Principles of AI Planning

13. Planning with binary decision diagrams

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel and Robert Mattmüller and David Speck

January 15th, 2018

Binary decision diagrams

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.

Basic Ideas

- Come up with a good data structure for sets of states.
- Hope: (at least some) exponentially large state sets can be represented as polynomial-size data structures.
- Simulate a standard search algorithm like breadth-first search using these set representations.

Breadth-first search with progression and state sets

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
  goal := *models*(γ)
  reached := $\{I\}$
  loop:
    if *reached* $\cap$ *goal* $\neq \emptyset$:
      return solution found
    new-reached := *reached* $\cup$ *image*(*reached*, O)
    if *new-reached* = *reached*:
      return no solution exists
    reached := *new-reached*

$\rightsquigarrow$ If we can implement operations *models*, $\{I\}$, $\cap$, $\neq \emptyset$, $\cup$, *img* 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.

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
$s \in S$?	$O(k \log \lvert S \rvert)$	$O(k)$	$O(\lVert S \rVert)$
$S := S \cup \{s\}$	$O(k \log \lvert S \rvert + \lvert S \rvert)$	$O(k)$	$O(k)$
$S := S \setminus \{s\}$	$O(k \log \lvert S \rvert + \lvert S \rvert)$	$O(k)$	$O(k)$
$S \cup S'$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(1)$
$S \cap S'$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(1)$
$S \setminus S'$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(k \lvert S \rvert + k \lvert S' \rvert)$	$O(1)$
$\overline{S}$	$O(k2^k)$	$O(k2^k)$	$O(1)$
$\{s \mid s(v) = 1\}$	$O(k2^k)$	$O(k2^k)$	$O(1)$
$S = \emptyset$?	$O(1)$	$O(1)$	co-NP-complete
$S = S'$?	$O(k \lvert S \rvert)$	$O(k \lvert S \rvert)$	co-NP-complete
$\lvert S \rvert$	$O(1)$	$O(1)$	#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: $S \neq \emptyset$?, $S = S'$?

Canonical Representations

- 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.
- Reduced ordered binary decision diagrams (BDDs) are an example of an efficient canonical representation.

Performance characteristics

Formulae vs. BDDs

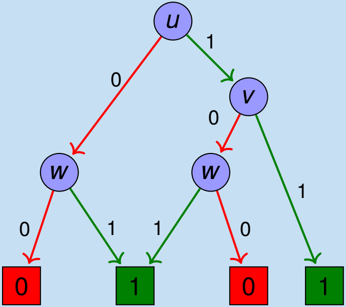
Let k be the number of state variables, $|S|$ the number of states in S and $\|S\|$ the size of the representation of S.

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

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

BDD example

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



Definition (BDD)

Let V be a set of propositional variables.

A binary decision diagram (BDD) over V 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 $v \in V$ and have exactly two outgoing arcs, labeled 0 and 1.

UNI FREIBURG

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.

Testing whether a BDD includes a variable assignment

def bdd-includes(*B*: BDD, *I*: variable assignment):
    Set *n* to the root of *B*.
    while *n* is not a sink:
        Set *v* to the decision variable of *n*.
        Set *n* to the *I*(*v*)-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 V. The set represented by B, in symbols $r(B)$ consists of all variable assignments $I : V \to \{0, 1\}$ for which *bdd-includes*(B, I) returns true.

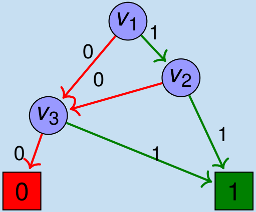
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- $\neg v_1 \land \neg v_2 \land \neg v_3$
- $\neg v_1 \land \neg v_2 \land v_3$
- $\neg v_1 \land v_2 \land \neg v_3$
- $\neg v_1 \land v_2 \land v_3$
- $v_1 \land \neg v_2 \land \neg v_3$
- $v_1 \land \neg v_2 \land v_3$
- $v_1 \land v_2 \land \neg v_3$
- $v_1 \land v_2 \land v_3$

Which states are represented by this BDD?



