Chapter 5
그래프의 비밀

```
// 버그 수정
void Pathfinder::Render()
{
  gdi->TransparentText();
  if (m_pGraph == NULL) {     // m_pGraph가 NULL이 되는 경우가 있음
        return;
  }
```

그래프 클래스 구현하기

- GraphNode 클래스

```
class GraphNode {
protected:
  //every node has an index. A valid index is >= 0
  int       m_iIndex;
public:
  GraphNode():m_iIndex(invalid_node_index){}
  GraphNode(int idx):m_iIndex(idx){}
  GraphNode(std::ifstream& stream){char buffer[50]; stream >> buffer >>
    m_iIndex;}
  virtual ~GraphNode(){}
  int  Index()const{return m_iIndex;}
  void SetIndex(int NewIndex){m_iIndex = NewIndex;}
};
```



```
template <class extra_info = void*>
class NavGraphNode : public GraphNode {
protected:
  //the node's position
  Vector2D    m_vPosition;

  extra_info  m_ExtraInfo;
public:
  ...
};
```

GraphEdge 클래스

- 두 개의 그래프 노드 사이의 연결을 나타내기 위해 필요한 기본 정보를 캡슐화

```
class GraphEdge {
protected:
  //An edge connects two nodes. Valid node indices are always positive.
  int     m_iFrom;
  int     m_iTo;
  //the cost of traversing the edge
  double  m_dCost;
public:
  GraphEdge(int from, int to, double cost):
     m_dCost(cost), m_iFrom(from), m_iTo(to) {}
  ...
};
```

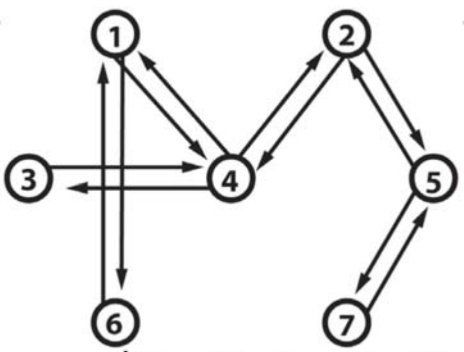

SparseGraph 클래스

// 각 노드의 인덱스 숫자가 그래프 노드들의 벡터(m_Nodes)와 인접 에지 리스트 (m_Edges)로 직접 연결

```
template <class node_type, class edge_type>
class SparseGraph {
public:
  typedef edge_type              EdgeType;
  typedef node_type              NodeType;
  //a couple more typedefs to save my fingers and to help with the formatting
  //of the code on the printed page
  typedef std::vector<node_type>  NodeVector;
  typedef std::list<edge_type>         EdgeList;
  typedef std::vector<EdgeList>        EdgeListVector;
};
```

타입정의

```
template <class node_type, class edge_type>
class SparseGraph {
public:
  typedef std::vector<node_type>  NodeVector;
  typedef std::list<edge_type>         EdgeList;
  typedef std::vector<EdgeList>        EdgeListVector;   -> 2중 list
private:
  NodeVector       m_Nodes; // 이 그래프를 구성하는 노드들
  // 인접 에지 리스트의 벡터 (각 노드 인덱스는 그 노드와 관련된 에지의     실제변수
  //                              리스트로 키(key)화 된다)
  EdgeListVector   m_Edges;
public:
  // invalid_node_index를 그 인덱스에 지정하여 노드를 제거
  void RemoveNode(int node);
};
```

	①	②	③	④	⑤	⑥	⑦
①	0	0	0	1	0	1	0
②	0	0	0	1	1	0	0
③	0	0	0	1	0	0	0
④	1	1	1	0	0	0	0
⑤	0	1	0	0	0	0	1
⑥	1	0	0	0	0	0	0
⑦	0	0	0	0	1	0	0

