## Mô tả game play, xử lý va chạm và xử lý bom nổ

* Trong một màn chơi, Bomber sẽ được người chơi di chuyển, đặt và kích hoạt Bomb với mục tiêu chính là tiêu diệt tất cả Enemy và tìm ra vị trí Portal để có thể qua màn mới
* Bomber sẽ bị giết khi va chạm với Enemy hoặc thuộc phạm vi Bomb nổ. Lúc đấy trò chơi kết thúc.
* Enemy bị tiêu diệt khi thuộc phạm vi Bomb nổ
* Một đối tượng thuộc phạm vi Bomb nổ có nghĩa là đối tượng đó va chạm với một trong các tia lửa được tạo ra tại thời điểm một đối tượng Bomb nổ.
* Khi Bomb nổ, một Flame trung tâm tại vị trí Bomb nổ và bốn Flame tại bốn vị trí ô đơn vị xung quanh vị trí của Bomb xuất hiện theo bốn hướng trên/dưới/trái/phải. Độ dài bốn Flame xung quanh mặc định là 2 đơn vị, được tăng lên khi Bomber sử dụng các FlameItem.

Khi các Flame xuất hiện, nếu có một đối tượng thuộc loại Brick/Wall nằm trên vị trí một trong các bombBang thì độ dài Flame đó sẽ được giảm đi để sao cho Flame chỉ xuất hiện đến vị trí đối tượng Brick/Wall theo hướng xuất hiện. Lúc đó chỉ có đối tượng Brick/Wall bị ảnh hưởng bởi Flame, các đối tượng tiếp theo không bị ảnh hưởng. Còn nếu vật cản bombBang là một đối tượng Bomb khác thì đối tượng Bomb đó cũng sẽ nổ ngay lập tức.

**Mô tả starter project**

Xem comment ở starter project

**Yêu cầu chung**

* Có thể chơi được ít nhất cho một màn chơi (chiến thắng một màn chơi)
* Có thể thay đổi được tệp cấu hình khác cho màn chơi (tương tự mẫu cho trước)

**Nhiệm vụ của Nhóm**

1. Xây dựng bản đồ màn chơi từ tệp cấu hình
2. Di chuyển Bomber theo sự điều khiển từ người chơi
3. Tự động di chuyển các Enemy Xử lý va chạm cho các đối tượng Bomber, Monster, Wall, Stone, Bomb Xử lý bom nổ
4. Xử lý khi Bomber sử dụng các Item và khi đi vào vị trí Portal

* Tự xây dựng project mà không dùng starter code Gói tùy chọn 2

1. Nâng cấp thuật toán tìm đường cho Monster
2. Cài đặt thuật toán AI cho Bomber ,Xử lý hiệu ứng âm thanh (thêm music & sound effects)
3. Những ý tưởng khác sẽ được đánh giá và cộng điểm theo mức tương ứng

Project được xây dựng xây dựng hoàn toàn bằng các thư viện có sẵn của java với đồ họa sử dụng Java Swing.

1. Xây dựng bản đồ màn chơi từ tệp cấu hình:

Nhiệm vụ của bạn là lấy các thông tin về vị trí của các đối tượng ở trong tệp cấu hình và hiển thị vào trong game.

Trong đó:

Dòng đầu tiên gồm 3 số nguyên bao gồm: số thứ tự của màn chơi (ví dụ level 1 thì có giá trị là 1), số hàng R và số cột C của màn chơi.

Tiếp là ma trận R x C có R dòng, mỗi dòng có chứa C ký tự chứa toàn bộ thông tin vị trí của các đối tượng trong game.

