컴실1 최종 프로젝트 보고서

20180501 김연수

1. 실험 목표 & 실험 환경:

화면상으로 우주복을 입은 팬더가 움직이는 모습을 통해서 DFS와 BFS 알고리즘이 동작하는 방식을 더 명확하게 표현한다. 미로 상에 랜덤하게 배치되는 안경은 벽을 꿰뚫어 보는 능력을 제공한다. 우주복을 입은 팬더가 이 안경을 집으면, 목표지점까지 헤매지 않고 최단거리로 이동할 수 있다.

팬더가 움직이는 방식은 DFSMODE, BFSMODE 두 가지 방식이 존재한다. DFSMODE의 경우 팬더는 걸어서 이동한다. BFSMODE의 경우 팬더는 순간이동 능력을 사용한다.

실험 환경 : visual studio 2022 release, open framework

Os : window

1. 변수 설명 : 주석 참고, global variable위주로 설명.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

1. 함수 설명 : maze project 이후 최종 프로젝트에서 추가한 기능들 위주로 설명.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

Dfs의 함수 내에 panda의 이동경로를 저장하기 위해서 임시 변수 pan에 a, b좌표를 저장하고, “panda.push(pan)”으로 panda에 pan을 push해준다. 위 변수 설명을 참고하면, Panda는 global queue 자료구조로 선언되어 있다. Queue 자료구조를 여기서 사용한 이유는 판다가 이동하는 경로를 화면에서 순서대로 보여주기 위함이다. 따라서 bfs든 dfs든 방문하는 노드의 a, b 좌표가 순서대로 panda에 저장된다. 위 코드는 dfs지만, bfs에도 똑같이 위 코드를 추가해줬다.

함수의 마지막 줄에서 리턴하기 전에, BFS\_from\_glasses()함수를 호출해준다. 함수의 자세한 코드 설명은 바로 아래에서 설명하겠지만, 주석에도 쓰여 있듯 이 함수는 안경의 위치로부터 목표지점까지의 최단 경로를 계산해 저장한다. dfsmode에서나 bfsmode에서나 완전미로에서 경로는 하나이기 때문에 dfs와 bfs모두 각각에 대해 함수를 만들지 않고, BFS\_from\_glasses로 통일했다.

BFS\_from\_glasses 함수로부터 구한 최단경로를 저장하기 위해서 다른 구조체를 선언하거나 하지 않고, 기존 GRAPH 구조체에 glasses를 위한 field를 추가해주는 형태로 작성했다. GRAPH에 추가한 field는 visit\_glasses, prev\_glasses, next\_glasses로 총 세가지다. 순서대로 방문 여부, 전 노드, 다음 노드를 의미한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

위 함수의 기본적인 틀은 BFS()와 같다. Graph 상에서 안경이 위치하는 노드에서 시작해 bfs알고리즘으로 목표지점(last value를 갖는 노드)까지 탐색해 경로를 알아낸다. 경로는 prev\_glasses노드를 통해 연결되어 있으므로 position의 위치까지 전 노드로 이동하면서, 전 노드의 next\_glasses를 현 노드로 연결시켜 준다. 이 작업을 하고 나면, position에서 마지막 노드까지 완전 경로를 통해 next로 연결된다.

Next로 노드를 연결시켜 준 후에, glasses\_path에 next\_glasses를 따라가며 glasses\_path에 각 노드의 좌표 정보를 push해준다. Glasses\_path는 queue 자료구조이면서 전역변수로 선언되어 있기 때문에, update할 때마다 판다가 방문한 노드를 순서대로 따라가는 모습을 연출할 수가 있다.

Prev와 prev\_glasses, next와 next\_glasses가 bfs와 bfs\_from\_glasses간의 가장 큰 차이라고 볼 수 있다. 코드를 직접 작성해보기 전엔 이미 bfs 또는 dfs함수를 통해 만들어진 graph상의 next로 이어진 경로를 통해 안경으로부터 목표지점까지 최단거리도 구할 수 있을 것이라고 예상했다. 하지만, 안경의 위치가 완전미로의 경로 위에 위치할 것이라는 보장이 없기 때문에, 안경의 위치로 부터 다시 목표지점까지 경로를 생성해야 함을 깨달았다. 따라서 위와 같이 새로운 필드와 새로운 함수를 추가해줬고, 예상했던 것보다 까다로운 작업이 됐다.

