Constraint Handling Rules -
My first CHR programs

Table of Contents

- My first CHR programs
  - CHR as a database language
  - Multiset transformation
  - Procedural algorithms
  - Graph-based algorithms
Overview

► Introduction of simple, yet concise and effective CHR programs
► Informal discussion of basic properties of CHR programs
  ► Anytime and online algorithm property
  ► Logical correctness
  ► Rule confluence
  ► Declarative Concurrency
  ► Worst-case time complexity

CHR as database language

► CHR can be used as information store
► CHR as deductive database
  ► Relations modeled as CHR constraints
  ► Database tuple is instance of constraint
  ► Query contains/generates tuples of database as CHR constraints
  ► Queries, views, integrity constraints formulated as CHR propagation rules
    ⇒ New data constraints (i.e. database tuples) can be deducted
Examples (III)

Example (Crossword)

- Words represented by CHR constraints
  \(\text{word}(n,o), \text{word}(d,o,g), \text{word}(b,o,o,k)\)

- Crossword problem represented as sequence of \text{word} constraints with variables as arguments
  - One variable corresponds to one field
  - Same variable for fields shared by two words

- Sequence of words as head of propagation rule
  \(\text{word}(A,B,C,D), \text{word}(E,F,G), \text{word}(A,E,H)\ldots \Rightarrow \text{solution}(A,B,C,D,E,F,G,H)\).

- \text{solution} is auxiliary CHR constraint for output

Multiset transformation

- Programs consisting of essentially one constraint
- Constraint represents active data
- Pairs of constraints rewritten by single simplification rule
- Often possible: more compact notation with simpagation rule
- Simpagation rule removes one constraint, keeps (and updates) other
Minimum I

**Minimum program**

\[
\text{min}(N) \backslash \text{min}(M) \iff N \leq M \ | \ \text{true}.
\]

- Computing minimum of multiset of numbers \(n_i\)
- Numbers given as query \(\text{min}(n_1), \text{min}(n_2), \ldots, \text{min}(n_k)\)
- \(\text{min}(n_i)\) means \(n_i\) is potential minimum
- Simpagation rule takes two \(\text{min}\) constraints and removes the one representing the larger value.
- Program continues until only one \(\text{min}\) constraint left
- This \(\text{min}\) constraint represents smallest value

Minimum II

**Minimum program**

\[
\text{min}(N) \backslash \text{min}(M) \iff N \leq M \ | \ \text{true}.
\]

- Rule corresponds to intuitive algorithm:  
  “Cross out larger numbers until one, the minimum remains”
- Illustrates use of multi-headed rule to iterate over data
  - No explicit loops or recursion needed
  - Keeps program code compact
  - Makes program easier to analyze
Example computations

Computation using top-down rule application and left-to-right goal processing order

```
Example computation
min(1), min(0), min(2), min(1)
min(0), min(2), min(1)
min(0), min(1)
min(0)
```

Computation using different order

```
Example computation
min(1), min(0), min(2), min(1)
min(1), min(0), min(1)
min(0), min(1)
min(0)
```

Program properties (I)

- Both example computations lead to same answer for the given query
- In general: answer (for given query) does not depend on order of rule applications
- This property is called **confluence**
- Rule can be applied in parallel without changing the program
  
  `Example computation (parallel)
  min(1), min(0), min(2), min(1)
  min(0), min(1)
  min(0)
  ` 
- This property is called **logical parallelism** or **declarative concurrency**
Program properties (II)

- Program is **terminating** because rule only removes constraints without adding new ones
- Number of rule applications is one less than number of min constraints
- Rule can be applied in constant time
- Given two min constraints rule can always be applied (in one order or the other)
  \[\Rightarrow\text{Complexity linear in number of min constraints}\]

Program properties (III)

- Program can be stopped at any time and intermediate answer (current store) can be observed
- Computation can be continued after that without restarting from scratch
- Intermediate results become closer and closer to final answer (fewer and fewer min constraints)
- Intermediate answers approximate final answer
- Called **anytime algorithm property**
- Makes algorithm also an **approximation algorithm**
Program properties (IV)

- Assuming \( \min \) constraint is added during computation
- Will eventually participate in computation
- Same situation as if added constraint was there from beginning but has been ignored for some time
  \( \Rightarrow \) Result will still be correct
- This property is called **online algorithm property** or **incrementality**

