Indexing Techniques for Vector Databases

(practice & HW)

실습 목표 및 과제 소개

- 실습 목표
  - IVF, HNSW 인덱스 동작 방식 이해
  - 인덱스 성능 평가 방식 이해

- 과제
  - IVF, HNSW + BQ 구현 (by python)

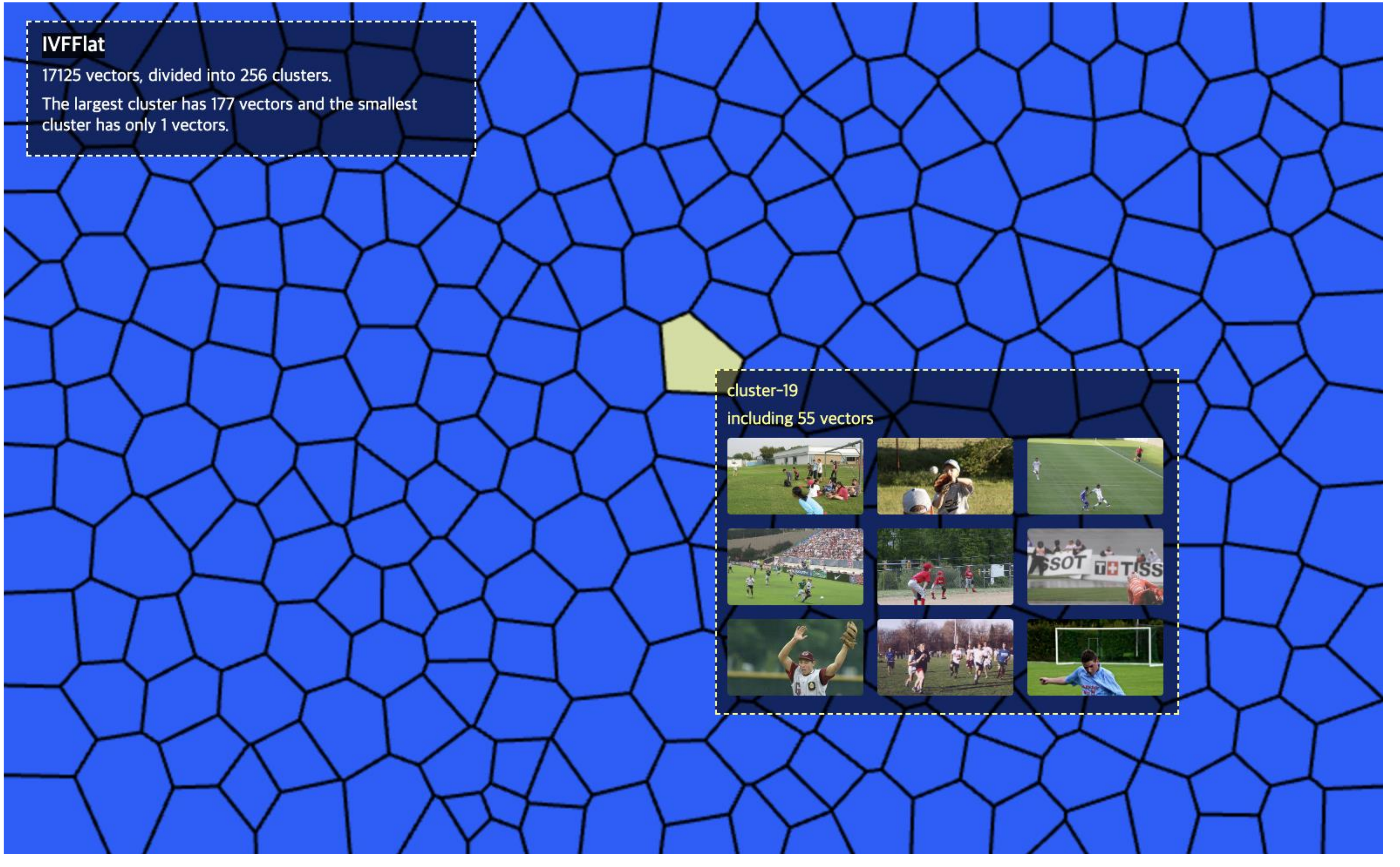
Vector Indexing

- Similarity search
  - 쿼리 벡터와 가장 유사한 벡터를 찾아내는 작업
  - 모든 벡터와 직접 비교할 경우 연산 비용이 크게 증가함

- ANN (Approximate Nearest Neighbor)
  - 검색 비용을 줄이기 위해 약간의 정확도를 희생하는 탐색 방법
  - EX.
    - IVF, HNSW: 인덱스를 활용해 비교 벡터 수를 줄임
    - Quantization: 벡터를 압축하여 연산 효율을 개선

