

# Principles of AI Planning

December 8th, 2006 — Planning with binary decision diagrams

## Binary Decision Diagrams

- Motivation

- Definition

## BDD operations

- Ideas

- Essential operations

- Derived operations

## Planning with BDDs

- Main algorithm

- The apply function

- Remarks

# Principles of AI Planning

## Planning with binary decision diagrams

Malte Helmert    Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

December 8th, 2006

## Dealing with large state spaces

- ▶ One way to explore very large state spaces is to use **selective** exploration methods (such as heuristic search) that only explore a fraction of states.
- ▶ Another method is to **concisely represent** large sets of states and deal with large state sets at the same time.

# Breadth-first search with progression and state sets

## Progression breadth-first search

**def** bfs-progression( $A, I, O, G$ ):

*goal* := formula-to-set( $G$ )

*reached* :=  $\{I\}$

**loop**:

**if**  $reached \cap goal \neq \emptyset$ :

**return** solution found

*new-reached* :=  $reached \cup apply(reached, O)$

**if**  $new-reached = reached$ :

**return** no solution exists

*reached* := *new-reached*

$\rightsquigarrow$  If we can implement operations *formula-to-set*,  $\{I\}$ ,  $\cap$ ,  $\neq \emptyset$ ,  $\cup$ , *apply* and  $=$  efficiently, this is a reasonable algorithm.

## Formulae to represent state sets

- ▶ We have previously considered **boolean formulae** as a means of representing set of states.
- ▶ Compared to **explicit representations** of state sets, boolean formulae have very nice performance characteristics.

**Note:** In the following, we assume that formulae are implemented as **trees**, not **strings**, so that we can e.g. compute  $\chi \wedge \psi$  from  $\chi$  and  $\psi$  in **constant time**.

# Performance characteristics

## Explicit representations vs. formulae

Let  $k$  be the **number of state variables**,  $|S|$  the **number of states** in  $S$  and  $\|S\|$  the **size of the representation** of  $S$ .

	Sorted vector	Hash table	Formula
$s \in S?$	$O(k \log  S )$	$O(k)$	$O(\ S\ )$
$S := S \cup \{s\}$	$O(k \log  S  +  S )$	$O(k)$	$O(k)$
$S := S \setminus \{s\}$	$O(k \log  S  +  S )$	$O(k)$	$O(k)$
$S \cup S'$	$O(k S  + k S' )$	$O(k S  + k S' )$	$O(1)$
$S \cap S'$	$O(k S  + k S' )$	$O(k S  + k S' )$	$O(1)$
$S \setminus S'$	$O(k S  + k S' )$	$O(k S  + k S' )$	$O(1)$
$\bar{S}$	$O(k2^k)$	$O(k2^k)$	$O(1)$
$\{s \mid s(a) = 1\}$	$O(k2^k)$	$O(k2^k)$	$O(1)$
$S = \emptyset?$	$O(1)$	$O(1)$	co-NP-complete
$S = S'?$	$O(k S )$	$O(k S )$	co-NP-complete
$ S $	$O(1)$	$O(1)$	#P-complete

## Which operations are important?

- ▶ **Explicit representations** such as hash tables are not suitable because their size grows linearly with the number of represented states.
- ▶ **Formulae** are very efficient for some operations, but not very well suited for other important operations needed by the progression algorithm.
  - ▶ Examples:  $S \neq \emptyset?$ ,  $S = S'?$
- ▶ One of the sources of difficulty is that formulae allow **many different representations** for a given set.
  - ▶ For example, all unsatisfiable formulae represent  $\emptyset$ .

This makes equality tests expensive.

⇒ We are interested in **canonical representations**, i.e. representations for which there is only **one possible representation** for every state set.

**Binary decision diagrams** (BDDs) are an example of an efficient canonical representation.

# Performance characteristics

## Formulae vs. BDDs

Let  $k$  be the **number of state variables**,  $|S|$  the **number of states** in  $S$  and  $\|S\|$  the **size of the representation** of  $S$ .

	Formula	BDD
$s \in S?$	$O(\ S\ )$	$O(k)$
$S := S \cup \{s\}$	$O(k)$	$O(k)$
$S := S \setminus \{s\}$	$O(k)$	$O(k)$
$S \cup S'$	$O(1)$	$O(\ S\  \ S'\ )$
$S \cap S'$	$O(1)$	$O(\ S\  \ S'\ )$
$S \setminus S'$	$O(1)$	$O(\ S\  \ S'\ )$
$\bar{S}$	$O(1)$	$O(\ S\ )$
$\{s \mid s(a) = 1\}$	$O(1)$	$O(1)$
$S = \emptyset?$	co-NP-complete	$O(1)$
$S = S'?$	co-NP-complete	$O(1)$
$ S $	#P-complete	$O(\ S\ )$

**Remark:** Optimizations allow BDDs with complementation ( $\bar{S}$ ) in constant time, but we will not discuss this here.

# Binary decision diagrams

## Definition

### Definition (BDD)

Let  $A$  be a set of propositional variables.

