ИННОТЕХ

АЛГОРИТМ ДЕЙКСТРЫ + 2D-ИГРА = ПРОСТО

Александр Пономаренко

АЛЕКСАНДР ПОНОМАРЕНКО

ГК «Иннотех», Frontend-разработчик React

В web-разработке 20 лет

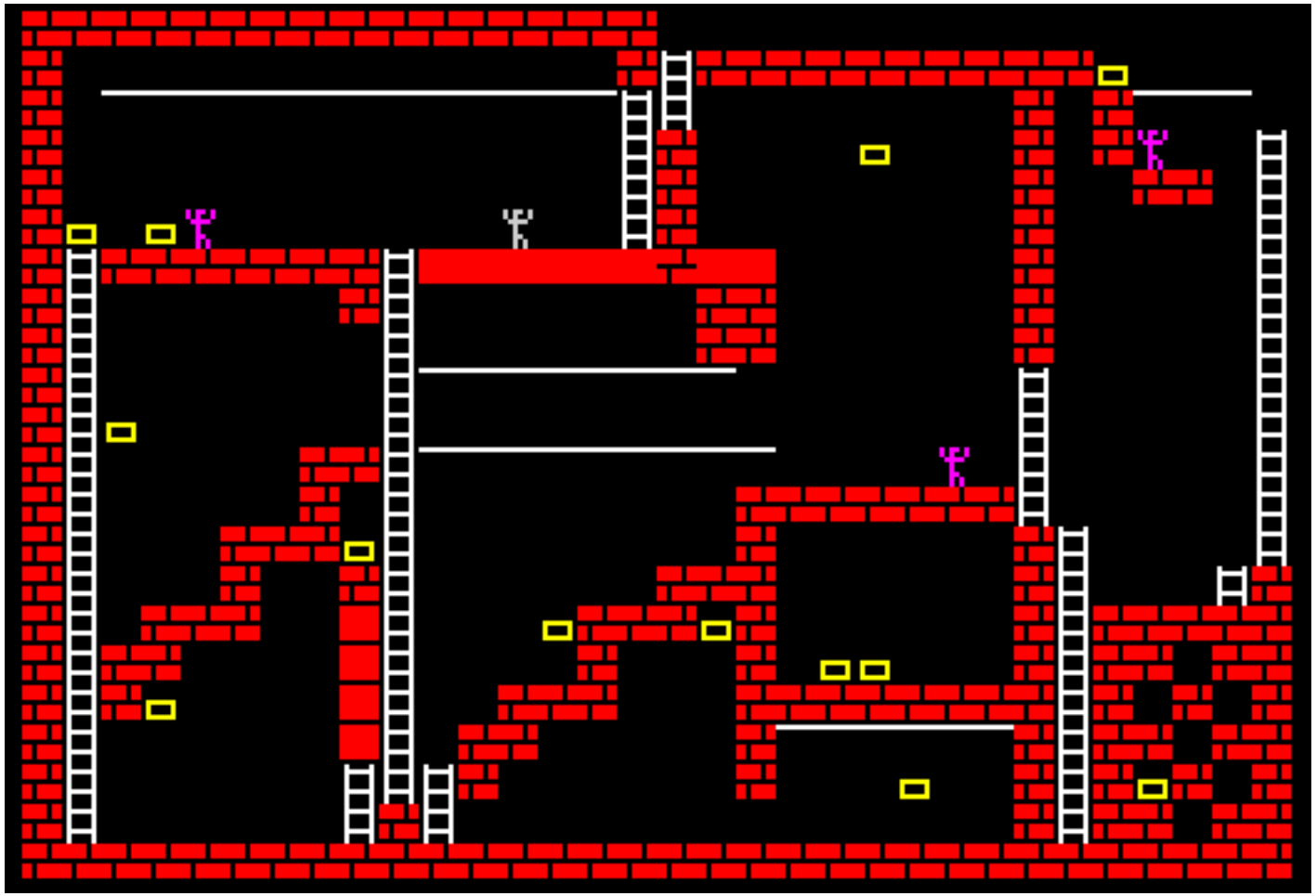
Создал 5 игр в рамках обучающих проектов для школьников

ИСТОЧНИК ВДОХНОВЕНИЯ

Игра «Lode runner»

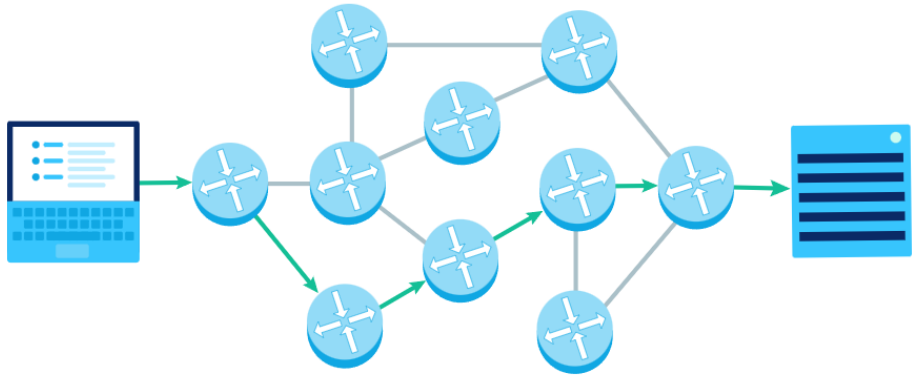
для ZX Spectrum

Broderbund 1984



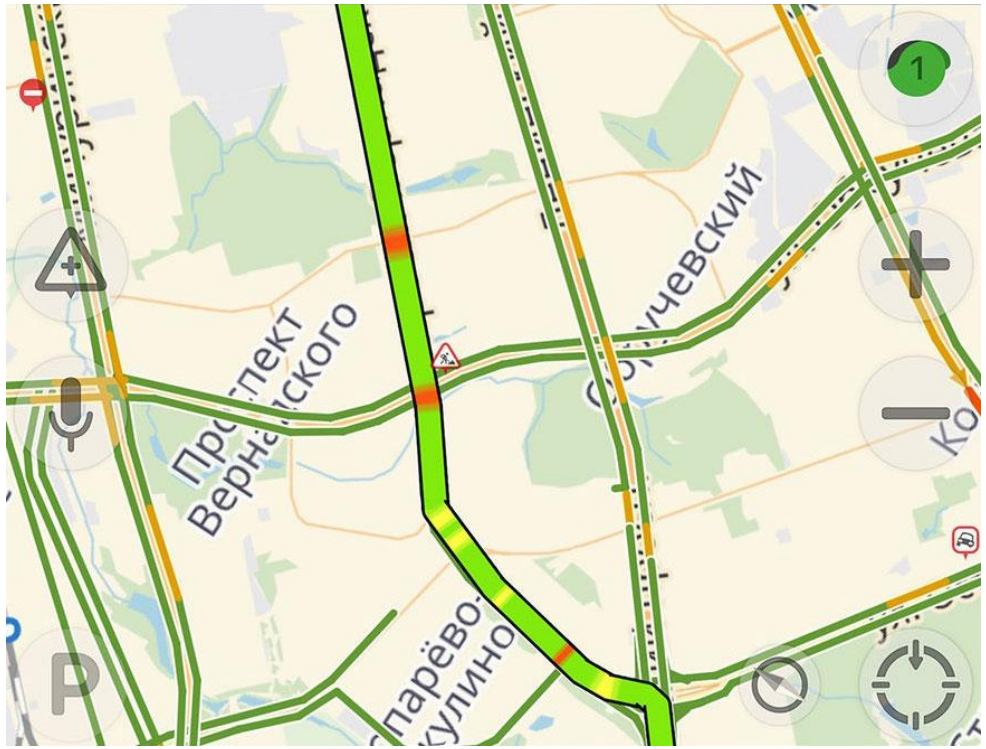
ЕСЛИ ВЫ — IT-СПЕЦИАЛИСТ И ВАША ЗАДАЧА:

- Маршрутизировать трафик по сети



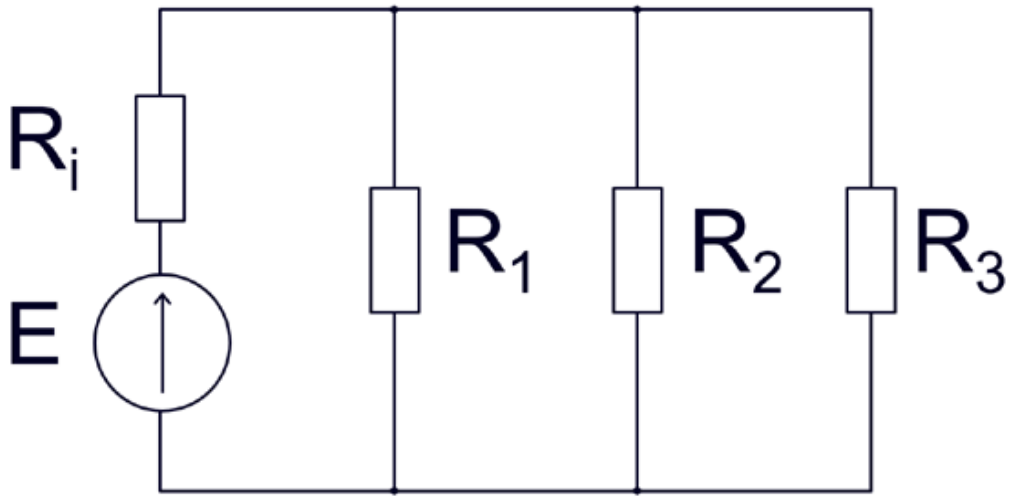
ЕСЛИ ВЫ — IT-СПЕЦИАЛИСТ И ВАША ЗАДАЧА:

- Маршрутизировать трафик по сети
- Или создать программу-навигатор, который будет находить кратчайший путь из точки А в точку Б



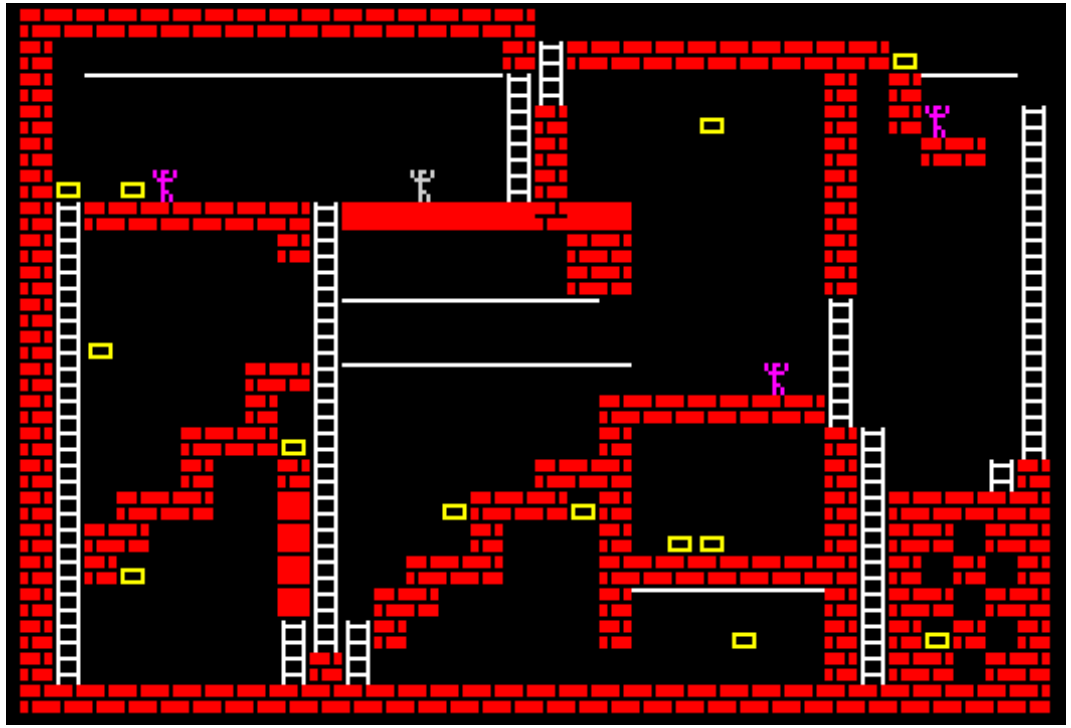
ЕСЛИ ВЫ — IT-СПЕЦИАЛИСТ И ВАША ЗАДАЧА:

- Маршрутизировать трафик по сети
- Или создать программу-навигатор, который будет находить кратчайший путь из точки А в точку Б
- Или создать программу расчета электрических цепей



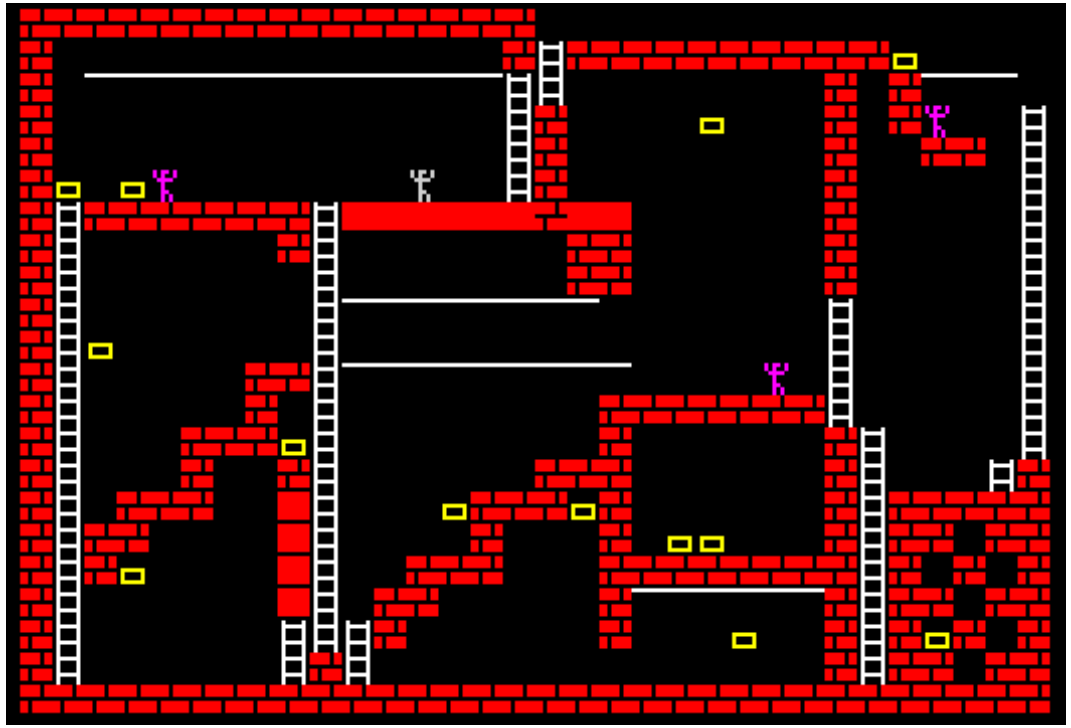
ЕСЛИ ВЫ — IT-СПЕЦИАЛИСТ И ВАША ЗАДАЧА:

- Маршрутизировать трафик по сети
- Или создать программу-навигатор, который будет находить кратчайший путь из точки А в точку Б
- Или создать программу расчета электрических цепей
- Или создать компьютерную игру —



ЕСЛИ ВЫ — IT-СПЕЦИАЛИСТ И ВАША ЗАДАЧА:

- Маршрутизировать трафик по сети
- Или создать программу-навигатор, который будет находить кратчайший путь из точки А в точку Б
- Или создать программу расчета электрических цепей
- Или создать компьютерную игру — **то этот доклад для вас**



ЧЕРЕЗ 40 МИНУТ

ИННОТЕХ

Вы будете знать:

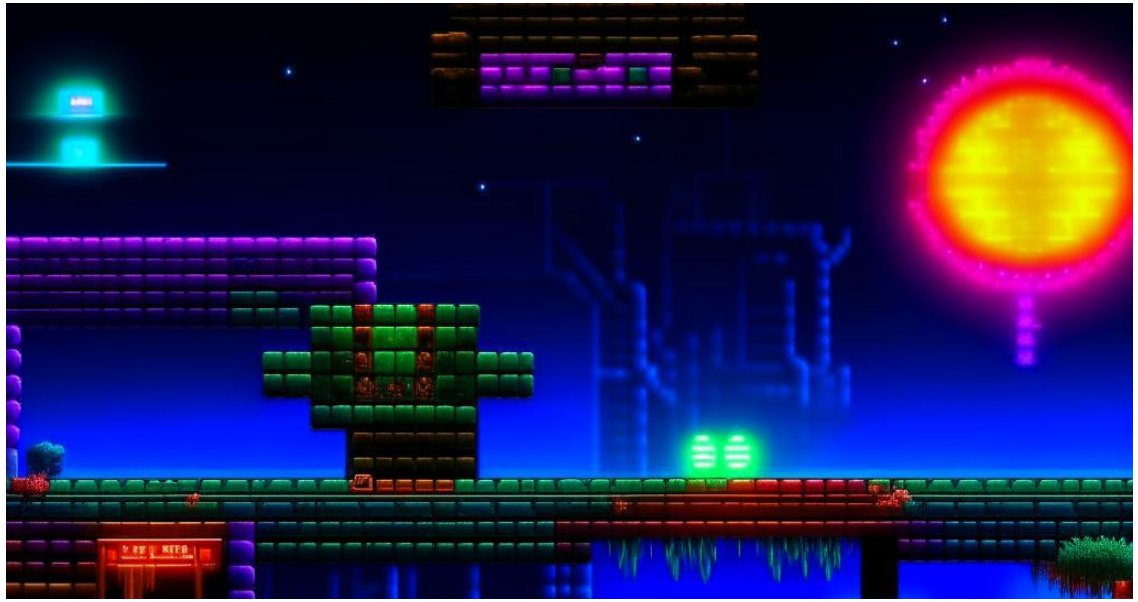
- Как реализовать алгоритм Дейкстры
- Как на его базе создать компьютерную игру, где персонаж обходит препятствия и движется к цели

У вас будет ссылка:

- На репозиторий с исходниками алгоритма и игры электрических цепей
- На работающую игру в интернете

ВЫ СМОЖЕТЕ

- Сами создавать собственные игры
- И решать задачи поиска оптимального пути, опираясь на представленный пример

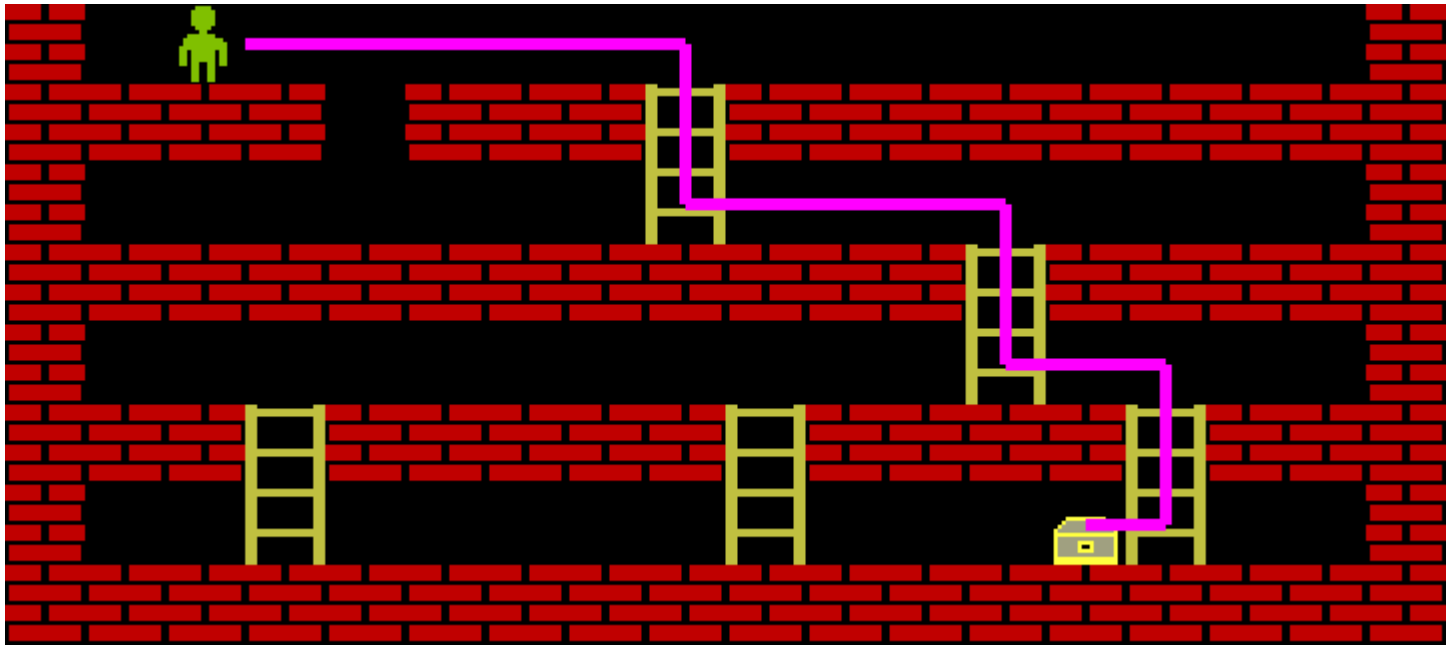


ВЫ СЭКОНОМИТЕ 1 НЕДЕЛЮ СВОЕГО ВРЕМЕНИ

ИННОТЕХ

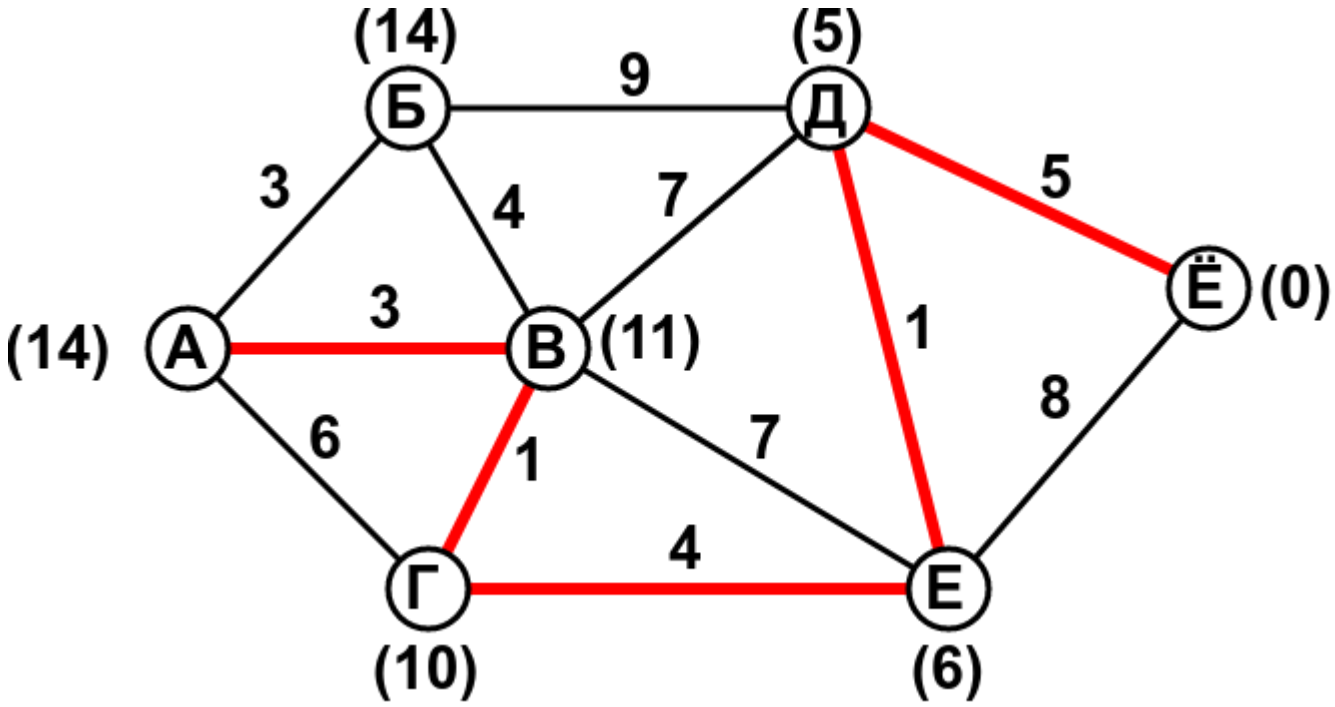
Потому что:

- Предложу вам готовую реализацию алгоритма
- Расскажу о подводных камнях



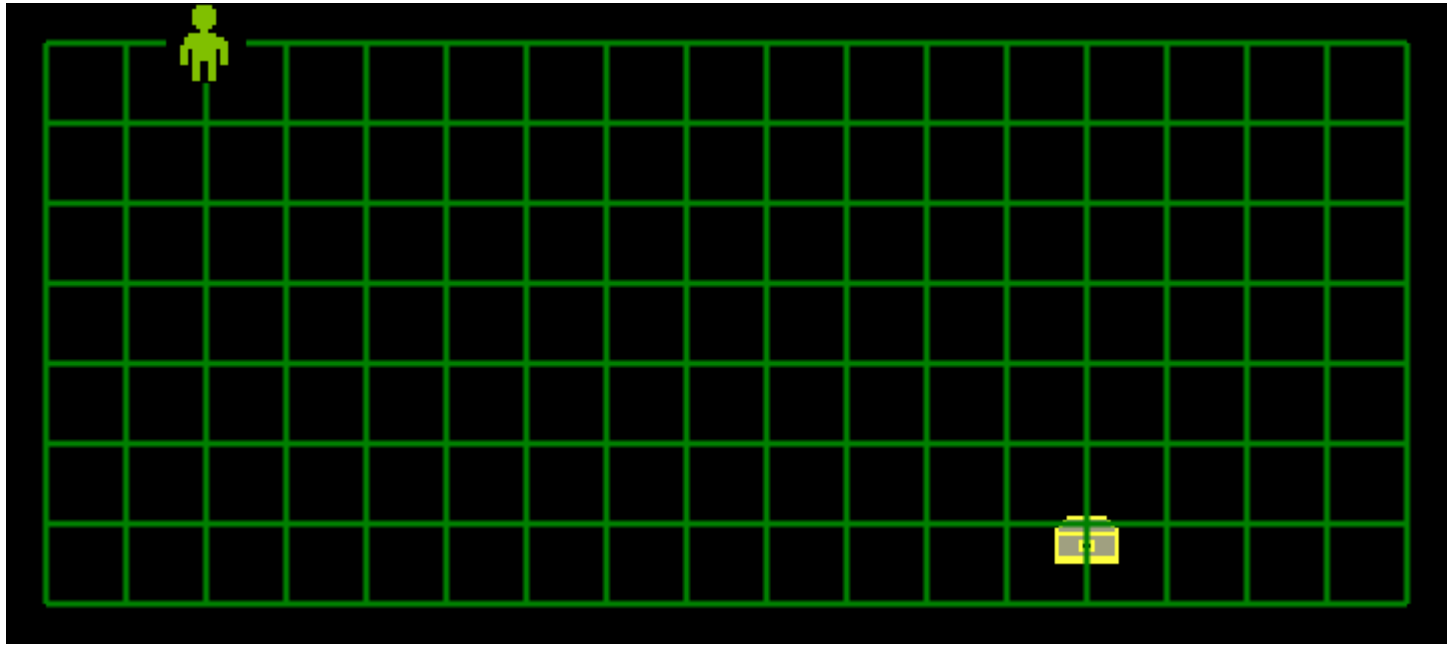
В ДОКЛАДЕ МЫ:

1. Применим алгоритм Дейкстры для поиска кратчайшего пути на графе



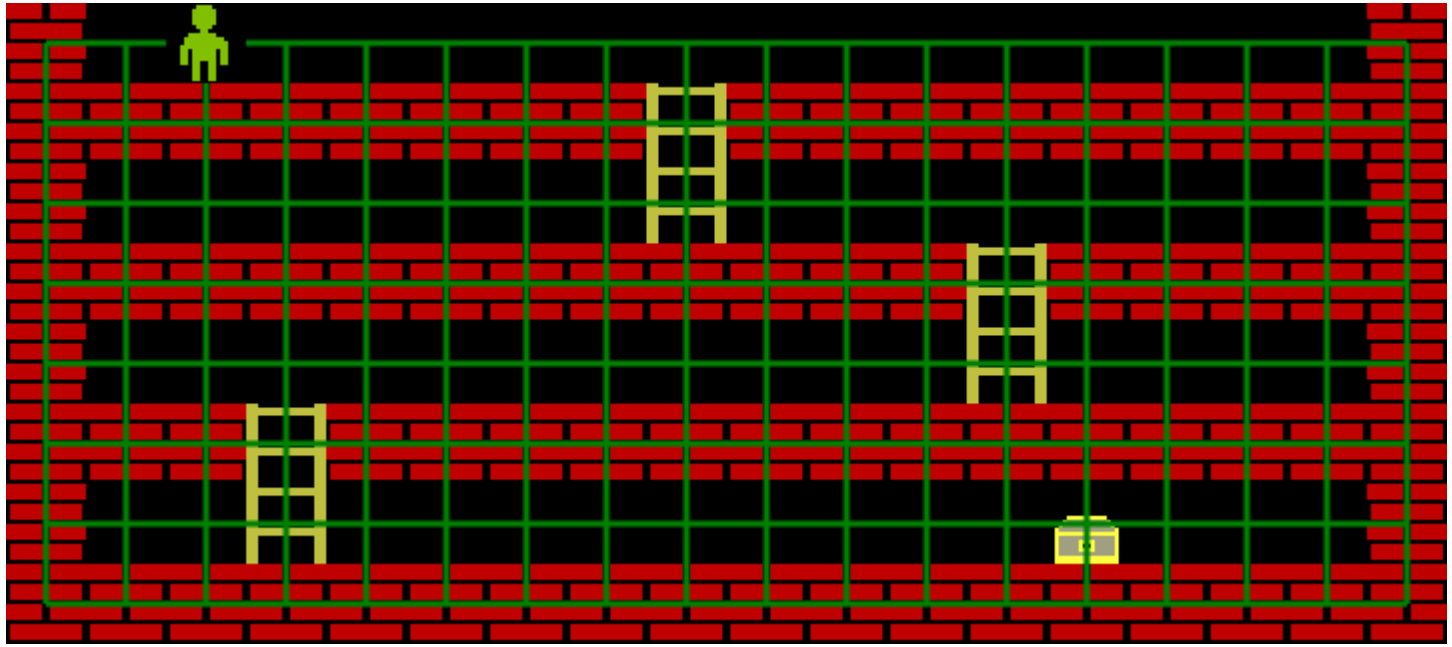
В ДОКЛАДЕ МЫ:

2. Преобразуем граф
   в квадратную сетку



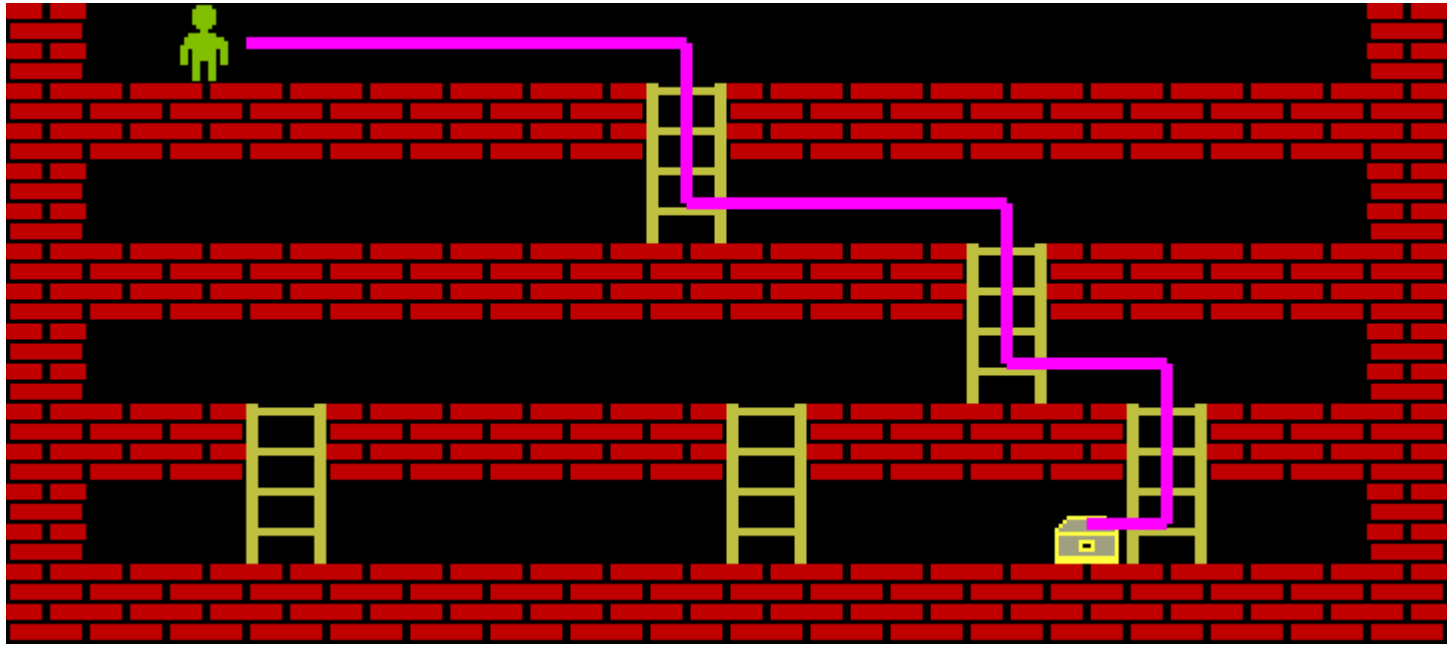
В ДОКЛАДЕ МЫ:

3. Наложим квадратную сетку на карту уровня 2D-игры



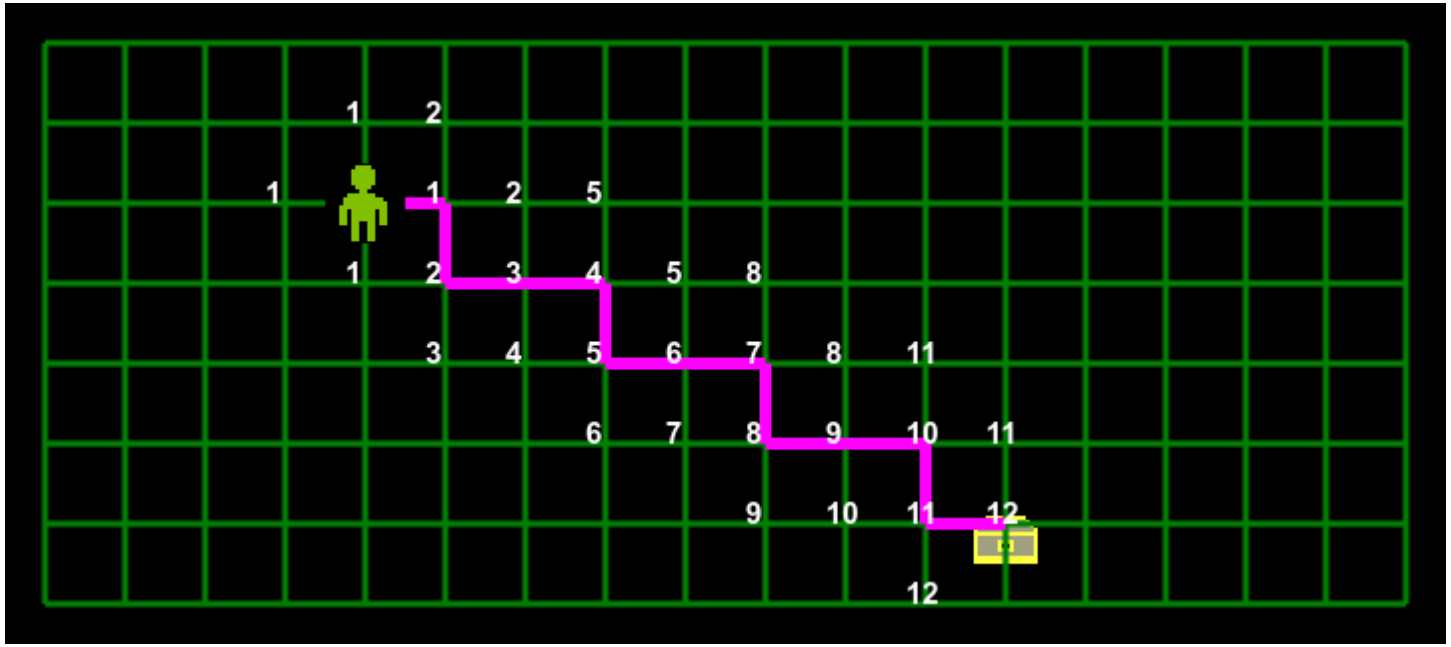
В ДОКЛАДЕ МЫ:

Найдем кратчайший путь, добавляя на карту разные элементы



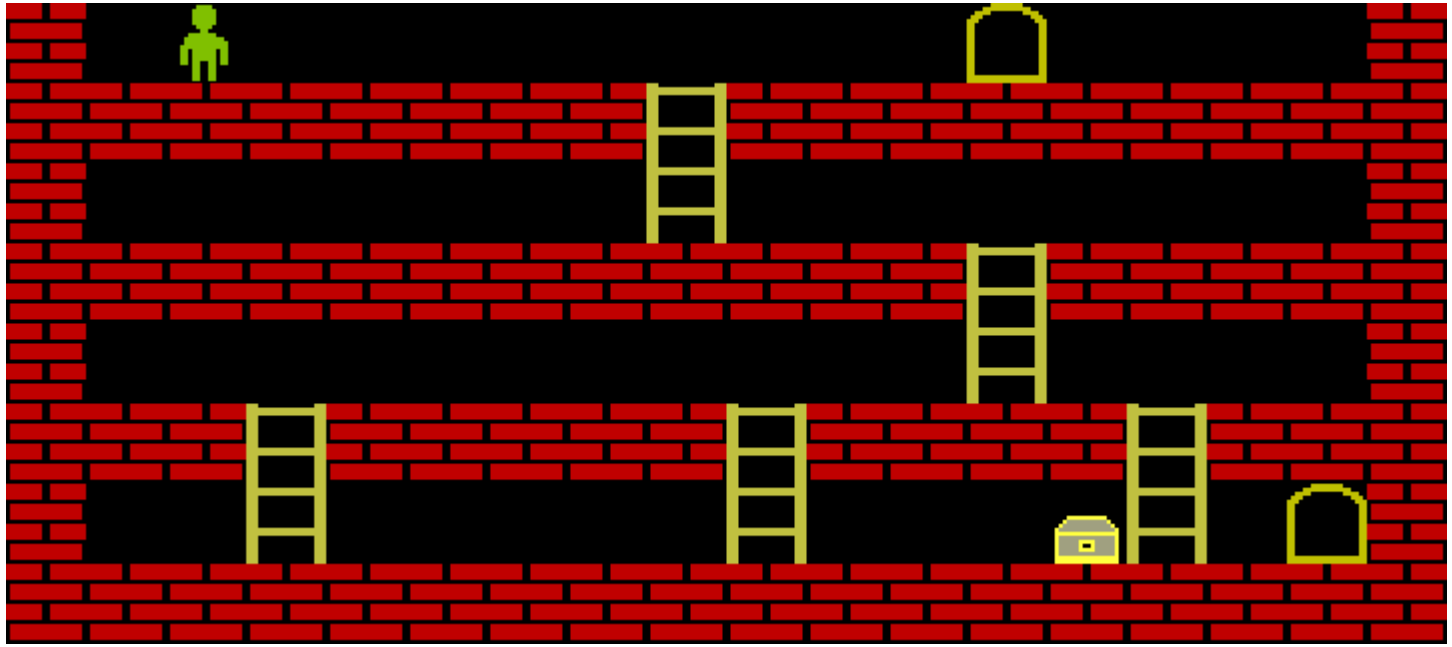
В ДОКЛАДЕ МЫ:

4. Оптимизируем алгоритм Дейкстры и получим вариант A*



В ДОКЛАДЕ МЫ:

5. Решим задачу телепортации персонажа



В ДОКЛАДЕ МЫ:

6. Сравним быстродействие алгоритмов Дейкстры и A*

Алгоритм Дейкстры

N, узлов	T, сек
…	…
…	…
…	…

A*

N, узлов	T, сек
…	…
…	…
…	…

СОДЕРЖАНИЕ

1. АЛГОРИТМ ДЕЙКСТРЫ

Алгоритм Дейкстры — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году.

Находит кратчайшие пути от одной из вершин графа до всех остальных

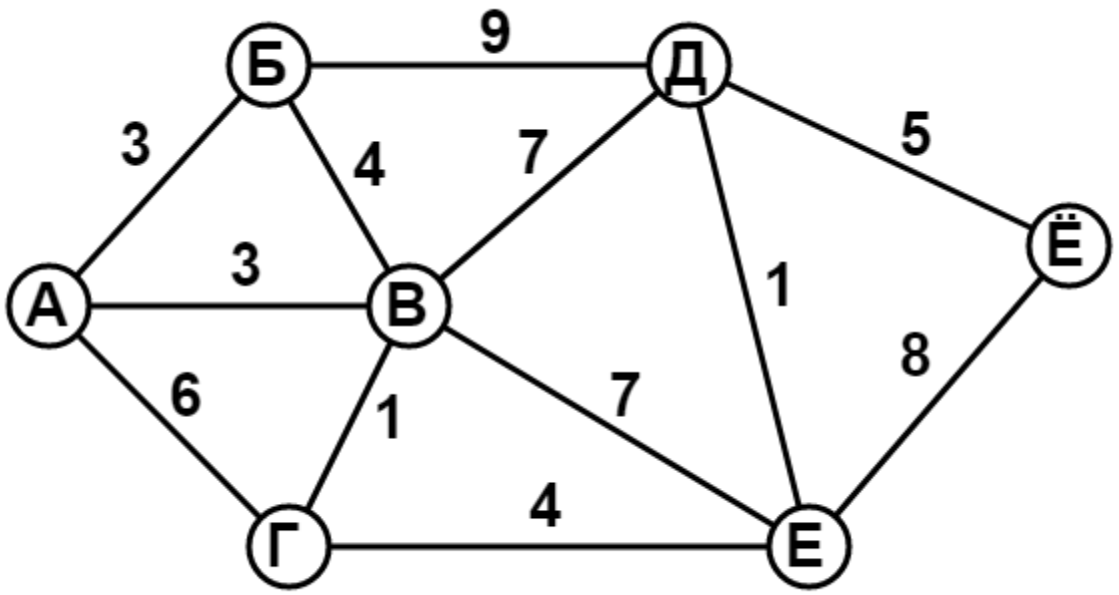
1. ГДЕ МОЖЕТ ПРИГОДИТЬСЯ?

- Маршрутизация в компьютерных сетях
- Программы-навигаторы: поиск маршрута, чтобы быстрее добраться из точки А в точку Б
- В играх (вычисление оптимальной траектории движения)
- Расчет электрических цепей: ток течет по пути наименьшего сопротивления
- ...

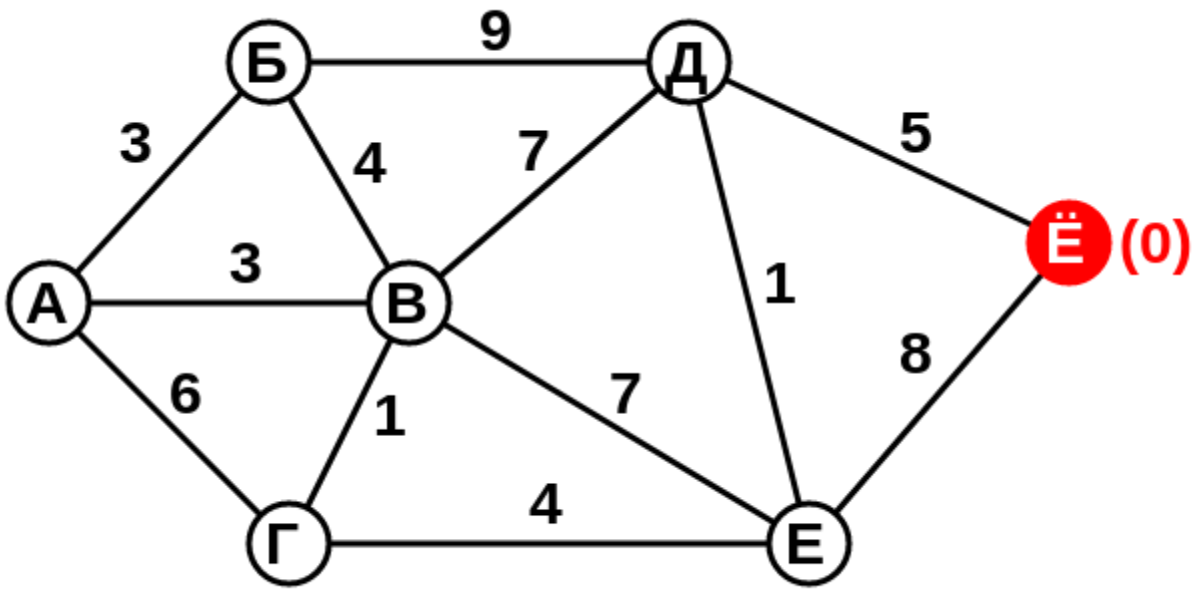
1. ОСНОВНЫЕ МОМЕНТЫ

- Внешний цикл – обходим все вершины графа
- Внутренний цикл – обходим соседние вершины относительно текущей
- Расчет стоимости перехода в соседнюю вершину из текущей
- При расчете стоимости запоминаем, по какому ребру перешли в данную вершину

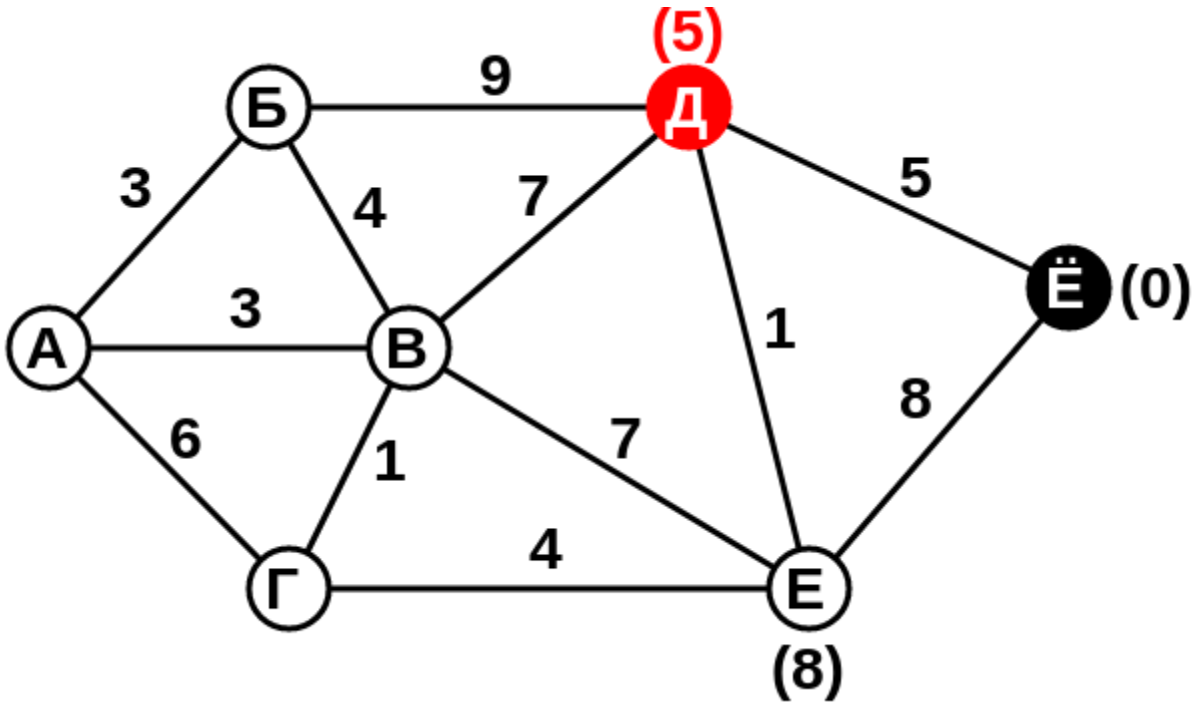
1. ИСХОДНЫЙ ГРАФ



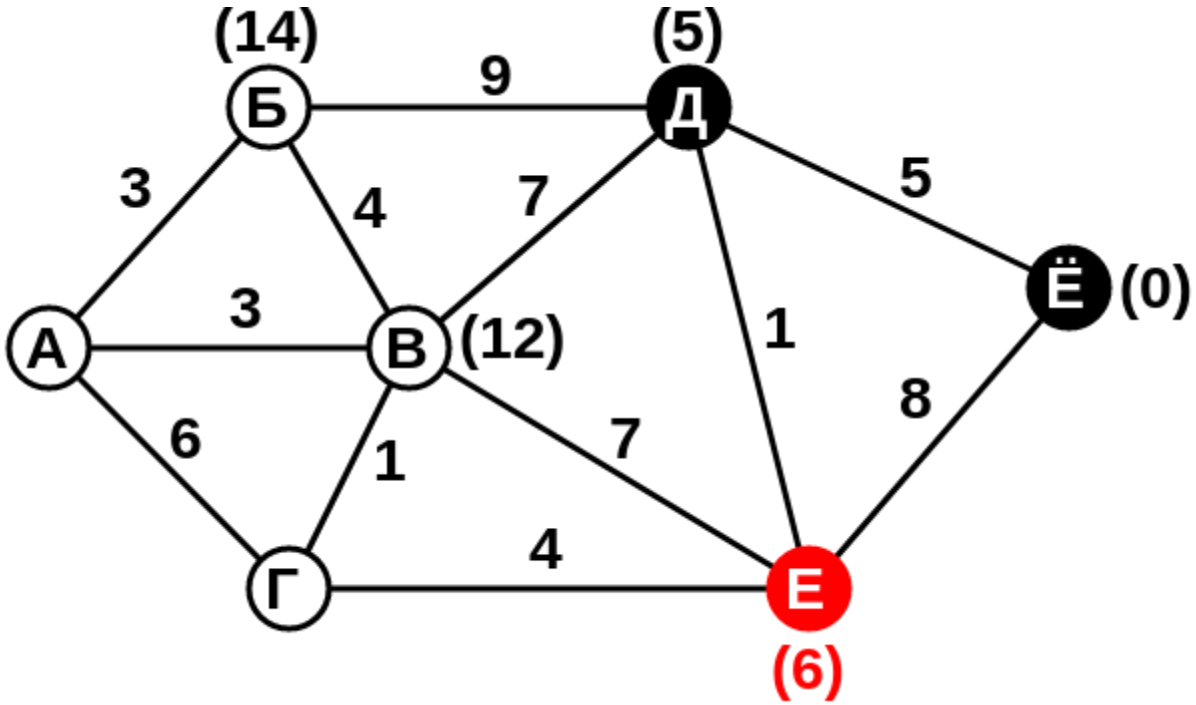
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 0



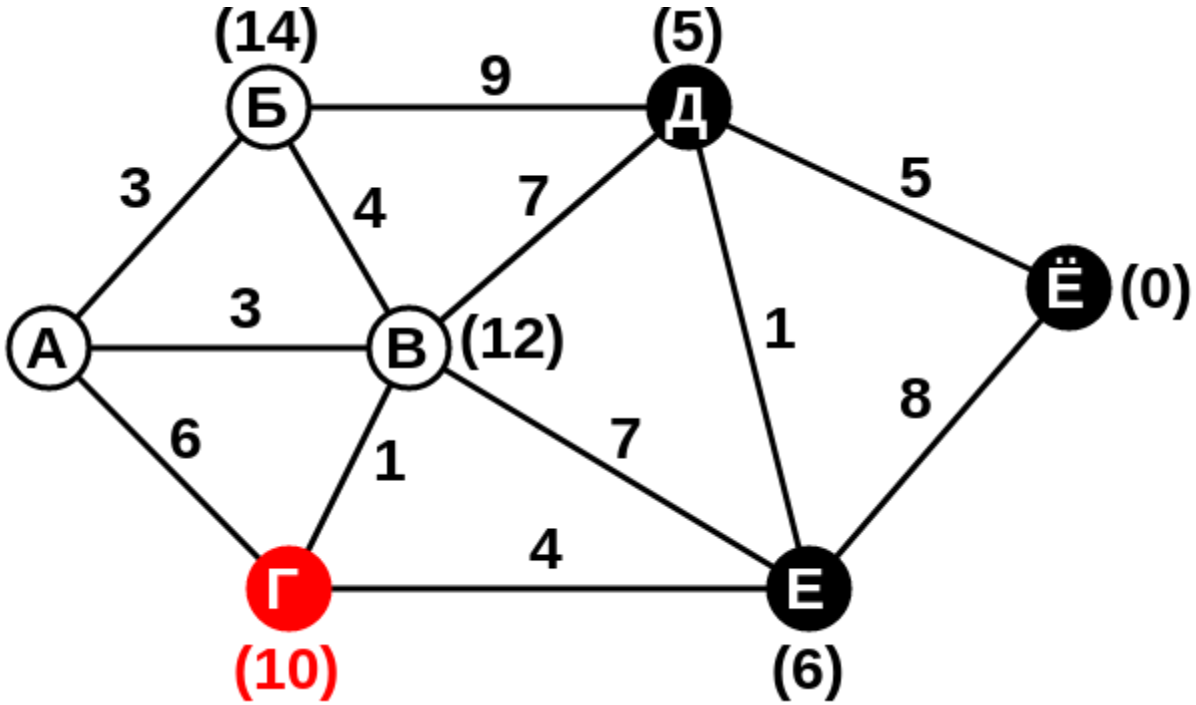
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 1



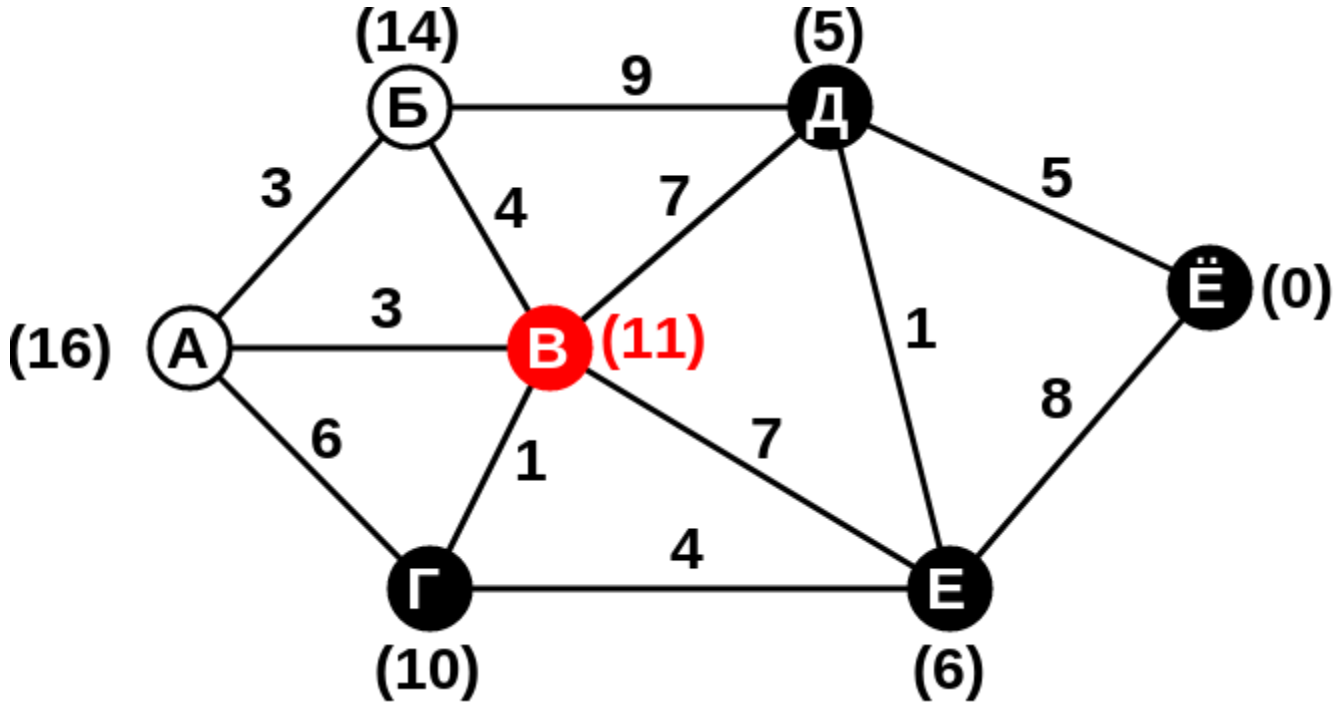
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 2