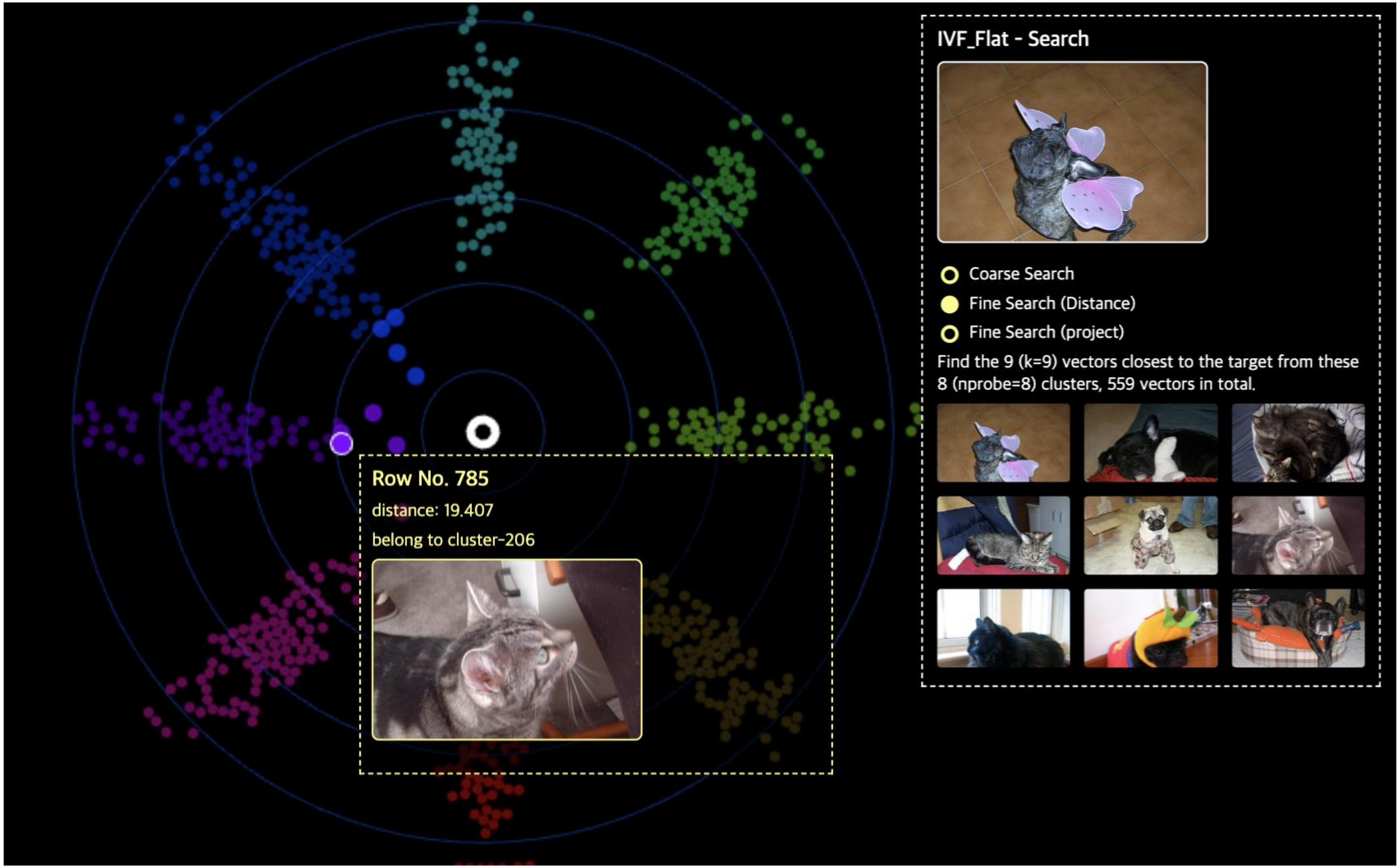
- 성능 평가 지표
  - Build 단계: build time, index size
  - Search 단계: search time, accuracy (ex. Recall)

IVF visualization (build)



- clustering 진행 (region 수를 결정)

IVF visualization (search)



- 탐색할 cluster 수 결정 (search time ↔ accuracy trade-off)

IVF python code

```
def __init__(self, distance_type, n_clusters):

    self.n_clusters = n_clusters
    self.distance_type = distance_type
    self.kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    self.inverted_index = {i: [] for i in range(n_clusters)}
    self.data = None
```

- n_cluster: cluster 개수
- distance_type: 사용할 distance measure
- kmeans: clustering 시에 사용할 모듈
- inverted_index: clustering 결과를 저장할 dictionary
- data: 원본 벡터 데이터

IVF python code (build)

```python
def fit(self, X):

    if self.distance_type == 'cosine':
        X = X / np.linalg.norm(X, axis=1, keepdims=True)

    self.kmeans.fit(X)
    labels = self.kmeans.labels_

    for idx, label in enumerate(labels):
        self.inverted_index[label].append(idx)

    self.data = X
```

- clustering 진행 후 inverted_index에 저장

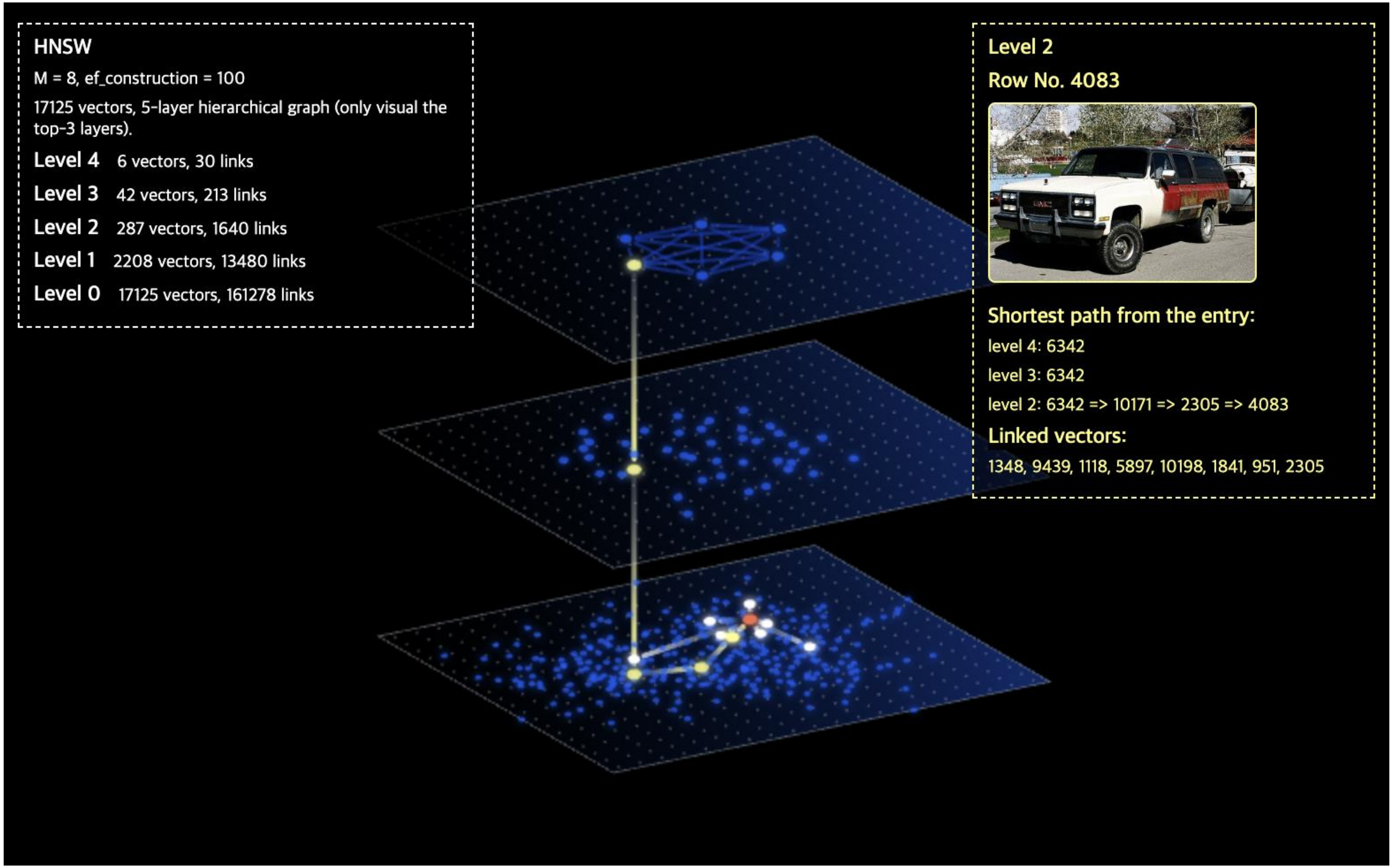
IVF python code (build)

```python
def search(self, q, k, n_probes=1):

    assert n_probes <= self.n_clusters

    if self.distance_type == "cosine":
        q = q / np.linalg.norm(q)

    cluster_distances = self._compute_distance(self.kmeans.cluster_centers_, q).flatten()
    probe_clusters = np.argpartition(cluster_distances, n_probes)[:n_probes]

    candidates = np.concatenate([self.inverted_index[cl] for cl in probe_clusters])

    if len(candidates) <= k:
        return candidates

    dists = self._compute_distance(self.data[candidates], q).flatten()
    top_idx = np.argpartition(dists, k)[:k]

    top_k = candidates[top_idx]
    top_dist = dists[top_idx]

    return [(k, dist) for k, dist in zip(top_k, top_dist)]
```

