Raport końcowy
Optymalizacja procesu kolejkowania

Jan Bronicki 249011, Piątek TN 13:15
Patryk Marciniak 248978, Środa TN 13:15
Borys Staszczak 248958, Piątek TN 11:15

Spis treści

1 Opis problemu

1.1 Algorytmy kolejkowania

Algorytmy kolejkowania nazywane też algorytmami szeregowania zadań to grupa algorytmów służących do optymalizacji harmonogramowania zadań. Podstawowym zestawem aksjomatów w problemie szeregowania zadań są zadania, czyli pewne czynności, które zajmują pewien określony czas do wykonania oraz maszyny, czyli obiekty, na których te zadania mogą zostać wykonane. W poniższym projekcie rozważone zostały problemy szeregowania zadań na jednej maszynie, czyli tzw. problemy jednomaszynowe. W tego typu problemach każde zadanie musi zostać wykonane przez jedną maszynę, która jednocześnie może wykonywać tylko jedno zadanie tzn. nawet jeśli wiele zadań jest gotowych do wykonania, to dopóki maszyna wykonuje już jakieś zadanie, dopóty nie może ona rozpocząć innego zadania (chyba że problem dopuszcza możliwość przerwania zadania, w celu wykonania zadania o większym priorytecie. W takim wariancie problemu, przerwane zadanie musi zostać dokończone w późniejszym terminie).

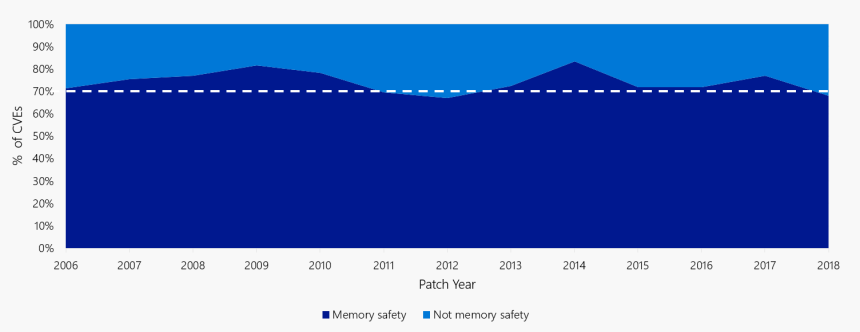
Algorytmy, jakie zostały w projekcie zaimplementowane, to algorytm Schrage, algorytm Schrage z podziałem oraz algorytm Carliera. W przypadku tych algorytmów, pojedyncze i-te zadanie jest opisywane przez trzy wartości:

- r_i - czas dostarczenia zadania, czyli najwcześniejsza chwila, w której i-te zadanie będzie dostępne do wykonania. Dopóki nie nastanie moment r_i, dopóty maszyna nie może zacząć wykonywać zadanie
- p_i - czas trwania zadania (czas wykonywania), czyli czas, jaki potrzebuje maszyna na wykonanie i-tego zadania
- q_i - czas stygnięcia zadania, czyli czas, jaki i-te zadanie potrzebuje na "ostygnięcie" po tym, jak zostanie wykonane

2 Projekt rozwiązania

2.1 Rust

Przestrzeń języków programowania służących do generalnego użytku była przez wiele dekad zdominowana przez dwa główne systemy zarządzania pamięcią programu. Pierwszym, a zarazem najstarszym z nich, stosowanych w takich językach jak C lub C++. Ten model zarządzania pamięcią opierał się głównie na tym, aby to programista alokujący dany obszar pamięci był za niego odpowiedzialny, samodzielnie nim zarządzał oraz był odpowiedzialny za jego dealokację. Takie rozwiązanie powoduje liczne błędy związane z zarządzaniem pamięcią, począwszy od jej nieefektywnego zarządzania, przez błędy programu, po luki w zabezpieczeniach programu, powodujące możliwość wykorzystania ich w złej wierze poprzez atak zewnętrzny na daną aplikację. Według raportu Microsoftu, około 70% wszystkich zgłaszanych błędów, umożliwiających ataki (CVE - Common Vulnerabilities and Exposures), na daną aplikację, jest związana, lub bezpośrednio spowodowana błędami w zarządzaniu pamięcią.