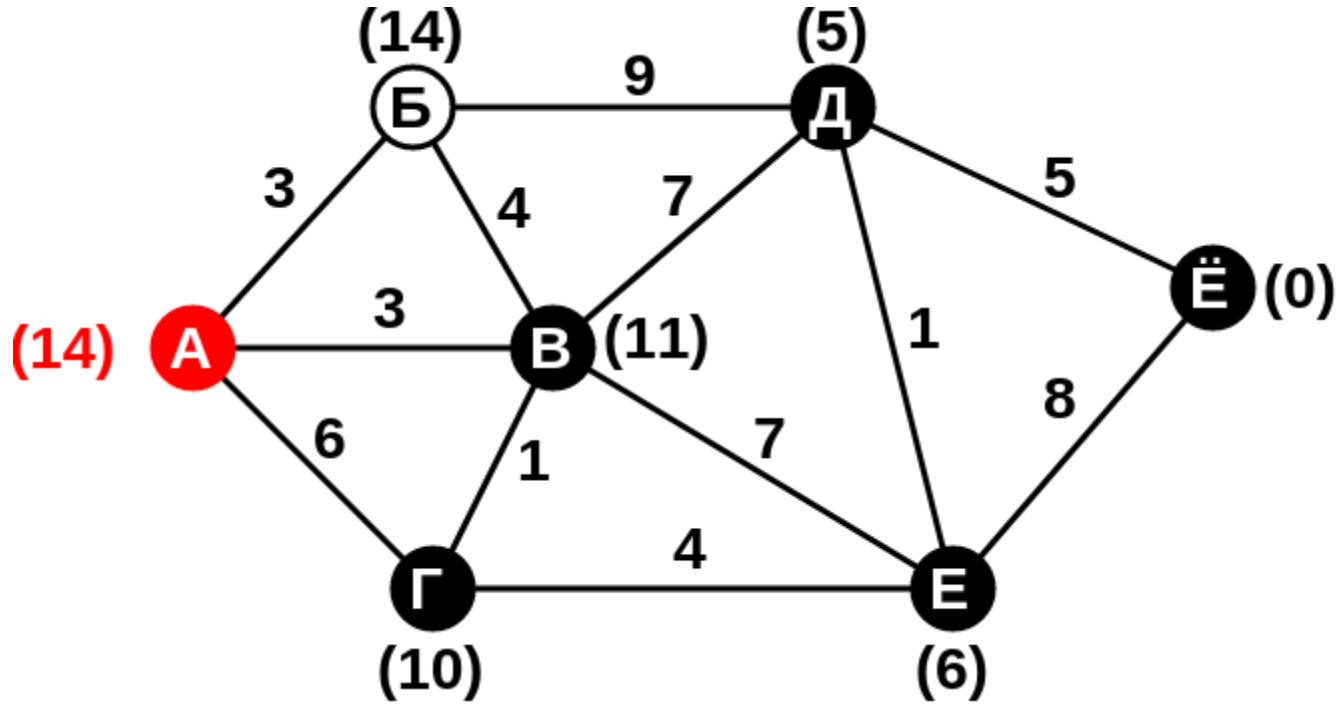
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 3



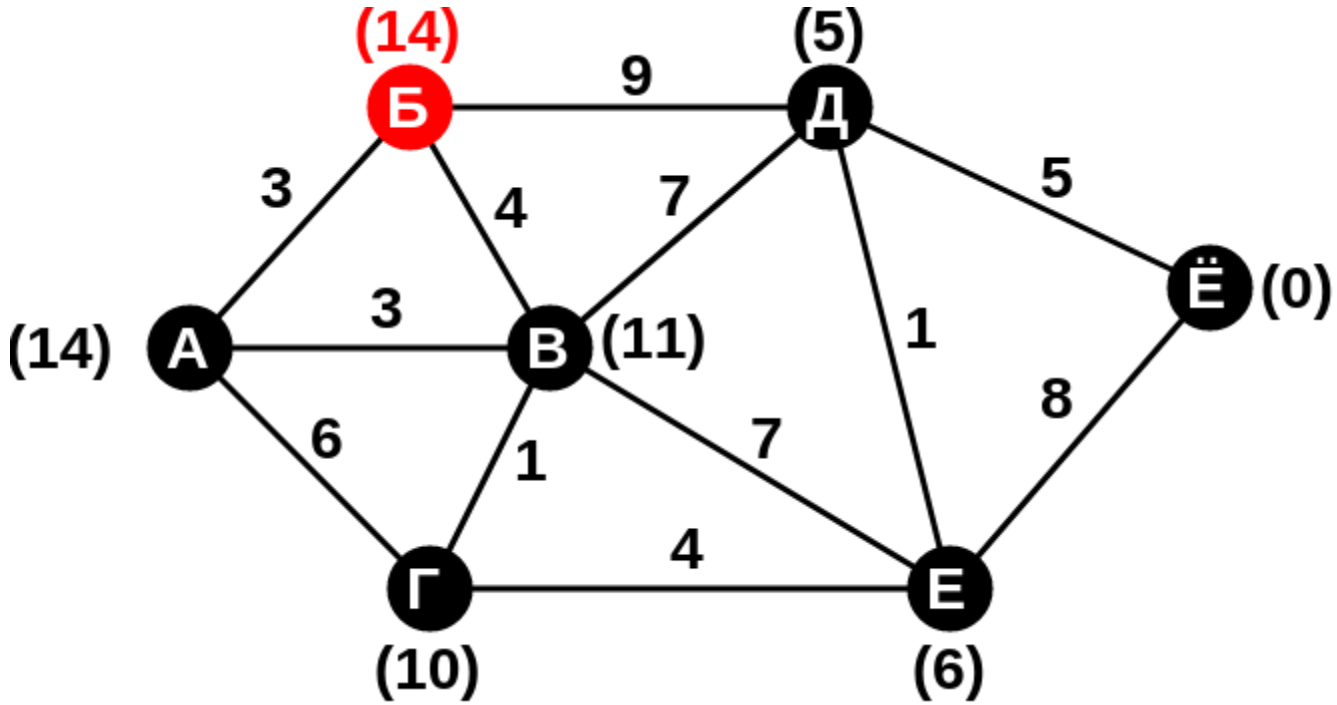
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 4



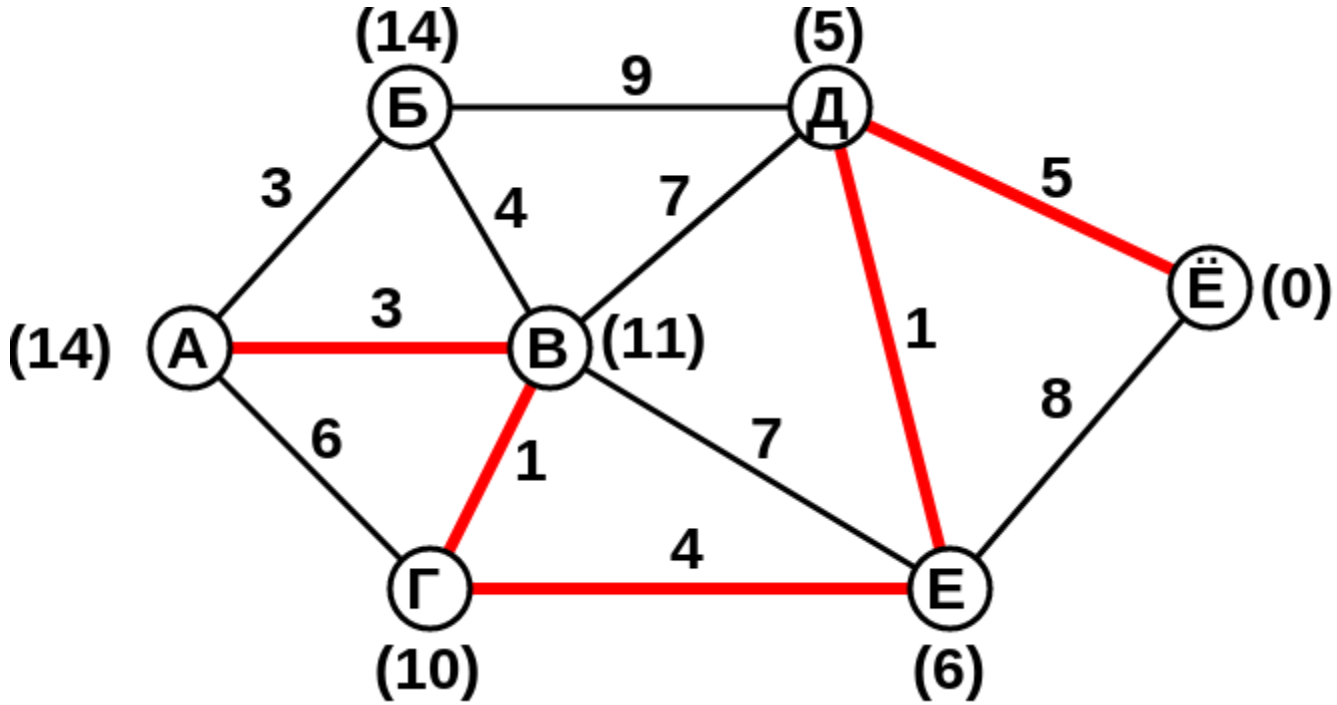
1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 5



1. АЛГОРИТМ ДЕЙКСТРЫ. ШАГ 6



1. КРАТЧАЙШИЙ ПУТЬ



1. АЛГОРИТМ ДЕЙКСТРЫ

1. ВНЕШНИЙ И ВНУТРЕННИЙ ЦИКЛЫ

ИННОТЕХ

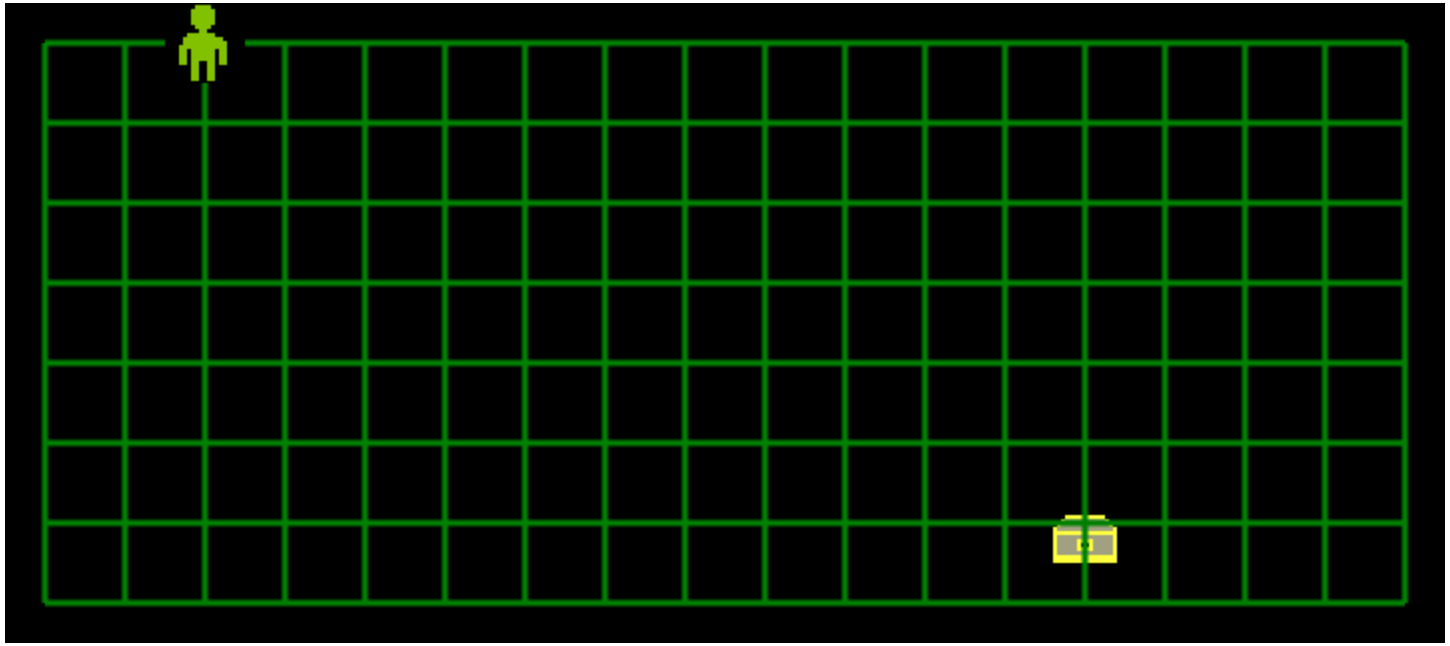
```
01. calcVerticesCost = (fromVertex: number) => {
02.     this.vertices[fromVertex].accessCost = 0;
03.     this.curVertexIndex = fromVertex;
04.     while (this.curVertexIndex !== -1) {
05.         const curVertex = this.vertices[this.curVertexIndex];
06.         this.vertices[this.curVertexIndex].processed = true;
07.         const edgesOfVertex = this.getEdgesOfVertex();
08.         for (let i = 0; i < edgesOfVertex.length; i++) { ... }
09.         const nextVertex = this.getNextVertex(edgesOfVertex);
10.         this.curVertexIndex = nextVertex;
11.     }
12. };
```

1. ВНУТРЕННИЙ ЦИКЛ

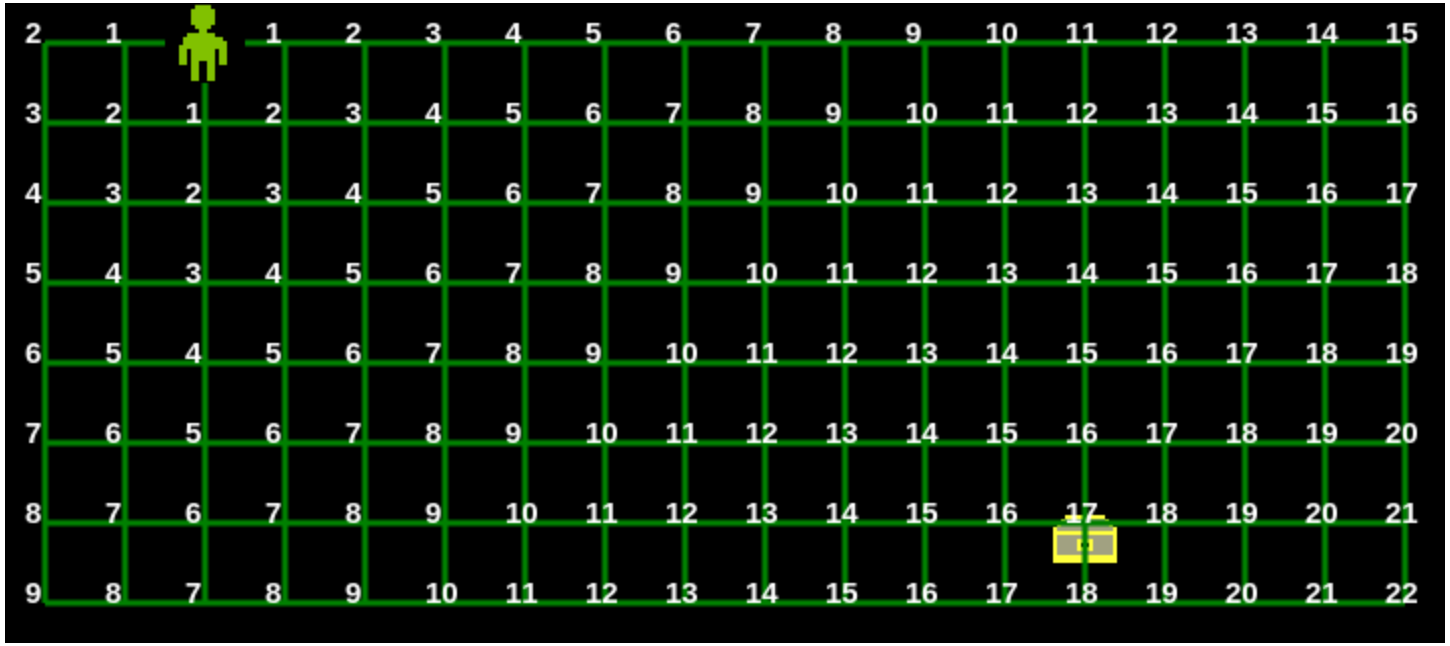
```
01. for (let i = 0; i < edgesOfVertex.length; i++) {
02.     const edgeIndex = edgesOfVertex[i];
03.     const adjVIndex = this.getAdjancedVIndex(edgeIndex);
04.     const adjVertex = this.vertices[adjacentVIndex];
05.     if (adjacentVertex.processed) {
06.         continue;
07.     }
08.     this.updateAccCostAndEdgeIdx(adjVertex, curVertex,
09.         edgeIndex);
10. }
```