- cluster centroid와 query의 거리 계산
- n-probes개의 근접 cluster 선택
- 해당 cluster 내에 있는 vector 중 상위 k개 반환

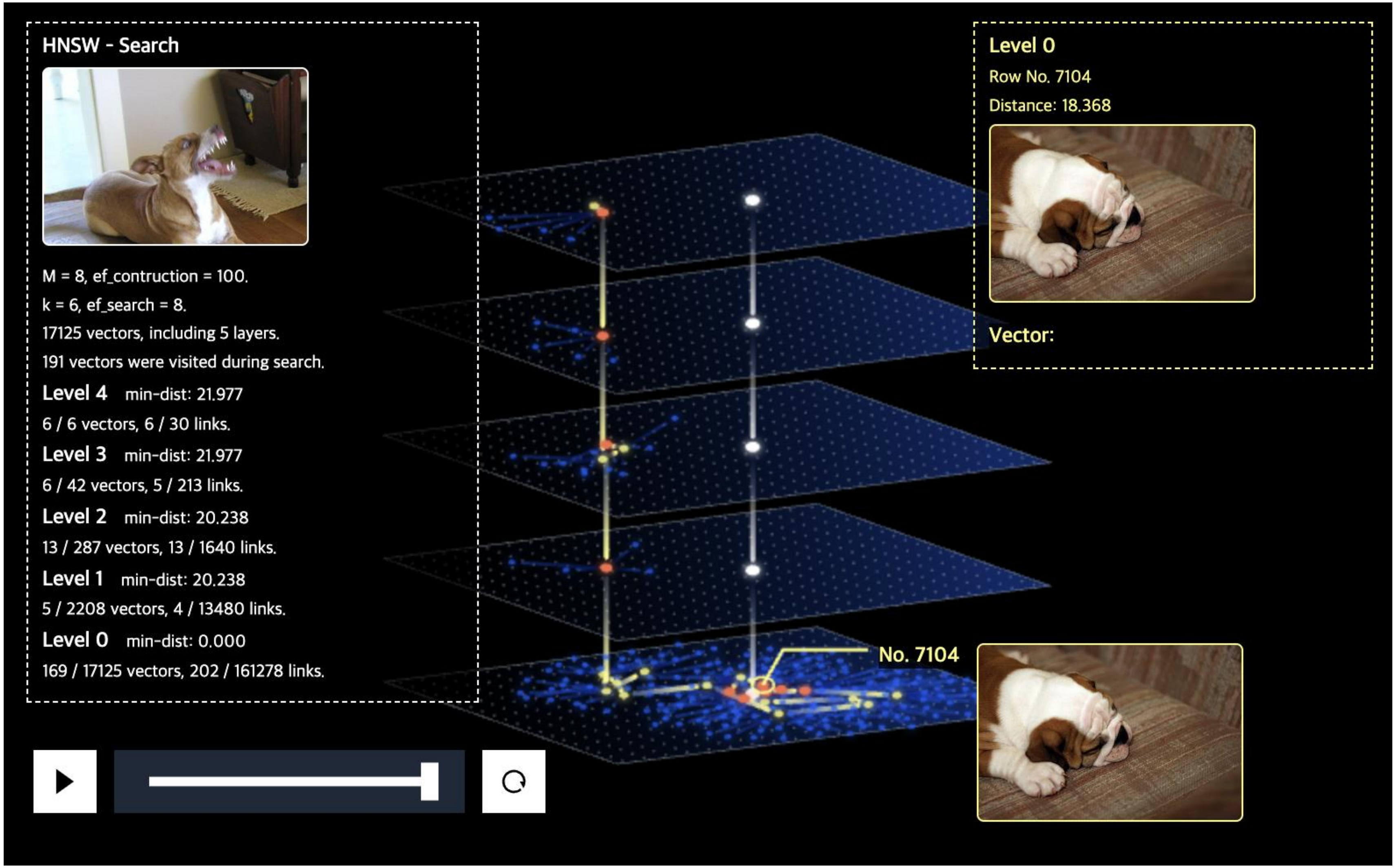
IVF 성능 평가 (practice.ipynb)

HNSW visualization (build)



- hierarchical graph 생성

HNSW visualization (search)



- Layer별로 search 진행

HNSW python code

```python
class HNSW(object):
    ### Algorithm 1: INSERT
    def insert(self, q, efConstruction=None): ...
    ### Algorithm 5: K-NN-SEARCH
    def search(self, q, K=5, efSearch=20): ...
    ### Algorithm 2: SEARCH-LAYER
    def _search_layer(self, q, W, lc, ef): ...
    ### Algorithm 3: SELECT-NEIGHBORS-SIMPLE
    def _select(self, R, C, M, lc, heap=False): ...
```

