ChatGPT

QMTL DAG 구조를 유지하면서 알파 성능 측정 및 매매 신호 발생 기능 확장을 위한 구현 계획

1. 배경 및 문제점

QMTL은 전략을 **DAG(Directed Acyclic Graph)** 형태로 표현하여 재사용성과 중복 제거를 극대화하는 백테스팅・실시간 거래 프레임워크이다. 기존 구현은 데이터 입력과 알파 계산 노드를 제공하지만, **알파 성능을 평가하고 실제 매매 신호를 발생·전송하는 기능이 부족**하다. compute 함수는 순수 함수여야 하고 I/O를 수행하지 않아야 하며, 모든 업스트림에서 필요한 기간 동안 데이터가 채워져야만 실행된다 [1]. 또한 NodeID는 `(node_type, code_hash, config_hash, schema_hash)`를 SHA-256으로 해싱해 결정적으로 생성된다 [2]. 따라서 기능 확장을 위해선 이 구조를 존중해야 하며, **DAG 내에서 노드 실행의 결정성과 결과 재사용**을 유지해야 한다.

2. 요구 사항 정리

1. **알파 성능 측정 기능**
2. 알파를 산출하는 노드의 결과를 일정 기간 축적하고, 통계 지표를 계산하는 노드가 필요하다.
3. 평가 지표는 최대 낙폭, 승률, 샤프지수, 이익인손비(profit factor) 등 다양한 퍼포먼스 지표를 지원해야 한다 [3]. 특히 **샤프지수**는 기대 초과수익을 표준편차로 나눈 값으로, 위험 조정 후 성과를 비교할 때 중요하다 [4].
4. **매매 신호 발생 및 전송 기능**
5. 알파의 시그널을 실제 매매 지시로 변환하는 노드와, 이 지시를 외부 브로커나 주문 시스템으로 전송하는 노드가 필요하다.
6. compute 함수는 I/O를 수행할 수 없으므로, 신호 데이터를 큐(예: Kafka)로 전달하거나 Runner에서 후속 처리해야 한다 [1].
7. **DAG 개념 유지**
8. 새로운 기능도 다른 노드처럼 DAG 상에서 표현되며, NodeID 계산 방식 등 기존 규칙을 따라야 한다.
9. 순수 함수 기반의 계산과 외부 시스템과의 인터페이스 분리를 유지해 재현 가능한 백테스트·실거래를 보장한다.
10. **확장성 및 모니터링**
11. 전략에 따라 다양한 지표를 선택할 수 있도록 설정 가능해야 한다.
12. 실행된 신호 및 성능 지표를 로그·모니터링할 수 있어야 한다.

3. 설계 제안

3.1 DAG 노드 확장

3.1.1 `AlphaHistoryNode` 재사용 및 개선

이미 존재하는 `alpha_history_node`는 알파 값을 sliding window로 모아 다운스트림에서 사용하도록 반환한다 [5]. 이 노드를 기반으로 알파 결과를 축적하고 성능 평가 노드로 전달한다. 필요에 따라 윈도우 크기와 저장 내용(알파 값, 수익률 등)을 설정 가능하도록 파라미터를 확장한다.

3.1.2 `AlphaPerformanceNode` (신규)

- **목적**: `AlphaHistoryNode` 또는 기타 알파 계산 노드의 출력 데이터를 사용하여 성과 지표를 계산하는 노드.
- **입력**: `CacheView`를 통해 과거 알파 시리즈(`alpha_history`)를 받아온다.
- **출력**: 지정된 성과 지표를 포함하는 딕셔너리(예: `{'sharpe': 0.8,'max_drawdown':-0.1,'profit_factor':1.5}` 등).
- **구현 방법**:
- compute 함수는 순수 함수로 작성하여 단순히 통계 계산만 수행한다.
- **샤프지수** 계산 시 평균 초과수익을 표준편차로 나누는 방식이며, 거래 비용을 반영할 수 있도록 옵션 파라미터를 지원한다 [4].
- 최대 낙폭(max drawdown), 승률(win ratio), 이익/손실 비율(profit factor), CAR/MDD(연복리 수익률 대비 최대 낙폭), RAR/MDD 등 필요한 지표를 선택적으로 계산한다 [3].
- NodeID는 `node_type='alpha_performance'`와 코드·설정 해시를 사용해 생성한다.

3.1.3 `TradeSignalGeneratorNode` (신규)

- **목적**: 알파나 기타 신호를 기반으로 실제 매수/매도/중립 포지션을 결정하는 노드.
- **입력**: 알파 값 시리즈와 리스크 관리 파라미터(예: 진입/청산 기준치, 포지션 한도).
- **출력**: 매매 신호 객체(`{'action':'BUY','size':0.5,'timestamp':t}` 등) 리스트.
- **구현 방법**:
- compute 함수는 알파 시리즈를 분석하여 진입·청산 시점을 결정한다. 예: 알파가 임계값 이상이면 매수, 이하이면 매도. 기존 예제 `general_strategy.py`의 모멘텀 기반 신호 생성 방식을 참고한다 [6].
- 손절/이익실현, 포지션 크기 결정 논리를 포함할 수 있도록 설정 항목을 추가한다.
- 신호는 순수 데이터 객체로 반환되며, 노드 내부에서는 외부 주문 API 호출을 하지 않는다.

