Overcooked Implementation

Robbie Buxton, Alfie Chenery, Daniel Marks

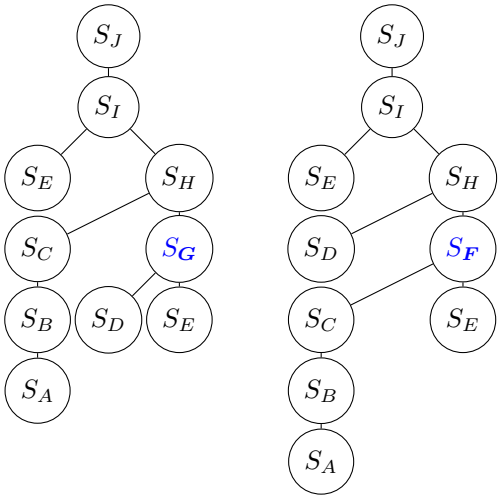
20 November 2023

1 Hidden Markov Model

Let S_i be a given state and E_A the number of occurrences to the produced item in the environment.

S_A = **grab_raw_burger**, E_A = **raw_burgers**.
S_B = **cook_raw_burger**, E_B = **burger_pans**.
S_C = **grab_cooked_burger**, E_C = **cooked_burgers**.
S_D = **grab_cheese**, E_D = **cheeses**.
S_E = **grab_bap**, E_E = **baps**.
S_F = **combine_bap_burger**, E_E = **bap_burgers**.
S_G = **combine_bap_cheese**, E_G = **bap_cheeses**.
S_H = **combine_bap_cheese_burger**
,E_H = **bap_cheese_burgers**.
S_I = **combine_bap_cheese_burger_bap**
,E_I = **bap_cheese_burger_baps**.
S_J = **deliver_cheeseburger**. E_j is irrelevant.

There are two different dependency trees for manufacturing a burger where S_G and S_F are the different mutually exclusive states.



Using the knowledge from these two trees and the limitations of the space (only two pans) we create some state dependency rules about if a state is available or not. $A(S_i|E)$ gives true if the state i is possible to reach this transition, false otherwise.

$A(S_J|E) = E_I > 0$
$A(S_I|E) = E_E > 0 \wedge E_H > 0$
$A(S_H|E) = (E_C > 0 \wedge E_G > 0) \vee (E_F > 0 \wedge E_D > 0)$
$A(S_G|E) = E_D > 0 \wedge E_E > 0$
$A(S_F|E) = E_E > 0 \wedge E_C > 0$
$A(S_E|E) = \top$
$A(S_D|E) = \top$
$A(S_C|E)|E) = E_B > 0$
$A(S_B|E) = E_A > 0 \wedge E_B < 2$
$A(S_A|E) = \top$

So for our transition matrix the unnormalised probability of transitioning from state i to j,

$$P_i, j = \begin{cases} w_{i,j}, & \text{if } A(S_j|E) = \top \\ 0, & \text{otherwise} \end{cases}$$

where $w_{i,j}$ is the transition weight (assuming it's possible) which will be composed of a few factors (more detail below), and the normalisation factor $k = \frac{\sum_{i=1}^{10} w_{i,j}}{\sum_{i=1}^{10} P_{i,j}}$ such that $\sum_{i=1}^{10} kP_{i,j} = 1$ for any j.

To calculate the transition weights. We first calculate all shortest paths to the input piece for all valid next states. We do this by using first a Breadth First Search to find the first occurrence of a valid input piece while labelling how many tiles we have traveled to get there. We then use a Depth First Search to find all paths back to the source that strictly decrease which gives us the set of shortest paths. The example below shows the paths generated and tiles labeled for the user looking for the shortest path for `cook_raw_burger` where S is the user and D are `empty_pans`.

The paths which are returned as arrays of positions are used to calculate the distance of the player away from the state they are trying to achieve and to calculate the next user input required to get closer to that

#	#	#	#	#	#	#
#	1	S	1	2	3	#
#	2	1	2	3	?	D
#	3	#	3	?	?	#
#	D	#	#	#	#	#

action. For example the example before would result in 0.5:`left` and 0.5:`down` as the first move.

So to calculate our weights we use a variety of metrics.

$$w_{i,j} = f(i)h(i)j(i)$$

where

$f(i)$ gets smaller the further away from the required piece the player is (sum of all distances / this distance).

