Chapter 8

실질적인 길 계획하기

그래프에 아이템 추가하기

```
template <class extra_info = void*>
class NavGraphNode : public GraphNode
{
protected:
  //the node's position
  Vector2D     m_vPosition;
  extra_info  m_ExtraInfo;   // 여기에 아이템 주소가 할당됨
public:
  //ctors
  NavGraphNode():m_ExtraInfo(extra_info()){}
  NavGraphNode(int      idx,
               Vector2D pos):GraphNode(idx),
                             m_vPosition(pos),
                             m_ExtraInfo(extra_info())
  {}
}
```

// Raven_Game 클래스는 맵이 로딩될 때 노드들을 셀 공간 방법으로 분할한다

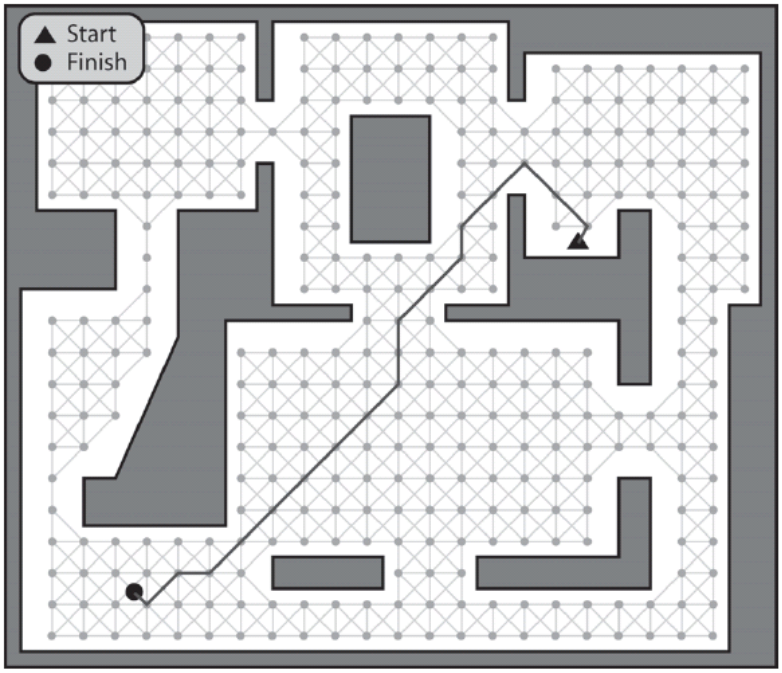
경로 계획자

```
class Raven_PathPlanner {
  typedef std::list<PathEdge>                    Path;
  //A pointer to the owner of this class
  Raven_Bot*                          m_pOwner;
  //a reference to the navgraph
  const Raven_Map::NavGraph&          m_NavGraph;
  //a pointer to an instance of the current graph search algorithm.
  Graph_SearchTimeSliced<EdgeType>*  m_pCurrentSearch;
  //this is the position the bot wishes to plan a path to reach
  Vector2D                            m_vDestinationPos;
  //returns the index of the closest visible and unobstructed graph node to
  //the given position
  int   GetClosestNodeToPosition(Vector2D pos)const;
  //smooths a path by removing extraneous edges. (may not remove all
  //extraneous edges)
  void  SmoothPathEdgesQuick(Path& path);
  //smooths a path by removing extraneous edges. (removes *all* extraneous
  //edges)
  void  SmoothPathEdgesPrecise(Path& path);
```



```
  //creates an instance of the A* time-sliced search and registers it with
  //the path manager
  bool       RequestPathToItem(unsigned int ItemType);
  //creates an instance of the Dijkstra's time-sliced search and registers
  //it with the path manager
  bool       RequestPathToPosition(Vector2D TargetPos);
  //called by an agent after it has been notified that a search has terminated
  //successfully. The method extracts the path from m_pCurrentSearch, adds
  //additional edges appropriate to the search type and returns it as a list of
  //PathEdges.
  Path       GetPath();
  //the path manager calls this to iterate once though the search cycle
  //of the currently assigned search algorithm. When a search is terminated
  //the method messages the owner with either the msg_NoPathAvailable or
  //msg_PathReady messages
  int        CycleOnce()const;
};
```

