Método de *Davis-Putnam*

- Verificar a validade de uma FNC φ consiste em determinar se existe algum modelo $\mathcal{M}$ tal que $\mathcal{M} \models \varphi$.
- Uma estratégia "*divide and conquer*" consiste em dividir a classe de modelos conforme estes contenham (ou não) um dado símbolo proposicional.
- Assim, vamos exprimir a validade da fórmula φ com base na validade de duas fórmulas φ_1, φ_2 tais que, para um dado símbolo proposicional p:

$$\mathcal{M} \models \varphi \quad \text{sse} \quad \begin{cases} \mathcal{M} \models \varphi_1 & \text{, se } p \in \mathcal{M} \\ \mathcal{M} \models \varphi_2 & \text{, se } p \notin \mathcal{M} \end{cases}$$

- As fórmulas φ_1 e φ_2 serão obtidas (de uma forma muito simples) da fórmula original φ.

Conectiva de *Shannon*

- A obervação básica para determinar φ_1 e φ_2 por simplificação de φ consiste em notar que, quando dispomos da informação que uma dada proposição é válida (ou não), podemos eliminar essa proposição da FNC φ.
- Considere-se por exemplo a FNC $\varphi = [[a, b], [\neg a, b], [a, \neg b], [\neg a, \neg b]]$. Vejamos como simplificar esta FNC com base no pressuposto que a é válido (ou não):
  - Admitamos que $a \in \mathcal{M}$. Nesse caso temos podemos simplificar φ observando que certas cláusulas são simplificáveis ou mesmo redundantes.

$$\mathcal{M} \models [[a, b], [\neg a, b], [a, \neg b], [\neg a, \neg b]] \quad \text{sse} \quad \mathcal{M} \models [[b], [\neg b]]$$

  - Quando que $a \notin \mathcal{M}$, temos:

$$\mathcal{M} \models [[a, b], [\neg a, b], [a, \neg b], [\neg a, \neg b]] \quad \text{sse} \quad \mathcal{M} \models [[b], [\neg b]]$$

- É conveniente introduzir uma notação específica para denotar esta partição dependente da validade de uma proposição p:

$$\varphi = p \rightarrow \varphi_1 ; \varphi_2 \qquad (\textit{conectiva de Shannon}).$$

Partição de FNCs (*Splitting*)

Definição

Seja S um conjunto de cláusulas e p um símbolo proposicional. Define-se $\mathrm{Split}^p(S)$ como sendo o par (S^{+p}, S^{-p}) onde:

$$S^{+p} = \{c \setminus \{\neg p\} \quad | \quad c \in S, \quad p \notin c\}$$
$$S^{-p} = \{c \setminus \{p\} \quad | \quad c \in S, \quad \neg p \notin c\}$$

Exemplo: Considere-se a FNC

$$S = [[\neg p, \neg q, \neg r], [\neg q, \neg r], [q], [r]]$$

Ao efectuarmos o *splitting* por r obtemos $(S^{+r}, S^{-r}) = \mathrm{Split}^r(S)$ em que

$$S^{+r} = [[\neg p, \neg q], [\neg q], [q]]$$
$$S^{-r} = [[q], []]$$

Na proposição que se segue identificamos o conjunto de cláusulas S com a FNC associada.

Proposição

Seja $(S^{+p}, S^{-p}) = \mathrm{Split}^p(S)$. Então:

1. $p \wedge S \equiv p \wedge S^{+p}$
2. $\neg p \wedge S \equiv \neg p \wedge S^{-p}$

Demonstração.

