
TRAVERSAL COMPLETE LOGIC

DESIGN OF A COMPREHENSIVE SYSTEM FOR MULTI-SKILL INTERVIEW ACCOMMODATING VARIOUS EXPERIENCE LEVELS.

1. Setup:

- Interview length: 30 minutes
- Skills: Python, Data Analysis, Machine Learning, Deep Learning, NLP, Cloud
- Experience range: 0-15 years
- Question time: Vary between 60-120 seconds based on difficulty

2. Initial Assessment:

- Start with a quick self-assessment questionnaire to gauge the candidate's experience in each skill area. This helps in choosing the starting point and overall difficulty.

3. Time Allocation:

- 2 minutes for initial assessment
- 26 minutes for main interview (allows for ~13-26 questions)
- 2 minutes for wrap-up and final assessment

4. Starting Point:

- Based on the self-assessment, start with the skill where the candidate claims the most experience, at an intermediate difficulty level.

5. Traversal Logic:

- a) Inter-skill Movement:
 - Move to a different skill every 3-4 questions or if the candidate shows mastery/struggle in the current skill
- b) Intra-skill Movement:
 - Use the previous tree traversal logic within each skill
- c) Difficulty Adjustment:
 - Adjust difficulty based on performance, but keep it within ± 2 of the initial assessed level

6. Question Selection Logic:

```
def select_next_question(candidate_profile, current_state, performance_history):
    if len(performance_history)%4==0:
        return select_new_skill(candidate_profile, performance_history)
    else:
        return select_within_skill(current_state, performance_history)

def select_new_skill(candidate_profile, performance_history):
    least_covered_skill = get_least_covered_skill(performance_history)
    starting_difficulty = calculate_starting_difficulty(candidate_profile,
least_covered_skill)
    return generate_question(least_covered_skill, starting_difficulty)

def select_within_skill(current_state, performance_history):
    if performance_history[-1].score > 4:
        return move_forward(current_state)
    elif performance_history[-1].score < 2:
        return move_backward(current_state)
    else:
        return move_lateral(current_state)

def calculate_starting_difficulty(candidate_profile, skill):
    claimed_experience = candidate_profile.experience[skill]
    return min(3 + claimed_experience // 2, 8) # Cap at 8
```

7. Backtracking:

Implement a stack to keep track of the path:

```
class InterviewState:
    def __init__(self):
        self.path = []
        self.current_skill = None
        self.current_node = None
    def move_to(self, skill, node):
        self.path.append((self.current_skill, self.current_node))
        self.current_skill = skill
```

```

        self.current_node = node
    def backtrack(self):
        if self.path:
            self.current_skill, self.current_node = self.path.pop()
        return self.current_skill, self.current_node

```

8. Main Interview Loop:

```

def conduct_interview(candidate_profile):
    state = initialize_state(candidate_profile)
    performance_history = []
    start_time = time.time()
    while time.time() - start_time < 1680: # 28 minutes
        question = select_next_question(candidate_profile, state,
performance_history)
        response, response_time = ask_question(question)
        score = evaluate_response(response, question.difficulty)

        performance_history.append(QuestionPerformance(question, score,
response_time))
        if score < 2 and len(state.path) > 0:
            state.backtrack()
        else:
            update_state(state, score, question)
    return generate_final_assessment(performance_history)

```

9. Skill Transition Logic:

```

def should_transition_skill(performance_history):
    recent_questions = performance_history[-3:]
    avg_score = sum(q.score for q in recent_questions) / len(recent_questions)
    return avg_score > 4.5 or avg_score < 1.5 or len(recent_questions) >= 4

def select_next_skill(candidate_profile, current_skill, performance_history):
    if should_transition_skill(performance_history):
        skills_score = calculate_skills_score(performance_history)
        next_skill = min(skills_score, key=skills_score.get)
        return next_skill
    return current_skill

```

10. Experience Level Adaptation:

- Adjust the difficulty and depth of questions based on experience:
- 0-2 years: Focus on fundamentals, max difficulty 6
- 3-5 years: Cover intermediate topics, max difficulty 8
- 6-10 years: Include advanced topics, max difficulty 9
- 11+ years: Focus on expert-level and theoretical questions, max difficulty 10

11. Time Management:

```
def get_question_time(difficulty):  
    return 60 + (difficulty - 1) * 10  # 60s for easiest, 150s for hardest  
  
def adjust_remaining_questions(remaining_time, avg_question_time):  
    return max(1, remaining_time // avg_question_time)
```

12. Final Assessment:

```
def generate_final_assessment(performance_history):  
    skill_scores = {}  
    for skill in SKILLS:  
        skill_questions = [q for q in performance_history if q.question.skill ==  
skill]  
        if skill_questions:  
            avg_score = sum(q.score for q in skill_questions) /  
len(skill_questions)  
            skill_scores[skill] = avg_score  
  
    strengths = [skill for skill, score in skill_scores.items() if score > 4]  
    weaknesses = [skill for skill, score in skill_scores.items() if score < 2]  
  
    return FinalAssessment(  
        overall_score=sum(skill_scores.values()) / len(skill_scores),  
        skill_breakdown=skill_scores,  
        strengths=strengths,  
        weaknesses=weaknesses  
    )
```

- This system provides a flexible, adaptive interview experience that:
- Covers multiple skills
- Adjusts to the candidate's experience level
- Balances depth and breadth of questioning
- Implements backtracking when needed
- Manages time effectively
- Provides a comprehensive final assessment
- The AI chatbot uses this logic to navigate through the various skill trees, adjusting difficulty and focus based on the candidate's performance and claimed experience level. It ensures a thorough evaluation across all relevant skills within the 30-minute timeframe.

