

Prompt

You are a language model agent tasked with carrying out a problem-solving algorithm. The algorithm is called LLM-PFC. The algorithm utilizes a number of modules. The modules are described below:

- **TaskDecomposer:** The TaskDecomposer receives the current state x and a goal y and generates a set of subgoals Z that will allow you to gradually work toward the final goal.
- **Actor:** The Actor receives the current state x and a subgoal z and proposes B potential actions $A = a_{b=1} \dots a_{b=B}$. The Actor can also receive feedback ϵ from the Monitor about its proposed actions.
- **Monitor:** The Monitor assesses the actions proposed by the Actor to determine whether they are valid (e.g., whether they violate the rules of a task). It emits an assessment of validity σ , and also feedback ϵ in the event the action is deemed invalid.
- **Predictor:** The Predictor receives the current state x and a proposed action a and predicts the resulting next state \tilde{x} .
- **Evaluator:** The Evaluator receives a next-state prediction \tilde{x} and produces an estimate of its value v in the context of goal y .
- **TaskCoordinator:** The TaskCoordinator receives the current state x and a subgoal z and emits an assessment Ω of whether the subgoal has been achieved. When the TaskCoordinator determines that all subgoals (including the final goal) have been achieved, the plan is emitted to the environment as a series of actions.

The LLM-PFC algorithm takes a state x and a goal y and generates a plan P , a series of actions with a maximum length of T . The TaskDecomposer first generates a set of subgoals Z . The agent then pursues each individual subgoal z in sequence, followed by the final goal y . At each time step, Search (a subroutine) is called to generate an action and a predicted next-state. Actions are added to the plan until the TaskCoordinator determines that the goal has been achieved, or the plan reaches the maximum length T . Here is a formal specification of the LLM-PFC algorithm:

Function: LLM-PFC(x , y , T , L , B):

```
 $P \leftarrow [ ]$  // Initialize plan
 $Z \leftarrow \text{TaskDecomposer}(x, y)$  // Generate subgoals
for  $g$  in  $1 \dots \text{length}(Z) + 1$  do
    if  $g \leq \text{length}(Z)$ 
        then  $z \leftarrow Z_{\{g\}}$  // Update current subgoal
    else
         $z \leftarrow y$  // Final goal
```

```

end
 $\Omega \leftarrow \text{TaskCoordinator}(x, z)$  // Initialize subgoal assessment
while  $\Omega$  is false and  $\text{length}(P) < T$  do
     $a, x, v \leftarrow \text{Search}(l = 1, L, B, x, z)$  // Perform search
     $P \leftarrow [P, a]$  // Update plan
     $\Omega \leftarrow \text{TaskCoordinator}(x, z)$  // Determine if subgoal is achieved
end
end
return P

```

LLM-PFC uses a Search subroutine. This involves a tree search with a depth of L layers, with B branches at each layer l . For each branch, a proposed action is sampled, and the Predictor predicts the next state \tilde{x} . This process continues recursively until the terminal layer L , at which point the value $v_{l=L}$ of the terminal states is estimated by the Evaluator. The values are backpropagated to their parent states in the first layer, and the action that leads to the most valuable state is selected. This process is accelerated by caching the actions and predicted states from deeper search layers and then reusing them in subsequent searches. The TaskCoordinator is also employed to prematurely terminate search if the goal state is achieved. Here is a formal specification of the Search subroutine:

```

Function: Search( $l, L, B, x, y$ ):
 $V_l \leftarrow \{ \}$  // Initialize value record
 $\tilde{X}_l \leftarrow \{ \}$  // Initialize next-state record
 $A_l \leftarrow \text{ProposeAction}(x, y, B)$  // Propose B actions
for  $b$  in  $1 \dots B$  do
     $\tilde{x}_b \leftarrow \text{Predictor}(x, A_b)$  // Predict next state
     $\tilde{X}_l \leftarrow \tilde{X}_l \cup \{ \tilde{x}_b \}$  // Update next-state record
     $\Omega \leftarrow \text{TaskCoordinator}(\tilde{x}_b, y)$  // Terminate search if goal
    // achieved
    if  $l < L$  and  $\Omega$  is false then
         $a_{l+1}, \tilde{x}_{l+1}, v_{l+1} \leftarrow \text{Search}(l + 1, L, B, \tilde{x}_b, y)$ 
        // Advance search depth
         $V_l \leftarrow V_l \cup \{ v_{l+1} \}$  // Update value record
    else
         $v_b \leftarrow \text{Evaluator}(\tilde{x}_b, y)$  // Evaluate predicted state
         $V_l \leftarrow V_l \cup \{ v_b \}$  // Update value record
    end
end
 $v_l \leftarrow \max(V_l)$  // Maximum value (randomly sample if equal value)
 $a_l \leftarrow A_l[\text{argmax}(V_l)]$  // Select action
 $\tilde{x}_l \leftarrow \tilde{X}_l[\text{largmax}(V_l)]$  // Predicted next-state
return  $a_l, \tilde{x}_l, v_l$ 

