Decoding (Additional Notes)

Jakub Waszczuk

January 2019

1 Search tree

The space of possible translations is represented in the form of a *search tree*:

- Each node in the tree represents a hypothesis (partial translation)
- Each arc represents expansion of a hypothesis to another hypothesis; the arc is labeled with the corresponding translation step (translation of a not-yet-translated phrase in the input sentence to an output phrase) as well as the corresponding score (i.e. the probability of the effectuated phrase translation $\times$ the incurred language model cost $\times$ reordering cost)
- Each branch from the root to a leaf node represents a particular translation process and – if the leaf represents a complete hypothesis – a complete translation (output sentence)
- The overall score of the branch[1] represents the score of the corresponding translation

2 Pruning

One of the main techniques to optimize the search for best the translation is pruning. It consists in removing some of the branches of the search tree and, consequently, making the search space smaller.

Pruning can be safe, in the sense that it can be performed in a way which does not make the exploration fail (even though a complete hypothesis exists) or find a sub-optimal solution. This is the case if we use AI techniques such as *dead-end detection* or *hypothesis recombination*.

We can also use pruning *heuristics*, which are more aggressive and do not provide the guarantees of safe pruning. On the other hand, these can lead to more significant search space reductions and, as a consequence, practical runtime. An example of a heuristic pruning technique is *beam search*.

[1] Multiplication of the scores of the arcs on the branch in normal domain, or sum in logarithmic domain

2.1 Dead-end detection

Dead-end detection simply consists in determining if, from a given node, a path to a complete hypothesis (and, therefore, complete translation) exists at all. If not, the search algorithm is faced with a „dead-end" which is not worth further exploring.

A generalized form of dead-end detection can involve determining if a best-score solution can be reached from the given node. If the answer is „no", such a branch can be safely pruned as well. A simple criterion can be used here – if the partial score of the current node is lower than the score of the best complete solution found so far (if any), we can safely skip the current branch. This is safe because, while expanding hypotheses, the score only gets lower. This technique can be especially effective in depth-first tree exploration (which will typically find some reference complete translation much faster than breadth-first search).

2.2 Hypothesis recombination

Another safe technique is hypothesis/node recombination. It involves pruning newly generated nodes equivalent to other, already existing nodes.

Definition 1. *We say that two nodes v and v' are* equivalent *if:*

- *Each sequence of translation steps leading from v to a complete translation l can be also applied to v', resulting in a complete translation l'*
- *And vice versa, each sequence of translation steps leading from v' to a complete translation l' can be also applied to v, also resulting in some complete translation l*

If v and v' are equivalent, and we know that we can reach a complete translation from v, we can be sure to be able to reach a complete translation from v'. Moreover, we can be sure that the minimal remaining score s needed to reach a complete translation from v is the same as the minimal remaining score needed to reach a complete translation from v'. Therefore, if the partial score p of v is higher than the partial score p' of v', than we can drop v', because the overall cost $p \times s$ of the best translation reachable from v is higher than $p' \times s$, the overall cost of the best translation reachable from v'.

Hypothesis recombination alone is not enough to obtain efficient decoding.

Proposition 1. *The time complexity of decoding (e.g. stack decoding) with hypothesis recombination is exponential in the worst case.*

Proof. Recombination only applies to hypotheses with the same set of covered input positions M.[2] Even if we recombine all hypotheses with the same set M, there are still 2^n distinct coverage sets that can be created (all subsets of $\{1, \ldots, n\}$, where n is the length of the input sentence). In the worst case, one or more hypotheses (differing on the last translated position e or the last translated phrase p) will be created for each distinct coverage set, resulting in the number of distinct hypotheses exceeding 2^n ($\implies$ exponential search space). The time complexity is also exponential because a standard decoding algorithm has to explore the entire search space. □

[2] Recall from the lecture that each hypothesis has the form of a tuple $\langle M, p, e, w \rangle$.