1. Considere-se o impacto da operação $\mathrm{Split}^p(S)$ nas cláusulas que envolvam literais com o símbolo p:
   - Se $S = (p \vee \alpha) \wedge S'$ temos que $p \wedge S = p \wedge (p \vee \alpha) \wedge S' \equiv$ $\equiv (p \wedge S') \vee (p \wedge \alpha \wedge S') \equiv p \wedge S' \wedge (\top \vee \alpha) \equiv p \wedge S'$. Logo verificamos que podemos remover cláusulas contendo o literal p.
   - Se $S = (\neg p \vee \alpha) \wedge S'$, então $p \wedge S = p \wedge (\neg p \vee \alpha) \wedge S' \equiv$ $\equiv (p \wedge \neg p \wedge S') \vee (p \wedge \alpha \wedge S') \equiv p \wedge \alpha \wedge S'$. Verificamos assim que podemos remover $\neg p$ da cláusula $(\neg p \vee \alpha)$.
2. Raciocinando de forma análoga ao ponto anterior.

Algoritmo *Davis-Putnam*

O algoritmo de *Davis-Putnam* permite verificar se uma fórmula (na FNC) é uma contradição.

Definição

Seja S o conjunto de cláusulas de uma fórmula na FNC.

$$\mathrm{DP}(S) \doteq \begin{cases} \mathrm{False} & , \textit{se } S = \emptyset; \\ \mathrm{True} & , \textit{se } \emptyset \in S; \\ \mathrm{DP}(S^{+p}) \wedge \mathrm{DP}(S^{-p}) & , \textit{noutro caso.} \end{cases}$$

onde $(S^{+p}, S^{-p}) = \mathrm{Split}^{p}(S)$ *para a dada escolha de um símbolo proposicional p.*

Exemplo de Aplicação

Considere-se novamente a fórmula $(\neg p \vee \neg q \vee \neg r) \wedge (\neg q \vee \neg r) \wedge q \wedge r$. Apliquemos o algoritmo *Davis-Putnam* para verificar se é uma contradição:

- $\mathrm{DP}([[\neg p, \neg q, \neg r], [\neg q, \neg r], [q], [r]]) =$
  $\mathrm{Split}^{p}([[\neg p, \neg q, \neg r], [\neg q, \neg r], [q], [r]]) = ([[\neg q, \neg r], [\neg q, \neg r], [q], [r]], [[\neg q, \neg r], [q], [r]])$
  - $\mathrm{DP}([[\neg q, \neg r], [q], [r]]) =$
    $\mathrm{Split}^{q}([[\neg q, \neg r], [\neg q, \neg r], [q], [r]]) = ([[\neg r], [r]], [[], [r]])$
    - $\mathrm{DP}([[\neg r], [r]]) =$
      $\mathrm{Split}^{r}([[\neg r], [r]]) = ([[]], [[]])$
      - $\mathrm{DP}([[]]) = \mathrm{True}$
      - $\mathrm{DP}([[]]) = \mathrm{True}$

      $= \mathrm{True}$
    - $\mathrm{DP}([[], [r]]) = \mathrm{True}$

    $= \mathrm{True}$
  - $\mathrm{DP}([[\neg q, \neg r], [q], [r]]) =$
    $\ldots$
    $\ldots$

    $= \mathrm{True}$

$= \mathrm{True}$

Exemplo de Aplicação (2)

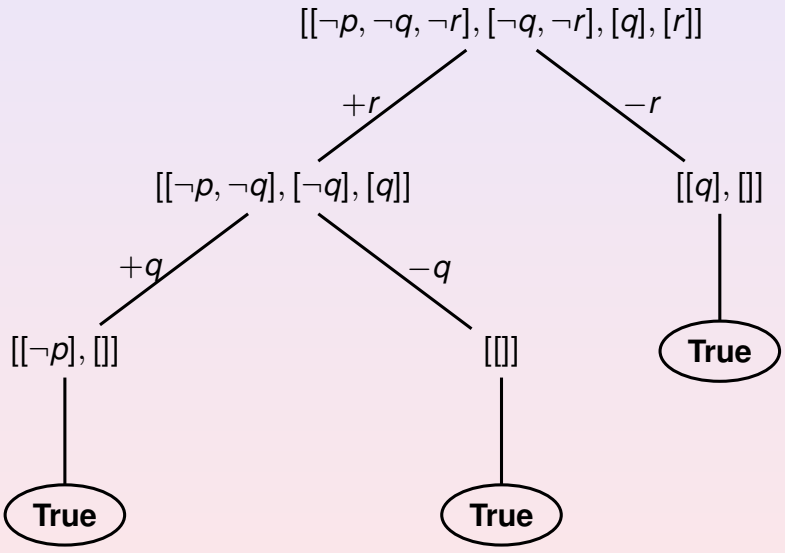
Considere-se ainda a fórmula $(\neg p \vee \neg q \vee \neg r) \wedge (\neg q \vee \neg r) \wedge q \wedge r$, mas inverte-se a ordem na escolhas dos literais utilizados nas simplificações:

$\bullet \mathrm{DP}([[\neg p, \neg q, \neg r], [\neg q, \neg r], [q], [r]]) =$
$\quad \mathrm{Split}^{r}([[\neg p, \neg q, \neg r], [\neg q, \neg r], [q], [r]]) = ([[\neg p, \neg q], [\neg q], [q]], [[q], []])$
$\quad \bullet \mathrm{DP}([[\neg p, \neg q], [\neg q], [q]]) =$
$\quad\quad \mathrm{Split}^{q}([[\neg p, \neg q], [\neg q], [q]]) = ([[\neg p], []], [[]])$
$\quad\quad \bullet \mathrm{DP}([[\neg p], []]) = \mathrm{True}$
$\quad\quad \bullet \mathrm{DP}([[]]) = \mathrm{True}$
$\quad = \mathrm{True}$
$\quad \bullet \mathrm{DP}([[q], []]) = \mathrm{True}$
$= \mathrm{True}$

Verifica-se então que um factor determinante na eficiência do algoritmo é a estratégia de selecção dos literais.

Exemplo de Aplicação (3)

Também é usual apresentar a aplicação do Algoritmo *Davis-Putnam* pela árvore das invocações recursivas:



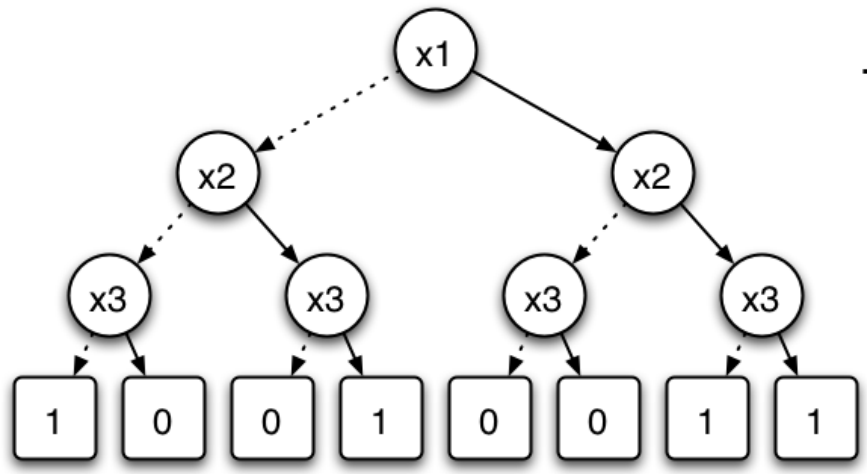
Sendo todas as folhas da árvore **True**, a proposição é uma **contradição**.

Comentários

- Uma optimização habitualmente considerada no algoritmo Davis-Putnam é a *Unit Propagation* — sempre que é detectada uma cláusula singular, esta é utilizada para simplificar a FNC (nesse caso não é necessário considerar ambas as valorações para a fórmula).
- Outra optimização (ainda que o seu impacto não seja tão relevante como a anterior) é a simplificação de *Literais Puros* (literais que ocorrem sempre com a mesma "polaridade" na FNC). Existem sempre modelos que fazem estes literais puros verdadeiros, pelo que se podem omitir as cláusulas onde ocorrem.
- Um factor determinante para a eficiência do método é a "estratégia" para escolha dos literais a eliminar. Interessa escolher literais que ocorram no maior número de cláusulas possível.