A **binary decision diagram** (BDD) over  $A$  is a directed acyclic graph with labeled arcs and labeled vertices satisfying the following conditions:

- ▶ There is exactly one node without incoming arcs.
- ▶ All sinks (nodes without outgoing arcs) are labeled **0** or **1**.
- ▶ All other nodes are labeled with a variable  $a \in A$  and have exactly two outgoing arcs, labeled **0** and **1**.

# Binary decision diagrams

## Terminology

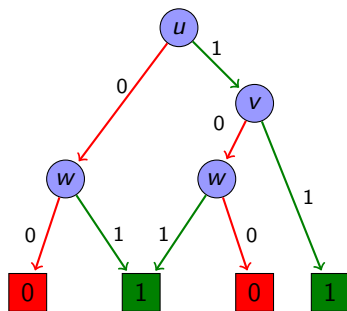
### BDD terminology

- ▶ The node without incoming arcs is called the **root**.
- ▶ The labeling variable of an internal node is called the **decision variable** of the node.
- ▶ The nodes reached from node  $n$  via the arc labeled  $i \in \{0, 1\}$  is called the  **$i$ -successor** of  $n$ .
- ▶ The BDDs which only consist of a single sink are called the **zero BDD** and **one BDD**, respectively.

**Observation:** If  $B$  is a BDD and  $n$  is a node of  $B$ , then the subgraph induced by all nodes reachable from  $n$  is also a BDD.

- ▶ This BDD is called the **BDD rooted at  $n$** .

## BDD example

Possible BDD for  $(u \wedge v) \vee w$ 

## BDD semantics

### Testing whether a BDD includes a valuation

**def** `bdd-includes`( $B$ : BDD,  $v$ : valuation):

    Set  $n$  to the root of  $B$ .

**while**  $n$  is not a sink:

        Set  $a$  to the decision variable of  $n$ .

        Set  $n$  to the  $v(a)$ -successor of  $n$ .

**return** true if  $n$  is labeled 1, false if it is labeled 0.

### Definition (set represented by a BDD)

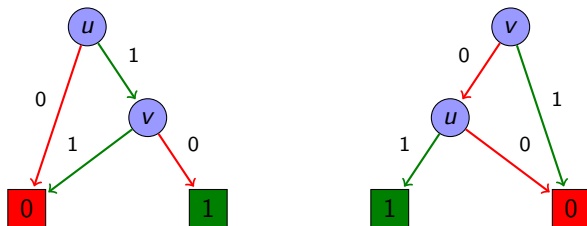
Let  $B$  be a BDD over variables  $A$ . The **set represented by  $B$** , in symbols  $r(B)$  consists of all valuations  $v : A \rightarrow \{0, 1\}$  for which `bdd-includes`( $B, v$ ) returns true.

# Ordered BDDs

## Motivation

In general, BDDs are not a canonical representation for sets of valuations. Here is a simple counter-example ( $A = \{u, v\}$ ):

BDDs for  $u \wedge \neg v$  with different variable order



Both BDDs represent the same state set, namely the singleton set  $\{\{u \mapsto 1, v \mapsto 0\}\}$ .

# Ordered BDDs

## Definition

- ▶ As a first step towards a canonical representation, we will in the following assume that the set of variables  $A$  is **totally ordered** by some ordering  $\prec$ .
- ▶ In particular, we will only use variables  $v_1, v_2, v_3, \dots$  and assume the ordering  $v_i \prec v_j$  iff  $i < j$ .

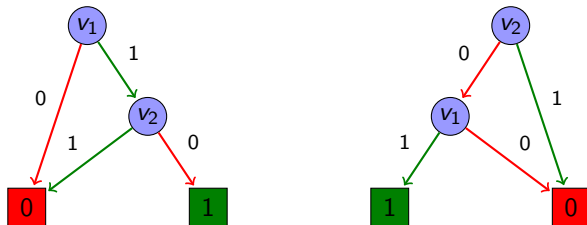
## Definition (ordered BDD)

A BDD is **ordered** iff for each arc from an internal node with decision variable  $u$  to an internal node with decision variable  $v$ , we have  $u \prec v$ .

# Ordered BDDs

## Example

### Ordered and unordered BDD

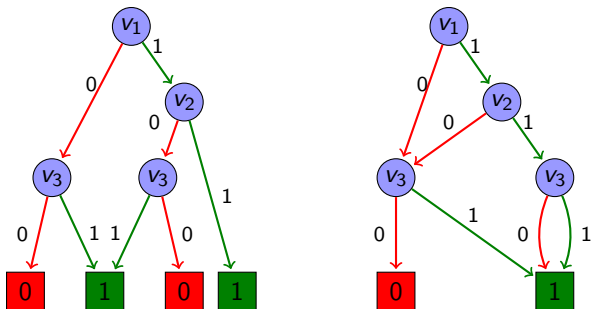


The left BDD is ordered, the right one is not.

# Reduced ordered BDDs

Are ordered BDDs canonical?

Two equivalent BDDs that can be reduced



- ▶ Ordered BDDs are not canonical: Both ordered BDDs represent the same set.
- ▶ However, ordered BDDs can easily be **made** canonical.

