Algorithm 1: The proposed APRMCTS algorithm

```
Input: orig_buggy_program $bp_o$, orig_test_result $t_o$,
    patch_generator $\pi_p$, patch_evaluator $\pi_r$, max_round $T$,
    branch $b$, exploration_constant $\epsilon$, quit_at_first_plausible
    $quit$.
 1: $T_q$ ←Initialize_Tree($bp_o, t_o$)
 2: $\pi_p, \pi_r$ ←Initialize_Models($\pi_\theta, \pi_\theta$)
 3: for $i$ in range($T$) do
 4:   $C$ ←root($T_q$)
 5:   ————Patch Selection————
 6:   while $C$ is not leaf node do
 7:     $C \leftarrow argmax_{C' \in \text{children}(C)}(\bar{X}_C + \epsilon\sqrt{\frac{\ln n_C}{n_{C'}}})$
 8:   end while
 9:   ————Patch Generation————
10:   $bp_{old}, t_{old}$ ←Extract_Bug_Info($C$)
11:   for $j$ in range($b$) do
12:     $cot$ ←Get_CoT($\pi, bp, t$)
13:     $P_j, ref$ ←Repair($\pi, bp_{old}, t_{old}, cot$)
14:     $t_{new}$ ←Validate_Patch($P_j$)
15:     if Is_Pass($t_{new}$) and $quit$ then
16:       Return $P_j$
17:     else
18:       ————Patch Evaluation————
19:       $r$ ←Get_Reward($\pi_r, P_j, t_{new}, cot, ref$)
20:       $b_{new}$ ←Construct_Bug($bp_{old}, P_j$)
21:       $T_q$ ←Update_Tree($b_{new}, t_{new}$)
22:     end if
23:   end for
24:   ————Patch Tree Updating————
25:   Back_Propagate($C, r$)
26: end for
27: $P$=Get_Plausible_Patches($T_q$)
28: Return $P$
Output: $P$
```

APPENDIX

A. Algorithm Detail

We present the pseudocode of APRMCTS as shown in the Algorithm 1. We have annotated the range of each stage in the algorithm, and the comments describe what is done in each stage. It is worth noting that although a fixed number of searches max_round is used for APRMCTS in the experiments of this paper, in real-world scenarios, we improve the performance and efficiency of the search by enabling the $quit\_at\_first\_plausible$ switch. When this switch is turned on, APRMCTS returns after finding the first plausible patch. Compared to other repair methods, APRMCTS offers greater flexibility and does not require the rigid process of generating N patches followed by sequential patch filtering and ranking.

B. Prompt Design

We use perfect fault localization in this paper, following previous work [1] we divide bugs into 3 different types: single-line (can be fixed by replacing/adding a single line), single-hunk (can be fixed by replacing/adding a continuous code hunk), single-function (can be fixed by generating a new function to replace the original buggy version). We also design different prompt according to types. For single-line and single-hunk bugs, prompts for repair are like prompt (a) shown in Figure 1, we provide masked code and removed buggy line or hunk. The prompt for repairing single-function bugs is like prompt (b), only the buggy function is needed.

Prompt for CoT Before we ask the patch generator to generate patches, we first use prompt (c) in Figure 1 to direct it to generate CoT. CoT is further combined with prompt (a) and (b) together used for repairing.

Prompt for LLM-as-Judge After the patch generator generates patches, we use prompt (d) in Figure 1 to ask the patch evaluator to give a score on the generated patches.

C. Full Results on Defects4J and QuixBugs

We have implemented APRMCTS with 14 different LLMs in total, and the full experimental results on Defects4J and QuixBugs are shown in Table I.

D. Usage of APRMCTS

APRMCTS is designed to generate plausible patches for bugs under the conditions of provided fault locations and test cases. In practice, APRMCTS is very useful in software testing scenarios, particularly during routine software testing (e.g., regression testing). When code triggers bugs that fail to pass all test cases, APRMCTS can automatically generate patches for the bugs, which are then subject to manual validation, significantly reducing the time required for manual bug fixing. Additionally, APRMCTS can be integrated into an automated program repair framework as a component for patch generation, working in conjunction with other components such as fault localization and patch ranking to achieve full automation. Furthermore, APRMCTS can leverage the latest LLMs as both the patch generator and evaluator, continuously improving the effectiveness of repairs.

E. Repo-level Context Helps Fix Complex Bugs: A Case Study

We analyze the bugs that are not correctly fixed by APRMCTS and find that an important reason for the repair failure is lack of sufficient context. In our study, in order to facilitate comparison with baselines, we follow the baseline's setup by limiting the context scope to the function level. However, in Defects4J, some function-level bugs still rely on cross-function and even cross-class context, thus function-level context only is not sufficient for repairing such bugs.

