Introduction

Belief revision is the process of changing beliefs to take into account a new piece of information. This is a cognitive process that happens in our life every day as we learn and process information. When we are exposed to a new piece of information, it is probably not that case that we will believe it without a second thought. Our likelihood of accepting new information and using it to refine our beliefs can be based on a lot of things such as personal experience, source of information, frequency of exposure, and biases. Moreover, when we are refining our beliefs, we may be able to generate new beliefs by combining new information with existing beliefs, and this can be done in several ways. In this project, we aim to study a problem which involves belief revision and build a computational model for that. The problem we have picked is task prioritization, which is to determine the order of completing some tasks.

Background

There has been some research done on task prioritization, but they mostly focus on prioritization when multitasking, or prioritization in computer systems. Although these do not quite match the scenario we imagine, there are some components that we can take out and apply to our situation.

In [1], Parasuraman and Rovira suggest potential enhancements for the Improved Performance Research Integration Tool (IMPRINT), which is a discrete-event simulation and human performance modeling software tool primarily used by the U.S. Department of Defense to evaluate human performance when interacting with new and existing technology [2]. In the section where they talk about task scheduling and prioritization, they show that a general priority scheme makes the process of task prioritization more efficient by using the pilot training

example, in which pilots were separated into groups to receive different training. Some pilots were taught a specific task management procedure, while the rest were not exposed to the procedure. Results show that pilots who knew the procedure performed better with fewer mistakes. We can see that in order to perform efficient prioritization, we need to define high and low priority in specific procedures. We can do this by setting up clear guidelines to determine which task should be processed first. For example, we can look at what kind of task it is, how urgent it is, how hard it is, etc. In this paper, they also mentioned that the time deadline to complete all the tasks must also be taken into account. That means we have to do things in a way that ensures we have enough time for all awaiting tasks in our hands. In usual cases, we definitely perform the tasks with high priority first. But if those tasks require some resources that may take longer to acquire, maybe picking up other tasks with lower priority to "get them out of the way" is a more efficient choice.

In [3], Rhodes et al. focused on how memory storage can affect one's ability of processing information. When there are multiple tasks needed to be processed concurrently, we need a way to determine which task to give our attention and time to. One way to decide which task to prioritize is by looking at the cost of each task. They introduce two terms for that - prioritization cost and concurrence cost. Prioritization cost refers to any decline in performance when we shift our emphasis away from a task, and concurrence cost refers to any drop in performance when we shift from single-task processing to dual-task processing. Concurrence cost might not be relevant in our project since we assume that we are able to only process one task at a time. Prioritization cost might be important to our project but it probably will be defined in a little different way. In our scenario, the prioritization cost can be related to the time we have left and the number of remaining tasks. If we choose to prioritize one task, can we make sure that

the resources it consumes will not affect the completion of the other tasks? Currently the only given resource in our project is time. To be specific, time is what we will trade for completion of tasks. We might also want to consider having a time limit for each day because in real life, we only have a certain amount of time allocated for certain tasks, and we would like to get tasks done while utilizing our limited time. Similar to the idea of time deadline in the previous paragraph, if a task with high priority is hard to complete and will take a very long time, maybe shifting our focus to some easier low-priority tasks would be more efficient. In order to do that, we need to think about how to represent the difficulty of a task, so we went on to do some research on that.

Difficulty representation is actually very common in video games. Let's take Super Mario as an example. The way they measure difficulty and set an order to the levels is by testing and calculating the amount of players that are able to get through the level without losing a life and such. If all the players are able to get through the level without losing a life, the difficulty compared to a level where one player lost a life, would be easier since more people were able to complete it without consequence. But this is pretty complicated to implement and doesn't apply to our world that well. What we might end up doing would just be categorizing task difficulty, such as easy, medium, difficult, or extra credit. We will try to mimic the tasks we have in real life and take our personal experience into consideration.