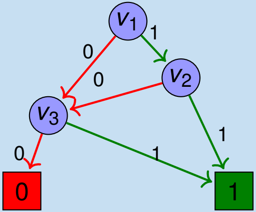
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- $\neg v_1 \wedge \neg v_2 \wedge v_3$
- $\neg v_1 \wedge v_2 \wedge \neg v_3$
- $\neg v_1 \wedge v_2 \wedge v_3$
- $v_1 \wedge \neg v_2 \wedge \neg v_3$
- $v_1 \wedge \neg v_2 \wedge v_3$
- $v_1 \wedge v_2 \wedge \neg v_3$
- $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



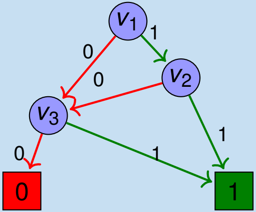
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge \neg v_2 \wedge v_3$
- $\neg v_1 \wedge v_2 \wedge \neg v_3$
- $\neg v_1 \wedge v_2 \wedge v_3$
- $v_1 \wedge \neg v_2 \wedge \neg v_3$
- $v_1 \wedge \neg v_2 \wedge v_3$
- $v_1 \wedge v_2 \wedge \neg v_3$
- $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



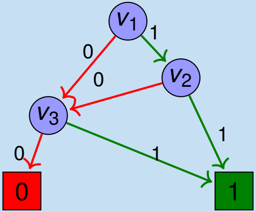
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \land \neg v_2 \land \neg v_3$
- ✓ $\neg v_1 \land \neg v_2 \land v_3$
- ✗ $\neg v_1 \land v_2 \land \neg v_3$
- $\neg v_1 \land v_2 \land v_3$
- $v_1 \land \neg v_2 \land \neg v_3$
- $v_1 \land \neg v_2 \land v_3$
- $v_1 \land v_2 \land \neg v_3$
- $v_1 \land v_2 \land v_3$

Which states are represented by this BDD?



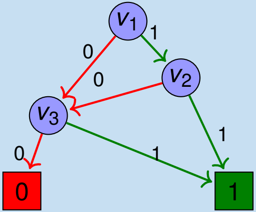
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \land \neg v_2 \land \neg v_3$
- ✓ $\neg v_1 \land \neg v_2 \land v_3$
- ✗ $\neg v_1 \land v_2 \land \neg v_3$
- ✓ $\neg v_1 \land v_2 \land v_3$
- $v_1 \land \neg v_2 \land \neg v_3$
- $v_1 \land \neg v_2 \land v_3$
- $v_1 \land v_2 \land \neg v_3$
- $v_1 \land v_2 \land v_3$

Which states are represented by this BDD?



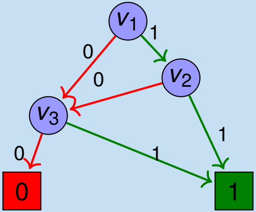
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✔ $\neg v_1 \wedge \neg v_2 \wedge v_3$
- ✗ $\neg v_1 \wedge v_2 \wedge \neg v_3$
- ✔ $\neg v_1 \wedge v_2 \wedge v_3$
- ✗ $v_1 \wedge \neg v_2 \wedge \neg v_3$
- ■ $v_1 \wedge \neg v_2 \wedge v_3$
- ■ $v_1 \wedge v_2 \wedge \neg v_3$
- ■ $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



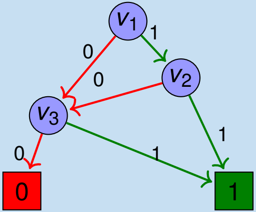
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge \neg v_2 \wedge v_3$
- ✗ $\neg v_1 \wedge v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge v_2 \wedge v_3$
- ✗ $v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $v_1 \wedge \neg v_2 \wedge v_3$
- ■ $v_1 \wedge v_2 \wedge \neg v_3$
- ■ $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



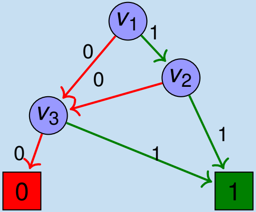
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge \neg v_2 \wedge v_3$
- ✗ $\neg v_1 \wedge v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge v_2 \wedge v_3$
- ✗ $v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $v_1 \wedge \neg v_2 \wedge v_3$
- ✓ $v_1 \wedge v_2 \wedge \neg v_3$
- $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