Trong đó:

 # - Wall: tường

\* - Stone : đá

 x - Portal: cổng kết thúc game

p - Bomber: Nhân vật chính

1 - Ghost: Nhân vật Ghost hình bóng bay

b - Item: Vật phẩm tăng số lượng bom

 f – Item\_bomSize: Vật phẩm tăng sức công phá của bom

s - Item\_shoe: Vật phẩm tăng tốc độ của người chơi

Kí tự khác các kí tự trên - Background\_hightscore: Cỏ nền

Để thực hiện bài tập này, các bạn sẽ phải sửa lại lớp **FileLevelLoader** được kế thừa từ LevelLoader. Trong đó, ở hàm loadLevel(), các bạn phải đọc file cấu hình và lưu trữ ma trận chứa thông tin của các đối tượng vào mảng ma trận 2 chiều cho trước **\_map**. Ngoài mảng **\_map** , bạn cũng cần cập nhật các thuộc tính **\_level** , **\_width** , **\_height** trong lớp **FileLevelLoader** để các đối tượng khác trong game có thể lấy được thông tin. Chú ý, file cấu hình lưu trong resources của project nên bạn phải đọc file theo resources chứ không phải đường dẫn bình thường, tham khảo các hàm Class.getResourceAsStream và Class.getResource() .

Sau đó, bạn phải sửa lại hàm **createEntities** () để đọc các thông tin trong mảng **\_map** và hiển thị trong game. Trong hàm đã có ghi chú và các ví dụ để tạo mới và hiển thị các đối tượng trong game.

Lưu ý là trong Game có 2 dạng tọa độ chính: Tile và Pixel. Trong đó tọa độ tile là tọa độ số nguyên tương ứng với tọa độ của đối tượng trong mảng **\_map**. Còn tọa độ Pixel là tọa độ số thập phân tương ứng với tọa độ được hiển thị trong Game. Bạn tham khảo lớp Coordinates để chuyển đổi giữa 2 loại tọa độ. Khi hiển thị đối tượng (bằng cách gọi hàm **Entity.render()**), ta sử dụng tọa độ Pixel. Mặc định, các đối tượng kế thừa lớp Character có tham số x, y ở Constructor là tọa độ Pixel vì các đối tượng này có thể di chuyển nên vị trí thường là số thập phân. Còn các đối tượng tĩnh, không di chuyển kế thừa lớp Tile sẽ nhận tham số x, y là tọa độ Tile. Tham khảo hàm **Tile.render()** để tìm hiểu thêm.

1. Di chuyển Bomber theo sự điều khiển từ người chơi

Nhiệm vụ này yêu cầu điều khiển nhân vật bomber sử dụng bàn phím và di chuyển màn hình theo nhân vật bomber.

Trong phần này, các bạn tham khảo 2 lớp **Keyboard** và **Bomber**. Lớp **Keyboard** có nhiệm vụ nhận và xử lý input từ bàn phím của người dùng, các bạn không phải sửa lớp này.

Lớp Bomber đại diện cho nhân vật bomber. Để di chuyển, các bạn cần sửa lại các hàm **calculateMove** (), **canMove** () và **move** ().

Hàm **move** () di chuyển nhân vật đến địa điểm xác định. Ở đây, bạn sử dụng **canMove** () để kiểm tra xem nhân vật có thể di chuyển tới điểm đã tính toán hay không và thực hiện thay đổi các tọa độ **\_x** , **\_y.** Lưu ý bạn phải cập nhật giá trị thuộc tính **\_direction** (hướng đi của nhân vật) khi di chuyển để animation được hiển thị chính xác.

Hàm **calculateMove** () xử lý nhận tín hiệu điều khiển hướng đi từ **\_input** và gọi **move** () để thực hiện di chuyển. Các bạn phải tích toán khoảng cách theo cả hai trục x và y mà nhân vật bomber sẽ dịch chuyển dựa trên tốc độ của nhân vật. Để lấy tốc độ di chuyển của nhân vật, các bạn sử dụng **Game.getPlayerSpeed** (). Các bạn lưu ý nhớ cập nhật lại giá trị cờ **\_moving** khi thay đổi trạng thái di chuyển, khi di chuyển thì để giá trị bằng **true** , còn không thì để là **false**. Giá trị này sẽ kiểm tra để hiển thị animation cho nhân vật.