# Reduced ordered BDDs

## Reductions

There are two important operations on BDDs that do not change the set represented by it:

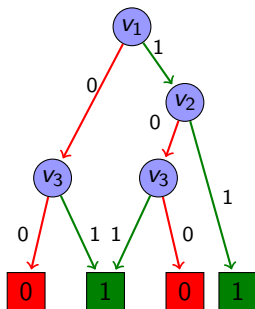
### Definition (Isomorphism reduction)

If the BDDs rooted at two different nodes  $n$  and  $n'$  are **isomorphic**, then all incoming arcs of  $n'$  can be redirected to  $n$ , and all parts of the BDD no longer reachable from the root removed.

# Reduced ordered BDDs

## Reductions

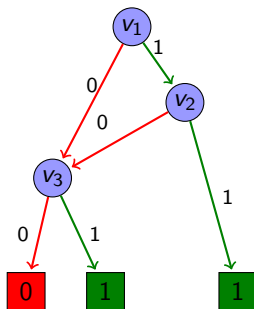
### Isomorphism reduction



# Reduced ordered BDDs

## Reductions

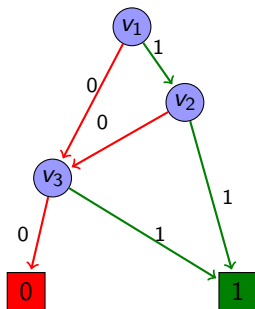
### Isomorphism reduction



# Reduced ordered BDDs

## Reductions

### Isomorphism reduction



# Reduced ordered BDDs

## Reductions

There are two important operations on BDDs that do not change the set represented by it:

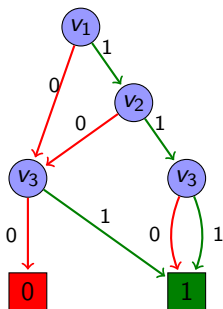
### Definition (Shannon reduction)

If both outgoing arcs of an internal node  $n$  of a BDD lead to the same node  $m$ , then  $n$  can be removed from the BDD, with all incoming arcs of  $n$  going to  $m$  instead.

# Reduced ordered BDDs

## Reductions

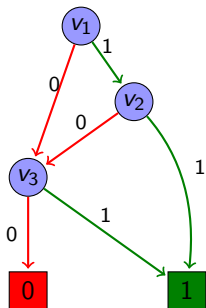
### Shannon reduction



# Reduced ordered BDDs

## Reductions

### Shannon reduction



# Definition

## Definition (reduced ordered BDD)

An ordered BDD is **reduced** iff it does not admit any isomorphism reduction or Shannon reduction.

## Theorem (Bryant 1986)

*For every state set  $S$  and a fixed variable ordering, there exists exactly one reduced ordered BDD representing  $S$ .*

*Moreover, given any ordered BDD  $B$ , the equivalent reduced ordered BDD can be computed in linear time in the size of  $B$ .*

↪ Reduced ordered BDDs are the canonical representation we were looking for.

From now on, we simply say **BDD** for **reduced ordered BDD**.

# Efficient BDD implementation

## Ideas

- ▶ Earlier, we showed some BDD performance characteristics.
  - ▶ Example:  $S = S'$ ? can be tested in time  $O(1)$ .
- ▶ The critical idea for achieving this performance is to **share structure** not only within a BDD, but also between **different BDDs**.

## BDD representation

- ▶ Every BDD (including sub-BDDs)  $B$  is represented by a single natural number  $id(B)$  called its **ID**.
  - ▶ The zero BDD has ID  $-2$ .
  - ▶ The one BDD has ID  $-1$ .
  - ▶ Other BDDs have IDs  $\geq 0$ .
- ▶ The BDD operations must satisfy the following invariant: Two BDDs with different ID are **never** identical.

# Efficient BDD implementation

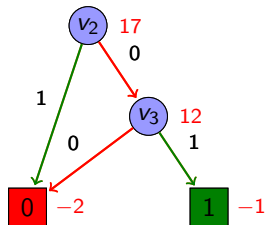
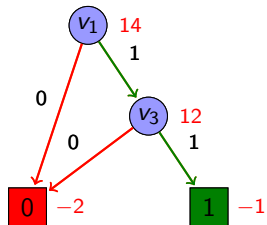
## Data structures

### Data structures

- ▶ There are three global vectors (dynamic arrays) to represent information on non-sink BDDs with ID  $i \geq 0$ :
  - ▶  $var[i]$  denotes the decision variable.
  - ▶  $low[i]$  denotes the ID of the 0-successor.
  - ▶  $high[i]$  denotes the ID of the 1-successor.
- ▶ There is some mechanism that keeps track of IDs that are currently unused (garbage collection, reference counting).
  - ▶ This can be implemented without amortized overhead.
- ▶ There is a global hash table *lookup* which maps, for each ID  $i \geq 0$  representing a BDD in use, the triple  $\langle var[i], low[i], high[i] \rangle$  to  $i$ .
  - ▶ Randomized hashing allows constant-time access in the **expected case**. More sophisticated methods allow deterministic constant-time access.

