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

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪_{o ∈ O} img_o(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

⇒ If we can implement operations `formula-to-set`, `{I}`, ∩, ≠ ∅, ∪, apply and = efficiently, this is a reasonable algorithm.
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 $\chi \land \psi$ from $\chi$ and $\psi$ 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 $S$.

| Operation               | Sorted vector $|S|$ | Hash table $|S|$ | Formula $|S|$ |
|-------------------------|----------------------|-----------------|------------------|
| $s \in S?$              | $O(k \log |S|)$      | $O(k)$          | $O(\|S\|)$      |
| $S := S \cup \{s\}$    | $O(k \log |S| + |S|)$ | $O(k)$          | $O(k)$           |
| $S := S \setminus \{s\}$| $O(k \log |S| + |S|)$ | $O(k)$          | $O(k)$           |
| $S \cup S'$             | $O(k |S| + k |S'|)$   | $O(k |S| + k |S'|)$ | $O(1)$          |
| $S \cap S'$             | $O(k |S| + k |S'|)$   | $O(k |S| + k |S'|)$ | $O(1)$          |
| $S \setminus S'$        | $O(k |S| + k |S'|)$   | $O(k |S| + k |S'|)$ | $O(1)$          |
| $\bar{S}$               | $O(k 2^k)$           | $O(k 2^k)$      | $O(1)$           |
| $\{s \mid s(a) = 1\}$  | $O(k 2^k)$           | $O(k 2^k)$      | $O(1)$           |
| $S = \emptyset?$        | $O(1)$               | $O(1)$          | co-NP-complete   |
| $S = S'$                 | $O(k |S|)$           | $O(k |S|)$      | co-NP-complete   |
| $|S|$                    | $O(1)$               | $O(1)$          | #P-complete      |
Which operations are important?

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

This makes equality tests expensive.

⇒ We are interested in **canonical representations**, i.e. representations for which there is only one possible representation for every state set. **Binary decision diagrams** (BDDs) are an example of an efficient canonical representation.
Let \( k \) be the number of state variables, \(|S|\) the number of states in \( S \) and \( \|S\| \) the size of the representation of \( S \).

<table>
<thead>
<tr>
<th></th>
<th>Formula</th>
<th>BDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s \in S? )</td>
<td>( O(|S|) )</td>
<td>( O(k) )</td>
</tr>
<tr>
<td>( S := S \cup {s} )</td>
<td>( O(k) )</td>
<td>( O(k) )</td>
</tr>
<tr>
<td>( S := S \setminus {s} )</td>
<td>( O(k) )</td>
<td>( O(k) )</td>
</tr>
<tr>
<td>( S \cup S' )</td>
<td>( O(1) )</td>
<td>( O(|S| |S'|) )</td>
</tr>
<tr>
<td>( S \cap S' )</td>
<td>( O(1) )</td>
<td>( O(|S| |S'|) )</td>
</tr>
<tr>
<td>( S \setminus S' )</td>
<td>( O(1) )</td>
<td>( O(|S| |S'|) )</td>
</tr>
<tr>
<td>( \overline{S} )</td>
<td>( O(1) )</td>
<td>( O(|S|) )</td>
</tr>
<tr>
<td>( { s \mid s(a) = 1 } )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( S = \emptyset? )</td>
<td>co-NP-complete</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( S = S'? )</td>
<td>co-NP-complete</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>( |S| )</td>
<td>#P-complete</td>
<td>( O(|S|) )</td>
</tr>
</tbody>
</table>

**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.
Possible BDD for \((u \land v) \lor w\)
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

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

Definition (set represented by a BDD)

Let $B$ be a BDD over variables $A$. The set represented by $B$, in symbols $r(B)$ consists of all valuations $v : A \rightarrow \{0, 1\}$ for which $\text{bdd-includes}(B, v)$ returns true.
In general, BDDs are not a canonical representation for sets of valuations. Here is a simple counter-example ($A = \{u, v\}$):

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 $A$ 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 BDDs

Ordered and unordered BDD

The left BDD is ordered, the right one is not.
Reduced ordered BDDs
Are ordered BDDs canonical?