Operational equivalence

**Different minimum program**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \min(N) \setminus \min(M) \iff N &lt; M \mid \text{true} )</td>
<td>First rule does not remove duplicates of final minimum constraint</td>
</tr>
<tr>
<td>( \min(N) \setminus \min(M) \iff N = M \mid \text{true} )</td>
<td>Second rule only removes duplicates</td>
</tr>
<tr>
<td></td>
<td>Both rules together have same behavior as original program when working with known values</td>
</tr>
<tr>
<td></td>
<td>Query ( \min(A), \min(B), A = B ) will not reduce in new program under abstract semantics (( = ) does not imply = neither (&lt;)</td>
</tr>
<tr>
<td></td>
<td>Programs are not <strong>operationally equivalent</strong> but <strong>logically equivalent</strong></td>
</tr>
</tbody>
</table>
Greatest common divisor (I)

**XOR program**

\[ \text{gcd}(N) \setminus \text{gcd}(M) \iff 0 < N, N = M \mid \text{gcd}(M-N). \]

- Computes greatest common divisor of natural number represented as \( \text{gcd}(N) \)
- Result is remaining nonzero \( \text{gcd} \) constraint

**Example computation**

1. \( \text{gcd}(12), \text{gcd}(8) \)
2. \( \text{gcd}(8), \text{gcd}(4) \)
3. \( \text{gcd}(4), \text{gcd}(4) \)
4. \( \text{gcd}(4), \text{gcd}(0) \)

Greatest common divisor (II)

- **Condition** \( 0 < N \) in guard leads to ignoring \( \text{gcd}(0) \) constraints
- **Usability** can be improved by adding \( \text{gcd}(0) \iff \text{true} \)
- **Efficiency** can be improved by replacing subtraction with modulo operation

\[ \text{gcd}(N) \setminus \text{gcd}(M) \iff 0 < N, N = M \mid \text{gcd}(M \text{ mod } N). \]

**Example computation**

1. \( \text{gcd}(7), \text{gcd}(12) \)
2. \( \text{gcd}(7), \text{gcd}(5) \)
3. \( \text{gcd}(5), \text{gcd}(2) \)
4. \( \text{gcd}(2), \text{gcd}(1) \)
5. \( \text{gcd}(1), \text{gcd}(0) \)
6. \( \text{gcd}(1) \)
Greatest common divisor (III)

- Program also works for several \(\text{gcd}\) constraints
  - Query \(\text{gcd}(94017), \text{gcd}(1155), \text{gcd}(2035)\) results in \(\text{gcd}(11)\)
- Code also works for rational numbers (given according arithmetic operations)
- Termination is ensured for natural numbers
  - New value always smaller than \(M\)
  - New value cannot be negative due to guard
- Program is confluent when used with known numbers

Prime sieve

- Rule removes multiples of each of the numbers
- Query: Prime number candidates from 2 to up to \(N\)
  - i.e. \(\text{prime}(2), \text{prime}(3), \text{prime}(4), \ldots \text{prime}(N)\)
- Each number absorbs multiples of itself, eventually only prime numbers remain

Example computation

- \(\text{prime}(7), \text{prime}(6), \text{prime}(5), \text{prime}(4), \text{prime}(3), \text{prime}(2)\)
- \(\text{prime}(7), \text{prime}(5), \text{prime}(4), \text{prime}(3), \text{prime}(2)\)
- \(\text{prime}(7), \text{prime}(5), \text{prime}(3), \text{prime}(2)\)
Prime sieve (II)

- Rule can be seen as specialization of modulo version of gcd
  - Result of modulo operation required to be zero
- Also similar to minimum rule
  - Two numbers compared, one removed
  - But not applicable to arbitrary pairs
- Program is terminating (only removes constraints without introducing new ones)
- Anytime and online algorithm property also given

Exchange sort (I)

**Exchange sort program**

\[
a(I, V), a(J, W) \iff I > J, V < W \mid a(I, W), a(J, V).
\]

- Rule sorts array by exchanging values which are in wrong order
- Array is sequence of constraints \(a(\text{Index}, \text{Value})\)
  - i.e. \(a(1, A1), \ldots, a(n, An)\)