Binary Decision Trees

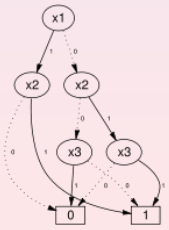
- Podemos representar a tabela de verdade de uma fórmula proposicional por intermédio de uma **árvore binária de decisão** com tantos níveis como o número de símbolos proposicionais utilizados:



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Binary Decision Diagrams

- As árvores de decisão contém muita redundância (repetições). E.g. os 0 e 1 que ocorrem nos nós terminais...
- A representação pode ser consideravelmente reduzida se maximizarmos a **partilha** de sub-árvores. Em rigor, deixamos de trabalhar com *árvores* — estamos perante *grafos dirigidos e acíclicos* (DAGs).
- Para realizar a partilha referida acima, é importante convencionar-se uma **ordem** nos símbolos proposicionais — essa ordem determina a sequência dos símbolos em todos os caminhos do DAG.
- A função exemplificada atrás pode então ser representada por:



BDDs

- Quando ambos os descendentes de um nó são grafos iguais (o mesmo, quando maximizamos a partilha) podemos omitir o nó em questão. Uma BDD que não tenha nós nas condições referidas diz-se **reduzida**. Note que esta simplificação corresponde à observação da equivalência

$$\varphi \equiv p \rightarrow \varphi; \varphi$$

- Quando nos referimos a BDDs, referimo-nos às **Ordered, Reduced Binary Decision Diagrams**. Em rigor, deveríamos designa-las por ORBDDs.
- Formalmente, diremos que uma (OR)BDD é um DAG *ordenado* que satisfaz:

  Unicidade – Não podem existir dois nodos distintos para uma mesma variável que partilhem os mesmos filhos, i.e.

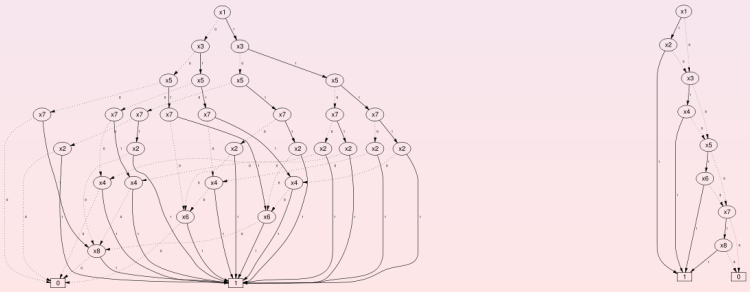
$$\text{var}(u) = \text{var}(v) \& \text{lo}(u) = \text{lo}(v) \& \text{hi}(u) = \text{hi}(v) \Rightarrow u = v$$

  Não-Redundância – Nenhum nodo pode ter filhos iguais, i.e.