СОДЕРЖАНИЕ

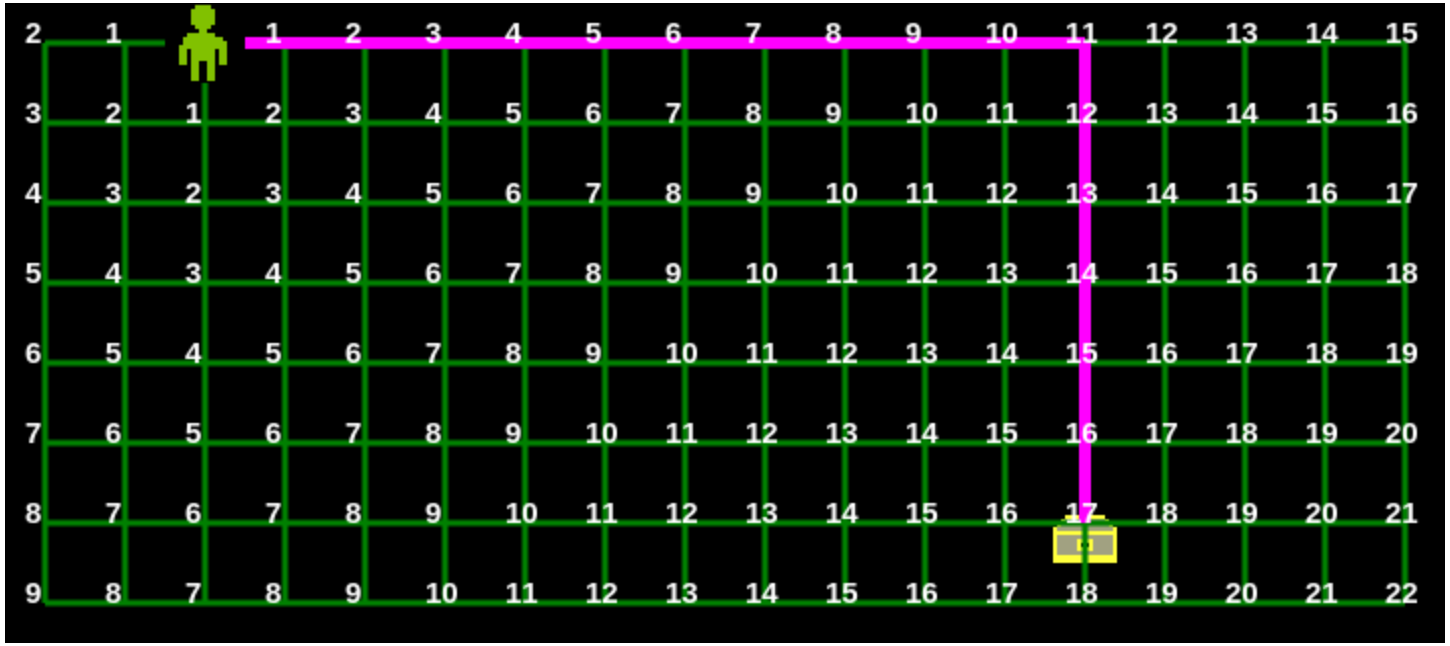
2. КВАДРАТНАЯ СЕТКА



2. КВАДРАТНАЯ СЕТКА

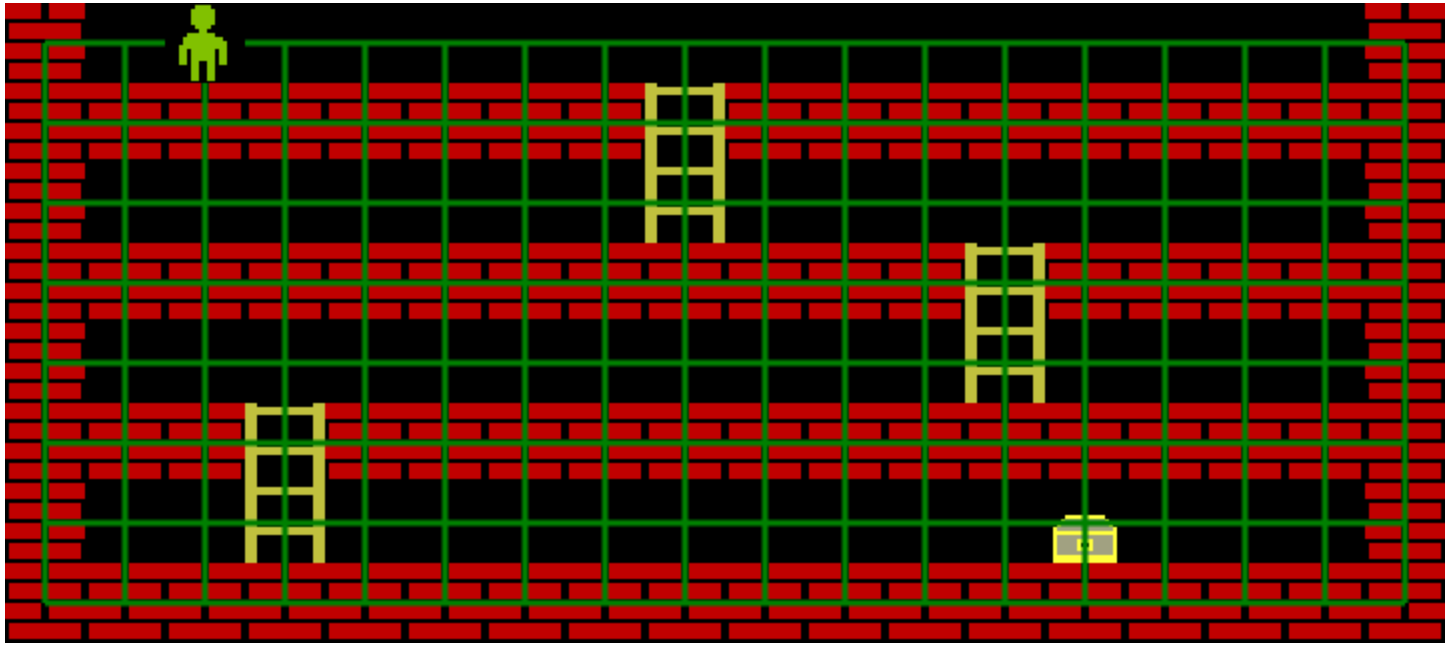


2. КВАДРАТНАЯ СЕТКА

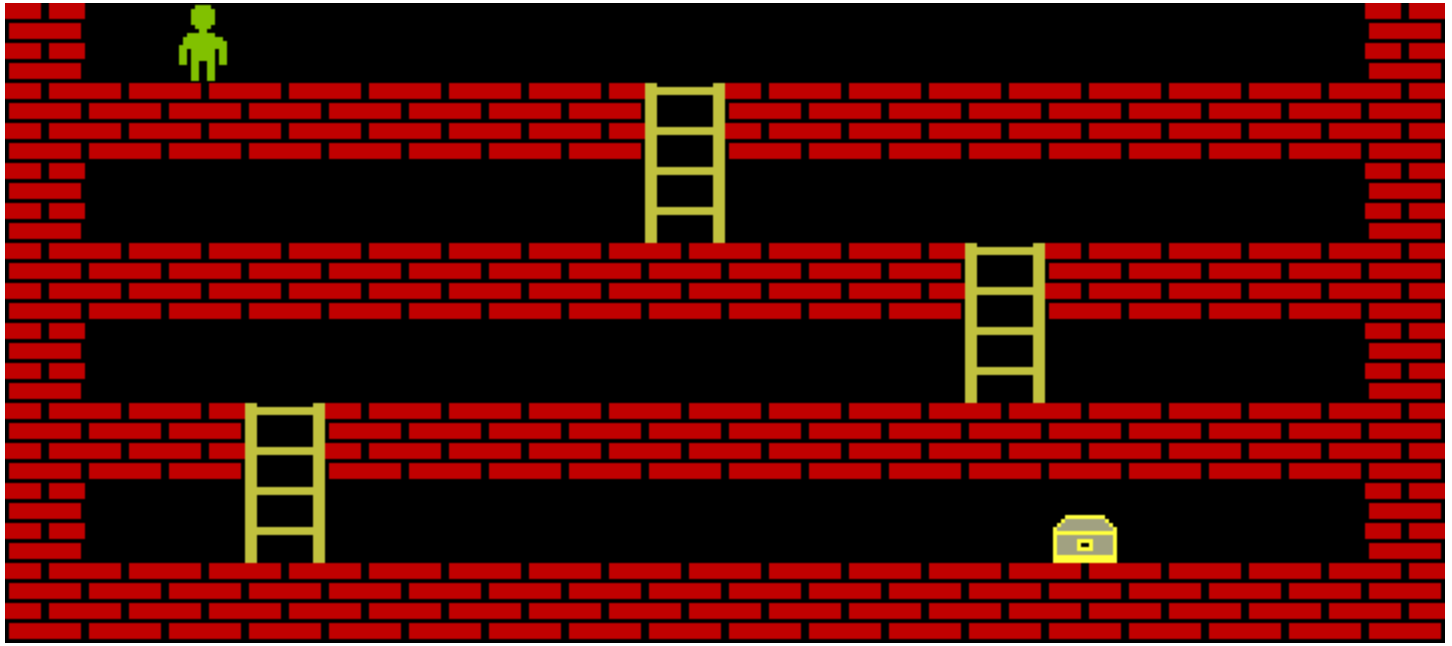


СОДЕРЖАНИЕ

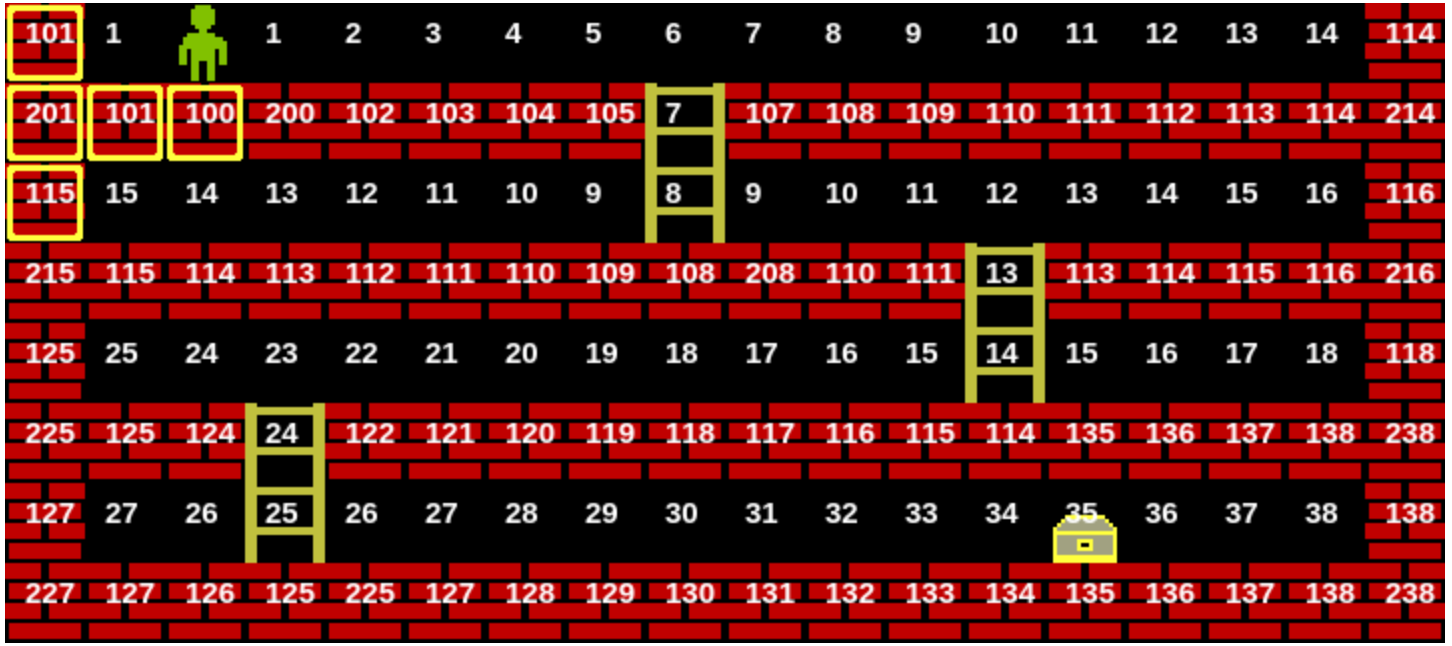
3. ОБХОД ПРЕПЯТСТВИЙ



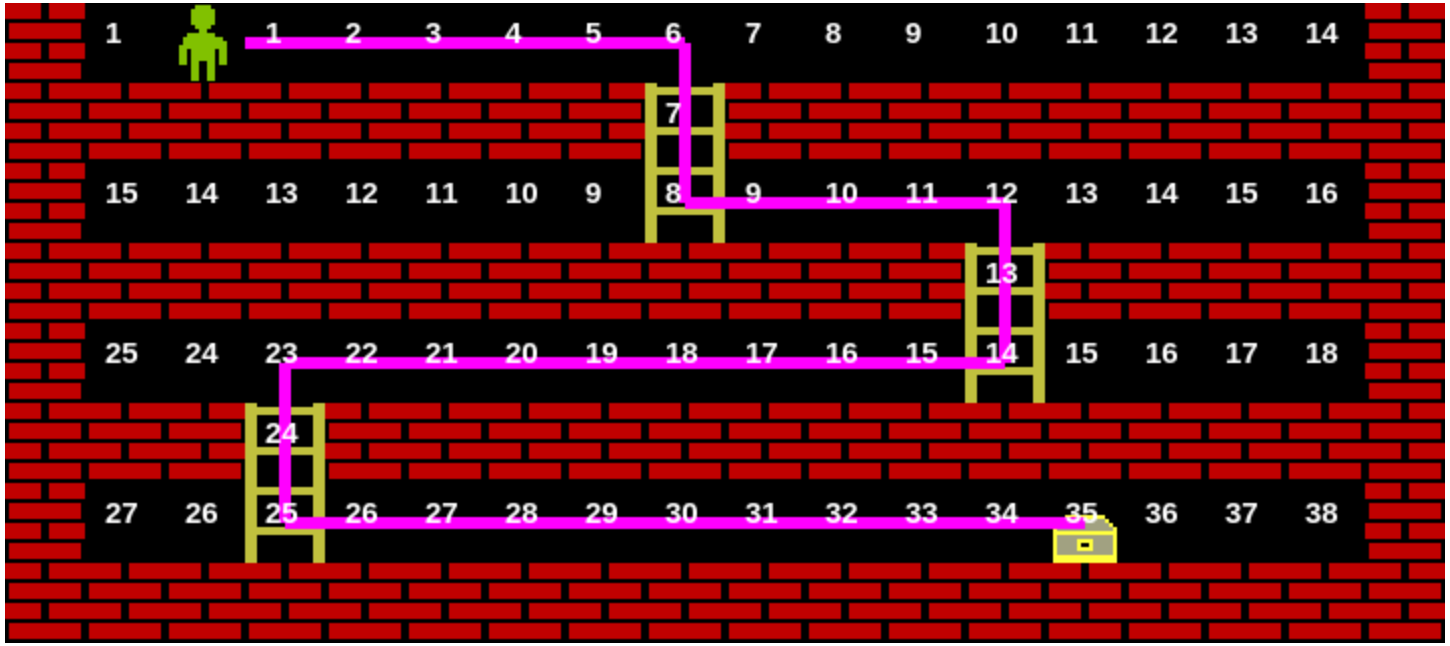
3. ОБХОД ПРЕПЯТСТВИЙ



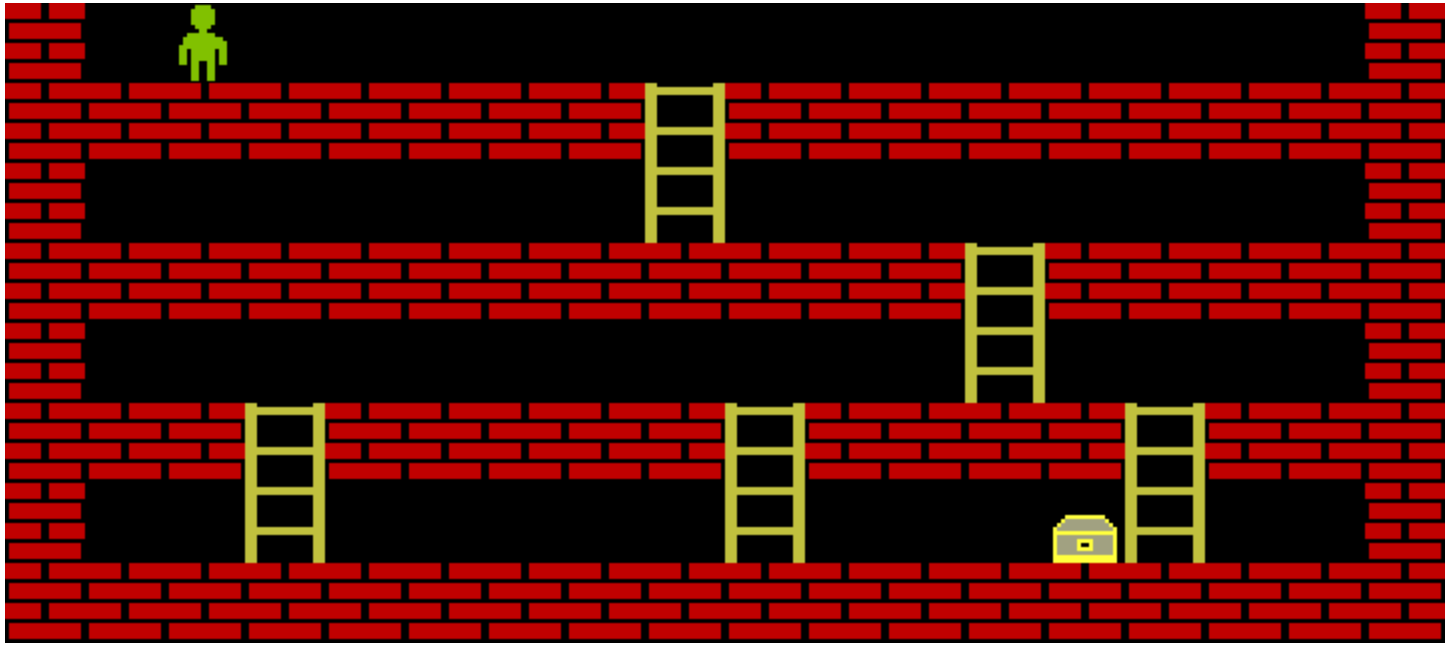
3. ОБХОД ПРЕПЯТСТВИЙ



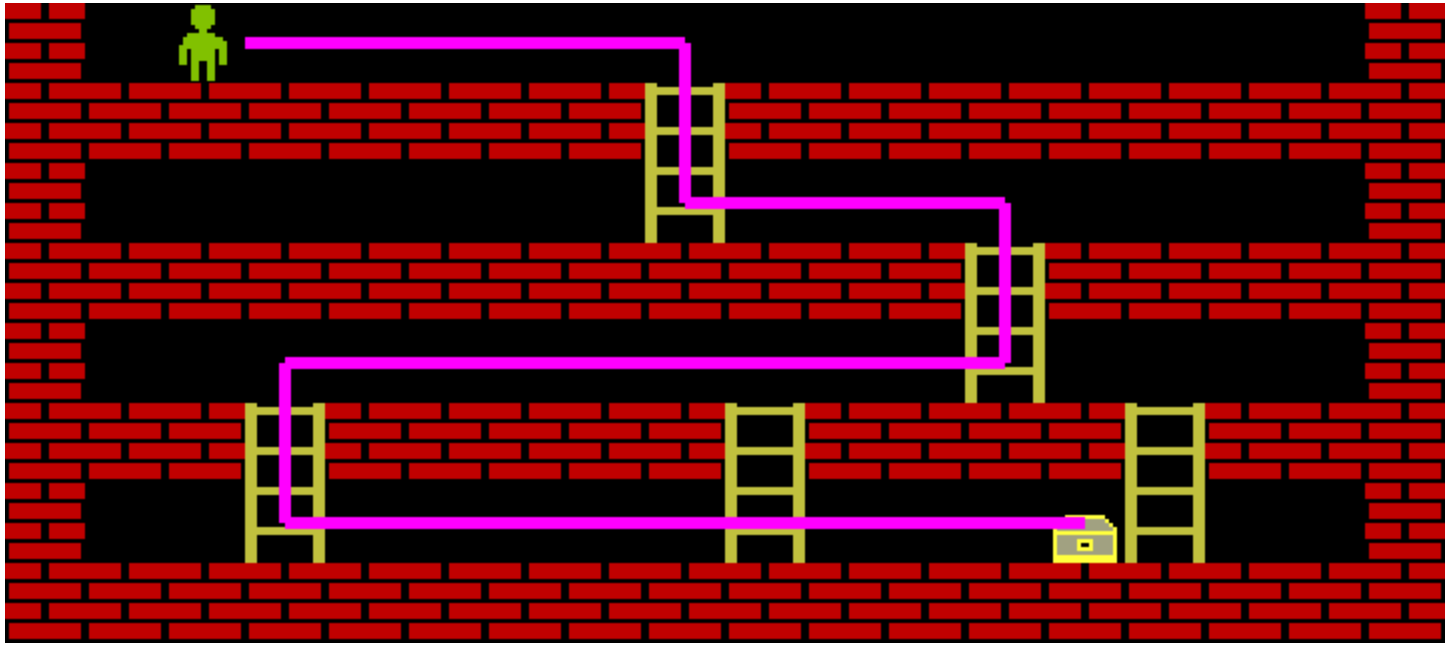
3. ОБХОД ПРЕПЯТСТВИЙ



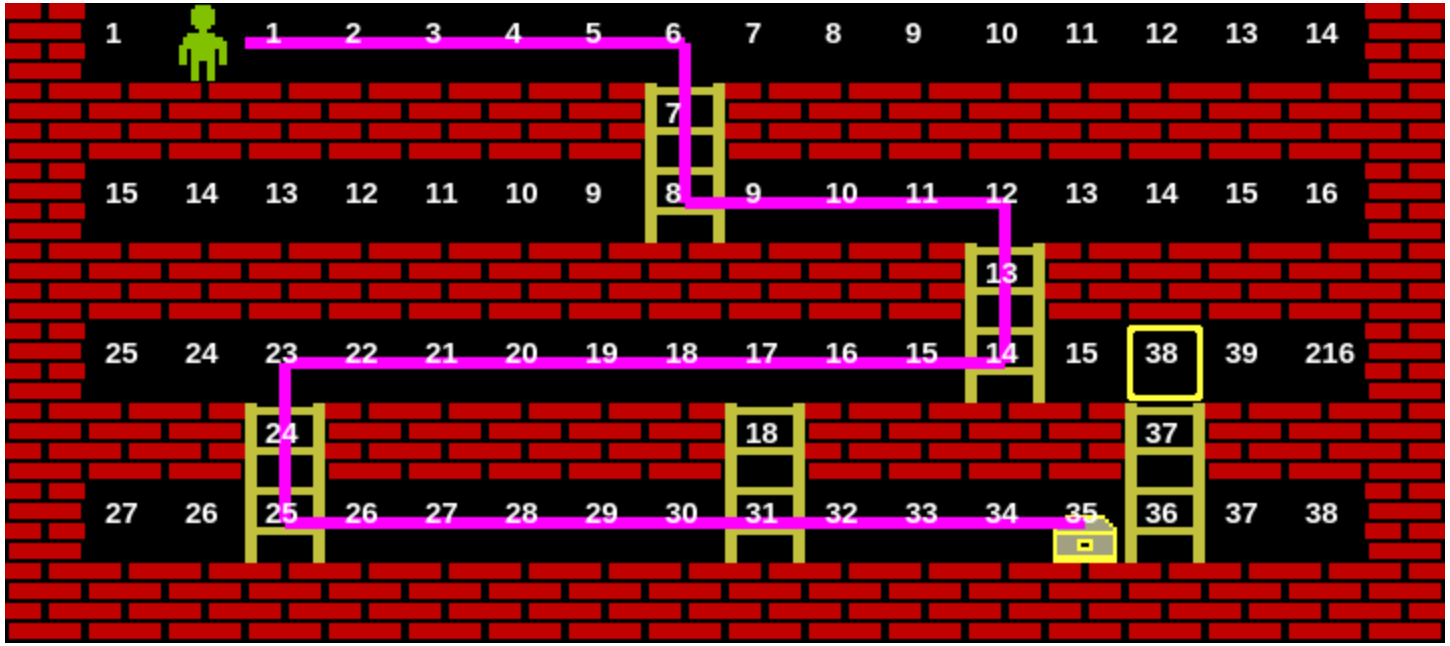
3. ОШИБКА РЕАЛИЗАЦИИ



3. ОШИБКА РЕАЛИЗАЦИИ



3. ОШИБКА РЕАЛИЗАЦИИ

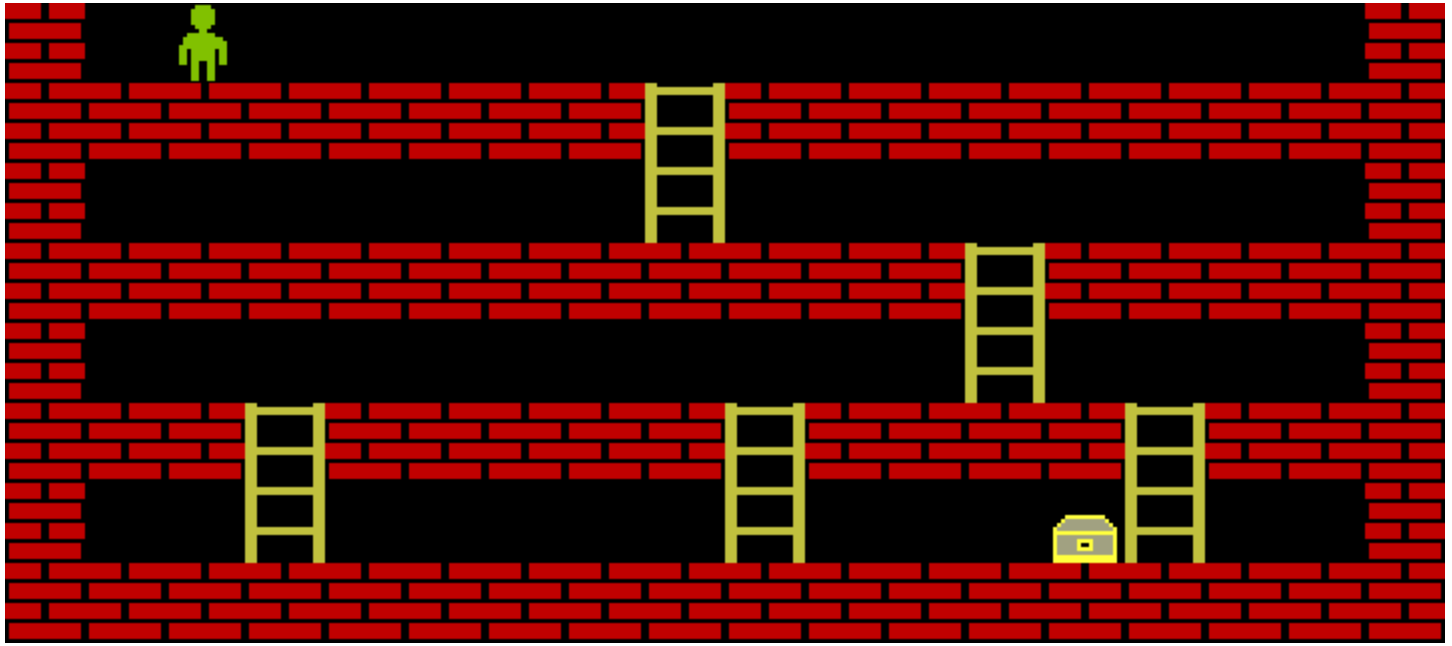


3. ИСПРАВЛЕННАЯ ФУНКЦИЯ

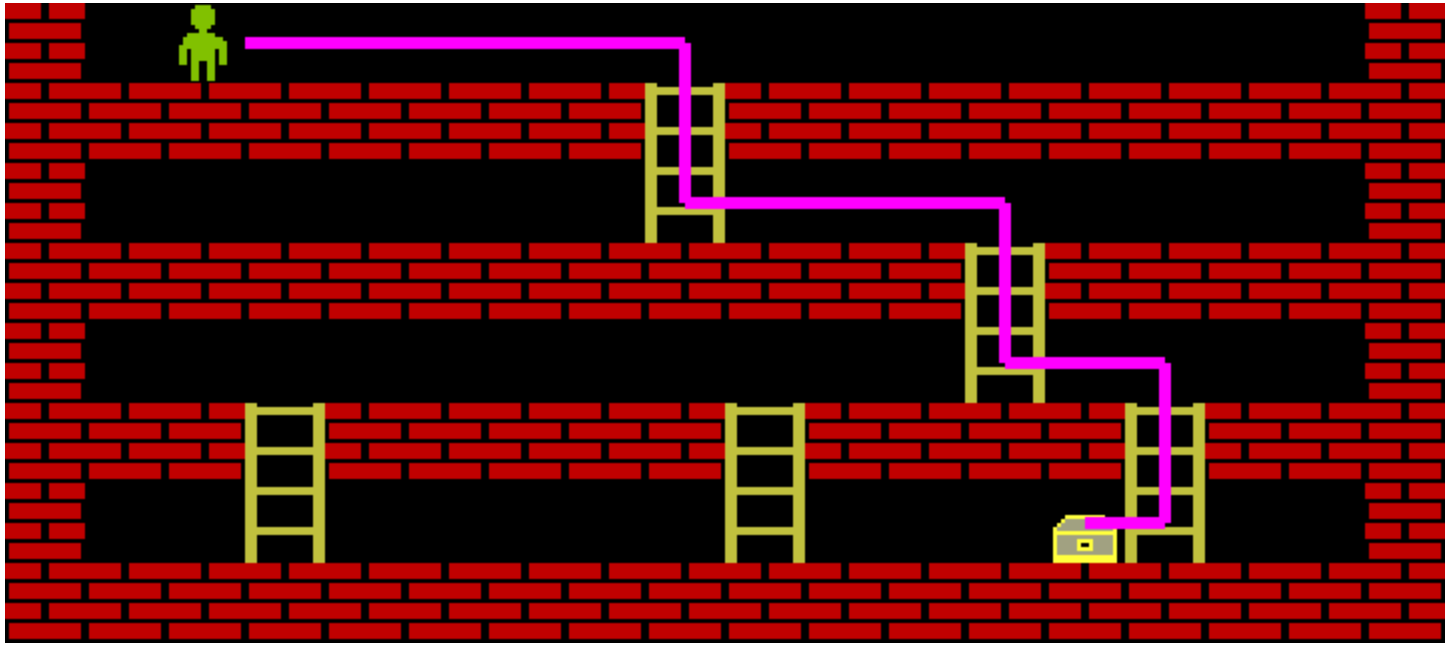
ИННОТЕХ

```
01. getNextVertex = (verticesToProcess: number[]): number => {
02.     let minAccessCost = Number.MAX_SAFE_INTEGER;
03.     let result = -1;
04.     verticesToProcess.forEach((nodeIndex) => {
05.         const vertex = this.vertices[nodeIndex as number];
06.         if (vertex.processed === false &&
07.             vertex.accessCost < minAccessCost)
08.         {
09.             minAccessCost = vertex.accessCost;
10.             result = nodeIndex as number;
11.         }
12.     });
13.     return result;
14. };
```

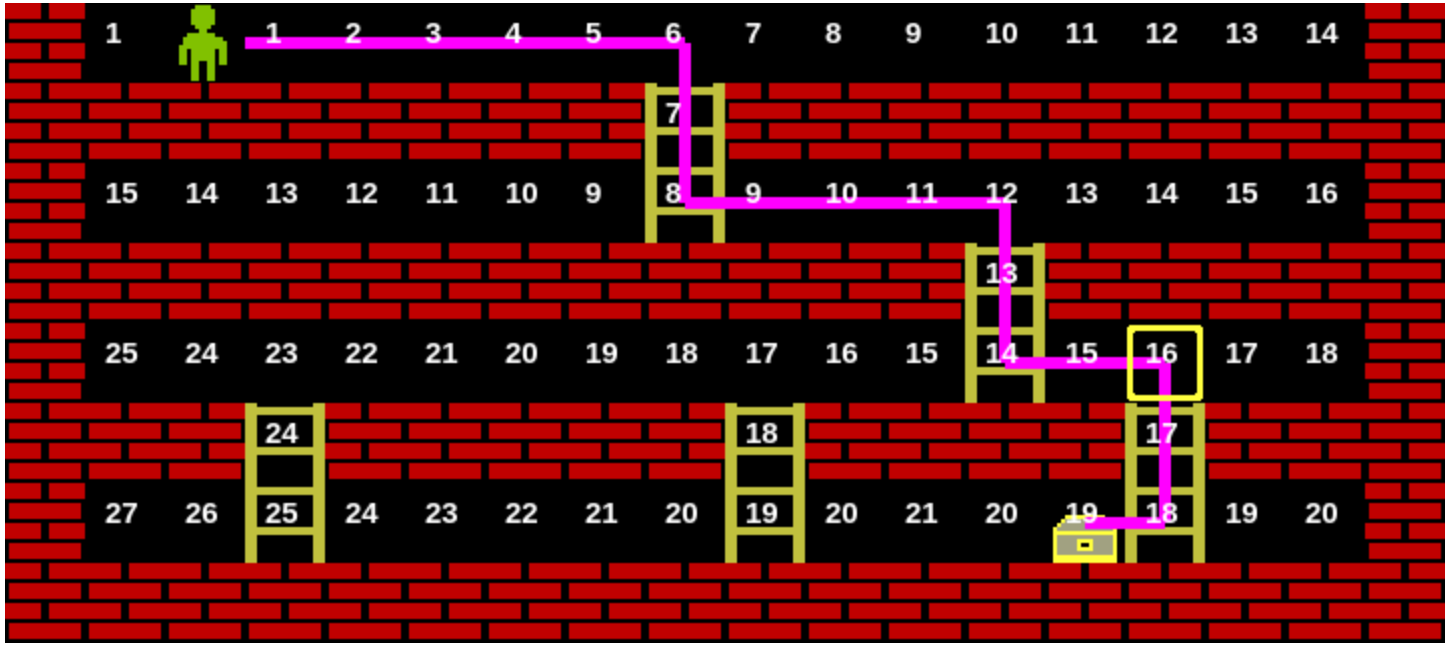
3. ВЫБОР ОПТИМАЛЬНОГО ПУТИ



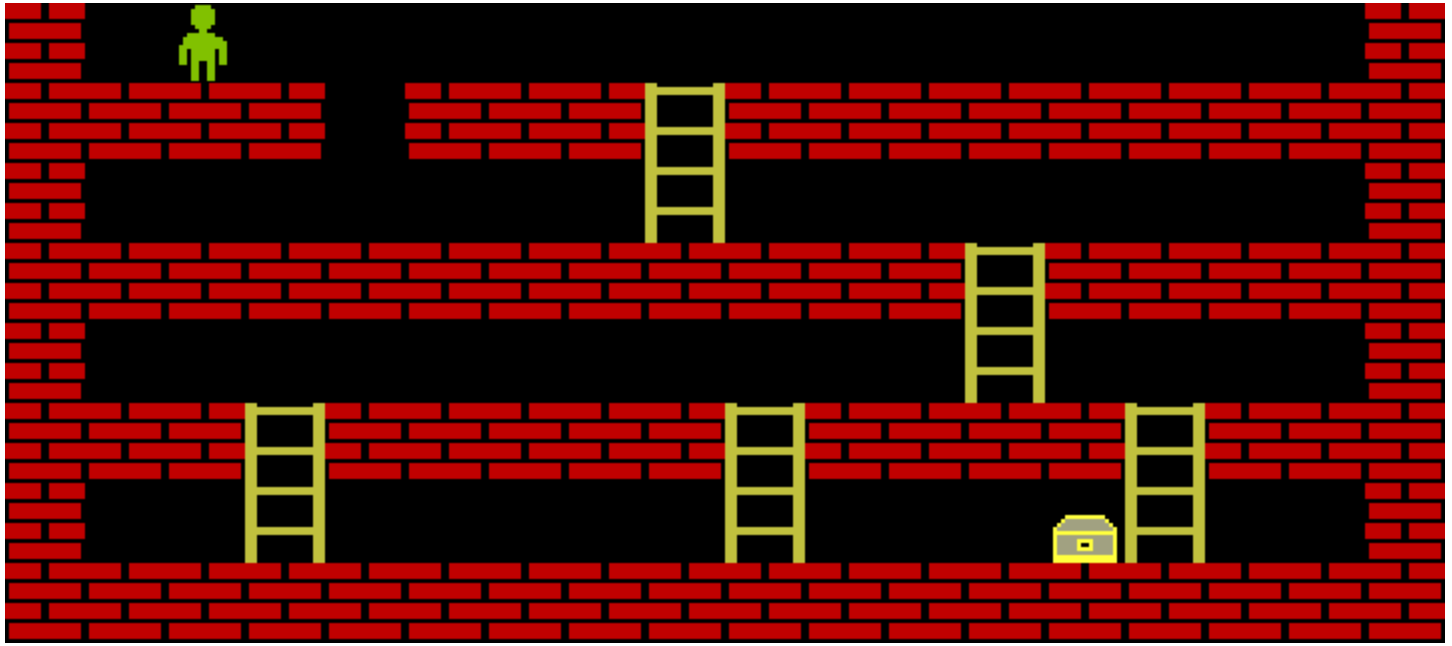
3. ВЫБОР ОПТИМАЛЬНОГО ПУТИ



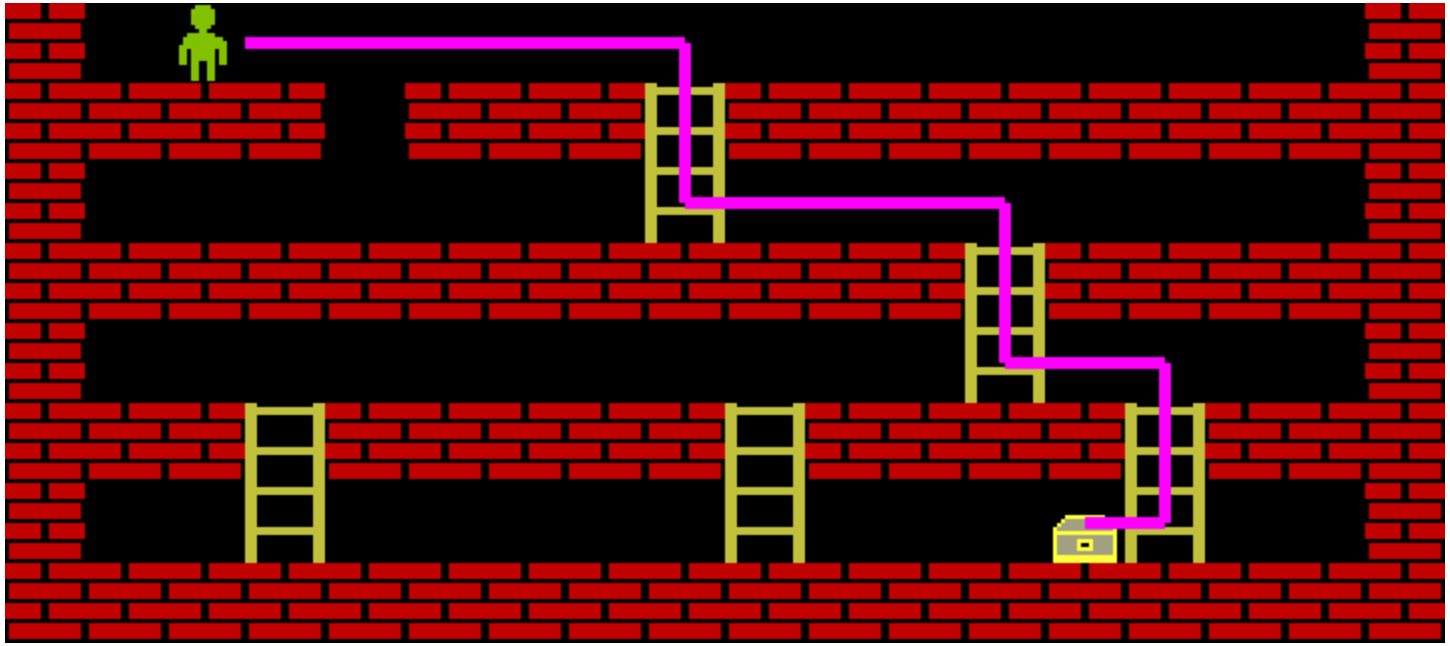
3. ВЫБОР ОПТИМАЛЬНОГО ПУТИ



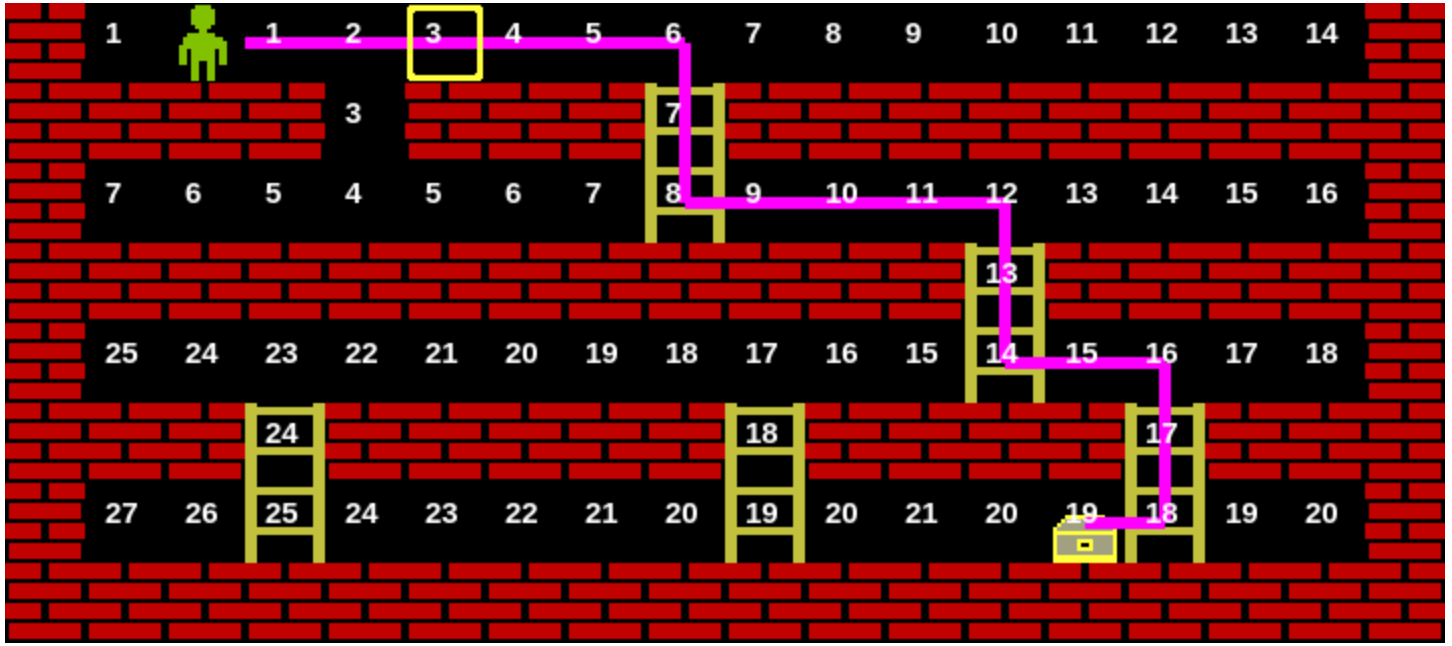
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



