2d/sparse_pose_graph.cc

Go to the documentation of this file.

```
1  /*
2   * Copyright 2016 The Cartographer Authors
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *      http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
15  */
16
17 #include "cartographer/mapping_2d/sparse_pose_graph.h"
18
19 #include <algorithm>
20 #include <cmath>
21 #include <cstdio>
22 #include <functional>
23 #include <iomanip>
24 #include <iostream>
25 #include <limits>
26 #include <memory>
27 #include <set>
28 #include <sstream>
29 #include <string>
30
31 #include "Eigen/Eigenvalues"
32 #include "cartographer/common/make_unique.h"
33 #include "cartographer/common/math.h"
34 #include "cartographer/mapping/sparse_pose_graph/proto/constraint_builder_options.pb.h"
35 #include "cartographer/sensor/compressed_point_cloud.h"
36 #include "cartographer/sensor/voxel_filter.h"
37 #include "glog/logging.h"
38
39 namespace cartographer {
40 namespace mapping_2d {
41
42 SparsePoseGraph::SparsePoseGraph(
43     const mapping::proto::SparsePoseGraphOptions& options,
44     common::ThreadPool* thread_pool)
45     : options_(options),
46       optimization_problem_(options_.optimization_problem_options()),
47       constraint_builder_(options_.constraint_builder_options(), thread_pool) {}
48
49 SparsePoseGraph::~SparsePoseGraph() {
50   WaitForAllComputations();
51   common::MutexLocker locker(&mutex_);
52   CHECK(scan_queue_ == nullptr);
53 }
54
55 void SparsePoseGraph::GrowSubmapTransformsAsNeeded(
56     const std::vector<const mapping::Submap*>& insertion_submaps) {
57   CHECK(!insertion_submaps.empty());
58   const mapping::SubmapId first_submap_id = GetSubmapId(insertion_submaps[0]);
59   const int trajectory_id = first_submap_id.trajectory_id;
60   CHECK_GE(trajectory_id, 0);
61   const auto& submap_data = optimization_problem_.submap_data();
62   if (insertion_submaps.size() == 1) {
63     // If we don't already have an entry for the first submap, add one.
```

```
    CHECK_EQ(first_submap_id.submap_index, 0);
    if (static_cast<size_t>(trajectory_id) >= submap_data.size() ||
        submap_data[trajectory_id].empty()) {
      optimization_problem_.AddSubmap(trajectory_id,
                                      transform::Rigid2d::Identity());
    }
    return;
  }
  CHECK_EQ(2, insertion_submaps.size());
  const int next_submap_index = submap_data.at(trajectory_id).size();
  // CHECK that we have a index for the second submap.
  const mapping::SubmapId second_submap_id = GetSubmapId(insertion_submaps[1]);
  CHECK_EQ(second_submap_id.trajectory_id, trajectory_id);
  CHECK_LE(second_submap_id.submap_index, next_submap_index);
  // Extrapolate if necessary.
  if (second_submap_id.submap_index == next_submap_index) {
    const auto& first_submap_pose =
        submap_data.at(trajectory_id).at(first_submap_id.submap_index).pose;
    optimization_problem_.AddSubmap(
        trajectory_id,
        first_submap_pose *
            sparse_pose_graph::ComputeSubmapPose(*insertion_submaps[0])
                .inverse() *
            sparse_pose_graph::ComputeSubmapPose(*insertion_submaps[1]));
  }
}

void SparsePoseGraph::AddScan(
    common::Time time, const transform::Rigid3d& tracking_to_pose,
    const sensor::RangeData& range_data_in_pose, const transform::Rigid2d& pose,
    const int trajectory_id, const mapping::Submap* const matching_submap,
    const std::vector<const mapping::Submap*>& insertion_submaps) {
  const transform::Rigid3d optimized_pose(
      GetLocalToGlobalTransform(trajectory_id) * transform::Embed3D(pose));

  common::MutexLocker locker(&mutex_);
  trajectory_nodes_.Append(
      trajectory_id,
      mapping::TrajectoryNode{
          std::make_shared<const mapping::TrajectoryNode::Data>(
              mapping::TrajectoryNode::Data{
                  time, range_data_in_pose,
                  Compress(sensor::RangeData{Eigen::Vector3f::Zero(), {}, {}}),
                  tracking_to_pose}),
          optimized_pose});
  ++num_trajectory_nodes_;
  trajectory_connectivity_.Add(trajectory_id);

  if (submap_ids_.count(insertion_submaps.back()) == 0) {
    const mapping::SubmapId submap_id =
        submap_data_.Append(trajectory_id, SubmapData());
    submap_ids_.emplace(insertion_submaps.back(), submap_id);
    submap_data_.at(submap_id).submap = insertion_submaps.back();
  }
  const mapping::Submap* const finished_submap =
      insertion_submaps.front()->finished_probability_grid() != nullptr
          ? insertion_submaps.front()
          : nullptr;

  // Make sure we have a sampler for this trajectory.
