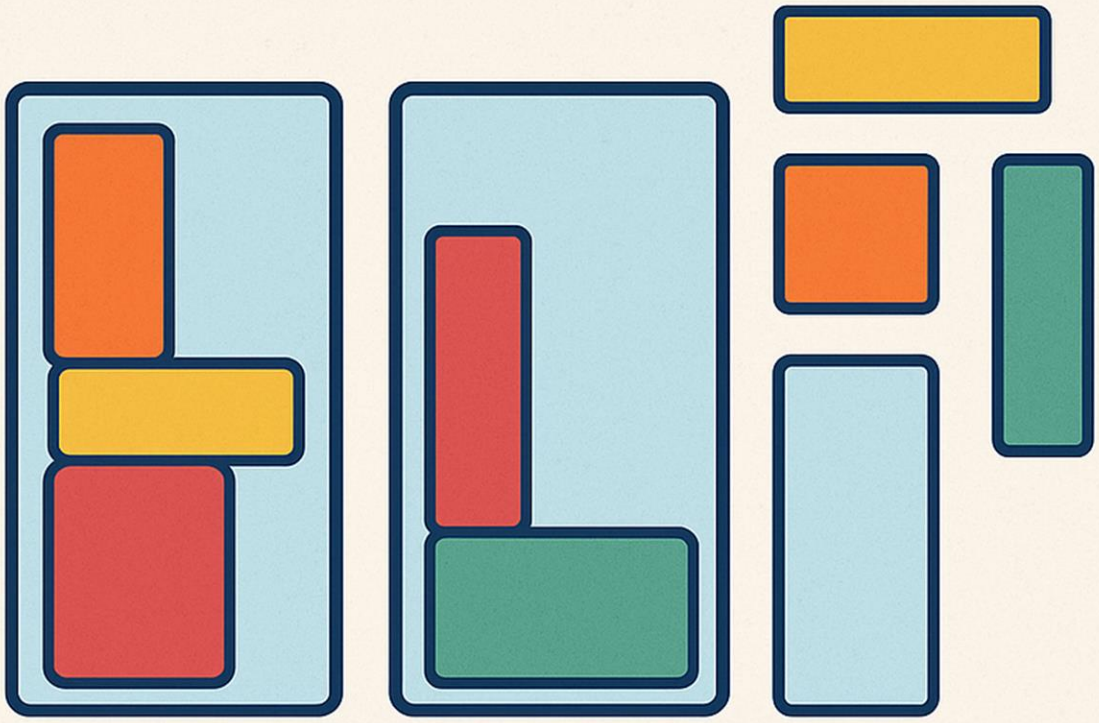
GENETIC ALGORITHM

BIN PACKING PROBLEM



حل مسئله بسته بندی کلاسیک با الگوریتم ژنتیک

یاسین پوررئیسی

خرداد 1404

—

الگوریتم ژنتیک

—

دکتر ابوذر زندوکیلی

چکیده

در این پروژه، از الگوریتم ژنتیک به عنوان روشی فراابتکاری برای حل مسئله‌ی بسته‌بندی (BinPacking)استفاده شده است. مسئله‌ی Bin Packing یکی از مسائل بهینه‌سازی ترکیبیاتی NP-Hardاست که هدف آن، قرار دادن مجموعه‌ای از اقلام با اندازه‌های متفاوت در کمترین تعداد ممکن از بسته‌ها با ظرفیت ثابت است، به‌گونه‌ای که ظرفیت هیچ بسته‌ای تجاوز نکند.

با توجه به پیچیدگی بالای این مسئله، استفاده از الگوریتم‌های سنتی مانند جستجوی کامل یا برنامه‌نویسی پویا در مسائل با ابعاد بزرگ عملی نیست. در این پروژه، الگوریتم ژنتیک با پیاده‌سازی مفاهیم انتخاب، ترکیب(Crossover) ، جهش (Mutation)و با بهبودهایی مانند «نخبه‌گرایی (Elitism) برای یافتن پاسخ‌های نزدیک به بهینه به کار گرفته شده است.

نتایج نشان می‌دهد که الگوریتم ژنتیک قادر است در زمان معقول، راه‌حل‌هایی با کیفیت بالا و استفاده‌ی حداقلی از بسته‌ها ارائه دهد. همچنین، با به‌کارگیری تکنیک‌های بهینه‌سازی در طراحی الگوریتم، همگرایی به سمت جواب بهینه سریع‌تر و دقیق‌تر شده است.