$h(i)$ reduces the probability of any state that have happened in the last two ticks to a quarter of it's value.

$j(i)$ reduces the probability of any state that didn't have a key press expected by the last emissions matrix.

Each key press we multiply the state probabilities from last tick by the new transmission matrix. This gives us the current state probabilities. We select the state with the highest probability as our prediction. We use these state probabilities to multiply with our emissions matrix to get the chances of each key press.

We use the chances of each key press in generating the next ticks transitions matrix.

2 Robot Collaboration Algorithm

To control the robot in the Coop Cookbot world we decided to use a fixed policy. This policy was used since we conclude that we can craft a near optimal policy simply by inspecting the problem, without the time overhead of training a reinforcement learning agent. However, this policy is equivalent to an MDP with deterministic state transitions. We have just defined these transitions our self, instead of learning them over time.

2.1 State Space

Conceptually we control the robot using a very similar approach to the states defined in Section 1 for the HMM. The robot holds some internal state which represents the high level action it is trying to complete. From looking at the world map we can see that the robot only has 3 high level actions it needs to perform. It needs to give the player baps, it needs to cook burgers that the player gives it, and it needs to deliver cooked burgers back to the player. It is also possible that there is no useful work the robot can do, if for example the player is not helping. We want our robot policy to be robust to a difficult player, and so we add an additional idle state. We decide that if the robot is idle it will check the environment for if it can do something useful and change state. Once the robot has a non idle state, it will remain in this state until the action is completed. We notice that all the non idle states can be broken down in the same way and handled the same. The robot will move to some start position, grab the desired item, move to some end position, and finally place the item. As such all states can be described as a single parameterised state move(start_pos, end_pos, desired_hand) or the idle state.

2.2 Algorithm

The robot control algorithm follows a simple structure.

```
if in idle state:
  update state by checking environment
if state is still idle:
  move to a convenient place which is close to where we might be needed soon
else:
  if hand is empty:
    move to start_pos, pickup item
  if hand contains desired item:
    move to end_pos, place item
```

There are some additional checks for if the pickup or place step fails. This is possible if the player interacts with the desired position while the robot is on its way there. In these cases the robot will enter a special "dump" state in which it will look for any free space to empty its hand before returning to idle and deciding a new state. Moving to a position is simply done by calculating the difference in x and y coordinates between the robot and the position. The robot then defines a path to the position and moves along this path. Note the path planning algorithm used is much simpler than the one used in Section 1, and is still optimal, but requires the assumption that the space the robot can move in is static and strictly convex, which for this map is true.
The most interesting step of this algorithm is updating the robot's state based on the environment.

2.3 Determining state from the environment

In order for the robot to determine which state it should be in, it considers the content of all the spaces it can interact with. The robot does not consider the contents of places it cant control, such as spaces it can't reach or the player's hand. The spaces the robot considers are broken into 3 main regions:
The Table: The 3 counter cells which separate the player and the robot. These 3 cells are the only ones that both the player and robot can interact with.
The Pans: The 2 cooking pan cells.
Storage: The 4 other counter cells that the robot can reach. These are used to dump the robot's hand contents if it was unable to place the item at its desired end_pos. Since the player can't reach these storage cells, the robot will not fail to put an item here, if it sees a free space. If the robot needs an item it always checks storage first in an attempt to prevent storage getting full.

