PART ONE

Introduction
shortest path (by car) according to Google
how to find the shortest path ???

how to find the shortest path ???

Edsger Dijkstra (1930-2002)
Dutch computer scientist

Dijkstra's algorithm:
1. distance(start-point) = 0
2. pick a (not-yet-considered) point x
   with smallest distance. LABEL(x)
3. if end-point is considered, stop;
   otherwise go to step 2

LABEL(x): for all arrows \( x \rightarrow y \):
set distance(y) = distance(x) + a
(If the new distance is shorter)

Edsger Dijkstra (1930-2002)
Dutch computer scientist

How to do this automatically ?
Implementing Dijkstra's algorithm

...in machine code

Implementing Dijkstra's algorithm

...in assembly

Implementing Dijkstra's algorithm

...in C

Implementing Dijkstra's algorithm

...in Java
CHR is a very high level programming language based on rules.

- propagation rules:
  - clouds ⇒ forecast(rainy).
  - forecast(rainy) ⇒ bring(coat).
  - forecast(sunny) ⇒ bring(sunscreen).

- simplification rules:
  - bring(coat), bring(sunscreen) ⇒ bring(umbrella).

CHR = Constraint Handling Rules

stand-alone (CHR-only) or extending a host language
Compilation

Syntax of CHR

- Propagation rule:
  \[ \text{head} \Rightarrow \text{guard} \mid \text{body}. \]
  Example: \[ \text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L). \]

- Simplification rule:
  \[ \text{head} \Leftrightarrow \text{guard} \mid \text{body}. \]
  Example: \[ \text{dist}(A,X), \text{dist}(A,Y) \Leftrightarrow X \leq Y \mid \text{dist}(A,X). \]

Operational semantics of CHR

**IF head IN STORE (AND guard HOLDS), THEN...**

- Propagation rule: \[ \text{ADD body TO STORE} \]
  \[ \text{head} \Rightarrow \text{guard} \mid \text{body}. \]
  Example: \[ \text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L). \]

- Simplification rule: \[ \text{REPLACE head BY body} \]
  \[ \text{head} \Leftrightarrow \text{guard} \mid \text{body}. \]
  Example: \[ \text{dist}(A,X), \text{dist}(A,Y) \Leftrightarrow X \leq Y \mid \text{dist}(A,X). \]
Features of CHR

**Embedded in a host language**

CHR extends an existing programming language, e.g.
CHR(Prolog)
CHR(Haskell)
CHR(Java)
CHR(C)

- **Simplification rule:**
  - \[ \text{head} \iff \text{guard} \land \text{body}. \]
  - example: \( \text{dist}(A,X), \text{dist}(A,Y) \iff X \leq Y \land \text{dist}(A,X). \)

**Propagation rule:**

- \[ \text{head} \Rightarrow \text{guard} \land \text{body}. \]
- example: \( \text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L). \)

**Multiple heads**

The head of a rule consists of an arbitrary number of CHR constraints (1 or more)

- cf. Prolog: single-headed

**Important remark:**

in CHR(Prolog), we can still use Prolog disjunction or nondeterministic predicates in the body of rules!

**Multi-set semantics**

The constraint store may contain the same constraint multiple times (c) is not the same as (c,c)
- cf. classical logic: \( p \iff p \land p \)

- **Committed-choice**

Once a rule has been applied, it remains applied - no backtracking to try different derivation paths
- cf. Prolog: choice-points and backtracking

- **Simplification rule**
  - \[ \text{head} \iff \text{guard} \land \text{body}. \]
  - example: \( \text{dist}(A,X), \text{dist}(A,Y) \iff X \leq Y \land \text{dist}(A,X). \)
Features of CHR

- **Propagation rule:**
  \[ \text{head} \Rightarrow \text{guard} \land \text{body}. \]
  
  Example: \( \text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L) \).  

- **Simplification rule:**
  \[ \text{head} \Leftrightarrow \text{guard} \land \text{body}. \]
  
  Example: \( \text{dist}(A,X), \text{dist}(A,Y) \Leftrightarrow X \leq Y \land \text{dist}(A,X) \).