$$\text{lo}(v) \neq \text{hi}(v)$$

Ordenação nas BDDs

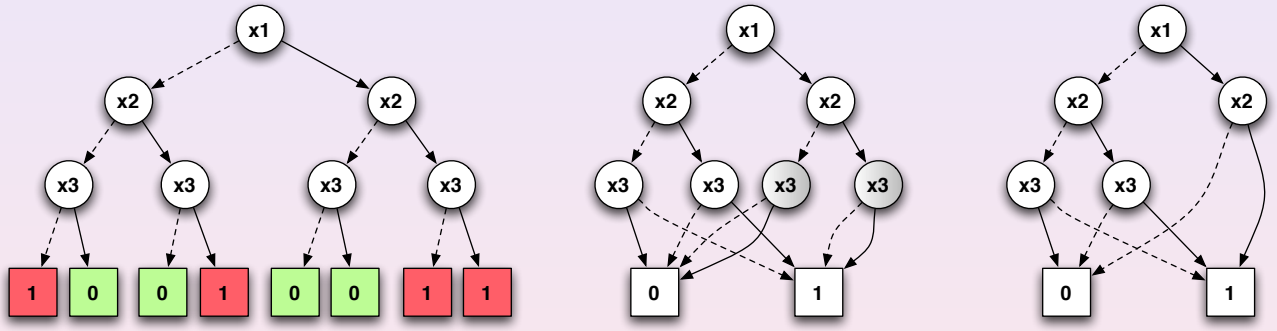
- Tal no método de *Davis Putnam*, a ordem adoptada na ordenação dos símbolos proposicionais tem um impacto enorme na eficácia do método.
- Infelizmente, o problema de determinar a "ordem óptima" para uma dada fórmula é, ele próprio, de resolução muito difícil. Na prática, as ferramentas de manipulação de BDDs fazem uso de heurísticas que procuram obter "boas ordenações" sem investir demasiado tempo nessa decisão...
- O exemplo que se segue mostra o impacto da escolha de diferentes ordenações na BDD referente a uma mesma fórmula.



Redução de Árvores de Decisão Binária

- Um BDD pode ser obtida partindo da árvore de decisão (ordenada), sobre a qual se aplicam as seguintes transformações:
  - *Remoção de Terminais Duplicados:* deixar apenas um terminal para cada etiqueta, redireccionando os arcos dos nós eliminados por forma a que mantenham a etiquetagem original.
  - *Remoção de Não-Terminais Duplicados:* se u e v verificam $\mathrm{var}(u) = \mathrm{var}(v)$, $\mathrm{lo}(u) = \mathrm{lo}(v)$ e $\mathrm{hi}(u) = \mathrm{hi}(v)$, então elimina-se um deles e redireccionam-se os seus arcos de entrada para o outro.
  - *Remoção de Testes Redundantes:* se um nodo v verifica $\mathrm{lo}(v) = \mathrm{hi}(v)$, então elimina-se v e redireccionam-se os seus arcos de entrada para $\mathrm{lo}(v)$.
- As transformações devem ser aplicadas até que nenhuma seja aplicável — nesse caso é óbvio que estamos perante uma (OR)BDD.

Exemplo de Redução



Observações

- As BDDs constituem representações canónicas para fórmulas proposicionais. Em particular, temos

Proposição

*Uma fórmula proposicional φ é uma tautologia se a sua representação numa BDD for o nó terminal **1**.*
*De forma dual, φ será uma contradição se a sua representação numa BDD for o nó terminal **0**.*

- Mas a obtenção da BDD de uma fórmula por redução da árvore de decisão associada não é um *procedimento eficiente para verificar a validade de fórmulas* — basta notar que uma árvore de decisão contém um número exponencial de nós (em função do número de símbolos proposicionais).

Construção *bottom-up* de BDDs

- A construção *bottom-up* de BDDs permite ultrapassar as dificuldades referidas. A ideia consiste em construir a BDD de uma fórmula a partir das BDDs das suas sub-fórmulas.
- Para que este processo se traduza em ganhos de eficiência, é crucial que seja possível verificar facilmente (em tempo constante) se já foi construída uma BDD para uma dada fórmula. Para isso faz-se uso de uma *Tabela de Hash*.
- A função de construção de uma nova BDD ($p \rightarrow \varphi_1; \varphi_2$) deve garantir que não se duplicam BDDs nem se criam BDDs não reduzidas. Assumindo uma tabela de hash T, inicializada com as BDDs terminais 0 e 1, e possuindo métodos para *lookup* e *insert* (em pseudo-código *haskell*).

```
mkBDD(p, bdd1, bdd2) =
  if bdd1 == bdd2
    then return bdd1
    else case T.lookup(p,bdd1,bdd2) of
           Just x -> return x
           Nothing -> return (T.insert(p,bdd1,bdd2))
```