Figure 5.12. An adjacency matrix

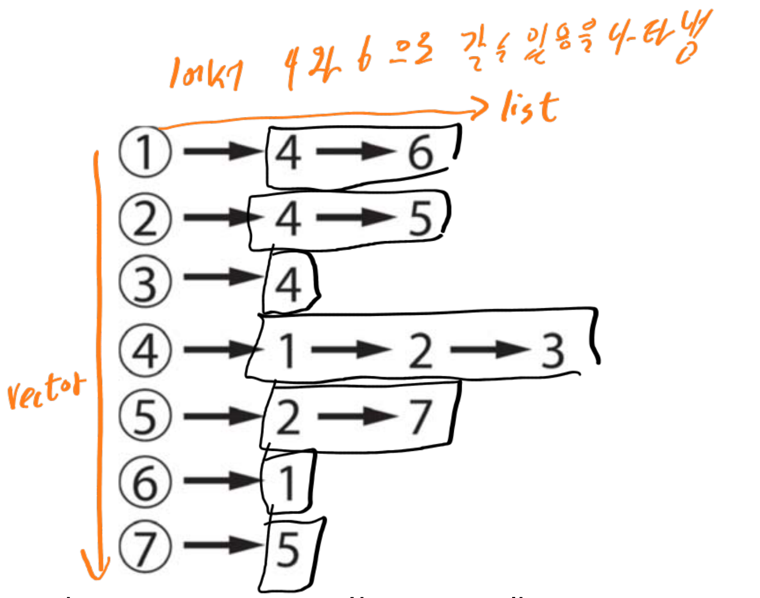


Figure 5.13. An adjacency list representation of the digraph from Figure 5.6

무정보 그래프 탐색

- 깊이 우선 탐색

```
template<class graph_type>
class Graph_SearchDFS {
  typedef typename graph_type::EdgeType Edge;
  typedef typename graph_type::NodeType Node;
private:
  //a reference to the graph to be searched
  const graph_type& m_Graph;
  std::vector<int>  m_Visited; // 방문할때마다 unvisited -> visited 설정
  std::vector<int>  m_Route;   // 각 노드의 부모를 기록하여 경로저장
                                  해답을 저장하는 용도
  //this method performs the DFS search
  bool Search();
  ...
};
```



```
template <class graph_type>
bool Graph_SearchDFS<graph_type>::Search()
{
  //create a std stack of edges                가장 최근의 분기점으로 돌아가기 위해
  std::stack<const Edge*> stack;
  //create a dummy edge and put on the stack
  Edge Dummy(m_iSource, m_iSource, 0);
  stack.push(&Dummy);
  //while there are edges in the stack keep searching
  while (!stack.empty())
  {
    //grab the next edge
    const Edge* Next = stack.top();
    //remove the edge from the stack
    stack.pop();
    //make a note of the parent of the node this edge points to
    m_Route[Next->To()] = Next->From();   내가 어디서 왔는지 저장
    //and mark it visited
    m_Visited[Next->To()] = visited;     방문처리
```

```
    //if the target has been found the method can return success
    if (Next->To() == m_iTarget) {
       return true;     }
     //push the edges leading from the node this edge points to onto
     //the stack (provided the edge does not point to a previously
     //visited node)
     graph_type::ConstEdgeIterator ConstEdgeItr(m_Graph, Next->To());
     for (const Edge* pE=ConstEdgeItr.begin(); !ConstEdgeItr.end();
          pE=ConstEdgeItr.next())
     {
       if (m_Visited[pE->To()] == unvisited) {
         stack.push(pE);
       }
     }
   }
   //no path to target
   return false;
 }
```

방문하지 않은 연결된 노드들을 탐색

실습

- DFS 개선하기
  - 반복적 깊이증가탐색 구현

너비우선탐색

```
template <class graph_type>
bool Graph_SearchBFS<graph_type>::Search()
{
  //create a std queue of edges
  std::queue<const Edge*> Q;
  const Edge Dummy(m_iSource, m_iSource, 0);
  //create a dummy edge and put on the queue
  Q.push(&Dummy);
  //mark the source node as visited
  m_Visited[m_iSource] = visited;
  //while there are edges in the queue keep searching
  while (!Q.empty())
  {
    //grab the next edge
    const Edge* Next = Q.front();
    Q.pop();
    //mark the parent of this node
    m_Route[Next->To()] = Next->From();
```