다음은 update함수이다. 구체적인 함수의 코드는 아래와 같다. 변수에 대한 설명은 주석을 참고하자.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

update함수는 판다를 움직이게 하는 기능을 주로 담당한다. 함수에 대한 대략적인 설명은 다음과 같다. GAME이 시작되고 나서 안경을 흭득하기 전까지는 panda라는 이름으로 선언한 queue에서 좌표를 가져온다. 안경을 흭득한 이후에는 glasses\_path로부터 좌표를 가져와 움직이도록 구현했다. Panda queue가 비어있으면서 erase\_glasses가 false이거나, glasses\_path queue가 비어있으면서 erase\_glasses가 true인 경우, 즉 안경을 흭득하지 않고 목표지점까지 도달하거나, 안경을 흭득하고 목표지점까지 도달한 경우, 게임이 종료되고, isGAME 은 0, GAMEOVER은 true로 초기화 해준다.

판다가 걸어가는 모습을 연출하기 위해서, pos\_x와 pos\_y변수를 두었다. queue에서 받아오는 것은 end\_x, end\_y이고, 그 end\_x와 end\_y의 지점으로 가기위해서 pos\_x와 pos\_y를 1만큼씩 조정해준다. 다만 여기서 bfs의 경우에는 판다가 순간이동을 할 수 있다는 설정이므로, end\_x로 이동하도록 설정한다. 보다 자연스럽게 걸어가는 모습을 연출하기 위해서 setup()함수에서 “ofSetFrameRate(60)”으로 조정했다.

마지막으로 update된 내용을 반영하기 위해서 draw()함수 내에서 update() 함수를 호출했다.

Draw()함수를 아래와 같이 update했다. 기존에 있던 미로를 그리는 코드에 추가적으로 랜덤한 위치에 배정되는 안경을 그려줬고, 게임이 시작하기 전에 우주복을 입은 판다 캐릭터를 출발지점에 표시해주는 디테일을 추가했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. 플로우 차트 & 자료구조와 알고리즘 & 시공간 복잡도 :

Flowchart :

도표, 텍스트, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

자료구조 및 알고리즘 :

전역변수로 선언한 자료구조를 위주로 설명하자면, maze, path, panda glasses\_path, graph가 있다. 각각에 대해 공간 복잡도를 분석하면 다음과 같다.

Maze는 미로를 그리기 위해 구현한 vector 자료구조이다. 따라서 Maze에 입력으로 들어오는 원소의 숫자는 행 \* 열. 여기서 행과 열은 n\*m 미로에서 각각 2 \* n + 1, 2 \* m + 1을 뜻한다. 따라서 O(n\*m)의 공간 복잡도를 가진다.

Path는 DFS또는 BFS 알고리즘의 결과로 생성된 경로를 저장하는 vector 자료구조다. 이 또한 최악의 경우 모든 칸을 거친다고 가정하면, 공간 복잡도는 O(n\*m)이라고 할 수 있다.

Panda는 판다 캐릭터가 지나갈 경로의 좌표를 저장한다. 따라서 이 또한 최악의 경우 모든 칸을 지나갈 것이고, 공간 복잡도는 O(n\*m)이라고 할 수 있다.

Glasses\_path는 안경으로부터 목표지점까지 경로의 좌표를 저장한다. 이 또한 위와 마찬가지 이유로 O(n\*m)의 공간 복잡도를 가진다.

graph를 표현하기 위해 만들어지는 노드의 개수는 n\*m의 미로에서 정확히 n \* m 개이다. 다른 자료구조들과 비교했을 때 대략 4분의 1만큼이나 작지만, 이것 역시 O(n\*m)의 공간 복잡도를 가진다.

