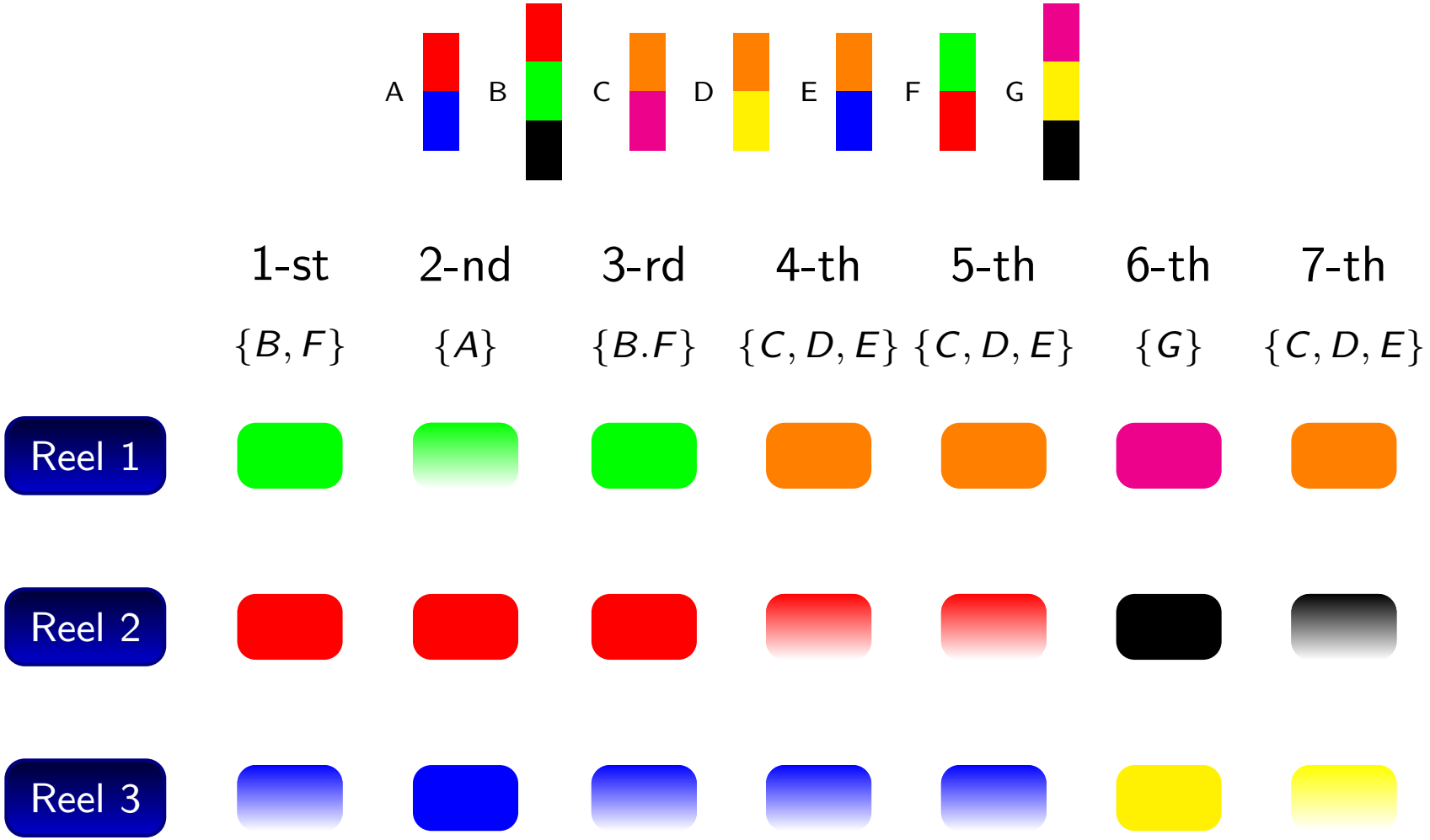
Reel 2

Reel 3

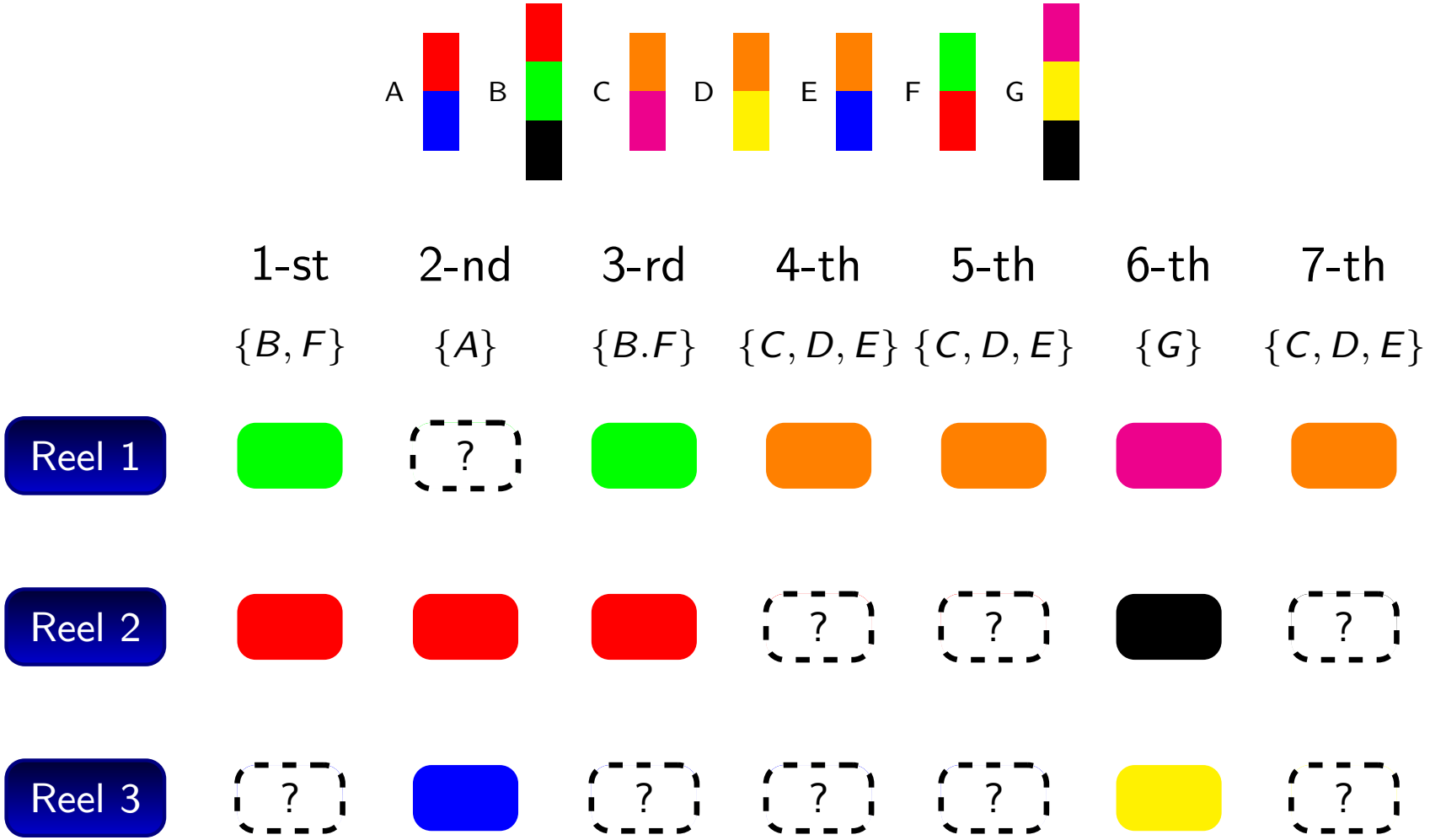
Propagating the SWITCH Constraint



Propagating the SWITCH Constraint



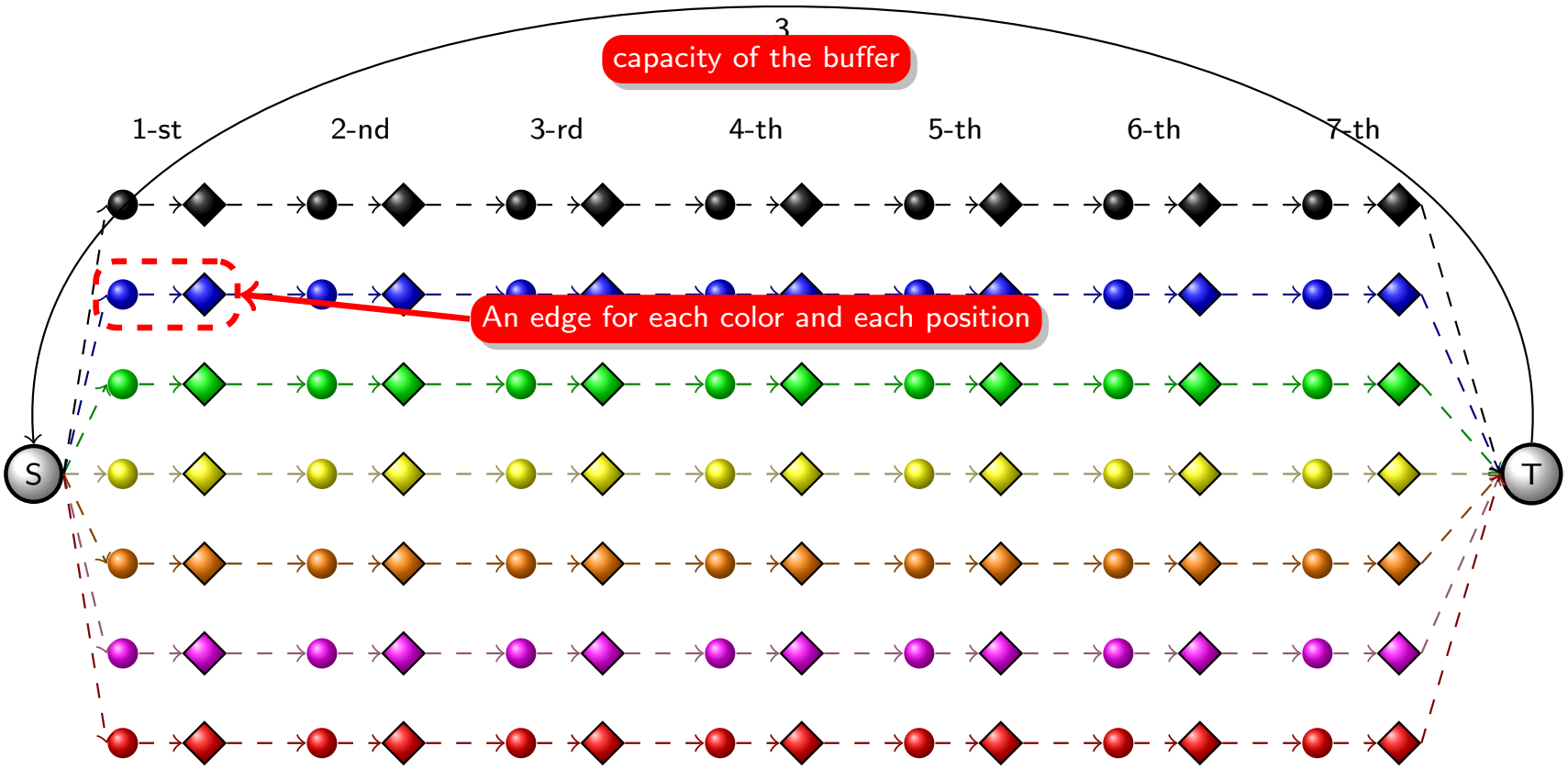
Propagating the SWITCH Constraint



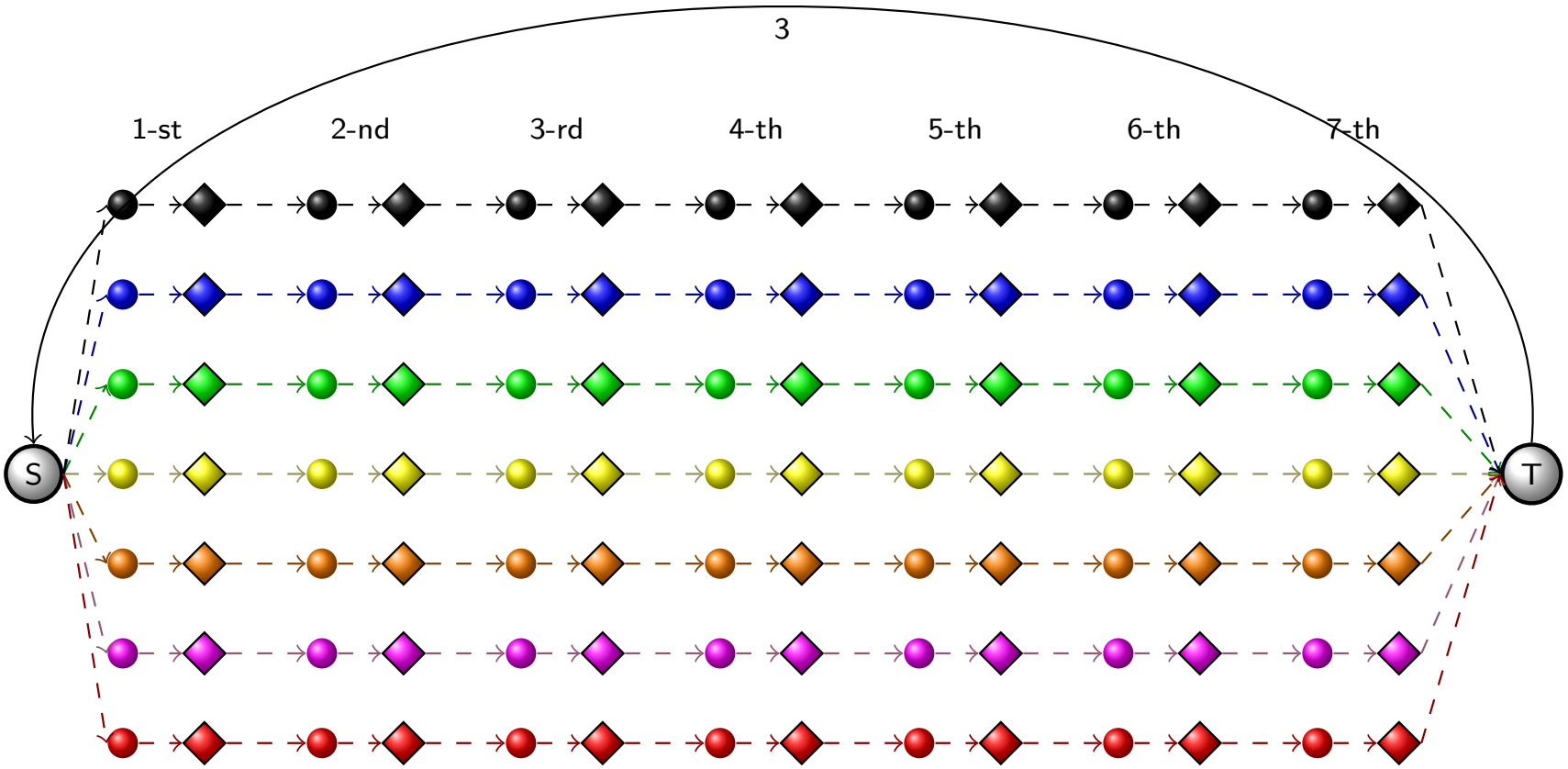
- How many switches at least? (compute a lower bound for M)
- What are the possible colors (given the upper bound on M)