**Example computation**

- \(a(0, 1), a(1, 7), a(2, 5), a(3, 9), a(4, 2)\)
- \(a(0, 1), a(1, 5), a(2, 7), a(3, 2), a(4, 9)\)
- \(a(0, 1), a(1, 5), a(2, 2), a(3, 7), a(4, 9)\)
- \(a(0, 1), a(1, 2), a(2, 5), a(3, 7), a(4, 9)\)
Exchange sort (II)

- In sorted array for each pair \( a(I, V), a(J, W) \) with \( I > J \) it holds that \( V > W \)
- Rule ensures this by exchanging values if necessary
  \( \Rightarrow \) array sorted if rule not applicable anymore
- Every rule application corrects at least one ordering without introducing wrong orderings
  \( \Rightarrow \) program terminates
- Program is confluent for queries with know numbers, but not in general

Newton’s method for square roots (I)

**Square root program**

\[
\sqrt{x, G} \iff \text{abs}(G*G/X-1) > 0 \land \sqrt{x, (G+X/G)/2)}.
\]

- \( \sqrt{x, G} \) means that the square root of \( x \) is approximated by \( G \)
- Rule is based on formula \( G_{i+1} = (G_i + X/G_i)/2 \)
- Straightforward implementation because CHR programs already anytime, i.e. approximation algorithms
- Rule replaces \( \sqrt{\cdot} \) constraint with one containing next approximation
- Query is \( \sqrt{\text{GivenNumber}, \text{Guess}} \) (both numbers positive)
Newton’s method for square roots (II)

Square root program

\[
\text{sqrt}(X, G) \iff \text{abs}(G \times G / X - 1) > 0 \quad \text{or} \quad \text{sqrt}(X, (G + X / G) / 2).
\]

- Guard stops rule application if approximation is exact
- Unlikely in praxis, hence 0 should be replaced by small \( \varepsilon \)
- Demand driven version:
  \[
  \text{improve(sqrt(X))}, \quad \text{sqrt}(X, G) \iff \text{sqrt}(X, (G + X / G) / 2).
  \]
  - Approximation step only performed on demand (expressed by improve)
  - Can be extended with counter or quality check

Procedural algorithms

- More traditional style of programming used in this section
- Constraints as relations that resemble procedures
- Results returned as values bound to variables
- Fibonacci example will show CHR’s support for different programming styles
Top-down evaluation

Fibonacci top-down

\[
\begin{align*}
  f_0 & @ \text{fib}(0, M) \iff M = 1. \\
  f_1 & @ \text{fib}(1, M) \iff M = 1. \\
  f_n & @ \text{fib}(N, M) \iff N \geq 2 \mid \text{fib}(N-1, M_1), \text{fib}(N-2, M_2), \\
  & \text{M is } M_1 + M_2.
\end{align*}
\]

- \( \text{fib}(N, M) \) holds if \( M \) is \( N \)th Fibonacci number
- Recursive approach starting with highest Fibonacci number
- Also called goal-driven or backward-chaining approach
- Examples: \( \text{fib}(8, A) \) yields \( A = 34 \), \( \text{fib}(12, 233) \) succeeds, \( \text{fib}(11, 233) \) fails, and \( \text{fib}(N, 233) \) delays
- Problem: exponential complexity

Tabulation and memorization (I)

- Store and look up already computed Fibonacci numbers
- Easy implementation because CHR constraints are both data and operations
  - Turn simplification into propagation rules
  - This will keep constraints in store as data
- Rule for look-up of already computed numbers has to come first

Fibonacci with memorization

\[
\begin{align*}
  \text{mem} & @ \text{fib}(N, M_1) \setminus \text{fib}(N, M_2) \iff M_1 = M_2. \\
  f_0 & @ \text{fib}(0, M) \implies M = 1. \\
  f_1 & @ \text{fib}(1, M) \implies M = 1. \\
  f_n & @ \text{fib}(N, M) \implies N \geq 2 \mid \text{fib}(N-1, M_1), \text{fib}(N-2, M_2), \\
  & \text{M is } M_1 + M_2.
\end{align*}
\]
Tabulation and memorization (II)

