graphrag 笔记

1. graphrag.index --init 的工作机制

```
python -m graphrag.index --init --root ./ragtest
```

1.1 _initialize_project_at

初始化目录，并创建文件，并添加内容

- settings.yaml： llm 和 embedding 模型的配置信息
- .env： 配置环境变量，api_key
- prompts/entity_extraction.txt
- summarize_descriptions.txt
- claim_extraction.txt
- community_report.txt

2. 构建索引

2.1 _create_default_config

- 指定配置文件。支持 yaml,yml, json，若没有指定，则使用 root/settings.xx 否则从环境变量中获取
- create_graphrag_config。将配置文件读取按指定格式返回 GraphRagConfig
- 创建 pipeline 参数。

```
    PipelineConfig(
            root_dir=settings.root_dir,
            input=_get_pipeline_input_config(settings),
            reporting=_get_reporting_config(settings),
            storage=_get_storage_config(settings),
            cache=_get_cache_config(settings),
            workflows=[
                *_document_workflows(settings, embedded_fields),
                *_text_unit_workflows(settings, covariates_enabled,
    embedded_fields),
                *_graph_workflows(settings, embedded_fields),
                *_community_workflows(settings, covariates_enabled,
    embedded_fields),
                *(_covariate_workflows(settings) if
    covariates_enabled else []),
            ],
        )
```

2.2 pipeline 的 workflow

在创建 pipeline 时，分为以下几个部分，每个部分有会创建一个或多个 workflow。

2.2.1 _document_workflows

- create_base_documents：
- create_final_documents：

2.2.2 _text_unit_workflows

- create_base_text_units
- join_text_units_to_entity_ids
- join_text_units_to_relationship_ids
- join_text_units_to_covariate_ids
- create_final_text_units

2.2.3 _graph_workflows

- create_base_extracted_entities
- create_summarized_entities
- create_base_entity_graph

- create_final_entities
- create_final_relationships
- create_final_nodes

2.2.4 _community_workflows

- create_final_communities
- create_final_community_reports

2.2.5 _covariate_workflows

- create_final_covariates

2.3 _run_workflow_async

- 创建存储，缓存，report 等
- 加载文件 txt or csv
- 创建工作流依赖，构建任务拓扑图
- 执行任务
- 更新工作流状态和保存任务执行结果

3.索引(搜索)

入口为 graphrag.query.__main__

- run_local_search
- run_global_search

3.1 run_local_search

- 加载索引文件。包括的文件 create_final_nodes，create_final_community_reports，create_final_text_units， create_final_relationships， create_final_entities，create_final_covariates，这些文件在构建索引时生成，比如官方文档的例子中在 ragtest/output/{timestamp}/artifacts
- 创建向量存储索引，description_embedding_store = __get_embedding_description_store()
- 实体加载及向量索引生成（实体 embedding）:read_indexer_entities(), store_entity_semantic_embeddings()
- 创建搜索引擎。search_engine = get_local_search_engine()
- 查询。search_engine.search(query=query)

3.1.1 创建搜索引擎

```
search_engine = get_local_search_engine(
    config,
    reports=read_indexer_reports(
        final_community_reports, final_nodes, community_level
    ),
    text_units=read_indexer_text_units(final_text_units),
    entities=entities,
    relationships=
    read_indexer_relationships(final_relationships),
    covariates={"claims": covariates},
    description_embedding_store=description_embedding_store,
    response_type=response_type,
)
```