Methods

In our proposed computational model, the beliefs would be the tasks we want to prioritize. Our belief set will include the type, the due date, and the difficulty of each task. To reason with the beliefs, we reorder the tasks based on the three aforementioned factors. Tasks

with a more important type (e.g. academic) will be on the top of the list. If two tasks have the same type, then priority is determined by their due dates. The task with the sooner due dates will be prioritized. If two tasks have the same due date, our prioritization will be determined by their difficulties. We can pick whichever is easier to do first.

There are two conditions for possible belief revision, which are 1) when the user adds a new task; or 2) when the user modifies an existing task. When adding a task, we first check if the task exists in the belief set. We do not do anything if it is an existing task. If this task does not exist in the belief set, we do not simply append it to the belief set. Instead, we compare this new task with every existing task in the belief set to decide where to place it, then add it at the determined position. When modifying a task, we have to remove the task being modified from the belief set because it is contradicting the new version we are going to have. After that, we follow a similar approach in adding, where we compare this modified task to the rest of the belief set to determine its new position in the set. The techniques for prioritization of two tasks are mentioned in the first paragraph.

Implementation

In this section, we will show how our model works with a valid running example:

```
?- prioritizer.
Task Prioritizer...
There is no task currently.
Add a task...
|: cs project due 12/1 hard

Here is the list of tasks, ordered in terms of their priority levels:
```

```
1. cs project  due:12/1 difficulty:hard
```

When we start the program, there is no task in the belief set yet, so we prompt the user to add a task. Since the belief set is empty, there is no task to compare to, and the new task automatically becomes our priority regardless of its type, due date, or difficulty.

```
Type 1 to add a task...
Type 2 to modify a task...
Type  to end this program...
|: 1
Add a task...
|: slides for club meeting on 11/30 easy

Here is the list of tasks, ordered in terms of their priority levels:
1. cs project  due:12/1 difficulty:hard
2. slides for club meeting due:11/30 difficulty:easy
```

Now that we have an existing task, modifying a task becomes an option. When the user adds another task, there are two tasks in the belief sets, which is enough to perform prioritization. In this case, the CS project is of type “academic,” while the slides for club meeting is of type “eca.” We defined “academic” as a type with higher priority, so the CS project is prioritized.

```
Type 1 to add a task...
Type 2 to modify a task...
Type  to end this program...
|: 1
Add a task...
```

```
|: math homework due 12/3 normal

Here is the list of tasks, ordered in terms of their priority levels:
1. cs project  due:12/1 difficulty:hard
2. math homework  due:12/3 difficulty:normal
3. slides for club meeting due:11/30 difficulty:easy
```

Here, a new task math homework of type “academic” is added to the belief set. Based on the reasoning we had, the math homework has a higher priority than the slides for club meeting. However, it has the same type as the CS project, which is currently our priority. To determine whether it will replace our current priority, we have to look at their due dates now. Since the CS project is due earlier, it should be placed before the math homework, thus it remains as our priority.

```
Type 1 to add a task...
Type 2 to modify a task...
Type  to end this program...
|: 2
Type the index of the task you want to modify...
1. cs project  due:12/1 difficulty:hard
2. math homework  due:12/3 difficulty:normal
3. slides for club meeting due:11/30 difficulty:easy
|: 2
Type 1 to modify task type...
Type 2 to modify task name...
Type 3 to modify task due date...
```

```
Type 4 to modify task difficulty...
|: 3
Type task new due date (Month/Date)...
|: 11/30

Here is the list of tasks, ordered in terms of their priority levels:
1. math homework  due:11/30 difficulty:normal
2. cs project  due:12/1 difficulty:hard
3. slides for club meeting due:11/30 difficulty:easy
```

Let's say the user later realizes that the math homework is actually due earlier, so now the due date for that needs to be modified. The user can pick the task that needs to be modified and change the corresponding details. Since the math homework is modified, it has to be removed from the belief set and added again with different information. In this case, the math homework is now due earlier than the CS project, so it becomes the current priority, and the CS project gets pushed back.

```
Type 1 to add a task...
Type 2 to modify a task...
Type  to end this program...
|: exit
Bye!
true.
```

There is an option for the user to end this program. The belief set will be cleared once the program ends.