Position은 head를 루트로 삼는 graph의 노드 중 하나를 가리키므로 다른 공간을 필요로 하지 않는다.

전체 프로그램에 대한 알고리즘 설명은 다음과 같다. 자세한 알고리즘은 앞선 함수 설명을 참고하라. GAME이 시작되고, 모드를 선택하고 나면, 모든 자료구조에 정보가 업데이트 된다. 각 실행마다 pos\_x와 pos\_y 그리고 end\_x와 end\_y만 변동하고, 나머지 자료구조는 프로그램 실행에 영향을 받지 않는다.

프로그램의 전체적인 시간 복잡도를 살펴보면 다음과 같다. 파일을 오픈할 때 READFILE()함수가 실행되는데, READFILE()의 시간 복잡도는 O(n\*m)이다. 또 DFS함수, BFS 함수, dfsdraw, bfsdraw도 마찬가지로 O(n\*m)의 시간 복잡도를 가진다. Update 함수의 시간 복잡도는 게임 도중인 경우 O(1)이고 게임이 끝난 경우 dfsdraw or bfsdraw 함수를 호출하게 되므로 O(n\*m)의 시간 복잡도를 가진다. Draw 함수에서는 미로를 그리는 작업을 수행해야 하므로 O(n\*m)의 시간 복잡도를 가진다.

1. 창의적 구현 :

판다 캐릭터가 화면상에서 움직이는 모습을 연출하기 위해서 처음 생각한 것은 1초마다 팬더의 위치를 다음 칸으로 update하는 것이었다. 따라서 update함수 내에서 queue에서 좌표를 얻어와 이동하도록 했다. 하지만 바로 다음 칸으로 이동하는 것이 부자연스러워 보였기 때문에 더 자연스럽게 움직이도록 pos\_x와 pos\_y를 전역변수에 추가하여, end\_x와 end\_y에 도달할 때까지 1씩 천천히 증가하도록 설정했다. 그리고 pos\_x와 pos\_y가 모두 end\_x와 end\_y와 같아질 때(즉 다음 칸에 도착했을 때), queue에서 다음 칸의 정보를 받아와 end\_x, end\_y에 저장한다.

게임을 시작하면, 캐릭터가 DFS 또는 BFS에 맞춰서 알고리즘에 맞게 칸을 방문하면서 목표지점을 탐색한다. 미로를 탐색하는 와중에 미로에 떨어진 안경을 장착하면, 목표지점으로 최단거리를 찾아 이동하게 된다.

위 기능을 구현하기 위해서 처음에 생각한 것은 저장되어 있는 dfs 또는 bfs상의 next로 연결된 path를 이용하는 것이었다. Dfs 또는 Bfs 함수에서 이미 graph를 각 알고리즘에 맞게 목표 지점까지 갈 수 있는 path를 next로 연결해 놓았기 때문에, 그냥 이를 이용하면 쉽게 구현 할 수 있을 것이고 코드 몇 줄만 추가하면 될 일이라고 보았다.

하지만, 코드를 작성하면서 놓치고 있는 것이 있다는 것을 깨달았다. 안경의 위치는 랜덤이기 때문에 완전 미로의 경로에 안경이 위치할 것이라는 보장이 없는 것이다. 이를 생각하지 못하고 프로그램을 돌려보니 캐릭터가 벽을 넘어서 이동하는 것을 관찰할 수 있었다. 디버깅 도중에 이를 깨닫고나서 코드를 재작성 해야만 했다.