2.3 Beam search

In beam search the nodes of the search tree are organized into *stacks* of mutually comparable[3] hypotheses. The basic idea of beam search is very simple – if a stack gets too large, the lowest-scoring nodes are removed from it. The easiest flavor of beam search-related pruning is the so-called *histogram pruning*, which involves keeping a maximum K number of nodes in each stack, K being a predetermined parameter.

Proposition 2. *Let n be the length of the input sentence and T be the maximum number of possible translations for any given input phrase in the phrase translation table. Let also assume that the cost of placing a hypothesis in a stack has constant time (i.e., cost 1). Then, the time complexity of beam search decoding is polynomial in n, K, and T.*

Proof. Beam search decoding looks like this:

- For each stack k from 0 to $n-1$
- For each node/hypothesis h in stack k
- For each expansion h' of h, add h' to the corresponding stack $> k$

The first point involves n top-level steps. For each top-level step, we look at all (at most) K hypotheses h in the corresponding stack k. We then determine all possible expansions h' of h. To this end, we use the expansion algorithm given during the lecture, which involves looking at less than n^2 spans $(i,j)\colon 1 \leq i \leq j \leq n$ in the input sentence and, for each not-yet-translated span (i,j), considering at most T of its possible translations. Finally, we place h' in the corresponding stack, which is constant time. To summarize, the upper bound on the cost of the entire process is:

$$n \times K \times n^2 \times T = n^3 \times K \times T$$

Therefore, the algorithm is polynomial in n, K, and T. □

3 Remaining cost estimation

The effectiveness of beam search directly depends on the quality of partial scores. Put differently, if the score of a node (partial translation) really reflects its quality, then we can simply compare the scores of any given two nodes in order to determine which of the two is better (and which can be pruned).

Unfortunately, the link between the partial score of a node and the node's quality is not so direct. In fact, to be really able to ascertain its quality in a reliable manner, we would have to first determine the complete translations to which it can expand, as well as their overall translation scores. This would allow to avoid situations where a partial translation has a (relatively) high score, even though it does not lead to any good translation (or, worse, doesn't lead to any translation at all!). But, of course, this is not realistic – when we look at a particular node and try to make the decision whether it should be pruned or

[3] Two hypotheses are comparable if they have the same number of translated input words. As a result, both involve the same number of cost multiplications.

not, we don't know to what complete translations it can evolve, nor what their overall scores would be.

This leads to the idea of *estimating the remaining cost*. Instead of determining the minimum remaining cost $r(v)$ of reaching a complete translation from a given node v, we design a heuristic function h which estimates r. Then, when trying to decide whether a particular node v is useful, we take into account its total score $s(v) \times h(v)$ rather than just its partial score $s(v)$.

Definition 2. *Let V be the set of nodes in the search graph, $v \in V$ be a node, $s(v)$ be its partial score, and $h\colon V \to [0,1]$ be a heuristic. Then, we define v's* total score *as:*

$$t(v) = s(v) \times h(v) \tag{1}$$

In general, we could use any heuristic function $h\colon V \to [0,1]$, but that wouldn't necessarily be useful. In practice, we typically want the heuristic to be *admissible*.

Definition 3. *We say that $h\colon V \to [0,1]$ is* admissible *if, for any give node v:*

$$h(v) \geq r(v) \tag{2}$$

The advantage of an admissible heuristic is that, whatever the cost $h(v)$ it gives for a given node v, we can be sure that it will have to be incurred before expanding v to a complete translation. Hence, it makes sense to use $t(v) = s(v) \times h(v)$ in place of $s(v)$ as the measure of the quality of node v. Another advantage is that an admissible heuristic can be used in the $A^\star$ algorithm (see Sec. 4).

Apart from being admissible, the general goal when designing a heuristic is to make it as close to r as possible. Indeed, if $h = r$, then we get perfect estimations and search can be simply reduced to a traversal of a single path from the root to an optimal leaf obtained by following the nodes with the best total scores.

3.1 Remaining cost estimation in SMT

For simplicity, let's assume a word-based translation model (e.g. IBM-1) P_T. What follows could be generalized to phrase-based models, but the calculations would be significantly more complicated then (even if still feasible).