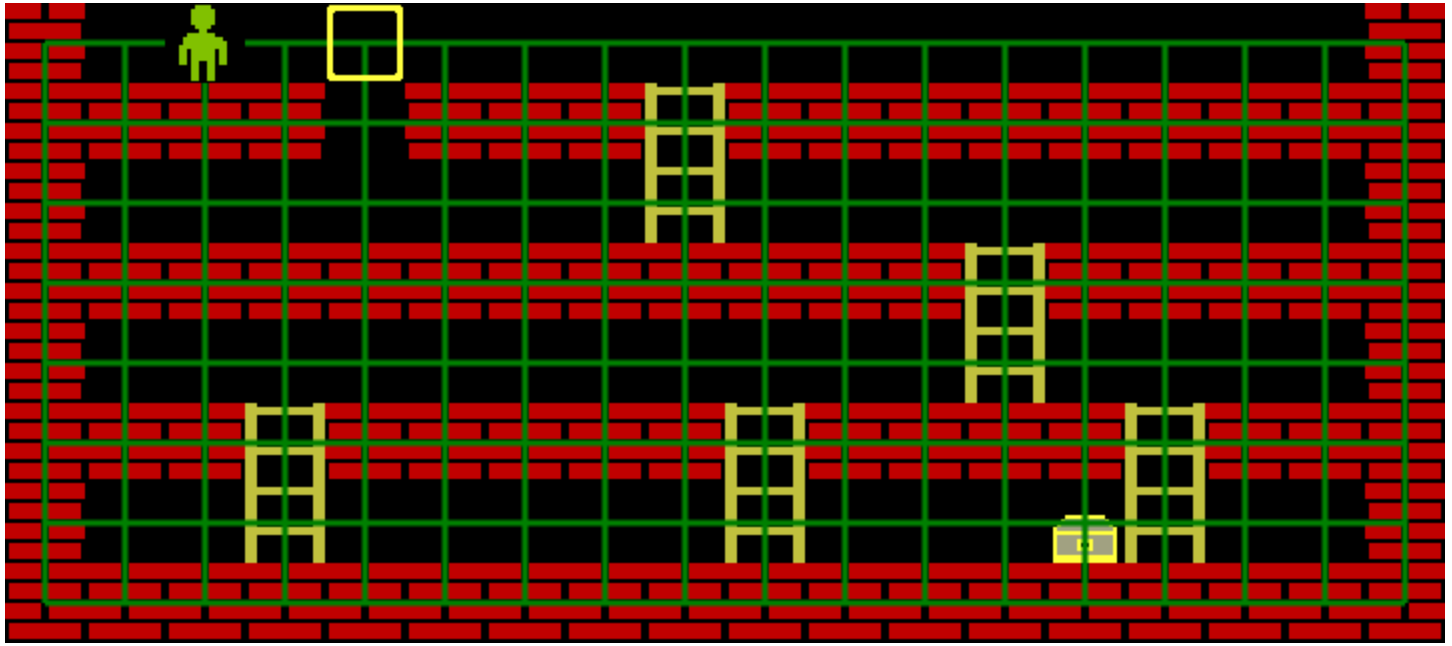
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



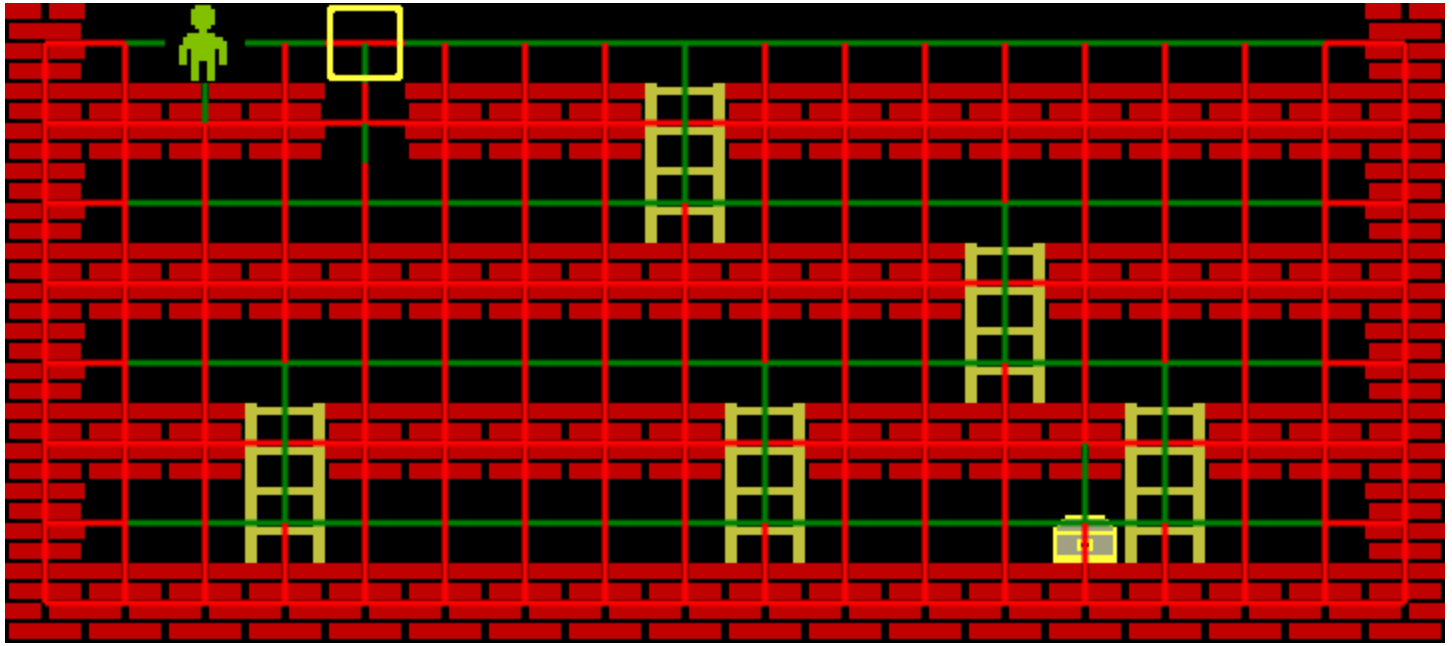
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



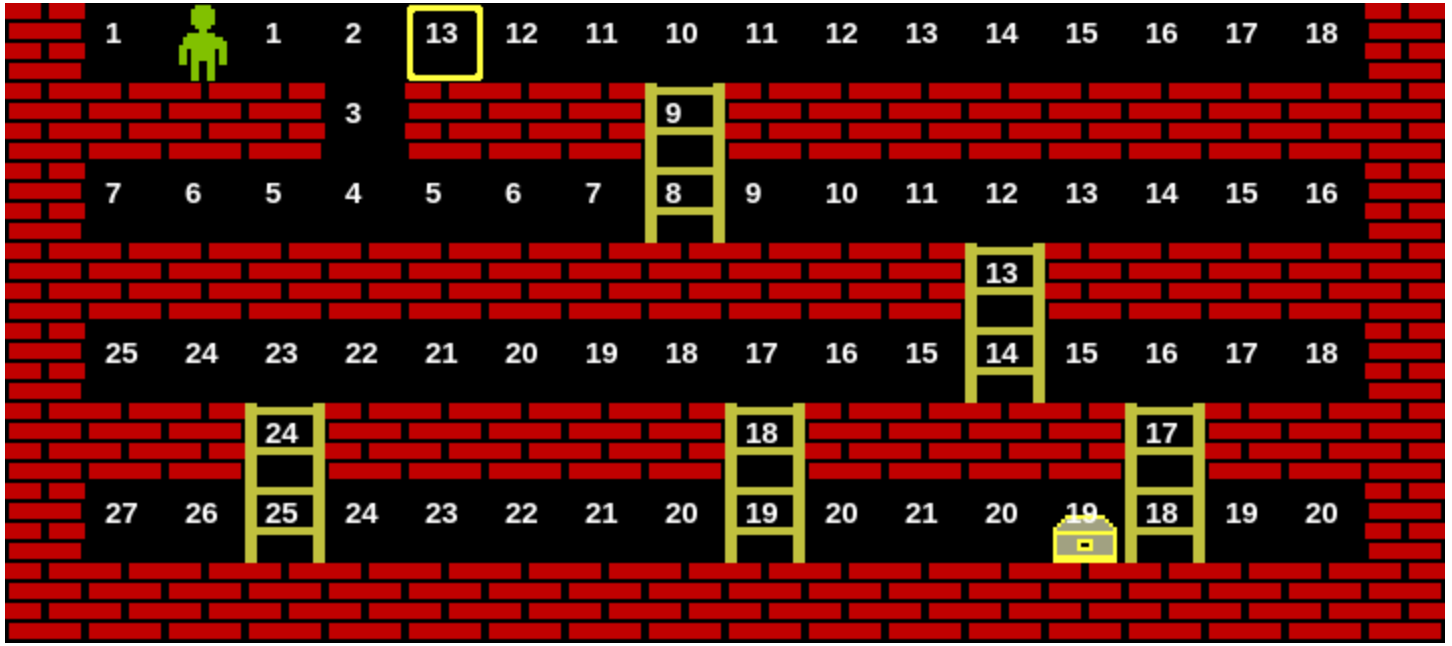
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



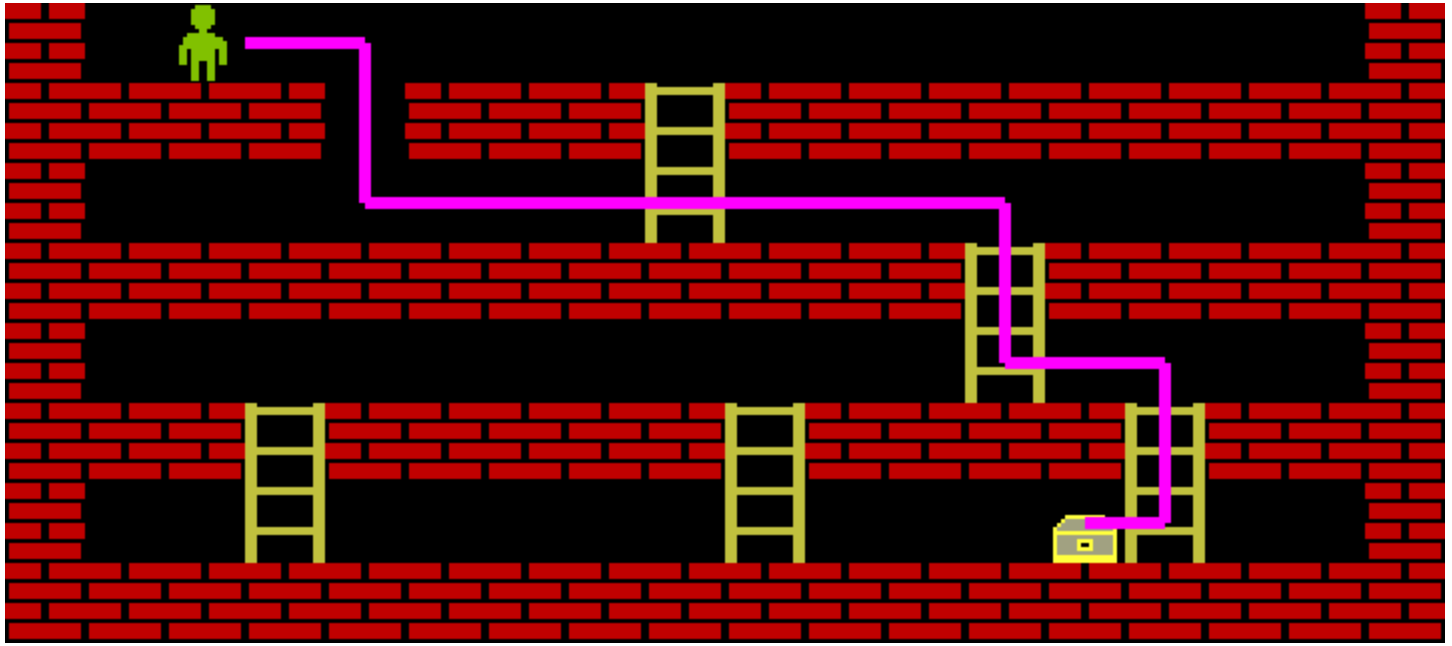
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



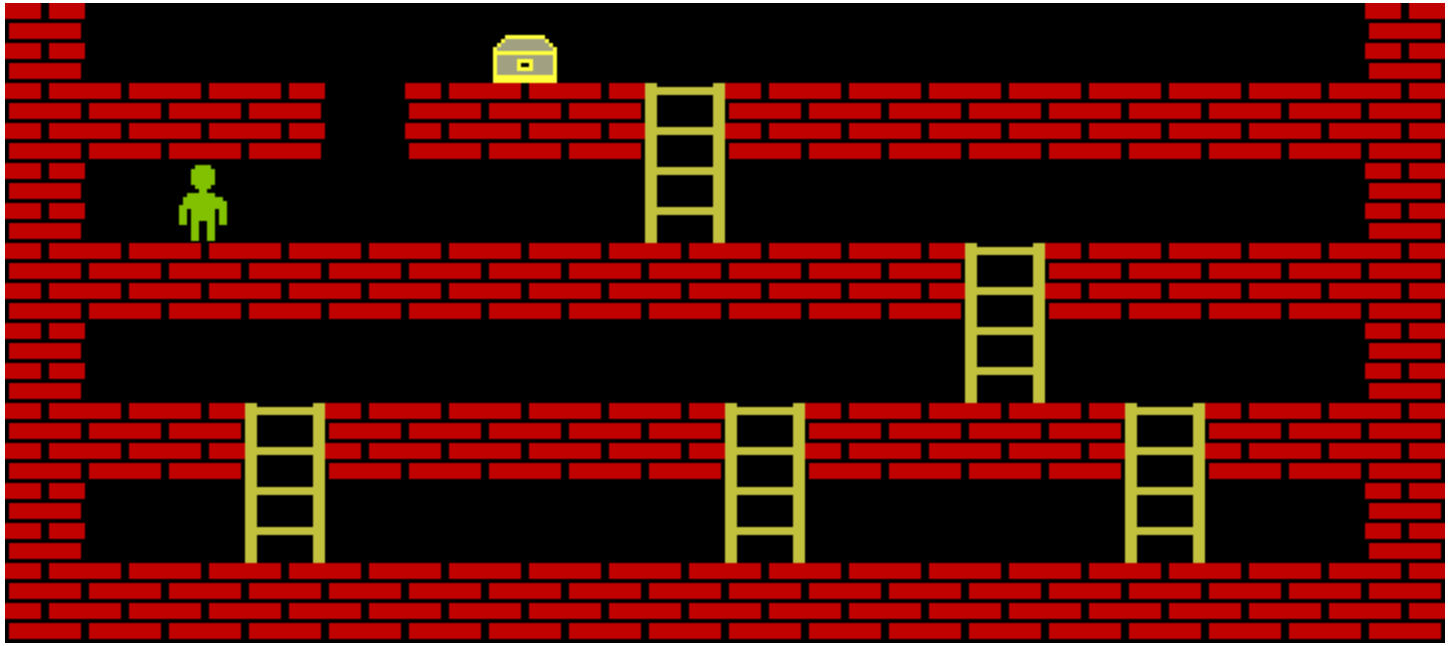
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



