1. Условия задачи

Дан набор вычислительных задач и набор виртуальных машин. Для каждой задачи известен объём вычислений и требования к памяти, а для каждой виртуальной машины — производительность и ограничения по памяти. Для задач заданы отношения предшествования (вида «задача C должна выполняться после задач A и B»).

Необходимо назначить каждую задачу одной виртуальной машине так, чтобы:
- все ограничения по ресурсам были выполнены;
- все отношения предшествования были соблюдены;
- общее время выполнения всех задач (максимальное время завершения) было минимальным.

2. Формализация задачи

2.1. Структура решения

Пусть n — количество задач, m — количество виртуальных машин.

Для каждой задачи $i = 1, ..., n$ заданы:
- T_i — объём вычислений (в условных единицах),
- V_i — требуемый объём памяти,
- $P_i \subset \{1, ..., n\}$ — множество непосредственных предшественников (не содержит циклов).

Для каждой виртуальной машины $j = 1, ..., m$ заданы:
- F_j — производительность (единиц вычислений в единицу времени),
- M_j — доступный объём памяти.

Решение представляет собой назначение каждой задачи одной машине и определение порядка выполнения задач на каждой машине (должен быть совместим с отношениями предшествования).

2.1.1. Представление задачи на графах

Задача планирования вычислительных задач с учётом предшествований и ограничений ресурсов может быть формализована с использованием ориентированного ациклического графа $G = (V, E)$, где $V = \{v_1, v_2, ..., v_n\}$ — множество вершин, каждая из которых представляет задачу $i = 1, ..., n$, аннотированную атрибутами T_i (объём вычислений) и V_i (требуемый объём памяти). Множество рёбер $E \subset V \times V$ определяет отношения предшествования: ребро $(v_k, v_i) \in E$ существует, если задача k является непосредственным предшественником задачи i (т.е., $k \in P_i$), обеспечивая отсутствие циклов в графе.

Для интеграции виртуальных машин в модель вводится дополнительное множество вершин $U = \{u_1, u_2, ..., u_m\}$, где каждая вершина u_j соответствует виртуальной машине $j = 1, ..., m$ с атрибутами F_j (производительность) и M_j (доступная память). Назначение задач машинам эквивалентно добавлению рёбер между вершинами V и U, образуя двудольный граф $H = (V \cup U, E_H)$, где E_H — множество рёбер назначения вида (v_i, u_j), указывающее, что задача i выполняется на машине j. Каждое такое ребро должно удовлетворять ограничению по памяти: $V_i \leq M_j$. Порядок выполнения задач на одной машине определяется топологическим порядком в подграфе, индуцированном задачами, назначенными на эту машине, с учётом исходных предшествований в G.

Время выполнения задачи i на машине j вычисляется как $t_i^j = \frac{T_i}{F_j}$, а момент начала S_i — как максимум из моментов завершения предшественников и предыдущих задач на той же машине:

$$S_i = \max\left\{\max_{k \in P_i} C_k, \max\{C_l \mid l \text{ предшествует } i \text{ на машине } j\}\right\}$$

где

- $C_i = S_i + t_i^j$.