특정 위치까지의 경로 계획하기



봇에서 가장 가까운 노드

특정 위치까지의 경로 계획하기

1. bot의 현재 위치에서 가장 가까우면서 장애물에 방해받지 않는 가시의 그래프 노드를 찾는다.
2. 목표 위치까지의 가장 가까우면서 장애물에 방해받지 않는 가시의 그래프 노드를 찾는다.
3. 둘 사이의 최소 비용 경로를 찾아내는 탐색 알고리즘을 사용한다.

bool Raven_PathPlanner::RequestPathToPosition(Vector2D TargetPos)

```
bool Raven_PathPlanner::RequestPathToPosition(Vector2D TargetPos)
{
 m_vDestinationPos = TargetPos;
 //find the closest visible node to the bots position int ClosestNodeToBot =
GetClosestNodeToPosition(m_pOwner->Pos());
 //find the closest visible node to the target position
  int ClosestNodeToTarget = GetClosestNodeToPosition(TargetPos);
 //create an instance of a the distributed A* search class
  typedef Graph_SearchAStar_TS<Raven_Map::NavGraph, Heuristic_Euclid> AStar;

  m_pCurrentSearch = new AStar(m_NavGraph,
                               ClosestNodeToBot,
                               ClosestNodeToTarget);

  //and register the search with the path manager
  m_pOwner->GetWorld()->GetPathManager()->Register(this);
  return true;
}
```

time slice 버전 : 루프가 한번만 돌도록 한 버전

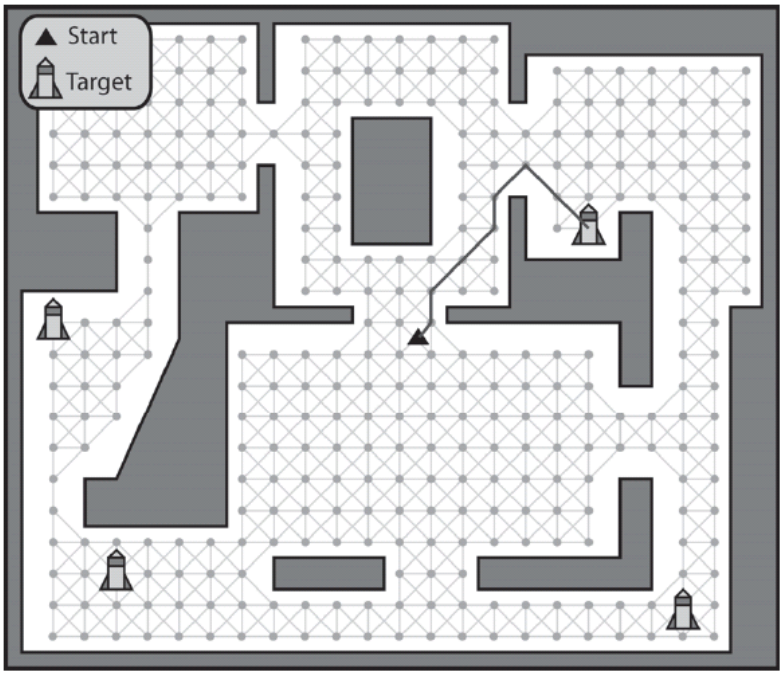
뇌에서 해야할 일

```
void Goal_MoveToPosition::Activate()
{
  m_iStatus = active;
  //make sure the subgoal list is clear.
  RemoveAllSubgoals();
  //requests a path to the target position from the path planner. Because, for
  //demonstration purposes, the Raven path planner uses time-slicing when
  //processing the path requests the bot may have to wait a few update cycles
  //before a path is calculated. Consequently, for appearances sake, it just
  //seeks directly to the target position whilst it's awaiting notification
  //that the path planning request has succeeded/failed
  if (m_pOwner->GetPathPlanner()->RequestPathToPosition(m_vDestination))
  {
    AddSubgoal(new Goal_SeekToPosition(m_pOwner, m_vDestination));
  }
}
```

2번째 고비

"찾아줘"해놓고 목적지로 직접헤엄

어떤 아이템 타입까지의 경로 계획하기



A*를 목적지가 일어야만 사용 가능하다.