Drugim sposobem zarządzania pamięcią funkcjonującym w takich językach, jak Python, C#, czy JavaScript jest tzw. “garbage collector”, który jest automatycznym zarządcą, zarządzającym alokacją i zwalnianiem pamięci dla aplikacji. “Garbage collector” zwalnia programistów z konieczności ręcznego zarządzania pamięcią, a także efektywnie ją przydziela, czyści i utrzymuje w stanie, który umożliwia jej ponowną alokację. Ponadto zapewnia bezpieczeństwo, poprzez

zapewnienie, że dany obiekt nie może używać pamięci przydzielonej innemu obiektowi, co pomaga wyeliminować typowe problemy, takie jak zapomnienie o dealokacji pamięci, po usunięciu obiektu, czy wycieki pamięci. Minusem tej metody jest dodatkowe obciążenie programu równolegle działającym "garbage collectorem", który spowalnia działanie naszego oprogramowania.

Rust zaprezentował nowy sposób zarządzania pamięcią opierający się o tzw. "borrow checker", czyli funkcjonalność kompilatora, która w już w czasie kompilacji, a nie w czasie działania programu dba o bezpieczeństwo i poprawne użytkowanie pamięci programu. Metoda działania "borrow checkera" opiera się na prostej zasadzie limitowania dostępu do danego zasobu częścią programu, tylko i wyłącznie w dwóch przypadkach: pierwszym, będącym pojedynczy właściciel, mający wyłączne prawo do czytania oraz pisania do danego zasobu i drugi, gdzie zasób może być odczytywany przez różne elementy programu, z takim ograniczeniem, że żaden z nich nie może tego zasobu edytować. Te fundamentalne zasady następnie wpływają na całą strukturyzację i działanie mechanizmów języka, w ten sposób eliminując problemy zarządzania zasobami, występującymi w wyżej obu wymienionych przypadkach.

2.2 Cargo

Cargo jest systemem budowy oraz menadżerem paczek języka Rust. Jego głównych zadań zalicza się:

- Budowa i kompilacja paczek
- Pobieranie bibliotek
- Przygotowanie paczek do dystrybucji
- Publikacja paczek na crates.io

Crates.io to internetowy rejestr paczek języka Rust, który pozwala na publikację oraz dostęp do publicznych bibliotek tego języka.

Jednym z założeń projektu było opublikowanie opracowywanej biblioteki na crates.io. Celem tego założenia było wzbogacenie zbioru bibliotek Rusta o nieobecne implementacje algorytmów kolejkowania, a także zapoznanie się z samym systemem publikacji projektów.

2.3 Schrage

Algorytm Schrage jest algorytmem rozwiązującym problem $1|r_j, q_j|C_{max}$. Polega on na dokładaniu nieuszeregowanych jeszcze zadań na koniec bieżącej kolejki. W pierwszej kolejności algorytm umieszcza nieuszeregowane zadania, o ile takowe istnieją, w zbiorze zadań gotowych tj. zadania o czasie dostarczenia r_i mniejszym lub równym aktualnemu czasowi t. Następnie, z tego zbioru, wybierane jest zadanie o największym czasie stygnięcia q_i. To zadanie zostaje usunięte ze zbioru i zostaje wykonane na maszynie. Po zakończeniu zadania proces dodawania zadań do zbioru i wybierania zadania o największym q_i powtarza się, dopóki istnieją jakiekolwiek niewykonane zadania.

