DONKEY algorithms

parakram majumdar

January 28, 2010

1 General BFS

In the previous class, we had seen how to Explore all the nodes in a BFS fashion
:

```
Let Q be an empty queue;
Ans = φ;
Q.push(φ);
while (Q is not empty) do
    t = Q.pop();
    if (qualifies(t)) then
        Ans = Ans ∪ {t};
    end if

    for every u in ρ(t) do
        //ρ is the complete downward closure
        if (u.visited == False) then
            Q.push(u);
            u.visited = True;
        end if
    end for

end while
```

2 Truncated BFS algorithm

In the simple BFS, notice how u is pushed into the queue even if t does not qualify. We now improve over the BFS algorihm using the fact that the qualification function is anti-monotonic. Thus, if a hypothesis h is unqualified, there is no point in adding its children into the queue.

Truncated BFS algorithm:

```
Let Q be an empty queue;
Ans = φ;
Q.push(φ);
while (Q is not empty) do
   t = Q.pop();
   if (not qualifies(t)) then
      continue;
   end if

   Ans = Ans ∪ {t};

   for every u in ρ(t) do
      //ρ is the complete downward closure
      if (u.visited == False) then
         Q.push(u);
         u.visited = True;
      end if
   end for

end while
```

Notice that for any hypothesis t, each element $u \in \rho(t)$ can be generated as

```
for each σ ∈ Σ \ t do
   u = t ∪ {σ};
   ...
end for
```

However, using the monotonicity property, we know that for any set to be qualified, all its subsets should be qualified. Thus, for any u generated above, it is worth checking whether the generating $\{\sigma\}$ is qualified. This leads us to the slightly better truncated BFS :

```
Let Q be an empty queue;
Ans = ϕ;
Q.push(ϕ);
while (Q is not empty) do
   t = Q.pop();
   if (not qualifies(t)) then
      continue;
   end if

   Ans = Ans ∪ {t};

   for each σ ∈ Σ' \ t do
      //Σ' is the set of all σ' ∈ Σ such that {σ'} is qualified
      u = t ∪ {σ};
      if (u.visited == False) then
         Q.push(u);
         u.visited = True;
      end if
   end for

end while
```

Note that this algorithm is hardly worth the complication if we need to check the qualification of each σ' again and again. Hence, instead, it is better to generate the set Σ' once and later simply pick the elements from Σ' rather than Σ.

3 Final Algorithm

Taking this idea of truncation a step further, we now want to push a hypotheses h into the queue only if all subsets are qualified. A necessary and sufficient condition for all subsets of h to be qualified is that all subsets h' such that $|h'| = |h| - 1$ are qualified. Thus, we modify the algorithm to :

```
Let Q be an empty queue;
Ans = φ;
Q.push(φ);
while (Q is not empty) do
   t = Q.pop();
   if (not qualifies(t)) then
      continue;
   end if
   Ans = Ans ∪ {t};
   for each σ ∈ Σ \ t do
      u = t ∪ {σ};
      if (u.visited == True) then
         continue;
      end if
      Let test be a boolean variable set to False;
      for all s ∈ u do
         if (not qualifies(u \ s)) then
            test = True;
            break;
         end if
      end for
      if (test == False) then
         Q.push(u);
         u.visited = True;
      end if
   end for
end while
```

The problem with this algorithm, just like the earlier one, is that checking for qualification is expensive. Instead, it makes more sense to store the results of the qualification checks so that we do not compute it twice for the same hypothesis.

An elegant way to do this is to notice that every qualifying hypothesis of size n can be obtained by the union of two qualifying sets of size $n - 1$ (where $n > 1$). Writing this idea down as an algorithm leads to the final algorithm that we discussed in class.

```
S = φ;
Ans = φ;
for each element σ in Σ do
    if (qualifies({σ})) then
        S = S ∪ {σ};
    end if
end forAns = Ans ∪ S;
for i = 1 to (|Σ| − 1) do
    S_2 = S;
    S = φ;
    for all u, v ∈ S differing by exactly one element do
        w = u ∪ v;
        if (qualifies(w)) then
            S = S ∪ {w};
        end if
    end for
    Ans = Ans ∪ S;
end for
```

Here an interesting thing to notice is how u and v must necessarily differ in exactly one element, because otherwise $|w|$ will become greater than $|u| + 1$.