  if (!global_localization_samplers_[trajectory_id]) {
    global_localization_samplers_[trajectory_id] =
        common::make_unique<common::FixedRatioSampler>(
            options_.global_sampling_ratio());
  }

  AddWorkItem([=]() REQUIRES(mutex_) {
    ComputeConstraintsForScan(matching_submap, insertion_submaps,
                              finished_submap, pose);
  });
}

void SparsePoseGraph::AddWorkItem(std::function<void()> work_item) {
  if (scan_queue_ == nullptr) {
    work_item();
  } else {
```

```
140     scan_queue_->push_back(work_item);
141   }
142 }
143
144 void SparsePoseGraph::AddImuData(const int trajectory_id, common::Time time,
145                                  const Eigen::Vector3d& linear_acceleration,
146                                  const Eigen::Vector3d& angular_velocity) {
147   common::MutexLocker locker(&mutex_);
148   AddWorkItem([=]() REQUIRES(mutex_) {
149     optimization_problem_.AddImuData(trajectory_id, time, linear_acceleration,
150                                      angular_velocity);
151   });
152 }
153
154 void SparsePoseGraph::ComputeConstraint(const mapping::NodeId& node_id,
155                                         const mapping::SubmapId& submap_id) {
156   CHECK(submap_data_.at(submap_id).state == SubmapState::kFinished);
157
158   // Only globally match against submaps not in this trajectory.
159   if (node_id.trajectory_id != submap_id.trajectory_id &&
160       global_localization_samplers_[node_id.trajectory_id]->Pulse()) {
161     constraint_builder_.MaybeAddGlobalConstraint(
162         submap_id, submap_data_.at(submap_id).submap, node_id,
163         &trajectory_nodes_.at(node_id).constant_data->range_data_2d.returns,
164         &trajectory_connectivity_);
165   } else {
166     const bool scan_and_submap_trajectories_connected =
167         reverse_connected_components_.count(node_id.trajectory_id) > 0 &&
168         reverse_connected_components_.count(submap_id.trajectory_id) > 0 &&
169         reverse_connected_components_.at(node_id.trajectory_id) ==
170             reverse_connected_components_.at(submap_id.trajectory_id);
171     if (node_id.trajectory_id == submap_id.trajectory_id ||
172         scan_and_submap_trajectories_connected) {
173       const transform::Rigid2d initial_relative_pose =
174           optimization_problem_.submap_data()
175               .at(submap_id.trajectory_id)
176               .at(submap_id.submap_index)
177               .pose.inverse() *
178           optimization_problem_.node_data()
179               .at(node_id.trajectory_id)
180               .at(node_id.node_index)
181               .point_cloud_pose;
182
183       constraint_builder_.MaybeAddConstraint(
184           submap_id, submap_data_.at(submap_id).submap, node_id,
185           &trajectory_nodes_.at(node_id).constant_data->range_data_2d.returns,
186           initial_relative_pose);
187     }
188   }
189 }
190
191 void SparsePoseGraph::ComputeConstraintsForOldScans(
192     const mapping::Submap* submap) {
193   const auto submap_id = GetSubmapId(submap);
194   const auto& submap_data = submap_data_.at(submap_id);
195
196   const auto& node_data = optimization_problem_.node_data();
197   for (size_t trajectory_id = 0; trajectory_id != node_data.size();
198        ++trajectory_id) {
199     for (size_t node_index = 0; node_index != node_data[trajectory_id].size();
200          ++node_index) {
201       const mapping::NodeId node_id{static_cast<int>(trajectory_id),
202                                     static_cast<int>(node_index)};
203       if (!trajectory_nodes_.at(node_id).trimmed() &&