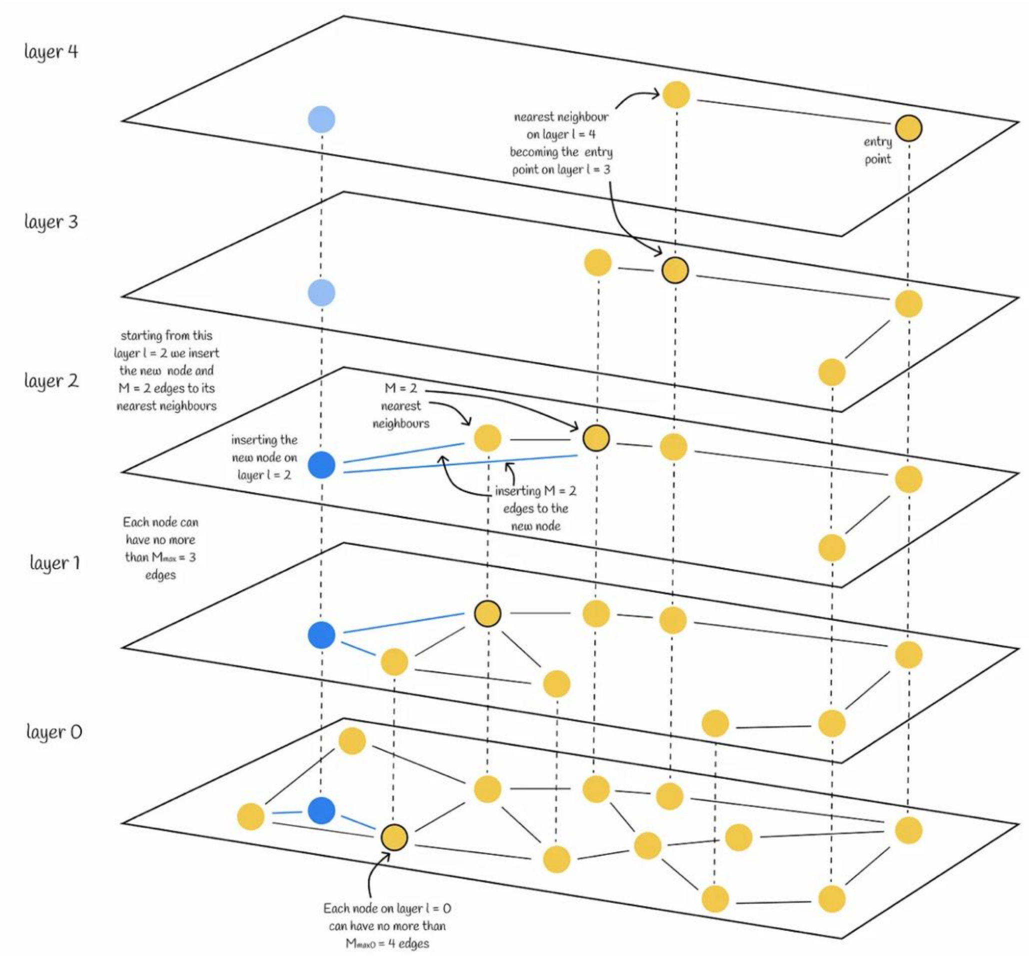
- Main method: insert, search
- Sub-algorithm:
  - _search_layer: 주어진 계층에서 노드 탐색
  - _select: 후보 노드 중 가장 가까운 이웃을 찾아 연결

HNSW python code

- Insert
- Search

_search_layer(1)

_search_layer(ef_construction)
_select



_search_layer(1)

_search_layer(ef_search)

https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

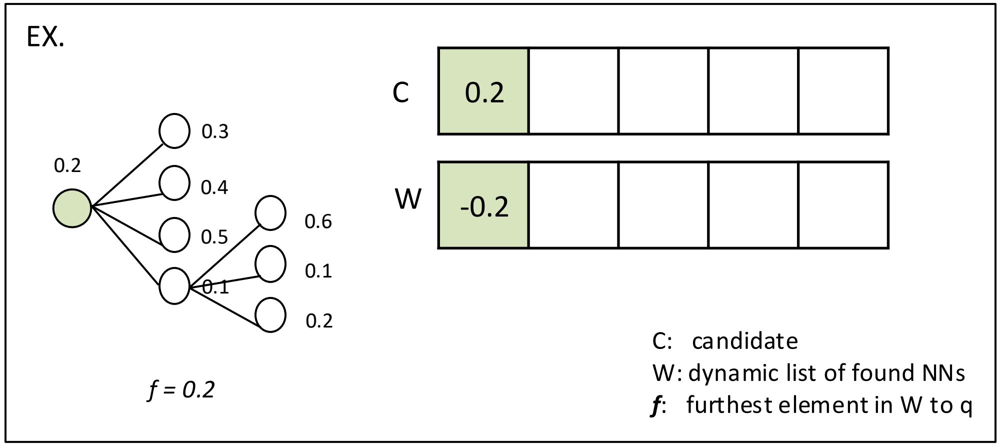
HNSW python code

```python
def __init__(self, distance_type, M=5, efConstruction=200, Mmax=None):
    if distance_type == "l2":…
    elif distance_type == "cosine":…
    else:…
    self.distance_func = distance_func
    self.vectorized_distance = self.vectorized_distance_
    self._M = M
    self._efConstruction = efConstruction
    self._Mmax = 2 * M if Mmax is None else Mmax
    self._level_mult = 1 / log2(M)
    self._graphs = []
    self._enter_point = None
    self.data = []
```

- M: 이웃 노드 수
- efConstruction: build 시 고려할 이웃 후보 수
  - cf. efSearch: 검색 시 유지할 후보 노드 수
- level_mult: 레벨 결정 시 사용되는 파라미터
  - level = (-ln(uniform(0, 1)) * level_mult)
- Graph, enter point, data: 벡터 데이터 및 그래프 저장

HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```

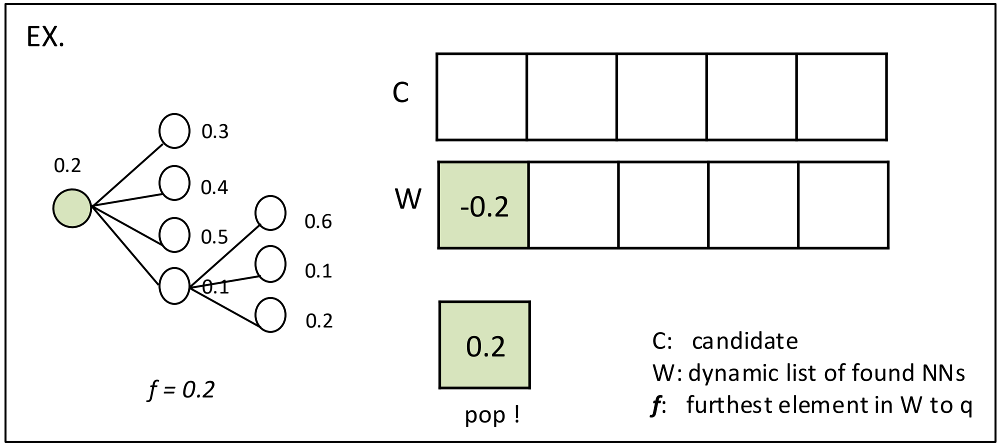


- 특정 layer에서 ef 개의 이웃 노드를 찾는 알고리즘
- C: 탐색 후보를 나타내는 min heap
- W: 검색 결과를 나타내는 max heap
  - 최대 길이 ef

- Python의 경우 min heap만 제공
- 따라서 W의 경우, distance를 음수로 저장

HNSW python code

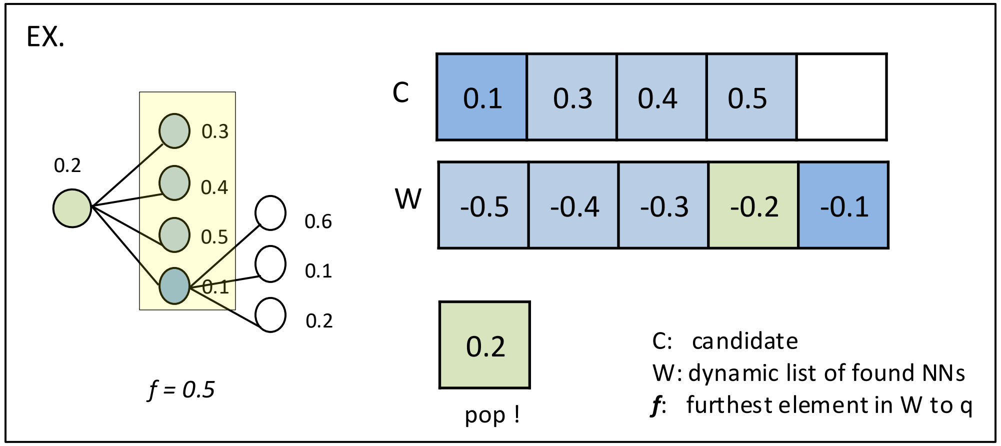
```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- C에서 pop
  - 가장 가까운 이웃 후보 노드를 가져옴
- W의 최댓값(f)과 비교
  - 현재 검색 결과 중 가장 먼 이웃과 비교
- C에서 pop한 노드가 더 먼 경우 종료
- 그렇지 않으면 이후 과정으로 이동

HNSW python code

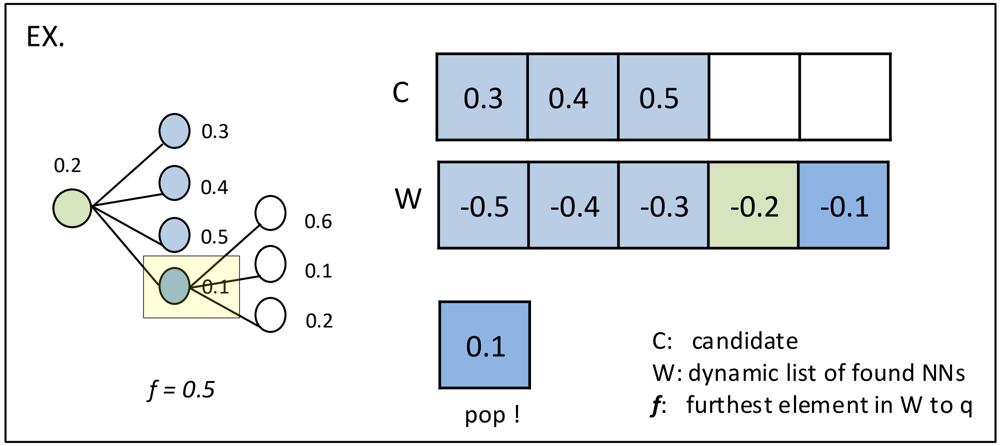
```
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- pop한 노드의 neighbor를 조건에 따라 C, W에 추가
  - W의 길이가 ef보다 작은 경우, 모든 노드를 추가

HNSW python code

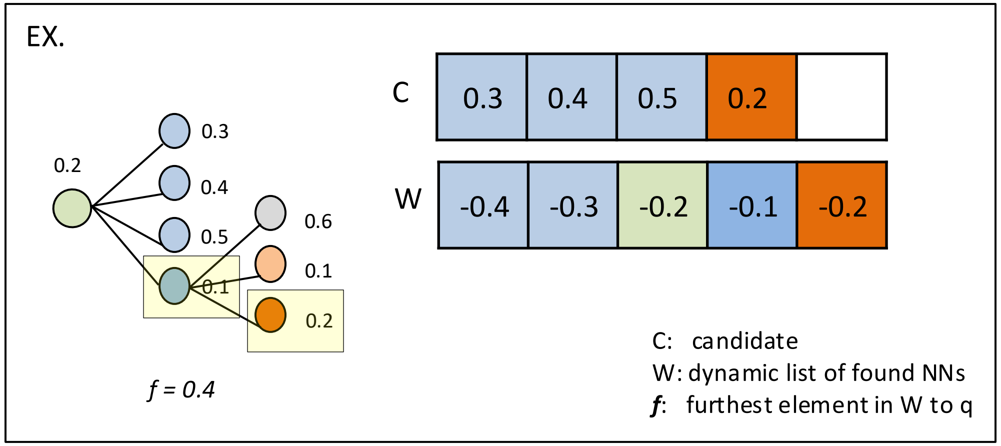
```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- C에서 pop
  - 가장 가까운 이웃 후보 노드를 가져옴
- W의 최댓값(f)과 비교
  - 현재 검색 결과 중 가장 먼 이웃과 비교
- C에서 pop한 노드가 더 먼 경우 종료
- 그렇지 않으면 이후 과정으로 이동

HNSW python code

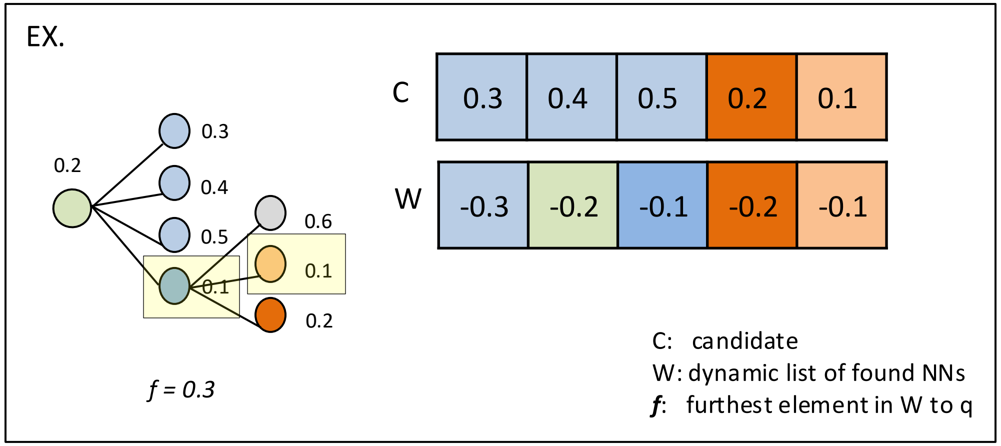
```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- pop한 노드의 neighbor를 조건에 따라 C, W에 추가
  - 이웃 노드와의 거리를 f와 비교
  - 작은 경우 C, W에 추가