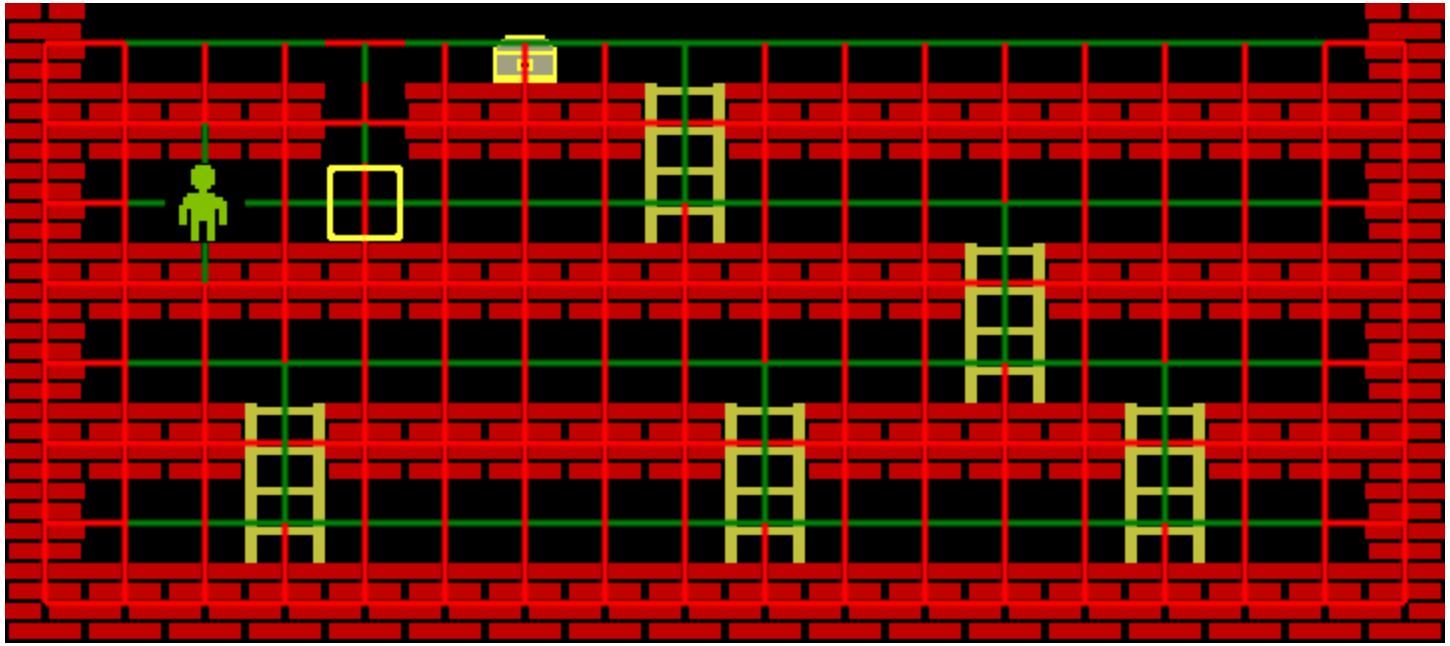
3. ПАДЕНИЕ В ОТВЕРСТИЕ В ПОЛУ



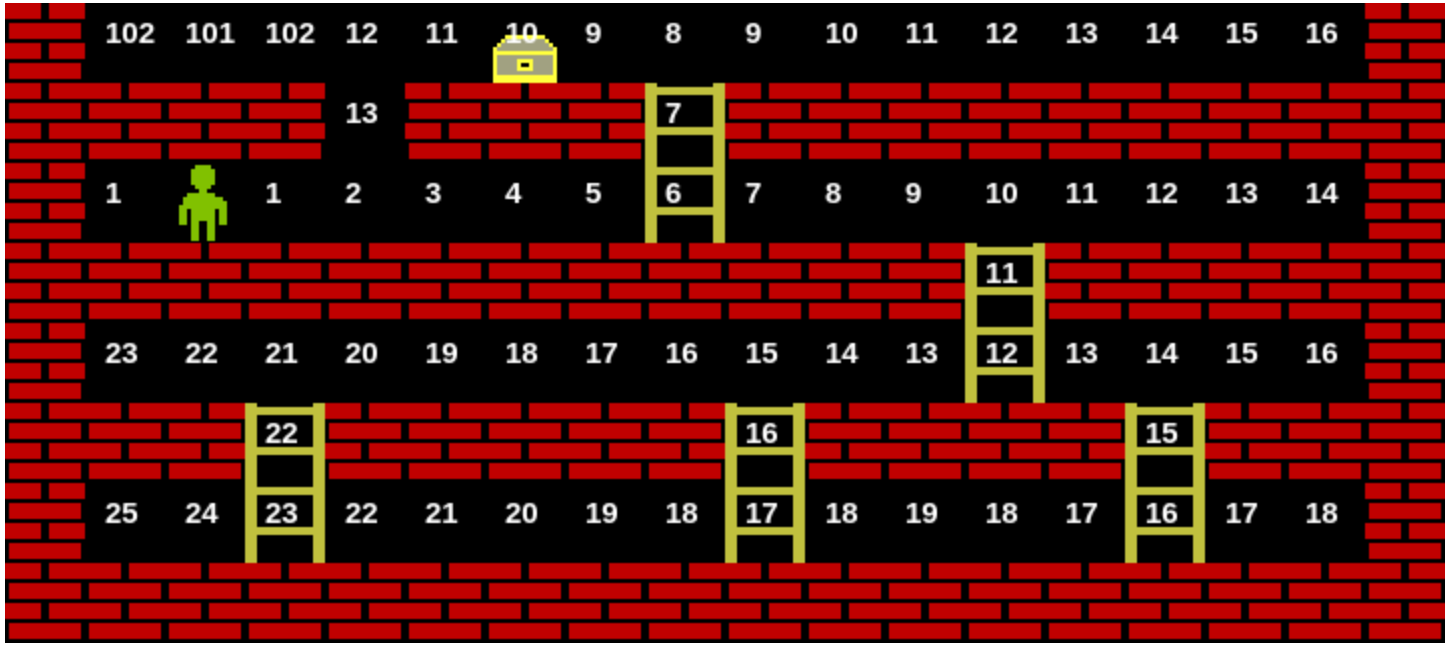
3. ДВИЖЕНИЕ ВВЕРХ



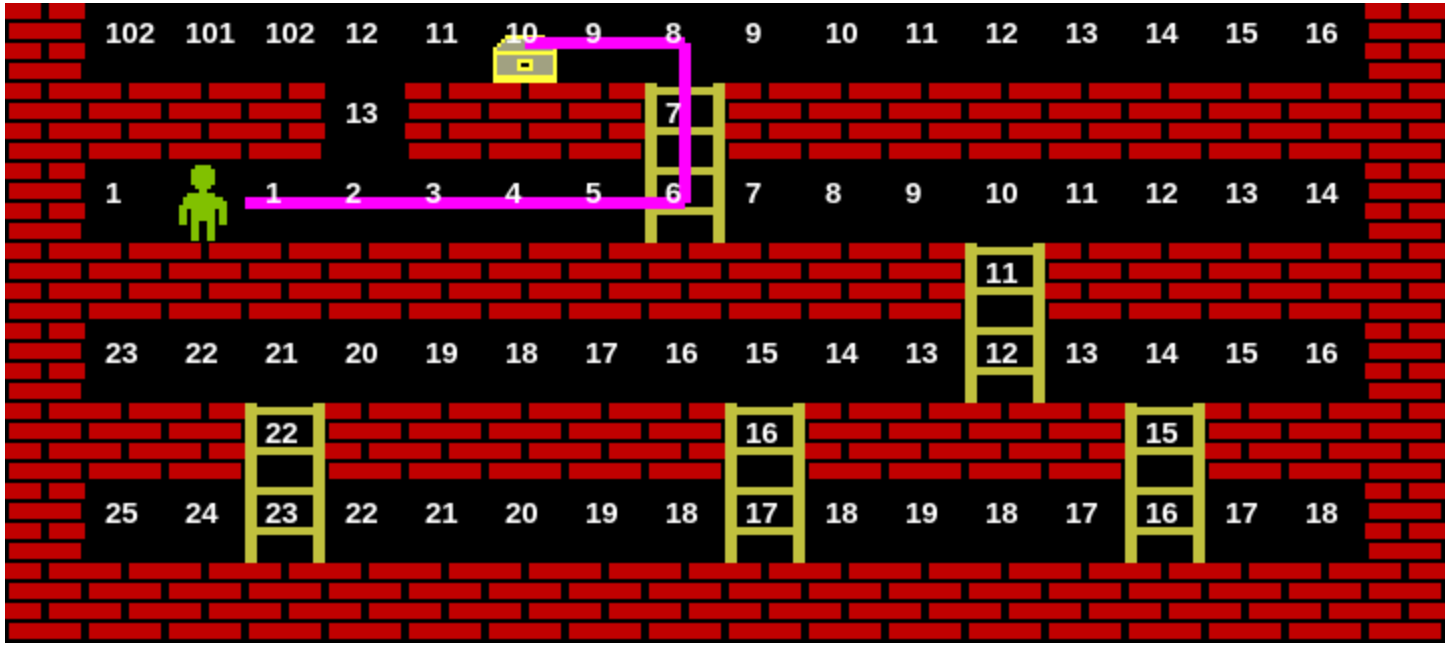
3. ДВИЖЕНИЕ ВВЕРХ



3. ДВИЖЕНИЕ ВВЕРХ

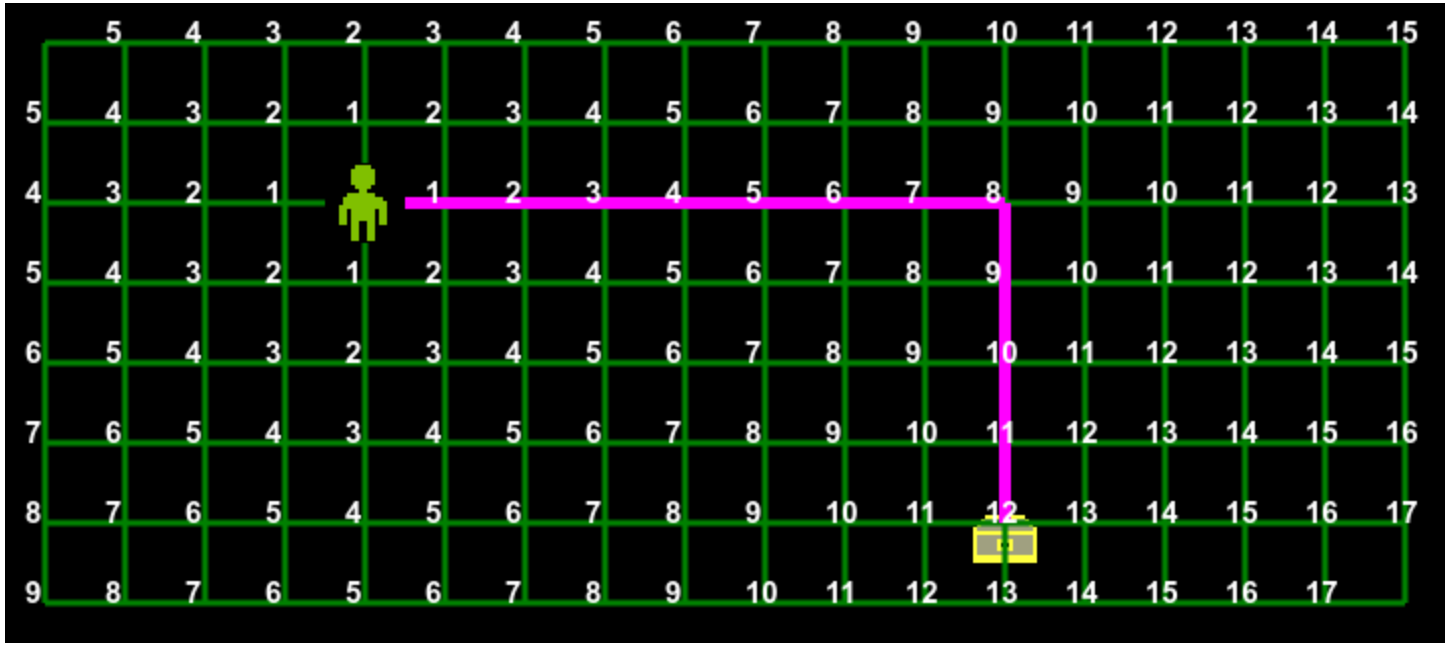


3. ДВИЖЕНИЕ ВВЕРХ

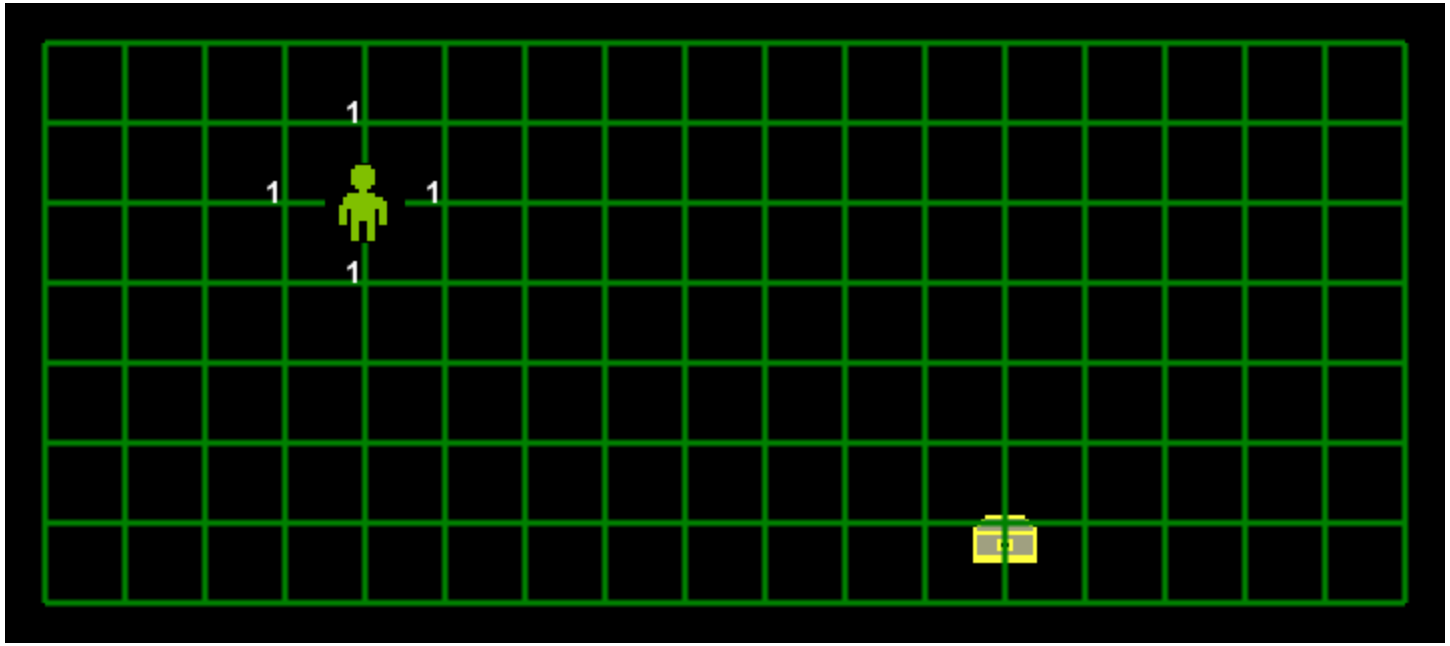


СОДЕРЖАНИЕ

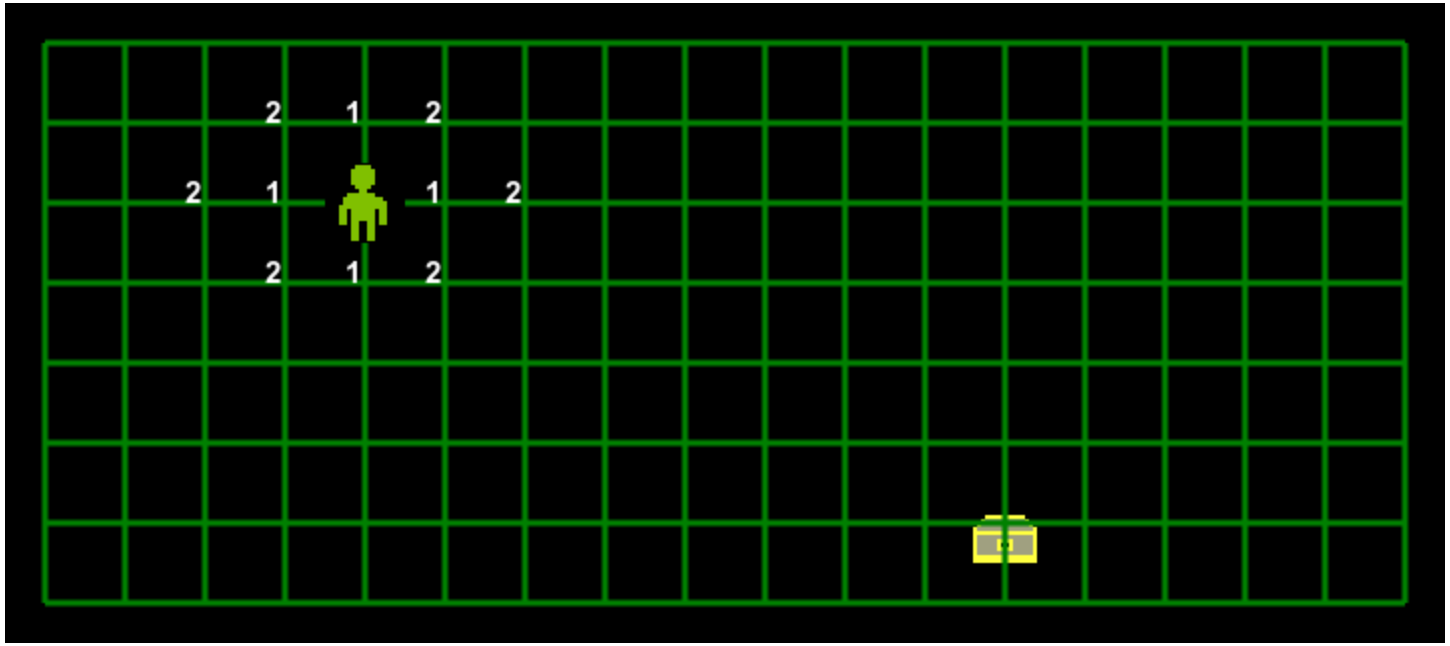
4. АЛГОРИТМ ДЕЙКСТРЫ. РЕЗУЛЬТАТ РАСЧЕТА ГРАФА



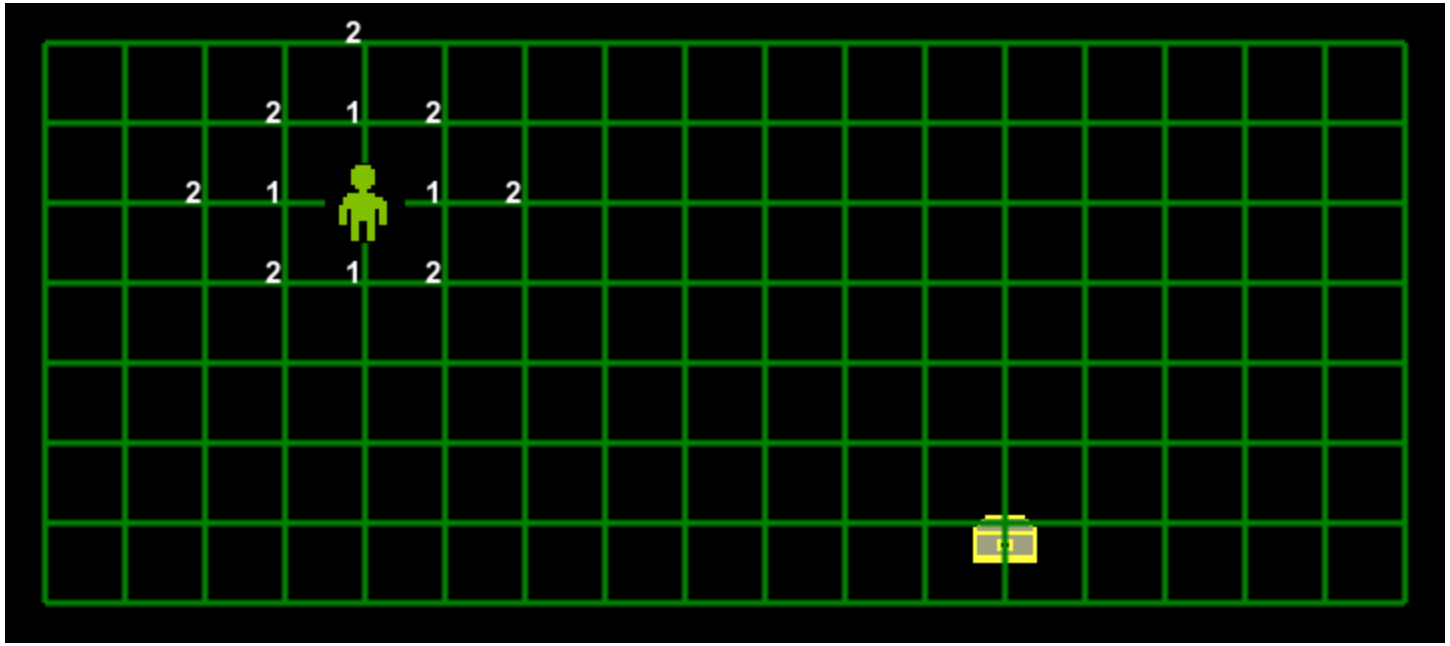
4. АЛГОРИТМ ДЕЙКСТРЫ. В КОНЦЕ ШАГА 1



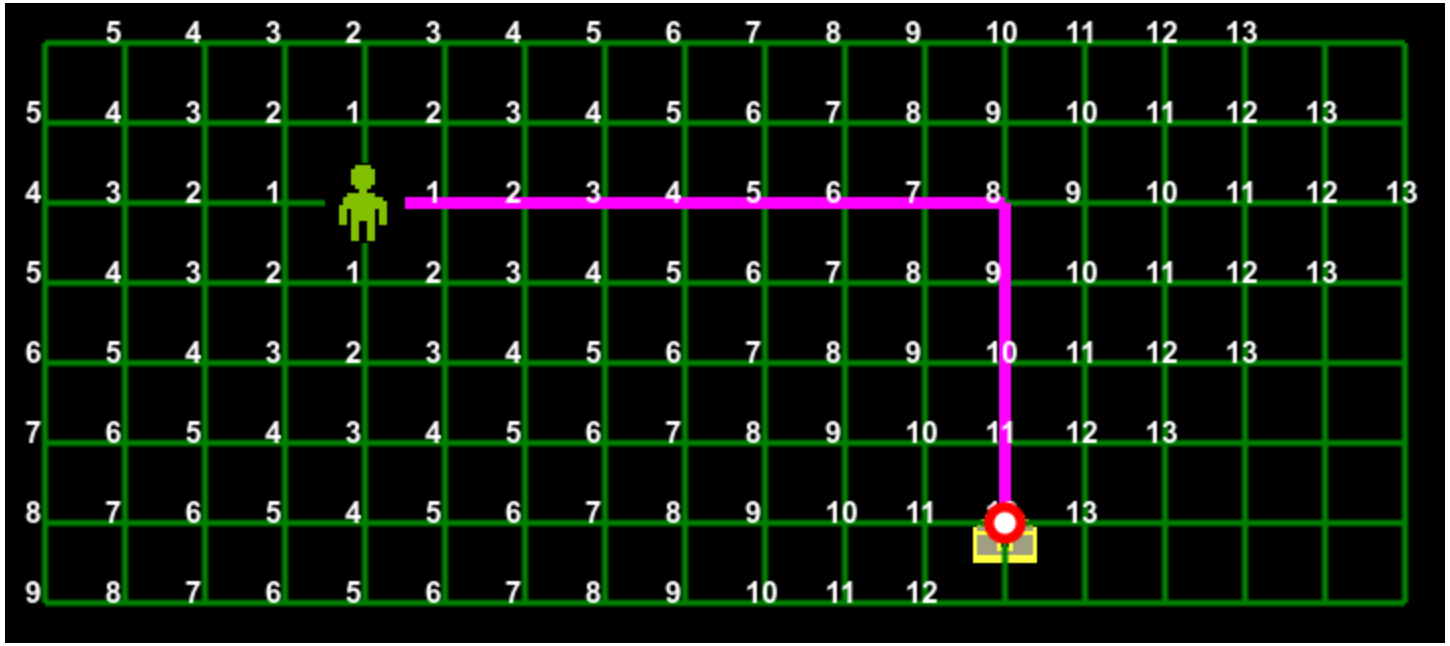
4. АЛГОРИТМ ДЕЙКСТРЫ. В КОНЦЕ ШАГА 2



4. АЛГОРИТМ ДЕЙКСТРЫ. В КОНЦЕ ШАГА 3



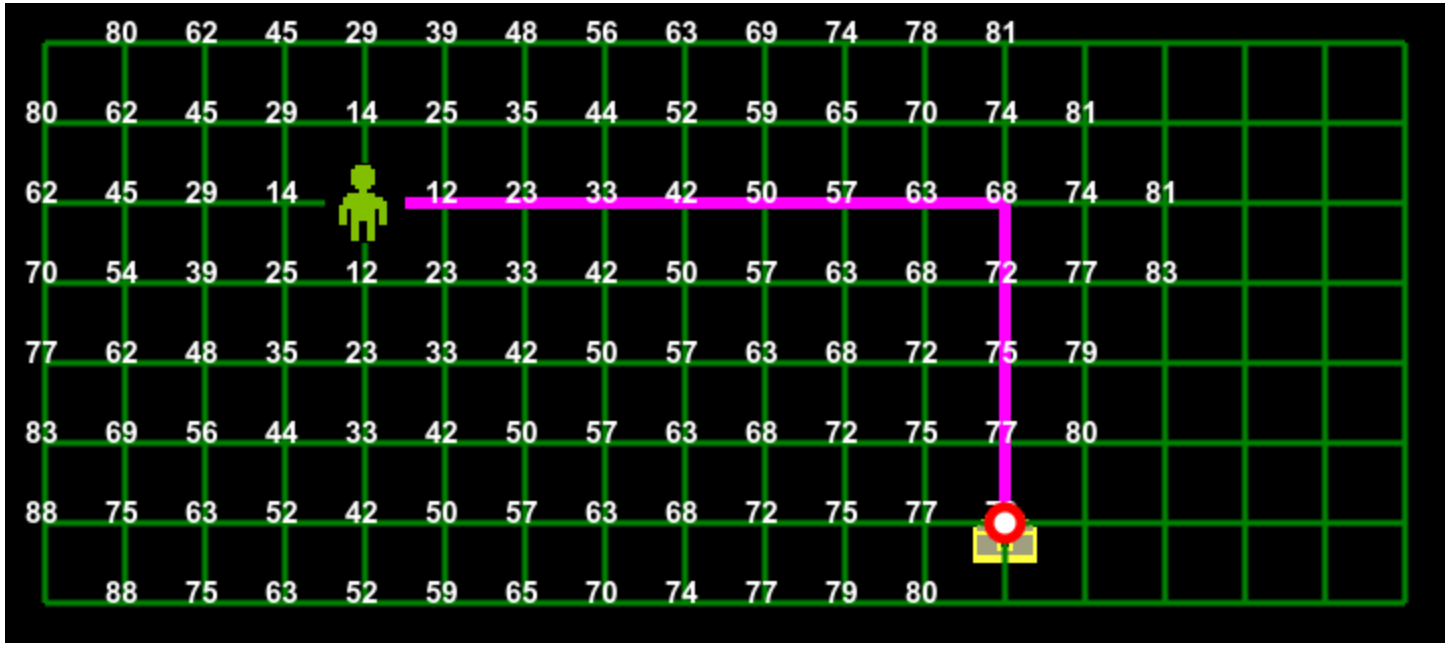
4. ОПТИМИЗАЦИЯ: РАННИЙ ВЫХОД



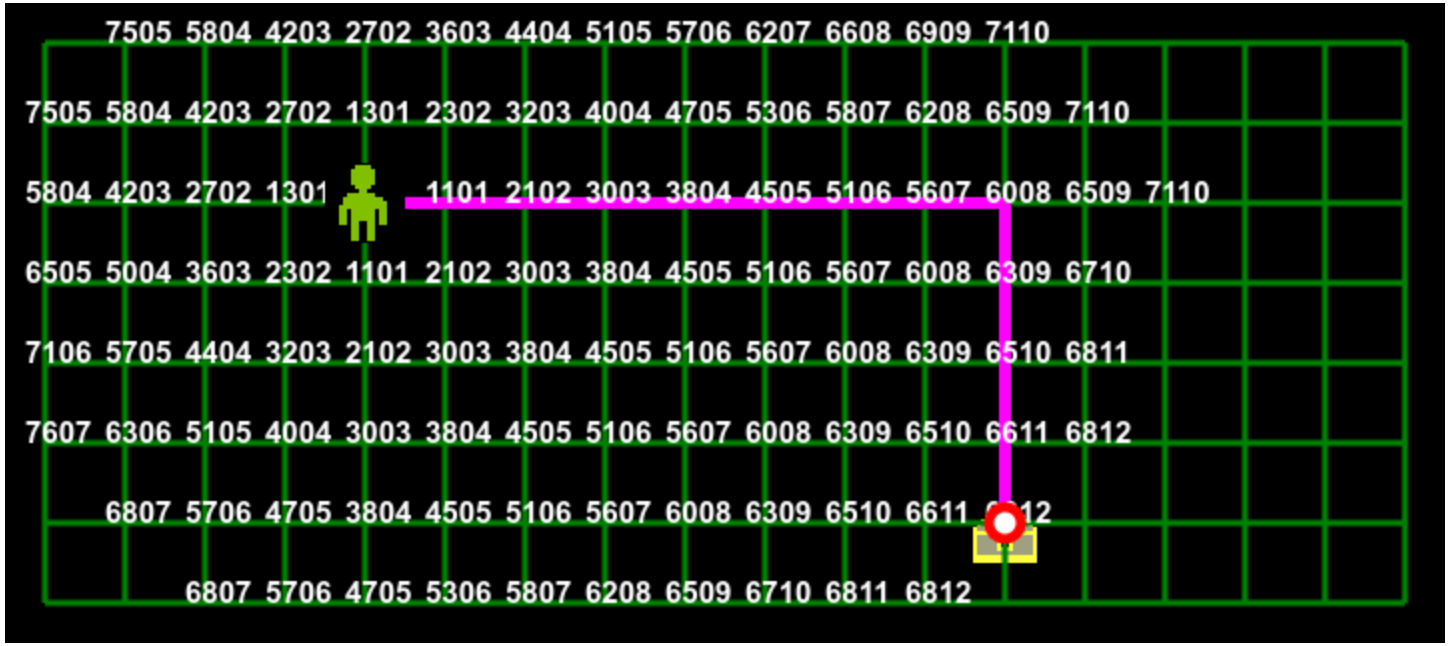
4. ОПТИМИЗАЦИЯ: ЭВРИСТИЧЕСКАЯ ФУНКЦИЯ H()

H(x0, x1, y0, y1) = |x1 — x0| + |y1 — y0|

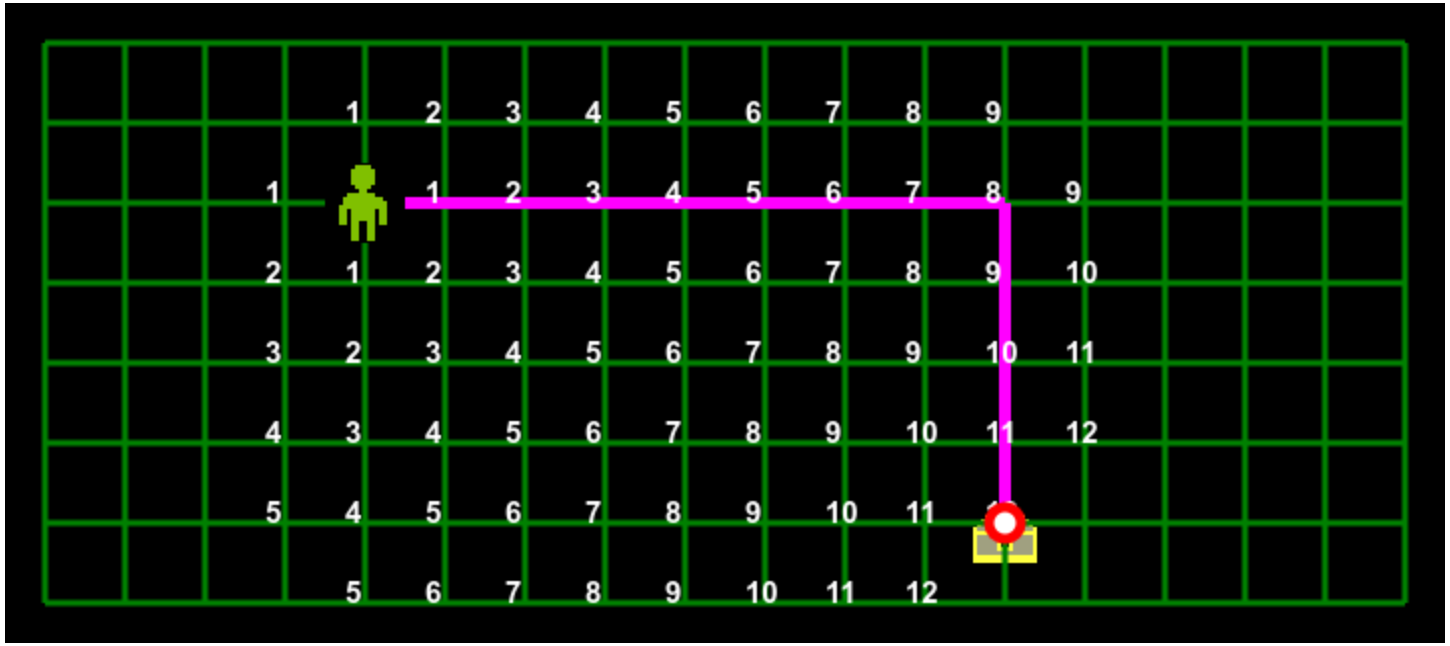
4. СТОИМОСТЬ ПЕРЕХОДА С УЧЕТОМ ЭВРИСТИКИ



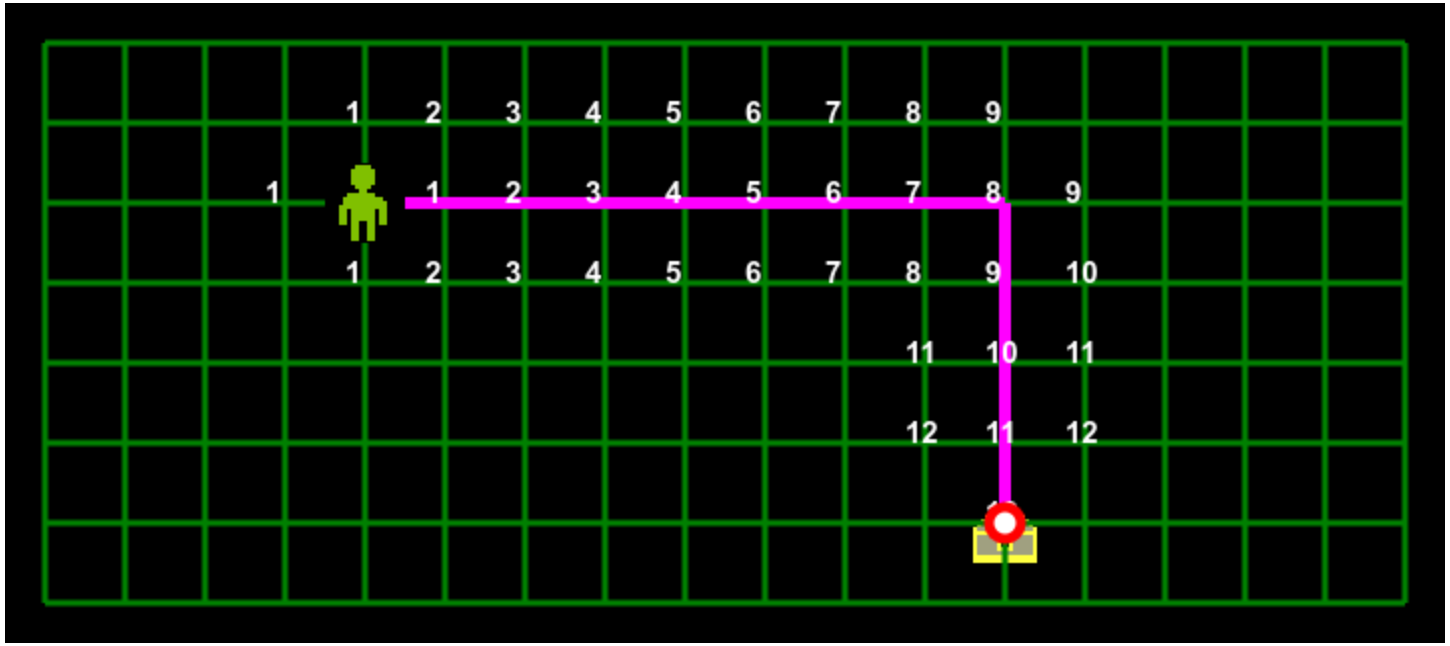
4. СТОИМОСТЬ ПЕРЕХОДА. УВЕЛИЧИВАЕМ ВЕС H()*100



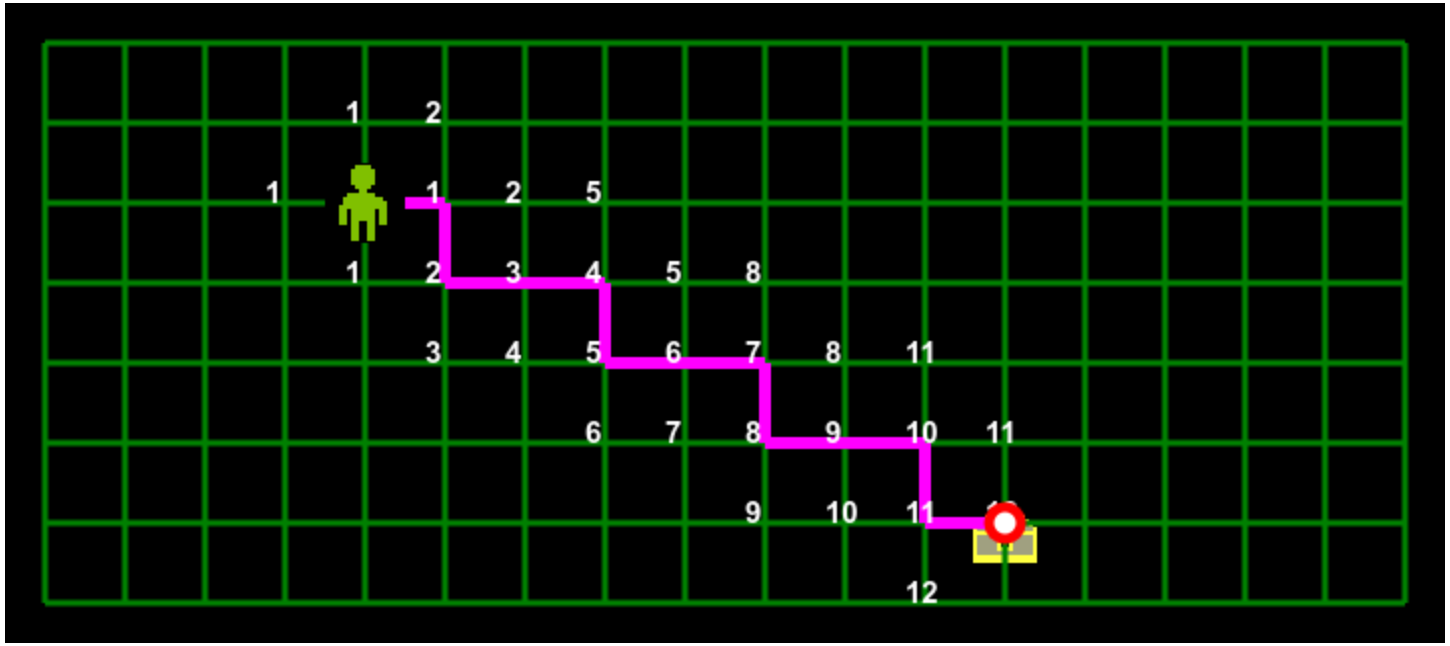
4. ЭВРИСТИКА H1(). ВЫБОР СЛЕДУЮЩЕЙ ВЕРШИНЫ