204           submap_data.node_ids.count(node_id) == 0) {
205         ComputeConstraint(node_id, submap_id);
206       }
207     }
208   }
209 }
210
211 void SparsePoseGraph::ComputeConstraintsForScan(
212     const mapping::Submap* matching_submap,
213     std::vector<const mapping::Submap*> insertion_submaps,
214     const mapping::Submap* finished_submap, const transform::Rigid2d& pose) {
215   GrowSubmapTransformsAsNeeded(insertion_submaps);
```

```
216   const mapping::SubmapId matching_id = GetSubmapId(matching_submap);
217   const transform::Rigid2d optimized_pose =
218       optimization_problem_.submap_data()
219           .at(matching_id.trajectory_id)
220           .at(matching_id.submap_index)
221           .pose *
222       sparse_pose_graph::ComputeSubmapPose(*matching_submap).inverse() * pose;
223   const mapping::NodeId node_id{
224       matching_id.trajectory_id,
225       static_cast<size_t>(matching_id.trajectory_id) <
226               optimization_problem_.node_data().size()
227           ? static_cast<int>(optimization_problem_.node_data()
228                                  .at(matching_id.trajectory_id)
229                                  .size())
230           : 0};
231   const auto& scan_data = trajectory_nodes_.at(node_id).constant_data;
232   optimization_problem_.AddTrajectoryNode(
233       matching_id.trajectory_id, scan_data->time, pose, optimized_pose);
234   for (const mapping::Submap* submap : insertion_submaps) {
235     const mapping::SubmapId submap_id = GetSubmapId(submap);
236     CHECK(submap_data_.at(submap_id).state == SubmapState::kActive);
237     submap_data_.at(submap_id).node_ids.emplace(node_id);
238     const transform::Rigid2d constraint_transform =
239         sparse_pose_graph::ComputeSubmapPose(*submap).inverse() * pose;
240     constraints_.push_back(Constraint{submap_id,
241                                       node_id,
242                                       {transform::Embed3D(constraint_transform),
243                                        options_.matcher_translation_weight(),
244                                        options_.matcher_rotation_weight()},
245                                       Constraint::INTRA_SUBMAP});
246   }
247
248   for (int trajectory_id = 0; trajectory_id < submap_data_.num_trajectories();
249        ++trajectory_id) {
250     for (int submap_index = 0;
251          submap_index < submap_data_.num_indices(trajectory_id);
252          ++submap_index) {
253       const mapping::SubmapId submap_id{trajectory_id, submap_index};
254       if (submap_data_.at(submap_id).state == SubmapState::kFinished) {
255         CHECK_EQ(submap_data_.at(submap_id).node_ids.count(node_id), 0);
256         ComputeConstraint(node_id, submap_id);
257       }
258     }
259   }
260
261   if (finished_submap != nullptr) {
262     const mapping::SubmapId finished_submap_id = GetSubmapId(finished_submap);
263     SubmapData& finished_submap_data = submap_data_.at(finished_submap_id);
264     CHECK(finished_submap_data.state == SubmapState::kActive);
265     finished_submap_data.state = SubmapState::kFinished;
266     // We have a new completed submap, so we look into adding constraints for
267     // old scans.
268     ComputeConstraintsForOldScans(finished_submap);
269   }
270   constraint_builder_.NotifyEndOfScan();
271   ++num_scans_since_last_loop_closure_;
272   if (options_.optimize_every_n_scans() > 0 &&
273       num_scans_since_last_loop_closure_ > options_.optimize_every_n_scans()) {
274     CHECK(!run_loop_closure_);
275     run_loop_closure_ = true;
276     // If there is a 'scan_queue_' already, some other thread will take care.
