Principles of AI Planning

13. Planning with binary decision diagrams

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel and Robert Mattmüller

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.

Breadth-first search with progression and state sets

Progression breadth-first search

def bfs-progression(V, I, O, γ):
  $goal$:= $formula\text{-}to\text{-}set(\gamma)$
  $reached$:= $\{I\}$
  loop:
    if $reached \cap goal \neq \emptyset$:
      return solution found
    $new\text{-}reached := reached \cup \bigcup_{o \in O} img_o(reached)$
    if $new\text{-}reached = reached$:
      return no solution exists
    $reached := new\text{-}reached$

$\leadsto$ 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 χ and ψ in constant time.

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

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

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

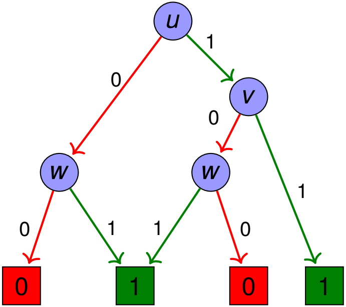
Definition (BDD)

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.

BDD example

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



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 semantics

Testing whether a BDD includes a valuation

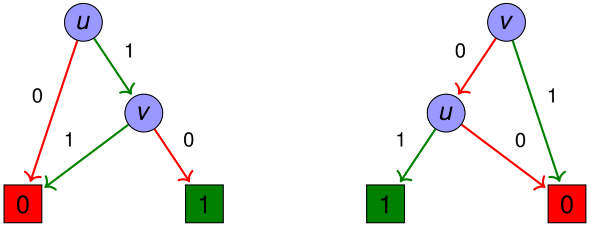
def bdd-includes(B: BDD, v: valuation):
  Set n to the root of B.
  while n is not a sink:
    Set a to the decision variable of n.
    Set n to the $v(a)$-successor of n.
  return true if n is labeled 1, false if it is labeled 0.

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 $v : A \to \{0, 1\}$ for which *bdd-includes*(B, v) returns true.

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

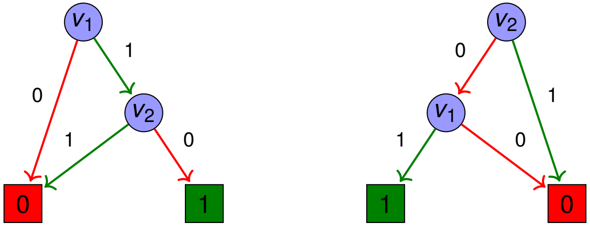
Definition

- 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 $v_1, v_2, v_3, \ldots$ and assume the ordering $v_i \prec v_j$ iff $i < j$.

Definition (ordered BDD)

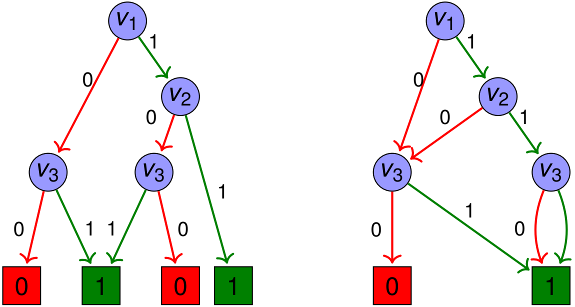
A BDD is ordered iff for each arc from an internal node with decision variable u to an internal node with decision variable v, we have $u \prec v$.

Ordered and unordered BDD



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

Two equivalent BDDs that can be reduced



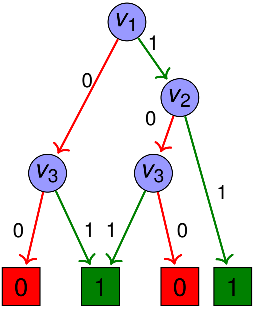
- Ordered BDDs are not canonical: Both ordered BDDs represent the same set.
- However, ordered BDDs can easily be made canonical.