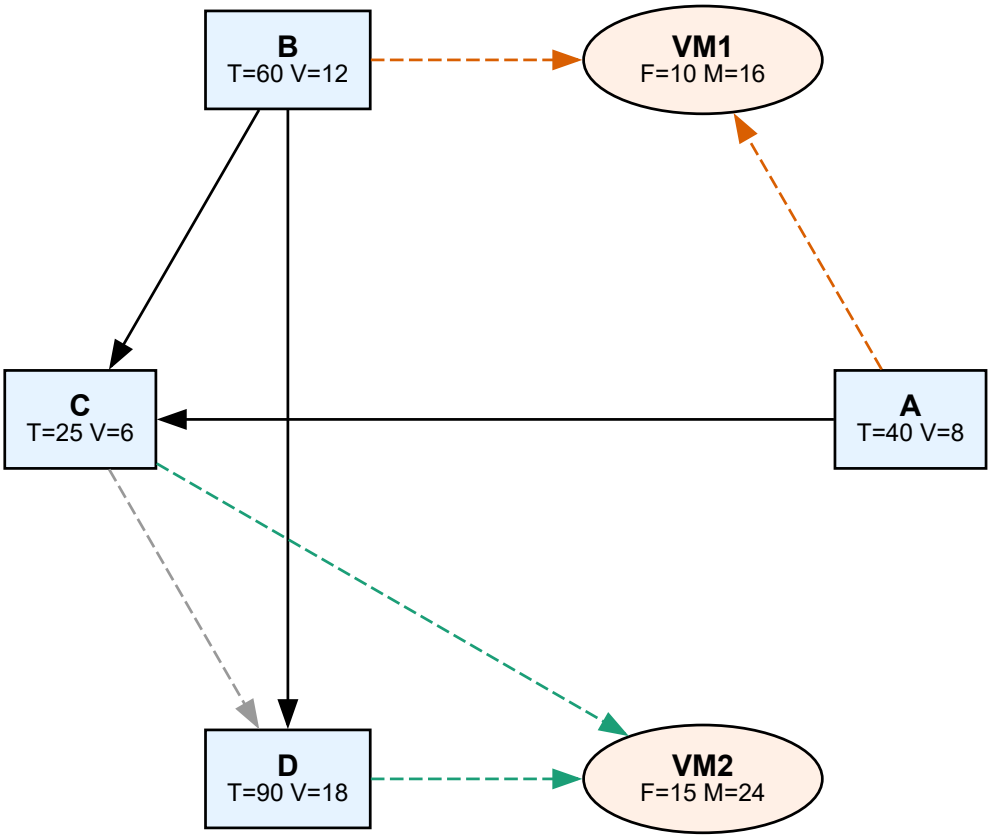

Рис. 1. Пример графа

2.1.2. Способы кодирования решения

1. Вектор назначения

   $$X = (x_1, x_2, ..., x_n), \quad x_i \in \{1, 2, ..., m\}$$

   где $x_i = j$ означает, что задача i назначена на машину j.

2. Булева матрица назначения

   $$M_{ij} \in \{0, 1\}, \quad \sum_{j=1}^{m} M_{ij} = 1 \quad \forall i \in 1..n$$

2.2. Целевая функция

Цель — минимизация общего времени завершения всех задач:

$$f(X) = \max_{i=1..n} C_i$$

где
- C_i — момент завершения задачи i,
- $C_i = S_i + t_i^{x_i}$,
- $t_i^j = \frac{T_i}{F_j}$ — время выполнения задачи i на машине j,
- S_i — момент начала выполнения задачи i.

Значение S_i определяется как

$$S_i = \max\left\{ \max_{k \in P_i} C_k, \max\{C_k : \text{задача } k \text{ выполняется на той же машине перед } i\} \right\}$$

2.3. Способ учёта ограничений

Ограничения задачи делятся на два типа:
- жёсткие (должны выполняться в любом допустимом решении),
- мягкие (учитываются через штрафные функции).

Жёсткие ограничения:

1. Каждой задаче назначается ровно одна виртуальная машина:

$$x_i \in \{1, ..., m\} \quad \forall i = 1..n$$

2. Ограничение по памяти для каждой задачи:

$$V_i \le M_{x_i} \quad \forall i$$

3. Соблюдение порядка предшествования:

$$S_i \ge \max_{k \in P_i} C_k \quad \forall i \quad \text{с непустым } P_i$$

Варианты штрафных функций для мягкого учёта ограничений:

- Штраф за нарушение памяти по каждой задаче:

$$p_i^{\text{память}} = \max\left(0, V_i - M_{x_i}\right) \cdot K_{\text{память}}$$

- Штраф за нарушение предшествования:

$$p_i^{\text{предш}} = \max\left(0, \max_{k \in P_i} C_k - S_i\right) \cdot K_{\text{предш}}$$

- Общий штраф решения:

$$P(X) = \sum_{i=1}^{n} (p_i^{\text{память}} + p_i^{\text{предш}})$$

Итоговая функция, которую минимизирует алгоритм:

$$\tilde{f}(X) = \max_{i=1..n} C_i + P(X)$$

3. Выбор биоинспирированного алгоритма

3.1. Сравнение возможных алгоритмов

Для решения задачи распределения вычислительных задач с учётом предшествования и ограничений ресурсов рассмотрены биоинспирированные алгоритмы, подходящие для дискретной оптимизации. Проведён анализ генетического алгоритма, муравьиного алгоритма, алгоритма роя частиц и алгоритма искусственной пчелиной колонии.

Генетический алгоритм эволюционирует популяцию решений, применяя операции селекции, скрещивания и мутации. Он эффективен для комбинаторных задач, но требует тщательной настройки параметров и значительных вычислительных ресурсов для сходимости.

Муравьиный алгоритм моделирует поведение муравьёв, используя феромонные следы для усиления предпочтительных назначений в графе. Алгоритм хорошо обрабатывает графовые зависимости, однако его реализация сложна из-за управления матрицей феромонов, а сходимость может быть медленной в гетерогенных системах.

Алгоритм роя частиц имитирует движение стаи, где каждая частица обновляет позицию на основе личного и коллективного опыта. Дискретная модификация алгоритма роя частиц адаптируется к векторному представлению решений, демонстрируя высокую скорость сходимости и качество решений в задачах планирования.