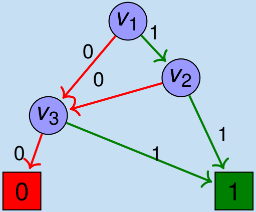
Set represemted by a BDD

Example

Possible states for $V = \{v_1, v_2, v_3\}$

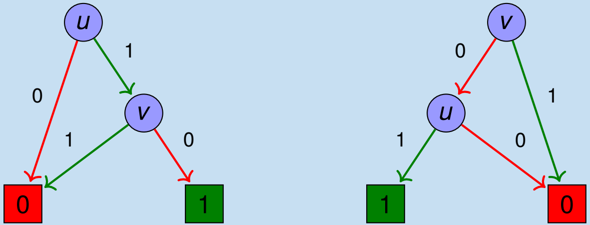
- ✗ $\neg v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge \neg v_2 \wedge v_3$
- ✗ $\neg v_1 \wedge v_2 \wedge \neg v_3$
- ✓ $\neg v_1 \wedge v_2 \wedge v_3$
- ✗ $v_1 \wedge \neg v_2 \wedge \neg v_3$
- ✓ $v_1 \wedge \neg v_2 \wedge v_3$
- ✓ $v_1 \wedge v_2 \wedge \neg v_3$
- ✓ $v_1 \wedge v_2 \wedge v_3$

Which states are represented by this BDD?



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

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



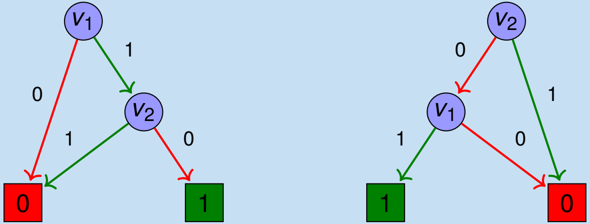
Both BDDs represent the same state set, namely the singleton set $\{\{u \mapsto 1, v \mapsto 0\}\}$.

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

Definition (ordered BDD)

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

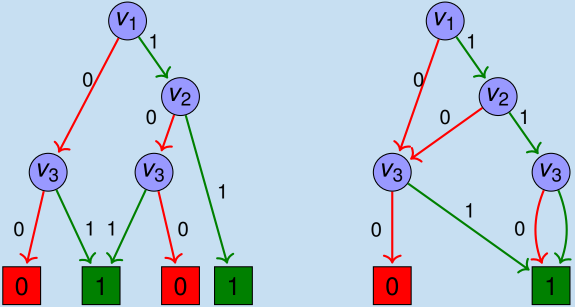
Ordered and unordered BDD



According to our definitions, the left BDD is ordered, the right one is not.

Note: Often in literature, a BDD is called ordered if on all paths from the root to a sink variables appear in the same order.

Two equivalent BDDs that can be reduced



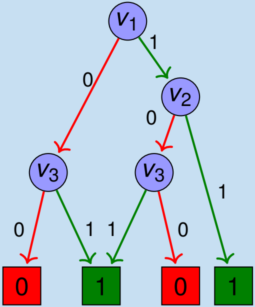
- Ordered BDDs are not canonical: Both ordered BDDs represent the same set.
- However, ordered BDDs can easily be made canonical.

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

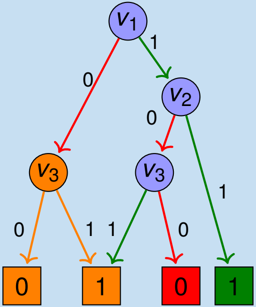
Definition (Isomorphism reduction)

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.

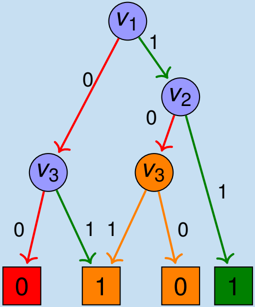
Isomorphism reduction



Isomorphism reduction



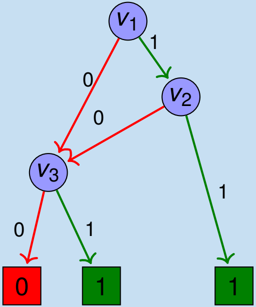
Isomorphism reduction



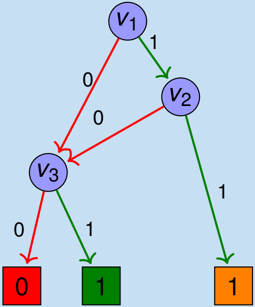
Reduced ordered BDDs

Reductions

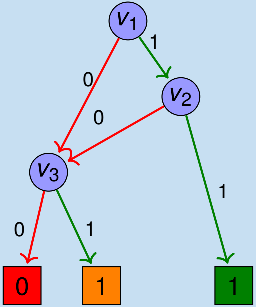
Isomorphism reduction



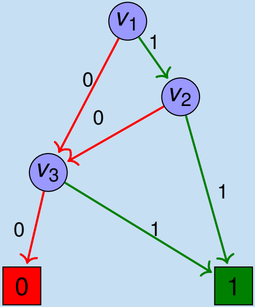
Isomorphism reduction



Isomorphism reduction



Isomorphism reduction

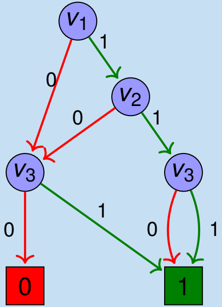


There are two important operations on BDDs that do not change the set represented by it:

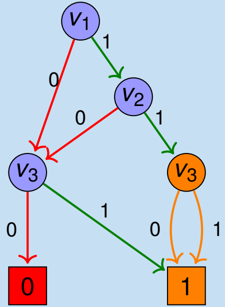
Definition (Shannon reduction)

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.

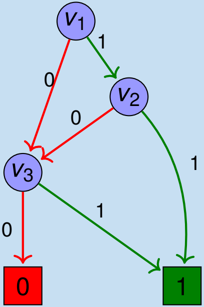
Shannon reduction



Shannon reduction



Shannon reduction



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.

$\rightsquigarrow$ Reduced ordered BDDs are the canonical representation we were looking for.

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

BDD operations

Goal: 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

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: $\bot$
  - 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: $O(\|B\|)$ (or $O(1)$)
- 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:

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

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:

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

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

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:

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

Forgetting: Example

Examples:

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

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

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

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

Formulas to BDDs

- With the logical/set operations, we can convert propositional formulas φ into BDDs representing the models of φ.
  - bdd-atom, bdd-complement, bdd-union, ...
- 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.
  - bdd-atom, bdd-complement and bdd-intersection
- We denote this computation with bdd-singleton(I).
- When done in the correct order, this takes time $O(k)$.

Renaming

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

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

We require that Y is not present in φ initially.

Example:

- $\varphi = (A \wedge (B \vee \neg C))$

$\leadsto$ $\varphi[A \to D] = (D \wedge (B \vee \neg C))$

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 $\varphi[X \to Y] \equiv \exists X(\varphi \wedge (X \leftrightarrow Y))$