## Efficient BDD implementation

## Data structures example



formula	ID $i$	$var[i]$	$low[i]$	$high[i]$
$\perp$	-2	-	-	-
$\top$	-1	-	-	-
$v_3$	12	3	-2	-1
$v_1 \wedge v_3$	14	1	-2	12
$\neg v_2 \wedge v_3$	17	2	12	-2

# Core BDD operations

## Building the zero BDD

```
def zero():  
    return -2
```

## Building the one BDD

```
def one():  
    return -1
```

# Core BDD operations

## Building other BDDs

```

def bdd(v: variable, l: ID, h: ID):
    if l = h:
        return l
    if  $\langle v, l, h \rangle \notin \text{lookup}$ :
        Set i to a new unused ID.
        var[i], low[i], high[i] := v, l, h
        lookup[ $\langle v, l, h \rangle$ ] := i
    return lookup[ $\langle v, l, h \rangle$ ]
  
```

We only create BDDs with **zero**, **one** and **bdd** (i.e., function `bdd` is the only function writing to `var`, `low`, `high` and `lookup`). Thus:

- ▶ BDDs are guaranteed to be reduced.
- ▶ BDDs with different IDs always represent different sets.

# BDD operations

## Notations

For convenience, we introduce some additional notations:

- ▶ We define  $\mathbf{0} := \text{zero}()$ ,  $\mathbf{1} := \text{one}()$ .
- ▶ We write *var*, *low*, *high* as attributes:
  - ▶  $B.\text{var}$  for  $\text{var}[B]$
  - ▶  $B.\text{low}$  for  $\text{low}[B]$
  - ▶  $B.\text{high}$  for  $\text{high}[B]$

# Essential vs. derived BDD operations

We distinguish between

- ▶ **essential BDD operations**, which are implemented directly on top of **zero**, **one** and **bdd**, and
- ▶ **derived BDD operations**, which are implemented in terms of the essential operations.

# Essential BDD operations

We study the following essential operations:

- ▶ `bdd-includes( $B, s$ )`: Test  $s \in r(B)$ .
- ▶ `bdd-equals( $B, B'$ )`: Test  $r(B) = r(B')$ .
- ▶ `bdd-atom( $a$ )`: Build BDD representing  $\{s \mid s(a) = 1\}$ .
- ▶ `bdd-state( $s$ )`: Build BDD representing  $\{s\}$ .
- ▶ `bdd-union( $B, B'$ )`: Build BDD representing  $r(B) \cup r(B')$ .
- ▶ `bdd-complement( $B$ )`: Build BDD representing  $\overline{r(B)}$ .
- ▶ `bdd-countmodels( $B$ )`: Compute  $|r(B)|$ .
- ▶ `bdd-forget( $B, a$ )`: Described later.

# Essential operations

## Memoization

- ▶ The essential functions are all defined recursively and are free of side effects.
- ▶ We assume (without explicit mention in the pseudo-code) that they all use **dynamic programming** (memoization):
  - ▶ Every **return** statement stores the arguments and result in a memo hash table.
  - ▶ Whenever a function is invoked, the memo is checked if the same call was made previously. If so, the result from the memo is taken to avoid recomputations.
- ▶ The memo may be cleared when the “outermost” recursive call terminates.
  - ▶ The bdd-forget function calls the bdd-union function internally. In this case, the memo for bdd-union may only be cleared once bdd-forget finishes, **not** after each bdd-union invocation finishes.

Memoization is critical for the mentioned runtime bounds.

# Essential BDD operations

bdd-includes

Test  $s \in r(B)$

```
def bdd-includes( $B, s$ ):  
    if  $B = \mathbf{0}$ :  
        return false  
    else if  $B = \mathbf{1}$ :  
        return true  
    else if  $s[B.var] = 1$ :  
        return bdd-includes( $B.high, s$ )  
    else:  
        return bdd-includes( $B.low, s$ )
```

- ▶ Runtime:  $O(k)$
- ▶ This works for partial or full valuations  $s$ , as long as all variables appearing in the BDD are defined.

# Essential BDD operations

bdd-equals

Test  $r(B) = r(B')$

```
def bdd-equals( $B, B'$ ):  
    return  $B = B'$ 
```

► Runtime:  $O(1)$

# Essential BDD operations

bdd-atom

Build BDD representing  $\{s \mid s(a) = 1\}$

```
def bdd-atom(a):  
    return bdd(a, 0, 1)
```

► Runtime:  $O(1)$

# Essential BDD operations

bdd-state

Build BDD representing  $\{s\}$

**def** bdd-state( $s$ ):

$B := \mathbf{1}$

**for each** variable  $v$  of  $s$ , in reverse variable order:

**if**  $s(v) = 1$ :

$B := \text{bdd}(v, \mathbf{0}, B)$

**else**:

$B := \text{bdd}(v, B, \mathbf{0})$

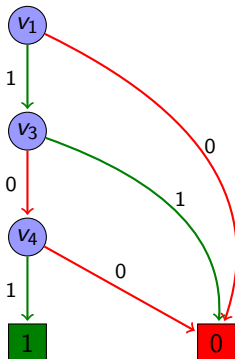
**return**  $B$

- ▶ Runtime:  $O(k)$
- ▶ Works for partial or full valuations  $s$ .

