Approach C (Modified): The LLM-Powered Vision Approach

Core Idea: Treat each page of the flowchart PDF as an image. Feed this image to a multimodal LLM and instruct it, via a detailed prompt, to return a structured JSON object representing all the elements and their relationships on that page.

Why this is a paradigm shift:

- **From Geometry to Semantics:** Traditional methods require you to write code that says, "Find a rectangle here, find an arrow whose tail is near the rectangle and whose head is near a diamond." The LLM approach is semantic: "Tell me what process this rectangle represents and which decision it connects to."
- **Robustness:** It's more resilient to variations in flowchart style, hand-drawn elements, or unusual connector paths that would break a rigid, code-based geometric analysis.
- **Reduced Complexity:** It outsources the most difficult computer vision tasks (shape detection, line tracing, text association) to the pre-trained model.

The Step-by-Step Extraction Workflow

Step 1: Preparation and Pre-processing

Your input is a multi-page PDF. LLMs that accept images typically work with standard image formats (PNG, JPG).

1. **Convert PDF to Images:** Convert each page of your flowchart PDF into a high-resolution PNG image. A resolution of 150 or 300 DPI is a good starting point.
    - **Tool:** Python's `pdf2image` library is perfect for this.

```
from pdf2image import convert_from_path

images = convert_from_path('my_flowchart.pdf', dpi=300)

for i, image in enumerate(images):
    image.save(f'page_{i+1}.png', 'PNG')
```

2. Set Up API Access: You will need an API key for a multimodal LLM provider.

- **OpenAI:** For `gpt-4-vision-preview` (or the latest model).
- **Google:** For `gemini-pro-vision`.

Step 2: The Core Extraction Loop (Per-Page Analysis)

You will loop through each generated image (`page_1.png`, `page_2.png`, etc.) and send it to the LLM with a carefully crafted prompt.

The goal is to get a structured JSON output for *each page*.

Step 3: Designing the "Master Prompt"

This is the most critical part. The quality of your extraction depends almost entirely on the quality of your prompt. You need to tell the model its role, what you want it to do, and *exactly* what format you expect the output in.

Here is a powerful, detailed prompt template:

Instructions:

1. **Identify all Nodes:** A "node" is any shape that contains text (e.g., ovals for start/end, rectangles for processes, diamonds for decisions).
2. **Assign Unique IDs:** Assign a unique integer ID to each node you find on the page, starting from 1.
3. **Extract Node Details:** For each node, capture its ID, shape, and the exact text inside it.
4. **Identify all Edges:** An "edge" is an arrow or line connecting two nodes.
5. **Map Edges to Nodes:** Define each edge by the ID of its source node and its target node.
6. **Capture Decision Logic:** For edges coming out of a decision diamond, capture the condition text (e.g., "Yes", "No", "True", "False") associated with that edge.
7. **Identify Page Connectors:** Pay special attention to standalone connector symbols (often circles or special shapes) that indicate the flow continues on another page. List these separately.

Output Format:

Your output MUST be a single, valid JSON object. Do not include any other text or explanations outside of the JSON block. The JSON object should have two main keys: "nodes" and "edges".

```
{
  "nodes": [
    {
      "id": <integer>,
      "shape": "<Oval | Rectangle | Diamond | Parallelogram | Other>",
      "text": "<The full text inside the shape>",
      "is_page_connector": false,
      "connector_label": null
    }
```

```
 ],
 "edges": [
  {
   "source_id": <integer>,
   "target_id": <integer>,
   "label": "<'Yes', 'No', or any other text on the arrow, otherwise null>"
  }
 ],
 "off_page_connectors": [
  {
    "id": <integer>, // The ID of the node this connector is attached to
    "direction": "<Out | In>", // Is this an outgoing or incoming connector?
    "label": "<The label of the connector, e.g., 'A', 'B-1'>"
  }
 ]
}
```

Example of a simple node:
{ "id": 1, "shape": "Oval", "text": "Start Process", "is_page_connector": false, "connector_label": null }

Example of an off-page connector node:
{ "id": 5, "shape": "Circle", "text": "Go to A", "is_page_connector": true, "connector_label": "A" }

Now, analyze the following image and provide the JSON output.