Figure out which skill to go?

Here's a more detailed approach to deciding which skill to transition to next:

1. Skill Relationship Matrix:

- First, we establish a relationship matrix between skills. This helps in making more logical transitions:

```
SKILL_RELATIONSHIPS = {  
    'Python': ['Data Analysis', 'Machine Learning', 'Cloud'],  
    'Data Analysis': ['Python', 'Machine Learning', 'Deep Learning'],  
    'Machine Learning': ['Data Analysis', 'Deep Learning', 'NLP'],  
    'Deep Learning': ['Machine Learning', 'NLP'],  
    'NLP': ['Machine Learning', 'Deep Learning'],  
    'Cloud': ['Python', 'Machine Learning']  
}
```

2. Skill Coverage Tracker:

Keep track of how much each skill has been covered:

```
skill_coverage = {skill: 0 for skill in SKILLS}
```

3. Performance Tracker:

Track performance in each skill:

```
skill_performance = {skill: [] for skill in SKILLS}
```

4. Transition Decision Function:

```
def should_transition_skill(current_skill, performance_history, skill_coverage):
    recent_questions = [q for q in performance_history[-3:] if q.skill ==
current_skill]

    if len(recent_questions) < 3:
        return False # Not enough questions in current skill to judge

    avg_score = sum(q.score for q in recent_questions) / len(recent_questions)
    time_in_skill = sum(skill_coverage.values()) - skill_coverage[current_skill]
    return avg_score > 4.5 or avg_score < 1.5 or time_in_skill >= 4
```

5. Next Skill Selection:

```
def select_next_skill(current_skill, candidate_profile, performance_history,
skill_coverage):
    if not should_transition_skill(current_skill, performance_history,
skill_coverage):
        return current_skill

    related_skills = SKILL_RELATIONSHIPS[current_skill]

    # Calculate a score for each related skill
    skill_scores = {}
    for skill in related_skills:
        coverage_score = 1 / (skill_coverage[skill] + 1) # Favor less covered
skills
```

```

        performance_score = calculate_skill_performance(skill,
performance_history)
        relevance_score = calculate_skill_relevance(skill, candidate_profile)
        skill_scores[skill] = coverage_score * 0.4 + performance_score * 0.3 +
relevance_score * 0.3

    # Select the skill with the highest score
    next_skill = max(skill_scores, key=skill_scores.get)
    return next_skill

def calculate_skill_performance(skill, performance_history):
    skill_questions = [q for q in performance_history if q.skill == skill]
    if not skill_questions:
        return 0.5 # Neutral score if no questions asked yet
    return sum(q.score for q in skill_questions) / len(skill_questions) / 5 #
Normalize to 0-1

def calculate_skill_relevance(skill, candidate_profile):
    # Based on candidate's claimed experience and job role
    return min(1, candidate_profile.experience[skill] / 10) # Normalize to 0-1

```

6. Main Interview Loop Integration:

```

def conduct_interview(candidate_profile):
    current_skill = select_initial_skill(candidate_profile)
    state = initialize_state(candidate_profile, current_skill)
    performance_history = []
    skill_coverage = {}
    skill_performance = {}
    start_time = time.time()

    while time.time() - start_time < 1680: # 28 minutes
        question = select_next_question(current_skill, state,
performance_history)
        response, response_time = ask_question(question)
        score = evaluate_response(response, question.difficulty)

        performance_history.append(QuestionPerformance(question, score,
response_time))
        skill_coverage[current_skill] = skill_coverage.get(current_skill, 0) + 1
        skill_performance.setdefault(current_skill, []).append(score)

        if score < 2 and len(state.path) > 0:

```

```

        state.backtrack()
    else:
        update_state(state, score, question)

    current_skill = select_next_skill(current_skill, candidate_profile,
performance_history, skill_coverage)

    return generate_final_assessment(performance_history, skill_coverage,
skill_performance)

```

This approach ensures that:

1. Skills are transitioned logically based on their relationships.
 2. Less covered skills are favoured to ensure broad coverage.
 3. The candidate's performance in a skill influence whether to stay or move.
 4. The candidate's profile (experience, job role) influences skill selection.
 5. There's a balance between depth (staying in a skill) and breadth (moving between skills).
- For example, if a candidate is doing well in `data_analysis` and has covered it sufficiently, the system might choose to move to `machine_learning` next, based on their relationship to `data_analysis` and the candidate's profile. If the candidate struggles with `data_analysis`, it might move back to `data_analysis` or try `machine_learning`, depending on the overall coverage and performance.
 - This method provides a dynamic and adaptive way to navigate between different skills during the interview, ensuring a comprehensive assessment while still maintaining a logical flow.