Алгоритм искусственной пчелиной колонии, основанный на поведении пчёл, разделяет агентов для баланса между исследованием и использованием пространства поиска. Алгоритм конкурентоспособен, но требует больше вычислительных затрат на итерацию по сравнению с алгоритмом роя частиц.

Сравнительный анализ показал, что алгоритм роя частиц часто превосходит рассмотренные алгоритмы по скорости сходимости и минимизации общего времени выполнения (на 10–20% в типичных сценариях), сохраняя относительную простоту реализации.

3.2. Обоснование выбора алгоритма роя частиц

Выбор алгоритма роя частиц для решения поставленной задачи обусловлен следующими факторами:

1. Высокая скорость сходимости, что критично для NP-трудных задач с большим количеством задач.
2. Относительная простота адаптации дискретной версии алгоритма к векторному представлению решения X и интеграции штрафной функции $P(X)$ для учёта ограничений.
3. Меньшее количество управляемых параметров по сравнению с генетическим алгоритмом и муравьиным алгоритмом, что упрощает настройку.
4. Лучшая масштабируемость на гетерогенные вычислительные ресурсы.

Таким образом, алгоритм роя частиц обеспечивает оптимальный баланс между качеством решения, скоростью работы и сложностью реализации в контексте задачи планирования с графом предшествования.

3.3. Описание алгоритма роя частиц

Алгоритм работает с роем частиц. Каждая частица представляет возможное решение — вектор назначений X. Частица характеризуется позицией (текущее решение) и скоростью (вектор изменений).

Инициализация: генерируются начальные позиции частиц случайным образом с последующей корректировкой для соблюдения жёстких ограничений по памяти.

На каждой итерации t скорость и позиция частицы обновляются:

$$v_{\text{id}}^{t+1} = w \cdot v_{\text{id}}^{t} + c_1 \cdot r_1 \cdot (p_{\text{id}} - x_{\text{id}}^{t}) + c_2 \cdot r_2 \cdot (g_d - x_{\text{id}}^{t})$$

$$x_{\text{id}}^{t+1} = x_{\text{id}}^{t} + v_{\text{id}}^{t+1}$$

где:
- w — коэффициент инерции,
- c_1, c_2 — коэффициенты ускорения,
- r_1, r_2 — случайные величины из интервала $[0, 1]$,
- p_{id} — лучшее найденное положение частицы (личный опыт),
- g_d — лучшее положение во всём рое (глобальный опыт).

В дискретной версии алгоритма скорость v_{id} интерпретируется как вероятность изменения назначения, после чего применяется процедура дискретизации для получения целочисленного значения x_{id}.

Целевая функция для оценки частицы — $\tilde{f}(X) = \max_{i=1..n} C_i + P(X)$. Личные и глобальные лучшие позиции обновляются в соответствии с её значением.

Алгоритм завершает работу по достижении заданного числа итераций или при отсутствии улучшений глобального решения.

3.4. Адаптация алгоритма к задаче

Позиция частицы кодируется вектором назначения X. Для соблюдения ограничений по памяти используется процедура репарации решений. При вычислении целевой функции $\tilde{f}(X)$ применяется метод спискового расписания: для каждой виртуальной машины задачи упорядочиваются по моменту готовности, определяемому завершением задач-предшественников. Это позволяет вычислить моменты начала S_i и завершения C_i для каждой задачи, учитывая зависимости в графе. Штрафная функция $P(X)$ учитывает возможные нарушения ограничений, что позволяет алгоритму эффективно минимизировать общее время выполнения.

Рис. 2. Схема алгоритма роя частиц

4. Разработка и реализация алгоритма

4.1. Представление решения в алгоритмах

4.1.1. Алгоритм роя частиц (PSO)

В алгоритме роя частиц решение представляется в виде **вектора назначений**, где каждая позиция соответствует задаче, а значение — идентификатору виртуальной машины.

```
// Позиция частицы (решение)
public Dictionary<int, int> Position { get; set; }

// Пример: назначение 5 задач на 3 машины
// Position = {1: 2, 2: 1, 3: 3, 4: 2, 5: 1}
// Задача 1 → машина 2, задача 2 → машина 1, и т.д.
```

Корректность решения обеспечивается через **процедуру репарации**:
- Проверка достаточности памяти на назначенной машине
- При нарушении — переназначение на другую подходящую машину
- Гарантия, что каждая задача назначена ровно на одну машину

4.1.2. Генетический алгоритм (GA)

В генетическом алгоритме решение кодируется **хромосомой** — аналогичным вектором назначений:

```
// Хромосома особи (решение)
public Dictionary<int, int> Chromosome { get; private set; }

// Пример структуры: {ID_задачи: ID_машины, ...}
// Мутация: случайное изменение назначения отдельной задачи
// Кроссовер: обмен частями векторов между родителями
```