2.4 Schrage z podziałem

Algorytm Schrage z podziałem zadań jest w działaniu bardzo podobny do klasycznego algorytmu Schrage. Ten algorytm rozwiązuje problem $1|r_j, q_j, pmtn|C_{max}$ i od podstawowej wersji algorytmu Schrage różni się tym, że przy planowaniu szeregowania zadań dopuszcza się możliwość przerwania aktualnie wykonywanego zadania w pewnej chwili t_0 równej czasowi dostarczenia r_i pewnego zadania, takiego że wykonanie go jak najszybciej prowadzi do lepszego rozwiązania. Przerwane zadanie nie musi być wykonane od razu po zakończeniu wykonywania zadania, które spowodowało przerwanie, ale konieczne jest aby to zadanie zostało dokonczone. Warto też zaznaczyć, że:

- każde rozwiązanie problemu $1|r_j, q_j|C_{max}$ jest także rozwiązaniem problemu $1|r_j, q_j, pmtn|C_{max}$
- optymalne rozwiązanie problemu $1|r_j, q_j, pmtn|C_{max}$ jest lepsze lub tak samo dobre, jak rozwiązanie problemu $1|r_j, q_j|C_{max}$

2.5 Carlier

Algorytm Carliera jest algorytmem dla problemu $1|r_j, q_j|C_{max}$ i w przeciwieństwie do opisanych wcześniej wariantów algorytmu Schrage, jest on algorytmem dokładnym, tj. wyznacza on rozwiązanie optymalne, czyli rozwiązanie minimalizujące kryterium C_{max}. Algorytm Carliera bazuje na metodzie podziału i ograniczeń (B&B - branch and bound), a także wykorzystuje algorytmy Schrage oraz Schrage z podziałem do wyznaczenia odpowiednio: górnego i dolnego ograniczenia.

W pierwszej kolejności algorytm wyznacza górne ograniczenie (upper bound) za pomocą algorytmu Schrage, a następnie wyznacza blok (a, b) zadań oraz zadanie referencyjne c. Blok (a, b) to tzw. ścieżka krytyczna rozwiązania, czyli zadania (w danej kolejności), których jakiekolwiek opóźnienie spowoduje opóźnienie zakończenia całego procesu wykonywania zadań na maszynie. Zadanie referencyjne c to tzw. zadanie krytyczne, takie że $q_C < q_B$. Jeśli w danym rozwiązaniu nie istnieje zadanie krytyczne c, to jest to rozwiązanie optymalne w danym węźle. W przeciwnym wypadku algorytm, w celu wyznaczenia optymalnego rozwiązania węzła wybiera najlepsze z następujących rozwiązań:

- bieżące rozwiązanie węzła, wyznaczone z algorytmu Schrage
- najlepsze z rozwiązań, gdzie zadanie referencyjne c znajduje przed zadaniami z bloku $(c + 1, b)$
- najlepsze z rozwiązań, gdzie zadanie referencyjne c znajduje się za zadaniami z bloku $(c + 1, b)$

3 Implementacja rozwiązania

3.1 Struktury danych

Dane są zapisywane w prostej strukturze. Pozwala to na łatwe posługiwanie się nimi bez potrzeby wywoływania za każdym razem wielu różnych zmiennych. Takie podejście umożliwia również zdefiniowanie powtarzających się operacji umożliwiając w ten sposób używanie ich w różnych częściach projektu, bez potrzeby definiowania ich za każdym razem.

```rust
#[derive(Copy, Clone, Debug)]
pub struct Job {
    pub delivery_time: u32,   // r
    pub processing_time: u32, // p
    pub cooldown_time: u32,   // q
}

impl Job {
    pub fn new(delivery_time: u32, processing_time: u32, cooldown_time: u32) -> Job {
        Job {
            delivery_time,
            processing_time,
            cooldown_time,
        }
    }

    #[allow(dead_code)]
    pub fn total_time(&self) -> u32 {
        self.delivery_time + self.processing_time + self.cooldown_time
    }
}
```