4. ВЫБОР СЛЕДУЮЩЕЙ ВЕРШИНЫ. H2() = H1()*2



4. H3(): УЧИТЫВАЕМ ОТКЛОНЕНИЕ ОТ ПРЯМОЙ



4. НАСТРАИВАЕМ ЭВРИСТИЧЕСКУЮ ФУНКЦИЮ

H1(x, y, xT, yT) = |xT — x| + |yT — y|

H2(x, y, xT, yT) = 2 * H1()

H3(x, y, xT, yT, xS, yS) = 2 * H1() + D(), где

D(x, y, A, B, C) = |A * x + B * y + C| / sqrt(A * A + B * B)

A = yT — yS

B = xS — xT

C = xS * (yS — yT) + yS * (xT — xS)

(xT, yT) — целевая вершина

(xS, yS) — исходная вершина

(x, y) — вершина, для которой вычисляем значение H()

4. ФУНКЦИЯ ВЫЧИСЛЕНИЯ СЛЕДУЮЩЕЙ ВЕРШИНЫ

ИННОТЕХ

```
01. getNextVertex = (verticesToProcess: number[]): number => {
02.     let minAccessCost = Number.MAX_SAFE_INTEGER;
03.     let result = -1;
04.     verticesToProcess.forEach((nodeIndex) => {
05.         const vertex = this.vertices[nodeIndex as number];
06.         const h = this.heuristic(nodeIndex, this.toVertex);
07.         if (vertex.processed === false &&
08.             vertex.accessCost + h < minAccessCost)
09.         {
10.             minAccessCost = vertex.accessCost + h;
11.             result = nodeIndex as number;
12.         }
13.     });
14.     return result;
15. };
```

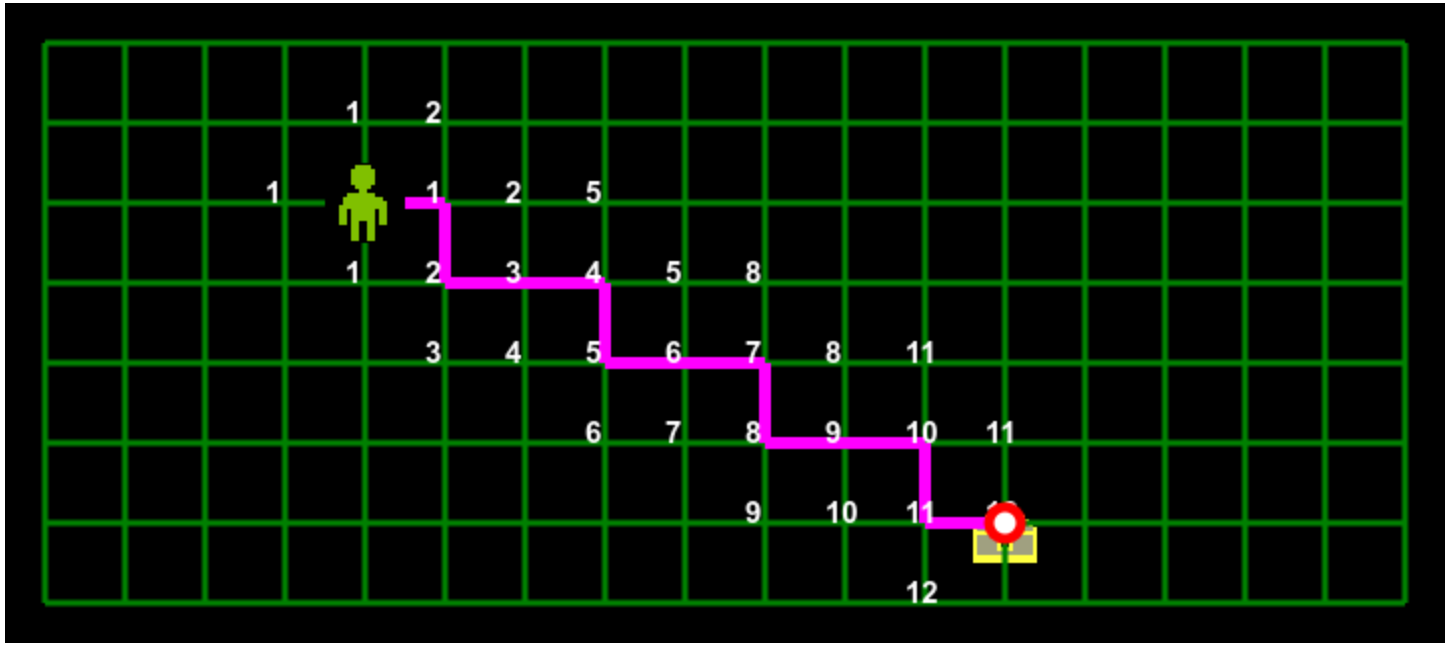
4. ЭВРИСТИЧЕСКАЯ ФУНКЦИЯ H2()

```
01. function heuristic(v0Index: number, v1Index: number) {
02.     const v0 = field.idxToPoint2D(v0Index, field.getWidth());
03.     const v1 = field.idxToPoint2D(v1Index, field.getWidth());
04.     const h = Math.abs(v0.x − v1.x) + Math.abs(v0.y − v1.y);
05.     return h * 2;
06. };
```

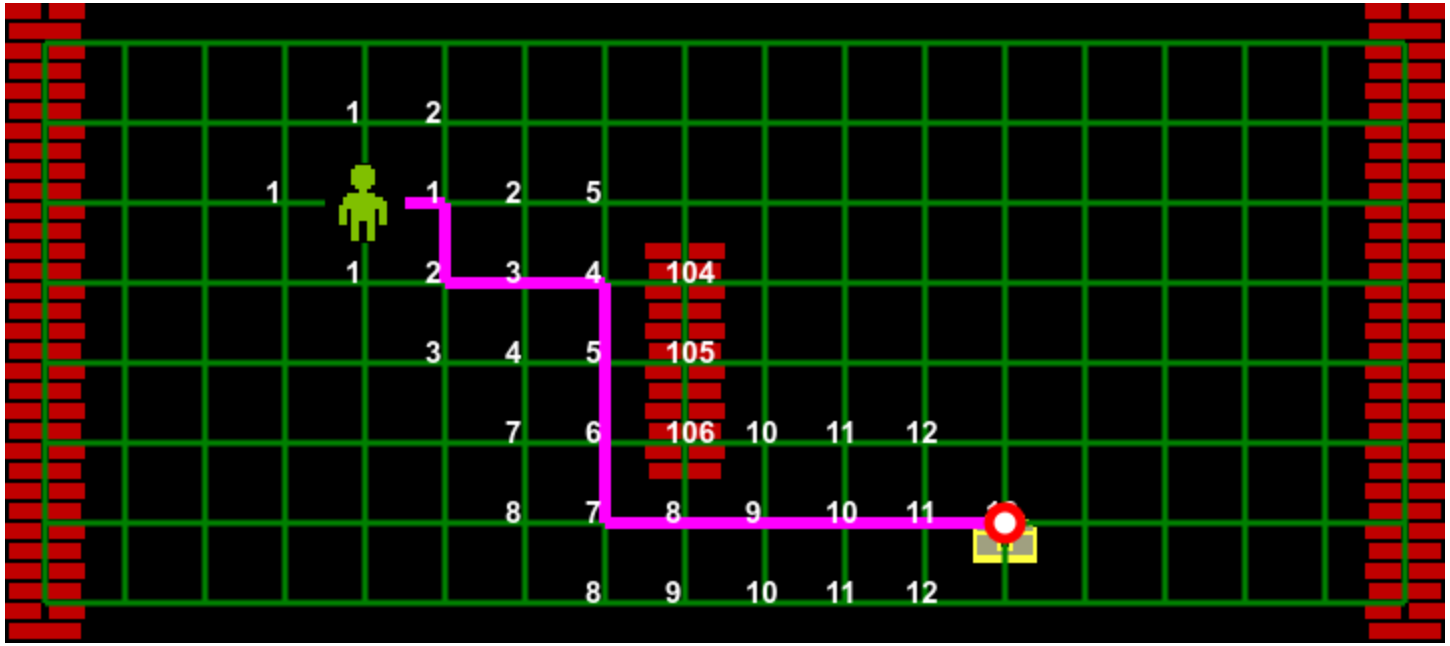
4. ЭВРИСТИЧЕСКАЯ ФУНКЦИЯ H3()

```
01. function heuristic(v0Index: number, v1Index: number) {
02.     const v0: Point2D = field.vIndexToPoint2D(v0Index);
03.     const v1: Point2D = field.vIndexToPoint2D(v1Index);
04.     const h = Math.abs(v0.x – v1.x) + Math.abs(v0.y – v1.y);
05.     const d = getDistance(A, B, C, v0);
06.     return h * 2 + d;
07. };
08. const getDistance = (A, B, C: number, v: Point2D) =>
09.     Math.abs(A * v.x + B * v.y + C) / Math.sqrt(A*A + B*B);
10. const getA = (vS: Point2D, vT: Point2D) => vT.y – vS.y;
11. const getB = (vS: Point2D, vT: Point2D) => vS.x – vT.x;
12. const getC = (vS: Point2D, vT: Point2D) =>
13.     vS.x * (vS.y – vT.y) + vS.y * (vT.x – vS.x);
```

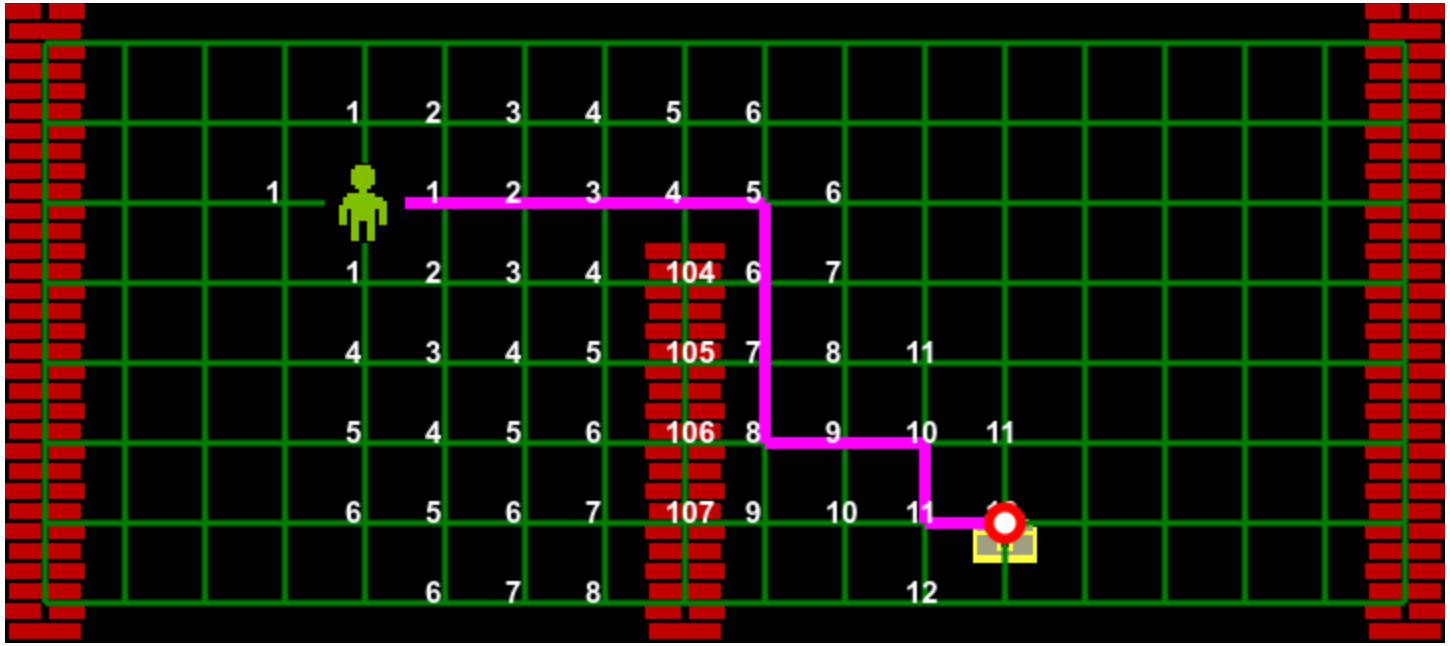
4. H3(): УЧИТЫВАЕМ ОТКЛОНЕНИЕ ОТ ПРЯМОЙ



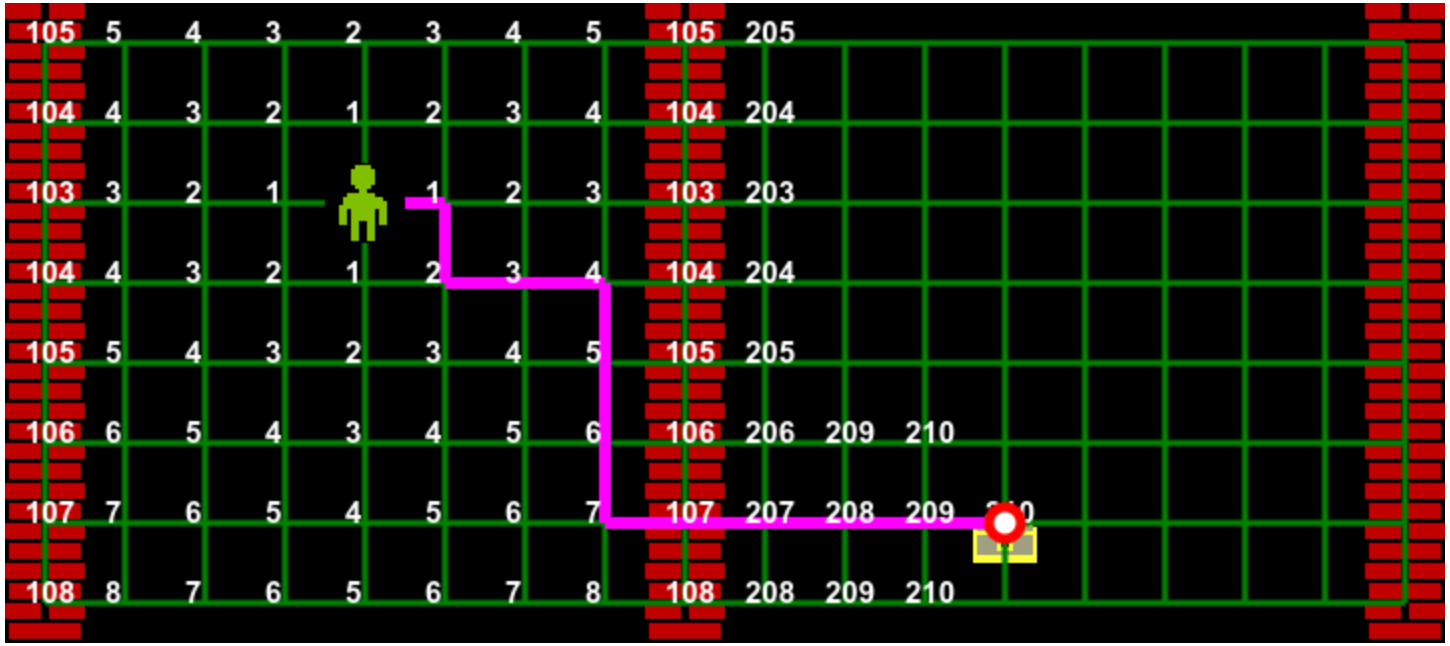
4. ДОБАВЛЯЕМ БАРЬЕР



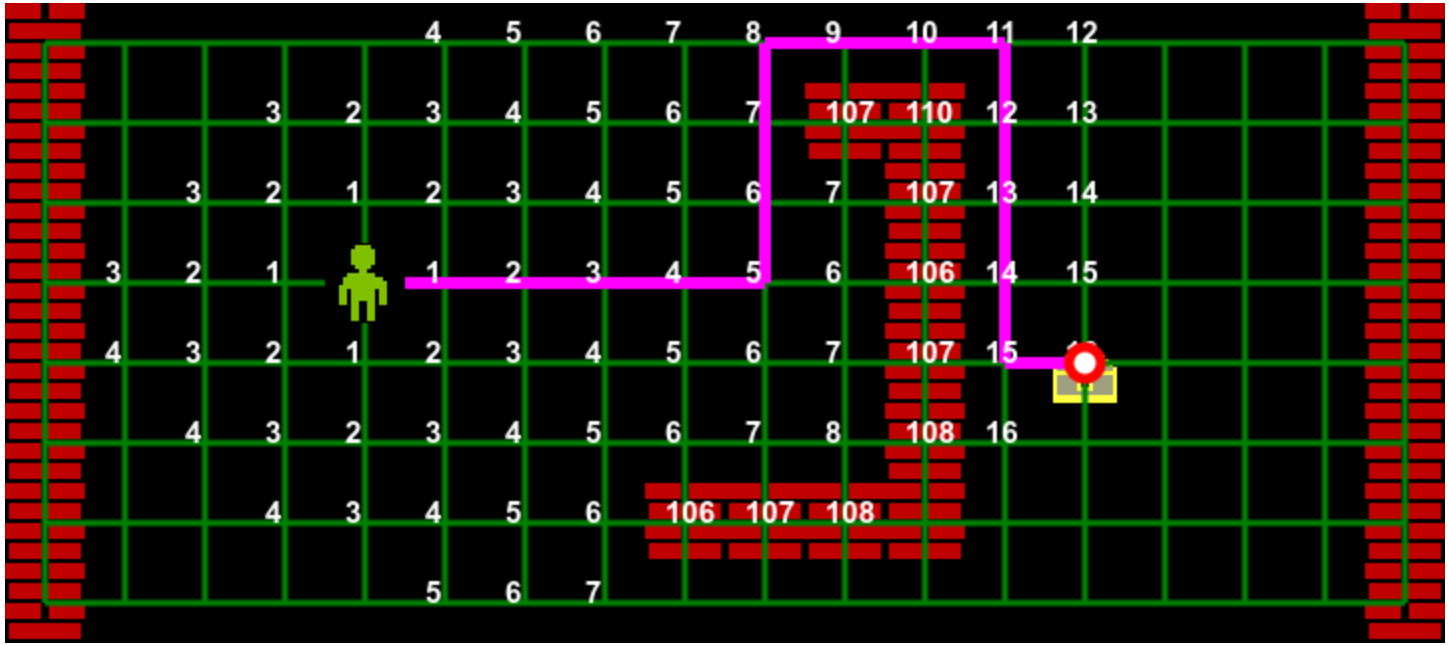
4. СПЛОШНОЙ БАРЬЕР СНИЗУ



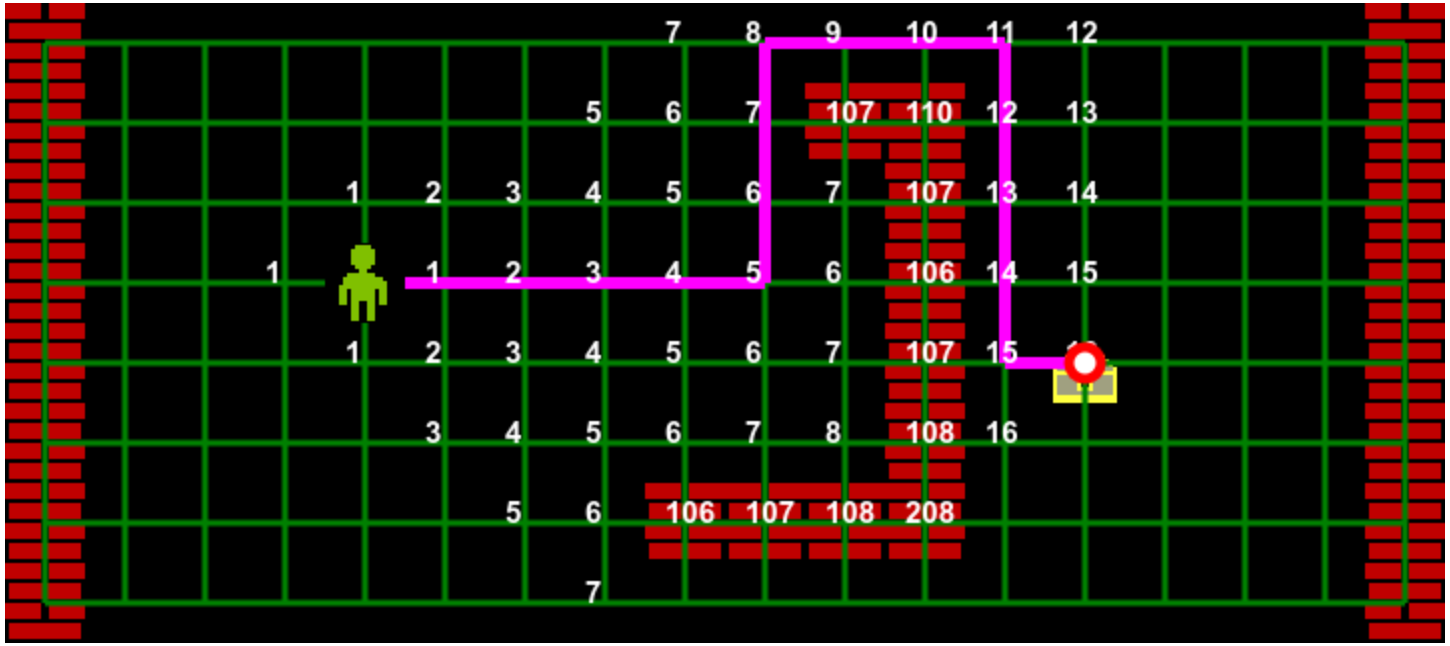
4. ПРОХОДИМ СКВОЗЬ СТЕНУ



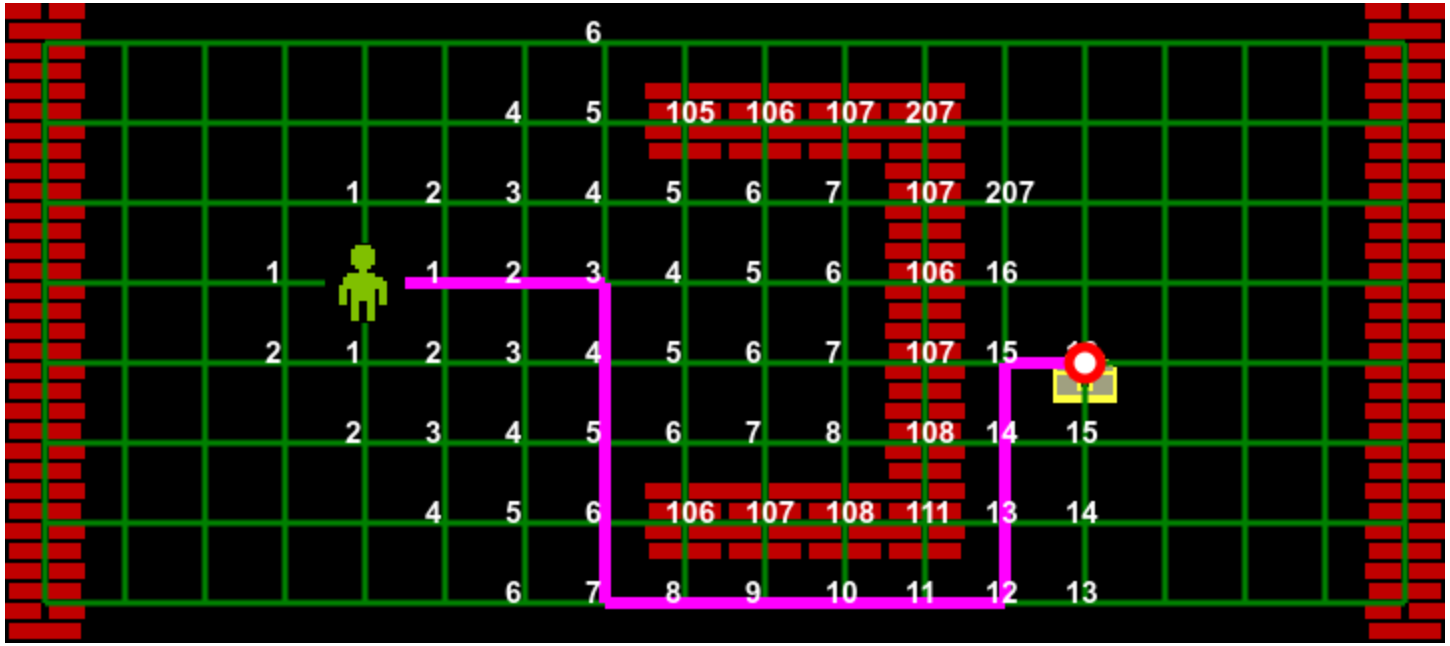
4. ДРУГОЙ БАРЬЕР



4. H4() – С НОВЫМИ КОЭФФИЦИЕНТАМИ



4. БАРЬЕР: МЕНЯЕМ ФОРМУ ПРЕПЯТСТВИЯ



4. ЭВРИСТИЧЕСКАЯ ФУНКЦИЯ H4()

H1(x, y, xT, yT) = |xT — x| + |yT — y|

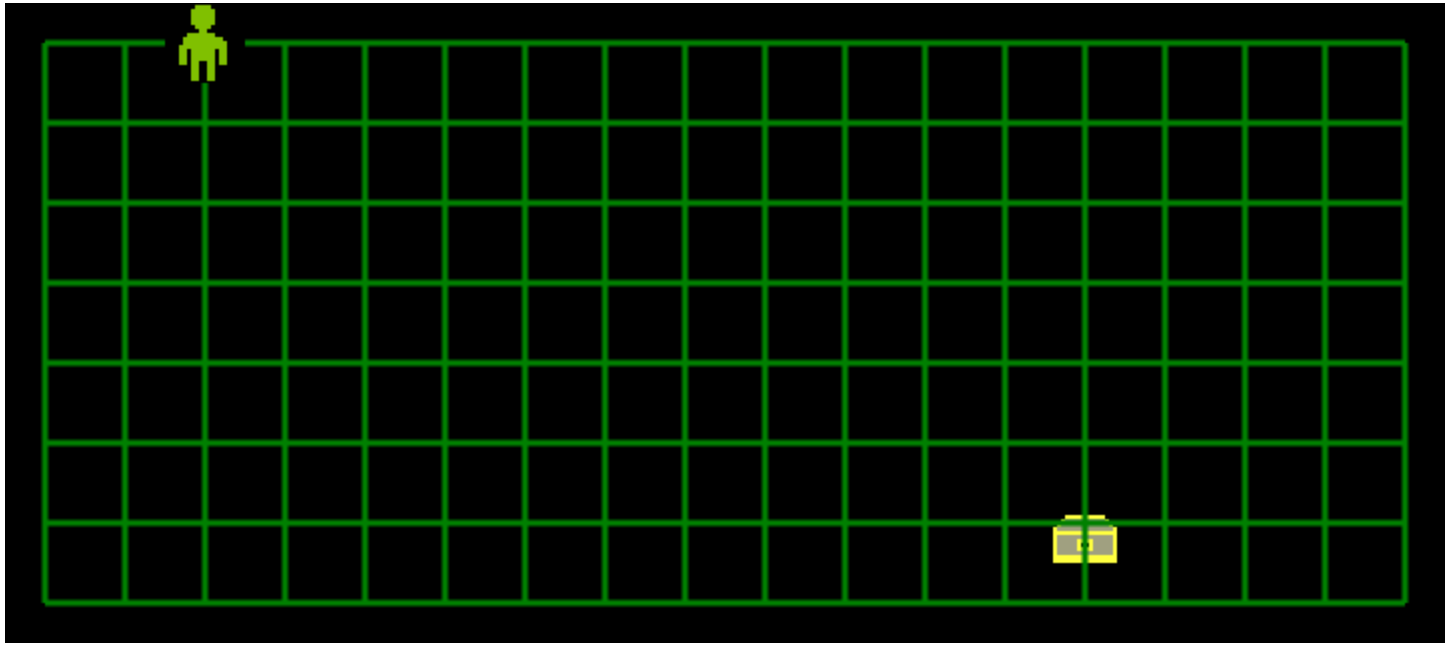
H2(x, y, xT, yT) = 2 * H1()

