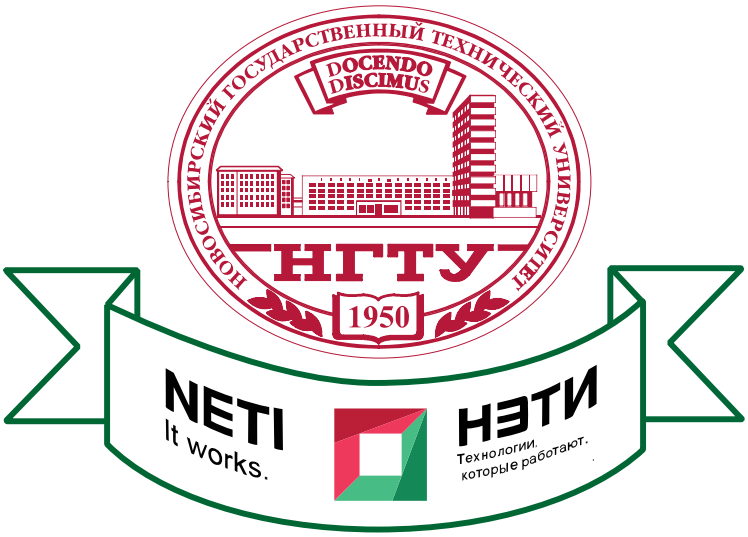
Министерство науки и высшего образования
Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение высшего образования

«Новосибирский государственный технический университет»

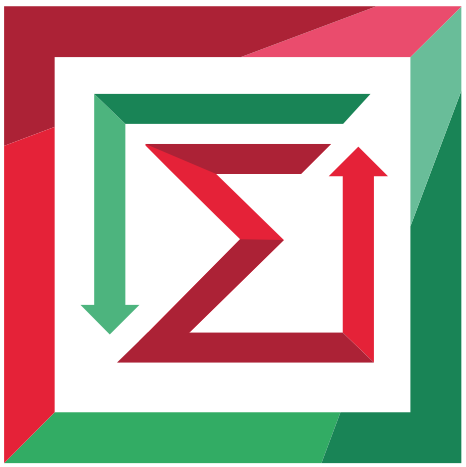


Теоретической и прикладной информатики

Лабораторная работа № 1

по дисциплине «Интеллектуальные системы»

Стратегии решения задач и программирование игр



Факультет:	ПМИ
Группа:	ПМИ-02
Студент:	Сидоров Даниил,
	Дюков Богдан
Преподаватель:	Дворецкая Виктория Константиновна

Новосибирск

2026

1. **Цель работы**

Изучение основных стратегий решения задач. Приобретение навыков выбора адекватных стратегий в зависимости от типа задач. Выбор инструмента для реализации этих стратегий. Применение базовых стратегий решения задач для программирования игр двух лиц с полной информацией.

2. **Задание**

1. Сформулируйте задачу, выбрав комбинацию из следующих подходящих стратегий решения задачи:

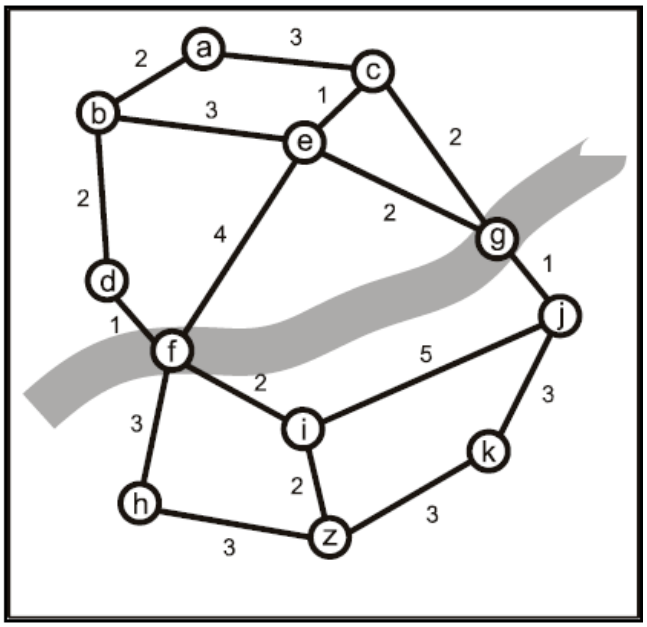
- представление в пространстве состояний;
- сведения задач к подзадачам;
- поиск в глубину с возвратом;
- поиск в ширину;
- поиск с предпочтением (эвристический поиск);
- выбор подходящей стратегии обхода дерева при реализации игры двух лиц с полной информацией.

2. Разработайте адекватную структуру данных, максимально учитывающую специфику предметной области задачи. Обоснуйте выбор структуры. В случае представления задачи с помощью пространства состояний нарисуйте несколько первых уровней графа переходов. В случае сведения задач к подзадачам - несколько уровней И/ИЛИ дерева.

3. Реализуйте формальное описание проблемы на Прологе, снабдив программу достаточным количеством средств ввода-вывода для наглядного отображения результатов.

3. **Вариант задания**

Пусть требуется найти маршрут из a в z на карте дорог, каждая из которых имеет свою стоимость:



На карте, как видно, присутствует река, и путь через нее лежит только через пункты f и g. Таким образом, искомый маршрут должен непременно проходить через один из этих двух пунктов. Найти путь с минимальной стоимостью.