Discussion

Our implemented computational model is successful in prioritizing tasks based on three factors: types of tasks, due dates, and difficulties of tasks. A user is able to add new tasks to the belief set and modify the details of existing tasks. When any action that may change the belief set is performed, the model will reconsider the order of the tasks. So the main goal of this model is accomplished. Using this model as a framework, one can easily make modifications to make it fit into a different scenario, e.g., a business determining which orders to fulfill first. Similar to our world, the order that was placed the earliest should be processed first. There are also cases where a customer would pay extra to have express shipping or such, then that order should be moved up to the top of the list. This can be taken care of by specifying the type of the order.

There are a few limitations in our implemented computational model. To start off, we simplified our prioritization factors down to just three as mentioned above. We did not consider the specific amount of time needed for a task, which is actually important in a real world scenario. However, one can claim that time is considered one of the factors, which is the difficulty of a task. Since we did not define difficulties explicitly, when a user is deciding the difficulty of a particular task, one can take the amount of time needed into consideration. Another limitation is that this model does not show any bias, which is something present in humans. When we are deciding which task to prioritize, we always have certain tasks in our heads that we definitely want to get done first, and some that we just want to postpone forever. We do not really have an effective way to show that in the model reasonably. Moreover, this model only depicts the mindset of one person, but everybody may have a unique way to decide one's priority. We do not know if the mindset we have chosen is efficient enough, and we don't really have other systems to compare our model to. Therefore, this model may not be as good

when it comes to generality. If we have more time to work on this model, one thing we would definitely like to improve is error handling. In the current version, if a user enters some inputs that cannot be parsed, the program will end with a "false" without any explanation of failure. To fix this, we will have to add an extra case in the code every time we are reading an input from the user. If the put cannot be parsed, we have to let the user know the correct format of input, and repeat the action of reading input until the program receives a reasonable input and can parse it without any problem.

Conclusion

Belief revision is a complex process that is really difficult to be accurately represented by a computer program, so our computational model is definitely not enough to show how humans prioritize tasks in general. In limited time, we successfully implemented a model that only depicts specific situations, and we identified some limitations of our current model. Improvements can be made to make this model accept more variations of inputs and become more user-friendly. Nonetheless, we are satisfied with the outcome.

Bibliography

[1] Parasuraman, R., & Rovira, E. (2005). Workload Modeling and Workload Management: Recent theoretical developments. https://doi.org/10.21236/ada432181

[2] Rusnock, C. F., & Geiger, C. D. (2013). Using discrete-event simulation for cognitive workload modeling and system evaluation. IIE Annual Conference. Proceedings, 2485-2494.

[3] Rhodes, S., Jaroslawska, A. J., Doherty, J. M., Belletier, C., Naveh-Benjamin, M., Cowan, N., Camos, V., Barrouillet, P., & Logie, R. H. (2019). Storage and processing in working memory: Assessing dual-task performance and task prioritization across the adult lifespan. Journal of Experimental Psychology: General, 148(7), 1204–1227. https://doi.org/10.1037/xge0000539

