

Planning and Optimization

C7. Symbolic Search: Full Algorithm

Malte Helmert and Gabriele Röger

Universität Basel

October 17, 2022

Planning and Optimization

October 17, 2022 — C7. Symbolic Search: Full Algorithm

C7.1 Basic BDD Operations

C7.2 Formulas and Singletons

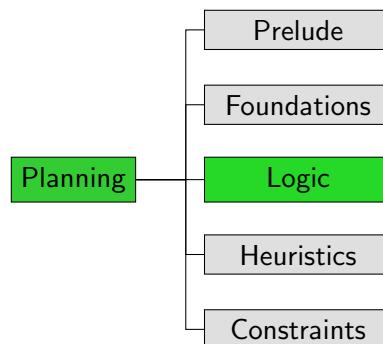
C7.3 Renaming

C7.4 Symbolic Breadth-first Search

C7.5 Discussion

C7.6 Summary

Content of this Course



Devising a Symbolic Search Algorithm

- ▶ We now put the pieces together to build a symbolic search algorithm for propositional planning tasks.
- ▶ use BDDs as a **black box** data structure:
 - ▶ care about provided operations and their time complexity
 - ▶ do not care about their internal implementation
- ▶ Efficient implementations are available as libraries, e.g.:
 - ▶ **CUDD**, a high-performance BDD library
 - ▶ **libbdd**, shipped with Ubuntu Linux

C7.1 Basic BDD Operations

BDD Operations: Preliminaries

- ▶ All BDDs work on a **fixed** and **totally ordered** set of propositional variables.
- ▶ Complexity of operations given in terms of:
 - ▶ k , the number of **BDD variables**
 - ▶ $\|B\|$, the number of **nodes** in the BDD B

BDD Operations (1)

BDD operations: **logical/set atoms**

- ▶ **bdd-true()**: build BDD representing all assignments
 - ▶ in logic: \top
 - ▶ time complexity: $O(1)$
- ▶ **bdd-false()**: build BDD representing \emptyset
 - ▶ in logic: \perp
 - ▶ time complexity: $O(1)$
- ▶ **bdd-atom(v)**: build BDD representing $\{s \mid s(v) = 1\}$
 - ▶ in logic: v
 - ▶ time complexity: $O(1)$

BDD Operations (2)

BDD operations: **logical/set connectives**

- ▶ **bdd-complement(B)**: build BDD representing $\overline{r(B)}$
 - ▶ in logic: $\neg\varphi$
 - ▶ time complexity: $O(\|B\|)$
- ▶ **bdd-union(B, B')**: build BDD representing $r(B) \cup r(B')$
 - ▶ in logic: $(\varphi \vee \psi)$
 - ▶ time complexity: $O(\|B\| \cdot \|B'\|)$
- ▶ analogously:
 - ▶ **bdd-intersection(B, B')**: $r(B) \cap r(B')$, $(\varphi \wedge \psi)$
 - ▶ **bdd-setdifference(B, B')**: $r(B) \setminus r(B')$, $(\varphi \wedge \neg\psi)$
 - ▶ **bdd-implies(B, B')**: $\overline{r(B)} \cup r(B')$, $(\varphi \rightarrow \psi)$
 - ▶ **bdd-equiv(B, B')**: $(r(B) \cap r(B')) \cup (\overline{r(B)} \cap \overline{r(B')})$, $(\varphi \leftrightarrow \psi)$

BDD Operations (3)

BDD operations: **Boolean tests**

- ▶ **bdd-includes**(B, I): return **true** iff $I \in r(B)$
 - ▶ in logic: $I \models \varphi?$
 - ▶ time complexity: $O(k)$
- ▶ **bdd-equals**(B, B'): return **true** iff $r(B) = r(B')$
 - ▶ in logic: $\varphi \equiv \psi?$
 - ▶ time complexity: $O(1)$ (due to canonical representation)

Conditioning: Formulas

The last two basic BDD operations are a bit more unusual and require some preliminary remarks.

Conditioning a variable v in a **formula** φ to **T** or **F**, written $\varphi[\mathbf{T}/v]$ or $\varphi[\mathbf{F}/v]$, means restricting v to a particular truth value:

Examples:

- ▶ $(A \wedge (B \vee \neg C))[\mathbf{T}/B] = (A \wedge (\mathbf{T} \vee \neg C)) \equiv A$
- ▶ $(A \wedge (B \vee \neg C))[\mathbf{F}/B] = (A \wedge (\perp \vee \neg C)) \equiv A \wedge \neg C$

Conditioning: Sets of Assignments

We can define the same operation for sets of assignments S :
 $S[\mathbf{F}/v]$ and $S[\mathbf{T}/v]$ restrict S to elements with the given value for v and **remove** v from the domain of definition:

Example:

- ▶ $S = \{ \{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}, C \mapsto \mathbf{F}\}, \{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{F}\}, \{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{T}\} \}$
- ↔ $S[\mathbf{T}/B] = \{ \{A \mapsto \mathbf{T}, C \mapsto \mathbf{F}\}, \{A \mapsto \mathbf{T}, C \mapsto \mathbf{T}\} \}$

Forgetting

Forgetting (a.k.a. **existential abstraction**) is similar to conditioning: we allow **either** truth value for v and remove the variable.