**Logical semantics**

CHR has a declarative semantics!

PART TWO

Writing CHR programs

CHR(Prolog) by example

- Simple example: color mixing in CHR
- We first declare CHR constraints as follows:
  \[ :- \text{chr}\_constraint \text{red}, \text{blue}, \text{yellow}, \text{purple}, \ldots \]
- Then we write the rules:
  \[
  \begin{align*}
  &\text{red}, \text{blue} \Leftrightarrow \text{purple}. \\
  &\text{blue}, \text{yellow} \Leftrightarrow \text{green}. \\
  &\text{yellow}, \text{red} \Leftrightarrow \text{orange}.
  \end{align*}
  \]

CHR(Prolog) by example

- Simple example: color mixing in CHR
- rules are applied exhaustively
- the remaining constraints are the result
CHR(Prolog) by example

- Simple example: color mixing
  red, blue <=> purple.
  blue, yellow <=> green.
  yellow, red <=> orange.

- Example interaction:
  ?- blue, red.
  purple
  ?- yellow, blue, red.
  green
  red

Why this answer? (and not, say, "yellow, purple")

Refined semantics
Execution from left to right and from top to bottom (cf. Prolog)

Confluence

- Simple example: color mixing in CHR
  r1 @ red, blue --> purple.
  r2 @ blue, yellow --> green.
  r3 @ yellow, red --> orange.

- Example interaction:
  ?- yellow, blue, red.

Why this answer? (and not, say, "yellow, purple")

Refined semantics
Execution from left to right and from top to bottom (cf. Prolog)

Abstract semantics
allows rule application in any order

Confluence

A CHR program is called confluent if for any given goal, there is only one result, regardless of the order in which rules are applied. (so the color mixing program is not confluent)

Abstract semantics
allows rule application in any order

Constraints with arguments

- Add anything to brown and it remains brown:
  red, blue <=> purple.
  blue, yellow <=> green.
  yellow, red <=> orange.
  brown, red <=> brown.
  brown, blue <=> brown.
  brown, yellow <=> brown.
  brown, purple <=> brown.
  ...

Constraints with arguments
This becomes a bit tedious, can't we write something like this instead?

\[ \text{brown, } _ <\Rightarrow\text{ brown.} \]

The above will not work in CHR (but it does work in a related formalism called ACD term rewriting)

But we can write our program in a different way...

From many 0-ary constraints to one unary constraint:

\[
\text{:- chr_constraint red, blue, yellow, purple, ... } \\
\text{red, blue } \Rightarrow \text{ purple. } \\
\text{blue, yellow } \Rightarrow \text{ green. } \\
\text{yellow, red } \Rightarrow \text{ orange. } \\
\text{:- chr_constraint color/1. } \\
\text{color(red), color(blue) } \Rightarrow \text{ color(purple). } \\
\ldots
\]

Now we can write more general rules:

\[
\text{:- chr_constraint color/1. } \\
\text{color(X), color(Y) } \Rightarrow \text{ mix(X, Y, Z) } \mid \text{ color(Z). } \\
\text{color(brown), color(_) } \Rightarrow \text{ color(brown).} \\
\]

Optionally, we can specify types and modes:

\[
\text{% no type/mode declaration: } \\
\text{:- chr_constraint color/1. } \\
\text{% only mode declaration: } \\
\text{:- chr_constraint color(+). } \quad \text{% ground argument} \\
\text{% type and mode declaration: } \\
\text{:- chr_constraint color(+colormap). } \\
\text{:- chr_type colormap } \Rightarrow \text{ red ; blue ; yellow ;...}
\]
Simpagation rules

- So far we have only used simplification rules.
- Simpagation rules can be more concise/efficient:

  % simplification rule:
  color(brown), color(_) <=> color(brown).

  "true"?
  "In Prolog, "true" is a built-in that does not do
  anything. We use it to indicate an empty body.

  % simpagation rule:
  color(brown) \ color(_) <=> true.