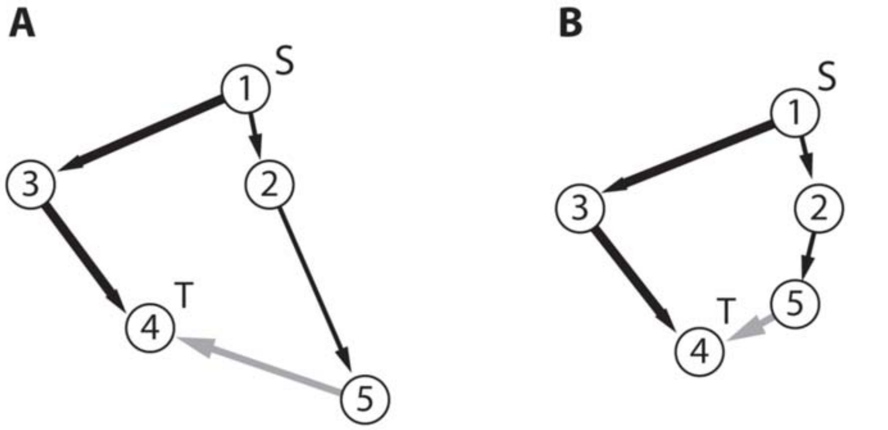


```
    //exit if the target has been found
    if (Next->To() == m_iTarget) {
      return true;
    }
    //push the edges leading from the node at the end of this edge
    //onto the queue
    graph_type::ConstEdgeIterator ConstEdgeItr(m_Graph, Next->To());
    for (const Edge* pE=ConstEdgeItr.begin(); !ConstEdgeItr.end();
         pE=ConstEdgeItr.next()) {
      //if the node hasn't already been visited we can push the
      //edge onto the queue
      if (m_Visited[pE->To()] == unvisited) {
        Q.push(pE);
        //and mark it visited
        m_Visited[pE->To()] = visited;
      }
    }
  }
  //no path to target
  return false;
}
```

이 노드에서 나가는 edge들을 push

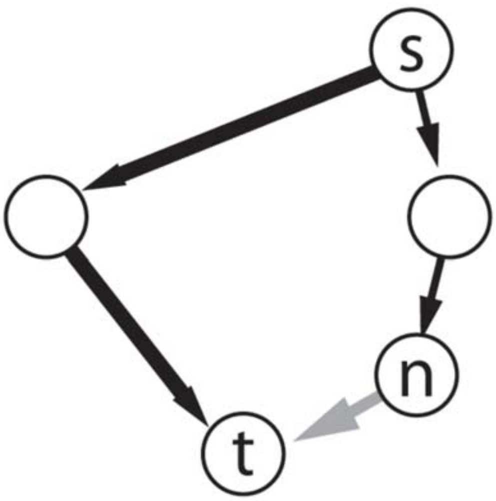
비용 기반 그래프 탐색

- 에지 완화
  - BPSF: Best Path found So Far
  - 어떤 노드까지의 경로가 기존의 최상 경로 대신 새롭게 검사되는 에지를 사용함으로써 더 짧아질 것으로 추론되면, 그 에지가 추가되고 그 경로는 적절하게 갱신됨