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

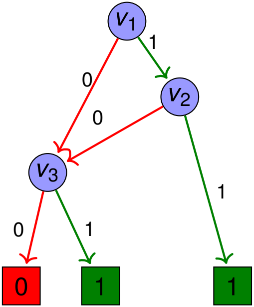
Definition (Isomorphism reduction)

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.

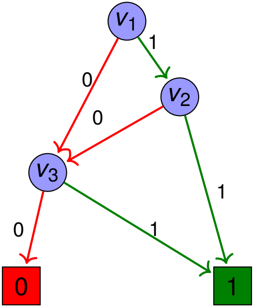
Isomorphism reduction



Isomorphism reduction



Isomorphism reduction

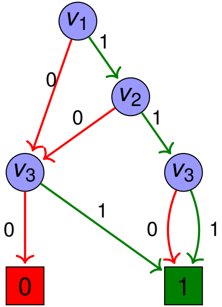


There are two important operations on BDDs that do not change the set represented by it:

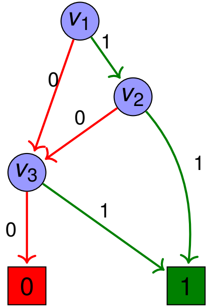
Definition (Shannon reduction)

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



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.

$\rightsquigarrow$ Reduced ordered BDDs are the canonical representation we were looking for.
From now on, we simply say BDD for reduced ordered BDD.

BDD operations

- 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

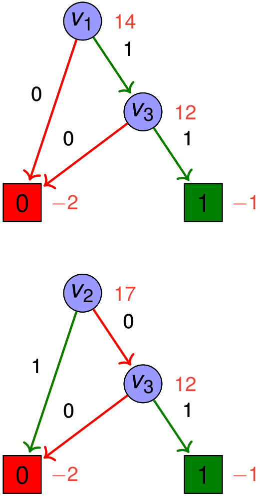
- Every BDD (including sub-BDDs) B is represented by a single natural number $id(B)$ called its ID.
  - The zero BDD has ID -2.
  - The one BDD has ID -1.
  - Other BDDs have IDs ≥ 0.
- The BDD operations must satisfy the following invariant: Two BDDs with different ID are never identical.

Data structures

- There are three global vectors (dynamic arrays) to represent information on non-sink BDDs with ID $i \geq 0$:
  - $\mathit{var}[i]$ denotes the decision variable.
  - $\mathit{low}[i]$ denotes the ID of the 0-successor.
  - $\mathit{high}[i]$ 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 $\langle \mathit{var}[i], \mathit{low}[i], \mathit{high}[i] \rangle$ 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 $\langle v, l, h \rangle \notin \mathit{lookup}$:
        Set i to a new unused ID.
        $\mathit{var}[i], \mathit{low}[i], \mathit{high}[i] := v, l, h$
        $\mathit{lookup}[\langle v, l, h \rangle] := i$
    return $\mathit{lookup}[\langle v, l, h \rangle]$

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 **0** := *zero*(), **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(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-forget(B, a): Described later.

- 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 $s[B.\mathrm{var}] = 1$:
        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 $bdd(a, \mathbf{0}, \mathbf{1})$

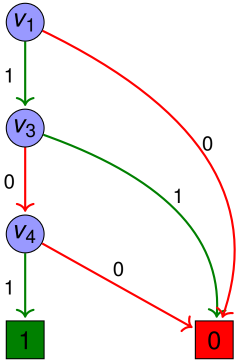
- Runtime: $O(1)$

Build BDD representing $\{s\}$

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

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

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



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.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: $O(\|B\| \cdot \|B'\|)$

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

def bdd-complement(B):
    if $B = \mathbf{0}$:
        return 1
    else if $B = \mathbf{1}$:
        return 0
    else:
        return *bdd*(B.var, *bdd-complement*(B.low),
                    bdd-complement(B.high))

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

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

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

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

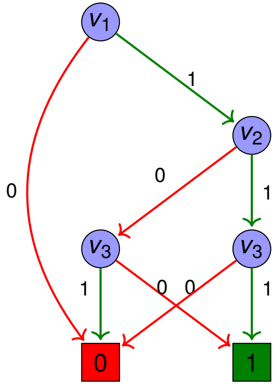
Existential abstraction is also called forgetting.