چرا **Bin Packing** مهم است؟

مسئله‌ی **Bin Packing** مستقیماً در مسائل واقعی قابل استفاده است، مثل:

چیدمان اقلام در جعبه‌ها با حداقل تعداد بسته	بسته‌بندی کالا
قرار دادن بار در کامیون‌ها بدون تجاوز از ظرفیت	حمل‌ونقل
تخصیص فرآیندها یا داده‌ها در بلوک‌های حافظه در سیستم‌عامل	تخصیص حافظه
برنامه‌ریزی برای کارهای با اندازه‌ی مختلف در صف چاپ	زمان‌بندی چاپگر
ارسال داده‌ها با اندازه‌های مختلف در بسته‌های شبکه	ارسال داده‌ها

برتری الگوریتم ژنتیک:

Algorithm	Time Complexity	Approximation	Notes
Brute Force	`O(n!)` or worse	Exact	Extremely slow for `n > 20`
First-Fit (Greedy)	`O(n log n)`	~1.7 optimal	Fast but suboptimal
Best-Fit	`O(n log n)`	~1.7 optimal	Better than First-Fit in practice
Genetic Algorithm	`O(G * P * n)`	~1.5–1.1x optimal	Balances quality and speed for large `n`

در مسائلی که زمان و پاسخ تقریبی کافی است، الگوریتم حریص سریع و مناسب است. اما اگر هدف پیدا کردن **پاسخ‌های با کیفیت بالا برای مسائل بزرگ و پیچیده** است، الگوریتم ژنتیک گزینه‌ای بسیار بهتر و قابل تنظیم‌تر است.

صورت مسئله:

تعدادی آیتم به اندازه های [4, 8, 1, 4, 2, 1, 7, 3, 6, 5, 2, 9, 3, 2] داریم
می خواهیم در بسته های به اندازه 10 آن ها را بسته بندی کنیم
طوری که حداقل تعداد بسته استفاده شود

```
import random

ITEM_SIZES = [4, 8, 1, 4, 2, 1, 7, 3, 6, 5, 2, 9, 3, 2]
BIN_CAPACITY = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.05
```

مقادیر دادن اولیه (Initialization) :

هر کروموزوم یک لیست مانند [1,2,5,7,…] است یعنی:

Item(1) in bin(1)
Item(2) in bin(2)
Item(3) in bin(5)
Item(4) in bin(7)
...

تعداد اعضا لیست = تعداد آیتم ها

هر عضو لیست عددی رندوم بین 0 تا بیشینه تعداد بسته ها (در بدترین حالت هر آیتم در بسته اختصاصی خود قرار می گیرد پس بیشینه برابر تعداد آیتم هاست)

```
def create_chromosome(num_items, max_bins):
return [random.randint(0, max_bins - 1) for _ in range(num_items)]

def initial_population(pop_size, num_items, max_bins):
    return [create_chromosome(num_items, max_bins) for _ in range(pop_size)]
```

تابع برازندگی (Fitness Function) :

حال هر کروموزوم از نسل را با تابع فیتنس می سنجیم

- Fitness= (تعداد سطل‌های استفاده‌شده+مجموع اضافه‌بارها×ضریب جریمه)-
- ضریب جریمه = 1000
- علامت منفی به این دلیله که الگوریتم ژنتیک به‌صورت پیش‌فرض دنبال بیشینه‌سازی fitness هست.
- ما در واقع می‌خوایم تعدادbin ها و مقدار اضافه‌بار رو کم کنیم.

```
def fitness(chromosome, item_sizes, bin_capacity):
    bins = {}
    overflow_penalty = 1000

    for i, bin_idx in enumerate(chromosome):
        bins.setdefault(bin_idx, []).append(item_sizes[i])

    total_bins_used = len([b for b in bins.values() if sum(b) > 0])
    overflow = sum(max(0, sum(items) - bin_capacity) for items in bins.values())

    return - (total_bins_used + overflow * overflow_penalty)
```

انتخاب (Selection) :