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.
Reduced ordered BDDs

Isomorphism reduction

\[
\begin{array}{c}
\text{v}_1 \\
\begin{array}{c}
\text{v}_2 \\
\begin{array}{c}
\text{v}_3 \\
\text{v}_3
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
0 \\
1
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
1 \\
0
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
0 \\
1
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
0 \\
1
\end{array}
\]
Reduced ordered BDDs

Isomorphism reduction

![Diagram of isomorphism reduction](image-url)
Isomorphism reduction

Reduced ordered BDDs

January 30th, 2013
Isomorphism reduction

\[ \begin{align*}
  v_1 & \rightarrow 1 \\
  v_2 & \rightarrow 0 \\
  v_3 & \rightarrow 0 \\
\end{align*} \]

\[ \begin{align*}
  v_1 & \rightarrow 0 \\
  v_2 & \rightarrow 0 \\
  v_3 & \rightarrow 1 \\
\end{align*} \]
Isomorphism reduction

Diagram showing a reduced ordered BDD with nodes $v_1$, $v_2$, and $v_3$, labeled with values 0 and 1 along the paths.
Isomorphism reduction
Reduced ordered BDDs

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
Reduced ordered BDDs

Shannon reduction

BDDs
Motivation
Definition
Operations
BDD Planning
Reduced ordered BDDs

Shannon reduction

\[
\begin{aligned}
\text{Shannon reduction} & \\
\begin{array}{c}
\text{BDDs} \\
\text{Motivation} \\
\text{Definition} \\
\text{Operations} \\
\text{BDD} \\
\text{Planning}
\end{array}
\end{aligned}
\]
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 $B$.

$\Rightarrow$ 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 $\geq 0$.

- The BDD operations must satisfy the following invariant: Two BDDs with different ID are never identical.
Efficient BDD implementation

Data structures

- There are three global vectors (dynamic arrays) to represent information on non-sink BDDs with ID $i \geq 0$:
  - $var[i]$ denotes the decision variable.
  - $low[i]$ denotes the ID of the 0-successor.
  - $high[i]$ denotes the ID of the 1-successor.

- There is some mechanism that keeps track of IDs that are currently unused (garbage collection, reference counting). This can be implemented without amortized overhead.

- There is a global hash table $lookup$ which maps, for each ID $i \geq 0$ representing a BDD in use, the triple $\langle var[i], low[i], high[i] \rangle$ to $i$.
  - Randomized hashing allows constant-time access in the expected case. More sophisticated methods allow deterministic constant-time access.
Efficient BDD implementation

Data structures example

```
\[
\begin{array}{c|c|c|c|c}
\text{formula} & \text{ID} i & \text{var}[i] & \text{low}[i] & \text{high}[i] \\
\hline
\bot & -2 & - & - & - \\
\top & -1 & - & - & - \\
\var_3 & 12 & 3 & -2 & -1 \\
\var_1 \land \var_3 & 14 & 1 & -2 & 12 \\
\neg \var_2 \land \var_3 & 17 & 2 & 12 & -2 \\
\end{array}
\]
```
Efficient BDD implementation

Data structures example

\[
\begin{array}{c|c|c|c|c}
\text{formula} & \text{ID } i & \text{var}[i] & \text{low}[i] & \text{high}[i] \\
\hline
\bot & -2 & - & - & - \\
\top & -1 & - & - & - \\
v_3 & 12 & 3 & -2 & -1 \\
v_1 \land v_3 & 14 & 1 & -2 & 12 \\
\neg v_2 \land v_3 & 17 & 2 & 12 & -2 \\
\end{array}
\]
Efficient BDD implementation
Data structures example

\[
\begin{array}{c|c|c|c|c}
\text{formula} & \text{ID } i & \text{var}[i] & \text{low}[i] & \text{high}[i] \\
\hline
\bot & -2 & - & - & - \\
\top & -1 & - & - & - \\
v_3 & 12 & 3 & -2 & -1 \\
v_1 \land v_3 & 14 & 1 & -2 & 12 \\
\neg v_2 \land v_3 & 17 & 2 & 12 & -2 \\
\end{array}
\]

\[
\begin{align*}
0 & \rightarrow 0,0,1 \\
1 & \rightarrow 1,0,1 \\
v_1 & \rightarrow 0,1,1 \\
v_2 & \rightarrow 0,1,0 \\
v_3 & \rightarrow 1,0,1 \\
\end{align*}
\]
Efficient BDD implementation

Data structures example

<table>
<thead>
<tr>
<th>formula</th>
<th>ID i</th>
<th>var[i]</th>
<th>low[i]</th>
<th>high[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>−2</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>⊤</td>
<td>−1</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>v₃</td>
<td>12</td>
<td>3</td>
<td>−2</td>
<td>−1</td>
</tr>
<tr>
<td>v₁ ∧ v₃</td>
<td>14</td>
<td>1</td>
<td>−2</td>
<td>12</td>
</tr>
<tr>
<td>¬v₂ ∧ v₃</td>
<td>17</td>
<td>2</td>
<td>12</td>
<td>−2</td>
</tr>
</tbody>
</table>
Efficient BDD implementation
Data structures example
Efficient BDD implementation