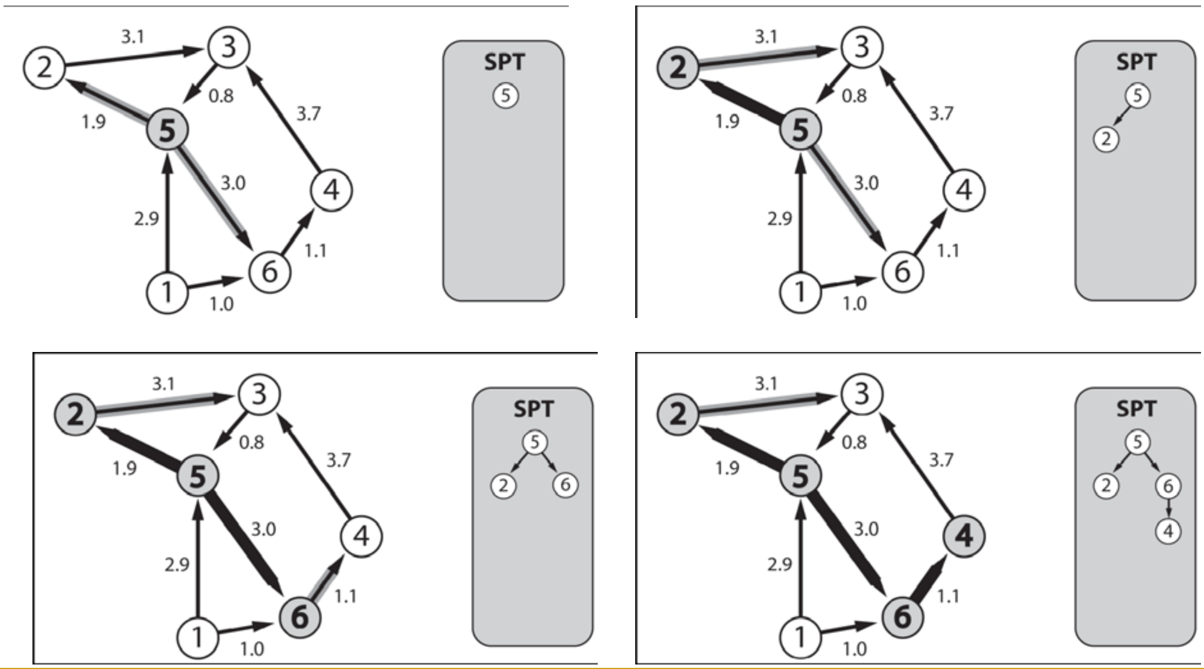
경로 갱신이 안되는 경우 경로가 갱신되는 경우

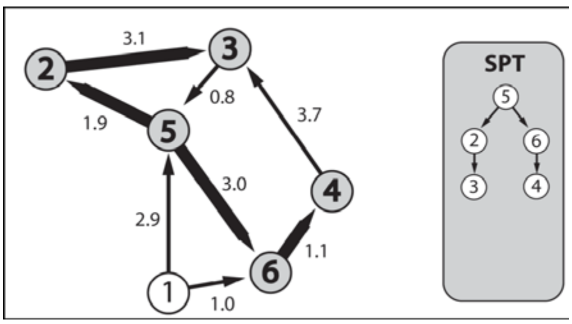

```
if (TotalCostToThisNode[t] > TotalCostToThisNode[n] + EdgeCost(n-to-t))
{
  TotalCostToThisNode[t] = TotalCostToThisNode[n] + EdgeCost(n-to-t));
  Parent(t) = n;
}
```



Dijkstra 알고리즘

- Root 노드에서 시작하여 한번에 하나의 최단경로트리(SPT: Shortest Path Tree)에 없는 에지를 추가하여 최단경로트리를 구축





* 유의점은 frontier에 후보 edge를 넣는다

```
//this is indexed into by node index and holds the total cost of the best
//path found so far to the given node. For example, m_CostToThisNode[5]
//will hold the total cost of all the edges that comprise the best path
//to node 5, found so far in the search (if node 5 is present and has
//been visited)
std::vector<double>              m_CostToThisNode;

//this is an indexed (by node) vector of 'parent' edges leading to nodes
//connected to the SPT but that have not been added to the SPT yet. This is
//a little like the stack or queue used in BST and DST searches.
std::vector<const Edge*>      m_SearchFrontier;
```

```
template <class graph_type>
class Graph_SearchDijkstra
{
  typedef typename graph_type::EdgeType Edge;
  const graph_type&           m_Graph;
  //this vector contains the edges that comprise the shortest path tree -
  //a directed subtree of the graph that encapsulates the best paths from
  //every node on the SPT to the source node.
  std::vector<const Edge*>     m_ShortestPathTree;
  std::vector<double>           m_CostToThisNode;
  //this is an indexed (by node) vector of 'parent' edges leading to nodes
  //connected to the SPT but that have not been added to the SPT yet. This
  // is a little like the stack or queue used in BST and DST searches.
  std::vector<const Edge*>     m_SearchFrontier;//그 노드로의 부모edge 유지
```