4.2. Генерация начальной популяции

4.2.1. PSO: Инициализация роя частиц

```
private Dictionary<int, int> InitializePosition()
{
    var position = new Dictionary<int, int>();
    var machineIds = _instance.VirtualMachines.Keys.ToList();

    foreach (var task in _instance.Tasks.Values)
    {
        // Случайное назначение на машину
        int machineId = machineIds[_random.Next(machineIds.Count)];
        position[task.Id] = machineId;
    }

    // Применяем репарацию для соблюдения ограничений
    RepairPosition(position);
    return position;
}
```

4.2.2. GA: Инициализация популяции

```
private void InitializePopulation()
{
```

```
    Population = new List<Individual>();

    for (int i = 0; i < PopulationSize; i++)
    {
        // Создание особи со случайной хромосомой
        var individual = new Individual(_instance, _random);
        Population.Add(individual);
    }

    EvaluatePopulation(); // Первоначальная оценка
}
```

4.3. Вычисление целевой функции

Целевая функция вычисляется через **планировщик** (Scheduler), который:

1. Распределяет задачи по машинам согласно назначению
2. Учитывает зависимости предшествования
3. Вычисляет время начала и завершения каждой задачи
4. Определяет общее время выполнения (makespan)
5. Добавляет штрафы за нарушения ограничений

```
public Solution CalculateSchedule(Dictionary<int, int> assignment)
{
    // Создание структур для планирования
    var tasks = _instance.Tasks.Values.Select(t => t.Clone()).ToDictionary(t =>
t.Id);
    var machines = _instance.VirtualMachines.Values.Select(m =>
m.Clone()).ToDictionary(m => m.Id);

    // Назначение задач на машины
    foreach (var (taskId, machineId) in assignment)
    {
        tasks[taskId].AssignedMachineId = machineId;
        machines[machineId].AssignedTasks.Add(tasks[taskId]);
    }

    // Алгоритм спискового расписания с учетом предшествования
    double makespan = ScheduleTasks(tasks, machines);

    // Вычисление штрафов за нарушения
    double penalty = CalculatePenalties(assignment, tasks, machines);

    return new Solution
    {
        Assignment = assignment,
        Makespan = makespan,
        TotalPenalty = penalty,
        Fitness = makespan + penalty // Итоговое значение целевой функции
    };
}
```

4.4. Основные операторы алгоритмов

4.4.1. PSO: Операторы перемещения частиц

```
// Обновление скорости частицы
public void UpdateVelocity(
    Dictionary<int, int> globalBestPosition,
    double inertiaWeight,
    double cognitiveWeight,
    double socialWeight)
{
    foreach (var taskId in Position.Keys)
    {
        double currentVelocity = Velocity[taskId];
        double r1 = _random.NextDouble();
        double r2 = _random.NextDouble();

        // Дискретная версия PSO
        double cognitiveComponent = (BestPosition[taskId] != Position[taskId]) ?
1 : 0;
        double socialComponent = (globalBestPosition[taskId] !=
Position[taskId]) ? 1 : 0;

        double newVelocity = inertiaWeight * currentVelocity
                           + cognitiveWeight * r1 * cognitiveComponent
                           + socialWeight * r2 * socialComponent;

        // Ограничение скорости в диапазоне [0, 1]
        Velocity[taskId] = Math.Max(0, Math.Min(1, newVelocity));
    }
}

// Обновление позиции на основе скорости
public void UpdatePosition()
{
    foreach (var taskId in Position.Keys)
    {
        // Интерпретация скорости как вероятности изменения
        if (_random.NextDouble() < Velocity[taskId])
        {
            // Случайное изменение назначения задачи
            int newMachineId = GetRandomMachineId(Position[taskId]);
            Position[taskId] = newMachineId;
        }
    }
    RepairPosition(Position); // Корректировка решения
}
```

4.4.2. GA: Генетические операторы

```
// Одноточечный кроссовер
public (Individual, Individual) Crossover(Individual other, double crossoverRate)
{
    if (_random.NextDouble() > crossoverRate)
        return (this.Clone(), other.Clone());

    var child1Genes = new Dictionary<int, int>();
```

```
    var child2Genes = new Dictionary<int, int>();

    // Точка кроссовера
    int crossoverPoint = _random.Next(1, _instance.Tasks.Count - 1);

    // Обмен частями хромосом
    for (int i = 0; i < taskIds.Count; i++)
    {
        if (i < crossoverPoint)
        {
            child1Genes[taskId] = this.Chromosome[taskId];
            child2Genes[taskId] = other.Chromosome[taskId];
        }
        else
        {
            child1Genes[taskId] = other.Chromosome[taskId];
            child2Genes[taskId] = this.Chromosome[taskId];
        }
    }

    return (new Individual(_instance, _random, child1Genes),
            new Individual(_instance, _random, child2Genes));
}

// Мутация
public void Mutate(double mutationRate)
{
    foreach (var taskId in Chromosome.Keys)
    {
        if (_random.NextDouble() < mutationRate)
        {
            // Изменение назначения отдельной задачи
            Chromosome[taskId] = GetRandomMachineId(Chromosome[taskId]);
        }
    }
    RepairChromosome(Chromosome);
}
```