따라서, 안경의 위치로부터 새로운 path를 생성해야만 하는 것이다. 새로운 graph를 생성해 구현하는 것은 코드 상에서 가독성을 떨어뜨릴 뿐만 아니라, 메모리 공간을 낭비하는 결과를 낳는다. 따라서 기존의 graph의 노드를 구현하기 위해 생성한 GRAPH 구조체의 field에 새로운 path를 연결해주기 위한 요소를 추가했다. 이를 이용해 BFS\_from\_glasses 함수에서 기존 graph에 안경으로부터 목표지점을 잇는 path를 생성할 수 있었다. 그리고 이 안경으로부터 목표지점까지 이어주는 path를 DFS, BFS함수 내에서 호출했다. 따라서 프로그램이 시작하면 파일을 읽어올 때 안경의 위치를 랜덤으로 지정하고, DFS, BFS 함수가 호출될 때 시작지점부터 목표지점까지의 path와 안경의 위치로부터 목표지점까지 path모두 업데이트 된다.

1. 실행 결과는 사진과 함께 설명하도록 한다.

file open이후 게임 시작 전 화면 :

도표, 스케치, 패턴이(가) 표시된 사진

자동 생성된 설명

판다 캐릭터가 이동하는 화면 :

도표, 스케치이(가) 표시된 사진

자동 생성된 설명

안경 획득 후 안경이 사라지고 최단 경로를 통해 이동 :

도표, 직사각형, 디자인, 패턴이(가) 표시된 사진

자동 생성된 설명

목표지점에 도착하면 아래와 같이 path를 화면에 표시(DFS) :

도표, 라인, 직사각형, 평면도이(가) 표시된 사진

자동 생성된 설명

1. 느낀 점 & 개선 사항

중간에 프로젝트를 하면서 느낀 것은 공간 복잡도를 희생해 시간 복잡도를 줄일 수 있고, 반대로 시간 복잡도를 희생해 공간 복잡도를 줄일 수 있다는 사실이다. 미리 계산해 놓고 그 결과를 적절한 자료구조에 저장해 놓는다면 후에 그 자료를 이용할 때, 걸리는 시간 복잡도가 감소한다. 하지만 결과를 저장하지 않고 필요할 때 직접 계산해서 사용할 수도 있다. 예를 들면, 이번 프로젝트처럼 DFS의 path를 필요할 때 graph를 직접 탐색해 얻을 수도 있지만, 필요한 경우를 대비해 적절한 자료구조에 저장해 놓으면 후에 access time을 감소시킬 수 있는 것이다. 그렇기 때문에 어떤 것이 더 효율적인지를 잘 고민해봐야 한다는 점을 깨달았다.

하다 보니 계속해서 아쉬운 부분들이 보이고 기능을 더 추가하고 싶은 마음이 있었지만, 시험 기간이고 프로젝트가 겹쳐서 더 많은 기능을 구현하지 못했다는 점이 아쉬움으로 남는다. 사실 처음 생각했던 기능은 망치 기능이었다. 망치를 획득하면 주변 벽을 부수고 이동할 수 있도록 게임을 구현할 생각이었다. 하지만, 코드를 직접 구현해봤을 때, 코드가 복잡해지고 결코 쉽지 않은 작업이 될 것이라는 직감이 들었다. 안경 기능을 구현하고 나서 시간이 남으면 구현할 생각이었지만, 시간 부족으로 구현하지 못했다. 더불어 수동 작동 방식 생각했지만, 같은 이유로 구현하지 못했다. 안경 기능 하나만으로 만족해야 했다.

전체적으로 예상했던 것보다 생각했던 내용들을 구현하는 것이 상당히 어려웠다. Open framework에 대한 지식이 충분하지 못했던 탓도 있지만, 디버깅 하기가 어려웠던 탓도 있었다. 지금까지 mac을 이용해 실험을 진행했지만, 이번 maze project에서는 윈도우와 visual studio를 이용하다 보니 visual studio와 윈도우 키보드 단축키 사용에 미숙했던 것도 이번 프로젝트를 하는데 걸림돌이 되었다.

추가할 기능을 직접 생각해보고 구현하는 작업을 하기 위해서 자연스럽게 능동적인 자세를 가질 수밖에 없었다. 능동적인 자세로 프로젝트에 임하고 구현한 코드가 생각했던 대로 실행되었을 때 그냥 주어진 과제를 하는 것보다 더 큰 성취감을 느꼈다.