- `mem` enforces functional dependency of input and output of Fibonacci relation
- Query `fib(8,A)` now returns all Fibonacci numbers up to the eighth
- Only linear complexity
  - Each Fibonacci number only computed once
  - Recursive call is only a look-up
- `mem` rule also merges two computations which have the same result

Bottom-up evaluation

- Use only data, compute larger Fibonacci numbers from smaller ones
- Starting from given fact and proceeding towards solution (also called forward-chaining or data-driven)
- Reverse head and body of top-down rules
  \[
  \text{fn } \oplus \text{fib}(N_1,M_1), \text{fib}(N_2,M_2) \Rightarrow \text{N}_2\text{:=N}_1+1 \mid \text{fib}(N_2+1,M_1+M_2).
  \]
- Include `fib(0,1), fib(1,1)` in query instead of first two rules
- Computation infinity, add rule to observe result
  \[
  \text{fib}(N,M) \Rightarrow \text{write}(\text{fib}(N,M)).
  \]
Termination

- Make computation terminating by introducing \( \text{fib–upto}(\text{Max}) \)
- Also used to introduce first two Fibonacci numbers

**Fibonacci bottom-up and terminating**

\[
\begin{align*}
\text{f01} & @ \text{fib–upto}(\text{Max}) \quad \Rightarrow \quad \text{fib}(0,1), \; \text{fib}(1,1). \\
\text{fn} & @ \text{fib–upto}(\text{Max}), \; \text{fib}(\text{N1},\text{M1}), \; \text{fib}(\text{N2},\text{M2}) \quad \Rightarrow \\
& \quad \text{Max}\geq\text{N2}, \; \text{N2}\!:=\!\text{N1}+1 \; | \; \text{fib}(\text{N2}+1,\text{M1}\!+\!\text{M2}).
\end{align*}
\]

Faster version

- Even faster version: Turn propagation rule into simpagation rule

\[
\begin{align*}
\text{fn} & @ \text{fib}(\text{Max}), \; \text{fib}(\text{N2},\text{M2}) \; \backslash \; \text{fib}(\text{N1},\text{M1}) \quad \iff \\
& \quad \text{Max}\geq\text{N2}, \; \text{N2}\!:=\!\text{N1}+1 \; | \; \text{fib}(\text{N2}+1,\text{M1}\!+\!\text{M2}).
\end{align*}
\]

- Only keeps the last two Fibonacci constraints
- Exchanged order in rule head to remove smaller Fibonacci number
Destructive assignment (I)

- Update or override of bound variables not possible in declarative programming languages
- Simulation of destructive assignment possible in CHR
  - Remove constraint with old value and add on with new value
- Optimizing compiler can translate this into in-place operation
  ⇒ Constant time for simulating destructive assignment
- This is not known for other purely declarative languages
- One source of CHR's efficiency

Destructive assignment (II)

Destructive assignment

\[ \text{assign}(\text{Var}, \text{New}), \ \text{cell}(\text{Var}, \text{Old}) \iff \text{cell}(\text{Var}, \text{New}). \]

- Storing name-value pairs as constraint \text{cell}
- Use \text{assign} constraint to update values
- Variables introduced by a \text{cell} constraint
- New cell constraint may trigger new computations (data-driven)
- Order matters for destructive assignments
  - Program containing such a rule is not confluent
  - Cannot be run in parallel without modification
- Standard first-order declarative semantics does not reflect intended meaning
Graph-based algorithms

- Algorithms working on generic class of relations: graphs
- Graph is binary relation over nodes
- Programs in this section deal with
  - Transitive closure
  - Shortest paths
  - Partial order constraints
  - Grammar parsing
  - Ordered merging and sorting

Implementation (I)

Transitive closure

\[ \text{pl} @ e(X, Y) \implies p(X, Y). \]
\[ \text{pn} @ e(X, Y), p(Y, Z) \implies p(X, Z). \]

- \( R \) implemented as edge constraint, \( R^+ \) as path constraint
- Propagation rules compute transitive closure bottom-up
- First rule adds path for every edge
- Second rule extends existing path by adding an edge in front
Implementation (II)

Transitive closure

\[
\begin{align*}
p_1 & \leftarrow e(X, Y) \implies p(X, Y). \\
p_n & \leftarrow e(X, Y), \ p(Y, Z) \implies p(X, Z).
\end{align*}
\]

- Query \( e(1, 2), e(2, 3), e(2, 4) \) adds path constraints \( p(1, 4), p(2, 4), p(1, 3), p(2, 3), p(1, 2) \)
- Query \( e(1, 2), e(2, 3), e(1, 3) \) computes \( p(1, 3) \) twice (two ways to get to node 3 from node 1)
- When distinction of edges and paths dropped program can be simplified to \( p(X, Y), \ p(Y, Z) \implies p(X, Z) \).