Typical pattern #1: flattening lists

- We want to convert "colors([red,green,blue])" to "color(red), color(green), color(blue)"

  :- chr_constraint color(+colorname).
  :- chr_type colorname ---> red ; blue ; yellow ;... 
  :- chr_constraint colors(+list(colorname)).
  :- chr_type list(T) ---> [ ] ; [T|list(T)].

  colors([]) <=> true.
  colors([C|Rest]) <=> color(C), colors(Rest).

  (just like how you would do this in Prolog)

More complex color mixing

- Now we also specify the amount of paint:

  :- chr_constraint color(+colorname,+amount).
  :- chr_type colorname ---> red ; blue ; yellow ;...
  :- chr_type amount == float. % in liters

  ("float" is a built-in type for floating point numbers)

Typical pattern #2: “default constructor”

- For backwards compatibility, we still have color/1

  :- chr_constraint color(+colorname).
  :- chr_constraint color(+colorname,+amount).
  :- chr_type colorname ---> red ; blue ; yellow ;...
  :- chr_type amount == float.

  % we assume 1 liter of paint:
  color(C) <=> color(C,1).
Typical pattern #3: maintaining a sum

```prolog
:- chr_constraint color(+colorname,+amount).

color(C,A1), color(C,A2)
  <=> TA is A1+A2, color(C,TA).

color(C,0) <=> true.
color(X,A1), color(Y,A2)
  <=> mix(X,Y,Z) | TA is A1+A2, color(Z,TA).
```

CHR(Prolog) one-liners (1)

- Finding the minimum:
  ```prolog
  min(A) \ min(B) <=> A =< B | true.
  ?- min(8), min(3), min(6), min(7).
  min(3)
  ```

- Computing the sum:
  ```prolog
  sum(A), sum(B) <=> C is A+B, sum(C).
  ?- sum(3), sum(5), sum(6).
  sum(14)
  ```

CHR(Prolog) one-liners (2)

- Transitive closure
  ```prolog
  :- op(700,xfx,before).
  :- chr_constraint before(+any,+any).
  A before B, B before C ==> A before C.
  ?- a before b, b before c, c before d.
  a before d
  b before d
  ```
CHR(Prolog) one-liners (3)

- Naive merge-sort in $O(n^2)$ time

```prolog
:- op(700,xfx,before).
:- chr_constraint before(+any,+any).
A before B \ A before C <= B ¥ C | B before C.
?- 0 before foo, 0 before bar, 0 before baz, 0 before quux.
  0 before bar
  bar before baz
  baz before foo
  foo before quux
```

CHR(Prolog) two-liners (1)

- Greatest common divisor (Euclid’s algorithm)

```prolog
:- chr_constraint gcd(+int).
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N ¥ M | L is M mod N, gcd(L).
?- gcd(94017), gcd(1155), gcd(2035).
gcd(1)
```

CHR(Prolog) two-liners (2)

- Prime number generator (sieve of Eratosthenes)

```prolog
:- chr_constraint prime(+int).
prime(N) ==> N>2 | M is N-1, prime(M).
prime(A) \ prime(B) <=> B mod A =:= 0 | true.
?- prime(10).
  prime(2)
  prime(3)
  prime(5)
  prime(7)
```

CHR(Prolog) two-liners (3)

- Fibonacci numbers

```prolog
:- chr_constraint fib(+int,+int), upto(+int).
upto(_) => fib(0,1), fib(1,1).
upto(Max), fib(N1,M1), fib(N2,M2) => Max>N2, N2 is N1+1 | N is N2+1, M is M1+M2, fib(N,M).
?- upto(10).
  fib(0,1)
  fib(1,1)
  fib(2,3)
  fib(3,5)
  fib(4,8)
  fib(5,13)
  fib(6,21)
  fib(7,34)
  fib(8,55)
  fib(9,89)
  fib(10,144)
```
CHR(Prolog) two-liners (4)