Propagating the Switch Constraint

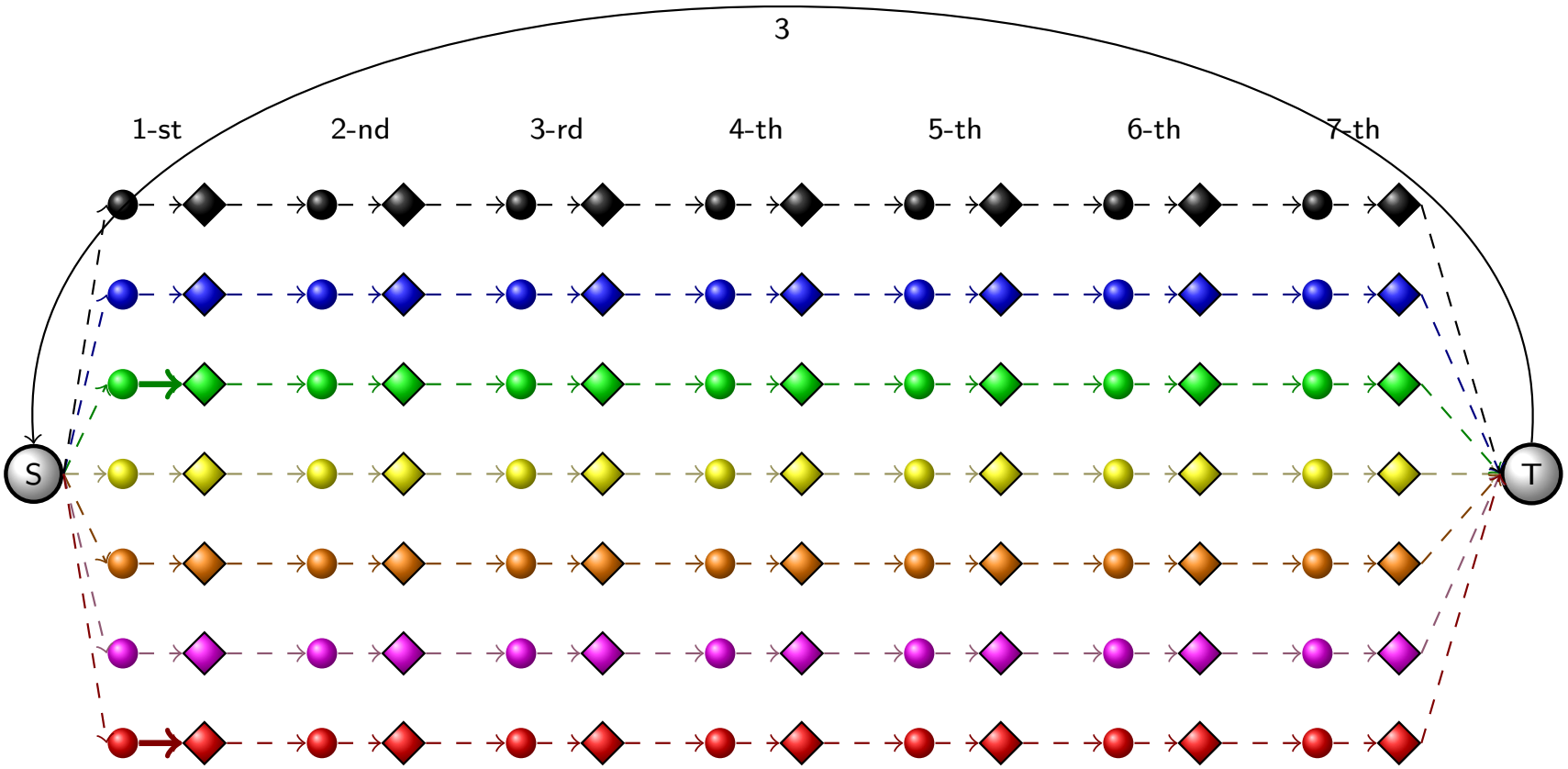
The Switch Constraint



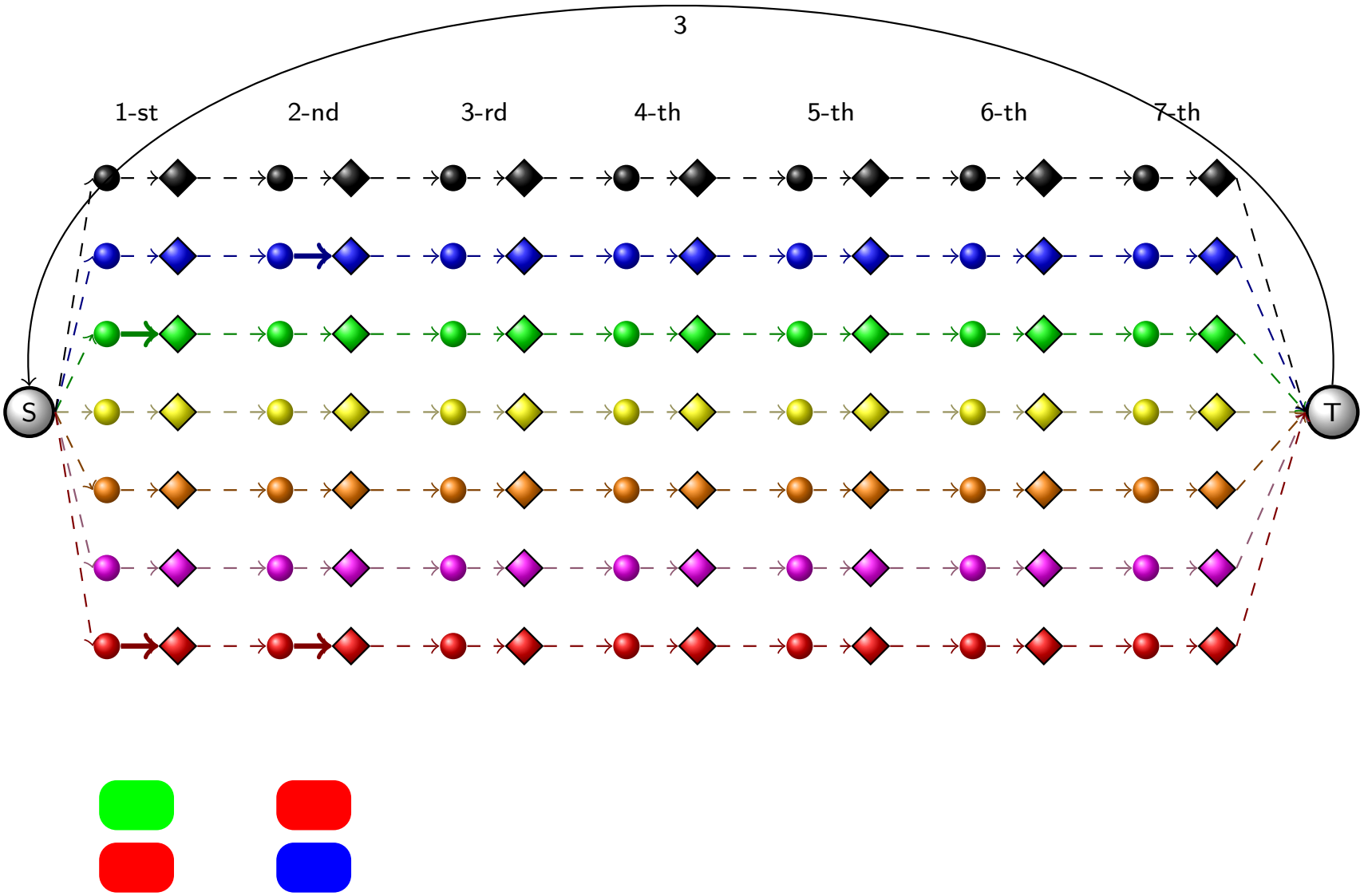
The Switch Constraint



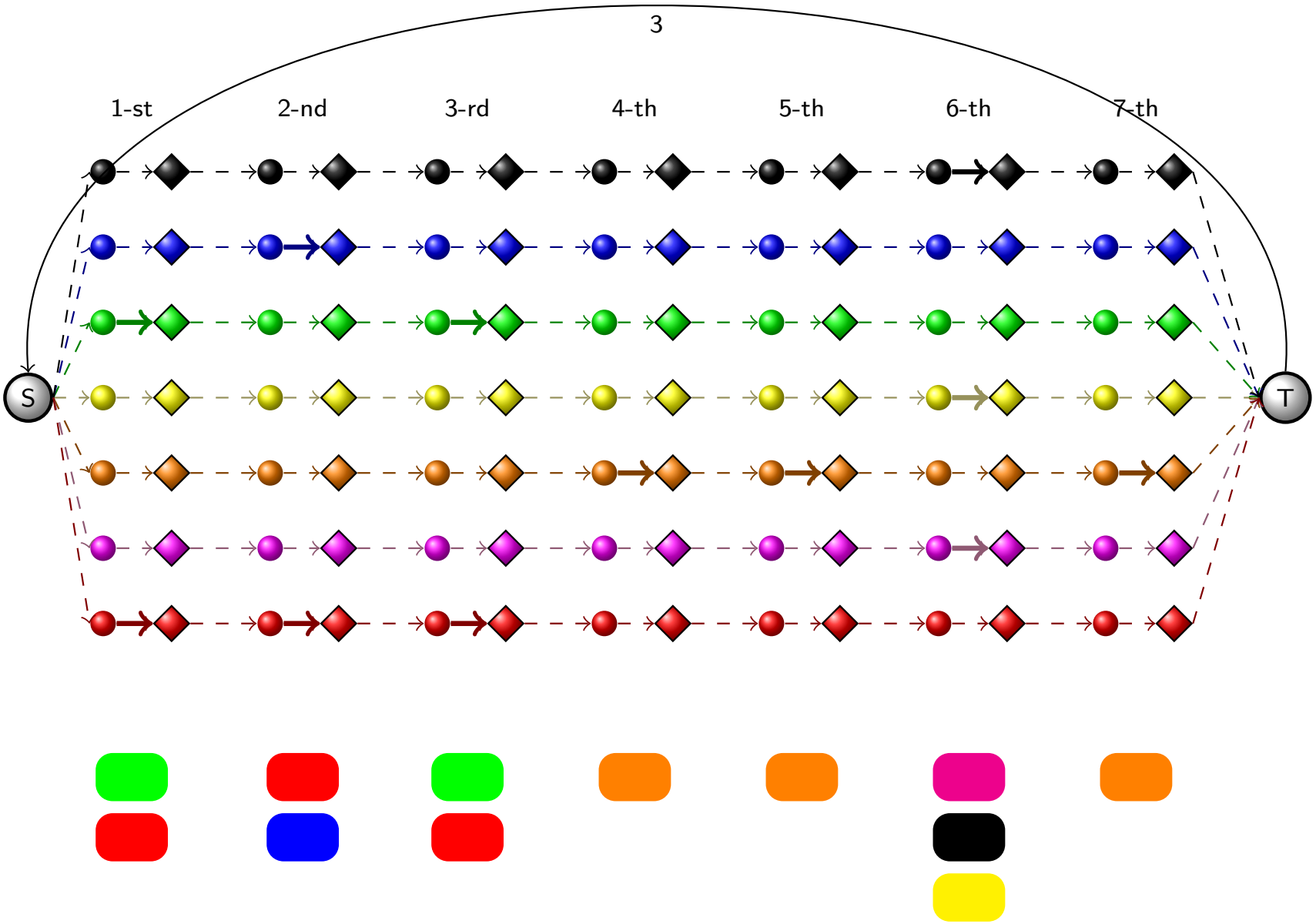
The Switch Constraint



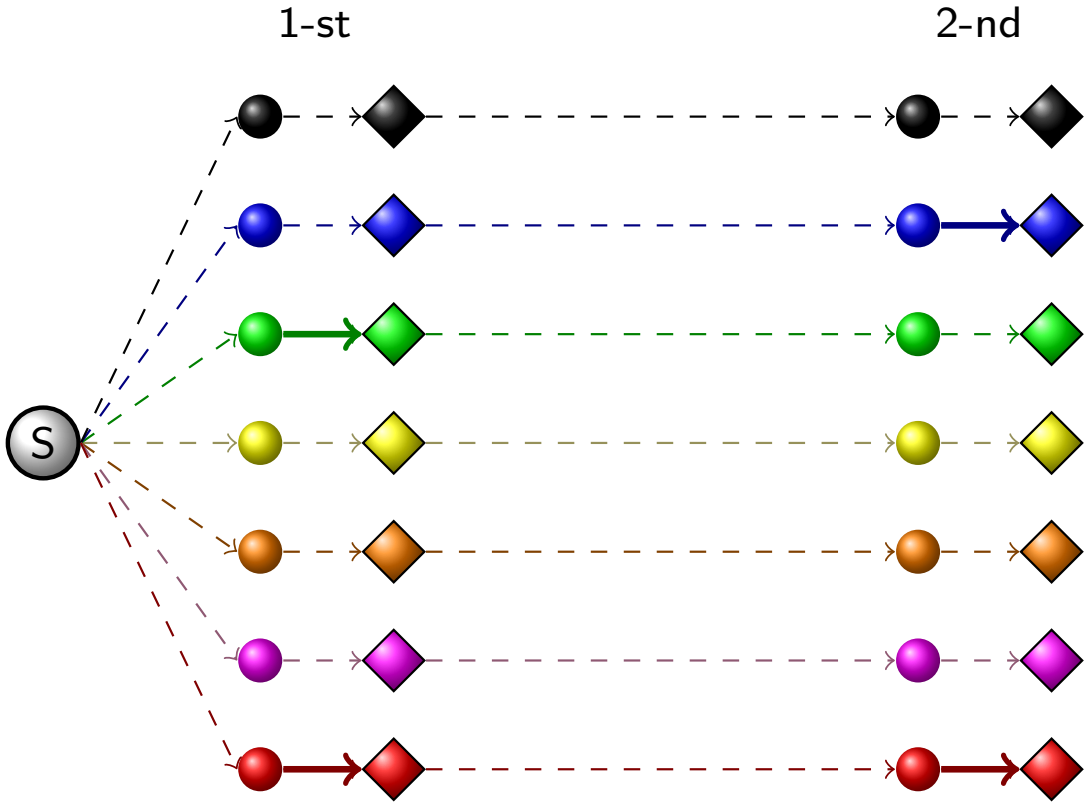
The Switch Constraint



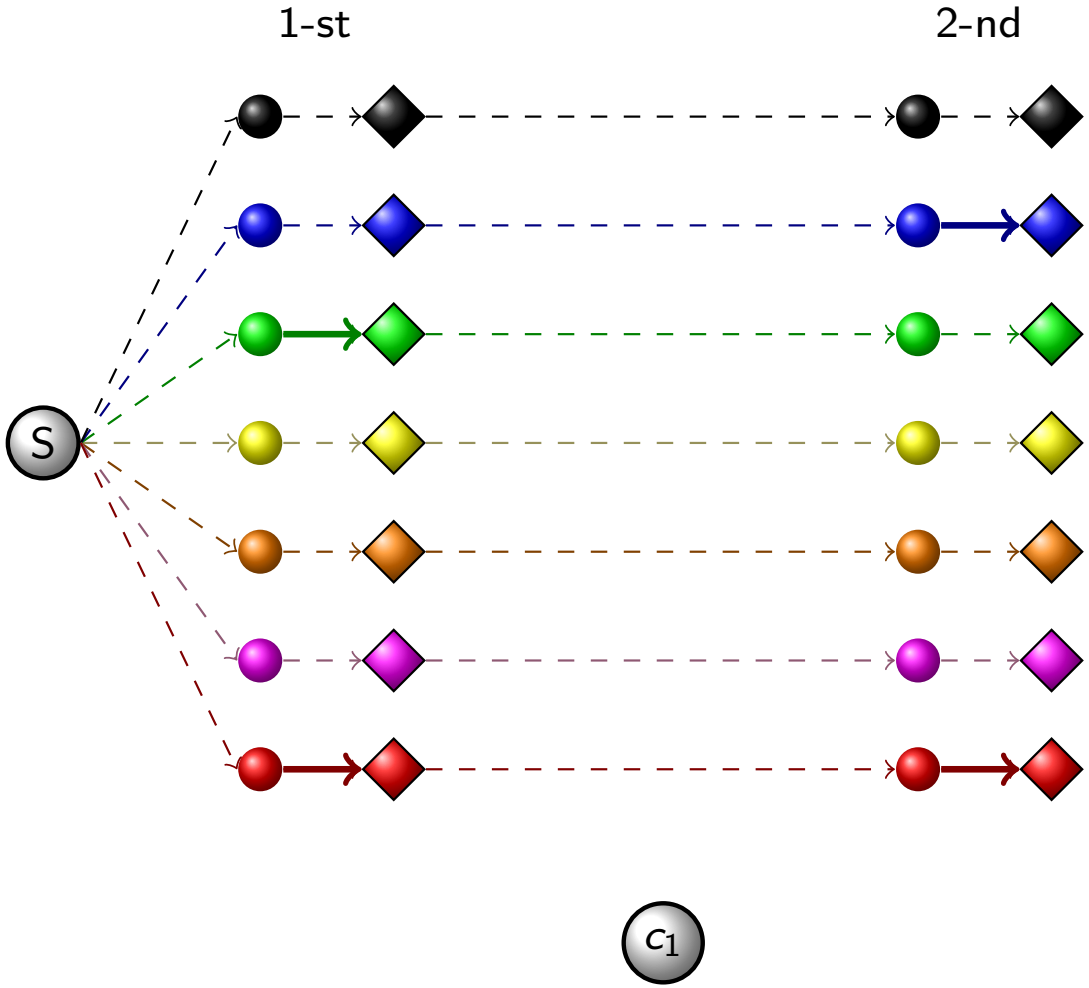
The Switch Constraint



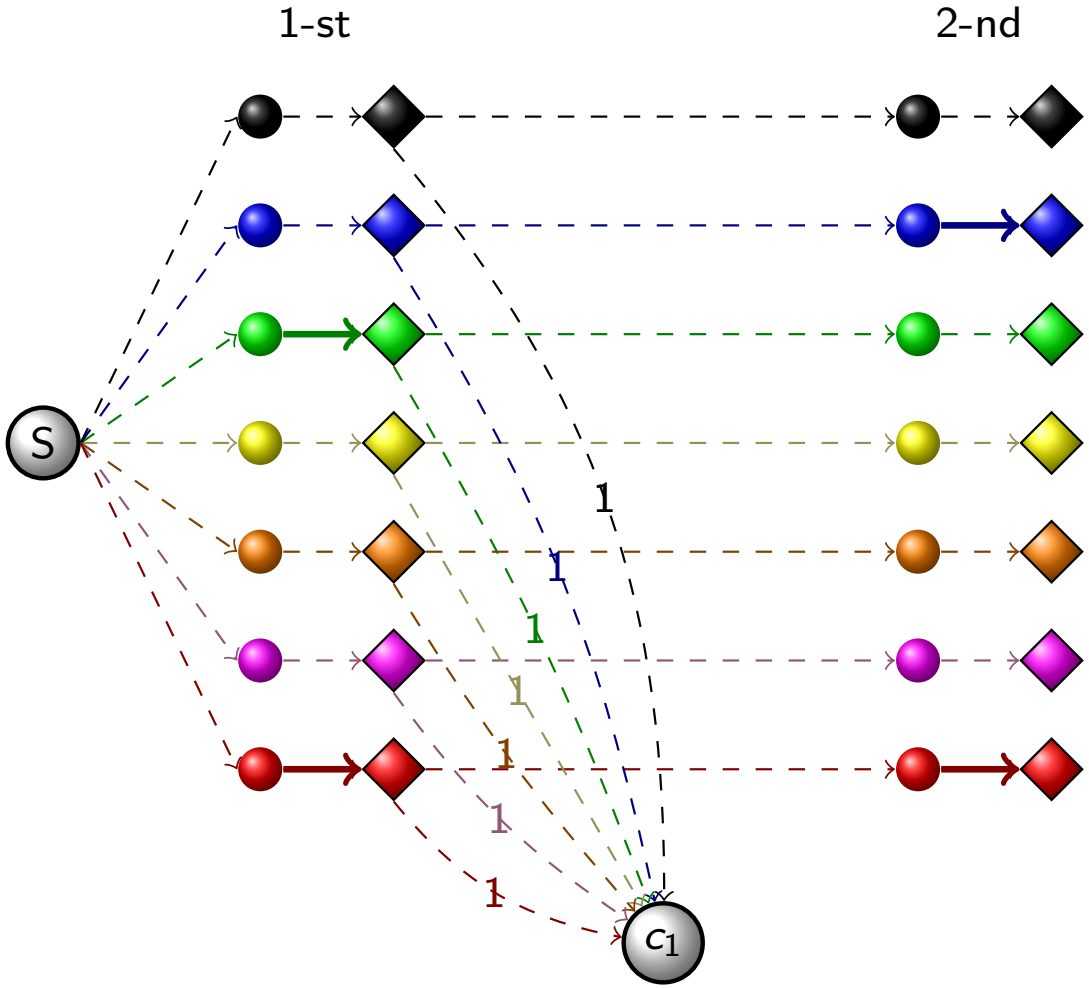
The Switch Constraint



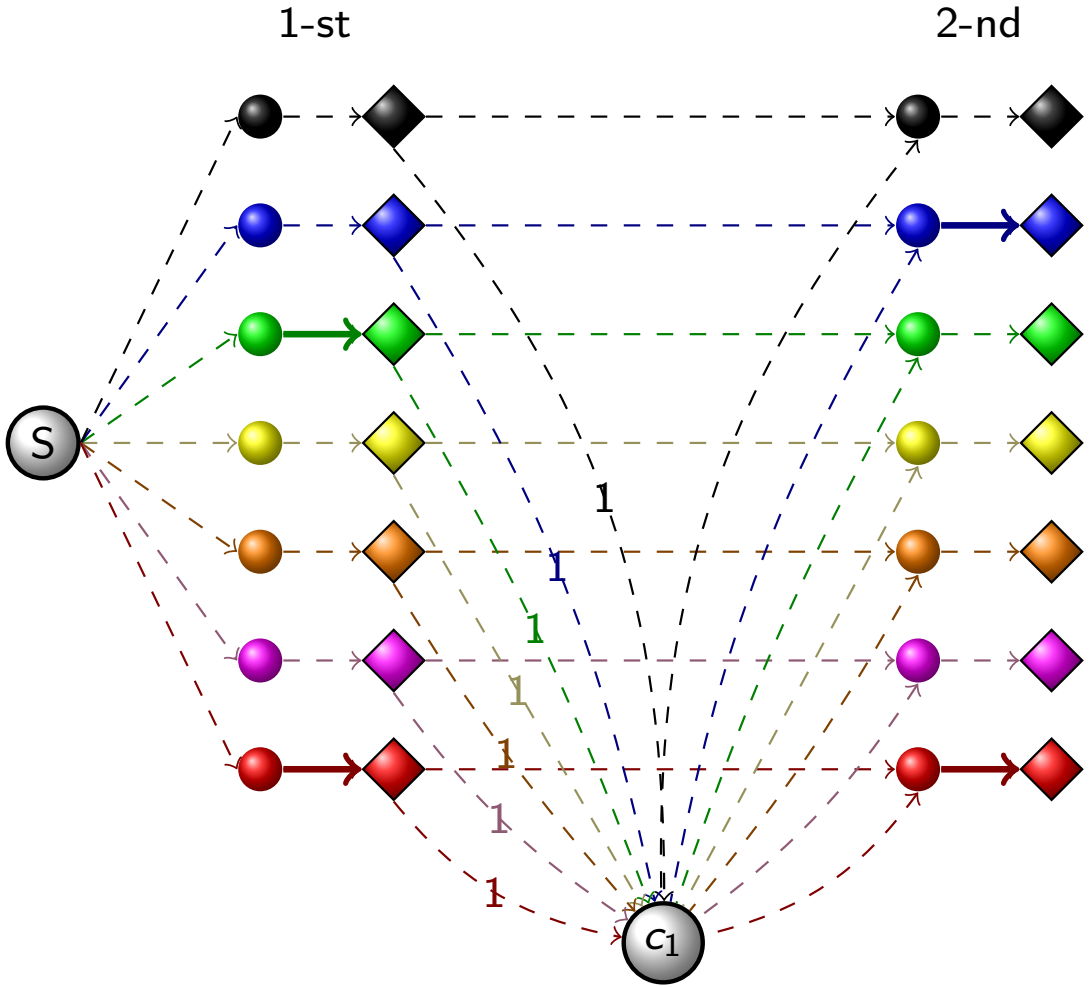
The Switch Constraint



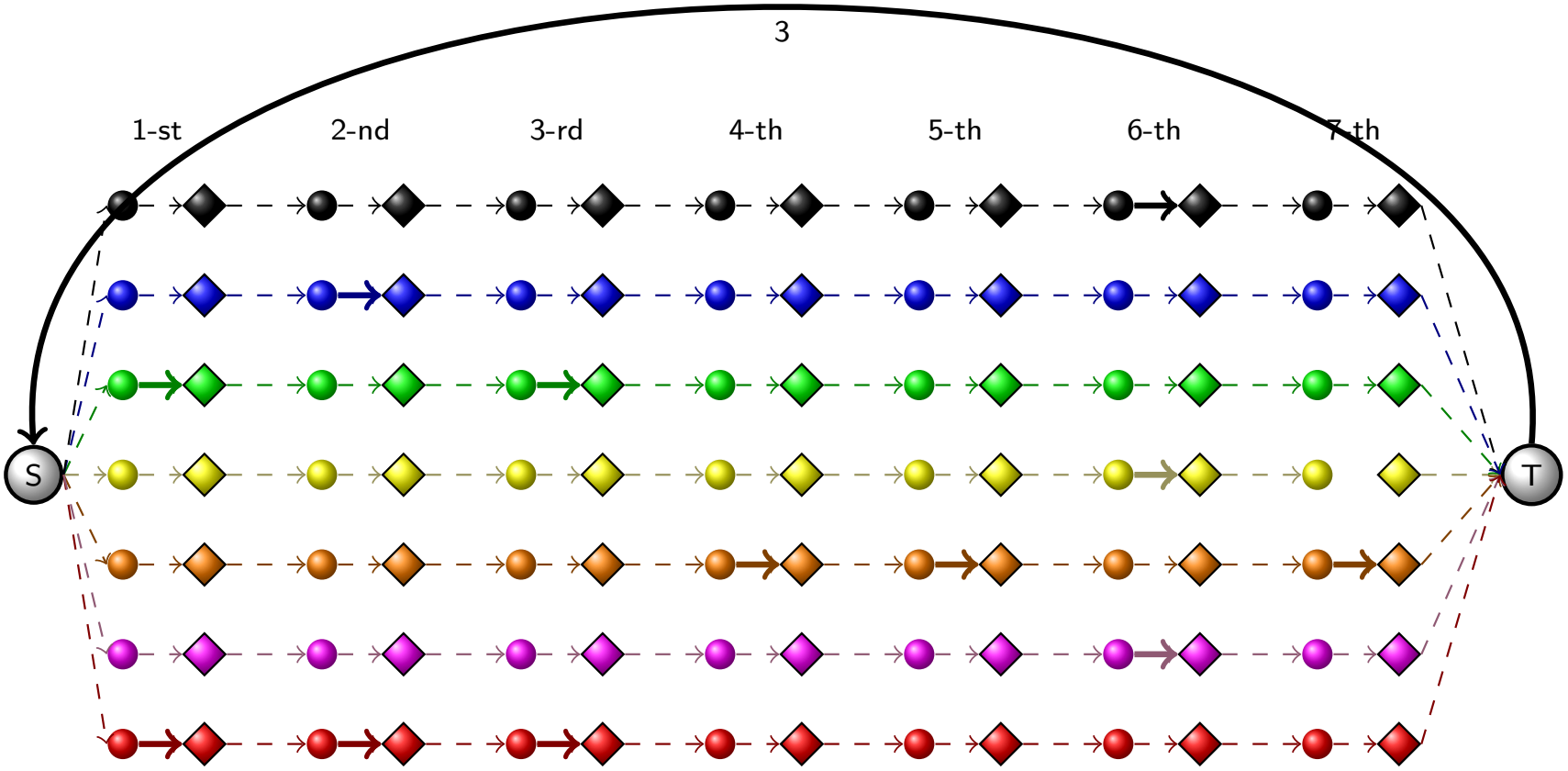
The Switch Constraint



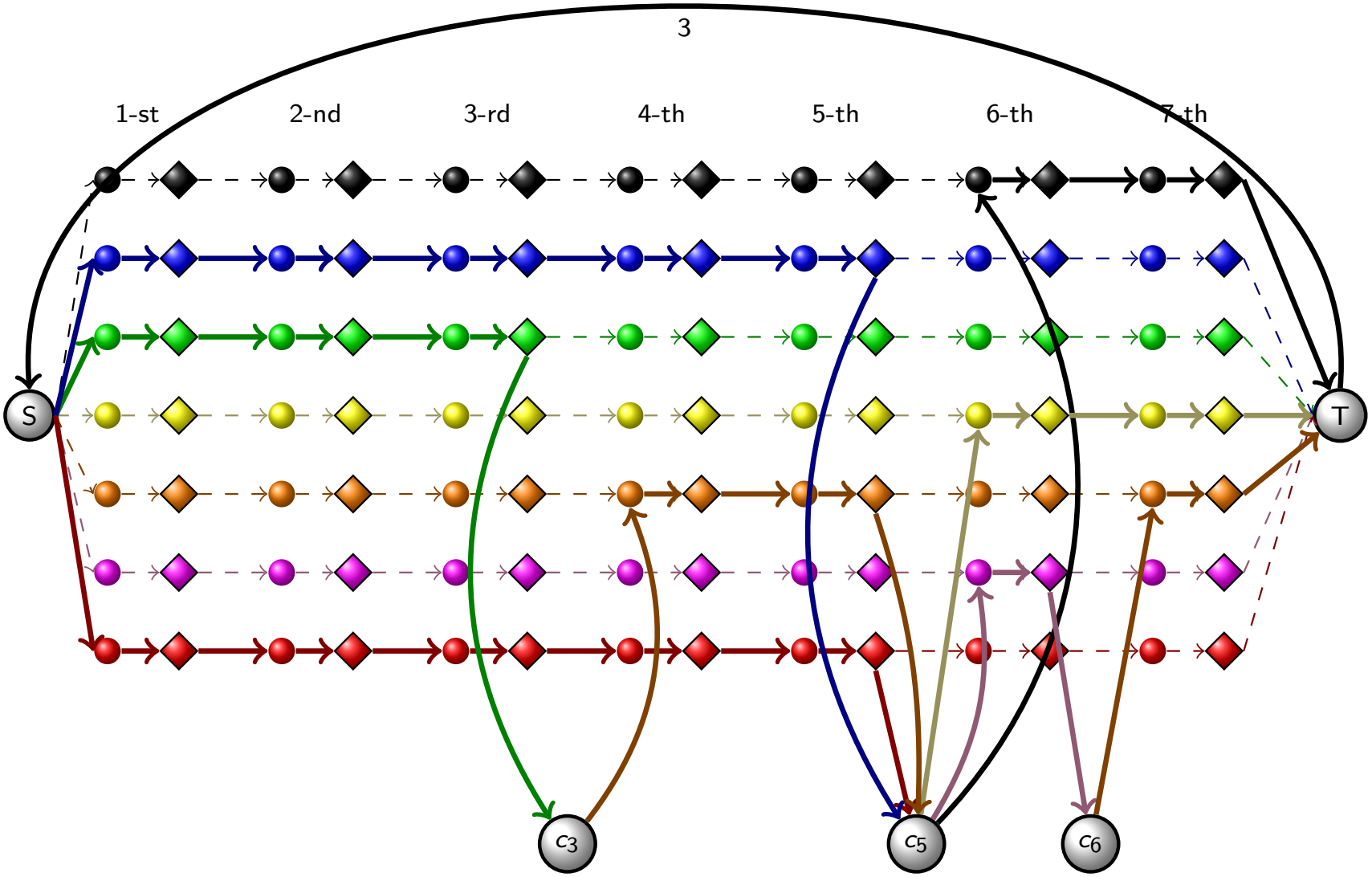
The Switch Constraint



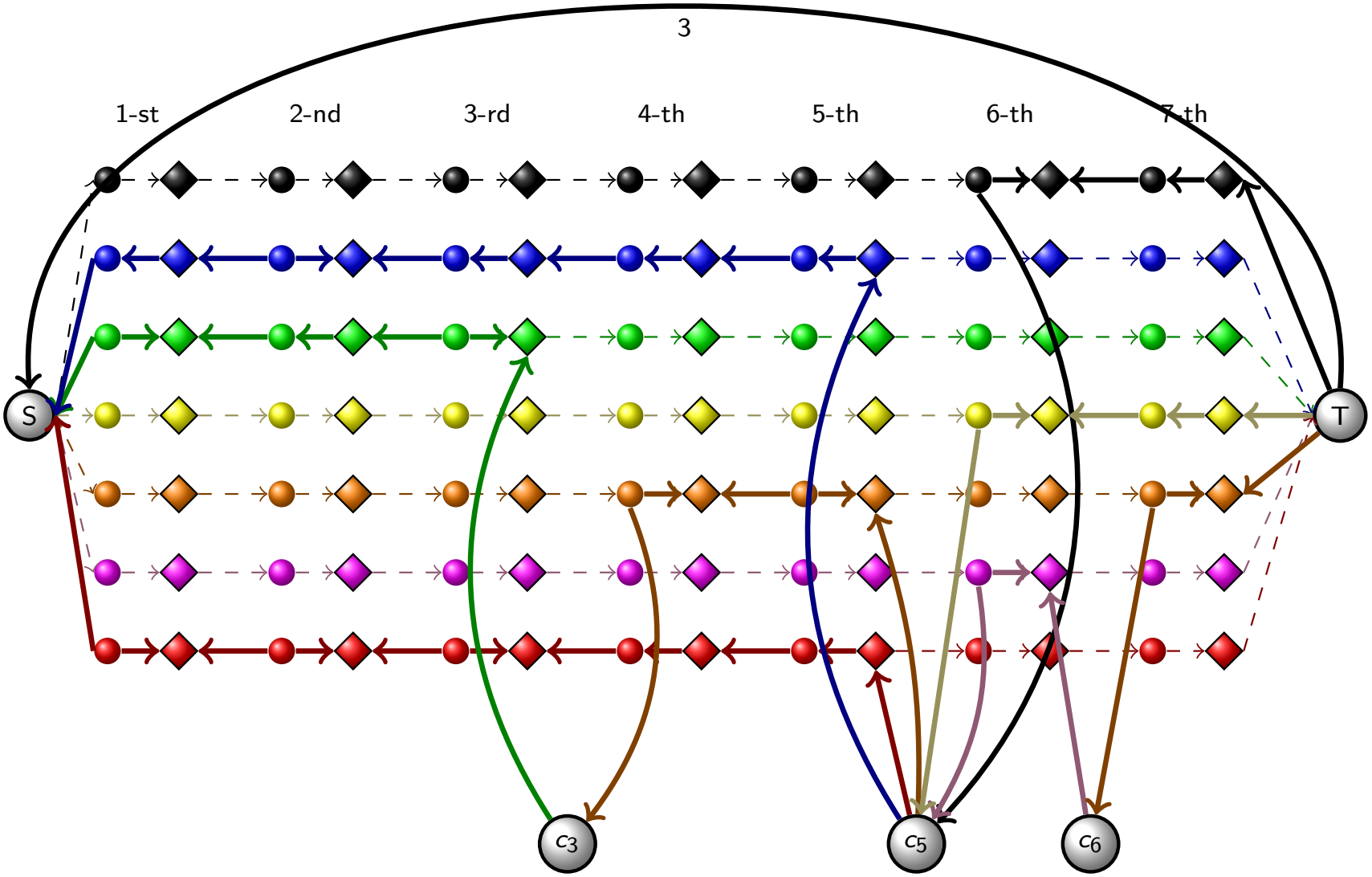
The Switch Constraint



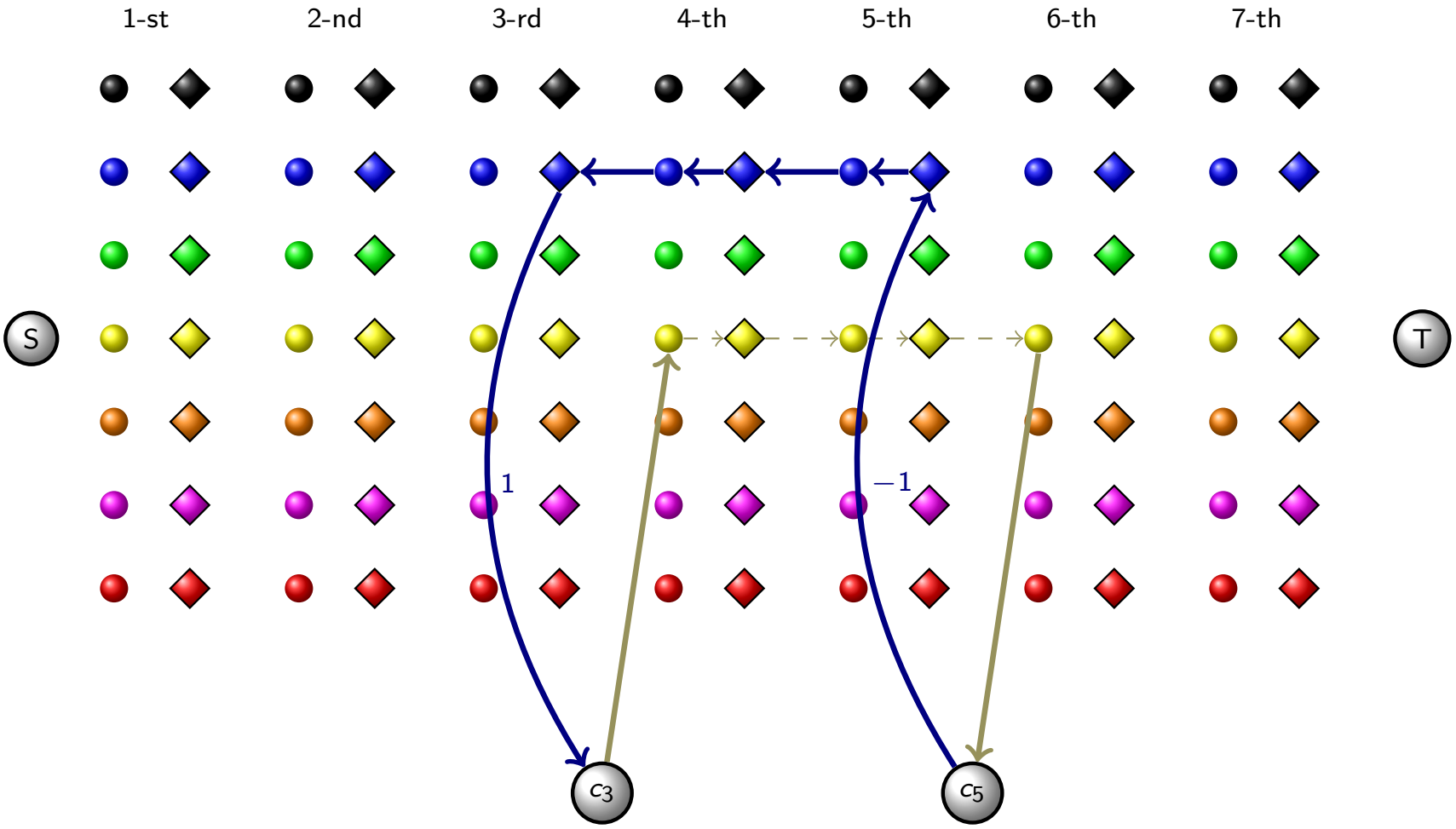
The Switch Constraint



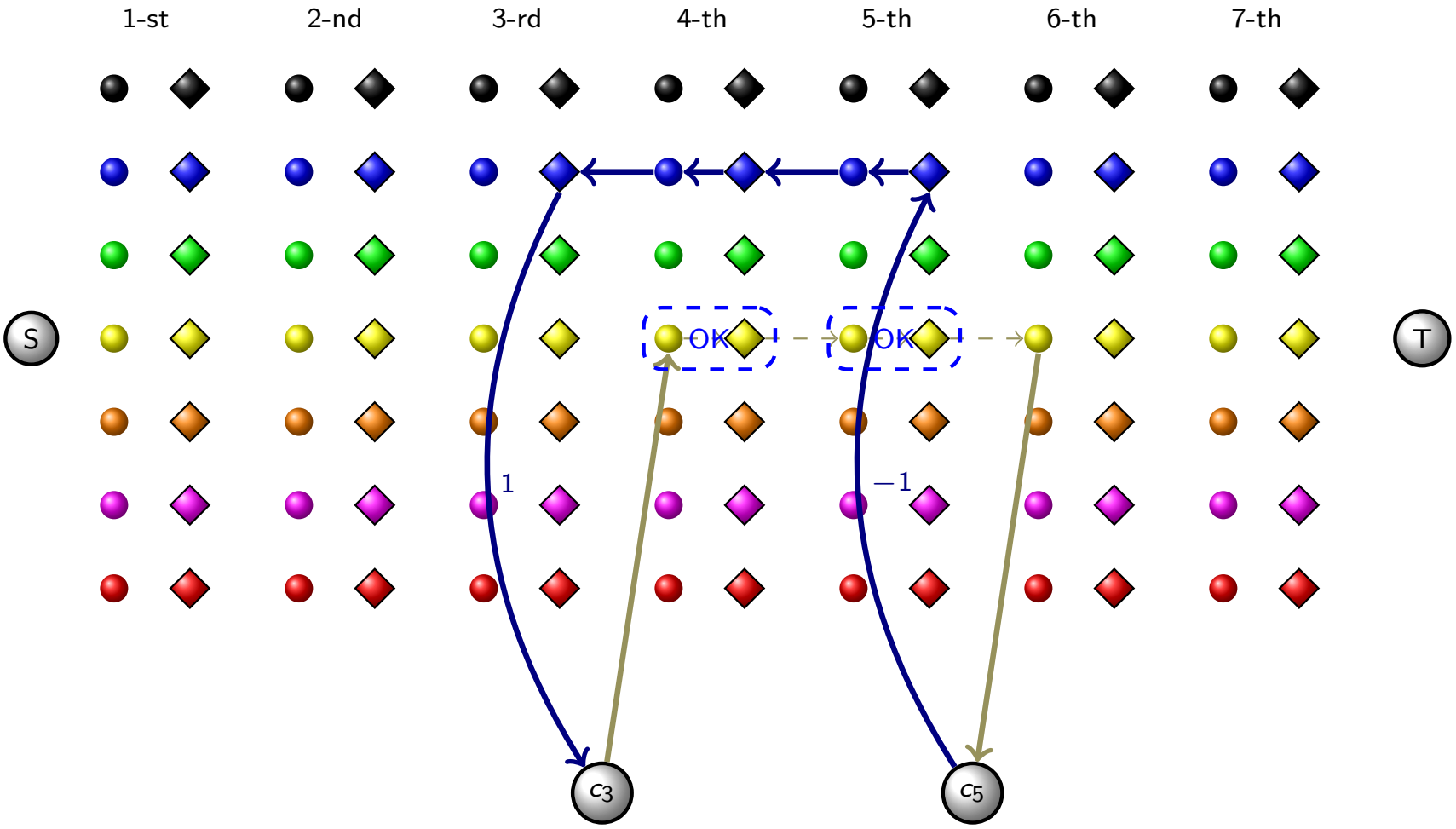
The Switch Constraint



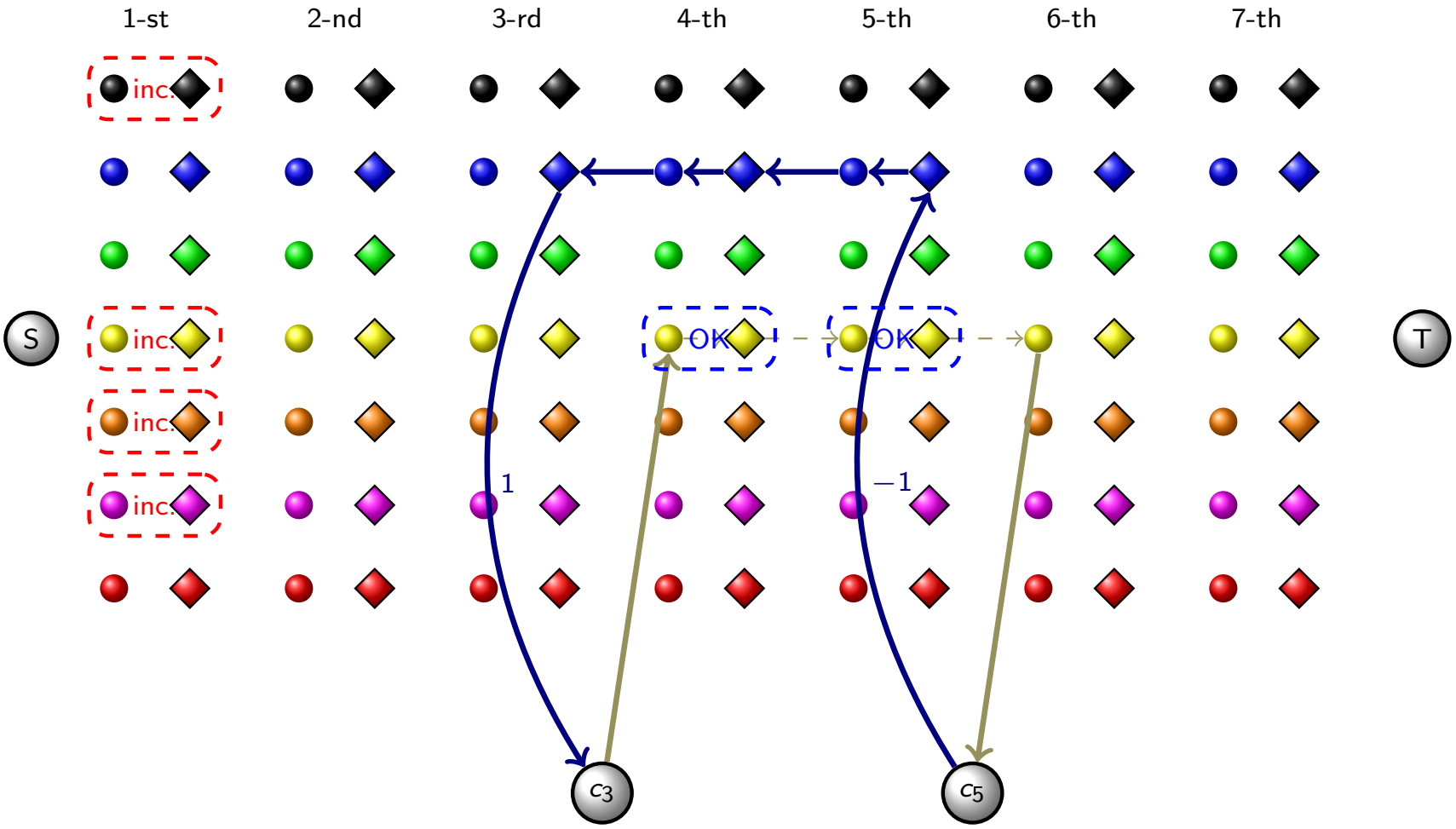
The Switch Constraint



The Switch Constraint



The Switch Constraint



Propagation Algorithm (details at CP-AI-OR)

Propagation Algorithm (details at CP-AI-OR)

- Find a maximum flow of minimum cost: $O(nd)$

Propagation Algorithm (details at CP-AI-OR)

- Find a maximum flow of minimum cost: $O(nd)$
- Eliminate negative costs of the residual graph: $O((nd)^{1.5})$

Propagation Algorithm (details at CP-AI-OR)

- Find a maximum flow of minimum cost: $O(nd)$
- Eliminate negative costs of the residual graph: $O((nd)^{1.5})$
- Find all cycles of costs 0 and 1 (but not 2 or greater): $O(n^2 d)$

Propagation Algorithm (details at CP-AI-OR)

- Find a maximum flow of minimum cost: $O(nd)$
- Eliminate negative costs of the residual graph: $O((nd)^{1.5})$
- Find all cycles of costs 0 and 1 (but not 2 or greater): $O(n^2 d)$

Total time complexity: $O(n^2 d + nd^{1.5})$

experiments

Complexity of Propagating vs. Complexity of Solving

- Propagating is often harder
  - E.g. SWITCH: solving in $O(nd)$, propagating in $O(n^2d + nd^{1.5})$

Complexity of Propagating vs. Complexity of Solving

- Propagating is often harder
  - E.g. SWITCH: solving in $O(nd)$, propagating in $O(n^2d + nd^{1.5})$
- Sometimes, we can propagate "for free", i.e., with the same time complexity as for solving

Propagating for "Free": the AtMostSeqCard Constraint

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d)$$

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type
- They should not occur alltogether (capacity constraints)
  - On each subsequence of size q, at most u are of this type

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d) \wedge \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u)$$

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type
- They should not occur alltogether (capacity constraints)
  - On each subsequence of size q, at most u are of this type
- Example: $u = 2$, $q = 4$, $d = 10$, $[x_1, \ldots, x_{22}]$