⇒ 다익스트라 사용

어떤 아이템 타입까지의 경로 계획하기

- 인스턴스가 많은 아이템에 대한 경로
  - A*
    - 근원위치와 목표위치 둘 다를 가져야 함
    - 게임 세계에 있는 각각의 인스턴스에 대한 탐색을 모두 해야 함
    - 단지 하나의 아이템을 찾기 위해 많은 A* 탐색이 요구됨
  - Dijkstra 알고리즘
    - 목표를 찾거나 그래프 전체를 탐색할 때까지 루트 노드로부터 바깥 방향으로 최단경로트리(shortest path tree, SPT)를 성장
    - 많은 유사한 아이템 타입이 있는 경우, 찾고자 하는 아이템이 발견되자마자, 종료
    - SPT는 루트로부터 원하는 타입의 가장 가까운 아이템까지의 경로를 포함

```
bool Raven_PathPlanner::RequestPathToItem(unsigned int ItemType)
{
  //find the closest visible node to the bots position
  int ClosestNodeToBot = GetClosestNodeToPosition(m_pOwner->Pos());
 //create an instance of the search algorithm
  typedef FindActiveTrigger<Trigger<Raven_Bot> > t_con;
  typedef Graph_SearchDijkstras_TS<Raven_Map::NavGraph, t_con> DijSearch;

  m_pCurrentSearch = new DijSearch(m_NavGraph,
                                   ClosestNodeToBot,
                                   ItemType);
 //register the search with the path manager
  m_pOwner->GetWorld()->GetPathManager()->Register(this);

  return true;
}
```

↳ 봇에서 가장 가까운 노드 찾는 과정


```
void Goal_GetItem::Activate()
{
  m_iStatus = active;

  m_pGiverTrigger = 0;

  //request a path to the item
  m_pOwner->GetPathPlanner()->RequestPathToItem(m_iItemToGet);

  //the bot may have to wait a few update cycles before a path is calculated
  //so for appearances sake it just wanders
  AddSubgoal(new Goal_Wander(m_pOwner));

}
```

↓
배회하기

```
int Raven_PathPlanner::CycleOnce()const
{
 int result = m_pCurrentSearch->CycleOnce();
 if (result == target_not_found) {
     Dispatcher->DispatchMsg(SEND_MSG_IMMEDIATELY, SENDER_ID_IRRELEVANT,
                             m_pOwner->ID(), Msg_NoPathAvailable,
                             NO_ADDITIONAL_INFO);
  }
 else if (result == target_found) {
  void* pTrigger =
    m_NavGraph.GetNode(m_pCurrentSearch->GetPathToTarget().back()).ExtraInfo();
    Dispatcher->DispatchMsg(SEND_MSG_IMMEDIATELY, SENDER_ID_IRRELEVANT,
                            m_pOwner->ID(), Msg_PathReady,
                            pTrigger);
  }
  return result;
}
```


Paths as Nodes or Paths as Edges?