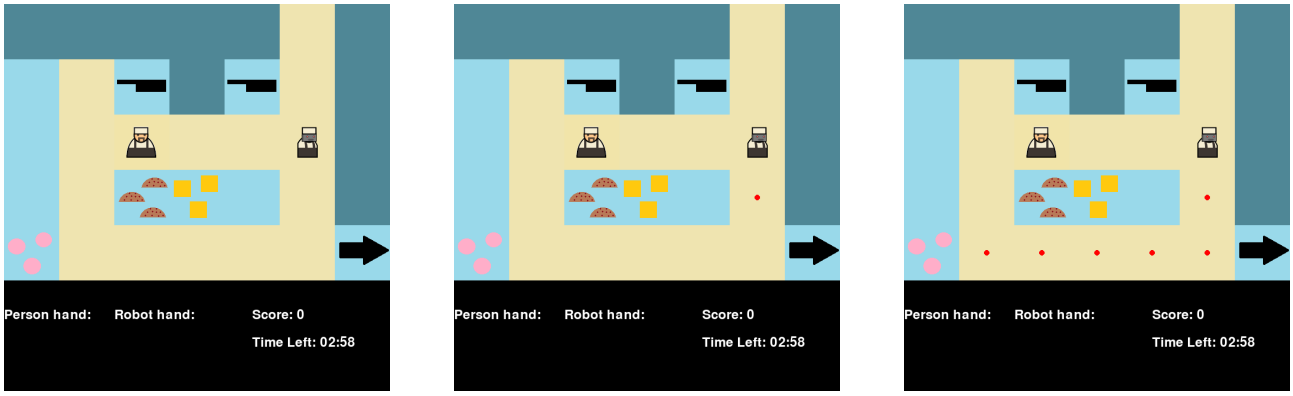
The robot will assign its state defined by setting its internal start_pos, end_pos and desired_hand fields according to a set of rules. For this full set of rules, see `robot_determine_state(robot)` of the `Cookbot_Coop.py file`.

```
If a partially completed burger on the table needs a bun or burger, place it on it.
If a raw burger is on the table and there is an empty pan, cook the burger.
If both pans are full, put one of the burgers on the table.
If the table has 2 free slots, place a bun in it.
Else idle.
```

3 User Study Conditions

Our user study investigates to what extent the robot communicating its intended movement assists the user, allowing them to perform better? The 3 conditions are described below:

Condition 1: No plan information: In the first condition the user gets no information about the plan of the robot. The player will need to infer the robot's plan from its current position and the world environment for there self.
Condition 2: Plan direction information: In the second condition the user can see the direction the robot plans to move next. At the next 0.5 second time step the robot will move in this direction, and will then display its plan for the next time step. If the path the robot was planning to take becomes blocked, it will recalculate, and may change which direction it wishes to move next.
Condition 3: Full plan path information: In the third condition, the user can see the full path which the robot is planning to take from its current position to its target cell. As with Condition 2, if the player blocks the robot's desired path, it will recalculate a new one and the player will see this new path.



(a) Condition 1 - The player does not see any of the robot's planned path

(b) Condition 2 - The player can see the direction the robot intends to move in next

(c) Condition 3 - The player can see the full planned path of the robot

Figure 1: Three conditions of the user study

The implementation for the conditions for our user study involved first improving the game in general. We first implemented different sprites for the player and robot which convey which way the respective agent is facing. This information is regarding the current state of the player and robot and does not give information about future plans, so will be available in all conditions.
After this we implemented a robot control algorithm similar to the approach used in Section 2. As part of this algorithm the robot finds the shortest path to the item it wants to interact with and stores this as a field of the robot object.
As a result, the implementation for specific conditions is very simple. Since the robot already calculates its entire desired path, we simply decide how much of this path to display to the user. For every cell coordinate in the robot's desired path, we can overlay a special image which will appear on top of the floor. In condition 1, the player receives none of the path information so no cells will be overlayed. In Condition 2, the player receives only the first element of the path. This is the coordinate of the next cell the robot wishes to move to, conveying to the player the general direction the robot wishes to move. In condition 3 the player receives the full path. They are able to see every cell the robot wishes to traverse in order to reach its goal. These different conditions are shown in Figure 1.
In Figure 1 you can also see the updated map we use to test our user study. This map has been specifically designed to create interesting challenges for the player and robot. In particular, the island counter which allows a short path between the burger mount and the pans, or a longer route around the outside. This short path will be preferred by the robot unless the player is blocking the path, in which case it will be forced to take the long way. As such knowing the robot's planned route in later conditions will likely be helpful.
The map contains a new cell type added specifically for this user study. The wall, shown by the darker blue squares, are just impassable cells. They are like counters but the player and robot can not place items on them. This new cell type was added to force the player and robot to interact more, as most of the space for assembling burgers is now concentrated mostly on the left of the map, blocking the shorter path to the pans.
Additionally, we decided to reverse our initial decision to shorten the game time. We believe that a 2 minute game would not be long enough for significant differences between the conditions to become apparent. As such we leave the game time as 3 minutes.