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d) \wedge \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u)$$

1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type
- They should not occur alltogether (capacity constraints)
  - On each subsequence of size q, at most u are of this type
- Example: $u = 2$, $q = 4$, $d = 10$, $[x_1, \ldots, x_{22}]$

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d) \wedge \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u)$$

1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type
- They should not occur alltogether (capacity constraints)
  - On each subsequence of size q, at most u are of this type
- Example: $u = 2$, $q = 4$, $d = 10$, $[x_1, \ldots, x_{22}]$

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d) \wedge \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u)$$

1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1

The AtMostSeqCard Constraint

- A sequence of variables $[x_1, \ldots, x_n]$
  - $x_i = 1$ iff the bolt is of the type that Charlot cares about
- There is a given demand for bolts of this type
- They should not occur alltogether (capacity constraints)
  - On each subsequence of size q, at most u are of this type
- Example: $u = 2$, $q = 4$, $d = 10$, $[x_1, \ldots, x_{22}]$

Definition: $\textsc{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \iff$

$$(\sum_{i=1}^{n} x_i = d) \wedge \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u)$$

1 0 0 1 1 0 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1

Solving the AtMostSeqCard Constraint

- Finding a support (solving)

Solving the AtMostSeqCard Constraint

- Finding a support (solving)
  - We can be greedy and assign 1 whenever possible ⇒ `leftmost`

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$. 0 . 1 . . . 0 . 0 1 . . 1 1

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$																						

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$		0		1				0		0	1			1								1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$		0		1				0		0	1	0	0	1										1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0		1				0		0	1	0	0	1									1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1				0		0	1	0	0	1										1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1	1			0		0	1	0	0	1								1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1	1	0	0	0		0	1	0	0	1									1

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints
  - Greedily inserting 1 works

Example: $\overrightarrow{w}$ = `leftmost` ($u = 2$, $q = 4$)

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	

- Support: maximizing the cardinality while respecting capacities
  - Changing the value 1 to 0 has no impact on capacity constraints
  - Greedily inserting 1 works
- We can find a support in $O(nq)$
- We can achieve AC in $O(n^2q)$

Filtering the AtMostSeqCard Constraint

- Can we propagate in the same time complexity?