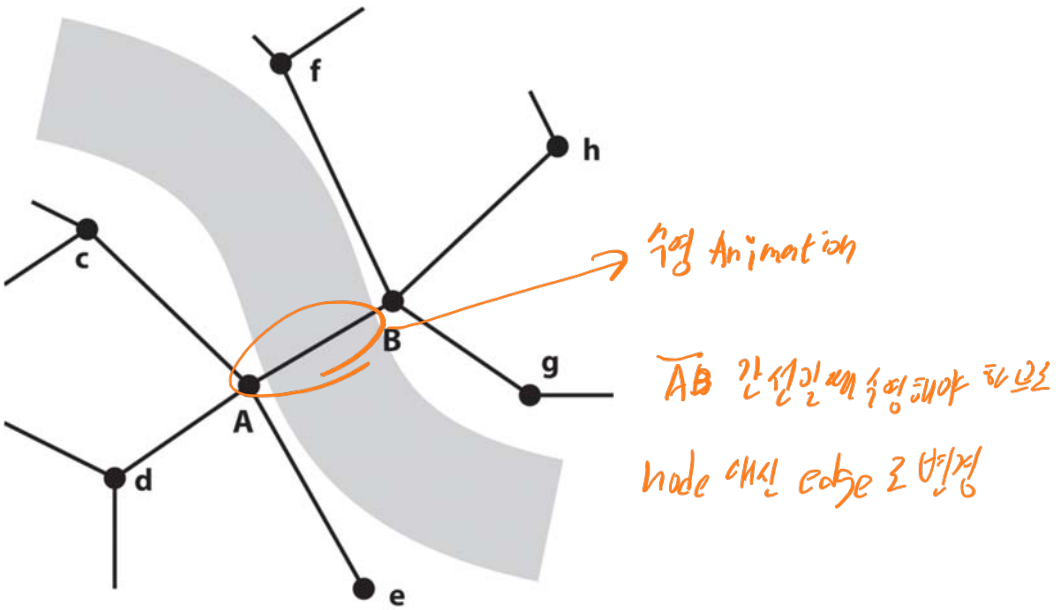


Figure 8.10. A navgraph spanning a river

An Annotated Edge Class Example

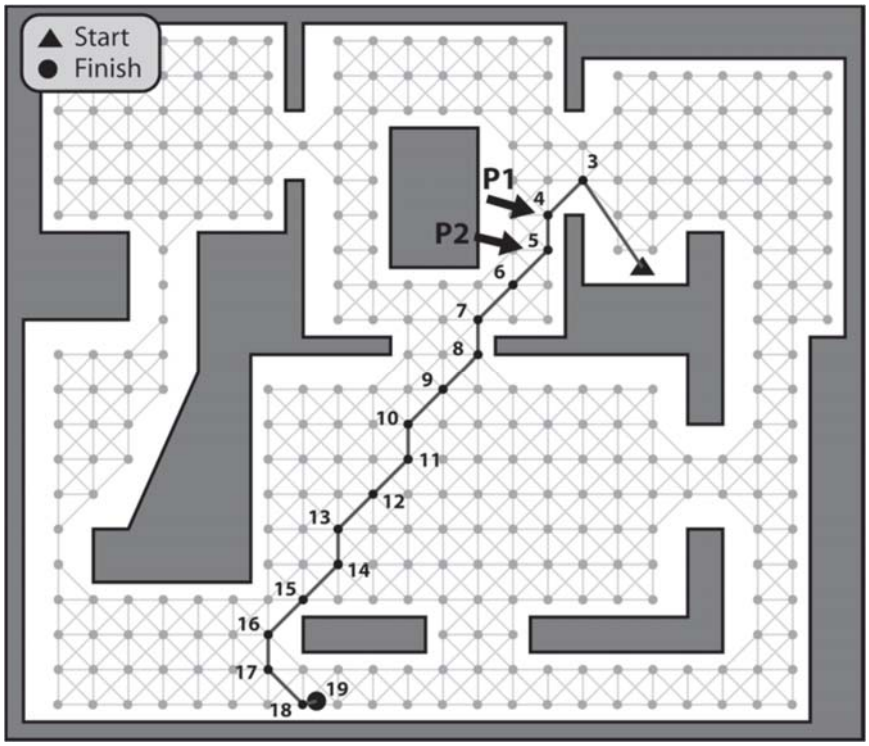
```
class NavGraphEdge : public GraphEdge {
  //examples of typical flags
  enum  {
    normal            = 0,
    swim              = 1 << 0,
    crawl             = 1 << 1,
    creep             = 1 << 3,
    jump              = 1 << 3,
    fly               = 1 << 4,
    grapple           = 1 << 5,
    goes_through_door = 1 << 6
  };
  int   m_iFlags;
  //if this edge intersects with an object (such as a door or lift), then
  //this is that object's ID.
  int  m_ilDofIntersectingEntity;
...
};
```