277     if (scan_queue_ == nullptr) {
278       scan_queue_ = common::make_unique<std::deque<std::function<void()>>>();
279       HandleScanQueue();
280     }
281   }
282 }
283
284 void SparsePoseGraph::HandleScanQueue() {
285   constraint_builder_.WhenDone(
286       [this](const sparse_pose_graph::ConstraintBuilder::Result& result) {
287         {
288           common::MutexLocker locker(&mutex_);
289           constraints_.insert(constraints_.end(), result.begin(), result.end());
290         }
291         RunOptimization();
```

```
292
293         common::MutexLocker locker(&mutex_);
294         num_scans_since_last_loop_closure_ = 0;
295         run_loop_closure_ = false;
296         while (!run_loop_closure_) {
297           if (scan_queue_->empty()) {
298             LOG(INFO) << "We caught up. Hooray!";
299             scan_queue_.reset();
300             return;
301           }
302           scan_queue_->front()();
303           scan_queue_->pop_front();
304         }
305         // We have to optimize again.
306         HandleScanQueue();
307       });
308 }
309
310 void SparsePoseGraph::WaitForAllComputations() {
311   bool notification = false;
312   common::MutexLocker locker(&mutex_);
313   const int num_finished_scans_at_start =
314       constraint_builder_.GetNumFinishedScans();
315   while (!locker.AwaitWithTimeout(
316       [this]() REQUIRES(mutex_) {
317         return constraint_builder_.GetNumFinishedScans() ==
318                num_trajectory_nodes_;
319       },
320       common::FromSeconds(1.))) {
321     std::ostringstream progress_info;
322     progress_info << "Optimizing: " << std::fixed << std::setprecision(1)
323                   << 100. *
324                          (constraint_builder_.GetNumFinishedScans() -
325                           num_finished_scans_at_start) /
326                          (num_trajectory_nodes_ - num_finished_scans_at_start)
327                   << "%...";
328     std::cout << "\r\x1b[K" << progress_info.str() << std::flush;
329   }
330   std::cout << "\r\x1b[KOptimizing: Done.     " << std::endl;
331   constraint_builder_.WhenDone(
332       [this, &notification](
333           const sparse_pose_graph::ConstraintBuilder::Result& result) {
334         common::MutexLocker locker(&mutex_);
335         constraints_.insert(constraints_.end(), result.begin(), result.end());
336         notification = true;
337       });
338   locker.Await([&notification]() { return notification; });
339 }
340
341 void SparsePoseGraph::AddTrimmer(
342     std::unique_ptr<mapping::PoseGraphTrimmer> trimmer) {
343   common::MutexLocker locker(&mutex_);
344   // C++11 does not allow us to move a unique_ptr into a lambda.
345   mapping::PoseGraphTrimmer* const trimmer_ptr = trimmer.release();
346   AddWorkItem([this, trimmer_ptr]()
347                   REQUIRES(mutex_) { trimmers_.emplace_back(trimmer_ptr); });
348 }
349
350 void SparsePoseGraph::RunFinalOptimization() {
351   WaitForAllComputations();
352   optimization_problem_.SetMaxNumIterations(
353       options_.max_num_final_iterations());
354   RunOptimization();
355   optimization_problem_.SetMaxNumIterations(
356       options_.optimization_problem_options()
357           .ceres_solver_options()
358           .max_num_iterations());
359 }
360
361 void SparsePoseGraph::RunOptimization() {
362   if (optimization_problem_.submap_data().empty()) {
363     return;
364   }
365   optimization_problem_.Solve(constraints_);
366   common::MutexLocker locker(&mutex_);
367
```

```
368   const auto& node_data = optimization_problem_.node_data();
369   for (int trajectory_id = 0;
370        trajectory_id != static_cast<int>(node_data.size()); ++trajectory_id) {
371     int node_index = 0;
372     const int num_nodes = trajectory_nodes_.num_indices(trajectory_id);
373     for (; node_index != static_cast<int>(node_data[trajectory_id].size());
374          ++node_index) {
375       const mapping::NodeId node_id{trajectory_id, node_index};
376       trajectory_nodes_.at(node_id).pose = transform::Embed3D(
377           node_data[trajectory_id][node_index].point_cloud_pose);
378     }
379     // Extrapolate all point cloud poses that were added later.