pq → 우선순위 큐


```
template <class graph_type>
void Graph_SearchDijkstra<graph_type>::Search()
{
    //get lowest cost node from the queue. Don't forget, the return value
    //is a *node index*, not the node itself. This node is the node not
    // already on the SPT that is the closest to the source node
    int NextClosestNode = pq.Pop(); // 최단 비용 순서는 pq에서 유지
    //move this edge from the frontier to the shortest path tree
    m_ShortestPathTree[NextClosestNode] = m_SearchFrontier[NextClosestNode];
    //if the target has been found exit
    if (NextClosestNode == m_iTarget) return;
    //now to relax the edges.
    graph_type::ConstEdgeIterator ConstEdgeItr(m_Graph, NextClosestNode);
    //for each edge connected to the next closest node
    for (const Edge* pE=ConstEdgeItr.begin();
        !ConstEdgeItr.end();
        pE=ConstEdgeItr.next())  {
      //the total cost to the node this edge points to is the cost to the
      //current node plus the cost of the edge connecting them.
      double NewCost = m_CostToThisNode[NextClosestNode] + pE->Cost();
```

노드번호

그 노드로의 edge임

그 노드에서 나가는 edge임

```
      //if this edge has never been on the frontier make a note of the cost
      //to get to the node it points to, then add the edge to the frontier
      //and the destination node to the PQ.  → 처음 확장된 노드일 경우
      if (m_SearchFrontier[pE->To()] == 0) {
        m_CostToThisNode[pE->To()] = NewCost;//소스부터 그 노드까지의 총비용
        pq.insert(pE->To());  → 큐에 넣는다
        m_SearchFrontier[pE->To()] = pE;//NextClosestNode부터 이 노드로의
      }                                  //edge를 탐색 경계에 후보로 추가
      //else test to see if the cost to reach the destination node via the
      //current node is cheaper than the cheapest cost found so far. If
      //this path is cheaper, we assign the new cost to the destination
      //node, update its entry in the PQ to reflect the change and add the
      //edge to the frontier                  → 기존 비용보다 더 적고
      else if ( (NewCost < m_CostToThisNode[pE->To()]) &&    → 해답에 이미 들어가 있는것이
                (m_ShortestPathTree[pE->To()] == 0) ) {          아니라면
        m_CostToThisNode[pE->To()] = NewCost;
        //because the cost is less than it was previously, the PQ must be
        //re-sorted to account for this.
        pq.ChangePriority(pE->To());
        m_SearchFrontier[pE->To()] = pE;
      }                        //pE->To()로의 기존 등록edge를 pE edge로 교체한다
    }
  }
}
```

아직 m_ShortestPathTree
에는 등록되지 않은 경우

- IndexedPriorityQLow<double> 의 작동 방법을 설명하시오.

  void Graph_SearchDijkstra<graph_type>::Search()

  IndexedPriorityQLow<double> pq

  - 구현 내용 설명
  - 작동 방법 설명

특별한 Dijkstra: A* g(x)+h(x)

Cost = AccumulativeCostTo(E.From) + E.Cost + CostTo(Target)

```
template <class graph_type, class heuristic>
class Graph_SearchAStar
{
private:
  //create a typedef for the edge type used by the graph
  typedef typename graph_type::EdgeType Edge;
private:
  const graph_type&             m_Graph;
  //indexed into my node. Contains the 'real' accumulative cost to that
node
  std::vector<double>            m_GCosts;
  //indexed into by node. Contains the cost from adding m_GCosts[n] to
  //the heuristic cost from n to the target node. This is the vector the
  //iPQ indexes into.
  std::vector<double>            m_FCosts;
  std::vector<const Edge*>       m_ShortestPathTree;
  std::vector<const Edge*>       m_SearchFrontier;//그 노드로의 부모edge
```


휴리스틱 클래스

Graph_SearchAStar

```
template <class graph_type, class heuristic>
class Graph_SearchAStar { ... };
class Heuristic_Euclid
{
public:
  Heuristic_Euclid(){}
  //calculate the straight line distance from node nd1 to node nd2
  template <class graph_type>
  static double Calculate(const graph_type& G, int nd1, int nd2)
  {
    return Vec2DDistance(G.GetNode(nd1).Pos(), G.GetNode(nd2).Pos());
  }
};
class Pathfinder {
  typedef SparseGraph<NavGraphNode<void*>, GraphEdge> NavGraph;...};
void Pathfinder::CreatePathAStar(){
  typedef Graph_SearchAStar<NavGraph, Heuristic_Euclid> AStarSearch;
  //create an instance of the A* search using the Euclidean heuristic
  AStarSearch AStar(*m_pGraph, m_iSourceCell, m_iTargetCell);
...}
```