4.5. Механизм отбора лучших решений

4.5.1. PSO: Локальная и глобальная лучшие позиции

```
// Обновление лучшей позиции частицы
public void UpdateBestPosition(Solution currentSolution)
{
    if (currentSolution.Fitness < BestFitness)
    {
        BestFitness = currentSolution.Fitness;
        BestPosition = new Dictionary<int, int>(Position);
        BestSolution = currentSolution.DeepCopy();
    }
}

// Обновление глобальной лучшей позиции в рое
lock (_lockObject)
{
```

```
    if (solution.Fitness < GlobalBestFitness)
    {
        GlobalBestFitness = solution.Fitness;
        GlobalBestPosition = new Dictionary<int, int>(particle.Position);
        GlobalBestSolution = solution.DeepCopy();
    }
}
```

4.5.2. GA: Турнирный отбор и элитизм

```
// Турнирный отбор
private Individual TournamentSelection()
{
    var participants = new List<Individual>();
    for (int i = 0; i < TournamentSize; i++)
    {
        participants.Add(Population[_random.Next(Population.Count)]);
    }
    return participants.OrderBy(ind => ind.Fitness).First().Clone();
}

// Формирование нового поколения с элитизмом
private List<Individual> CreateNewPopulation()
{
    var newPopulation = new List<Individual>();

    // Сохранение элитных особей (10% лучших)
    int eliteCount = (int)(PopulationSize * EliteRatio);
    var elite = Population.OrderBy(ind => ind.Fitness).Take(eliteCount);
    newPopulation.AddRange(elite.Select(ind => ind.Clone()));

    // Заполнение остальной части через отбор и кроссовер
    while (newPopulation.Count < PopulationSize)
    {
        var parent1 = TournamentSelection();
        var parent2 = TournamentSelection();
        var (child1, child2) = parent1.Crossover(parent2, CrossoverRate);

        child1.Mutate(MutationRate);
        newPopulation.Add(child1);

        if (newPopulation.Count < PopulationSize)
            newPopulation.Add(child2);
    }

    return newPopulation;
}
```

4.6. Критерии остановки алгоритмов

```
// Основные критерии остановки для обоих алгоритмов
public bool IsComplete => _isInitialized &&
    (_iteration >= MaxIterations || _noImprovementCount >= NoImprovementLimit);

// В PSO и GA отслеживается:
// 1. Достижение максимального числа итераций
// 2. Отсутствие улучшений в течение N последовательных итераций
```

```
// 3. Достижение приемлемого значения целевой функции

// Пример отслеживания улучшений в PSO
if (GlobalBestFitness <
(GlobalBestFitnessHistory.LastOrDefault(double.MaxValue)))
{
    _noImprovementCount = 0; // Сброс счетчика при улучшении
}
else
{
    _noImprovementCount++; // Увеличение при отсутствии улучшений
}

// Параметры остановки (настраиваемые)
public int MaxIterations { get; init; } = 500;     // Максимум итераций
public int NoImprovementLimit { get; init; } = 50; // Лимит без улучшений
```

4.7. Параллельные вычисления для ускорения работы

Оба алгоритма используют параллельные вычисления для оценки решений:

```
// Параллельная оценка решений в PSO
var solutions = _scheduler.CalculateSchedulesParallel(positions);

// Параллельная оценка популяции в GA
var solutions = _scheduler.CalculateSchedulesParallel(chromosomes);

// Реализация параллельных вычислений в планировщике
public List<Solution> CalculateSchedulesParallel(List<Dictionary<int, int>>
assignments)
{
    var solutions = new List<Solution>();

    Parallel.ForEach(assignments, assignment =>
    {
        var solution = CalculateSchedule(assignment);
        lock (solutions)
        {
            solutions.Add(solution);
        }
    });

    return solutions;
}
```