HNSW python code

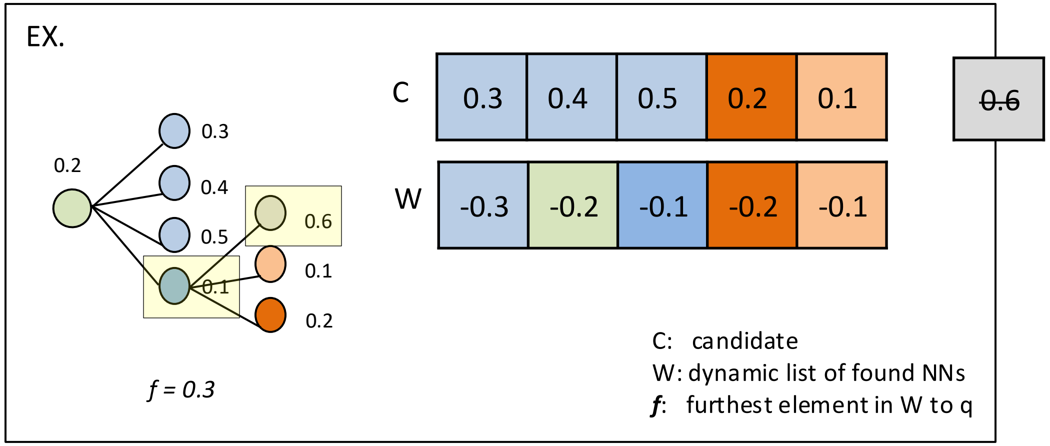
```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- pop한 노드의 neighbor를 조건에 따라 C, W에 추가
  - 이웃 노드와의 거리를 f와 비교
  - 작은 경우 C, W에 추가

HNSW python code

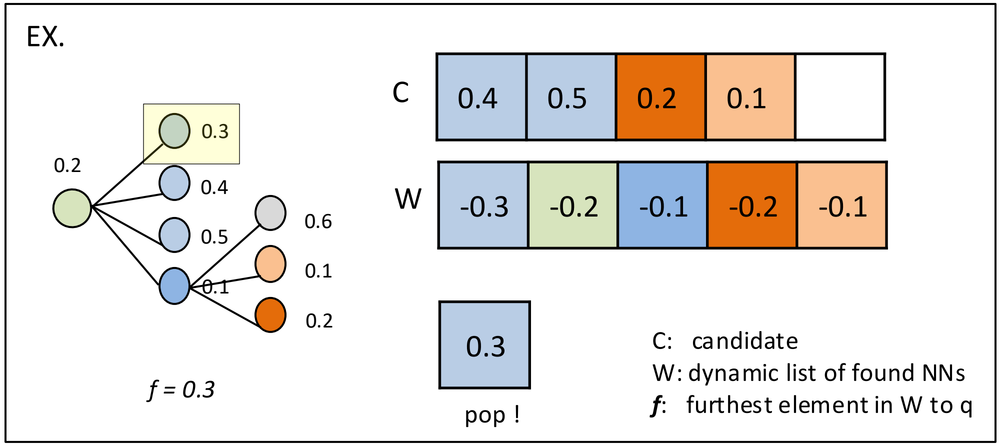
```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- pop한 노드의 neighbor를 조건에 따라 C, W에 추가
  - 이웃 노드와의 거리를 f와 비교
  - 큰 경우 pass

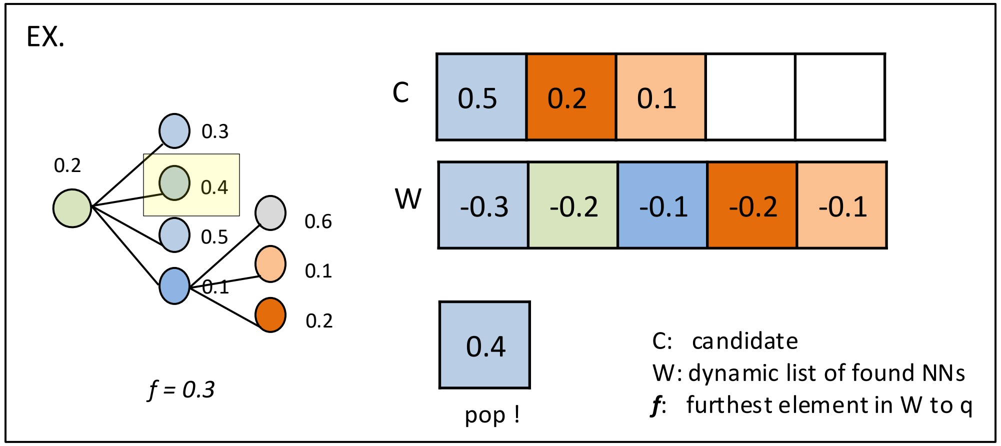
HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



