Planning and Optimization

G2. Symbolic Search: BDD Operations and Breadth-First Search

Malte Helmert and Gabriele Röger

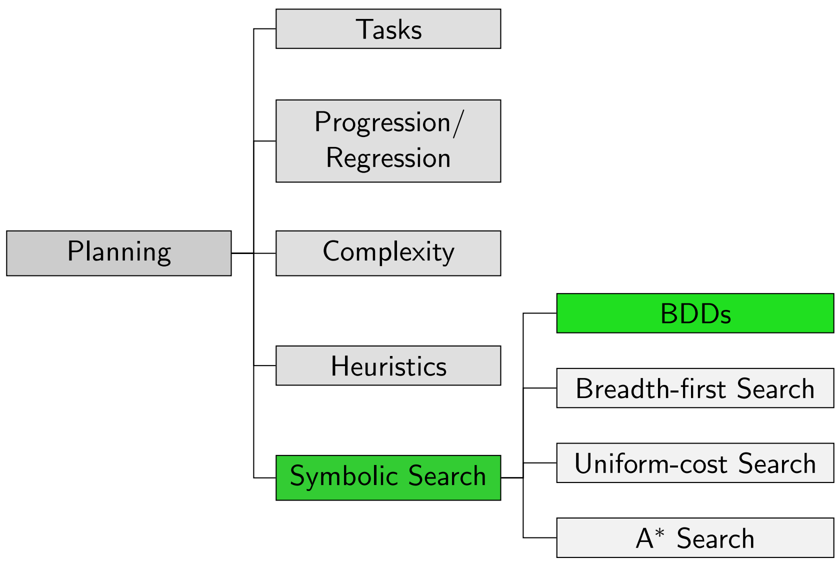
Universität Basel

December 13, 2017

Planning and Optimization

December 13, 2017 — G2. Symbolic Search: BDD Operations and Breadth-First Search

Content of this Course



G2.1 BDD Operations

Reminder: BDD Implementation – Data Structures

Data Structures

- 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 ID -1.
- There are three global vectors to represent the decision variable, the 0- and the 1-successor of non-sink BDDs:
- 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.

BDD Operations: Notations

For convenience, we introduce some additional notations:

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

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(v): Build BDD representing $\{s \mid s(v) = 1\}$.
- 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-forget(B, v): 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 $s[B.\text{var}] = 1$:
        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 $\{s \mid s(v) = 1\}$

def bdd-atom(v):
    return $bdd(v, \mathbf{0}, \mathbf{1})$

- Runtime: $O(1)$

Essential BDD Operations: bdd-state

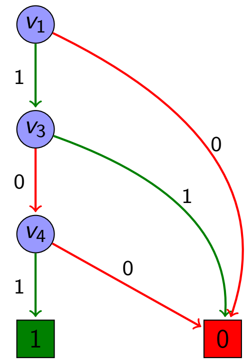
Build BDD representing $\{s\}$

```
def bdd-state(s):
    B := 1
    for each variable v of s, in reverse variable order:
        if s(v) = 1:
            B := bdd(v, 0, B)
        else:
            B := bdd(v, B, 0)
    return B
```

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

Essential BDD Operations: bdd-state Example

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 = 0 and B' = 0: return 0
    else if B = 1 or B' = 1: return 1
    else if B = 0: return B'
    else if B' = 0: return B
    else if B.var < B'.var:
        return bdd(B.var, bdd-union(B.low, B'),
                          bdd-union(B.high, B'))
    else if B.var = B'.var:
        return bdd(B.var, bdd-union(B.low, B'.low),
                          bdd-union(B.high, B'.high))
    else if B.var > B'.var:
        return bdd(B'.var, bdd-union(B, B'.low),
                           bdd-union(B, B'.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-forget (1)

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 V be a set of propositional variables, let S be a set of variable assignments over V, and let $v \in V$.
The existential abstraction of v in S, in symbols $\exists v.S$, is the set of valuations

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

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

Existential abstraction is also called forgetting.