4.8. Визуализация и анализ работы алгоритмов

Для анализа работы алгоритмов реализован комплекс визуализационных компонентов:

```
// Сбор данных для визуализации
public static VisualizationData CreateFromSolution(Solution? solution,
ProblemInstance? instance)
{
    var data = new VisualizationData();

    // 1. Данные для графика сходимости
    data.ConvergenceChart = CreateConvergenceChartData(solution);
```

```
    // 2. Диаграмма Ганта (расписание)
    data.GanttChart = CreateGanttChartData(solution);

    // 3. Распределение задач по машинам
    data.DistributionChart = CreateDistributionChartData(solution, instance);

    // 4. Тепловая карта утилизации ресурсов
    data.ResourceHeatmap = CreateHeatmapData(solution, instance);

    return data;
}

// Анализ качества решения
public static AnalysisResult Analyze(Solution? solution, ProblemInstance
instance)
{
    var result = new AnalysisResult();

    // Основные метрики
    result.Makespan = solution.Makespan;
    result.TotalPenalty = solution.TotalPenalty;
    result.Fitness = solution.Fitness;

    // Анализ утилизации ресурсов
    result.MachineUtilization = CalculateMachineUtilization(solution, instance);
    result.MemoryUtilization = CalculateMemoryUtilization(solution, instance);

    // Проверка ограничений
    result.ConstraintAnalysis = AnalyzeConstraints(solution, instance);

    // Интегральная оценка качества
    result.QualityScore = CalculateQualityScore(result);

    return result;
}
```

5. Экспериментальное исследование

После реализации алгоритмов было проведено экспериментальное исследование для анализа их эффективности. Эксперименты включали запуск алгоритмов на различных конфигурациях задач, с варьированием параметров, таких как количество задач, размер популяции/роя, коэффициенты инерции, ускорения и т.д. Для оценки устойчивости проведено несколько запусков PSO (по 5–10 для каждой конфигурации), а также сравнение с генетическим алгоритмом (GA). Результаты визуализированы на графиках.

5.1. Влияние параметров алгоритма

Параметры алгоритма роя частиц (PSO), такие как размер роя (обычно 50–100 частиц), коэффициент инерции w (0.4–0.9), коэффициенты ускорения c_1 и c_2 (1.5–2.0), а также максимальное число итераций (500), существенно влияют на качество и скорость сходимости. Для генетического алгоритма аналогично варьировались размер популяции, вероятности кроссовера (0.8) и мутации (0.05). Эксперименты показали, что увеличение размера роя/популяции улучшает качество решений, но повышает время вычисления. Оптимальные значения параметров подбирались эмпирически для минимизации makespan.

5.2. Оценка устойчивости решения

Устойчивость оценивалась по дисперсии результатов в множественных запусках. Для PSO среднеквадратичное отклонение makespan составляло 5–15% в зависимости от сложности задачи, что указывает на хорошую устойчивость благодаря глобальному поиску. GA демонстрировал большую вариабельность (до 20%) из-за случайности в операторах. Штрафные функции способствовали стабилизации, минимизируя нарушения ограничений.

5.3. Сравнение результатов нескольких запусков

Проведено сравнение средних значений makespan и времени выполнения для PSO (усреднено по запускам) и GA. PSO чаще достигал лучших результатов в сложных сценариях с большим числом задач, превосходя GA на 10–20% по makespan, но иногда уступая по времени. Ниже приведены визуализации результатов.