- A BDD de uma fórmula lógica é construída seguindo a sua estrutura:

$$\begin{aligned}
\text{buildBdd}(\text{p}) &= \text{mkBdd}(p, 0, 1) \\
\text{buildBdd}(\neg\varphi) &= \textit{neg}(\text{buildBdd}(\varphi)) \\
\text{buildBdd}(\varphi_1 \vee \varphi_2) &= \textit{or}(\text{buildBdd}(\varphi_1), \text{buildBdd}(\varphi_2)) \\
\text{buildBdd}(\varphi_1 \wedge \varphi_2) &= \textit{and}(\text{buildBdd}(\varphi_1), \text{buildBdd}(\varphi_2))
\end{aligned}$$

- As funções *neg*, *or* e *and* manipulam BDDs: estabelecem como são construídas as BDDs de fórmulas estruturadas a partir das BDDs das sub-fórmulas.

- Definimos essas funções como (assume-se uma ordem pré-definida nos símbolos proposicionais):

$$
\begin{aligned}
neg(1) &= 0 \\
neg(0) &= 1 \\
neg(p \rightarrow \varphi_1, \varphi_2) &= \mathrm{mkBdd}(p, neg(\varphi_1), neg(\varphi_2))
\end{aligned}
$$

$$
\begin{aligned}
or(1, \varphi) = or(\varphi, 1) &= 1 \\
or(0, \varphi) = or(\varphi, 0) &= \varphi
\end{aligned}
$$

$$
or(\varphi = (p \rightarrow \varphi_1; \varphi_2), \varphi' = (p' \rightarrow \varphi'_1; \varphi'_2)) = \begin{cases} \mathrm{mkBdd}(p, or(\varphi_1, \varphi'), or(\varphi_2, \varphi')) & , \text{ se } p < p' \\ \mathrm{mkBdd}(p', or(\varphi, \varphi'_1), or(\varphi, \varphi'_2)) & , \text{ se } p' < p \\ \mathrm{mkBdd}(p, or(\varphi_1, \varphi'_1), or(\varphi_2, \varphi'_2)) & , \text{ se } p = p' \end{cases}
$$

$$
\begin{aligned}
and(1, \varphi) = and(\varphi, 1) &= \varphi \\
and(0, \varphi) = and(\varphi, 0) &= 0
\end{aligned}
$$

$$
and(\varphi = (p \rightarrow \varphi_1; \varphi_2), \varphi' = (p' \rightarrow \varphi'_1; \varphi'_2)) = \begin{cases} \mathrm{mkBdd}(p, and(\varphi_1, \varphi'), and(\varphi_2, \varphi')) & , \text{ se } p < p' \\ \mathrm{mkBdd}(p', and(\varphi, \varphi'_1), and(\varphi, \varphi'_2)) & , \text{ se } p' < p \\ \mathrm{mkBdd}(p, and(\varphi_1, \varphi'_1), and(\varphi_2, \varphi'_2)) & , \text{ se } p = p' \end{cases}
$$

Observações

- A utilização da função $\mathrm{mkBdd}(-)$ garante os invariantes das BDDs construidas.
- Podem ser considerados outros operadores (e.g. implicação, equivalência, xor, etc.). As correspondentes funções de construção são facilmente deriváveis dos exemplos apresentados.
- Mas, uma vez mais, devemos recorrer a *truques de implementação* para que o ganho na eficiência seja efectivo...
- As invocações às funções que combinam BDDs (`neg`, `or`, `and`, ...) devem ser *memorizadas*. Essa técnica é conhecida por ***memoization*** e consiste em armazenar os pares $(x, f(x))$ para todos os argumentos para os quais já foi calculado o valor da função — só quando se pretende o valor da função num ponto não contido nessa tabela e que é efectivamente calculada a função (e armazenado o novo par na tabela).