4. Решение

Для решения нашей задачи представим нашу карту дорог в виде графа. Пункты представим, как вершины графа, а дороги ребрами. Соответственно стоимость дорог будет представлена, как вес ребер. Для нахождения кратчайшего пути от одной вершины графа до другой будем использовать алгоритм Дейкстры.

Алгоритм работы Дейкстры в общем виде:

Каждой вершине сопоставим метку — минимальное известное расстояние от этой вершины до ***a*** (стартовая вершина).

Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки.

Работа алгоритма завершается, когда все вершины посещены или достигнута заданная конечная вершина.

Инициализация.

Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности.

Это отражает то, что расстояния от a до других вершин пока неизвестны.

Все вершины графа помечаются как непосещённые.

Шаг алгоритма.

Если все вершины посещены или достигнута заданная конечная вершина, алгоритм завершается.

В противном случае, из ещё не посещённых вершин выбирается вершина ***u***, имеющая минимальную метку.

Мы рассматриваем всевозможные маршруты, в которых ***u*** является предпоследним пунктом. Вершины, в которые ведут рёбра из ***u***, назовём соседями этой вершины. Для каждого соседа вершины ***u***, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки ***u*** и длины ребра, соединяющего ***u*** с этим соседом.

Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину ***u*** как посещённую и повторим шаг алгоритма.

5. Листинг программы

```
% Функция для поиска кратчайшего пути в графе
result(Start, End, Path, Cost) :-
    dijkstra([(0, [Start], Start)], End, ReversedPath, Cost),
    reverse(ReversedPath, Path).

% Алгоритм Дейкстры
dijkstra([(Cost, Path, End) | _], End, Path, Cost).
dijkstra([(CurrentCost, CurrentPath, CurrentEnd) | Rest], End, Path, Cost) :-
    findall((NewCost, [Next | CurrentPath], Next),
            (edge(CurrentEnd, Next, EdgeCost),
             \+ member(Next, CurrentPath),
             NewCost is CurrentCost + EdgeCost),
            NextSteps),
    append(Rest, NextSteps, UpdatedQueue),
    sort(UpdatedQueue, SortedQueue),
```

```
    dijkstra(SortedQueue, End, Path, Cost).

% Граф
edge(a,b,2).
edge(a,c,3).
edge(b,e,3).
edge(c,e,1).
edge(b,d,2).
edge(e,f,4).
edge(e,g,2).
edge(c,g,2).
edge(d,f,1).
edge(f,h,3).
edge(f,i,2).
edge(g,j,1).
edge(i,j,5).
edge(i,z,2).
edge(j,k,3).
edge(h,z,3).
edge(k,z,3).
% В обратную сторону
edge(b,a,2).
edge(c,a,3).
edge(e,b,3).
edge(e,c,1).
edge(d,b,2).
edge(f,e,4).
edge(g,e,2).
edge(g,c,2).
edge(f,d,1).
edge(h,f,3).
edge(i,f,2).
edge(j,g,1).
edge(j,i,5).
edge(z,i,2).
edge(k,j,3).
edge(z,h,3).
edge(z,k,3).
```

6. Тестирование

```
?- result(a,i,Path,Cost).
Path = [a, b, d, f, i],
Cost = 7 .

?- result(a,z,Path,Cost).
Path = [a, b, d, f, i, z],
Cost = 9 .

?- result(z,a,Path,Cost).
Path = [z, i, f, d, b, a],
Cost = 9 .

?- result(a,u,Path,Cost).
false.

?- result(a,a,Path,Cost).
Path = [a],
Cost = 0
```

7. Правила

1. dijkstra([(Cost, Path, End) | _], End, Path, Cost) - это базовый случай, когда достигнута конечная вершина. Если текущая вершина End соответствует конечной вершине, то Path и Cost принимают значения из текущего состояния (текущий путь и текущая стоимость).

2. dijkstra([(CurrentCost, CurrentPath, CurrentEnd) | Rest], End, Path, Cost) - это правило осуществляет шаг алгоритма. Принимается текущее состояние, включающее стоимость CurrentCost, текущий путь CurrentPath и текущую вершину CurrentEnd.

3. findall((NewCost, [Next | CurrentPath], Next), (edge(CurrentEnd, Next, EdgeCost), \+ member(Next, CurrentPath), NewCost is CurrentCost + EdgeCost), NextSteps) - используется для поиска всех возможных следующих шагов из текущей вершины. edge(CurrentEnd, Next, EdgeCost) проверяет наличие ребра между текущей вершиной и следующей вершиной, а \+ member(Next, CurrentPath) убеждается, что следующая вершина не содержится в текущем пути. Затем вычисляется новая стоимость NewCost путем добавления веса ребра к текущей стоимости.

4. append(Rest, NextSteps, UpdatedQueue) - объединяет текущий список состояний Rest с новыми состояниями NextSteps, формируя обновленную очередь состояний UpdatedQueue.

5. sort(UpdatedQueue, SortedQueue) - сортирует обновленную очередь по возрастанию стоимости. Это важно для выбора наименьшей стоимости на следующем шаге.

6. dijkstra(SortedQueue, End, Path, Cost) - Рекурсивный вызов алгоритма с обновленной отсортированной очередью. Алгоритм продолжает выполнение с новым состоянием.