Robert Johns
CS 7637: Knowledge-Based AI
Final Exam
April 17, 2018

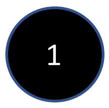
Knowledge-Based AI: Final Exam

Part I: An Affordable Economy Car

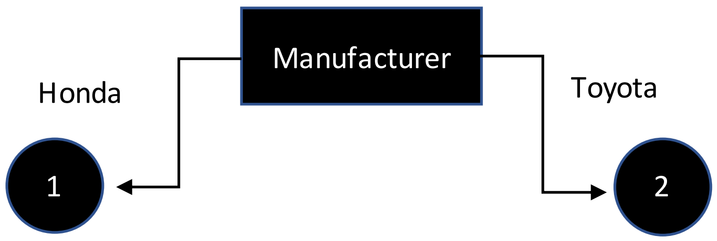
Example	Origin	Company	Color	Year	Type	Result
1	Japan	Honda	Blue	2008	Economy	Positive
2	Japan	Toyota	Green	2007	Sports	Negative
3	Japan	Toyota	Blue	2009	Economy	Positive
4	USA	Chrysler	Red	2008	Economy	Negative
5	Japan	Honda	White	2008	Economy	Positive

1. *Show how the examples given would be stored incrementally in a discrimination tree for case-based reasoning*

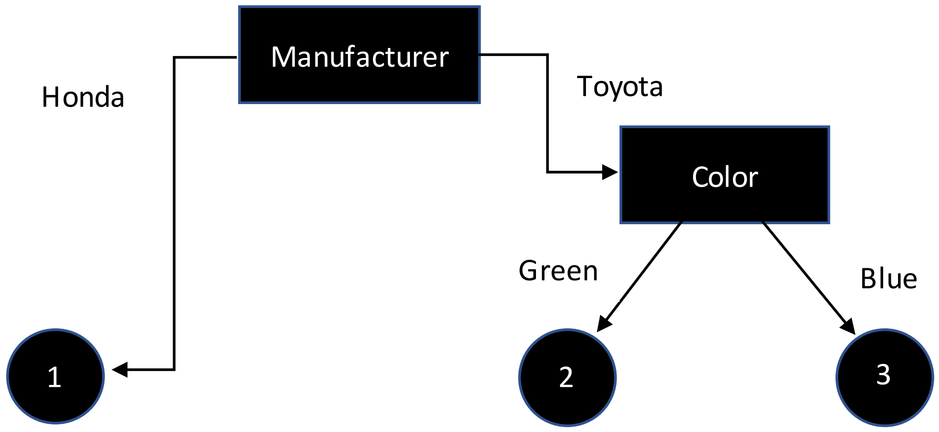
When building a discrimination tree incrementally, we change the structure of the tree each time we add new knowledge (Goel, Joyner & Thaker, 2016, p. 111). When our first example is processed, there is nothing to discriminate as it is the only example in the tree. Our initial tree thus looks like this:



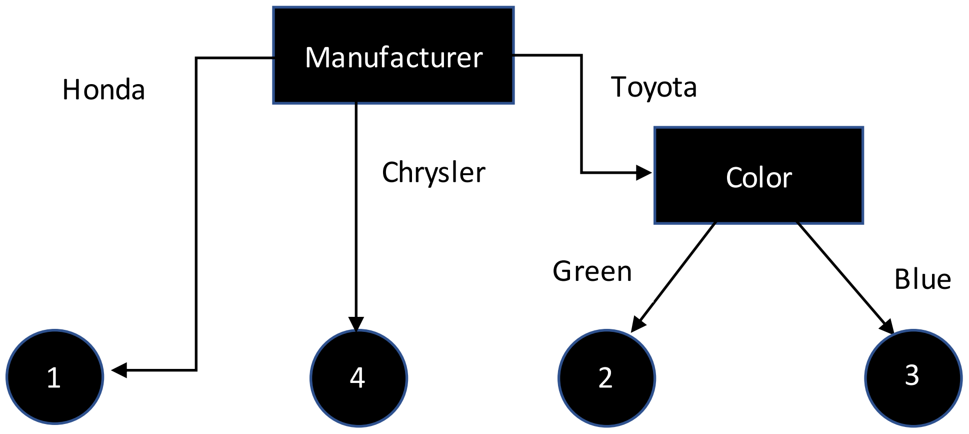
When our second example is processed, we must discriminate between the two. Our algorithm will discriminate based on the first column on which the two examples differ. In this case, that difference is in the "Company" column: Example 1 is a Honda, Example 2 a Toyota. Our second tree will thus look like this:



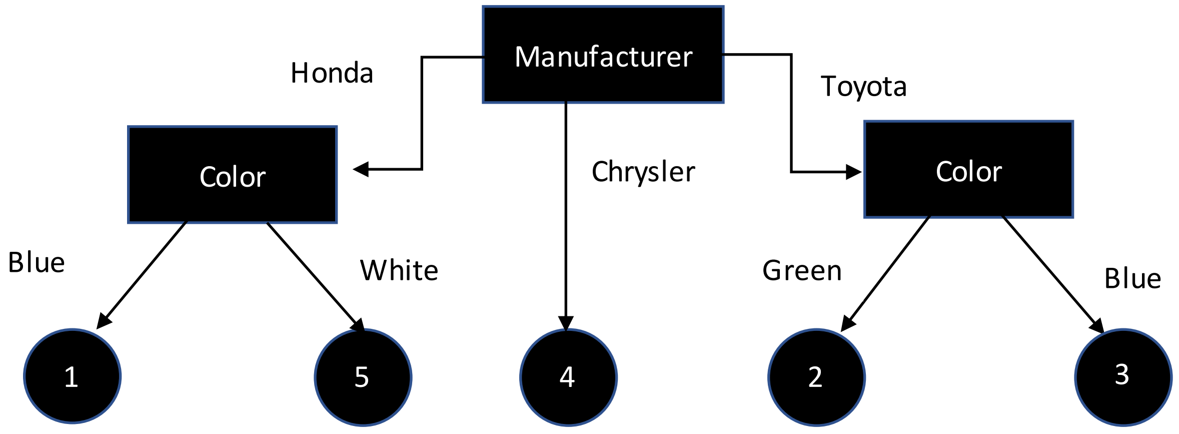
Continuing this process, our third example is, like Example 2, a Toyota; the two first differ in the "color" column; Example 3 is blue, while Example 2 is green. As such, we will discriminate in the right branch on color. Our third discrimination tree will look like this:



With the fourth example, we will discriminate based on “Manufacturer”, as it has a different value for “Manufacturer” than any of the previous examples. Our fourth tree will look like this:



The final example is a Honda, like Example 1, but differs first in the “Color” column; as such, we will differentiate based on color again. Our final tree will look like this:



Several advantages to this knowledge structure are apparent. First, it is algorithmically and computationally quick and efficient to execute: when a new node would occupy a taken space on the existing tree, simply differentiate those two examples based on the first column for which they have different values. Also, the tree has potential to be very wide, enabling fast search, as clearly shown in this problem, where five examples can be differentiated based on the answers to two questions.

However, a potential drawback exists: since the examples arrive one at a time, an optimal structure is not known beforehand, and an incremental building of the tree may result in a tree that is taller than it needs to be.

2. *Show how the concept of an affordable economy car will incrementally evolve with each example.*

In incremental concept learning, we start with one model of a concept and refine it incrementally based on sequential positive or negative examples, as opposed to version spaces, which have us converging two models together: one specific and one general (Goel, Joyner & Thaker, 2016, p. 122). The algorithm for incremental concept learning is as follows:

```
for each object:

    if the object is a positive example of the concept:
        if the object fits our current model:
            do nothing
        if the object does not fit our current model:
            generalize the model to fit the object

    if the object is a negative example of the concept:
        if the object fits our current model:
            specialize our model to exclude the object
        if the object does not fit our current model:
            do nothing
```

Figure 1: pseudo-algorithm for Incremental Concept Learning. (Goel, Joyner & Thaker, 2016, p. 122)

When given our first example, our model will be as specific as possible: each value will need to match the example exactly.

Origin	Company	Color	Year	Type
Japan	Honda	Blue	2008	Economy

Our second example is a negative example that does not fit our current model. As such, we will do nothing, and our model is unchanged:

Origin	Company	Color	Year	Type
Japan	Honda	Blue	2008	Economy

Our third example is a positive example that does not fit our model. As such, we need to generalize. The two columns in which the third example does not match our model are "Company" and "Year". There is an algorithmically simple ways to generalize these: we could simply allow any value for the two columns. While this method is simple and has its advantages, we will elect to employ more nuanced heuristics: the "close-interval" heuristic for "Year" and the "expand-set" heuristic for "Company". The close-interval heuristics has us expand the range of values allowed in a column (Goel, Joyner & Thaker, 2016, p. 130). This makes sense for the "Year" column, as it stands to reason that if, for example, a 1992 and a 1994 car are affordable, then a 1993 car will be as well. The "enlarge-set" heuristic (in which we allow, discretely, other options in a column) is logical for the "Company" column, as simply allowing any company into the column will likely result in luxury brands being labeled as "affordable" (Goel, Joyner & Thaker, 2016, p. 130). Our third iteration looks like this:

Origin	Company	Color	Year	Type
Japan	Honda | Toyota	Blue	2008-2009	Economy

Our fourth example is a negative example that does not fit our model. As such, we will do nothing, and our model is unchanged:

Origin	Company	Color	Year	Type
Japan	Honda | Toyota	Blue	2008-2009	Economy

Our fifth example is a positive example that does not fit our model. As such, we need to generalize. The only column in which the fifth example does not match our model is "Color"; thus far, all positive examples are required to be blue, whereas this positive example is white. In general, this would be a good opportunity to use the "climb-tree" heuristic, in which we use background knowledge to generalize the values allowed (Goel, Joyner & Thaker, 2016, p. 130). If our fifth example were a yellow car instead of a white car, our agent could use its knowledge that yellow and blue are both primary colors to generalize in a more nuanced way, allowing for any primary color to be used in a positive example. However, white is not a primary color, so we will allow any value for the "Color" column (designated by "*"), and we arrive at our final concept:

Origin	Company	Color	Year	Type
Japan	Honda | Toyota	*	2008-2009	Economy

In lecture, we discussed three other heuristics for incremental concept learning: require-link, forbid-link, and drop-link (Goel, Joyner & Thaker, 2016, p. 130). Those three are used for concepts that can be represented as graphs, like the "Foo" example given in lecture. A graph representation does not make as much sense in this case, as all columns (with the possible exception of "Origin" and "Company") are independent of each other and aren't related spatially or semantically.

Part II: AI Agent Design

3. Design an AI agent that can alternate between case-based and map-based methods to navigate urban areas.

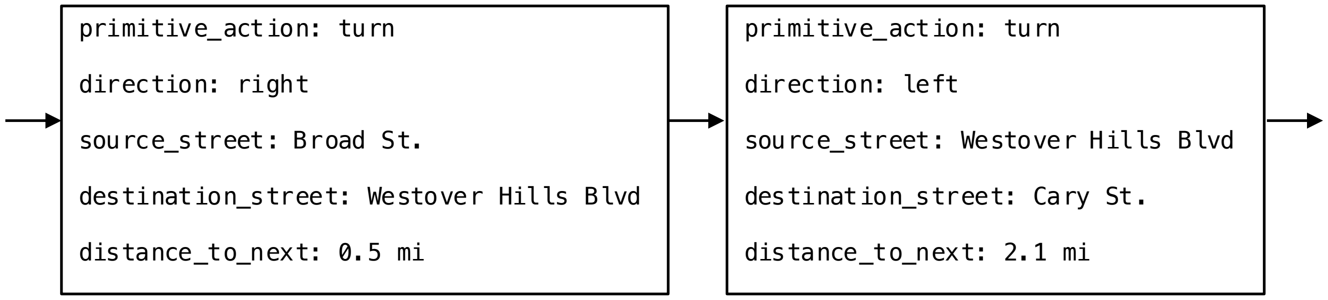
First, let us give an overall outline for the functionality of the agent. The agent will two locations take as primary input: a starting point A and a destination B. The agent will design a route, using either a case-based or map-based method (or both) to design the route. Once the route has been generated and sent to the user, the agent will prompt the user for feedback about the route. This feedback will be stored in two databases: one containing user data and driving preferences, and the other containing route/traffic data in a given area. This data will be used to better design routes for a particular user, and to improve information about an area for all users.

Next, let us define the knowledge representations and data structures the agent will use. The agent makes use of two main sources of knowledge: user knowledge and route knowledge. The user data will be stored as frames, like the one given below:

```
user: Robert Johns

lives-in: Richmond, VA

preferred-route-type: highway

traffic-aversion-ratio: 3.14

other-preferences: [list of preferences]

favorite-routes: [links to routes]
```

Each piece of this data can be used to personalize routes designed for the user. For example, if the system discerns that a user prefers to drive on a highway, it may design city-routes that have the user drive on an interstate for a short period of time, rather than on smaller city streets. The traffic-aversion-ratio will determine how many extra miles a user is willing to go to avoid one minute of traffic (defined as extra time to traverse a section of the route under current conditions); if Robert Johns has a traffic-aversion-ratio of 3.14, then the system will take him a maximum of 31.4 miles out of the way to avoid 10 minutes of traffic. This is an important parameter for people like me; I hate traffic enough that I have gone sixty miles out of my way to avoid traffic on I-95. The other-preferences slot is a catch-all for other information; for example, the user may be afraid to turn left in high-traffic areas or may be willing to drive a few extra miles to enjoy some nice scenery.