Essential BDD Operations: bdd-forget (2)

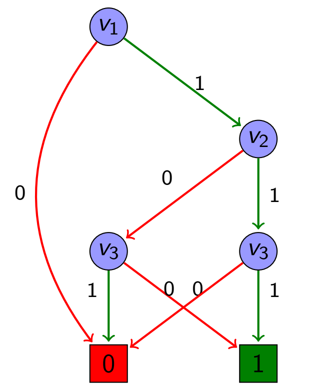
Build BDD representing $\exists v.r(B)$

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

- Runtime: $O(\|B\|^2)$

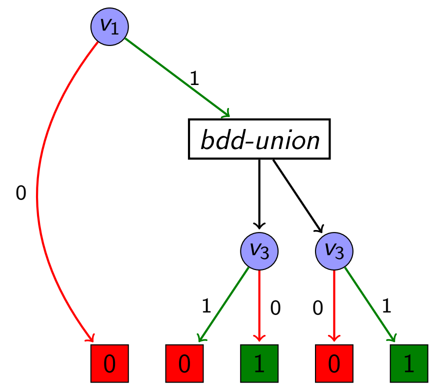
Essential BDD Operations: bdd-forget Example

Example (Forgetting v_2)



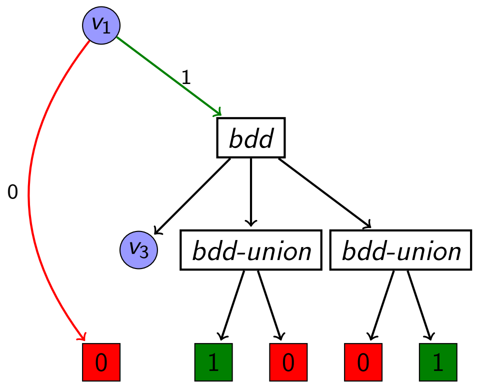
Essential BDD Operations: bdd-forget Example

Example (Forgetting v_2)



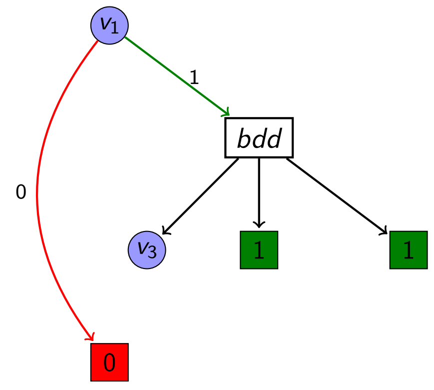
Essential BDD Operations: bdd-forget Example

Example (Forgetting v_2)



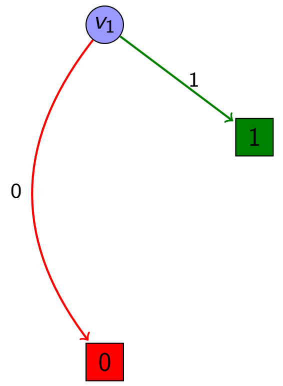
Essential BDD Operations: bdd-forget Example

Example (Forgetting v_2)



Essential BDD Operations: bdd-forget Example

Example (Forgetting v_2)



Derived BDD Operations

We study the following derived operations:

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

Derived 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, *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 $\{ rename(s, v, v') \mid s \in r(B) \}$

def bdd-rename(B, v, v'):
  v-and-v' := *bdd-intersection*(*bdd-atom*(v), *bdd-atom*(v'))
  not-v := *bdd-complement*(*bdd-atom*(v))
  not-v' := *bdd-complement*(*bdd-atom*(v'))
  not-v-and-not-v' := *bdd-intersection*(*not-v*, *not-v'*)
  v-eq-v' := *bdd-union*(*v-and-v'*, *not-v-and-not-v'*)
  return *bdd-forget*(*bdd-intersection*(B, *v-eq-v'*), v)