H3(x, y, xT, yT, xS, yS) = 2 * H1() + D()

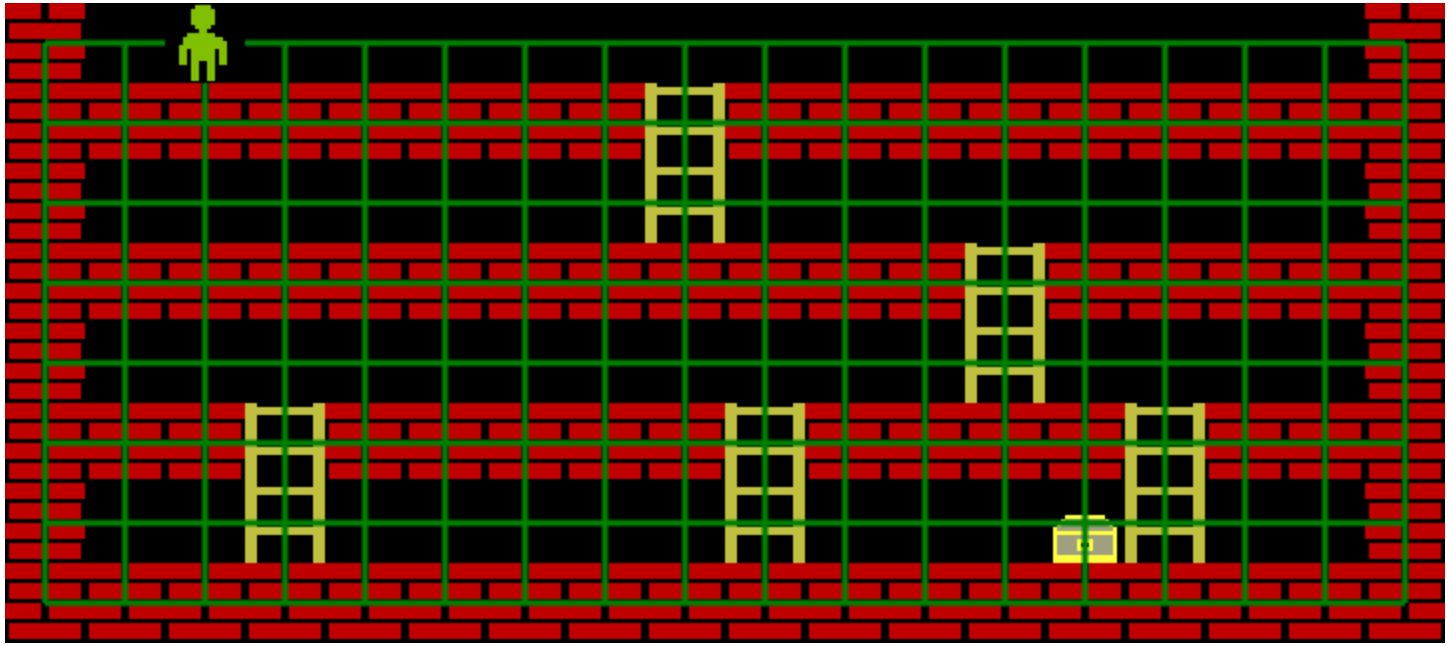
H4(x, y, xT, yT, xS, yS) = 5 * (4 * H1() + D())

СОДЕРЖАНИЕ

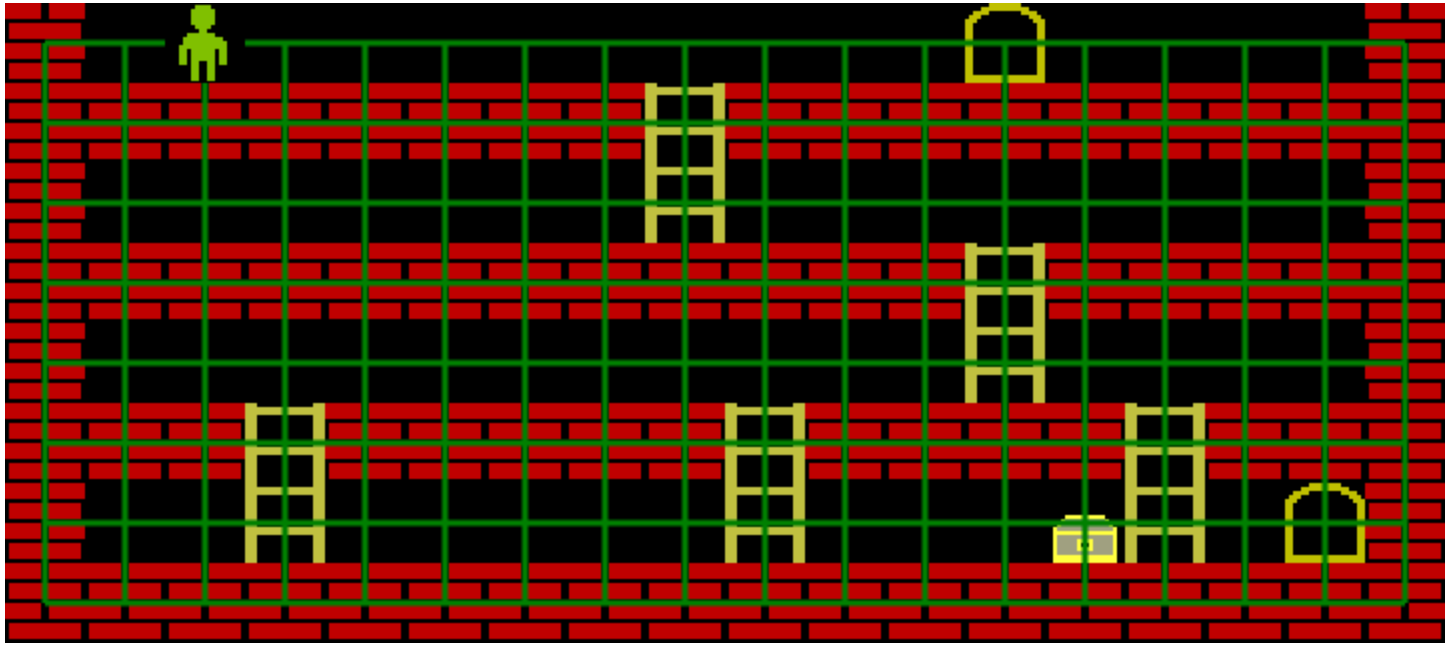
5. ОГРАНИЧЕНЫ ЛИ МЫ 2D-ПРОСТРАНСТВОМ?



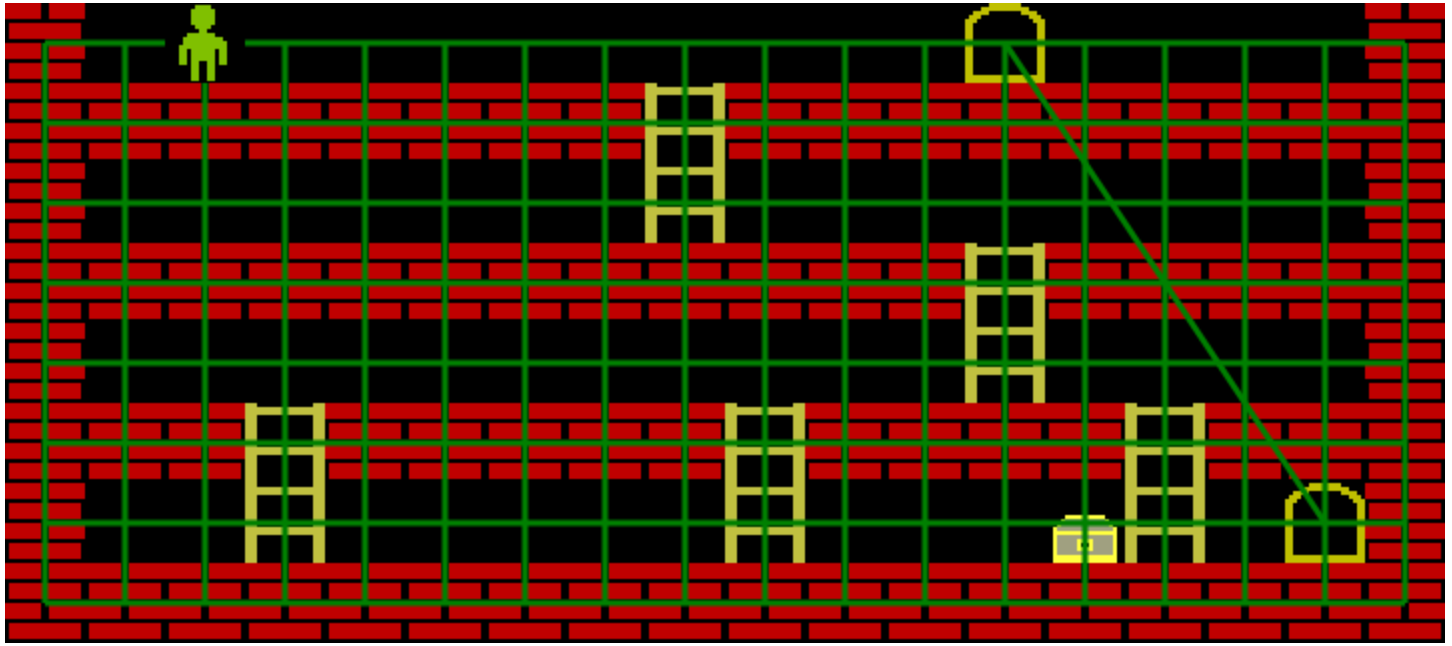
5. ОГРАНИЧЕНЫ ЛИ МЫ 2D-ПРОСТРАНСТВОМ?



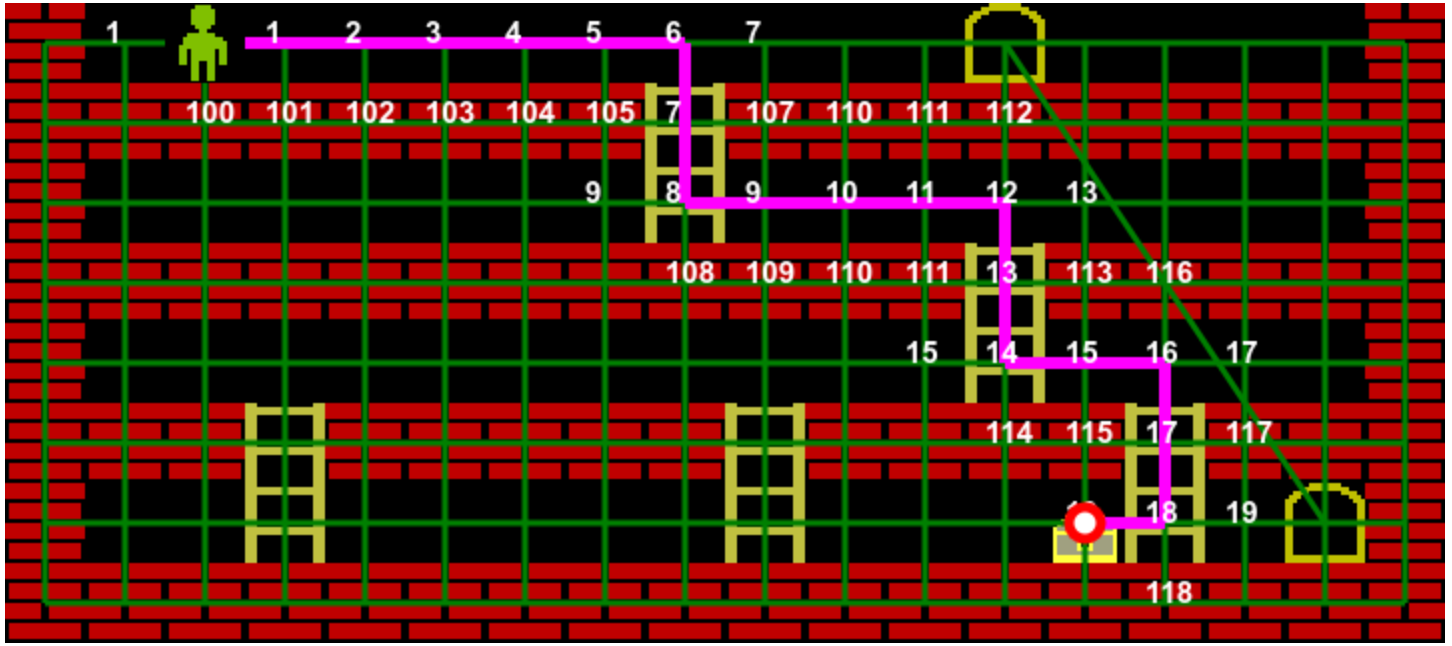
5. ОГРАНИЧЕНЫ ЛИ МЫ 2D-ПРОСТРАНСТВОМ?



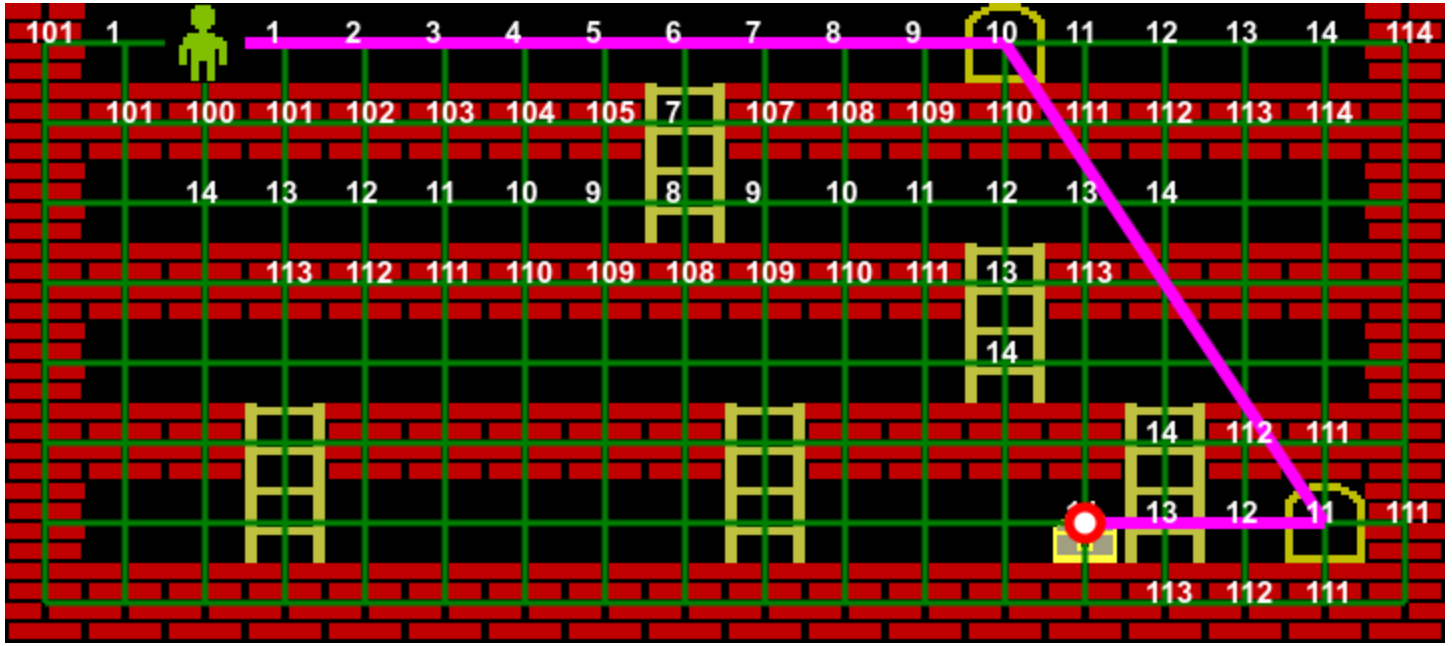
5. ОГРАНИЧЕНЫ ЛИ МЫ 2D-ПРОСТРАНСТВОМ?



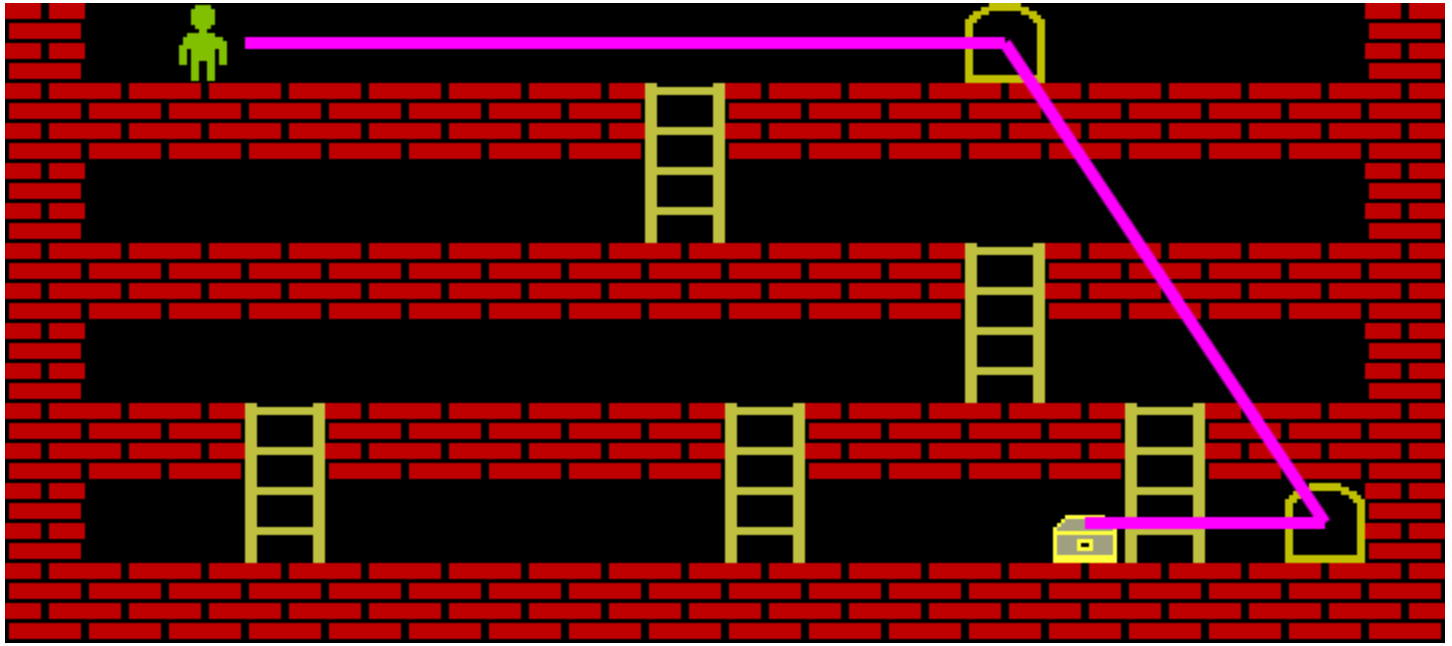
5. ТЕЛЕПОРТАЦИЯ И A*: НЕ НАШЛИ КРАТЧАЙШИЙ ПУТЬ



5. ВОЗВРАЩАЕМСЯ К АЛГОРИТМУ ДЕЙКСТРЫ

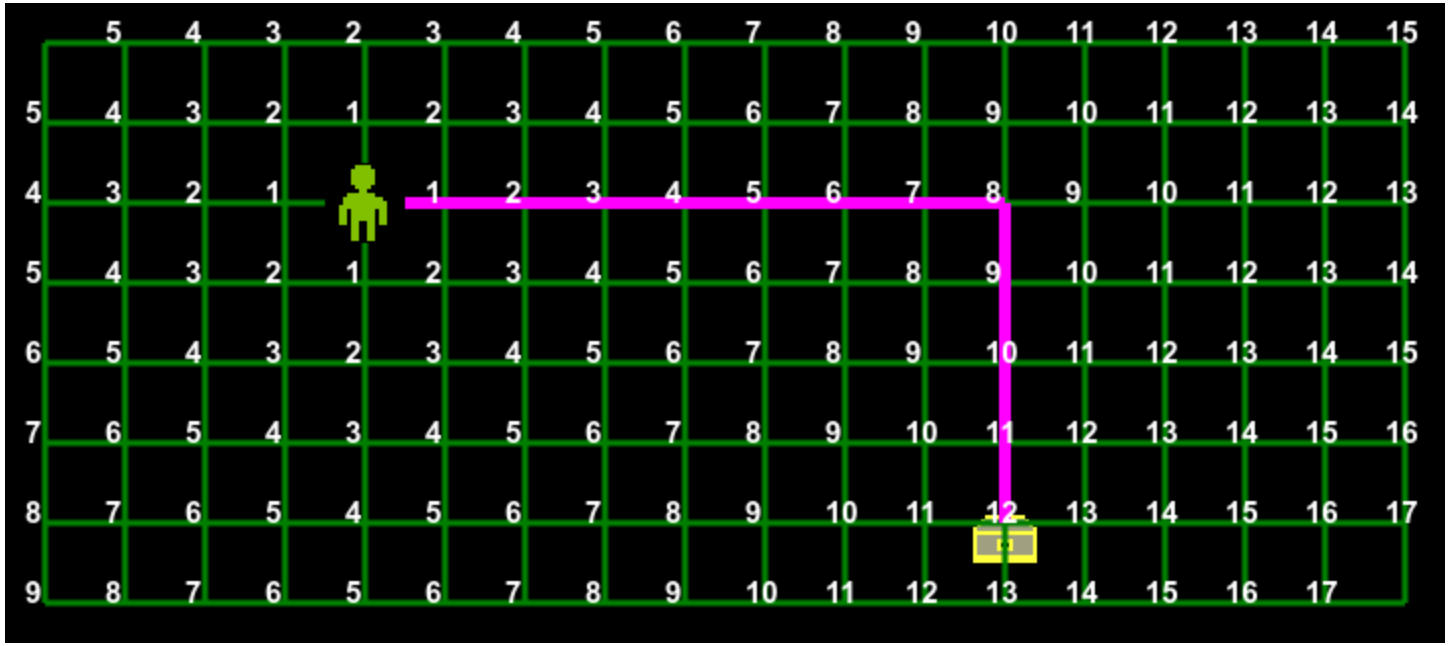


5. ТЕЛЕПОРТАЦИЯ: КАК ЭТО РАБОТАЕТ



СОДЕРЖАНИЕ

6. АЛГОРИТМ ДЕЙКСТРЫ. РЕЗУЛЬТАТ РАСЧЕТА ГРАФА



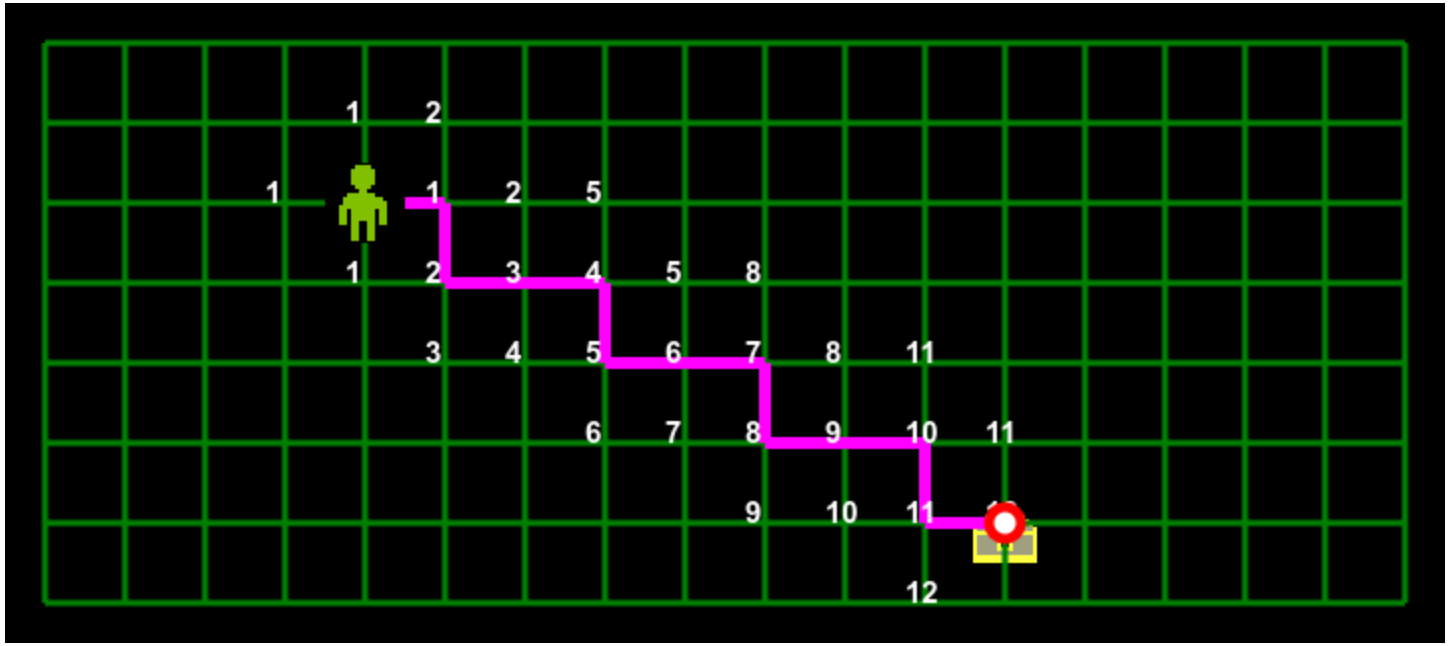
6. T(N) – ВРЕМЯ РАСЧЕТА

ИННОТЕХ

Реализация алгоритма Дейкстры

N, сторона квадрата	N2, узлов в графе	T, миллисекунд
10	100	3
100	10 000	557
1 000	1 000 000	51 555

6. A* С ЭВРИСТИКОЙ H4()



6. T(N) – ВРЕМЯ РАСЧЕТА

ИННОТЕХ

Реализация алгоритма Дейкстры

N, сторона квадрата	N2, узлов в графе	T, миллисекунд
10	100	3
100	10 000	557
1 000	1 000 000	51 555

Реализация алгоритма A*

N, сторона квадрата	N2, узлов в графе	T, миллисекунд
10	100	2
100	10 000	14
1 000	1 000 000	835
10 000	100 000 000	*out of memory

СОДЕРЖАНИЕ

ВЫВОДЫ

1. Алгоритм Дейкстры проще в реализации, чем A*
2. A* эффективнее, чем алгоритм Дейкстры
3. Можно создавать свои игры на основе примера: https://github.com/alexanderpono/bricks-runner
4. Не бойтесь экспериментировать. Преимущество фронтенда – можно сразу увидеть результат!

ЧТО ЕЩЕ ПОЧИТАТЬ?

- A* – развитие алгоритма Дейкстры
  https://www.redblobgames.com/pathfinding/a-star/introduction.html
- Еще об алгоритмах на графах
  https://habr.com/ru/companies/timeweb/articles/751762/

И НАКОНЕЦ...

СПАСИБО ЗА ВНИМАНИЕ!
ВОПРОСЫ?

ИННОТЕХ

Александр Пономаренко

Описание A*

Алгоритмы на графах

Исходники игры

Новости ИТ в Иннотехе и Т1