- Optimal merge-sort

\[ \text{op}(780, 	ext{xfr}, 	ext{before}) \]
\[ \text{op}(	ext{chr}, 	ext{constraint}, 	ext{before} (+\text{any}, +\text{any}) \Rightarrow \text{sort} (+\text{int}, +\text{any}) \Rightarrow \text{X before A} \land \text{X before B} \Rightarrow \text{A} \lessdot \text{B} | \text{A before B}. \]

\[ \text{sort}(\text{N}, \text{A}), \text{sort}(\text{N}, \text{B}) \Rightarrow \text{A} \lessdot \text{B} | \text{M is N}+1, \text{sort}(\text{M}, \text{A}), \text{A before B}. \]

?- \text{sort}(0, \text{foo}), \text{sort}(0, \text{bar}), \text{sort}(0, \text{baz}), \text{sort}(0, \text{quux}).
\text{bar before bar}
\text{bar before foo}
\text{foo before quux}
\text{sort(1, bar)}

One last example...

- Simple less-than-or-equal constraint solver

\[ \text{op}(700, \text{xfr}, \text{leq}) \]
\[ \text{op}(	ext{chr}, \text{constraint}, \text{leq}/2). \]

\[ \text{reflexivity} @ \text{X leq X} \Rightarrow \text{true}. \]
\[ \text{idempotence} @ \text{X leq Y} \land \text{X leq Y} \Rightarrow \text{true}. \]
\[ \text{antisymmetry} @ \text{X leq Y}, \text{Y leq X} \Rightarrow \text{X=Y}. \]
\[ \text{transitivity} @ \text{X leq Y}, \text{Y leq Z} \Rightarrow \text{X leq Z}. \]

?- \text{A leq B}, \text{B leq C}, \text{C leq A}.
\text{A = B}
\text{B = C}

CHR(Prolog) two-liners (5)

- Soduko puzzle solver in CHR

\[ \text{given}(\text{P}, \text{V}) \land \text{maybe}(\text{P}, \text{L}) \]
\[ \Leftrightarrow \text{sees}(\text{P}, \text{P2}), \text{select}(\text{V}, \text{L}, \text{L2}) \mid \text{maybe}(\text{P2}, \text{L2}). \]
\[ \text{maybe}(\text{P}, \text{L}) \Leftrightarrow \text{member}(\text{V}, \text{L}), \text{given}(\text{P}, \text{V}). \]

?- \text{given}(1,1,5), \text{given}(1,2,3), ..., \text{maybe}(1,3, \{1, 2, 3, ..., 9\}), ...
\text{given}(a,1,5)
\text{given}(a,2,3)
\text{given}(a,3,7)
...

Differences between CHR and Prolog

\begin{tabular}{|c|c|c|}
\hline
\textbf{basic elements} & \textbf{Prolog} & \textbf{CHR} \\
\hline
\text{predicates} & \text{constraints} \\
\hline
\text{rules} & \text{coordinates} \\
\hline
\text{matching + guard} & \text{unification} \\
\hline
\text{committed-choice} & \text{failure} \\
\hline
\text{partial result} & \text{suspension (delay)} \\
\hline
\end{tabular}
Committed-choice – different from Prolog!

- In Prolog, **backtracking** (proof search) is used to find a non-failing derivation.
- In CHR, there is no backtracking.

```
?- chr_constraint chr/0, output/1.
chr <=> output(foo).
prolog :- output(foo).
prolog :- output(bar).
```

Head matching – different from Prolog!

- In Prolog, **unification** is used to match clause heads.
- In CHR, **matching** (one-way unification) is used.

```
?- chr_constraint chr/1, output/1.
chr(foo) <=> output(bar).
prolog(foo) :- output(bar).
```

### History of CHR: some milestones

- **1991** CHR is born, Thom Frühwirth
- **1995** Christian Holzbaur implements CHR(SICStus)
- **1998** Confluence, program analysis (PhD Slim Abdennadher)
- **2002** Tom Schrijvers implements Leuven CHR system
- **2002** Optimized compilation (PhDs Gregory Duck, Tom Schrijvers)
- **2003** First CHR book (Frühwirth & Abdennadher, Essentials of Constraint Programming)
- **2004** Refined semantics, Gregory Duck et al.
- **2004** First CHR workshop
- **2005** Computational complexity (PhD Jon Sneyers)
- **2005** Peter Van Weert implements Leuven JCHR (Java)
- **2007** Sulzmann & Lam implement first concurrent system
- **2009** Second CHR book, sixth CHR workshop