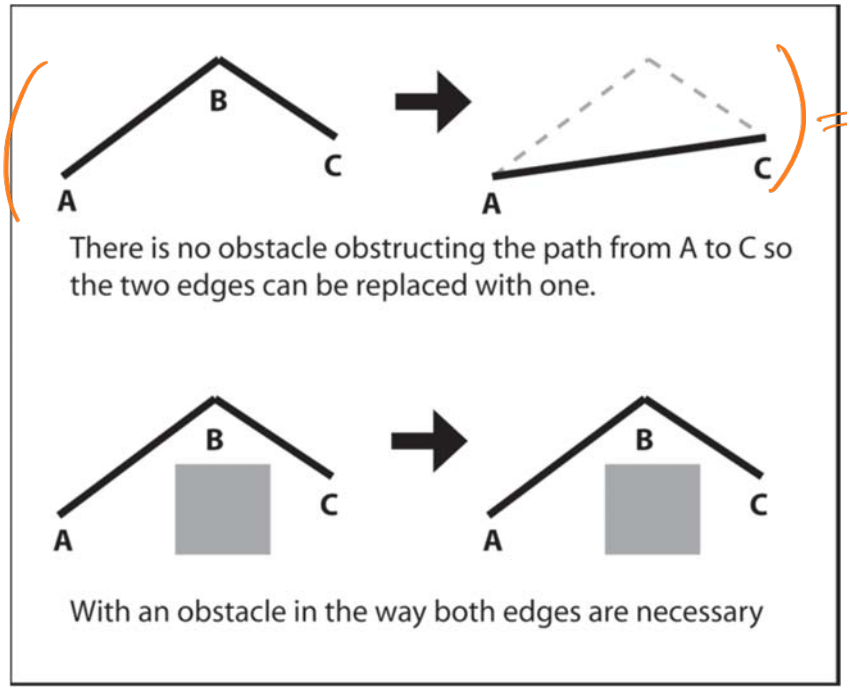
```
template <class graph_type, class heuristic>
std::list<int>
Graph_SearchAStar_TS<graph_type, heuristic>::GetPathToTarget()const
{
  std::list<int> path;
 //just return an empty path if no target or no path found
  if (m_iTarget < 0)  return path;
  int nd = m_iTarget;
  path.push_back(nd);

  while ((nd != m_iSource) && (m_ShortestPathTree[nd] != 0))
  {
    nd = m_ShortestPathTree[nd]->From();

    path.push_front(nd);
  }
  return path;
}
```

```
template <class graph_type, class heuristic>
std::list<PathEdge>
Graph_SearchAStar_TS<graph_type, heuristic>::GetPathAsPathEdges()const
{
  std::list<PathEdge> path;
  //just return an empty path if no target or no path found
  if (m_iTarget < 0)  return path;
  int nd = m_iTarget;
  while ((nd != m_iSource) && (m_ShortestPathTree[nd] != 0))
  {
    path.push_front(PathEdge(m_Graph.GetNode(m_ShortestPathTree[nd]->From()).Pos(),
                             m_Graph.GetNode(m_ShortestPathTree[nd]->To()).Pos(),
                             m_ShortestPathTree[nd]->Flags(),
                             m_ShortestPathTree[nd]->IDofIntersectingEntity()));

    nd = m_ShortestPathTree[nd]->From();
  }
  return path;
}
```