380     const auto local_to_new_global = ComputeLocalToGlobalTransform(
381         optimization_problem_.submap_data(), trajectory_id);
382     const auto local_to_old_global = ComputeLocalToGlobalTransform(
383         optimized_submap_transforms_, trajectory_id);
384     const transform::Rigid3d old_global_to_new_global =
385         local_to_new_global * local_to_old_global.inverse();
386     for (; node_index < num_nodes; ++node_index) {
387       const mapping::NodeId node_id{trajectory_id, node_index};
388       trajectory_nodes_.at(node_id).pose =
389           old_global_to_new_global * trajectory_nodes_.at(node_id).pose;
390     }
391   }
392   optimized_submap_transforms_ = optimization_problem_.submap_data();
393   connected_components_ = trajectory_connectivity_.ConnectedComponents();
394   reverse_connected_components_.clear();
395   for (size_t i = 0; i != connected_components_.size(); ++i) {
396     for (const int trajectory_id : connected_components_[i]) {
397       reverse_connected_components_.emplace(trajectory_id, i);
398     }
399   }
400
401   TrimmingHandle trimming_handle(this);
402   for (auto& trimmer : trimmers_) {
403     trimmer->Trim(&trimming_handle);
404   }
405 }
406
407 std::vector<std::vector<mapping::TrajectoryNode>>
408 SparsePoseGraph::GetTrajectoryNodes() {
409   common::MutexLocker locker(&mutex_);
410   return trajectory_nodes_.data();
411 }
412
413 std::vector<SparsePoseGraph::Constraint> SparsePoseGraph::constraints() {
414   common::MutexLocker locker(&mutex_);
415   return constraints_;
416 }
417
418 transform::Rigid3d SparsePoseGraph::GetLocalToGlobalTransform(
419     const int trajectory_id) {
420   common::MutexLocker locker(&mutex_);
421   return ComputeLocalToGlobalTransform(optimized_submap_transforms_,
422                                        trajectory_id);
423 }
424
425 std::vector<std::vector<int>> SparsePoseGraph::GetConnectedTrajectories() {
426   common::MutexLocker locker(&mutex_);
427   return connected_components_;
428 }
429
430 int SparsePoseGraph::num_submaps(const int trajectory_id) {
431   common::MutexLocker locker(&mutex_);
432   if (trajectory_id >= submap_data_.num_trajectories()) {
433     return 0;
434   }
435   return submap_data_.num_indices(trajectory_id);
436 }
437
438 transform::Rigid3d SparsePoseGraph::GetSubmapTransform(
439     const mapping::SubmapId& submap_id) {
440   common::MutexLocker locker(&mutex_);
441   // We already have an optimized pose.
442   if (submap_id.trajectory_id <
443           static_cast<int>(optimized_submap_transforms_.size()) &&
```

```
444       submap_id.submap_index < static_cast<int>(optimized_submap_transforms_
445                                                     .at(submap_id.trajectory_id)
446                                                     .size())) {
447     return transform::Embed3D(
448         optimized_submap_transforms_.at(submap_id.trajectory_id)
449             .at(submap_id.submap_index)
450             .pose);
451   }
452   // We have to extrapolate.
453   return ComputeLocalToGlobalTransform(optimized_submap_transforms_,
454                                        submap_id.trajectory_id) *
455          submap_data_.at(submap_id).submap->local_pose();
456 }
457
458 transform::Rigid3d SparsePoseGraph::ComputeLocalToGlobalTransform(
459     const std::vector<std::vector<sparse_pose_graph::SubmapData>>&
460         submap_transforms,
461     const int trajectory_id) const {
462   if (trajectory_id >= static_cast<int>(submap_transforms.size()) ||
463       submap_transforms.at(trajectory_id).empty()) {
464     return transform::Rigid3d::Identity();
465   }
466   const mapping::SubmapId last_optimized_submap_id{
467       trajectory_id,
468       static_cast<int>(submap_transforms.at(trajectory_id).size() - 1)};
469   // Accessing 'local_pose' in Submap is okay, since the member is const.