- Runtime: $O(\|B\|^2)$

Derived BDD Operations: bdd-rename Remarks

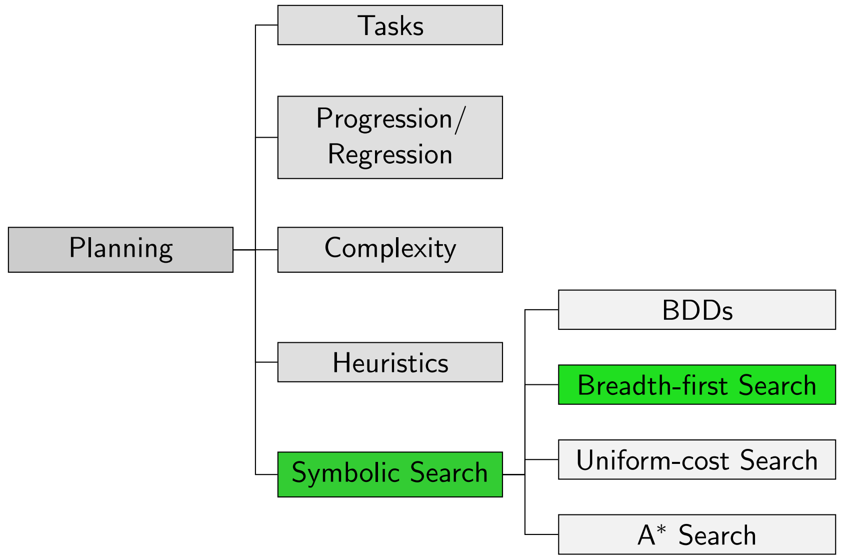
- Renaming sounds like a simple operation.
- Why is it so expensive?

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.

G2.2 Symbolic Breadth-first Search

Content of this Course



Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
- $goal := formula\text{-}to\text{-}set(\gamma)$
- $reached_0 := \{I\}$
- $i := 0$
- **loop**:
  - **if** $reached_i \cap goal \neq \emptyset$:
    - **return** solution found
  - $reached_{i+1} := reached_i \cup apply(reached_i, O)$
  - **if** $reached_{i+1} = reached_i$:
    - **return** no solution exists
  - $i := i + 1$

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
- $goal := formula\text{-}to\text{-}set(\gamma)$
- $reached_0 := \{I\}$
- $i := 0$
- **loop**:
  - **if** $reached_i \cap goal \neq \emptyset$:
    - **return** solution found
  - $reached_{i+1} := reached_i \cup apply(reached_i, O)$
  - **if** $reached_{i+1} = reached_i$:
    - **return** no solution exists
  - $i := i + 1$

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

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
  $goal := formula\text{-}to\text{-}set(\gamma)$
  $reached_0 := \{I\}$
  $i := 0$
  loop:
    if $reached_i \cap goal \neq \emptyset$:
      return solution found
    $reached_{i+1} := reached_i \cup apply(reached_i, O)$
    if $reached_{i+1} = reached_i$:
      return no solution exists
    $i := i + 1$

Use *bdd-state*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
  $goal := formula\text{-}to\text{-}set(\gamma)$
  $reached_0 := \{I\}$
  $i := 0$
  loop:
    if $reached_i \cap goal \neq \emptyset$:
      return solution found
    $reached_{i+1} := reached_i \cup apply(reached_i, O)$
    if $reached_{i+1} = reached_i$:
      return no solution exists
    $i := i + 1$

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

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
  $goal := formula\text{-}to\text{-}set(\gamma)$
  $reached_0 := \{I\}$
  $i := 0$
  loop:
    if $reached_i \cap goal \neq \emptyset$:
      return solution found
    $reached_{i+1} := reached_i \cup apply(reached_i, O)$
    if $reached_{i+1} = reached_i$:
      return no solution exists
    $i := i + 1$

Use *bdd-union*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
  $goal := formula\text{-}to\text{-}set(\gamma)$
  $reached_0 := \{I\}$
  $i := 0$
  loop:
    if $reached_i \cap goal \neq \emptyset$:
      return solution found
    $reached_{i+1} := reached_i \cup apply(reached_i, O)$
    if $reached_{i+1} = reached_i$:
      return no solution exists
    $i := i + 1$

Use *bdd-equals*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

def bfs-progression(V, I, O, γ):
- $goal := formula\text{-}to\text{-}set(\gamma)$
- $reached_0 := \{I\}$
- $i := 0$
- **loop**:
  - **if** $reached_i \cap goal \neq \emptyset$:
    - **return** solution found
  - $reached_{i+1} := reached_i \cup apply(reached_i, O)$
  - **if** $reached_{i+1} = reached_i$:
    - **return** no solution exists
  - $i := i + 1$

How to do this?