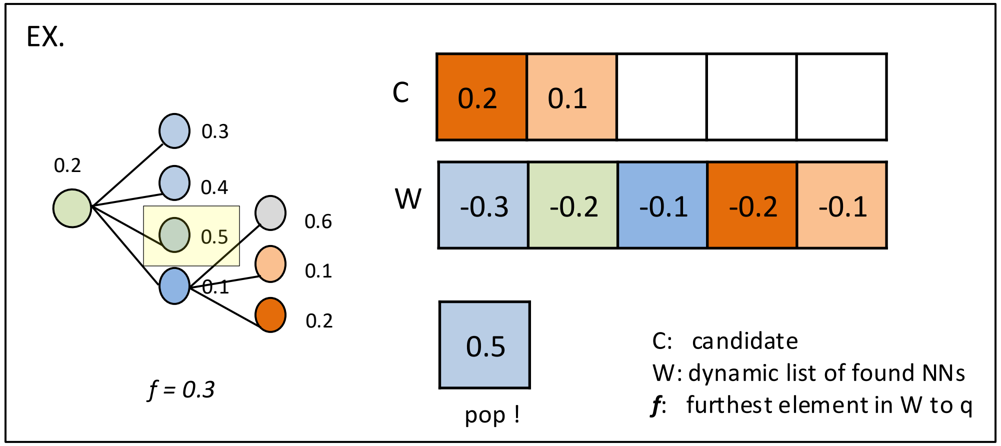
HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



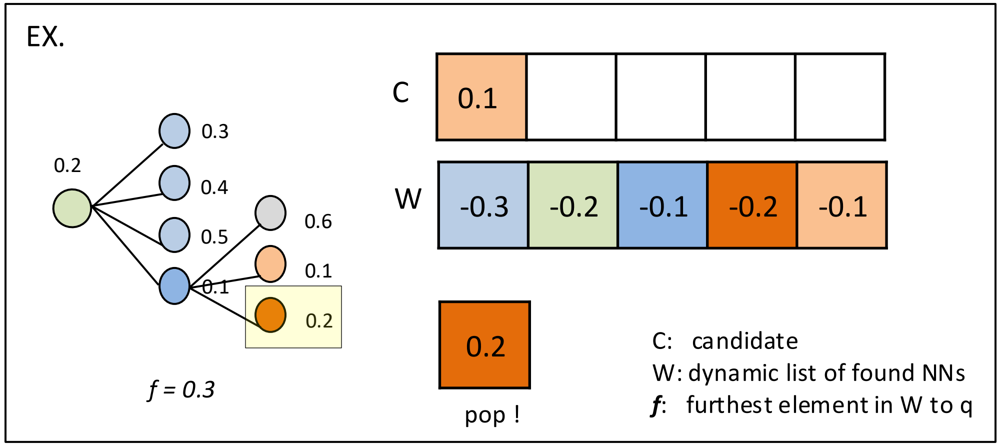
HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



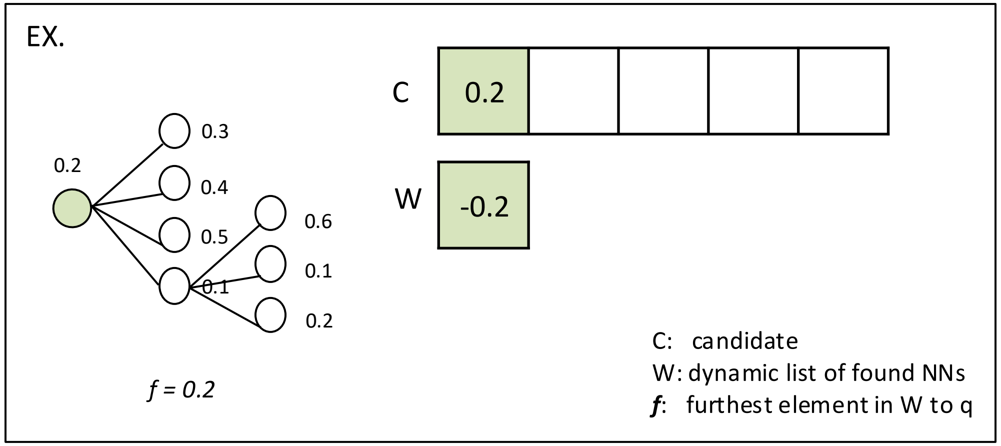
HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



HNSW python code

```python
### Algorithm 2: SEARCH-LAYER
def _search_layer(self, q, W, lc, ef):

    vectorized_distance = self.vectorized_distance
    data = self.data

    # Step 1: Initialize candidate list and visited set
    C = [(-neg_dist, idx) for neg_dist, idx in W]
    heapify(C)
    heapify(W)
    visited = set(idx for _, idx in W)

    # Step 4-17: Explore neighbors until candidate list is exhausted
    while C:
        dist, c = heappop(C)
        furthest = -W[0][0]
        if dist > furthest:
            break
        neighbors = [e for e in lc[c] if e not in visited]
        visited.update(neighbors)
        dists = vectorized_distance(q, [data[e] for e in neighbors])
        for e, dist in zip(neighbors, dists):
            neg_dist = -dist
            if len(W) < ef:
                heappush(C, (dist, e))
                heappush(W, (neg_dist, e))
                furthest = -W[0][0]
            elif dist < furthest:
                heappush(C, (dist, e))
                heapreplace(W, (neg_dist, e))
                furthest = -W[0][0]

    return W
```



- Parameter
  - ef (= W의 최대 길이)
  - 탐색 속도 및 정확도를 결정
  - Ex. (ef = 1)
    - 가까운 이웃 노드 하나만 빠르게 탐색
  - Ex. (ef = 10)
    - 가까운 이웃 10개를 유지하며 탐색

HNSW python code

```python
### Algorithm 3: SELECT-NEIGHBORS-SIMPLE
def _select(self, R, C, M, lc, heap=False):

    if not heap:
        idx, dist = C
        if len(R) < M:
            R[idx] = dist
        else:
            max_idx, max_dist = max(R.items(), key=itemgetter(1))
            if dist < max_dist:
                del R[max_idx]
                R[idx] = dist
        return

    else:
        C = nlargest(M, C)
        R.update({idx: -neg_dist for neg_dist, idx in C})
```