انتخاب به صورت تورنومنت

به طور اتفاقی سه کروموزوم را انتخاب می کنیم و کروموزوم با بهترین فیتنس را انتخاب می کنیم.

```
def tournament_selection(population, fitnesses, k=3):
    selected = random.sample(list(zip(population, fitnesses)), k)
    selected.sort(key=lambda x: x[1], reverse=True)
    return selected[0][0]
```

تولید فرزند (Crossover) :

- در این حل از برش استفاده نمی کنیم.
- برای هر ژن به صورت اتفاقی از یکی از والدین اش ژن را انتخاب می کنیم.

```
def crossover(parent1, parent2):
    return [parent1[i] if random.random() < 0.5 else parent2[i] for i in range(len(parent1))]
```

جهش(Mutation) :

هر ژن به احتمال (نرخ جهش) به صورت اتفاقی تغییر می کند

- نرخ جهش = 5٪
- ژن i (آیتم i) به احتمال نرخ جهش به عددی اتفاقی از صفر تا تعداد بیشینه بسته ها تغییر می کند.
- آیتم به یک بسته دیگر منتقل می شود.

```
def mutate(chromosome, mutation_rate, max_bins):
    return [
        random.randint(0, max_bins - 1) if random.random() < mutation_rate else gene
        for gene in chromosome]
```

حلقه تشکیل نسل:

در پایان نسل جدیدی تشکیل میشود و این فرآیند را به اندازه تعداد نسل مورد نظر ادامه می دهیم.

```
def genetic_algorithm(item_sizes, bin_capacity, population_size, generations):
    num_items = len(item_sizes)
    max_bins = num_items  # worst case: each item in its own bin
    population = initial_population(population_size, num_items, max_bins)

    for gen in range(generations):
        fitnesses = [fitness(ch, item_sizes, bin_capacity) for ch in population]
        next_gen = []

        # Elitism: keep the best
        best = population[fitnesses.index(max(fitnesses))]
        next_gen.append(best)

        while len(next_gen) < population_size:
            p1 = tournament_selection(population, fitnesses)
            p2 = tournament_selection(population, fitnesses)
            child = crossover(p1, p2)
            child = mutate(child, MUTATION_RATE, max_bins)
            next_gen.append(child)

        population = next_gen

        if gen % 50 == 0:
            print(f"Generation {gen} - Best fitness: {max(fitnesses)}")

    # Final result
    final_fitnesses = [fitness(ch, item_sizes, bin_capacity) for ch in population]
    best_solution = population[final_fitnesses.index(max(final_fitnesses))]
     return best_solution
```

نتیجه گیری :

مسئله‌ی Bin Packingیکی از مسائل پایه‌ای و بسیار مهم در علوم کامپیوتر و مهندسی است، و دلایل زیادی برای اهمیت آن وجود دارد.
در بسیاری از کاربردها، مثل حمل‌ونقل یا انبارداری، استفاده‌ی بهینه از فضا = صرفه‌جویی در هزینه.
مثلاً اگر بتوانی با یک الگوریتم بهتر، فقط یک کامیون کمتر استفاده کنی، ممکن است هزاران دلار در ماه صرفه‌جویی شود.

حل این مسئله با روش Bruteforce پیچیدگی بالایی دارد و کاربردی نیست.
همچنین حل آن با الگوریتم های حریص از دقت پایین تری نسبت به الگوریتم ژنتیک است.
در مسائلی که زمان و پاسخ تقریبی کافی است، الگوریتم حریص سریع و مناسب است. اما اگر هدف پیدا کردن پاسخ‌های با کیفیت بالا برای مسائل بزرگ و پیچیده است، الگوریتم ژنتیک گزینه‌ای بسیار بهتر و قابل تنظیم‌تر است.

با توجه به مزایای بالقوه‌ی الگوریتم ژنتیک و چالش‌های موجود در حل بهینه‌ی مسئلهBin Packing ، انگیزه‌ی اصلی این پروژه، بررسی و پیاده‌سازی یک الگوریتم ژنتیک بهینه‌شده برای حل این مسئله، تحلیل عملکرد آن، و مقایسه با روش‌های سنتی است. همچنین، هدف آن است که نشان داده شود چگونه تنظیم مناسب پارامترها (مانند نرخ جهش، اندازه‌ی جمعیت و تعداد نسل‌ها) می‌تواند در کیفیت پاسخ نهایی تأثیرگذار باشد.