Hàm **canMove** () kiểm tra xem nhân vật có thể di chuyển đến vị trí đó không. Bạn cần kiểm tra xem vị trí cần di chuyển tới có đối tượng nào ở đó chưa. Để tìm đối tượng tại 1 vị trí nhất định, bạn sử dụng hàm **Board.getEntity** (double **x** , double **y** , Character **m** ) trong đó **m** là đối tượng cần xét va chạm. Hàm sẽ trả về đối tượng nằm ở vị trí x, y và không phải là **m.** Thêm nữa, bạn cần sử dụng hàm **collide** () trong lớp entity để kiểm tra 2 đối tượng có được tính là va chạm hay không (phần 4 sẽ mô tả chi tiết). Nếu có va chạm thì không cho di chuyển.

Lưu ý, khi di chuyển nhân vật bomber, bạn sẽ gặp trường hợp nhân vật không di chuyển hoàn toàn vào chính giữa ô dẫn đến khi di chuyển 1 số hướng sẽ bị vướng không đi được. Do đó, khi nhân vật di chuyển đến gần sát điểm chính giữa, bạn nên lập trình di chuyển hẳn nhân vật đó ra giữa.

1. Tự động di chuyển các Monster

Tương tự như di chuyển nhân vật bomber, bạn sẽ phải chỉnh sửa các hàm **calculateMove** (), **canMove** () và **move** () trong lớp **Monster**. Lưu ý là thay vì lấy input nhập từ bàn phím thì bạn sẽ lấy hướng đi của đối tượng từ thuộc tính **\_ai** qua hàm **AI.calculateDirection** ().

Với AI đơn giản nhất là đi ngẫu nhiên, bạn phải sửa lại các lớp **AILow** để trả về giá trị ngẫu nhiên. Việc quy định giá trị số nguyên trả về tương ứng với hướng đi nào là do bạn tự quyết định.

1. Xử lý va chạm cho các đối tượng Bomber, Monster, Brick, Bomb

Ở đây các bạn sẽ phải sửa hàm **collide** () của tất cả các đối tượng **Bomber** , **Monster**, **Wall** , **Brick** , **Bomb**. Các hàm nhận tham số là một đối tượng Entity bất kỳ này trả về false nếu xác định có va chạm với đối tượng đó và true trong trường hợp không va chạm. Đồng thời, hàm cũng xử lý các sự kiện cần thiết trong game:

\* Nếu **Brick** va chạm với **Bombang** thì **Brick** sẽ bị hủy diệt. Tham khảo hàm **DestroyableTile.destroy** ().

\* Nếu **Bomb** va chạm với **Bombang**  thì sẽ tự động nổ.  Tham khảo hàm **Bomb.explode** ()

\* Nếu **Bomber** va chạm với **Bombang**  hoặc **Monster** thì sẽ bị chết.  Tham khảo hàm **Bomber.kill** ().

\* Nếu **Monster** va chạm với **Bombang**  thì sẽ bị chết.

Các hàm này được gọi trong các trường hợp muốn xác định một đối tượng có thể được di chuyển hay không (hàm **Bomber.canMove** ()và **Monster.canMove** ()) hay xử lý khi bom nổ (hàm **Flame.calculatePermitedDistance** ()).

Bạn lưu ý với **Bomb** , khi nhân vật bomber khi bắt đầu đặt bom thì nhân vật bomber nằm cùng vị trí với bom. Khi đó để nhân vật bomber di chuyển được thì hàm **collide** () của **Bomb** phải xác định xem bomber đang có cùng vị trí (nằm trong) bom hay không, nếu có thì phải trả về **false** để nhân vật có thể đi qua. Ngược lại, khi nhân vật đã đi ra bên ngoài bom và đi vào, phải trả về **true** để chặn lại.

1. Xử lý bom nổ

Nhiệm vụ này yêu cầu phải cài đặt để bomber có thể đặt bom và kích hoạt bom nổ.

Để đặt bom, hãy tham khảo hai phương thức **detectPlaceBomb** () và **placeBomb** () trong lớp **Player**.