Step 4: Execute the Extraction and Store Results

You will write a script that:

1. Iterates through your `page_N.png` files.
2. Encodes each image into a format the API accepts (usually base64).
3. Sends the master prompt along with the encoded image to the LLM API.
4. Receives the response.
5. Parses the JSON string from the response and saves it (e.g., `page_1_data.json`, `page_2_data.json`).

Here's a conceptual Python snippet using the OpenAI API:

```
import base64
import requests
import json
import os
```

```
# Your API key
api_key = "YOUR_OPENAI_API_KEY"

def encode_image(image_path):
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode('utf-8')

def analyze_flowchart_page(image_path, prompt):
    base64_image = encode_image(image_path)

    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {api_key}"
    }

    payload = {
        "model": "gpt-4-vision-preview",
        "messages": [
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": prompt},
                    {
                        "type": "image_url",
                        "image_url": {
                            "url": f"{base64_image}"
                        }
                    }
                ]
            }
        ],
        "max_tokens": 4096
    }

    response = requests.post("https://api.openai.com/v1/chat/completions", headers=headers,
json=payload)

    # Extract the JSON content from the response
    response_json = response.json()
    content = response_json['choices'][0]['message']['content']

    # Clean up the response to get only the JSON part
    json_str = content.strip().lstrip('```json').rstrip('```')
    return json.loads(json_str)
```

```
# --- Main Execution ---
master_prompt = "..." # Paste the detailed prompt from Step 3 here
flowchart_data = {}

for i in range(1, 4): # Assuming 3 pages
    image_file = f'page_{i}.png'
    if os.path.exists(image_file):
        print(f"Analyzing {image_file}...")
        try:
            page_data = analyze_flowchart_page(image_file, master_prompt)
            flowchart_data[f'page_{i}'] = page_data

            # Save individual page data
            with open(f'page_{i}_data.json', 'w') as f:
                json.dump(page_data, f, indent=2)

        except Exception as e:
            print(f"Error analyzing {image_file}: {e}")

# At this point, `flowchart_data` holds the structured data for all pages
print("All pages analyzed and data extracted.")
```

Step 5: Stitching the Pages Together (Post-Processing)

You now have a set of JSON files, one for each page. The final step is to combine them into a single, global process graph.

1. **Load all JSON data.**
2. **Create a Global Node Map:** Create a dictionary to map a global ID to each node. You can create global IDs like `p1_n1`, `p1_n2`, `p2_n1`, etc. (page 1-node 1, page 1-node 2, page 2-node 1).
3. **Link Connectors:** Iterate through the `off_page_connectors` from all pages.
    - Find an "Out" connector with `label: "A"`.
    - Find the corresponding "In" connector with `label: "A"` on another page.
    - Create a new edge in your global graph linking the source node of the "Out" connector to the target node of the "In" connector.

You can then use this final, unified graph data structure as the input for Phase 3 (Synthesis and Summary Generation).

Advantages and Challenges of this Approach

Advantages:

- **Massively Reduced Development Time:** You are writing prompts, not complex CV algorithms.
- **High Accuracy for Standard Flowcharts:** LLMs are excellent at OCR and understanding common diagrammatic conventions.
- **Flexibility:** It can handle imperfect diagrams, different drawing styles, and even some hand-drawn elements better than rigid algorithms.

Challenges and Considerations:

- **Cost:** API calls to powerful models are not free. Processing a 50-page document could become expensive.
- **Rate Limits:** You may be constrained by API rate limits.
- **Hallucinations/Inaccuracy:** The model can still make mistakes, misread text, or incorrectly identify a connection. The post-processing and final human review step remains crucial.
- **Non-Determinism:** Running the same image twice might produce slightly different (though usually functionally identical) results.
- **Prompt Sensitivity:** The results are highly dependent on your prompt. Minor changes can lead to different output formats.

This LLM-powered approach represents the state-of-the-art for this kind of unstructured data extraction and is a fantastic way to tackle your project.