- 创建 llm。llm = get_llm(config)
- 创建 embedding 对象。text_embedder = get_text_embedder(config)
- token_encoder。token_encoder = tiktoken.get_encoding(config.encoding_model)
- LocalSearch 创建。根据创建的 llm 等信息，生成本地搜索引擎

```
LocalSearch(
        llm=llm,
        context_builder=LocalSearchMixedContext(
            community_reports=reports,
            text_units=text_units,
            entities=entities,
            relationships=relationships,
            covariates=covariates,
            entity_text_embeddings=description_embedding_store,
            embedding_vectorstore_key=EntityVectorStoreKey.ID,  # if the vectorstore uses entity title as ids, set this to EntityVectorStoreKey.TITLE
            text_embedder=text_embedder,
            token_encoder=token_encoder,
        ),
        token_encoder=token_encoder,
        llm_params={
            "max_tokens": ls_config.llm_max_tokens,  # change this based on the token limit you have on your model (if you are using a model with 8k limit, a good setting could be 1000=1500)
            "temperature": ls_config.temperature,
            "top_p": ls_config.top_p,
            "n": ls_config.n,
        },
        context_builder_params={
            "text_unit_prop": ls_config.text_unit_prop,
            "community_prop": ls_config.community_prop,
            "conversation_history_max_turns": ls_config.conversation_history_max_turns,
            "conversation_history_user_turns_only": True,
            "top_k_mapped_entities": ls_config.top_k_entities,
            "top_k_relationships": ls_config.top_k_relationships,
            "include_entity_rank": True,
            "include_relationship_weight": True,
            "include_community_rank": False,
            "return_candidate_context": False,
            "embedding_vectorstore_key": EntityVectorStoreKey.ID,  # set this to EntityVectorStoreKey.TITLE if the vectorstore uses entity title as ids
            "max_tokens": ls_config.max_tokens,  # change this based on the token limit you have on your model (if you are using a model with 8k limit, a good setting could be 5000)
        },
```

3.1.2 查询

构建上下文

```
    context_text, context_records =
    self.context_builder.build_context(
                query=query,
                conversation_history=conversation_history,
                **kwargs,
                **self.context_builder_params,
            )
```

- 根据 query 检索相关实体。将 query 向量化，然后根据向量检索获取实体（实体前面步骤已经向量化存储了），selected_entities = map_query_to_entities()
- 构建社区上下文内容。根据相关实体关联到社区信息，然后根据社区相关性排序取 top，接着根据社区报告获取文本快，并更新社区报告的属性权重信息。并根据 token 长度限制进行裁减。community_context, community_context_data = self._build_community_context()
- 构建局部实体上下文和实体关系上下文。这部分是跟全局搜索不一样的点，local_context, local_context_data = self._build_local_context()
- 构建实体关联的原文本块。根据实体找到关联的原文本块，text_unit_context, text_unit_context_data = self._build_text_unit_context()

prompt 生成

将构建上下文数据转成文本，然后填充 prompt 模板

```
    search_prompt = self.system_prompt.format(
                    context_data=context_text,
                    response_type=
                    self.response_type
                )
```

大模型调用

```
    search_messages = [
        {"role": "system", "content": search_prompt},
        {"role": "user", "content": query},
    ]
    response = self.llm.generate(
        messages=search_messages,
        streaming=True,
        callbacks=self.callbacks,
        **self.llm_params,
    )
```

3.2 run_global_search

- 加载索引文件。包括的文件 create_final_nodes，create_final_community_reports，create_final_text_units， create_final_relationships， create_final_entities，create_final_covariates
- 读取社区报告。reports = read_indexer_reports(final_community_reports, final_nodes, community_level)
- 实体加载:read_indexer_entities()，注意是所有的实体，不是检索的 topN 实体。
- 创建搜索引擎。earch_engine = get_global_search_engine()
- 查询。result = search_engine.search(query=query)

3.2.1 创建搜索引擎

```
    search_engine = get_global_search_engine(
            config,
            reports=reports,
            entities=entities,
            response_type=response_type,
        )
```

- 创建 llm。llm = get_llm(config)
- token_encoder。token_encoder = tiktoken.get_encoding(config.encoding_model)
- GlobalSearch 创建。根据创建的 llm 等信息，生成本地搜索引擎

3.2.2 全局搜索

1）异步执行 `asyncio.run(self.asearch(query, conversation_history))`

2）构建上下文

```
    context_text, context_records =
    self.context_builder.build_context(
                query=query,
                conversation_history=conversation_history,
                **kwargs,
                **self.context_builder_params,
            )
```

- 构建社区上下文内容。根据相关实体关联到社区信息，然后根据社区相关性排序取 top，接着根据社区报告获取文本快，并更新社区报告的属性权重信息。并根据 token 长度限制进行裁减。community_context, community_context_data = self.build_community_context()

3）map 执行(对每个社区获取与 query 相关的信息)