# Essential BDD operations

bdd-state: Example

$bdd\text{-state}(\{v_1 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1\})$



# Essential BDD operations

bdd-union

Build BDD representing  $r(B) \cup r(B')$

```

def bdd-union( $B, B'$ ):
    if  $B = \mathbf{0}$  and  $B' = \mathbf{0}$ :
        return  $\mathbf{0}$ 
    else if  $B = \mathbf{1}$  or  $B' = \mathbf{1}$ :
        return  $\mathbf{1}$ 
    else if  $B.\text{var} < B'.\text{var}$ :
        return  $\text{bdd}(B.\text{var}, \text{bdd-union}(B.\text{low}, B'),$ 
                 $\text{bdd-union}(B.\text{high}, B'))$ 
    else if  $B.\text{var} = B'.\text{var}$ :
        return  $\text{bdd}(B.\text{var}, \text{bdd-union}(B.\text{low}, B'.\text{low}),$ 
                 $\text{bdd-union}(B.\text{high}, B'.\text{high}))$ 
    else if  $B.\text{var} > B'.\text{var}$ :
        return  $\text{bdd}(B'.\text{var}, \text{bdd-union}(B, B'.\text{low}),$ 
                 $\text{bdd-union}(B, B'.\text{high}))$ 

```

► Runtime:  $O(\|B\| \cdot \|B'\|)$

# Essential BDD operations

## bdd-complement

Build BDD representing  $\overline{r(B)}$

```
def bdd-complement(B):  
    if B = 0:  
        return 1  
    else if B = 1:  
        return 0  
    else:  
        return bdd(B.var, bdd-complement(B.low),  
                    bdd-complement(B.high))
```

► Runtime:  $O(\|B\|)$

# Essential BDD operations

bdd-countmodels

Compute  $|r(B)|$

```
def bdd-countmodels( $B$ ):
    return count( $B$ , 0)
```

```
def count( $B$ ,  $i$ ):
```

```
    if  $B = 0$ :
```

```
        return 0
```

```
    else if  $B = 1$ :
```

```
        return  $2^{k-i}$ 
```

```
    else:
```

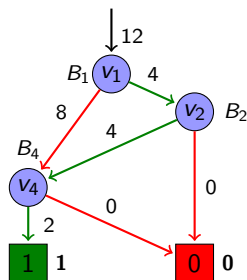
```
        Set  $j$  so that  $B.\text{var} = v_j$ .
```

```
        return  $2^{j-i-1} \cdot (\text{count}(B.\text{low}, j) + \text{count}(B.\text{high}, j))$ 
```

► Runtime:  $O(\|B\|)$

## Essential BDD operations

bdd-countmodels: Example



BDD represents  $v_4 \wedge (\neg v_1 \vee v_2)$  over variables  $\{v_1, v_2, v_3, v_4, v_5\}$ , i.e.  $k = 5$ .

$$\text{count}(B_1, 0) = 1 \cdot (\text{count}(B_4, 1) + \text{count}(B_2, 1)) = 12$$

$$\text{count}(B_4, 1) = 4 \cdot (\text{count}(\mathbf{0}, 4) + \text{count}(\mathbf{1}, 4)) = 8$$

$$\text{count}(\mathbf{0}, 4) = 0$$

$$\text{count}(\mathbf{1}, 4) = 2$$

$$\text{count}(B_2, 1) = 1 \cdot (\text{count}(\mathbf{0}, 2) + \text{count}(B_4, 2)) = 4$$

$$\text{count}(\mathbf{0}, 2) = 0$$

$$\text{count}(B_4, 2) = 2 \cdot (\text{count}(\mathbf{0}, 4) + \text{count}(\mathbf{1}, 4)) = 4$$

# Essential BDD operations

bdd-forget

The last essential BDD operation is a bit more unusual, but we will need it for defining the semantics of operator application.

## Definition (Existential abstraction)

Let  $A$  be a set of propositional variables, let  $S$  be a set of valuations over  $A$ , and let  $v \in A$ .

The **existential abstraction of  $v$  in  $S$** , in symbols  $\exists v.S$ , is the set of valuations

$$\{ s' : (A \setminus \{v\}) \rightarrow \{0, 1\} \mid \exists s \in S : s' \subset s \}$$

over  $A \setminus \{v\}$ .

Existential abstraction is also called **forgetting**.