Phương thức **detectPlaceBomb** () kiểm tra các điều kiện có thể đặt bom hay không, bao gồm: có nhận tín hiệu đặt bom từ người chơi ( **\_input.space** ), thời gian giữa hai lần đặt bom và người chơi còn bom để đặt hay không. Bạn cần tham khảo hàm **Game.getBombRate** () và thuộc tính **Game.\_timeBetweenPutBombs**. Hàm **Game.getBombRate** () sẽ trả về số lượng bom có thể đặt liên tiếp tại thời điểm hiện tại, tức số lượng bom còn lại của người chơi. Mặc định số bom đặt tối đa ở một thời điểm là 1, bạn có thể điều chỉnh giá trị này ở **Game.BOMBRATE**. Thuộc tính **\_timeBetweenPutBombs** dùng để ngăn chặn **Bomber** đặt 2 bom cùng tại 1 vị trí trong 1 khoảng thời gian quá ngắn. Khi đặt bom, bạn hay để giá trị này bằng 30 (tương ứng với 30 khung hình) hoặc một con số bất kì. Sau mỗi khung hình (mỗi lần gọi hàm **update** ()), giá trị này giảm đi 1 đơn vị. Ta sẽ quy ước khi giá trị này nhỏ hơn 0 là có thể đặt được bom. Nếu 3 điều kiện trên thỏa mãn thì thực hiện đặt bom bằng **Bomber** ()

Lưu ý, sau khi đặt bom, nhớ giảm số lượng **BombRate** đi 1 (tham khảo hàm **Game.addBombRate** ()).

Với phương thức **placeBomb** (), bạn chỉ đơn giản là thêm một đối tượng **Bomb** vào bản đồ ( **\_board** ).

Tiếp theo là xử lý bom nổ, trước tiên hãy tham khảo lớp **Bomb**. Thuộc tính **\_timeToExplode** xác định xem bom có được nổ hay không. Giá trị khởi tạo là 120, sau mỗi khung hình, giá trị này giảm đi 1 cho đến 0 thì bom có thể nổ được. Tức là sau 123 khung hình (khoảng 2 giây) thì bom sẽ tự động nổ. Khi đó, ta sẽ gọi phương thức **explode** () để kích hoạt nổ bom. Nhiệm vụ đầu tiên của bạn là sửa lại phương thức này. Trong phương thức, bạn sẽ phải kiểm tra xem có nhân vật (bomber hoặc kẻ thù) có ở vị trí đặt bom hay không, nếu có thì nhân vật sẽ chết ngay lập tức bằng cách gọi hàm **Character.kill** (). Khi bom nổ, bom sẽ sinh ra 4 tia lửa tương ứng với 4 hướng trái, phải, trên, dưới. Mỗi tia lửa sẽ là một đối tượng thuộc lớp Flame. Bạn cần gán giá trị vào mảng **\_flames** trong lớp Bomb để sử dụng sau này.

Tiếp theo, mỗi Flame có thể có độ dài ngắn khác nhau tùy thuộc vào sức mạnh của bom (có thể ăn item để tăng) và tia lửa có bị chặn lại bởi đối tượng khác hay không. Độ dài này là số nguyên tính bằng số ô mà tia lửa kéo dài. Mỗi **Flame** có thể có nhiều **FlameSegment** , tức là các đoạn lửa nhỏ, tương ứng với một ô trên bản đồ. Nhiệm vụ của bạn là sửa lại hàm **createFlameSegments** () và **calculatePermitedDistance** (). Phương thức **calculatePermitedDistance** () sẽ tính toán độ dài của tia lửa. Lưu ý là nếu tia lửa chạm với các đối tượng khác (ví dụ là tường) thì sẽ bị chặn lại. Phương thức **createFlameSegments** () sẽ tạo ra các tia lửa và gán vào mảng **\_flameSegments**. Chú ý, tham số **last** trong Constructor của **FlameSegment** để xác định tia lửa có phải cuối cùng hay không (vì phần đầu tia lửa có hình dạng khác so với các phần khác).

1. Xử lý khi Bomber sử dụng các Item và khi đi vào vị trí Portal

Nhiệm vụ này bạn sẽ phải xử lý trường hợp Bomber ăn các item. Có 3 loại Item: **item\_bomb** giúp tăng số lượng tối đa bomb được đặt cùng lúc ở một thời điểm, **item\_bombsize** giúp tăng độ dài của tia lửa khi bom nổ, **item\_Shoe** giúp tăng tốc độ chạy của nhân vật Bomber. Ở đây, bạn phải sửa hàm **collide** () trong các lớp tương ứng là **item\_bomb**, **item\_bombsize** và **item\_Shoe**. Với mỗi loại, bạn sẽ tăng thêm cho nhân vật tương ứng tác dụng.