针对社区上下文的多个块，分别执行 map 操作，

```
map_responses = await asyncio.gather(*[
    self._map_response_single_batch(
        context_data=data, query=query, **self.map_llm_params
    )
    for data in context_chunks
])
```

- 使用社区上下文信息及 query 填充 map_system_promp 模板
- 调用大模型
- 解析大模型结果

```
search_prompt =
self.map_system_prompt.format(context_data=context_data)
search_messages = [
    {"role": "system", "content": search_prompt},
    {"role": "user", "content": query},
]
search_response = await self.llm.agenerate(
                    messages=search_messages, streaming=False,
**llm_kwargs
)
def parse_search_response(self, search_response: str) ->
list[dict[str, Any]]:
    parsed_elements = json.loads(search_response)["points"]
    return [
        {
            "answer": element["description"],
            "score": int(element["score"]),
        }
        for element in parsed_elements
    ]
```

4）reduce 执行

- map 结果初步处理。获取每个 map 的结果，不满足条件的过滤掉，比如格式不对或字段不全。
- 过滤不相关结果。将 score 分=0 的过滤掉（完全不相关）
- 按相关性得分逆序排 map 的结果
- 根据 token 限制，取 top 的 map 结果填充 reduce_system_prompt 模板。

```
    key_points = []
    for index, response in enumerate(map_responses):
        if not isinstance(response.response, list):
            continue
        for element in response.response:
            if not isinstance(element, dict):
                continue
            if "answer" not in element or "score" not in element:
                continue
            key_points.append({
                "analyst": index,
                "answer": element["answer"],
                "score": element["score"],
            })

    # filter response with score = 0 and rank responses by descending
    order of score
    filtered_key_points = [
        point
        for point in key_points
        if point["score"] > 0  # type: ignore
    ]

    if len(filtered_key_points) == 0 and not
    self.allow_general_knowledge:
        # return no data answer if no key points are found
        return SearchResult(
            response=NO_DATA_ANSWER,
            context_data="",
            context_text="",
            completion_time=time.time() - start_time,
            llm_calls=0,
            prompt_tokens=0,
        )

    filtered_key_points = sorted(
        filtered_key_points,
        key=lambda x: x["score"],  # type: ignore
        reverse=True,  # type: ignore
    )

    data = []
    total_tokens = 0
```

4. 大模型调用

4.1 构建索引的大模型调用

在构建索阶段，使用了 datashaper 缏排工作流，没有显示调用大模型，可先了解 datashaper 的工作机制。具体调用大模型可参考 query 阶段的逻辑。

4.2 query 阶段的大模型调用

在 query 阶段，调用大模型的文本模型或是 embedding 模型都是显示的。在 run_local_search 中，会对实体构建 embedding 向量，在获取到实体、社区等局部上下文后调用大模型，得到查询结果。调用大模型使用 chat 方式，用户输入为 query，其他信息为 system_prompt。

在 run_global_search 总中，没有 embedding 部分的调用，而是先根据所有实体、社区信息分块（map 阶段）构建 map_system_prompt，用户输入为 query，调用大模型获取到相关的若干块。

在 recude 阶段，先处理 map 阶段处理得到的 top 信息，然后填充 redue_system_prompt，用户输入为 query，调用大模型获取到最终的结果。

5. 接入国内大模型的修改

将 graphrag 中调用的大模型改成国内或者内部自己部署的大模型，需要解决两个问题。

- 模型的 api_key 如何配置。因为不同平台的大模型调用 api_key 是不一样的，比如百度的千帆是两个 key 搭配使用，而 openai 只需要配置一个 key 值。没法完全复用。
- 支持不同平台的模型，如果区分。不同的平台模型调用大模型是需要创建不同的实例。
- 实际调用大模型的时机确定。越早创建大模型，与 graphrag 的流程越吻合，但破坏性也越强。越晚加入第三方大模型，对 graphrag 的破坏性越弱，但代码结构需要单独维护。

具体代码的实现见 git 仓库。

参考文献

https://microsoft.github.io/graphrag/
https://github.com/guoyao/graphrag-more

https://m1n9x.vercel.app/2024/07/09/%E6%BA%90%E7%A0%81%E8%A7%A3%E8%AF%BB%20-%20%E5%BE%AE%E8%BD%AFGraphRAG%E6%A1%86%E6%9E%B6/