# Essential BDD operations

bdd-forget

Build BDD representing  $\exists v.r(B)$

```

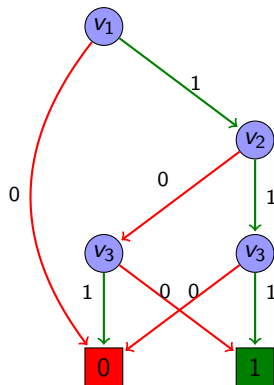
def bdd-forget( $B, v$ ):
    if  $B = \mathbf{0}$  or  $B = \mathbf{1}$  or  $B.\text{var} \succ v$ :
        return  $B$ 
    else if  $B.\text{var} \prec v$ :
        return  $\text{bdd}(B.\text{var}, \text{bdd-forget}(B.\text{low}, v),$ 
                 $\text{bdd-forget}(B.\text{high}, v))$ 
    else:
        return  $\text{bdd-union}(B.\text{low}, B.\text{high})$ 
  
```

► Runtime:  $O(\|B\|^2)$

# Essential BDD operations

bdd-forget: Example

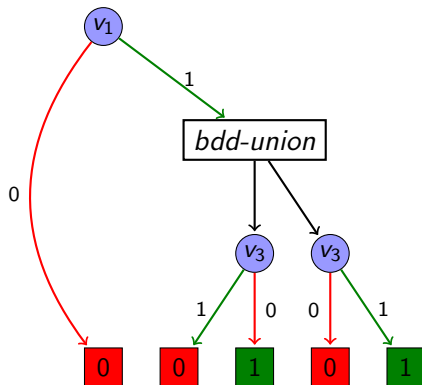
Forgetting  $v_2$



# Essential BDD operations

bdd-forget: Example

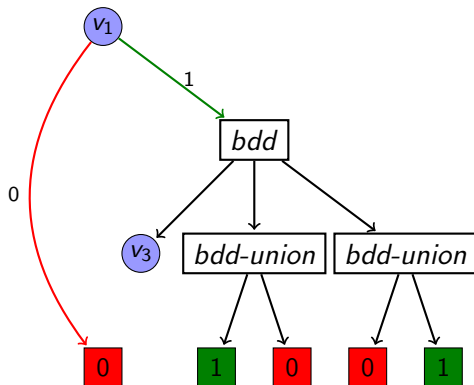
Forgetting  $v_2$



# Essential BDD operations

bdd-forget: Example

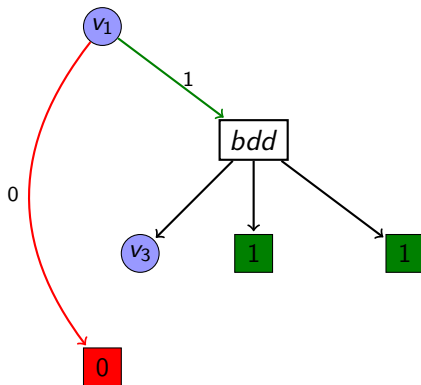
Forgetting  $v_2$



# Essential BDD operations

bdd-forget: Example

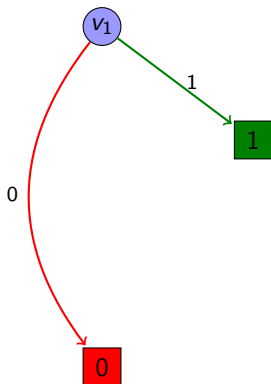
Forgetting  $v_2$



# Essential BDD operations

bdd-forget: Example

Forgetting  $v_2$



## Derived BDD operations

We study the following derived operations:

- ▶  $\text{bdd-intersection}(B, B')$ :  
Build BDD representing  $r(B) \cap r(B')$ .
- ▶  $\text{bdd-setdifference}(B, B')$ :  
Build BDD representing  $r(B) \setminus r(B')$ .
- ▶  $\text{bdd-isempty}(B)$ :  
Test  $r(B) = \emptyset$ .
- ▶  $\text{bdd-rename}(B, v, v')$ :  
Build BDD representing  $\{\text{rename}(s, v, v') \mid s \in r(B)\}$ , where  $\text{rename}(s, v, v')$  is the valuation  $s$  with variable  $v$  renamed to  $v'$ .
  - ▶ If variable  $v'$  occurs in  $B$  already, the result is undefined.

## Derived BDD operations

*bdd-intersection*, *bdd-setdifference*

Build BDD representing  $r(B) \cap r(B')$

**def** *bdd-intersection*( $B, B'$ ):

*not-B* := *bdd-complement*( $B$ )

*not-B'* := *bdd-complement*( $B'$ )

**return** *bdd-complement*(*bdd-union*(*not-B*, *not-B'*))

Build BDD representing  $r(B) \setminus r(B')$

**def** *bdd-setdifference*( $B, B'$ ):

**return** *bdd-intersection*( $B, \textit{bdd-complement}(B')$ )

- ▶ Runtime:  $O(\|B\| \cdot \|B'\|)$
- ▶ These functions can also be easily implemented directly, following the structure of *bdd-union*.