Routes will be stored as sequences of action frames representing the directions. In these action frames, the primitive actions will be different actions a driver can take on the road: turn, exit, u-turn, etc. Scripts will also be built, consisting of scenes representing generic expectations for different types of driving (interstate, city, etc.). An example portion of a route follows:



```
primitive_action: turn

direction: right

source_street: Broad St.

destination_street: Westover Hills Blvd

distance_to_next: 0.5 mi
```

```
primitive_action: turn

direction: left

source_street: Westover Hills Blvd

destination_street: Cary St.

distance_to_next: 2.1 mi
```

Now that our knowledge structures have been defined, let us describe our overall algorithm. Pseudocode follows:

```
get input (point_A, point_B) from user

if route exists in database that matches (point_A, point_B) exactly:
       return route

for each route in route_database that is close to (point A, point B):
       if route would satisfy user's preferences:
              adapt_route(route, user, point_A, point_B)
              return route

if no existing route is close enough:
       route = create_new_route(user, point_A, point_B)
       return route

feedback = get_user_feedback(route, user)
update_user_profile(route, user, feedback)
update_route_database(route, feedback)
```

We see that there are two main processes for this system: creating the route and obtaining/processing feedback. To accomplish these tasks, we make use of four functions: adapt_route() create_new_route(), get_user_feedback(), update_user_profile(), and update_route_database(). A couple of other important aspects of the agent are too simple to warrant their own function: for example, determining the closeness of a route to the user input can be done by simply obtaining the Euclidean distance to the start/endpoints of the two routes, and requiring that distance to be less than some threshold.

The function adapt_route() is where the case-based portions and map-based portions of the agent intersect. The agent will have searched through the route database to find a suitably close route in existence and will need to adapt that route to exactly match the input and suit the user's preferences. The agent can accomplish this by creating a new route (using the create_new_route() function) from the input start/end to the start/end of the route retrieved from the database, adjusting various aspects of the route to fit the user's profile.

The function create_new_route() relies purely on the map-based reasoning functionality of the agent. It will create a new route from scratch when no existing route is available connecting the start/end-points of the input. This function will also consider the user's preferences when designing the route.

The functions get_user_feedback(), update_user_profile(), and update_route_databases() are where the agent's learning happens: they are used to personalize the user's profile and update the route database with information the user may have provided about the conditions along the route. The useful data an agent is able to glean depends on the questions it asks: a potentially good set could consist of prompts like: "rate your satisfaction of this route from 1-10", or a dynamically generated set of yes/no questions based on choices the agent made, for instance whether the agent took the user *too* far out of the way to avoid traffic. Some of the route data will be stored only temporarily (it probably won't affect traffic in a week if a telephone line is down along a road), but some will be used to design routes into the future (e.g. determining how bad traffic gets in a particular section of the city at rush hour). These functions are what really drive the *learning* of the agent.

With the knowledge representations, basic functions and overall algorithm described, let us ruminate on the levels of functionality of our cognitive agent. Our agent functions at three cognitive levels: reaction, deliberation, and metacognition. At the reactive level, the agent will take the input from the user and will simply output the exact matching route from the database, if such a route exists.

At a higher level of cognition, the agent may need to deliberate before outputting a route for the user; this deliberation happens in the form of memory, learning and reasoning. The memory portion of the deliberation consists of the agent's two databases, and the cases stored therein. The reasoning occurs in the functions create_new_route() and adapt_route(); in these functions, the agent deliberates on how best to create a route for the user based on the user's personal driving preferences. The agent's learning occurs when it processes new data about a location; the agent is learning about traffic patterns, both immediate and long-term.

At the meta-cognition level, the agent will deliberate about its own deliberations. This mainly takes the form of adjusting the agent's parameters for a particular user. In re-adjusting the traffic-aversion-ratio of a user, or adding additional preferences to other-preferences, the agent is, in a way, adjusting its mode of thinking for a particular user, and improving its deliberation the next time a user asks for directions.

4. *Design an AI agent for a system that can deduce that an actor might be preparing to rob a bank.*

First, let us give an overview of the functionality of our agent. Our agent takes as inputs a series of events in the format (event (actor, object, location, time, instrument)), and will output an alert of some sort when it determines that a dangerous situation is imminent. The agent will accomplish this by working with two main knowledge structures: a short-term memory database containing recent events, and a long-term memory database containing cases from the past to which it can compare the current situation, and general information about objects and people (for example, the fact that a knife can be used as a weapon, or a list of suspected domestic terrorists). Our agent will use knowledge from these two sources, along with the steam of events, to determine if a situation is dangerous enough to warrant an alert. Our agent's main reasoning system will rest on analogical reasoning, and its meta-reasoning will come mainly in the form of learning by correcting mistakes.