نمونه خروجی نهایی:

```
Generation 0 - Best fitness: -8
Generation 50 - Best fitness: -6
Generation 100 - Best fitness: -6
Generation 150 - Best fitness: -6
Generation 200 - Best fitness: -6
Generation 250 - Best fitness: -6
Generation 300 - Best fitness: -6
Generation 350 - Best fitness: -6
Generation 400 - Best fitness: -6
Generation 450 - Best fitness: -6

Final Bin Packing Solution:
Bin 0: [6, 2, 2] (Total: 10/10)
Bin 5: [1, 9] (Total: 10/10)
Bin 6: [4, 1, 5] (Total: 10/10)
Bin 7: [8] (Total: 8/10)
Bin 8: [4, 3, 3] (Total: 10/10)
Bin 9: [2, 7] (Total: 9/10)

Total bins used: 6
```

پیوست کد :

```
import random

# ------------------------------
# Problem Setup
# ------------------------------
ITEM_SIZES = [4, 8, 1, 4, 2, 1, 7, 3, 6, 5, 2, 9, 3, 2]
BIN_CAPACITY = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.05

# ------------------------------
# Fitness Function
# ------------------------------
def fitness(chromosome, item_sizes, bin_capacity):
bins = {}
overflow_penalty = 1000

for i, bin_idx in enumerate(chromosome):
bins.setdefault(bin_idx, []).append(item_sizes[i])

total_bins_used = len([b for b in bins.values() if sum(b) > 0])
overflow = sum(max(0, sum(items) - bin_capacity) for items in bins.values())

return - (total_bins_used + overflow * overflow_penalty)

# ------------------------------
# Initialization
# ------------------------------
def create_chromosome(num_items, max_bins):
return [random.randint(0, max_bins - 1) for _ in range(num_items)]

def initial_population(pop_size, num_items, max_bins):
return [create_chromosome(num_items, max_bins) for _ in range(pop_size)]

# ------------------------------
# Selection
# ------------------------------
def tournament_selection(population, fitnesses, k=3):
selected = random.sample(list(zip(population, fitnesses)), k)
selected.sort(key=lambda x: x[1], reverse=True)
return selected[0][0]

# ------------------------------
# Crossover
# ------------------------------
def crossover(parent1, parent2):
return [parent1[i] if random.random() < 0.5 else parent2[i] for i in range(len(parent1))]

# ------------------------------
# Mutation
# ------------------------------
def mutate(chromosome, mutation_rate, max_bins):
return [
random.randint(0, max_bins - 1) if random.random() < mutation_rate else gene
for gene in chromosome
]

# ------------------------------
# GA Main Loop
```

```
# ------------------------------
def genetic_algorithm(item_sizes, bin_capacity, population_size, generations):
num_items = len(item_sizes)
max_bins = num_items  # worst case: each item in its own bin
population = initial_population(population_size, num_items, max_bins)

for gen in range(generations):
fitnesses = [fitness(ch, item_sizes, bin_capacity) for ch in population]
next_gen = []

# Elitism
best = population[fitnesses.index(max(fitnesses))]
next_gen.append(best)

while len(next_gen) < population_size:
p1 = tournament_selection(population, fitnesses)
p2 = tournament_selection(population, fitnesses)
child = crossover(p1, p2)
child = mutate(child, MUTATION_RATE, max_bins)
next_gen.append(child)

population = next_gen

if gen % 50 == 0:
print(f"Generation {gen} - Best fitness: {max(fitnesses)}")

# Final result
final_fitnesses = [fitness(ch, item_sizes, bin_capacity) for ch in population]
best_solution = population[final_fitnesses.index(max(final_fitnesses))]

return best_solution

if __name__ == "__main__":
best = genetic_algorithm(ITEM_SIZES, BIN_CAPACITY, POPULATION_SIZE, GENERATIONS)

# Group items by bin
packed_bins = {}
for i, bin_idx in enumerate(best):
packed_bins.setdefault(bin_idx, []).append(ITEM_SIZES[i])

# Clean empty bins and sort by bin index
final_bins = {k: v for k, v in sorted(packed_bins.items()) if sum(v) > 0}

print("\nFinal Bin Packing Solution:")
for bin_id, items in final_bins.items():
print(f"Bin {bin_id}: {items} (Total: {sum(items)}/{BIN_CAPACITY})")

print(f"\nTotal bins used: {len(final_bins)}")
```

End.