↳두 점의 거리를 구하는 것으로 h를 사용

↳생성자에서 search()가 불리도록 구현되어 있음

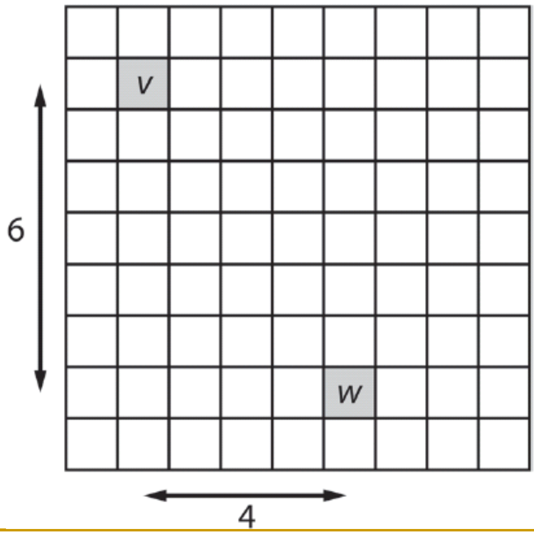
```
template <class graph_type, class heuristic>
void Graph_SearchAStar<graph_type, heuristic>::Search()
{ //create an indexed priority queue of nodes. The nodes with the
  //lowest overall F cost (G+H) are positioned at the front.
  IndexedPriorityQLow<double> pq(m_FCosts, m_Graph.NumNodes());
  //put the source node on the queue
  pq.insert(m_iSource);
  //while the queue is not empty
  while(!pq.empty()) {
    //get lowest cost node from the queue
    int NextClosestNode = pq.Pop();
    //move this node from the frontier to the spanning tree
    m_ShortestPathTree[NextClosestNode] = m_SearchFrontier[NextClosestNode];
    //if the target has been found exit
    if (NextClosestNode == m_iTarget) return;
    //now to test all the edges attached to this node
    graph_type::ConstEdgeIterator ConstEdgeItr(m_Graph, NextClosestNode);
    for (const Edge* pE=ConstEdgeItr.begin(); !ConstEdgeItr.end();
         pE=ConstEdgeItr.next()) {
      //calculate the heuristic cost from this node to the target (H)
      double HCost = heuristic::Calculate(m_Graph, m_iTarget, pE->To()); h값
```



```
      //calculate the 'real' cost to this node from the source (G)
      double GCost = m_GCosts[NextClosestNode] + pE->Cost();        g값
      //if the node has not been added to the frontier, add it and update
      //the G and F costs
      if (m_SearchFrontier[pE->To()] == NULL)  { → 목록에 없으면
        m_FCosts[pE->To()] = GCost + HCost;
        m_GCosts[pE->To()] = GCost;
        pq.insert(pE->To()); → 비용
        m_SearchFrontier[pE->To()] = pE; → 실제 엣지를 봅는다.
      }
      //if this node is already on the frontier but the cost to get here
      //is cheaper than has been found previously, update the node
      //costs and frontier accordingly. → 기존비용보다 더 적으면
      else if ((GCost < m_GCosts[pE->To()]) && (m_ShortestPathTree[pE-
>To()]==NULL))  {
        m_FCosts[pE->To()] = GCost + HCost;
        m_GCosts[pE->To()] = GCost;
        pq.ChangePriority(pE->To());
        m_SearchFrontier[pE->To()] = pE; //그 노드로의 edge를 변경
      }
    }
  }
}
```

A*

- 맨하탄 거리 휴리스틱으로 변환
  - 맨하탄 거리: 가로와 세로 방향의 변이의 합
  - Euclidean 거리와의 성능 비교
  - Euclidean 거리 제곱 휴리스틱으로 변환하고 시간 비교