```

The Search subroutine uses a ProposeAction subroutine. ProposeAction takes a state x and a goal y and generates B potential actions $A = a_{b=1} \dots a_{b=B}$. This is implemented via a loop, in which the Actor first proposes potential actions, and the Monitor then assesses those actions according to certain constraints (e.g., task rules), providing feedback if any of the actions are deemed to be invalid. This continues until the proposed actions are considered valid. Here is a formal specification of the ProposeAction subroutine:

```
Function: ProposeAction( $x, y, B$ ):
 $\sigma \leftarrow$  false // Initialize validity
 $E \leftarrow \{ \}$  // Initialize feedback
while  $\sigma$  is false do
     $A \leftarrow$  Actor( $x, y, E, B$ ) // Sample B actions
     $\sigma, \epsilon \leftarrow$  Monitor( $x, A$ ) // Determine validity and provide feedback
     $E \leftarrow E \cup \{ \epsilon \}$  // Accumulate feedback
end
return  $A$ 
```

You will implement LLM-PFC to solve the following puzzle problem:

Problem description:

- There are three lists labeled A, B, and C.
- There is a set of numbers distributed among those three lists.
- You can only move numbers from the rightmost end of one list to the rightmost end of another list.

Rule #1: You can only move a number if it is at the rightmost end of its current list.

Rule #2: You can only move a number to the rightmost end of a list if it is larger than the other numbers in that list.

A move is valid if it satisfies both Rule #1 and Rule #2. A move is invalid if it violates either Rule #1 or Rule #2.

Each of the modules also receive their own specific instructions. Here are the instructions that are given to each module:

TaskDecomposer instructions:

...

Goal: The goal is to generate a single subgoal from the current configuration, that helps in reaching the goal configuration using the minimum number of moves.

To generate a subgoal use the goal recursion strategy. First if the smallest number isn't at the correct position in list C, then set the subgoal of moving the smallest number to its correct

position in list C. But before that, the numbers larger than the smallest number and present in the same list as the smallest number must be moved to a list other than list C. This subgoal is recursive because in order to move the next smallest number to the list other than list C, the numbers larger than the next smallest number and present in the same list as the next smallest number must be moved to a list different from the previous other list and so on.

Note in the subgoal configuration all numbers should always be in ascending order in all the three lists.

Here are two examples:

Example 1:

This is the current configuration:

A = [0, 1]

B = [2]

C = []

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Answer:

I need to move 0 from list A to list C.

Step 1. Find the numbers to the right of 0 in list A. There is 1 to the right of 0.

Step 2. Find the numbers larger than 0 in list C. There are none.

I will move the numbers found in Step 1 and Step 2 to list B. Hence I will move 1 from list A to list B. Also numbers should be in ascending order in list B.

Subgoal:

A = [0]

B = [1, 2]

C = []

Example 2:

This is the current configuration:

A = [1]

B = [0]

C = [2]

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Answer:

I need to move 0 from list B to list C.

Step 1. Find the numbers to the right of 0 in list B. There are none.

Step 2. Find the numbers larger than 0 in list C. There is 2 which is larger than 0.

I will move the numbers found in Step 1 and Step 2 to list A. Hence, I will move 2 from list C to list A. Also numbers should be in ascending order in list A.

Subgoal:

A = [1, 2]

B = [0]

C = []

...

Actor instructions:

...

Goal: The goal is to end up in the goal configuration using the minimum number of moves.

Here are two examples:

Example 1:

This is the starting configuration:

A = [0, 1]

B = [2]

C = []

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Here is the sequence of minimum number of moves to reach the goal configuration from the starting configuration:

Move 2 from B to C.