Appendix

Current version of the prolog code:

```
academic(project).

academic(homework).

work(tutor).

work(ta).

eca(club).


dueMonth(1).

dueMonth(2).

dueMonth(3).

dueMonth(4).

dueMonth(5).

dueMonth(6).

dueMonth(7).

dueMonth(8).

dueMonth(9).

dueMonth(10).

dueMonth(11).

dueMonth(12).


dueDay(1).

dueDay(2).

dueDay(3).

dueDay(4).

dueDay(5).
```

```
dueDay(6).

dueDay(7).

dueDay(8).

dueDay(9).

dueDay(10).

dueDay(11).

dueDay(12).

dueDay(13).

dueDay(14).

dueDay(15).

dueDay(16).

dueDay(17).

dueDay(18).

dueDay(19).

dueDay(20).

dueDay(21).

dueDay(22).

dueDay(23).

dueDay(24).

dueDay(25).

dueDay(26).

dueDay(27).

dueDay(28).

dueDay(29).

dueDay(30).

dueDay(31).
```

```
parse_due(Type, String, [due, DueMonth, (/), DueDay, Difficulty],
Task) :-
    dueMonth(DueMonth), dueDay(DueDay),
    Task = [Type, String, [DueMonth, DueDay], Difficulty].
parse_due(Type, String, [on, DueMonth, (/), DueDay, Difficulty],
Task) :-
    dueMonth(DueMonth), dueDay(DueDay),
    Task = [Type, String, [DueMonth, DueDay], Difficulty].
parse_due(Type, String, [First, due, DueMonth, (/), DueDay,
Difficulty], Task) :-
    string_concat(String, First, Name),
    dueMonth(DueMonth), dueDay(DueDay),
    Task = [Type, Name, [DueMonth, DueDay], Difficulty].
parse_due(Type, String, [First, on, DueMonth, (/), DueDay,
Difficulty], Task) :-
    string_concat(String, First, Name),
    dueMonth(DueMonth), dueDay(DueDay),
    Task = [Type, Name, [DueMonth, DueDay], Difficulty].
parse_due(Type, String, [First|Rest], Task) :-
    string_concat(String, First, Temp),
    string_concat(Temp, " ", NextString),
    parse_due(Type, NextString, Rest, Task).
parse_task([First|Rest], Task) :- string_concat(First, " ", String),
    parse_task(String, Rest, Task).
parse_task([First|Rest], Task) :-
```

```
    downcase_atom(First, LowerFirst), academic(LowerFirst),
    string_concat(First, " ", NextString),
    parse_due(academic, NextString, Rest, Task).
parse_task([First|Rest], Task) :-
    downcase_atom(First, LowerFirst), work(LowerFirst),
    string_concat(First, " ", NextString),
    parse_due(academic, NextString, Rest, Task).
parse_task([First|Rest], Task) :-
    downcase_atom(First, LowerFirst), eca(LowerFirst),
    string_concat(First, " ", NextString),
    parse_due(academic, NextString, Rest, Task).
parse_task(String, [First|Rest], Task) :-
    downcase_atom(First, LowerFirst), academic(LowerFirst),
    string_concat(String, First, Temp),
    string_concat(Temp, " ", NextString),
    parse_due(academic, NextString, Rest, Task).
parse_task(String, [First|Rest], Task) :-
    downcase_atom(First, LowerFirst), work(LowerFirst),
    string_concat(String, First, Temp),
    string_concat(Temp, " ", NextString),
    parse_due(work, NextString, Rest, Task).
parse_task(String, [First|Rest], Task) :-
    downcase_atom(First, LowerFirst), eca(LowerFirst),
    string_concat(String, First, Temp),
    string_concat(Temp, " ", NextString),
    parse_due(eca, NextString, Rest, Task).
```

```
parse_task(String, [First|Rest], Task) :- string_concat(String,
First, Temp),
    string_concat(Temp, " ", NextString),
    parse_task(NextString, Rest, Task).

indexList([_], [1]).
indexList([Element|Rest], Indices) :- indexList(Rest, RestIndices),
    length([Element|Rest], Length),
    append(RestIndices, [Length], Indices).