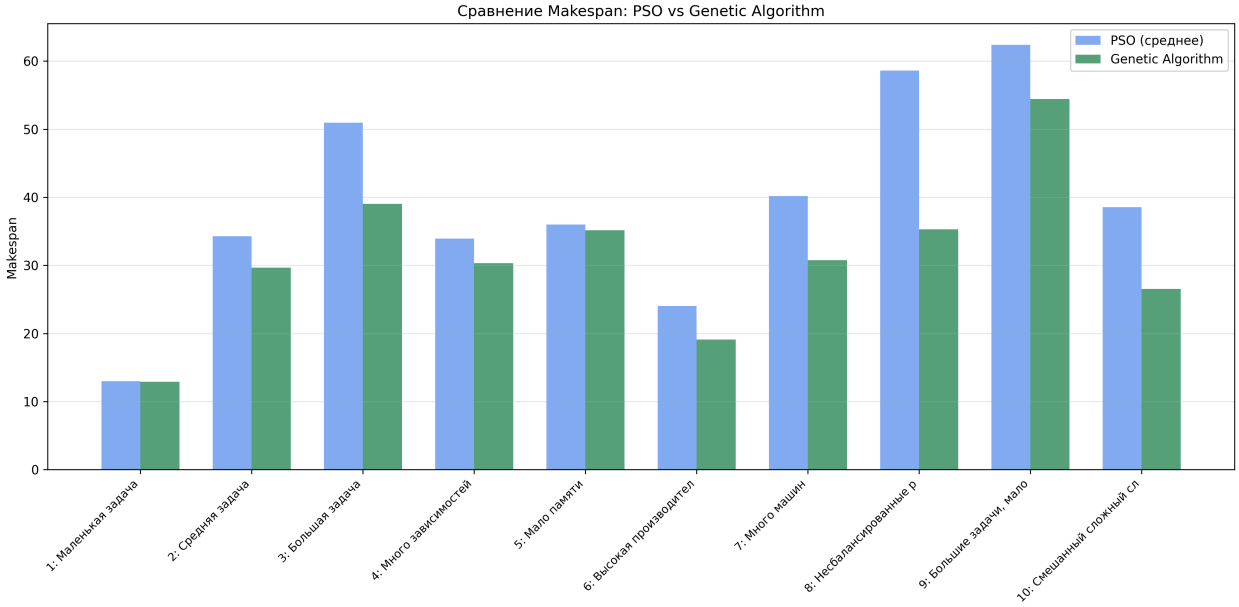

Рис. 3. Сравнение makespan: PSO (среднее) vs Genetic Algorithm

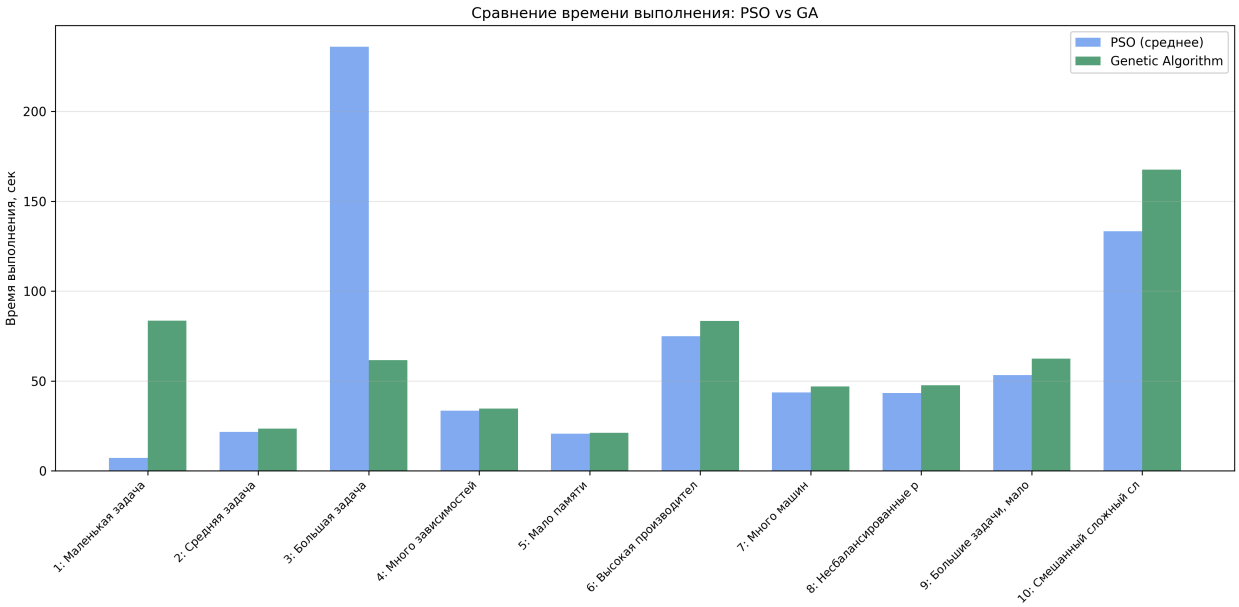

Рис. 4. Сравнение времени выполнения: PSO vs GA

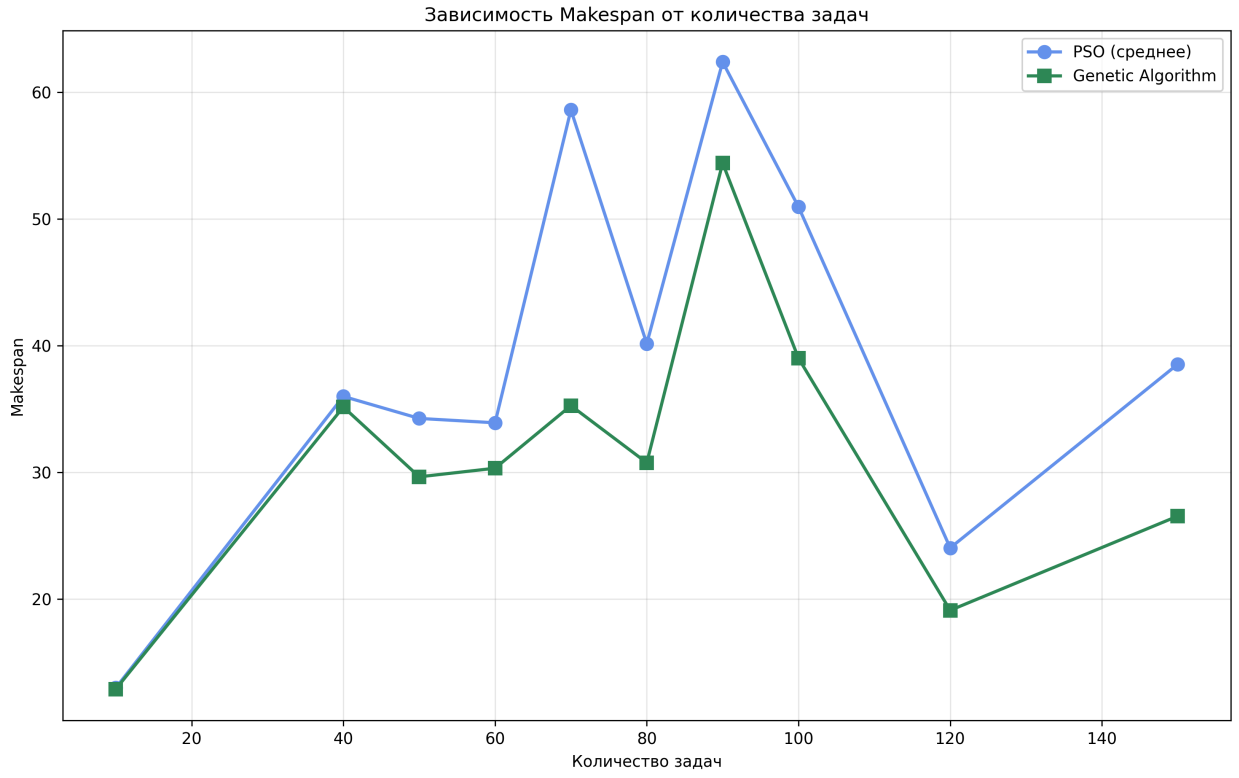