A = [0, 1]

B = []

C = [2]

Move 1 from A to B.

A = [0]

B = [1]

C = [2]

Move 2 from C to B.

A = [0]

B = [1, 2]

C = []

Move 0 from A to C.

A = []

B = [1, 2]

C = [0]

Move 2 from B to A.

A = [2]

B = [1]

C = [0]

Move 1 from B to C.

A = [2]

B = []

C = [0, 1]

Move 2 from A to C.

A = []

B = []

C = [0, 1, 2]

Example 2:

This is the starting configuration:

A = [1]

B = [0]

C = [2]

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Here is the sequence of minimum number of moves to reach the goal configuration from the starting configuration:

Move 2 from C to A.

A = [1, 2]

B = [0]

C = []

Move 0 from B to C.

A = [1, 2]

B = []

C = [0]

Move 2 from A to B.

A = [1]

B = [2]

C = [0]

Move 1 from A to C.

A = []

B = [2]

C = [0, 1]

Move 2 from B to C.

A = []

B = []

C = [0, 1, 2]

At each time step, give two different valid next moves possible from the starting configuration that would help in reaching the goal configuration using as few moves as possible.

Your answer should be in the format as below:

1. Move <N> from <src> to <trg>.

...

Monitor instructions:

...

Goal: The goal is to check if the proposed move satisfies or violates Rule #1 and Rule #2 and based on that if it is a valid or invalid move.

Here are two examples:

Example 1:

This is the initial configuration:

A = []

B = [1]

C = [0, 2]

Proposed move:
Move 0 from C to B.

Answer:

First check whether the move satisfies or violates Rule #1. Index of 0 in list C is 0. Length of list C is 2. The difference in length of list C and index of 0 in list C is 2, which is not equal to 1. Hence 0 is not at the rightmost end of list C, and the move violates Rule #1.

Next check whether the move satisfies or violates Rule #2. For that compute the maximum of list B, to which 0 is moved. Maximum of list B is 1. 0 is not larger than 1. Hence the move violates Rule #2.

Since the Move 0 from list C to list B violates both Rule #1 and Rule #2, it is invalid.

Example 2:

This is the initial configuration:

A = []

B = [1]

C = [0, 2]

Proposed move:
Move 2 from C to B.

Answer:

First check whether the move satisfies or violates Rule #1. Index of 2 in list C is 1. Length of list C is 2. The difference in length of list C and index of 2 in list C is 1. Hence 2 is at the rightmost end of list C, and the move satisfies Rule #1.

Next check whether the move satisfies or violates Rule #2. For that compute the maximum of list B, to which 2 is moved. Maximum of list B is 1. 2 is larger than 1. Hence the move satisfies Rule #2.

Since the Move 2 from list C to list B satisfies both Rule #1 and Rule #2, it is valid.

...

Predictor instructions:

...

Goal: The goal is to predict the configuration of the three lists, if the proposed move is applied to the current configuration.

Here are two examples:

Example 1:

This is the current configuration:

A = []
B = [1]
C = [0, 2]

Proposed move:
Move 2 from list C to list B.

Answer:
A = []
B = [1, 2]
C = [0]

Example 2:
This is the current configuration:
A = []
B = [1]
C = [0, 2]

Proposed move:
Move 1 from list B to list A.

Answer:
A = [1]
B = []
C = [0, 2]

...

Evaluator instructions:

...

Goal: The goal is to predict the minimum number of valid moves required to reach the goal configuration from the current configuration.

Here are two examples:

Example 1:

This is the current configuration:
A = [0, 1, 2]
B = []
C = []

This is the goal configuration:

A = []
B = []
C = [0, 1, 2]

Answer:

The minimum number of valid moves required to reach the goal configuration from the current configuration is 7.

Example 2:

This is the current configuration:

A = [1, 2]
B = [0]
C = []

This is the goal configuration:

A = []
B = []
C = [0, 1, 2]

Answer:

The minimum number of valid moves required to reach the goal configuration from the current configuration is 4.

The following heuristic function (proposed by GPT-4) may be helpful when estimating distance:

GPT-4: "A suitable heuristic function for this problem is the "sum of the distances" heuristic. This heuristic estimates the minimum number of valid moves required to reach the goal configuration by calculating the sum of the distances each number needs to travel to reach its final position in the goal configuration.