Next, let us describe the knowledge structures the agent will employ. The agent needs to have three main types of knowledge in its short-term and long-term memory. First, the agent must have knowledge of actors and objects, stored in short-term memory to keep tabs on developing situations, and in long-term memory to aid in the judgement of the seriousness of future situations. These can be stored in frames, as follows:

```
object: .44—Magnum

object-type: gun

object-intent: personal harm

potential-dangers: robbery, murder, ...

danger-factor: 8.6

notable-incidents: [list of incidents]
```

```
person: Robert Johns

based-in: Richmond, VA

currently-in: Nowhere, VA

danger-factor: 0.0

associates: [other person frames]

notable-incidents: [list of incidents]
```

The agent must also have a knowledge structure for current situations, which will be stored as a connected sequence of action frames, (these will be the frames the agent gets from the input stream). The agent will connect action frames together by recognizing events that are occurring in the same location, or that involve the same actor(s). Depending on an event, the agent may also need to build a spatial model for the actors and object in a situation (for instance, the arrangement of suspicious persons in the lobby of a bank). The agent can store this spatial information as a semantic network, where the nodes are actors and objects and the edges describe the relationships between those objects. Lastly, the agent must have knowledge of past cases. These past cases are crucial for the agent to understand; they will form the basis of comparison for any developing situation. These past cases will be stored as scripts, abstract representations of different situation types (robbery, assault, etc.) to which the agent can compare the current situation.

Now that our knowledge structures are defined, let us describe our overall algorithm:

```
get input (event (actor, object, location, time, instrument)) from stream

if actor or location has an open situation in short-term memory:
      if get_threat_level(event) > 0:
            update_situation(situation, event)
            threat_level = update_threat_level(situation, event)

else if get_threat_level(event) > 0:
      create_new_situation(event)
      threat_level = update_threat_level(event, location)

if threat_level > threshold:
      send_alert(situation)
      get_feedback(situation)
      process_feedback(situation)
```

While the structure is clear, a few details deserve explanation. The send_alert() function is the foundation for the reactive functionality of the cognitive agent. The agent's reactive processing comes when a situation is clearly benign enough to be discarded (a child eating ice cream), or dangerous enough to warrant an immediate alert (a group of robbers swarming a bank).

The functions get_threat_level() and update_situation() form the core of the agent's deliberation structure, which consists of memory, reasoning, and learning. The memory aspect of its deliberation comes in the form of its short- and long-term memory databases about people and/or objects; that memory is crucial for the agent's judgements. For reasoning, the function get_threat_level() will process the threat level for any individual event in isolation: if the object or actor involved is considered dangerous, or if the location involved already has a developing situation, the threat level will be higher than if the actor and object are benign. The function update_situation() will add the event being processed to an open situation in the agent's short-term memory. The function update_threat_level() will process the threat level for a particular situation, and update it based on the event. This function is mainly based on analogical reasoning, where the agent will retrieve from memory a list of past events (stored as scripts, as described above) similar to the current situation, map and transfer details from the current event to those from the past, and return a value representing the similarity to the past event, and that past event's direness. The agent's deliberative learning takes place in the parameter adjustments that occur in update_threat_level(), in which a situation or actor will have their frames adjusted to match their role in the current situation.

Two other functions require explanation: get_feedback() and process_feedback(). These functions form the meta-cognitive processing of the agent and will help the agent learn not about specific people or objects, but about its own reasoning process to make better judgements in future situations. The function get_feedback() will prompt a user for the results of a given event: questions like "was the situation actually dangerous?", "if not, what was the actual situation", and "did authorities arrive in time?" are all useful for the agent to determine how to react in the future. Once the feedback has been obtained, process_feedback() will tweak parameters for people/objects, situational scripts, and update the agent's reasoning as necessary, mainly employing the methodology of learning by correcting mistakes. For example, the agent may initially treat anybody who is filming a bank as extremely suspicious, as it seems that they are surveying the bank before a possible robbery. However, if the agent were to send an alert after such an incident, and then learn that the actor was merely Michael Moore, filming a documentary about capitalism, the agent would then adjust its parameters, lowering the threat value for Michael Moore, or adding more nuance to its reasoning about surveillance devices.

Works Cited

Goel, Ashok; Joyner, David; Thaker, Bhavin (2016). *KBAI Ebook: Knowledge-Based Artificial Intelligence*. Publisher, city and state not given.