Xử lý khi đi vào vị trí **Portal** , tức là kết thúc game, bạn cần chỉnh sửa lại hàm **collide** () trong lớp **Portal**. Lưu ý là khi tất cả các kẻ địch đều bị tiêu diệt thì mới có thể kết thúc trò chơi. Để kết thúc màn chơi cũ và chuyển sang màn mới, hãy sử dụng **\_board.nextLevel** ().

1. Nâng cấp thuật toán tìm đường cho Monster và cài đặt thêm các Monster khác

Bạn có thể cài đặt lại thuật toán tìm đường cho các Enemy để tăng độ khó của game. Hãy viết thêm AI mới bằng cách kế thừa lớp **AI** và cài đặt hàm **calculateDirection** (). Bạn có thể tùy ý sử dụng bất cứ thông tin nào trong game (ví dụ thông tin bản đồ, vị trí của Bomber,...) để phục vụ cho việc tính toán (hãy truyền các thông tin này qua Constructor).

Bạn có thể cài đặt thêm các Enemy khác bằng cách kế thừa lớp Monster (tương tự như nhân vật **Ghost** ). Bạn phải cài đặt hàm **chooseSprite** (), hàm này lấy các thông tin cần thiết (thường là **\_direction** và **\_moving** ) để chọn ra Sprite tương ứng để hiển thị (gán vào trường **\_sprite** ).

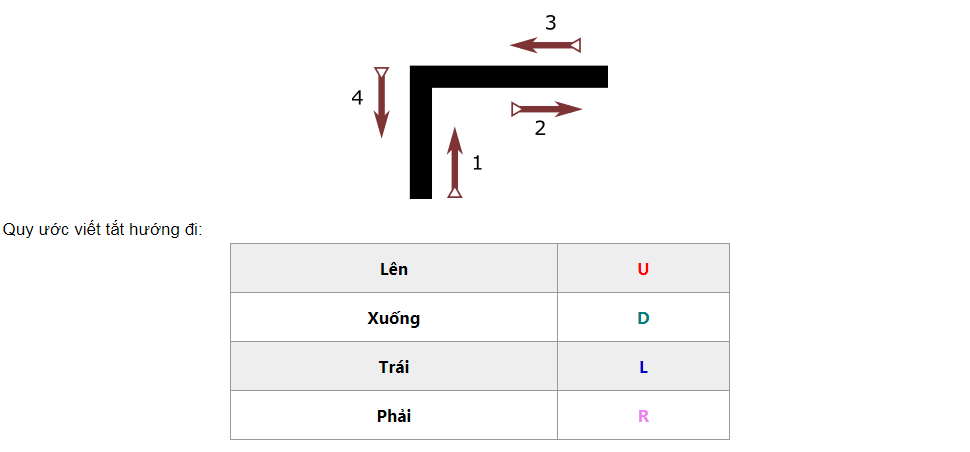
Các sprite có sẵn cho các nhân vật [**Oneal**](http://bomberman.wikia.com/wiki/Onil), [**Dall**](http://bomberman.wikia.com/wiki/Dahl), [**Minvo**](http://bomberman.wikia.com/wiki/Minvo) và [**Doria**](http://bomberman.wikia.com/wiki/Doria). Các nhân vật này có nhiều phiên bản game khác nhau với các đặc tính khác nhau. Do đó, ta thống nhất là cài đặt đặc tính của nhân vật theo phiên bản của [NES](http://bomberman.wikia.com/wiki/Bomberman_(NES)). Bạn có thể phải thêm AI, xử lý va chạm hoặc các phần khác để cài đặt các đặc tính riêng của nhân vật.

Các nhân vật [**Ovape**](http://bomberman.wikia.com/wiki/Ovape), [**Pontan**](http://bomberman.wikia.com/wiki/Pontan) và [**Pass**](http://bomberman.wikia.com/wiki/Pass) cũng đã có sẵn trong file textures nhưng chưa được load vào game).