# Derived BDD operations

`bdd-isempty`

Test  $r(B) = \emptyset$

```
def bdd-isempty(B):  
    return bdd-equals(B, 0)
```

► Runtime:  $O(1)$

# Derived BDD operations

## bdd-rename

Build BDD representing  $\{ \text{rename}(s, v, v') \mid s \in r(B) \}$

**def** `bdd-rename`( $B, v, v'$ ):

$v\text{-and-}v' := \text{bdd-intersection}(\text{bdd-atom}(v), \text{bdd-atom}(v'))$

$\text{not-}v := \text{bdd-complement}(\text{bdd-atom}(v))$

$\text{not-}v' := \text{bdd-complement}(\text{bdd-atom}(v'))$

$\text{not-}v\text{-and-not-}v' := \text{bdd-intersection}(\text{not-}v, \text{not-}v')$

$v\text{-eq-}v' := \text{bdd-union}(v\text{-and-}v', \text{not-}v\text{-and-not-}v')$

**return**  $\text{bdd-forget}(\text{bdd-intersection}(B, v\text{-eq-}v'), v)$

► Runtime:  $O(\|B\|^2)$

# Derived BDD operations

## bdd-rename: Remarks

- ▶ Renaming sounds like a simple operation.
- ▶ Why is it so expensive?

This is **not** because the algorithm is bad:

- ▶ Renaming **must** take at least quadratic time:
  - ▶ There exist families of BDDs  $B_n$  with  $k$  variables such that renaming  $v_1$  to  $v_{k+1}$  increases the size of the BDD from  $\Theta(n)$  to  $\Theta(n^2)$ .
- ▶ However, renaming is cheap in **some cases**:
  - ▶ For example, renaming to a **neighboring** unused variable (e.g. from  $v_i$  to  $v_{i+1}$ ) is always possible in linear time by simply relabeling the decision variables of the BDD.
- ▶ In practice, one can usually choose a variable ordering where renaming only occurs between neighboring variables.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

**def** bfs-progression( $A, I, O, G$ ):

*goal* := formula-to-set( $G$ )

*reached* :=  $\{I\}$

**loop:**

**if**  $reached \cap goal \neq \emptyset$ :

**return** solution found

*new-reached* :=  $reached \cup apply(reached, O)$

**if**  $new-reached = reached$ :

**return** no solution exists

*reached* := *new-reached*

# Breadth-first search with progression and BDDs

## Progression breadth-first search

**def** bfs-progression( $A, I, O, G$ ):

$goal := formula-to-set(G)$

$reached := \{I\}$

**loop:**

**if**  $reached \cap goal \neq \emptyset$ :

**return** solution found

$new-reached := reached \cup apply(reached, O)$

**if**  $new-reached = reached$ :

**return** no solution exists

$reached := new-reached$

Use *bdd-atom*, *bdd-complement*, *bdd-union*, *bdd-intersection*.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

```

def bfs-progression(A, I, O, G):
  goal := formula-to-set(G)
  reached := {I}
  loop:
    if reached ∩ goal ≠ ∅:
      return solution found
    new-reached := reached ∪ apply(reached, O)
    if new-reached = reached:
      return no solution exists
    reached := new-reached
  
```

Use *bdd-state*.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

```

def bfs-progression(A, I, O, G):
    goal := formula-to-set(G)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ apply(reached, O)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
  
```

Use *bdd-intersection*, *bdd-isempty*.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

```

def bfs-progression( $A, I, O, G$ ):
     $goal := formula-to-set(G)$ 
     $reached := \{I\}$ 
    loop:
        if  $reached \cap goal \neq \emptyset$ :
            return solution found
         $new-reached := reached \cup apply(reached, O)$ 
        if  $new-reached = reached$ :
            return no solution exists
         $reached := new-reached$ 
  
```

Use *bdd-union*.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

```

def bfs-progression( $A, I, O, G$ ):
     $goal := formula-to-set(G)$ 
     $reached := \{I\}$ 
    loop:
        if  $reached \cap goal \neq \emptyset$ :
            return solution found
         $new-reached := reached \cup apply(reached, O)$ 
        if  $new-reached = reached$ :
            return no solution exists
         $reached := new-reached$ 
  
```

Use *bdd-equals*.

# Breadth-first search with progression and BDDs

## Progression breadth-first search

**def** bfs-progression( $A, I, O, G$ ):

*goal* := formula-to-set( $G$ )

*reached* :=  $\{I\}$

**loop:**

**if**  $reached \cap goal \neq \emptyset$ :

**return** solution found

*new-reached* :=  $reached \cup \text{apply}(reached, O)$

**if**  $new-reached = reached$ :

**return** no solution exists

*reached* := *new-reached*

How to do this?

## The *apply* function

- ▶ We need an operation that, for a set of states *reached* (given as a BDD) and a set of operators  $O$ , computes the set of states (as a BDD) that can be reached by applying some operator  $o \in O$  in some state  $s \in \textit{reached}$ .
- ▶ We have seen something similar already. . .

# Translating operators into formulae

(slide taken from the “planning by satisfiability testing” chapter)

## Definition (operators in propositional logic)

Let  $o = \langle c, e \rangle$  be an operator and  $A$  a set of state variables.  
Define  $\tau_A(o)$  as the conjunction of

$$c \quad (1)$$

$$\bigwedge_{a \in A} (EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))) \leftrightarrow a' \quad (2)$$

$$\bigwedge_{a \in A} \neg (EPC_a(e) \wedge EPC_{\neg a}(e)) \quad (3)$$

Condition (1) states that the precondition of  $o$  is satisfied.