470   return transform::Embed3D(submap_transforms.at(trajectory_id).back().pose) *
471          submap_data_.at(last_optimized_submap_id)
472              .submap->local_pose()
473              .inverse();
474 }
475
476 SparsePoseGraph::TrimmingHandle::TrimmingHandle(SparsePoseGraph* const parent)
477     : parent_(parent) {}
478
479 int SparsePoseGraph::TrimmingHandle::num_submaps(
480     const int trajectory_id) const {
481   return parent_->optimization_problem_.submap_data().at(trajectory_id).size();
482 }
483
484 void SparsePoseGraph::TrimmingHandle::MarkSubmapAsTrimmed(
485     const mapping::SubmapId& submap_id) {
486   // TODO(hrapp): We have to make sure that the trajectory has been finished
487   // if we want to delete the last submaps.
488   CHECK(parent_->submap_data_.at(submap_id).state == SubmapState::kFinished);
489
490   // Compile all nodes that are still INTRA_SUBMAP constrained once the submap
491   // with 'submap_id' is gone.
492   std::set<mapping::NodeId> nodes_to_retain;
493   for (const Constraint& constraint : parent_->constraints_) {
494     if (constraint.tag == Constraint::Tag::INTRA_SUBMAP &&
495         constraint.submap_id != submap_id) {
496       nodes_to_retain.insert(constraint.node_id);
497     }
498   }
499   // Remove all 'constraints_' related to 'submap_id'.
500   std::set<mapping::NodeId> nodes_to_remove;
501   {
502     std::vector<Constraint> constraints;
503     for (const Constraint& constraint : parent_->constraints_) {
504       if (constraint.submap_id == submap_id) {
505         if (constraint.tag == Constraint::Tag::INTRA_SUBMAP &&
506             nodes_to_retain.count(constraint.node_id) == 0) {
507           // This node will no longer be INTRA_SUBMAP contrained and has to be
508           // removed.
509           nodes_to_remove.insert(constraint.node_id);
510         }
511       } else {
512         constraints.push_back(constraint);
513       }
514     }
515     parent_->constraints_ = std::move(constraints);
516   }
517   // Remove all 'constraints_' related to 'nodes_to_remove'.
518   {
519     std::vector<Constraint> constraints;
```

```
520       for (const Constraint& constraint : parent_->constraints_) {
521         if (nodes_to_remove.count(constraint.node_id) == 0) {
522           constraints.push_back(constraint);
523         }
524       }
525       parent_->constraints_ = std::move(constraints);
526     }
527
528     // Mark the submap with 'submap_id' as trimmed and remove its data.
529     parent_->submap_data_.at(submap_id).state = SubmapState::kTrimmed;
530     parent_->constraint_builder_.DeleteScanMatcher(submap_id);
531     // TODO(hrapp): Make 'Submap' object thread safe and remove submap data in
532     // there.
533
534     // Mark the 'nodes_to_remove' as trimmed and remove their data.
535     for (const mapping::NodeId& node_id : nodes_to_remove) {
536       CHECK(!parent_->trajectory_nodes_.at(node_id).trimmed());
537       parent_->trajectory_nodes_.at(node_id).constant_data.reset();
538     }
539
540     // TODO(whess): The optimization problem should no longer include the submap
541     // and the removed nodes.
542
543     // TODO(whess): If the first submap is gone, we want to tie the first not
544     // yet trimmed submap to be set fixed to its current pose.
545
546     // TODO(hrapp): Delete related IMU data.
547   }
548
549   }  // namespace mapping_2d
550   }  // namespace cartographer
```

---

cartographer

Author(s):

autogenerated on Mon Jun 10 2019 12:51:39