Filtering the AtMostSeqCard Constraint

- Can we propagate in the same time complexity?
  - There is nothing to propagate unless $|\overrightarrow{w}| = d$

Filtering the AtMostSeqCard Constraint

- Can we propagate in the same time complexity?
  - There is nothing to propagate unless $|\overrightarrow{w}| = d$
  - Two runs of `leftmost` are enough

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$	.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$	1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	.	0	1	.	.	1	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	1	0	1	.	.	1	.	.	.	.	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	1	0	1	.	.	1	.	.	.	1	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	1	0	1	.	.	1	.	.	.	1	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

- We can achieve AC in $O(nq)$, that is, $O(n^2)$

Example: $\overrightarrow{w}$ = `leftmost` $(u = 2,\ q = 4,\ d = 10)$

$\mathcal{D}(x_i)$		.	0	.	1	.	.	.	0	1	0	1	.	.	1	.	.	.	1	.	.	.	1	
$\overrightarrow{w}[i]$		1	0	0	1	1	0	0	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	
$\overleftarrow{w}[i]$		0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	1	1	
$L[i]$	0	1	1	1	2	3	3	3	3	4	4	5	5	5	6	7	7	7	8	9	9	9	d	
$R[i]$		d	d	d	9	8	8	8	7	7	6	6	5	5	5	4	4	4	3	2	2	2	1	0

- We can achieve AC in $O(nq)$, that is, $O(n^2)$
- We can do better: `leftmost` can run in $O(n)$ details
- Total complexity of $O(n)$, optimal!

NP-hard Constraints

- If solving is NP-hard, then achieving AC is NP-hard

NP-hard Constraints

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?

NP-hard Constraints: the SoftAllEqual Constraints

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]
- What about an "AllEqual" constraint?

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]
- What about an "AllEqual" constraint?
  - Trivial to solve and propagate

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]
- What about an "AllEqual" constraint?
  - Trivial to solve and propagate
  - However, the soft version is far from trivial

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]
- What about an "AllEqual" constraint?
  - Trivial to solve and propagate
  - However, the soft version is far from trivial
  - Try to make the variables as equal as possible

The AllDifferent Constraint

- Each variable should take a distinct value (Matching)
  - Propagation algorithm [Régin 1994, Costa 1994]
- What about an "AllEqual" constraint?
  - Trivial to solve and propagate
  - However, the soft version is far from trivial
  - Try to make the variables as equal as possible
- Two versions
  - Maximise the number of equalities
  - Minimise the number of value skip

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

- Meeting Scheduling

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

- Meeting Scheduling
  - Each researcher states his/her availabilities

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

- Meeting Scheduling
  - Each researcher states his/her availabilities
  - Each researcher goes to at most one meeting

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

- Meeting Scheduling
  - Each researcher states his/her availabilities
  - Each researcher goes to at most one meeting
  - Schedule meetings so that:
    - ⋆ The number of interactions is maximum

The SoftAllEqual Constraint

Definition: $\textsc{SoftAllEqual}(k, [x_1, \ldots, x_n]) \iff$

At least k pairs of equal variables in the set $\{x_1, \ldots, x_n\}$

- Meeting Scheduling
  - Each researcher states his/her availabilities
  - Each researcher goes to at most one meeting
  - Schedule meetings so that:
    - ★ The number of interactions is maximum
    - ★ Interaction: a pair of researchers attend the same meeting

The SoftAllEqual Constraint

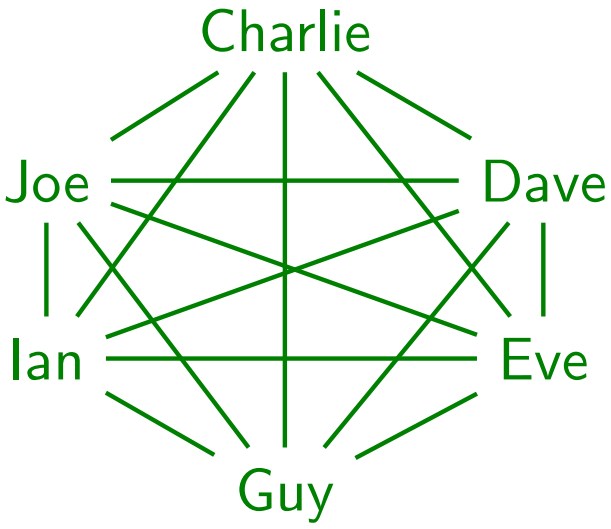
	Tue. am	Tue. pm	Wed. am	Wed. pm
Alice	✓			✓
Bob		✓		
Charlie	✓		✓	✓
Dave		✓	✓	
Eve		✓	✓	
Frank				✓
Guy	✓		✓	✓
Hugo	✓	✓		
Ian			✓	✓
Joe	✓	✓	✓	

The SoftAllEqual Constraint

	Tue. am	Tue. pm	Wed. am	Wed. pm
Alice	✓			✓
Bob		✓		
Charlie	✓		✓	✓
Dave		✓	✓	
Eve		✓	✓	
Frank				✓
Guy	✓		✓	✓
Hugo	✓	✓		
Ian			✓	✓
Joe	✓	✓	✓	

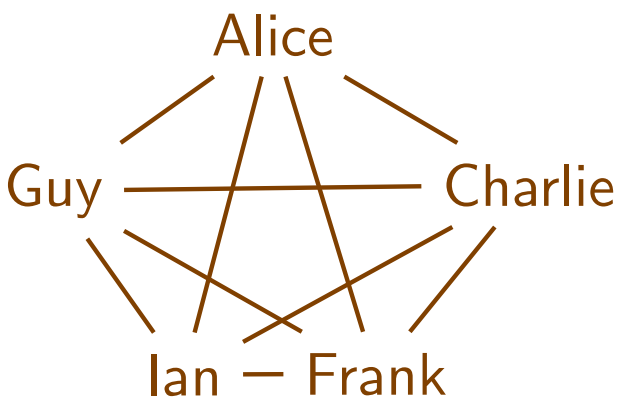
Alice — Hugo

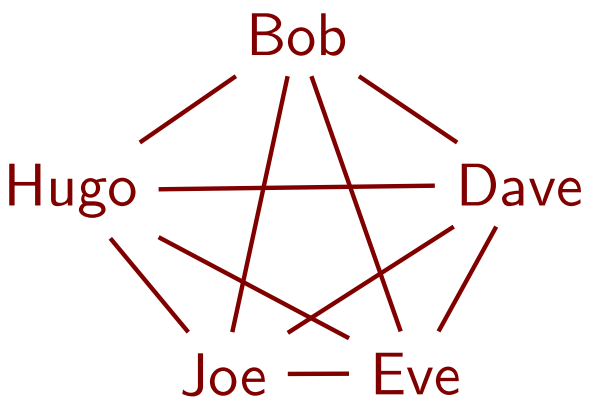
Bob



The SoftAllEqual Constraint

	Tue. am	Tue. pm	Wed. am	Wed. pm
Alice	✓			✓
Bob		✓		
Charlie	✓		✓	✓
Dave		✓	✓	
Eve		✓	✓	
Frank				✓
Guy	✓		✓	✓
Hugo	✓	✓		
Ian			✓	✓
Joe	✓	✓	✓	





The SoftAllEqual Constraint

- Finding an optimal solution is NP-hard (and so is Achieving AC)

The SoftAllEqual Constraint

- Finding an optimal solution is NP-hard (and so is Achieving AC)
- Given an ordering (red, blue, green, ...), we can define a greedy algo:
  - Assign X to the first value whenever possible, then the second, etc.

The SoftAllEqual Constraint

- Finding an optimal solution is NP-hard (and so is Achieving AC)
- Given an ordering (red, blue, green, ...), we can define a greedy algo:
  - Assign X to the first value whenever possible, then the second, etc.

If values are ordered by occurrences: 2-approximation

The SoftAllEqual Constraint

- Finding an optimal solution is NP-hard (and so is Achieving AC)
- Given an ordering (red, blue, green, ...), we can define a greedy algo:
  - Assign X to the first value whenever possible, then the second, etc.

If values are ordered by occurrences: 2-approximation

There exists such an ordering that gives an optimal solution

- There is an algorithm exponential only in the number of values (V)

The SoftAllEqual Constraint

- Finding an optimal solution is NP-hard (and so is Achieving AC)
- Given an ordering (red, blue, green, ...), we can define a greedy algo:
  - Assign X to the first value whenever possible, then the second, etc.

If values are ordered by occurrences: 2-approximation

There exists such an ordering that gives an optimal solution

- There is an algorithm exponential only in the number of values (V)
- Exponential in something that is potentially smaller than the data nd
  - Fixed Parameter Tractable

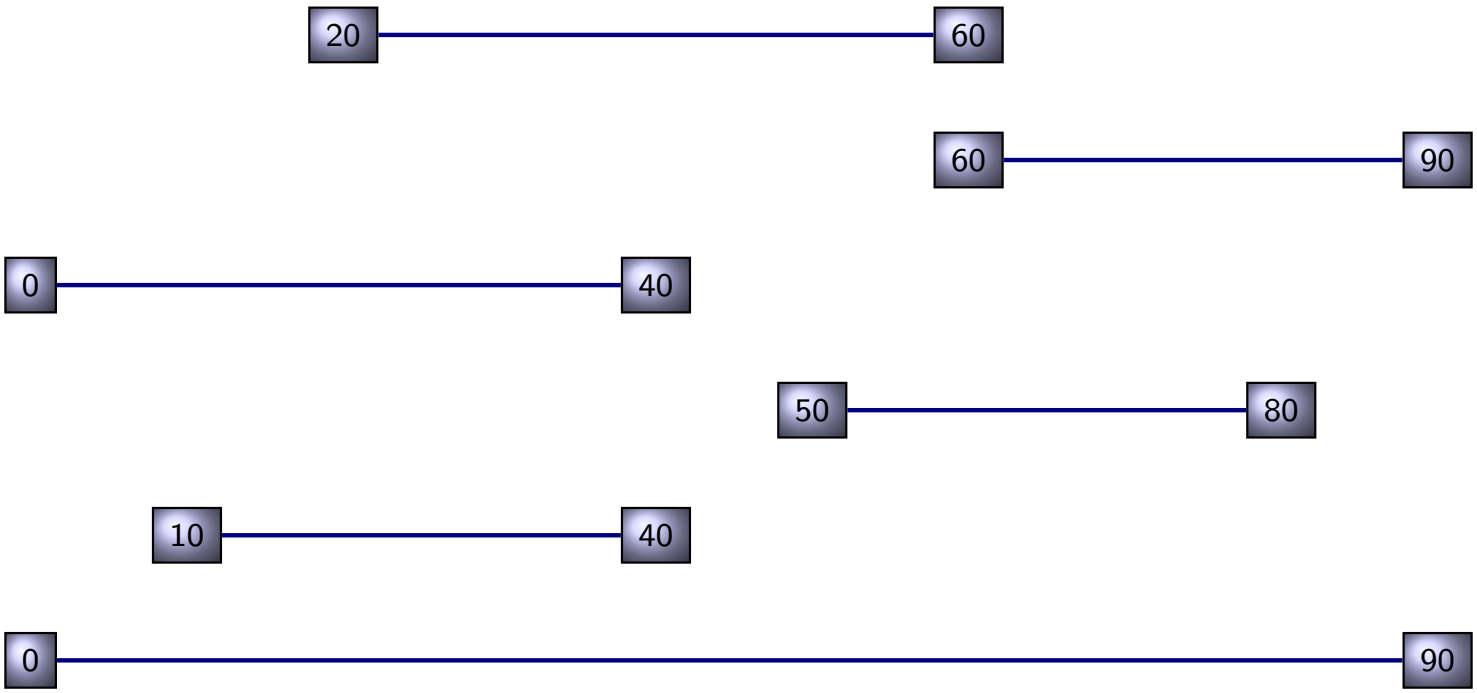
The SoftAllEqual Constraint

- Achieving AC is NP-hard

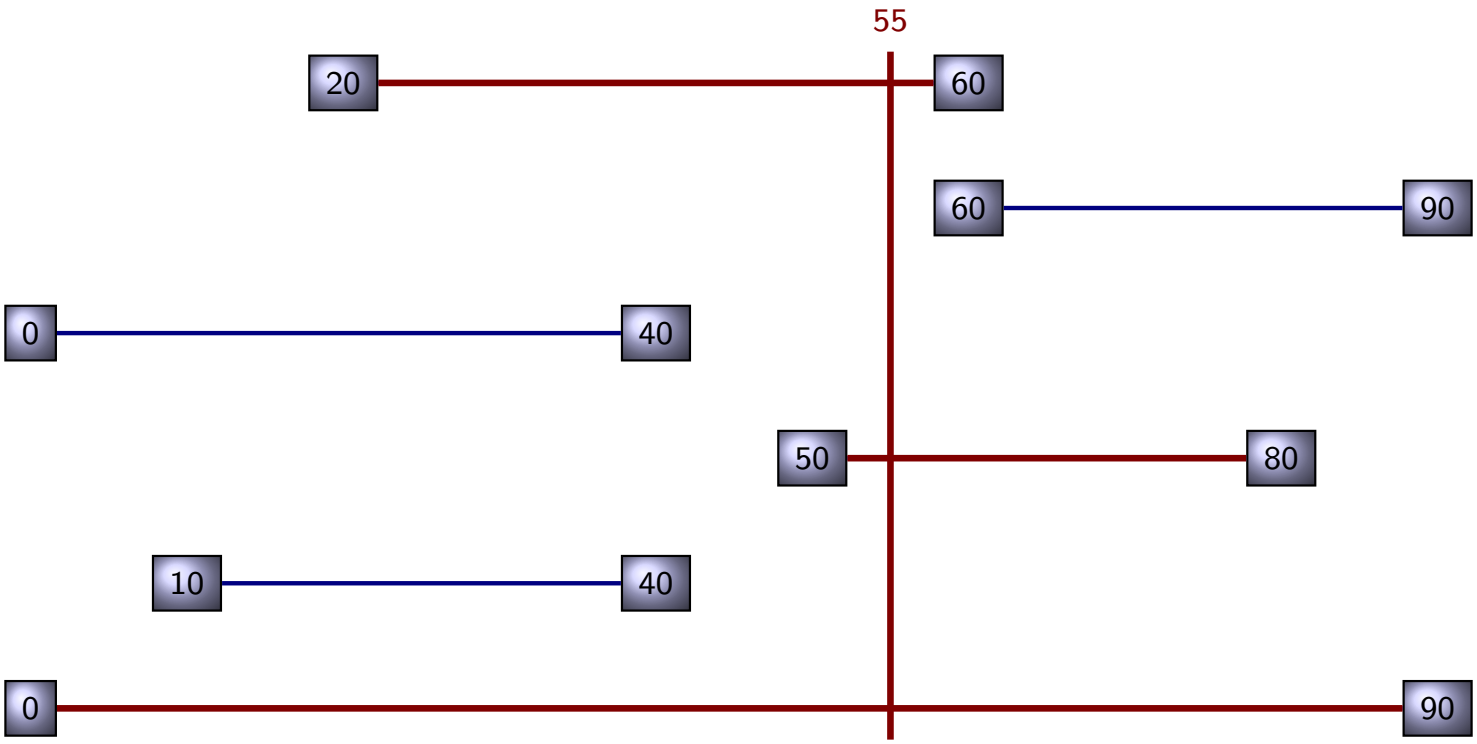
The SoftAllEqual Constraint

- Achieving AC is NP-hard
- What about BC?

A bc Algorithm for SoftAllEqual

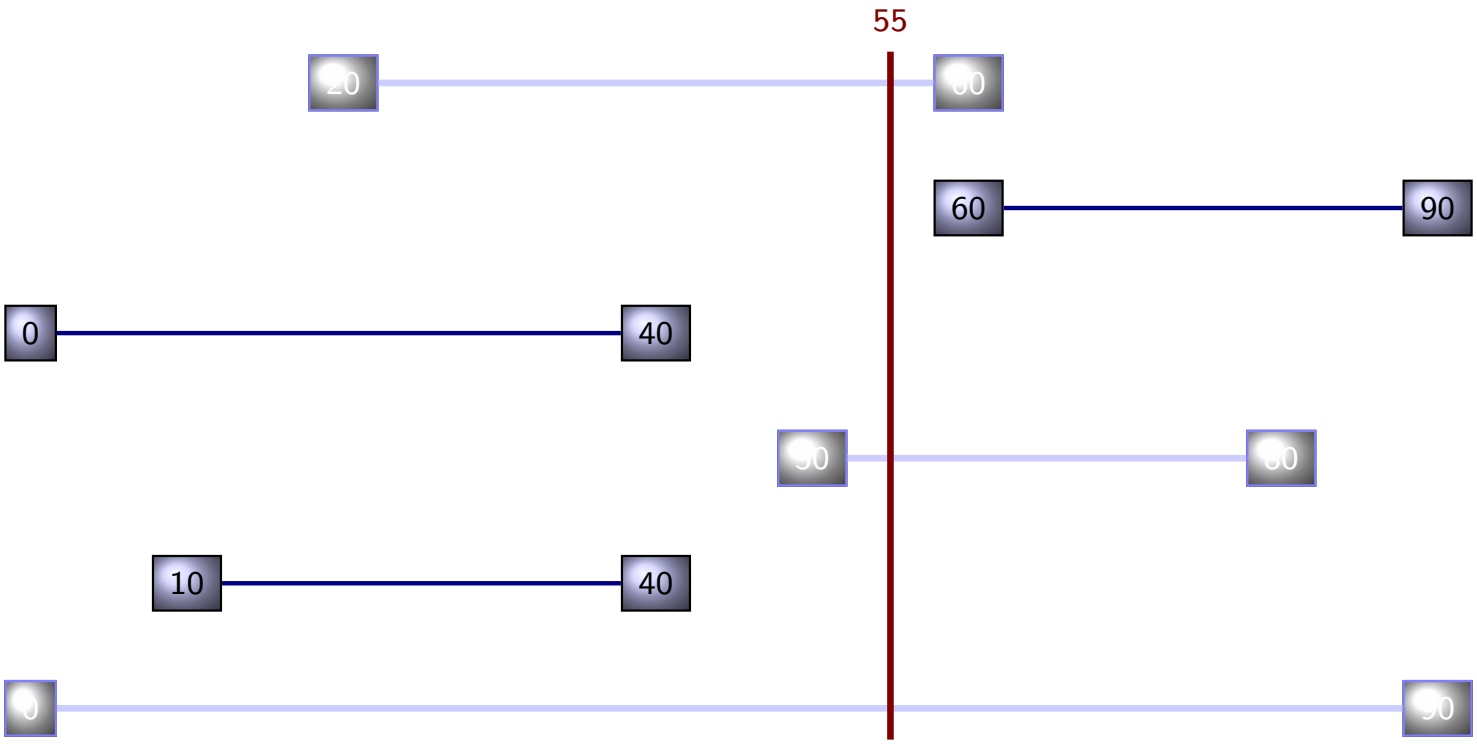


A bc Algorithm for SoftAllEqual



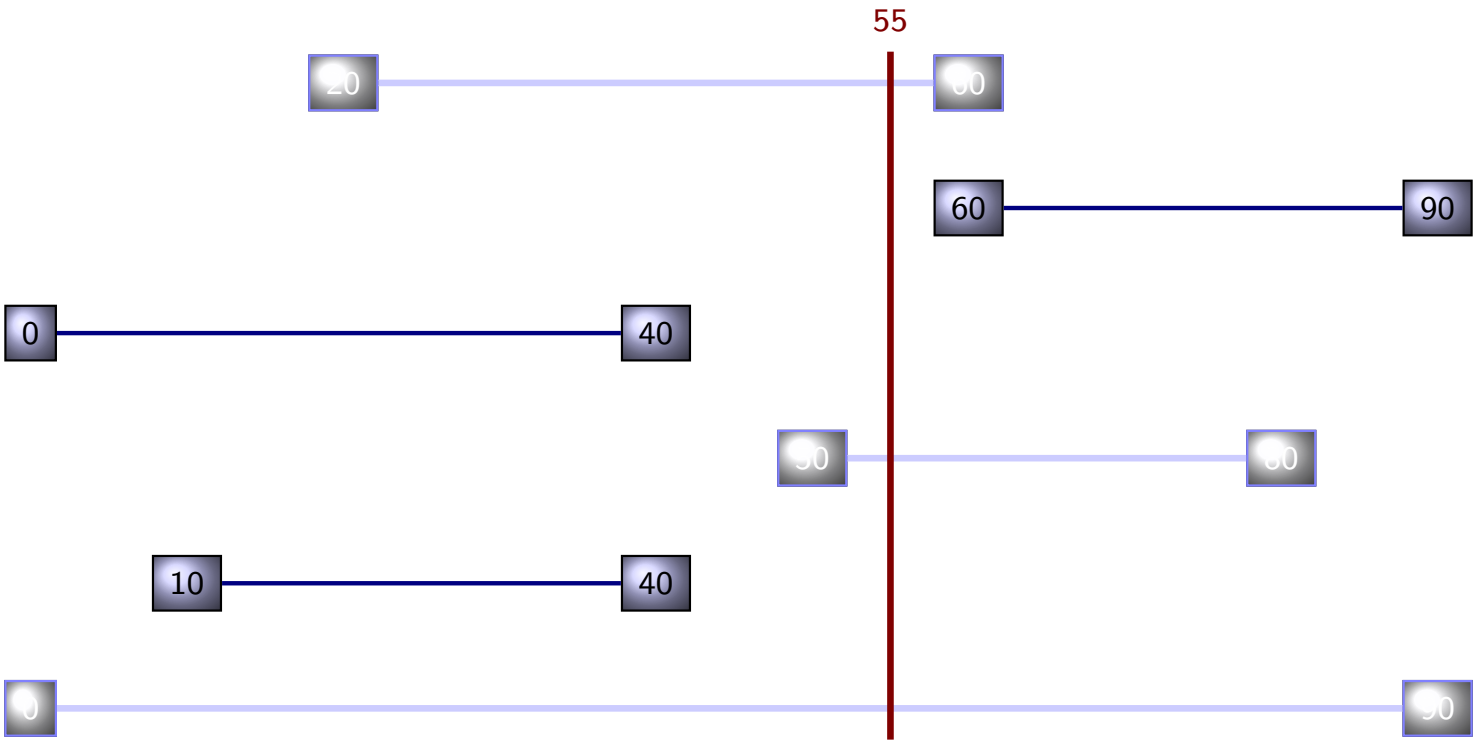
- If we assign the value 55 first, we assign it to all variables