Data structures example

<table>
<thead>
<tr>
<th>formula</th>
<th>ID $i$</th>
<th>$\text{var}[i]$</th>
<th>$\text{low}[i]$</th>
<th>$\text{high}[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$-2$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$-1$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$12$</td>
<td>$3$</td>
<td>$-2$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$v_1 \land v_3$</td>
<td>$14$</td>
<td>$1$</td>
<td>$-2$</td>
<td>$12$</td>
</tr>
<tr>
<td>$\neg v_2 \land v_3$</td>
<td>$17$</td>
<td>$2$</td>
<td>$12$</td>
<td>$-2$</td>
</tr>
</tbody>
</table>
Efficient BDD implementation
Data structures example

<table>
<thead>
<tr>
<th>formula</th>
<th>ID $i$</th>
<th>var[$i$]</th>
<th>low[$i$]</th>
<th>high[$i$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$-2$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$\top$</td>
<td>$-1$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$12$</td>
<td>$3$</td>
<td>$-2$</td>
<td>$-1$</td>
</tr>
<tr>
<td>$v_1 \land v_3$</td>
<td>$14$</td>
<td>$1$</td>
<td>$-2$</td>
<td>$12$</td>
</tr>
<tr>
<td>$\neg v_2 \land v_3$</td>
<td>$17$</td>
<td>$2$</td>
<td>$12$</td>
<td>$-2$</td>
</tr>
</tbody>
</table>
Core BDD operations

Building the zero BDD

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

Building the one BDD

```python
def one():
    return -1
```
Core BDD operations

Building other BDDs

```python
def bdd(v: variable, l: ID, h: ID):
    if l == h:
        return l
    if ⟨v, l, h⟩ ∉ lookup:
        Set i to a new unused ID.
        var[i], low[i], high[i] := v, l, h
        lookup[⟨v, l, h⟩] := i
    return lookup[⟨v, l, h⟩]
```

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.
For convenience, we introduce some additional notations:

- We define $0 := \text{zero}()$, $1 := \text{one}()$.
- We write $\text{var}$, $\text{low}$, $\text{high}$ as attributes:
  - $B.\text{var}$ for $\text{var}[B]$
  - $B.\text{low}$ for $\text{low}[B]$
  - $B.\text{high}$ for $\text{high}[B]$
Essential vs. derived BDD operations

We distinguish between

- **essential BDD operations**, which are implemented directly on top of `zero`, `one` and `bdd`, and
- **derived BDD operations**, which are implemented in terms of the essential operations.
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 \( s \in r(B) \)

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

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

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

```python
def bdd-equals(B, B_prime):
    return B == B_prime
```

- Runtime: $O(1)$
Build BDD representing \( \{ s \mid s(a) = 1 \} \)

```python
def bdd_atom(a):
    return bdd(a, 0, 1)
```

- Runtime: \( O(1) \)
Essential BDD operations

**bdd-state**

Build BDD representing \{s\}

```python
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

\[
\text{bdd-state}\left( \{ v_1 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1 \} \right)
\]
Essential BDD operations

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

\[ \text{bdd-state}(\{v_1 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1\}) \]
Essential BDD operations

\[ \text{bdd-state}(\{v_1 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1\}) \]
Essential BDD operations

bdd-state: Example

\[ \text{bdd-state}(\{v_1 \mapsto 1, v_3 \mapsto 0, v_4 \mapsto 1\}) \]
Build BDD representing $r(B) \cup r(B')$

```python
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: $O(\|B\| \cdot \|B'\|)$
```
Essential BDD operations

**bdd-complement**

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

```python
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:** \( O(\|B\|) \)
The last essential BDD operation is a bit more unusual, but we will need it for defining the semantics of operator application.

**Definition (Existential abstraction)**

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

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

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

Existential abstraction is also called **forgetting**.
Build BDD representing $\exists v. r(B)$

```python
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: $O(\|B\|^2)$
Essential BDD operations

bdd-forget: Example

Forgetting $v_2$
**Essential BDD operations**

**bdd-forget: Example**

**Forgetting $v_2$**

![BDD Diagram](image)

- **bdd-union**
- $v_1$ connected to $v_3$ with $v_1 = 1$
- $v_3$ connected to $v_3$ with $v_3 = 0$
- $v_3$ connected to $v_3$ with $v_3 = 1$
- $v_1$ connected to $v_3$ with $v_1 = 0$
- $v_3$ connected to $v_3$ with $v_3 = 0$
- $v_3$ connected to $v_3$ with $v_3 = 1$
- $v_3$ connected to $v_3$ with $v_3 = 1$
- $v_3$ connected to $v_3$ with $v_3 = 1$

Forgetting $v_2$