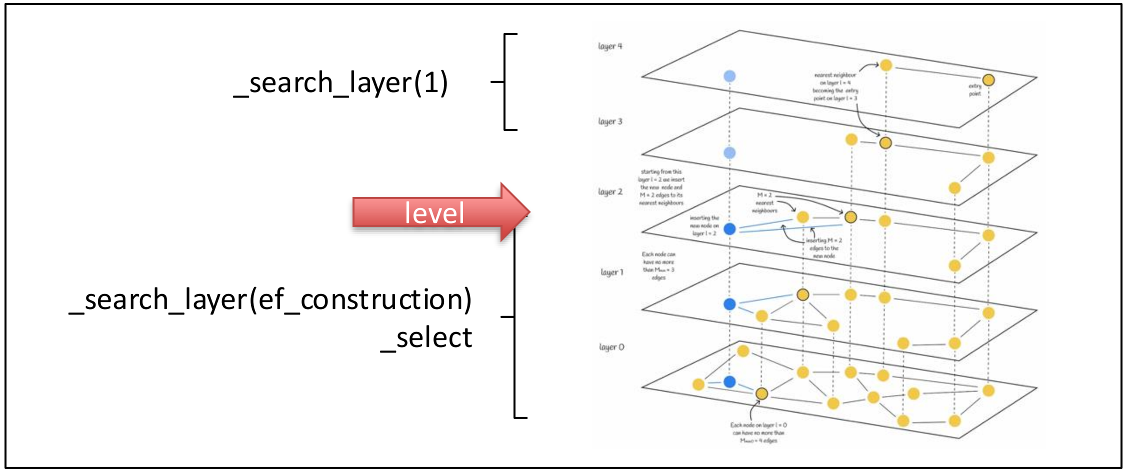
- Index build 시 그래프에 이웃 노드를 업데이트하는 알고리즘

- R: 기존 이웃 노드 목록
- C: 추가할 노드 목록
- M: 최대 이웃 개수

- => R+C 중에서 M개를 선택

HNSW python code

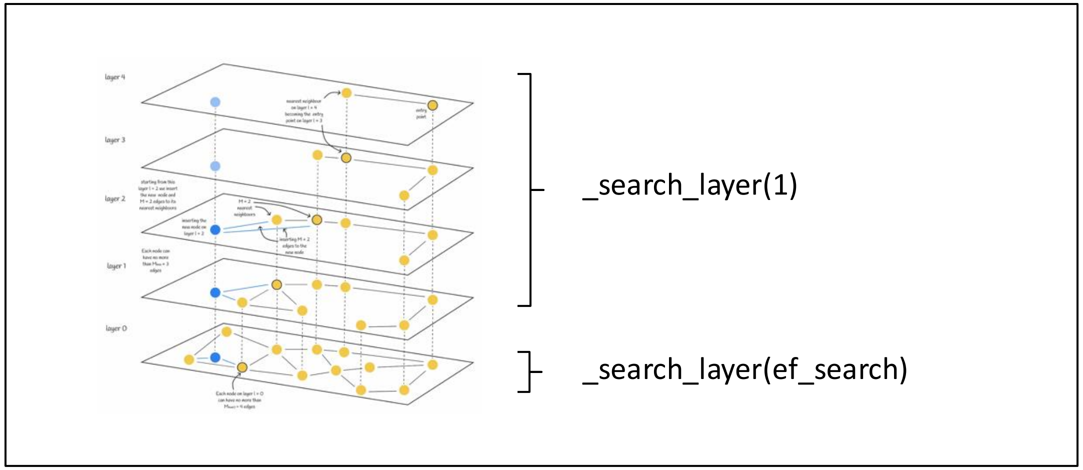
```python
### Algorithm 1: INSERT
def insert(self, q, efConstruction=None):

    if efConstruction is None:
        efConstruction = self._efConstruction

    distance = self.distance_func
    data = self.data
    graphs = self._graphs
    ep = self._enter_point
    M = self._M

    # line 4: determine level for the new element q
    l = int(-log2(random()) * self._level_mult) + 1
    idx = len(data)
    data.append(q)

    if ep is not None:
        neg_dist = -distance(q, data[ep])
        # distance(q, data[ep])

        # line 5-7: find the closest neighbor for levels above the insertion level
        for lc in reversed(graphs[l:]):
            neg_dist, ep = self._search_layer(q, [(neg_dist, ep)], lc, 1)[0]

        # line 8-17: insert q at the relevant levels; W is a candidate list
        layer0 = graphs[0]
        for lc in reversed(graphs[:l]):
            M_layer = M if lc is not layer0 else self._Mmax

            # line 9: update W with the closest nodes found in the graph
            W = self._search_layer(q, [(neg_dist, ep)], lc, efConstruction)

            # line 10: insert the best neighbors for q at this layer
            lc[idx] = layer_idx = {}
            self._select(layer_idx, W, M_layer, lc, heap=True)

            # line 11-13: insert bidirectional links to the new node
            for j, dist in layer_idx.items():
                self._select(lc[j], (idx, dist), M_layer, lc)

    # line 18: create empty graphs for all new levels
    for _ in range(len(graphs), l):
        graphs.append({idx: {}})
        self._enter_point = idx
```



- Level 결정
- Level보다 높은 layer에서는 가장 가까운 노드 하나만 탐색 (ef = 1)
  - Insert 진행 X
- Level 이하의 layer에서는 가까운 노드 efConstruction개 탐색
- 그중 가장 가까운 M개를 골라 이웃 노드로 저장
- 해당 이웃 노드들에 대해서도 이웃 업데이트 진행

HNSW python code

```python
### Algorithm 5: K-NN-SEARCH
def search(self, q, K=5, efSearch=20):
    """Find the K points closest to q."""

    distance = self.distance_func
    graphs = self._graphs
    ep = self._enter_point

    if ep is None:
        raise ValueError("Empty graph")

    neg_dist = -distance(q, self.data[ep])

    # line 1-5: search from top layers down to the second level
    for lc in reversed(graphs[1:]):
        neg_dist, ep = self._search_layer(q, [(neg_dist, ep)], lc, 1)[0]

    # line 6: search with efSearch neighbors at the bottom level
    W = self._search_layer(q, [(neg_dist, ep)], graphs[0], efSearch)

    if K is not None:
        W = nlargest(K, W)
    else:
        W.sort(reverse=True)

    return [(idx, -md) for md, idx in W]
```



- 마지막 layer 제외, 가장 가까운 노드 하나만 탐색 (ef = 1)
- 마지막 layer에서는 가까운 노드 efSearch개 탐색
- 그중 가장 가까운 K개를 골라 검색 결과로 반환

HNSW 성능 평가 (practice.ipynb)

과제 소개 (homework.ipynb)