CUDD: Colorado University Decision Diagram Package

Systems Design Laboratory (2023/2024)

Computer Engineering for Robotics and Smart Industry

Luigi Capogrosso Davide Bresolin Tiziano Villa

Outline

CUDD

- CUDD is the **C**olorado **U**niversity **D**ecision **D**iagram package.
- It is a C/C++ library for creating the following different types of decision diagrams:
  - BDD: **B**inary **D**ecision **D**iagrams.
  - ZDD: **Z**ero-Suppressed B**DD**s.
  - ADD: **A**lgebraic **D**ecision **D**iagrams.
- The slides, source code, and all documentation related to this lecture are available here:
  `https://github.com/luigicapogrosso/SDL`

Getting CUDD

- The CUDD package is **available via anonymous FTP** from `vlsi.colorado.edu`.
- You can download the CUDD package from the server using an FTP client such as FileZilla or you can use the `ftp` command from the command line.
- Alternatively, you can download the latest version of CUDD **directly from the SDL GitHub repository**, so:

```
$ git clone
https://github.com/luigicapogrosso/SDL.git
```

Getting CUDD (cont'd)

- The library is tested using GCC (9.4.0) and GNU Make (9.4.0). To build the library from sources in a clean way, it is preferable that you set up a build subdirectory, say:

```
$ cd SDL/lecture_01/cudd-3.0.0
$ mkdir objdir && cd objdir
$ ../configure --prefix=$HOME/<path>
$ make && make install
```

Including and linking the CUDD library

- To build an application that uses the CUDD package, you should **add, in your source code, the following lines**:
  - `#include "cudd.h"`
  - `#include "util.h"`
- To **compile** and link a C program that uses CUDD:

```
$ gcc -o main main.c -lcudd -lutil
```

- Or, you can refer to the following **Makefile**: `https://github.com/luigicapogrosso/SDL/blob/master/lecture_01/code/Makefile`

Outline

Garbage Collection

- CUDD has a **built-in garbage collection** system.
- When a BDD is not used anymore, its memory can be reclaimed.
- To facilitate the garbage collector, **we need to "reference" and "dereference"** each node in our BDD:
  - `Cudd_Ref(DdNode*)` to reference a node.
  - `Cudd_RecursiveDeref(DdNode*)` to dereference a node and all its descendants.

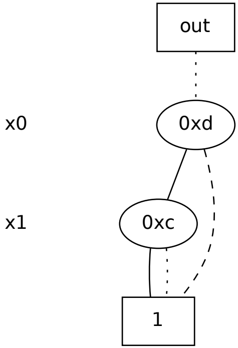
Complemented arcs

- Each node of a BDD can be:
  - **A variable** with two children.
  - **A leaf** with a constant value.
- The two children of a node are referred to as the *"then"* child and the *"else"* child.
- To assign a value to a BDD, we follow *"then"* and *"else"* children until we reach a leaf:
  - **The value of our assignment is the value of the leaf we reach**.
- **However:** *"else"* children can be **complemented**:
  - When an *"else"* child is complemented, then we take the **complement of the value of the leaf**:
    - ★ *i.e.*, if the value of the leaf is 1 and we have traversed an odd number of complement arcs, the value of our assignment is 0.

Complemented arcs: example

- $out = x_0\overline{x}_1$
- *"then"* arcs are solid.
- Normal *"else"* arcs are dashed.
- Complemented *"else"* arcs are dotted.
- The `out` arc is complemented:

$$\begin{aligned}\overline{out} &= \overline{x}_0 + x_1 \\ &= \overline{x}_0 + x_0x_1\end{aligned}$$



The `DdManager`

- The `DdManager` is the **key data structure of CUDD**:
  - It must be created before calling any other CUDD function.
  - It needs to be passed to almost every CUDD function.
- To initialize the `DdManager`, we use the following function:

```
DdManager * Cudd_Init(
  unsigned int numVars,     // initial number of BDD variables (i.e., subtables)
  unsigned int numVarsZ,    // initial number of ZDD variables (i.e., subtables)
  unsigned int numSlots,    // initial size of the unique tables
  unsigned int cacheSize,   // initial size of the cache
  unsigned long maxMemory   // target maximum memory occupation.(0 means unlimited)
);
```

The `DdManager`: C code

```
#include<stdio.h>
#include"cudd.h"

int main()
{
  DdManager* manager = Cudd_Init(0, 0,
      CUDD_UNIQUE_SLOTS,  CUDD_CACHE_SLOTS, 0);
  if(manager == NULL)
  {
    printf("Error when initalizing CUDD.\n");
    return 1;
  }
  ...

  return 0;
}
```

The DdNode

- The `DdNode` is the **core building block of BDDs**:

```
struct DdNode {
    DdHalfWord index;        // Index of the variable reprented by this node
    DdHalfWord ref;          // reference count
    DdNode *next;            // next pointer for unique table
    union {
    CUDD_VALUE_TYPE value;   // for constant nodes
    DdChildren kids;         // for internal nodes
    } type;
};
```

- `index` is a unique index for the variable represented by this node.
  - It is permanent: if we reorder variables, the idx remains the same.
- `ref` stores the reference count for this node.
  - It is incremented by `Cudd_Ref()` and decremented by `Cudd_Recursive_Deref()`.