display_tasks([Index], [[_, Name, [DueMonth, DueDay], Diff]]) :-
write(Index), write(". "),
    write(Name), write(" due:"), write(DueMonth), write("/"),
write(DueDay),
    write(" difficulty:"), write(Diff), nl.
display_tasks([Index|RestIndices], [[_, Name, [DueMonth, DueDay],
Diff]|Rest]) :- write(Index), write(". "),
    write(Name), write(" due:"), write(DueMonth), write("/"),
write(DueDay),
    write(" difficulty:"), write(Diff), nl,
display_tasks(RestIndices, Rest).

show_current_state([]) :- write("There is no task currently."), nl.
show_current_state(Tasks) :- write("Here is the list of tasks,
ordered in terms of their priority levels:"),
    nl, indexList(Tasks, Indices), display_tasks(Indices, Tasks).
```

```
add_task(Task, Tasks, Tasks) :- member(Task, Tasks),
    write("This task exists in the list already."), nl.
add_task(Task, [], [Task]).
add_task(Task, Tasks, NewTasks) :- prioritize(Task, Tasks, NewTasks).

dueSooner([DueMonth1, _], [DueMonth2, _]) :-
    DueMonth1 < DueMonth2.
dueSooner([DueMonth, DueDay1], [DueMonth, DueDay2]) :-
    DueDay1 < DueDay2.

prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]], NewTasks) :-
    (Type1 = academic, Type2 = work ;
     Type1 = academic, Type2 = eca ;
     Type1 = work, Type2 = eca),
    NewTasks = [[Type1, Name1, Due1, Diff1], [Type2, Name2, Due2,
Diff2]].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]], NewTasks) :-
    (Type2 = academic, Type1 = work ;
     Type2 = academic, Type1 = eca ;
     Type2 = work, Type1 = eca),
    NewTasks = [[Type2, Name2, Due2, Diff2], [Type1, Name1, Due1,
Diff1]].
```

```
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]], NewTasks) :-
    Type1 = Type2, Due1 = Due2,
    (Diff1 = easy, Diff2 = normal ;
     Diff1 = normal, Diff2 = hard ;
     Diff1 = easy, Diff2 = hard),
    NewTasks = [[Type1, Name1, Due1, Diff1], [Type2, Name2, Due2,
Diff2]].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]], NewTasks) :-
    Type1 = Type2, Due1 = Due2,
    (Diff2 = easy, Diff1 = normal ;
     Diff2 = normal, Diff1 = hard ;
     Diff2 = easy, Diff1 = hard),
    NewTasks = [[Type2, Name2, Due2, Diff2], [Type1, Name1, Due1,
Diff1]].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    (Type1 = academic, Type2 = work ;
     Type1 = academic, Type2 = eca ;
     Type1 = work, Type2 = eca),
    NewTasks = [[Type1, Name1, Due1, Diff1], [Type2, Name2, Due2,
Diff2]|Rest].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    (Type2 = academic, Type1 = work ;
```

```
     Type2 = academic, Type1 = eca ;
     Type2 = work, Type1 = eca),
    prioritize([Type1, Name1, Due1, Diff1], Rest, PartNewTasks),
    NewTasks = [[Type2, Name2, Due2, Diff2]|PartNewTasks].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    Type1 = Type2, dueSooner(Due1, Due2),
    NewTasks = [[Type1, Name1, Due1, Diff1], [Type2, Name2, Due2,
Diff2]|Rest].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    Type1 = Type2, dueSooner(Due2, Due1),
    prioritize([Type1, Name1, Due1, Diff1], Rest, PartNewTasks),
    NewTasks = [[Type2, Name2, Due2, Diff2]|PartNewTasks].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    Type1 = Type2, Due1 = Due2,
    (Diff1 = easy, Diff2 = normal ;
     Diff1 = normal, Diff2 = hard ;
     Diff1 = easy, Diff2 = hard),
    NewTasks = [[Type1, Name1, Due1, Diff1], [Type2, Name2, Due2,