Termination

- Program does not terminate for cyclic graphs
- For query \( e(1, 1) \) infinitely many \( p(1, 1) \) generated by \( p_n \)
- Various compiler optimizations and options to avoid repeated generation of same constraint exists
- But program is not terminating for any implementation
Duplicate removal

- Restoring termination by removing duplicate constraints before they are used
- Enforcing set-based semantics for path constraint
- Ensures termination (finite graph contains only finite number of paths)
- Duplicate removal rule (has to come first in program)
  \[ dp \cap p(X,Y) \setminus p(X,Y) \leftrightarrow \text{true}. \]
- Implementations try to remove new constraint and keep old one

Reachability: single-source and single-target paths (I)

- Specialize transitive closure so that only paths which reach single target are computed
  \[
  \begin{align*}
  \text{target}(Y), \ e(X,Y) & \implies p(X,Y). \\
  \text{target}(Z), \ e(X,Y), \ p(Y,Z) & \implies p(X,Z).
  \end{align*}
  \]
- (Almost) analogous for source
  \[
  \begin{align*}
  \text{source}(X), \ e(X,Y) & \implies p(X,Y). \\
  \text{source}(X), \ p(X,Y), \ e(Y,Z) & \implies p(X,Z).
  \end{align*}
  \]
- Can be simplified (any path produced has same first argument)
  \[
  \begin{align*}
  p(X) \setminus p(X) & \leftrightarrow \text{true}. \\
  \text{source}(X), \ e(X,Y) & \implies p(Y). \\
  p(Y), \ e(Y,Z) & \implies p(Z).
  \end{align*}
  \]
- If source replaced with \( p \), second rule no longer needed
Shortest path (I)

```
<table>
<thead>
<tr>
<th>Shortest path</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X, Y, N) \ p(X, Y, M) &lt;=&lt; N=&lt;M</td>
</tr>
<tr>
<td>e(X, Y) ==&gt; p(X, Y, 1).</td>
</tr>
<tr>
<td>e(X, Y), p(Y, Z, N) ==&gt; p(X, Z, N+1).</td>
</tr>
</tbody>
</table>
```

- Adding argument which stores length of path
- Keeping shorter path in duplicate removal (ensures termination)
- Path propagated from edge has length 1
- Path of length \(n\) extended by edge has length \(n + 1\)

Query \(e(X, X)\) reduces to \(p(X, X, 1)\)

For query \(e(X, Y), e(Y, Z), e(X, Z)\) answer is
\(e(X, Y), e(Y, Z), e(X, Z), p(X, Z, 1), p(Y, Z, 1), p(X, Y, 1)\)

Rules can be generalized to compute shortest distance
(replace constant \(1\) by distance \(D\))

```
<table>
<thead>
<tr>
<th>Shortest path</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X, Y, N) \ p(X, Y, M) &lt;=&lt; N=&lt;M</td>
</tr>
<tr>
<td>e(X, Y, D) ==&gt; p(X, Y, D).</td>
</tr>
</tbody>
</table>
```
Partial order constraint

Partial order constraint (I)

duplicate \( @ X \leq Y \ \& \ \& X \leq Y \iff \text{true} \).

reflexivity \( @ X \leq X \iff \text{true} \).

antisymmetry \( @ X \leq Y, \ Y \leq X \iff X=Y \).

transitivity \( @ X \leq Y, \ Y \leq Z \Rightarrow X \leq Z \).

- Solver for partial order constraint \( \leq \), represented by \( \leq \).
- Implements duplicate removal and the three axioms.
- Reflexivity removes constraints matching \( X \leq X \).
- Antisymmetry replaces \( X \leq Y \) and \( Y \leq X \) by \( X=Y \).
- Reflexivity adds \( X \leq Z \) as redundant constraint.