### PART THREE

Theory & Applications
Theory topics (1)

- Semantics
  - Declarative (logical) semantics
    - Classical logic (Frühwirth)
    - Linear logic (Hariolf Betz)
    - Transaction logic, ...
  - Compositional semantics (Gabbrielli et al)
- Operational semantics
  - Abstract semantics
  - Refined semantics (Duck et al)
  - Priority semantics (Leslie De Koninck)

Theory topics (2)

- Extensions / variants of CHR
  - Adaptive CHR (Armin Wolf)
  - Disjunction, search (Abdennadher, Wolf, De Koninck, …)
  - Negation, aggregates (Van Weert & Sneyers, …)
  - Modularity, solver hierarchies (Duck et al, Schrijvers et al, Fages et al)
  - Probabilistic CHR (Frühwirth et al, Sneyers et al)
  - ...

Theory topics (3)

- Relationship to other formalisms
  - Term rewriting (ACD term rewriting, Duck, Stuckey et al)
  - Production rules / business rules (Van Weert)
  - Join-Calculus (Sulzmann and Lam)
  - Logical Algorithms (De Koninck)
  - Graph Transformation Systems (Raifer)
  - Petri nets (Betz)
  - ...
Theory topics (4)

- Program analysis
  - Confluence (Abdennadher, Duck et al., Raiser&Tacchella, Haemmerl&Fages, …)
  - Operational equivalence (Abdennadher&Frühwirth)
  - Termination (Frühwirth, Paolo Pizzi, Dean Voets)
  - Complexity (Frühwirth&Schiëlers, Sneyers, De Koning)
- Abstract interpretation (Schiëlers, Stuckey, Duck)
- …

Computational Complexity Theory

- How does an algorithm scale with the input size?

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Algorithm A</th>
<th>Algorithm B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leuven 5000</td>
<td>2 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>Brussels 50000</td>
<td>23 ms</td>
<td>2.5 seconds</td>
</tr>
<tr>
<td>New York City 277863</td>
<td>151 ms</td>
<td>1 min 17 seconds</td>
</tr>
<tr>
<td>Florida 1228116</td>
<td>747 ms</td>
<td>25 min, 8 seconds</td>
</tr>
<tr>
<td>North America 29883886</td>
<td>22 seconds</td>
<td>10 days, 8 hours, 4 min</td>
</tr>
</tbody>
</table>

Back to the shortest path problem...

- How long does it take?
  - It depends...
    - which algorithm is used?
    - how is it implemented?
    - how large is the map (graph)?

What about Dijkstra’s algorithm?

- Dijkstra’s algorithm is $O(n \log n)$
  - for sparse graphs (in general: $O(m + n \log n)$)
  - if implemented in a good way, e.g. using Fibonacci-heaps
- This is optimal: you cannot do better
- Dijkstra’s algorithm can be implemented in CHR (with the optimal complexity)
Some other examples...

- Dijkstra's algorithm can be implemented efficiently in CHR
- The Union-Find algorithm can be implemented efficiently in CHR
- Hopcroft's algorithm can be implemented efficiently in CHR

...can everything be implemented efficiently in CHR?

Can we implement everything efficiently in CHR?

Yes we can!

Complexity-wise completeness result for CHR

More information: my talk on Thursday

Application domains

- Constraint solvers
  - CHR was specifically designed for this
  - Some domains where CHR has been used:
    - Scheduling
    - Soft constraints
    - Spatio-temporal reasoning
    - Multi-agent systems
    - Semantic web
- General-purpose programming language
  - Many classical algorithms have been implemented in CHR in a very elegant and natural way - often more concise than pseudocode!

Application domains

- Programming language development
  - Type systems (e.g. Haskell type classes)
  - Abductive reasoning
  - Computational linguistics (NLP)
    - CHR Grammars (Dahl & Christiansen)
  - Meta-programming
  - Testing & verification
- CHR can be used as a high-performance business rule engine (integrated in your favorite host language)
Further reading...