3.1.4 `TradeOrderPublisherNode` (신규, optional)

- **목적**: `TradeSignalGeneratorNode`의 출력 신호를 Kafka 큐 등에 게시하여 실제 주문 처리 서비스가 구독하도록 하는 노드.
- **특징**: compute 함수는 단순히 신호 데이터를 반환하고, 파이프라인에서 후속 처리로 Kafka producer가 메시지를 게시한다. 기존 `Pipeline`의 Kafka 연동 기능을 활용한다 [7].

3.2 Runner 및 Pipeline 수정

1. **알파 성과 지표 수집**: Runner는 `AlphaPerformanceNode`의 출력 결과를 로그 또는 파일로 저장할 수 있도록 후처리를 지원한다. compute 함수는 I/O를 하지 않으므로, Runner에서 결과를 Neo4j 또는 모니터링 시스템으로 전송하는 별도 콜백을 추가한다.

2. **실제 주문 인터페이스**: 실거래 모드(`mode=live`)에서 Runner는 `TradeOrderPublisherNode`의 결과를 읽어 외부 브로커 API로 주문을 실행할 수 있는 서비스(예: `TradeExecutionService`)에 전달한다. 이 서비스는 QMTL 외부 모듈로 구현하며, 주문 처리 실패 시 재시도, 슬리피지 적용 등 리스크 관리 로직을 포함한다.

3. **설정 파일 확장**: 전략 설정 YAML/JSON에 새 노드 구성(`performance_metrics`, `signal_thresholds`, `risk_limits`)을 정의할 수 있도록 스키마를 확장한다.

3.3 외부 모듈 연동

- **Broker/Order API**: 실거래를 위해 별도의 주문 실행 모듈을 개발한다. Runner는 메시지 큐를 통해 신호를 전달하고, 이 모듈은 인증/주문/상태 확인 등의 작업을 담당한다.
- **모니터링**: 성과 지표와 신호 데이터를 Grafana나 Prometheus와 같은 모니터링 도구에 노출할 수 있도록 exporter를 추가한다.

3.4 테스트 및 검증

1. **단위 테스트**: 각 노드의 compute 함수가 기대한 값을 출력하는지 검증한다. 랜덤 데이터에 대해 샤프지수, 승률 계산 결과를 확인하고 경계 상황(데이터 없음, NaN 등)을 처리한다.
2. **통합 테스트**: DAG를 구성하여 가상의 알파를 입력한 뒤 `AlphaPerformanceNode`와 `TradeSignalGeneratorNode`의 동작이 연속적으로 이루어지는지 검증한다.
3. **백테스트 검증**: 기존 Runner의 `mode=backtest`를 사용해 성능 지표와 매매 신호를 계산하고, 실제 시장 데이터로 백테스트한 결과가 합리적인지 확인한다.
4. **실거래 Dryrun**: 모의 거래 환경에서 주문 API 호출을 모의 객체로 대체하여 전체 파이프라인을 검증한다.

4. 구현 일정 및 우선순위

단계	주요 작업	설명	예상 기간
1	요구사항 세분화 및 설계 승인	위 설계를 팀과 공유하고 피드백 반영	1주
2	`AlphaPerformanceNode` 구현	샤프지수 등 지표 계산 모듈 구현 및 테스트	2주
3	`TradeSignalGeneratorNode` 구현	알파 시그널을 포지션 신호로 변환하는 로직 개발	2주
4	Runner/Pipeline 수정	성능 지표 후처리 및 신호 전송 기능 추가	1주
5	외부 주문 모듈 개발	주문 API 연동 모듈 구현 및 보안 검증	3주
6	통합 테스트 및 문서화	테스트 자동화, 사용자 문서 작성, 예제 추가	2주

5. 결론

이 계획은 QMTL의 **결정론적 DAG 구조**를 유지하면서 알파 성능 측정과 실제 매매 신호 발생 기능을 확장하는 전략이다. compute 함수는 여전히 순수함을 유지하고, I/O는 Runner나 외부 모듈에서 처리하도록 분리했다 [1]. NodeID 생성 규칙과 캐시 모델을 그대로 사용함으로써 재현 가능성과 결과 공유를 보장한다. 전략 개발자는 성능 지표 계산과 매매 신호 로직을 모듈화된 노드로 쉽게 조합할 수 있게 되며, 실거래 환경에서도 동일한 DAG를 재사용할 수 있게 된다.

---

1 architecture.md
https://github.com/hyophyop/qmtl/blob/ee8b8d032f441fb6baf1b8a53f2ced2313b90151/architecture.md

2 dag-manager.md
https://github.com/hyophyop/qmtl/blob/ee8b8d032f441fb6baf1b8a53f2ced2313b90151/dag-manager.md

3 Trading Performance: Strategy Metrics, Risk-Adjusted Metrics, And Backtest - QuantifiedStrategies.com
https://www.quantifiedstrategies.com/trading-performance/

4 Sharpe Ratio for Algorithmic Trading Performance Measurement | QuantStart
https://www.quantstart.com/articles/Sharpe-Ratio-for-Algorithmic-Trading-Performance-Measurement/

5 alpha_history.py
https://github.com/hyophyop/qmtl/blob/ee8b8d032f441fb6baf1b8a53f2ced2313b90151/qmtl/transforms/alpha_history.py

6 general_strategy.py
https://github.com/hyophyop/qmtl/blob/ee8b8d032f441fb6baf1b8a53f2ced2313b90151/qmtl/examples/strategies/general_strategy.py

7 pipeline.py
https://github.com/hyophyop/qmtl/blob/ee8b8d032f441fb6baf1b8a53f2ced2313b90151/qmtl/pipeline/pipeline.py