The DPLL Algorithm

Leo Ware

This is my implementation of the DPLL algorithm, which is a heuristic algorithm for solving the general SAT problem. This implementation includes a nice Literal class for definition knowledgebases (KBs) and a function that implements the DPLL algorithm using different levels of heuristics according to a setting.

First we can import the algorithm from the main file:

In [1]:
```
from main import *
```

Next, we can create an example knowledgebase and run the algorithm on it. We can get away with such a sparse representation because we know this is a propositional logic knowledgebase in conjunctive normal form. That means that each of the letters represents a logical sentence that can be true or false. Each set represents a disjunction---aka we a claiming that one of the sentences is true. And, the list represents a conjunction. We are saying that all of the disjunctions are true.

In [2]:
```
# Define some literals
A = Literal('A')
B = Literal('B')
C = Literal('C')
D = Literal('D')
E = Literal('E')
F = Literal('F')

# the KB from R&N
KB = [
    {-A, B, E},
    {-B, A},
    {-E, A},
    {-E, D},
    {-C, -F, -B},
    {-E, B},
    {-B, F},
    {-B, C}
]
```

The algorithm will try to find an assignment such that the knowledgebase itself is true. This will look like a dictionary mapping from literals to booleans. This is known as the general SAT problem. This problem is NP-Complete, meaning that there is no known algorithm that can reliably solve it in less than exponential time. However, most (random) instances can be solved quite quickly by using efficient heuristics. Obviously, this example is small enough that it doesn't matter.

In [3]:
```
# run the algorithm
dpll_satisfiable(KB)
```

Out[3]:
```
(True,
 {Literal(name='B', sign=True): False,
  Literal(name='A', sign=True): False,
  Literal(name='D', sign=True): True,
  Literal(name='E', sign=True): False},
 [Literal(name='B', sign=False),
  Literal(name='C', sign=True),
  Literal(name='A', sign=True),
  Literal(name='F', sign=False),
  Literal(name='A', sign=True),
  Literal(name='E', sign=True),
  Literal(name='D', sign=True),
  Literal(name='E', sign=False)])
```

The function returns three things: a boolean indicating whether the KB was successfully satisfied, an assignment, and a symbol trace. The assignment means that if you substitute each literal for the corresponding boolean, then the KB will be satisfied. An assignment like this is called a model.

The trace shows which literals the algorithm visited while constructing this assignment. This is interesting because most of the cost of the algorithm comes from how many literals it visits, so we can compare different heuristics based on how many they visit. By default, the function runs with no heuristics---it just does depth first search on possible assignments. This is tractable for such a small KB, but we can see that it made a fair number of mistakes and had to backtrack multiple times.

In [4]:
```
# with better heuristics
for h_lvl in range(4):
    print("heuristic level:", h_lvl)
    _, _, trace = dpll_satisfiable(KB, heuristic_level=h_lvl)
    print("trace length:", len(trace))
    print("trace", trace)
    print()
```

```
heuristic level: 0
trace length: 8
trace [Literal(name='B', sign=False), Literal(name='C', sign=True), Literal(name='A', sign=Tr
ue), Literal(name='F', sign=False), Literal(name='A', sign=True), Literal(name='E', sign=Tru
e), Literal(name='D', sign=True), Literal(name='E', sign=False)]

heuristic level: 1
trace length: 6
trace [Literal(name='B', sign=False), Literal(name='A', sign=True), Literal(name='C', sign=Fa
lse), Literal(name='F', sign=False), Literal(name='E', sign=False), Literal(name='A', sign=Fa
lse)]

heuristic level: 2
trace length: 7
trace [Literal(name='D', sign=True), Literal(name='B', sign=False), Literal(name='A', sign=Tr
ue), Literal(name='C', sign=True), Literal(name='F', sign=False), Literal(name='E', sign=Fals
e), Literal(name='A', sign=False)]

heuristic level: 3
trace length: 7
trace [Literal(name='D', sign=True), Literal(name='B', sign=False), Literal(name='A', sign=Tr
ue), Literal(name='C', sign=True), Literal(name='F', sign=False), Literal(name='E', sign=Fals
e), Literal(name='A', sign=False)]
```

What is the algorithm doing here? Well, at `h_lvl=0`, the algorithm was doing dfs. Basically, it would keep guessing until it got a model that satisfied the KB. At higher levels, it employs smarter strategies.

- At `h_lvl=1`, the algorithm employs the degree heuristic. It substitutes for the Literal that appears most often. This means that bad ideas die quickly. The more often a variable appears, the more likely it has to take a specific value. So, by guessing on the most frequent first, we avoid going down paths that are doomed to fail.
- At `h_lvl=2`, the algorithm uses two heuristics which allows it to make substitutions with absolute certainty, instead of just guessing. If a variable appears only negated or only unnegated, then it can simply set it to be true (if unnegated) or false (if negated). Additionally, if a variable appears on its own, then it must be true (if unnegated) or false (if negated). By making these free substitutions, the algorithm saves on a lot of guessing.
- At `h_lvl=3`, the algorithm combines these. It looks at all the literals nominated by the level 2 heuristic and picks the one that occurs the most times.

We can see that in this instance, the cheapest heuristic actually performed best. However, all three heuristic algorithms outperformed brute force, and for larger problems though, you would definitely want to be running with all three.