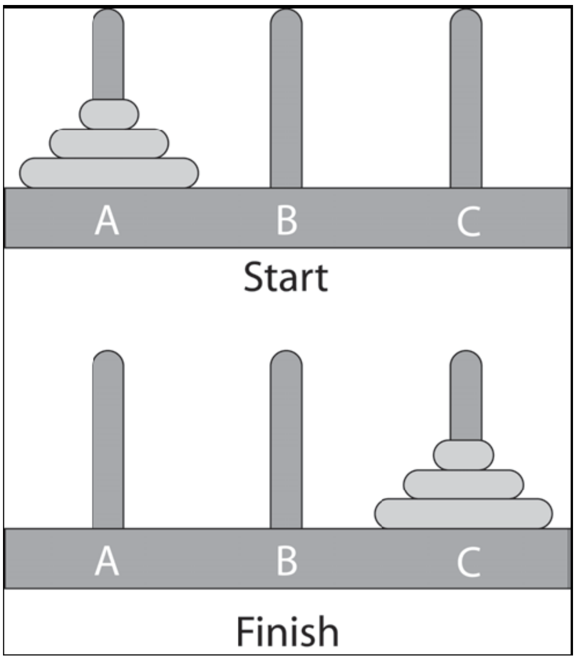
N 퀸(queen) 퍼즐

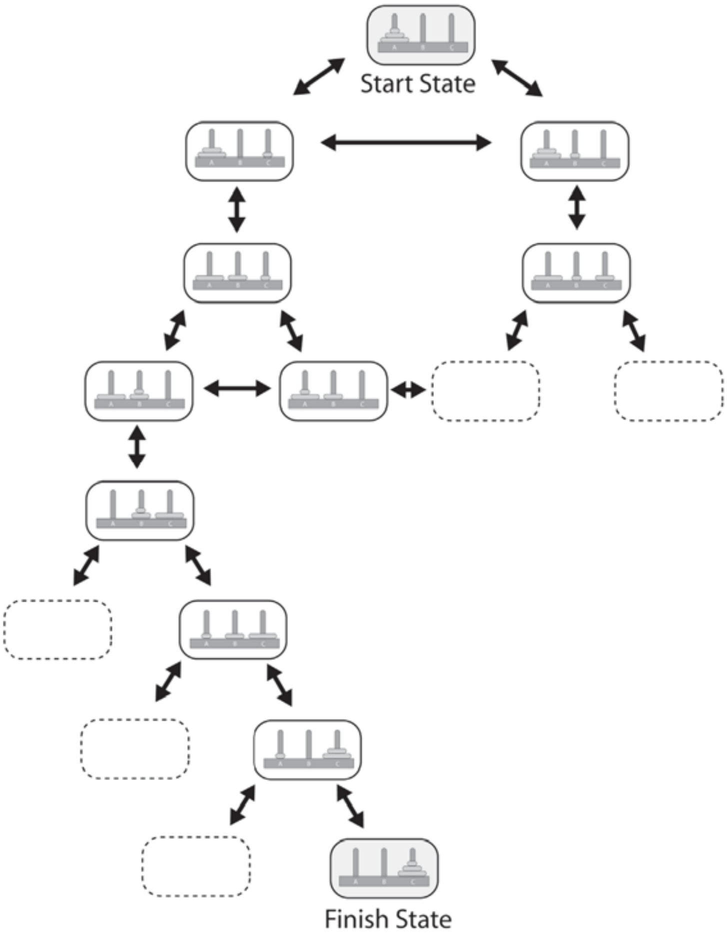
- N×N 체스 판 위에 N개의 퀸을 서로 잡지 못하도록 배치하는 문제 (어느 행, 열 또는 대각선에도 반드시 하나의 퀸만이 있어야 한다).
  - 시작 노드는 빈 N×N 배열
  - 후속자 함수는 하나의 퀸을 정당한 임의의 위치에 추가한 새로운 N×N 배열을 만들어낸다
  - 목표 술어는 배열 안에(정당한 위치에) N개의 퀸이 존재하는 경우에만 만족

실습

- N 퀸 퍼즐에 대한 해
  - $\hat{h}(n) \geq 0$, A^* 그래프 탐색으로 풀이
  - 언덕 오르기(Hill-climbing)
  - 무작위 재시작 언덕 오르기(Random restart)
    - 옵션: 횡이동 (sideway move)

n 디스크 하노이 탑





실습

- n 디스크 하노이 탑 퍼즐에 대한 해
  - $\hat{h}(n) \geq 0$, A* 그래프 탐색으로 풀이