We write this as $\exists v \varphi$ (for formulas) and $\exists v S$ (for sets).

Formally:

- ▶ $\exists v \varphi = \varphi[\mathbf{T}/v] \vee \varphi[\mathbf{F}/v]$
- ▶ $\exists v S = S[\mathbf{T}/v] \cup S[\mathbf{F}/v]$

Forgetting: Example

Examples:

$$\blacktriangleright S = \{ \{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}, C \mapsto \mathbf{F}\}, \\ \{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{F}\}, \\ \{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}, C \mapsto \mathbf{T}\} \}$$

$$\rightsquigarrow \exists B S = \{ \{A \mapsto \mathbf{F}, C \mapsto \mathbf{F}\}, \\ \{A \mapsto \mathbf{T}, C \mapsto \mathbf{F}\}, \\ \{A \mapsto \mathbf{T}, C \mapsto \mathbf{T}\} \}$$

$$\rightsquigarrow \exists C S = \{ \{A \mapsto \mathbf{F}, B \mapsto \mathbf{F}\}, \\ \{A \mapsto \mathbf{T}, B \mapsto \mathbf{T}\} \}$$

BDD Operations (4)

BDD operations: **conditioning and forgetting**

- ▶ **bdd-condition**(B, v, t) where $t \in \{\mathbf{T}, \mathbf{F}\}$:
build BDD representing $r(B)[t/v]$

- ▶ in logic: $\varphi[t/v]$
- ▶ time complexity: $O(\|B\|)$

- ▶ **bdd-forget**(B, v):
build BDD representing $\exists v r(B)$

- ▶ in logic: $\exists v \varphi \quad (= \varphi[\mathbf{T}/v] \vee \varphi[\mathbf{F}/v])$
- ▶ time complexity: $O(\|B\|^2)$

C7.2 Formulas and Singletons

Formulas to BDDs

- ▶ With the logical/set operations, we can convert propositional **formulas** φ into BDDs representing the **models** of φ .
- ▶ We denote this computation with **bdd-formula**(φ).
- ▶ Each individual logical connective takes **polynomial** time, but converting a full formula of length n can take $O(2^n)$ time. (How is this possible?)

Singleton BDDs

- ▶ We can convert a **single truth assignment** I into a BDD representing $\{I\}$ by computing the conjunction of all literals true in I (using `bdd-atom`, `bdd-complement` and `bdd-intersection`).
- ▶ We denote this computation with `bdd-singleton(I)`.
- ▶ When done in the correct order, this takes time $O(k)$.

C7.3 Renaming

Renaming

We will need to support one final operation on formulas: **renaming**.

Renaming X to Y in formula φ , written $\varphi[X \rightarrow Y]$, means **replacing** all occurrences of X by Y in φ .

We require that Y is **not present** in φ initially.

Example:

$$\text{▶ } \varphi = (A \wedge (B \vee \neg C))$$

$$\rightsquigarrow \varphi[A \rightarrow D] = (D \wedge (B \vee \neg C))$$

How Hard Can That Be?

- ▶ For formulas, renaming is a **simple** (linear-time) operation.
- ▶ For a BDD B , it is equally simple ($O(\|B\|)$) when renaming between variables that are **adjacent** in the variable order.
- ▶ In general, it requires $O(\|B\|^2)$, using the equivalence $\varphi[X \rightarrow Y] \equiv \exists X(\varphi \wedge (X \leftrightarrow Y))$

C7.4 Symbolic Breadth-first Search

Planning Task State Variables vs. BDD Variables

Consider propositional planning task $\langle V, I, O, \gamma \rangle$ with states S .

In symbolic planning, we have **two BDD variables** v and v' for every state variable $v \in V$ of the planning task.

