1 Постановка задачи

Алгоритм A* (A-star).

Проанализировать поставленную задачу, реализовать последовательный алгоритм ее решения, при необходимости вспомнить соответствующие дисциплины. Выделить подзадачи, которые могут быть вычислены параллельно и реализованы в виде потоков с барьерной(fork-join) синхронизацией.

2 Последовательный алгоритм

Input: Поле Map с размеченными проходимыми и не проходимыми вершинами

Init: $Start, Target$

$Open, Closed$

push $Start \rightarrow Open$

while $open$ содержит элементы и путь не найден

do

- Удаляем из $Open$ узел $Current = min_F(Open)$
- ***push*** $Current \rightarrow Closed$
- Для каждой из восьми смежных(Adj) вершин:

Если вершина Adj проходима и не в $Closed$:

- Если вершина не в $Open$:
  - $Adj.Parrent = Current$;
  - Пересчитать F, G, H
  - **push** $Adj \rightarrow Open$
- Если вершина в $Open$:
  - Пересчитать F, G. Если новое G меньше старого, поменять $Parent$
- Если $target$ в $open$, путь найден.

end.

3 Параллельный алгоритм

3.1 Разбиение на кластеры

Метод заключается в разбиении карты на кластеры и нахождение всех точек входа и выхода в каждом из них. Чтобы добиться эффективности, необходимо запускать программу на многоядерном устройстве (например, на видеокарте с поддержкой технологии *CUDA*). Но зато такой алгоритм лучше всего подходит для нахождения всех путей на поле.

3.2 Простой параллельный алгоритм

Единственная возможность применить к алгоритму барьерную синхронизацию – это заменить цикл по восьми смежным вершинам параллельной секцией, равномерно распределив обработку вершин по потокам. Этот метод выглядит логичным, однако на практике не приносит результатов, на что есть несколько причин.

Во-первых, передача сообщений между потоками – очень дорогая операция. Во-вторых, к потокам неприменимы оптимизации компилятора. Поэтому барьерная синхронизация целесообразна только в случае обработки больших данных, но не восьми ячеек.

Таким образом, было решено использовать использовать немодифицированный последовательный алгоритм для нахождения всех путей на поле, разделяя между отдельными потоками вычисление каждого пути.

4 Вычислительная сложность

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где h^* – оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка $h(x)$ не должна расти быстрее, чем логарифм от оптимальной эвристики.

Но ещё большую проблему, чем временная сложность, представляют собой потребляемые алгоритмом ресурсы памяти. В худшем случае ему приходится помнить экспоненциальное количество узлов. Для борьбы с этим было предложено несколько вариаций алгоритма, таких как алгоритм A* с итеративным углублением (iterative deeping A*, IDA*), A* с ограничением памяти (memory-bounded A*, MA*), упрощённый MA* (simplified MA*, SMA*) и рекурсивный поиск по первому наилучшему совпадению (recursive best-first search, RBFS).

5 Результаты тестов производительности

Тест	Результат	Время, с
16x16-multi	T	40.33
16x16-single	T	90.46
10x10-multi	T	0.74
10x10-multi	T	1.53

Приложение А

Тестирование функции pmap

Функция $pmap$ действует абсолютно так же, как и map, за исключением того, что функция f применяется к элементам параллельно. Её очень удобно использовать для организации барьерной **fork-join** синхронизации. Так же, как и map, $pmap$ возвращает ленивую последовательность и порядок применения функции не стандартизирован.

Не стоит забывать, что $pmap$ полезна только если f – относительно долгая операция, на порядки превышающая расходы на создание потоков. Иначе, функция $pmap$ работает гарантированно дольше однопоточного аналога.

```
(defn str-seq [n]
  (if (= n 0)
    "0"
    (str n "," (str-seq (- n 1)))))

(defn run-test [f n label]
  (println label)
  (time (dorun (f str-seq (range 1 n)))))

(defn compare-map-pmap [n]
  (run-test map n "Single:")
  (run-test pmap n "Multi:"))


(compare-map-pmap 1500)
```

f	Время, *msecs*
map	5465.595298
$pmap$	3760.315602

Приложение Б

Код программного продукта

graph-all-single.lisp

```
(load "graph-all.fasl")
(use-package :graph-all)

(defparameter *test* 10)

(print
 (time
  (graph-all:process-graph 1  (- *test* 2)  (- *test* 2))))
```

graph-all-multi.lisp

```
(load "graph-all.fasl")

(defpackage :graph-all-multi
  (:use "CL" "SB-THREAD" "SB-EXT"))
(use-package :graph-all)
(in-package :graph-all-multi)

(defparameter *barrier* 0)
(defparameter *parallel-end* (make-waitqueue))
(defparameter *lock* (make-mutex :name "lock"))
(defparameter *test* 10)

(defun create-process (from to &optional (end to))
  #'(lambda nil
      (graph-all:process-graph from to end)
      (when (>= (incf *barrier*) 2)
        (condition-notify *parallel-end*))
      (return-from-thread nil)))

(with-mutex (*lock*)
  (make-thread (create-process 1 (floor (/ *test* 2)) (- *test* 2)))
  (make-thread (create-process (floor (/ *test* 2)) (- *test* 2)))
  (print
```

```
   (time
    (when (condition-wait *parallel-end* *lock*)))))
```

test-single.lisp

```
(defun a-star-single (test)
  (defparameter *graph* nil)
  (graph-all::open-graph (format nil "tests/~a.txt" test))
  (graph-all::a-star
   (graph-all::select-v 1 1)
   (graph-all::select-v (- test 1) (- test 1))))

(deftest graph-10-single
  (check
    (a-star-single 10)))

(deftest graph-16-single
  (check
    (a-star-single 16)))

(deftest single
  (combine
    (graph-10-single)
    (graph-16-single)))

(single)
```