1. Xử lý hiệu ứng âm thanh

Bạn sẽ phải thêm âm thanh vào trong game, bao gồm âm thanh nền và âm thanh của các event trong game (bom nổ, nhân vật chết, thắng cuộc, ăn item,...). âm thanh nền với event là độc lập với nhau, tức việc phát âm thanh sự kiện không ảnh hưởng đến âm thanh nền.

* **thuật toán bám tường**
* Trước tiên, để thực hiện được thuật toán, xe dò đường của bạn tất nhiên phải có chức năng ghi lại những hướng đã đi tại các điểm rẽ. Những hướng đó được lấy theo hê tọa độ cố định gắn với mê cung (**ví dụ như lên, xuống, trái, phải**). Ta lưu lại những hướng này vào một chuỗi để tiến hành rút gọn.
* Nguyên tắc của thuật toán này là khử dần những đường đi ngược nhau, lặp lại liên tục, cho đến khi nào không thể rút gọn được nữa, thì đó sẽ là đường đi ngắn nhất. Trong chuỗi đã lưu sẽ có những cặp ký tự chỉ hướng ngược nhau và đứng liền nhau, ta sẽ nhận biết được đó là ngõ cụt. Để bỏ qua ngõ cụt đó, ta chỉ cần xóa chúng đi là xong.



Như vậy bắt đầu từ 1 ta có:

* 1: **U**
* 2: **L**
* 3: **R**
* 4: **D**

=> **ULRD**

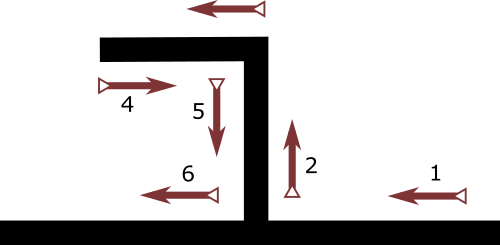
Ta nhận thấy những hướng ngược chiều nhau và đứng cạnh nhau là ngõ cụt (ở trên là L,R). Do đó ta có thể bỏ đi.

**U~~LR~~D** => **UD**

Chỉ còn lại **UD**. **UD** lại tiếp tục ngược nhau, ta lại rút gọn thêm một lần nữa...

...và chuỗi **ULRD** đã không còn, do đó xe sẽ bỏ qua đường này. devil

**Ví dụ 2:** Ví dụ tiếp theo với một đường đi thẳng và đường rẽ ngang.

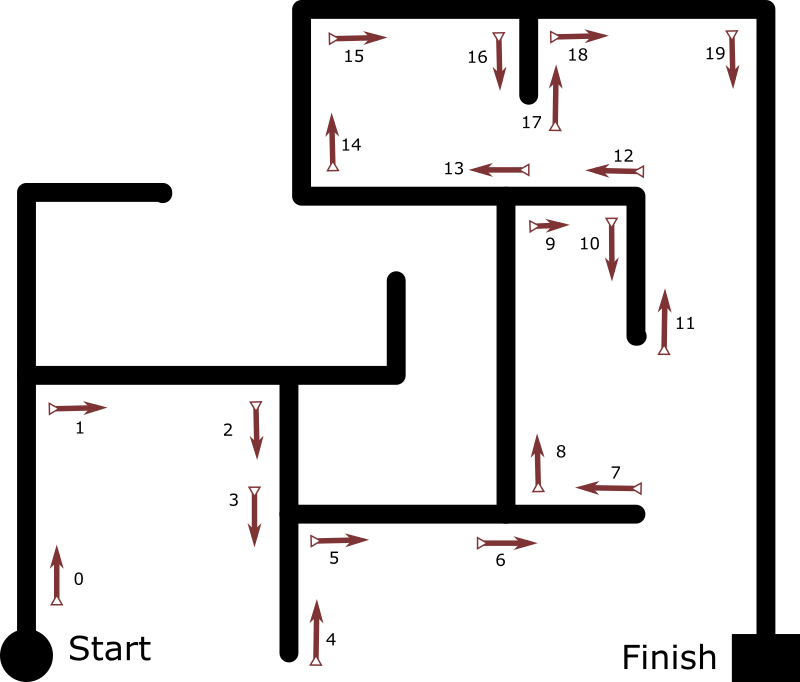
[](http://k3.arduino.vn/img/2018/01/27/0/4000_81257005-1517062999-0-text28620.png)