- ▶ use **unprimed** variables v to describe sets of **states**:
 $\{s \in S \mid \text{some property}\}$
- ▶ use combinations of **unprimed** and **primed** variables v, v' to describe sets of **state pairs**:
 $\{\langle s, s' \rangle \mid \text{some property}\}$

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

```

def bfs-progression( $V, I, O, \gamma$ ):
   $goal\_states := models(\gamma)$ 
   $reached_0 := \{I\}$ 
   $i := 0$ 
  loop:
    if  $reached_i \cap goal\_states \neq \emptyset$ :
      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( $V, I, O, \gamma$ ):
   $goal\_states := models(\gamma)$ 
   $reached_0 := \{I\}$ 
   $i := 0$ 
  loop:
    if  $reached_i \cap goal\_states \neq \emptyset$ :
      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-formula*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

```

def bfs-progression( $V, I, O, \gamma$ ):
    goal_states := models( $\gamma$ )
    reached0 := { $I$ }
    i := 0
    loop:
        if  $reached_i \cap goal\_states \neq \emptyset$ :
            return solution found
        reachedi+1 := reachedi  $\cup$  apply(reachedi,  $O$ )
        if  $reached_{i+1} = reached_i$ :
            return no solution exists
        i := i + 1

```

Use *bdd-singleton*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

```

def bfs-progression( $V, I, O, \gamma$ ):
    goal_states := models( $\gamma$ )
    reached0 := { $I$ }
    i := 0
    loop:
        if  $reached_i \cap goal\_states \neq \emptyset$ :
            return solution found
        reachedi+1 := reachedi  $\cup$  apply(reachedi,  $O$ )
        if  $reached_{i+1} = reached_i$ :
            return no solution exists
        i := i + 1

```

Use *bdd-intersection*, *bdd-false*, *bdd-equals*.

Breadth-first Search with Progression and BDDs

Progression Breadth-first Search

```

def bfs-progression( $V, I, O, \gamma$ ):
    goal_states := models( $\gamma$ )
    reached0 := { $I$ }
    i := 0
    loop:
        if  $reached_i \cap goal\_states \neq \emptyset$ :
            return solution found
        reachedi+1 := reachedi  $\cup$  apply(reachedi,  $O$ )
        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( $V, I, O, \gamma$ ):
    goal_states := models( $\gamma$ )
    reached0 := { $I$ }
    i := 0
    loop:
        if  $reached_i \cap goal\_states \neq \emptyset$ :
            return solution found
        reachedi+1 := reachedi  $\cup$  apply(reachedi,  $O$ )
        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( $V, I, O, \gamma$ ):
     $goal\_states := models(\gamma)$ 
     $reached_0 := \{I\}$ 
     $i := 0$ 
    loop:
        if  $reached_i \cap goal\_states \neq \emptyset$ :
            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* (given as a BDD)
- ▶ and a set of operators *O*
- ▶ computes the set of states (as a BDD) that result from applying some operator $o \in O$ in some state $s \in reached$.

We have seen something similar already...

Translating Operators into Formulas

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 how

- ▶ the **new value of v** , represented by v' ,
- ▶ must relate to the **old state**, described by variables V .

The *apply* Function (2)

- ▶ The formula $\tau_V(o)$ describes all transitions $s \xrightarrow{o} s'$
 - ▶ induced by a **single** operator o
 - ▶ in terms of variables V describing s
 - ▶ and variables V' describing s' .
- ▶ 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 $V \cup V'$) with **bdd-formula**.
- ▶ 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 := TV(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
```

The *apply* Function (3)

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

The *apply* function

```
def apply(reached, O):
  B := TV(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 **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 := TV(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 state pairs $\langle s, s' \rangle$ where s' is a successor of s and $s \in \text{reached}$ 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 := TV(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** $s \in \text{reached}$ 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 := TV(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 $s \in \textit{reached}$ 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 := TV(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 .

C7.5 Discussion

- ▶ This completes the discussion of a (basic) symbolic search algorithm for classical planning.
- ▶ We ignored the aspect of **solution extraction**. This needs some extra work, but is not a major challenge.
- ▶ In practice, some steps can be performed slightly more efficiently, but these are comparatively minor details.

Discussion

Variable Orders

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.

Extensions

Symbolic search can be extended to . . .

- ▶ **regression and bidirectional search:**
this is very easy and often effective
- ▶ **uniform-cost search:**
requires some work, but not too difficult in principle
- ▶ **heuristic search:**
requires a heuristic representable as a BDD;
has not really been shown to outperform blind symbolic search

Literature

-  Randal E. Bryant.
Graph-Based Algorithms for Boolean Function Manipulation.
IEEE Transactions on Computers 35.8, pp. 677–691, 1986.
Reduced ordered BDDs.
-  Kenneth L. McMillan.
Symbolic Model Checking.
PhD Thesis, 1993.
Symbolic search with BDDs.
-  Álvaro Torralba.
Symbolic Search and Abstraction Heuristics
for Cost-Optimal Planning.
PhD Thesis, 2015.
State of the art of symbolic search planning.

C7.6 Summary

Summary

- ▶ **Symbolic search** operates on **sets of states** instead of individual states as in explicit-state search.
- ▶ State sets and transition relations can be represented as **BDDs**.
- ▶ Based on this, we can implement a blind breadth-first search in an efficient way.
- ▶ A good variable ordering is crucial for performance.