3.2 Algorytm Schrage

Jako pierwszy zaimplementowano algorytm schrage.

```rust
pub fn schrage(jobs: &JobList) -> SchrageJobTable {
    // N
    // A list of jobs to be completed
    let mut shortest_delivery_jobs = JobList::new(jobs.sorted_by_delivery_time());
    // G
    // A list of jobs that in a current moment are ready to run
    let mut ready_to_run = JobList::new(Vec::new());
    // Time tracking variable
    let mut t: u32 = 0;
    // The final sequence in which the jobs should be run
    let mut pi: JobList = JobList::new(Vec::new());

    // Iterate over all of the jobs until we ran out of them
    while !shortest_delivery_jobs.jobs.is_empty() || !ready_to_run.jobs.is_empty() {
        // Find all jobs that are available
        while !shortest_delivery_jobs.jobs.is_empty()
            && shortest_delivery_jobs.jobs[0].delivery_time <= t
        {
            ready_to_run
                .jobs
                .append(&mut vec![shortest_delivery_jobs.jobs[0]]);
            shortest_delivery_jobs.jobs.remove(0);
        }
        // If there are jobs that are ready to run schedule them
        if !ready_to_run.jobs.is_empty() {
            let vec_by_processing_time = JobList {
                jobs: ready_to_run.sorted_by_processing_time(),
            };
            let reversed: JobList = JobList {
                jobs: vec_by_processing_time.jobs.into_iter().rev().collect(),
            };
            let cooldown_times: Vec<Job> = reversed.sorted_by_cooldown_time();

            let max_cooldown_time = cooldown_times.last().unwrap();
            let position = ready_to_run
                .jobs
                .iter()
                .position(|&n| &n == max_cooldown_time)
                .unwrap();
            ready_to_run.jobs.remove(position);
            // Add a job to the final sequence
            pi.jobs.push(*max_cooldown_time);
            t += max_cooldown_time.processing_time;
        } else {
            // If there aren't any jobs that can be run,
            // skip to when the nearest job is available
            t = shortest_delivery_jobs.jobs[0].delivery_time;
        }
    }
    SchrageJobTable { job_list: pi }
}
```

3.3 Algorytm Schrage z podziałem

Implementacja Schrage z podziałem:

```rust
pub fn part_time_schrage(jobs: &JobList) -> u32 {
    // N
    let mut shortest_delivery_jobs = JobList::new(jobs.sorted_by_delivery_time());
    // G
    let mut ready_to_run = JobList::new(Vec::new());
    let mut current_job = Job::new(0, 0, 0);
    let mut t: u32 = 0;
    let mut c_max: u32 = 0;
    let mut pi: JobList = JobList { jobs: Vec::new() };

    while !shortest_delivery_jobs.jobs.is_empty() || !ready_to_run.jobs.is_empty() {
        while !shortest_delivery_jobs.jobs.is_empty()
            && shortest_delivery_jobs.jobs[0].delivery_time <= t
        {
            ready_to_run
                .jobs
                .append(&mut vec![shortest_delivery_jobs.jobs[0]]);
            let next_job = shortest_delivery_jobs.jobs.remove(0);

            if next_job.cooldown_time > current_job.cooldown_time {
                current_job.processing_time = t - next_job.delivery_time;
                t = next_job.delivery_time;

                if current_job.processing_time > 0 {
                    ready_to_run.jobs.append(&mut vec![current_job]);
                    ready_to_run.jobs = ready_to_run.sorted_by_delivery_time().clone();
                }
            }
        }

        if !ready_to_run.jobs.is_empty() {
            let cooldown_times = ready_to_run.sorted_by_cooldown_time();
            let max_cooldown_time = cooldown_times.last().unwrap();
            let position = ready_to_run
                .jobs
                .iter()
                .position(|&n| n.cooldown_time == max_cooldown_time.cooldown_time)
                .unwrap();
            current_job = ready_to_run.jobs.remove(position);

            // Add a job to the final sequence
            pi.jobs.push(*max_cooldown_time);
            t += max_cooldown_time.processing_time;
            c_max = cmp::max(t + max_cooldown_time.cooldown_time, c_max)
        } else {
            t = shortest_delivery_jobs.jobs[0].delivery_time;
        }
    }
    c_max
}
```