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

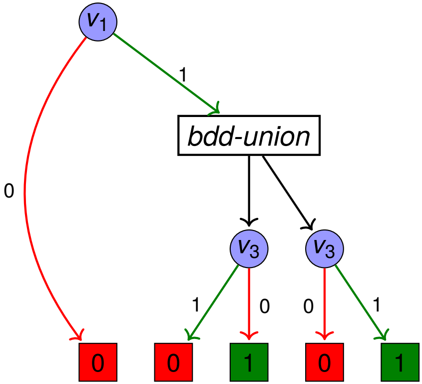
def bdd-forget(B, v):
    if B = **0 or** B = **1 or** B.var $\succ v$:
        return B
    else if B.var $\prec v$:
        return *bdd*(B.var, *bdd-forget*(B.low, v),
                        bdd-forget(B.high, v))
    else:
        return *bdd-union*(B.low, B.high)

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

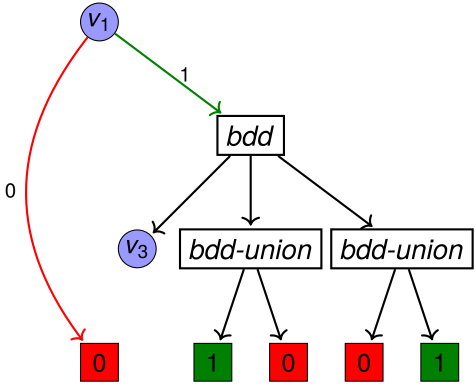
Forgetting v_2



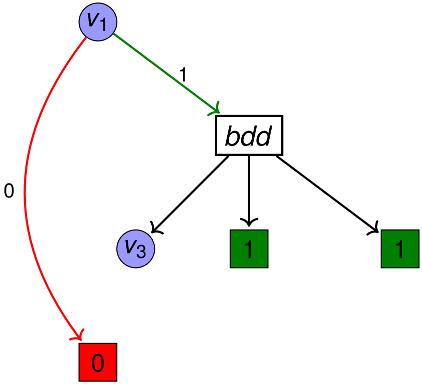
Forgetting v_2



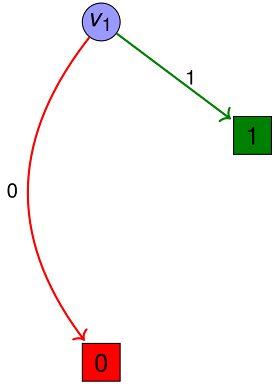
Forgetting v_2



Forgetting v_2



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 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, *bdd-complement*(B'))

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

Test $r(B) = \emptyset$

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

- Runtime: $O(1)$

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.

Planning with BDDs

Breadth-first search with progression and BDDs

Progression breadth-first search

def bfs-progression(V, I, O, γ):
    goal := *formula-to-set*(γ)
    reached := $\{I\}$
    loop:
        if $reached \cap goal \neq \emptyset$:
            return solution found
        $new\text{-}reached := reached \cup \bigcup_{o \in O} img_o(reached)$
        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(V, I, O, γ):
  goal := *formula-to-set*(γ)
  reached := $\{I\}$
  loop:
    if $reached \cap goal \neq \emptyset$:
      return solution found
    $new\text{-}reached := reached \cup \bigcup_{o \in O} img_o(reached)$
    if $new\text{-}reached = reached$:
      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(V, I, O, γ):
    goal := *formula-to-set*(γ)
    reached := $\{I\}$
    loop:
        if $reached \cap goal \neq \emptyset$:
            return solution found
        $new\text{-}reached := reached \cup \bigcup_{o \in O} img_o(reached)$
        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(V, I, O, γ):
    goal := *formula-to-set*(γ)
    reached := $\{I\}$
    loop:
        if $\mathit{reached} \cap \mathit{goal} \neq \emptyset$:
            return solution found
        $\mathit{new\text{-}reached} := \mathit{reached} \cup \bigcup_{o \in O} \mathit{img}_o(\mathit{reached})$
        if *new-reached* = *reached*:
            return no solution exists
        reached := *new-reached*