The goal is to define a heuristic $h(v)$ which, for a given node v, estimates the cost remaining to reach a complete translation starting from v. Recall from the lecture, that each node/hypothesis is a tuple $\langle M, p, e, s\rangle$, where:

- $M \subseteq \{1, \ldots, n\}$ is the set of input positions translated so far (n is the length of the input sentence)
- p is the last output phrase of the partial translation generated so far[4]
- e is the ending input position corresponding to the last translated phrase
- s is the partial score of the generated partial translation

[4]This is enough in case of the bigram model.

A simple way to obtain an admissible heuristic is to optimistically assume that each of the words remaining to translate in the input sentence will be translated using the best possible entry in the underlying lexical translation table. Formally:

$$h(v) = \prod_{k \in \{1,\ldots,n\} \setminus M} \max_y P(x \mid y) \tag{3}$$

What about other costs – reordering, fertility, language model, etc.? In the formula above we just assume that they don't cost anything. It's not a problem because it doesn't break the admissibility of the heuristic. In general, the rule of thumb when designing admissible cost estimation heuristics is – be optimistic. Of course, we could try to estimate and include these additional costs in our estimation and, provided that we manage to preserve admissibility, the resulting heuristic would be better and lead to more reliable pruning.

4 Shortest-path decoding

The A* algorithm is an instance of a shortest-path algorithm, i.e., a method which allows to find a shortest path in a *weighted* graph.

Definition 4. *Let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of arcs. A* weighted *graph is a pair (G, w), where G is a graph and $w\colon E \to [0, \infty)$ is a function which assigns non-negative weights to arcs in E.*

From now on, we assume that we have a pre-determined graph $G = (V, E)$ weighted with function w. Moreover, we assume that there is single root node (with no incoming arcs) $R \in V$ and a set of target nodes $T \subset V$. In the context of SMT decoding, R will represent the empty hypothesis, and T – the set of complete hypotheses.

Definition 5. *We define a* path *in G as a sequence of arcs connecting two nodes $v, v' \in V$. We then also say that such a path* leads from v to v'.

Definition 6. *Let p be a path in G. We define the* weight of path p *as the sum of the weights of the arcs in p.*

$$w(p) = \sum_{e \in p} w(e)$$

Definition 7. *Given $v \in V$, we denote with $w(v)$ the* weight of node $v \in V$, *i.e., the weight of the minimal-weight path leading from R to v.*

Definition 8. *Let $\mathcal{A}$ be an algorithm which returns a path in G leading from R to a target node. We call $\mathcal{A}$ a* shortest-path algorithm *if the path it returns is the one with the lowest weight among all the paths leading from R to a target node.*

What is the relation between a shortest-path algorithm and our problem? At first sight, there are important differences:

- The search space in SMT decoding is a tree and not a graph.
- The score of a branch in the search tree is the product of the scores assigned to its arcs, not their sum.

- In SMT search we look for a complete leaf with the highest score, while a shortest-path algorithm looks for the shortest path to a target node.

The first point is not really an issue, though, since a tree is simply a graph with the additional constraint that any node can have at most one incoming arc (a restriction which a graph does not have to satisfy). Moreover, a search tree in SMT can take the form of a graph if hypothesis recombination is used to merge rather than to prune spurious hypotheses.

Note that in case we stick to a tree-structured search space, the definition of the weight of a node (Def. 7) becomes simpler – given a node $v \in V$, it is the weight of the *only* path from the root (empty hypothesis) to v.

The second and the third point can be solved at the same time. To this end, we transform each score s to the corresponding weight $w = -log(s)$. Since scores belong to $[0, 1]$, the resulting value is within $[0, \infty]$, with the smallest possible weight 0 corresponding to the highest possible score 1 and the highest possible weight ∞ corresponding to the smallest possible score 0. Moreover, since we transform scores from normal to (negated) log domain,

- the product of scores becomes the (negated) sum of weights, and
- the shortest path corresponds to the highest score solution.