Diff2]|Rest].
prioritize([Type1, Name1, Due1, Diff1], [[Type2, Name2, Due2,
Diff2]|Rest], NewTasks) :-
    Type1 = Type2, Due1 = Due2,
    (Diff2 = easy, Diff1 = normal ;
```

```
     Diff2 = normal, Diff1 = hard ;
     Diff2 = easy, Diff1 = hard),
    prioritize([Type1, Name1, Due1, Diff1], Rest, PartNewTasks),
    NewTasks = [[Type2, Name2, Due2, Diff2]|PartNewTasks].

find_task(1, [Task|_], Task).
find_task(Index, [_|Rest], Task) :-
    Next is Index - 1, find_task(Next, Rest, Task).

remove_task([Task|Rest], Task, Rest).
remove_task([DiffTask|Rest], Task, TempTasks) :-
    remove_task(Rest, Task, Temp), TempTasks = [DiffTask|Temp].

determine_type([1], academic).
determine_type([2], work).
determine_type([3], eca).

determine_due([Month, (/), Due], [Month, Due]).

determine_diff([1], easy).
determine_diff([2], normal).
determine_diff([3], hard).

modify([1], [_, Name, Due, Diff], [NewType, Name, Due, Diff]) :-
    write("Type 1 for academic..."), nl,
    write("Type 2 for work..."), nl,
```

```
    write("Type 3 for eca..."), nl,
    read_word_list(List),
    determine_type(List, NewType).
modify([2], [Type, _, Due, Diff], [Type, String, Due, Diff]) :-
    write("Type task new name..."), nl,
    read_word_list(List), atomics_to_string(List, String).
modify([3], [Type, Name, _, Diff], [Type, Name, NewDue, Diff]) :-
    write("Type task new due date (Month/Date)..."), nl,
    read_word_list(List),
    determine_due(List, NewDue).
modify([4], [Type, Name, Due, _], [Type, Name, Due, NewDiff]) :-
    write("Type 1 for easy..."), nl,
    write("Type 2 for normal..."), nl,
    write("Type 3 for hard..."), nl,
    read_word_list(List),
    determine_diff(List, NewDiff).

modify_task([Index], Tasks, NewTasks) :-
    find_task(Index, Tasks, Task),
    remove_task(Tasks, Task, TempTasks),
    write("Type 1 to modify task type..."), nl,
    write("Type 2 to modify task name..."), nl,
    write("Type 3 to modify task due date..."), nl,
    write("Type 4 to modify task difficulty..."), nl,
    read_word_list(List),
    modify(List, Task, ModifiedTask),
```

```
    prioritize(ModifiedTask, TempTasks, NewTasks).

process_action([1], Tasks, NewTasks) :-
    write("Add a task..."), nl,
    read_word_list(List),
    parse_task(List, Task),
    add_task(Task, Tasks, NewTasks).
process_action([2], Tasks, NewTasks) :-
    write("Type the index of the task you want to modify..."), nl,
    indexList(Tasks, Indices), display_tasks(Indices, Tasks),
    read_word_list(List),
    modify_task(List, Tasks, NewTasks).

prioritizer :-
    Tasks = [],
    write("Task Prioritizer..."), nl,
    show_current_state(Tasks),
    write("Add a task..."), nl,
    read_word_list(List),
    (List = [exit] -> write("Bye!"), nl, !;
    (parse_task(List, Task),
    add_task(Task, Tasks, NewTasks),
    nl, prioritizer(NewTasks))), !.

prioritizer(Tasks) :- show_current_state(Tasks), nl,
    write("Type 1 to add a task..."), nl,
```

```
    write("Type 2 to modify a task..."), nl,
    write("Type <exit> to end this program..."), nl,
    read_word_list(List),
    (List = [exit] -> write("Bye!"), nl, !;
    (process_action(List, Tasks, NewTasks),
    nl, prioritizer(NewTasks))), !.

read_word_list(Ws) :-
    read_line_to_codes(user_input, Cs),
    atom_codes(A, Cs),
    tokenize_atom(A, Ws).
```