EECS 647 - BCNF Implementation

Dain Vermaak
9/22/2015

Forward

This algorithm is intended for reproduction and sacrifices adherence to many accepted coding standards in exchange for compactness and clarity. This implementation represents a single approach to BCNF decomposition and ignores many optimizations in favor of simplicity.

Table of Contents

BCNF

BCNF.cpp

```
#include "BCNFTester.h"

int main() {
        BCNFTester* pTester = new BCNFTester();
        pTester->RunTests(new BCNFCalculator());
        std::string pWait;
        std::cin >> pWait;
        return 0;
}
```

BCNFTester

BCNFTester.h

```
#pragma once
#include "BCNFCalculator.h"

class BCNFTester {
public:
        BCNFTester() {};
        ~BCNFTester() {};

        void RunTests(BCNFCalculator* calc);
        void RunSingleTest(BCNFCalculator* calc, Set* R, FDSet S, std::vector<Set*>& expected);
        void RunSingleTest(BCNFCalculator* calc, Set* R, FDSet S, std::vector<Set*>& expected, std::string title);
        void TestA(BCNFCalculator* calc);
        void TestB(BCNFCalculator* calc);
        void TestWikipedia(BCNFCalculator* calc);
}
```

BCNFTester.cpp

```
#include "BCNFTester.h"
void BCNFTester::RunTests(BCNFCalculator* calc) {
        TestA(calc);
        TestB(calc);
        TestWikipedia(calc);
}

void BCNFTester::TestA(BCNFCalculator* calc) {
        Set* pSet = new Set("EID PID Ename Pname Hours");
        std::vector<FunctionalDependency*> pS;
        pS.push_back(new FunctionalDependency("EID --> Ename"));
        pS.push_back(new FunctionalDependency("PID --> Pname"));
        pS.push_back(new FunctionalDependency("EID PID --> Hours"));
        std::vector<Set*> pExpectedRelations;
        pExpectedRelations.push_back(new Set("EID Ename"));
        pExpectedRelations.push_back(new Set("PID Pname"));
        pExpectedRelations.push_back(new Set("EID PID Hours"));
        RunSingleTest(calc, pSet, pS, pExpectedRelations, "Test A");
}

void BCNFTester::TestB(BCNFCalculator* calc) {
        Set* pSet = new Set("Property_ID County_Name Lot# Area Price Tax_Rate");
        std::vector<FunctionalDependency*> pS;
        pS.push_back(new FunctionalDependency("Property_ID --> County_Name Lot# Area Price Tax_Rate"));
        pS.push_back(new FunctionalDependency("County_Name Lot# --> Property_ID Area Price Tax_Rate"));
        pS.push_back(new FunctionalDependency("County_Name --> Tax_Rate"));
        pS.push_back(new FunctionalDependency("Area --> Price"));
        std::vector<Set*> pExpectedRelations;
        pExpectedRelations.push_back(new Set("County_Name Tax_Rate"));
        pExpectedRelations.push_back(new Set("Area Price"));
        pExpectedRelations.push_back(new Set("Property_ID Area County_Name Lot#"));
        RunSingleTest(calc, pSet, pS, pExpectedRelations, "Test B");
}
```

```
void BCNFTester::TestWikipedia(BCNFCalculator* calc) {
        Set* pSet = new Set("Court StartTime EndTime RateType Member");
        std::vector<FunctionalDependency*> pS;
        pS.push_back(new FunctionalDependency("Member Court --> RateType"));
        std::vector<Set*> pExpectedRelations;
        pExpectedRelations.push_back(new Set("RateType Court Member"));
        pExpectedRelations.push_back(new Set("Member Court StartTime EndTime"));
        RunSingleTest(calc, pSet, pS, pExpectedRelations, "Test Wikipedia");
}


void BCNFTester::RunSingleTest(BCNFCalculator* calc, Set* R, FDSet S, std::vector<Set*>& expected, std::string title) {
        std::cout << "\n===========================\nTitle: " << title << "\n";
        RunSingleTest(calc, R, S, expected);
}

void BCNFTester::RunSingleTest(BCNFCalculator* calc, Set* R, FDSet S, std::vector<Set*>& expected) {
        std::vector<Set*> pFinalRelations = calc->BCNF(R, S);
        if (pFinalRelations.size() != expected.size()) {
                std::cout << "Invalid size: " << pFinalRelations.size();
                std::cout << ", expected " << (int)expected.size() << " relations.\n\n";
                for (size_t i = 0; i < pFinalRelations.size(); i++) {
                        std::cout << "\tInvalid Relation: " << pFinalRelations[i]->ToString() << "\n";
                }
                return;
        }

        std::string pRelationToString;
        for (size_t i = 0; i < pFinalRelations.size(); i++) {
                bool isExpected = false;
                for (size_t j = 0; j < expected.size(); j++) {
                        if (pFinalRelations[i]->IsEqual(expected[j])) {
                                isExpected = true;
                                expected.erase(expected.begin() + j);
                                break;
                        }
                }
                if (!isExpected) {
                        std::cout << "Invalid Relation: " << pFinalRelations[i]->ToString() << "\n";
                        return;
                }
        }

        std::cout << "Test Passed!\nOutput: \n";
        for (size_t i = 0; i < pFinalRelations.size(); i++) {
                std::cout << " > " << pFinalRelations[i]->ToString() << "\n";
        }
}
```

