Erlang SAT Demo

Running the code

Start the "Eshell"

```
erl
```

Eshell should now be running and present something like the following:

```
Erlang/OTP 20 [erts-9.2] [source] [64-bit] [smp:8:8] [ds:8:8:10]
              [async-threads:10] [kernel-poll:false]

Eshell V9.2  (abort with ^G)
1>
```

To load and compile the sat.erl module type the following on the eshell prompt.

```
c(sat).
```

Eshell should reply with `{ok, sat}`.

When `sat` has been loaded you can see information about the loaded module by issuing the command:

```
sat:module_info().
```

Trying some SAT

The algorithm we parallelize in this demo solves the boolean satisfiability problem. The goal of the algorithm is to decide if, or if not, a boolean formula can be made true by a particular interpretation of the variables it is made up from.

For example:

```
((a or b) and (not a))
```

which can be made true if we do the following instantiation of the variables.

```
a = false
b = true
```

A simple example of a formula that cannot be satisfied is of course:

```
(a and (not a))
```

The demo code operates on boolean formulas transformed into a disjunctive normal form. That is they will be built exclusively from the connectives `and`, `or` and `not` and have the form:

```
(a1 or a2 or ... or aN) and
(b1 or b2 or ... or bN) and
```

...

where each variable may also occur negated (with `not`).

The demo code utilizes an encoding of variables as non-zero values where a negative value represents a negated variable. For example if you want to use variables `a` and `b`, assign to them the numerical values `1` and `2`. If the formula is `(a or (not b))` this will be represented as the erlang list `[1, -2]`.

An input formula to the solver should be turned into a list of lists where each of the inner lists represent a (clause) disjunction of terms. Between each clause there is an `and` connective.

Now we can see if the formula is satisfiable:

```
sat:solve([[1, -2]]).
```

This will result in the value `[1]` which should be interpreted to mean that it is enough to make `a` true to make the whole formula true.

If we instead were interested in solving `((not a) or (not b))` we would get:

```
sat:solve([[-1, -2]]).
```

and the result `[-1]` which means that it is enough to make the assignment `a = false` to make the formula true.

Let's try something with a conjunction in it as well. `(a or b) and ((not a) or (not b))` for example. Using the same encoding of `a` and `b` as earlier this leads to the following:

```
sat:solve([[1, 2], [-1,-2]]).
```

And it turns out that to make the formula true we much create an interpretation of the variables where one of them is true and the other is false. For example `a = true` and `b = false`. This is the answer the solve function provides as `[1, -2]`.

Above, the completely opposite interpretation `[-1, 2]` would also have worked. That this is not the result returned may be a hint to in what order the solver is trying out instantiations.

About the algorithm `solve`

The sequential solver first tries to satisfy the formula assuming L is true, (L could it be a negated variable). If that assumption turns out wrong, try again assuming L is false.

```
solve([]) ->
    [];
solve([[]|_]) ->
    false;
```

```
solve([[L|Lits]|Clauses]) ->
    case solve(unit_propagation(L, Clauses)) of
        false ->
            case solve(unit_propagation(-L,[Lits|Clauses])) of
                false ->
                    false;
                Solution ->
                    [-L|Solution]
            end;
        Solution ->
            [L|Solution]
    end.
```

Note that if we assume L to be true as we do in the first case, the entire first clause can be satisfied and we do not need to look in it any further. On the other hand if we assume L to be false, for the first clause to become true some other L' must be true.

The `unit_propagation` function deletes all occurrences of -L from all clauses that do not contain L. If we assume L there is no way to make -L true. All clauses containing L can be removed because we can consider them already satisfied.

```
unit_propagation(L, Clauses) ->
    NewClauses =
        [lists:delete(-L,C) || C <- Clauses,
                               not lists:member(L,C)],
    sort_by_length(NewClauses).

sort_by_length(Clauses) ->
    [C || {_,C} <- lists:usort([{length(C),C} || C <- Clauses])].
```

Sorting on the length of the remaining clauses is to ensure that if there are empty clauses (clauses that given our current assumptions cannot be satisfied) these end up at the head.

Parallelization by speculation

Instead of trying L and -L sequentially we can start a process to try and find a solution using -L in parallel to the process trying with L. If we then fail to find a solution based on L we can check if the speculative process found one.

```
par_solve([]) ->
    [];
par_solve([[]|_]) ->
    false;
par_solve([[L|Lits]|Clauses]) ->
```

```
    Rest = speculate(fun()-> par_solve(unit_propagation(-L,[Lits|Clauses])) end),
    case par_solve(unit_propagation(L,Clauses)) of
    false ->
        case value_of(Rest) of
        false ->
            false;
        Solution ->
            [-L|Solution]
        end;
    Solution ->
        [L|Solution]
    end.
```

The `speculate` function takes a function as argument and starts a thread to evaluate it.

```
%% Speculation

speculate(F) ->
    Parent = self(),
    Pid = spawn_link(fun() -> Parent ! {self(),F()} end),
    {speculating,Pid}.
```

In `speculate` a unnamed function is created to `F` run in parallel with the parent (`self()`) and sends the result of `F` to parent.

