Guía Técnica del Código (`optimizer.py`)

Este documento está dirigido a desarrolladores que necesiten mantener, modificar o extender el script `optimizer.py`.

1. Estructura General del Código

El script está organizado en regiones lógicas para facilitar su lectura y mantenimiento.

- **ESTRUCTURAS DE DATOS**: Se usan `dataclasses` de Python para modelar de forma tipada y predecible todos los objetos del `input.json`. Esto previene errores de tipeo y facilita el autocompletado en los IDEs.
- **CALLBACK PARA MÚLTIPLES SOLUCIONES**: La clase `SolutionCallback` hereda de la clase base `cp_model.CpSolverSolutionCallback`. Su propósito es ser invocada por el solver cada vez que encuentra una solución válida. La clase formatea la solución encontrada y la añade a una lista, deteniéndose cuando alcanza el `numberOfSolutions` solicitado.
- **LÓGICA DE CÁLCULO DE COSTOS**: Contiene funciones puras y modulares que replican la lógica de negocio.
    - `get_printing_needs` / `calculate_printing_cost`: Calculan el costo de impresión (setup, lavado, impresiones) basándose en las tintas, el tipo de impresión (duplex/simplex) y las reglas de la máquina definidas en el input.
    - `calculate_material_needs` / `calculate_total_layout_cost`: Orquestan el cálculo completo, sumando el costo de material (con merma y optimización de corte) y el de impresión, y devuelven un objeto con todo el desglose.
- **FASES DEL ALGORITMO**:
    - `calculate_base_solution`: Implementa el cálculo de la solución individual para cada trabajo. Itera sobre trabajos, máquinas y pliegos para encontrar el costo mínimo individual.
    - `generate_candidate_layouts`: Es la función más compleja. Realiza la poda de árbol descrita en la documentación de criterios para encontrar los "layouts campeones" de ganging.
    - `solve_optimal_plan`: Construye el modelo matemático para OR-Tools. Define las variables binarias, las restricciones de cantidad y la función de costo con penalizaciones. Implementa el bucle iterativo para encontrar las N-mejores soluciones.
- **PARSEO Y EJECUCIÓN**:

- `parse_input_data`: Convierte el diccionario JSON de entrada en la estructura de `dataclasses` anidada, asegurando la integridad de los datos.
- `main`: Es el orquestador principal. Llama a las fases en orden, procesa los resultados (filtra, ordena, limita) y finalmente construye y escribe el `output.json`.

2. Flujo de Ejecución Detallado

1. **Inicio (`main`)**: Se lee el `input.json` y se parsea a `dataclasses` con `parse_input_data`.
2. **Fase 1 (`calculate_base_solution`)**: Se calcula el costo de imprimir cada trabajo de forma individual. El resultado es una lista de "layouts base" y un costo total de referencia.
3. **Fase 2 (`generate_candidate_layouts`)**: Es el núcleo de la búsqueda de gangings.
   - Se crean combinaciones de trabajos (de 2 en 2, de 3 en 3...).
   - Para cada combinación y cada pliego posible, se generan todas las "recetas" de cantidades.
   - Se filtran por área y se ordenan por tiraje.
   - Se usa `rectpack.newPacker()` para validar geométricamente cada receta en orden. La primera que funciona se guarda como un "layout campeón".
4. **Fase 3 (`solve_optimal_plan`)**:
   - Se recopilan todos los layouts viables (base + campeones).
   - Se construye un modelo de `cp_model` donde cada layout es una variable **binaria** (`BoolVar`), representando si se usa o no.
   - Se añade la restricción de que la producción total de los layouts seleccionados debe cubrir la cantidad de cada trabajo.
   - La función objetivo es minimizar el costo total, que se pondera con las penalizaciones.
   - Se implementa un **bucle iterativo**:
     - Se llama a `solver.Solve()`.
     - Si se encuentra una solución, se guarda.
     - Se añade una nueva restricción al modelo: `total_cost_var > costo_encontrado`.
     - Se repite hasta alcanzar el `numberOfSolutions` deseado.
5. **Final (`main`)**:
   - Las soluciones encontradas por el solver se filtran (solo las que son mejores que la base).
   - Se ordenan por costo.

    - Se limita al `numberOfSolutions`.
    - Se formatea el `output.json` final y se escribe en disco.

3. Dependencias Externas

- **`ortools`**: Librería de Google para optimización y resolución de problemas de investigación de operaciones. Se usa su `CpSolver` para la Fase 3.
- **`rectpack`**: Librería para resolver el problema de empaquetado 2D. Se usa en la Fase 2 para la validación geométrica de los layouts.

4. Consideraciones de Mantenimiento

- **Lógica de Costos**: La mayoría de los cambios futuros probablemente ocurrirán en las funciones de cálculo de costos (ej. añadir costo de guillotinado). Estas funciones están aisladas y se pueden modificar sin afectar el resto del flujo.
- **Rendimiento de Fase 2**: Si la generación de candidatos es lenta, la función `generate_candidate_layouts` es el lugar a optimizar, por ejemplo, implementando una heurística más avanzada en lugar de la generación exhaustiva.
- **Modelo del Solver**: Si se necesita añadir restricciones más complejas (ej. tiempos de entrega), la modificación se centrará en la función `solve_optimal_plan`.
    -