Using Closure_20 as an example, the developer patch for Closure_20 is shown in Figure 2, which includes three if conditions. APRMCTS (GPT-4o-mini) is able to generate the first two if conditions, but due to the third if condition depending on a method in the utility class $NodeUtil$, APRMCTS (GPT-4o-mini) cannot generate a completely correct answer. Instead, APRMCTS (GPT-4o-mini) generates a plausible patch that satisfies some of the conditions based on the function-level context.

~~~
The following code contains a buggy line/hunk that has been removed.
```java
......
>>>[[INFILL]]<<<
......
~~~
This was the original buggy line/hunk which was removed by the infill location:
......
Please provide the correct line/hunk at the infill location.
```

(a) Prompt for repairing single-line/single-hunk bugs

```
The following code contains a bug.
```java
......
```
Please provide the correct function.
```

(b) Prompt for repairing single-function bugs

```
Before you give the final answer, let's think step by step. You need to
explain where bug happens and how your answer can avoid it.
```

(c) Prompt for getting chain of thought before repairing

```
Please give a score between 0 and 100, the score stands for quality of the patch, 0 means the patch is of very poor quality, 100 means the patch is correct.

**(d) Prompt for evaluation after repairing and testing**

Fig. 1: Prompt design of AprMcts

```java
if (value != null && value.getNext() == null
    && NodeUtil.isImmutableValue(value))
```

Fig. 2: Developer patch of Closure_20.

For bugs like Chart_20 that require repository-level context, we develop a tool for extracting code-level context. specifically, we extract the following types of context as shown in Table II. In terms of implementation, we exclude the test classes from the project and perform static analysis on the remaining classes using Spoon. We use Neo4J to build a knowledge graph that contains six types of relationships: Import, Extend, Contain, Implement, Invoke, and Dependency. Then, for each bug in the project, we extract both in-class and cross-class contexts from the knowledge graph for persistence. We further utilize LLMs to generate code summary for excessively long methods. To limit the context window, we adopt both agentless and agent methods. The agentless method relies on developers to select the necessary context based on experience, while the agent method allows LLMs to extract context as needed.

We find that both the agent and agentless methods can help fix Chart_20. After the model analyzes the need to determine whether $value$ is immutable, it utilizes the utility methods from the $NodeUtil$ class to fill in the buggy code.

In summary, the repository-level context provides additional information that aids in fixing bugs, especially in cases where external class information is required and cross-class calls are necessary.

References

[1] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

TABLE I: Comparison of correct/plausible fix between Vanilla LLMs and APRMCTS on Defects4J and QuixBugs, including three types of bugs, single-line (SL), single-hunk (SH) and single-function (SF).

Category	Model	Patch Size	SL	SH	SF	Defects4J	QuixBugs
3B	Stable-Code-3B	16	39/56	4/12	15/31	58/99	20
	Stable-Code-3B (APRMCTS)	16	40/58	5/13	17/35	62/106	-
	Calme-3.1-3B	16	28/46	2/3	15/37	45/86	19
	Calme-3.1-3B (APRMCTS)	16	26/41	2/3	16/39	44/83	-
	Starcoder2-3b	16	30/52	8/16	5/11	43/79	18
	Starcoder2-3b (APRMCTS)	16	32/55	9/18	7/15	48/88	-
	Qwen2.5-Coder-3B	16	56/79	13/25	18/34	87/138	27
	Qwen2.5-Coder-3B (APRMCTS)	16	60/86	13/24	22/43	95/153	-
	Llama-3.2-3B	16	41/57	2/8	16/30	59/95	21
	Llama-3.2-3B (APRMCTS)	16	15/34	2/9	9/17	26/60	-
7-9B	Phi-3.5-mini	16	33/52	9/17	15/36	57/105	19
	Phi-3.5-mini (APRMCTS)	16	34/55	11/20	13/35	58/110	-
	DeciLM-7B	16	31/51	2/9	12/23	45/83	19
	DeciLM-7B (APRMCTS)	16	32/57	3/12	13/25	48/94	-
	Falcon-7B	16	13/34	5/10	0/2	18/46	4
	Falcon-7B (APRMCTS)	16	7/22	3/8	0/2	10/32	-
	Deepseek-Coder-6.7B	16	59/80	8/18	22/43	89/141	27
	Deepseek-Coder-6.7B (APRMCTS)	16	54/76	11/21	23/47	88/144	-
	Yi-Coder-9B	16	60/77	16/30	30/59	106/166	31
	Yi-Coder-9B (APRMCTS)	16	73/90	26/37	44/63	143/190	-
	Llama-3.1-8B	16	48/63	12/21	26/55	86/139	25
	Llama-3.1-8B (APRMCTS)	16	54/75	14/26	27/61	95/162	-
	Qwen2.5-Coder-7B	16	46/62	11/18	22/52	79/132	25
	Qwen2.5-Coder-7B (APRMCTS)	16	61/78	16/34	30/59	107/171	-
API	GPT-4o-mini	16	65/72	27/37	36/61	128/170	35
	GPT-4o-mini (APRMCTS)	16	78/92	32/45	48/71	158/208	40
	GPT-3.5	16	67/73	29/38	36/65	132/176	36
	GPT-3.5 (APRMCTS)	16	84/96	31/46	44/74	159/216	40
	GPT-3.5 (APRMCTS)	32	104/121	42/64	55/95	201/280	40

TABLE II: Implementation of extracting repo-level context.

Scope	Type	Example
In-class	Field	final string a;
	Method	public void check(){...}
	Invocation	boolean isValid = check();
Cross-class	Dependency	AnotherClass field;
	Extend	public class A extends AnotherClass{...}
	Invocation	AnotherClass.method();
	Util	from com.example.util import NodeUtil;