Ta có chuỗi: **LULRDL**. Bắt đầu rút gọn:

**LU~~LR~~DL** => **L~~UD~~**L => **LL**

Như vậy, ban đầu xe đi về phía bên trái tại số 1. Đến ngã rẽ, xe sẽ bỏ qua để tiếp tục đi về phía trái. Như thế ta đã rút gọn bớt được một ngõ cụt.

**Ví dụ 3:** Đây sẽ là một mê cung hoàn chỉnh đơn giản. Nếu bạn hiểu rồi thì hãy thử tự giải và rút gọn trước khi xem đáp án.

[](http://k3.arduino.vn/img/2018/01/27/0/3999_81257005-1517062995-0-text20204.png)

Giả sử xe dò đường của chúng ta sử dụng thuật toán bám tường phải. Sau một hồi mò mẫm ta có đường đi như hình, tại mỗi điểm rẽ ta đều ghi lại hướng tiếp theo sẽ di chuyển. Chuỗi đường đi sẽ như sau: **URDD** **URRL** **URDU** **LLUR** **DURD** (mình viết cách ra cho dễ nhìn thôi nhé, thực tế chuỗi được ghi liền)

Tiến hành khử các cặp **~~UD~~**, **~~DU~~**, **~~LR~~**, **~~RL~~** cạnh nhau, cho đến khi không còn nữa thì thôi.

**URD~~D~~** **~~U~~R~~RL~~** **UR~~DU~~** **LLUR** **~~DU~~RD => URDR~~RL~~U~~RL~~LURRD => URDRULURRD**

Hãy thử dùng kết quả và dò trên hình, bạn sẽ thấy con đường tới đích đã ngắn hơn rất nhiều.

Vấn đề còn lại cho bạn tự giải quyết, đó là làm thế nào để ghi lại hướng đi tại từng điểm rẽ theo hệ tọa độ gắn với mê cung, làm thế nào để triển khai thuật toán trên dưới dạng code C++ nhằm rút gọn chuỗi đường đi, và cuối cùng để xe tự di chuyển theo lộ trình của chuỗi đã rút gọn.

1. ​// quy chuoi ra man tra
2. ​for (int i = 0; i < size1; i++)
3. ​if (x[i] == 'L'
4. ​a[i] = 3
5. ​else if (x[i] == 'R'
6. ​a[i] = 2
7. ​else if (x[i] == 'D'
8. ​a[i] = 10
9. ​else if (x[i] == 'U'
10. ​a[i] = 1
11. System.***out***.println(a[i])
12. ​
13. ​int i = 0
14. ​while (i < size1)
15. ​int size\_current = size1
16. ​if (a[i] \* a[i + 1] == 10 || a[i] \* a[i + 1] == 6) { // co su rut go
17. ​if (a[i + 2] == size1
18. ​a[i] = a[i + 2]
19. ​else
20. ​for (int j = i; j < size1 - 2; j++)
21. ​a[j] = a[j + 2]
22. ​
23. ​
24. ​// XOA PHAN TU CU THU
25. ​for (int j = size1 - 2; j < size1; j++)
26. ​a[j] = 0
27. ​
28. ​size1 = size1 - 2
29. ​
30. ​// neu k cos su rut gon thi tag i
31. ​if (size\_current == size1
32. ​i++
33. ​
34. ​// doc nguoc tu mang a de ra mang
35. ​for (int j = 0; j < size1; j++)
36. ​if (a[j] == 1
37. ​x[j] = 'U'
38. ​else if (a[j] == 10
39. ​x[j] = 'D'
40. ​else if (a[j] == 3
41. ​x[j] = 'L'
42. ​else if (a[j] == 2
43. ​x[j] = 'R'
44. ​
45. ​x[size1] = '\0'
46. ​ System.***out***.println(" duong di ngan nhat ");
47. ​return 0
48. }