The `spawn_link` command spawns a process and creates a link between the parent and spawned child. If one of the processes terminates a message will be sent to linked process containing an `exit reason`.

```
value_of({speculating,Pid}) ->
    receive {Pid,X} ->
        X
    end.
```

The `value_of` function waits for a result from a speculating process.

Parallelization by worker pool

The previous approach `par_solve` spawns many worker processes, too many. This leads to some overhead from context switching between all these processes.

To solve this performance bottle-neck we can use a pool of workers with exactly as many workers as we want! We can this way ensure we do not run more processes than we have processor cores and there will be no scheduling overhead to talk about.

```
pool_solve(P) ->
    start_pool(erlang:system_info(schedulers)-1),
```

```
    S = pool_solve1(P),
    pool ! {stop,self()},
    receive {pool,stopped} -> S end.

pool_solve1([]) ->
    [];
pool_solve1([[]|_]) ->
    false;
pool_solve1([[L|Lits]|Clauses]) ->
    Rest = speculate_on_worker(
         fun()-> pool_solve1(unit_propagation(-L,[Lits|Clauses])) end),
    case pool_solve1(unit_propagation(L,Clauses)) of
    false ->
        case worker_value_of(Rest) of
        false ->
            false;
        Solution ->
            [-L|Solution]
        end;
    Solution ->
        [L|Solution]
    end.
```

The code below is all "pool management".

```
start_pool(N) ->
    true = register(pool,spawn_link(fun()->pool([worker() || _ <- lists:seq(1,N)]) end)).

pool(Workers) ->
    pool(Workers,Workers).

pool(Workers,All) ->
    receive
    {get_worker,Pid} ->
        case Workers of
        [] ->
            Pid ! {pool,no_worker},
            pool(Workers,All);
        [W|Ws] ->
            Pid ! {pool,W},
            pool(Ws,All)
        end;
    {return_worker,W} ->
        pool([W|Workers],All);
    {stop,Pid} ->
        [unlink(W) || W <- All],
        [exit(W,kill) || W <- All],
```

```
        unregister(pool),
        Pid ! {pool,stopped}
    end.

worker() ->
    spawn_link(fun work/0).

work() ->
    receive
    {task,Pid,R,F} ->
        Pid ! {R,F()},
        catch pool ! {return_worker,self()},
        work()
    end.
```

The `speculate_on_worker` function tries to get a worker process from the pool. If the pool has no worker available you get `{not_speculating,F}`.

```
speculate_on_worker(F) ->
    case whereis(pool) of
    undefined ->
        ok; %% we're stopping
    Pool -> Pool ! {get_worker,self()}
    end,
    receive
    {pool,no_worker} ->
        {not_speculating,F};
    {pool,W} ->
        R = make_ref(),
        W ! {task,self(),R,F},
        {speculating,R}
    end.
```

Checking if a speculation has resulted in a value is similar to `par_solve` but in the case when there was no free worker in the pool, the function `F` be called here instead!

```
worker_value_of({not_speculating,F}) ->
    F();
worker_value_of({speculating,R}) ->
    receive
    {R,X} ->
        X
    end.
```

Limit Parallelism to large problems

This last piece of code is simple modification of the pool solver that only parallelizes is the problem size is "large.

On my machine this does not seem to improve things but please experiment with this in the erlang parallelization lab!

```
limit_solve(P) ->
    start_pool(erlang:system_info(schedulers)-1),
    S = limit_solve1(P),
    pool ! {stop,self()},
    receive {pool,stopped} -> S end.

limit_solve1([]) ->
    [];
limit_solve1([[]|_]) ->
    false;
limit_solve1([[L|Lits]|Clauses]) when length(Clauses) < 250 ->
    solve([[L|Lits]|Clauses]);
limit_solve1([[L|Lits]|Clauses]) ->
    Rest = speculate_on_worker(
         fun()-> limit_solve1(unit_propagation(-L,[Lits|Clauses])) end),
    case limit_solve1(unit_propagation(L,Clauses)) of
    false ->
        case worker_value_of(Rest) of
        false ->
            false;
        Solution ->
            [-L|Solution]
        end;
    Solution ->
        [L|Solution]
    end.
```