Example

Query \( A \leq B, \ C \leq A, \ B \leq C \) leads to
\[
\begin{align*}
A & \leq B, \ C \leq A, \ B \leq C \\
A & \leq B, \ C \leq A, \ \underline{B \leq C}, \ C \leq B \\
A & \leq B, \ C \leq A, \ B=C \\
A=B, \ B=C
\end{align*}
\]

- Example shows use of propagation rule.
- Starting from circular relationship equality of variables has been proven.
Ordered merging and sorting

Notation and representation

- Directed edge (arc) from node $A$ to node $B$ represented by binary constraint $A \rightarrow B$ (infix)
- Sequences represented as chain of arcs
- Example: sequence $0, 2, 5$ encoded as $0 \rightarrow 2, 2 \rightarrow 5$

Ordered merging (I)

**Ordered merging program**

$$A \rightarrow B \setminus A \rightarrow C \implies A\lessdot B, B\lessdot C \mid B \rightarrow C.$$  

- Assuming ascendingly order chains  
  ($A \rightarrow B$ implies $A\lessdot B$, $B$ is immediate successor of $A$)
- Program merges to chains starting with same smallest value by zipping
- Given $A \rightarrow B$ and $A \rightarrow C$, arc $B \rightarrow C$ is added and now redundant arc $A \rightarrow C$ is removed
- Query $0 \rightarrow 2, 0 \rightarrow 5$ will result in $0 \rightarrow 2, 2 \rightarrow 5$
- Rule undoes transitive closure, flattens out branch in graph
Ordered merging (II)

**Ordered merging program**

\[
A \rightarrow B \ \ \ \ A \rightarrow C \leftrightarrow A < B, B < C \ \ | \ \ B \rightarrow C.
\]

- A denotes current position with branch
- All notes up to A have already been merged
- Both branches examined, smaller successor B kept, other arc replaced by B \rightarrow C
- If first chain not finished, now branch at node B
- Exhaustive rule application removes all such branches

**Example computation**

\[
\begin{align*}
0 \rightarrow 2, & \ 2 \rightarrow 5, \ 0 \rightarrow 3, \ 3 \rightarrow 7. \\
0 \rightarrow 2, & \ 2 \rightarrow 5, \ 2 \rightarrow 3, \ 3 \rightarrow 7. \\
0 \rightarrow 2, & \ 3 \rightarrow 5, \ 2 \rightarrow 3, \ 3 \rightarrow 7. \\
0 \rightarrow 2, & \ 2 \rightarrow 3, \ 3 \rightarrow 5, \ 5 \rightarrow 7.
\end{align*}
\]

**Termination**

- Rule application does not change number of arcs or node values
- Nodes on right side of arc do not change either
- Nodes on left side of arc might be replaced by larger node value
- Only finite number of values (and therefore of larger values)
  \(\Rightarrow\) Program terminates
Correctness

- Query: two ordered chains with common smallest node
- Chains share longer and longer common prefix
- Rule applications maintains following invariant
  - Set of values does not change
  - Individual arcs are ordered
  - Graph is connected
  - Each node reachable from smallest node
  - Along all paths node values in ascending order
- No branch left when rule not applicable (except duplicate arcs)
  \[ \Rightarrow \] Each node has unique immediate successor
  \[ \Rightarrow \] Each path from smallest node most be in ascending order
  \[ \Rightarrow \] Chains have been merged (only one chain which is ordered)

Sorting (I)

- Merging more than two chains simultaneously is possible
- Set of values represented as set of arcs of form \(0 \rightarrow v\)
- Sorting by merging those arcs into one chain

Example computation

\[
\begin{align*}
0 & \rightarrow 2, \quad 0 \rightarrow 5, \quad 0 \rightarrow 1, \quad 0 \rightarrow 7. \\
0 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 1, \quad 0 \rightarrow 7. \\
1 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 1, \quad 0 \rightarrow 7. \\
1 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 1, \quad 1 \rightarrow 7. \\
1 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 1, \quad 2 \rightarrow 7. \\
1 & \rightarrow 2, \quad 2 \rightarrow 5, \quad 0 \rightarrow 1, \quad 5 \rightarrow 7.
\end{align*}
\]