Ken Hartsook
CS 8803 KBA
Project 1

1. Design

While designing my system, I attempted several different techniques for drawing inferences and forming causal chains.

a. Graph Algorithms

The first such technique was graph algorithms. While reading in the data, I created two types of frames: NounFrames and ActionFrames (for both facts and events). The fillers of these frames were all pointers to other frames so that the knowledge base could be traversed like a graph. This presented several unique opportunities to try and obtain knowledge from the graph. For instance, I applied a depth first search to the graph in order to find connected subsets which were disconnected from one another (essentially, isolated trees). I thought that since each of these trees represented nouns and actions which were connected, each such tree might contain a causal chain, and the trick would be flattening it into one.

Unfortunately, this technique did not work for two reasons. First, several of the causal chains contain items which are not connected via any nouns or verbs to other events in the chain. An example of this is (grabbed (dog, robber pants)). To connect this item to its chain, bootstrapped knowledge such as (wearing(robber, pants)) or (member-of(robber, robbers)) would have to be added. However, this was not the only issue. Completely unrelated causal chains were linked together as well. For example, the chains involving the robbers and the man assaulting the employee both contain a generic “car” noun. This noun caused the items from these two event chains to become linked. Finally, flattening these trees was next to impossible without a significant amount of bootstrapped knowledge.

Although this method was not completely effective for determining causal chains, the code was left in the system; it has the potential to be useful for determining an ordering between events if there are gaps in inferences. Network flow or shortest-path algorithms could also be used in the future to estimate connectivity between various people or events in the knowledge base.

b. First-Order Logic

The next approach I attempted was one which utilized first-order logic to represent the preconditions and postconditions of actions. Unfortunately, the complexity of attempting to implement a full FOL system quickly exploded and I was limited to a small subset. However, the remnants of this can be seen in the implementation of the preconditions and postconditions, as some variable values are allowed.

c. STRIPS-like Bootstrapping

The final approach I used was that recommended by Professor Goel in class; namely, hard-coding various preconditions and postconditions for action operators such that the events can be linked together by those conditions. While I am not completely satisfied with this approach due to the fact that it seems to require far more human effort than computer, it does produce acceptable results.

2. Architecture

This section will detail the architecture as I have implemented it.

Step 1: Initialize operators and knowledge base

First, my system reads an "operators.txt" file which contains a sizable set of STRIPS-like operators. That is, they are actions with preconditions and postconditions. For example:

> opened {%AGENT, %OBJECT}
> pre: {%OBJECT, state, closed}, {%AGENT, next-to, %OBJECT}
> post: {%OBJECT, state, open}

A condition is a 3-tuple, where the first item is either a role (action, object, location, coagent, time) or an atom (john-torch, boynton-labs, etc.), the second item is a slot name, and the third item is a filler value. In this example, in order for an AGENT to perform "opened" on an OBJECT, the specified OBJECT must be have a slot of "state" with value "closed"; similarly, AGENT must be "next-to OBJECT". When the operator is applied, OBJECT's "state" value changes to "open."

To initialize the knowledge base, the system adds initial states to a few of the objects in the world (four, actually). While this could have been accomplished by providing operators with no preconditions, it seemed to make more sense to do it this way.

Step 2: Read, parse, and apply facts

Next, my system reads in and parses facts. Since the facts for this system are in the form of actions, it creates an ActionFrame for each such action with slots for the following:

> Agents, Direct Objects, Locations, Co-agents, Times

While reading in each of these fillers, it also creates a NounFrame for each such noun. NounFrames are much more generic then ActionFrames, and contain two types of slots. The

first type of slot is a “connection” which links that noun (via a pointer) to an action in which it was a participant. The second type of slot is a “state slot” which contains a descriptor of the state or other useful information about the noun, as determined by facts and the effects of actions.

Some of these facts have “side effects” on the nouns in the form of postconditions. These are immediately applied as part of the initial world state to the nouns.

Step 3: Read and parse events

Events are read in just like the facts, with one exception – their postconditions are not applied to the world state, and they are not linked to the other data structures quite yet. Instead, they are stored in an “ActionHolder” queue.

Step 4: Determine Causal Chains

To find causal chains, the system scans the ActionHolder queue for all events whose preconditions are met by the initial world state. These are the initial events of causal chains. Then, for each of these chains, the system applies that initial event to the world state – this means changing NounFrames based on the operator’s postconditions as well as linking that event to its associated NounFrames. This process repeats itself until no more events can be added; that is the end of a causal chain. We then apply another initial event and continue doing so until all events have been applied.

While connecting causal chains, the system prints out each event as it is applied to the world state. It also prints out the postconditions of the event; since these postconditions are what are used to apply the next event in the chain, these are essentially the inferences being made to connect these events together.

There are some events whose preconditions are never met by the initial state or by the postconditions of other events. These events seem to be essentially “standalone”, and although they might be related to other causal chains, they are not direct causes or effects. As such, they are printed by themselves.

Step 5: Allow Querying

Now that all facts and events have been applied to the knowledge base, the knowledge base can be fully explored. The system provides a prompt asking the user for the name of a noun, fact, or event. These system will search for all frames of that name and display them to the user; this includes links to all of the frames to which it is connected. While not particularly sophisticated, this is a simple tool which allows the user to see all of the information stored in the frames and how they are linked together.

3. Evaluation

Given that a significant amount of hard-coding was a necessity for this assignment in order to produce meaningful causal chains, I believe that my system is quite effective.

My knowledge base of interleaving NounFrames and ActionFrames guarantees that absolutely no information is lost during the reading of facts and events. It also opens up several opportunities for performing graph algorithms on the data (and iterator abstractions have been made on the frames to make this easy) in order to analyze connectivity between various nouns and actions.

My operator evaluation system is also rather extensible. Although many of the operators have data-specific atoms hard-coded into them, many of them do not. They accomplish this by my usage of variables in the operator specifications. While standard STRIPs operators contain no arguments, many of my operators do, such as (grabbed(AGENT, OBJECT)). This makes many of my operators more general-purpose as they could potentially be used outside of this data set. Furthermore, it makes the operators that much more powerful as a new operator does not have to be defined for every combination of literals which could undertake a specific type of action.

One mistake I made in the implementation of my system was that I did not use a separate add-list and delete-list for the operators. As such, when a state slot on a noun is changed, that value is overridden and cannot be added. This means that a noun cannot have both “has (sword)” and “has (shield)” because changing the “has” slot overwrites the previous filler. Another problem of this is that to take away a slot, for instance if we want to delete “has (sword)” we have to either use “has (empty)” or “has(sword), has-not(sword)” (and assume precedence for later state slots). I chose the latter approach for this assignment but intend to change it to allow for multiple slots of the same name (with separate add and delete lists).

There are a very small number of operators which do not work quite properly, and such chains become broken. Considering the noisiness of the data and the fact that numerous facts and events do not have atoms in the correct positions, I consider this a very acceptable trade off. Second, after reviewing the data myself, I determined that some of the events were formed in such a way that most highly-intelligent systems would not be able to place them correctly. I made no such extra effort to place these in chains correctly.

Finally, I would like to point out one last benefit of my system. By going through the (painstaking) effort of creating pseudo-realistic operators with preconditions and postconditions, my system is more realistic in its ordering of events in a causal sequence. Specifically, if there are some events whose order is interchangeable, it has the potential to interchange them (unlike other potential approaches in which exact events were ordered).

While currently it does not print out these different partial-orderings, it could easily be extended to do so by scanning for all possible following events and displaying them simultaneously.