Outline

BDD of Boolean functions

- **Common manipulations** of BDDs can be accomplished by **calling operators on variables**.
- The CUDD package includes Boolean functions that can be used for BDD operations such as: *NOT*, *AND*, *NAND*, *OR*, *NOR*, *Exclusive-OR*, *XNOR*, and etc.

BDD for the NOT Boolean function

- For the **NOT Boolean function**, we use `Cudd_Not()`.
- The truth table for a NOT:

$\mathbf{x_1}$	**f**
0	1
1	0

- Exercise: write the code to build the BDD for the function $f = \neg x_1$.

BDD for the AND Boolean function

- For the **AND Boolean function**, we use `Cudd_bddAnd()`.
- The truth table for an AND:

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

- Exercise: write the code to build the BDD for the function $f = x_1 \wedge x_2$.

BDD for the NAND Boolean function

- For the **NAND Boolean function**, we use `Cudd_bddNand()`.
- The truth table for a NAND:

x_1	x_2	f
0	0	1
0	1	1
1	0	1
1	1	0

- Exercise: write the code to build the BDD for the function $f = \neg(x_1 \wedge x_2)$.

BDD for the OR Boolean function

- For the **OR Boolean function**, we use `Cudd_bddOr()`.
- The truth table for a logic OR:

$\mathbf{x_1}$	$\mathbf{x_2}$	**f**
0	0	0
0	1	1
1	0	1
1	1	1

- Exercise: write the code to build the BDD for the function $f = x_1 \vee x_2$.

BDD for the NOR Boolean function

- For the **NOR Boolean function**, we use `Cudd_bddNor()`.
- The truth table for a NOR:

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	0

- Exercise: write the code to build the BDD for the function $f = \neg(x_1 \vee x_2)$.

BDD for Exclusive-OR Boolean function

- For the **Exclusive-OR Boolean function**, we use `Cudd_bddXor()`.
- The truth table for an Exclusive-OR:

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

- Exercise: write the code to build the BDD for the function $f = x1 \oplus x2$.

BDD for the XNOR Boolean function

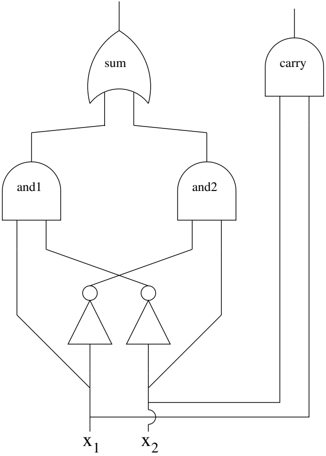
- For the **XNOR Boolean function**, we use `Cudd_bddXnor()`.
- The truth table for an XNOR:

x_1	x_2	f
0	0	1
0	1	0
1	0	0
1	1	1

- Exercise: write the code to build the BDD for the function $f = \neg(x_1 \oplus x_2)$.

Outline

The Half-Adder circuit



This is the schematic of a **half-adder circuit** that we want to compile into an OBDD. It has the following truth table:

$\mathbf{x_1}$	$\mathbf{x_2}$	**sum**	**carry**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Create the BDD for `sum`

```
DdNode *x1 = Cudd_bddIthVar(manager, 0);
DdNode *x2 = Cudd_bddIthVar(manager, 1);

DdNode *and1;
and1 = Cudd_bddAnd(manager, x1, Cudd_Not(x2));
Cudd_Ref(and1);

DdNode *and2;
and2 = Cudd_bddAnd(manager, Cudd_Not(x1), x2);
Cudd_Ref(and2);

DdNode *sum;
sum = Cudd_bddOr(manager, and1, and2);
Cudd_Ref(sum);

Cudd_RecursiveDeref(manager, and1);
Cudd_RecursiveDeref(manager, and2);
```

- Exercise: write the code for `carry`.