Condition (2) states that the **new value of  $a$** , represented by  $a'$ , is 1 if the old value was 1 and it did not become 0, or if it became 1.

Condition (3) states that none of the state variables is assigned both 0 and 1. Together with (1), this encodes applicability of the operator.

## The *apply* function

- ▶ The formula  $\tau_A(o)$  describes the applicability of a **single** operator  $o$  and the effect of applying  $o$  as a binary formula over variables  $A$  (describing the state in which  $o$  is applied) and  $A'$  (describing the resulting state).
- ▶ The formula  $\bigvee_{o \in O} \tau_A(o)$  describes state transitions by **any** operator.
- ▶ We can translate this formula to a BDD (over variables  $A \cup A'$ ) using *bdd-atom*, *bdd-complement*, *bdd-union*, *bdd-intersection*.
- ▶ The resulting BDD is called the **transition relation** of the planning task, written as  $T_A(O)$ .

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

**def** *apply*(*reached*, *O*):

$B := T_A(O)$

$B := \text{bdd-intersection}(B, \text{reached})$

**for each**  $a \in A$ :

$B := \text{bdd-forget}(B, a)$

**for each**  $a \in A$ :

$B := \text{bdd-rename}(B, a', a)$

**return**  $B$

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

**def** *apply*(*reached*, *O*):

$B := T_A(O)$

$B := \text{bdd-intersection}(B, \text{reached})$

**for each**  $a \in A$ :

$B := \text{bdd-forget}(B, a)$

**for each**  $a \in A$ :

$B := \text{bdd-rename}(B, a', a)$

**return**  $B$

This describes the set of **state pairs**  $\langle s, s' \rangle$  where  $s'$  is a successor of  $s$  in terms of variables  $A \cup A'$ .

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

```

def apply(reached, O):
     $B := T_A(O)$ 
     $B := \text{bdd-intersection}(B, \textit{reached})$ 
    for each  $a \in A$ :
         $B := \text{bdd-forget}(B, a)$ 
    for each  $a \in A$ :
         $B := \text{bdd-rename}(B, a', a)$ 
    return  $B$ 
  
```

This describes the set of state pairs  $\langle s, s' \rangle$  where  $s'$  is a successor of  $s$  and  $s \in \textit{reached}$  in terms of variables  $A \cup A'$ .

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

```

def apply(reached, O):
     $B := T_A(O)$ 
     $B := \text{bdd-intersection}(B, \textit{reached})$ 
    for each  $a \in A$ :
         $B := \text{bdd-forget}(B, a)$ 
    for each  $a \in A$ :
         $B := \text{bdd-rename}(B, a', a)$ 
    return  $B$ 
  
```

This describes the set of states  $s'$  which are successors of some state  $s \in \textit{reached}$  in terms of variables  $A'$ .

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

```

def apply(reached, O):
     $B := T_A(O)$ 
     $B := \text{bdd-intersection}(B, \textit{reached})$ 
    for each  $a \in A$ :
         $B := \text{bdd-forget}(B, a)$ 
    for each  $a \in A$ :
         $B := \text{bdd-rename}(B, a', a)$ 
    return  $B$ 
  
```

This describes the set of states  $s'$  which are successors of some state  $s \in \textit{reached}$  in terms of variables  $A$ .

## The *apply* function

Using the transition relation, we can compute *apply(reached, O)* as follows:

### The *apply* function

```

def apply(reached, O):
     $B := T_A(O)$ 
     $B := \text{bdd-intersection}(B, \textit{reached})$ 
    for each  $a \in A$ :
         $B := \text{bdd-forget}(B, a)$ 
    for each  $a \in A$ :
         $B := \text{bdd-rename}(B, a', a)$ 
    return  $B$ 
  
```

Thus, *apply* indeed computes the set of successors of *reached* using operators *O*.

# Planning with BDDs

## Summary and conclusion

- ▶ **Binary decision diagrams** are a data structure to compactly represent and manipulate sets of valuations.
- ▶ They can be used to implement a blind breadth-first search algorithm in an efficient way.

# Planning with BDDs

## Performance

- ▶ For good performance, we need a **good variable ordering**.
  - ▶ Variables that refer to the same state variable before and after operator application ( $a$  and  $a'$ ) should be **neighbors** in the transition relation BDD.
- ▶ Use **mutexes** to reformulate as a multi-valued task.
  - ▶ Use  $\lceil \log_2 n \rceil$  BDD variables to represent a variable with  $n$  possible values.

With these two ideas, performance is not bad for an algorithm that generates optimal (sequential) plans.

# Planning with BDDs

## Outlook

Is this all there is to it?

- ▶ For classical deterministic planning, **almost**.
  - ▶ Practical implementations also perform **regression** or **bidirectional** searches.
  - ▶ This is only a minor modification.
- ▶ However, BDDs are more commonly used for **non-deterministic** planning.
- ▶ More about this later.