BCNFCalculator

BCNFCalculator.h

```
#pragma once
#include <bitset>
#include <iostream>
#include "Set.h"
#include "FunctionalDependency.h"
class BCNFCalculator{
public:
        BCNFCalculator() {};
        ~BCNFCalculator() {};
        std::vector<Set*> BCNF(Set* R, FDSet S);
        Set* Closure(Set* Q, FDSet S);
private:
        bool IsKey(Set* possibleKey, Set* relation);
        void Split(Set* R, FDSet S, Set*& R1, Set*& R2, FDSet& S1, FDSet& S2, Set* problemClosure);
        void SplitFDs(Set* R, Set* R1, FDSet S, FDSet& S1);
        std::vector<Set*> GetAllSubsets(Set* S);
};
```

BCNFCalculator.cpp

```
#include "BCNFCalculator.h"

std::vector<Set*> BCNFCalculator::BCNF(Set* R, FDSet S) {
        if (S.size() > 0) {
                for (size_t i = 0; i < S.size(); i++) {

                        Set* pAs = S[i]->GetA();
                        Set* pClosure = Closure(pAs, S);
                        if (!IsKey(pClosure, R)) {
                                // Define the variables for the split
                                Set* R1;
                                Set* R2;
                                std::vector<FunctionalDependency*> S1;
                                std::vector<FunctionalDependency*> S2;

                                // Split the relation and the functional deps
                                Split(R, S, R1, R2, S1, S2, pAs);

                                // Recursively call BCNF on the new relations and FD sets
                                std::vector<Set*> pR1BCNF = BCNF(R1, S1);
                                std::vector<Set*> pR2BCNF = BCNF(R2, S2);

                                // Merge the sets of relations
                                for (size_t i = 0; i < pR2BCNF.size(); i++) {
                                        pR1BCNF.push_back(pR2BCNF[i]);
                                }

                                // Return the final set in BCNF
                                return pR1BCNF;
                        }
                }
        }

        // All keys checked out so return the relation we were given
        std::vector<Set*> pRelationSet;
        pRelationSet.push_back(R);
        return pRelationSet;
}

Set* BCNFCalculator::Closure(Set* Q, FDSet S) {
        Set* pC = new Set(Q);                            // Build a new set to work with (so we don't modify Q)

        int pClosureSize;
        do {
                pClosureSize = pC->Size();
                for (size_t i = 0; i < S.size(); i++) {  // Look at each FD in S
                        Set* pAs = S[i]->GetA();         // Get the As
                        Set* pBs = S[i]->GetB();         // Get the Bs

                        if (pAs->IsSubsetOf(pC)) {       // Are the As a subset of the closure
                                pC->Union(pBs);          // Add to closure
                        }
                }
        } while (pClosureSize != pC->Size());
        return pC;
}


bool BCNFCalculator::IsKey(Set* possibleKey, Set* relation) { return (relation->IsSubsetOf(possibleKey));}

void BCNFCalculator::Split(Set* R, FDSet S, Set*& R1, Set*& R2, FDSet& S1, FDSet& S2, Set* problemAs) {
        R1 = Closure(problemAs, S);
        R1->Intersect(R);
        R2 = new Set(R);
        R2->Subtract(R1);
        R2->Union(problemAs);

        SplitFDs(R, R1, S, S1);
        SplitFDs(R, R2, S, S2);
}
```

```
void BCNFCalculator::SplitFDs(Set* R, Set* R1, FDSet S, FDSet& S1) {
        std::vector<Set*> pSubsets = GetAllSubsets(R1);

        for (int i = 0; i < pSubsets.size(); i++) {      // Check each S for allowable
                Set* pAllowable = pSubsets[i];
                Set* pAllowableClosure = Closure(pAllowable, S);
                pAllowableClosure->Subtract(pAllowable);
                pAllowableClosure->Intersect(R1);

                for (size_t j = 0; j < pAllowableClosure->Size(); j++) {
                        std::string pRawSet = pAllowable->ToRawSet();
                        S1.push_back(new FunctionalDependency(pRawSet + " --> " + pAllowableClosure->Get(j)));
                }
        }
}

std::vector<Set*> BCNFCalculator::GetAllSubsets(Set* S) {
        std::vector<Set*> pSubsets;
        int numElements = S->Size();
        int numSubsets = pow(2, numElements);

        Set* pNewSubset;
        for (int i = 1; i < numSubsets; i++) {
                pNewSubset = new Set();
                std::bitset<32> x(i);
                std::ostringstream pTempOut;
                pTempOut << x;
                std::string pX = pTempOut.str();
                for (int q = 0; q < numElements; q++) {
                        std::string pTempString = std::string("") + pX[31 - q];
                        std::istringstream pTempIn(pTempString);
                        int oneOrZero;
                        pTempIn >> oneOrZero;
                        if (oneOrZero == 1) {
                                pNewSubset->Union(new Set(S->Get(q)));
                        }
                }
                pSubsets.push_back(pNewSubset);
        }
        return pSubsets;
}
```