Here's how the heuristic function works:

1. For each number in the current configuration, determine its current position (list and index) and its goal position (list and index) in the goal configuration.
2. Calculate the distance between the current position and the goal position for each number. The distance can be calculated as the absolute difference between the indices of the current and goal positions, plus a penalty if the number needs to move to a different list.
3. Sum the distances calculated in step 2 for all numbers.

The heuristic function will return the sum of the distances, which is an estimate of the minimum number of valid moves required to reach the goal configuration from the current configuration.

This heuristic is admissible because it never overestimates the cost of reaching the goal configuration. It considers the minimum number of moves required for each number to reach its goal position, without taking into account the constraints imposed by the rules of the puzzle. Therefore, the actual number of moves required to reach the goal configuration will always be greater than or equal to the heuristic value.”

Use this heuristic function to predict the minimum number of valid moves required to reach the goal configuration from the current configuration.

Please provide your answer according to the heuristic function in the format as below:
The minimum number of valid moves required to reach the goal configuration from the current configuration is <N>.

...

TaskCoordinator instructions:

...

Goal: The goal is to predict whether the current configuration matches the goal configuration or not.

Here are two examples:

Example 1:

This is the current configuration:

A = []

B = []

C = [0, 1, 2]

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Answer:

The current configuration matches the goal configuration. Hence yes.

Example 2:

This is the current configuration:

A = [0, 1]

B = [2]

C = []

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Answer:

The current configuration doesn't match the goal configuration. Hence no.

...

You will now solve an actual problem. Remember to do so by implementing the algorithms stated above (LLM-PFC, Search, and ProposeAction), using the modules described above (TaskDecomposer, Actor, Monitor, Predictor, Evaluator, and TaskCoordinator), and keeping in mind the specific instructions for each module.

Here is the problem:

This is the current configuration:

A = [2]

B = [1]

C = [0]

This is the goal configuration:

A = []

B = []

C = [0, 1, 2]

Response

First, let's use the TaskDecomposer to generate a subgoal:

Current configuration:

A = [2]

B = [1]

C = [0]

Goal configuration:

A = []

B = []

C = [0, 1, 2]

We need to move 0 from list C to list C (its final position).

Step 1. Find the numbers to the right of 0 in list C. There are none.

Step 2. Find the numbers larger than 0 in list C. There are none.

Since there are no numbers to move, the subgoal is the same as the current configuration.

Now, let's use the LLM-PFC algorithm to find the plan:

1. Initialize the plan $P = []$
2. Set the subgoal $Z = \text{TaskDecomposer}(x, y) = [A = [2], B = [1], C = [0]]$
3. For each subgoal, perform the following steps:
 - a. Check if the subgoal is achieved using TaskCoordinator. If yes, move to the next subgoal.
 - b. If the subgoal is not achieved, perform a search using the Search subroutine to find the best action.
 - c. Update the plan with the best action found in the search.
4. Repeat steps 3a-3c until all subgoals are achieved or the plan reaches the maximum length T .

Now, let's perform the search using the Search subroutine:

1. Initialize the value record V_I and the next-state record X_I .
2. Propose B actions using the ProposeAction subroutine.
3. For each proposed action, predict the next state using the Predictor module.
4. Update the next-state record with the predicted next state.
5. If the current search depth is less than L and the goal is not achieved, perform a search at the next depth.
6. If the current search depth is L or the goal is achieved, evaluate the predicted state using the Evaluator module.
7. Update the value record with the evaluated value.
8. Select the action with the maximum value and return the action, predicted next state, and value.

Here's the plan found using the LLM-PFC algorithm:

1. Move 1 from B to A .

$A = [1, 2]$

$B = []$

$C = [0]$

2. Move 0 from C to B .

$A = [1, 2]$

$B = [0]$

$C = []$

3. Move 2 from A to C .

$A = [1]$

$B = [0]$

$C = [2]$

4. Move 0 from B to A .

$A = [0, 1]$

$B = []$

$C = [2]$

5. Move 1 from A to C .

$A = [0]$

$B = []$

$C = [1, 2]$

6. Move 0 from A to C .

$A = []$

$B = []$

$C = [0, 1, 2]$

The plan is now complete, and the goal configuration has been achieved.