Symbolic Breadth-first Search

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

```
def bfs-progression(V, I, O, γ):
    goal := models(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ image(reached, O)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
    goal := *models*(γ)
    reached := $\{I\}$
    loop:
        if *reached* $\cap$ *goal* $\neq \emptyset$:
            return solution found
        new-reached := *reached* $\cup$ *image*(*reached*, O)
        if *new-reached* = *reached*:
            return no solution exists
        reached := *new-reached*

Use *bdd-formula* (*bdd-complement*, *bdd-union* and *bdd-intersection*).

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
    goal := *models*(γ)
    reached := $\{I\}$
    loop:
        if $reached \cap goal \neq \emptyset$:
            return solution found
        new-reached := *reached* $\cup$ *image*(*reached*, O)
        if *new-reached* = *reached*:
            return no solution exists
        reached := *new-reached*

Use *bdd-singleton* (*bdd-complement*, *bdd-union* and *bdd-intersection*).

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
  $goal := models(\gamma)$
  $reached := \{I\}$
  loop:
    if $reached \cap goal \neq \emptyset$:
      return solution found
    $new\text{-}reached := reached \cup image(reached, O)$
    if $new\text{-}reached = reached$:
      return no solution exists
    $reached := new\text{-}reached$

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

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
  goal := *models*(γ)
  reached := $\{I\}$
  loop:
    if *reached* $\cap$ *goal* $\neq \emptyset$:
      return solution found
    new-reached := *reached* $\cup$ *image*(*reached*, O)
    if *new-reached* = *reached*:
      return no solution exists
    reached := *new-reached*

Use *bdd-union*.

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
    goal := *models*(γ)
    reached := $\{I\}$
    loop:
        if *reached* $\cap$ *goal* $\neq \emptyset$:
            return solution found
        new-reached := *reached* $\cup$ *image*(*reached*, O)
        if *new-reached* = *reached*:
            return no solution exists
        reached := *new-reached*

Use *bdd-equals*.

Symbolic Breadth-first search with progression and BDDs

Symbolic progression breadth-first search

def bfs-progression(V, I, O, γ):
    $goal := models(\gamma)$
    $reached := \{I\}$
    loop:
        if $reached \cap goal \neq \emptyset$:
            return solution found
        $new\text{-}reached := reached \cup image(reached, O)$
        if $new\text{-}reached = reached$:
            return no solution exists
        $reached := new\text{-}reached$

How to do this?

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

Definition (operators in propositional logic)

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

$$\chi \quad (1)$$

$$\bigwedge_{v \in V} (EPC_v(e) \vee (v \wedge \neg EPC_{\neg v}(e))) \leftrightarrow v' \quad (2)$$

$$\bigwedge_{v \in V} \neg(EPC_v(e) \wedge EPC_{\neg v}(e)) \quad (3)$$

(1) The precondition of o is satisfied

(2) The new value of v, represented by v', is 1 if it became 1 or if the old value was 1 and it did not become 0.

(3) None of the state variables is assigned both 0 and 1.

Note: (1) + (3) encodes applicability of the operator.

The *image* function
Idea

- 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 $T_V(O)$.

- $V = \{v_1, v_2\}$ and $V' = \{v_1', v_2'\}$
- $O = \{\langle v_1, \neg v_1 \rangle\}$

Transition Relation

$$\begin{aligned} T_V(O) = \bigvee_{o \in O} \tau_V(o) &= \tau_V(\langle v_1, \neg v_1 \rangle) \\ &= v_1 \\ &\quad \wedge (EPC_{v_1}(\neg v_1) \vee (v_1 \wedge \neg EPC_{\neg v_1}(\neg v_1))) \leftrightarrow v_1') \\ &\quad \wedge (EPC_{v_2}(\neg v_1) \vee (v_2 \wedge \neg EPC_{\neg v_2}(\neg v_1))) \leftrightarrow v_2') \\ &\quad \wedge (\neg(EPC_{v_1}(\neg v_1) \wedge EPC_{\neg v_1}(\neg v_1))) \\ &\quad \wedge (\neg(EPC_{v_2}(\neg v_1) \wedge EPC_{\neg v_2}(\neg v_1))) \\ &= ? \end{aligned}$$

- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\}$

Transition Relation

$$
\begin{aligned}
T_V(O) = \bigvee_{o \in O} \tau_V(o) = \tau_V(\langle v_1, \neg v_1 \rangle) \\
&= v_1 \\
&\quad \wedge (EPC_{v_1}(\neg v_1) \vee (v_1 \wedge \neg EPC_{\neg v_1}(\neg v_1))) \leftrightarrow v'_1) \\
&\quad \wedge (EPC_{v_2}(\neg v_1) \vee (v_2 \wedge \neg EPC_{\neg v_2}(\neg v_1))) \leftrightarrow v'_2) \\
&\quad \wedge (\neg(EPC_{v_1}(\neg v_1) \wedge EPC_{\neg v_1}(\neg v_1))) \\
&\quad \wedge (\neg(EPC_{v_2}(\neg v_1) \wedge EPC_{\neg v_2}(\neg v_1))) \\
&= v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)
\end{aligned}
$$

Transition Relation as BDD

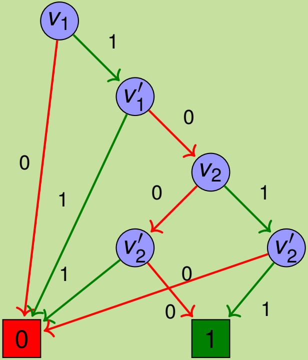
Example

- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

Transition Relation as BDD

States:

- $v_1 \wedge \neg v'_1 \wedge v_2 \wedge v'_2$