A bc Algorithm for SoftAllEqual



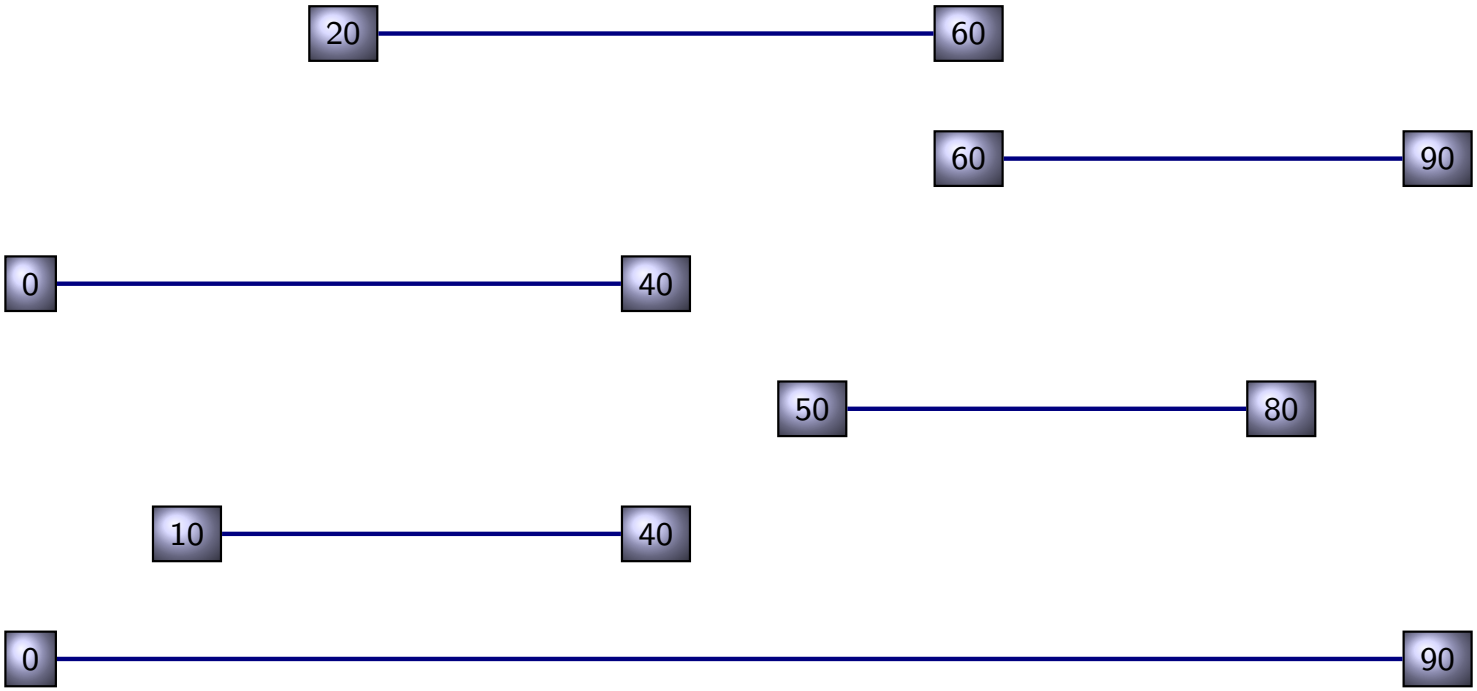
- If we assign the value 55 first, we assign it to all variables
- The problem is split into two smaller parts

A bc Algorithm for SoftAllEqual



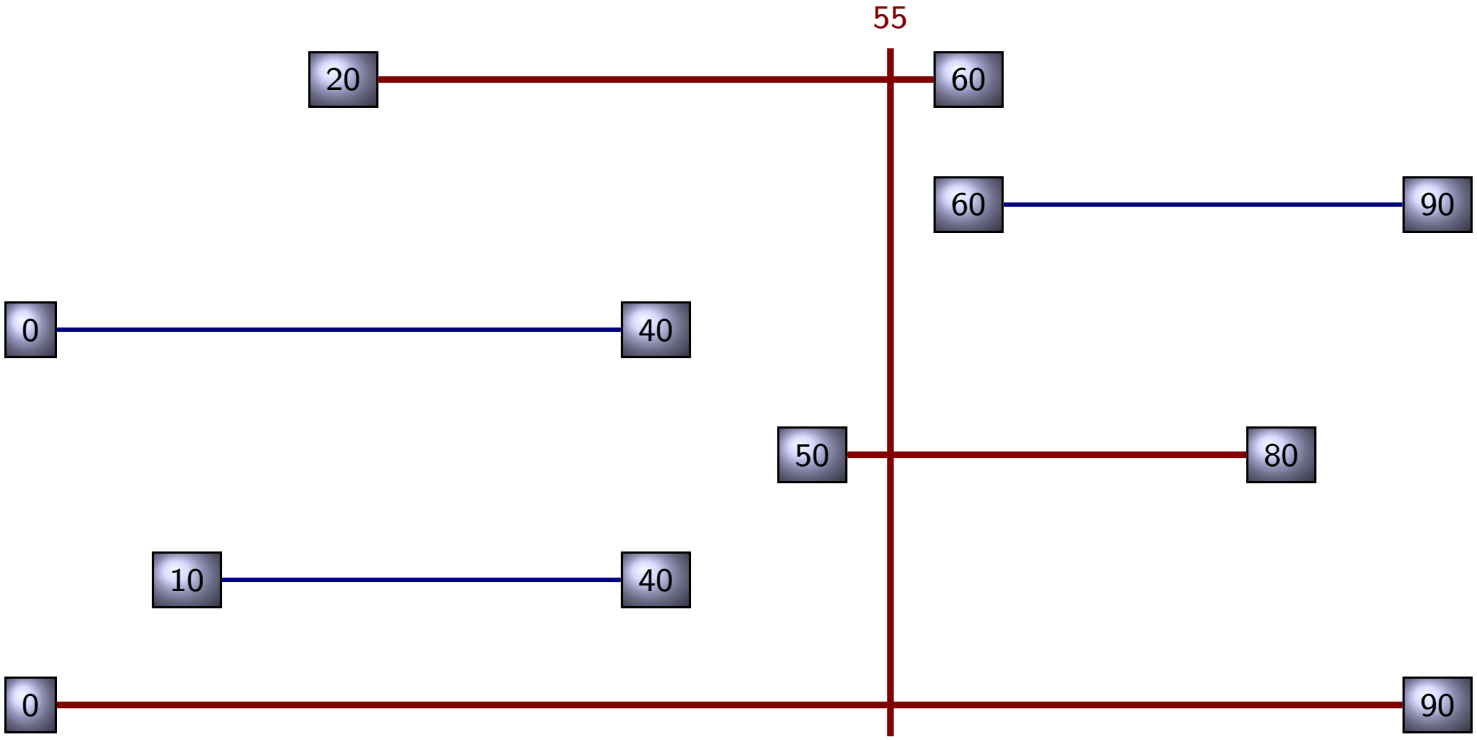
- If we assign the value 55 first, we assign it to all variables
- The problem is split into two smaller parts
  - Polynomial size search tree: Dynamic Programming

Dynamic Programming



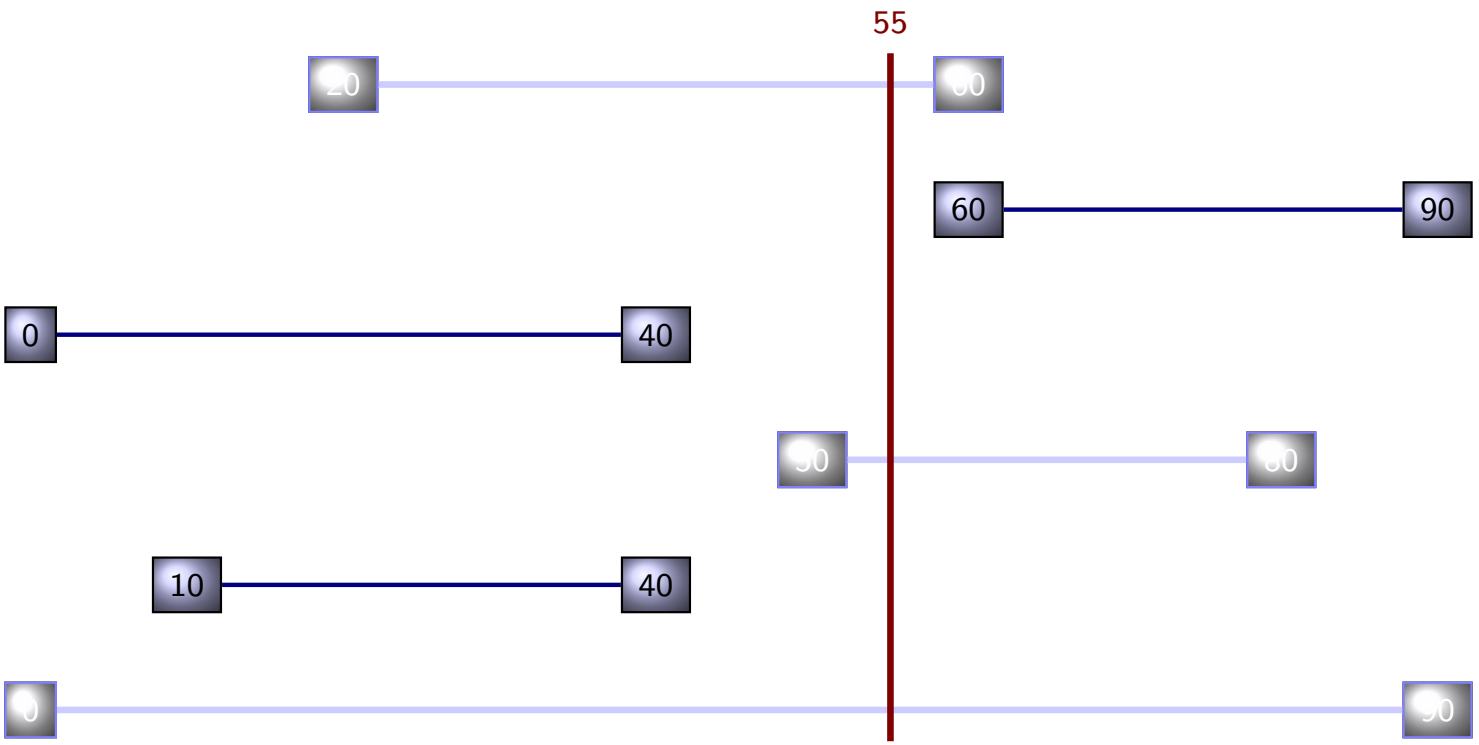
- $T_{a,b}$ = Maximum equalities on variables entirely contained in $[a, b]$

Dynamic Programming



- $T_{a,b}$ = Maximum equalities on variables entirely contained in $[a, b]$
- Given a value $v \in [a, b]$, we must assign variables to v if possible
  - $T_{a,b}(v)$ = (Binomial coeff of) the number of variables hit by v +

Dynamic Programming



- $T_{a,b}$ = Maximum equalities on variables entirely contained in $[a, b]$
- Given a value $v \in [a, b]$, we must assign variables to v if possible
  - $T_{a,b}(v)$ = (Binomial coeff of) the number of variables hit by v +
  - table content on the splitted parts: $T_{a,v-1} + T_{v+1,b}$

Dynamic Programming

- Time complexity: $O(nV^3)$
  - Table of size V^2, for each cell, go through the V values and count occurrences ($O(n)$)
- Since we consider interval domains, V can be very large!

Dynamic Programming

- Time complexity: $O(nV^3)$
  - Table of size V^2, for each cell, go through the V values and count occurrences ($O(n)$)
- Since we consider interval domains, V can be very large!
  - We can count occurrences of all values in $O(n + V)$

Dynamic Programming

- Time complexity: $O(nV^3)$
  - Table of size V^2, for each cell, go through the V values and count occurrences ($O(n)$)
- Since we consider interval domains, V can be very large!
  - We can count occurrences of all values in $O(n + V)$
  - Only maximal cliques matter

Dynamic Programming

- Time complexity: $O(nV^3)$
  - Table of size V^2, for each cell, go through the V values and count occurrences ($O(n)$)
- Since we consider interval domains, V can be very large!
  - We can count occurrences of all values in $O(n + V)$
  - Only maximal cliques matter
  - We can reformulate in an equivalent problem with as many values as maximal cliques
  - Interval graph, we can find all ($O(n)$) cliques in $O(n \log n)$ time

Dynamic Programming

- Time complexity: $O(nV^3)$
  - Table of size V^2, for each cell, go through the V values and count occurrences ($O(n)$)
- Since we consider interval domains, V can be very large!
  - We can count occurrences of all values in $O(n + V)$
  - Only maximal cliques matter
  - We can reformulate in an equivalent problem with as many values as maximal cliques
  - Interval graph, we can find all ($O(n)$) cliques in $O(n \log n)$ time
- Algorithm in $O(n^3)$ to find a solution and $O(n^4)$ for BC

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?
- There are alternatives

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?
- There are alternatives
  - Achieve a weaker consistency (e.g., BC)

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?
- There are alternatives
  - Achieve a weaker consistency (e.g., BC)
  - Approximate AC

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?
- There are alternatives
  - Achieve a weaker consistency (e.g., BC)
  - Approximate AC
  - Use an exponential algorithm (good case: Fixed Parameter Tractable)

Assessing the Tradeoff

- If solving is NP-hard, then achieving AC is NP-hard
- Should we decompose into simpler constraints?
- There are alternatives
  - Achieve a weaker consistency (e.g., BC)
  - Approximate AC
  - Use an exponential algorithm (good case: Fixed Parameter Tractable)
- The best tradeoff might not be obvious
  - Theoretical comparison
  - Empirical comparison

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

- NP-hard, equivalent to Minimum Hitting Set

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

- NP-hard, equivalent to Minimum Hitting Set

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

- NP-hard, equivalent to Minimum Hitting Set
  - Domains = collection of sets ($\mathcal{D}(x_i) = S_i$)

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

- NP-hard, equivalent to Minimum Hitting Set
  - Domains = collection of sets ($\mathcal{D}(x_i) = S_i$)
  - x_i = an element in $H \cap S_i$

Assessing the Tradeoff: The AtMostNValue Constraint

Definition: $\textsc{AtMostNValue}([x_1, \ldots, x_n], N) \iff$

$[x_1, \ldots, x_n]$ assigned using at most N distinct values

- NP-hard, equivalent to Minimum Hitting Set
  - Domains = collection of sets ($\mathcal{D}(x_i) = S_i$)
  - x_i = an element in $H \cap S_i$
  - $N = |H|$

Hitting Set

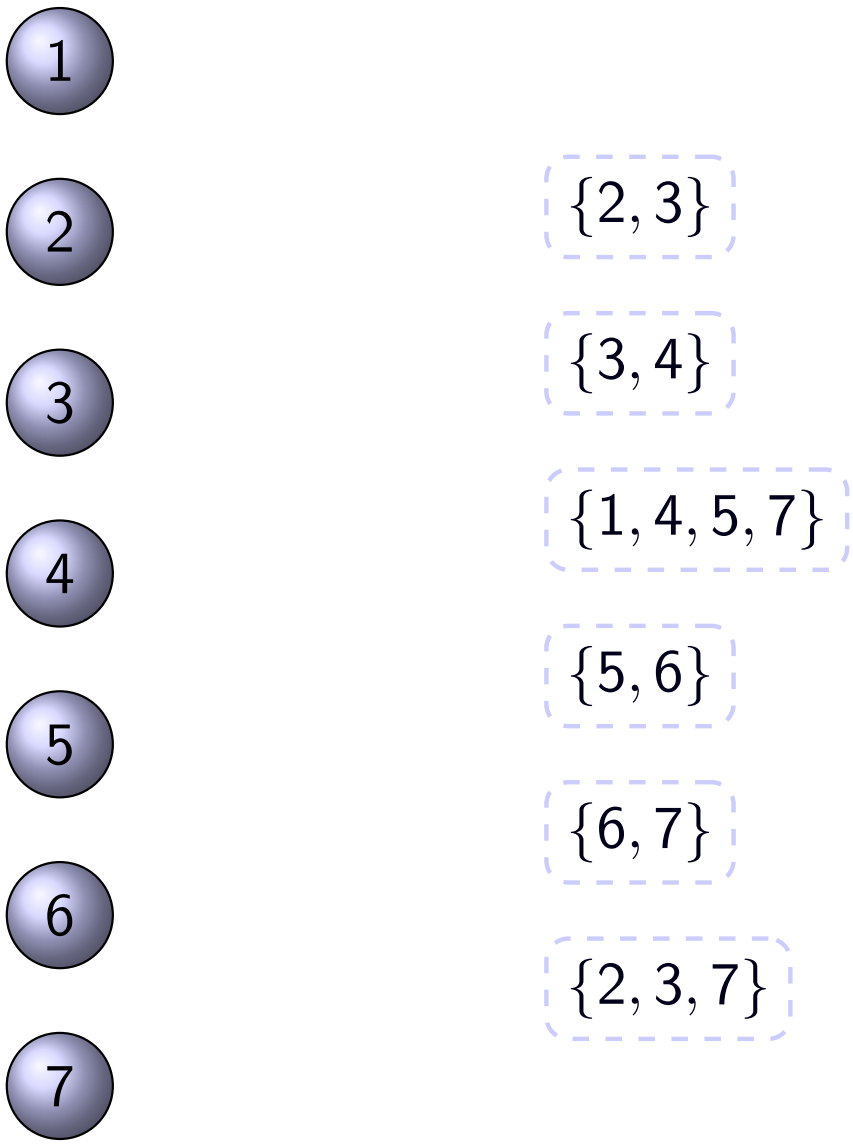
$$\mathcal{D}(a) = \{2, 3\}$$
$$\mathcal{D}(b) = \{3, 4\}$$
$$\mathcal{D}(c) = \{1, 4, 5, 7\}$$
$$\mathcal{D}(d) = \{5, 6\}$$
$$\mathcal{D}(e) = \{6, 7\}$$
$$\mathcal{D}(f) = \{2, 3, 7\}$$

Hitting Set

$$\mathcal{D}(a) = \{2, 3\}$$
$$\mathcal{D}(b) = \{3, 4\}$$
$$\mathcal{D}(c) = \{1, 4, 5, 7\}$$
$$\mathcal{D}(d) = \{5, 6\}$$
$$\mathcal{D}(e) = \{6, 7\}$$
$$\mathcal{D}(f) = \{2, 3, 7\}$$