4.1 A* decoding

The A* algorithm (see Alg. 1) is an example of a shortest-path algorithm. It relies on a heuristic estimation of the remaining distance.

Definition 9. *Let $r\colon V \to [0, \infty)$ be a function which, for any given $v \in V$, returns the weight of the minimal path leading from v to a target node. Then, we call r a* remaining distance function*.*

Definition 10. *Let $h\colon V \to [0, \infty)$ be a function which heuristically estimates (approximates) r. Then, we call h a* remaining distance estimation heuristic*.*

Even though A* relies on a heuristic estimating the remaining distance, it is not heuristic itself – provided that the underlying heuristic satisfies certain properties, the A* algorithm guarantees to find the shortest path in the graph, and therefore it is *exact*. Moreover, the better the heuristic estimates the remaining distance, the smaller the part of the search tree/graph the algorithm will have to effectively explore.

The A* algorithm resembles a lot the breadth-first search algorithm presented during the lecture. The crucial difference is that, instead of using a standard queue to store newly discovered hypotheses, it uses a *priority queue*. The hypotheses are stored in this queue not in the first-in-first-out fashion, but in the order consistent with their total weights (see also Def. 2):

Definition 11. *Let $v \in V$. We define v's* total weight *as:*

$$t(v) = w(v) + h(v) \tag{4}$$

Each time a hypothesis is removed from the queue (cf. line 1.5), it is the hypothesis with the smallest total weight.

Algorithm 1 A*

```
1: let Q be an empty priority queue of hypotheses
2: let G be an empty set of completed hypotheses
3: place empty hypothesis in Q
4: while Q not empty do
5:     x ← DELETE-MIN(Q)
6:     if h complete then
7:         add h to G
8:     else
9:         for each expansion h' of h do
10:            add h' to Q
11:        end for
12:    end if
13: end while
```

Definition 12. *We say that h is* admissible *if, for any give node $v \in V$:*

$$h(v) \leq r(v) \tag{5}$$

In words, an admissible heuristic must never overestimate the remaining distance. Note that this definition is an inverse of Def. 3 – this is because we are working in the negated log-domain now.

Definition 13. *We say that h is* monotonic *if, for each hypothesis v and its expansion v' in the search graph:*

$$t(v) \leq t(v') \tag{6}$$

In different terms, the heuristic is monotonic if the total weights never decrease as the algoritm explores the search space, they only increase or stay level.

The following propositions refer all to Algorithm 1.

Proposition 3. *The hypotheses are removed from the priority queue in an ascending order of their total weights.*

Proof. Follows from

- the definition of the priority queue – we know that, when a hypothesis is removed from the queue, it is the one with the smallest total weight,
- the fact that once a hypothesis v is removed from the queue, it is not possible to generate another hypothesis v' with $t(v') < t(v)$ – that would contradict the monotonicity of the heuristic.

□

Proposition 4. *Assuming a tree-structured search space, the first complete hypothesis removed from the priority queue corresponds to the shortest path from R (empty hypothesis) to a target node (complete hypothesis) and, equivalently, to the highest-score translation of the input sentence.*[5]

[5] In fact, Prop. 4 also holds within the context where the search space is graph-structured. But in order to find the best-score translation, search tree is sufficient.

Proof. Follows directly from Prop. 3 and from the fact that, once a complete hypothesis v is removed from the queue, $t(v) = w(v)$. Therefore, for each subsequent (complete or not) hypothesis v' removed from the queue, $t(v') \geq t(v') = t(v)$. □

Finally, let's note that the hypothesis recombination-based pruning can be easily combined with A*. We just need to (i) keep track of the already visited nodes, and (ii) check, each time a hypothesis v is removed from the queue, that another hypothesis v' with which v can be recombined was not already visited; if so, v can be safely ignored.[6]

[6] An alternative is to recombine hypotheses already when adding them to the priority queue, but that is a bit more complicated and goes some way in the direction of a graph-based search space.