- $v_1 \wedge \neg v'_1 \wedge \neg v_2 \wedge \neg v'_2$



The *image* function

Definition

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

The image function

def image(*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

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

The image function

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

This describes the set of state pairs in terms of variables $V \cup V'$.

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

The image function

def image(*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 *image* function

Definition

Using the transition relation, we can compute $image(reached, O)$ as follows:

The image function

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

$B := T_V(O)$

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

for each $v \in V$:

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

for each $v \in V$:

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

return B

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

Using the transition relation, we can compute $image(reached, O)$ as follows:

The image function

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

    $B := T_V(O)$

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

    for each $v \in V$:

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

    for each $v \in V$:

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

    return B

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

The *image* function

Definition

Using the transition relation, we can compute $image(reached, O)$ as follows:

The image function

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

  $B := T_V(O)$

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

  for each $v \in V$:

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

  for each $v \in V$:

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

  return B

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

The *image* function

Example

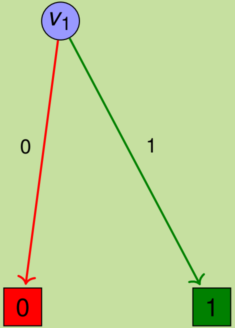
- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

Let *reached* = v_1

States:

- $v_1 \wedge \neg v_2$
- $v_1 \wedge v_2$

How many states considering $V \cup V'$?

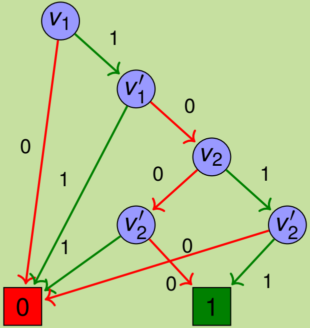


- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

B = bdd-intersection($T_V(O)$, *reached* = v_1)

States:

- $v_1 \wedge \neg v'_1 \wedge v_2 \wedge v'_2$
- $v_1 \wedge \neg v'_1 \wedge \neg v_2 \wedge \neg v'_2$



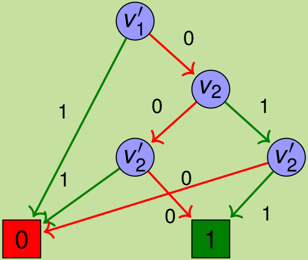
The *image* function
Example

- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

B = bdd-forget(B, v_1)

States:

- $\neg v'_1 \wedge v_2 \wedge v'_2$
- $\neg v'_1 \wedge \neg v_2 \wedge \neg v'_2$



The *image* function

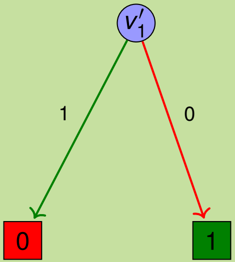
Example

- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \land \neg v'_1 \land (v_2 \leftrightarrow v'_2)$

B = bdd-forget(B, v_2)

States:

- $\neg v'_1 \land v'_2$
- $\neg v'_1 \land \neg v'_2$

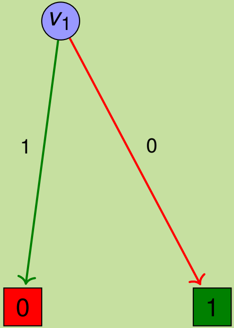


- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

B = bdd-rename(B, v'_1, v_1)

States:

- $\neg v_1 \wedge v'_2$
- $\neg v_1 \wedge \neg v'_2$



The *image* function

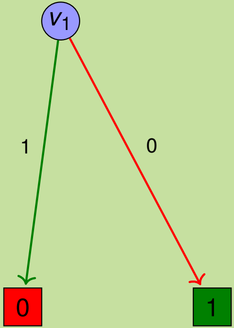
Example

- $V = \{v_1, v_2\}$ and $V' = \{v'_1, v'_2\}$
- $O = \{\langle v_1, \neg v_1 \rangle\} \rightsquigarrow T_V(O) = v_1 \wedge \neg v'_1 \wedge (v_2 \leftrightarrow v'_2)$

B = bdd-rename(B, v'_2, v_2)

States:

- $\neg v_1 \wedge v_2$
- $\neg v_1 \wedge \neg v_2$



Discussion

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.

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.

Finite-Domain Variables and Variable Orders

The algorithm can easily be extended to FDR tasks by using $\lceil \log_2 n \rceil$ BDD variables to represent a state variable with n possible values.

- Variables related to the same FDR variable should be kept together in the BDD variable ordering (but still interleaving primed and unprimed variables).
- Automatic conversion from STRIPS to SAS$^+$ was first explored in the context of symbolic search.
- It was found critical for performance.

Extensions

Overview

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?

- represent heuristic as multiple BDDs $H_0, H_1, \ldots$
- split BDD B according to their h-value
  - bdd-intersection(B, H_0), bdd-intersection(B, H_1), ...
  - can be costly
- can increase or decrease the sizes of the BDDs
  - in the worst case exponentially
  - even with the perfect heuristic h^*
- no theoretical guarentees
- Does not pay off in practice!
- explicit search + symbolic heuristics: very effective

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.
- David Speck, Florian Geißer and Robert Mattmüller.
  When Perfect is not Good Enough: On the Search Behaviour of Symbolic Heuristic Search
  Proc. ICAPS 2020, 2020. ⇒ Symbolic Heuristic Search.

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.