Restricting the BDD

- **Restricting a BDD** means **assigning a truth value to some of the variables.**
- The `Cudd_bddRestrict()` function returns the restricted BDD.

```
DdNode * Cudd_bddRestrict(
  DdManager * manager,   // DD manager
  DdNode * BDD,          // The BDD to restrict
  DdNode * restrictBy)   // The BDD to restrict by.
```

- `BDD` is the original BDD to restrict.
- `restrictBy` is the truth assignment of the variables.

Print the truth table

```
DdNode *restrictBy;
restrictBy = Cudd_bddAnd(manager, x1, Cudd_Not(x2));
Cudd_Ref(restrictBy);

DdNode *testSum;
testSum = Cudd_bddRestrict(manager, sum, restrictBy);
Cudd_Ref(testSum);
DdNode *testCarry;
testCarry = Cudd_bddRestrict(manager, carry, restrictBy);
Cudd_Ref(testCarry);

printf("x1 = 1, x2 = 0: sum = %d, carry = %d\n",
       1 - Cudd_IsComplement(testSum),
       1 - Cudd_IsComplement(testCarry));

Cudd_RecursiveDeref(manager, restrictBy);
Cudd_RecursiveDeref(manager, testSum);
Cudd_RecursiveDeref(manager, testCarry);
```

- Exercise: Write the code for the complete truth table.

Print the BDD with `graphviz`

- The function `Cudd_DumpDot()` dumps the BDD to a file in **GraphViz** format.
- The `.dot` file can be converted to a PDF by the command `dot`:

```
$ dot -O -Tpdf half_adder.dot
```

Print the BDD: C code

```
char *inputNames[2];
inputNames[0] = "x1";
inputNames[1] = "x2";
char *outputNames[2];
outputNames[0] = "sum";
outputNames[1] = "carry";

DdNode *outputs[2];
outputs[0] = sum;
Cudd_Ref(outputs[0]);
outputs[1] = carry;
Cudd_Ref(outputs[1]);

FILE *f = fopen("half_adder.dot", "w");

Cudd_DumpDot(manager, 2, outputs, inputNames, outputNames, f);

Cudd_RecursiveDeref(manager, outputs[0]);
Cudd_RecursiveDeref(manager, outputs[1]);
fclose(f);
```

Variable reordering

- **The order** of variables can have a **tremendous effect on the size of BDDs**.
- CUDD provides a rich set of tools for reordering BDDs:
  - Automatic reordering (using heuristics) when the number of nodes in the BDD passes a certain threshold.
  - Manual reordering using different heuristics.
  - Manual reordering with a user-specified variable order.
- The function `Cudd_ShuffleHeap()` is used to define the variable order:

```
int Cudd_ShuffleHeap(
  DdManager * manager,    // DD manager
  int * permutation       // required variable permutation
)
```

Exercise: play with the variable order!

- Create the BDD for the function $x_1x_2 + x_3x_4 + x_5x_6$.
- Try the following variable orders and compare the results:
  - $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$
  - $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$

HINTS

- `int Cudd_ReadPerm(manager, x2->index)` returns the position of variable `x2` in the order.
- `int Cudd_ReadNodeCount(manager)` returns the number of nodes in the BDD.

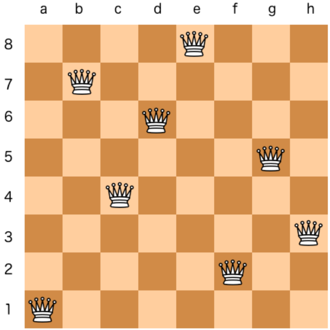
Converting BDDs to ZDDs

- Many applications **first build a set of BDDs and then derive ZDDs** from the BDDs.
- These applications should create the manager with 0 ZDD variables and create the BDDs.
- Then they should call `Cudd_zddVarsFromBddVars()` to create the necessary ZDD variables–whose number is known once the BDDs are available.

Converting BDDs to ZDDs (cont'd)

- The simplest conversion from BDDs to ZDDs is a simple change of representation, which preserves the functions.
- Simply put, given a BDD for *f*, a ZDD for *f* is requested. In this case the correspondence between the BDD variables and ZDD variables is *one-to-one*.
- Hence, `Cudd_zddVarsFromBddVars()` should be called with the *multiplicity* parameter equal to 1.
- The **conversion can then be performed by calling:** `Cudd_zddPortFromBdd()`.
- The **inverse transformation is performed by calling:** `Cudd_zddPortToBdd()`.

The N-Queens problem



- The **N-Queens problem** is the problem of placing N non-attacking queens on an $N \times N$ chessboard.
- Our implementation of these benchmarks is based on the description of Kunkle10. We construct a ZDD row-by-row to represent whether the row is in a legal state.
- On the accumulated ZDD we then count the number of satisfying assignments.