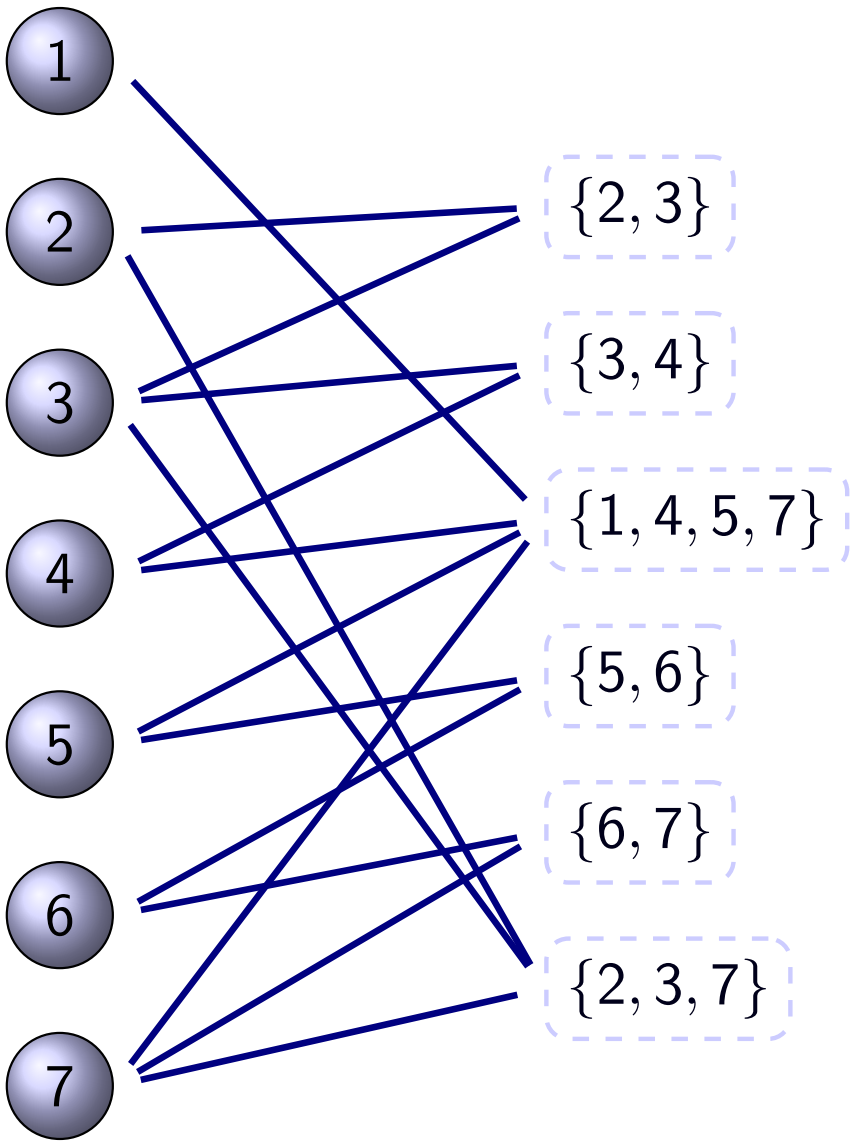
Hitting Set

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



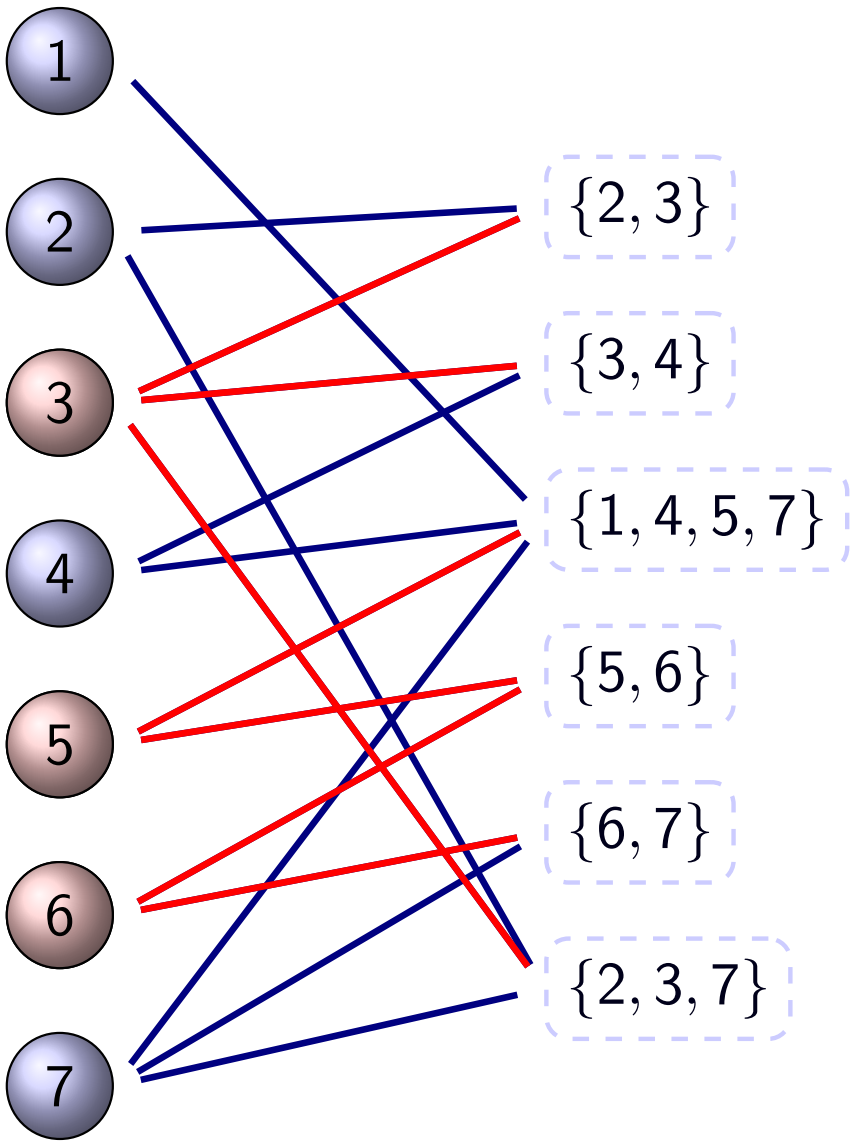
Hitting Set

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



Hitting Set

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



Hitting Set, Clique Cover

Hitting Set, Clique Cover

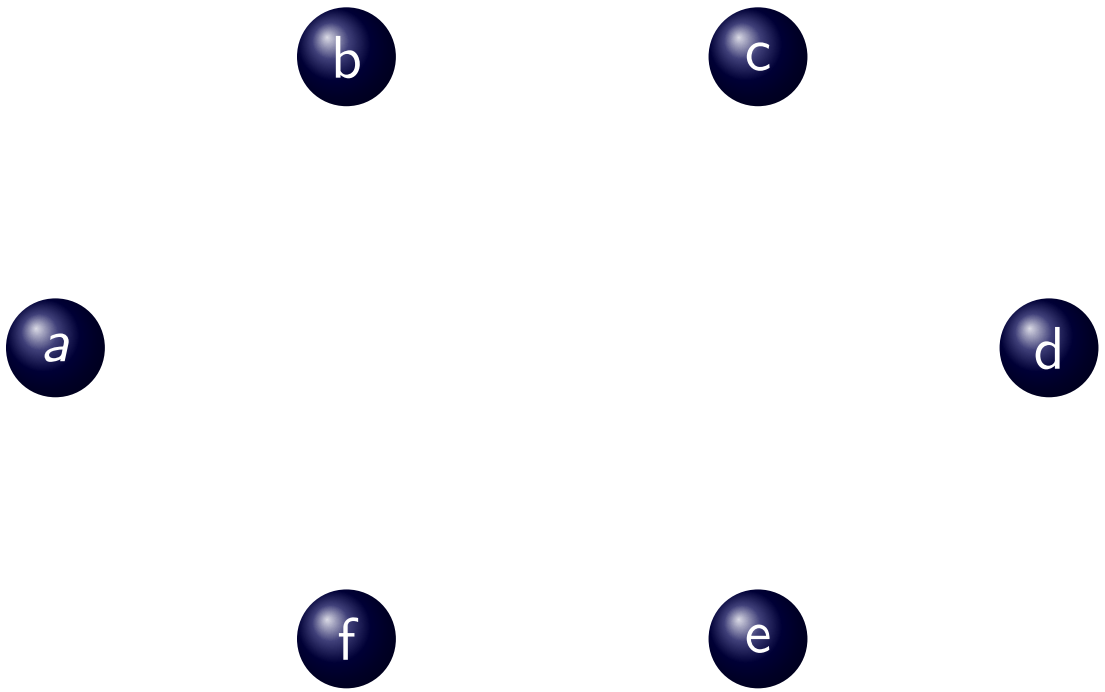
- Can we approximate Minimum Hitting Set to filter AtMostNValue?

Hitting Set, Clique Cover

- Can we approximate Minimum Hitting Set to filter AtMostNValue?
  - We need a lower bound; approximating MHS gives us an upper bound
  - Another approach: Clique Cover of the Intersection Graph

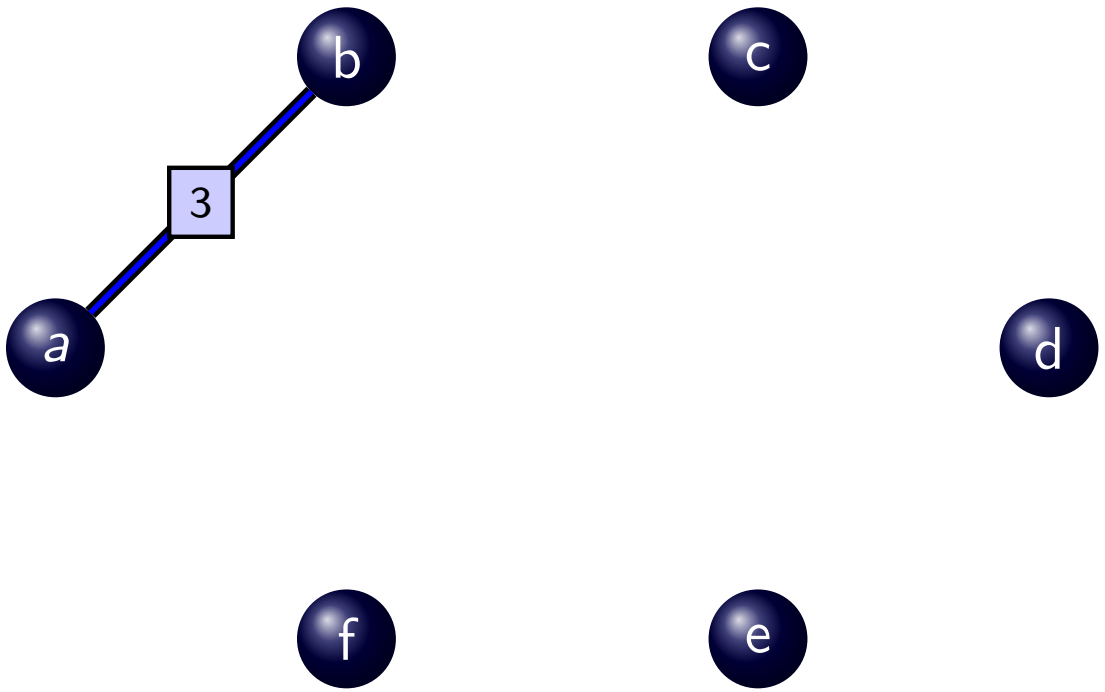
Intersection Graph: Discrete domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



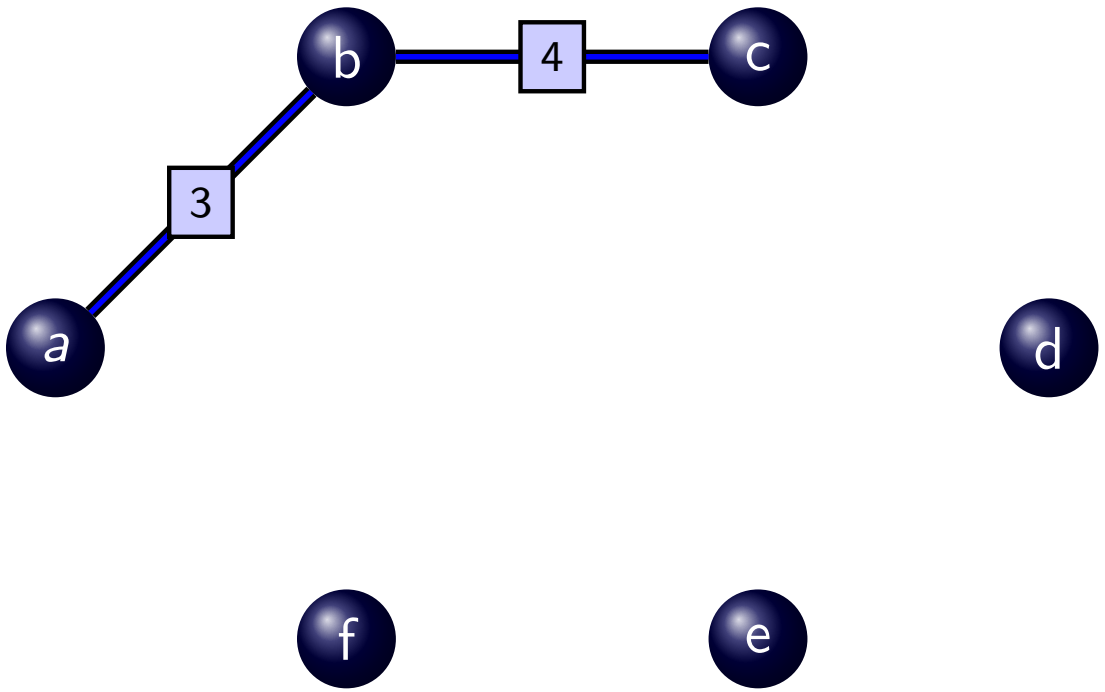
Intersection Graph: Discrete domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



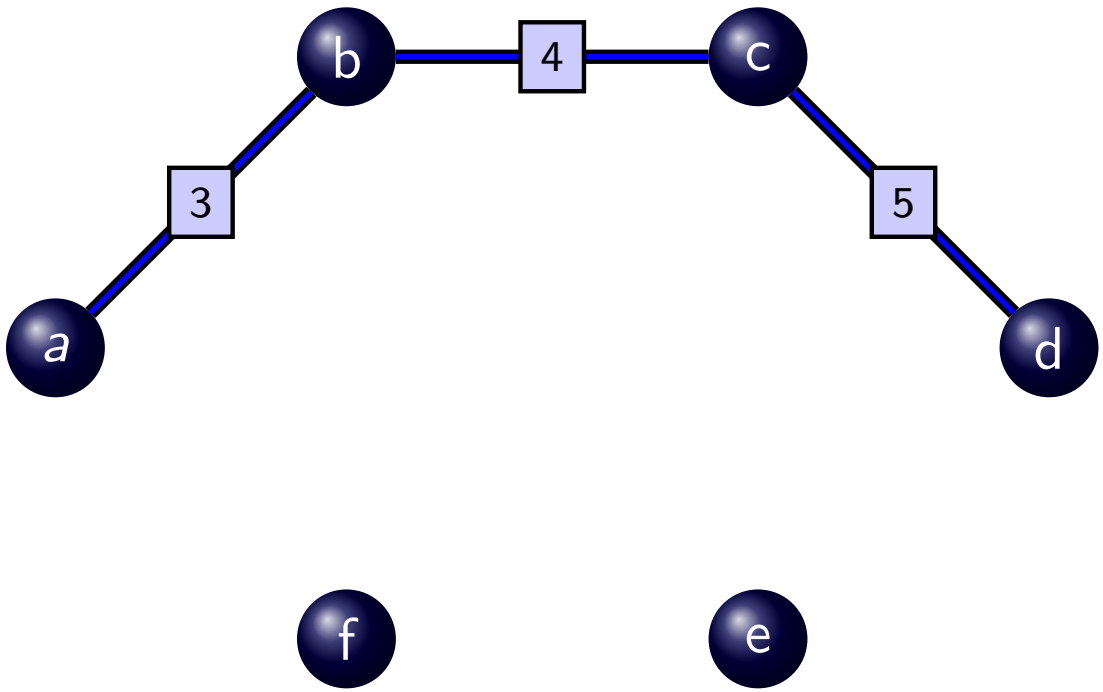
Intersection Graph: Discrete domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



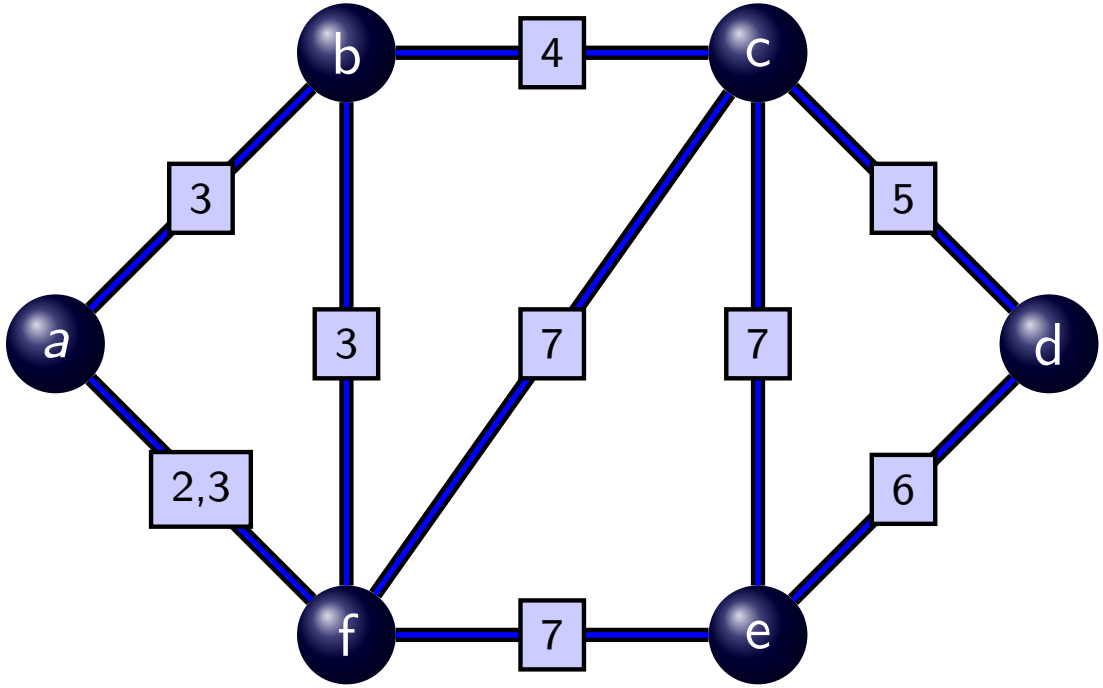
Intersection Graph: Discrete domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



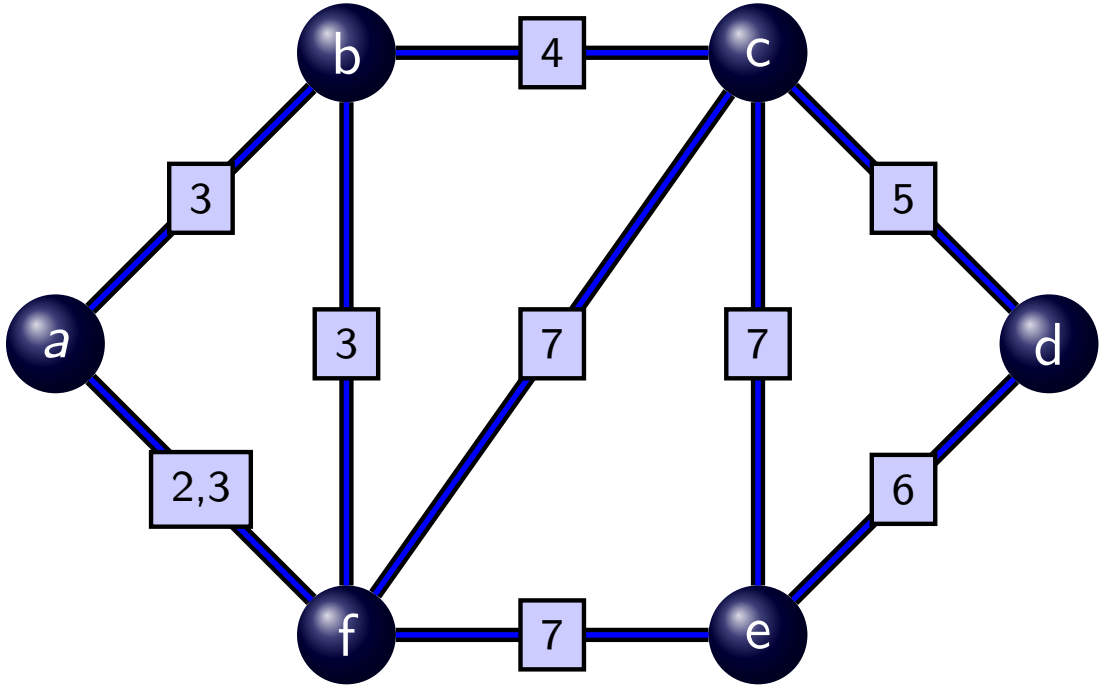
Intersection Graph: Discrete domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



Intersection Graph: Discrete domains

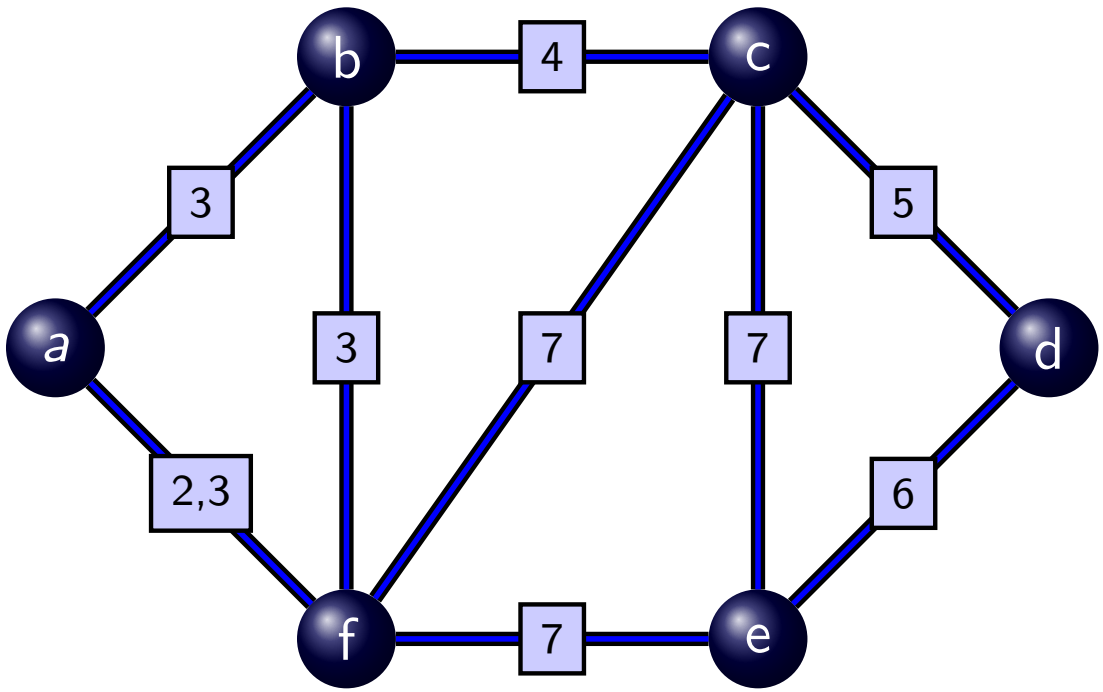
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



