Mit 6.043 - Artificial Intelligence

Marco Filippone

November 25, 2020

1 Reasoning: goal trees and rule-based expert systems

A **semantic net** is a representation in which:

- Lexically, there are nodes, links, and application-specific link labels
- Structurally, each link conects a tail node to a head node
- Semantically, the nodes and links denote application-specific entities

With constructors that:

- Construct a node
- Construct a link, given a link label and two nodes to be connected

With readers that:

- Produce a list of all links departing from a given node
- Produce a list of all links arriving at a given node
- Produce a tail node, given a link
- Produce a head node, given a link
- Produce a link label, given a link

A **Semantic tree** is a representation, that is a semantic net in which:

- certain links are called *branches*. Each banch connects two nodes; the head node is called the *parent node* and the tail node is called the *child node*
- One node has no parent; it is called the root node. Other nodes have exactly one parent
- Some nodes have no children, they are called *leaf nodes*. When two nodes are connected to each other by a chain of two or more branches, one is said to be the *ancestor*; the other if said to be the descendant

With constructors that:

- Connect a parent node to a child node with a branch links

With readers that:

- Produce a list of a given node's children
- Produce a given node's parent

A **goal tree** is a semantic tree in which: nodes represent goals and branches indicate how you can achive goals by solving one or more subgoals. Each node's children corresponds to **immediate subgoals**; each node's parent corresponds to the **immediate supergoal**. The top node, the one with no parent, is the **root** goal.

Some goals are satisfied directly, without reference to any other subgoals. These goals are called **leaf goals**, and the corresponding nodes are called **leaf nodes**.

Because goal trees always involve *And* nodes, or *Or* nodes, or both. they are often called **And-Or trees**.

To determine whether a goal has been achieved, you need a testing procedure. The key procedure, *REDUCE*, channels action into the *REDUCE-AND* and the *REDUCE-OR*.

Goal trees enable introspective question answering:

- how: the immediate subgoal (downstream)
- why the immediate supergoal (downstream)

1.1 Eliciting expert systems features

- Heuristic of specific situations: it is dangerous to limit inquiry to office interviews
- Heuristic of situation comparison: ask a domain expert for clarification whenever the domain expert's behavior varies in situations that look identical to the knowledge enginner.
- You should build a system and see when it cracks. Helps identifiying missing rules.

2 Nets and Basic Search

A **search tree** is a representation, that is a semantic tree, in which:

- Nodes denote paths
- Branches connect paths to one-set path extensions

With writers that:

- Connect a path to a path description

WIth reades that:

- Produce a path's description

Each child denotes a path that is a one-city extension of the path denoted by its parent.

If a node has b children, it is said to have a **branching** factor of b. If the number of children is always b for every nonleaf node, the the tree is said to have a branching factor of b.

Each path that does not reach the goal is called a **partial path**. Each path that does reach the goal is called a **complete path**, and the corresponding node is called a **goal node**.

Nodes are said to be **open** until they can be expanded, whereupon they become **closed**.

2.1 Blind methods

2.1.1 Depth-first Search

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - Add the new paths, if any to the **front** of the queue
- If the goal node is found, annouce success; otherwise, annouce failure

Good when you are confident that all partial paths either reach dead ends or become complete paths after a reasonable number of steps.

Bad with long or infinite paths, that neither reach dead ends nor become complete paths.

2.1.2 Breadth-first Search

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node

  - Reject all new paths with loops
  - Add the new paths, if any to the **back** of the queue
- If the goal node is found, annouce success; otherwise, annouce failure

Works with trees with infinitely deep paths.
Wasteful when all paths lead to the goal node at more or less the same depth.

2.1.3 Nondeterministic search

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - Add the new paths **at random places in the queue**
- If the goal node is found, annouce success; otherwise, annouce failure

2.2 Heuristically informed methods

2.2.1 Hill climbing

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - **Sort the new paths, if any, by the estimated distances between their terminal nodes and the goal**
  - Add the new paths, if any to the **front** of the queue
- If the goal node is found, annouce success; otherwise, annouce failure

Possible issues:

- Foothills: an optimal point is found, but it is a local maximum
- Plateaus: for all but a small number of positions, all standard-step probes leave the quality measurement unchanged
- Ridges

2.2.2 Beam search

Like BFS in that ir progresses level by level. Unlike BFS, beam search moves downward only through the best w nodes at each level; the other nodes are ignored.

2.2.3 Beast-first search

While hill climbing demands forward motion from the most recently created open node. In the best-first seach, **forward motion is from the best open node so far** no matter where that node is i the partially developed tree.

2.3 Which search type is good for me?

- Depth-first search is good when unproductive partial paths are never too long
- Breadth-first search is good when the branching factor is never too large
- Nondeterministic search is good when you are not sure whether depth-first search or breadth-first search would be better
- Hill climbing is good when there is a natural measurement of goal distance form each place to the goal and a good path is likely to be among the partial paths that appear to be good at all levels
- Beam search is good when there is a natural measure of goal distance and a good path is likely to be among the partial paths that appear to be good at all levels
- Best-first search is good when there is a natural measure of the goal distance and a good partial path may look like a bad option before more promising partial paths are played out

3 Nets and Optimal Search

3.1 British Museum procedure

Find all possible paths and select the best one from them.

3.2 Branch and Bound

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empth:

  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - add the remaining new paths, if any, to the queue
  - Sort the entire queue by path length with least-cost paths in front
- If the goal node is gound, annouce success; otherwise, annouce failure

3.3 Branch and Bound with lower-bound estimate

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empth:
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - Add the remaining new paths, if any, to the queue
  - Sort the entire queue by **the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front**
- If the goal node is gound, annouce success; otherwise, annouce failure

3.4 Branch and Bound with dynamic programming

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empth:
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - Add the remaining new paths, if any, to the queue
  - **if two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost**
  - Sort the entire queue by **the sum of the path length with least-cost paths in front**
- If the goal node is gound, annouce success; otherwise, annouce failure

3.5 A* procedure - Branch and bound with Underestimates and Dynamic Programming

- Form a one-element queue consisting of a zero-length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empth:
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node
  - Reject all new paths with loops
  - Add the remaining new paths, if any, to the queue
  - **if two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost**
  - Sort the entire queue by **the sum of the path length and a lower-bound estimate of the cost remaining, with least-cost paths in front**
- If the goal node is gound, annouce success; otherwise, annouce failure

3.6 Which optimal search method is good for me?

- The British Museum procedure is good only when the search tree is small
- Branch-and-bound search is good when the tree is big and bad paths turn distinctly bad quickly
- Branch-and-bound search with a guess is good when there is a good lower-bound estimate of the distance remaining to the goal
- Dynamic programming is good when many paths convers on the same place
- The A* procedure is good when both branch-and-bound search with a guess and dynamic programming are good

4 Trees and Adversarial Search