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

Progression breadth-first search

def bfs-progression(V, I, O, γ):
  $goal$:= $formula\text{-}to\text{-}set(\gamma)$
  $reached$:= $\{I\}$
  loop:
    if $reached \cap goal \neq \emptyset$:
      return solution found
    $new\text{-}reached$:= $reached \cup \bigcup_{o \in O} img_o(reached)$
    if $new\text{-}reached = reached$:
      return no solution exists
    $reached$:= $new\text{-}reached$

Use *bdd-union*.

UNI FREIBURG

Progression breadth-first search

def bfs-progression(V, I, O, γ):
  goal := *formula-to-set*(γ)
  reached := $\{I\}$
  loop:
    if *reached* $\cap$ *goal* $\neq \emptyset$:
      return solution found
    new-reached := *reached* $\cup \bigcup_{o \in O} img_o(reached)$
    if *new-reached = reached*:
      return no solution exists
    reached := *new-reached*

Use *bdd-equals*.

Progression breadth-first search

def bfs-progression(V, I, O, γ):
  goal := *formula-to-set*(γ)
  reached := $\{I\}$
  loop:
    if *reached* $\cap$ *goal* $\neq \emptyset$:
      return solution found
    new-reached := *reached* $\cup \bigcup_{o \in O} img_o(reached)$
    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 $s \in reached$.
- 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 A a set of state variables. Define $\tau_A(o)$ as the conjunction of

$$
\begin{aligned}
&\chi && (1)\\
&\textstyle\bigwedge_{a \in A} (EPC_a(e) \vee (a \wedge \neg EPC_{\neg a}(e))) \leftrightarrow a' && (2)\\
&\textstyle\bigwedge_{a \in A} \neg (EPC_a(e) \wedge EPC_{\neg a}(e)) && (3)
\end{aligned}
$$

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*):

$B := T_A(O)$

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

for each $a \in A$:

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

for each $a \in A$:

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

return B

The *apply* function

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

The apply function

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

$B := T_A(O)$

$B :=$ *bdd-intersection*$(B, reached)$

for each $a \in A$:

$B :=$ *bdd-forget*(B, a)

for each $a \in A$:

$B :=$ *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 $s \in reached$ 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*):
  $B := T_A(O)$
  $B := \mathit{bdd\text{-}intersection}(B, \mathit{reached})$
  for each $a \in A$:
    $B := \mathit{bdd\text{-}forget}(B, a)$
  for each $a \in A$:
    $B := \mathit{bdd\text{-}rename}(B, a', a)$
  return B

This describes the set of states s' which are successors of some state $s \in \mathit{reached}$ 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*):
  $B := T_A(O)$
  $B := bdd\text{-}intersection(B, reached)$
  for each $a \in A$:
    $B := bdd\text{-}forget(B, a)$
  for each $a \in A$:
    $B := bdd\text{-}rename(B, a', a)$
  return B

This describes the set of states s' which are successors of some state $s \in reached$ 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*):
  $B := T_A(O)$
  $B := \textit{bdd-intersection}(B, \textit{reached})$
  for each $a \in A$:
    $B := \textit{bdd-forget}(B, a)$
  for each $a \in A$:
    $B := \textit{bdd-rename}(B, a', a)$
  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.

- 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.

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 also often used for non-deterministic planning.