1. Conflicts & Trust mappings in Community DBs

Background 1: Conflicting beliefs

```
  1. Bill
  2. Charlie
  3. Alice

Belief: annotated (has value) pairs
```

Priorities:

```
- 100: Prioritize Alice over Bill over Charlie
- 90: Prioritize Alice over Bill
- 10: Prefer Bill over Charlie
- 5: Prefer Alice
```

*Replicas are selected based on community databases:

- Includes (Haggle, VLDB’07)
- Examples (VLDB’09, Beliefs’VLDB’09)

Recent work on community databases:

How to unambiguously assign beliefs to all users?

2. Stable solutions

- Priority trust network (TN)
  - users: A, B, C
  - weights: w, w, w
  - trust mappings (pairs) from “parents”

- Stable solution
  - assignment of values to each node, s.t. each belief has a “non-dominated” lineage to an explicit belief

- Possible / Certain semantics
  - a stable solution determines, for each node, a possible value (“pos”) and certain value (“cert”) = intersection of all stable solutions, per user

```
A: [1, 1, 1]
B: [1, 1, 1]
C: [1, 1, 1]
```

3. Logic programs with stable model semantics

Step 1: Binarization

```
A  B  C  D
10  10  10  10
```

Step 2: Logic program

```
1: accept all poss of preferred parent
2: accept poss from non-preferred parent, that are not conflicting with an existing value
```

4. Resolution Algorithm (1/2)

- Keep 2-ary closed/open
- Inititalize closed with explicit beliefs

```
Step 1: From preferential edges from open to closed:
  - follow

X: pos(x) cert(x)
A (x)
B (x) (x)
C (x) (x)
D (x) (x)
E (x) (x)
```

5. Resolution Algorithm (2/2)

- Keep 2-ary closed/open
- Inititalize closed with explicit beliefs

```
Step 2: From preferential edges from open to closed:
  - follow

X: pos(x) cert(x)
A (x)
B (x) (x)
C (x) (x)
D (x) (x)
E (x) (x)
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```