- A Minimum Clique Cover is a lower bound

Intersection Graph: Discrete domains

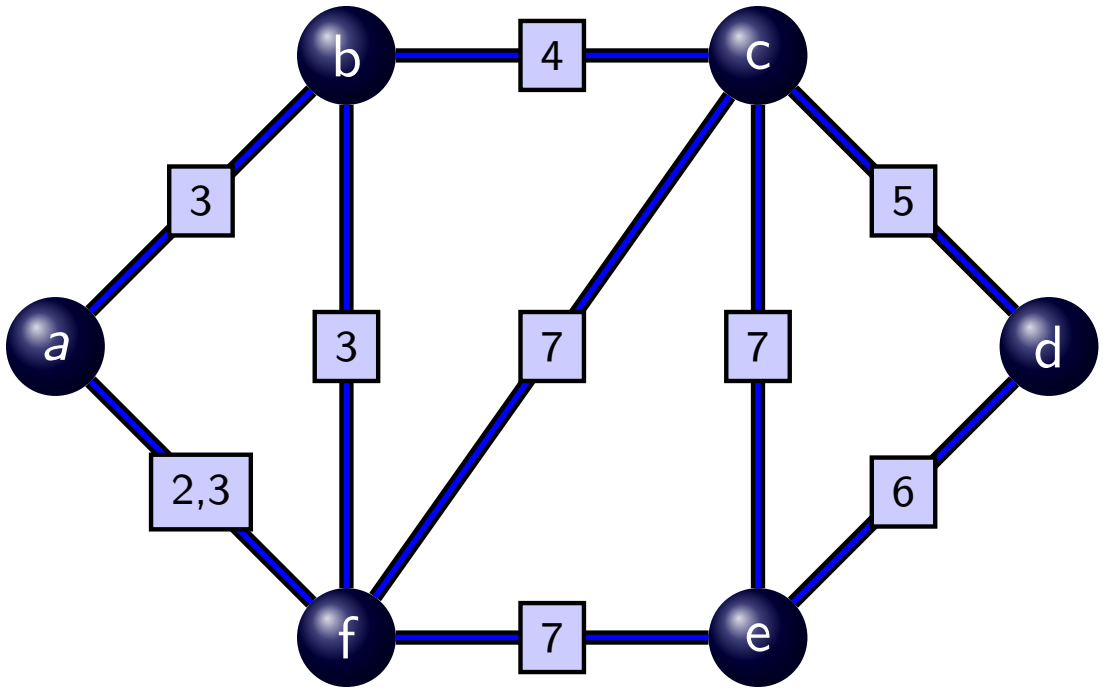
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



- A Minimum Clique Cover is a lower bound
- However, finding a Minimum Clique Cover is NP-hard

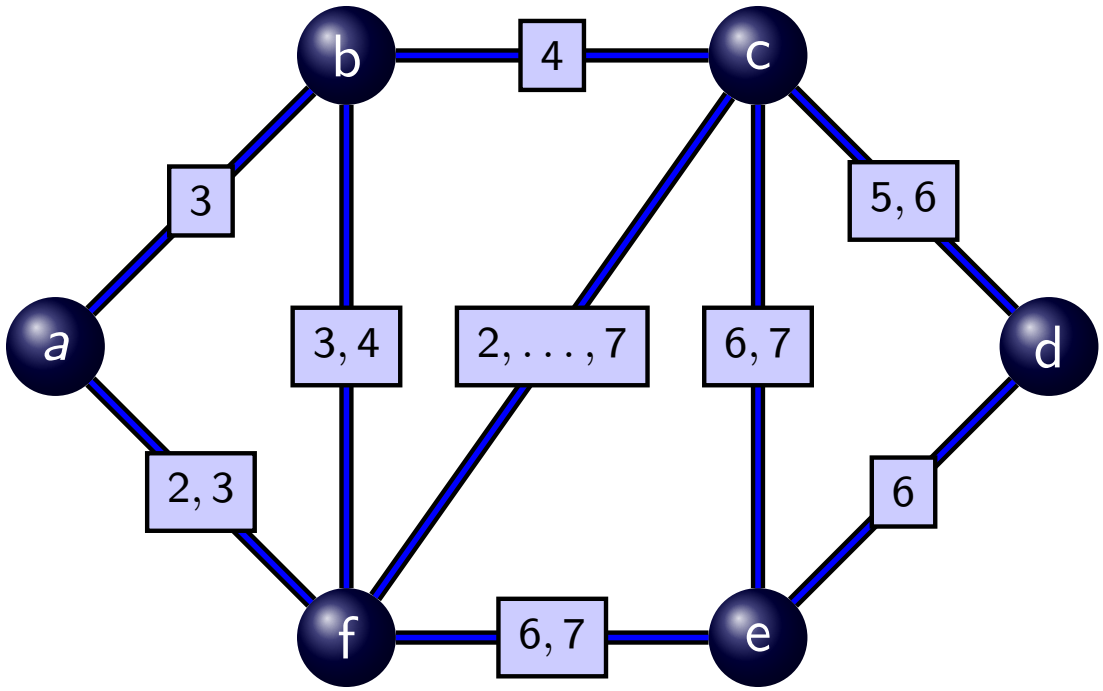
Intersection Graph: Interval domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



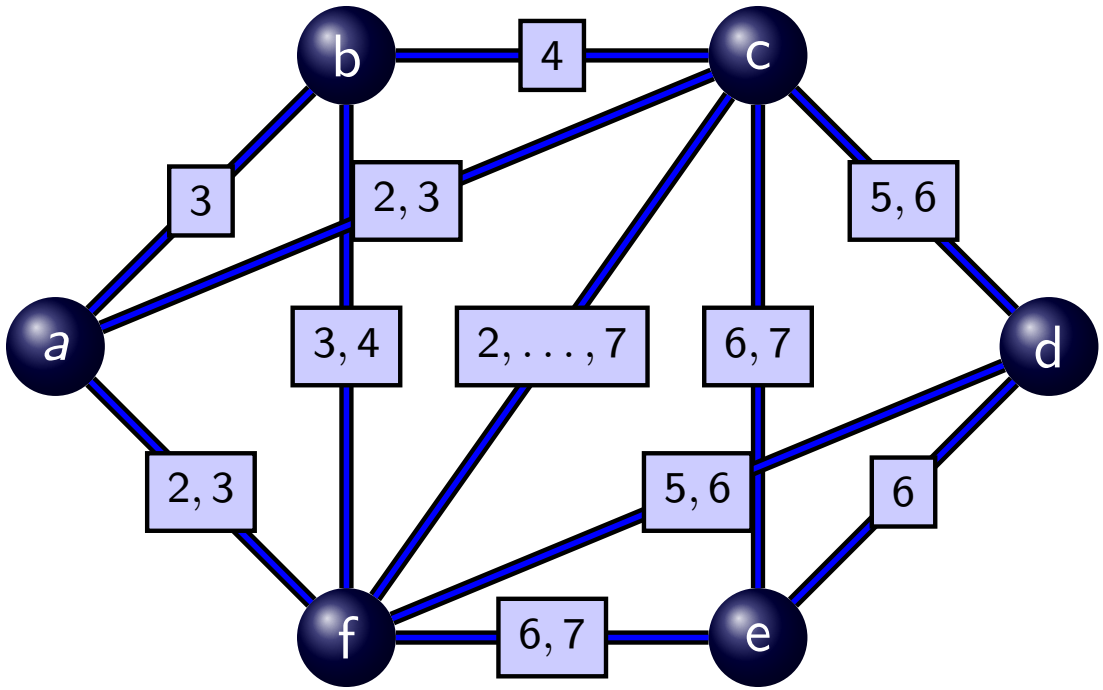
Intersection Graph: Interval domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, \ldots, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, \ldots, 7\}$



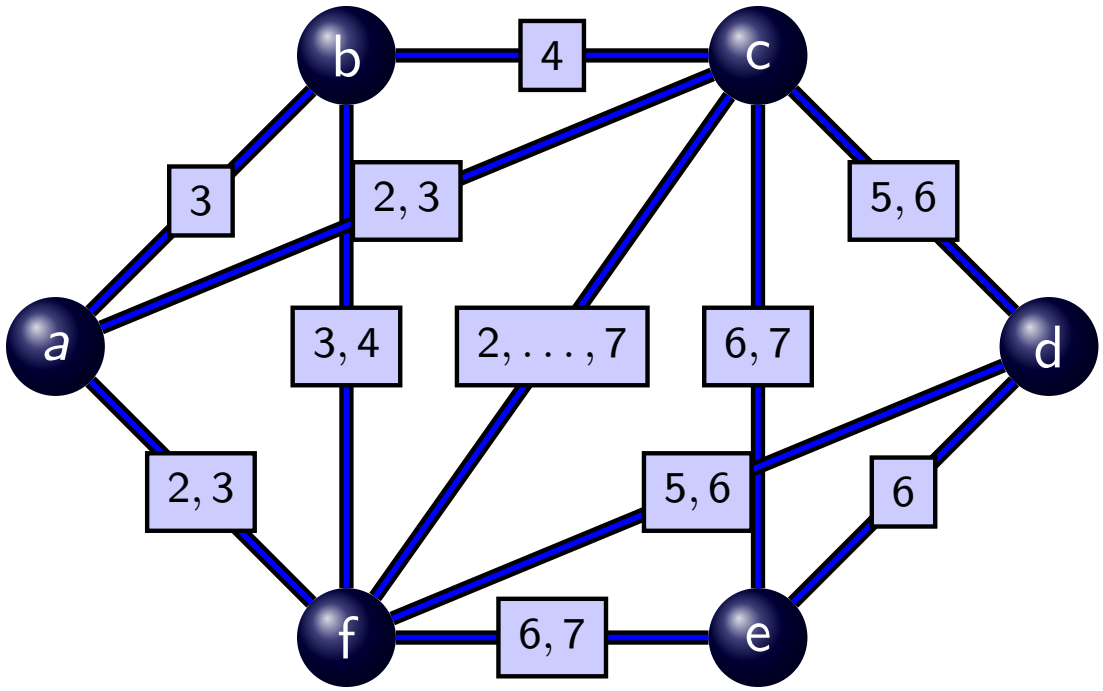
Intersection Graph: Interval domains

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, \ldots, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, \ldots, 7\}$



Intersection Graph: Interval domains

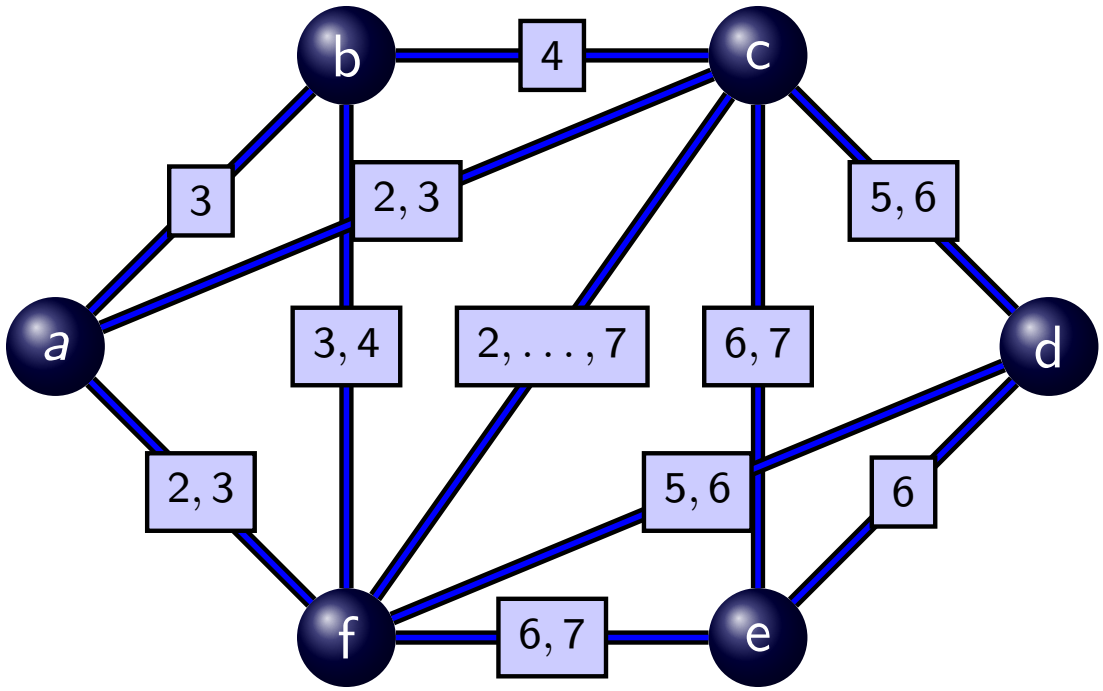
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, \ldots, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, \ldots, 7\}$



- Finding a Minimum Clique Cover is polynomial ($O(n \log n)$)

Intersection Graph: Interval domains

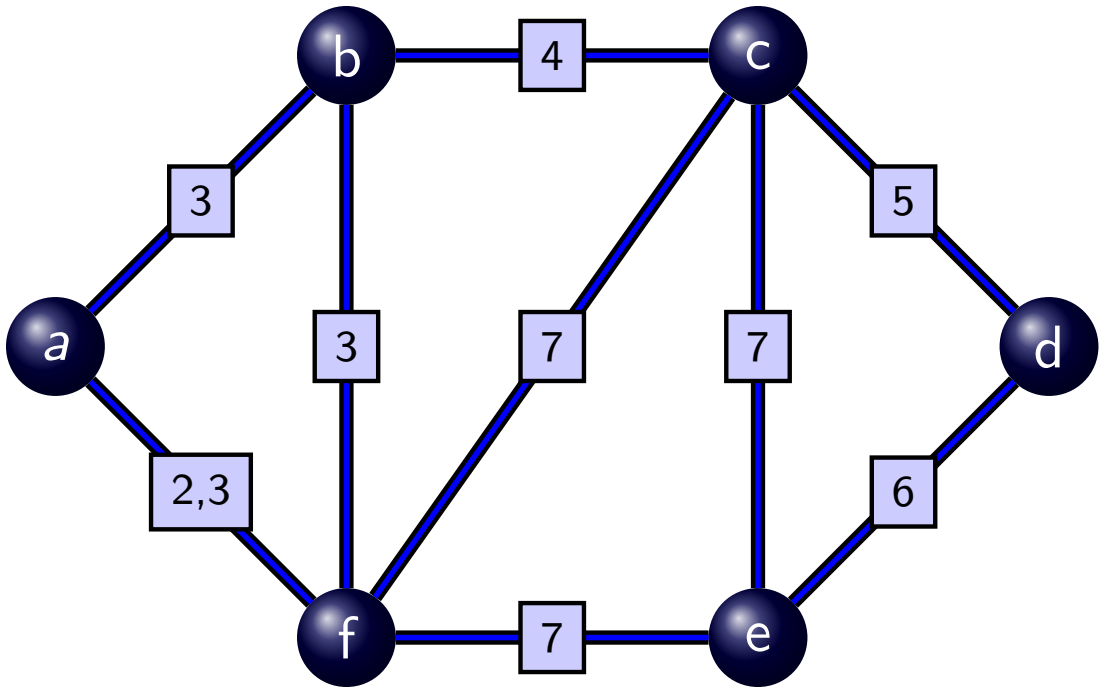
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, \ldots, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, \ldots, 7\}$



- Finding a Minimum Clique Cover is polynomial ($O(n \log n)$)
- A Minimum Clique Cover is an exact lower bound [Beldiceanu 2001]

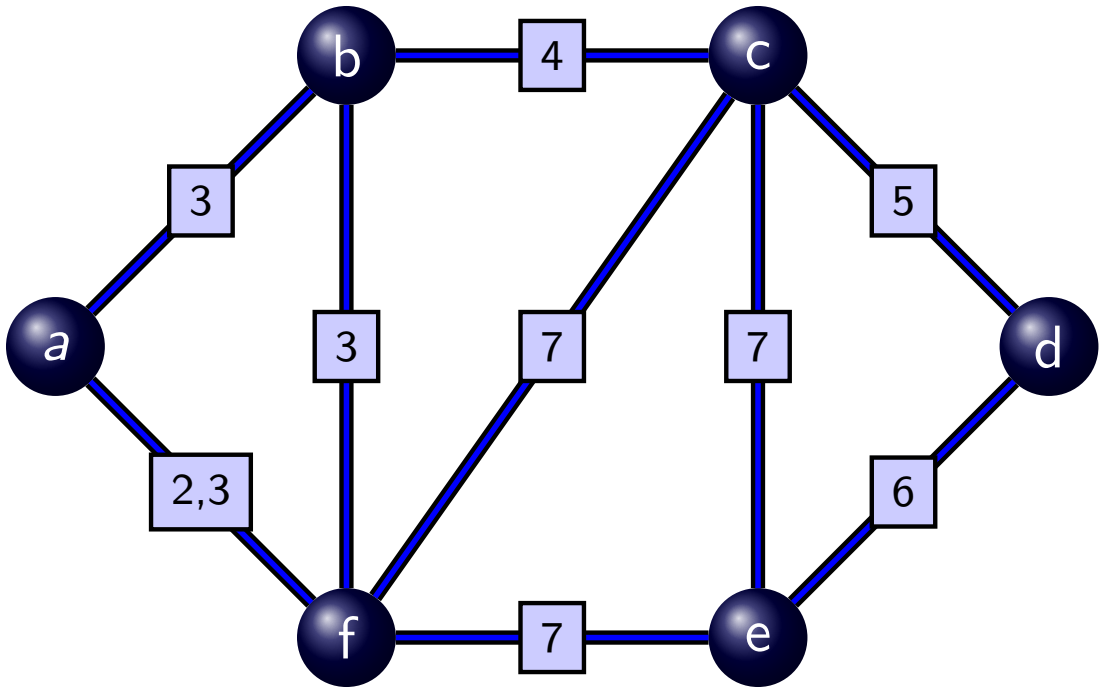
Independent Set

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



Independent Set

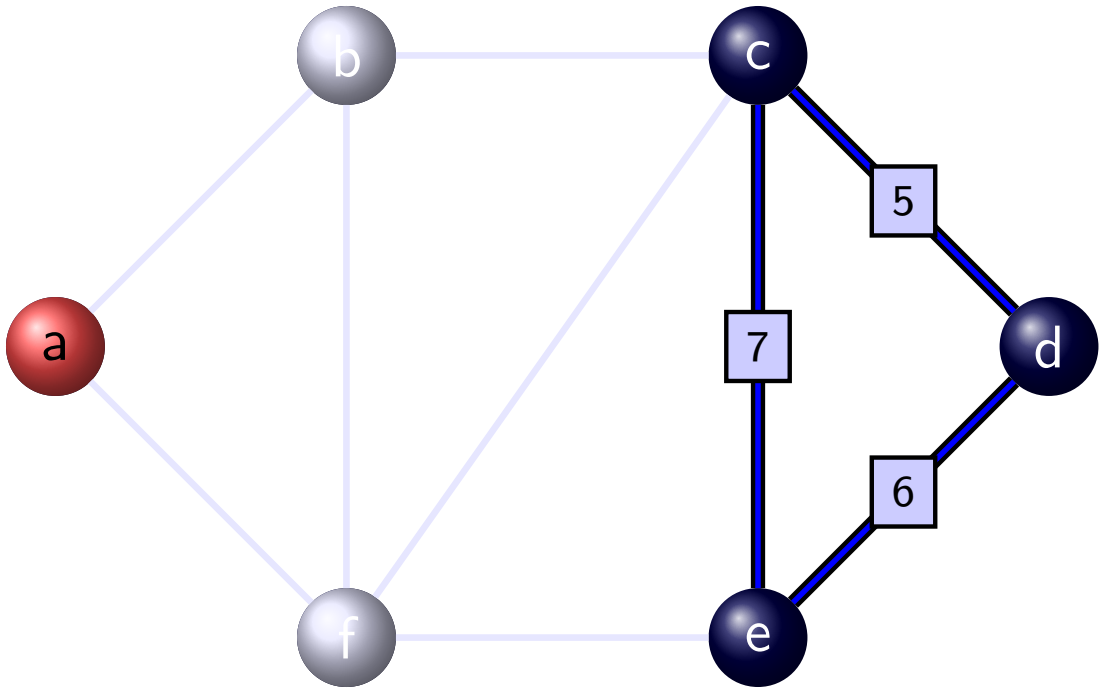
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



- Find an Independent Set
- Independance Number of $G \leq$ Intersection Number of $\theta(G)$

Independent Set

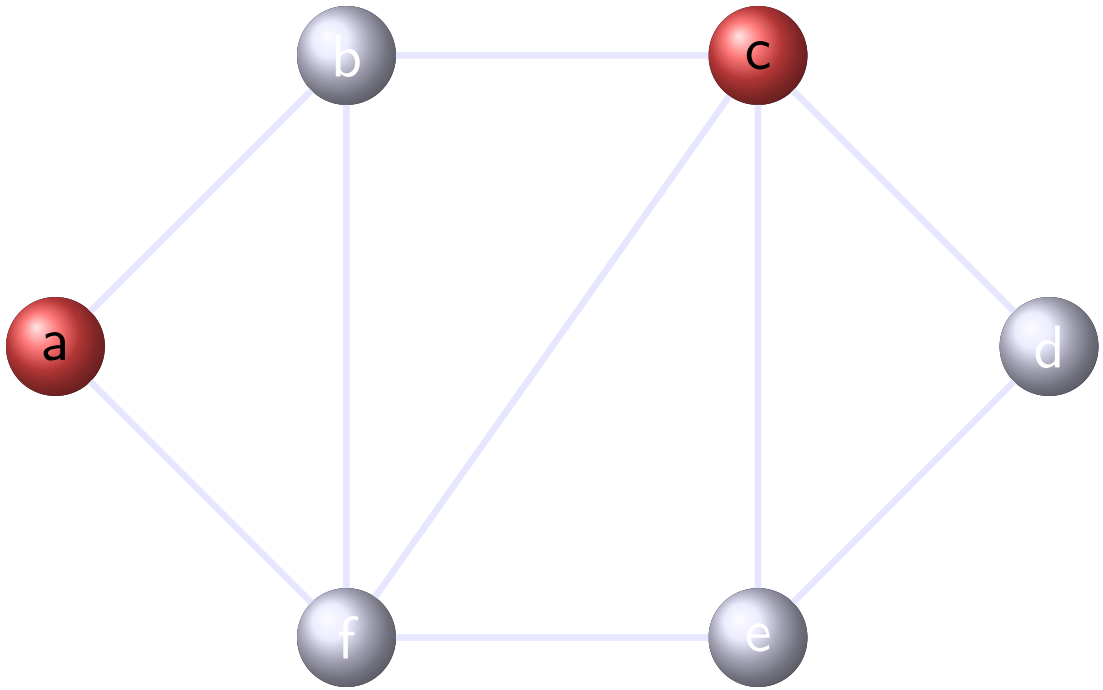
$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



- Find an Independent Set
- Independance Number of $G \leq$ Intersection Number of $\theta(G)$