Path Smoothing



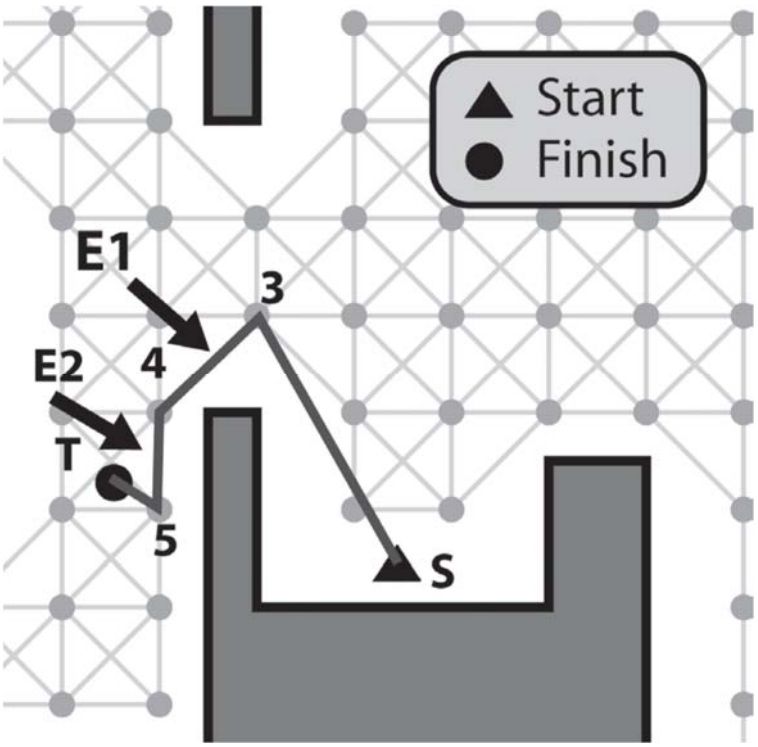
거칠지만 빠르게 경로 부드럽게 하기



=> 할 수 있는 데까지 수행


```
void Raven_PathPlanner::SmoothPathEdgesQuick(Path& path)
{
  //create a couple of iterators and point them at the front of the path
  Path::iterator e1(path.begin()), e2(path.begin());
  //increment e2 so it points to the edge following e1.
  ++e2;
 while (e2 != path.end())
  {
    //check for obstruction, adjust and remove the edges accordingly
    if ( (e2->Behavior() == EdgeType::normal) &&
         m_pOwner->canWalkBetween(e1->Source(), e2->Destination()) )
    {
      e1->SetDestination(e2->Destination());
      e2 = path.erase(e2);
    }
    else
    {
      e1 = e2;
      ++e2;
    }
  }
}
```

정교하지만 느린 경로 부드럽게 하기



4-5-T를 펴면

3-4-T로 펼쳐질 것

정교하지만 느린 경로 부드럽게 하기

```
void Raven_PathPlanner::SmoothPathEdgesPrecise(Path& path)
{
  //create a couple of iterators
  Path::iterator e1, e2;

  //point e1 to the beginning of the path
  e1 = path.begin();

  while (e1 != path.end())
  {
    //point e2 to the edge immediately following e1
    e2 = e1;
    ++e2;
```

```
  while (e2 != path.end())
     {
       //check for obstruction, adjust and remove the edges accordingly
       if ( (e2->Behavior() == EdgeType::normal) &&
            m_pOwner->canWalkBetween(e1->Source(), e2->Destination()) )
       {
         e1->SetDestination(e2->Destination());
         e2 = path.erase(++e1, ++e2);
         e1 = e2;
         --e1;
       }
       else
       {
         ++e2;
       }
     }
     ++e1;
   }
 }
```



```
template <class path_planner>
class PathManager
{
private:
  //a container of all the active search requests
  std::list<path_planner*>  m_SearchRequests;
  //this is the total number of search cycles allocated to the manager.
  //Each update-step these are divided equally amongst all registered path
  //requests
  unsigned int              m_iNumSearchCyclesPerUpdate;
public:
  //every time this is called the total amount of search cycles available will
  //be shared out equally between all the active path requests. If a search
  //completes successfully or fails the method will notify the relevant bot
  void UpdateSearches();
  //a path planner should call this method to register a search with the
  //manager. (The method checks to ensure the path planner is only registered
  //once)
  void Register(path_planner* pPathPlanner);
  void UnRegister(path_planner* pPathPlanner);
  //returns the amount of path requests currently active.
  int  GetNumActiveSearches()const{return m_SearchRequests.size();}
};
```

시간별 경로 계획하기

- 시간 쪼개기(Time Slicing)
  - 갱신 단계마다 일정한 양의 CPU 자원을 할당
    - 검색들에 균등하게 배분
  - 검색은 다수의 갱신 단계에 걸쳐서 그래프를 탐색
  - 에이전트가 검색을 요청
    - 경로 계획자(Path Planner)가 관련 검색(A*나 Dijkstra) 인스턴스를 만들고 경로 운영자(Path Manager) 클래스에 등록
    - 경로 운영자는 모든 활성화된 경로 계획자의 포인터 리스트를 유지
      - 매 시간 단계마다 경로 계획자 각각에 CPU 자원을 균등하게 배분
      - 성공 또는 실패 결과를 경로 계획자는 소유자에게 메시지로 알림

실습

- 부분 경로 생성(검색) 기능 추가
  - 사용자가 정의한 수의 검색 사이클이나 검색 깊이가 도달된 후에는 목표에 가장 가까운 노드에 이르는 경로를 반환하도록 A* 알고리즘을 변경

경로가 완성되기전에 부분경로 등을 이용해서 사용하도록 해보자