FunctionalDependency

FunctionalDependency.h

```
#pragma once
#include <sstream>
#include <string>
#include <vector>
#include "Set.h"

class FunctionalDependency{
public:
        FunctionalDependency(std::string aArrowB);
        ~FunctionalDependency();
        Set* GetA();
        Set* GetB();
        bool IsEqual(FunctionalDependency* fd);
private:
        Set* mA;
        Set* mB;
        void ParseArrow(std::string aArrowB, std::string& pA, std::string& pB);
};

typedef std::vector<FunctionalDependency*> FDSet;
```

FunctionalDependency.cpp

```
#include "FunctionalDependency.h"

FunctionalDependency::FunctionalDependency(std::string aArrowB) {
        std::string pA;
        std::string pB;

        ParseArrow(aArrowB, pA, pB);
        mA = new Set(pA);
        mB = new Set(pB);
}

FunctionalDependency::~FunctionalDependency() {
        delete mA;
        delete mB;
}

void FunctionalDependency::ParseArrow(std::string aArrowB, std::string& pA, std::string& pB) {
        std::string pInString;
        std::istringstream in(aArrowB);

        bool pInA = true;
        while (in >> pInString) {
                if (pInString == "-->") {
                        pInA = false;
                } else if (pInA) {
                        pA = pA + " " + pInString;
                } else {
                        pB = pB + " " + pInString;
                }
        }
}

Set* FunctionalDependency::GetA() { return mA; }
Set* FunctionalDependency::GetB() { return mB; }
bool FunctionalDependency::IsEqual(FunctionalDependency* fd) {
        return (!mA->IsEqual(fd->GetA()) && !mB->IsEqual(fd->GetB()));
}
```

Set

Set.h

```
#pragma once
#include <string>
#include <vector>
#include <sstream>

class Set {
public:
        Set() {};
        Set(std::string spaceDelimitedSet);
        Set(Set* S);
        ~Set() {};

        std::string Get(int index);
        void Intersect(Set* s);
        bool IsEqual(Set* relation);
        bool IsSubsetOf(Set* s);
        int Size();
        void Subtract(Set* s);
        std::string ToString();
        std::string ToRawSet();
        void Union(Set* s);
protected:
        bool VContains(std::string v);
        std::vector<std::string> VUnion(std::vector<std::string> s);
        bool VIsEqual(std::vector<std::string> s);
private:
        std::vector<std::string> mSet;
};
```

Set.cpp

```
#include "Set.h"

Set::Set(std::string spaceDelimitedSet) {
        std::string pAtomicValue;
        std::istringstream in(spaceDelimitedSet);

        while (in >> pAtomicValue) {
                mSet.push_back(pAtomicValue);
        }
}

Set::Set(Set* S) { mSet = S->VUnion(mSet); }
std::string Set::Get(int index) { return mSet[index]; }

bool Set::IsSubsetOf(Set* s) {
        for (size_t i = 0; i < mSet.size(); i++) {
                if (!s->VContains(mSet[i])) {
                        return false;
                }
        }
}

void Set::Intersect(Set* s) {
        for (size_t i = 0; i < mSet.size(); i++) {
                if (!s->VContains(mSet[i])) {
                        mSet.erase(mSet.begin() + i);
                        i--;
                }
        }
}

bool Set::IsEqual(Set* relation) { return relation->VIsEqual(mSet);}
int Set::Size() { return (int)mSet.size();}
```

```
void Set::Subtract(Set* s) {
        for (size_t i = 0; i < mSet.size(); i++) {
                if (s->VContains(mSet[i])) {
                        mSet.erase(mSet.begin() + i);
                        i--;
                }
        }
}

std::string Set::ToRawSet() {
        std::string outString;
        for (size_t i = 0; i < mSet.size(); i++) {
                outString = outString + mSet[i] + " ";
        }
        return outString;
}

std::string Set::ToString() { return "{ " + ToRawSet() + "}";}
void Set::Union(Set* s) { mSet = s->VUnion(mSet); }

std::vector<std::string> Set::VUnion(std::vector<std::string> s) {
        std::vector<std::string> pUnion;
        for (std::string u : mSet) { // Grab all our values
                pUnion.push_back(u);
        }
        for (std::string q : s) { // For each s check and see if its in our set.
                if (!VContains(q)) { // If not then add it
                        pUnion.push_back(q);
                }
        }
        return pUnion;
}

bool Set::VContains(std::string v) {
        for (std::string s : mSet) {
                if (s == v) {
                        return true;
                }
        }
        return false;
}

bool Set::VIsEqual(std::vector<std::string> s) {
        if (s.size() != mSet.size()) { return false; }
        for (size_t i = 0; i < s.size(); i++) { // Compare set a to set b
                if (!VContains(s[i])) { return false; }
        }
        return true;
}
```

Works Cited

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). *Database Systems: The Complete Book (2nd Edition).* Pearson.