Рис. 5. Зависимость makespan от количества задач

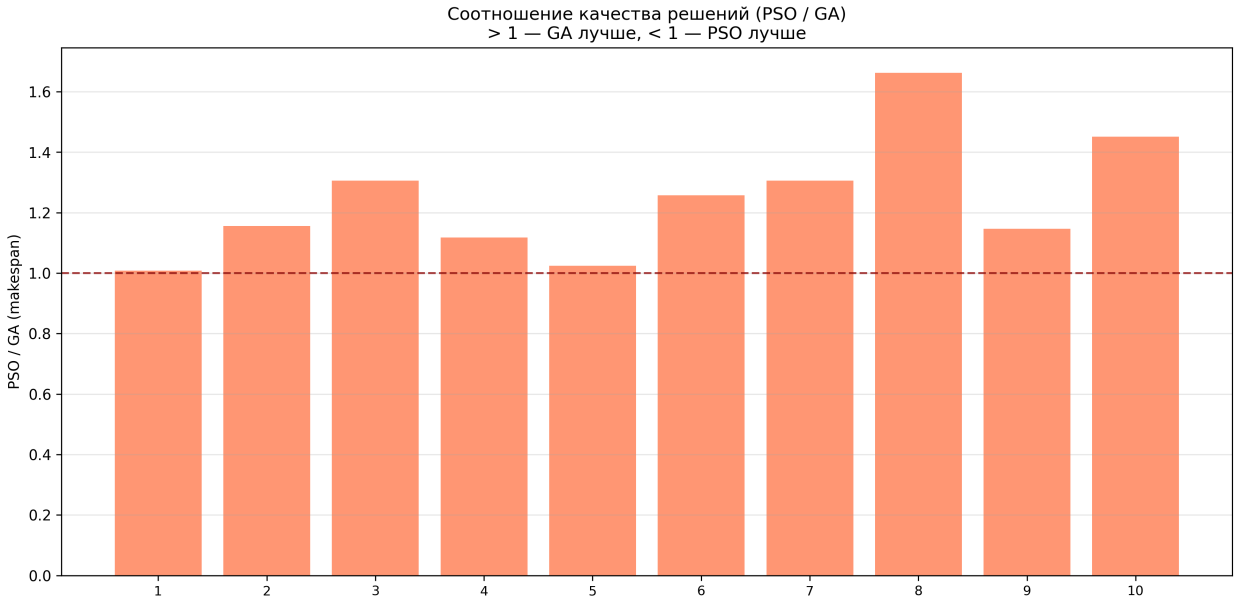

Рис. 6. Соотношение качества решений (PSO / GA по makespan)

В целом, PSO показал превосходство в большинстве случаев, особенно при росте размерности задачи, подтверждая выбор алгоритма.