The *apply* Function (1)

- We need an operation that, for a set of states $reached_i$ (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 $s \in reached$.
- We have seen something similar already. . .

Translating Operators into Formulae

Definition (Operators in Propositional Logic)

Let o be an operator and V a set of state variables.
Define $\tau_V(o) := pre(o) \wedge \bigwedge_{v \in V}(regr(v, eff(o)) \leftrightarrow v')$.

States that o is applicable and describes when the new value of v, represented by v', is $\mathbf{T}$.

The *apply* Function (2)

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

The *apply* Function (3)

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

The apply function

def apply(*reached*, O):
  $B := T_V(O)$
  $B :=$ *bdd-intersection*(B, *reached*)
  for each $v \in V$:
    $B :=$ *bdd-forget*(B, v)
  for each $v \in V$:
    $B :=$ *bdd-rename*(B, v', v)
  return B

The *apply* Function (3)

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

The apply function

def apply(*reached*, O):
  $B := T_V(O)$
  $B :=$ *bdd-intersection*(B, *reached*)
  for each $v \in V$:
    $B :=$ *bdd-forget*(B, v)
  for each $v \in V$:
    $B :=$ *bdd-rename*(B, v', v)
  return B

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

The *apply* Function (3)

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

The apply function

def apply(*reached*, O):
  $B := T_V(O)$
  $B :=$ *bdd-intersection*(B, *reached*)
  for each $v \in V$:
    $B :=$ *bdd-forget*(B, v)
  for each $v \in V$:
    $B :=$ *bdd-rename*(B, v', v)
  return B

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

The *apply* Function (3)

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

The apply function

def apply(*reached*, O):
  $B := T_V(O)$
  $B :=$ *bdd-intersection*(B, *reached*)
  for each $v \in V$:
    $B :=$ *bdd-forget*(B, v)
  for each $v \in V$:
    $B :=$ *bdd-rename*(B, v', v)
  return B

This describes the set of states s' which are successors of some state $s \in$ *reached* in terms of variables V'.

The *apply* Function (3)

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

The apply function

```
def apply(reached, O):
    B := T_V(O)
    B := bdd-intersection(B, reached)
    for each v ∈ V:
        B := bdd-forget(B, v)
    for each v ∈ V:
        B := bdd-rename(B, v', v)
    return B
```

This describes the set of states s' which are successors of some state $s \in$ *reached* in terms of variables V.

The *apply* Function (3)

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

The apply function

```
def apply(reached, O):
    B := T_V(O)
    B := bdd-intersection(B, reached)
    for each v ∈ V:
        B := bdd-forget(B, v)
    for each v ∈ V:
        B := bdd-rename(B, v', v)
    return B
```

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

Plan Extraction

We can construct a plan from the BDDs $reached_i$ (set given as parameter $reached_*$):

Construct Plan

```
def construct_plan(I, O, γ, reached_*, i_max):
    goal := BDD for γ
    s := arbitrary state from bdd-intersection(goal, reached_{i_max})
    π := ⟨⟩
    for i = i_max − 1 to 0:
        for o ∈ O:
            p := BDD for regr(s, o)
            if c := bdd-intersection(p, reached_i) ≠ 0:
                s := arbitrary state from c
                π := ⟨o⟩π
                break
    return π
```


Remarks

BDDs can be used to implement a blind breadth-first search algorithm in an efficient way.

- For good performance, we need a good variable ordering.
  - Variables that refer to the same state variable before and after operator application (v and v') 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.

G2.3 Summary

Summary

- 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.