```
0

v_1

1
```

```
0 1 0 0 1

v_3

bdd

bdd-union  bdd-union

bdd-union
```
Essential BDD operations

bdd-forget: Example

Forgetting $v_2$
Essential BDD operations

bdd-forget: Example

Forgetting $v_2$
Derived BDD operations

We study the following derived operations:

- **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 $\{\text{rename}(s, v, v') \mid s \in r(B) \}$,
  where $\text{rename}(s, v, v')$ is the valuation $s$ with variable $v$
  renamed to $v'$.
    - If variable $v'$ occurs in $B$ already, the result is undefined.
Derived BDD operations

bdd-intersection, bdd-setdifference

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

```python
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 $r(B) \setminus r(B')$

```python
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 $r(B) = \emptyset$

```python
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)\}

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

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪ o∈O img_o(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

January 30th, 2013 B. Nebel, R. Mattmüller – AI Planning
Breadth-first search with progression and BDDs

Progression breadth-first search

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪_{o∈O} img_o(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

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

January 30th, 2013 B. Nebel, R. Mattmüller – AI Planning
Progression breadth-first search

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪_{o∈O} img_{o}(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

Use `bdd-state`.
Progression breadth-first search

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reaching := reached ∪ \bigcup_{o \in O} \text{img}_o(reached)
        if new-reaching = reached:
            return no solution exists
        reached := new-reaching
```

Use `bdd-intersection, bdd-isempty`. 
Breadth-first search with progression and BDDs

Progression breadth-first search

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪_{o∈O} img_o(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

Use `bdd-union`.
Breadth-first search with progression and BDDs

Progression breadth-first search

```python
def bfs-progression(V, I, O, γ):
    goal := formula-to-set(γ)
    reached := {I}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ ∪_{o∈O} img_o(reached)
        if new-reached = reached:
            return no solution exists
        reached := new-reached
```

Use `bdd-equals`. 
Breadth-first search with progression and BDDs

**Progression breadth-first search**

```python
def bfs-progression(V, l, O, γ):
    goal := formula-to-set(γ)
    reached := {l}
    loop:
        if reached ∩ goal ≠ ∅:
            return solution found
        new-reached := reached ∪ \bigcup_{o \in O} \text{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 $s \in \textit{reached}$.

- We have seen something similar already...
Translating operators into formulae

Definition (operators in propositional logic)

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

\[
\begin{align*}
  c & \quad \text{(1)} \\
  \bigwedge_{a \in A} (EPC_a(e) \lor (a \land \neg EPC_{\neg a}(e))) & \leftrightarrow a' \quad \text{(2)} \\
  \bigwedge_{a \in A} \neg (EPC_a(e) \land EPC_{\neg a}(e)) & \quad \text{(3)}
\end{align*}
\]

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 $T_A(O)$. 
Using the transition relation, we can compute \( \text{apply}(\text{reached}, O) \) as follows:

```python
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
```
The *apply* function

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

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

This describes the set of state pairs $\langle s, s' \rangle$ where $s'$ is a successor of $s$ in terms of variables $A \cup A'$. 
Using the transition relation, we can compute $\text{apply}(\text{reached}, O)$ as follows:

The apply function

```python
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 $s \in \text{reached}$ in terms of variables $A \cup A'$. 
Using the transition relation, we can compute $\text{apply}(\text{reached}, O)$ as follows:

```python
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 states $s'$ which are successors of some state $s \in \text{reached}$ in terms of variables $A'$. 
Using the transition relation, we can compute \( \text{apply}(\text{reached}, O) \) as follows:

```python
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 states \( s' \) which are successors of some state \( s \in \text{reached} \) in terms of variables \( A \).
Using the transition relation, we can compute $\text{apply}(\text{reached}, O)$ as follows:

**The apply function**

```python
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
```

Thus, $\text{apply}$ indeed computes the set of successors of $\text{reached}$ using operators $O$. 

The function `apply` computes the set of successors of a state $\text{reached}$ using operators $O$. It starts by applying the transition function $T_A$ to $O$, then intersects the result with $\text{reached}$. For each action $a$, it forgets $a$ and renames it to $a'$ to avoid conflicts, and then returns the result.
**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 \((a\text{ and }a')\) should be **neighbors** in the transition relation BDD.

- Use **mutexes** to reformulate as a multi-valued task.
  - Use \([\log_2 n]\) 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 more commonly used for *non-deterministic* planning (not covered in this course).