3.4 Algorytm Carliera

Implementacja algorytmu Carlier:

```rust
pub fn carlier(jobs: &mut JobList, upper_bound: &mut u32) {
    let result: SchrageJobTable = schrage(jobs);
    let mut pi: JobList = result.job_list.clone();
    let c_max_from_schrage: u32 = result.c_max();

    if c_max_from_schrage < *upper_bound {
        *upper_bound = c_max_from_schrage;
    }

    let critical_path_end_index: u32 = find_critical_path_end(pi.clone(), c_max_from_schrage);
    let critical_path_start_index: u32 =
        find_critical_path_start(pi.clone(), c_max_from_schrage, critical_path_end_index);
    let critical_job_index: i32 = find_critical_job(
        pi.clone(),
        critical_path_end_index,
        critical_path_start_index,
    );

    if critical_job_index == -1 {
        return;
    }

    let mut rj: u32 = u32::MAX;
    let mut pj: u32 = 0;
    let mut qj: u32 = u32::MAX;

    for i in (critical_job_index as usize + 1)..=critical_path_end_index as usize {
        if pi.jobs[i as usize].delivery_time < rj {
            rj = pi.jobs[i as usize].delivery_time;
        }

        if pi.jobs[i as usize].cooldown_time < qj {
            qj = pi.jobs[i as usize].cooldown_time;
        }

        pj += pi.jobs[i as usize].processing_time;
    }

    let c_job_delivery: u32 = pi.jobs[critical_job_index as usize].delivery_time;
    pi.jobs[critical_job_index as usize].delivery_time = cmp::max(c_job_delivery, rj + pj);

    let mut lower_bound: u32 = part_time_schrage(jobs);
    if lower_bound < *upper_bound {
        carlier(&mut pi, upper_bound);
    }

    pi.jobs[critical_job_index as usize].delivery_time = c_job_delivery;

    let c_job_cooldown: u32 = pi.jobs[critical_job_index as usize].cooldown_time;
    pi.jobs[critical_job_index as usize].cooldown_time = cmp::max(c_job_cooldown, pj + qj);
    lower_bound = part_time_schrage(jobs);

    if lower_bound < *upper_bound {
        carlier(&mut pi, upper_bound)
    }
    pi.jobs[critical_job_index as usize].cooldown_time = c_job_cooldown;
}
```

```rust
fn find_critical_path_end(pi: JobList, c_max: u32) -> u32 {
    let mut b_value: i32 = -1;
    let mut t: u32 = pi.jobs[0].delivery_time;

    for i in 0..pi.jobs.len() {
        let current_job: Job = pi.jobs[i];
        t = cmp::max(t, current_job.delivery_time) + current_job.processing_time;

        if c_max == (current_job.cooldown_time + t) {
            b_value = i as i32;
        }
    }
    b_value as u32
}

fn find_critical_path_start(pi: JobList, c_max: u32, b_value: u32) -> u32 {
    let mut sum: u32;
    let mut a_value: i32 = -1;
    let mut t: u32 = pi.jobs[0].delivery_time;

    for i in 0..pi.jobs.len() {
        let current_job: Job = pi.jobs[i];
        t = cmp::max(t, current_job.delivery_time) + current_job.processing_time;

        if a_value == -1 {
            sum = 0;

            for j in i..=b_value as usize {
                sum += pi.jobs[j].processing_time;
            }
            sum += pi.jobs[b_value as usize].cooldown_time;

            if c_max == (current_job.delivery_time + sum) {
                a_value = i as i32;
            }
        }
    }
    a_value as u32
}

fn find_critical_job(pi: JobList, b_value: u32, a_value: u32) -> i32 {
    let mut c_value: i32 = -1;

    for i in a_value..=b_value {
        if pi.jobs[i as usize].cooldown_time < pi.jobs[b_value as usize].cooldown_time {
            c_value = i as i32;
        }
    }
    c_value
}
```

4 Publikacja biblioteki

Paczka została opublikowana na rejestrze crates.io i obecnie została pobrana ponad 100 razy. Przyjęcie takiej formy projektu wymagało zapewnienia jak najlepszego standardu kodu oraz dokumentacji. W tym celu przeprowadzono odpowiednią restrukturyzację kodu, co pomogło pozbyć się nieoptymalnych części kodu jak i wychwycić drobne błędy. Ponadto w celu odpowiedniej walidacji poprawności działania algorytmów zaimplementowano wiele testów jednostkowych które przydatne były nie tylko podczas projektowania i implementacji paczki, ale mogą również posłużyć użytkownikom, którzy z opublikowanej biblioteki skorzystają.