Independent Set

$\mathcal{D}(a) = \{2, 3\}$
$\mathcal{D}(b) = \{3, 4\}$
$\mathcal{D}(c) = \{1, 4, 5, 7\}$
$\mathcal{D}(d) = \{5, 6\}$
$\mathcal{D}(e) = \{6, 7\}$
$\mathcal{D}(f) = \{2, 3, 7\}$



- Find an Independent Set
- Independance Number of $G \leq$ Intersection Number of $\theta(G)$

Turán's Lower Bound

- There are even simpler lower bounds
  - [Turán 1941]

$$\alpha(G) \leq \left\lceil \frac{n^2}{2m+n} \right\rceil$$

Turán's Lower Bound

- There are even simpler lower bounds
  - [Turán 1941]
  - [Favaron et al. 1993]

$$\alpha(G) \leq \left\lceil \frac{n^2}{2m+n} \right\rceil \leq \left\lceil \frac{2n - \frac{2m}{\lceil \frac{2m}{n} \rceil}}{\lceil \frac{2m}{n} \rceil + 1} \right\rceil$$

Linear Program

Linear Program

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in \mathcal{D}} y_v \\ \text{subject to} \quad & \sum_{v \in x_i} y_v \geq 1, \; x_i \in \mathcal{X} \\ & y_v \geq 0 \qquad , \; v \in \mathcal{D} \end{aligned}$$

number of values

each variable takes a value

- An integral solution is a Hitting Set

Linear Program

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in \mathcal{D}} y_v \\ \text{subject to} \quad & \sum_{v \in x_i} y_v \geq 1, \; x_i \in \mathcal{X} \\ & y_v \geq 0 \qquad , \; v \in \mathcal{D} \end{aligned}$$

number of values

each variable takes a value

- An integral solution is a Hitting Set
- The linear relaxation gives a lower bound

Tradeoff: Theory

- Which solution is best?

experiments

Tradeoff: Theory

- Which solution is best?
  - Better time complexity vs. better propagation

experiments

Tradeoff: Theory

- Which solution is best?
  - Better time complexity vs. better propagation
- Comparing propagation ϕ against ψ, constraint C domain $\mathcal{D}$

experiments

Tradeoff: Theory

- Which solution is best?
  - Better time complexity vs. better propagation
- Comparing propagation ϕ against ψ, constraint C domain $\mathcal{D}$
  - $\phi(C, \mathcal{D})$ the domain representation / set of solutions after applying ϕ
  - $\psi(C, \mathcal{D})$ the domain representation / set of solutions after applying ψ

experiments

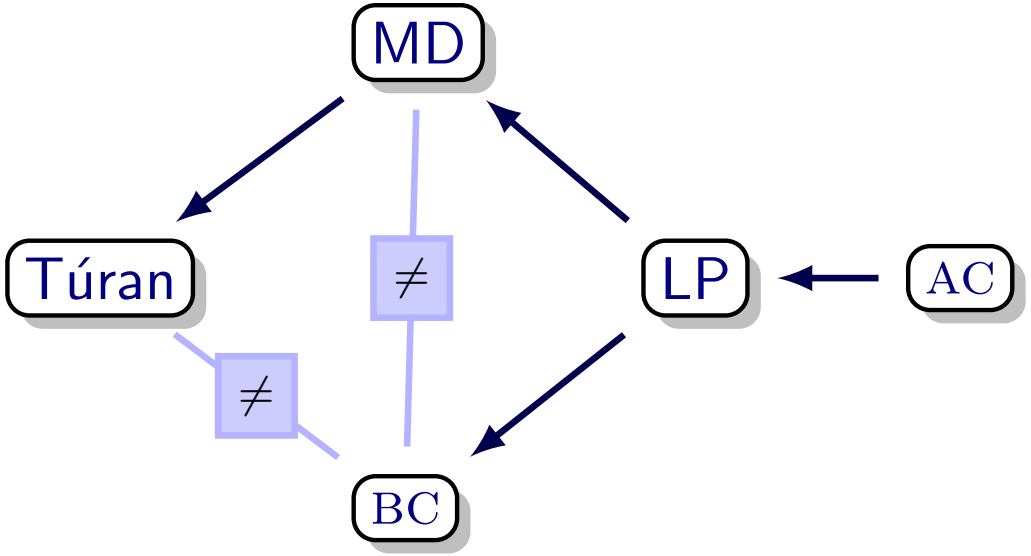
Tradeoff: Theory

- Which solution is best?
  - Better time complexity vs. better propagation
- Comparing propagation ϕ against ψ, constraint C domain $\mathcal{D}$
  - $\phi(C, \mathcal{D})$ the domain representation / set of solutions after applying ϕ
  - $\psi(C, \mathcal{D})$ the domain representation / set of solutions after applying ψ
- ϕ is as strong as ψ is $\phi(C, \mathcal{D}) \subseteq \psi(C, \mathcal{D})$

experiments

Tradeoff: Theory

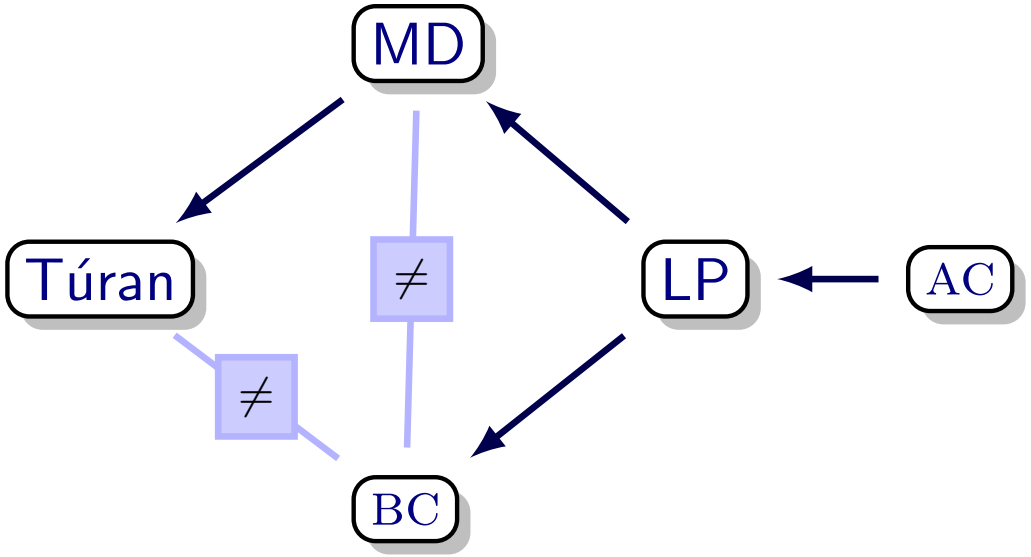
- Which solution is best?
  - Better time complexity vs. better propagation
- Comparing propagation ϕ against ψ, constraint C domain $\mathcal{D}$
  - $\phi(C, \mathcal{D})$ the domain representation / set of solutions after applying ϕ
  - $\psi(C, \mathcal{D})$ the domain representation / set of solutions after applying ψ
- ϕ is as strong as ψ is $\phi(C, \mathcal{D}) \subseteq \psi(C, \mathcal{D})$



experiments

Tradeoff: Theory

- Which solution is best?
  - Better time complexity vs. better propagation
- Comparing propagation ϕ against ψ, constraint C domain $\mathcal{D}$
  - $\phi(C, \mathcal{D})$ the domain representation / set of solutions after applying ϕ
  - $\psi(C, \mathcal{D})$ the domain representation / set of solutions after applying ψ
- ϕ is as strong as ψ is $\phi(C, \mathcal{D}) \subseteq \psi(C, \mathcal{D})$



Túran	amortised
BC	$O(n \log n)$
MD	$O(n^2 d)$
LP	$O(nd^4)$
AC	NP-hard

experiments

Conclusion

- When solving a problem: find the algorithm with best complexity

Conclusion

- When solving a problem: find the algorithm with best complexity
  - Here we also want to achieve the highest possible consistency
    - ★ AC? BC? MDD-C? (SAC, MaxRPC, ...)

Conclusion

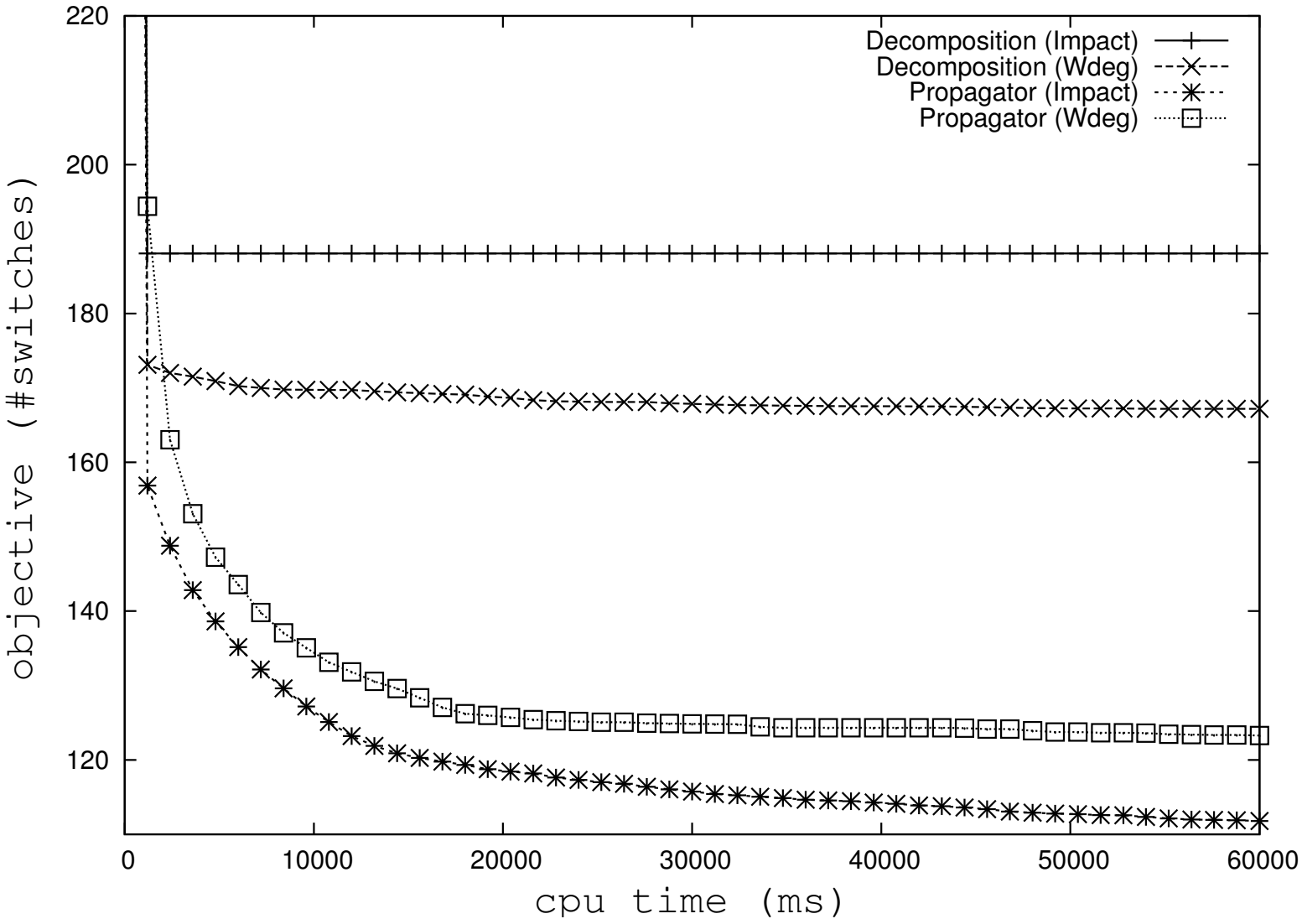
- When solving a problem: find the algorithm with best complexity
  - Here we also want to achieve the highest possible consistency
    - ★ AC? BC? MDD-C? (SAC, MaxRPC, ...)
    - ★ Even something undefined

Conclusion

- When solving a problem: find the algorithm with best complexity
  - Here we also want to achieve the highest possible consistency
    - ★ AC? BC? MDD-C? (SAC, MaxRPC, ...)
    - ★ Even something undefined
- When achieving a given consistency is NP-hard
  - Consider a lower consistency
  - Consider approximations
  - Consider parameterized complexity
- Other related questions:
  - Is it decomposable?
  - What consistency/decomposition is the strongest?
- Experimental tradeoff!

Questions?

Experimental Evaluation (120 garments, 3 machines, 7 colors)



back

Experimental results

[label=expeamsc]

Table: Car-sequencing

Models	G1 ($70 \times 34 \times 5$) 11900		G2 ($4 \times 34 \times 5$) 680		G3 ($5 \times 34 \times 5$) 850		G4 ($7 \times 34 \times 5$) 1190	
	#sol	time	#sol	time	#sol	time	#sol	time
sum	8480	13.93	95	76.60	0	> 1200	64	43.81
gsc	11218	3.60	325	110.99	31	276.06	140	56.61
amsc	10702	4.43	**360**	**72.00**	16	8.62	**153**	**33.56**
amsc+gsc	**11243**	**3.43**	339	106.53	**32**	**285.43**	147	66.45

Experimental results

[label=expeamsc]

Table: Car-sequencing

Models	G1 ($70 \times 34 \times 5$) 11900		G2 ($4 \times 34 \times 5$) 680		G3 ($5 \times 34 \times 5$) 850		G4 ($7 \times 34 \times 5$) 1190	
	#sol	time	#sol	time	#sol	time	#sol	time
sum	8480	13.93	95	76.60	0	> 1200	64	43.81
gsc	11218	3.60	325	110.99	31	276.06	140	56.61
amsc	10702	4.43	**360**	**72.00**	16	8.62	**153**	**33.56**
amsc+gsc	**11243**	**3.43**	339	106.53	**32**	**285.43**	147	66.45

- SUM: simple decomposition

Experimental results

[label=expeamsc]

Table: Car-sequencing

Models	G1 ($70 \times 34 \times 5$) 11900		G2 ($4 \times 34 \times 5$) 680		G3 ($5 \times 34 \times 5$) 850		G4 ($7 \times 34 \times 5$) 1190	
	#sol	time	#sol	time	#sol	time	#sol	time
sum	8480	13.93	95	76.60	0	> 1200	64	43.81
gsc	11218	3.60	325	110.99	31	276.06	140	56.61
amsc	10702	4.43	**360**	**72.00**	16	8.62	**153**	**33.56**
amsc+gsc	**11243**	**3.43**	339	106.53	**32**	**285.43**	147	66.45

- SUM: simple decomposition
- GSC (Global Sequencing Constraint)

Experimental results

[label=expeamsc]

Table: Car-sequencing

Models	G1 ($70 \times 34 \times 5$) 11900		G2 ($4 \times 34 \times 5$) 680		G3 ($5 \times 34 \times 5$) 850		G4 ($7 \times 34 \times 5$) 1190	
	#sol	time	#sol	time	#sol	time	#sol	time
sum	8480	13.93	95	76.60	0	> 1200	64	43.81
gsc	11218	3.60	325	110.99	31	276.06	140	56.61
amsc	10702	4.43	**360**	**72.00**	16	8.62	**153**	**33.56**
amsc+gsc	**11243**	**3.43**	339	106.53	**32**	**285.43**	147	66.45

- SUM: simple decomposition
- GSC (Global Sequencing Constraint)
  - same as ATMOSTSEQCARD for an option + demand for each type of car requiring this option
  - NP-hard, approximation

Experimental results

Table: Crew-Rostering

Heuristic	Lexicographic					
Model	satifisable (1140)			unsatisfiable (385)		
	#sol	time	avg bts	#sol	time	avg bts
sum	0	-	-	170	0.05	86
gsc	25	308.93	74074	175	2.56	262
amsc	**125**	**164.36**	**1828347**	**213**	**1.76**	**22621**
mamsc	**534**	**87.29**	**685720**	**271**	**2.80**	**27150**

Experimental results

Table: Crew-Rostering

Heuristic	Lexicographic					
Model	satifisable (1140)			unsatisfiable (385)		
	#sol	time	avg bts	#sol	time	avg bts
sum	0	-	-	170	0.05	86
gsc	25	308.93	74074	175	2.56	262
amsc	**125**	**164.36**	**1828347**	**213**	**1.76**	**22621**
mamsc	**534**	**87.29**	**685720**	**271**	**2.80**	**27150**

- MultiAtMostSeqCard: several capacity constraints together

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$

1 0 0 1 1 0 . 0 . 0 1 . . 1 1

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$

1 0 0 1 1 0 . 0 . 0 1 . . 1 1

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(7, 0) = 2$
- $card(7, 1) = 1$
- $card(7, 2) = 0$
- $card(7, 3) = 0$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(7, 0) = 2$
- $card(7, 1) = 1$
- $card(7, 2) = 0$
- $card(7, 3) = 0$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(8, 0) = 1$
- $card(8, 1) = 0$
- $card(8, 2) = 0$
- $card(8, 3) = 1$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(7, 0) = 2$
- $card(7, 1) = 1$
- $card(7, 2) = 0$
- $card(7, 3) = 0$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(7, 0) = 3 - 1 = 2$
- $card(7, 1) = 3 - 2 = 1$
- $card(7, 2) = 3 - 3 = 0$
- $card(7, 3) = 3 - 3 = 0$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(8, 0) = 3 - 2 = 1$
- $card(8, 1) = 3 - 3 = 0$
- $card(8, 2) = 3 - 3 = 0$
- $card(8, 3) = 3 - 2 = 1$

- $card(i, j) = \sum_{k=1}^{i} + c[i + j \bmod q]$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}
    - ★ increment all subsequences containing x_i

1 0 0 1 1 0 0 0 . 0 1 . . 1 1

- $card(8, 0) = 3 - 2 = 1$
- $card(8, 1) = 3 - 3 = 0$
- $card(8, 2) = 3 - 3 = 0$
- $card(8, 3) = 3 - 2 = 1$

- $card(i, j) = \sum_{k=1}^{i} + c[i + j \bmod q]$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}
    - ★ increment all subsequences containing x_i

1 0 0 1 1 0 0 0 1 0 1 . . 1 1

- $card(8, 0) = 3 - 2 = 1$
- $card(8, 1) = 3 - 3 = 0$
- $card(8, 2) = 3 - 3 = 0$
- $card(8, 3) = 3 - 2 = 1$

- $card(i, j) = \sum_{k=1}^{i} + c[i + j \bmod q]$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - access the cardinality of the jth subsequence containing x_i: $card(i, j)$
    - ★ jump from x_i to x_{i+1}
    - ★ increment all subsequences containing x_i

1 0 0 1 1 0 0 0 1 0 1 . . 1 1

- $card(8, 0) = 4 - 2 = 2$
- $card(8, 1) = 4 - 3 = 1$
- $card(8, 2) = 4 - 3 = 1$
- $card(8, 3) = 4 - 2 = 2$

- $card(i, j) = \sum_{k=1}^{i} + c[i + j \bmod q]$

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - check whether there exists a subsequence of cardinality u: $\#sub(i, k)$
    - ★ jump from x_i to x_{i+1}
    - ★ increment all subsequences containing x_i

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - check whether there exists a subsequence of cardinality u: $\#sub(i, k)$
    - ★ jump from x_i to x_{i+1}
    - ★ increment all subsequences containing x_i

0	0	0	0	0	0	3	1	2	0
						0	1	2	3

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - check whether there exists a subsequence of cardinality u: $\#sub(i, k)$
    - ★ jump from x_i to x_{i+1} $\rightarrow$ OK using $card(i, 0)$ and $card(i + 1, 4)$
    - ★ increment all subsequences containing x_i

0	0	0	0	0	0	3	1	2	0
						0	1	2	3

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - check whether there exists a subsequence of cardinality u: $\#sub(i, k)$
    - ★ jump from x_i to x_{i+1} $\rightarrow$ OK using $card(i, 0)$ and $card(i + 1, 4)$
    - ★ increment all subsequences containing x_i $\rightarrow$ decrement pointer

0	0	0	0	0	0	3	1	2	0
						0	1	2	3

back

Linear time algorithm

- We need to do each of the following in $O(1)$:
  - check whether there exists a subsequence of cardinality u: $\#sub(i, k)$
    - ★ jump from x_i to x_{i+1} $\rightarrow$ OK using $card(i, 0)$ and $card(i+1, 4)$
    - ★ increment all subsequences containing x_i $\rightarrow$ decrement pointer

0	0	0	0	0	0	3	1	2	0
					0	1	2	3	

back

Tradeoff: In Practice

- Random instances
- Dominating set of the Queens Graph of order n
  - One vertex per square of the chessboard (n^2)
  - An edge iff if the two squares are attacked (same row, column or diagonal)

Tradeoff: In Practice

- Random instances
- Dominating set of the Queens Graph of order n
  - One vertex per square of the chessboard (n^2)
  - An edge iff if the two squares are attacked (same row, column or diagonal)
- minimum number of Queens to attack the whole chessboard

problem	Túran		BC		MD		LP	
	time	bkt	time	bkt	time	bkt	time	bkt
`Random` $n = 50, d = 15$								
`Queens` (9×9)								

Tradeoff: In Practice

- Random instances
- Dominating set of the Queens Graph of order n
  - One vertex per square of the chessboard (n^2)
  - An edge iff if the two squares are attacked (same row, column or diagonal)
- minimum number of Queens to attack the whole chessboard

problem	Túran		BC		MD		LP	
	time	bkt	time	bkt	time	bkt	time	bkt
`Random` $n = 50, d = 15$	7.2	90786	6.9	72773	2.5	18885	15.8	17866
`Queens` (9×9)								

Tradeoff: In Practice

- Random instances
- Dominating set of the Queens Graph of order n
  - One vertex per square of the chessboard (n^2)
  - An edge iff if the two squares are attacked (same row, column or diagonal)
- minimum number of Queens to attack the whole chessboard

problem	Túran		BC		MD		LP	
	time	bkt	time	bkt	time	bkt	time	bkt
`Random` $n = 50, d = 15$	7.2	90786	6.9	72773	2.5	18885	15.8	17866
`Queens` (9×9)	-	-	-	-	203	880669	16	2243